PO - Java

Nadjib Achir

nadjib.achir@univ-paris13.fr

Université Paris 13

Sources & remerciements

- Ces transparents doivent beaucoup à :
  - **José Paumard** (Université Paris 13),
    - http ://blog.paumard.org
  - **Julien Sopena** (Université Paris 6),
    - http ://julien.sopena.fr
- dont les cours et/ou les transparents sont extraordinairement complets et bien faits.

Plan du cours

Plan du cours

Java c'est quoi ?

- Un langage : Orienté **objet** fortement **typé** avec classes
- Un environnement d'exécution (Java Runtime Environment – **JRE**) : Une machine virtuelle et un ensemble de bibliothèques
- Un environnement de développement (Java Development Kit – **JDK**) : Une machine virtuelle et un ensemble d'outils
- Java **hérite** principalement sa syntaxe (procédurale) du `C`
- Langage **généraliste**, aussi versatile que `C++`
- Plusieurs **simplifications** notables par rapport au `C++`
- Très **vaste bibliothèque** de classes standard (plus de 3000 classes dans plus de 160 paquetages pour le **JDK 1.5**)

Naissance de Java 1/2

- A partir de **1993**, chez Sun, développement d'un langage adapté à l'Internet
- En **1995**, annonce officielle de `Java` (conçu, entre autres, par *James Gosling*, *Patick Naughton*, *Crhis Warth*, *Ed Frank*, *Mike Sheridan* et *Bill Joy*)
- Milieu **1996**, sortie de `Java 1.02` (Oak), première version distribuée par *JavaSoft* (filiale de *Sun*)
- Début **1997**, sortie de `Java 1.1`. Beaucoup d'améliorations significatives. Première version à être jugée sérieuse du langage
- Été **2004**, sortie de `Java 1.5` (Tiger) ; diverses améliorations et ajouts intéressants
- **2005**, sortie de `Java 1.6` (Mustang)
- ...
- **2011**, sortie de `Java 1.7` (Java 7)

Naissance de Java 2/2

- **2014**, sortie de `Java 1.8` (Java 8 – LTS) $\rightarrow$ Lambda
- **2017**, sortie de `Java 9` $\rightarrow$ Module
- **2018**, sortie de `Java 11` LTS
- **2021**, sortie de `Java 17` LTS
- **2023**, sortie de `Java 21` LTS

Particularité de Java

- Un programme Java est tout d'abord écrit sous forme de **code source** dans un ou **plusieurs fichiers** de type **texte**
- Par convention, ces fichiers portent l'extension **.java**
- ... d'autres contraintes sur le **nommage** et la façon dont on les **ranger** dans des répertoires
- Pour pouvoir être exécutés, ces fichiers doivent être **compilés**, ensemble ou séparément
- **Compilation** en Java $\neq$ **compilation** en C / C++
- Le compilateur est une application fournie avec le JDK, qui s'appelle `javac`

Plan du cours

Java versus C++

- Filiation historique :
  - **1983** (*AT&T Bell*) : C++
  - **1991** (*Sun Microsystems*) : Java
  - **2011** (*Oracle*) : Java

- Java est **très proche** du langage C++ (et donc du langage C)

- Toutefois Java est **plus simple** que le langage C++, car les points "critiques" du langage C++ (*à l'origine des principales erreurs*) ont été supprimés

- Cela comprend :
  - Les **pointeurs**
  - La **surcharge d'opérateurs**
  - L'**héritage multiple**

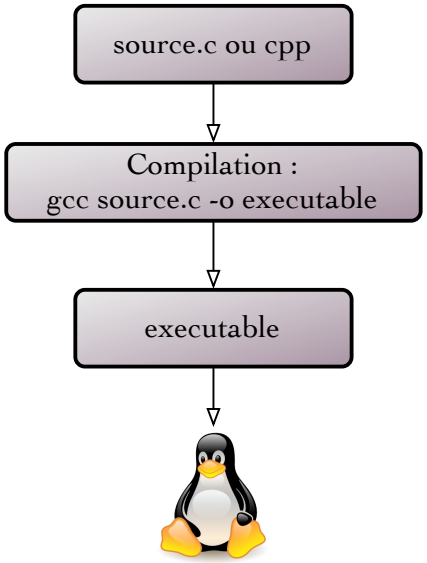
Java versus C++

- **Tout est dynamique** : les instances d'une classe sont instanciées dynamiquement
- **La libération de mémoire est transparente** pour l'utilisateur
- **Pas nécessaire de spécifier de mécanisme de destruction**
- La libération de l'espace mémoire est prise en charge par un gestionnaire appelé **garbage collector** $\Rightarrow$ chargé de détecter les objets à détruire
- Avantages/Inconvénients :
  - gain de **fiabilité** (pas de désallocation erronée)
  - a un **coût** (perte en rapidité par rapport au C++)

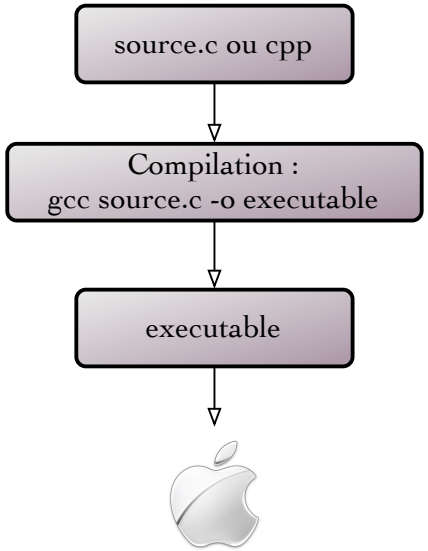
Java versus C++

- Une fois achevée la production du logiciel, un choix doit être fait entre fournir le **source** ou le **binaire** pour la machine du client
- Généralement, un développeur ou une entreprise souhaitent **protéger le code source** et **distribuer le code binaire**
- Le **code binaire** doit donc être **portable** sur des **architectures différentes** (processeur, système d'exploitation, etc.)
- À l'instar du compilateur C, le compilateur C++ produit du code **natif**, i.e., qu'il produit un exécutable propre à l'environnement de travail ou le code source est compilé
- On doit donc **créer** les exécutables pour **chaque type d'architecture** potentielle des clients

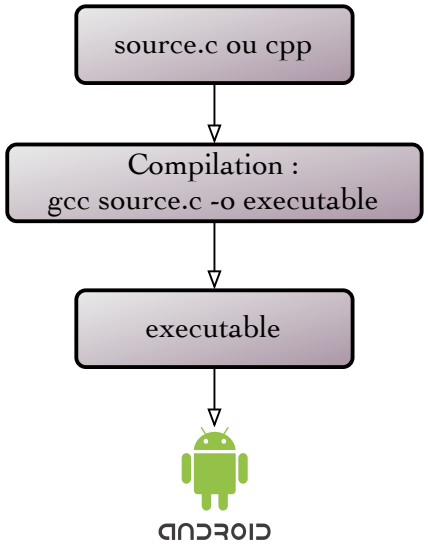
Java versus C++



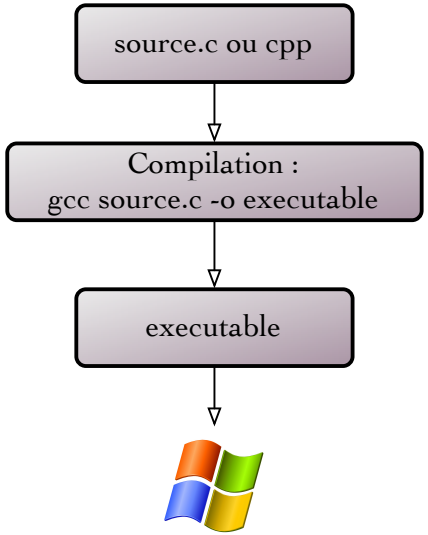
C/C++ sur **linux**



C/C++ sur **mac**



C/C++ sur **android**



C/C++ sur **windows**

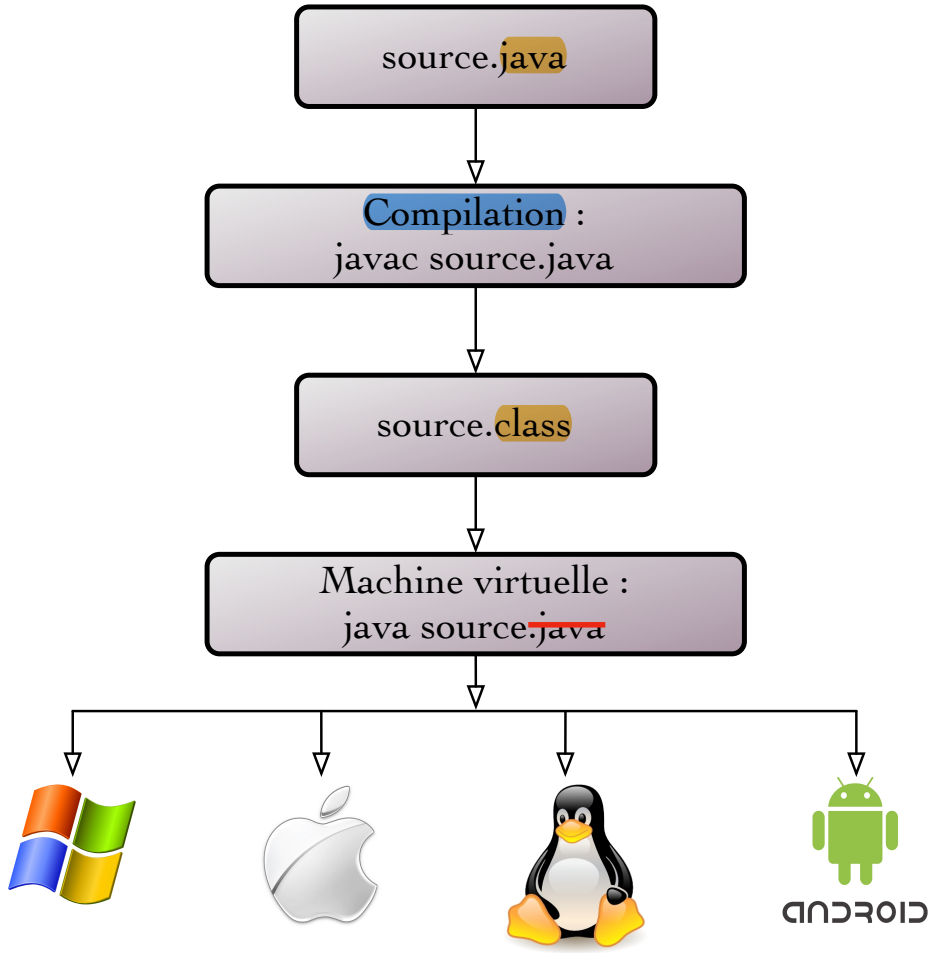
Java versus C++

- En Java, le **code source n'est pas traduit directement dans le langage de l'ordinateur**
- Il est d'abord traduit dans un langage appelé "**bytecode**", langage d'une machine virtuelle (**JVM – Java Virtual Machine**) définie par *Sun*
- Le **bytecode** généré par le compilateur **ne dépend pas de l'architecture de la machine** où a été compilé le code source
- Les **bytecodes** produits sur une machine **pourront s'exécuter** (au travers d'une machine virtuelle) **sur des architectures différentes**

Java versus C++

- Le **bytecode** doit être **exécuté** par une **Machine Virtuelle Java**
- Cette **JVM n'existe pas**. Elle est **simulée** par un programme qui :
  1. **lit** les instructions (en **bytecode**) du programme `.class`
  2. fait une passe de **vérification** (*type opérande, taille de pile, flot données, variable bien initialisé,...*) pour s'assurer qu'il n'y a aucune action dangereuse
  3. fait plusieurs passes d'**optimisation** du code
  4. les **traduit** dans le langage natif du processeur de l'ordinateur
  5. lance leur **exécution**

Java versus C++



Compilation et execution en JAVA

Plan du cours

Cycle de vie

- La commande de base pour **compiler** le fichier `application.java` est donc :

```
$ javac application.java
```

- S'il ne donne **pas** de messages **d'erreur**, `javac` produit alors un fichier au même endroit que `application.java` : `application.class`

- Le fichier `application.class` (format **bytecode**) contient le code qui va être exécuté dans la machine virtuelle Java

- On peut **copier** ce fichier sur toute autre machine que celle sur laquelle il a été compilé, et l'exécuter

- L'exécution du programme est appelée par la commande suivante :

```
$ java application
```

Plan du cours

Un premier programme

- Comme le veux la tradition, nous commençons par l'écriture d'un petit programme capable d'écrire **Bonjour le monde** sur un écran

```
1 public class Bonjour {
2     public static void main (String [] arguments) {
3         System.out.println("Bonjour le monde") ;
4         System.exit(0) ;
5     }
6 }
```

- On remarque tout d'abord que le code est contenu dans un bloc : **public class Bonjour ...**

- Tout code Java doit être déclaré à l'intérieur d'une classe, et la fonction de ce bloc est de délimiter une telle classe

- À l'intérieur d'une classe, on peut déclarer trois types de choses : *des champs*, *des méthodes* et *des blocs*

Un premier programme

- La première déclaration que l'on trouve dans ce bloc est celle d'une méthode :

```
1 public static void main{...}
```

- Cette méthode `main` a un statut particulier : c'est elle qui est appelée en premier quand on lance un programme `Java`

- À l'intérieur de ce bloc se trouvent deux commandes :
  - La première affiche "`Bonjour le monde`" sur la console
  - La deuxième ferme le programme

Plan du cours

2. Programmation orientée objets
   - Principe de la programmation & concept objet
   - Protection de l'information & encapsulation
   - Notion de classe
   - Une première classe
   - Qlqs règles

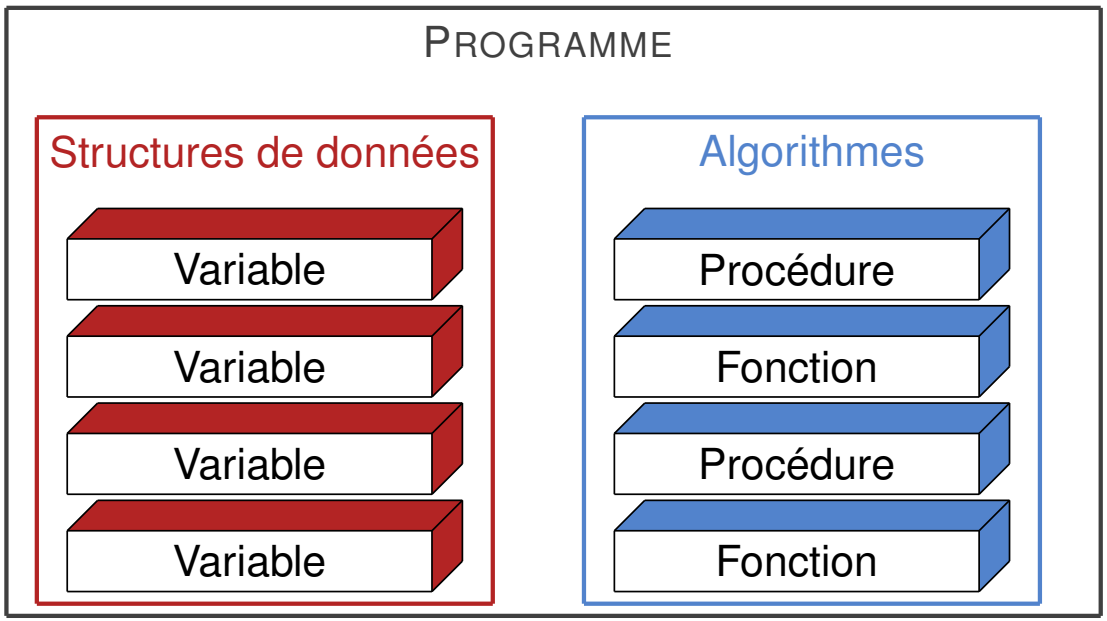
Plan du cours

Principe de la programmation

- Historiquement, le schéma **simplifié** d'un système informatique peut se résumer par la formule :

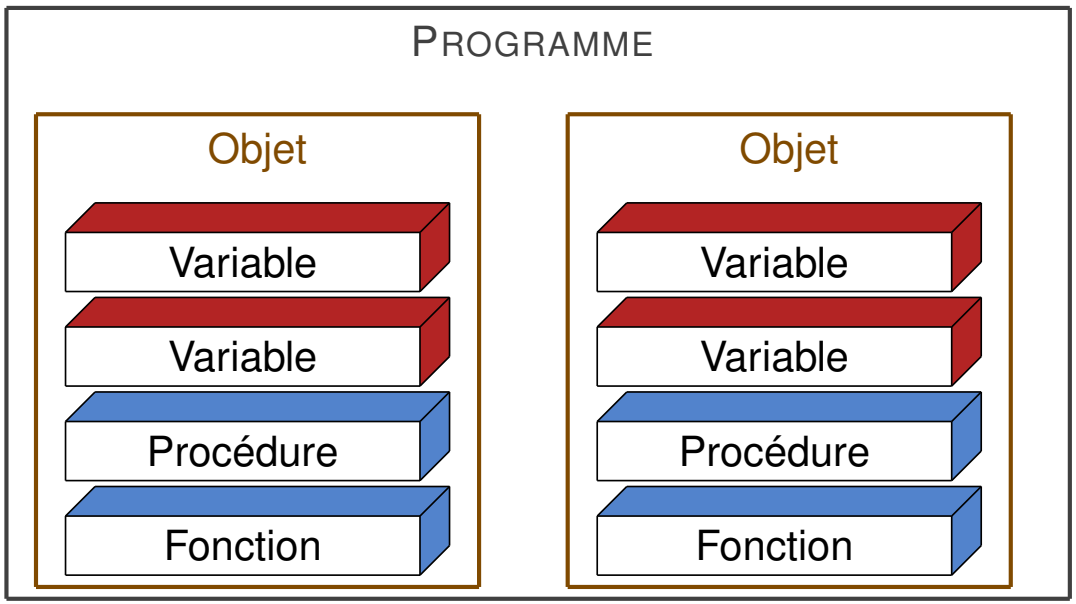
Système informatique = *Structures de données* + *Traitements*

- On sépare les données des moyens de traitement de ces données



Conception par objets : principe

- Afin d'établir de façon **stable** et **robuste** l'architecture d'un système informatique, il semble raisonnable de s'organiser autour des données manipulées.



Le concept d'objet

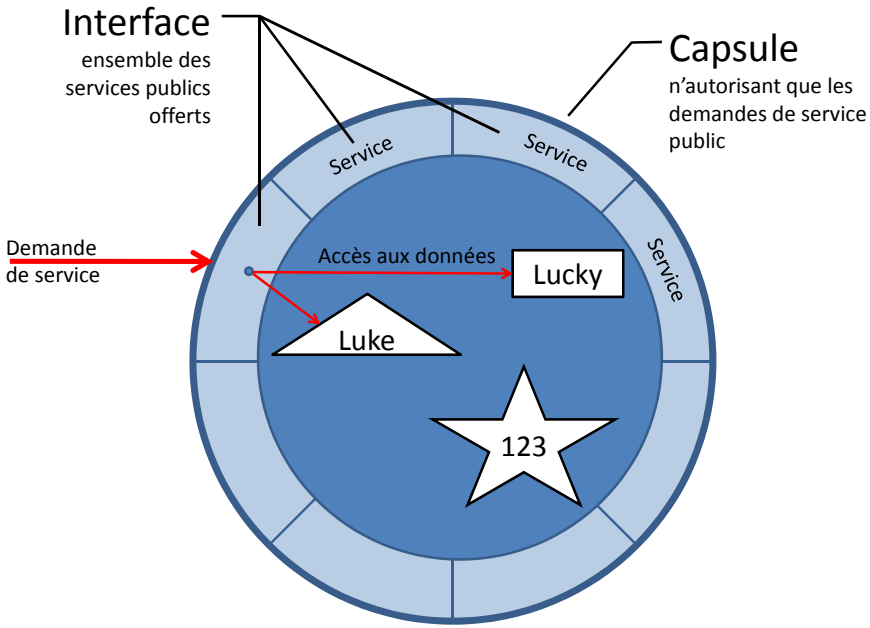
- *Qu'est-ce qu'un objet ?*
  - Le monde qui nous entoure est composé d'objets
  - Ces objets ont tous deux caractéristiques
    - un *état* (**attributs**)
    - un *comportement* (**méthodes**/**opérations**)

- Exemples d'objets du monde réel
  - chien
    - **état** : *nom*, *couleur*, *race*, *poids*....
    - **comportement** : *manger*, *aboyer*, *renifler*...
  - Bicyclette
    - **état** : nombre de vitesses, vitesse courante, couleur
    - **comportement** : tourner, accélérer, changer de vitesse

Plan du cours

Protection de l'information et encapsulation

- Les **données d'un objet** (son *état*) peuvent être **lues ou modifiées uniquement** par les services proposés par l'**objet lui-même** (ses *méthodes*)



- Le terme **encapsulation** désigne le principe consistant à **cacher l'information contenue dans un objet** et de ne proposer que des **méthodes de modification/accès** à ces **propriétés** (attributs)

Protection de l'information et encapsulation

- L'objet est vu de l'extérieur comme une **boîte noire** ayant certaines propriétés et ayant un comportement spécifié

- La manière dont le **comportement** a été implémenté est **cachée** aux **utilisateurs** de l'objet

- **Protéger la structure interne** de l'objet contre toute manipulation non contrôlée, produisant une incohérence

- L'encapsulation nécessite la spécification de parties **publics** et **privées** de l'objet
  - **éléments publics** : *Partie visible de l'objet depuis l'extérieur*
    - Un ensemble de méthodes utilisables par d'autres objets
  - **éléments privées** : *Partie non visible de l'objet*
    - Constitué des éléments de l'objet visibles uniquement de l'intérieur de l'objet et de la définition des méthodes

Plan du cours

Java et le concept d'objet : notion de classe

- Pour être véritablement intéressante, la notion d'objet doit permettre un certain degré d'abstraction $\Rightarrow$ **notion de classe**

Définition

classe = **structure d'un objet** $\Rightarrow$ *la déclaration de l'ensemble des membres qui composeront un objet*

Définition

La classe peut être vue comme un **moule** pour la **création des objets**, qu'on appelle alors des **instances de la classe**

Relation entre classe et objet

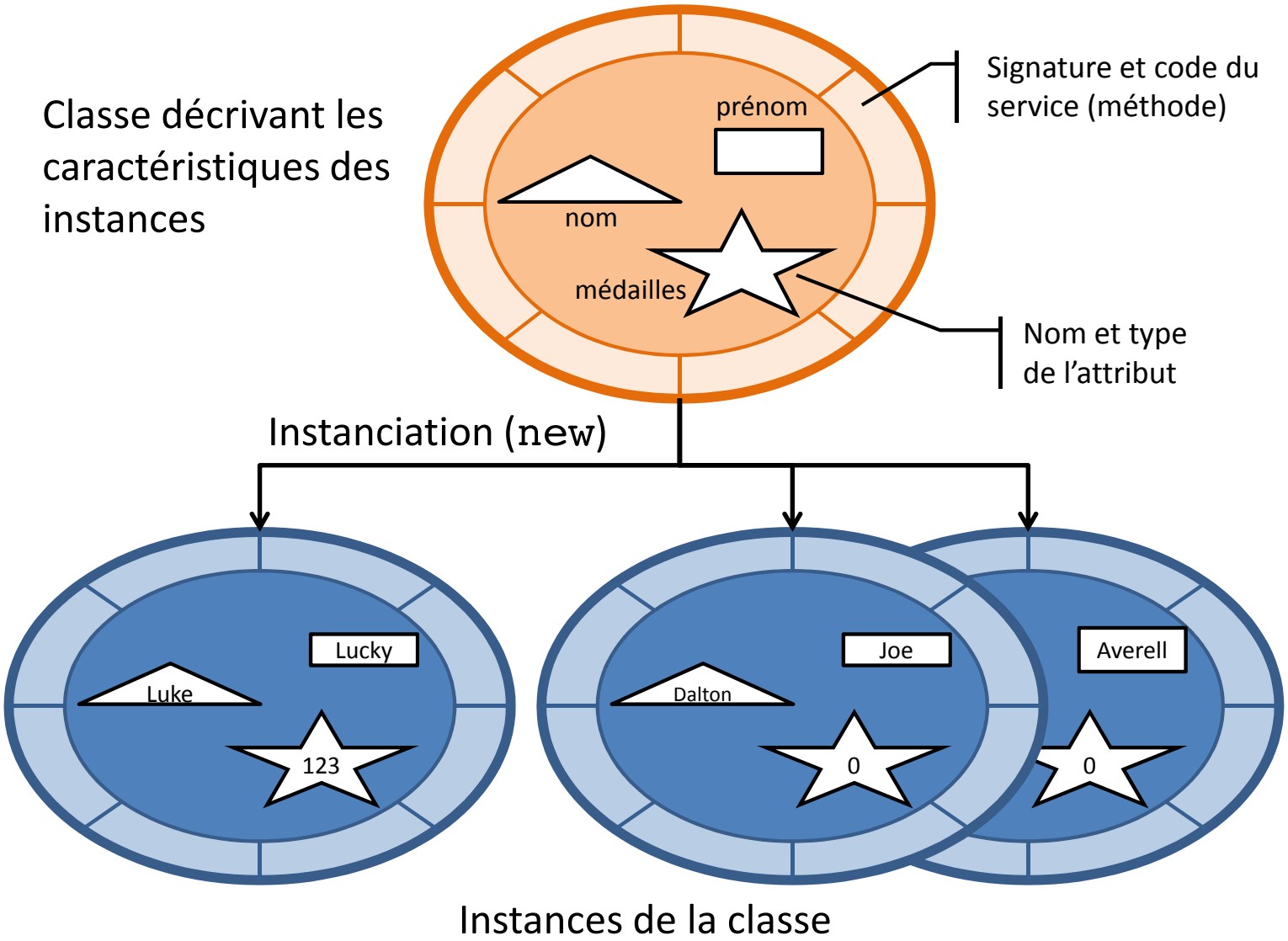
- Il est important de saisir les différences entre les notions de **classe** et **instance de la classe** :

classe = *champs* + *méthodes* + *mécanismes d'instanciation* + *mécanismes de destruction*

instance de la classe = *valeurs des champs* + *accès aux méthodes*

- L'instanciation est le mécanisme qui permet de **créer** des instances dont les **traits** sont **décrits par la classe**
- La **destruction** est le mécanisme qui permet de **détruire** une **instance de la classe**
- L'ensemble des **instances** d'une classe constitue l'**extension de la classe**

Classes vs objets



Plan du cours

Écriture d'une première classe

- Voyons ici un exemple simple pour introduire les concepts généraux
- Nous allons écrire une classe `Marin`, afin de modéliser des marins avec les attribues :
  - *nom*, *prénom* (deux chaînes de caractères) ; *salaire* (entier)

Écriture d'une classe

```
1  // nom de la classe
2  public class Marin {
3  // (1) champs
4      String nom, prenom ;
5      int salaire ;
6  // (2.a) mecansime d'instantiation ou constructeur
7      public Marin (String nouveauNom, String nouveauPrenom, int
           nouveauSalaire) {
8          nom = nouveauNom ;
9          prenom = nouveauPrenom ;
10         salaire = nouveauSalaire ;
11     }
12 // (2.b) mecanisme d'instantiation ou constructeur
13     public Marin (String nouveauNom, int nouveauSalaire) {
14         nom = nouveauNom ;
15         prenom = "" ;
16         salaire = nouveauSalaire ;
17     }
18 // (3) methode
19     public void augmenteSalaire (int montant) {
20         salaire = salaire + montant ;
21     }
22 }
```

Écriture d'une classe

- On reconnaît la structure classique d'une classe :
  - le nom de la classe : **public class Marin**
  - les champs : **nom** et **prenom** de type **String**, et **salaire** de type **int**
  - deux éléments : des méthodes, `public Marin` (il y en a deux), et **public void augmenteSalaire**.
- Les deux premières méthodes **public Marin** sont d'un type particulier : les **constructeurs** de la classe `Marin`.
- Un **constructeur** est une méthode qui doit respecter quelques contraintes :
  - *elle doit porter le même nom que la classe*
  - *ne pas avoir de déclaration de type de retour* (pas même void)
  - notons ici qu'un constructeur n'est *pas nécessairement* **public**
- La méthode **public void augmenteSalaire** est une méthode classique.
  - **public** : signifie qu'elle peut être appelée en dehors de la classe
  - **void** : indique que cette méthode ne renvoie rien
  - **augmenteSalaire** : nom de la méthode

Instanciation de cette classe

- On peut créer un **objet instance** de cette classe par une déclaration du type

```
1 Marin marin1 = new Marin ("Surcouf", "Robert", 25000) ;
```

- **new** réserve un espace mémoire capable de contenir un objet de type `Marin`, et l'initialise
  - appel au constructeur `Marin (String, String, int)`
- Créons un autre objet de type Marin de cette façon :

```
1 Marin marin2 = marin1 ;
```

- **Affectation d'objet** $\Rightarrow$ la machine Java recopie la référence vers l'objet cible dans la variable de destination $\Rightarrow$ l'objet **marin2** référence la même zone mémoire que **marin1** $\Rightarrow$ pas de duplication de la zone mémoire $\Rightarrow$ Les deux objets **marin1** et **marin2** sont donc les mêmes.

Instanciation de cette classe

- Exemple :

```
1 marin1.augmenteSalaire(100) ;
2 marin1.nom = marin1.nom.toUpperCase() ;
3 boolean b1 = (marin1 == marin2) ;
```

1. Le salaire de **marin2** est aussi augmenté, puisqu'il s'agit physiquement du même objet
2. On convertit de ce fait le champ nom de **marin1** en majuscule. Comme **marin2** partage ce champ avec **marin1**, **marin2.nom** est aussi en majuscules
3. l'operateur == compare les valeurs des références, et non pas leur contenu. Si les deux références **marin1** et **marin2** référencent la même zone mémoire $\Rightarrow$ la valeur de **b1** est **true**

Instanciation de cette classe

- Instancions maintenant **marin1** et **marin2** de la façon suivante :

```
1  Marin marin1 = new Marin ("Surcouf", "Robert", 25000) ;
2  Marin marin2 = new Marin ("Surcouf", "Robert", 25000) ;
3  boolean b2 = (marin1 == marin2) ;
```

- Cette fois-ci, chaque variable porte une référence vers une zone mémoire qui lui est propre (deux opérations **new** sont réalisées)

- Changer un champ de **marin1** n'aura pas d'effet sur **marin2**

- Comparons **marin1** et **marin2** de la même façon que précédemment. La valeur de **b2** est dans ce cas **false**

Plan du cours

Qlqs règles

- Il existe deux types de règles à suivre lorsque l'on écrit du code `Java`
  - **les obligations** : imposées par la **JLS** (*Java Language Specification*)
  - **les bonnes habitudes**

- Les bonnes habitudes :
  - Ecrire correctement ses identificateurs : *noms de classes*, *des méthodes* ou *des champs*

  - Un identificateur d'un attribut ou d'une méthode **commence par une minuscule**, et ne comporte pas de caractère '_'

  - Lorsqu'un identificateur est composé de plusieurs mots, alors on met en **majuscule la première lettre des mots qui le composent** (exp : `ageMarin`)

Plan du cours

Plan du cours

Types de classes

- Il existe *trois types de classe* en `Java` :
  - les classes **publiques**, (de loin les plus utilisées)
  - les classes **locales**
  - les classes **membres**

Type de Classes : publiques

- Classes **publiques**
  - Une classe publique est déclarée par les mots-clés **public class**
  - Doit être enregistrée dans un **fichier** qui porte le **même nom** qu'elle
  - Son nom **commence** en général par une **majuscule**

```
1 public class Marin { // doit être écrite dans le fichier Marin.java !
2     ... // code de la classe
3 }
```

Type de Classes : internes

- Classes **internes**
  - On peut pas avoir plus d'**une classe publique** dans un **fichier** donné
  - On peut ajouter d'autres classes, **non publiques**
  - Légale mais **pas conseillée**

```
1 public class Marin { // doit être écrite dans le fichier Marin.java
2     ... // code de la classe
3 }
4 class Capitaine { // écrite dans le êmme fichier
5     ... // code de la classe Capitaine
6 }
```

Type de Classes : membre

- Classe **membre**
  - Une classe **membre** est une classe déclarée à l'**intérieur** d'une autre **classe**
  - Elle peut être **static** ou non, final ou non, **public**, **private** ou **protected**

```
1  public class Character { // declaree dans le fichier java/lang/Character
2      ... // code de la classe Character
3      public static class Subset {
4          ... // code de la classe Subset
5      }
6      public static final class UnicodeBlock extends Subset {
7          ... // code de la classe UnicodeBlock
8      }
9  }
```

- La classe `Character` contient une première classe membre : `Subset`

- Une classe membre peut **étendre** une autre classe, ici une autre **classe membre**

Type de Classes : locales

- Classes **locales**
  - Une classe **locale** est déclarée dans une méthode
  - Ne fait **pas partie des membres de la classe** qui contient cette méthode

```
1  public class Marin implements Comparable<Marin> {
2      private String nom, prenom ;
3      public int compareTo(Marin marin) {
4
5          class MarinComparator implements Comparator<Marin> {
6
7              public int compare(Marin m1, Marin m2) {
8                  int order = m1.nom.compareTo(m2.nom) ;
9                  return order != 0 ? order : m1.prenom.compareTo(m2.prenom)
                       ;
10             }
11         }
12         return (new MarinComparator()).compare(this, marin) ;
13     }
14 }
```

Plan du cours

Le mot-clé `this`

- Le mot clé `this` désigne l'instance sur laquelle est invoquée la méthode.
- Utilisé dans 3 circonstances :
  1. pour **accéder aux champs** de l'objets
  2. pour **comparer la référence** de l'objet invoquant la méthode à une autre référence
  3. pour **passer la référence** de l'objets invoquant la méthode en paramètre d'une autre méthode
- Le mot-clé `this` n'est **pas défini dans un élément statique**

```
1  public class Marin {  // écrit dans le fichier Marin.java
2      private String nom ;
3      public Marin(String nom) {
4          this.nom = nom ; // utilisation de this pour lever l'ambiguite
5                           // entre le parametre et le champ
6      }
7      public void setNom(String nom) {
8          this.nom = nom ; // utilisation de this pour lever l'ambiguite
9                          // entre le parametre et le champ
10     }
11 }
```

Le mot-clé `this` : exemple

- Dans le corps d'une méthode, `this` permet de **comparer la référence** de l'objet sur lequel est invoqué la méthode à une autre référence.

```
1  public class Animal {
2      Estomac estomac ;
3      ...
4      public void manger(Animal victime ) {
5          // On ne peut pas se manger completement soi-meme
6          if ( this != victime ) {
7             estomac.addNouriture ( victime );
8          }
9      }
10 }
```

Le mot-clé `this` : exemple

- Dans le corps d'une méthode, `this` permet de **passer la référence de l'objet** sur lequel est invoqué à une autre méthode.

```
1  public class Entreprise {
2      public int calculerSalaire(Individu x)
3          {...}
4  }
5  public class Individu {
6      int compte ;
7      Entreprise entreprise ; ...
8      public void demissionner () {
9          compte += entreprise.calculerSalaire(this);
10         entreprise = null;
11     }
```

Plan du cours

Plan du cours

Membres d'une classe

- Les **éléments** déclarés à l'intérieur d'une classe `Java` sont appelés des **membres**

- Il existe *cinq* types de membres :

  - des **attribues** ou **champs** ;
  - des **champs et blocs statiques** ;
  - des **méthodes** (les constructeurs d'une classe en font partie) ;
  - des **blocs non statiques** ;
  - des **classes membre**.

Plan du cours

Accès à un membre & visibilité

- Pour accéder à un membre non statique d'un objet, **il faut posséder une référence sur cet objet**

- La visibilité d'un membre est définie par un mot clé qui peut prendre trois valeurs : `private`, `protected` ou `public`
  - Les membres `private` ne sont pas accessibles de l'extérieur d'une classe $\Rightarrow$ On ne peut donc **ni les lire**, **ni les modifier**

  - Les membres `public` sont accessibles de toute classe. Ils sont donc **lisibles et modifiables de tout objet**

  - L'accès aux membres `protected` sont **accessibles que des instances de classes** :
    - qui étendent la classe qui contient le membre `protected`

    - qui se trouvent dans le même package que celle qui contient le membre `protected`

Accès à un membre & visibilité

```
1  public class Marin {
2
3      protected String nom ;
4
5      protected String getNom() {
6          return this.nom ;
7      }
8  }
```

- Toutes les classes se trouvant dans le **même package** que `Marin` ont accès aux deux membres `protected` de `Marin` : `nom` et `getNom()`

- Les instances des classes qui **étendent** `Marin` ont également accès à ces membres

Plan du cours

Les champs

- Les **champs** sont les **membres** qui permettent de **stocker** des valeurs de types de base ou des objets
- Un champ est déclaré suivant la syntaxe suivante :

```
1 [visibilite] [static] typeDeChamp nomDuChamp [ = initialisation] ;
```

Déclaration d'un attribut/champ

- En Java, **toutes les variables doivent être déclarées** avant d'être utilisées (les champs comme les variable locales des méthodes)

- La **déclaration** des champs se fait de **préférence en début de classe** et leurs noms commencent par des minuscules

- On indique au compilateur :
  1. Un ensemble de modificateurs (facultatif)
  2. Le type de la variable
  3. Le nom de la variable

- Exemple

```
1 private double z ;
2 public String str ;
```

Plan du cours

Types de base pour les champs

- Il existe huit types de base en `Java` :
  - des types **entier** : `byte, short, int, long` ;
  - un type **caractère** : `char` ;
  - un type **booléen** : `boolean` ;
  - deux types **flottants** : `float et double`.

Type de base	Type	Nombre de bits	Valeurs possibles
`boolean`	booléen	32 (effectifs)	true et false
`byte`	entier	8	signées
`short`	entier	16	signées
`int`	entier	32	signées
`long`	entier	64	signées
`float`	virgule flottante	32	IEEE 754
`double`	virgule flottante	64	IEEE 754
`char`	caractère	16	Unicode

Types de base pour les champs

- Les types de base ne définissent pas d'objet. Quand on déclare une variable de type `int`, la machine `Java` nous donne en retour une **zone mémoire de 4 octets**.
- L'opérateur d'affectation recopie la valeur de la variable cible dans la variable destination. Par exemple :

```
1  int i, j ;
2  i = 0 ;
3  j = i ;
4  i = 1 ;
```

- On a bien, à l'issue de `cette` série d'instructions j = 0 et i = 1.
- **Remarque** : Java se charge de l'initialisation des variables avant que l'on puisse les utiliser. Les **nombres** sont mis à **0**, les **booléens** à **false** et les **caractères** à "**chaine vide**" avant d'être utilisés par une application.

Plan du cours

Champ statique

- **Par défaut** : chaque instance de chaque classe occupe son propre espace mémoire
- Deux objets instances d'une même classe occupent donc deux espaces mémoire ⇒ **ne se recouvrent pas**
- Par contre, déclarer un membre statique (`static`) ⇒ placé dans un espace mémoire commun à tous les objets de la classe
- Si un des objets modifie la valeur ⇒ tous les objets verront la valeur de ce champ modifiée.
- Il est même possible d'invoquer un élément statique d'une classe sans que celle-ci n'ait jamais été instanciée
- Bonne pratique :
  - Appeler les membres statiques d'une classe que de façon "statique", c'est-à-dire en utilisant le nom de la classe plutôt qu'une de ses instances

Champ statique

```
1  public class Marin { // dans le fichier Marin.java
2      public static int nombreMarins = 0 ;
3
4      public Marin() {
5          nombreMarins ++ ;
6      }
7  }
8  public class Application { // dans le fichier Application.java
9      public static void main(String [] args) {
10         Marin marin1 = new Marin() ;
11         Marin marin2 = null ;
12
13          // affiche 1
14         System.out.println("Nombre de marins = " + Marin.nombreMarins) ;
15
16          // affiche 1, methode non recommandee
17         System.out.println("Nombre de marins = " + marin1.nombreMarins) ;
18
19          // affiche 1, bien que marin2 soit null
20         System.out.println("Nombre de marins = " + marin2.nombreMarins) ;
21     }
22 }
```

Cas des constantes

- La bonne façon de définir une **constante** en `Java` est de la définir dans un champ `public static final`
- Le fait qu'il soit publique et statique permet d'y accéder de n'importe où
- Le déclarer `final` interdit sa modification, ce qui est en général recherché pour une constante !

```
1 public class Math {
2
3     public static final double PI = 3.14159265358979323846 ;
4
5 }
```

- Il est donc possible d'utiliser la constante `Math.PI` dans n'importe quel programme `Java`

Bloc statique

- Un **bloc statique** est un ensemble d'instructions déclaré suivant une syntaxe particulière

```
1 public class Marin {
2     static {
3         // ceci est un bloc statique
4     }
5 }
```

- Ces blocs sont à utiliser avec précautions

Plan du cours

Les méthodes

- Parties importantes d'un objet qui permettent de :
  - effectuer des **traitements**
  - **dialoguer** avec d'autres objets
  - **accéder** à des **ressources** telles que des fichiers, des bases de données, des terminaux graphiques ou non, ...

- Une méthode peut donc être `private`, `protected`, ou `public` :

```
1  [visibilite] [static] [signature] [throws exception] { ... }
```

- Une méthode peut être déclarée `static` $\Rightarrow$ **méthode de classe**

- Une méthode statique **ne peut accéder elle-même** aux champs et méthodes *non statiques* de cette classe

Les méthodes

- Une méthode peut **rencontrer une erreur** lors de son exécution $\Rightarrow$ la dernière déclaration permet de préciser le comportement face à cette erreur

- Une méthode pour laquelle on déclare un type de retour (`return`) doit obligatoirement retourner une valeur

- Dans le cas contraire le compilateur génèrera une erreur

```
1  public boolean testParite(int i) { // erreur de compilation : la
       methode ne retourne pas toujours une valeur
2      if (i % 2 == 0)
3          return true ;
4  }
```

- Le code précédent ne compile pas

Les méthodes

- À l'inverse, les deux méthodes suivantes compilent correctement

```
1  public boolean testParite1(int i) {
2      if (i % 2 == 0)
3          return true ;
4      else
5          return false ;
6  }
7
8   public boolean testParite2(int i) {
9      if (i % 2 == 0)
10         return true ;
11
12     return false ;
13 }
```

Signature d'une méthode

- Une méthode est caractérisée par sa **signature**
- La signature d'une méthode est composée :
  - de son **nom**
  - de la **liste ordonnée des paramètres** qu'elle accepte en entrée
- Les éléments suivants ne font pas partie de la signature d'une méthode :
  - le modificateur de `visibilité`
  - le type de retour
  - le modificateur `static`
  - la clause throws
- **Une classe ne peut pas avoir deux méthodes avec la même signature**

Plan du cours

Getters et Setters

- Deux types de méthodes ont un statut particulier en Java
  - les **getters** et les **setters**

- Afin de respecter le principe d'encapsulation, les champs non statiques d'une méthode sont déclarés `private`

- Afin de **pouvoir lire**/**modifier** ces champs $\Rightarrow$ on ajoute à la classe un getter et un setter pour chacun de ses champs `private`

- Le nom de ces deux méthodes est fixé par une convention
  - Un **getter** commence par **get**, suivi d'un **nom**, qui est en général le nom du **champ** commençant par une majuscule. Il retourne une valeur en général du même type que le champ associé

  - Le **setter** associé commence par **set**, suivi du même **nom** que le get. Il prend en paramètre un unique argument, du même type que le type de retour du getter associé

Getters et Setters

```
1  public class Marin { // La classe Marin comporte deux proprietes :
2      private String nomDuMarin ; // nom
3      private String prenomDuMarin ; // prenom
4
5      public Marin(String nom) { // Le nom d'un marin est fixe lors de sa
           construction
6       nomDuMarin = nom ;
7      }
8
9      public String getNom() { // Seulement en lecture, ne peut plus etre
           modifie
10        return nomDuMarin ;
11     }
12
13     public String getPrenom() { // prenom, que l'on peut lire et ...
14        return prenomDuMarin ;
15     }
16
17     public void setPrenom(String prenom) { // ... modifier
18        this.prenomDuMarin = prenom ;
19     }
20 }
```

Plan du cours

Classes n'ayant que des méthodes static

- Une classe peut ne contenir que des membres static. Dans ce cas elle ne sert pas à créer des objets ! ! !

- **Exemple 1** - La classe Math est une classe boîte à outils ne possédant que des constantes (E et PI) et méthodes de classes

```
1 int rayon = 30 ;
2 double surface = Math.PI * Math.pow(rayon, 2) ;
```

- **Exemple 2** - La classe System qui représente la VM ne possédant que des champs et méthodes de classes : il n'y a pas de raison d'en avoir plusieurs instances

```
1 // Permet d'afficher des messages dans la console
2 System.out.println("Hello, World !");
3 // Permet d'arreter brutalement un programme
4 System.exit (0);
```

Plan du cours

Plan du cours

Chargement d'une classe

- Etapes de l'instanciation d'un objet :

  1. *Charger la classe à laquelle il appartient*. Une classe n'est chargée qu'une seule fois, et reste ensuite présente dans la machine Java
  2. *Une classe n'est chargée que lorsqu'elle est référencée par un objet*
  3. `La machine Java charge toutes les classes dont cette classe hérite`, si elle ne l'ont pas déjà été

Plan du cours

Constructeurs d'une classe

- Un **constructeur** d'une classe est une méthode qui **n'a pas de type de retour**, et **porte le même nom que la classe** dans laquelle elle se trouve

- Une classe peut avoir **autant de constructeurs que l'on a le courage de lui en créer** $\Rightarrow$ **différentes signatures** $\Rightarrow$ des paramètres différents

- Une classe qui **ne déclare aucun constructeur** explicitement en possède en fait toujours un : **le constructeur vide par défaut**, qui ne prend aucun paramètre

- Bonne habitude de programmation : **toujours ajouter un constructeur par défaut vide** (sans paramètres)

Constructeurs d'une classe : exemple constructeur par defaut

```
1  public class Marin {
2      private String nom ;
3
4      public String getNom() {
5          return this.nom ;
6      }
7
8      public void setNom(String nom) {
9          this.nom = nom ;
10     }
11 }
```

- La machine Java a créé un constructeur vide par défaut dans cette classe, on peut donc l'instancier de la façon suivante :

```
1  Marin marin = new Marin() ;
```

Constructeurs d'une classe : exemple constructeur explicite

```
1  public class Marin {
2
3      private String nom ;
4
5      public Marin(String nom) {
6          this.nom = nom;
7      }
8
9      public String getNom() {
10         return this.nom ;
11     }
12 }
```

- On ne peut plus instancier cette classe comme précédemment. On ne peut l'instancier que de la façon suivante :

```
1  Marin marin = new Marin("Surcouf") ;
```

Constructeurs d'une classe : exemple constructeur explicite

```
1  public class Marin { // dans le fichier Marin.java
2       private String nom ;
3       // constructeur vide de la classe Marin
4       public Marin() {
5          nom = "indefini" ;
6      }
7  }
8  public class Capitaine extends Marin { // dans le fichier Capitaine.java
9       private String grade
10      public Capitaine(String grade) {
11          this.grade = grade ;
12     }
13 }
```

- Sur ce code, instancions un objet de type Capitaine :

```
1  Capitaine capitaine = new Capitaine("Capitaine de vaisseau") ;
```

Constructeurs d'une classe : exemple constructeur explicite

- L'instanciation de l'objet capitaine déclenche les opérations suivantes :
  1. le **constructeur** de la classe `Capitaine` est appelé
  2. ce constructeur **appelle** tout d'abord **implicitement** le **constructeur** vide de la classe `Marin`, qui initialise le champ `nom`
  3. le constructeur de `Capitaine` **initialise** le champ `grade`

Constructeurs d'une classe : Cas d'une erreur

- Lors qu'un objet instance d'une classe qui en étend une autre est construit, au moins un constructeur de cette super classe doit être appelé
- Si aucun appel explicite n'est écrit $\Rightarrow$ la JVM exécute le constructeur vide par défaut
- S'il n'existe pas $\Rightarrow$ la JVM génère une erreur à la compilation

```
1 Exemple ...
```

Constructeurs d'une classe : exemple ...

- Il est également possible pour un constructeur d'appeler explicitement un unique constructeur. Cet appel ne peut être que la première instruction de ce constructeur.

```
1  public class Marin { // dans le fichier Marin.java
2      private String nom ;
3      public Marin(String nom) {
4          this.nom = nom ;
5      }
6  }
7  public class Capitaine extends Marin { // dans le fichier Capitaine.java
8      private String grade
9      public Capitaine(String nom, String grade) {
10         super(nom) ; // appel du constructeur de la super classe
11         this.grade = grade ;
12     }
13     public Capitaine(String grade) {
14         this("indefini", grade) ; // appel du constructeur de meme classe
15     }
16 }
```

Plan du cours

Instanciation d'un objet

- Voyons à présent l'ensemble des opérations effectuées lors de la création d'un objet
  1. la machine Java **réserve de la mémoire** pour stocker l'objet à créer ;
  2. cette mémoire est **effacée de toute ce qu'elle pouvait contenir** auparavant : *les champs sont mis à 0, false ou null suivant leur type* ;
  3. le **constructeur** invoqué est **appelé** ;
  4. si ce constructeur appelle un autre constructeur de la même classe, alors il est appelé ;
  5. une fois que la chaîne d'appel des constructeurs a été épuisée, alors les initialiseurs de champs sont appelés ;
  6. les blocs non statiques sont exécutés ;
  7. le constructeur de la super classe dans laquelle on se trouve est exécuté ;
  8. on passe à la sous-classe suivante, en répétant les mêmes opérations dans le même ordre, jusqu'à la classe dont on construit finalement une instance.

Instanciation d'un objet : Exemple

```java
public class Marin { // dans le fichier Marin.java
    private long dateCreation = System.currentTimeMillis() ;
    {
       // ceci est un bloc non statique
       System.out.println(i) ;
    }
    private String nom ;
    public Marin() {
        this.nom = "indefini" ;
    }
    public Marin(String nom) {
        this.nom = nom ;
    }
    public String getNom() {
        return nom;
    }
    public void setNom(String nom) {
        this.nom = nom;
    }
}
```

```java
public class Capitaine extends Marin { // dans le fichier Capitaine.java
    private int grade ;
    private long dateCreation = System.currentTimeMillis() ;
    {
       // ceci est un bloc non statique
       System.out.println(i) ;
    }
    public Capitaine(String nom) {
        super(nom) ;
    }
    public Capitaine(String nom, int grade) {
        this(nom) ;
        this.grade = grade ;
    }
}
```

Instanciation d'un objet : Exemple

- Créons un objet `Capitaine` avec l'instruction suivante :

```
1 Capitaine m = new Capitaine("Surcouf", 2) ;
```

- Les opérations s'enchaînent de la façon suivante :

1. appel du constructeur `(String, int)` de `Capitaine` ;
2. appel du constructeur `(String)` de `Capitaine` ;
3. appel du constructeur `(String)` de `Marin` ;
4. initialisation de la variable `dateCreation` de `Marin` ;
5. exécution du bloc non statique de `Marin` ;
6. exécution du constructeur `(String)` de `Marin` ;
7. initialisation de la variable `dateCreation` de `Capitaine` ;
8. exécution du bloc non statique de `Capitaine` ;
9. exécution du constructeur `(String)` de `Capitaine` ;
10. exécution du constructeur `(String, int)` de `Capitaine`.

Plan du cours

Destruction d'objets

- Rappelons que l'utilisateur n'a pas à se préoccuper de la destruction de ses objets : la notion de destructeur n'existe pas en `Java`
- Il existe une méthode `finalize()` dans la classe Object qui joue le rôle de callback avant que le **garbage collector** ne détruise un objet

Plan du cours

Le mot-clé `final`

- Le mot-clé `final` peut être utilisé comme modificateur de plusieurs choses :
  1. Il peut être utilisé **sur une classe** $\Rightarrow$ il n'est pas possible de l'étendre. De nombreuses classes sont final dans l'API standard (`String`)
  2. Il peut être utilisé **sur une méthode** $\Rightarrow$ ne peut pas être surchargée
  3. Il peut être utilisé **sur un champ** d'une classe $\Rightarrow$ une fois intialisé, ce champ ne pourra plus être modifié
  4. Il peut être posé **sur un paramètre** reçu par une méthode $\Rightarrow$ ce paramètre ne pourra être modifié
  5. Il peut être posé **sur une variable** définie dans une méthode $\Rightarrow$ la valeur de cette variable ne pourra être modifiée

Plan du cours

Introduction

- Le langage Java est indissociable de sa librairie standard
  - Librairie standard, ou API standard

- Étudier le langage `Java` $\Rightarrow$ étudier les points les plus importants de cette librairie

  1. classe `Object`
     - Toutes les classes `Java` héritent de la classe `Object`
     - Toutes les méthodes de la classe `Object` sont donc disponibles dans toutes les classes `Java`

  2. classe `String`
     - Sert à coder les chaînes de caractères en `Java`

Plan du cours

La classe Object

- Voyons tout d'abord les méthodes de cette classe
- Pour l'instant, nous nous intéresserons pas des clauses `throws ... Exception`, ainsi que les méthodes `notify()`, `notifyAll()`, `wait(long)` et `wait(long, int)`

```
1  public class Object {
2      public Object() {...} // contructeur
3
4      public String toString() {...}
5
6      protected native Object clone() throws CloneNotSupportedException {...}
7
8      public equals(java.lang.Object) {...}
9      public native int hashCode() {...}
10
11     protected void finalize() throws Throwable {...}
12
13     public final native Class getClass() {...}
14
15     // methodes utilisees dans la gestion des threads
16     public final native void notify() {...}
17     public final native void notifyAll() {...}
18
19     public final void wait(long) throws InterruptedException {...}
20     public final void wait(long, int) throws InterruptedException {...}
21 }
```

Plan du cours

La méthode toString()

- Utilisée par la machine Java toutes les fois où elle a besoin de représenter un objet sous forme d'une chaîne de caractères
- Exemple

```
1 Marin marin = new Marin ("Surcouf", "Robert", 25000) ;
2 System.out.println("Marin : " + marin) ;
```

- Résultat : le nom de la classe, suvie du caractère @, et une adresse en hexadécimal, qui est l'adresse mémoire où l'objet considéré est enregistré

```
1 Marin : Marin@b82e3f203
```

- Résultat souhaité :

```
1 Marin :
2 Marin@b82e3f203
3 Nom : Surcouf
4 Prenom : Robert
5 Salaire : 25000
```

La méthode toString() : comment faire ?

- Surcharge de `toString()` dans la classe `Marin`
- Exemple de surcharge de la méthode `toString()` de la classe `Marin`

```
1 public String toString() {
2     String resultat = super.toString() ;
3     resultat += "\nNom : " + nom ;
4     resultat += "\nPrenom : " + prenom ;
5     resultat += "\nSalaire : " + salaire ;
6     return resultat ;
7 }
```

- `super.toString()` : appelle une méthode `toString()` qui doit être définie parmi les super-classes de `Marin`
  - Sur notre exemple la méthode `toString()` de la class `Object`
- Les lignes suivantes ajoutent des éléments supplémentaires.
- Résultat :

```
1 Marin@b82e3f203
2 Nom : Surcouf
3 Prenom : Robert
4 Salaire : 25000
```

Plan du cours

La méthode `clone()`

- La méthode `clone()` est une méthode déclarée **native** $\Rightarrow$ n'est pas écrite en Java, mais dans un autre langage, qui peut être le `C`, le `C++`
- La méthode `clone()` : **dupliquer** un objet rapidement, en **dupliquant la zone mémoire** dans laquelle il se trouve
- **Attention** : le clonage des objets est **interdit par défaut**
- Afin de l'utiliser, il faut **surcharger la méthode** `clone()` de la classe Object
- Il faut que la classe dont on veut cloner les instances, implémente l'interface `Cloneable` $\Rightarrow$ juste là pour autoriser le clonage

```
1  public class Marin
2    implements Cloneable { // declaration indispensable
3      // ici on propage l'exception, on aurait pu aussi
4      // l'attraper localement
5      public Object clone() throws CloneNotSupportedException {
6         return super.clone() ;
7         // ...
8     }
9  }
```

La méthode clone() : Exemple

```java
package clonables;
public class EntierA implements
      Cloneable {
   private int entier = 10;
   public int getEntier() {
      return entier;
   }
   public void setEntier(int entier) {
      this.entier = entier;
   }
   public EntierA clone() throws
         CloneNotSupportedException {
      return (EntierA)super.clone();
   }
}
```

```java
package clonables;
public class EntierB {
   private int entier = 1000;
   public int getEntier() {
      return entier;
   }
   public void setEntier(int entier) {
      this.entier = entier;
   }
}
```

```java
package clonables;
public class ObjetClonable implements Cloneable {
   private int entierLocal = 1;
   private EntierA entierA = new EntierA();
   private int[] tableau = {0, 100};
   private EntierB entierB = new EntierB();

   public ObjetClonable clone() throws
         CloneNotSupportedException {
      ObjetClonable copie = (ObjetClonable)super.
            clone();
      copie.entierA = entierA.clone();
      copie.tableau = new int[tableau.length];
      System.arraycopy(tableau, 0, copie.tableau, 0 ,
            tableau.length);
      return copie;
  }
   public int getEntierLocal() { return entierLocal;
         }
   public void setEntierLocal(int entier) { this.
         entierLocal = entier; }
   public int[] getTableau() { return tableau; }
   public EntierA getEntierA() { return entierA; }
   public EntierB getEntierB() { return entierB; }
}
```

La méthode clone() : Exemple

```
1  import clonables.ObjetClonable;
2  class EssaiClone {
3  public static void main(String arg[]) throws CloneNotSupportedException {
4      ObjetClonable I = new ObjetClonable(), J;
5      J = I.clone();
6      System.out.println("Dans l'original " + I.getEntierLocal() + " " +
7          I.getEntierA().getEntier() + " " +
8          I.getTableau()[1] + " " +
9          I.getEntierB().getEntier());
10     System.out.println("Dans la copie " + J.getEntierLocal() + " " +
11         J.getEntierA().getEntier() + " "
12         + J.getTableau()[1] + " " +
13         J.getEntierB().getEntier());
14     I.setEntierLocal(2);
15     I.getEntierA().setEntier(20);
16     I.getTableau()[1] = 200;
17     I.getEntierB().setEntier(2000);
18     System.out.println("\nApres changement de tout ce que contient l'original :");
19     System.out.println("Dans l'original " + I.getEntierLocal() + " " +
20         I.getEntierA().getEntier() + " " +
21         I.getTableau()[1] + " " +
22         I.getEntierB().getEntier());
23     System.out.println("Dans la copie " + J.getEntierLocal() + " " +
24         J.getEntierA().getEntier() + " " +
25         J.getTableau()[1] + " " +
26         J.getEntierB().getEntier());
27    }
28 }
```

La méthode `clone()` : Exemple

- On obtient à l'exécution :

```
1 Dans l'original 1 10 100 1000
2 Dans la copie  1 10 100 1000
3
4 Apres changement de tout ce que contient l'original :
5 Dans l'original 2 20 200 2000
6 Dans la copie  1 10 100 2000
```

Plan du cours

La méthode equals()

- Permet de de **comparer** deux objets pour savoir s'ils sont égaux
- Nous avions précédement que l'operateur == comparait les adresses mémoire des objets ⇒ donne **false** si deux objets identiques mais dans des adresses memoire différentes

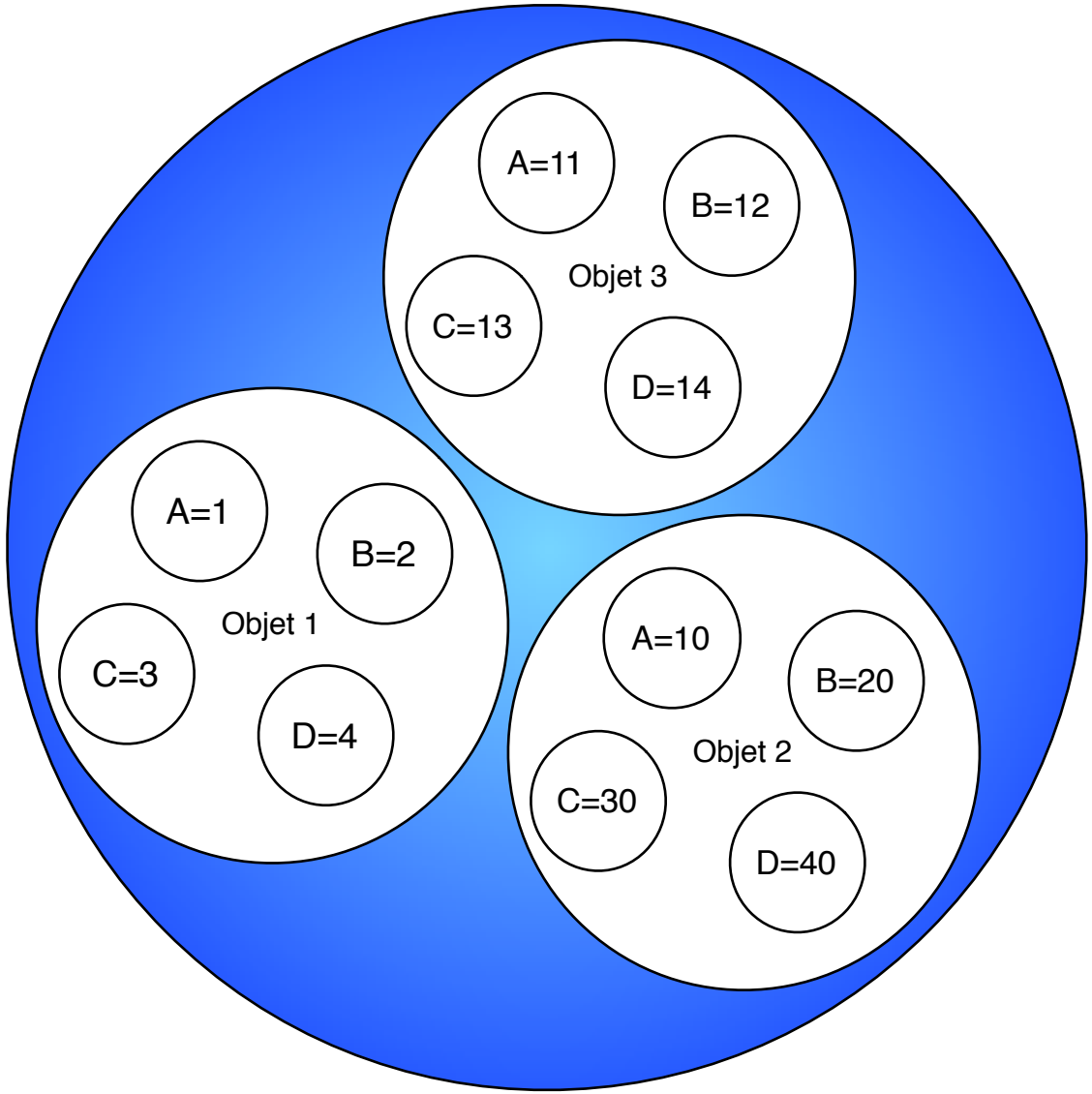
```
1  public boolean equals(Object o) {
2      if (!(o instanceof Marin))
3          return false ;
4
5      Marin marin = (Marin) o ;
6
7      return nom.equals(marin.nom) &&
8           prenom.equals(marin.prenom) &&
9           salaire == marin.salaire ;
10 }
```

La méthode `equals()` : Analysons du code

1. L'objet passé en paramètre doit étre de type `Object`
2. `instanceof` retourne systématiquement false si l'objet testé est **null** ou s'il n'est pas de type `Marin`
3. Il faut le convertir en objet de la bonne classe
   - Cette opération s'appelle un cast , elle consiste à déclarer un objet (ici marin), et à lui affecter la valeur de l'objet à convertir, en mettant devant et entre parenthèses le type dans lequel on veut faire cette conversion
4. la méthode `equals()` prend en paramètre un objet de type `Object`
5. Une fois que nous sommes sûr d'avoir un objet Marin en paramètre, alors il nous faut comparer ses champs un par un
6. nous remarquerons que la comparaison des chaînes de caractères se fait en utilisant aussi la méthode equals(), de la classe String

Plan du cours

La méthode `hashCode()`



La méthode `hashCode()`

- Pour retrouver rapidement une instance dans un ensemble, l'approche naïve consiste à parcourir toutes les instances de l'ensemble et d'appeler la méthode `equals()` sur chacun d'un $\Rightarrow$ Cela prend un temps proportionnel au nombre d'élément de l'ensemble

- Le rôle de la méthode `hashCode()` est de calculer un code numérique pour l'objet dans lequel on se trouve

- Le code numérique est censé être représentatif de l'objet

- La méthode `hashCode()` est une méthode native qui permet de calculer un nombre (**int**) unique

- Par défaut, la méthode de la classe Object retourne l'adresse à laquelle est rangé cet objet, nombre effectivement unique

La méthode `hashCode()`

- Surcharger une méthode hashCode() se fait en respectant un algorithme précis
  1. Choisir deux nombres entiers, pas trop petits, (exp. 17 et 31)
  2. On initialise l'algorithme en prenant hashCode = 17
  3. Pour chacun des autres champs c pris en compte par la méthode equals(), on construit l'entier hash suivant :
     - si c est un **booléen**, hash vaut 1 si c est true, 0 s'il est false ;
     - si c est de type **byte**, short, int ou char, alors hash vaut (int)c ;
     - si c est de type **long**, alors hash vaut `(int)(c^(c >>> 32))` ;
     - si c est de type **float**, alors hash vaut Float.floatToIntBits(f) ;
     - si c est de type **double**, alors hash vaut `Double.doubleToLongBits(f)`, et l'on prend le code de hachage du **long** que l'on récupère ;
     - si c est **nul** alors hash vaut 0 ;
     - si c est un **objet non nul**, alors hash vaut `c.hashCode()` ;
     - si c est un **tableau**, alors chacun des éléments du tableau est traité comme un champ à part entière.
  4. On met à jour `hashCode` : `hashCode = 31 * hashCode + hash`

La méthode hashCode()

```
1  public int hashCode() {
2     int hashCode = 17 ;
3     hashCode = 31 * hashCode + ((nom == null) ? 0 : nom.hashCode());
4     hashCode = 31 * hashCode + ((prenom == null) ? 0 : prenom.hashCode());
5     hashCode = 31 * hashCode + salaire;
6     return hashCode ;
7  }
```

```
1  import java.util.Date;
2
3  public class Personne {
4
5    private String nom;
6    private String prenom;
7    private long id;
8    private Date dateNaiss;
9    private boolean adulte;
10
11   public int hashCode() {
12     final int prime = 31;
13     int result = 1;
14     result = prime * result + (adulte ? 1231 : 1237);
15     result = prime * result + ((dateNaiss == null) ? 0 : dateNaiss.hashCode());
16     result = prime * result + (int) (id ^ (id >>> 32));
17     result = prime * result + ((nom == null) ? 0 : nom.hashCode());
18     result = prime * result + ((prenom == null) ? 0 : prenom.hashCode());
19     return result;
20   }
21 }
```

Plan du cours

La méthode `finalize()`

- La méthode `finalize()` est appelé par la machine `Java` juste avant l'effacement d'un objet
- L'appel de cette méthode ne se fait pas au moment où un objet n'est plus référencé, mais au moment où la machine Java décide de l'effacer
- Normalement, avec le ramasse-miettes (garbage collector) aucune fuite de mémoire ne peut avoir lieu... **Sauf que ce mécanisme ne fonctionne pas dans tous les cas**
- Il existe des cas dans lesquels un paquet d'objets qui se référencent entre eux, ne sont plus référencés par rien, et ne sont pas effacés par la machine Java

Plan du cours

La méthode getClass()

- La méthode `getClass()` retourne un objet, instance d'une classe particulière appelée `Class`

Remarque

Comme tout est objet en Java, y compris les classes elles-mêmes ! Il existe donc une classe `Class`, qui modélise les classes Java

- Cette classe est à la base des mécanismes d'**introspection**

```
1 Marin m = new Marin("Surcouf", "Robert") ;
2 System.out.println("Classe de marin : " + m.getClass()) ;
3 // Si l'on veut juste le nom de la classe, il faut invoquer sa methode
      getName().
4 System.out.println("Classe de marin : " + m.getName()) ;
```

```
1 > Classe de marin : class Marin
2 > Classe de marin : Marin
```

Plan du cours

La méthode `String`

- Classe **fondamentale** du langage Java
- Permet de **gérer** les chaînes de caractères
- Comporte une quarantaine de méthodes
- La classe `String` est déclarée **final** $\Rightarrow$ pas possible de l'étendre
- Un objet `String` contient un **tableau** de `char`, qui stocke la chaîne de caractères

Plan du cours

Construction d'un objet de type `String`

- On peut construire un objet String de deux façons

```
1 String s1 = "Bonjour le monde !" ;
2 String s11 = "Bonjour le monde !" ;
3 String s2 = new String("Bonjour le monde !") ;
```

- Différence entre les deux façons :
  - Si on évalue `(s1 == s11)` $\Rightarrow$ la valeur retournée est **true** $\Rightarrow$ la machine Java n'a en fait créé qu'un seul objet, qu'elle a affecté aux deux variables `s1` et `s11`
  - Si on évalue `(s1 == s2)`, là le résultat est `false`, on a forcé la création d'un nouvelle chaîne de caractères par un appel explicite au `new`

Plan du cours

Concaténation : utilisation de l'opérateur +

- L'opérateur de concaténation est le +

```
1 String s1 = "Bonjour" ;
2 String s2 = "le monde !" ;
3 String s3 = s1 + " " + s2 ;
```

- Les trois opérations de concaténation s'enchaînent. Une première chaîne de caractères est créée, concaténation de `s1` et `" "`, puis une deuxième, concaténation de cette première chaîne et de `s2`

- Ces opérations de concaténation peuvent mener à la création de nombreuses chaînes intermédiaires $\Rightarrow$ coût non négligeable en calculs et en ressources mémoire

- On préfèrera utiliser un `StringBuilde`

Concaténation : utilisation de StringBuilder

```
1 String s1 = "Bonjour" ;
2 String s2 = "le monde !" ;
3 StringBuilder sb = new StringBuilder() ;
4 sb.append(s1).append(" ").append(s2) ;
5 String s3 = sb.toString() ;
```

- Le résultat dans s3 est identique au précédent, mais dans ce cas, aucune chaîne de caractères intermédiaire n'a été utilisée

Concaténation de String et d'objets quelconques

- La concaténation peut opérer sur des objets qui ne sont pas des chaînes de caractères, ou encore entre une chaînes de caractères avec un objet
- Dans ce cas, la machine Java appelle la méthode `toString()` des objets à convertir

```
1 Marin marin = new Marin("Surcouf", "Robert") ;
2 String s1 = "Bonjour " + marin ;
3
4 StringBuilder sb = new StringBuilder() ;
5 sb.append("Bonjour ").append(marin) ;
6 String s2 = sb.toString() ;
```

- Les deux chaînes `s1` et `s2` contiennent des résultats identiques.
- Encore une fois, il a été plus rapide de calculer `s2` que `s1`

Bonne habitude

On doit donc privilégié l'utilisation de la méthode `StringBuilder`

Plan du cours

Extraction d'une sous-chaîne de caractères

- On peut extraire une chaîne de caractères d'une autre à l'aide de la méthode `substring()`
- Deux versions :
  1. `public String substring(int debut, int fin)`
  2. `public String substring(int debut)`
- Exemple

```
1 String s1 = "Bonjour le monde !" ;
2 String s2 = s1.substring(8) ;      // du 8eme caractere
3 String s3 = s1.substring(8, 10) ; // du 8eme au 10eme caractere
```

- A la différence de la concaténation, la méthode `substring()` est très peu coûteuse en calculs

Plan du cours

Comparaison de deux chaînes de caractères

- On peut être tenté de comparer deux strings à l'aide de l'opérateur == comme dans :

```
1 if (str1 == str2) ...
```

- Cette comparaison, bien que correcte, ne compare pas si les deux chaînes sont égales, mais si `str1` et `str2` pointent vers le même objet
- Une comparaison de chaînes s'effectue de la manière suivante :

```
1 if (str1.equals(str2)) ...
```

- Il existe aussi la méthode `compareTo()` :

```
1 str1.compareTo(str2);
```

- Cette méthode **retourne 0** si les deux chaînes sont égales, une **valeur négative** si `str1` est plus petit que `str2`, ou une **valeur positive** si `str2` est plus petit que `str1`

Plan du cours

Méthodes de comparaisons lexicographiques

- La comparaison lexicographique consiste à comparer deux chaînes caractère par caractère
- Les méthodes qui permettent de ce faire sont :
  1. `compareTo()` et `compareToIgnoreCase()`
  2. `startsWith()`, `endsWith()` : renvoient `true` si la chaîne commence ou se termine par la chaîne passée en paramètre

```
1  String s1 = "Bonjour" ; String s2 = "le monde !" ;
2
3  int c = s1.compareTo(s2) ;
4  -> c = -42 ;
5
6  c = s2.compareToIgnoreCase(s1) ;
7  -> c = 10 ;
8
9  boolean b = s1.startsWith("a") ;
10 -> b = false
```

Ordre : *du plus petit au plus grand*

A, B, C, . . ., Z, a, b, c, . . ., z

Plan du cours

Méthode de recherche de caractères

- Les méthodes de recherche de caractères ou de groupes de caractères permettent de localiser des éléments d'une chaîne de caractères

1. `indexOf()` et `lastIndexOf()` : permettent de localiser un caractère donné dans une chaîne
   - `indexOf()` : une première version permet de localiser la première occurrence d'un élément dans une chaîne
   - `indexOf()` : la seconde version prend en plus un index en paramètre, à partir duquel on recherche l'élément passé
   - Les deux méthodes retournent **-1** si l'élément n'est pas trouvé, ou une valeur entière correspondant à la position de l'élément trouvé dans la chaîne
2. `substring()` : permet d'extraire une sous-chaîne d'une chaîne donnée
3. `charAt()` : permet d'extraire un caractère à une position donnée d'une chaîne

Méthode de recherche de caractères

```
1  String str = "Hello World !";
2  int p1 = str.indexOf ("Hell"); // p1 vaut 0
3  int p2 = str.indexOf ("World"); // p2 vaut 6
4  int p3 = str.indexOf ("z");     // p3 vaut -1
5  int p4 = str.indexOf ("o");     // p4 vaut 4
6  int p5 = str.indexOf ("o", 5); // p5 vaut 7
7  int p6 = str.lastIndexOf ("o");    // p6 vaut 7
```

```
1  String str = "Hello World !";
2  String s1 = str.substring (0);     // s1 vaut "Hello World !"
3  String s2 = str.substring (4);     // s2 vaut "o World !"
4  String s3 = str.substring (0, 4);    // s3 vaut "Hell"
```

```
1  String str = "Hello World !";
2  System.out.print (str.charAt (4)); // affiche "o"
```

Plan du cours

Méthode de modification de chaîne

- Les méthodes de modification de chaînes retournent toutes une nouvelle chaîne, sans modifier la chaîne originale
- Elles peuvent référencer des portions du tableau de caractères qui contient la chaîne originale
  1. `replace()` : cette méthode prend des variables char en paramètre, et se borne à remplacer toutes les occurrences du premier caractère par le second
  2. `replaceFirst()` et `replaceAll()` : ces deux méthodes prennent deux chaînes de caractères en paramètre. La première contient une expression régulière. Toutes les occurrences de cette expression sont remplacées par la deuxième chaîne
  3. `toUpperCase()`, `toLowerCase()` : retourne une chaîne résultat de la mise en majuscules ou en minuscules de la chaîne originale
  4. `trim()` : retourne une chaîne de laquelle les caractères blancs en tête ou en début de chaîne ont été retirés

Méthode de modification de chaîne

```
1 String str = "Hello World !";
2 String s1 = str.replace ('o', 'e');    // s1 vaut "Helle Werld !"
3 String s2 = str.replace ("ll", "l");    // s2 vaut "Helo World !"
```

```
1 String Str = new String("Welcome to University Paris 13");
2
3 System.out.print("Return Value :" );
4 System.out.println(Str.replaceFirst("(.*)Paris(.*)","PARIS" )); // Affiche
      "Return Value :PARIS"
5
6 System.out.print("Return Value :" );
7 System.out.println(Str.replaceFirst("Tutorials", "AMROOD" )); // Affiche "
     Return Value :Welcome to University PARIS 13"
```

```
1 String str = "      Hello World ! ";
2 String s1 = str.toLowerCase();    // s1 vaut "      hello world ! "
3 String s2 = str.toUpperCase();    // s2 vaut "      HELLO WORLD ! "
4 String str = "      Hello World ! ";
```

Plan du cours

Méthode de duplication

1. `getBytes()`, `getChars()` : retourne un tableau de `byte` ou de `char` correspondant à cette chaîne de caractères

2. `split()` : découpe une chaîne de caractères à l'aide d'une expression régulière passée en paramètre. Le résultat du découpage est retourné dans un tableau de `String`

3. `toCharArray()` : retourne un tableau de char correspondant à cette chaîne de caractères

Plan du cours

Plan du cours

Opérateurs

Définition

Les opérateurs permettent d'effectuer une opération bien définie sur des valeurs, appelées opérandes, en produisant un résultat appelé valeur qui est une donnée d'un certain type

- De nombreux opérateurs en Java
  1. les opérateurs **unaires** requièrent un unique opérande ;
  2. les opérateurs **binaires** en requièrent deux ;
  3. et enfin les opérateurs **ternaires** en nécessitent trois.

Opérateurs

- Java dispose d'un ensemble d'opérateurs
- Chaque opérateur dispose d'une priorité
- Plus l'indice de priorité est élevé, plus l'opérateur est prioritaire

Symbole	Note	Priorité	Associativité
`++a --a`	Préincrémentation, prédécrémentation	16	Droite à gauche
`a++ a--`	Postincrémentation, postdécrémentation	15	Gauche à droite
$\sim$	Inversion des bits d'un entier	14	Droite à gauche
`!`	Non logique pour un booléen	14	Droite à gauche
`- +`	Moins et plus unaire	14	Droite à gauche
`(type)`	Conversion de type (cast)	13	Droite à gauche
`* / %`	Opérations multiplicatives	12	Gauche à droite
`- +`	Opérations additives	11	Gauche à droite
$\ll \gg \ggg$	Décalage de bits, à gauche et à droite	10	Gauche à droite
`instanceof < <= > >=`	Opérateurs relationnels	9	Gauche à droite
`== !=`	Opérateurs d'égalité	8	Gauche à droite
`&`	Et logique bit à bit	7	Gauche à droite
`^`	Ou exclusif logique bit à bit	6	Gauche à droite
`|`	Ou inclusif logique bit à bit	5	Gauche à droite
`&&`	Et conditionnel	4	Gauche à droite
`||`	Ou conditionnel	3	Gauche à droite
`? :`	Opérateur conditionnel	2	Droite à gauche

- La signification de l'ordre de priorité est classique. La multiplication a une priorité plus forte que l'addition, donc `2 + 3*4` vaudra bien 14 et non pas 20 si l'addition avait été calculée la première

Opérateurs

- Les opérateurs ++ et −− : appelés respectivement post et préincrémentation ou post et prédécrémentation

```
1  int i = 5;
2  int j = i++;
3  //Here, i will contain the value 6, but j will contain the value 5.
4
5  int i = 5;
6  int j = ++i;
7  //Here, i will contain the value 6, and j will also contain the value
       6.
```

- Les opérateurs % et / :
  - Le signe / est utilisé pour les deux divisions : entière et flottante. Dans le cas de la division entière, le résultat est le quotient de la division (en entier, 7/3 vaut 2)
  - L'opérateur % (modulo) donne le reste de la division entière du numérateur par le dénominateur. Ainsi, 7%3 vaut 1.

Opérateurs

- Les opérateurs $\ll$ et $\gg$:
  - Les opérateurs $\ll$ et $\gg$ fonctionnent sur des `int`, et réalisent un décalage à gauche ou à droite du nombre de bits indiqué

```
1 int i = 8 >> 2 ; // la valeur finale de i est 2
```

  - Le décalage de bits à droite revient à diviser un entier par 2, et à gauche à le multiplier par deux. Ces opérations sont souvent utilisées en tant qu'optimisation

- L'opérateur $\ggg$ est un décalage à droite non signé, ce qui signifie que le bit de signe n'est pas propagé

Opérateurs

- L'opérateur `instanceof` : L'opérateur `instanceof` est une originalité du langage Java
  - `instanceof` s'utilise dans un cas bien précis : quand on veut tester si un objet appartient à une classe donnée

- Les opérateurs `&`, `|` et `^` :
  - Ces opérateurs correspondent respectivement au **AND**, **OR** et **XOR** logiques
  - Ils peuvent opérer sur des types booléen, ou entier, auquel cas ils fonctionnent en bit à bit

- Les opérateurs `&&` et `||`
  - Ces deux opérateurs correspondent respectivement au **AND** et **OR** logiques
  - Ne peuvent opérer que sur des booléens

Opérateurs

- L'opérateur ? ... : ... : appelé "opérateur ternaire"
  - Syntaxe :

```
1  Action action = ilPleut() ? ouvertureParapluie() :
       fermetureParapluie() ;
```

  - Dans notre exemple, si le retour de la méthode `ilPleut()` vaut **true**, alors l'objet `action`, de type `Action` prend la valeur de retour de la méthode `ouvertureParapluie()`. Dans le cas contraire, c'est l'autre méthode qui est utilisée

  - Formellement, cette façon d'écrire est équivalente à :

```
1  Action action ;
2  if (ilPleut()) {
3     action = ouvertureParapluie() ;
4  } else {
5     action = fermetureParapluie() ;
6  }
```

Plan du cours

Ordre d'exécution

- Les opérateurs et l'ordre d'exécution des calculs ont été spécifiés très précisément en Java

```
1  int j = i + tab[i] + fonction() ;
```

- Les spécifications nous disent :
  1. S'il y a un élément de tableau, les éléments entre crochets ([]) sont calculés en premier
  2. Lors d'un appel de méthode du type : objet.methode(argument), objet est calculé en premier, puis methode, puis argument. Ce point est utilisé lorsque les mécanismes d'héritage sont utilisés
  3. Lors de l'allocation d'un tableau à plusieurs dimensions, les expressions sont calculées une par une, de gauche à droite

Plan du cours

Plan du cours

Introduction

- Les tableaux ont plusieurs particularités en `Java`
  - Le premier indice est `0`, comme en `C / C++`
  - Ce sont presque des objets. Ils ne peuvent pas être étendus, n'ont pas de nom de classe, sont utilisés avec une syntaxe qui leur est propre, et ils héritent des méthodes de la classe `Object`, notamment `toString()`
  - Une variable de type tableau est une référence vers un objet
  - La validité des indices d'un tableau est systématiquement vérifiée lors de l'exécution d'un code Java. Il n'est pas possible de lire au-delà du dernier indice existant d'un tableau. Si un code tente de le faire, la machine Java génère une exception

Plan du cours

Création d'un tableau

- La création d'un tableau se déroule en deux temps : la déclaration et la création
  1. **Déclaration** : déclarer cette variable est comme déclarer un objet. Aucun espace mémoire n'est réservé, autre celui qui permet de stocker cette variable

```
1 int [] tab1 ; // declaration d'un tableau pouvant contenir des
      entiers
2 int tab2 [] ; // declaration equivalente a la precedente
```

  2. **Création** : Réserver de la mémoire pour stocker un tableau. Nécessite de fournir la taille de ce tableau. Une fois cette taille fixée, il n'est plus possible de la modifier

```
1 int nombreMarins = 10 ; // taille initiale du tableau
2 Marin [] marins = new Marin [nombreMarins] ; // creation du
      tableau
3 for (int i = 0 ; i < marins.length ; i++) { // le champ length
      donne la taille du tableau
4     marins[i] = new Marin() ; // initialisation de chaque case
          du tableau
5 }
```

Création d'un tableau

Important

Créer un tableau d'objets n'initialise pas les objets qu'il contient

```
1 Marin [] marins = new Marin[10] ;
2 String nom = marins[3].getNom() ; // ERREUR !!!!!
```

Plan du cours

Initialisation d'un tableau

- Il est possible d'initialiser un tableau avec des valeurs explicites

```
1 int tableau [] = {0, 1, 2, 3,  4} ; // on peut aussi placer les []
      juste apres int
2 int [] autreTableau ;
3 autreTableau = new int [] {0, 1, 2,  3,  4} ;
```

- Les deux premières syntaxe ne peut être utilisée que lors de la déclaration du tableau
- La dernière syntaxe est utilisable une fois que le tableau a été déclaré
- Il est possible d'utiliser des expressions entre les paires d'accolades

```
1 Marin marins [] ;
2 marins = new Marin [] {
3      new Marin(),
4      new Marin("Surcouf"),
5      null
6 } ;
```

Plan du cours

Initialisation d'un tableau

- Comme il a déjà été dit, les méthodes de la classe Object sont accessibles sur toute variable de type tableau
- Malheureusement, il n'est pas conseillé d'utiliser les méthodes `toString()`, `equals()` ni même `hashCode()`

```
1 int [] tab1 = {0, 1, 2, 3, 4} ;
2 int [] tab2 = {0, 1, 2, 3, 4} ;
3
4 tab1.toString() // -> [I@18d107f
5 tab1.equals(tab2) // -> false
```

Plan du cours

Initialisation d'un tableau

- Comme dans la plupart des langages, les tableaux multidimensionnels sont en fait des tableaux de tableaux

- La syntaxe pour créer de tels tableaux se déduit aisément :

```
1 int [][] tableauBidi ;
2 tableauBidi = new int [10][5] ;
```

- La syntaxe pour initialiser un tableau bidimensionnel se déduit elle aussi :

```
1 int tableauBidi [][] ;
2 tableauBidi = new int [][] {
3     {0, 1, 2},
4     {1, 2, 3},
5     {2, 3, 4}
6 } ;
```

Initialisation d'un tableau

- Il est également possible d'initialiser un tableau ligne par ligne

```
1  int tableauBidi [][] = new int [3][] ;
2
3   for (int i = 0 ; i < tableauBidi.length ; i++) { // tableauBidi.
        length vaut 3
4      tab[i] = new  int [] {i, i + 1, i + 2} ;
5  }
```

- Dans la mesure où tableauBidi est un tableau de tableau, tableauBidi.length est défini, de même que tableauBidi[i].length, qui, dans notre exemple, vaut 3 pour le valeurs de i : 0, 1, 2

Plan du cours

Copie de tableaux

- La copie d'un tableau dans l'autre peut toujours se faire par itération sur les éléments du premier tableau, et recopie dans le second
- l'API standard de Java nous fournit une méthode particulièrement rapide, qui fonctionne par copie de zone mémoire, sur le modèle du clonage d'objets
- La méthode à invoquer est System.arraycopy, dont voici la signature

```
1  public static void arraycopy(Object src, int src_pos, Object dest,
       int dest_pos, int length)
```

- Les objets src et dest doivent être des tableaux de même type, sans quoi une exception est générée. Ces deux tableaux doivent avoir été correctement déclarés et initialisés

Copie de tableaux

- Lors de cette opération, la machine Java va tenter de copier `length` éléments du tableau `src` à partir de l'index `src_pos`, vers le tableau `dest`, à partir de l'index `dest_pos`
- Si un dépassement de capacité a lieu, une exception est générée
- Ce dépassement peut avoir lieu en lecture (si `src_pos + length` dépasse la taille du tableau `src`), ou en écriture (si `dest_pos + length` dépasse la taille du tableau `dest`)

- Exemple :

```
1 int tab1 [] = new int [] {0, 1, 2, 3, 4, 5} ;
2 int tab2 [] = new int [] {0, 10, 20, 30, 40} ;
3
4 System.arraycopy(tab1, 1, tab2, 1, 2) ; // tab2 -> {0, 1, 2, 30, 40}
```

Plan du cours

Plan du cours

Blocs

- Par définition, un bloc est un ensemble de commandes, d'instructions et de déclarations compris entre deux accolades
- Un bloc peut se trouver dans une classe, en tant que membre (cas des blocs statiques et non statiques, que nous avons déjà vus), ou dans une méthode
- On peut définir des blocs dans des blocs
- Un bloc définit la portée d'une variable $\Rightarrow$ Une variable définie à l'intérieur d'un bloc n'est pas connue à l'extérieur de ce bloc
- Une variable définie dans un bloc ne peut pas avoir le même nom qu'une autre variable définie dans un bloc englobant

Blocs : Exemple

```
1  public class Marin {
2      private String nom ;
3      static {
4          // ceci est un bloc statique
5      }
6      {
7          // ceci est un bloc non statique
8      }
9      public void setNom(String nom) {
10         String nom ; // ERREUR !!! impossible de definir une variable portant le nom d'un èparamtre
11          this.nom = nom ;
12         {
13              // nous sommes dans un bloc dans la émthode setNom(String)
14              int i ;
15             {
16                 // nous sommes dans un sous-bloc de ce bloc
17                 int i = 0 ; // ERREUR !!! il existe une variable i dans le bloc englobant
18             }
19         }
20         i = 0 ; // ERREUR !!! i n'est pas connue, nous ne sommes pas
21                 // dans son bloc de édfinition
22     }
23     public void augmenteSalaire(int montant) {
24
25         for (int i = 0 ; i < 10 : i++) {
26             // bloc de la boucle for, i est definie dans ce bloc
27             int i = 0 ; // ERREUR !!! impossible de definir i, qui existe deja
28         }
29         System.out.println("i = " + i) ; // ERREUR !!! i n'est pas connue, nous sommes sorti de la
               boucle for
30     }
31 }
```

Plan du cours

Mots-clés réservés

- Il existe une cinquantaine de mots réservés dans le langage Java
- Un bloc peut se trouver dans une classe, en tant que membre (cas des blocs statiques et non statiques, que nous avons déjà vus), ou dans une méthode

abstract	class	extends	implements	null	strictfp	true
assert	const	false	import	package	super	try
boolean	continue	final	instanceof	private	switch	void
break	default	finally	int	protected	synchronized	volatile
byte	do	float	interface	public	this	while
case	double	for	long	return	throw	
catch	else	goto	native	short	throws	
char	enum	if	new	static	transient	

Plan du cours

Tests : if

- Ces deux instructions sont quasiment identiques à celles du `C` ou du `C++`

- Le **if** permet de tester si une valeur est vraie (**true**) ou fausse (**false**)

  - Si elle est vraie, alors certaines instructions, regroupées dans un bloc, sont exécutées, sinon ce sont d'autres instructions, optionnelles, qui le sont

```
1 if (expression éboolenne)
2     commandes on bloc de commandes
3  [ else
4     commandes on bloc de commandes ]
```

- Un bloc de commande est une suite de commandes encadrée par des accolades, comme nous l'avons déjà vu. Il est une bonne habitude de programmation de systématiquement utiliser des blocs dans un **if**, même pour n'enserrer qu'une unique commande

Tests : switch

- Le switch fonctionne un peu différemment
- Il s'agit d'une instruction de branchement, qui "branche" le code sur une instruction en fonction d'une valeur, constante, que prend le paramètre du `switch`
- La syntaxe du `switch` est la suivante :

```
1 switch (expression)
2     case constante1 : commande1 ;
3     [ break ; ]
4     [ case constante2 : commande2 ;
5     [ break ; ]]
6     [ default : commande ; ]
```

- Les valeurs `constante1` , `constante2` , etc... sont des valeurs constantes

Tests : switch

- Ce peut donc être :
  - des valeurs écrites en dur dans le code, ce qui est évidemment à proscrire !
  - des valeurs constantes, donc déclarées comme étant final. Ces valeurs peuvent être importées statiquement
  - des valeurs énumérées
- Les commandes sont des instructions Java classiques, rangées ou non dans un bloc, cela n'a pas d'importance (en général elles ne le sont pas)
- La machine Java teste séquentiellement si expression a pour valeur constante1 puis constante2 , etc. . .
- Dès qu'elle rencontre une valeur qui correspond, alors elle exécute les commandes qui se trouvent à la suite de cette valeur
- Enfin, si aucun case ne correspond à la valeur d'expression, alors le branchement s'effectue sur default, s'il existe. En général, default est mis en dernier, après tous les case, mais ce n'est pas une obligation

Tests : switch

- Exemple :

```
1  public void testBreak(Civility civility) {
2      switch (civility) {
3
4          case MADAME :
5             System.out.println("Madame") ;
6          case MADEMOISELLE :
7             System.out.println("Mademoiselle") ;
8
9          break ;
10
11         case MONSIEUR :
12            System.out.println("Monsieur") ;
13
14         default :
15            System.out.println("Default") ;
16     }
17
18     System.out.println("Sortie de la émthode") ;
19 }
```

Tests : switch

```
1  testBreak(Civility.MADAME) ;
2   > Madame
3   > Mademoiselle
4   > Sortie de la émthode
5
6   testBreak(Civility.MADEMOISELLE) ;
7   > Mademoiselle
8   > Sortie de la émthode
9
10  testBreak(Civility.MONSIEUR) ;
11  > Monsieur
12  > Default
13  > Sortie de la émthode
```

Plan du cours

Boucles : for, while, do ... while

- Il existe trois contrôles de boucles en `Java` :
  - le for pour les boucles indexées
  - le while pour les boucles avec test avant l'itération
  - le do ... while pour les boucles avec test en fin d'itération
- La syntaxe du **for** est la suivante :

```
1 for (initialisation ; test ; incrementation)
2   commandes on bloc de commandes
```

- Le for fonctionne précisément de la façon suivante :
  1. initialisation est exécuté ;
  2. test est évalué, s'il est false, le programme sort de la boucle ;
  3. la commande, ou le bloc de commandes du for est exécuté ;
  4. incrémentation est exécuté ;
  5. test est évalué, s'il est vrai alors on reprend le processus en 3, sinon on sort de la boucle.

Boucles : for, while, do ... while

```
1  // boucle infinie
2   for ( ; ; ) {
3      ...
4  }
5
6   // boucle classique
7   for (int i = 0 ; i < 100 ; i++) {
8      ...
9  }
10
11  // iteration sur les elements d'un tableau
12  int [] tableau = new int [10] ;
13  for (int i = 0 ; i < tableau.length ; i++) {
14     ...
15 }
16
17  // boucle a double index
18  for (int i = 0, j = 1 ; i < 100 ; i += 2, j += 2) {
19     ...
20 }
```

Boucles : for, while, do ... while

- La syntaxe du **while** est plus simple que celle du for :

```
1 while (expression) commandes
```

- Le fonctionnement de cette commande est le suivant :
  1. expression est évalué, s'il est false alors on sort de la boucle ;
  2. commande est exécuté, et on reprend en 1.

Boucles : for, while, do ... while

```
1 int compteur = 1;
2 while (compteur <= 5)
3 {
4     System.out.println (compteur);
5     compteur++;
6 }
7 System.out.println ("Done");
```

Boucles : for, while, do ... while

- La syntaxe du `do ... while` et la suivante :

```
1 do commandes while (expression)
```

- Le fonctionnement de cette commande est le suivant :
  1. commande est exécuté ;
  2. expression est évalué, s'il est false alors on sort de la boucle, sinon on reprend en 1.

- La différence fondamentale entre le **while** et le **do ... while**, est que le **do ... while** exécute toujours, au moins une fois, son bloc de commande.

Boucles : for, while, do ... while

```
1  int compteur = 0;
2  do
3  {
4      compteur++;
5      System.out.println (compteur);
6  }
7  while (compteur < 5);
8  System.out.println ("Done");
```

Plan du cours

Plan du cours

Héritage

- Nous avons déjà utilisé de nombreuses classes dans ce cours, et également utilisé le mécanisme de l'héritage
- Une classe **B** qui *hérite* d'une classe **A** est une sous-classe de **A**, et **A** est la *super-classe* de **B**
- La classe **java.lang.Object** est la *super-classe* de toutes les classes Java, directement ou indirectement

```
1  public class A { // éédclare dans le fichier A.java
2      ...
3  }
4
5  public class B extends A { // éédclare dans le fichier B.java
6      ...
7  }
```

Conséquences pour les membres

- Nous avons déjà utilisé de nombreuses classes dans ce cours, et également utilisé le mécanisme de l'héritage
- Une classe **B** qui *hérite* d'une classe **A** est une sous-classe de **A**, et **A** est la *super-classe* de **B**
- La classe **java.lang.Object** est la *super-classe* de toutes les classes Java, directement ou indirectement

```
1  public class A { // éédclare dans le fichier A.java
2      ...
3  }
4
5  public class B extends A { // éédclare dans le fichier B.java
6      ...
7  }
```

Plan du cours

Conséquences pour les membres

- Toutes les membres public et protected d'une classe parent sont accessibles à ses classes enfant
- Une classe parent ne peut accéder à ses classes enfant
- Si un champ d'une classe enfant porte le même nom qu'un champ d'une classe parent $\Rightarrow$ le champ enfant masque le champ parent
- Par défaut, le code de la classe enfant accède au champ enfant
- Il est possible pour un code de la classe enfant de lire le champ de la classe parent en utilisant le mot-clé **super**

Conséquences pour les membres

```
1  public class A { // éédclare dans le fichier A.java
2      protected String nom = "Je suis dans A" ;
3
4      public void uneMethode() {
5          System.out.println(nom) ; // imprime "Je suis dans A"
6      }
7  }
8
9  public class B extends A { // éédclare dans le fichier B.java
10     protected String nom = "Je suis dans B" ;
11
12     public void uneAutreMethode() {
13         System.out.println(nom) ; // imprime "Je suis dans B"
14         System.out.println(super.nom) ; // imprime "Je suis dans A"
15     }
16 }
```

Conséquences pour les membres

```
1  public class Main { // éédclare dans le fichier Main.java
2
3      public static void main(String... args) {
4          A a = new A() ; // la classe A est celle que nous venons de édfinir
5          B b = new B() ; // idem
6          A ba = b ;      // cette declaration est legale, car b est un
                element de A,
7                          // du fait que B etend A ;
8                          // b et ba designent le meme objet
9          System.out.println("a.nom = " + a.nom) ; // a.nom = Je suis dans A
10         System.out.println("b.nom = " + b.nom) ; // b.nom = Je suis dans B
11         System.out.println("ba.nom = " + ba.nom) ; // ba.nom = Je suis
                dans A
12     }
13 }
```

Overloading et overriding

```
1  public class A { // declaree dans le fichier A.java
2      public String ouSuisJe() {
3       return "Je suis dans A...";
4      }
5  }
6  public class B extends A { // declaree dans le fichier B.java
7      public String ouSuisJe() {
8         return "Je suis dans B...";
9     }
10 }
11 public class Tests { // declaree dans le fichier Main.java
12     public static void main(String[] args) {
13         A a = new A() ; // memes declarations que dans l'exemple ééprcdent
14         B b = new B() ;
15         A ba = b ;
16         System.out.println("a.ouSuisJe ? " + a.ouSuisJe()) ; // a.ouSuisJe
               ? Je suis dans A !
17         System.out.println("b.ouSuisJe ? " + b.ouSuisJe()) ; // b.ouSuisJe
               ? Je suis dans B...
18         System.out.println("ba.ouSuisJe ? " + ba.ouSuisJe()) ; // ba.
               ouSuisJe ? Je suis dans B...
19     }
20 }
```

Empêcher l'héritage

```
1  public final class A { // declaree dans le fichier A.java, ne peut etre
       etendue
2      public void ouSuisJe() {
3          System.out.println("Je suis dans A !") ;
4      }
5  }
6
7  public B { // declaree dans B.java
8      public final void ouSuisJe() { // ne peut etre surchargee, bien que
9                                     // B puisse etre étendue
10         System.out.println("J'y suis j'y reste !") ;
11     }
12 }
```

Plan du cours

Introduction

- La notion d'**interface** est absolument centrale en `Java`
- Massivement utilisée dans le design des `API` du `JDK` et de `JEE`
- Utilisée pour représenter des propriétés transverses de classes
- Là où une classe abstraite doit être étendue et spécialisée, une interface nous dit juste que telle classe possède telle propriété, indépendamment de ce qu'elle représente

Exemple

Soit une hiérarchie de classes dont le but est de coder des moyens de locomotion. On peut imaginer une classe **Transport**, classe de base de laquelle toutes les autres classes vont hériter. Puis des classes **Avion**, **Voiture**, **Moto**, **Camion**, etc...
Nos moyens de locomotion ont besoin de faire le plein. Pour cela ils se rendent dans une station service. Cette station service possède une méthode **faireLePlein(...)**, censée prendre *un moyen de transport en paramètre*. Ecrivons tout d'abord notre jeu de classes.

Exemple : Les classes "moyen de locomotion"

```
1  public class Transport {
2      public void roule() ;
3  }
4
5   public class Voiture extends Transport {
6      public void conduit() ;
7  }
8
9   public class Avion extends Transport {
10     public void vole() ;
11 }
12
13  public class Moto extends Transport {
14     public void seFaufile() ;
15 }
16
17  public class Velo extends Transport {
18     public void pedale() ;
19 }
```

Exemple : la classe StationService

- Codons à présent notre station service, notamment sa méthode **faireLePlein(...)**
- On pourrait penser que cette méthode **faireLePlein(...)** peut prendre un objet instance de Transport en paramètres, après tout cette classe est la super classe de toute notre hiérarchie
- Malheureusement, dans la hiérarchie de **Transport**, il y a la classe **Velo**, et un vélo ne fréquente pas les stations service.

```
1  public class StationService {
2      public void faireLePlein(Transport transport) {
3          if (transport instanceof Velo) {
4              // ne pas faire le plein
5          } else {
6              // faire le plein
7          }
8      }
9  }
```

Exemple : la classe StationService

- Cette méthode fonctionne dans notre cas, mais elle est **catastrophique**
- Si un autre développeur reprenne notre code, et sans connaître l'implémentation de **StationService**, écrive une autre extension de **Transport**, **Tricycle**. Si un tricycle se présente à la station service, notre système aura un problème...
- Le principal problème de cette approche est qu'il faut modifier le code de cette méthode à chaque fois que l'on ajoute des classes dans la hiérarchie de Transport
- C'est là que les interfaces entrent en jeu et nous aident à résoudre notre problème.
- Écrivons une **interface Motorise**, et utilisons-la dans notre hiérarchie d'objets.

Exemple : Les classes “moyen de locomotion”

```
1  public interface Motorise { // notre interface
2      public void faisLePlein() ;
3  }
4
5  public class Transport { // une instance de Transport ne sait pas toujours faire le plein
6      public void roule() {}
7  }
8
9  public class Voiture extends Transport implements Motorise {
10     public void conduit() {}
11     public void faisLePlein() {}
12 }
13
14 public class Avion extends Transport implements Motorise {
15     public void vole() ;
16     public void faisLePlein() {}
17 }
18
19 public class Moto extends Transport implements Motorise {
20     public void seFaufile() ;
21     public void faisLePlein() {}
22 }
23
24 public class Velo extends Transport { // ne sait pas faire le plein
25     public void pedale() ;
26 }
```

Ecriture de StationService avec Motorise

- On peut alors écrire notre classe **StationService** de la façon suivante
- Le code ne dépend plus des classes de la hiérarchie de Transport
- La classe **StationService** accepte toute instance d'une classe qui possède une méthode **faisLePlein()**, dont l'existence est spécifiée par l'interface **Motorise**
- Que cette classe soit une extension de **Transport** ou non n'a pas d'importance

```
1 public class StationService {
2     public void faireLePlein(Motorise motorise) {
3        motorise.faisLePlein() ;
4     }
5 }
```

Définition

- Une interface s'écrit comme une classe
- La différence que l'on remplace le mot-clé class par **interface**
- Une interface ne peut pas posséder de méthode concrète, ni de paramètres

```
1 public interface Motorise { // éédclare dans Motorise.java
2     public void faisLePlein() ;
3 }
```

- Une interface peut en étendre une autre, et même plusieurs. Elle ne peut pas étendre de classe
- Une classe n' étend pas une interface, elle l'implémente
- Une classe peut implémenter autant d'interfaces que l'on veut

```
1 public class Voiture implements Motorise {
2     public void faisLePlein() {
3         // corps de la classe
4     }
5 }
```

Définition de constantes dans les interfaces

- Les interfaces peuvent aussi être utilisées pour définir des constantes
- Une façon de faire est de placer ces constantes dans une interface
- Toutes les classes qui ont besoin d'accéder à ces constantes n'ont plus qu'à déclarer cette interface dans leur clause implements

```
1  public interface Constantes { // dans le fichier Constantes.java
2      public static final double G = 9.81 ;
3  }
4
5   public class ChampGravitationnel // dans le fichier
        ChampGravitationnel.java
6   implements Constantes {
7      private double vitesse ;
8
9      public double calculeVitesse(double temps) {
10         return G*temps ;
11     }
12 }
```

Utilité des interfaces

- Les interfaces sont massivement utilisées dans les API Java, que ce soit celles du JDK ou les API avancées de JEE (par exemple)

Plan du cours

Introduction

- Java préconisait l'écriture d'une classe par fichier comme "*bonne habitude de programmation*"
- Un projet complet pouvant être constitué d'un grand nombre de **classes** $\Rightarrow$ il devient vite nécessaire de pouvoir les ranger, les trier, les structurer convenablement
- Analogue à celui que l'on rencontre lorsque l'on veut classer un très grand nombre de fichiers sur un disque $\Rightarrow$ *on crée une structure de répertoires*
- Java propose la même approche : ranger ses classes dans des répertoires
- plutôt que de parler de répertoire, on parle de paquet (**package**)
- Permettent de ranger des classes Java dans une structure **hiérarchique** que l'on peut définir soi-même

Déclaration d'appartenance à un paquet

- Déclarer qu'une classe appartient à un paquet est une chose très simple : il suffit de mettre en première ligne de cette classe la directive package.

```
1  package transport ; // la classe Voiture appartient au paquet transport
2
3   public class Voiture { // érange dand un fichier transport/Voiture.java
4      // corps de la classe
5  }
```

- La convention de rangement des fichiers impose que le fichier **Voiture.java** soit rangé dans un répertoire transport
- Il est bien sûr possible de créer des sous-paquets dans des paquets parent. Il suffit pour cela de le déclarer dans la directive package

```
1  package transport.motorise ; // la classe Voiture appartient au paquet transport.motorise
2
3   public class Voiture { // érange dans un fichier transport/motorise/Voiture.java
4      // corps de la classe
5  }
```

Introduction

- le compilateur **javac** ne fait pas ce rangement automatiquement, si l'on compile avec la ligne de commande, il faut le faire manuellement, en respectant bien sûr les majuscules / minuscules au niveau des noms de répertoires

- Le nom complet d'une classe **Voiture** rangée dans le paquet **transport.motorise** sera **transport.motorise.Voiture**. Le fichier compilé **Voiture.class** se trouvera lui dans le répertoire **transport/motorise**

- La bibliothèque standard Java est entièrement structurée de cette façon là

Chargement d'une classe

```
1  package vehicule.motorise ;
2
3   import java.util.* ; // importation de tout le package java.util, dont la
          classe Date
4
5   public class Voiture {
6
7       private Date dateAchat ;
8
9  }
```

- Cette directive indique à la machine Java qu'elle doit aller regarder dans le répertoire **java/util** pour trouver les classes utilisées dans ce fichier source
- Si elle rencontre une classe inconnue, et qu'elle ne la trouve pas dans **java/util**, elle génèrera une erreur à la compilation
- La machine Java ne cherche les classes que dans le répertoire **java/util**, elle ne descend pas dans les sous-niveaux d'arborescence

Chargement d'une classe

- Notons également que l'on aurait pu importer la classe **Date** explicitement avec la directive suivante

- Le répertoire **java.lang** est toujours importé par défaut, il n'y a pas besoin de le déclarer explicitement

```
1  package vehicule.motorise ;
2
3   import java.util.Date ; // importation d'une classe unique
4
5   public class Voiture {
6
7       private Date dateAchat ;
8
9  }
```

Choix de nom

- Nom d'une classe
  - Le nom d'une classe reste le nom qui est déclaré dans la directive **public class** mais on ne peut y accéder que grâce à son chemin complet, tout comme un fichier

```
1  public class Voiture {
2
3      private java.util.Date dateAchat ;
4      private java.sql.Data dateMiseEnCirculation ;
5
6  }
```

Choix de nom

- Nom d'un package
  - Il est bien sûr possible d'ajouter des classes dans une hiérarchie existante de packages
  - Il est possible aussi de créer de nouvelles branches d'une hiérarchie existante
  - Il est cependant dangereux de nommer ses branches au hasard ou avec des noms trop génériques
  - Les spécifications de Java préconisent donc un pattern pour nommer sa hiérarchie, que tout le monde respecte, y compris et surtout, les éditeurs de logiciels
  - Cette convention consiste, pour une société de développement, à prendre son nom de domaine lu à l'envers comme base de sa hiérarchie
  - Ainsi, **IBM**, propriétaire de **ibm.com**, placera les classes de ses applications et bibliothèques dans une hiérarchie **com.ibm**
  - La bibliothèque standard utilise deux hiérarchies qu'il est interdit de modifier : **java.** et **javax.**

Plan du cours

Introduction

- Tout programme comporte des erreurs (*bugs*) ou est susceptible de générer des erreurs (e.g suite à une action de l'utilisateur, de l'environnement, etc ...)

- Le langage Java inclus plusieurs mécanismes permettant d'améliorer la fiabilité des programmes :

- Une exception caractérise le déroulement **non nominal** d'un programme

  - les exceptions pour la robustesse ;

  - les assertions pour la correction.

Les exceptions

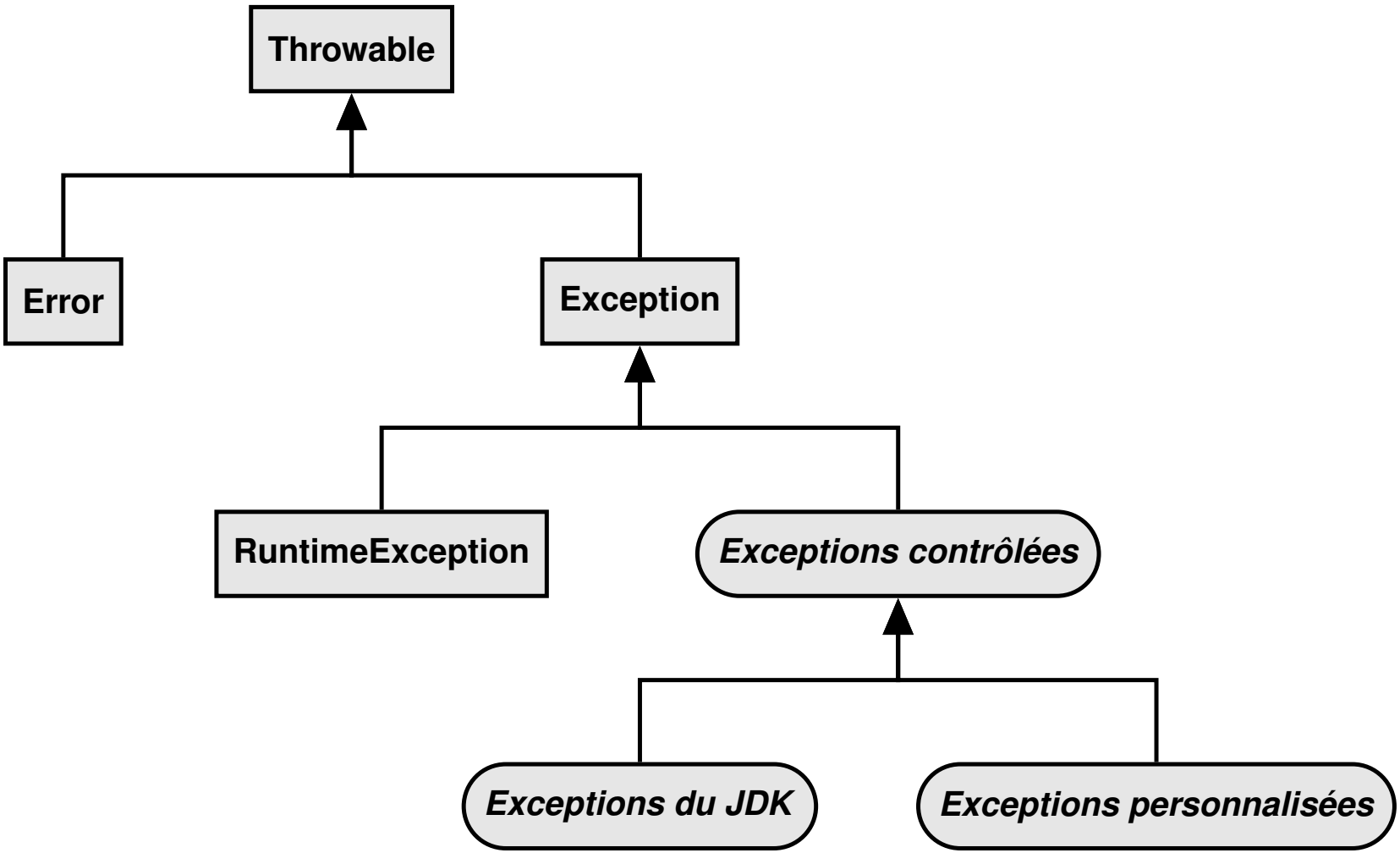
- Le langage Java propose un mécanisme particulier pour gérer les erreurs : les *exceptions*. Ce mécanisme repose sur deux principes :
  - Les différents types d'erreurs sont modélisées par des classes ;
  - Les instructions susceptibles de générer des erreurs sont séparées du traitement de ces erreurs : concept de **bloc d'essai** et de **bloc de traitement d'erreur**

Exception : définition

Le terme exception désigne tout événement arrivant durant l'exécution d'un programme interrompant son fonctionnement normal.

- En Java, les exceptions sont matérialisées par des instances de classes héritant de la classe **java.lang.Throwable**
- A chaque évènement correspond une sous-classe précise, ce qui peut permettre d'y associer un traitement approprié

Arbre d'héritage des exceptions



Arbre d'héritage des exceptions

Les blocs : **try**

Définition

La clause **try** s'applique à un bloc d'instructions correspondant au fonctionnement normal mais pouvant générer des erreurs

```
1 try {
2   ...
3   ...
4 }
```

Attention

Un bloc **try** ne compile pas si aucune de ses instructions n'est susceptible de lancer une exception.

Les blocs : **catch**

Définition

La clause **catch** s'applique à un bloc d'instructions définissant le traitement d'un type d'erreur. Ce traitement sera lancé sur une instance de la classe d'exception passée en paramètre

```
1  try{
2    ...
3  }
4  catch(TypeErreur1 e) {
5    ...
6  }
7  catch(TypeErreur2 e) {
8    ...
9  }
```

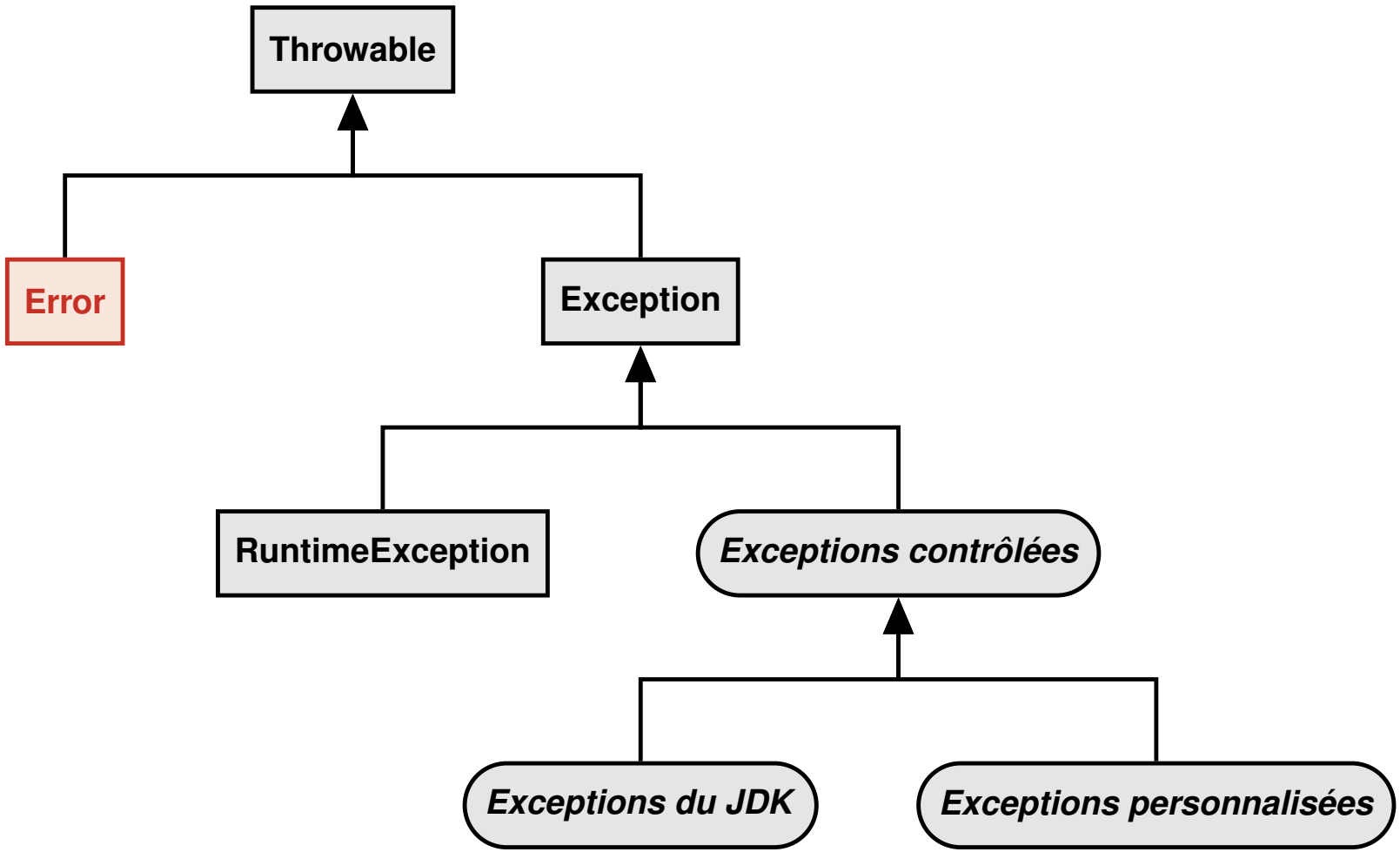
Règles sur les blocs : **try-catch**

- Si les blocs **try** et **catch** corresponde à deux types de traitement (resp. normal et erreur), ils n'en sont pas moins liés. Ainsi :
  - Tout bloc **try** doit être suivit par au moins un bloc **catch** ou par un bloc **finally** (étudié plus loin)
  - Tout bloc **catch** doit être précédé par un autre bloc **catch** ou par un bloc **try**
- Un ensemble composé d'un bloc *try* et d'au moins un bloc **catch** est communément appelé bloc **try-catch**
- Lorsqu'une instruction du bloc d'essai génère une erreur et y associe une exception, on dit qu'elle lance cette exception
- Lorsqu'un bloc de traitement d'erreur est déclenché par une exception, on dit qu'il lève cette exception

Les bloc try-catch : fonctionnement

- Le fonctionnement d'un bloc **try-catch** est le suivant :
  - si aucune des instructions du bloc d'essai ne lance d'exception, il est entièrement exécuté et les blocs de traitement d'erreur sont ignorés
  - si une des instructions du bloc d'essai lance une exception, alors toutes les instructions du bloc d'essai après elle sont ignorées et le premier bloc de traitement d'erreur correspondant au type d'exception lancée
  - Tous les autres blocs de traitement d'erreur sont ignorés

Arbre d'héritage des exceptions : Exception de type error



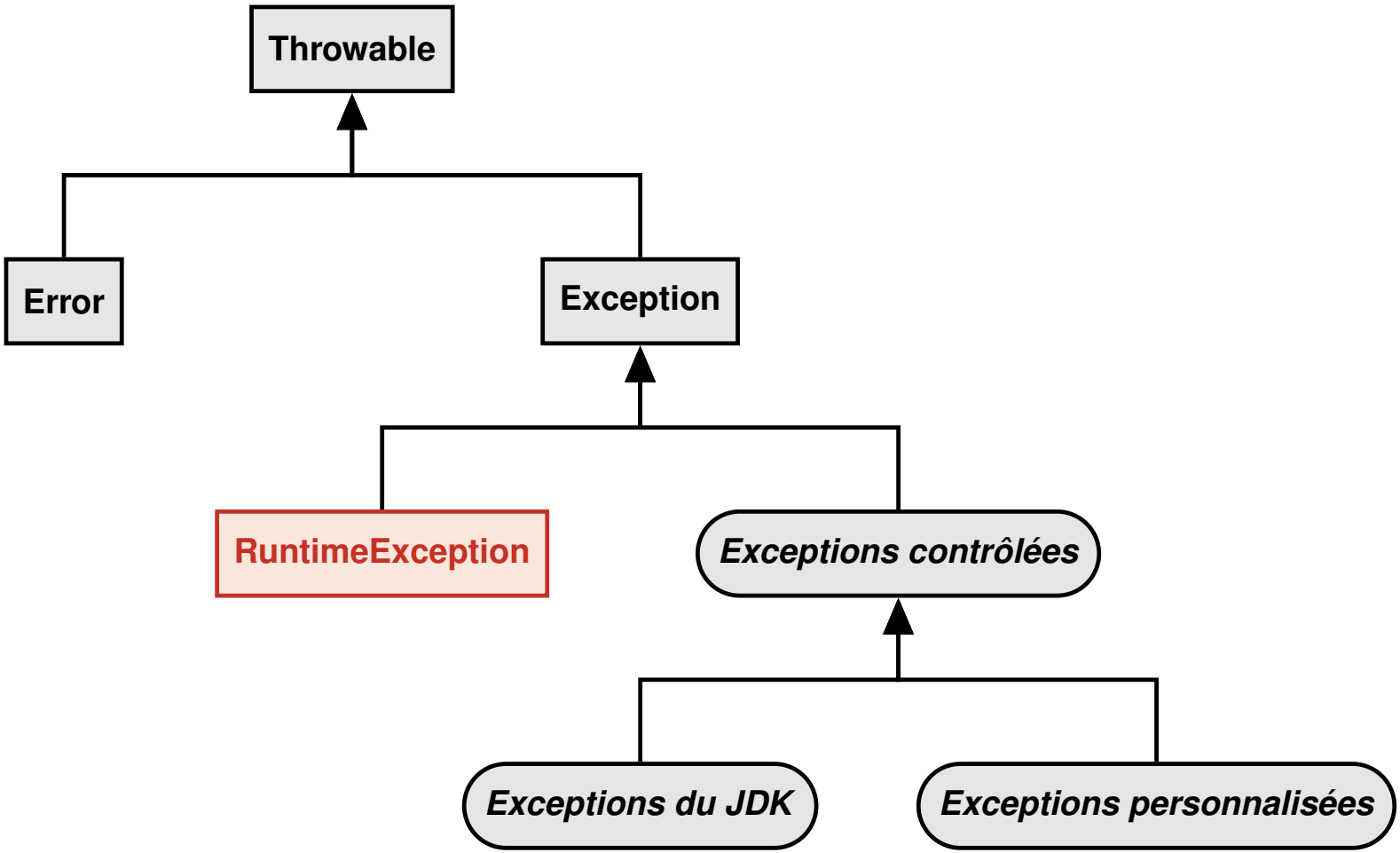
Arbre d'héritage des exceptions

Exception de type **error**

- Les exceptions de type Error sont réservées aux erreurs qui surviennent dans le fonctionnement de la JVM. Elles peuvent survenir dans toutes les portions du codes

- `Java` définit de nombreuses sous-classes de `Error` :
  - **OutOfMemoryError** : survient lorsque la machine virtuelle n'a plus de place pour faire une allocation et que le GC ne peut en libérer
  - **NoSuchMethodError** : survient lorsque la machine virtuelle ne peut trouver l'implémentation de la méthode appelée
  - **StackOverflowError** : survient lorsque la pile déborde après une série d'appel récursif trop profond
  - etc...

Arbre d'héritage des exceptions : Exception de type RuntimeException



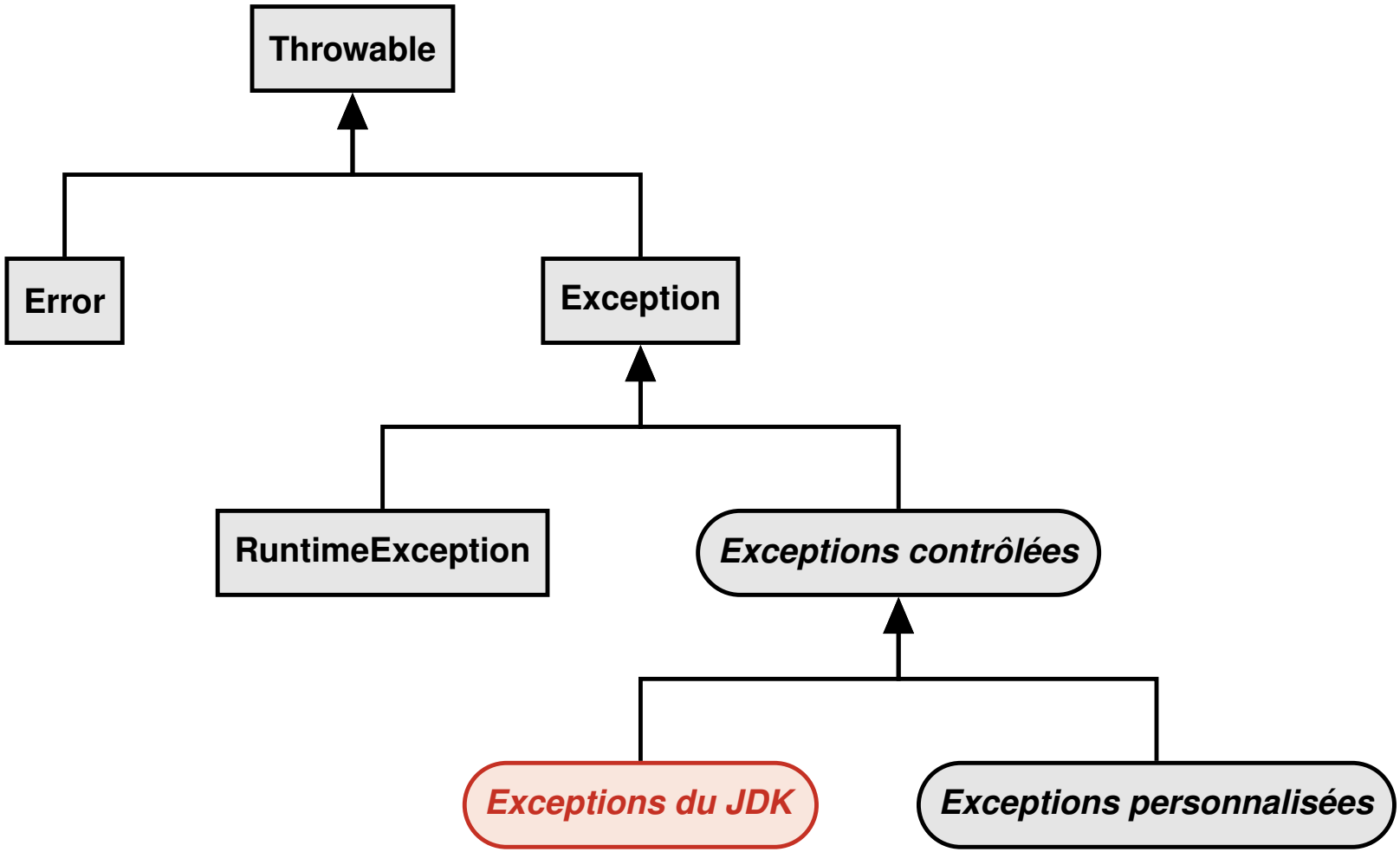
Exception de type **RuntimeException**

- Les exceptions de type **RuntimeException** correspondent à des erreurs qui peuvent survenir dans toutes les portions du codes
- Java définit de nombreuses sous-classes de **RuntimeException** :
  - **ArithmeticException** : division par zéro (entiers), etc ...
  - **IndexOutOfBoundsException** : dépassement d'indice dans un tableau
  - **NullPointerException** : référence null alors qu'on attendait une référence vers une instance
  - etc...

Exception de type RuntimeException

```
1  int a=0;
2  try {
3    int x = 1 / a;
4    System.out.println ("␣␣x␣=" + x );
5  }
6  catch ( ArithmeticException e ) { // division par 0 : 1 / 0
7           System.out.println("␣␣␣divisionpar0␣␣␣:1/" + a);
8  }
```

Arbre d'héritage des exceptions : Les exceptions contrôlées



Arbre d'héritage des exceptions

Les exceptions contrôlées

- On appelle exception contrôlée, toute exception qui hérite de la classe Exception et qui n'est pas une **RuntimeException**. Elle est dite contrôlée car le compilateur vérifie que toutes les méthodes l'utilisent correctement

- Le `JDK` définit de nombreuses exceptions :
  - **EOFException** : fin de fichier
  - **FileNotFoundException** : erreur dans l'ouverture d'un fichier
  - **ClassNotFoundException** : erreur dans le chargement d'une classe
  - etc...

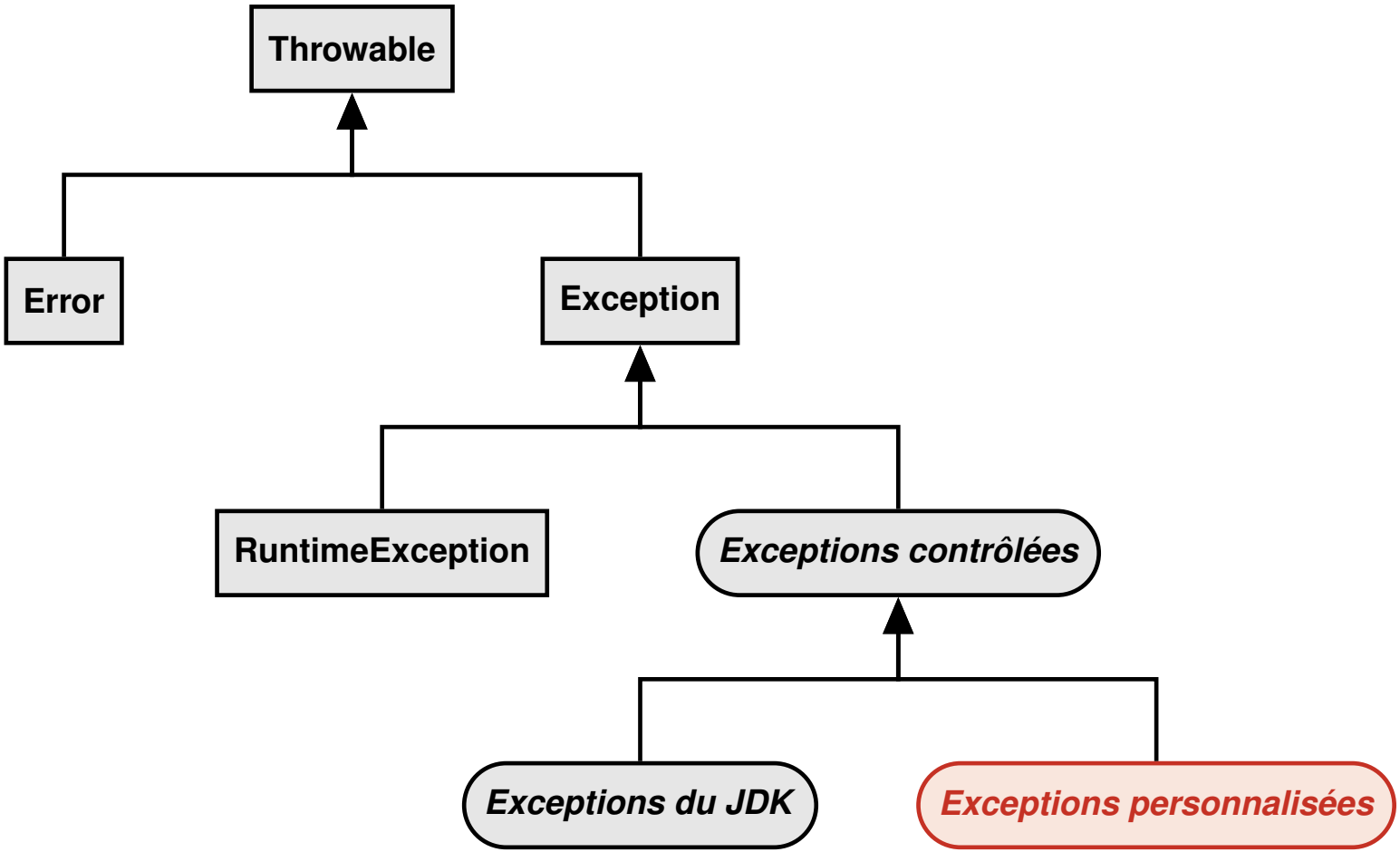
Les exceptions contrôlées

- Toute exception contrôlée, du JDK ou personnalisée, pouvant être émise dans une méthode doit être :
  - soit levée dans cette méthode. Elle est alors lancée dans un bloc try auquel est associé un catch lui correspondant
  - soit être indiquées dans le prototype de la méthode à l'aide du mot clé throws

Les exceptions contrôlées

```
1  public void f () throw FileNotFoundException {
2       FileInputStream monFichier ;
3       // Ouvrir un fichier peut éégnrer une exception
4       monFichier = new FileInputStream("./ essai . txt");
5  }
6
7  public void g() {
8       try {
9            // Un appel a f() peut éégnrer une exception
10           f();
11      }
12      catch (FileNotFoundException e) {
13          System.out.println(e) ;
14      }
15 }
```

Arbre d'héritage des exceptions : Les exceptions contrôlées personnalisées



Les exceptions contrôlées personnalisées

- On peut définir ses propres exceptions en définissant une sous-classe de la classe **Exception**

```
1  public class MonException extends Exception {
2      private int x;
3      public MonException(int x) {
4          this.x = x;
5      }
6
7      public String toString () {
8          return "␣␣valeurincorrecte␣:" + x;
9      }
10 }
```

Les exceptions contrôlées personnalisées

- Pour lancer une exception, on peut utiliser la clause throw.

```
1  class TestTableau {
2      public void main ( String args ) {
3          try {
4              int n = args[0];
5              if (n < 0) {
6                      throw new MonException(n) ;
7              }
8              System.out.println("␣␣␣Tailledutableau:" + n);
9          } catch ( MonException e ) {
10             System.err.println(e) ;
11         }
12     }
13 }
```

```
1  java TestTableau 3
2     Taille du tableau : 3
```

```
1  java TestTableau -1
2     valeur incorrecte : -1
```

La clause finally : définition

- La clause **finally** définit un bloc d'instruction qui sera exécuté même si une exception est lancée dans le bloc d'essai
- Elle permet de forcer la bonne terminaison d'un traitement en présence d'erreur, par exemple : la fermeture des fichiers ouverts

```
1  try {
2       ...
3  }
4  catch (...) {
5       ...
6  }
7  finally {
8       ...
9  }
```

La clause finally : quand ?

- Le code de la clause finally sera toujours exécuté :
  - si la clause **try** ne lève pas d'exception : exécution après le **try** (même s'il contient un **return**)
  - si la clause **try** lève une exception traité par un **catch** : exécution après le catch (même s'il contient un **return**)
  - si la clause **try** lève une exception non traité par un **catch** : exécution après le lancement de l'exception
- **Attention** : Un appel à la méthode **System.exit()** dans le bloc **try** ou dans un bloc **catch** arrête l'application sans passer par la clause **finally**

La clause finally : exemple

```
1  class TestTableau b {
2      public void main ( String args ) {
3          try {
4              int x = -1;
5              if (x < 0) {throw new MonException(x);}
6                  System.out.println(x) ;
7           }
8          catch ( MonException e ) {
9              System.err.println(e) ;
10         }
11         finally {
12             System.out. println("Tout est bien qui ...");
13         }
14     }
15 }
```

```
1  java TestTableau -1
2    valeur incorrecte : -1
3    Tout est bien qui ...
```

Plan du cours

Introduction

- Le package **java.io**, qui contient les classes de base pour gérer les flux de données, de type caractère ou binaire, et les fichiers
- Fait partie des premiers packages disponibles dès la création du langage
- **API** fournies sont extrêmement puissantes
- Elément important : la portabilité
- Par exemple : différences entre l'écriture des chemins et noms de fichiers entre les systèmes `Unix` et `Windows` pour entrevoir l'étendue du problème

Plan du cours

Notion de fichier

- Un fichier en `Java` est un objet instance de la classe **File**
- La classe **File** définit la notion de chemin dans un système de fichiers. Ce chemin se compose de deux éléments :
  - Un préfixe qui dépend du système d'exploitation sur lequel on est
  - Une séquence de chaînes de caractères
- Le premier nom de cette séquence peut être un nom de répertoire
- Le dernier nom de cette séquence peut être un nom de fichier ou un nom de répertoire
- Tous les noms intermédiaires de cette séquence sont des noms de répertoire

Caractère de séparation

- Un chemin vers un fichier utilise toujours un caractère de séparation
- Dépend du système d'exploitation ⇒ défini dans un champ public statique de la classe **File**, sous deux formes (`separator` : sous forme de chaîne de caractère & `separatorChar` : sous forme de caractère)
- Une chemin vers un fichier doit toujours utiliser l'un de ces champs, ce qui permet de le rendre portable d'un système à l'autre

```
1 // utilisation de StringBuffer pour eviter les concatenation
2  // de chaines de ècaractres
3 StringBuffer accessFileName = new StringBuffer() ;
4 accessFileName.append("tmp").append(File.separator).append("access.
      log") ;
5 System.out.println(accessFileName) ;
```

- Ce code affiche "tmp/access.log" sous Unix et "tmp\access.log" sous Windows

Chemin absolu ou relatif, répertoire courant

- Une instance de **File** peut représenter un chemin vers un répertoire ou un fichier (absolu ou relatif)
- Un chemin relatif est toujours mesuré par rapport au répertoire courant dans lequel s'exécute l'application
- On y a accès par une propriété système de la machine **Java** : **user.dir**

```
1 System.out.println(System.getProperty("user.dir")) ;
```

- Ce code affiche "**D :\projets\TD\workspace\exo-03**" sous `Windows`
- On peut noter aussi que la classe **File** est *immutable* . Une fois fixé le chemin d'une instance, on ne peut plus le modifier

Construction d'une instance de File

- La classe **File** admet quatre constructeurs

1. **File(String pathname)** : prend en paramètre une chaîne de caractères qui indique un chemin, relatif ou absolu, vers un fichier ou un répertoire
2. **File(String parent, String child)** : prend deux chaînes de caractères en paramètre. La première indique le chemin vers le fichier ou le répertoire. La seconde le nom de ce ficher ou répertoire
3. **File(File parent, String child)** : ce constructeur est analogue au précédent, sauf que le chemin est exprimé sous la forme d'une instance de File
4. **File(URI uri)** : ce dernier constructeur prend en paramètre une URI

Surcharge des méthodes de Object

- La classe **File** surcharge **equals()** et **hashCode()**, de même que **toString()**
- La méthode **toString()** affiche juste le fichier ou le répertoire représenté par cette instance de File
- La classe **File** implémente Comparable :
  - L'ordre choisi est simplement l'ordre lexicographique du nom du fichier ou du répertoire représenté
  - On prendra garde que sous Unix les différences entre majuscules et minuscules sont supportées, ce qui n'est pas le cas sous Windows

toString() : Exemple

```java
import java.io.File;

public class FileDemo {
   public static void main(String[] args) {
      File f = null;
      String str = "";
      boolean bool = false;
      try{
         // create new File object
         f = new File("test.txt");
         // returns true if file exists
         bool = f.exists();
         // if file exists
         if(bool)
         {
            // pathname string of this abstract pathname
            str = f.toString();
            // print
            System.out.println("pathname string: "+str);
         }
      }catch(Exception e){
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

```
pathname string: test.txt
```

equals() : Exemple

```java
import java.io.File;

public class FileDemo {
   public static void main(String[] args) {
      File f = null;
      File f1 = null;
      boolean bool = false;
      try{
         // create new files
         f = new File("test.txt");
         f1 = new File("test1.txt");
         // returns boolean
         bool = f.equals(f);
         // prints
         System.out.println("Equal: "+bool);
         // returns boolean
         bool = f.equals(f1);
         // prints
         System.out.print("Equal: "+bool);
      }catch(Exception e){
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

```
Equal: true
Equal: false
```

hashCode() : Exemple

```java
import java.io.File;

public class FileDemo {
   public static void main(String[] args) {
      File f = null;
      int v;
      boolean bool = false;
       try{
         // create new file
         f = new File("C:\\test.txt");
          // returns hash code for this abstract pathname
         v = f.hashCode();
          // true if the file path exists
         bool = f.exists();
          // if file exists
         if(bool)
         {
            // prints
            System.out.print("The hash code for this abstract pathname: "+v);
         }
      }catch(Exception e){
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

```
The hash code for this abstract pathname: -246273912
```

Interrogation du type de ressources et de ses droits

- **isFile()**, **isDirectory()** :
  - permettent de tester si cette instance de File représente un fichier ou un répertoire
- **exists()**, **canRead()**, **canWrite()**, **canExecute()** :
  - permettent de tester différents éléments sur le fichier ou le répertoire représenté par cette instance de File
- **setReadable(boolean b)**, **setWritable(boolean b)**, **setExecutable(boolean b)** :
  - permettent de modifier la propriété correspondante du fichier ou du répertoire. Ces méthodes peuvent prendre un booléen supplémentaire, qui indique si la propriété correspondante doit être modifiée pour tous les utilisateurs, ou uniquement pour le propriétaire du fichier ou du répertoire
- **lastModified()** et **setLastModified()** :
  - permettent de manipuler la date de dernière modification de ce fichier ou de ce répertoire

isFile() : Exemple

```
1  import java.io.File;
2  public class FileDemo {
3     public static void main(String[] args) {
4        File f = null;
5        String path;
6        boolean bool = false;
7         try{
8           // create new file
9           f = new File("c:");
10           // true if the file path is a file, else false
11          bool = f.isFile();
12           // get the path
13          path = f.getPath();
14           // prints
15          System.out.println(path+" is file? "+ bool);
16       }catch(Exception e){
17          // if any error occurs
18          e.printStackTrace();
19       }
20    }
21 }
```

```
1  c: is file? false
```

Interrogation du type de ressources et de ses droits

- **length()** : retourne la taille du fichier sous forme d'un long quand cette instance de File représente un fichier. Le résultat n'est pas prévisible si cette instance représente un répertoire
- **isHidden()** : permet de savoir si ce fichier ou répertoire est caché ou non. Notons que la notion de fichier caché est différente sous Unix et Windows. Sous Unix, un fichier est caché s'il commence par un point (.). Sous Windows, chaque fichier possède un attribut particulier qui indique s'il est caché ou non

Remarque

Certaines de ces opérations peuvent nécessiter des droits particuliers sur les fichiers ou répertoires. Dépend du système de fichier utilisé

length() : Exemple

```java
import java.io.File;
public class FileDemo {
   public static void main(String[] args) {
      File f = null;
      String path;
      long len;
      boolean bool = false;
      try{
         // create new file
         f = new File("c:/test.txt");
         // true if the file path is a file, else false
         bool = f.exists();
         // if path exists
         if(bool)
         {
            // returns the length in bytes
            len = f.length();
            // path
            path = f.getPath();
            // print
            System.out.print(path+" file length: "+len);
         }
      }catch(Exception e){
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

```
c:\test.txt file length: 5
```

Interrogation du nom et du chemin

- **getName()** : retourne le nom de ce fichier ou répertoire, sans son chemin d'accès s'il est précisé. Ce nom correspond au dernier élément de la séquence de nom de cette instance de File

- **getParent()** : retourne le nom du parent de ce fichier ou répertoire. Le parent est défini par la séquence complète des noms, stockée dans cette instance de File, de laquelle on a retiré le dernier élément, qui correspond par convention au nom de ce fichier ou répertoire

- **getPath()** : retourne le chemin complet de ce fichier. Le retour de cette méthode est le même que celui de la méthode toString()

getName() : Exemple

```
1  import java.io.File;
2   public class FileDemo {
3     public static void main(String[] args) {
4         File f = null;
5        String v;
6        boolean bool = false;
7         try{
8           // create new file
9           f = new File("C:\\test.txt");
10           // get file name or directory name
11          v = f.getParent();
12           // true if the file path exists
13          bool = f.exists();
14           // if file exists
15          if(bool)
16          {
17             // prints
18             System.out.print("parent name: "+v);
19          }
20       }catch(Exception e){
21          // if any error occurs
22          e.printStackTrace();
23       }
24    }
25 }
```

```
1  parent name: C:\
```

Interrogation du nom et du chemin

- **getAbsolutePath()** : le retour de cette méthode diffère si l'instance de File représente un chemin absolu ou pas. Si ce chemin est absolu, le retour de cette méthode est le même que getPath(). S'il ne l'est pas, un chemin absolu est calculé, relativement au répertoire courant de l'application

- **getCanonicalPath()** : Retourné est le chemin direct vers le fichier ou le répertoire représenté. Supprime de ce chemin les mouvements via le répertoire ".." notamment, et identifie les raccourcis ou liens symboliques, qu'elle remplace par les vrais chemins dans le système de fichier

getAbsolutePath() : Exemple

```
1  import java.io.File;
2  public class PathTesting {
3    public static void main(String [] args) {
4      File f = new File("test/.././file.txt");
5      System.out.println(f.getPath());
6      System.out.println(f.getAbsolutePath());
7      try {
8        System.out.println(f.getCanonicalPath());
9      }
10     catch(Exception e) {}
11   }
12 }
```

```
1  test\..\.\file.txt
2  C:\projects\sandbox\trunk\test\..\.\file.txt
3  C:\projects\sandbox\trunk\file.txt
```

Création de fichier

- **createNewFile()** :
  - demande la création de ce fichier au système de fichier. Cette création ne peut se faire que si le fichier à créer n'existe pas déjà. Si la création n'a pu avoir lieu alors cette méthode retourne **false**. Si une erreur a été rencontrée, alors la méthode jette une **IOException**
- **delete()** :
  - demande l'effacement de ce fichier ou répertoire. Retourne **false** si cet effacement n'a pu avoir lieu
- **mkdir()** et **mkdirs()** :
  - ces deux méthodes créent le répertoire représenté par cette instance de File. **mkdirs()** peut créer une série de répertoires imbriqués, ce qui n'est pas le cas de **mkdir()**. Retourne **false** si la création n'a pas pu se faire
- **deleteOnExit()** :
  - demande à la machine **Java** d'effacer automatiquement ce fichier ou ce répertoire quand l'application se termine. Si plusieurs fichiers ou répertoires sont enregistrés de la sorte, alors ils sont effacés en commençant par le dernier qui a été enregistré. L'effacement ne peut se faire que si la `JVM` s'éteint normalement, sans plantage

createNewFile() : Exemple

```java
import java.io.File;

public class FileDemo {
   public static void main(String[] args) {
      File f = null;
      boolean bool = false;
      try{
         // create new file
         f = new File("test.txt");
         // tries to create new file in the system
         bool = f.createNewFile();
         // prints
         System.out.println("File created: "+bool);
         // deletes file from the system
         f.delete();
         // delete() is invoked
         System.out.println("delete() method is invoked");
         // tries to create new file in the system
         bool = f.createNewFile();
         // print
         System.out.println("File created: "+bool);
      }catch(Exception e){
         e.printStackTrace();
      }
   }
}
```

```
Absolute Path D:\EclipseAndroid\IO\BufferedInputStream\test.txt
```

Création de fichiers temporaires

- Deux méthodes statiques de la **classe File** pour créer des fichiers temporaires :
  - **createTempFile(String prefix, String suffix, File directory)**
  - **createTempFile(String prefix, String suffix)**
- La machine Java garantit deux choses sur ces fichiers :
  - le fichier retourné est toujours un nouveau fichier, qui n'existait pas auparavant
  - deux appels successifs à cette méthode, avec les mêmes arguments, ne retournent pas le même fichier
- Le nom du fichier créé commence par prefix, puis est complété par un code unique, qui garantit l'unicité du fichier créé. L'extension de ce fichier est prefix, ou .tmp si prefix est nul
- Si le paramètre suffix est nul, alors le fichier aura pour extension .tmp

createTempFile() : Exemple

```
1  import java.io.File;
2
3  public class FileDemo {
4     public static void main(String[] args) {
5        File f = null;
6        try{
7           // creates temporary file
8           f = File.createTempFile("tmp", ".txt", new File("C:/"));
9           // prints absolute path
10          System.out.println("File path: "+f.getAbsolutePath());
11          // deletes file when the virtual machine terminate
12          f.deleteOnExit();
13          // creates temporary file
14          f = File.createTempFile("tmp", null, new File("D:/"));
15          // prints absolute path
16          System.out.print("File path: "+f.getAbsolutePath());
17          // deletes file when the virtual machine terminate
18          f.deleteOnExit();
19       }catch(Exception e){
20          // if any error occurs
21          e.printStackTrace();
22       }
23    }
24 }
```

```
1  File path: C:\tmp3602253894598046604.txt
2  File path: D:\tmp587577452036748166.tmp
```

Capacité du système de fichier

- Ces méthodes permettent d'interroger le système de fichier pour connaître la capacité de stockage du système de fichier sur lequel on est :
  - **getFreeSpace()** : retourne un long qui porte le nombre d'octets utilisables sur la partition sur laquelle se trouve le fichier interrogé. On prendra garde qu'aucun test n'est fait pour savoir si cette l'application a le droit d'écrire sur cette partition. Il est par conséquent possible qu'aucun de ces octets ne soit disponible
  - **getUsableSpace()** : retourne un long qui porte le nombre d'octets utilisables sur la partition sur laquelle se trouve le fichier interrogé. Cette estimation est plus précise que **getFreeSpace()**. Effectivement, la machine Java vérifie un certain nombre de choses pour déterminer cet espace, comme les droits en écriture
  - **getTotalSpace()** : retourne un long qui porte le nombre d'octets de cette partition

Contenu d'un répertoire

- **list()** : retournent tous les fichiers et répertoires de ce répertoire
- **listeFiles()** : retournent tous les fichiers (sans les répertoires) de ce répertoire

```java
import java.io.File;

public class FileDemo {
   public static void main(String[] args) {
      File f = null;
      File[] paths;

      try{
         // create new file
         f = new File("./test");
         // returns pathnames for files and directory
         paths = f.listFiles();
         // for each pathname in pathname array
         for(File path:paths)
         {
            // prints file and directory paths
            System.out.println(path);
         }
      } catch(Exception e){
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

```
./test/child_test
./test/child_test.java
./test/child_test.txt
./test/child_test.xls
```

Contenu d'un répertoire

- **listFiles(FileFilter)** et **listFiles(FilenameFilter)** : retournent tous les fichiers (sans les répertoires) de ce répertoire, qui satisfont le filtre passé en paramètre

```
1  import java.io.File;
2  import java.io.FilenameFilter;
3  public class FileDemo {
4     public static void main(String[] args) {
5        File f = null;
6        File[] paths;
7        try{
8           // create new file
9           f = new File("./test");
10          // returns pathnames for files and
                directory
11          paths = f.listFiles(new
                FilenameFilter() {
12             @Override
13             public boolean accept(File dir,
                   String name) {
14                if(name.lastIndexOf('.')>0)
                      {
15                   // get last index for '.'
                         char
16                   int lastIndex = name.
                         lastIndexOf('.');
17                   // get extension
18                   String str = name.
                         substring(lastIndex
                         );
19                   // match path name
                         extension
20                   if(str.equals(".txt"))
21                      return true;
22                }
23                return false;
24             }
25          });
26          // for each pathname in pathname
                array
27          for(File path:paths) {
28             // prints file and directory
                   paths
29             System.out.println(path);
30          }
31       } catch(Exception e){
32          // if any error occurs
33          e.printStackTrace();
34       }
35    }
36 }
```

Utilisation de **File.listRoots()**

- **listFiles(FileFilter)** et **listFiles(FilenameFilter)** : retournent tous les fichiers (sans les répertoires) de ce répertoire, qui satisfont le filtre passé en paramètre

```
1  // liste des éélments racine du èsystme de fichier courant
2  File [] roots = File.listRoots() ;
3
4   // affichage du tableau érsultat
5  System.out.println(Arrays.toString(roots)) ;
```

- Sur une machine Windows, l'exécution de ce code donne un résultat du type : `[C:\, D:\, E:\, F:\, G:\]`

Plan du cours

Introduction, notion de flux

- La première chose à définir lorsque l'on parle de flux de sortie est la notion de sortie

- Les sorties les plus courantes sont :
  1. un fichier, qu'il soit à accès séquentiel ou aléatoire ;
  2. une chaîne de caractères, instance de String ;
  3. la console système ;
  4. un tableau de caractères, ou d'octets ;
  5. une URL (ou URI), dans le cas des flux HTTP ;
  6. une socket, pour la communication ;
  7. un tuyau de communication entre threads

- Pour chacune de ces sorties, il existe une classe **Java** qui permet d'écrire dessus
- L'action d'écrire se fait en utilisant un objet qui modélise le flux des données vers chacun de ces média de sortie

Écriture de caractères, classe **Writer**

- La classe de base utilisée pour écrire des caractères est la classe **Writer**

- Cette classe est abstraite, et expose les méthodes suivantes :
  1. **write(char[] buffer), write(char[] buffer, int offset, int length), write(String s), write(String s, int offset, int length)** et **write(int c)**
     - Toutes ces méthodes permettent d'écrire des caractères en provenance de différentes sources (tableau ou chaîne de caractères). La dernière écrit un unique caractère stocké dans les 16 bits de poids faible de l'entier passé en paramètre
  2. **append(char c), append(CharSequence seq)** et **append(CharSequence seq, int start, int end)**
     - Ces méthodes fonctionnent de façon analogue aux précédentes. La différence est qu'elles retournent cette instance de **Writer**, ce qui permet de chaîner ces appels
  3. **flush()** : permet de vider les buffers d'écriture vers le médium de sortie.
  4. **close()** : ferme ce flux.

Écriture de caractères, classe Writer : Remarques

Cette classe **Writer** est abstraite, on ne peut donc pas l'instancier directement. Elle est étendue par une série de classes, qui correspondent à chacun des média de sortie dont nous avons donné la liste en introduction : **FileWriter**, **StringWriter** et **CharArrayWriter** sont les principales

Chacune des classes concrètes qui étendent Writer exposent leurs propres constructeurs. **FileWriter** se construit sur un fichier (instance de **File**). Pour **StringWriter** et **CharArrayWriter** on indique juste la taille de la chaîne ou du tableau dans lequel l'écriture va se faire

Écriture de caractères, classe Writer

```
1  import java.io.*;
2
3  public class WriterDemo {
4  public static void main(String[] args) {
5
6      // édfinition d'un fichier
7     File fichier = new File("tmp/bonjour.text") ;
8
9      try {
10
11        // ouverture d'un flux de sortie sur un fichier
12        // a pour effet de écrer le fichier
13       Writer writer = new FileWriter(fichier) ;
14
15        // écriture dans le fichier
16       writer.write("Bonjour le monde !") ;
17
18        // la émthode close de FileWriter appelle elle-êmme flush()
19       writer.close() ;
20
21    }  catch (IOException e) {
22
23        // affichage du message d'erreur et de la pile d'appel
24       System.out.println("Erreur " + e.getMessage()) ;
25       e.printStackTrace() ;
26    }
27 }
28 }
```

Écriture de caractères, classe Writer (V2 : version corrigée)

```
1  import java.io.*;
2
3  public class WriterDemo {
4  public static void main(String[] args) {
5
6      // édfinition d'un fichier
7     File fichier = new File("tmp/bonjour.
           text") ;
8
9      // la édfinition du writer doit se faire
           ici
10     // pour des raisons de évisibilit
11    Writer writer = null ;
12
13     try {
14
15        // ouverture d'un flux de sortie sur
              un fichier
16        // a pour effet de écrer le fichier
17       writer = new FileWriter(fichier) ;
18
19        // écriture dans le fichier
20       writer.write("Bonjour le monde !") ;
21
22    }  catch (IOException e) {
23
24        // affichage du message d'erreur et
              de la pile d'appel
25        System.out.println("Erreur " + e.
              getMessage()) ;
26        e.printStackTrace() ;
27
28    }  finally {
29
30        // il se peut que l'ouverture du flux
              ait ééchou,
31        // et que ce writer n'ait pas éét
              éinitialis
32        if (writer != null) {
33
34           try {
35
36              // la émthode close de
                    FileWriter appelle elle-
                    êmme flush()
37              writer.close() ;
38
39           }  catch (IOException e) {
40
41              System.out.println("Erreur " + e
                    .getMessage()) ;
42              e.printStackTrace() ;
43           }
44        }
45    }
46 }
47 }
```

Écriture d'octets, **OuputStream**

- L'écriture d'octets suit exactement la même logique que l'écriture de caractères
- Ces écritures sont gérées par la classe abstraite **OutputStream**
- Cette classe est abstraite, et étendue par les classes **FileOutputStream** pour l'écriture dans des fichiers, et **ByteArrayOutputStream** pour l'écriture dans des tableaux d'octets
- La classe OutputStream expose les méthodes suivantes :
  1. **write(byte [] b)**, **write(byte [] b, int offset, int length)** et **write(int b)**
     - Ces méthodes permettent d'écrire des octets directement sur le flux. Dans le cas de l'écriture d'un entier, seuls les 8 bits de poids faibles sont pris en compte
  2. **close()** et **flush()**
     - Ont la même sémantique que celles de la classe **Writer**

Écriture de caractères, classe **OuputStream** : **FileOutputStream**

```
1  import java.io.*;
2
3  public class WriterDemo {
4  public static void main(String[] args) {
5
6      // édfinition d'un fichier
7     File fichier = new File("tmp/array.bin")
              ;
8     OutputStream os = null ;
9
10     // écration d'un tableau d'octets, qui
             sera
11     // écrit dans le fichier
12     byte [] byteArray = { 0, 1, 2 } ;
13
14     try {
15
16        // ouverture d'un flux de sortie sur
               un fichier
17       os = new FileOutputStream(fichier) ;
18        // écriture proprement dite
19       os.write(byteArray) ;
20
21    } catch (IOException e) {
22
23       System.out.println("Erreur " + e.
               getMessage()) ;
24       e.printStackTrace() ;
25    } finally {
26
27        // fermeture du fichier dans le bloc
               finally
28        if (os != null) {
29
30           try {
31
32              // la émthode close de
                     FileOutputStream appelle
                     elle-êmme flush()
33              os.close() ;
34
35           } catch (IOException e) {
36
37              System.out.println("Erreur " + e
                     .getMessage()) ;
38              e.printStackTrace() ;
39           }
40    }
41 }
42 }
```

Écriture d'octets, **OuputStream** : classe **ByteArrayOutputStream**

- La classe **ByteArrayOutputStream** permet de diriger le flux de sortie vers un tableau d'octets, qui sert alors de buffer
- Cette classe expose quelques méthodes supplémentaires pour la gestion de ce buffer :
  1. **reset()** :
     - permet de remettre à zéro le pointeur d'écriture de ce buffer, ce qui a pour effet d'annuler tout ce qui a été écrit
  2. **size()** :
     - retourne la taille du buffer. Notons que la taille de ce buffer peut être fixée à la construction de cette instance de **ByteArrayOutputStream**
  3. **toByteArray()**
     - retourne le tableau d'octets sur lequel ce buffer est construit
  4. **toString(String charsetName)**
     - permet de convertir le contenu de ce tableau d'octets en utilisant le jeu de caractères précisé en argument. La liste des jeux de caractères supportés est donnée dans la **javadoc** de la classe **Charset**

Écriture de caractères, classe **OuputStream** : **ByteArrayOutputStream**

```
1
2  import java.io.ByteArrayOutputStream;
3  import java.io.IOException;
4
5  public class ByteArrayOutputStreamDemo {
6     public static void main(String[] args)
            throws IOException {
7
8        String str = "";
9
10       byte[] bs = {65, 66, 67, 68, 69};
11       ByteArrayOutputStream baos = null;
12
13       try{
14
15          // create new ByteArrayOutputStream
16          baos = new ByteArrayOutputStream();
17
18          // write byte array to the output
                stream
19          baos.write(bs);
```

```
20
21          // converts buffers content using
                Cp1047 character set
22          str = baos.toString("Cp1047");
23          System.out.println(str);
24
25          // converts buffers contents using
                UTF-8 character set
26          str = baos.toString("UTF-8");
27          System.out.println(str);
28
29       }catch(Exception e){
30
31          // if I/O error occurs
32          e.printStackTrace();
33       }finally{
34          if(baos!=null)
35             baos.close();
36       }
37    }
38 }
```

```
1  âäàá
2
3  ABCDE
```

Écriture de types primitifs, classe **DataOutputStream**

- L'écriture de types primitifs est fournie par la classe **DataOutputStream**

```
1
2  import java.io.ByteArrayOutputStream;
3  import java.io.IOException;
4
5  public class DataOutputStreamDemo {
6  public static void main(String[] args) {
7
8     File fichier = new File("tmp/integers.
           bin") ;
9     OutputStream os = null ;
10
11     try {
12        // ouverture d'un flux de sortie sur
              un fichier
13       os = new FileOutputStream(fichier) ;
14        // ouverture d'un flux de type
              DataOutputStream
15        // sur ce êmme fichier
16       DataOutputStream dos = new
              DataOutputStream(os) ;
17
18        for (int i : Arrays.asList(1, 2, 3,
              4, 5)) {
19          dos.writeInt(i) ;
20       }
21
22       } catch (IOException e) {
23          // gestion de l'erreur
24       } finally {
25          // fermeture du flux
26       }
27    }
28 }
29 }
```

- Les méthodes d'écriture suivent toutes le même modèle de nommage : **writeInt(int)**, **writeLong(long)**, etc...

Écriture d'objets : la classe **ObjectOutputStream**

- La classe **ObjectOutputStream** supporte l'écriture directe d'objets sur des flux
- Ce mécanisme particulier s'appelle serialization , et a un fonctionnement particulier
- Il permet de garantir, entre autres, qu'une instance d'une classe écrite dans un fichier est bien recréée dans la bonne classe, identique à la première
- Ce mécanisme peut aussi être surchargé de différentes façons

Plan du cours

Introduction

- À chaque type de flux de sortie est associé un flux d'entrée
- permet de lire les données qui ont été écrites, dans les mêmes conditions
- Aux classes **Writer** et **OutputStream**, que nous venons de voir dans la section précédente, sont donc associées les classes **Reader** et **InputStream** respectivement

Lecture de caractères, classe **Reader**

- La classe Reader est une classe abstraite, étendue par **CharArrayReader**, **StringReader** et **FileReader**
- Ces trois classes permettent de lire des flux de caractères en provenance de tableaux de char, de chaînes de caractères, ou de fichiers
- La classe **Reader** expose plusieurs versions d'une méthode **read()**
- Permet de lire des caractères, un par un, ou en les stockant dans un tableau
- Toutes ces versions de **read()** retournent un **int**
- Ce nombre correspond au nombre de caractères effectivement lus sur le flux
- appel à **read()** peut bloquer , c'est-à-dire, ne retourner de valeur qu'au bout d'un certain temps, durant lequel le flux est en attente de l'arrivée de nouveaux caractères

Utilisation de **FileReader**

```
1  public static void main(String[] args) {
2      // édclaration de notre FileReader àl'
           éextieur des
3     FileReader fr = null ;
4      try {
5        // ouverture du flux de lecture
6        // peut jeter une
             FileNotFoundException
7       fr = new FileReader("tmp/bonjour.text
           ") ;
8        // édfinition du buffer : un tableau
             de char
9        int bufferSize = 1024 ;
10       char [] buffer = new char[bufferSize
           ] ;
11       // édfinitions de variables pour
             suivre notre
12       // lecture
13       int n = 0 ;
14       int total = 0 ;
15       int loops = 0 ;
16       do {
17          // remplissage du buffer
18          // n contient le nombre de
                ècaractres effectivement lus
19         n = fr.read(buffer) ;
20         total += n ;
21         loops++ ;
22        // si le nombre lu est -1,
23        // c'est que l'on a atteint la fin du
              flux
24       } while (n != -1) ;
25        // quelques informations sur la
              lecture
26       System.out.println(
27             "Nombre de ècaractres lus au
                   total = " + total +
28             " en " + loops + " boucles.") ;
29    } catch (FileNotFoundException e) {
30        // gestion de l'erreur
31    } catch (IOException e) {
32        // gestion de l'erreur
33    } finally {
34        // pattern de fermeture d'un flux
35        if (fr != null) {
36           try {
37             fr.close() ;
38           } catch (IOException e) {
39              // gestion de l'erreur
40           }
41        }
42    }
43 }
44 }
```

Lecture d'octets : classe **InputStream**

- Le pendant de la classe **OutputStream** est la classe **InputStream**. Cette classe abstraite est la base des classes de lecture des flux d'octets
- Cette classe est étendue par :
  1. **FileInputStream** pour la lecture dans des fichiers ;
  2. **ByteArrayInputStream** pour la lecture dans des tableaux d'octets ;
  3. **DataInputStream** pour la lecture des types primitifs Java ;
  4. Ce nombre correspond au nombre de caractères effectivement lus sur le flux
  5. **ObjectInputStream** pour la lecture des objets sérialisés.
- Le fonctionnement de la lecture d'octets sur un flux suit le même pattern que la lecture de caractères
- Les octets sont lus dans un tableau, qui sert de buffer
- La méthode **read()** retourne le nombre d'octets effectifs qui ont été lus. Lorsque ce nombre est -1, la fin du flux a été atteinte.

Lecture d'octets : classe InputStream

- La classe InputStream expose les méthodes suivantes :
  1. **read(), read(byte[] buf) et read(byte[] buf, int offset, int length)** : ces méthodes permettent de lire les octets dans un tableau. Elles retournent toutes le nombre d'octets effectivement lus
  2. **skip(long n)** : saute le nombre d'octets passé en paramètre dans le flux de lecture
  3. **reset()** : réinitialise la lecture de ce flux
  4. **available()** : retourne le nombre d'octets disponibles dans un int. Ce nombre d'octets peut être lu, ou sauté. La lecture qui suit ne bloquera pas, tant que ce nombre ne sera pas dépassé

Utilisation de FileReader

```
1  public static void main(String[] args) {
2      // declaration de notre InputStream à l'
           exetieur des
3      // blocs try {} catch {}
4     InputStream is = null ;
5      try {
6        // ouverture du flux d'entree sur un
             fichier
7        // peut jeter une
             FileNotFoundException
8       is = new FileInputStream(fichier) ;
9        // édfinition du buffer : un tableau
             d'octets
10       int bufferSize = 1024 ;
11       byte [] buffer = new byte[bufferSize
             ] ;
12       // definitions de variables pour
             suivre notre
13       // lecture
14       int n = 0 ;
15       int total = 0 ;
16       int loops = 0 ;
17       do {
18          // remplissage du buffer
19          // n contient le nombre d'octets
                effectivement lus
20          n = is.read(buffer) ;
21          total += n ;
22          loops++ ;
23
24        // si le nombre lu est -1,
25        // c'est que l'on a atteint la fin du
              flux
26       } while (n != -1) ;
27
28        // quelques informations sur la
              lecture
29       System.out.println(
30             "Nombre d'octets lus au total =
                    " + total +
31             " en " + loops + " boucles.") ;
32    } catch (FileNotFoundException e) {
33        // gestion de l'erreur
34    } catch (IOException e) {
35        // gestion de l'erreur
36    } finally {
37        // pattern de fermeture d'un flux
38        if (fr != null) {
39           try {
40             fr.close() ;
41           } catch (IOException e) {
42              // gestion de l'erreur
43           }
44       }
45    }
46 }
```

Lecture de types primitifs : **DataInputStream**

- La classe **DataInputStream** expose un jeu de méthodes qui permet de lire les types primitifs Java directement dans des flux binaires : **readInt(), readLong()**, etc...

Lecture d'objets : ObjectInputStream

- De même que pour l'écriture directe d'objets, la lecture d'objets (appelée aussi désérialisation) sera vue dans une section à part

Plan du cours

Fonctionnement de la sérialization

- Sérializer un objet consiste à le convertir en un tableau d'octets, que l'on peut ensuite écrire dans un fichier, envoyer sur un réseau au travers d'une socket etc...
- Il suffit de passer tout objet qui implémente l'interface **Serializable** à une instance de **ObjectOutputStream** pour sérialiser un objet
- Si cet objet ne comporte pas de champ trop exotique, comme des connexions à des bases de données, des fichiers ou des **threads**, cette sérialization se déroulera sans problème
- Des problèmes peuvent se poser pour les objets qui possèdent des champs eux-mêmes non sérializables
- Dans ce cas, ces champs doivent être marqués avec le mot-clé transient. Cela a pour effet de les retirer du flux sérializé. Après désérialization, ces champs seront à **null**

Fonctionnement de la sérialization

```
1  // écration d'une classe Serializable
2   public class Marin implements Serializable {
3
4      // la classe String est Serializable, donc ces champs sont élgaux
5      private String nom, prenom ;
6
7
8      // en revanche la classe Connection ne l'est pas,
9      // il faut donc retirer ce champ de la serialization
10     private transient Connection con ;
11 }
```

Notion de serialVersionUID

- Il est nécessaire de vérifier que la classe que l'on possède est bien la même que celle qui a servi à la création de ces octets
- Vérifier son nom complet n'est pas suffisant, elle doit définir les mêmes champs, de même nom et de même type
- Java introduit un code de hachage associé aux classes qui implémentent **Serializable**, stocké dans un champ standard
- Ce champ standard s'appelle **serialVersionUID**, doit être de type **long** et doit être **private static final**
- Ce champ est systématiquement enregistré dans tout paquet d'octets qui représente un objet sérialized

Sérialisation d'un objet

- La sérialization d'un objet consiste à passer un objet à la méthode **writeObject(Object)** de la classe **ObjectOutputStream**.

```
1  import java.io.Serializable;
2
3  public class Marin implements Serializable {
4
5      private static final long serialVersionUID = 77085381207865022l9L;
6
7      private String nom, prenom ;
8
9      public Marin(String nom, String prenom) {
10        this.nom = nom ;
11        this.prenom = prenom ;
12     }
13
14     public String toString() {
15       StringBuffer sb = new StringBuffer() ;
16        return sb.append(nom).append(" ").append(prenom).toString() ;
17     }
18 }
```

Serialization d'un objet

```
1  import java.io.*;
2  public class Marin implements Serializable {
3      private static final long serialVersionUID = 7708538120786502219L;
4      private String nom, prenom ;
5      public Marin(String nom, String prenom) {
6         this.nom = nom ;
7         this.prenom = prenom ;
8     }
9      public String toString() {
10       StringBuffer sb = new StringBuffer() ;
11        return sb.append(nom).append(" ").append(prenom).toString() ;
12    }
13     public static void main(String[] args) throws FileNotFoundException, IOException {
14        Marin m = new Marin("Nadjib","Achir");
15        System.out.println(m);
16        // dans une émthode main on simplifie le code en retirant la gestion des exceptions
17        File fichier = new File("./tmp/marin.ser") ;
18        // ouverture d'un flux sur un fichier
19        ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(fichier)) ;
20        // écration d'un objet àésrializer
21        // ésrialization de l'objet
22        oos.writeObject(m) ;
23        // fermeture du flux dans le bloc finally
24        oos.close();
25     }
26 }
```

deserialization d'un objet

```
1  import java.io.*;
2  public class Marin implements Serializable {
3      private static final long serialVersionUID = 7708538120786502219L;
4      private String nom, prenom ;
5      public Marin(String nom, String prenom) {
6         this.nom = nom ;
7         this.prenom = prenom ;
8     }
9      public String toString() {
10       StringBuffer sb = new StringBuffer() ;
11        return sb.append(nom).append(" ").append(prenom).toString() ;
12    }
13     public static void main(String[] args) throws FileNotFoundException, IOException {
14         Marin m = new Marin("Nadjib","Achir");
15         System.out.println(m);
16         // dans une émthode main on simplifie le code en retirant la gestion des exceptions
17         File fichier = new File("./tmp/marin.ser") ;
18    // ouverture d'un flux sur un fichier
19         ObjectInputStream ois = new ObjectInputStream(new FileInputStream(fichier)) ;
20         // éédsrialization de l'objet
21         Marin m = (Marin)ois.readObject() ;
22         System.out.println(m) ;
23    // fermeture du flux dans le bloc finally
24         oos.close();
25     }
26 }
```

Personnalisation de la sérialization

- Dans certains cas applicatifs, le mécanisme standard de sérialization proposé par Java peut se révéler inadapté, ou tout simplement ne pas convenir.

- Il est possible de le surcharger de trois manières :
  1. méthode **writeObject() readObject()**
  2. utilisation d'un externalizer
  3. utilisation d'un objet proxy

Première surcharge : méthode writeObject() readObject()

- Créer deux méthodes dans la classe que l'on souhaite sérializer
  1. private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {} : cette méthode est appelée pour reconstituer l'objet à partir d'un flux sérializé. Elle doit avoir exactement cette signature, et être privée
  2. private void writeObject(ObjectOutptStream oos) throws IOException {} : cette méthode est appelée pour écrire l'objet sur un flux sérializé. Elle doit avoir exactement cette signature, et être privée
- Lorsque la machine Java constate qu'une classe Serializable comporte ces deux méthodes, alors elle les appelle plutôt que d'utiliser ses mécanismes internes de sérialization
- writeObject() a la responsabilité d'écrire les champs de l'objet sur le flux sérializé passé en paramètre
- readObject() a la responsabilité de restaurer les valeurs des champs de l'objet $\Rightarrow$ doit correspondre au processus d'écriture utilisé par la méthode writeObject()

deserialization d'un objet

```
1  public class Marin implements Serializable {
2
3      private String nom, prenom ;
4      private int salaire ;
5
6      // suivent les getters / setters
7
8      // émthode readObject, éutilise pour reconstituer un objet éésrializ
9      private void readObject(ObjectInputStream ois)
10     throws IOException, ClassNotFoundException {
11
12        // l'ordre de lecture doit être le êmme que l'ordre dé'criture d'un objet
13        this.nom = ois.readUTF() ;
14        this.prenom = ois.readUTF() ;
15        // le salaire n'est pas relu, vu qu'il n'a pas éét écrit
16     }
17
18     // émthode writeObject, éutilise lors de la ésrialization
19     private void writeObject(ObjectOutputStream oos)
20     throws IOException {
21
22        // écriture de toute ou partie des champs d'un objet
23       oos.writeUTF(nom) ;
24       oos.writeUTF(prenom) ;
25        // on choisit de ne pas écrire le salaire, qui ne fait
26        // pas partie de lé'tat d'une instance de marin
27     }
28 }
```

Deuxième surcharge : utilisation d'un externalizer

- L'utilisation d'un externalizer fonctionne différemment
- La classe que l'on veut sérializer, dans ce cas, doit implémenter Externalizable plutôt que Serializable
- Elle doit posséder un constructeur vide, explicite ou par défaut
- L'interface Externalizable impose deux méthodes :
  1. public void writeExternal(ObjectOutput out) throws IOException est appelée pour écrire l'objet sur le flux sérializé passé en paramètre
  2. public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException est appelée pour relire cet objet
- Avantages :
  - Le flux sérializé généré est beaucoup plus court (moins d'octets) qu'un flux sérializé standard
  - Plus rapide à écrire
- Inconvénients :
  - moins sécurisée que la sérialization standard (la méthode readExternal() est ici publique, alors que readObject() est privée)

Plan du cours

Introduction

- L'API Java I/O propose le support direct pour écrire et lire des fichiers au format :
  1. GZip
  2. Zip

Flux de type gzip

- GZip est un format compressé, qui ne supporte l'écriture que d'un unique fichier
- Ne peut pas contenir de structure de répertoire, ou plusieurs fichiers
- L'écriture d'un fichier GZip utilise la classe GZIPOutputStream
- La lecture utilise la classe GZIPInputStream
- Ces classes se trouvent dans le package java.util.zip

Flux de type gzip

- Voyons maintenant la classe Marin

```
1  // dans une émthode main
2   // édclaration d'un fichier
3  File fichier = new File("tmp/bonjour.txt.gz") ;
4
5   // écration d'un flux écompress sur ce fichier
6  GZIPOutputStream gzos = new GZIPOutputStream(new FileOutputStream(fichier)) ;
7
8   // écration d'un flux data sur ce flux écompress
9  DataOutputStream dos = new DataOutputStream(gzos) ;
10
11  // écriture de Bonjour le monde !
12 dos.writeUTF("Bonjour le monde !") ;
13
14  // fermeture de ce flux, àmettre dans un bloc finally
15 gzos.close() ;
16
17  // puis ouverture du flux écompress sur ce fichier
18 GZIPInputStream gzis = new GZIPInputStream(new FileInputStream(fichier)) ;
19
20  // ouverture d'un flux data sur ce flux écompress
21 DataInputStream dis = new DataInputStream(gzis) ;
22
23  // lecture d'une îchane de ècaractres
24 String lue = dis.readUTF() ;
25
26  // fermeture du flux dans un bloc finally
27 gzis.close() ;
28
29 System.out.println(lue) ;
```

Flux de type Zip

- L'écriture de fichier **Zip** est un peu plus complexe
- Un fichier **Zip** possède une *structure interne*, et qu'il faut créer cette structure
- On peut donc créer des *répertoires*, des *fichiers*, des *fichiers dans des répertoires*, etc...
- Chaque élément, répertoire ou fichier, est modélisé par une instance de **ZipEntry**

Flux de type gzip

- Voyons maintenant la classe Marin

```
1  // dans une émthode main
2  // édclaration d'un fichier
3  File fichier = new File("tmp/bonjour.zip")
       ;
4
5   // ouverture d'un flux Zip sur ce fichier
6  ZipOutputStream zos = new ZipOutputStream(
       new FileOutputStream(fichier)) ;
7
8  // écration d'un érpertoire : il s'agit d'
       une éentre dont le nom
9  // se termine par un /
10 ZipEntry entry = new ZipEntry("vide/") ;
11 zos.putNextEntry(entry) ;
12
13 // écration d'un autre érpertoire
14 entry = new ZipEntry("bonjour/") ;
15 zos.putNextEntry(entry) ;
16
17 // écration d'un fichier Bonjour-1.txt dans
       ce érpertoire
18 entry = new ZipEntry("bonjour/Bonjour-1.txt
       ") ;
19 zos.putNextEntry(entry) ;
20
21 // ouverture d'un flux Data sur ce flux Zip
22 DataOutputStream dos = new DataOutputStream
       (zos) ;
23 // écriture d'un message dans ce fichier
24 dos.writeUTF("Bonjour le monde ! [1]") ;
25
26 // écration d'un autre fichier
27 entry = new ZipEntry("Bonjour-2.txt") ;
28 zos.putNextEntry(entry) ;
29
30 // écriture d'un message dans ce fichier
31 dos.writeUTF("Bonjour le monde ! [2]") ;
32
33 // fermeture d'un fichier dans le bloc
       finally
34 dos.close() ;
```

Flux de type gzip

- Voyons maintenant la classe Marin

```
1  // ouverture d'un flux Zip sur le fichier
       de l'exemple éèprcdent
2  ZipInputStream zis = new ZipInputStream(new
       FileInputStream(fichier)) ;
3   // ouverture d'un flux Data sur ce flux
4  DataInputStream dis = new DataInputStream(
       zis) ;
5
6   // lecture de la èpremire èentre
7   // cette émthode retourne null s'il n'y a
       plus d'éentre àlire
8  entry = zis.getNextEntry() ;
9   while (entry != null) {
10
11     if (entry.isDirectory()) {
12
13         // cette éentre est un érpertoire, on
               affiche son nom
14        System.out.println("Repertoire : " +
              entry) ;
15     } else {
16
17         // cette éentre est un fichier, on
               affiche son contenu
18        System.out.println("Fichier : " +
              entry) ;
19        System.out.println(dis.readUTF()) ;
20     }
21
22      // on boucle sur toutes les éentres
23     entry = zis.getNextEntry() ;
24 }
```

```
1  Repertoire : vide/
2  Repertoire : bonjour/
3  Fichier : bonjour/Bonjour-1.txt
4  Bonjour le monde ! [1]
5  Fichier : Bonjour-2.txt
6  Bonjour le monde ! [2]
```

Plan du cours

Introduction

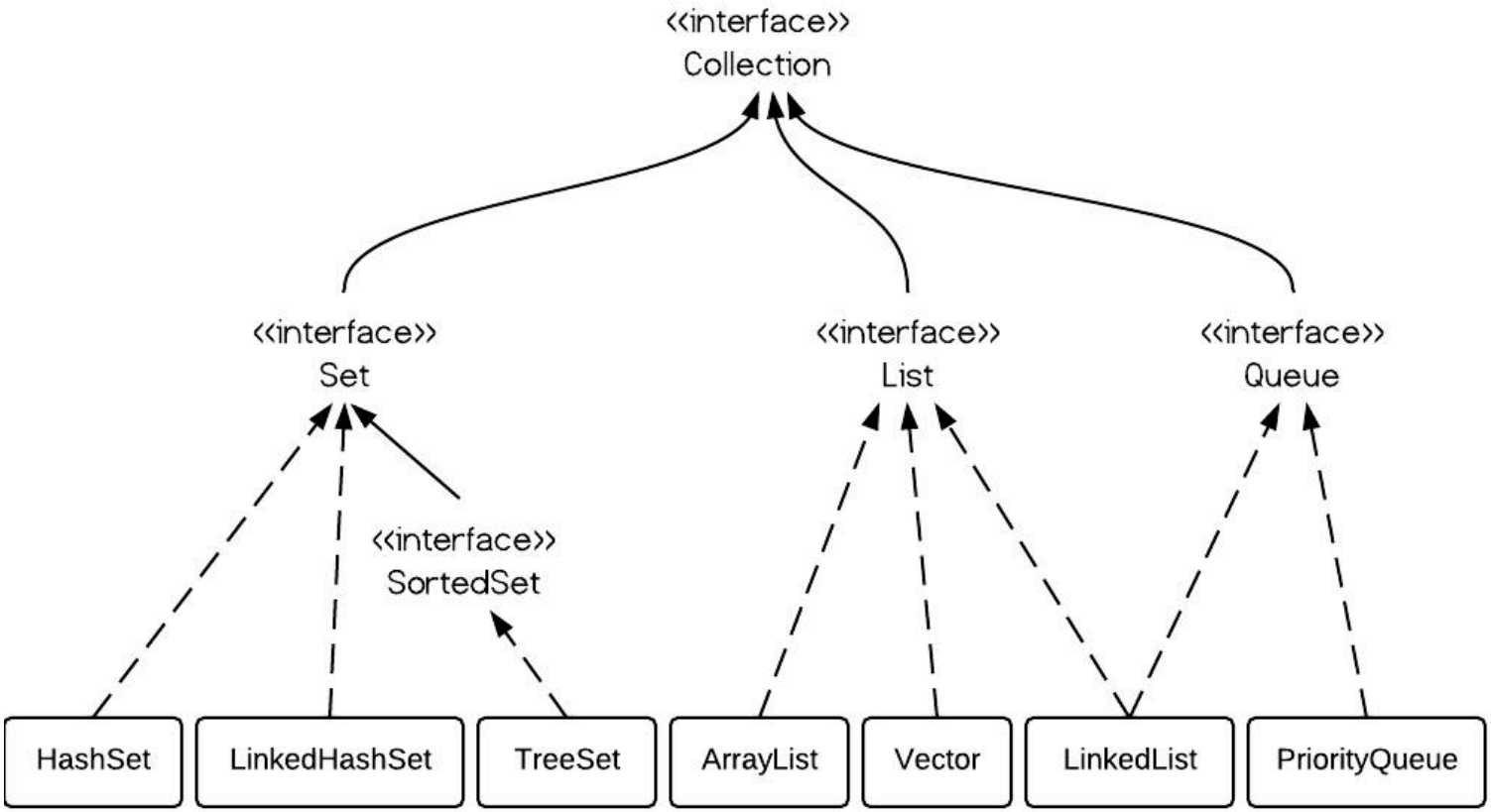
- L'API Collection est une des API utilitaires les plus utilisées en Java
- Ces interfaces nous fournissent un ensemble de méthodes qui permettent d'accéder à ces données de façon naturelle
- Les interfaces sont rangées dans le paquet java.util
- À chacune de ces interfaces sont associées des classes concrètes qui permettent de les instancier

Plan du cours

Notion de Collection

- Une collection d'objets est un ensemble d'objets
- Les opérations que l'on peut définir sur un tel ensemble sont basiques :
  1. on peut y ajouter des objets, un par un ou par paquets ;
  2. on peut retirer des objets précis, un par un ou par paquets ;
  3. on peut tester si un objet appartient à cet ensemble ;
  4. on peut balayer l'ensemble de ces objets, mais sans prévoir l'ordre dans lequel les objets contenus seront balayés ;
  5. on peut déterminer le cardinal de cet ensemble ;
  6. on peut en effacer le contenu.
- L'interface Collection est parente de trois autres interfaces : **List**, **Set** et sa descendante **SortedSet**

Notion de Collection



Détail des méthodes disponibles

- **size()** et **isEmpty()** : retourne le nombre d'éléments portés par cette collection, et un booléen qui permet de tester si cette collection est vide ou pas
- **contains(T t)** : retourne true si l'objet passé en paramètre est contenu dans cette collection
- **add(T t)** et **remove(T t)** : permet d'ajouter (resp. de retirer) un objet à cette collection
- **iterator()** : retourne un itérateur sur les éléments de cette collection
- **addAll(Collection<? extends T> collection)** et **removeAll(Collection<? extends T> collection)** : permet d'ajouter (resp. de retirer) l'ensemble des objets passés dans la collection en paramètre

Détail des méthodes disponibles

- **retainAll(Collection< ? extends T> collection)** : permet de retirer tous les éléments de la collection qui ne se trouvent pas dans la collection passée en paramètre. Cette opération réalise l'intersection des deux collections
- **containsAll(Collection< ? extends T> collection)** : retourne true si tous les éléments de la collection passée en paramètre se trouvent dans la collection courante. Cette opération teste l'inclusion
- **clear()** : efface la collection courante
- **toArray(T[] a)** : convertit la collection courante en tableau

Interface Iterator

- La méthode **iterator()** de l'interface Collection retourne une instance d' Iterator
  - **hasNext()** : retourne **true** si la collection possède encore des éléments à itérer
  - **next()** : retourne l'élément suivant
  - **remove()** : permet de retirer de la collection l'élément courant. Cette opération est optionnelle, et peut ne pas être supportée par toutes les implémentations. Si une implémentation ne supporte pas cette méthode, alors elle doit jeter l'exception UnsupportedOperationException

Exemples : Création d'une collection de `String`

```
1  // écration d'une collection de String
2  Collection<String> collection = new ArrayList<String>() ;
3
4   // ajout dé'élments àcette collection
5  collection.add("un") ;
6  collection.add("deux") ;
7  collection.add("trois") ;
```

- Pas à se soucier des problèmes de dépassement de capacité

Exemples : Test d'appartenance d'un objet à une collection

```
1  // test d'appartenance de "deux"
2  boolean b1 = collection.contains("deux") ;
3  System.out.println(b1) ; // affiche true
4
5  // test d'appartenance de "DEUX"
6  boolean b2 = collection.contains("DEUX") ;
7  System.out.println(b2) ; // affiche false
```

- Dans le cas de **ArrayList** cette méthode appelle la méthode **equals()** de l'objet passé en paramètre, pour tous les objets de la collection

Exemples : Parcourir les éléments d'une collection avec un itérateur

- Il existe deux façons de faire pour parcourir les éléments d'une collection
- La première consiste à créer un itérateur sur la collection, et à l'utiliser

```
1  // parcourir les éélments de la collection avec
2   // un éitrateur
3  Iterator<String> it = collection.iterator() ;
4   while (it.hasNext()) {
5
6      String element = it.next() ; // retourne un objet de type String
7      System.out.println(element) ;
8  }
```

- La seconde consiste à utiliser la nouvelle syntaxe **for each** , introduite en Java 5

```
1  // balayer les éélments de la collection avec
2   // un for each
3   for (String element : collection) {
4
5      System.out.println(element) ;
6  }
```

Exemples : Conversion d'une collection en tableau

- créer un tableau à partir des éléments contenus dans une collection

```
1  // pattern de conversion d'une collection en tableau
2  String [] tab = collection.toArray(new String[] {}) ;
3
4   // affichage du contenu du tableau
5   for (String element : tab) {
6
7      System.out.println(element) ;
8  }
```

Plan du cours

Notion de Collection

- L'interface **List** modélise une liste indexée par des entiers
- Lorsque l'on ajoute un objet à une liste, il prend un numéro d'ordre, géré par la liste
- Lorsque l'on en retire un, il est de la responsabilité de l'implémentation de la liste de conserver une numérotation cohérente
- On peut donc toujours demander le n *ième* élément d'une liste
- L'interface **List** étend **Collection**, et lui ajoute essentiellement deux types de méthodes :
  1. celles qui permettent de manipuler les objets directement à partir de leur numéro d'ordre
  2. celles qui permettent de parcourir la liste dans un sens ou dans l'autre

Méthodes disponibles

- **add(int index, T t)** et **addAll(int index, Collection<? extends T> collection)** : permettent d'insérer un ou plusieurs éléments à la position notée par index
- **set(int index, T t)** : permet de remplacer l'élément placé à la position index par celui passé en paramètre. L'élément qui existait est retiré de la liste, et retourné par cette méthode
- **get(int index)** : retourne l'élément placé à l'index passé en paramètre
- **remove(int index)** : retire l'élément placé à l'index passé en paramètre. Cet élément est retourné par la méthode

Méthodes disponibles

- **indexOf(Object o)** et **lastIndexOf(Object o)** : retournent respectivement le premier et le dernier index de l'objet passé en paramètre dans cette liste
- **subList(int debut, int fin)** : retourne la liste composé des éléments compris entre l'index debut, et l'index fin **-1**
  - cette sous-liste n'est pas une copie de la liste existante, mais une vue sur cette liste. Toutes les modifications de la liste principale sont donc vues au travers de cette sous-liste, et réciproquement. Nous verrons cela sur un exemple simple

Interface ListIterator

- L'interface List supporte la méthode **iterator()**
- **ListIterator** permet de parcourir une liste dans le sens croissant de ses index, ou dans le sens décroissant, et ajoute quelques méthodes supplémentaires
  - **hasPrevious()** : retourne un booléen qui est vrai s'il existe un élément à itérer dans l'ordre décroissant des index.
  - **previous()** : retourne l'élément précédent
  - **previousIndex()** et nextIndex() : retournent respectivement l'index de l'élément précédent, et l'index de l'élément suivant

Interface ListIterator

- L'interface **ListIterator** permet aussi de manipuler directement la liste que l'on est en train d'itérer
  - **add(T t)** : permet d'insérer un élément dans la liste à l'endroit où l'on se trouve, c'est-à-dire avant l'élément qui aurait été retourné par un appel à **next()**. Notons que si l'on fait un appel à **next()** après une telle insertion, ce n'est pas l'élément que l'on vient d'insérer qui est retourné, mais l'élément suivant
  - **remove(T t)** : permet de retirer de la liste l'élément que l'on vient d'itérer. Cette méthode ne peut être appelée qu'après un appel à **next()** ou **previous()**. Il n'est donc pas légal de faire plusieurs appel de suite à **remove(T t)**. Il est illégal d'appeler cette méthode juste après un appel à **add(T t)** ou **remove(T t)**
  - **set(T t)** : remplace l'élément que l'on vient d'itérer par l'élément passé en paramètre. Cette méthode ne peut être appelée qu'après un appel à **next()** ou **previous()**. Il est illégal d'appeler cette méthode juste après un appel à **add(T t)** ou **remove(T t)**

Exemples d'utilisation

- L'interface List possède trois implémentations standard dans l'API Java Collection :
  1. Vector
  2. ArrayList
  3. LinkedList
- Ces deux implémentations exposent bien sûr les mêmes fonctionnalités et la même sémantique. Cela dit, elles ne doivent pas être utilisées dans les mêmes cas
  - Un tableau permet d'accéder très rapidement à un élément donné si l'on possède son index, ce qui n'est pas le cas d'une liste chaînée
  - l'insertion d'un élément dans un tableau est un processus lourd (il faut décaler des éléments du tableau). Dans une liste chaînée ce processus est rapide : il s'agit juste d'un mouvement de pointeurs
  - augmenter la capacité d'un tableau est également un processus lourd, alors qu'une liste chaînée, par définition, n'a pas de capacité maximale
- Le type d'implémentation sera donc choisi en fonction du type de problème que l'on a à traiter

Exemples d'utilisation : Création d'une liste de String

```
1  // écration d'une liste de String
2  List<String> liste = new ArrayList<String>() ;
3
4   // ajout dé'élments àcette liste
5  liste.add("un") ;
6  liste.add("deux") ;
7  liste.add("trois") ;
8
9   // ajout d'un éélment àun index
10 liste.add(1, "avant deux") ;
11
12  // positionnement d'un éélment édonn
13 liste.set(3, "TROIS") ;
```

Exemples d'utilisation : Parcourir les éléments d'une liste avec un ListIterator

```
1  // dans une émthode main
2  // écration d'une liste
3  List<String> liste = new ArrayList<String
       >() ;
4  liste.add("un") ;
5  liste.add("deux") ;
6  liste.add("trois") ;
7
8   // écration d'un listIterator sur cette
       liste
9  ListIterator<String> it = liste.
       listIterator() ;
10
11  while(it.hasNext()) {
12     // on ajoute un éélment ésupplmentaire
           èaprs chaque éélment de la liste
13    String element = it.next() ;
14    it.add(element + " et demi") ;
15 }
16
17  // évrification du érsultat
18  for (String s : liste) {
19    System.out.println(s) ;
20 }
```

- Ce code affiche bien :

```
1  un
2  un et demi
3  deux
4  deux et demi
5  trois
6  trois et demi
```

Plan du cours

Introduction

- L'interface Set modèlise un ensemble d'objets dans lequel on ne peut pas trouver de doublons
- Necessite la définition la méthode equals() de l'objet
- À la différence des interfaces List et Collection, l'ajout d'un élément dans un Set peut donc échouer, si cet élément s'y trouve déjà
- L'implémentation fournie par l'API Java sont : HashSet et LinkedHashSet

Implémentations HashSet

- HashSet : cette classe fonctionne avec une table de hachage, dont les clés sont les codes de hachage des objets ajoutés, et les valeurs les objets eux-mêmes
- Il est donc crucial que les méthodes equals() et hashCode() des objets ajoutés à un HashSet respectent leur contrat
- Cette implémentation offre des performances constantes pour les opérations add(T t), remove(T t), contains(T t) et size()

Exemples :

```
1  // dans une émthode main
2  // écration du set
3  Set<String> set = new HashSet<String>() ;
4
5  // ajout dé'élment
6  System.out.println("J'ajoute un : " + set.
       add("un")) ;
7  System.out.println("J'ajoute deux : " + set
       .add("deux")) ;
8
9  // ajout d'un doublon : échec
10 System.out.println("J'ajoute encore un : "
       + set.add("un")) ;
11
12 // affichage de la taille du set
13 System.out.println("Taille du set : " + set
       .size()) ;
```

- L'exécution de ce code affiche le résultat suivant :

```
1  J'ajoute un : true
2  J'ajoute deux : true
3  J'ajoute encore un : false
4  Taille du set : 2
```

Plan du cours

Notion de SortedSet

- L'interface SortedSet est une extension de l'interface Set
- impose de plus que tous les objets enregistrés dans cet ensemble sont automatiquement triés dans un ordre que nous allons préciser
- L'itération sur les éléments d'un SortedSet se fait dans l'ordre croissant associés aux objets de cet ensemble
- La comparaison de deux objets n'est pas définie au niveau de la classe Object
- Il y a deux façons de faire pour comparer deux objets
  1. Implémenter l'interface Comparable : Cette interface expose une unique méthode : compareTo(T t), qui retourne un entier. Le fait que cet entier soit négatif ou positif nous dit si l'objet comparé est plus grand ou plus petit que notre obje
  2. Fournir au SortedSet, lors de sa construction, une instance de Comparator. L'interface Comparator n'expose qu'une unique méthode : compare(T t1, T t2), qui a la même sémantique que la méthode compareTo(T t) de Comparable
  3. l'implementation de SortedSet est TreeSet

Détails des méthodes disponibles

- L'interface SortedSet propose les méthodes supplémentaires suivantes :
  1. comparator() : retourne l'objet instance de Comparator qui permet la comparaison, s'il existe
  2. first() et last() : retournent le plus petit objet de l'ensemble, et le plus grand, respectivement
  3. headSet(T t) : retourne une instance de SortedSet contenant tous les éléments strictement plus petit que l'élément passé en paramètre. Ce sous-ensemble est une vue sur l'ensemble sur lequel il est construit. Il reflète donc les changements de cet ensemble, et réciproquement
  4. tailSet(T t) : retourne une instance de SortedSet contenant tous les éléments plus grands ou égaux que l'élément passé en paramètre. Ce sous-ensemble est une vue sur l'ensemble sur lequel il est construit. Il reflète donc les changements de cet ensemble, et réciproquement
  5. subSet(T inf, T sup) : retourne une instance de SortedSet contenant tous les éléments plus grands ou égaux que inf, et strictement plus petits que sup. Là encore, ce sous-ensemble est une vue sur l'ensemble sur lequel il est construit, qui reflète donc les changements de cet ensemble, et réciproquement

Exemples d'utilisation : classe Comparable

```
1  public class ComparableMarin implements Comparable<Marin> {
2
3      // deux champs classiques
4      private String nom, prenom ;
5
6      // suivent les getters et les setters
7
8      // suit la surcharge de equals() et de hashCode()
9
10     // surcharge de toString()
11     public String toString() {
12
13        // un bonne émthode toString() est une émthode
14        // qui ne fait pas de éconcatnation de îchane !
15       StringBuffer sb = new StringBuffer() ;
16       sb.append(nom).append(" ").append(prenom) ;
17        return sb.toString() ;
18    }
19
20     // émthode éimpose par Comparable<Marin>
21     public int compareTo(Marin m) {
22
23        // une version ècomplte de cette émthode
24        // doit égrer le cas ùo nom et prenom sont nuls
25        if (getNom().equals(m.getNom())) {
26           return getPrenom().compareTo(m.getPrenom()) ;
27       }  else {
28           return getNom().compareTo(m.getNom()) ;
29       }
30    }
31 }
```

Exemples d'utilisation

```
1  // dans une émthode main
2  SortedSet<ComparableMarin> set = new TreeSet<ComparableMarin>() ;
3
4  ComparableMarin m1 = new ComparableMarin("Surcouf", "Alain") ;
5  ComparableMarin m2 = new ComparableMarin("Tabarly", "Eric") ;
6  ComparableMarin m3 = new ComparableMarin("Auguin", "Christophe") ;
7  ComparableMarin m4 = new ComparableMarin("Surcouf", "Robert") ;
8
9  set.add(m1) ;
10 set.add(m2) ;
11 set.add(m3) ;
12 set.add(m3) ;
13 set.add(m4) ;
14
15  for (ComparableMarin m : set) {
16    System.out.println(m) ;
17 }
```

```
1  Auguin Christophe
2  Surcouf Alain
3  Surcouf Robert
4  Tabarly Eric
```

Exemples d'utilisation : avec Comparator

```
1  // dans une émthode main édfinition d'une classe anonyme, éimplmentation de Comparator<Marin>
2  SortedSet<Marin> set = new TreeSet<Marin>(new Comparator<Marin>() {
3      public int compare(Marin m1, Marin m2) {
4         if (m1.getNom().equals(m2.getNom())) {
5            return m1.getPrenom().compareTo(m2.getPrenom()) ;
6        } else {
7            return m1.getNom().compareTo(m2.getNom()) ;
8        }
9     }
10 }) ;
11 Marin m1 = new Marin("Surcouf", "Alain") ;
12 Marin m2 = new Marin("Tabarly", "Eric") ;
13 Marin m3 = new Marin("Auguin", "Christophe") ;
14 Marin m4 = new Marin("Surcouf", "Robert") ;
15
16 set.add(m1) ;
17 set.add(m2) ;
18 set.add(m3) ;
19 set.add(m3) ;
20 set.add(m4) ;
21
22  for (Marin m : set) {
23    System.out.println(m) ;
24 }
```

```
1  Auguin Christophe
2  Surcouf Alain
3  Surcouf Robert
4  Tabarly Eric
```

Plan du cours

Notion de file d'attente

- Une file d'attente est une collection normale
- L'utilisation classique d'une file d'attente, est de servir de tampon entre une source d'objets et un consommateur de ces mêmes objets
- Traditionnellement, une file d'attente expose trois types de méthodes :
  - ajout d'un objet dans la file ;
  - examen de l'objet suivant disponible ;
  - consommation de l'objet suivant disponible, et retrait de la file.

Détail des méthodes disponibles : Interface Queue

- L'interface Queue, qui modélise une file d'attente simple, expose six méthodes, qui sont les suivantes
  - add(T t) et offer(T t) permettent d'ajouter un élément à la liste. Si la capacité maximale de la liste est atteinte, alors add() jette une exception de type IllegalStateException, et offer() retourne false
  - remove() et poll() retirent toutes les deux de cette file d'attente. Si aucun élément n'est disponible, alors remove() jette une exception de type NoSuchElementException, tandis que poll() retourne null
  - element() et peek() examinent toutes les deux l'élément disponible, sans le retirer de la file d'attente. Si aucun élément n'est disponible, alors element() jette une exception de type NoSuchElementException, tandis que peek() retourne null

Détail des méthodes disponibles : Interface Deque

- Elle définit la notion de file d'attente à double extrémité
- Il est possible d'ajouter des éléments au début de la file ou à la fin, avec la même sémantique que pour Queue
  - addFirst(T t) / addLast(T t)
  - removeFirst(T t) / removeLast(T t)
  - getFirst(T t) / getLast(T t)

Plan du cours

Notion de table de hachage

- Une table de hachage est une structure de données qui associe des clés à des valeurs
- Une telle structure doit au moins exposer les fonctionnalités suivantes :
  - une méthode de type put(key, value), qui permet d'associer un objet à une clé ;
  - une méthode de type get(key), qui retourne la valeur qui a été associée à cette clé, ou null s'il n'y en a pas.
  - une méthode de type remove(key), qui supprime la clé de cette table, et la valeur qui lui est associée.

Interface Map

- Cette interface modélise la forme la plus simple d'une table de hachage
- Elle expose les méthodes de base suivantes :
  - put(K key, V value) et get(K key) : ces deux méthodes permettent d'associer une clé à une valeur, et de récupérer cette valeur à partir de cette clé, respectivement
  - remove(K key) : permet de supprimer la clé passée en paramètre de cette table, et la valeur associée
  - keySet() : retourne l'ensemble de toutes les clés de cette table de hachage. Cet ensemble ne peut pas contenir de doublons, il s'agit d'un Set<K>, donc les éléments sont de type K. Cet ensemble est une vue sur les clés de la table de hachage. Donc les éléments ajoutés à cette table seront vus dans ce Set. Il supporte les méthodes remove() et removeAll(), mais pas les méthodes d'ajout d'éléments. Retirer une clé de cet ensemble retire également la valeur qui lui est associée

Interface Map

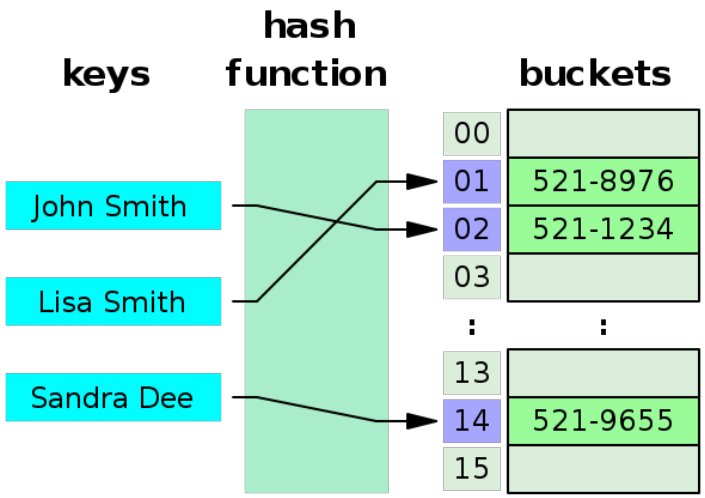
- Elle expose les méthodes de base suivantes :
  - values() : retourne l'ensemble de toutes les valeurs stockées dans cette table de hachage. À la différence de l'ensemble des clés, l'ensemble des valeurs peut contenir des doublons. Il est donc de type Collection<V>. Cette collection est également une vue sur la table : toute valeur ajoutée à la table sera vue dans cette collection. Elle supporte les méthode remove() et removeAll(), qui ont pour effet de retirer également la clé associée à cette valeur, mais pas les méthodes d'ajout
  - entrySet() : retourne l'ensemble des entrées de cette table de hachage. Cet ensemble est un Set, dont les éléments sont de type Map.Entry. Nous allons voir l'interface Map.Entry dans la suite de cette partie. Cet ensemble est lui aussi une vue sur la table, qui reflète donc les modifications qui peuvent y être faites. Il supporte les opérations de retrait d'éléments, mais pas les opérations d'ajout

Interface Map

- Plusieurs autres méthodes utilitaires sont ajoutées à cette interface
  - clear() : efface tout le contenu de la table
  - size() et isEmpty() : retourne le cardinal de la table, et un booléen qui indique si cette table est vide ou pas
  - putAll(Map map) : permet d'ajouter toutes les clés de la table passée en paramètre à la table courante
  - containsKey(K key) et containsValue(V value) : permettent de tester si la clé ou la valeur passée en paramètre sont présentes dans cette table

Classe HashMap

- Cette classe implémente l'interface Map, en utilisant une table hachée
- Les valeurs ou les clés peuvent être null
- Il n'y a aucun ordre garanti sur les éléments stockés dans le HasMap, et de plus l'ordre des éléments n'est pas forcément le même à deux instants différents



HashMap

Exemples d'utilisation : HashMap

```
1  // dans une émthode main écration de quelques marins
2  Marin m1 = new Marin("Surcouf", "Alain") ;
3  Marin m2 = new Marin("Tabarly", "Eric") ;
4  Marin m3 = new Marin("Auguin", "Christophe") ;
5  Marin m4 = new Marin("Surcouf", "Robert") ;
6
7   // écration d'une table dont les écls sont des String
8   // et les valeurs des marins
9  Map<String, Marin> map = new HashMap<String, Marin>() ;
10
11  // ajout de nos marins àcette table
12  // remarquons que deux ajouts èpossdent la êmme écl
13 map.put(m1.getNom(), m1) ;
14 map.put(m2.getNom(), m2) ;
15 map.put(m3.getNom(), m3) ;
16 map.put(m4.getNom(), m4) ;
17
18  // èpremire interrogation de la table
19 System.out.println("[Surcouf] -> " + map.get("Surcouf")) ;
20 System.out.println("éElments : " + map.size()) ;
21
22  // parcours de la table par ses éentres
23  for (Map.Entry<String, Marin> entry : map.entrySet()) {
24    System.out.println("[" + entry.getKey() + "] -> " + entry.getValue()) ;
25 }
```

Exemples d'utilisation : HashMap

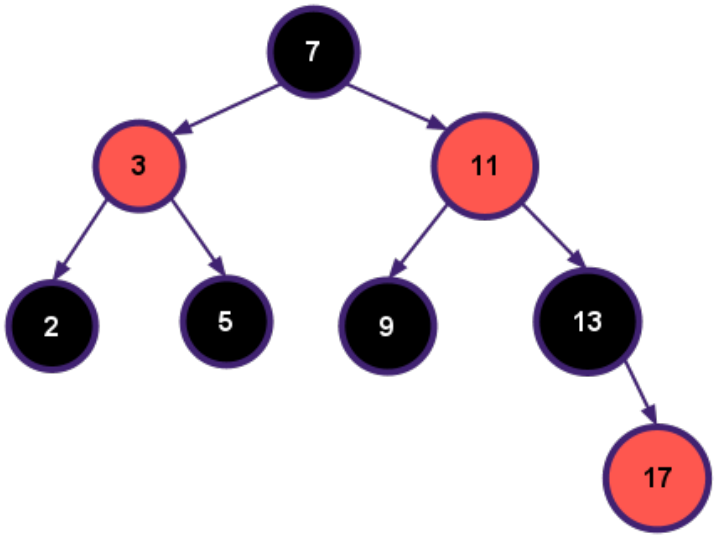
```
1 [Surcouf] -> Surcouf Roberté
2 Elments : 3
3 [Tabarly] -> Tabarly Eric
4 [Surcouf] -> Surcouf Robert
5 [Auguin] -> Auguin Christophe
```

Interfaces SortedMap et NavigableMap

- L'interface SortedMap permet de stocker des tables de hachage triées dans l'ordre de ses clés
- cet ordre peut être défini de deux façons, soit par le fait que les clés de cette table sont des objets Comparable, soit en donnant un objet Comparator à la construction de la table
- On retrouve les méthodes :
  - firstKey() et lastKey() : retournent respectivement la plus petite clé et la plus grande
  - headMap(K toKey) et tailMap(K fromKey) : retournent des vues sur la table maître, contenant les premières clés jusqu'à toKey, et les dernières clés à partir de fromKey, respectivement
  - subMap(K fromKey, K, toKey) : retourne une vue sur la table maître, de la clé fromKey, à la clé toKey
- L'interface NavigableMap est une extension de SortedMap, qui complète le jeu de méthodes de sélection de vues sur la table maître

Classe TreeMap

- Le conteneur TreeMap permet de stocker des couples (clé, valeur), dans une structure d'arbre binaire équilibré rouge-noir
- Cette classe garantit que la collection Map sera triée selon un ordre croissant, conformément à l'ordre naturel des clés ou à l'aide d'un comparateur fourni au moment de la création de l'objet TreeMap



`TreeMap`

Exemples d'utilisation : TreeMap

```
1  Marin m1 = new Marin("Surcouf", "Robert") ;
2  Marin m2 = new Marin("Tabarly", "Eric") ;
3  Marin m3 = new Marin("Auguin", "Christophe") ;
4
5  NavigableMap<String, Marin> map = new TreeMap<String, Marin>() ;
6  map.put(m1.getNom(), m1) ;
7  map.put(m2.getNom(), m2) ;
8  map.put(m3.getNom(), m3) ;
9
10 System.out.println("éElments : " + map.size()) ;
11
12  for (Map.Entry<String, Marin> entry : map.entrySet()) {
13    System.out.println("[" + entry.getKey() + "] -> " + entry.getValue()) ;
14 }
15
16  // construction d'une vue en ordre inverse
17 System.out.println("Inversion...") ;
18 NavigableMap<String, Marin> reversedMap = map.descendingMap() ;
19  for (Map.Entry<String, Marin> entry : reversedMap.entrySet()) {
20    System.out.println("[" + entry.getKey() + "] -> " + entry.getValue()) ;
21 }
22
23  // suppression de Tabarly de la table îmatre
24  // observons ce qui se passe pour la vue
25 System.out.println("Suppression de Tabarly...") ;
26 map.remove("Tabarly") ;
27  for (Map.Entry<String, Marin> entry : reversedMap.entrySet()) {
28     System.out.println("[" + entry.getKey() + "] -> " + entry.getValue()) ;
29 }
```

Exemples d'utilisation : TreeMap

```
1  Elements : 3
2  [Auguin] -> Auguin Christophe
3  [Surcouf] -> Surcouf Robert
4  [Tabarly] -> Tabarly Eric
5  Inversion...
6  [Tabarly] -> Tabarly Eric
7  [Surcouf] -> Surcouf Robert
8  [Auguin] -> Auguin Christophe
9  Suppression de Tabarly...
10 [Surcouf] -> Surcouf Robert
11 [Auguin] -> Auguin Christophe
```

Plan du cours

Introduction

- Nouveauté la plus significative du langage de programmation Java depuis la version 1.0
- L'introduction des génériques dans Java 5
- La programmation générique implique d'écrire du code qui puisse être réutilisé pour des objets de types différent
- Nous n'avons plus besoin, par exemple, de programmer des classes différentes pour collecter nos objets

Une première classe générique

```
1  // Classe ArrayList de Java 4
2  public class ArrayList {
3
4      public Object get() { ... }
5
6      public void add(Object object) { ... }
7  }
```

- Deux reproches :
  - À chaque fois que l'on lit un objet d'une liste, on doit le caster dans le bon type. Par exemple, pour une liste de String, on devra écrire (String)list.get()
  - La classe n'impose aucun contrôle lorsque l'on ajoute des objets. Si, par erreur, on introduit d'autres objets que des String dans une telle liste, on ne pourra s'en rendre compte qu'à la lecture du contenu de la liste
- pouvoir déclarer qu'une liste contient des String, pouvoir détecter dès la compilation que l'on est en train d'ajouter d'autres éléments que des String dans notre liste

Une première classe générique

- En utilisant les génériques, on peut écrire notre classe ArrayList de la façon suivante :

```
1  // Classe ArrayList de Java 5
2  public class ArrayList<T> {
3      public T get() { ... }
4      public void add(T t) { ... }
5  }
```

- Dans cette écriture, T est le type générique
- Lors de l'instanciation de cette classe, on doit lui donner une valeur, qui doit être une classe ou une interface
- Dans le cas où T représente le type String, notre classe devient :

```
1  public class ArrayList<String> {
2      public String get() { ... }
3      public void add(String s) { ... }
4  }
5  // declaration
6  ArrayList<String> listOfString = new ArrayList<String>() ;
7  ArrayList<Integer> listOfInteger = new ArrayList<Integer>() ;
```

Une première méthode générique

- On peut également déclarer un type générique à une méthode, que cette méthode appartienne à une classe générique ou non
- La déclaration est la suivante :

```
1 public class ArrayUtils {
2     public static <T> T getFirst(T[] arrayOfT) { return arrayOfT[0]; }
3 }
```

- La déclaration du type générique doit se faire en déclarant ce type entre brackets après les modificateurs de méthode (ici public static)
- la syntaxe générale d'appel d'une telle méthode :

```
1 String[] arrayOfT = {"Cheese", "Pepperoni", "Black Olives"};
2 String first = ArrayUtils.<String>getFirst(arrayOfT) ;
3 System.out.println(first);
```

Contraindre un type générique

- getFirst $\Rightarrow$ déterminer le plus petit élément d'un tableau :

```
1  public class ArrayUtils {
2      public static <T extends Comparable<T>> T getFirst(T[] arrayOfT) {
3        T min = arrayOfT[0] ;
4
5         for (T t : arrayOfT) {
6            if (t.compareTo(min) < 0) {
7              min = t ;
8            }
9        }
10        return min ;
11     }
12 }
```

- Contraintes multiples sur un type générique

```
1  public class ArrayUtils {
2      public static <T extends Comparable & Serializable> T getFirst(T[] arrayOfT) { ... }
3  }
```

Exemple

```
1  public class Solo<T> {
2
3    //Variable d'instance
4    private T valeur;
5
6    //Constructeur par édfaut
7    public Solo(){
8      this.valeur = null;
9    }
10
11   //Constructeur avec èparamtre inconnu pour l'instant
12   public Solo(T val){
13     this.valeur = val;
14   }
15
16   //éDfinit la valeur avec le èparamtre
17   public void setValeur(T val){
18     this.valeur = val;
19   }
20
21   //Retourne la valeur éàdj «écaste »par la signature de la émthode !
22   public T getValeur(){
23     return this.valeur;
24   }
25 }
```

Exemple

```
1  public static void main(String[] args) {
2    Solo<Integer> val = new Solo<Integer>(12);
3    int nbre = val.getValeur();
4  }
```

```
1  public static void main(String[] args) {
2    Solo<Integer> val = new Solo<Integer>();
3    Solo<String> valS = new Solo<String>("TOTOTOTO");
4    Solo<Float> valF = new Solo<Float>(12.2f);
5    Solo<Double> valD = new Solo<Double>(12.202568);
6  }
```

Généricité et collections

```
1  public static void main(String[] args) {
2
3    System.out.println("Liste de String");
4    System.out.println("
           ------------------------------");
5    List<String> listeString= new ArrayList<
           String>();
6    listeString.add("Une îchane");
7    listeString.add("Une autre");
8    listeString.add("Encore une autre");
9    listeString.add("Allez, une èdernire");
10
11   for(String str : listeString)
12     System.out.println(str);
13
14   System.out.println("\nListe de float");
15   System.out.println("
           ------------------------------");
16
17   List<Float> listeFloat = new ArrayList<
           Float>();
18   listeFloat.add(12.25f);
19   listeFloat.add(15.25f);
20   listeFloat.add(2.25f);
21   listeFloat.add(128764.25f);
22
23   for(float f : listeFloat)
24     System.out.println(f);
25 }
```

```
1  Liste de String
2  ------------------------------
3  Une îchane
4  Une autre
5  Encore une autre
6  Allez, une èdernire
7
8  Liste de float
9  ------------------------------
10 12.25
11 15.25
12 2.25
13 128764.25
```

Plan du cours

Introduction

Objectif

L'**introspection**, consiste à découvrir de façon dynamique des informations relatives à une classe ou à un objet. C'est notamment utilisé au niveau de la machine virtuelle Java lors de l'exécution du programme. En gros, la machine virtuelle stocke les informations relatives à une classe dans un objet.

L'**introspection** n'est que le moyen de connaître toutes les informations concernant une classe donnée. Vous pourrez même créer des instances de classe de façon dynamique grâce à cette notion.

Introduction

Comme nous l'avons déjà vu, une classe en Java est représentée par une instance de la classe **Class**. Lorsqu'une classe est chargée par un des **ClassLoader** de la JVM, cet objet **Class** est créé.

Cet objet comporte un jeu de méthodes qui permet d'explorer le contenu d'une classe. Notamment :

- les annotations qu'elle porte
- la classe qu'elle étend, et les interfaces qu'elle implémente, directement ou via ses super-classes
- ses champs, statiques ou non, modélisés par la classe **Field**
- ses constructeurs, modélisés par la classe **Constructor**
- ses méthodes, statiques ou non, modélisées par la classe **Method**

Plan du cours

Utilisation de **Class**

La classe **Class** est le point d'entrée de l'exploration du contenu d'une classe par introspection

On peut obtenir une instance de **Class** de trois façons :

- en invoquant la méthode **getClass()** sur n'importe quel objet
- en appelant cet objet directement, par exemple **String.class** ou **Marin.class** sont des objets instances de **Class**
- en utilisant la méthode statique **Class.forName(String)**, à laquelle on passe le nom complet d'une classe, avec le nom du package dans lequel elle se trouve

Utilisation de Class

- Une classe est toujours chargée par un class *loader* en `Java`
- Un class *loader* est une instance de la classe **ClassLoader**
- On peut obtenir une référence sur le class *loader* qui a chargé une classe donnée en utilisant la méthode **getClassLoader()**

Méthodes disponibles

La classe Class expose un jeu de méthodes qui permet d'obtenir :

- les constructeurs,
- les champs, et
- les méthodes de cette classe

Chacune de ces méthodes existe en quatre versions.

Méthodes disponibles

- Les méthodes susceptibles de retourner plusieurs champs les retournent dans un tableau
- Si aucun champ ne correspond à ce qui est demandé, ce tableau est vide
- Aucune méthode qui doit retourner un tableau ne retourne null dans cette `API`
- Exemple : **getDeclaredField(String name)**, retourne le champ dont le nom est précisé, qu'il soit public ou non, mais uniquement s'il se trouve dans cette classe

En résumé, on peut demander la liste des champs, constructeurs ou méthodes, qui est retournée dans un tableau

Méthodes disponibles : Constructeurs

- **getConstructors()** et **getConstructor(Class< ?>... types)** : retournent le tableau des constructeurs publics de cette classe, ou le constructeur qui prend en paramètre la liste des classes (donc des types) indiquée

- **getDeclaredConstructors()** et **getDeclaredConstructor(Class< ?>... types)** : retournent le tableau des constructeurs de cette classes, publics ou non, ou le constructeur qui prend en paramètre la liste des classes (donc des types) indiquée

Méthodes disponibles : Constructeurs

```
1  // nom de la classe
2  public class Marin {
3  // (1) champs
4      String nom, prenom ;
5      int salaire ;
6  // (2.a) mecansime d'instantiation ou constructeur
7      public Marin (String nouveauNom, String nouveauPrenom, int nouveauSalaire) {
8          nom = nouveauNom ;
9          prenom = nouveauPrenom ;
10         salaire = nouveauSalaire ;
11     }
12 // (2.b) mecanisme d'instantiation ou constructeur
13     public Marin (String nouveauNom, int nouveauSalaire) {
14         nom = nouveauNom ;
15         prenom = "" ;
16         salaire = nouveauSalaire ;
17     }
18 // (2.c) mecanisme d'instantiation ou constructeur
19     private Marin (int nouveauSalaire) {
20         nom = "Nadjib" ;
21         prenom = "Achir" ;
22         salaire = nouveauSalaire ;
23     }
24 // (3) methode
25     public void augmenteSalaire (int montant) {
26         salaire = salaire + montant ;
27     }
28 }
```

Méthodes disponibles : Constructeurs

```
1  import java.lang.reflect.*;
2
3  public class ClassDemo_getConstructor {
4
5     public static void main(String[] args) {
6
7       try {
8          Class cls = Class.forName("Marin");
9          System.out.println("getConstructor ==> \n Marin Constructors =");
10
11         /* returns the array of Constructor objects representing the public
12         constructors of this class */
13         Constructor c[] = cls.getConstructors();
14         for(int i = 0; i < c.length; i++) {
15            System.out.println(c[i]);
16         }
17
18         System.out.println("getDeclaredConstructors ==> \n Marin Constructors =");
19         /* returns the array of Constructor objects representing the public
20         constructors of this class */
21         Constructor cd[] = cls.getDeclaredConstructors();
22         for(int i = 0; i < cd.length; i++) {
23            System.out.println(cd[i]);
24         }
25       }
26       catch (Exception e) {
27          System.out.println("Exception: " + e);
28      }
29    }
30 }
```

Méthodes disponibles : Constructeurs

```
1  getConstructor ==>
2  Marin Constructors =
3  public Marin(java.lang.String,int)
4  public Marin(java.lang.String,java.lang.String,int)
5
6  getDeclaredConstructors ==>
7  Marin Constructors =
8  private Marin(int)
9  public Marin(java.lang.String,int)
10 public Marin(java.lang.String,java.lang.String,int)
```

Méthodes disponibles : Méthodes

- **getMethods()** et **getMethod(String, Class<?>... types)** : retournent le tableau des méthodes publiques de cette classe, et de ses super-classes, ou la méthode qui correspond aux paramètres spécifiés

- **getDeclaredMethods()** et **getDeclaredMethod(String, Class<?>... types)** : retournent le tableau des méthodes de cette classe, publiques ou non, ou la méthode qui correspond aux paramètres précisés.

Méthodes disponibles : Méthodes

```java
// nom de la classe
public class Marin {
// (1) champs
    String nom, prenom ;
    int salaire ;
    public Marin () {
    }
// (2.a) mecansime d'instantiation ou constructeur
    public Marin (String nouveauNom, String nouveauPrenom, int nouveauSalaire) {
        nom = nouveauNom ; prenom = nouveauPrenom ; salaire = nouveauSalaire ;
    }
// (2.b) mecanisme d'instantiation ou constructeur
    public Marin (String nouveauNom, int nouveauSalaire) {
        nom = nouveauNom ; prenom = "" ; salaire = nouveauSalaire ;
    }
// (2.c) mecanisme d'instantiation ou constructeur
    private Marin (int nouveauSalaire) {
        nom = "Nadjib" ; prenom = "Achir" ; salaire = nouveauSalaire ;
    }
// (3) methode
    public void augmenteSalaire (int montant) {
        salaire = salaire + montant ;
    }
// (4) methode
    public int getSalaire() {
        return salaire;
    }
// (5) methode
    public void SetSalaire(int l) {
        this.salaire = l;
    }
}
```

Méthodes disponibles : Méthodes

```
1  import java.lang.reflect.*;
2
3  public class ClassDemo_getMethods {
4
5     public static void main(String[] args) {
6
7       try {
8          Class cls = Class.forName("Marin");
9          System.out.println("getMethods ==> ");
10
11         Method m[] = cls.getMethods();
12         for(int i = 0; i < m.length; i++) {
13            System.out.println(m[i]);
14         }
15
16         System.out.println("\ngetDeclaredMethods ==> ");
17         Method dm[] = cls.getDeclaredMethods();
18         for(int i = 0; i < dm.length; i++) {
19            System.out.println(dm[i]);
20         }
21      }
22      catch (Exception e) {
23         System.out.println("Exception: " + e);
24      }
25    }
26 }
```

Méthodes disponibles : Méthodes

```
1  getMethods ==>
2  public void Marin.augmenteSalaire(int)
3  public int Marin.getSalaire()
4  public void Marin.SetSalaire(int)
5  public final void java.lang.Object.wait(long,int) throws java.lang.
       InterruptedException
6  public final native void java.lang.Object.wait(long) throws java.lang.
       InterruptedException
7  public final void java.lang.Object.wait() throws java.lang.
       InterruptedException
8  public boolean java.lang.Object.equals(java.lang.Object)
9  public java.lang.String java.lang.Object.toString()
10 public native int java.lang.Object.hashCode()
11 public final native java.lang.Class java.lang.Object.getClass()
12 public final native void java.lang.Object.notify()
13 public final native void java.lang.Object.notifyAll()
14
15 getDeclaredMethods ==>
16 public void Marin.augmenteSalaire(int)
17 public int Marin.getSalaire()
18 public void Marin.SetSalaire(int)
```

Méthodes disponibles : Champs

- **getFields()** et **getField(String field)** : retournent le tableau des champs publics de cette classe, ou le champ dont le nom est passé en paramètre. Les champs publics des super-classes sont présents dans la liste

- **getDeclaredFields()** et **getDeclaredField(String field)** : retournent le tableau des champs déclarés dans cette classes seulement, publics ou non.

Méthodes disponibles : Champs

```
1  // nom de la classe
2  public class Marin {
3  // (1) champs
4      private String nom, prenom ;
5      private int salaire ;
6      public int tmp1;
7      protected int tmp2;
8
9      public Marin () {
10     }
11 }
```

Méthodes disponibles : Champs

```
1  import java.lang.reflect.*;
2
3  public class ClassDemo_getFields {
4
5     public static void main(String[] args) {
6
7       try {
8          Class cls = Class.forName("Marin");
9          System.out.println("getFields ==>");
10
11         // returns the array of Field objects representing the public fields
12         Field f[] = cls.getFields();
13         for (int i = 0; i < f.length; i++) {
14            System.out.println(f[i]);
15         }
16
17         System.out.println("\ngetDeclaredFields ==> ");
18         Field df[] = cls.getDeclaredFields();
19         for (int i = 0; i < df.length; i++) {
20            System.out.println(df[i]);
21         }
22      }
23      catch (Exception e) {
24         System.out.println("Exception: " + e);
25      }
26   }
27 }
```

Méthodes disponibles : Champs

```
1  getFields ==>
2  public int Marin.tmp1
3
4  getDeclaredFields ==>
5  private java.lang.String Marin.nom
6  private java.lang.String Marin.prenom
7  private int Marin.salaire
8  public int Marin.tmp1
9  protected int Marin.tmp2
```

Méthodes disponibles : autres

Les méthodes suivantes permettent d'obtenir des informations sur cette classe

- **getCanonicalName()** : retourne le nom complet de cette classe, s'il existe. Certaines classes n'ont pas de nom, notamment les classes anonymes (c'est d'ailleurs la raison pour laquelle on les appelle anonymes !)

- **getName()** : retourne le nom du type de cette classe (class, interface, type primitif ou tableau)

- **getSimpleName()** : retourne le nom de cette classe, sans son nom de package

Méthodes disponibles : autres

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

public class ClassDemo_getCanonicalName {

   public static void main(String[] args) {
     try {
      Class cls = Class.forName("Marin");

      // returns the canonical name of the underlying class if it exists
      System.out.println("Class (getCanonicalName) = " + cls.getCanonicalName());
      System.out.println("Class (getName) = " + cls.getName());
      System.out.println("Class (getSimpleName) = " + cls.getSimpleName());
      System.out.println("Class (getPackage) = " + cls.getPackage());
     }
     catch (Exception e) {
      System.out.println("Exception: " + e);
     }
   }
}
```

```
Class (getCanonicalName) = Marin
Class (getName) = Marin
Class (getSimpleName) = Marin
Class (getPackage) = null
```

Méthodes disponibles : autres

Les méthodes suivantes permettent d'obtenir des informations sur cette classe

- **getPackage()** : retourne une instance de la classe Package, qui modélise le package dans lequel se trouve cette classe

- **getSuperClass()** : retourne la super-classe de cette classe, sous forme d'une instance de **Class**

Méthodes disponibles : **getPackage()**

```
1  import java.lang.*;
2
3  public class ClassDemo {
4
5     public static void main(String[] args) {
6
7       try {
8          Class cls = Class.forName("java.lang.Integer");
9
10         // returns the name and package of the class
11         System.out.println("Class = " + cls.getName());
12         System.out.println("Package = " + cls.getPackage());
13      }
14      catch(ClassNotFoundException ex) {
15         System.out.println(ex.toString());
16      }
17    }
18 }
```

```
1  Class = java.lang.Integer
2  Package = package java.lang, Java Platform API Specification, version 1.8
```

Méthodes disponibles : **getSuperClass()**

```
1  import java.lang.*;
2  class superClass { // super class
3  }
4  class subClass extends superClass { // sub class
5  }
6  public class ClassDemo_getSuperclass {
7     public static void main(String args[]) {
8       superClass val1 = new superClass();
9       subClass val2 = new subClass();
10      Class cls;
11
12      cls = val1.getClass();
13      System.out.println("val1 is object of type = " + cls.getName());
14
15      cls = cls.getSuperclass();
16      System.out.println("super class of val1 = " + cls.getName());
17
18      cls = val2.getClass();
19      System.out.println("val2 is object of type = " + cls.getName());
20
21      cls = cls.getSuperclass();
22      System.out.println("super class of val2 = " + cls.getName());
23    }
24 }
```

```
1  val1 is object of type = superClass
2  super class of val1 = java.lang.Object
3  val2 is object of type = subClass
4  super class of val2 = superClass
```

Plan du cours

Type d'une classe

- La classe **Class** expose également un jeu de méthodes qui permettent de tester à quel type de classe nous avons affaire
- Ce type peut être de différentes natures. Voyons ces méthode
  - **isInterface()** : retourne **true** si cette classe est une interface
  - **isEnum()** : retourne **true** si cette classe est une énumération
  - **isLocalClass()** et **isMemberClass()** : retourne **true** s'il s'agit d'une classe locale ou membre, respectivement

Méthodes disponibles : **isInterface()**

```
1  import java.lang.*;
2
3  public class ClassDemo {
4
5     public static void main(String[] args) {
6
7       ClassDemo c = new ClassDemo();
8       Class cls = c.getClass();
9
10      // determines if the specified Class object represents an interface type
11      boolean retval = cls.isInterface();
12      System.out.println("It is an interface ? " + retval);
13    }
14 }
```

```
1  It is an interface ? false
```

Méthodes disponibles : **isEnum()**

```
1  import java.lang.*;
2
3  // enum showing programming languages
4  enum Language {
5     C, Java;
6  }
7
8  public class ClassDemo {
9
10    public static void main(String args[]) {
11
12      System.out.println(Language.class.isEnum());
13    }
14 }
```

```
1  true
```

Méthodes disponibles : **isLocalClass()**

```
1  import java.lang.*;
2
3  public class ClassDemo {
4
5     public static void main(String[] args) {
6
7       ClassDemo c = new ClassDemo();
8       Class cls = c.getClass();
9
10      // returns the name of the class
11      String name = cls.getName();
12      System.out.println("Class Name = " + name);
13
14      // returns true if and only if this class is a local class
15      boolean retval = cls.isLocalClass();
16       System.out.println("Is this LocalClass? " + retval);
17    }
18 }
```

```
1  Class Name = ClassDemo
2  Is this LocalClass? false
```

Type d'une classe

- **isPrimitive()** : retourne **true** si cette classe modélise un type primitif. Notons que tous les types primitifs sont associés à des classes : **int.class, float.class**, etc

- **isArray()** : retourne **true** si cette classe modélise un tableau

- **isSynthetic()** : retourne **true** si cette classe est une classe synthétique. Une classe synthétique est une classe créée dynamiquement, par le compilateur ou à l'exécution du code. Une classe synthétique est par exemple associée à chaque clause **switch**

Méthodes disponibles : **isPrimitive()**

```java
import java.lang.*;

public class ClassDemo_isPrimitive {

   public static void main(String[] args) {

     // returns the Class object associated with this class
     ClassDemo_isPrimitive cl = new ClassDemo_isPrimitive();
     Class c1Class = cl.getClass();

     // returns the Class object associated with an integer
     Class kClass = int.class;

     // checking for primitive type
     boolean retval1 = c1Class.isPrimitive();
     System.out.println("c1 is primitive type? = " + retval1);

     // checking for primitive type?
     boolean retval2 = kClass.isPrimitive();
     System.out.println("k is primitive type? = " + retval2);
   }
}
```

```
c1 is primitive type? = false
k is primitive type? = true
```

Méthodes disponibles : **isArray()**

...

Méthodes disponibles : **isSynthetic()**

```
1  import java.lang.*;
2
3  public class ClassDemo {
4
5     public static void main(String[] args) {
6
7       ClassDemo c = new ClassDemo();
8       Class cls = c.getClass();
9
10      // returns true if this class is a synthetic class, else false
11      boolean retval = cls.isSynthetic();
12      System.out.println("It is a synthetic class ? " + retval);
13    }
14 }
```

```
1  It is a synthetic class ? false
```

Plan du cours

Création d'une instance à partir d'un objet Class

- La classe **Class** expose une méthode **newInstance()** qui permet de créer une nouvelle instance de cette classe
- Cette méthode invoque le constructeur vide de cette classe, qui doit donc exister
- Permet de créer des instances d'une classe, à partir du nom d'une classe sous forme d'une chaîne de caractères (**String**)

Méthodes disponibles : **newInstance()**

```
1  import java.lang.*;
2  public class ClassDemo_newInstance {
3      public static void main(String[] args) {
4      try {// dans une émthode main
5          String className = "Marin" ;
6          // chargement d'une classe àpartir de son nom
7          // jette une exception du type ClassNotFoundException
8          Class marinClass = Class.forName(className) ;
9          try {// dans une émthode main
10             // instanciation d'un objet àpartir de sa classe
11             // jette deux exceptions : IllegalAccessException, InstantiationException
12             // l'objet o est en fait de type org.paumard.model.Marin
13             Object o = marinClass.newInstance() ;
14             System.out.println(o);
15         }
16         catch(InstantiationException e) {
17             System.out.println(e.toString());
18         }
19         catch(IllegalAccessException e) {
20             System.out.println(e.toString());
21         }
22     }
23     catch(ClassNotFoundException ex) {
24         System.out.println(ex.toString());
25     }
26     }
27 }
```

```
1  Marin [nom=null, prenom=null, salaire=0]
```

Plan du cours

Cas des énumérations

- Dans le cas où la classe que l'on manipule est une énumération, on peut obtenir le tableau des constantes définies dans cette énumération

```
1 public enum Grade {
2    MOUSSE, CRABE, CHOUFFE, BOSCO, PACHA
3 }
```

- Interrogeons maintenant cette classe pour découvrir ses constantes

```
1 public static void main(String[] args) {
2    Grade[] grades = Grade.class.getEnumConstants() ;
3    System.out.println(Arrays.toString(grades)) ;
4 }
```

- L'exécution de ce code affiche le résultat suivant

```
1 [MOUSSE, CRABE, CHOUFFE, BOSCO, PACHA]
```

Plan du cours

La classe **Method**

- La classe **Method** permet de modéliser les méthodes d'une classe ou d'une interface, concrètes ou abstraites
- La façon la plus simple d'obtenir une référence sur une méthode, est d'utiliser une des méthodes **getMethod()** de la classe **Class**
- La classe **Method** fonctionne sur le même principe que la classe **Class**. Elle permet d'obtenir toutes les informations sur cette méthode
  - son modificateur de visibilité
  - son type de retour
  - ses paramètres, et leurs types
  - les exceptions qu'elle jette
  - son nom
  - les annotations posées sur cette méthode, et sur ses paramètres

La classe **Method** : Méthodes disponibles

- Les premières méthodes exposées permettent de lire le nom de la méthode, et son modificateur
  - **getName()** : retourne le nom de cette méthode
  - **getModifiers()** : retourne les modificateurs de cette méthode, sous forme d'un **int**. Cet entier peut être décodé par les méthodes de la classe Modifier, comme nous le verrons dans le paragraphe suivant

Méthodes disponibles : **getModifiers()**

```
1  import java.lang.reflect.*;
2
3  public class ClassDemo_getModifiers {
4
5     public static void main(String[] args) {
6
7       try {
8        Class cls = Class.forName("Marin");
9
10       // returns the Java language modifiers for this class
11       int i = cls.getModifiers();
12       String retval = Modifier.toString(i);
13       System.out.println("Class Modifier = " + retval);
14      }
15      catch (Exception e) {
16       System.out.println("Exception: " + e);
17      }
18    }
19 }
```

```
1  Class Modifier = public
```

La classe **Method** : Méthodes disponibles

- Viennent ensuite les méthodes qui permettent d'obtenir des informations sur le fonctionnement de cette méthode : son type de retour, les types de ses paramètres et les exceptions qu'elle jette
  - **getDeclaringClass()** : retourne l'objet Class qui correspond à la classe qui porte cette méthode
  - **getExceptionTypes()** : retourne le tableau des classes d'exception jetées par cette méthode. Si cette méthode ne jette pas d'exception, alors ce tableau est vide
  - **getReturnType()** : retourne la classe du type retourné. Si ce type est un type primitif **Java (int, float, etc...)**, alors cette classe est une classe de type primitif **(int.class, float.class, etc...)**. Si ce type est **void**, alors la classe est **void.class**
  - **getParameterTypes()** : retourne le tableau des types des paramètres définis par cette méthode. L'ordre des paramètres est bien sûr conservé. On peut accéder aux annotations sur ces paramètres par la méthode getParameterAnnotations().

Méthodes disponibles

```
1  import java.lang.reflect.*;
2  public class ClassDemo_getDeclaringClass {
3     public static void main(String[] args) {
4       try {
5          Class cls = Class.forName("Marin");
6          Method[] m = cls.getMethods();
7          for (int i = 0; i < m.length; i++) {
8              // returns te declaring class
9              Class dec = m[i].getDeclaringClass();
10             Class ret = m[i].getReturnType();
11             Class[] exept = m[i].getExceptionTypes();
12             Class[] params = m[i].getParameterTypes();
13
14             // displays all methods
15             System.out.println("Method = " + m[i].toString());
16             System.out.println(" --> Declaring class: " + dec.toString());
17             System.out.println(" --> Returned Type: " + ret.toString());
18             System.out.print(" --> Parameters Type: ") ;
19             for(int j = 0; j < params.length; j++)
20                 System.out.print(" --> param " + j + ": "+ params[j].toString() + "; ");
21             System.out.println();
22             System.out.print(" --> Exceptions Type: ") ;
23             for(int j = 0; j < exept.length; j++)
24                 System.out.print(" --> exception " + j + ": "+ exept[j].toString() + "; ");
25             System.out.println();
26         }
27     }
28     catch (Exception e)
29         System.out.println("Exception: " + e);
30 }
31 }
```

Méthodes disponibles

```
1  Method = public java.lang.String Marin.toString()
2   --> Declaring class: class Marin
3   --> Returned Type: class java.lang.String
4   --> Parameters Type:
5   --> Exceptions Type:
6  Method = public void Marin.augmenteSalaire(int)
7   --> Declaring class: class Marin
8   --> Returned Type: void
9   --> Parameters Type: --> param 0: int;
10  --> Exceptions Type:
11 Method = public int Marin.getSalaire()
12  --> Declaring class: class Marin
13  --> Returned Type: int
14  --> Parameters Type:
15  --> Exceptions Type:
16 Method = public void Marin.SetSalaire(int)
17  --> Declaring class: class Marin
18  --> Returned Type: void
19  --> Parameters Type: --> param 0: int;
20  --> Exceptions Type:
21 Method = public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
22  --> Declaring class: class java.lang.Object
23  --> Returned Type: void
24  --> Parameters Type: --> param 0: long; --> param 1: int;
25  --> Exceptions Type: --> exception 0: class java.lang.InterruptedException;
26 Method = public final native void java.lang.Object.wait(long) throws java.lang.
       InterruptedException
27  --> Declaring class: class java.lang.Object
28  --> Returned Type: void
29  --> Parameters Type: --> param 0: long;
30  --> Exceptions Type: --> exception 0: class java.lang.InterruptedException;
```

Méthodes disponibles

```
1  Method = public final void java.lang.Object.wait() throws java.lang.InterruptedException
2   --> Declaring class: class java.lang.Object
3   --> Returned Type: void
4   --> Parameters Type:
5   --> Exceptions Type: --> exception 0: class java.lang.InterruptedException;
6  Method = public boolean java.lang.Object.equals(java.lang.Object)
7   --> Declaring class: class java.lang.Object
8   --> Returned Type: boolean
9   --> Parameters Type: --> param 0: class java.lang.Object;
10  --> Exceptions Type:
11 Method = public native int java.lang.Object.hashCode()
12  --> Declaring class: class java.lang.Object
13  --> Returned Type: int
14  --> Parameters Type:
15  --> Exceptions Type:
16 Method = public final native java.lang.Class java.lang.Object.getClass()
17  --> Declaring class: class java.lang.Object
18  --> Returned Type: class java.lang.Class
19  --> Parameters Type:
20  --> Exceptions Type:
21 Method = public final native void java.lang.Object.notify()
22  --> Declaring class: class java.lang.Object
23  --> Returned Type: void
24  --> Parameters Type:
25  --> Exceptions Type:
26 Method = public final native void java.lang.Object.notifyAll()
27  --> Declaring class: class java.lang.Object
28  --> Returned Type: void
29  --> Parameters Type:
30  --> Exceptions Type:
```

Invocation d'une méthode par introspection

- Il est possible d'invoquer une méthode par introspection, sur un objet donné, avec des paramètres
- Très utilisée, dans les frameworks web et ailleurs
- Elle repose sur l'utilisation de la méthode **invoke()**

Exemple

```
1  public class Marin {
2
3      private String nom ;
4
5      public Marin () {
6          System.out.println("Invocation du constructeur vide") ;
7      }
8
9      public Marin (String nouveauNom, String nouveauPrenom, int nouveauSalaire) {
10         nom = nouveauNom ;
11         prenom = nouveauPrenom ;
12         salaire = nouveauSalaire ;
13     }
14
15     public void setNom(String nom) {
16         System.out.println("Invocation de setNom()") ;
17         this.nom = nom ;
18     }
19
20     public String getNom() {
21         System.out.println("Invocation de setNom()") ;
22         return this.nom ;
23    }
24
25     public String toString() {
26     return "Marin [nom=" + nom + ", prenom=" + prenom + ", salaire="
27         + salaire + "]";
28     }
29 }
```

Exemple

```
1  import java.lang.reflect.*;
2  public class ClassDemo_method {
3     public static void main(String[] args) {
4       try {
5          String className = "Marin" ;
6          String propertyName = "nom" ;
7          String value = "Barberousse" ;
8          Class cls = Class.forName(className) ;
9          try {
10             // invoque le constructeur vide, qui doit exister
11             Object o = cls.newInstance() ; // instanciation de cette classe
12             // construction du nom du setter, on utilise la definition classique d'un setter
13             StringBuilder sb = new StringBuilder() ;
14             sb.append("set").append(propertyName.substring(0, 1).toUpperCase()).append(propertyName
                     .substring(1)) ;
15             String setterName = sb.toString() ;
16             // interrogation de la classe pour recuperer la bonne methode
17             Method setter = cls.getMethod(setterName, value.getClass()) ;
18             // invocation de la methode
19             setter.invoke(o, value) ;
20             // affichage de l'objet, invocation de sa methode toString()
21             System.out.println(o) ;
```

Exemple

```
1
2           // construction du nom du getter
3           sb = new StringBuilder() ;
4           sb.append("get").append(propertyName.substring(0, 1).toUpperCase()).append(propertyName
                .substring(1)) ;
5           String getterName = sb.toString() ;
6           // interrogation de la classe pour recuperer la bonne methode
7           Method getter = cls.getMethod(getterName) ;
8           // invocation de la methode, recuperation de l'objet retourne
9           Object returnedValue = getter.invoke(o) ;
10          // affichage de la valeur éretourne
11          System.out.println(returnedValue) ;
12      }
13      catch (IllegalAccessException e) {
14          System.out.println("Exception: " + e);
15      }
16      catch (InstantiationException e) {
17          System.out.println("Exception: " + e);
18      }
19  }
20  catch (ClassNotFoundException e) {
21      System.out.println("Exception: " + e);
22  }
23  }
24  }
```

```
1  Invocation du constructeur vide
2  Invocation de setNom()
3  Marin [nom=Barberousse, prenom=null, salaire=0]
4  Invocation de setNom()
5  Barberousse
```

Plan du cours

Méthodes disponibles : Les champs (**Field**)

- La classe **Field** modélise les champs d'une classe
- La façon la plus simple d'obtenir un objet de type **Field** est d'utiliser l'une des méthodes getField() de la classe **Class**
- On peut obtenir ainsi une référence sur tous les champs d'une classe, qu'il soit statique ou non, privé, protégé ou public
- Elle permet d'obtenir les informations sur le champ d'une classe
  - son modificateur de visibilité
  - son type
  - son nom
  - les annotations qu'il porte
- Cette classe permet également de fixer la valeur de ce champ directement, ou de la lire

Méthodes disponibles : Les champs (Field)

- Les méthodes exposées sont plus simples que celles de la classe Method
  - **getName()** : retourne le nom de ce champ
  - **getModifiers()** : retourne un entier qui code les modificateurs de ce champ sous forme d'un **int**. Cet entier peut être décodé grâce à la classe utilitaire Modifier
  - **getInt(Object), getFloat(Object), get(Object)**, etc... : retournent la valeur de ce champ sous la forme indiquée par le nom de la méthode, pour l'objet passé en paramètre
  - **setInt(Object, int), setFloat(Object, float), setObject(Object, Object)** etc... : permettent de fixer la valeur de ce champ sous la forme indiquée par le nom de la méthode, pour l'objet passé en paramètre

Exemple

```
1  import java.lang.reflect.*;
2
3  public class ClassDemo_field {
4
5     public static void main(String[] args) {
6
7       try {
8          // dans une methode main
9          // definition de la classe a utiliser
10         String className = "Marin" ;
11         // definition de la propriete utilisee
12         String propertyName = "nom" ;
13         // valeur de cette propriete
14         String value = "Barberousse" ;
15         // jette une ClassNotFoundException
16         // on utilise cls car une variable ne peut pas s'appeler class
17         // creation de l'objet Class àpartir du nom complet de cette classe
18         Class cls = Class.forName(className) ;
19         try {
20             // jette IllegalAccessException et InstantiationException
21             // instanciation de cette classe
22             // invoque le constructeur vide, qui doit exister
23             Object o = cls.newInstance() ;
24             // jette NoSuchFieldException
25             // attention : getField() ne retourne que les champs publics, ici notre
26             // champ est prive, donc il faut utiliser cls.getDeclaredField()
27             Field champNom = cls.getDeclaredField(propertyName) ;
```

Exemple

```
1            // lecture des modificateurs de ce champ
2            int mod = champNom.getModifiers() ;
3            // on verifie que notre champ est bien prive
4            System.out.println("Champ [" + propertyName + "] prive : " + Modifier.isPrivate(mod)) ;
5            // on le rend accessible, pour pouvoir le modifier
6            boolean isAccessible = champNom.isAccessible() ;
7            if (!isAccessible) {
8                champNom.setAccessible(true) ;
9            }
10           // on positionne sa valeur
11           champNom.set(o, value) ;
12           // on remet la propriete accessible a sa valeur
13           // precedente
14           if (!isAccessible) {
15               champNom.setAccessible(false) ;
16           }
17           // on affiche notre marin
18           System.out.println(o) ;
19       }
20       catch (IllegalAccessException e) {
21           System.out.println("Exception: " + e);
22       }
23       catch (InstantiationException e) {
24           System.out.println("Exception: " + e);
25       }
26   }
27   catch (Exception e) {
28       System.out.println("Exception: " + e);
29   }
30 }
31 }
```

Exemple

```
1 Invocation du constructeur vide
2 Champ [nom] prive : true
3 Marin [nom=Barberousse, prenom=null, salaire=0]
```

Plan du cours

Exemple

- Ajout de Java 8 $\Rightarrow$ Réécriture complete de l'ensemble du JDK