Chapitre 8
Chaînes de caractères

Introduction

Considérons l'exemple suivant :

```
import java.util.Scanner;

public class TestComparaisonChaines {
  public static void main(String[] args) {
    String nom = "smi", str;
    Scanner input = new Scanner(System.in);
    System.out.print("Saisir le nom : ");
    str = input.nextLine();
    if (nom == str)
      System.out.println(nom + " = " + str);
    else
      System.out.println(nom + " different de " +
          str);
  }
}
```

Introduction

Dans cet exemple, on veut comparer les deux chaînes de caractères `nom` initialisée avec `"smi"` et `str` qui est saisie par l'utilisateur.

Lorsque l'utilisateur exécute le programme précédent, il aura le résultat suivant :

```
Saisir le nom de la filiere : smi
smi different de smi
```

Il semble que le programme précédent produit des résultats incorrects. Le problème provient du fait, que dans java, **String** est une classe et par conséquent chaque chaîne de caractères est un objet.

Remarque :

L'utilisation de l'opérateur == implique la comparaison entre les références et non du contenu.

En java, il existe des classes qui permettent la manipulation des caractères et des chaînes de caractères :

- **Character :** une classe qui permet la manipulation des caractères (un seul caractère).
- **String :** manipule les chaînes de caractères fixes.
- **StringBuilder** et **StringBuffer** : manipulent les chaînes de caractères modifiables.

Manipulation des caractères

Nous donnons quelques méthodes de manipulation des caractères.

L'argument passé pour les différentes méthodes peut être un caractère ou son code unicode.

Majuscule

isUpperCase() : test si le caractère est majuscule

toUpperCase() : si le caractère passé en argument est une lettre minuscule, elle retourne son équivalent en majuscule. Sinon, elle retourne le caractère sans changement.

Majuscule : Exemple

```
public class TestUpper {
    public static void main(String[] args) {
        char test='a';

        if (Character.isUpperCase(test))
            System.out.println(test + " est
                majuscule");
        else
            System.out.println(test + " n'est pas
                majuscule");

        test = Character.toUpperCase(test);
        System.out.println("Apres toUpperCase() : "
            + test);
    }
}
```

Minuscule

isLowerCase() : test si le caractère est minuscule

toLowerCase() : Si le caractère passé en argument est une lettre majuscule, elle retourne son équivalent en minuscule. Sinon, elle retourne le caractère sans changement.

isDigit() : Retourne `true` si l'argument est un nombre (0−9) et `false` sinon

isLetter() : Retourne `true` si l'argument est une lettre et `false` sinon

isLetterOrDigit() : Retourne `true` si l'argument est un nombre ou une lettre et `false` sinon

isWhitespace() : Retourne `true` si l'argument est un caractère d'espacement et `false` sinon. Ceci inclue l'espace, la tabulation et le retour à la ligne

Déclaration

Comme on l'a vu dans les chapitres précédents, la déclaration d'une chaîne de caractères se fait comme suit :

```
String  nom;
```

L'initialisation se fait comme suit :

```
nom="Oujdi";
```

Les deux instructions peuvent être combinées :

```
String  nom = "Oujdi";
```

L'opérateur **new** peut être utilisé :

```
String  nom = new String("Oujdi");
```

Pour créer une chaîne vide : `String  nom = new String();`

ou bien : `String  nom = "";`

Remarques

Une chaîne de type "Oujdi" est considérée par java comme un objet.
Les déclarations suivantes :

```
String  nom1 = "Oujdi";
String  nom2 = "Oujdi";
```

déclarent deux variables qui référencent le même objet ("Oujdi").

Par contre, les déclarations suivantes :

```
String  nom1 = new String("Oujdi");
String  nom2 = new String("Oujdi"); // ou nom2 = new String(nom1)
```

déclarent deux variables qui référencent deux objets différents.

Méthodes de traitement des chaînes de caractères

La compilation du programme suivant génère l'erreur « `array required, but String found System.out.println("Oujdi"[i]);` »

```
public class ProblemeManipString {
    public static void main(String[] args) {
        for(int i=0; i<5; i++)
            System.out.println("Oujdi"[i]);
    }
}
```

Pour éviter les erreurs de ce type, la classe « String » contient des méthodes pour manipuler les chaînes de caractères. Dans ce qui suit, nous donnons quelques unes de ces méthodes.

Méthode **charAt()**

Retourne un caractère de la chaîne.

Une correction de l'exemple précédent est :

```
public class ProblemeManipStringCorrection {
    public static void main(String[] args) {
        for(int i=0; i<5; i++)
            System.out.println("Oujdi".charAt(i));
    }
}
```

Méthode **concat()**

Permet de concaténer une chaîne avec une autre.

```
nom = "Oujdi".concat(" Mohammed");
//nom<--"Oujdi Mohammed"
```

Méthode **trim()**

supprime les séparateurs de début et de fin (espace, tabulation, ...)

```
nom = "\n Oujdi"+" Mohammed \n\t";
nom = nom.trim(); //nom<--"Oujdi Mohammed"
```

Méthodes **replace()** et **replaceAll()**

- **replace()** : Remplace toutes les occurrences d'une chaîne de caractères avec une autre chaîne de caractères
- **replaceAll()** : Remplace toutes les occurrences d'une expression régulière par une chaîne

```
String str;
str = "Bonjour".replace( "jour", "soir" );
//str <--"Bonsoir"
str = "soir".replaceAll("[so]", "t");
//str <-- "ttir"
str = "def".replaceAll("[a-z]","A");
//str <-- "AAA"
```

Méthode compareTo()

Permet de comparer un chaîne avec une autre chaîne.

```
String abc = "abc";
String def = "def";
String num = "123";
if ( abc.compareTo( def ) < 0 )    // true
  if ( abc.compareTo( abc ) == 0 )  // true
    if ( abc.compareTo( num ) > 0 )    // true
      System.out.println(abc);
```

Méthode **indexOf()**

Cherche la première occurrence d'un caractère ou d'une sous-chaîne de la chaîne

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
int i = abcs.indexOf( 's' );   // 18
int j = abcs.indexOf( "def" ); // 3
int k = abcs.indexOf( "smi" );   // -1
```

Méthode **valueOf()**

Retourne la chaîne qui est une représentation d'une valeur

```
double x=10.2;
str = String.valueOf(x); //str <-- "10.2"
int i = 20;
str = String.valueOf(i); //str <-- "20"
```

Méthode **substring()**

Retourne une sous-chaîne de la chaîne :

- **substring(debut) :** la sous-chaîne commence à partir du caractère qui se trouve à l'indice **debut** et se termine à la fin de la chaîne ;
- **substring(debut, fin) :** la sous-chaîne commence à partir du caractère qui se trouve à l'indice **debut** et se termine au caractère qui se trouve à l'indice **fin-1**.

Exemple :

```
str = "Bonjour".substring(3); //str <-- "jour"
str = "toujours".substring(3,7); //str <-- "jour"
```

Méthodes **startsWith()** et **endsWith()**

- **startsWith()** : Vérifie si une chaîne commence par un suffixe
- **endsWith()** : Vérifie si une chaîne se termine par un suffixe

```
String url = "http://www.ump.ma";
if ( url.startsWith("http") ) // true
  if ( url.endsWith("ma") ) // true
      System.out.println(str);
```

Méthodes equals() et equalsIgnoreCase()

La méthode :

- **equals()** compare une chaîne avec une autre chaîne en tenant compte de la casse (majuscule ou minuscule) ;
- **equalsIgnoreCase()** compare une chaîne avec une autre chaîne en ignorant la casse.

Exemple :

```
String str1="test", str2="Test";
if(str1.equals(str2))
   System.out.println("Les 2 chaines sont egaux");
else
   System.out.println("Les 2 chaines differents");

if(str1.equalsIgnoreCase(str2))
   System.out.println("Les 2 chaines sont egaux (
      sans tenir compte de la casse)");
```

Méthode getBytes()

Copie les caractères d'une chaîne dans un tableau de bytes.

Exemple :

```
String str1="abc_test_123";
byte[] b;
b = str1.getBytes();
for (int i = 0; i < b.length; i++)
  System.out.println(b[i]);
```

Méthode getChars()

Copie les caractères d'une chaîne dans un tableau de caractères.

Utilisation :
Soit **str** une chaîne de caractères. La méthode **getChars()** s'utilise comme suit :

```
str .getChars(srcDebut, srcFin, dst,  dstDebut)
```

Paramètres :

- **srcDebut :** indice du premier caractère à copier ;
- **srcFin :** indice après le dernier caractère à copier ;
- **dst :** tableau de destination ;
- **dstDebut :** indice du premier élément du tableau de destination.

Méthode getChars() :
Exemple :

```
str = "Ceci est un test";
char [] c = new char[str.length()];

str.getChars(0, str.length(), c, 0);
for (int i = 0; i < c.length; i++)
    System.out.println(c[i]);
```

Méthode toCharArray()

Met la chaîne dans un tableau de caractères.

Exemple :

```
str="Test";
char[] tabC=str.toCharArray();
for (int ind = 0; ind < tabC.length; ind++)
    System.out.println(tabC[ind]);
```

Méthode isEmpty()

Retourne **true** si la chaîne est de taille nulle.

Exemple :

```
if(str.isEmpty())
    System.out.println("Chaine vide");
else
    System.out.println("Chaine non vide");
```

Méthode indexOf()

Cherche la première occurrence d'un caractère ou d'une sous-chaîne de la chaîne. Elle retourne l'indice du premier occurrence du caractère dans la chaîne ou **-1** si le caractère n'existe pas.

Exemple :

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
int i = abcs.indexOf( 's' );   // 18
int j = abcs.indexOf( "def" ); // 3
int k = abcs.indexOf( "smi" );   // -1
```

Méthode lastIndexOf()

Cherche la dernière occurrence d'un caractère ou d'une sous-chaîne dans une chaîne. Elle retourne l'indice du dernier occurrence du caractère dans la chaîne ou **-1** si le caractère n'existe pas.

Exemple :

```
String abcs = "abcdefghijklmnopqrstuvwxyz+str";
int i = abcs.lastIndexOf( 's' );   // 27
int j = abcs.lastIndexOf( "def" ); // 3
int k = abcs.lastIndexOf( "smi" );   // -1
```

Méthode replaceFirst()

Remplace la première occurrence d'une expression régulière par une chaîne.

Exemple :

```
str = "Test1test2".replaceFirst("[0-9]", "_");
//Test_test2
```

Méthodes toLowerCase() et toUpperCase()

La méthode :

- **toLowerCase()** convertie la chaîne en minuscule ;
- **toUpperCase()** convertie la chaîne en majuscule.

Exemple :

```
String nom = "Oujdi";
nom = nom.toUpperCase(); //nom <-- "OUJDI"

str = "TEst";
str = str.toLowerCase(); //str <-- "test"
```

Méthode split()

split() sépare la chaîne en un tableau de chaînes en utilisant une expression régulière comme délimiteur.

```
nom = "Oujdi Mohammed";
String[] tabStr = nom.split(" ");
for (int ind = 0; ind < tabStr.length; ind++)
  System.out.println(tabStr[ind]); //Affichera :
//Oujdi
//Mohammed
str="Un:deux,trois;quatre";
tabStr = str.split("[:,;]");
for (int ind = 0; ind < tabStr.length; ind++)
    System.out.println(tabStr[ind]);  //Affichera :
//Un
//deux
//trois
//quatre
```

Méthode hashCode()

Retourne le code de hachage (hashcode) d'une chaîne. Pour une chaîne **s**, le code est calculé de la façon suivante :

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \cdots + s[n-1]$$

avec : $s[i]$ est le code du caractère à la position i (voir la méthode **getBytes()**).

Exemple :

```
str = "12";
System.out.println("code " + str.hashCode());
//retourne : 1569 = 49*31+50
```

Méthode valueOf()

Retourne la chaîne qui est une représentation d'une valeur.

Exemple :

```
double x = 10.2;
str = String.valueOf(x); //str <-- "10.2"

int test = 20;
str = String.valueOf(test); //str <-- "20"
```

Conversion entre String et types primitifs

Comme on l'a vu précédemment, la méthode **`valueOf()`** de la classe **`String`** permet de récupérer la valeur d'un type primitif.

Les classes **Byte, Short, Integer, Long, Float,** et **Double**, disposent de la méthode statique **toString()** qui permet de convertir un type primitif vers un **String**. Elles disposent respectivement des méthodes statiques **parseByte(), parseShort(), parseInt(), parseLong(), parseFloat()** et **parseDouble()**, qui permettent de convertir une chaîne en un type primitif.

Exemple :

```
public class TestConversion{
  public static void main(String[] args){
    double x = 4.5;
    String str = Double.toString(x);//str<--"4.5"

    int test = 20;
    str = Integer.toHexString(test); //str <-- "20"

    str = "2.3";
    x = Double.parseDouble(str); //x <-- 2.3

    str = "5";
    test = Integer.parseInt(str); //test <-- 5
  }
}
```

Affichage des objets

Le code suivant :

```
Etudiant etud = new Etudiant("Oujdi","Ali","A20");
System.out.println(etud);
```

affichera quelque chose comme : `Etudiant@1b7c680`.

Cette valeur correspond à la référence de l'objet.

Affichage des objets

Si on souhaite afficher le contenu de l'objet en utilisant le même code, il faut utiliser la méthode **toString** prévu par Java.

Affichage des objets : Exemple

```
public class TestEtudiantToString {
  public static void main(String[] args) {
    Etudiant etud = new Etudiant("Oujdi","Ali","A20"
        );
    System.out.println(etud);
  }
}

class Etudiant {
  private String nom, prenom, cne;
  ...
  public String toString(){
    return "Nom : "+nom+"\nPrenom : "+prenom+"\nCNE
        : "+cne;
  }
  ...
}
```

Affichage des objets : Exemple

Affichera :

```
Nom : Oujdi
Prenom : Ali
CNE : A20
```

Comparaison des objets

Le code suivant :

```
Etudiant etud1 = new Etudiant("Oujdi","Ali","A20");
Etudiant etud2 = new Etudiant("Oujdi","Ali","A20");

if(etud1 == etud2)
    System.out.println("Identiques");
else
    System.out.println("Differents");
```

affichera toujours « `Differents` », du fait que la comparaison s'est faite entre les références des objets.

Comparaison des objets

Comme pour la méthode **toString**, Java prévoit l'utilisation de la méthode **equals** qui pourra être définie comme suit :

```
class Etudiant {
  private String nom, prenom, cne;
  ...
  public boolean equals(Etudiant bis){
    if (nom == bis.nom && prenom == bis.prenom  &&
        cne == bis.cne)
        return true;
    else
        return false;
  }
}
```

Comparaison des objets : utilisation de equals

```
public class TestEtudiantToString {
  public static void main(String[] args) {
  Etudiant et = new Etudiant("Oujdi","Ali","A20");
  Etudiant et1 = new Etudiant("Oujdi","Ali","A20");

  if(et.equals(et1))
    System.out.println("Identiques");
  else
    System.out.println("Differents");
  }
}
```

Les classes StringBuilder et StringBuffer

Lorsqu'on a dans un programme l'instruction

```
str = "Bonjour";
```

suivie de

```
str = "Bonsoir";
```

le système garde en mémoire la chaîne "Bonjour" et crée une nouvelle place mémoire pour la chaîne "Bonsoir". Si on veut modifier "Bonsoir" par "Bonsoir SMI5", alors l'espace et "SMI5" ne sont pas ajoutés à "Bonsoir" mais il y aura création d'une nouvelle chaîne. Si on fait plusieurs opérations sur la chaîne str, on finira par créer plusieurs objets dans le système, ce qui entraîne la consommation de la mémoire inutilement.

Les classes StringBuilder et StringBuffer

Pour remédier à ce problème, on peut utiliser les classes **StringBuilder** ou **StringBuffer**. Les deux classes sont identiques à l'exception de :

- **StringBuilder** : est plus efficace.
- **StringBuffer** : est meilleur lorsque le programme utilise les threads.

Puisque tous les programmes qu'on va voir n'utilisent pas les threads, le reste de la section sera consacré à la classe **StringBuilder**.

Déclaration et création

Pour créer une chaîne qui contient "Bonjour", on utilisera l'instruction :

```
StringBuilder  message = new StringBuilder("Bonjour");
```

Pour créer une chaîne vide, on utilisera l'instruction :

```
StringBuilder  message = new StringBuilder();
```

Remarque :

L'instruction :

```
StringBuilder  message = "Bonjour";
```

est incorrecte. Idem, si « message » est un StringBuilder, alors l'instruction :

```
message = "Bonjour";
```

est elle aussi incorrecte.

Méthodes de StringBuilder

length()	Retourne la taille de la chaîne
charAt()	Retourne un caractère de la chaîne
substring()	Retourne une sous-chaîne de la chaîne
setCharAt(i,c)	permet de remplacer le caractère de rang i par le caractère c.
insert(i,ch)	permet d'insérer la chaîne de caractères ch à partir du rang i
append(ch)	permet de rajouter la chaine de caractères ch à la fin
deleteCharAt(i)	efface le caractère de rang i.
toString()	Convertie la valeur de l'objet en une chaîne (conversion de StringBuilder vers String)
concat()	Concaténer une chaîne avec une autre
contains()	Vérifie si une chaîne contient une autre chaîne

endsWith()	Vérifie si une chaîne se termine par un suffixe
equals()	Compare une chaîne avec une autre chaîne
getBytes()	Copie les caractères d'une chaîne dans un tableau de bytes
getChars()	Copie les caractères d'une chaîne dans un tableau de caractères
hashCode()	Retourne le code de hachage (hashcode) d'une chaîne
indexOf()	Cherche la première occurrence d'un caractère ou d'une sous-chaîne de la chaîne
lastIndexOf()	Cherche la dernière occurrence d'un caractère ou d'une sous-chaîne dans une chaîne
replace()	Remplace toutes les occurrences d'un caractère avec un autre caractère

Remarque :

On ne peut pas faire la concaténation avec l'opérateur + entre des StringBuilder. Par contre :

StringBuilder + String

produit une nouvelle chaîne de type String.

Exemple

```
public class TestStringBuilder {
  public static void main(String args[]) {
  StringBuilder strBuilder = new StringBuilder("
     Bonjour SMI5");
  int n = strBuilder.length(); // n <-- 12

  char c = strBuilder.charAt(2); // c <-- 'n'

  strBuilder.setCharAt(10, 'A');
  // remplace dans strBuilder le caractere 'I' par '
     A'.
  // strBuilder <-- "Bonjour SMA5"
```

Exemple (suite)

```
strBuilder.insert(10, " semestre ");
// insere dans strBuilder la chaine " semestre "
   a
// partir du rang 10.
// strBuilder <-- "Bonjour SMA semestre 5"


strBuilder.append(" (promo 14-15)");
// strBuilder <-- "Bonjour SM semestre A5 (promo
   14-15)"

strBuilder = new StringBuilder("Boonjour");
strBuilder.deleteCharAt(2);
// supprime de la chaine strBuilder le caractere
   de rang 2.
// strBuilder <-- "Bonjour"
```

Exemple (suite)

```
    String str = strBuilder.toString();
    // str <-- "Bonjour"

    str = strBuilder.substring(1, 4);
    // str <-- "onj"

    str = strBuilder + " tous le monde";
    // str <-- "Bonjour tous le monde"
    }
}
```

Chapitre 9
Interfaces et packages

Introduction

Le langage c++ permet l'héritage multiple, par contre Java ne permet pas l'héritage multiple. Pour remédier à ceci, Java utilise une alternative qui est la notion **d'interfaces**.

Définition

une **interface** est un ensemble de méthodes abstraites.

Déclaration

La déclaration d'une interface se fait comme celle d'une classe sauf qu'il faut remplacer le mot clé class par interface.

Exemple

```
public interface Forme {
    public abstract void dessiner ( ) ;
    public abstract void deplacer (int x , int y) ;
}
```

Propriétés

Les interfaces ont les propriétés suivantes :

- une interface est implicitement abstraite. On n'a pas besoin d'utiliser le mot clé abstract dans sa déclaration ;
- chaque méthode définie dans une interface est abstraite et public, par conséquent, les mots clés public et abstract peuvent être omis.

Propriétés

L'exemple précédent devient :

```
interface Forme {
    void dessiner ( ) ;
    void deplacer (int x , int y) ;
}
```

Règles

Une interface est similaire à une classe dans les points suivants :

- une interface peut contenir plusieurs méthodes ;
- une interface peut se trouver dans un fichier séparé (.java) ;
- peut se trouver dans un paquetage (voir section packages).

Restrictions

Il y a des restrictions concernant les interfaces. Une interface :

- ne peut pas instancier un objet et par conséquent ne peut pas contenir de constructeurs ;
- ne peut pas contenir d'attributs d'instances. Tous les attributs doivent être static et final ;
- peut hériter de plusieurs interfaces (héritage multiple autorisé pour les interfaces).

Remarque :

Dans Java 8, une interface peut contenir des méthodes statiques.

Implémenter une interface

Une classe peut implémenter une interface en utilisant le mot clé implements. On parle d'implémentation et non d'héritage.

Exemple

```
class Rectangle implements Forme {
    private double largeur, longueur;
    ...

    public double perimetre() {
        return 2 * (largeur + longueur);
    }

    public double surface() {
        return largeur * longueur;
    }
}
```

Implémentation partielle

Une classe doit implémenter toutes les méthodes de l'interface, sinon elle doit être abstraite.

Exemple

```
abstract class Rectangle implements Forme
    {
    private double largeur, longueur;

    public double surface() {
        return largeur * longueur;
    }
}
```

Implémentation multiple

Une classe peut implémenter plusieurs interfaces.

Exemple

Considérons les 2 interfaces **I1** et **I2** :

```
interface I1 {
    final static int MAX = 20;
    void meth1();
}
```

```
interface I2 {
    void meth2();
    void meth3();
}
```

Exemple

La classe **A** peut implémenter les 2 interfaces **I1** et **I2** :

```
class A implements I1 , I2{
    // attributs
    public void meth1(){
        int i=10;
        //on peut utiliser la constante MAX
        if (i < MAX)  i++;
        //implementation de la methode
    }
    public void meth2(){
        // ...
    }
    public void meth3(){
        // ...
    }
}
```

Implémentation et héritage

Une classe **B** peut hériter de la classe **A** et implémenter les 2 interfaces **I1** et **I2**, comme suit :

```
class B extends A implements I1,I2
```

Polymorphisme et interfaces

Déclaration :

On peut déclarer des variables de type interface :
Forme forme;

Pour instancier la variable « forme », il faut utiliser une classe qui implémente l'interface « Forme ». Par exemple :

```
forme = new Rectangle(2.5, 4.6);
```

Tableaux

Ceci peut être étendu pour les tableaux. Considérons la classe **Cercle** qui implémente elle aussi l'interface **Forme** et la classe **Carre** qui hérite de **Rectangle**.

Tableaux

Classe Cercle :

```
class Cercle implements Forme {
    private double rayon;

    public Cercle(double rayon) {
        this.rayon = rayon;
    }

    public double perimetre() {
        return 2 * Math.PI * rayon;
    }

    public double surface() {
        return Math.PI * rayon * rayon;
    }
}
```

Tableaux

Classe Carre :

```
class Carre extends Rectangle{
    private double largeur;

    public Carre(double largeur){
        super(largeur,largeur);
    }
}
```

Tableaux

On pourra écrire :

```
public static void main(String[] args) {
    Forme [] tabForme = new Forme[3];
    tabForme[0]=new Rectangle(10, 20);
    tabForme[1]=new Cercle(3);
    tabForme[2]=new Carre(10);

    for(int i=0; i<3; i++)
        System.out.printf("%.2f\t%.2f\n",
            tabForme[i].surface(),
            tabForme[i].perimetre());
}
```

Casting

Ajoutons à la classe **Cercle** la méthode « diametre() » :

```
class Cercle implements Forme {
    private double rayon;
    ...
    public double diametre() {
        return 2 * rayon;
    }
}
```

Casting

Considérons l'instruction :

```
Forme forme = new Cercle(5);
```

L'instruction :

```
double d = forme.diametre();
```

génère une erreur de compilation. Pour éviter cette erreur, il faut faire un cast comme suit :

```
double d = ((Cercle) forme).diametre();
```

Héritage

Une interface peut hériter d'une ou plusieurs interfaces.

Exemple :

Considérons les deux interfaces **I1** et **I2**

```
interface I1 {
    final static int MAX = 20;
    void meth1();
}
```

```
interface I2 {
    void meth2();
    void meth3();
}
```

Héritage

Une interface **I3** pourra hériter des deux interfaces **I1** et **I2**

```
interface I3 extends I1, I2 {
    void meth4();
}
```

On fait l'instruction interface I3 extends I1, I2, est équivalente à :

```
interface I3  {
    final static int MAX = 20;
    void meth1();
    void meth2();
    void meth3();
    void meth4();
}
```

Héritage

Important

L'héritage entre interfaces est différents de celui des classes.
L'héritage multiple est autorisé pour les interfaces et n'est pas autorisés pour les classes

Exercices

Exercice

Corrigez le programme suivant et justifiez chaque correction :

```
interface  Test {
    double m1();
    int m2(){}
}

public class ClasseA {
    public static void main(String[] args
        ) {
         Test i1 = new Test();
         Test i2;
    }
}
```

Solution

```
interface  Test {
    double m1();
    //methode abstraite, ne doit pas contenir un
       corps
    int m2();

}


public class ClasseA {
    public static void main(String[] args) {
        //On ne peut pas instancier une interface
        Test i1 ;
        Test i2;
    }
}
```

Exercice

Créez une interface nommée « **Tourner** » qui contient une seule méthode nommée « **tourner** ».

Créez une classe nommée :

- « **Page** » qui implémente **tourner** qui affiche « Tourner la page » ;
- « **Disque** » qui implémente **tourner** qui affiche « Faire une rotation ».
- « **Voiture** » qui implémente **tourner** qui affiche « Faire un tour ».

Utilisez la classe « **TestTourner** » pour tester les trois classes

Solution

L'interface **Tourner :**

```
public interface Tourner {
    void tourner();
}
```

Classe **Page :**

```
public class Page implements Tourner {
    public void tourner() {
        System.out.println("Tourner la page");
    }
}
```

Classe **Disque :**

```
public class Disque implements Tourner {
    public void tourner() {
        System.out.println("Faire une rotation");
    }
}
```

Classe **Voiture :**

```
public class Voiture implements Tourner {
    public void tourner() {
        System.out.println("Faire un tour");
    }
}
```

Classe **TestTourner :**

```
public class TestTourner {
    public static void main(String[] args) {
        Page page = new Page();
        Disque disque = new Disque();

        page.tourner();
        disque.tourner();

        Tourner[] tab = new Tourner[3];
        tab[0] = new Page();
        tab[1] = new Disque();
        tab[2] = new Voiture();

        for (Tourner tr : tab)
            tr.tourner();
    }
}
```

Packages

Introduction

Un package est un ensembles de classes. Il sert à mieux organiser les programmes. Si on n'utilise pas de packages dans nos programmes, alors on travail automatiquement dans le package par défaut (default package).

Pour la saisie à partir du clavier, nous nous avons utilisé la **Scanner**, pour cela nous avons importé le package **java.util**.

Création d'un package

Pour créer un package, on utilise le mot clé package. Il faut ajouter l'instruction

package nomPackage;

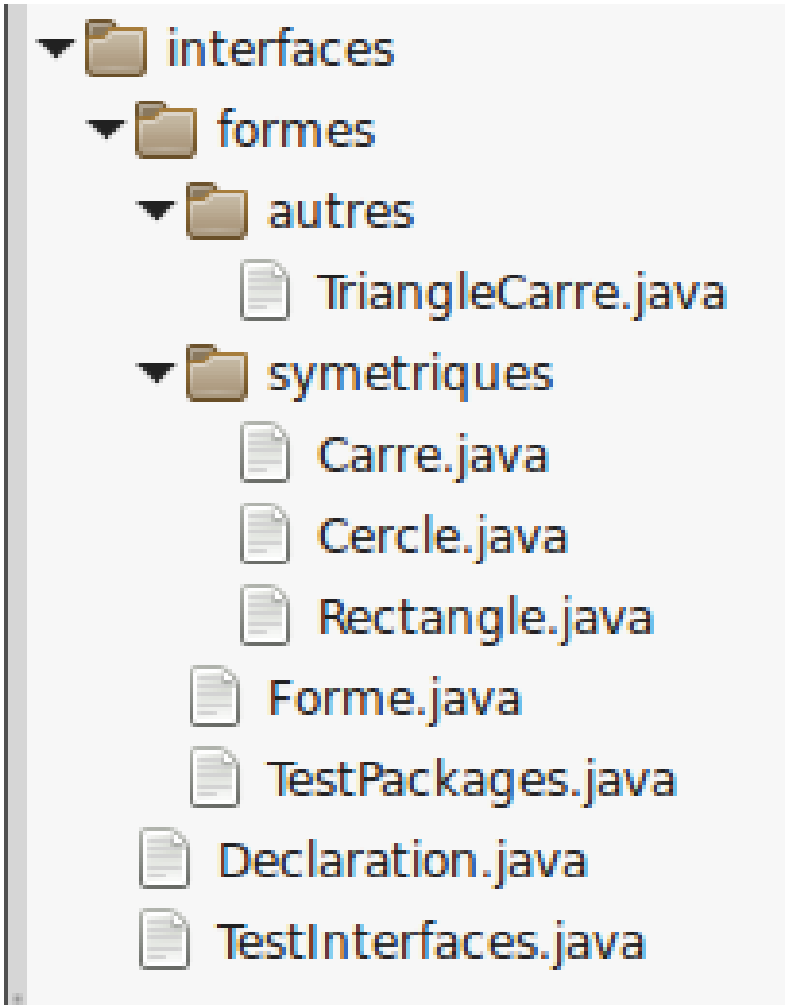
au début de chaque fichier.

Pour qu'elle soit utilisable dans un package, une classe publique doit être déclarée dans un fichier séparé.

On accède à une classe publique en utilisant le nom du package.

Exemple

Considérons la hiérarchie suivante :



Exemple

Nous avons la structure suivante :

- le répertoire **interfaces** contient le répertoire **formes** et les fichiers **Declaration.java** et **TestInterfaces.java** ;
- le répertoire **formes** contient les répertoires **symetriques** et **autres**, et les fichiers **Forme.java** et **TestPackages.java** ;
- le répertoire **autres** contient le fichier **TriangleCarre.java** ;
- le répertoire **symetriques** contient les fichiers **Carre.java**, **Cercle.java** et **Rectangle.java**.

Exemple

Nous avons la classe **TriangleCarre** se trouve dans le répertoire **autres**, donc on doit inclure, au début l'instruction :

```
package interfaces.formes.autres;
```

Nous avons aussi la classe **TriangleCarre** hérite de la classe **Rectangle**, donc, on doit inclure l'instruction :

```
import interfaces.formes.symetriques.Rectangle;
```

La classe **TriangleCarre** aura la structure suivante :

```
package interfaces.formes.autres;

import interfaces.formes.symetriques.Rectangle;

public class TriangleCarre extends Rectangle {
    private double cote;

    public TriangleCarre(double largeur, double
        longueur, double cote) {
        super(largeur, longueur);
        this.cote = cote;
    }

    public double surface() {
        return super.surface() / 2;
    }
}
```

La classe **TestPackages** sert pour tester les différentes classes, donc on doit inclure, au début les instructions :

```
package interfaces.formes.autres;

import interfaces.formes.autres.TriangleCarre;
import interfaces.formes.symetriques.Carre;
import interfaces.formes.symetriques.Cercle;
import interfaces.formes.symetriques.Rectangle;
```

Pour simplifier, on peut remplacer les trois dernières instructions par :

```
import interfaces.formes.symetriques.*;
```

La structure de la classe **TestPackages** est comme suit :

```
package interfaces.formes;
import interfaces.formes.autres.TriangleCarre;
import interfaces.formes.symetriques.*;

public class TestPackages {
    public static void main(String[] args) {
        Forme[] tabForme = new Forme[4];
        tabForme[0] = new Rectangle(10, 20);
        tabForme[1] = new Cercle(3);
        tabForme[2] = new Carre(10);
        tabForme[3] = new TriangleCarre(3,4,5);

        for (int i = 0; i < 3; i++)
            System.out.printf("%.2f\t%.2f\n",
                tabForme[i].surface(),
                    tabForme[i].perimetre());
    }
```

Classes du même package

Pour les classes qui sont dans le même package, on n'a pas besoin de faire des importations et on n'a pas besoin de mettre une classe par fichier si on veut que cette classe ne soit pas visible pour les autres packages.

Classes du même package

Soit le fichier **Carre.java** qui contient les 2 classes **Carre** et **Cube** :

```
package interfaces.formes.symetriques;

public class Carre extends Rectangle {
    private double largeur;

    public Carre(double largeur) {
        super(largeur, largeur);
    }
    ...
}
class Cube extends Carre {
    ...
    public double volume() {
        return surface() * super.getLargeur();
    }
```

Remarques

1. Les classes **Rectangle** et **Carre** sont dans le même package, donc on n'a pas besoin de faire importations.

2. La classe **Cube** est invisible dans les autres packages. Dans la classe **TestPackages**, une instruction comme :
   Cube cube =new Cube();
   génère une erreur de compilation, du fait que la classe **TestPackages** se trouve dans le package **interfaces.formes**

Fichiers jar

Un fichier **jar** (**J**ava **AR**chive) est un fichier qui contient les différentes classes (compilées) sous format compressé.

Pour générer le fichier **jar** d'un projet sous eclipse, il faut

- cliquer avec la souris sur le bouton droit sur le nom du projet puis cliquer sur **export**
- choisir après **JAR** dans la section **java**,
- cliquer sur **next**, et choisir un nom pour le projet dans la partie **JAR file** (par exemple **test.jar**)
- puis cliquer 2 fois sur **next**
- enfin, choisir la classe principale (qui contient **main**) et cliquer sur **Finish** pou terminer l'exportation.

Exécution

En se positionnant dans le répertoire qui contient le fichier **test.jar**, ce dernier peut être exécuté en utilisant la commande :

```
java -jar test.jar
```

Utilisation dans un autre projet

Vous pouvez utiliser le fichier jar dans un autre projet, en procédant, sous eclipse, comme suit :

- cliquer avec la souris sur le bouton droit sur le nom du projet puis cliquer sur **properties**
- choisir après **Java Build Path**
- cliquer ensuite sur l'onglet **Librairies**
- puis cliquer sur **Add External JARs**
- enfin, choisir le fichier **JAR** et valider par **ok**.

Vous pouvez ensuite importé les packages et utilisé les classes qui se trouvent dans le fichier **jar**.

Chapitre 10
Gestion des exceptions

Introduction

Définition

Une **exception** est une erreur qui se produit durant l'exécution d'un programme.

Les programmes peuvent générer plusieurs types d'exceptions :

- division par zéro ;
- une mauvaise valeur entré par l'utilisateur (une chaîne de caractères au lieu d'un entier) ;
- dépassement des capacités d'un tableau ;
- lecture ou écriture dans un fichier qui n'existe pas, ou pour lequel le programme n'a pas les droits d'accès ;
- ...

Ces erreurs sont appelés **exceptions** du fait qu'elles sont exceptionnelles.

Gestion des erreurs

Supposons qu'on veut écrire une fonction qui calcule la fonction $f(x) = \sqrt{x-1}$.

Une première version pourrait s'écrire comme suit :

```
public static double f(double x) {
        return Math.sqrt(x-1);
    }
```

Gestion des erreurs

Dans une méthode main(), l'instruction :

```
System.out.println("f(" + x + ") = " + f(x));
```

avec x = 0, on aura l'affichage :

```
f(0.0) = NaN
```

Gestion des erreurs

On pourra modifier le code de la fonction de la façon suivante :

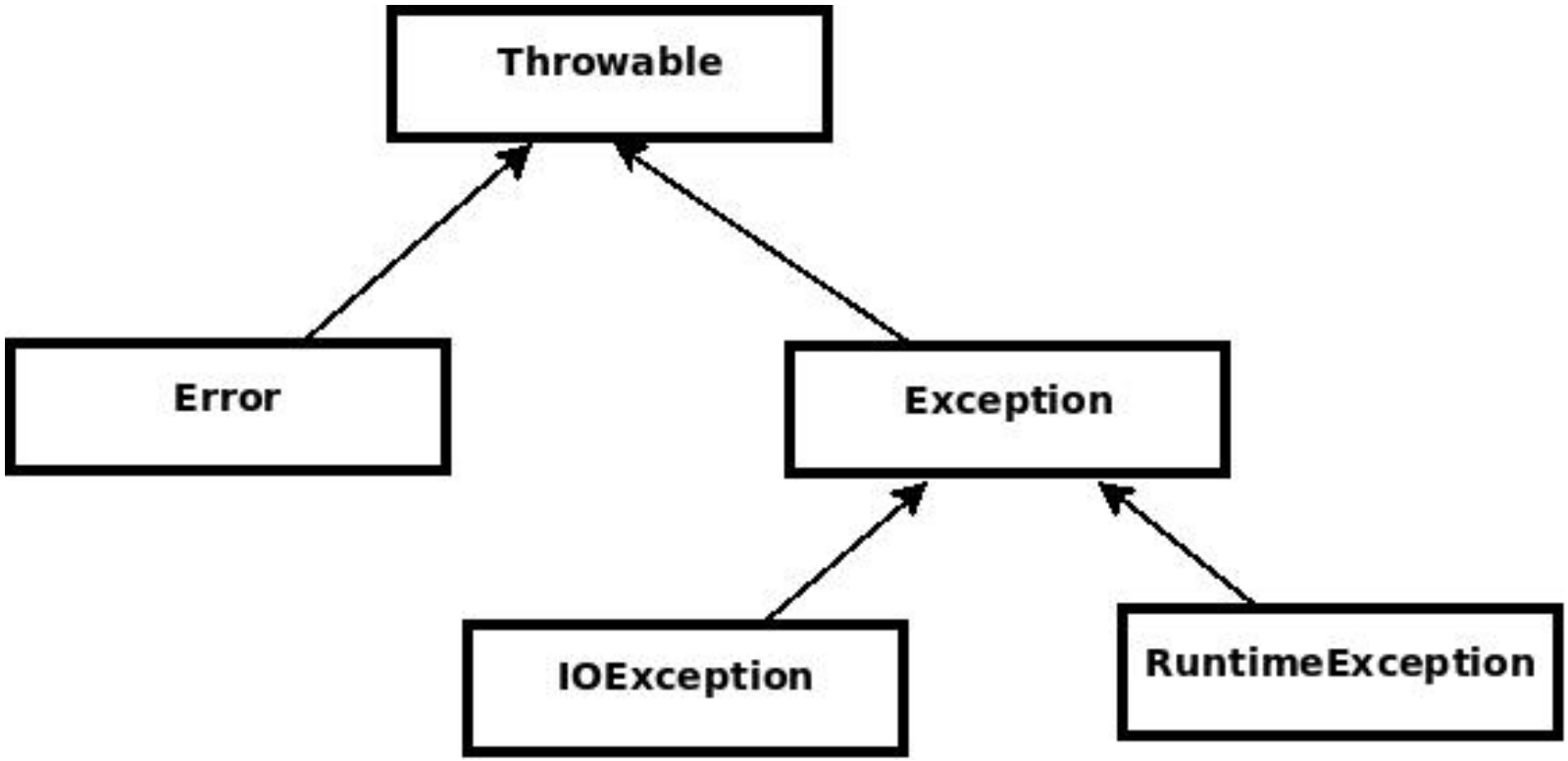
```
public static double f(double x) {
    if (x >= 1)
        return Math.sqrt(x - 1);
    else {
        System.out.println("Erreur");
        return -1;
    }
}
```

Le problème avec ce code c'est qu'il fait un affichage, qui n'est pas prévu par la fonction, de plus, il retourne une valeur qui n'est pas vraie qui pourrait aboutir à de mauvaises conséquences pour le reste du programme.

Le code précédent peut être généralisé en utilisant les exceptions.

En java, il y a deux classes pour gérer les erreurs : **Error** et **Exception**.

Les deux classes sont des sous-classe de la classe **Throwable** comme le montre la figure suivante :



Classe Error

La classe **Error** représente les erreurs graves qu'on ne peut pas gérer. Par exemple, il n'y a pas assez de mémoire pour exécuter un programme.

Le programme suivant :

```
package chap10exceptions;

//permet de gerer des tableaux dynamiques(vecteurs)
import java.util.Vector;

public class Erreurs {
    public static void main(String[] args) {
        Vector<Double> liste = new Vector<Double>();
        boolean b = true;
        while (b) {
            liste.add(2.0);
        }
```

Classe Error

génère l'erreur suivante :

```
Exception in thread "main" java.lang.OutOfMemoryError:
Java heap space
at java.util.Arrays.copyOf(Arrays.java:3210)
at java.util.Arrays.copyOf(Arrays.java:3181)
at java.util.Vector.grow(Vector.java:266)
at java.util.Vector.ensureCapacityHelper(Vector.java:2
at java.util.Vector.add(Vector.java:782)
at chap10.ClasseError.main(ClasseError.java:10)
```

Classe Exception

La classe **Exception** représente les erreurs les moins graves qu'on peut gérer dans les programmes.

Le programme suivant :

```
import java.util.Scanner;
public class SaisieEntier {
    public static void main(String[] args) {
        int n;
        Scanner clavier = new Scanner(System.in);
        System.out.print("Saisir un entier : ");
        n=clavier.nextInt();
        System.out.println("1/" + n + " = " + 1/n);
        clavier.close();
    }
}
```

Classe Exception

génère l'erreur suivante, si n=0 :

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
at chap10.SaisieEntier.main(SaisieEntier.java:11)
```

Si on saisi un caractère au lieu d'un entier, le programme précédent génère l'erreur suivante :

```
Saisir un entier : a
Exception in thread "main" java.util.InputMismatchExce
at java.util.Scanner.throwFor(Scanner.java:864)
at java.util.Scanner.next(Scanner.java:1485)
at java.util.Scanner.nextInt(Scanner.java:2117)
at java.util.Scanner.nextInt(Scanner.java:2076)
at chap10.SaisieEntier.main(SaisieEntier.java:10)
```

Types d'exceptions

Dans ce qui suit, nous donnons quelques types d'exceptions :

ArithmeticException

Erreur arithmétique, comme une division par zéro.

ArrayIndexOutOfBoundsException

Indice d'un tableau qui dépasse les limites du tableau. Par exemple :
`double [] tab = new double[10];`
`tab[10]=1.0;` ou bien `tab[-1]=1.0;`

NegativeArraySizeException

Tableau créé avec une taille négative. Par exemple :
`double [] tab = new double[-12];`

Types d'exceptions

ClassCastException : cast invalide. Par exemple :

```
Object [] T = new Object[2];

T[0] = 1.0;
T[1] = 2;

//Trie d'un tableau qui contient des entiers et des
   reelles
Arrays.sort(T);
```

NumberFormatException

Mauvaise conversion d'une chaîne de caractères vers un type numérique. Par exemple :

```
String s = "tt";
x = Double.parseDouble(s);
```

Types d'exceptions

StringIndexOutOfBoundsException

Indice qui dépasse les limites d'une chaîne de caractères. Par exemple :

```
String s = "tt";
char c = s.charAt(2);
```

ou bien

```
char c = s.charAt(-1);
```

NullPointerException

Mauvaise utilisation d'une référence. Par exemple utilisation d'un tableau d'objets créé mais non initialisé :

```
ClasseA [] a = new ClasseA[2]; // ClasseA une classe
a[0].x = 2;   // l'attribut x est public
```

Méthodes des exceptions

Dans ce qui suit, une liste de méthodes disponible dans la classe **Throwable** :

Méthode	**Description**
public String getMessage()	Retourne un message concernant l'exception produite.
public String toString ()	Retourne le nom de la classe concaténé avec le résultat de « getMessage() »
public void printStackTrace()	Affiche le résultat de « toString() » avec la trace de l'erreur .

Capture des exceptions

Pour les différents tests, le programme s'est arrêté de façon brutale. Il est possible de capturer (to catch) ces exceptions et continuer l'exécution du programme en utilisant les 5 mots clés **try**, **catch**, **throw**, **throws** et **finally**.

Forme générale d'un bloc **try**

```
try {
    //Code susceptible de generer une erreur
} catch (TypeException1 excepObj) {
    //traitement en cas d'exception de type TypeException1
} catch (TypeException2 excepObj) {
    //traitement en cas d'exception de type TypeException2
//  ...
finally{
//code a executer avant la fin du bloc try
}
//code après try
```

TypeException est le type d'exception généré. **excepObj** est un objet.

Remarque :

TypeException peut-être une classe prédéfinie de Java ou une classe d'exception créée par l'utilisateur.

Un seul catch

Lors de l'exécution du programme :

```
public class DiviseZero {
    public static void main(String[] args) {
        int n = 0;
        System.out.println("1/" + n + " = " + 1/n);
    }
}
```

on a obtenu l'exception **ArithmeticException**.

Pour capturer cette exception, on pourra modifier le programme en utilisant le bloc **try** de la façon suivante :

```
public class DiviseZero {
  public static void main(String[] args) {
     int n = 0;
     try {
        System.out.println("1/" +n+ " = " + 1/n);
        System.out.println("Ne sera pas affiche");
     } catch (ArithmeticException ex) {
        System.out.println("Division par zero");
     }
     System.out.println("Reste du programme");
  }
}
```

plusieurs catch

Reprenons le programme :

```
import java.util.Scanner;
public class Exceptions {
    public static void main(String[] args) {
        int n;
        Scanner clavier = new Scanner(System.in);
        System.out.print("Saisir un entier : ");
        n=clavier.nextInt();
        System.out.println("1/" + n + " = " + 1/n);
        System.out.println("Fin du programme");
        clavier.close();
    }
}
```

Traitons les différentes exceptions générées par le programme et améliorons ce dernier pour forcer l'utilisateur à saisir un entier :

```
//Pour pouvoir utiliser "InputMismatchException"
import java.util.InputMismatchException;
import java.util.Scanner;

public class Exceptions {
public static void main(String[] args) {
  int n;
  Scanner clavier = new Scanner(System.in);
  boolean b = true;
  while (b) {
    try {
      System.out.print("Saisir un entier : ");
      n = clavier.nextInt();
      System.out.println("1/" + n + " = " + 1 / n);
      break;
    }
```

Suite du programme

```
 catch (ArithmeticException e) {
    System.out.println("Division impossible par 0");
    System.out.println(e.getMessage());
 } catch (InputMismatchException e) {
    System.out.println("Vous n'avez pas saisi un
        entier");
    System.out.println(e);
    //equivalent a System.out.println(e.toString())
    //pour recuperer le retour a la ligne
    clavier.nextLine();
 }
}//Fin de while
 System.out.println("Fin du programme");
 clavier.close();
}
```

Remarque :

Puisque les classe **ArithmeticException** et **InputMismatchException** sont des sous classes de la classe **Exception**, on peut les combiner dans bloc **catch** générique qui utilise la classe **Exception**

```
public class ExceptionsGeneriques {
  public static void main(String[] args) {
     int n;
     Scanner clavier = new Scanner(System.in);
     try {
        System.out.print("Saisir un entier : ");
        n = clavier.nextInt();
        System.out.println("1/" + n + " = " + 1/n);
     } catch (Exception e) {
        System.out.println("Erreur\n"+e.toString());
     }
     System.out.println("Fin du programme");
     clavier.close();
  }
}
```

Bloc finally

Le bloc **finally** est toujours exécuté, même si aucune exception ne s'est produite.

Exemple :

```
public class BlocFinally {
    public static void main(String[] args) {
      int tab[] = new int[2];
      try {
          tab[2] = 1;
      } catch (ArrayIndexOutOfBoundsException e) {
          System.out.println("Exception : ");
          e.printStackTrace();
          //ou bien e.printStackTrace(System.out);
      } finally {
        tab[0] = 6;
        System.out.println("tab[0] = " + tab[0]);
        System.out.println("Le bloc finally est
            execute");
     }
    }
}
```

Blocs try imbriqués

On peut mettre un bloc **try** dans un autre bloc **try**.

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class MultipleTry {
    public static void main(String[] args) {
```

```
try {
    //Permet d'ouvrir un fichier en ecriture
    BufferedWriter out = new BufferedWriter(
        new FileWriter("test.txt"));
    out.append("Simple test\n");
    for (int i = 0; i < 5; i++)
        out.append("i = " + i + "\n");
    out.close();
} catch (IOException e) {
    e.printStackTrace();
}
```

```
try {
    //Permet d'ouvrir un fichier en lecture
    BufferedReader fichier = new BufferedReader(
        new FileReader("test.txt"));
    String ligne;

    try {
        //lecture du fichier ligne par ligne
        while((ligne = fichier.readLine()) != null){
            System.out.println(ligne);
        }
    } catch (IOException e) {
            e.printStackTrace();
    }

} catch (FileNotFoundException e1) {
        e1.printStackTrace();
}
```

Exceptions personnalisées

Soit la classe **Personne** définie comme suit :

```
class Personne {
    private String nom, prenom;
    private int age;
    public Personne(String nom, String prenom, int
        age){
      this.nom = nom.toUpperCase();
      this.prenom =
          prenom.substring(0,1).toUpperCase()
          + prenom.substring(1).toLowerCase();
      this.age = age;
    }
    public String toString() {
      return nom + " " + prenom + " " + age + " ans";
    }
}
```

Gestion d'une seule exception

Supposons qu'on veut générer une exception si l'utilisateur veut instancier un objet **Personne** avec une valeur négative pour l'age. Par exemple :

```
Personne p = new Personne("Oujdi","Ali",−9)
```

Pour cette raison, nous allons définir une nouvelle classe, appelée **AgeException** qui hérite de la classe **Exception** :

```
class AgeException extends Exception {
    public AgeException() {
        System.out.println("L'age ne doit pat etre
            negatif");
    }
}
```

La classe **Personne** va être modifiée de la façon suivante :

```
class Personne {
    private String nom, prenom;
    private int age;
    public Personne(String nom, String prenom, int
       age)
     throws AgeException {
        if (age < 0)
            throw new AgeException();
        else {
            this.nom = nom;
            this.prenom = prenom;
            this.age = age;
        }
    }
    ...
}
```

Remarque :

Dans une méthode, une instruction de type :

```
Personne p = new Personne("Oujdi","Ali",18);
```

génère une erreur de compilation du fait que le constructeur génère des exceptions, il faut gérer les exceptions en utilisant le bloc **try**.

Exemple :

```
public static void main(String[] args) {
   try {
      Personne p = new Personne("Oujdi", "Ali", 18);
      System.out.println(p);
   } catch (AgeException e) {
      System.out.println(e);
      System.exit(-1);
   }
}
```

Gestion de plusieurs exceptions

Supposons maintenant qu'on veut, en plus de l'exception pour l'age, on veut générer une exception si l'utilisateur veut instancier un objet **Personne** avec un **nom** et/ou un **prénom** qui contiennent un chiffre (par exemple : Personne p = new Personne("Oujdi12","Ali",19)).

Nous allons définir une nouvelle classe nommé **PersonneException** comme suit :

```
class PersonneException extends Exception {
    public PersonneException() {
        System.out.println("Un nom ne doit pas
            contenir de chiffres!");
    }
}
```

La classe **Personne** va être modifiée de la façon suivante :

```
class Personne {
  private String nom, prenom;
  private int age;

  public Personne(String nom, String prenom,
    int age)throws AgeException, PersonneException {
    boolean b = false;

    for (int i = 0; i <= 9; i++)
      if (nom.contains(String.valueOf(i))
          || prenom.contains(String.valueOf(i))) {
        b = true;
        break;
      }
```

suite de la classe Personne

```
    if (age < 0)
        throw new AgeException();
    else if (b)
        throw new PersonneException();
    else {
        this.nom = nom;
        this.prenom = prenom;
        this.age = age;
    }
  }
    ...
}
```

Puisque le constructeur doit gérer deux exceptions, on doit les tester tous les deux lors de l'instanciation d'un nouveau objet :

Exemple :

```
public static void main(String[] args) {
    try {
        Personne p = new Personne("OUjdi", "a1Li",
            12);
        System.out.println(p);
    } catch (AgeException e) {
        System.out.println(e);
        System.exit(-1);
    } catch (PersonneException e) {
        System.out.println(e);
        System.exit(-1);
    }
}
```

Spécifier l'exception qu'une méthode peut générer

Une méthode qui peut générer une exception mais qui ne peut être capturée mais qui peut être capturée par une autre méthode doit spécifier l'exception en utilisant le mot clé **throws**.

Exemple :

Soit la classe **Etudiant** définie comme suit :

```
class Etudiant {
    private String nom, prenom, cne;
    private double[] note = new double[6];

    public Etudiant(String nom, String prenom,
        String cne) {
        ...
    }

    public void setNote(int i, double x) {
        note[i] = x;
    }
    ...
}
```

Supposons qu'on veut générer une exception si l'utilisateur veut mettre une note différente de -1 ou non comprise entre 0 et 20 (par exemple : setNote(3,29) ou setNote(3,−10)).

Pour cette raison, nous allons définir une nouvelle classe, appelée **NoteException** qui hérite de la classe **Exception** :

```
class NoteException extends Exception {
    public NoteException() {
        System.out.println("Une note doit etre egale
            a -1 ou comprise entre 0 et 20");
    }
}
```

La classe **Etudiant** va être modifiée de la façon suivante :

```
class Etudiant {
    ...
    public void setNote(int i, double x) throws
        NoteException {
        if ((x != -1) && (x < 0 || x > 20))
            throw new NoteException();
        else
            note[i] = x;
    }
    ...
}
```

Pour une utilisation dans une méthode « main() », on appelle la méthode « setNote() » comme suit :

```
try {
    et1.setNote(3, 23);
    System.out.println(et1.getNote(3));
} catch (NoteException e) {
    System.out.println(e);
}
```