Programmation Objet Avancée

Kevin Heraud

Cours 4 : collections, types énumérés et fichiers

Département Informatique
Université de Tours

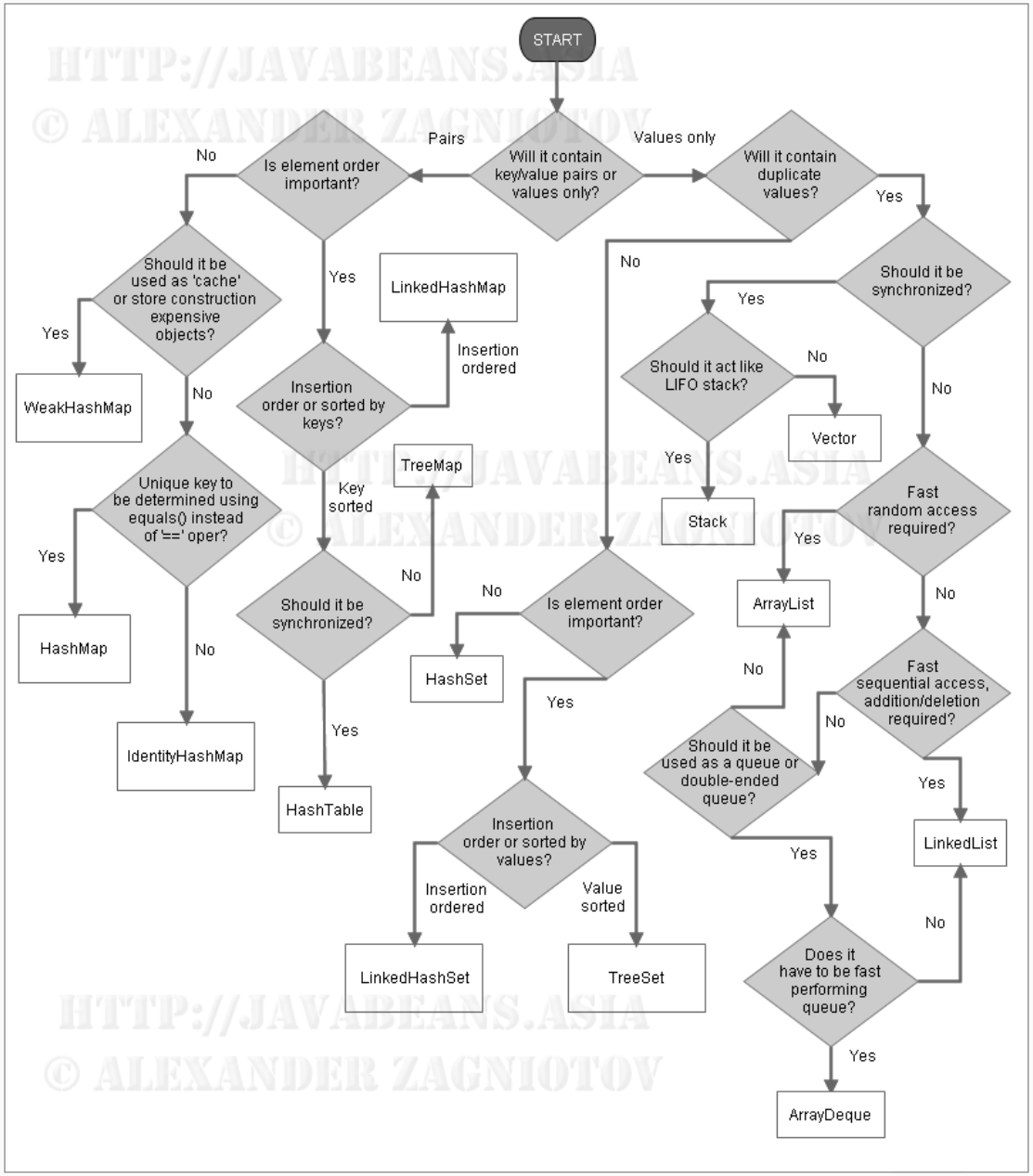
Les collections

- les **collections** en Java permettent de manipuler des structures de données génériques telles que
  - les **vecteurs dynamiques** `ArrayList`
  - les listes chaînées `LinkedList`
  - les ensembles `hashSet` et `TreeSet`
  - les queues avec priorité `PriorityQueue`
  - les queues avec double entrée `ArrayDeque`
- il existe d'autres types pour les structures de données
  - les tableaux `int[]` ...
  - les **tables associatives** `HashMap` issue de l'interface `Map`

Critères de choix d'une collection

- critères de choix
  - éléments ordonnées ou non
  - l'accès direct a un élément est possible ou pas
  - l'accès à la valeur se fait en connaissant une clé correspondante
  - les doublons sont-ils possibles ?
- dans ce cours, focus sur les `ArrayList` et les `HashMap`

Critères de choix d'une collection



Principales propriétés des collections

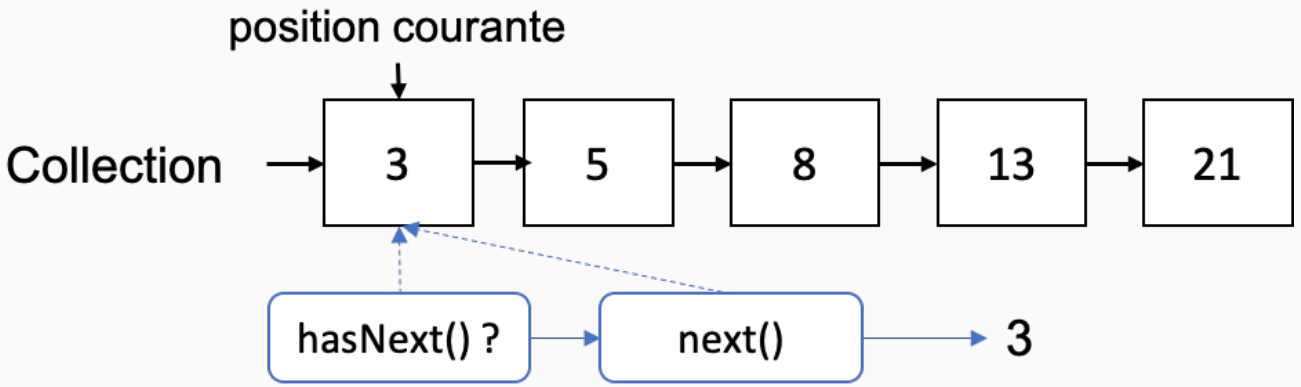
- contenues dans le package `java.util`
- depuis le JDK5, les collections sont **génériques** : `Collection<E>`
- certaines collections sont **ordonnées** : `ArrayList`, `LinkedList` ...
  - possibilité d'accéder au 1er, 2nd, i-ème élément
  - possibilité de les trier selon les valeurs
  - les éléments `E` de la collection implémentent l'interface `Comparable<E>` et recourent à la méthode `compareTo()`

Parcours des éléments d'une collection

- un **itérateur** permet de parcourir un par un les éléments d'une collection
- deux types d'itérateurs
  1. les itérateurs **monodirectionnels** parcourent la collection du début à la fin
  2. les itérateurs **bidirectionnels** parcourent la collection dans les deux sens

Itérateurs monodirectionnels

- accessibles par la méthode `iterator()` qui implémente l'interface `Iterator<E>`
- un itérateur indique une position courante qui désigne un élément de la collection
- la méthode `hasNext()` permet de savoir s'il y a un élément accessible dans la collection
- pour se placer sur et récupérer l'élément courant, il faut appeler la méthode `next()`



Utiliser un itérateur monodirectionnel

- exemple de parcours

```
1 Iterator<E> iter = c.iterator();
2 while (iter.hasNext()){
3  E elem = iter.next();
4  // utilisation de elem ...
5 }
```

Utiliser un itérateur monodirectionnel

- exemple de suppression du dernier élément retourné

```
1  Iterator<E> iter = c.iterator();
2  while (iter.hasNext()){
3   E elem = iter.next(); // avance d'une position et retourne
4                         // l'element courant
5   if (condition) elem.remove();
6  }
```

Supprimer le 1er élément d'une collection

1. récupérer l'itérateur de la collection
2. récupérer l'objet courant et se positionner sur la position suivante
3. supprimer l'objet courant

```
1 Iterator<E> iter = c.iterator();
2 E elem = iter.next(); // recuperation du 1er objet
3 iter.remove(); // suppression du 1er objet
```

- `remove` ne supprime pas l'élément à la position courante mais l'objet courant qui a été accédé précédemment
- il est nécessaire d'avoir un objet courant pour appeler la méthode `remove`

Parcours unidirectionnel avec for ... each

- il est aussi possible de parcourir une collection avec une boucle `for ... each`

```
1 for (E elem : c){
2 }
```

Les vecteurs dynamiques `ArrayList`

- structure ordonnée, à accès direct par indice, qui autorise les doublons
- comparable à un tableau :
  - offre un accès rapide aux éléments
  - les éléments sont contigus en mémoire
- mais le nombre d'éléments peut varier au cours de l'exécution
- accepte n'importe quels types d'objets, y compris la valeur `null`
- dérive de l'interface `List<E>`

Opérations usuelles : ajout

- **construction** d'un vecteur vide ou bien à partir d'une autre collection existante `c`

```
1 List<E> v1 = new ArrayList<E>();
2 List<E> v2 = new ArrayList<E>(c);
```

- **ajout** d'un ou de plusieurs éléments, à une certaine position

```
1 v1.add(e1); // ajout de e1 a la fin
2 v1.add(3, e2); // ajout de e2 en 4eme position
3 v2.addAll(c); // ajout de tous les elements de c a la fin
4 v2.addAll(2, c);; // ajout des elements de c a l'indice 2
```

Opérations usuelles : suppression

- **suppression** d'un ou plusieurs éléments avec décalage de l'indice de tous les éléments suivants

```
1 E elem = v1.remove(3); // ici elem = e2
2 v2.removeRange(1, 4);
```

Opérations usuelles : parcours et accesseurs

- **taille** du vecteur accessible par la méthode `size()`
- **accès** aux valeurs du vecteur avec la méthode

  `get(int index)`

Autres méthodes de la classe `ArrayList`

- `isEmpty()` : renvoie vrai si la liste est vide
- `indexOf(objet)` : retourne l'indice i de l'objet en paramètre ou -1 s'i l'objet n'est pas dans la liste
- `contains(objet)` : retourne vrai si l'objet est dans la liste
- `set(i, objet)` : remplace l'élément situé en i par l'objet en paramètre
- `clear()` : supprime tous les objets de la liste
- `isEmpty()` : renvoie vrai si la liste est vide
- `contains(objet)` retourne vrai si l'objet est dans la liste

Exemple d'utilisation

```
1  List<Point> points = new ArrayList<Point>();
2  points.add(p1);
3  points.add(p2);
4  points.add(p3);
5
6  // ou
7  List<Point> points = Arrays.asList(p1, p2, p3);
8
9  for (int i=0;i<points.size();i++) points.get(i).affiche();
```

Autre notation utilisant un `for ... each`

```
1  for (Point point : points) point.affiche();
```

Vecteurs dynamiques et héritage

- on suppose avoir les classes `Cercle`, `Triangle` et `Parallelogramme` qui héritent de la classe abstraite `Forme`
- la classe `Forme` définit la méthode abstraite `affiche`

```
1  Cercle c1 = new Cercle(p1,12.6); // centre p1, rayon 12.6
2  Triangle t1 = new Triangle(p2,v1,v2); // 1 point et 2 vecteurs
3  Parallelogramme pa1 = new Parallelogramme(p3,v1,v3);
4
5  List<Forme> formes = new ArrayList<Forme>();
6  formes.add(pa1);  // un parallelogramme
7  formes.add(c1); // un cercle
8  formes.add(t1);  // un triangle
9
10 // ou
11 List<String> formes = Arrays.asList(c1, t1, pa1);
12
13 for (Forme forme: formes) forme.affiche();
```

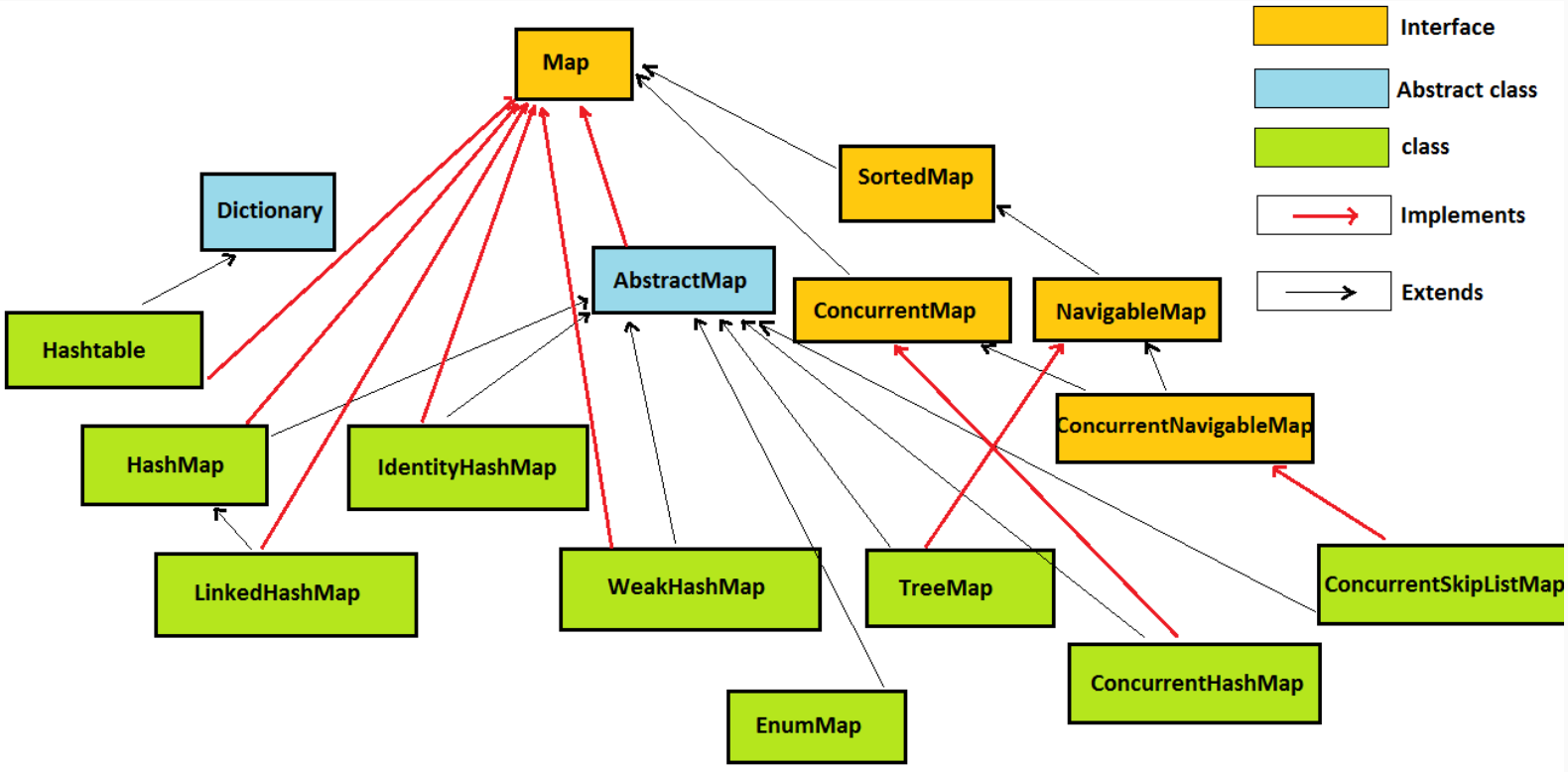
ArrayList personnalisée

```
1  public class EnvelopeArrayList<Page> extends ArrayList<Page> {
2      private final int defaultMaxPages;
3
4      public EnvelopeArrayList(int defaultMaxPages) {
5          this.defaultMaxPages = defaultMaxPages;
6      }
7
8      @Override
9      public boolean add(Page page) {
10         return super.size() < defaultMaxPages ?
11             super.add(page) : false;
12     }
13 }
14
15 List<Page> pages = new EnvelopeArrayList(5)<>;
```

Map

- `Map` est une interface représentant une table associative (<K,V>, "Key"/"Value")

Map : la hiérarchie



HashMap

- `HashMap<K,V>` est une classe implémentant l'interface `Map<K,V>`
- les clés ne peuvent pas être dupliquées
- une clé peut être `null`
- les valeurs `null` sont acceptées
- l'ordre des clés n'est pas maintenu
- fonctionne sur la base de la technique de hachage (bucketIndex = key.hashCode() % bucketSize)

HashMap : quelques méthodes

- `put(K, V)` : Insère l'entrée (K, V). Si la clé est déjà présente, la nouvelle valeur remplace l'ancienne
- `putIfAbsent(K, V)` : Insère l'entrée (K, V) si la clé K n'est pas déjà associée à la valeur V
- `get(K)` : Retourne la valeur associée à la clé K spécifiée. Si la clé n'est pas trouvée, elle retourne null
- `containsKey(K)` : Vérifie si la clé spécifiée K est présente dans le Map ou non
- `containsValue(V)` : Vérifie si la valeur spécifiée V est présente dans le Map ou non
- `remove(K)` : Supprime l'entrée du Map représentée par la clé K
- `keySet()` : Retourne l'ensemble de toutes les clés présentes dans une Map
- `values()` : Retourne un ensemble de toutes les valeurs

HashMap

```
1  Map<String, String> phonebook = new HashMap<>();
2
3  phonebook.put("01000005", "Tom");
4  phonebook.put("01000006", "Jerry");
5  phonebook.put("01000003", "Tom");
6  phonebook.put("01000004", "Donald");
7
8  // ou
9  Map<String, String> phonebook = Map.of("01000004", "Donald");
10
11 for (Map.Entry entry : phonebook.entrySet()) {
12 System.out.println(entry.getKey() + "|" + entry.getValue());
13 }
```

Les types énumérés

- le mot-clé `enum` permet de déclarer un type énuméré
- type de données comportant en ensemble fini de constantes
- les constantes seront séparées par une virgule et en MAJUSCULES (en SNAKE_CASE)
- utilisées lorsque nous connaissons toutes les valeurs possibles au moment de la compilation
- Très utile dans les `switch`

Les types énumérés : les méthodes associées

- une énumération peux contenir un constructeur privé
- `values()` : renvoie toutes les valeurs présentes dans l'énumération
- `valueOf()` : renvoie la constante d'énumération de la valeur de chaîne spécifiée, si elle existe

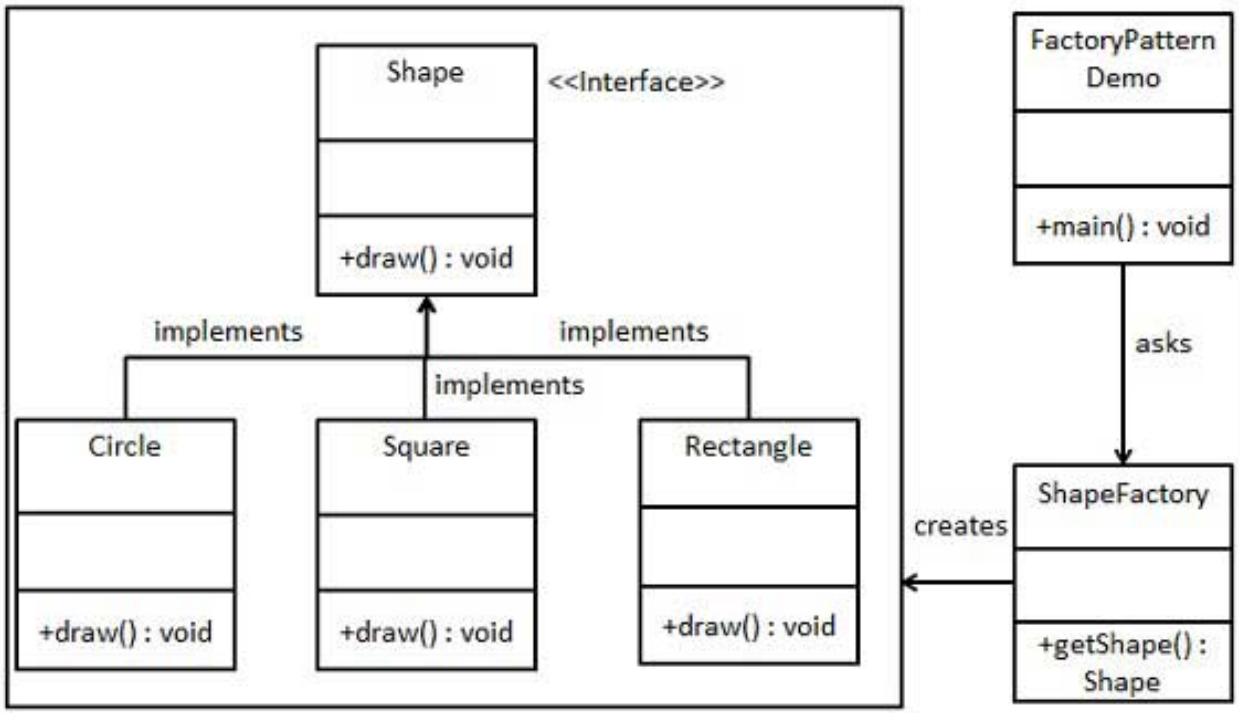
Les types énumérés : exemple

```
1  public enum CurrencyEnum {
2  EUR("euro", "E", 978), USD("dollar", "S", 840);
3
4  private final String displayName;
5  private final String symbol;
6  private final int code;
7
8  CurrencyEnum(String displayName, String symbol, int code) { }
9
10 public String getDisplayName() { return displayName; }
11 public String getSymbol() { return symbol; }
12 public int getNumericCode() { return numericCode; }
13 }
14
15 public static void main(String[] args) {
16     CurrencyEnum eur = CurrencyEnum.EUR;
17     System.out.println(eur.getDisplayName());
18 }
```

Les types énumérés : le pattern "Singleton"

```
1 public enum SingletonEnum {
2   INSTANCE;
3 }
4
5 public static void main(String[] args) {
6     SingletonEnum singleton = SingletonEnum.INSTANCE;
7 }
```

Les types énumérés : le pattern "Factory"



Gestion des fichiers en Java

- Java distingue les fichiers **binaires** des fichiers **textes**
- Java permet des accès **séquentiels** (lire les n premiers octets pour accéder au n+1 ème) ou **directs**
- Java permet soit d'accéder en **lecture**, soit en **écriture** (aussi utilisé pour créer un fichier)

Création d'un fichier binaire

- utilisation de la classe `FileOutputStream` qui gère les **flux binaires de sortie sur des fichiers**
- `FileOutputStream` hérite de `OutputStream` qui sert de base à tous les **flux binaires** de sortie
- les méthodes de `FileOutputStream` sont rudimentaires, il est préférable d'utiliser `DataOutputStream` (permet l'écriture de données de type primitif Java, et de chaînes de caractères)

```
1 FileOutputStream fop = new FileOutputStream("test.dat");
2 DataOutputStream dos = new DataOutputStream(fop);
3 dos.writeInt(42); // throws IOException
4 dos.close(); // fermeture du flux, impossible d'y acceder
```

Utilisation d'une mémoire tampon

- il est possible d'utiliser un **buffer** ou mémoire tampon pour optimiser les échanges avec le flux
- on utilise pour cela la classe **BufferedOutputStream**

```
1 FileOutputStream f = new FileOutputStream("test.dat");
2 BufferedOutputStream buf = new BufferedOutputStream(f);
3 DataOutputStream sortie = new DataOutputStream(buf);
4 sortie.close();
```

- l'appel à la méthode `close` vide le tampon dans le flux de sortie

Accès séquentiel à un fichier binaire

- similairement à l'écriture, la lecture dispose de la classe `FileInputStream`
- qui hérite de la classe abstraite `InputStream` qui gère tout type de flux binaire en entrée
- il est préférable d'utiliser la classe `DataInputStream` qui ajoute des méthodes plus pratiques pour la lecture de données issues d'un fichier binaire

```
1 FileInputStream f = new FileInputStream("test.dat");
2 BufferedInputStream buf = new BufferedInputStream(f);
3 DataInputStream entree = new DataInputStream(buf);
4
5 int valeur = entree.readInt(); // valeur = 42
6 entree.close(); // fermeture du flux, impossible d'y acceder
```

Accès direct à un fichier binaire

- la classe `RandomAccessFile` permet d'accéder en lecture **et** en écriture à un fichier binaire
- contient les méthodes de `DataInputStream` et `DataOutputStream`
  - `readInt()`, `readFloat()`, `writeInt(int)`, `writeFloat(float)`
- ajoute une méthode `seek()` pour placer le pointeur de lecture / écriture à l'endroit désiré du fichier
  - le premier octet est placé à la position 0
  - attention : `seek()` travaille sur un nombre d'octets
  - il faut prendre en compte la taille de chaque type de données lors du calcul du décalage dans le fichier

Écriture dans un flux texte

- la classe abstraite `Writer` sert de base aux flux de sortie texte
- la classe dérivée `FileWriter` permet de créer ou d'écrire dans un fichier texte
- pour faciliter le **formatage automatique** des contenus (`int`, `float`) vers du texte on pourra utiliser `PrintWriter`
  - comportement similaire à `System.out.println();`

```
1 public static void exText1() throws IOException {
2  FileWriter fw = new FileWriter("test.dat");
3  PrintWriter out = new PrintWriter(fw);
4  int n = 42;
5  out.println(n + " a pour carre " + (n * n));
6  out.close();
7 }
```

Utilisation d'un buffer

- comme pour les fichiers binaires, il est possible d'utiliser une mémoire tampon intermédiaire nommée **buffer**

```
1 public static void exText2() throws IOException {
2  PrintWriter out = new PrintWriter(
3    new BufferedWriter(  // add a buffer!
4     new FileWriter("test.dat")));
5  int n = 42;
6  out.println(n + " a pour carre " + (n * n));
7  out.close();
8 }
```

Lecture d'un fichier texte

- pas d'équivalent à `PrintWriter` qui permettrait de lire les valeurs contenues dans une ligne et de les interpréter
- deux étapes
  1. lire une ligne complète du fichier
  2. analyser et interpréter la ligne
- on utilise la classe `FileReader` qui est équivalente à `FileWriter`
- il est toujours possible d'utiliser un buffer avec `BufferedReader` qui ajoute la méthode de lecture d'une ligne `readLine()`

Exemple de lecture d'un fichier texte

```
1 public static void exText3() throws IOException {
2  FileReader fr = new FileReader("test.dat");
3  BufferedReader br = new BufferedReader(fr);
4  String ligne;
5  while ((ligne = br.readLine()) != null) {
6   System.out.println(ligne);
7  }
8  br.close();
9 }
```

La classe `File`

- la classe `File` représente
  - soit un nom de fichier, avec ou sans chemin
  - soit un nom de répertoire

- le chemin fourni peut-être
  - absolu : spécifié intégralement depuis la racine du système :
    - `/` : sous Unix (Linux, Mac OS)
    - `C:` sous Windows
  - relatif : spécifié à partir du répertoire courant

- attention aux écritures de chemin qui diffèrent selon les systèmes
  - possibilité d'utiliser `File.separator` qui fournit le séparateur en vigueur dans le système d'exploitation

Exemple de notations de chemins

- sous Windows, **en absolu**

```
1 File fwin = new File("C:/ens/l1/algo/test.java");
```

- sous Unix (Linux, MacOS), **en absolu**

```
1 File funx = new File("/ens/l1/algo/test.java");
```

- avec séparateur déterminé par Java **en relatif**

```
1 String sep = File.separator;
2 File fboth = new File("ens"+sep+"l1"+sep+"algo"+
3                       sep+"test.java");
```

- avec les notations `.` et `..` **en relatif**

```
1 File fboth2 = new File("/ens/l1/algo/2018/../test.java");
```

Fonctions avancées de `File`

- on peut utiliser `File` au sein des constructeurs de classes de gestion de flux comme `DataInputStream` ou `DataOutputStream`

```
1 File f = new File("test.dat");
2 DataOutputStream dos = new DataOutputStream (
3   new FileOutputStream(f));
```

- il y a de nombreuses autre fonctionnalités pratiques

```
1 System.out.println("Le fichier est " + f);
2 System.out.println("Existe ? " + f.exists());
3 System.out.println("Repertoire ? " + f.isDirectory());
4 System.out.println("Taille ? " + f.length());
5 System.out.println("Parent ? " + f.getParentFile());
6 System.out.println("Lisible ? " + f.canRead());
7 System.out.println("En ecriture ? " + f.canWrite());
8 System.out.println("Executable ? " + f.canExecute());
```

Package `NIO.2` et classe `Path`

- le package `java.nio.file` contient les classes `Path` ainsi que les classes `Paths` et `Files`

```
1 import java.nio.file.Path;
2 import java.nio.file.Paths;
3 import java.nio.file.Files;
```

- la classe `Path`
  - améliore la gestion des erreurs de la classe `File`
  - permet la prise en compte de métadonnées (date de création)
  - gère directement le séparateur du système d'exploitation

Package NIO.2 et classe Path

```
1 // chemin = "test/test.dat" (Unix) ou "test\\test.dat" (Win)
2 Path p = Paths.get("test", "test.dat");
3 // utilisation d'une fabrique pas d'un constructeur
4 boolean b = Files.exists(p);
5 System.out.println(b);
```

Comparaison entre `Path` et `File`

- il est possible de récupérer un objet `File` à partir de `Path` et inversement

```
1 Path p = Paths.get("test.dat");
2 File f = p.toFile();
3 Path p2 = f.toPath();
```

- le séparateur est désormais accessible en utilisant la nouvelle notion de “système de fichiers”

```
1 String sep = FileSystem.getDefault().getFileSeparator();
```

Principales méthodes de `Files`

- les méthodes statiques de la classe `Files` permettent d'accéder aux propriétés d'un objet `Path`

```
1  Path p = Paths.get("ens", "l1", "algo", "test.java");
2  System.out.println(p.toAbsolutePath());
3
4  System.out.println("Le fichier est " + p); // toString() auto
5  System.out.println("Existe ? " + Files.exists(p));
6  System.out.println("Repertoire ? " + Files.isDirectory(p));
7  System.out.println("Taille " + Files.size(p));
8  System.out.println("Parent ? " + p.getParent());
9  System.out.println("Lisible ? " + Files.isReadable(p));
10 System.out.println("En ecriture ? " + Files.isWritable(p));
11 System.out.println("Executable ? " + Files.isExecutable(p));
```

Parcours d'un répertoire

- utiliser un flux de dossiers et répertoires de type `DirectoryStream<Path>`

```
1 Path source = Paths.get("/Users/toto/Desktop/");
2 DirectoryStream<Path> liste = Files.newDirectoryStream(source)
3 for (Path p : liste) {
4  System.out.println(p + "\t" + Files.isDirectory(p));
5 }
```

- utiliser un `FileVisitor` qui consiste à implémenter une classe dont les méthodes sont appelées par `Files.walkFileTree`
  - gestion événementielle du parcours des fichiers / répertoires : on appelle une méthode en fonction de ce que l'on observe
  - possibilité de parcourir des arborescences à plusieurs niveaux

Les différents formats textes

- **CSV** *(Comma-Separated Values)*
- **JSON** *(JavaScript Object Notation)*
- **YAML** *(Yet Another Markup Language)*
- **XML** *(Extensible Markup Language)*
- **JSONL**, **TOML**, **HCL**, **INI**, ...

Les différents formats textes

XML	JSON	YAML

<Servers>
<Server>
<name>Server1</name>
<owner>John</owner>
<created>123456</created>
<status>active</status>
</Server>
</Servers>
	{
	Servers: [
	{
	name: Server1,
	owner: John,
	created: 123456,
	status: active
	}
	]
	}
		Servers:
		- name: Server1
		owner: John
		created: 123456
		status: active