Informatique 1

Introduction à la programmation orientée objet

SUPMICROTECH
Semestre bleu
INFO 1
2024

Les objets

Compétences

- Connaître les concepts et la syntaxe de base de Java (instructions élémentaires, structures de contrôle) ;
- Savoir utiliser des objets simples comme les chaînes de caractères ou les complexes.

Le but de cette leçon est de se familiariser avec les concepts de base de la programmation objet et leur mise en œuvre en langage Java.

I Variables et types primitifs

Une variable est une cellule mémoire identifiée par un nom à laquelle est associée un contenu (valeur de la variable). Le contenu d'une variable peut évoluer au cours du temps. En Java, les variables nécessaires à un algorithme sont dites typées et doivent toujours être déclarées.

Exemple

```
int         unEntier ;
double      unReel, surface ;
boolean     finDeTraitement ;
```

Un type définit l'ensemble des valeurs que peut prendre une variable et auquel sont associées les opérations et fonctions qui peuvent lui être appliquées.
En Java, les types primitifs permettent de manipuler des données élémentaires. Comme dans la plupart des langages, on retrouve :

- Les entiers (`int`) : nombres relatifs bornés (sous-ensemble de $\mathbb{Z}$)
- Les réels (`double`) : nombres à virgule flottante double précision (sous-ensemble de $\mathbb{R}$)
- Les booléens (`boolean`) : deux états (`true`, `false`)
- Les caractères (`char`) : 'a', 'b', …, 'z', 'A', …, 'Z', '0', '+', '@', …

Les opérateurs mathématiques usuels sont définis sur les types de base et permettent de réaliser des calculs.

```
int x,y,z;

x = 5;
y = 13 + (56 * 3);
z = x + y;
```

II Classes et objets

Une classe est l'extension de la notion de type. Il s'agit d'une structure qui permet de modéliser des objets décrits à partir d'un ensemble d'attributs et de lister les opérations disponibles sur ces objets.

Par exemple, si une application doit manipuler des nombres complexes, la représentation informatique d'un complexe nécessite la connaissance de deux éléments que sont la valeur de la partie réelle et la valeur de la partie imaginaire, tout deux de type réel. On définit alors une classe `Complexe` qui regroupe pour chaque objet à manipuler les deux informations ainsi que l'ensemble des opérations réalisables (somme, produit, module, etc.).

Définition Une classe est décrite par un ensemble d'attributs et par un ensemble de méthodes. Les attributs peuvent être des types primitifs ou des classes. Les méthodes sont les opérations qui permettent de manipuler les objets de la classe.

Les classes définissent des concepts décrivant des propriétés et comportements. Pour programmer, on utilise des variables correspondantes à ces classes que l'on appelle objets.

Définition Un objet est une réalisation concrète d'une classe. On dit qu'un objet est une *instance* d'une classe. La valeur de ses attributs définit son *état*.

Avant de pouvoir utiliser un objet en Java, il faut d'abord déclarer une référence puis construire l'objet.

1 Déclaration

La déclaration d'un objet nécessite de préciser un nom de référence (identificateur) et la classe à laquelle il appartient.

```
String      uneChaineDeCaracteres;
Complexe    unComplexe;
```

2 Construction

Pour pouvoir utiliser un objet déclaré, il est indispensable de le construire. Lors de la création de l'objet le constructeur correspondant est automatiquement invoqué.

```
// construit la chaine "Bonjour"
uneChaineDeCaracteres = new String("Bonjour");
// construit le complexe 1.0 + 2.0i
unComplexe = new Complexe(1.0, 2.0);
```

Une fois l'objet construit, il est utilisable. Par exemple, l'utilisation de la classe `String` peut se faire de la façon suivante.

3 Utilisation

```
String uneChaine;
uneChaine = new String("Bonjour");
uneChaine = uneChaine + "␣Antoine";

String question = new String(",␣comment␣vas-tu␣?");
uneChaine = uneChaine + question;

String uneChaineEnMajuscule = uneChaine.toUpperCase();

if (uneChaine.equals("bonjour␣antoine,␣comment␣vas-tu␣?")) {
        ...
}
```

III Structures de contrôle

En Java, les instructions s'exécutent en séquence dans l'ordre de leur écriture (de gauche à droite et de haut en bas). Toutes les structures de contrôle usuelles sont définies. Nous donnons ici pour mémoire quelques exemples d'utilisation des principales structures de contrôles. Les autres sont détaillées dans les diapositives du cours.

1 Structure de choix *Si-alors-sinon*

Une structure de choix permet d'exécuter l'un ou l'autre des blocs d'instructions en fonction du résultat d'un test booléen. Si la valeur de la condition est *vraie* alors la première séquence est exécutée. Si la valeur de la condition est *fausse* alors la seconde séquence est exécutée. Après quoi, l'exécution reprend à la première instruction qui suit le deuxième bloc.

```
if (n == 10) {
    n = 0;
} else {
    n = n + 1;
}
```

Remarque

- La condition peut être une expression booléen complexe.
- Le deuxième bloc d'instructions est optionnel.

```
if (a >= 1 && b != 0) {
    c = a * 3 / b;
    a = 0;
}
```

Remarque

Attention aux pièges des opérateurs = en Java : l'affectation s'écrit `a=b;`, le test d'égalité s'écrit `a==b` pour les types primitifs et `a.equals(b)` pour les classes.

2 Structure d'itération *Tant que*

Une structure d'itération *Tant que* permet de répéter une séquence tant que la valeur de la condition est *vrai*. La condition n'est testée qu'au début de chaque boucle.

```
int somme = 0;
int i = 1;
while (i <= 10) {
    somme = somme + i;
    i = i + 1;
}
```

3 Structure d'itération *Pour*

Une structure d'itération *Pour* exécute une séquence pour des valeurs successives d'une variable. Le nombre d'itérations doit être connu à l'avance et les bornes doivent être fixes.

```
somme = 0;
for (int i = 1; i <= 10; i = i + 1) {
    somme = somme + i;
}
```

Dans le cas d'une boucle *Pour* décroissante, il faut adapter les bornes et le test en conséquence.

```
somme = 0;
for (int i = 20; i >= 2; i = i - 2) {
    somme = somme + i / 2;
}
```

IV Affichages dans la console

L'affichage dans la console se fait à l'aide des méthodes prédéfinies `System.out.print` (affichage sans retour à la ligne) et `System.out.println` (affichage avec retour à la ligne).

```
System.out.println("Hello␣world!");

int n = 37;
System.out.println("Valeur␣de␣n␣=␣" + n);
```

Il est possible de formater l'affichage d'un nombre réel avec la méthode `String.format`.

```
double pi = 3.1415926535897933;
System.out.println("Valeur␣de␣pi␣=␣" + pi);
System.out.println("Valeur␣approchee␣de␣pi␣=␣" + String.format("%1.2f", pi));
```

Travaux dirigés

Exercice 1

➽ *Écrire un programme qui vérifie si un nombre entier est premier ou non.*

Exercice 2

Pour cet exercice, on utilisera la classe `String` qui modélise des chaines de caractères en Java. La JavaDoc de cette classe est la suivante :

`String` Methods	
Modifier and Type	*Method and Description*
char	charAt(int index) Returns the char value at the specified index.
int	compareTo(String anotherString) Compares two strings lexicographically.
int	compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences.
String	concat(String str) Concatenates the specified string to the end of this string.
boolean	equals(Object anObject) Compares this string to the specified object.
boolean	equalsIgnoreCase(String anotherString) Compares this String to another String, ignoring case considerations.
boolean	isEmpty() Returns true if, and only if, length() is 0.
int	length() Returns the length of this string.
String	replace(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
String	substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
String	toLowerCase() Converts all of the characters in this String to lower case using the rules of the default locale.
String	toUpperCase() Converts all of the characters in this String to upper case using the rules of the default locale.
static	String valueOf(double d) Returns the string representation of the double* argument.

*des méthodes similaires existent aussi pour les autre types primitifs

➽ *Écrire un programme qui compte le nombre d'occurences d'une chaine de caractères dans une autre.*

Exercice 3

Soit la classe `Complexe` dont la JavaDoc est donnée ci-dessous.

***Complexe* Methods**	
Modifier and Type	*Method and Description*
	Complexe() Constructeur par défaut qui construit le complexe 0.
	Complexe(double a) Constructeur qui construit le complexe a.
	Complexe(double a, double b) Constructeur qui construit le complexe $a + i \times b$.
double	module() retourne le module de `this`.
double	argument() retourne l'argument de `this`.
Complexe	conjugue() retourne le conjugué de `this`.
Complexe	inverse() retourne l'inverse de `this`.
Complexe	plus(Complexe unComplexe) retourne `this` + unComplexe.
Complexe	moins(Complexe unComplexe) retourne `this` - unComplexe.
Complexe	fois(Complexe unComplexe) retourne `this` × unComplexe.
Complexe	diviserPar(Complexe unComplexe) retourne `this` / unComplexe.
String	toString() conversion du complexe `this` en chaine de caractères.

1 *Écrire un programme qui construit les complexes* $c_1 = 2.3$, $c_2 = 0$ *et* $c_3 = -1.5 + 3.4 \times i$.

2 *Le programme doit ensuite calculer puis afficher le résultats de* $c_1 + c_3$, $c_1 * c_3$ *et le module de* c_3.

3 *Enfin, le programme calculera la somme suivante :*

$$z = \sum_{k=1}^{n} \frac{1}{k+2i}$$

Travaux pratiques

Vous possédez un compte vous permettant de vous connecter aux machines des salles d'informatique. Vérifiez lorsque vous vous connectez que vous êtes bien sur le domaine ENSMM.

Le disque dur de chaque machine est organisé en 3 parties. Le disque C (`Système`) contient le système d'exploitation de l'ordinateur, le disque D (`Applications`) contient tous les logiciels installés sur l'ordinateur et le disque F (`ENSMM`) est vide (normalement!!!). Vous ne pouvez stocker des données sur ce dernier disque que de manière temporaire (vous devez effacer tous les fichiers que vous avez créés sur le disque F avant de vous déconnecter).

Les machines sont connectées en réseau. Sur un serveur central, vous disposez chacun d'un espace pour stocker vos données. Cet espace disque est disponible depuis n'importe quelle machine ; il est accessible par l'icône `Mon Répertoire Personnel` que vous trouvez sur votre bureau.

Les fichiers nécessaires aux TPs sont également accessibles sur un répertoire du serveur. Les fichiers sont situés dans le dossier : `//Hermes/Profs/Informatique/INFO`. L'accès à ces données se fait par l'icône `Répertoire Profs` que vous trouverez également sur le bureau.

L'environnement de développement (IDE) utilisé en TP est le logiciel libre **NetBeans** [1].

Pour commencer à travailler, il faut tout d'abord recopier le dossier `TP_Objets` dans votre répertoire personnel (ou sur votre bureau), puis ouvrir le projet `TP_Objets` à l'aide NetBeans.

Un projet est constitué de plusieurs *packages* contenant des classes (un package est un simple répertoire).

Exercice 1

Dans cet exercice, nous travaillons dans le package `PremierProgramme`.

1 *Créer une classe `ProgrammePrincipal`. Pour cela, choisir `New file` dans le menu déroulant `File`. Dans la fenêtre de dialogue qui s'ouvre, choisir `Java main class` puis cliquer sur `Next`. Entrer `ProgrammePrincipal` dans le champ `Class name` et `PremierProgramme` dans le champ `Package` puis cliquer sur `Finish`.*

2 *Dans la méthode `main`, écrire un programme qui calcule et affiche les 50 premiers termes de la suite de Fibonacci de la façon suivante :*

```
f(1) = 1
f(2) = 1
f(3) = 2
f(4) = 3
f(5) = 5
f(6) = 8
f(7) = 13
f(8) = 21
f(9) = 34
...
```

Remarque : Pour lancer l'exécution de la méthode `main`, utiliser la commande `Run file` du menu déroulant `Run` (ou par un clic droit sur le fichier).

1. Téléchargeable sur https://netbeans.org/downloads/ (la version de base suffit).

Exercice 2

Dans cet exercice, nous travaillons dans le package `Complexe`. Ce package contient la classe `Complexe` vue dans le TD.

1 *Dans la méthode `main` de la classe `TestComplexe`, écrire un programme réalisant les opérations suivantes (en testant l'exécution de chaque élément 1 par 1) :*

- *construire 3 complexes z_1, z_2 et z_3 ayant les valeurs respectives 1, i et 0.*
- *afficher les valeurs des objets z_1, z_2 et z_3.*
- *calculer l'expression $z_1 + \frac{z_2}{2}$ et affecter le résultats à z_3 puis afficher z_3.*
- *afficher le module de z_3.*
- *calculer et afficher la somme suivante :*

$$z = \sum_{k=1}^{n} \frac{1}{k+2i}$$

2 *Étudier le code proposé dans la méthode `main` de la classe `TestComplexe2` et expliquer ligne par ligne les résultats obtenus.*

Exercice 3

Dans le package `Tortue` ouvrir le programme de `TestTortue`. Ce programme montre le fonctionnement de la classe `Tortue`. Cette classe permet de tracer des lignes dans un environnement graphique en déplaçant une tortue virtuelle à l'aide des commandes `avance` et `tourne`.

La javadoc de la classe `Tortue` est la suivante :

Constructor Summary

Constructors

Constructor and Description
`Tortue(FenetreGraphique fenetre, double x, double y, double angle)`

Method Summary

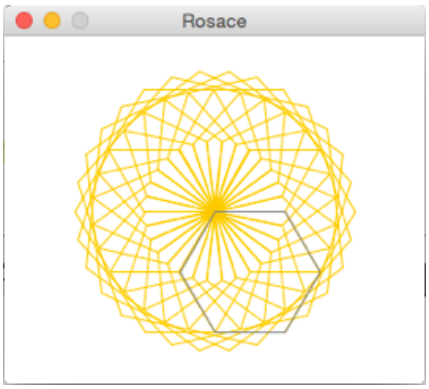
All Methods | **Instance Methods** | **Concrete Methods**

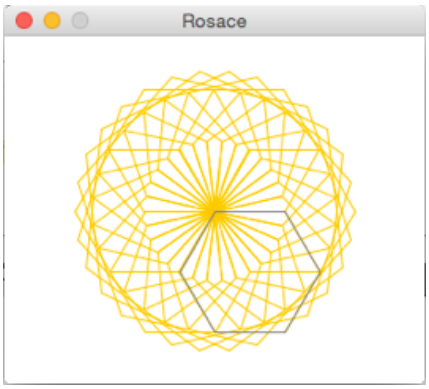
Modifier and Type	Method and Description
void	`avance(double dist)` fait avancer la tortue d'une distance dist
void	`setCouleur(java.awt.Color couleur)` change la couleur de tracé de la tortue
void	`tourne(double delta)` fait tourner la tortue dans le sens des aiguilles d'une montre d'un angle delta exprimé en degré

1 *Écrire un programme qui dessine, avec une `Tortue` un polygone régulier à n cotés de longueur l.*

2 *Transformer ce programme en une méthode de la classe `Tortue`.*

3 *Écrire un programme qui réalise les dessins suivants :*





Les classes

Compétences

- Être capable de modéliser une classe en UML et de la définir en Java ;
- Être capable d'écrire un programme testant les fonctionnalités de la classe.

Cette leçon aborde la modélisation des classes et leur implantation en Java. La définition d'une nouvelle classe suit les étapes de modélisation, de spécification et d'implantation.

I Modélisation d'une classe

Dans la phase de modélisation, on définit la représentation interne des futurs objets. Il s'agit d'identifier toutes les caractéristiques nécessaires au problème à traiter. Ces caractéristiques sont nommées *attributs*. Ils sont définis par un identifiant et la classe à laquelle ils appartiennent.

Exemple 1 : une application de gestion de stock devant manipuler des articles, nécessite la modélisation de l'entité article. Un article peut être caractérisé par un code barre, une désignation, un prix unitaire et une quantité en stock.
La classe `Article` est définie par le modèle suivant :

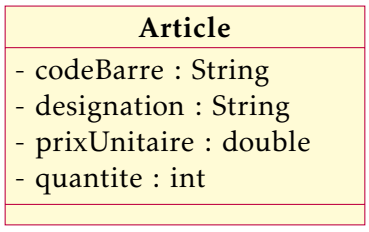


Figure 2.1 – Modèle UML de la classe `Article`

En plus de ses attributs, une classe possède un ensemble de méthodes (fonctionnalités) qui permettent de réaliser différentes opérations avec les objets de la classe. Il existe plusieurs types de méthodes :

- Les *constructeurs* qui initialisent l'état (valeurs des attributs) d'un objet
- Les *accesseurs* renseignent sur l'état d'un objet mais ne l'altère pas
- Les *modificateurs* permettent d'altérer l'état d'un objet

Tous les constructeurs portent le nom de la classe. Ils sont différenciés par la liste et le nombre de leurs arguments.
Les méthodes peuvent avoir des arguments (paramètres nécessaires à la réalisation de la fonction attendue) et fournissent au plus un résultat.

Exemple 2 : la classe `article` peut contenir les méthodes suivantes :

- un constructeur initialisant un article à partir d'informations fournies ;
- une méthode accédant à la désignation de l'article ;
- une méthode calculant la valeur du stock ;
- une méthode modifiant le prix unitaire ;
- une méthode indiquant si le stock est critique (moins de 10 articles) ;
- etc.

On ajoute donc à la classe `Article` des méthodes dans le diagramme UML (voir figure 2.2).

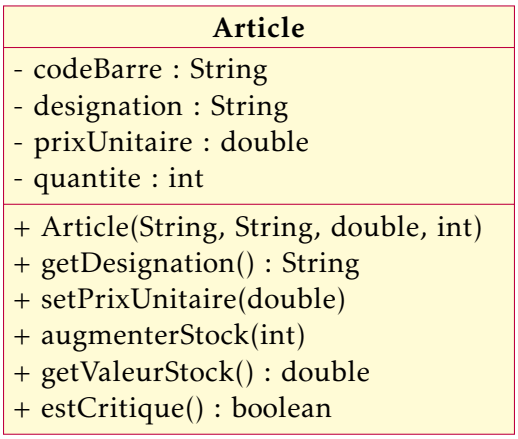


FIGURE 2.2 – Modèle UML complet de la classe `Article`

II Implantation d'une classe en Java

Lorsque le modèle est complet, il peut être codé dans un langage de programmation objet. Nous illustrons ici le codage de la classe `Article` avec le langage Java.

Implantation de la classe `Article`

```
public class Article {

    // attributs de la classe Article
    private String codeBarre;
    private String designation;
    private double prixUnitaire;
    private int quantite;

    // constructeurs de la classe Article
    public Article(String cb, String design, double pu) {
        this.codeBarre = cb;
        this.designation = design;
        this.prixUnitaire = pu;
        this.quantite = 0;
    }

    public Article() {
        this("1111111111111", "article sans nom", 0.0);
    }

    // méthodes de la classe Article
    public String getDesignation() {
        return designation;
    }

    public void augmenterStock(int nbUnite) {
```

```
        this.quantite = this.quantite + nbUnite;
    }

    public void setPrixUnitaire(double nouveauPU) {
        this.prixUnitaire = nouveauPU;
    }

    public double getValeurStock() {
        return this.prixUnitaire * this.quantite;
    }

    public boolean estCritique() {
        return this.quantite < 10;
    }

}
```

Une fois définie, la classe `Article` peut être utilisée de la façon suivante :

```
public class Test {
    public static void main(String arg[])
    {
        Article unArticle;

        unArticle = new Article("1234567890123", "Robinet", 56.32);
        unArticle.augmenterStock(15);
        System.out.println(unArticle.getDesignation());
        System.out.println("valeur␣du␣stock␣:␣" + unArticle.getValeurStock());
        System.out.println("Stock␣critique␣?␣" + unArticle.estCritique());
    }
}
```

III Quelques méthodes classique de Java

En Java, il existe quelques méthodes très classiques que l'on peut (ou doit) également écrire. Elles permettent de faciliter l'utilisation de la classe par la suite.

On en trouve principalement 3 :

- `String toString()`
  Permet de générer une chaine de caractères caractérisant l'état de l'objet pour lequel elle est invoquée. Cette méthode est utilisée de façon automatique pour aussitôt que l'objet doit être converti en une chaîne de caractères comme dans la méthode `System.out.println()` qui assure l'affichage de texte dans la console.
  Cette méthode n'a pas d'argument et retourne une chaîne de caractères de type `String`.

  ```
  public class Article {
          ...

     public String toString() {
          return this.codeBarre + "␣-␣" + this.designation + "␣-␣Prix:␣"
                  + this.prixUnitaire + "␣-␣Stock:" + this.quantite;
      }
  ```

- `boolean equals(Object obj)`
  Permet de programmer l'égalité entre deux objets d'une même classe. Elle doit retourner `true` si `this` et l'objet en argument sont égaux et `false` sinon.
  Cette méthode prend un `Object` en argument et retourne un booléen de type `boolean`.

```
public class Article {
        ...
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj == null) {
            return false;
        }
        if (!obj.getClass().equals(this.getClass())) {
            return false;
        }
        Article art = (Article) obj;
        if (! this.codeBarre.equals(art.codeBarre)) {
            return false;
        }
        return true;
    }
```

- `int compareTo(Object obj)`
  Permet de programmer la comparaison entre deux objets d'une même classe. Elle doit retourner `-1` si `this` est inférieur à l'objet en argument, `0` s'ils sont égaux et `1` sinon. Cette méthode n'a lieu d'être que si une relation d'ordre est nécessaire entre les objets de la classe.
  Cette méthode prend un `Object` en argument et retourne un entier de type `int`.

```
public class Article {
        ...
    public int compareTo(Object obj) {
        Article art = (Article) obj;
        if (this.getValeurStock() < art.getValeurStock()) {
            return -1;
        }
        if (this.getValeurStock() > art.getValeurStock()) {
            return 1;
        }
        return 0;
    }
```

L'exécution du code suivant :

```
Article artA, artB;

artA = new Article("1234567890123", "Robinet", 56.32);
artA.augmenterStock(15);
artB = new Article("1234567890123", "Robinet", 45.99);
artB.augmenterStock(10);
System.out.println("l'article␣A␣est␣le␣suivant␣:␣" + artA); // toString appelée
System.out.println(artA.equals(artB));
if (artA.compareTo(artB) < 0) {
    System.out.println("valeur␣du␣stock␣de␣\n" + artA + "\n␣inférieure␣à␣\n" + artB);
} else {
    System.out.println("valeur␣du␣stock␣de␣\n" + artB + "\n␣inférieure␣à␣\n" + artA);
}
```

Donne le résultat suivant :

```
run:
l'article␣A␣est␣le␣suivant␣:␣1234567890123␣-␣Robinet␣-␣Prix:␣56.32␣-␣Stock:15
true
valeur␣du␣stock␣de
1234567890123␣-␣Robinet␣-␣Prix:␣45.99␣-␣Stock:10
␣inférieure␣à
1234567890123␣-␣Robinet␣-␣Prix:␣56.32␣-␣Stock:15
BUILD␣SUCCESSFUL␣(total␣time:␣0␣seconds)
```

Travaux dirigés

Exercice 1

Soit la classe `Rationnel` modélisant un nombre rationnel (quotient de deux entiers relatifs). Cette classe doit contenir :

- un constructeur complet initialisant l'objet à $\frac{numerateur}{denominateur}$.
- un constructeur à partir d'un entier initialisant l'objet à $\frac{numerateur}{1}$.
- les accesseurs et les modificateurs pour le numérateur et le dénominateur.
- une méthode calculant la valeur décimale du rationnel (approximation).
- une méthode simplifiant le rationnel sous sa forme irréductible à l'aide de la méthode `long gcd(long u, long v)` de la classe `OutilsMath` qui renvoie le plus grand commun diviseur de deux nombres entiers.
- une méthode `fois` calculant le rationnel égal au produit du rationnel `this` avec un autre rationnel.
- une méthode `plus` calculant le rationnel égal à la somme du rationnel `this` avec un autre rationnel.
- une méthode `toString`.
- une méthode `equals`.

1 *Modéliser la classe en UML.*

2 *Spécifier et implanter la classe `Rationnel` en java.*

3 *Proposer un programme de test de la classe `Rationnel`.*

Exercice 2

On souhaite développer en Java une classe `Point` permettant de modéliser des points du plan. En plus de ses coordonnées, chaque point possède un nom qui peut contenir un ou plusieurs caractères.

La classe `Point` qui doit contenir :

- un constructeur complet qui permet la création d'un point à partir de coordonnées connues et de son nom.
- un constructeur par défaut qui permet la création d'un objet `Point` dont les coordonnées sont les coordonnées de l'origine du plan.
- une méthode calculant la longueur euclidienne qui sépare le point courant d'un autre point (à l'aide de la méthode `sqrt()` de la classe `Math`).
- une méthode calculant le point symétrique du point courant selon un point donné.
- une méthode calculant le point milieu entre le point courant et un point donné.

➽ *Spécifier et implanter la classe `Point` en java.*

Exercice 3

On souhaite développer en Java une classe `Triangle` permettant de modéliser des triangles. Un triangle est connu par les coordonnées de ses trois sommets. Ainsi un triangle est composé de trois points de la classe `Point` définie dans l'exercice précédent.

➽ *Écrire la classe `Triangle` qui doit contenir :*

- *un constructeur qui définit un triangle à partir de trois points.*
- *un constructeur par défaut qui définit un triangle avec les sommets de coordonnées (0,0), (1,0) et (0,1).*
- *une méthode calculant le périmètre du triangle.*
- *une méthode donnant le triangle médian du triangle courant (triangle qui joint les milieux des côtés).*
- *une méthode `compareTo` comparant deux triangles sur leur périmètre.*
- *une méthode construisant le triangle symétrique du triangle courant suivant une symétrie centrale selon un point donné.*

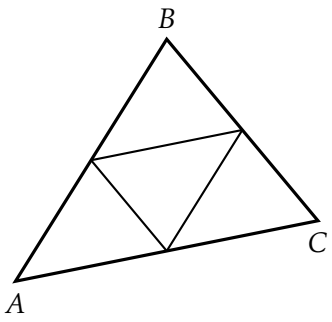


Figure 2.3 – Triangle médian du triangle ABC.

Travaux pratiques

Pour commencer à travailler, il faut tout d'abord recopier le dossier du `TP_Classes` dans votre répertoire personnel (ou sur votre bureau), puis ouvrir le projet `TP_Classes` à l'aide NetBeans.

Le package `ExempleCours` contient la classe `Article` qui donnée à titre d'exemple.

Exercice 1

Dans cette partie, nous travaillerons dans le package `Individu`.

On souhaite développer en Java une application manipulant des individus. Chaque individu est caractérisé par son nom, son prénom, son année de naissance, sa taille, son poids et son sexe.

Tout d'abord, il s'agit de définir une classe `Individu` :

1. *Choisir la commande `File->New File`, sélectionner `Java Class` dans la liste puis préciser le nom de la classe.*
2. *Définir les attributs de la classe `Individu`.*
3. *Définir un constructeur complet et un constructeur par défaut.*
4. *Ajouter les accesseurs et les modificateurs pour la taille.*
5. *Ajouter la méthode `toString`.*

Une fois ces méthodes définies, il est possible de tester la classe `Individu` :

6. *Définir une classe `TestIndividu` contenant une méthode `main` (à l'aide de la commande `File->New File->Java Main Class`).*
7. *Écrire un programme de test construisant un individu et affichant ses caractéristiques dans la console.*
8. *Il s'agit enfin d'étoffer la classe `Individu` en ajouter les méthodes ci-dessous* ***et en les testant au fur et à mesure*** *:*
   - *un modificateur du nom qui force la casse à majuscule (à l'aide de la méthode `toUpperCase()` de la classe `String`).*
   - *une méthode calculant l'IMC de l'individu (poids/taille*taille)*
   - *une méthode permettant de savoir si l'individu a une corpulence normal* ($18.5 < IMC < 25$).
   - *une méthode permettant de générer une adresse mail de la forme `nom.prenom@ens2m.org`.*
   - *une méthode `equals` (égalité sur le nom, le prénom et l'année de naissance).*

Exercice 2

Dans cette partie, nous travaillerons dans le package `Geometrie`.

Commencer par implanter et tester les classes `Point` et `Triangle` de l'exercice de TD (avec des coordonnées réelles).

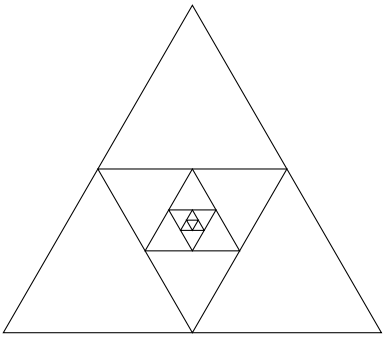
Remarque : les accesseurs, modificateurs, constructeurs et méthodes usuelles peuvent être générées automatiquement à l'aide de la commande `Source->Insert Code` une fois que les attributs de la classe ont été définis.

On souhaite maintenant dessiner les triangles dans une fenêtre graphique :

1 *Ouvrir la classe `TestFenetreGraphique` et étudier l'utilisation de la classe `FenetreGraphique`.*

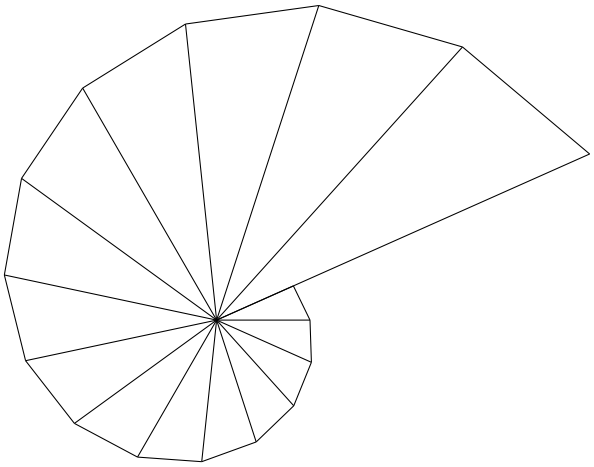
2 *Ajouter une méthode à la classe `Triangle` permettant de dessiner le triangle dans une fenêtre graphique.*

3 *Ajouter une méthode permettant de dessiner la suite des triangles médians illustrée ci-dessous :*



Exercice 3

➽ *Ajouter les méthodes nécessaires (rotations, homothéties, etc.) dans les classes `Point` et `Triangle` permettant de dessiner la spirale illustrée ci-dessous :*



Les tableaux

Compétences

- savoir définir et utiliser un tableau en attribut dans une classe ou en variable locale dans une méthode ;
- être capable d'écrire des algorithmes comme faire la somme des éléments d'un tableau, savoir si une certaine valeur est présente dans un tableau, connaître la valeur maximale des éléments d'un tableau, etc.

Un tableau est un ensemble d'éléments tous de même nature (type primitif ou classe). En Java, on déclare une variable de type tableau en ajoutant `[]` avant ou après le nom du tableau :

```
// tabInt est un tableau à une dimension de int
int tabInt[];
// tabChar est un tableau à une dimension de char
char[] tabChar ;
```

La représentation interne d'un tableau est une référence comme pour une classe. Il est nécessaire d'allouer la mémoire souhaitée pour utiliser le tableau. La création d'un tableau se fait avec l'instruction `new`. C'est au moment de la création d'un tableau que l'on fixe la taille (nombre cellules) de ce dernier en indiquant entre crochet à la suite du type des informations (voir l'exemple ci-dessous).

```
// tabInt est un tableau à une dimension de int
int tabInt [] ;

// création d'un tableau d'entiers de 10 cellules indicées de 0 à 9
tabInt = new int[10] ;
```

Lorsqu'un tableau est créé, l'accès aux cellules se fait en donnant le nom du tableau suivi de l'indice de la cellule entre crochets (`[]`). L'accès est possible en écriture et en lecture, ce qui signifie que l'on peut changer la valeur de la cellule ou simplement y accéder. L'indice de la première cellule du tableau est 0.

```
// utilisation en écriture
tabInt[0] = 10 * 20;
// accès en lecture
tabInt[1] = tabInt[0] + 10;
```

Remarque

L'indice utilisé doit exister. Si un indice hors borne est utilisé une exception est levée par Java (`ArrayIndexOutOfBoundsException`).

L'attribut `length` permet de connaître le nombre de cellules d'un tableau.

```
// déclaration et création d'un tableau tabInt de 10 int
int tabInt[] = new int[10];

// affiche la taille de tabInt, soit 10
System.out.println(tabInt.length);
```

```java
// somme des valeurs de tabInt
int somme = 0;
for (int i=0;i<tabInt.length;i++) {
    somme = somme + tabInt[i];
}
```

Il est possible de définir des tableaux à plusieurs dimensions.

```java
// déclaration et création d'un tableau tabInt de 5x10 int
int tabInt[][] = new int[5][10];

// utilisation en écriture
tabInt[0][0] = 1;
// accès en lecture
tabInt[1][0] = tabInt[0][0] + 1;
```

Travaux dirigés

Exercice 1

On souhaite développer en Java une application qui gère les relevés météorologiques annuels de différentes villes. Un relevé (`ReleveMeteo`) est caractérisé par le nom de la ville, l'année du relevé et l'ensemble des mesures de température moyenne journalière.

➽ *Définir la classe* `ReleveMeteo` *qui doit contenir au moins les méthodes suivantes :*

1. *un constructeur qui définit une relevé météo à partir d'un ville donnée, d'une année donnée et initialisant toutes les températures à 0 ;*
2. *une méthode qui retourne la température pour un jour donné ;*
3. *une méthode qui remplace une température pour un jour donné ;*
4. *une méthode qui calcule la moyenne des températures de toute une année ;*
5. *une méthode qui indique si il a gelé au moins une fois dans l'année ;*
6. *une méthode qui compte le nombre de jours dont la température est supérieure ou égale à une valeur donnée et strictement inférieure à une seconde valeur donnée ;*
7. *une méthode qui donne la température maximale atteinte dans l'année ;*
8. *une méthode* `toString` *;*
9. *une méthode* `compareTo` *qui compare deux relevés météorologiques par la moyenne des températures ;*
10. *une méthode qui donne le nombre de périodes au cours desquels la température est inférieure à une valeur donnée.*
11. *une méthode qui retourne la durée de la plus grande période au cours de laquelle la température est supérieure à une valeur donnée ;*
12. *une méthode qui retourne l'histogramme des températures du relevé météo. Pour cela, on comptabilise le nombre d'occurrences des températures comprises entre deux valeurs entières consécutives (en pratique entre la partie entière inférieure au sens large et la partie entière supérieure au sens strict de la température examinée). Le nombre de cases de l'histogramme est donc déterminé par les valeurs extrêmes du relevé météo et non par le nombre d'intervalles pour lesquels le nombre d'occurrences est non nul sachant que cette information n'est pas connue à l'avance.*

Exercice 2

Dans cet exercice, on voudrait développer une classe pour manipuler des polygones réguliers et convexes. Un tel Polygone possède `n` cotés et donc `n` sommets. Tous les sommets appartiennent à un même cercle et l'angle défini par deux cotés consécutifs est le même quels que soient les cotés consécutifs considérés.

➽ *Définir les éléments suivants :*

1 *les attributs utiles à la description d'un tel polygone en utilisant une classe* `Point` *supposée préalablement définie (voir TD précédent).*

2 *un constructeur permettant la création d'un polygone à* `n` *cotés tel que tous les sommets du polygone appartiennent au cercle de centre l'origine du repère et dont le rayon ait pour valeur* `1`*. Nous faisons le choix de créer les points du polygone dans le sens anti-horaire ou sens trigonométrique. Le premier point du polygone peut être par exemple le point de coordonnées* $(0,1)$*. Les autres points sont alors obtenus en appliquant des rotations successives d'angle* $2\pi/n$.

3 *une méthode permettant de faire une translation du polygone courant suivant un vecteur connu par ses deux coordonnées.*

4 *un constructeur permettant la création d'un polygone à* `n` *cotés tel que tous les sommets du polygone appartiennent au cercle de centre* `P`*, un point du plan, et dont le rayon ait pour valeur* `r`*.*

5 *une méthode permettant de trouver le point du polygone courant le plus haut dans le plan. On suppose dans la suite que des méthodes similaires sont définies pour le point le plus bas, le plus à gauche et le plus à droite.*

6 *une méthode permettant de connaître le sommet inférieur gauche du carré le plus petit qui recouvre exactement le polygone. On suppose dans la suite qu'une méthode similaire est définie pour le point supérieur droit.*

7 *une méthode permettant de calculer le périmètre du polygone courant.*

8 *une méthode qui permette de savoir si un point* `P` *quelconque du plan est à l'intérieur ou à l'extérieur du polygone courant.*

Le principe est de considérer chaque coté du polygone. Si le point `P` *est toujours situé à gauche des cotés du polygone, alors cela signifie que* `P` *est à l'intérieur du polygone. Par contre, si cela n'est pas le cas, même une seule fois, cela signifie que le* `P` *est à l'extérieur du polygone.*

Pour savoir si `P` *est à gauche d'un coté* `AB` *du polygone, alors le déterminant des vecteurs* $\overrightarrow{AB}$ *et* $\overrightarrow{AP}$ *doit être strictement positif. S'il est nul, alors* `P` *est sur la droite portée par* `AB`*. C'est donc le cas positif ou nul qu'il faut considérer.*

Nous considérons que nous disposons de la méthode `estAGaucheDe(Point a, Point b)` *de la classe* `Point` *qui indique si le point courant est à gauche du segment* `ab`*.*

Travaux pratiques

L'objectif de ce TP est d'utiliser les tableaux en Java pour réaliser quelques traitements audio que l'on retrouve dans les éditeurs audio numérique comme Audacity (voir figure 3.1).

Compétences

- savoir définir et utiliser un tableau en attribut dans une classe ou en variable locale dans une méthode ;
- être capable d'écrire des algorithmes comme faire la somme des éléments d'un tableau, savoir si une certaine valeur est présente dans un tableau, connaître la valeur maximale des éléments d'un tableau, etc.
- comprendre et coder des algorithmes élémentaires du traitement numérique du signal.

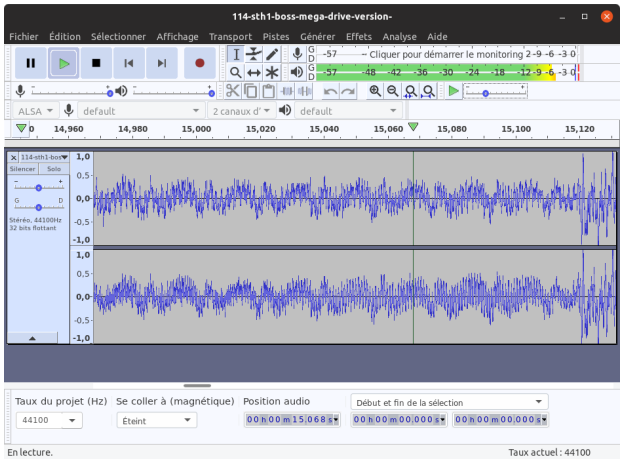


Figure 3.1 – Audacity, un éditeur audio numérique multi-piste, libre et téléchargeable à l'adresse https://www.audacityteam.org/.

I Représentation d'un son

La numérisation d'un signal comporte trois phases : l'échantillonnage, la quantification et le codage.

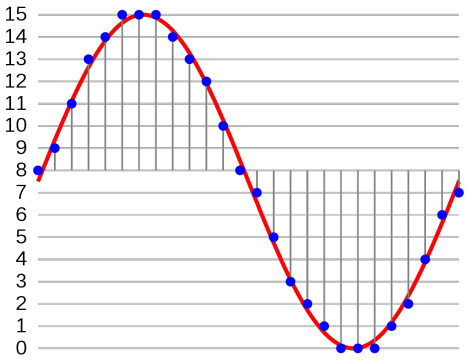


Figure 3.2 – Échantillonnage et quantification sur 4 bits d'un signal analogique.

L'échantillonnage consiste à prélever les valeurs d'un signal à intervalles définis, généralement réguliers. La période d'échantillonnage T_e définit la période entre la prise de deux échantillons. La fréquence échantillonnage f_e est égale à $1/T_e$.

La quantification transforme la valeur analogique du signal en une valeur prise dans une liste finie de valeurs valides réparties dans la plage de mesure. L'écart entre le signal original et le signal échantillonné est fonction du pas de quantification qui correspond à l'intervalle entre deux niveaux de quantification. Par exemple, quand les échantillons sont codés sur 16 bits, il y a 65536 pas de quantification.

Enfin, le codage fait correspondre un type de variable (entier ou flottant, signé ou non) à chaque valeur de signal quantifiée.

La première partie du TP s'intéresse à la définition de signaux audio que nous appelons communément des *sons*. Un son est défini par une fréquence d'échantillonnage et par un tableau contenant les valeurs x_k du signal à instants réguliers (échantillons).

Pour faciliter les calculs, nous nous limitons à des signaux audio monophoniques (un seul canal) contenant des échantillons représentés par des nombres réels compris entre -1.0 et 1.0 et codés par des `double`.

1 *Créer une classe* `Son` *dans le package* `son` *à l'aide de la commande* `File->New File->Java Class` *puis définir ses deux attributs et les méthodes suivantes :*

a. *un constructeur qui définit un son nul (silence) à partir une fréquence d'échantillonnage et une durée en seconde données ;*
b. *un constructeur qui définit une tonalité (signal sinusoïdale) à partir d'une fréquence d'échantillonnage, d'une durée, d'une amplitude et d'une fréquence de tonalité données ;*
c. *une méthode qui retourne la fréquence d'échantillonnage du son ;*
d. *une méthode permettant de modifier la fréquence d'échantillonnage du son ;*
e. *une méthode qui retourne la durée du son ;*
f. *une méthode qui retourne la valeur efficace du son (moyenne quadratique, RMS) :*

$$x_{eff} = \sqrt{\frac{1}{n}\sum_{k=0}^{n-1} {x_k}^2} \tag{3.1}$$

g. *une méthode qui indique si le son est constant (ne contient que des échantillons de valeurs identiques) ;*
h. *une méthode qui calcule la valeur crête-à-crête du son (valeur maximale moins valeur minimale des échantillons) ;*

$$x_{cac} = \max_{0 \le k < n} x_k - \min_{0 \le k < n} x_k \tag{3.2}$$

i. *une méthode* `toString` *indiquant la fréquence, la durée, la valeur efficace, la valeur crête-à-crête et si le son est constant ou non.*

2 *Créer une classe* `TestSon` *dans le package* `son` *à l'aide de la commande* `File->New File->Java Main Class` *puis construire et afficher un son nul puis une tonalité.*

II API Java Sound

Afin de rendre le projet plus ludique, une classe `OutilsAudio` est fournie dans le projet NetBeans. Cette classe permet de jouer un son sur l'ordinateur, de lire et d'écrire des fichiers audio au format WAV. La classe `OutilsAudio` utilise des commandes de l'API Java Sound qui est une bibliothèque bas niveau pour la manipulation des flux audio d'entrée/sortie.

Le Waveform Audio File Format (WAV) est un format conteneur destiné au stockage audio numérique et notamment de signaux sans compression et avec un pas de quantification constant. A l'aide d'Audacity, vous pourrez très facilement mettre un son au format du TP (PCM_SIGNED, 16 bits, mono, 2 bytes/frame, little-endian) et l'enregistrer dans un fichier WAV pour le traiter avec vos algorithmes.

Voici un aperçu des méthodes de la classe `OutilsAudio` :

```
public class OutilsAudio {

    public static void jouer(double[] echantillons, double frequenceEchantillonnage) {
        ...
    }

    public static double[] lireEchantillonsFichierWAV(String nomDuFichier) {
        ...
    }

    public static AudioFormat lireFormatFichierWAV(String nomDuFichier) {
        ...
    }

    public static void ecrireFichierWAV(String nomDuFichier,double[] echantillons, double
     frequenceEchantillonnage) {
        ...
    }

}
```

La classe `AudioFormat` contient notamment la fréquence d'échantillonnage et peut être récupérée en écrivant `OutilsAudio.lireFormatFichierWAV(nomDuFichier).getSampleRate();`

1 *A l'aide de la classe `OutilsAudio`, ajouter à la classe `Son` :*

   a. *un constructeur permettant de définir un son à partir d'un fichier WAV ;*
   b. *une méthode pour jouer un son ;*

2 *Écrire un programme de test dans `TestSon` permettant de :*

   a. *construire une tonalité de 440 Hz, d'afficher ses caractéristiques et de la jouer ;*
   b. *construire un son à partir du fichier `cable.wav`, d'afficher ses caractéristiques et de le jouer ;*
   c. *d'augmenter sa fréquence d'échantillonnage et de le jouer à nouveau.*

III Filtrage numérique

Un filtre numérique linéaire causal et invariant dans le temps est défini par une équation aux différences entre les entrées et les sorties :

$$y_k = b_0 x_k + b_1 x_{k-1} + b_2 x_{k-2} + \cdots + b_n x_{k-n} - a_1 y_{k-1} - a_2 y_{k-2} - \cdots - a_m y_{k-m} \tag{3.3}$$

ou de manière équivalente par sa fonction de transfert dans le domaine Z de la forme :

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_n z^{-n}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_m z^{-m}} \tag{3.4}$$

Cette partie propose d'aborder quelques filtres parmi les plus classiques du traitement numérique du signal et de les tester en direct avec des sons adaptés.

1 Amplification et saturation

Le filtre le plus simple est l'amplification qui consiste à multiplier l'entrée par un gain b :

$$y_k = b\, x_k \tag{3.5}$$

Un filtre numérique peut être calculé en place ou hors place. En informatique, un algorithme *en place* est un algorithme qui transforme les valeurs d'un tableau sans utiliser de tableau supplémentaire. L'entrée est remplacée par la sortie au fur et à mesure de l'exécution de l'algorithme. Dans la suite, nous écrirons uniquement des algorithmes *en place*.

1 *Ajouter à la classe* `Son` *une méthode qui amplifie un son avec un gain donné.*

2 *Écrire un programme de test qui construise un son à partir du fichier* `guitare.wav` *qui l'amplifie* 4× *et qui le joue. Qu'observe-t'on ?*

2 Écho

Un autre filtre très simple mais particulièrement amusant en traitement audio est l'écho. Pour obtenir un effet d'écho, il suffit de mixer le signal avec lui-même retardé de n périodes :

$$y_k = x_k + b\, x_{k-n} \qquad \text{ou} \qquad H(z) = 1 + b\, z^{-n} \tag{3.6}$$

3 *Ajouter à la classe* `Son` *une méthode qui ajoute un écho avec un gain donné et un retard donné en seconde.*

4 *Tester le filtre sur le son* `e-ho.wav`.

L'écho est un filtre à réponse impulsionnelle finie (RIF), en anglais FIR (*finite impulse response*). Ce type de filtre est dit fini, car sa réponse impulsionnelle se stabilise à zéro en un temps fini. Un filtre RIF est non récursif, c'est-à-dire que la sortie dépend uniquement de l'entrée du signal, il n'y a pas de contre-réaction. Ainsi, les coefficients a_k de la forme générale sont tous égaux à zéro.

3 Réverbération

Si on veut que l'écho se répète plusieurs fois pour produire un effet de réverbération, on peut utiliser un filtre à réponse impulsionnelle infini :

$$y_k = x_k - a\, y_{k-m} \qquad \text{ou} \qquad H(z) = \frac{1}{1 + a\, z^{-m}} \tag{3.7}$$

Les filtres à réponse impulsionnelle infinie (RII), en anglais IIR (*infinite impulse response*), possèdent une réponse impulsionnelle qui ne s'annule jamais définitivement même si elle converge éventuellement vers zéro à l'infini. Ce type de filtre est récursif, c'est-à-dire que la sortie du filtre dépend à la fois du signal d'entrée et du signal de sortie, il possède ainsi une boucle de contre-réaction (feedback).

5 *Ajouter à la classe* `Son` *une méthode qui ajoute de la réverbération avec un gain donné et un retard donnés en seconde.*

6 *Tester le filtre sur le son* `e-ho.wav`.

4 Filtres passe-bas

D'autres filtres permettent de réduire ou bien d'accentuer certaines parties du spectre sonore présentes dans un signal. Par exemple, un filtre passe-bas laisse passer les basses fréquences et atténue les hautes fréquences.

Le filtre moyenneur (appelé aussi moyenne mobile, ou moyenne glissante) est un filtre passe-bas à réponse impulsionnelle finie qui est défini par :

$$y_k = \frac{x_k + x_{k-1} + x_{k-2} + \cdots + x_{k-n+1}}{n} \tag{3.8}$$

où n est la taille de la fenêtre glissante.

7 *Ajouter à la classe* `Son` *une méthode qui applique le filtre moyenneur avec une taille de fenêtre donné.*

8 *Tester le filtre sur le son* `back.wav`.

En remarquant que :

$$y_k = \frac{x_k}{n} + \frac{(x_{k-1} + x_{k-2} + \cdots + x_{k-n+1})}{n} \tag{3.9}$$

$$= \frac{1}{n}x_k + \frac{(n-1)}{n} \cdot \frac{(x_{k-1} + x_{k-2} + \cdots + x_{k-n+1})}{(n-1)} \tag{3.10}$$

$$\approx \frac{1}{n}x_k + (1 - \frac{1}{n}) \cdot y_{k-1} \tag{3.11}$$

On obtient le filtre passe-bas du premier ordre suivant :

$$y_k = \alpha x_k + (1-\alpha)y_{k-1} \qquad \text{ou} \qquad H(z) = \frac{\alpha}{1-(1-\alpha)z^{-1}} \tag{3.12}$$

Ce filtre est appelé filtre moyenneur exponentiel. Contrairement au filtre moyenneur précédent, ce filtre a une réponse impulsionnelle infinie.

Sa fréquence de coupure est :

$$f_c = -\frac{\ln(1-\alpha)}{2\pi} f_e \tag{3.13}$$

et donc :

$$\alpha = 1 - e^{-\frac{2\pi f_c}{f_e}} \tag{3.14}$$

9 *Ajouter à la classe* `Son` *une méthode qui applique le filtre moyenneur exponentiel avec une fréquence de coupure donnée.*

10 *Tester le filtre sur le son* `back.wav`.

5 Filtres passe-haut

Un filtre passe-haut est un filtre qui laisse passer les hautes fréquences et qui atténue les basses fréquences. Une première idée pour accentuer les hautes fréquences est de simplement dériver le signal. En temps discret, on peut utiliser la formule des différences finies :

$$y_k = x_k - x_{k-1} \qquad \text{ou} \qquad H(z) = 1 - z^{-1} \tag{3.15}$$

Ce filtre dérivateur (RIF) n'est utilisable que si la fréquence d'échantillonnage est très grande devant la fréquence du signal. De plus, le filtre dérivateur a un gain proportionnel à la fréquence. Il est donc très sensible au bruit présent dans le signal, particulièrement les parties hautes fréquences du bruit.

11 *Ajouter à la classe `Son` une méthode qui applique le filtre dérivateur.*

12 *Tester le filtre sur le son `back.wav`.*

Afin de pouvoir régler la fréquence de coupure du filtre, on peut utiliser un filtre RII du premier ordre :

$$y_k = \alpha(x_k - x_{k-1}) + \alpha y_{k-1} \tag{3.16}$$

Sa fréquence de coupure est alors :

$$f_c = \frac{(1-\alpha)}{2\pi\alpha} f_e \tag{3.17}$$

et donc :

$$\alpha = \frac{1}{2\pi\frac{f_c}{f_e} + 1} \tag{3.18}$$

13 *Ajouter à la classe `Son` une méthode qui applique le filtre passe-haut du premier ordre avec une fréquence de coupure donnée.*

14 *Tester le filtre sur le son `back.wav`.*

6 Compression dynamique

L'un des effets les plus utilisés en production musicale est la compression dynamique. Cet effet permet de réduire la plage dynamique d'un enregistrement, c'est-à-dire, réduire l'écart entre les sons les plus forts et les sons les plus faibles de la piste.

La compression dynamique est notamment utilisée pour faciliter l'écoute dans un environnement bruité. Cependant, quand on compresse beaucoup avec un gain de sortie élevé, c'est dangereux pour le système auditif sur le long terme car le niveau moyen du son (RMS) devient et reste élevé en permanence. L'oreille interne n'a plus le temps de se relaxer mécaniquement et il s'en suit une usure prématurée du système auditif.

Plus d'informations sur la compression :
https://www.projethomestudio.fr/reglages-compresseur-audio/

Plus d'informations sur ses dangers :
https://fr.wikipedia.org/wiki/Guerre_du_volume et
http://voyard.free.fr/textes_audio/dangers.htm

La classe `ArrayList`

Compétences

- Savoir construire et utiliser une `ArrayList` (`add`, `remove`, `size`, `contains`, etc.).
- Savoir utiliser les méthodes de la classe `Collections` (`sort`, `min`, `max`).
- Etre capable de définir une classe contenant une liste d'objets.
- Etre capable d'écrire des algorithmes spécifiques de gestion d'une liste d'objets.

Cette leçon est consacrée à la découverte d'une structure de données dynamique. Nous avons précédemment abordé la notion de tableaux qui est déjà une structure de données permettant de gérer un ensemble d'objets. Un tableau (éventuellement à plusieurs dimensions) est dit *statique*, car une fois l'allocation effectuée (i.e., la taille exprimée en nombre d'objets à stocker), il n'est plus possible de la modifier (i.e., la taille est fixe). Une structure dynamique offre par contre l'avantage de permettre l'ajout de nouveaux éléments, même si cela n'était pas prévu au moment de sa création. Il existe plusieurs structures de données dynamiques en Java. Elles sont définies comme des classes et à ce titre, elles disposent de différentes fonctionnalités, ou méthodes, qui permettent une manipulation implicite et aisée des données qu'elles contiennent, chacune avec ses spécificités.

Nous proposons de découvrir ici l'une des structures dynamiques de Java, la classe `ArrayList` qui peut être vue comme un tableau dynamique.

La classe `ArrayList` peut être considérée comme un tableau à une dimension qui peut être agrandi ou rétréci à la demande sans que le programmeur ait à se soucier de la mise en œuvre de ces opérations mémoires. Les différents éléments peuvent être accédés grâce à leur indice dans la liste. Comme tous les indices en Java, leur valeur appartient à l'intervalle $[\![0, size - 1]\!]$. Dans la suite de ce cours, nous illustrons l'utilisation d'une `ArrayList`.

I Déclaration et construction d'une `ArrayList`

Afin de pouvoir utiliser toutes les facilités offertes par les `ArrayList`, il faut systématiquement la nature des objets qu'elle contient au moment de sa création. Cette précision est écrite entre chevrons ("<" et ">") au niveau de la déclaration et de l'appel du constructeur de la classe `ArrayList`.

Prenons l'exemple d'une `ArrayList` contenant des chaînes de caractères (`String`) et d'une autre `ArrayList` contenant des rationnels de la classe `Rationnel` vue dans la première leçon, la déclaration et la création de telles `ArrayList` se fait de la manière suivante :

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

ArrayList<Rationnel> listeRationnels ;
listeRationnels = new ArrayList<Rationnel>() ;
```

Nous disposons à présent de deux listes pouvant accueillir respectivement des chaînes de caractères et des rationnels.

II Méthodes de la classe `ArrayList`

Au niveau de la classe `ArrayList`, il existe plus de 30 méthodes. Nous ne les détaillons pas toutes ici. L'API complète peut être consultée directement pour le site d'Oracle pour plus d'information [1].
Dans la suite, au niveau des définitions des méthodes, nous considérons une `ArrayList` contenant les objets de la classe `Element` et qui aurait été déclarée de la manière suivante :

```
ArrayList<Element> listeElements = new ArrayList<Element>() ;
```

Un exemple d'utilisation d'une `ArrayList` d'entiers est donné à la fin de cette partie du cours afin d'illustrer simplement quelques une des méthodes décrites ici.

1 Ajouts

- `boolean add(Element element)`
  Cette méthode ajoute un objet de la classe `Element` à la fin de la liste sur laquelle elle est appelée. La complexité associée est $O(1)$, c'est à dire qu'un tel ajout a un coût constant quelque soit la taille de la liste.

- `void add(int i, Element element)`
  Cette méthode ajoute un objet de la classe `Element` au rang `i` de la liste sur laquelle est appelée cette méthode. Dans ce cas, tous les éléments situés après le rang `i` sont décalés de 1. Si la liste contient au départ n éléments, la complexité d'un tel ajout est $O(n-i+1)$ soit en moyenne $O(n)$. Il s'agit par conséquent d'une opération qui peut s'avérer coûteuse si elle est répétée souvent.

Exemple :

```
phrase.add("Jacques");  // ajout en fin
phrase.add(0, "Pierre");  // ajout en début
phrase.add(1, "Paul");  // ajout en deuxième position
```

2 Suppressions

- `Element remove(int i)`
  Cette méthode supprime et retourne l'élément placé au rang `i` de l'`ArrayList` sur laquelle elle est appelée. $n-i$ décalages sont nécessaires. La complexité est alors $O(n-i)$ soit $O(n)$ en moyenne.

- `boolean remove(Element element)`
  Cette méthode supprime la première occurrence d'un objet identique à `element`. Même remarque quant à la complexité de l'opération, avec en plus la nécessité de rechercher `element` grâce à la méthode `equals` de la classe `Element`. Cette méthode renvoie `false` si `element` ne faisait pas partie de la liste et `true` sinon.

- `void clear()`
  Cette méthode supprime tous les éléments de la liste.

Exemple :

```
phrase.remove(2);  // suppression du troisième mot de la phrase
String mot = phrase.remove(1);  // suppression et récupération du deuxième mot de la phrase
phrase.remove("Paul");  // suppression le première occurrence du mot "Paul" si il est présent
    dans la phrase
phrase.clear();  // vide la phrase
```

1. http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

3 Accès

- `Element get(int i)`
  Cette méthode retourne l'objet contenu dans la liste au rang `i`. Cette opération est assurée en temps constant 0(1). Aucune modification n'est apportée à la liste.

- `Element set(int i, Element element)`
  Cette méthode remplace la valeur de la liste au rang `i` par l'objet `element`. Cette méthode retourne également l'objet présent au rang `i` de la liste avant la modification. Cette opération est réalisée en temps constant $O(1)$.

Exemple :

```
String mot = phrase.get(2);  // lecture du troisième mot de la phrase
phrase.set(2, "Jean");  // le troisième mot est remplacé par "Jean"
```

4 Liste vide et taille de la liste

- `boolean isEmpty()` et `int size()` Ces méthodes permettent de savoir si la liste et vide et quelle est sa taille.

Exemple :

```
if (phrase.isEmpty()) {
    System.out.println("La phrase est vide.");
}

int nombreDeMots = phrase.size();
```

5 Affichage (méthode `toString`)

La classe `ArrayList` dispose de la méthode `toString` qui affiche les éléments de la liste dans l'ordre (de `0` à `size()-1`). La méthode `toString` doit être implantée dans la classe `Element` pour pouvoir utiliser la méthode `toString` au niveau de la liste.
Exemple :

```
System.out.println(phrase);  // affiche tous les mots de la phrase
```

6 Recherche

- `boolean contains(Element element)`
  Cette méthode retourne `true` si l'object `element` appartient à la liste et `false` sinon.

- `int indexOf(Element element)`
  Cette méthode retourne le rang de la première instance de l'objet `element` dans l'`ArrayList` sur laquelle la méthode est appelée. Le retour vaut `-1` lorsque l'objet `element` n'appartient pas à la liste.

Exemple :

```
if (phrase.contains("Paul")) {
    System.out.println("La phrase contient Paul en position " +phrase.indexOf("Paul"));
}
```

Remarque

Pour utiliser ces deux méthodes, la classe `Element` doit contenir la méthode `boolean equals(Object obj)`.

III Algorithmes génériques sur les listes

Des méthodes statiques sont disponibles au niveau de la classe `Collections` qui englobe différents algorithmes comme le tri par ordre croissant ou décroissant, les valeurs minimales et maximales, etc. Voici quelques méthodes qui manipulent une `ArrayList` :

- `Collections.sort(ArrayList<Element> element)`
  Cette méthode trie les éléments de la liste dans l'ordre croissant (la relation d'ordre étant définie par la méthode `compareTo` de la classe `Element`). La complexité du tri utilisé ici est $O(n \log n)$.
- `Collections.sort(ArrayList<Element> element, Collections.reverseOrder())`
  Cette méthode trie la liste `element` dans l'ordre décroissant. La complexité est la même que précédemment.
- `Collections.shuffle(ArrayList<Element> element)`
  Cette méthode mélange la liste `element` dans un ordre aléatoire.
- `Element Collections.min(ArrayList<Element> element)`
  Cette méthode retourne le plus petit élément de la liste au sens défini par la méthode `compareTo()` de la classe `Element`.
- `Element Collections.max(ArrayList<Element> element)`
  Cette méthode retourne le plus grand élément de la liste au sens défini par la méthode `compareTo()` de la classe `Element`.

Exemple :

```
Collections.sort(phrase);  // range les mots de la phrase par ordre alphabétique
String motMax = Collections.max(phrase); // renvoie le mot le plus "grand"
```

Remarque

En plus de définir la méthode `int compareTo(Object obj)`, la classe `Element` doit implanter l'interface `Comparable` :

```
public class Element implements Comparable {

  ...

}
```

IV Parcours d'une `ArrayList`

Il existe de nombreuses manières de parcourir une `ArrayList`. On peut naturellement utiliser une boucle *pour* si les bornes sont connues et l'incrémentation constante.

```
for (int i = 0; i < phrase.size(); i++) {
    System.out.println("Mot␣" + i + "␣=␣" + phrase.get(i));
}
```

Il également possible d'utiliser une boucle *pour chaque* si l'on souhaite parcourir un à un tous les éléments de la liste.

```
for (String mot : phrase) {
    System.out.println(mot);
}
```

Enfin, les structures d'itération `tant que` doivent être utilisées dans les autres cas.

```
int i = 0;
while (i < phrase.size()) {
    System.out.println("Mot " + i + " = " + phrase.get(i));
    i++;
}
```

V Exemple synthétique

Cet exemple consiste en la fabrication et la manipulation d'une liste de 10 entiers. Les éléments sont ajoutés, accédés, supprimés, affichés, mélangés, triés, modifiés, etc.

Remarque

Une `ArrayList` ne peut contenir des types primitifs en tant que tels. Par contre, tous les types primitifs possèdent une classe *enveloppe* qui les englobe. Par exemple, les entiers `int` peuvent également être définis à l'aide de la classe `Integer` dans laquelle nous retrouvons entre autres les méthodes `toString`, `equals` et `compareTo`.

Note : il est possible de passer d'un `int` à un objet `Integer` et inversement de manière naturelle depuis Java 5 (*boxing* et *unboxing*) :

```
int i = 10 ;
Integer objectI = i ;
int j = objectI ;
```

Le code suivant :

```
ArrayList<Integer> listeEntiers = new ArrayList<Integer>();
// ajout de 10 entiers Integer à la liste
for (int i = 0; i < 10; i++) {
    listeEntiers.add(i + 1);
}
System.out.println("La taille de l'ArrayList = " + listeEntiers.size());
// affichage grace à la get
System.out.print("listeEntiers = ");
for (Integer unEntier : listeEntiers) {
    System.out.print(unEntier + " ");
}
System.out.println();
// suppression d'éléments de la liste
System.out.println("Suppression des éléments de rang 0 puis 4");
listeEntiers.remove(0);
listeEntiers.remove(4);
System.out.println("Suppression de l'élément de rang 6 qui est :"
        + listeEntiers.remove(3));
// affichage grace a la méthode toString de ArrayList et de Integer
System.out.println("listeEntiers = " + listeEntiers);
// ajout d'elements a un certain rang
listeEntiers.add(5, 50);
listeEntiers.add(5, 75);
System.out.println("listeEntiers = " + listeEntiers);
// tests d'appartenance
System.out.println("6 est-il dans la liste ? "
        + listeEntiers.contains(6));
System.out.println("8 est au rang "
        + listeEntiers.indexOf(8)
        + " de la liste");
// max
System.out.println("la plus grande valeur de la liste est = "
        + Collections.max(listeEntiers));
// mélange et tri
Collections.shuffle(listeEntiers);
System.out.println("listeEntiers après le mélange = " + listeEntiers);
Collections.sort(listeEntiers);
System.out.println("listeEntiers après le tri     = " + listeEntiers);
// modification
```

```
System.out.println("Remplacement␣de␣"
        + listeEntiers.set(2, 100)
        + "␣par␣au␣rang␣2␣par␣la␣valeur␣" + listeEntiers.get(2));
System.out.println("listeEntiers␣après␣le␣remplacement␣=␣" + listeEntiers);
```

produit le résultat :

```
run:
La taille de l'ArrayList␣=␣10
listeEntiers␣=␣1␣2␣3␣4␣5␣6␣7␣8␣9␣10
Suppression␣des␣éléments␣de␣rang␣0␣puis␣4
Suppression␣de␣l'élément de rang 6 qui est : 5
listeEntiers = [2, 3, 4, 7, 8, 9, 10]
listeEntiers = [2, 3, 4, 7, 8, 75, 50, 9, 10]
6 est-il dans la liste ? false
8 est au rang 4 de la liste
la plus grande valeur de la liste est = 75
listeEntiers après le mélange = [10, 50, 9, 3, 4, 7, 2, 8, 75]
listeEntiers après le tri     = [2, 3, 4, 7, 8, 9, 10, 50, 75]
Remplacement de 4 par au rang 2 par la valeur 100
listeEntiers après le remplacement = [2, 3, 100, 7, 8, 9, 10, 50, 75]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Travaux dirigés

Exercice 1

On souhaite gérer le classement de joueurs dans un jeu en ligne. Chaque joueur est caractérisé par son nom, son prénom et son score. On suppose qu'une classe `Joueur` est déjà implantée (`int getScore()` existe dans cette classe). Un classement est défini par une liste de joueurs et par le nom du jeu.

1. *Représenter la relation entre les classes `Classement` et `Joueur` sous la forme d'un diagramme UML.*
2. *Définir la classe `Classement` avec un constructeur créant un classement vide et ayant un nom de jeu donné.*
3. *Ajouter une méthode permettant d'ajouter un joueur en première position dans la liste du classement.*
4. *Ajouter une méthode permettant de calculer le score moyen des joueurs.*
5. *Ajouter une méthode permettant de savoir si un joueur appartient au classement.*
6. *Ajouter une méthode permettant de connaître le score du meilleur joueur.*
7. *Ajouter une méthode permettant de ranger les joueurs du classement du plus petit au plus grand score.*

Exercice 2

Pour l'exercice suivant, nous proposons de définir des outils permettant d'exploiter un altimètre/-podomètre porté au poignet qui est utilisé par exemple par les randonneurs en montagne ou les amateurs de trails. La fonction altimètre permet de mesurer l'altitude. La fonction podomètre permet de mesurer la distance parcourue.

Nous nous intéressons tout d'abord à la notion de mesure. Une mesure prise à un instant donné est définie par trois informations : une altitude (en mètre), la distance relative parcourue depuis la mesure précédente (en mètre) et l'heure à laquelle a été effectuée la mesure (en seconde). L'altitude est un nombre entier. Les deux autres informations sont des réels.

Les mesures sont enregistrées au cours de la randonnée pour être traitées par la suite. La classe `Mesure` est définie comme suit :

```
package Podometre;

public class Mesure {
  private int altitude;
  private double distance;
  private double heure;

  public Mesure(int altitude, double distance, double heure) {
    this.altitude = altitude;
    this.distance = distance;
    this.heure = heure;
  }
  public int getAltitude() {
    return altitude;
  }
  public double getHeure() {
    return heure;
  }
```

```
  public double getDistance() {
    return distance;
  }
  public void setDistance(double distance) {
    this.distance = distance;
  }
  public Mesure clone(){
    return new Mesure(this.altitude,this.distance,this.heure);
  }
  public String toString() {
    return "Mesure{" + "altitude=" + altitude + ",␣distance="
      + distance + ",␣heure=" + heure + '}';
  }
}
```

Lorsque l'utilisateur le souhaite, il peut effectuer un parcours en mémorisant dans l'altimètre/podomètre les mesures associées à ce parcours. La fonction mémorisation est déclenchée puis arrêtée en appuyant sur un bouton. Les mesures effectuées entre le déclenchement et l'arrêt de la mémorisation sont stockées dans une `ArrayList` de mesures.

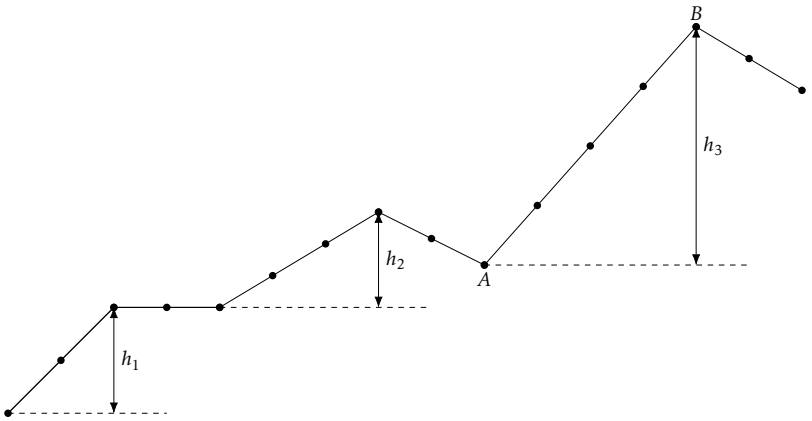


FIGURE 4.1 – Dénivelée cumulée positive : $h1 + h2 + h3$

➽ *Définir la classe `Parcours`. Cette classe doit contenir :*

1. *un constructeur construisant un parcours vide ;*
2. *une méthode fournissant la durée en seconde du parcours ;*
3. *une méthode fournissant l'altitude maximale du parcours ;*
4. *une méthode permettant de savoir si le parcours dépasse strictement une altitude donnée ;*
5. *une méthode fournissant la dénivelée cumulée positive du parcours (cf figure 4.1) ;*
6. *une méthode permettant de savoir si la vitesse entre deux mesures consécutives du parcours devient inférieure à une vitesse donnée.*
7. *une méthode qui supprime les mesures situées dans une portion plate du parcours afin de ne conserver que deux mesures de la dite portion ;*
8. *une méthode qui retourne sous la forme d'un parcours la dernière portion montante du parcours courant constitué d'au moins 2 mesures (portion située entre les mesures A et B sur la figure 4.1). La distance parcourue au démarrage de la portion montante retournée (mesure A sur la figure) doit être mise à zéro ;*
9. *une méthode donnant la plus grande dénivelée positive du parcours courant (dénivelée h3 sur la figure 4.1).*

Exercice 3

Un graphe est une structure formée par un ensemble de points, ou sommets, et par un ensemble d'arêtes qui relient certains sommets entre eux. Cette structure permet par exemple de représenter un réseau routier. Nous pouvons avoir une vue très macroscopique de ce réseau en imaginant que les sommets sont des villes et les arêtes les routes qui permettent de relier les villes voisines. Toutes les villes ne sont reliées les une aux autres par une route directe, d'où la notion de parcours dans un graphe, c'est à dire la liste des sommets pour passer d'un sommet à un autre. Étant donné un tel parcours, nous souhaitons supprimer les cycles sur ce parcours afin de réduire la longueur du chemin.
Nous proposons de maintenir un parcours dans une structure dynamique comme une `ArrayList` afin de profiter de sa plasticité. Pour simplifier l'exercice, nous codons les sommets par un numéro entier. Soit les questions suivantes :

1 *Écrire la classe `ParcoursGraphe` qui possède au minimum un attribut `parcours` de classe `ArrayList`. Écrire les méthodes `ajoutSommet(int s)`, `ajoutSommet(int index, int s)`, `remplacerSommet(int index, int s)` qui permettent respectivement d'ajouter le sommet `s` à la fin du parcours ou au rang `index`, et de remplacer le sommet au rang `index` dans le parcours par le sommet `s`. On rappelle qu'un type primitif ne peut apparaître dans une `ArrayList`, d'où l'obligation de passer par la classe `Integer` pour stocker des entiers dans une telle structure de données :*

```
ArrayList<Integer> parcours = new ArrayList<Integer>();
```

2 *Écrire les méthodes `int depart()` et `int arrivee()` qui retournent respectivement le numéro du sommet de départ et d'arrivée du parcours.*

3 *Écrire la méthode `estUneBoucle()` `ClasseParcours` qui répond à la question : le parcours est-il une boucle ?*

4 *Écrire la méthode `int occurrenceSuivante(int index, int s)` qui donne le rang de la prochaine occurrence d'un sommet `s` dans le sens du parcours à partir du rang `index`. La méthode retourne la valeur `-1` si aucune occurrence n'a pu être trouvée entre le rang `index` inclus et la fin de la liste.*

5 *Écrire la méthode `supprimeIntervalle(int index1, int index2)` qui supprime tous les sommets du parcours entre les rangs `index1` et `index2` inclus.*

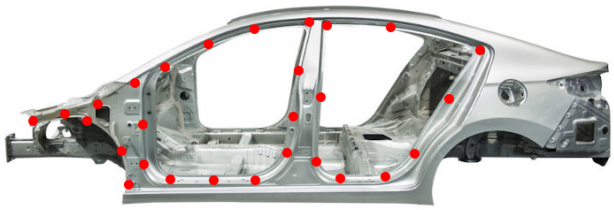
6 *Écrire la méthode `supprimeCycles()` qui élimine les cycles du parcours courant. Cela consiste à supprimer toutes sous-suites de numéros de sommets du parcours qui commence et termine par un même sommet. Pour cela, nous utilisons certaines des méthodes précédemment écrites ici.*

Travaux pratiques

L'objectif du TP est l'illustration des `ArrayList` à travers le problème ultra connu en algorithmique du Voyageur de Commerce. Ce problème est très connu car il possible de l'expliquer très facilement alors qu'il est très difficile à résoudre dans sa forme optimale.

Compétences

- construire et utiliser une `ArrayList` (`add`, `remove`, `size`, `contains`, etc.) ;
- utiliser les méthodes de la classe `Collections` (`sort`, `min`, `max`).
- définir une classe contenant une liste d'objets ;
- écrire des algorithmes spécifiques de gestion d'une liste d'objets ;
- écrire quelques heuristiques simples sur le problème du voyageur de commerce.



La soudure par point représente une part très importante de l'assemblage d'une caisse automobile. Équipé d'une pince à souder, le robot doit parcourir une liste de points sur le châssis. L'ordre de parcours de ces points est primordial pour le temps de cycle. Si il est mal choisi, le robot perdra beaucoup de temps à se déplacer d'un point à un autre.

Ce problème est équivalent au voyageur de commerce, qui consiste à déterminer, étant donné une liste de villes et les distances entre toutes les paires de villes, le plus court circuit qui passe par chaque ville une et une seule fois.

Ce problème est plus compliqué qu'il n'y paraît. Pour un ensemble de n villes, il existe au total $n!$ chemins possibles. On ne connaît pas de méthode de résolution permettant d'obtenir des solutions exactes en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances, on doit donc souvent se contenter de solutions approchées, car on se retrouve face à une explosion combinatoire.

L'objectif de ce TP est d'apprendre l'utilisation de la classe `ArrayList` en Java en réalisant quelques méthodes heuristiques permettant de trouver rapidement un chemin pas trop long !

VI Représentation d'un chemin

Pour simplifier, on traite le problème du voyageur de commerce dans le plan euclidien. Le but est de définir un ordre de parcours de n points 2D en minimisant la somme des distances euclidiennes entre chaque points.

Pour gagner du temps, une classe `Point` similaire à celle que vous avez codée précédemment est fournie. On définit alors un chemin comme une liste de points. L'ordre des points dans la liste définit le sens du parcours.

1 *Créer une classe* `Chemin` *dans le package* `chemin` *à l'aide de la commande* `File->New File->Java Class` *puis définir ses attributs et les méthodes suivantes :*

   a. *un constructeur qui définit un chemin vide ;*
   b. *une méthode permettant d'ajouter un point à la fin d'un chemin ;*
   c. *une méthode qui retourne la longueur totale du chemin ;*
   d. *une méthode permettant de savoir si le chemin contient un point donné ;*
   e. *une méthode qui mélange aléatoirement le chemin à l'aide de la commande* `Collections.shuffle( ... );`
   f. *une méthode* `toString` *avec tous les points du chemin.*

2 *Créer une classe* `TestChemin` *dans le package* `chemin` *à l'aide de la commande* `File->New File->Java Main Class` *puis construire et afficher un chemin comportant une dizaine de points dont on aura mélangé l'ordre juste avant.*

3 *Ajouter une méthode permettant de dessiner le chemin dans un contexte graphique 2D. Tester.*

VII Méthode du balayage angulaire

Dans le cas de points 2D, une méthode heuristique simple consiste à parcourir les points en fonction de leur angle en coordonnées polaires. Autrement dit, la méthode consiste à :

- translater les points pour que le barycentre des points corresponde au centre du repère ;
- ranger les points par ordre croissant de leurs angles en coordonnées polaires (ρ, θ) ;
- translater les points pour les remettre à leurs places initiales.

1 *Ajouter à la classe* `Chemin` *les méthodes suivantes :*

   a. *une méthode qui retourne le barycentre des points du chemin ;*
   b. *une méthode permettant de translater les points du chemin selon les coordonnées d'un point 2D donné illustrant le changement du repère du l'origine ;*
   c. *une méthode permettant de ranger les points par ordre croissant de leur angle en coordonnées polaires ;*
   d. *une méthode qui trie les points grâce à la méthode du balayage angulaire au chemin.*

2 *Tester avec les points précédents.*

3 *Tester avec les points du fichier* `TestChassis`.

VIII Méthode du plus proche voisin

Une autre méthode heuristique est de partir d'un point quelconque puis d'aller toujours au point le plus proche sans repasser par un point déjà parcouru. L'algorithme est résumé ci-dessous.

```
Q ← liste de points vide
P ← point quelconque de la liste initiale H
retirer P de la liste H
ajouter P à la liste Q
tant que H n'est pas vide faire
    P ← le point le plus proche de P dans H
    retirer P de la liste H
    ajouter P à la liste Q
fintantque
H ← Q
```

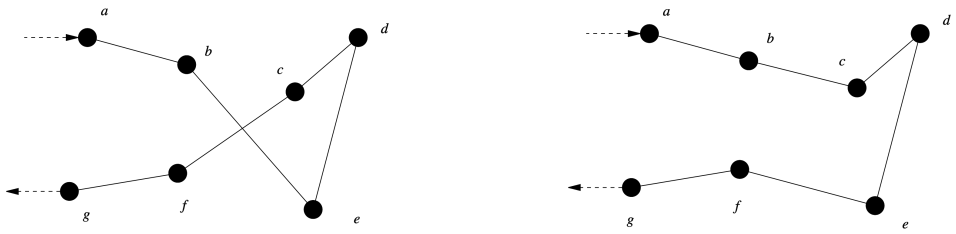
1 *Ajouter à la classe* `Chemin` *les méthodes suivantes :*

   a. *une méthode qui retourne l'indice du point du chemin le plus proche d'un point donné ;*
   b. *une méthode qui applique la méthode du plus proche voisin au chemin.*

2 *Tester.*

IX Méthode itérative 2-opt

Une heuristique classique, appelée 2-opt est une recherche locale qui consiste à partir d'une solution et à essayer de l'améliorer en échangeant itérativement les sommets de deux arêtes. Dans le cas du problème du voyageur de commerce géométrique, la permutation consiste généralement à remplacer les arêtes par leurs diagonales comme illustré ci-dessous.




```
amélioration ← vrai
tant que amélioration = vrai faire
    amélioration ← faux
    pour tout point Pi de H faire
        pour tout point Pj de H, avec j strictement supérieur à i+1 faire
            si distance(Pi, Pi+1) + distance(Pj, Pj+1) > distance(Pi, Pj) +
                distance(Pi+1, Pj+1) alors
                inverser l'ordre des points dans H entre les indices (i+1) et j
                    inclus
                amélioration ← vrai
            finsi
        finpour
    finpour
fintantque
```

1 *Ajouter à la classe* `Chemin` *une méthode qui applique la recherche locale 2-opt au chemin.*

2 *Tester.*

Lecture et écriture de fichiers textes

Compétences

- Lire et écrire des données d'un fichier texte structuré (csv)
- Utiliser ces données dans une classe contenant une collection d'objets (méthodes `lireFichier` et `ecrireFichier`)

I Fichiers textes

Un fichier informatique est une séquence d'octets enregistrée sur un support de stockage permanent comme un disque dur, une clé USB, ou une carte SD et désignée par un nom suivi généralement d'une extension (`.txt`, `.jpg`, `.exe`, etc.). Un fichier permet d'archiver des programmes et des données (textes, images, etc.).

Un fichier *texte* est un fichier dont le contenu représente uniquement des caractères. Les caractères considérés sont généralement les lettres, les chiffres, l'espace et des caractères spéciaux comme le retour à la ligne. La correspondance entre les octets du fichier et les caractères est décrite par un codage comme l'illustre la table ci-dessous.

Il existe de nombreux codages en fonction des systèmes d'exploitation et des pays. Pour la lecture et l'écriture de fichier, les méthodes que nous allons présenter utilisent le codage par défaut de l'ordinateur. Pour ce TDAO, nous utiliserons donc le codage *Windows-1252*, appelé aussi *latin 1*.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	
80	€	▫	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	▫	Ž	▫
90	▫	‘	’	“	”	•	–	—	˜	™	š	›	œ	▫	ž	Ÿ
A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B0	°	±	2	3	´	µ	¶	·	¸	1	º	»	¼	½	¾	¿
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Codage Windows-1252 (latin 1)

II Lecture d'un fichier texte

Dans cette partie, nous nous intéressons à la lecture d'un fichier texte, c'est-à-dire à l'extraction d'informations contenues dans un fichier. Dans ce mode d'accès, le fichier n'est pas modifié.

En Java, la lecture d'un fichier texte s'effectue très simplement à l'aide des classes `FileReader` et `BufferedReader`. La première étape consiste à ouvrir le fichier en lecture et à créer un buffer de lecture associé. L'exemple ci-dessous illustre comment ouvrir le fichier `test.txt`.

```
BufferedReader fichier = new BufferedReader(new FileReader("test.txt"));
```

Le fichier doit se trouver à la racine du répertoire du projet NetBeans, sinon il faut préciser le chemin complet (par exemple `"Z:\\Documents\\test.txt"`). On peut ensuite lire le fichier ligne par ligne à l'aide de la méthode `readLine`.

```
String ligne;
ligne = fichier.readLine();
```

Remarque

A chaque appel de la méthode `readLine`, le buffer avance d'une ligne dans le fichier. Ce mode de lecture est dit *séquentiel*. La méthode `ready` permet de savoir si il reste au moins une ligne à lire ou si la fin du fichier a été atteinte.

Une fois la lecture des informations effectuée, il faut libérer l'accès au fichier en le fermant à l'aide de la méthode `close`. En résumé, le code suivant permet la lecture intégrale d'un fichier texte de la première à la dernière ligne.

```
try {
    BufferedReader fichier = new BufferedReader(new FileReader("test.txt"));
    while (fichier.ready()) {
         String ligne;
         ligne = fichier.readLine();
         ... // Utilisation de la ligne lue
    }
    fichier.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

On remarquera la présence d'un bloc `try/catch` indispensable ici. Cette syntaxe permet de récupérer les erreurs pouvant survenir dans le bloc `try` et de les gérer dans le bloc `catch`. La déclaration d'une erreur (*levée d'une exception*) est réalisée par l'instruction `throw` qui interrompt la séquence du `try` pour exécuter celle du `catch` avec l'exception passée en argument.

Dans le cas de la lecture d'un fichier, une exception peut être levée si le fichier n'existe pas, si le fichier est corrompu (lecture impossible des caractères) ou si support de stockage est retiré pendant la lecture.

1 *Dans la classe `TestLecture`, écrire un programme permettant d'afficher le contenu du fichier texte `poeme.txt` dans la console*

III Écriture d'un fichier texte

Nous nous intéressons cette fois à l'écriture de données dans un fichier texte. En Java, l'écriture dans un fichier texte est réalisée par l'intermédiaire de la classe `FileWriter`. Comme pour la lecture, il faut commencer par ouvrir le fichier.

```
FileWriter fichier = new FileWriter("test.txt");
```

Remarque

Si le fichier `text.txt` n'existe pas dans le répertoire du projet courant, un fichier `text.txt` vide sera créé. Si un fichier `text.txt` est disponible dans le répertoire du projet, ce fichier sera totalement vidé.

Une fois le fichier ouvert en écriture, il est possible d'y écrire des chaînes de caractères à l'aide de la méthode `write` comme si il s'agissait de la console.

```
fichier.write("Hello world!");
fichier.write("Hello world!" + System.getProperty("line.separator"));
```

La méthode `System.getProperty("line.separator")` renvoie le ou les caractères spéciaux permettant un retour à la ligne dans le codage par défaut du système d'exploitation.

Comme pour la lecture, il faut fermer le fichier avec la méthode `close` quand toutes les données ont été écrites. L'exemple suivant crée le fichier `test.txt` contenant trois fois *hello world !*

```
try {
    FileWriter fichier = new FileWriter("test.txt");
    fichier.write("Hello world!");
    fichier.write("Hello world!" + System.getProperty("line.separator"));
    fichier.write("Hello world!" + System.getProperty("line.separator"));
    fichier.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

1 *Dans la classe `TestEcriture1`, écrire un programme permettant de générer le fichier `fibonacci.txt` contenant en colonne la suite des 50 premiers termes de la suite de Fibonacci : 1, 1, 2, 3, 5, 8, 13, 21, ...*

2 *Dans la classe `TestEcriture2`, écrire un programme qui à partir du fichier `poeme.txt` génère un nouveau fichier `poeme2.txt` avec le même contenu mis en caractères majuscules. On rappelle que la classe `String` dispose d'une méthode `toUpperCase` qui renvoie la chaîne en caractères majuscules.*

IV Fichiers textes à champs délimités (fichiers CSV)

Le contenu d'un fichier texte peut être organisé selon certaines structures appelées *formats*. Il existe des dizaines de formats différents en fonction des données à stocker.

L'un des formats les plus simples est sans doute le CSV (Comma-Separated Values). Les fichiers CSV sont dits délimités : un caractère spécial, appelé caractère de délimitation, permet de repérer quand finit un champ de donnée et quand commence le champ suivant. En français, le caractère de délimitation est généralement le point-virgule, alors que les anglo-saxons utilisent la virgule. Ce caractère de délimitation est naturellement proscrit dans les champs de donnée, faute de quoi la structure est illisible.

1 *Ouvrir le fichier* `stock.xlsx` *avec Excel et générer un fichier* `stock.csv` *en choisissant le format* `CSV (séparateur : point-virgule)`*. Ce fichier doit être enregistré dans le répertoire racine du projet (au même emplacement que* `stock.xlsx`*). Visualiser ensuite le contenu du fichier CSV avec Notepad++* (via *un clic droit sur le fichier).*

Pour lire les fichiers CSV en Java, il suffit d'utiliser l'algorithme de lecture séquentiel (ligne par ligne) d'un fichier texte. A chaque ligne lue, on utilise la méthode `split` de la classe `String` pour diviser cette ligne en un tableau de chaînes de caractères contenant les données dans le même ordre.

```
String ligne = "torchons;serviettes";
String champs[] = ligne.split(";");
System.out.println(champs[0] + "␣␣␣␣" + champs[1]);
```

Les champs récupérés sont des chaînes de caractères qui peuvent contenir des informations de différentes natures (entiers, réels, booléen, etc.). Pour convertir une chaîne de caractère dans un autre type, on utilise les *parsers* (analyseurs) fournis dans les classes enveloppes.

```
int n = Integer.parseInt("421");          // n reçoit 421
double x = Double.parseDouble("3.1415");  // x reçoit 3.1415
boolean b = Boolean.parseBoolean("true"); // b reçoit true
```

Enfin, pour les nombres réels, il peut être nécessaire de remplacer les virgules par des points. La méthode `replace` permet cela très facilement.

```
String nombreAvecVirgule = "3,1415";
double x = Double.parseDouble(nombreAvecVirgule.replace(",","."));
```

Pour cet exercice, nous allons étendre la classe `Stock` définie précédemment (vous pouvez au choix reprendre votre fichier `Stock.java` ou utiliser la version fournie dans le projet).

2 *Ajouter une méthode* `void lireFichier(String nomDuFichier)` *à la classe* `Stock` *permettant d'ajouter au stock les articles contenus dans un fichier CSV.*

3 *Tester la lecture et l'affichage du fichier* `stock.csv`*.*

4 *Vérifier que la valeur totale du stock est exacte.*

5 *Ajouter une méthode* `void ecrireFichier(String nomDuFichier)` *à la classe* `Stock` *permettant de générer un fichier CSV correspondant au stock ;*

6 *Générer le fichier* `reapprovisionnement.csv` *contenant les articles dont la quantité est inférieure à deux unités au sens large dans le stock en utilisant la méthode* `articlesACommander()`*.*

V Fichiers textes à champs de largeurs fixes

Un autre format courant consiste à structurer le fichier avec des champs de tailles fixes. Les données sont alors alignées en colonnes.

1 *Ouvrir le fichier* `releve.txt` *avec Notepad++ et étudier sa structure.*

Ce fichier a été automatiquement généré par une station météo. Le préambule contient des informations sur le lieu, l'année et l'intitulé des colonnes. Chaque ligne du préambule commence par le caractère # pour signifier qu'il ne s'agit pas des données.
Pour savoir si une chaîne de caractère commence par un caractère particulier, on peut écrire :

```
if (ligne.startsWith("#")) {
    ...
}
```

La suite du fichier contient les données en colonnes de largeurs fixes. Pour extraire une partie d'une chaîne de caractères, on peut utiliser la méthode `substring` de la classe `String` comme l'illustre l'exemple suivant :

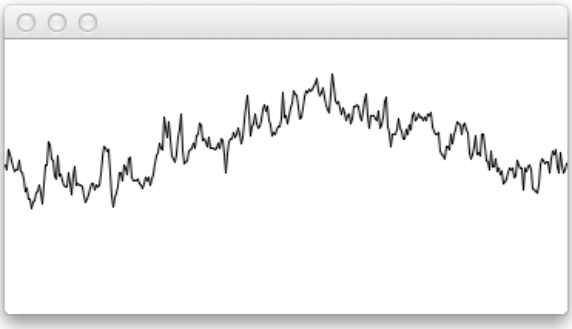
```
String ligne = "torchons␣␣serviettes";
String champ0 = ligne.substring(0,9);
String champ1 = ligne.substring(10,19);
System.out.println(champ0 + "␣␣␣␣" + champ1);
```

Pour cet exercice, nous allons étendre la classe ` releveMeteo` définie en TD et dont une implantation est proposée dans le projet.

2 *Ajouter une méthode* `void lireFichier(String nomDuFichier)` *à la classe* `releveMeteo` *permettant de lire la ville, l'année et les températures moyennes journalières contenues dans le fichier issu d'une station météo.*

3 *Tester la lecture et l'affichage du fichier* `releve.txt`.

4 *En s'inspirant du programme principal de la classe* `TestFenetreGraphique`*, ajouter une méthode permettant d'afficher le relevé météo sous forme d'un graphique 2D comme illustré ci-dessous.*



Les énumérations

Pour modéliser le fait qu'une variable ne prendre qu'un nombre fini de valeurs explicitement définies, il est recommandé d'utiliser une énumération.

I Définition

En Java, une énumération se déclare de la même façon qu'une classe. Par exemple, pour modéliser les jours de la semaine, on peut écrire le code suivant dans un fichier `Jour.java` :

```
package Exemples;

import java.util.EnumSet;

/** Définition de l'énumération Jour */
public enum Jour {

    LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE;

}
```

Il est d'usage de mettre les constantes en majuscules. Une énumération se représente par le diagramme UML ci-dessous.

Figure 6.1 – Modèle de la classe énumération `Jour` en UML

Exercice 1

➽ *Définir une classe `Direction` modélisant les quatre point cardinaux. On pourra pour cela utiliser la commande `New file` dans le menu déroulant `File` et choisir `Java enum` dans la fenêtre de dialogue.*

II Utilisation

Une énumération s'utilise comme un type de base :

```
Jour unJour = Jour.JEUDI;
System.out.println(unJour);

if (unJour == Jour.JEUDI) {
    System.out.println("Aujourd'hui, c'est jeudi");
}
```

Étant une classe, une énumération possède quelques méthodes. La méthode `ordinal` renvoie le numéro d'ordre d'une valeur énumérée (l'ordre est numéroté à partir de 0) :

```
Jour unJour = Jour.JEUDI;
int numJour = unJour.ordinal();
// numJour reçoit 3
```

La méthode statique `values` retourne un tableau contenant toutes les valeurs énumérées disponibles, ce qui permet par exemple de parcourir successivement toutes les valeurs d'une classe énumération :

```
Jour tableauJours[] = Jour.values();
for (int i=0; i<tableauJours.length;i++) {
    System.out.println("Jour " + i + " =" + tableauJours[i]);
}
```

Exercice 2

➽ *Tester les exemples ci-dessus sur la classe* `Direction`.

A l'aide de ces méthodes, l'on peut ajouter une méthode très pratique dans la classe énumération `Jour` :

```
public enum Jour {

    LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE;

    Jour suivant() {
        Jour tableauJours[] = Jour.values();
        return tableauJours[(this.ordinal() + 1)%tableauJours.length];
    }
}
```

Ce qui permet d'écrire :

```
Jour unJour = Jour.DIMANCHE;
System.out.println(unJour.suivant()); // affiche LUNDI
```

Exercice 3

➽ *Ajouter les méthodes* `quartDeTourHoraire` *et* `quartDeTourAntihoraire` *à la classe* `Direction`. *Tester.*

III Ensemble de valeurs énumérées

La classe générique `EnumSet` permet de gérer facilement des ensembles de valeurs énumérées. L'interface de `EnumSet` est similaire à celle d'`ArrayList` proposant notamment les méthodes `size`, `contains`, `add` et `remove`. Contrairement à `ArrayList`, `EnumSet` comme la classe `Set` modélisent des ensembles et n'autorisent pas de stocker deux fois le même élément (autrement dit un même élément appartient ou n'appartient pas à l'ensemble mais ne peut pas apparaître plusieurs fois).

L'exemple suivant crée un objet `ensembleDeJours` qui contient tous les jours de la semaine sauf le lundi.

```
EnumSet<Jour> ensembleDeJours = EnumSet.allOf(Jour.class) ;
ensembleDeJours.remove(Jour.LUNDI);
System.out.println(ensembleDeJours);
```

Exercice 4

➽ *Créer un `EnumSet<Direction>` vide puis tester l'ajout et l'appartenance de directions à cet ensemble.*

Les files et les piles

Les *files* et les *piles* sont des structures de données dynamiques dont l'utilisation est fréquente dans certains algorithmes faisant intervenir des priorités comme les algorithmes de parcours dans les graphes ou les algorithmes d'ordonnancement. Cette leçon présente les deux types de structure qui se différencient par la manière d'ajouter/supprimer des données. Les files obéissent au paradigme FIFO (First In First Out) ou en français premier entré, premier sorti alors que les piles obéissent au paradigme LIFO (Last In First Out) c'est-à-dire dernier entré, premier sorti.

I Définitions

1 Les Files

Une *file* est une structure de données dynamique et linéaire telle que les ajouts se font en fin de file et les suppressions, avec récupération des données supprimées, se font en début de file. On est bien dans un schéma FIFO car une fois ajoutée, une donnée ne sera retirée qu'après que toutes celles qui ont été ajoutées avant elle. La figure 7.1 illustre une file qui contient 9 entiers. L'entier qui sortira à la prochaine opération de d'extraction est le nombre "`3`" et le dernier entier ajouté dans la file est le nombre "`6`" en fin de file.

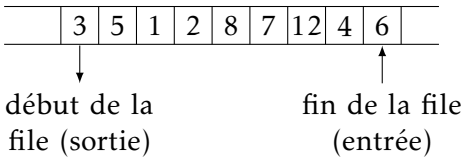


Figure 7.1 – Exemple de file.

Les 4 opérations utiles à la gestion d'une file sont les suivantes :

- l'opération qui interroge si la file est vide (`isEmpty`);
- l'opération qui retourne la taille de la file (`size`);
- l'opération d'ajout d'une donnée en fin de file (`push`);
- l'opération de suppression d'une donnée en début de file (`pop`) avec retour comme résultat la donnée supprimée;
- l'opération de consultation en lecture de la donnée en début de file, c'est-à-dire en position d'être extraite (`peek`).

2 Les Piles

Une *pile* est une structure de données dynamique et linéaire telle que les ajouts de données se font en sommet de pile (à la fin) et les suppressions, avec récupération de données, se font également au sommet de la pile (en fin). On est dans un schéma LIFO car une fois ajoutée, une donnée sera retirée avant toutes celles qui ont été ajoutées avant elle. C'est toujours les données les plus jeunes qui sont supprimées en premier. La pile d'assiettes est un bon exemple de pile LIFO.

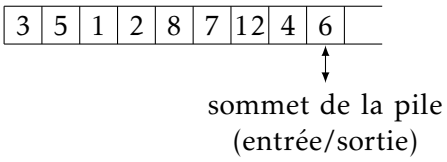


FIGURE 7.2 – Exemple de pile.

Les opérations utiles à la gestion d'une pile sont les suivantes :

- l'opération qui interroge si la pile est vide (`isEmpty`);
- l'opération qui retourne la taille de la pile (`size`);
- l'opération d'ajout d'une donnée au sommet de la pile (`push`), c'est-à-dire à la fin ;
- l'opération de suppression d'une donnée également au sommet de la pile (`pop`), avec comme retour la donnée supprimée ;
- l'opération de consultation en lecture de la donnée au sommet de la pile (`peek`).

II Mise en œuvre en Java

1 Classe `ArrayDeque`

En Java, il existe une structure dynamique linéaire particulièrement adaptée à la mise en œuvre des files et des piles. Il s'agit de la classe `ArrayDeque` qui permet de faire des ajouts/suppressions au début ou à la fin en temps constant, c'est-à-dire `O(1)`. En plus des méthodes dont dispose la classe `ArrayList` (comme `add`, `contains`, `isEmpty` ou `size`), la classe `ArrayDeque` possède des méthodes spécifiques pour réaliser des ajouts et des suppressions en fin ou en début.

```
ArrayDeque<String> phrase = new ArrayDeque<String>() ;

phrase.addLast("world");   // ajout à la fin
phrase.addFirst("Hello");  // ajout au début

String premierMot = phrase.getFirst(); // récupération de "Hello"
String dernierMot = phrase.getLast();  // récupération de "world"

premierMot = phrase.removeFirst(); // récupération et suppression de la "Hello"
dernierMot = phrase.removeLast();  // récupération et suppression de la "world"
```

2 Implantation de la classe `File`

```
import java.util.ArrayDeque;

public class File<Element> {

    private ArrayDeque<Element> file;

    public File() {
        file = new ArrayDeque<>();
    }
    public void push(Element e) {
        this.file.addLast(e);
    }
    public Element pop() {
        return this.file.removeFirst();
    }
    public Element peek() {
        return this.file.getFirst();
    }
    public boolean isEmpty() {
        return this.file.isEmpty();
    }
    public int size() {
        return this.file.size();
    }
    public String toString() {
        return "File␣=␣" + file;
    }
}
```

3 Implantation de la classe `Pile`

```
import java.util.ArrayDeque;

public class Pile<Element> {

    private ArrayDeque<Element> pile;

    public Pile() {
        pile = new ArrayDeque<>();
    }
    public void push(Element e) {
        this.pile.addLast(e);
    }
    public Element pop() {
        return this.pile.removeLast();
    }
    public Element peek() {
        return this.pile.getLast();
    }
    public boolean isEmpty() {
        return this.pile.isEmpty();
    }
    public int size() {
        return this.pile.size();
    }
    public String toString() {
        return "Pile␣=␣" + pile;
    }
}
```

4 Utilisation des classes `File` et `Pile`

Les classes présentées précédemment peuvent être utilisées, sans qu'il soit par ailleurs nécessaire de savoir comment elles sont faîtes comme l'illustrent les exemples suivants :

```
File<String> file = new File<String>() ;

file.push("Hello");
file.push("world");

String mot = file.pop(); // récupération et suppression de "Hello"
```

```
Pile<String> pile = new Pile<String>() ;

pile.push("Hello␣");
pile.push("world");

String mot = pile.pop(); // récupération et suppression de "world"
```

On retiendra que :

- `push(e)` ajoute l'élément `e` à la structure ;
- `pop()` supprime et retourne un élément conformément à son paradigme de fonctionnement FIFO ou LIFO ;
- `peek()` permet de consulter l'élément en position de quitter la structure (donc sans le supprimer).

Récursivité

La récursivité est un moyen élégant de résoudre certains problèmes. La récursivité est naturelle dans de nombreux cas d'application mathématique dont la définition est récurrente. Elle est souvent perçue comme "magique" par les débutants en informatique mais elle facilite souvent l'étude et la résolution de problèmes particuliers.

Le but de cette leçon est de présenter la récursivité, de dégager quelques grands principes de programmation à son sujet, et de parcourir un certain nombre d'exemples.

I Définition

Une fonction récursive est une fonction qui s'appelle elle même. Un exemple très classique issu des mathématiques est la fonction factorielle définie par :

$$\begin{cases} n! = 1 & \text{si } n = 0 \\ n! = n * (n-1)! & \text{sinon} \end{cases}$$

En Java, la programmation d'une telle définition récurrente peut se faire sous la forme suivante d'une méthode récursive :

```
public class Recursive {
  public static int factorielle ( int n ) {
    int resultat;
    if ( n==0 ) {
      resultat = 1;
    } else {
      resultat = n * factorielle ( n-1 );
    }
    return resultat;
  }
}
```

Le mot-clé `static` indique que la méthode est une méthode de classe, qui peut être utilisée directement sans construire d'objet de la classe `Recursive`. Ainsi, le code `System.out.println("4! " + Recursive.factorielle(4));` fournira le résultat suivant :

```
run:
4! = 24
BUILD SUCCESSFUL (total time: 0 seconds)
```

II Quelques principes

Un traitement récursif se fera forcément dans un sous-programme (méthode en java). Quelques grands principes doivent impérativement être respectés. Ils sont listés dans ce qui suit.

Principe 1 : toute définition récursive nécessite une terminaison (point d'arrêt ou point terminal).

La terminaison correspond à au moins un cas pour lequel la définition n'est pas récursive. Dans ce cas, la méthode doit fournir directement la solution au problème sans faire d'appel récursif. La terminaison est indispensable et tout aussi essentiel que la définition récursive elle-même.

Exemple

- factorielle : $n! = n * (n-1)!$ **avec** $0! = 1$
- suite récurrente : $u_n = f(u_{n-1})$ **avec** u_0 **donné**

Principe 2 : tout appel récursif ne peut être fait que si un test préalable de terminaison le précède. Autrement dit, on ne déclenche un appel récursif que dans les cas appropriés. On imagine facilement que l'absence de respect de ce principe générerait des traitements infinis.

On peut en déduire la forme générale suivante de toute fonction récursive :

```
typeRésultat fonctionRecursive(paramètres)
Début
    si ( terminaison )
        traitement non récursif
    sinon
        traitement récursif avec appel à fonctionRecursive(nouveau paramètres)
    finsi
Fin
```

Remarque

- On s'assurera de la terminaison effective de la récursivité lors de l'exécution et donc de sa "convergence". Par exemple, dans le calcul de la factorielle, n décroît à chaque appel et va bien atteindre la valeur 0 et ainsi assurer la terminaison.
- Les déclarations locales de la méthode récursive ne doivent pas être trop encombrantes, car chaque appel récursif va générer ses propres variables locales, d'où un risque de saturation de la mémoire.

Travaux dirigés

1 Quelques calculs

Exercice 1

➽ *Spécifier et écrire des méthodes récursives pour :*

1. *calculer le nombre de chiffres d'un nombre entier.*
2. *calculer la puissance nième d'un nombre entier de façon efficace.*
3. *calculer le plus grand commun diviseur de deux entiers positifs.*
4. *calculer le nombre de carrés, les plus grands possibles, que l'on peut découper dans un rectangle aux dimensions entières.*

Exercice 2

1. *Calculer le nombre de possibilités distinctes qu'un chaton a de monter un escalier de n marches, sachant que le chaton est capable de gravir un, deux ou trois marches d'un coup.*
2. *Donner une idée de la complexité algorithmique de la solution et proposer une piste pour réduire cette complexité.*
3. *Donner une autre solution basée sur la même idée, mais programmée dans avoir recourt à la récursivité.*

Exercice 3

➽ *Trouver un zéro, à ε près, d'une fonction continue f donnée, sachant que $f(a) < 0 < f(b)$ et $a < b$.*

Exercice 4

1. *Calculer le nombre de possibilités de se rendre d'un point A à un point B à Manhattan sans perte de temps.*
2. *Donner une idée de la complexité algorithmique de la solution et proposer une piste pour réduire cette complexité.*
3. *Donner une autre solution basée sur la même idée, mais programmée dans avoir recourt à la récursivité.*

2 Exercices sur des collections

Exercice 5

➽ *Spécifier et écrire des méthodes récursives pour :*

1. *inverser le contenu d'une collection.*
2. *savoir si une collection de caractères est un palindrome (exemple de palindrome : radar, ressasser, élu par cette crapule, ...).*
3. *savoir si un entier appartient à une collection d'entiers rangée par ordre croissant.*

3 Un peu plus dur !

Les exemples de récursivité que nous avons présentés jusqu'à maintenant sont assez facilement acceptés. En effet, la récursivité est terminale, c'est-à-dire que aucun traitement n'est à effectuer après l'appel récursif. Cette propriété entraîne une facile "dérécursivation" et une traduction aisée en itération.
Une récursivité non terminale comporte un ou plusieurs traitements postérieurs à l'appel récursif. Dans ce cas, l'exécution est plus difficile à "dérouler à la main", car la faiblesse de notre mémoire humaine fait oublier les traitements restants à effectuer. Une telle récursivité est moins facile à traduire en itération et nécessite l'emploi d'une structure de pile pour faire office de "mémoire". Pour s'en persuader, il suffit de jouer "à la main" aux tours de Hanoï avec 6 ou 7 disques (voir l'exercice à ce sujet un peu plus loin).

Exercice 6

1. *Afficher à l'écran un entier en notation binaire.*
2. *Calculer le nombre de possibilités d'écrire un nombre entier comme la somme de nombres entiers non nuls (décomposition additive).*

Exercice 7

Flocon de Koch

On suppose définit une classe `Tortue` qui modélise un outil capable de se déplacer et de tracer des lignes dans un environnement graphique.
La javadoc de la classe `Tortue` est donnée à la figure 8.1.

Class Tortue

java.lang.Object
    FenetreGraphique.Tortue

```
public class Tortue
extends java.lang.Object
```

Author:
christophe.varnier

Constructor Summary

Constructors

Constructor and Description
`Tortue(FenetreGraphique fenetre, double x, double y, double angle)`

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
void	`avance(double dist)` fait avancer la tortue d'une distance dist
void	`changeCouleur(java.awt.Color couleur)` change la couleur de tracé de la tortue
void	`tourne(double delta)` fait tourner la tortue dans le sens des aiguilles d'une montre d'un angle delta exprimé en degré

FIGURE 8.1 – Javadoc de la classe `Tortue`

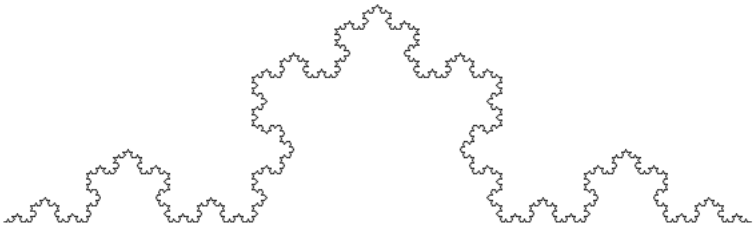


FIGURE 8.2 – Image du flocon de Von Koch

➽ *Spécifier et écrire une méthode récursive permettant dessiner avec une `Tortue` un côté du flocon de Koch (voir la figure 8.2).*

Exercice 8

Tour de Hanoï

Le but est ici de trouver une solution au problème dit des tours de Hanoï. Il s'agit d'un casse-tête qui consiste à partir de 3 tours pouvant accueillir des disques tous de tailles différentes empilés les uns sur les autres. Au début les disques sont rangés sur une seule tour et empilés du plus grand (en bas de la pile) au plus petit 8.3. Tous les disques doivent être déplacés sur une autre tour en respectant les règles suivantes : on ne peut déplacer qu'un seul disque à la fois et l'ordre de taille (grand sous petit) doit toujours être respecté.
On suppose qu'une classe `Jeu` est disponible. La javadoc de cette classe est donnée en figure 8.4.

➽ *Écrire une méthode récursive permettant de résoudre le jeu des tours de Hanoï.*

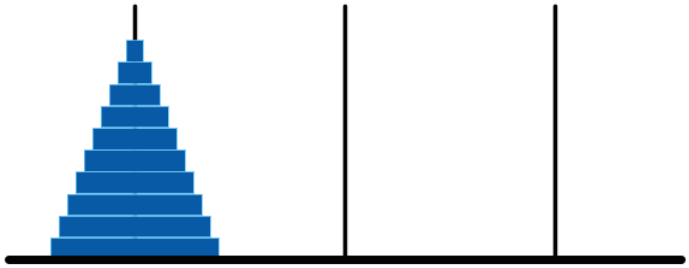


Figure 8.3 – Illustration du jeu des tours de Hanoï

Class Jeu

java.lang.Object
  Recursivite.Jeu

```
public class Jeu
extends java.lang.Object
```

Author:
christophe.varnier

Constructor Summary

Constructors
Constructor and Description
`Jeu(int nb)` construit un jeu avec nb disques rangés sur la tour 0

Method Summary

All Methods	Instance Methods	Concrete Methods

Modifier and Type	Method and Description
void	`deplacelDisque(int depart, int arrivee)` déplace un disque du sommet de la pile départ vers le sommet de la pile arrivee (les piles sont numérotées 0,1 et 2) Suppose la présence d'un disque dans la pile départ
void	`dessine(FenetreGraphique fenetre)` dessine le jeu dans la fenêtre fournie en entrée

Figure 8.4 – Javadoc de la classe `Jeu`

Travaux pratiques

Pour ce TP, un projet *NetBeans* est disponible sur le dossier `RépertoireProfs/Informatique/INFO/TP_Recursivite`.

Exercice 1

1. *Écrire une méthode récursive calculant le plus grand commun diviseur de deux entiers positifs non nuls.*
2. *Écrire une méthode récursive permettant de compter le nombre d'ascendants de rang n d'une abeille d'un sexe donné. Chez les abeilles les mâles sont issus d'une femelle seule, ils n'ont donc qu'un seul ascendant de rang 1. Les femelles sont issues quand à elles d'un mâle et d'une femelle, elles ont donc 2 ascendants de rang 1.*
3. *Écrire une méthode récursive permettant de savoir si une chaîne de caractère est un palindrome (exemple de palindrome : radar, ressasser, élu par cette crapule, ...).*
4. *Écrire une méthode récursive permettant un côté du flocon de Koch à l'aide de la classe `Tortue` (figure 8.1).*
5. *Écrire une méthode récursive permettant dessiner la courbe du dragon 8.6 à l'aide de la classe `Tortue`. Sur le même principe que le flocon de Von Koch, la construction de la courbe du dragon suit le processus de la figure 8.5.*

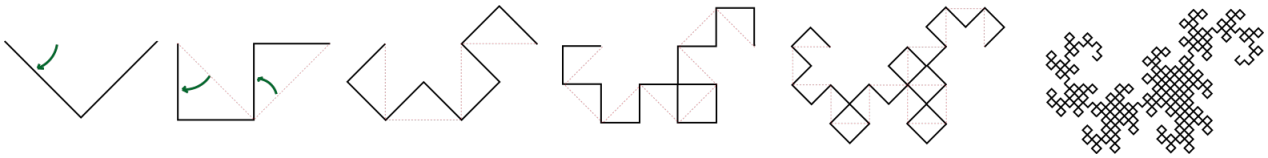


Figure 8.5 – construction de la courbe du dragon

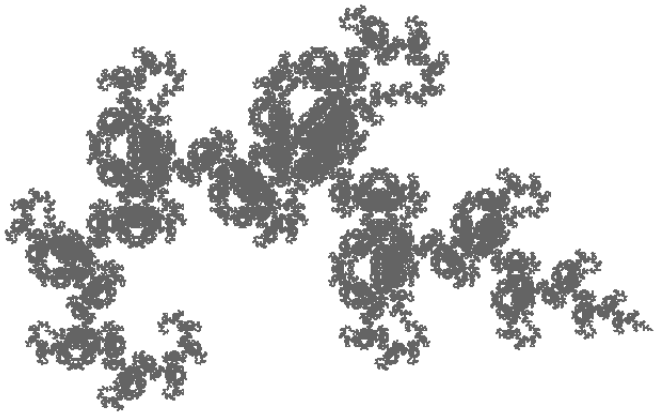


Figure 8.6 – Fractale : courbe du dragon

6. *Écrire une méthode récursive permettant de résoudre le jeu des tours de Hanoï.*

Les arbres

I Définitions

Les Graphes Un graphe $G = (X, U)$ est une structure constituée de 2 ensembles (fig. 9.1) : X un ensemble de nœuds (caractérisé par une simple étiquette en général) et U un ensemble d'arcs. Un arc est une paire de nœuds modélisant une relation entre les nœuds.

$X = \{1, 2, \ldots, 12\}$
$U = \{(1,2), (1,3), \ldots\}$

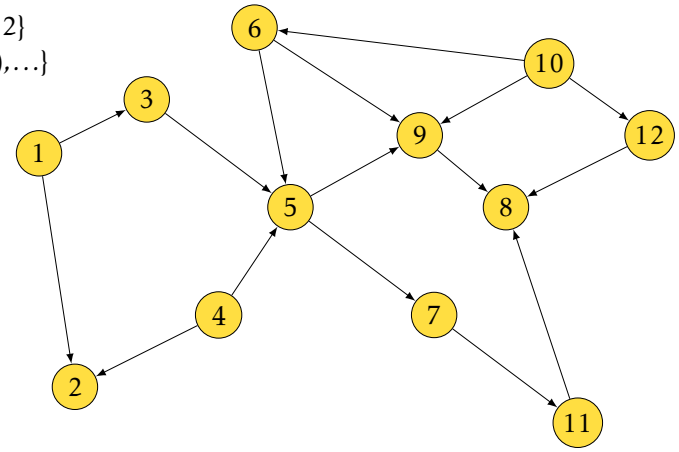


Figure 9.1 – Exemple de graphe orienté

Les Arbres Un arbre est un graphe connexe (il existe toujours un ensemble d'arc reliant deux nœuds) ne contenant aucun circuit (fig 9.2).

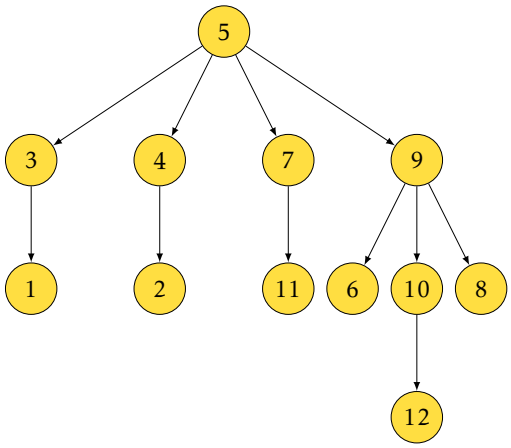


Figure 9.2 – Arbre

Les Arbres binaires Un arbre binaire est un arbre dans lequel tous les nœuds possèdent au plus 2 successeurs. On parle alors de fils gauche et de fils droit d'un nœud.

Les Arbres de recherche Un arbre de recherche est un arbre binaire dans lequel les étiquettes possèdent une relation d'ordre et construit tel que le fils droit d'un nœud x et tous ses descendants ont des étiquettes plus petites que celle de x, et le fils gauche de x et tous ses descendants ont des étiquettes plus grandes que celle de x.

Certains nœuds d'un arbre ont des propriétés particulières.

La Racine La racine d'un arbre est le seul nœud qui n'a aucun prédécesseurs.

Une Feuille Une feuille est un nœud qui n'a aucun successeurs.

II Modélisation en informatique

Un arbre est une structure de données qui généralise la notion de liste chaînée. Un arbre est constitué d'un nœud dit racine qui constitue le point d'"entrée" de la structure. Chaque possède une valeur (information que l'on souhaite stocker) et un ensemble de fils (références à d'autres nœuds). Un nœud père peut avoir plusieurs nœud fils. Un fils n'a qu'un seul père, et tous les nœuds ont un ancêtre commun appelé racine de l'arbre (le seul nœud qui n'a pas de père).

1 Modélisation des arbres binaires

Une modélisation informatique des arbres binaires (chaque nœud a au plus 2 fils) peut se faire de la façon suivante :
On définit simplement une classe modélisant un nœud :

```
public class Arbre {
    private ...    valeur ;    // ... : classe des informations à "stocker"
    private Arbre  filsGauche ;
    private Arbre  filsDroit ;
}
```

Remarque La définition de cette classe est récursive.
Cette classe devra bien sûr contenir des méthodes permettant de gérer la structure de données ainsi définie. On peut par exemple définir les 2 constructeurs suivants :

```
    /*
      *  Construit un arbre constitué d'un noeud de valeur fournie en entrée
      */
    public Arbre (... valeur) {
        this.valeur = valeur ;
        this.filsGauche = null ;    // null peut être utilisé pour indiquer
                                    // que le noeud n'a pas de fils gauche.
        this.filsDroit = null ;
    }

    /*
      *  Construit un père à fg et fd de valeur val
      */
    public Arbre (... val, Arbre fg, Arbre fd) {
        this.valeur = valeur ;
        this.filsGauche = fg ;
        this.filsDroit = fd ;
    }
```

2 Modélisation des arbres n-aires

Une modélisation informatique des arbres n-aires peut se faire de la façon suivante :
On définit simplement une classe modélisant un nœud :

```
public class Arbre {
    private ...              valeur ;
```

```
    private ArrayList<Arbre>  fils ;  // ArrayList ou autre ...
}
```

Travaux dirigés

Exercice 1

Arbres binaires

1. *Écrire une méthode permettant de savoir si un arbre binaire est une feuille.*
2. *Écrire une méthode permettant de savoir si une valeur donnée est présente ou non dans un arbre binaire.*
3. *Écrire une méthode calculant la hauteur d'un arbre.*

Exercice 2

1. *Écrire une méthode permettant d'afficher tous les nœuds d'un arbre binaire. On proposera un affichage dit en pré-ordre, en ordre puis en post-ordre.*

Exercice 3

Arbres de recherche :

1. *Écrire une méthode permettant de savoir si une valeur donnée est présente ou non dans un arbre de recherche.*
2. *Écrire une méthode permettant d'ajouter une nouvelle valeur dans un arbre de recherche.*

Exercice 4

➽ *Écrire une méthode permettant d'afficher toutes les valeurs d'un arbre n-aire (en pré-ordre).*

Les interfaces graphiques

Cette séance de TDAO est consacrée à la découverte des interfaces graphiques en utilisant la bibliothèque *Swing* et l'éditeur intégré de *NetBeans*.
Lorsqu'on souhaite développer une interface graphique, il est nécessaire de construire des programmes évènementiels.

I Programmation évènementielle

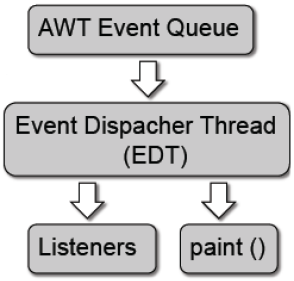
Un programme évènementiel est un programme qui réagit aux évènements. Après la création des objets son déroulement est contrôlé par les évènements qui surviennent. Les évènements peuvent avoir de nombreuses formes : entrée clavier, clic souris, déplacement souris, etc.

1 Notion de *thread*

Un *thread* est un fil d'exécution faisant partie d'un programme. Ce fil d'exécution fonctionne de façon autonome et parallèlement à d'autres threads.
Les threads permettent de répartir différents traitements d'un même programme en plusieurs unités distinctes qui peuvent ainsi être exécutées "simultanément".

Une application graphique avec Swing démarre toujours au minimum 3 threads :

- le programme principal `main()` ;
- l'*AWT event queue* qui reçoit les évènements provenant du système d'exploitation de la machine ;
- l'*event dispatcher thread* (EDT) qui gère les évènements en les envoyant aux écouteurs (*listeners*) implantés pour réagir et aux méthodes d'affichage (`paint()`).



En Java, la librairie Swing est dédiée à la conception et à l'utilisation d'interfaces graphiques. Swing intègre notamment tous les mécanismes de conception et de gestion des évènements.

2 Évènements

On trouve plusieurs types d'évènements :

- les évènements dits de "bas niveau" comme appuyer, relâcher, bouger la souris ou une touche du clavier, etc.
- les évènements dits de "haut niveau" comme l'activation d'un bouton, l'activation d'un champ de texte, la modification du texte dans un champ, etc.

Le package `java.awt.events` contient tous les évènements que peut gérer une interface graphique :

- `ActionEvent` évènement déclenché par un clic sur un bouton, sur le choix d'un menu, etc.
- `MouseEvent` évènement généré par la souris tels que un clic droit, un double clic, le survol d'un composant, un *drag and drop*, etc.
- `KeyEvent` évènement déclenché par l'utilisation du clavier.

II Fenêtres

Il existe deux types de fenêtres dans Swing :

- `JFrame` : fenêtre principale d'une application ;
- `JDialog` : fenêtre secondaire utile au dialogue avec l'utilisateur.

Les fenêtres de dialogues seront présentées dans la section suivante. Nous allons d'abord nous intéresser aux fenêtres principales de type `JFrame`.

1 JFrame

Une `JFrame` est une classe définissant une fenêtre et qui agit comme un conteneur pouvant contenir dans son `ContentPane` des composants Swing.

La classe `JFrame` possède quelques méthodes de base pour gérer entre autres sa dimension, son titre, son icône, et sa visibilité :

- `JFrame()` : constructeur par défaut.
- `JFrame(String titre)` : constructeur avec un titre.
- `void setTitle(String titre)` : change le titre de la fenêtre.
- `void setIconImage(Image image)` : affecte l'image d'icône de la fenêtre.
- `void setVisible(boolean estVisible)` : rend la fenêtre visible ou invisible.
- `void setSize(int largeur, int hauteur)` : fixe la dimension de la fenêtre à `largeur` pixels de large et `hauteur` pixels de haut.
- `void setLocation(int x, int y)` : positionne la fenêtre à `x` pixels du bord gauche de l'écran et `y` pixels du haut de l'écran (axe y vers le bas).
- etc.

Par exemple pour construire et afficher une fenêtre de 400 par 200 pixels et intitulée "Bienvenue", on peut écrire :

```
public static void main(String[] args) {
    JFrame fenetre = new JFrame("Bienvenue");
    fenetre.setSize(400,200);
    fenetre.setVisible(true);
}
```

2 Écouteurs (listeners)

Pour interagir avec l'utilisateur, il faut écouter les évènements des différents composants de l'application. Un écouteur (*listener*) est un objet destiné à recevoir et traiter les évènements. Toute classe peut être un *listener* si elle implante les méthodes correspondantes.

Par exemple, pour écouter des `KeyEvent`, il faut définir les méthodes `keyPressed`, `keyTyped` et `keyReleased` comme l'illustre la classe `EcouteurClavier` du package `Ecouteurs`.

```
public class EcouteurClavier implements KeyListener {
    public void keyPressed(KeyEvent event) {
        System.out.print("une touche a été appuyée - ") ;
        System.out.println("le code de la touche est "+event.getKeyCode());
    }

    public void keyTyped(KeyEvent event) {
        System.out.print("un caractère a été frappé - ");
        System.out.println("'"+event.getKeyChar()+"'");
    }

    public void keyReleased(KeyEvent event) {
        System.out.println("touche relachée : "+event.getKeyCode());
    }
}
```

La classe `EcouteurClavier` contient les trois méthodes `keyPressed`, `keyTyped` et `keyReleased` qui sont appelées lorsqu'un `KeyEvent` arrive. Le mot clé `implements KeyListener` signifie que la classe est un écouteur de clavier.

Le package `Ecouteurs` contient également la classe `TestEcouteurs` qui crée une fenêtre graphique et lui ajoute un écouteur interceptant les évènements provenant du clavier.

```
public class TestEcouteurs {
    public static void main(String[] args) {
        JFrame fenetre = new JFrame("Ma fenêtre");
        fenetre.setSize(500,400);
        fenetre.setVisible(true);
        fenetre.addKeyListener(new EcouteurClavier());
    }
}
```

1 *Tester et étudier le programme `TestEcouteurs`. Ce code ouvre une fenêtre et écoute les évènements clavier. Il suffit de taper quelques touches pour voir les effets de l'écouteur.*

Pour illustrer ces notions, nous allons maintenant construire une fenêtre simple capable d'écouter les clics et les mouvements de la souris.

2 *Écrire une classe `EcouteurSouris` sur le modèle de `EcouteurClavier` qui intercepte les évènements souris. Un écouteur souris implante l'interface `MouseListener` et doit donc définir les méthodes `mouseClicked(MouseEvent)`, `mousePressed(MouseEvent)`, `mouseReleased(MouseEvent)`, `mouseEntered(MouseEvent)` et `mouseExited(MouseEvent)`.*

Il est possible de retrouver des informations concernant la souris. Par exemple, les coordonnées de détection de l'évènement `MouseEvent` avec les méthodes `getX()` et `getY()`.

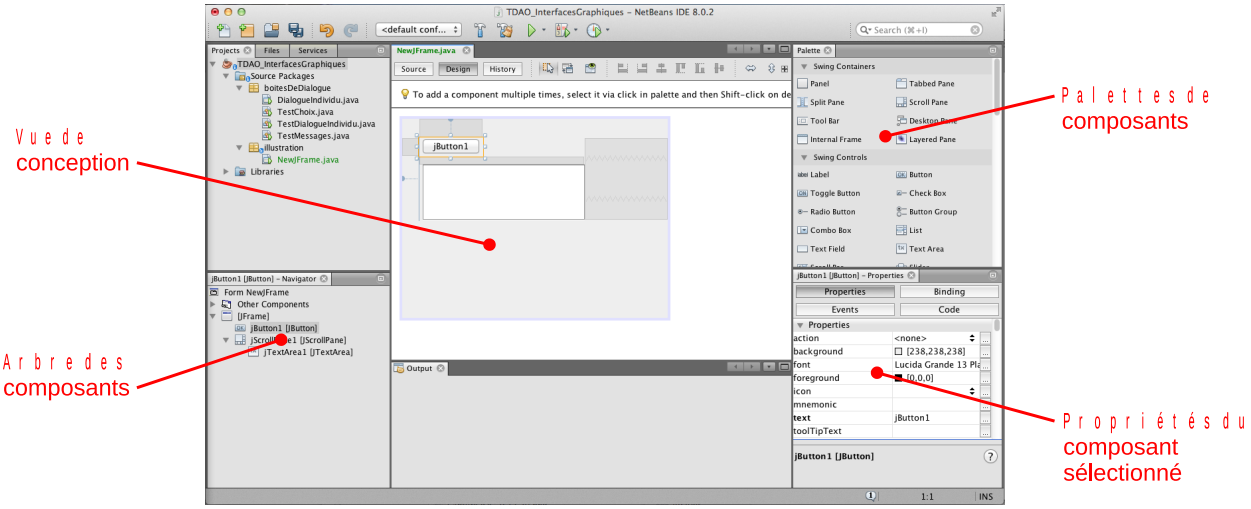
3 *Ajouter l'écouteur créé à la fenêtre avec `fenetre.addMouseListener(new EcouteurSouris());` et tester.*

3 Composants

De nombreux composants peuvent être ajoutés dans une fenêtre de type `JForm`. Parmi les plus utilisés, on retrouve les boutons `JButton`, les zones de texte `JTextField` (une seule ligne) et `jTextArea` (mini éditeur de texte), les cases à cocher `JCheckBox`, etc.

4 *Dans la méthode `main` de la classe `TestEcouteurs`, ajouter un bouton à la fenêtre à l'aide de la commande :* `fenetre.getContentPane().add(new JButton("Test"));`

Il peut être très fastidieux de coder une interface graphique à la main. Heureusement, NetBeans permet de créer une fenêtre graphique en glissant des composants directement sur la fenêtre en mode *design*. Le mode *Source* permet quant à lui de consulter le code généré automatiquement dans la classe correspondante et d'en ajouter le cas échéant.



5 *Ajouter au projet une nouvelle classe `ApplicationHello` de type* JFrame Form.

6 *Ajouter un `JButton` dans la fenêtre et changer son intitulé et la couleur du texte à l'aide du panneau de propriété.*

7 *Double-cliquer sur le bouton pour générer la méthode de gestion du clic. NetBeans génère automatiquement un écouteur associé pour gérer l'événement correspondant. Dans cette méthode, ajouter la commande* `System.out.println("Hello world!");`

8 *Tester l'exécution.*

9 *Ajouter un `JTextField` dans la fenêtre.*

10 *Modifier la méthode de gestion du clic en ajoutant la commande* `jTextField1.setText("Hello world!");`

11 *Tester l'exécution.*

Swing propose de très nombreux composants. Le tableau suivant liste les composants les plus usuels ainsi que les méthodes permettant de connaître ou de modifier leurs états.

Composant	Exemples d'accès	Exemples de modification
Text Field	`chaine = jTextField.getText();`	`jTextField.setText("Hello world!");`
Text Area	`chaine = jTextArea.getText();`	`jTextArea.append("Hello world!\n");` `jTextArea.setText("");`
Radio Button	`etat = jRadioButton.isSelected();`	`jRadioButton.setSelected(true);`
Check Box	`etat = jCheckBox.isSelected();`	`jCheckBox.setSelected(true);`
Spinner	`n = jSpinner.getValue();`	`jSpinner.setValue(17);`
Slider	`n = jSlider.getValue();`	`jSlider.setValue(17);`
Combo Box	`n = jComboBox.getSelectedIndex();` `objet = jComboBox.getSelectedItem();`	`jComboBox.addItem("ROUGE");` `jComboBox.setSelectedIndex(3);`

List	`n = jList.getSelectedIndex();`
	`objet = jList.getSelectedValue();`
		`DefaultListModel model = new DefaultListModel();`
		`jList.setModel(model);`
		`model.addElement("un");`

Table	`n = jTable.getRowCount();`
	`n = jTable.getColumnCount();`
	`ligne = jTable.getSelectedRow();`
	`col = jTable.getSelectedColumn();`
	`objet = jTable.getValueAt(ligne, col);`
		`DefaultTableModel model = (DefaultTableModel)jTable.getModel();`
		`model.setRowCount(6);`
		`model.setColumnCount(3);`
		`model.setValueAt(objet, ligne , col);`

12 *Ajouter quelques composants à la fenêtre `ApplicationHello` et modifier leurs contenus à l'aide de boutons.*

4 Dessiner dans une fenêtre

Le composant `JLabel` permet facilement d'intégrer des textes et des dessins dans une fenêtre.

13 *Ajouter un composant `JLabel` à la fenêtre `ApplicationHello` et tester.*

Pour dessiner, il faut associer une image (`BufferedImage`) au `JLabel` dans le constructeur de la fenêtre en écrivant :

```
public class Fenetre extends JFrame {

    private BufferedImage buffer;
    private Graphics2D contexteGraphique;

    public Fenetre() {
        initComponents();

        this.buffer = new BufferedImage(this.jLabel1.getWidth(), this.jLabel1.getHeight(), BufferedImage.
     TYPE_INT_ARGB);
        this.jLabel1.setIcon(new ImageIcon(this.buffer));
        this.contexteGraphique = this.buffer.createGraphics();
    }

    ...
```

On peut ensuite dessiner (des lignes, des cercles, des images, etc.) en utilisant le contexte graphique (voir leçon Java2D).

```
    this.contexteGraphique.setColor(Color.RED);
    this.contexteGraphique.drawLine(20,10, 150,30);

    this.contexteGraphique.setColor(Color.BLUE);
    this.contexteGraphique.fillRect(10,30, 50,40);
```

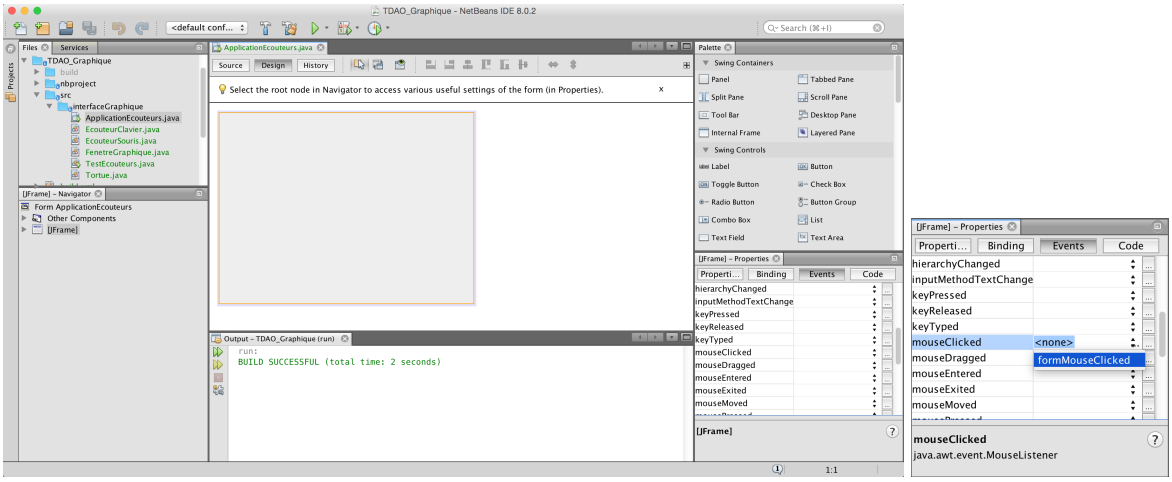
```
this.jLabel1.repaint();
```

La méthode `repaint()` demande au composant de se redessiner et d'afficher les dessins réalisés d'un seul coup.

14 *Ajouter un `JLabel` à la fenêtre et un bouton pour faire un petit dessin.*

5 Écouteurs de composant

L'environnement de développement *Netbeans* fournit également une assistance à la création d'écouteurs associés à tous les événements associés à un composant.



15 *Associer un écouteur souris au `JLabel` en sélectionnant l'évènement `mouseClicked` dans l'onglet `Events` des propriétés du `JLabel` .*

16 *Compléter le contrôleur associé en dessinant un petit rectangle aux coordonnées du clic.*

III Boîtes de dialogue

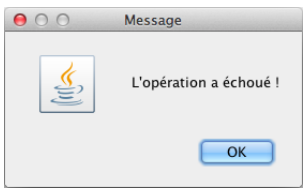
Les boîtes de dialogues sont des fenêtres très pratiques pour informer ou demander une information à l'utilisateur. De nombreuses boîtes de dialogues sont déjà définies dans *Swing* pour les usages courants (messages, confirmations, saisies ou choix d'un fichier). Il est également possible de définir des boîtes de dialogues personnalisées pour éditer des informations venant du modèle.

La plupart du temps, une boîte de dialogue est *modale*, ce qui signifie que toutes les interactions avec les autres fenêtres sont bloquées tant que l'utilisateur ne l'a pas fermé.

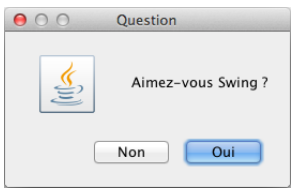
1 Boîtes de dialogue prédéfinies

La classe `JOptionPane` propose un certain nombre de boîtes de dialogue prédéfinies et très simples d'emploi. Par exemple, pour afficher un message, il suffit d'écrire :

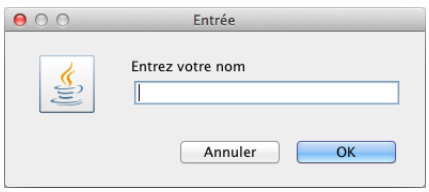
```
JOptionPane.showMessageDialog(null, "L'opération a échoué !");
```



(a) boîte de dialogue d'information



(b) boîte de dialogue de confirmation



(c) boîte de dialogue de saisie

Pour poser une question à l'utilisateur, on peut utiliser une boîte de confirmation :

```
int reponse = JOptionPane.showConfirmDialog(null, "Aimez-vous␣Swing␣?", "Question", JOptionPane.YES_NO_OPTION
    );

if (reponse == JOptionPane.YES_OPTION) {
    System.out.println("Réponse␣=␣oui");
} else if (reponse == JOptionPane.NO_OPTION) {
    System.out.println("Réponse␣=␣non");
}
```

Il est parfois utile de demander une information à l'utilisateur. La méthode `showInputDialog` permet de saisir une chaîne de caractère qui peut ensuite être convertie en d'autres types selon les besoins (avec un analyseur comme `Integer.parseInt(String)` par exemple).

```
String nom = JOptionPane.showInputDialog(null, "Entrez␣votre␣nom");

if (nom != null) {
    System.out.println("Saisie␣=␣" + nom);
}
```

Si l'utilisateur clique sur le bouton *Annuler*, la méthode renvoie `null`.

1 *En utilisant les exemples de la classe `TestMessages`, écrire un programme demandant deux nombres à l'utilisateur et affichant la division du premier par le second. Le programme devra afficher un message d'erreur en cas de division par zéro.*

2 Boîtes de dialogue de choix de fichier

Un autre type de dialogue permet la sélection d'un fichier en vue de la lecture de son contenu (ouverture d'un fichier existant). C'est le rôle de la classe `JFileChooser`. Le paramètre de son constructeur permet de définir le répertoire par défaut de la recherche (`"."` désigne le répertoire courant du programme). La méthode `showOpenDialog` ouvre une fenêtre de dialogue permettant de choisir un fichier puis renvoie le numéro du bouton qui a été cliqué. Le nom du fichier peut être récupéré grâce à la méthode `getSelectedFile`.

```
JFileChooser choixFichier = new JFileChooser(".");

int bouton = choixFichier.showOpenDialog(null);

if (bouton == JFileChooser.APPROVE_OPTION) {
    System.out.println("Fichier␣à␣lire␣:␣" + choixFichier.getSelectedFile());
}
```

De la même façon, la classe `JFileChooser` permet la sélection d'un nom de fichier en vue de son écriture (enregistrement sous un nouveau nom ou en remplacement d'un fichier existant).

```
JFileChooser choixFichier = new JFileChooser(".");

int bouton = choixFichier.showSaveDialog(null);

if (bouton == JFileChooser.APPROVE_OPTION) {
    System.out.println("Fichier␣à␣écrire␣:␣" + choixFichier.getSelectedFile());
}
```

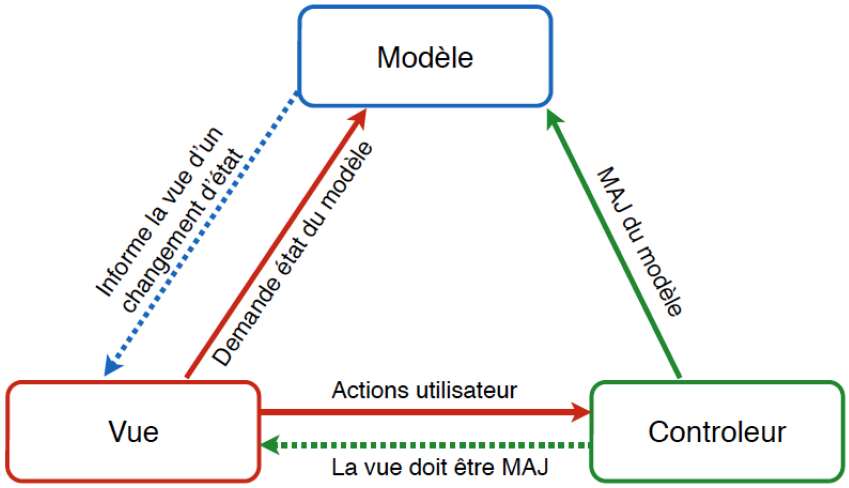
2 *Tester les exemples de la classe `TestChoix`.*

IV Applications et interfaces graphiques

Après avoir vu les concepts de base pour construire des applications graphiques, nous allons étudier l'organisation interne d'une application complète.

1 Architecture modèle-vue-contrôleur

L'architecture modèle-vue-contrôleur (MVC) est un modèle classiquement utilisé par les programmeurs pour séparer les différents types de traitements.



Ce paradigme regroupe les fonctions nécessaires en trois catégories :

- celles dédiées au modèle : modèle de données ;
- celles dédiées aux vues : présentation des données et interface avec l'utilisateur ;
- celles dédiées aux contrôleurs : logique de contrôle, gestion des évènements, synchronisation.

Les applications Swing se doivent de respecter au mieux ce modèle grâce notamment aux classes (modèle de données), aux écouteurs (contrôleurs) et aux composants graphiques (vues).

2 Création d'une application de gestion de stock

Nous allons maintenant créer une application pour gérer un stock. Le modèle de l'application sera constitué des classes `Stock` et `Article` définies dans les TPs précédents. La vue de l'application sera constituée d'une fenêtre principale et de quelques fenêtres de dialogue. Les contrôleurs permettront d'ouvrir et d'afficher un stock, d'ajouter un article, de modifier un article, de tracer l'histogramme des articles en stock, etc.

1 *Dans le package* `GestionStock`*, ajouter au projet une nouvelle classe* `ApplicationStock` *de type* `JFrame Form`*.*

2 *Ajouter à la classe `ApplicationStock` un attribut de type `Stock` comme illustré ci-dessous. Cet attribut permet d'associer un stock (le modèle) à la fenêtre (la vue).*

```
public class ApplicationStock extends javax.swing.JFrame {

    private Stock leStock;

    /**
      * Creates new form ApplicationStock
      */
    public ApplicationStock() {
        initComponents();

        this.leStock = new Stock("Castoramu");
    }
```

3 *Ajouter un bouton permettant d'ajouter un nouvel article au stock. On utilisera des boites de dialogues pour demander à l'utilisateur d'entrer une désignation, un code barre, un prix unitaire et une quantité.*

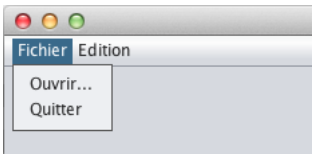
4 *Ajouter une zone de texte multiligne (`JTextArea`) et un bouton permettant d'afficher le stock dans cette zone de texte (se reporter au tableau page 65 pour ajouter une ligne à une zone de texte).*

5 *Ajouter un bouton permettant d'afficher la valeur totale du stock dans une boîte de texte (`JTextField`).*

3 Menu déroulant

Nous allons maintenant ajouter un menu déroulant.

6 *Ajouter un menu déroulant `Fichier` (avec `Menu Bar` et `Menu Item`) contenant les choix `ouvrir` et `quitter`.*



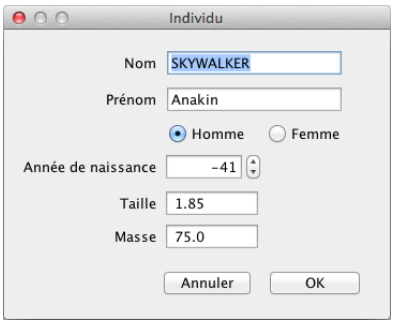
7 *Ajouter un contrôleur pour `quitter` du menu en double-cliquant sur l'item et en utilisant `dispatchEvent (new WindowEvent(this, WindowEvent.WINDOW_CLOSING));` pour déclencher la fermeture de la fenêtre.*

8 *Ajouter un contrôleur pour la commande `ouvrir`. Cette méthode doit d'abord afficher une fenêtre de dialogue de choix de fichier. Une fois le nom du fichier connu, elle doit lire le fichier en question et afficher le stock dans la zone de texte.*

4 Boites de dialogue personnalisées

Il est également possible de réaliser des boîtes de dialogues personnalisées permettant par exemple la modification des attributs d'un article du stock par l'utilisateur.

L'exemple ci-dessous illustre la création et l'utilisation d'une boîte de dialogue pour modifier un individu.



La première étape consiste à créer une nouvelle boîte de dialogue dérivant de `JDialog`. Pour cet exemple, nous avons créé la classe `DialogueIndividu` en ajoutant au projet un nouveau `JDialog Form` (disponible dans le groupe `Swing GUI Forms`).

La deuxième étape a pour objectif de réaliser la vue en ajoutant les composants nécessaires à l'édition des attributs du modèle (à l'aide de l'éditeur graphique). Il faut également ajouter les boutons `OK` et `Annuler`.

Il faut ensuite ajouter les contrôles indispensables pour gérer la fenêtre de dialogue. Si le bouton `Annuler` est cliqué, il faut fermer la fenêtre. On ajoute donc le code suivant (en double cliquant sur le bouton) :

```
private void jButtonAnnulerActionPerformed(java.awt.event.ActionEvent evt) {
    dispatchEvent(new WindowEvent(this, WindowEvent.WINDOW_CLOSING));
}
```

Si le bouton `OK` est cliqué, il faut également fermer la fenêtre mais en plus signaler qu'il faut prendre en compte les modifications. Dans ce but, on ajoute un attribut `miseAJourModele` à la classe `DialogueIndividu` qui prendra la valeur `true` si il faut tenir compte des modifications :

```
private void jButtonOKActionPerformed(java.awt.event.ActionEvent evt) {
    this.miseAJourModele = true;
    dispatchEvent(new WindowEvent(this, WindowEvent.WINDOW_CLOSING));
}
```

La dernière étape consiste à ajouter dans le fichier source l'attribut `miseAJourModele` et une méthode à la classe `DialogueIndividu` permettant de renseigner les contrôles de la fenêtre avec les attributs d'un individu puis de mettre à jour ceux-ci si le bouton `OK` est cliqué :

```
private boolean miseAJourModele = false;

public void editer(Individu unIndividu) {

    // Recopie des attributs de l'individu dans les contrôles
    jTextFieldNom.setText(unIndividu.getNom());
    jTextFieldPrenom.setText(unIndividu.getPrenom());
    jRadioButtonHomme.setSelected(unIndividu.estUnHomme());
    jRadioButtonFemme.setSelected(!unIndividu.estUnHomme());
    jSpinnerAnneeDeNaissance.setValue(unIndividu.getAnneeDeNaissance());
    jTextFieldTaille.setText(Double.toString(unIndividu.getTaille()));
    jTextFieldMasse.setText(Double.toString(unIndividu.getMasse()));
```

```
        // affichage de la fenêtre (modale)
        this.setVisible(true);

        // Recopie des valeurs des contrôles dans les attributs de l'individu
        // si le bouton OK est cliqué
        if (this.miseAJourModele) {
            unIndividu.setNom(jTextFieldNom.getText());
            unIndividu.setPrenom(jTextFieldPrenom.getText());
            unIndividu.setEstUnHomme(jRadioButtonHomme.isSelected());
            unIndividu.setAnneeDeNaissance((int) jSpinnerAnneeDeNaissance.getValue());
            unIndividu.setTaille(Double.parseDouble(jTextFieldTaille.getText()));
            unIndividu.setMasse(Double.parseDouble(jTextFieldMasse.getText()));
        }
    }
```

On peut alors utiliser la fenêtre de dialogue personnalisée pour modifier un individu :

```
Individu unIndividu = new Individu("Skywalker", "Anakin", -41, 1.85, 75.0, true);

DialogueIndividu dialogue = new DialogueIndividu(null, true);

dialogue.editer(unIndividu);

System.out.println(unIndividu);
```

9 *En vous inspirant de la classe `DialogueIndividu`, créer une boîte de dialogue `DialogueArticle` dans le package `GestionStock` permettant d'éditer un article. La classe `DialogueArticle` doit IMPERATIVEMENT être créée comme un nouveau `JDialog` et non un `JForm` (disponible dans le groupe `Swing GUI Forms`).*

10 *Ajouter un item au menu déroulant `Edition` permettant de modifier un article du stock avec la fenêtre de dialogue `DialogueArticle`.*

Java 2D

Compétences

- être capable de dessiner des formes simples (lignes, rectangles, ellipses) ou d'écrire du texte dans une fenêtre graphique.
- connaître la structure d'une image matricielle et être capable de la dessiner dans une fenêtre graphique

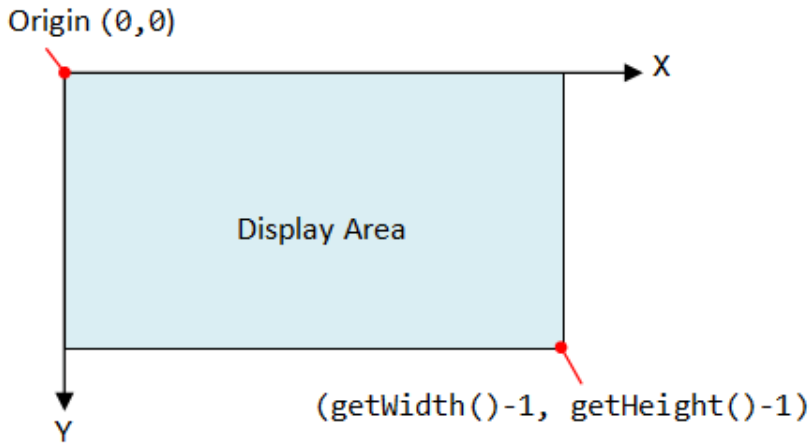
L'API Java 2D offre des fonctionnalités graphiques, textuelles et d'imagerie pour les programmes Java. Elle permet d'effectuer facilement les tâches suivantes :

- Dessiner des lignes, des rectangles et toute autre forme géométrique,
- Remplir ces formes avec des couleurs ou des gradients et des textures solides,
- Dessiner du texte avec des options pour un contrôle fin sur la police et le processus de rendu,
- Appliquer des transformations géométriques ou des opérations telles que l'union, l'intersection sur des formes définies ci-dessus,
- Dessiner des images, en appliquant des transformations géométriques ou des opérations de filtrage.

Cette leçon présente quelques possibilités de l'API Java 2D. Une documentation complète est disponible sur https://docs.oracle.com/javase/tutorial/2d/index.html

I Dessiner avec la classe `Graphics2D`

Pour dessiner sur différents périphériques, Java 2D utilise des classes spécifiques appelées *contextes graphiques*. Un contexte correspond à une surface de rendu graphique avec le repérage suivant :



L'API Java 2D comprend le contexte graphique `Graphics2D`, qui étend la classe `Graphics` pour donner accès à des fonctions graphiques améliorées. La classe `Graphics2D` contient des méthodes de dessin et des attributs définissant la couleur des tracés, la police de caractères, etc.

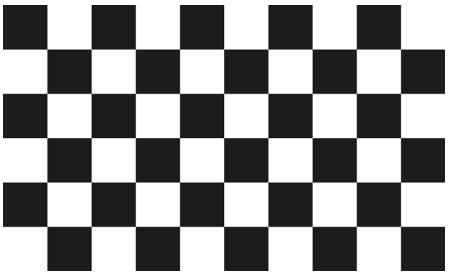
1 Dessiner des formes simples

L'exemple ci-dessous dessine une ligne rouge, un rectangle vert et un ellipse bleue pleine dans le contexte graphique d'une fenêtre. La classe `FenetreGraphique` est une classe « maison » permettant d'ouvrir une fenêtre graphique et d'actualiser le dessin une fois tous les tracés réalisés.

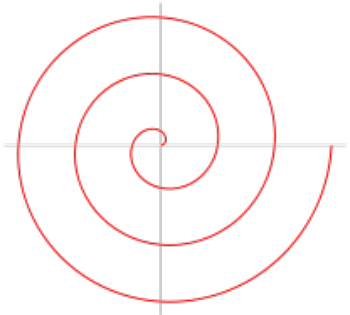
```
FenetreGraphique fenetre = new
 FenetreGraphique("Hello World!", 300, 250);

 Graphics2D contexte = fenetre.getGraphics2D();

 contexte.setColor(Color.RED); contexte.drawLine(10, 10, 60, 160);

 contexte.setColor(Color.GREEN); contexte.drawRect(100, 10, 50, 30);

 contexte.setColor(Color.BLUE); contexte.fillOval(200, 10, 20, 10);

 fenetre.actualiser();
```

Exercice 1

1. *Faire quelques essais en exécutant le fichier `TestDraw`.*
2. *Écrire une méthode dessinant un damier dans un contexte graphique*



3. *Écrire une méthode dessinant une courbe spirale de d'Archimède ($\rho = a\theta$) dans un contexte graphique.*



2 Gestion des couleurs

L'espace de couleur par défaut pour l'API Java 2D est sRGB, c'est-à-dire qu'une couleur est définie par la synthèse additive des trois composantes primaires rouge, vert et bleu. Chaque composante dans l'espace sRGB est une valeur entière comprise entre 0 et 255.

Un certain nombre de couleur sont déjà définies par défaut comme `Color.RED`, `Color.ORANGE`, `Color.WHITE`, etc. Il est possible de créer une couleur personnalisée en utilisant le constructeur de la classe `Color`.

```
Color rouge = new Color(255,0,0);
Color violet = new Color(255,0,255);
Color gris = new Color(128,128,128);
Color noir = new Color(0,0,0);
```

Inversement, on peut connaître les composantes d'une couleur à l'aide des accesseurs de la classe `Color`.

```
int rouge = couleur.getRed();
int vert = couleur.getGreen();
int bleu = couleur.getBlue();
```

Exercice 2

1 *Écrire une méthode dessinant un rectangle dégradé horizontal dans un contexte graphique.*

3 Ecrire du texte

La méthode `drawString` permet d'écrire du texte dans le contexte avec la couleur et la police de caractères choisies.

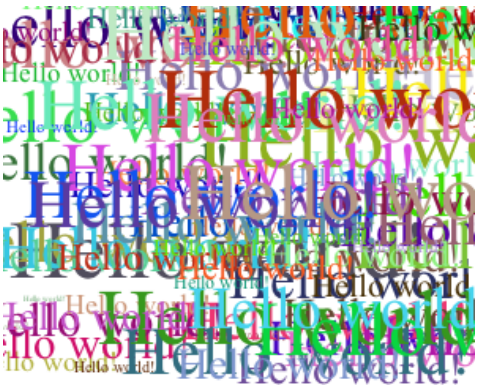
```
contexte.setColor(new Color(124,200,30));
contexte.setFont(new Font("Times New Roman", Font.PLAIN, 50));
contexte.drawString("Hello world!", 0, 125);
```

La classe `Font` permet de définir une nouvelle police de caractères à partir de son nom, de son style (normal : `Font.PLAIN`, italique : `Font.ITALIC`, gras : `Font.BOLD`, etc.) et de sa taille.

Exercice 3

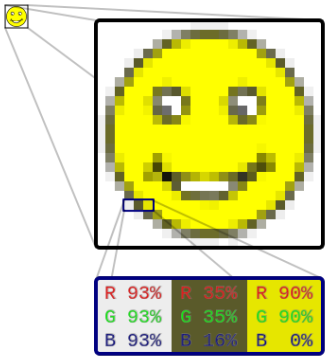
1 *Faire quelques essais en exécutant le fichier `TestDrawString`.*

2 *Écrire une méthode dessinant une chaine de caractère plusieurs fois au hasard avec des tailles et des couleurs aléatoires. On pourra utiliser la méthode `Math.random()` qui renvoie un nombre aléatoire entre 0.0 (inclu) et 1.0 (exclu).*



II Gestion des images

L'API Java 2D permet également de dessiner des images matricielles. Une image matricielle (bitmap) est constituée d'une matrice de points colorés appelés pixels. Chaque pixel possède une couleur qui lui est propre et est considérée comme un point. Une image matricielle est donc une juxtaposition de pixels de couleurs formant, dans leur ensemble, une image.



1 Stocker et dessiner des images

La classe `BufferedImage` permet de stocker des images matricielles qui peuvent être ensuite dessinées dans un contexte graphique.

```
BufferedImage image = ImageIO.read(new
 File("nyancat.png"));
 contexte.drawImage(image,150,100,null);
```

Exercice 4

1. *Faire quelques essais en exécutant le fichier* `TestDrawImage`.
2. *Modifier le fichier* `TestDrawImage` *pour faire défiler le nyan cat de gauche à droite dans la fenêtre. Pour adapter la vitesse de rafraichissement, on peut préciser un délai de 10 ms dans la méthode* `fenetre.actualiser(10)`.
3. *Modifier le fichier* `testDrawImage` *pour faire rebondir le nyan cat sur les bords droit et gauche de la fenêtre.*
4. *Modifier le fichier* `testDrawImage` *pour faire rebondir le nyan cat sur tous les bords de la fenêtre.*

2 Transformer les images

La classe `AffineTransform` de l 'API Java 2D permet d'appliquer une transformation affine (translation, rotation, échelle, etc.) à une image avant de la dessiner dans le contexte. Par exemple, le code ci-dessous agrandit une image d'un facteur 4 en X et -2 en Y et la place aux coordonnées (30, 150). Le coefficient négatif sur Y a pour effet de retourner l'image (miroir horizontal).

```
AffineTransform transform = new AffineTransform();
transform.translate(30, 150);
transform.scale(4.0, -2.0);

contexte.drawImage(image, transform, null);
```

Le code suivant tourne l'image de $-\pi/4$ et la place aux coordonnées (50; 50).

```
transform = new AffineTransform();
transform.translate(50, 50);
transform.rotate(-Math.PI / 4);

contexte.drawImage(image, transform, null);
```

Au besoin, il est possible de préciser le centre de rotation dans le repère de l'image, par exemple pour la faire tourner autour du centre de l'image comme dans l'exemple ci-dessous.

```
transform = new AffineTransform();
transform.translate(200, 50);
transform.rotate(-Math.PI / 4, image.getWidth() / 2, image.getHeight() / 2);

contexte.drawImage(image, transform, null);
```