INTRODUCTION À JAVA PARTIE 2

I.AZAIEZ

Février 2024

1

Vue d'ensemble

1. Les operateurs
2. Les Strings
3. Les exceptions
4. Les classes abstraites
5. Les interfaces
5. Design patterns

2

Les opérateurs de calculs

3

Il existe cinq opérateurs de calcul de base permettant de modifier mathématiquement la valeur d'une variable. Chacun est utilisé entouré de deux opérandes.

Opérateur	Opération	exemple
+	Addition	x+2
-	Soustraction	x-5
*	Multiplication	x*3
/	Diviser	x/7
%	Modulo (reste)	x%2

3

Les opérateurs de comparaison

4

Opérateur	Opérateur	Exemple	Résultat (booléen)
==	Égalité (ne pas confondre avec l'affectation =)	x==6	1 si x vaut "6", sinon 0
!=	Inégalité	x!=6	1 si x ne vaut pas "6", sinon 0
>	Supériorité stricte à	x>6	1 si x est supérieur à "6", sinon 0
>=	Supériorité à	x>=6	1 si x est supérieur ou égal à "6", sinon 0
<	Infériorité stricte à	x<6	1 si x est inférieur à "6", sinon 0
=<	Infériorité à	x<=6	1 si x est inférieur ou égal à "6", sinon 0

4

Les opérateurs logique

5

Opérateur	Nom	Opération	syntaxe
||	OU logique	Teste si UNE (au moins une) des conditions est réalisée	((condition1) II (condition2))
&&	ET logique	Teste si TOUTES les conditions sont réalisées	((condition1) && (condition2))
!	NON logique	S'applique à un booléen. Renvoie 1 si le booléen est false (0) et 0 si le booléen est true (1). Il inverse donc l'état du booléen	(!booléen)

5

Les opérateurs logiques

6

Ces opérateurs logiques sont cumulables.

On peut leur adjoindre l'opérateur "^" (OU exclusif ou XOR) qui teste si l'une des conditions est réalisée, mais qui ne renverra true que si l'autre condition N'EST pas réalisée (il faut donc que l'une des expressions soit vraie et l'autre fausse, alors que pour le OU logique || il suffit que l'une des deux soit vraie (donc les deux peuvent être vraies)).

Cependant, ce dernier opérateur (ainsi que d'autres: "|", "&") n'est en général employé que dans le cadre d'opérations logique "bit-à-bit" (opérations dans lesquelles on ne travaille pas au niveau des données typées dans leur globalité, mais directement au niveau des bits qui les composent. Comparaisons très pointues).

6

Les opérateurs pour les String

7

Java ne permet pas les surcharges d'opérateurs, une exception cependant: dans le cas des chaînes (Strings), le + est utilisé pour concaténer plusieurs Strings.

Rappelons que ces dernières sont des objets et que la classe String implémente un certain nombre de méthodes applicables aux objets String (en C++, les Strings sont des tableaux de caractères).

- Ex: **System.out.println("Hello" + "world");**

Dans ce cas, nous aurions pu coder "Hello world". L'intérêt principal de cette surcharge est de pouvoir concaténer des variables de type chaînes: ce sont leurs valeurs qui seront affichées.

- Ex: dans la classe Moto, nous avions deux variables de type String marque et couleur
- **System.out.println("La moto" + marque + "est" + couleur + ".");**

7

Les opérateurs pour les String

8

Si l'un des éléments concaténés n'est pas une chaîne, il est théoriquement converti en String en vu de l'affichage (avec plus ou moins de bonheur). Ceci est valable pour les types de base, mais également pour les objets.

- Un objet ou un type peut être converti via la méthode toString().
- Tout objet possède une représentation sous la forme d'une chaîne par défaut, cependant, la plupart des classes redéfinissent toString().

Afin d'assurer un résultat correct, il est souvent intéressant de redéfinir cette méthode dans le cadre des classes que l'on développe. La fonction devra indiquer quels éléments (caractéristiques) des objets devront être présentés au moment ou l'on tentera "d'afficher" ces objets (et comment le faire).

- L'écriture "S + O", avec l'élément de gauche (ici, S) qui est une String et celui de droite un objet, est équivalente à "S + (O.toString())". L'appel à toSting() est automatique.
- Il est donc plus rapide d'écrire ""+O que O.toString() (mais pas forcément plus clair).

8

Les opérateurs pour les String

9

Ex: les objets issus de la classe Moto ont des attributs marque et couleur. On peut redéfinir dans la classe Moto la méthode toString() fournie par défaut:

```
public String toString(){
   String s =("Moto"+this.marque+" "+this.couleur);
   return s;
}
```

Il sera alors possible "d'afficher une moto" de la manière que l'on souhaite. Pour l'objet moto_1:

System.out.println("nous avons une"+moto_1);

- Affichera, par exemple, "nous avons une moto Yamaha verte".

9

Les opérateurs pour les String

10

L'opérateur += est utilisable avec les Strings.

```
String s ="ceci est";
s+= "un exemple."; /* ajout de "un exemple" à la valeur
          de s et le résultat est affecter à s*/
System.out.println(s); /* affiche "ceci est un exemple*/
```

Pour les Strings, l'opérateur d'affectation est également "=". En revanche, la comparaison (l'égalité, plus précisément) est gérée par la méthode equals() de la classe String (et non pas par ==. Ce dernier existe, mais retourne true si les deux variables sont des références sur le même objet.

```
if (s.equals("table")){
/* code exécuté uniquement si la valeur de la String s est "table"
(attention aux majuscules)*/
}
```

10

Les opérateurs pour les String

11

La méthode compareTo() est équivalente à strcmp en C/C++

Quelques méthodes complémentaires :

- length() : renvoie la longueur de la String
- indexOf() : renvoie la position d'un caractère
- substring(): extraction d'une sous-chaîne
- toUpperCase() : mise en majuscules
- toLowerCase() : mise en minuscules

11

Exemple : String

12

indexOf

- String ch = "GCR1";
- ch.indexOf('G') renvoie le rang de la première apparition du caractère 'G' dans la chaîne (ou -1 si G n'est pas trouvé)
- ch.indexOf("GCR") renvoie le rang de la première apparition de l'objet String "GCR" dans ch (ou -1 si "GCR" n'est pas trouvé)

substring

- ch.substring(1,2) renvoie un nouvel objet String représentant la sous-chaîne extraite de ch depuis le caractère de rang 1 (le deuxième) jusqu'au caractère de rang 3, s'il existe.

12

Les fonctions pour les String

13

Java dispose d'une classe ***StringBuffer*** destinée elle aussi à la manipulation de chaînes, mais dans laquelle les objets sont modifiables.

Un objet de type ***StringBuffer*** peut modifier la chaine qu'il contient pour l'agrandir (par insertion ou par concaténation) ou la réduire

13

Les fonctions pour les String

- Il existe des méthodes :
  - de modification d'un caractère de rang donné : *setCharAt*,
  - d'accès à un caractère de rang donné : *charAt*,
  - d'ajout d'une chaîne en fin : la méthode *append* accepte des arguments de tout type primitif et de type String,
  - d'insertion d'une chaîne en un emplacement donné : *insert*,
  - de remplacement d'une partie par une chaîne donnée : replace,
  - de conversion de StringBuffer en String : *toString*.

14

Exemple : String

14

- Voici un programme utilisant ces différentes possibilités :

les StringBuffer

```
class Test {
public static void main (String args[]) {
        String ch = "la java" ;
        StringBuffer chBuf = new StringBuffer (ch) ;
        System.out.println (chBuf) ;
        chBuf.setCharAt (3, 'J'); System.out.println (chBuf) ;
        chBuf.setCharAt (1, 'e') ; System.out.println (chBuf) ;
        chBuf.append (" 2") ; System.out.println (chBuf) ;
        chBuf.insert (3, "langage ") ; System.out.println (chBuf) ;} }
// Résultat
```

☐ la java ☐ la Java ☐ le Java ☐ le Java 2 ☐ le langage Java 2

15

Les Exceptions

16

Les exceptions

18

- Exception = événement exceptionnel, sous-entendu, événement inattendu, donc généralement erreur. Pourtant, ce n'est pas une erreur au sens "mauvais codage", mais plus généralement un comportement incorrect, non désiré.
- Au moment de l'exécution (la compilation est correcte).
- Java permet au développeur de gérer certaines de ces exceptions de manière personnalisée, soit pour les identifier plus aisément en cours d'exécution, soit pour implémenter une parade.
- Exemples d'exception :
  - **IOException:** problème au niveau des entrées-sorties (fichier inconnu ou protégé en écriture/lecture, par exemple)
  - **IndexOutOfBoundsException**: tentative d'accéder une position dans un tableau qui dépasse les limites de celui-ci.
  - **ArithmeticException:** problème lors de l'exécution d'une expression mathématique (typiquement, la division par zéro)
  - Voir

http://download.oracle.com/javase/1.5.0/docs/api/java/lang/Exception.html

17

Les exceptions

19

- Une exception est un objet instancié au moment du dysfonctionnement à partir de la classe

  Java.lang.exception

- Cette dernière hérite de java.lang.Throwable (tout comme java.lang.Error)
- Certaines exceptions doivent obligatoirement être traitées (le compilateur le signale le cas échéant), d'autres de façon optionnelle (et le compilateur ne signale rien).

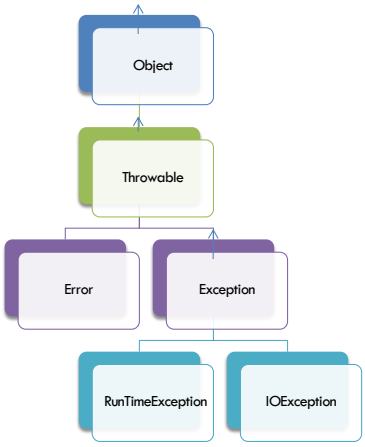
Donc, pour certains types d'exceptions, le compilateur ne permettra pas d'utiliser le code si elles ne sont pas explicitement gérées par le développeur.

18

Les exceptions

20

Un extrait de la hiérarchie des exceptions



19

Les exceptions

21

- La classe exception possède un grand nombre de sous-classes qui en dérivent et la spécialisent. Parmi elles, on distingue RuntimeException des autres classes filles.
- RuntimeException est elle-même la superclasse de plusieurs classes la spécialisant (ex: ArithmeticException et IndexOutOfBoundsException): l'ensemble de ces classes n'appellent pas à un traitement obligatoire dans le code. Le compilateur compilera le programme sans problème.
- En revanche, les sous classes de Exception (autres que RuntimeException et ses dérivées) nécessitent obligatoirement une gestion explicite au sein du code (ex: IOException). Dans le cas contraire, le compilateur refusera de compiler le programme.

20

Les exceptions

22

- public class TestDivision
```
{
    public static void main(String[] args)
    {
        int numerateur = 3;
        int denominateur = 0;
        System.out.println(numerateur/denominateur);
    }
}
```
- Il est évident que le programme échouera dans son exécution au moment ou une division par zéro va être tentée. Une exception de type ArithmeticException va être levée (un objet de type ArithmeticException va être instancier et constituera le résultat de l'exécution) ce qui aboutira à une cessation d'activité et un message d'erreur. Ce message comportera notamment la mention "/ by zero" fournit par l'objet.
- En revanche, ce type d'exception est dérivée de RuntimeException: la compilation ne posera aucun problème.

21

Les exceptions

23

- public class TestEntree
```
{
    public static void main (String args[])
    {
            System.in.read();
    }
}
```
- Ici, le système est en attente d'une action sur l'entrée standard (par défaut, le clavier). Tant qu'une touche ne sera pas pressée, le programme ne s'arrêtera pas.
- L'instruction est susceptible de lever une exception du type IOException, qui ne dérive pas de RuntimeException: le compilateur n'acceptera pas de compiler ce programme si l'exception n'est pas explicitement traitée.

22

Les exceptions

24

- Il apparaît donc deux cas ou la gestion d'une exception par le programmeur peut se faire:
  - Soit parce que cela est obligatoire
  - Soit parce que cela n'est pas obligatoire mais qu'il y a intérêt à le faire tout de même(code plus "propre", message d'erreur plus explicite, contrôle en finesse des sections critiques du programme, …). Cependant, la gestion à outrance des exceptions n'est pas non plus préconisée, le mécanisme pouvant se révéler lourd en ressources et en temps d'exécution.
- Pour cela, on utilise un couple d'instructions **try…catch** (essayer…attrape).
- **try** définit un bloc {…} qui enveloppe la ou les instruction(s) susceptible(s) de lever une exception.
- **catch** définit un bloc {…} où sont codées les actions à effectuer si effectivement l'exception attendue apparaît.
- Lorsque le bloc try lève une exception, le bloc catch (qui doit lui faire directement suite) "capture" l'objet exception et lance les actions à effectuer : le programme ne s'arrête pas.

23

Les exceptions

25

- Pour le premier exemple:

```
public class TestDivision
{
 public static void main(String[] args){
            int numerateur = 3;
            int denominateur = 0;
   try
   {
     System.out.println(numerateur/denominateur);
     System.out.println("message après la division");
   }
   catch(ArithmeticException e)
   {
     System.out.println("exception arithmétique");
   }
     System.out.println("message après catch");
   }
}
```

24

Les exceptions

26

- Ici, la division susceptible de lever l'exception est placée dans un bloc try.
- Le bloc catch qui le suit directement s'attend à recevoir une exception e de type ArithmeticException lancée par un bloc try (et seulement ce type d'exceptions précis)
- Si la division se déroule mal (ce qui est le cas):
  - Les instructions du bloc try qui la suivent ne sont pas exécutées.
  - Le bloc catch capture l'exception levée par le bloc try et les instructions qu'il contient sont exécutées.
  - Les instructions qui suivent ces deux blocs sont exécutées (alors que si l'exception n'était pas gérée, le programme s'arrêterait au moment de l'erreur).
  - Le résultat affiché est:

    Exception arithmétique

    Message après le catch

25

Les exceptions

27

- Pour le second exemple:

```
import java.io.*;       //pour gérer les entrées sorties

public class TestEntree {
   public static void main(String[] args)
   {
            try
            {
                     systeme.in.read();
            }
            catch(IOException e)
            {
                     System.out.println("Exception d'E/S");
            }
   }
}
```

26

Les exceptions

28

- Le processus est identique à deux éléments près:
  - L'exception attendue n'est pas la même et il faut adapter le bloc catch
  - Il est nécessaire d'importer java.io.IOException pour pouvoir gérer l'exception (mais si on importe déjà java.io.*, inutile car la classe IOException est déjà comprise dans la package java.io).
- Plusieurs blocs catch peuvent se suivre si l'on s'attend à ce que les instructions du bloc try lèvent plusieurs types d'exceptions. Cependant, ces blocs catch doivent être disposés d'une certaine manière: ceux qui capturent les exceptions les plus spécialisés (sous classe) avant ceux qui capturent les exceptions les plus générales (superclasses).

```
try{…}
catch(ArithmeticException e) {…}
catch(IOException e) {…}
catch(Exception e) {…}
```

27

Les exceptions

29

- Un bloc catch peut être suivi par un bloc finally{…}. Celui est optionnel, mais les instructions qu'il contient sont assurées d'être exécutées, qu'une exception soit levée ou non.
- Cela peut alors accorder une sécurité supplémentaire par rapport à certaines instructions cruciales pour la suite du programme (ex: fermer proprement un fichier ou une connexion à une base même si une exception est levée).
- Comme catch doit suivre directement try (sans instruction entre les deux), finally doit suivre directement catch.

28

Les exceptions

30

- Il est possible de préférer ne pas traiter l'exception dans la partie de code en cause mais de tenter de la faire traiter ailleurs.
- Ainsi, dans une méthode qui contient une instruction susceptible de lever une exception, on peut se dispenser d'entourer cette instruction d'un bloc try associé à un bloc catch.
- Le mécanisme, dans ce cas, est de renvoyer l'exception levée à la méthode qui a appelé le code en cause en espérant qu'elle sera la traiter…
- De la même manière que pour les blocs try…catch, ce mécanisme est obligatoire pour les exceptions dérivant de la classe Exception mais pas pour celles dérivant de RuntimeException. De ce fait, pour les exceptions de la première catégorie, il faut traiter l'exception soit par try…catch, soit par renvoi (sinon, le compilateur ne compile pas).

29

Les exceptions

31

- Pour faire cela, il faut adjoindre le mot clé throws (avec s) suivi du type d'exception attendue dans la déclaration de la méthode:

```
public void maMethode() throws IOException{…}
```

  - Cela indique que si, au sein de la méthode, une exception de type IOException est levée et que celle-ci n'est pas traitée par un bloc catch, elle est adressée aux méthodes "supérieures" afin d'être gérée.
- L'exception peut ainsi "voyager" jusqu'à la méthode main si aucune méthode ne la traite auparavant. Si celle-ci ne la traite pas non plus, ce sera la machine virtuelle qui le fera, entraînant une interruption de l'exécution du programme.

30

Les exceptions

32

Exemple

```
public class TestDivision {
    public static void main (String args[]) {
       int a =6, b=0, z=0;
       try {
                z=methode1(a,b);
                System.out.println(z);
       }
       catch(ArithmeticException e) {
                System.out.println("Exception arithmétique");
    }
    } // fin main
    public static int methode1(int c, int d) throws ArithmeticException {
       return methode2(c,d);
    }
    public static int methode2(int e, int f) throws ArithmeticException {
       return (e/f);
    }
 } //fin de la classe
```

31

Les exceptions

33

- Dans le cas présent, l'appel de la methode1 entraîne l'appel de la methode2
- Cette dernière aboutit à une ArithmeticException (division par zéro). Mais cette méthode ne la gère pas, elle l'envoie (grâce à throws) à la méthode qui a appelé methode2, c.-à-d. methode1.
- Methode1 ne gère pas l'exception: elle l'envoie à la méthode qui l'a appelée, c.-à-d. main.
- Au niveau de main, l'appel de methode1 était dans un bloc try: l'exception va être traitée par le bloc catch correspondant, ce qui aboutit à l'affichage "Exception arithmétique".

32

Les exceptions

34

- Les objets Exception implémentent un certain nombre de fonctions. Parmi elles :
  - toString()
  - getMessage()
  - printStackTrace()
- Elles peuvent être utiles à appeler dans le bloc catch afin d'avoir des informations complémentaires.
- getMessage() permet d'obtenir le message standard lié à l'erreur rencontrée et normalement affiché par le compilateur:

  System.out.println("Exception rencontrée:"+e.getMessage());

  Par exemple, pour une division par zéro, cela affichera "Exception rencontrer : / by zero"
- e.printStackTrace() affichera l'état de la pile, c'est-à-dire le cheminement de l'exception si elle se voit rejetée à travers plusieurs méthodes. Le développeur pourra ainsi déterminer les méthodes parcourues et accéder à celle qui a levée l'exception.

Les exceptions

35

- Une autre manière de faire afin de rejeter une exception aux méthodes supérieures peut être de coder un bloc catch au sein duquel on place uniquement la commande throw (sans s)

```
catch (typeException e) trows typeException{
        throw new(typeException(args));
}
```

Les exceptions

36

- L'une des caractéristiques les plus intéressantes de java est de permettre de créer ses propres types d'exceptions et de les gérer.
- Pour cela, il est nécessaire de créer une classe monException héritant de la classe Exception. L'ensemble des méthodes de la seconde est alors transmis à la première.

```
public class monException extends Exception
{
    public monException(String s)
    {
        super(s);
    }
}
```

35

Les exceptions

37

- Ici, nous créons un nouveau type d'exception avec un constructeur (ex: une exception qui se déclenche si le résultat d'un calcul est négatif ou s'il est dans une certaine plage de valeurs). Ce dernier appelle en fait le constructeur de la superclasse Exception afin que celui-ci instancie un objet de type *Exception* mais de type personnalisé avec le message passé en paramètre (String s)
- Le fichier .java est compilé et le fichier .class est importé dans le fichier de l'application.

36

Les exceptions

38

- Voici à quoi peut ressembler une application gérant monException:

```
import … .monException;
class monAppli {
   public static void main (String[] args) {
      …
      try {
      maMethode();      //risque de lever monException
      }
      catch (monException e) {
      System.out.println(e);
      }
}
public static typeRetour maMethode() throws monException
   …
   if(condition) {
                  throw new monException("monException rencontrée");
   }
   … } }//renvoi un nouvel objet : mon exception avec le message
```

37

Question

39

Les exceptions permettent de gérer les problèmes liés :

A. Aux instructions pouvant provoquer des erreurs d'exécutions
B. A la structure de contrôle try {…} catch {…}
C. A des erreurs de programmation rares
D. A l'instruction : test d'arrêt et appel récursif.

38

Question

40

```
1. public class Programme {
2. public static void main(String args[]){
3. try {
4. System.out.print("bonjour ");
5. }
6. finally {
7. System.out.println("exécution de Finally ");
8. }
9. }
10. }
```

Quelle est le résultat?

A. Rien. Le programme ne compile pas car aucune exception n'est spécifiée.
B. Rien. Le programme ne compile pas car aucune clause catch n'est spécifié.
C. bonjour
D. bonjour exécution de Finally

39

Les classes abstraite

40

Classes abstraites

42

- Une classe abstraite est une classe qui ne permet pas d'instancier des objets.
- Elle ne peut servir que pour des classes de base pour une dérivation.

```
abstract class Forme
{  public void f() { …} // f définie dans Forme
   public abstract void calculAir(); //claculAir n'est
                                     //pas définie dans Forme;
                                      //on n'en a fourni que l'entête.
}
```

41

Classes abstraites

43

- Une classe contenant une ou plusieurs méthodes abstraites, elle est abstraite.

```
abstract class A
{ public abstract void f();
        …
}//A classe abstraite
```

- Une méthode abstraite doit obligatoirement être déclarée public.
- Dans l'entête d'une méthode déclarée abstraite, les noms d'arguments effectifs doivent figurer.

```
abstarct class A {
public abstract void g(int);// erreur → g(int n)
}
```

42

Classes abstraites

44

Exemple

```
public abstract class Forme
{   public void f() { ...} // f définie dans Forme
    public abstract void calculAir(); //claculAir n'est
                                      //pas définie dans Forme;
                                //on n'en a fourni que l'entête.
}
public class Cercle extends Forme {
    public void calculAir(){
    ... }      // redéfinition de calculAir est obligatoire
               //pas de redéfinition de f }
```

43

Classes abstraites

45

- Une classe dérivée d'une classe abstraite n'est pas obliger de redéfinir les abstraites de sa classe de base.

```
abstract class A {
 public abstract void f1();
 public abstract void f2(char c);
 }

 abstract class B extends A
 { public void f1() {...}      // abstract obligatoire
                               // redéfinition de f1
                               //pas de redéfinition de f2
 }
```

- Une classe dérivée d'une classe non abstraite peut être déclarée abstraite et/ou contenir des méthodes abstraites.

44

LES INTERFACES

45

Les Interfaces

47

- Les Interfaces constituent un moyen d'attribuer un comportement a une classe. Elles permettent de définir des propriétés supplémentaires de cette classe et d'augmenter ses fonctionnalités.
- les interfaces ne sont pas des classes, elles ne sont pas instanciables (mais elles doivent être compilées). Elles sont constituées comme des ensembles de fonctionnalités pouvant être accordées à des classes.
- Dans le cadre de l'héritage, une sous-classe hérite des méthodes de sa superclasse. La relation qui lie les deux classes est du type "est une sorte de": un objet de la sous-classe "est une sorte" d'objet de la super-classes. Dans certains cas, ces deux types d'objets sont presque utilisables indifféremment (un objet de la sous-classe peut être utilisé à la place d'un objet de la super-classes).
- Dans un cas d'implémentation d'interface (classe utilisant les fonctionnalités d'une Interface), la classe acquiert simplement les fonctions décrites dans l'interface, mais ne devient pas un objet de type de l'interface. C'est uniquement comme un "emprunt" de méthode

46

Les Interfaces

48

- Une classe utilisant une interface est considérée comme "implémentant " cette interface.
- Une interface est un ensemble de méthodes *abstraites*, caractériser uniquement par leurs définitions sans aucune réalisation.
- Pour chaque méthode est précisé seulement son prototype, c'est-à-dire les droits d'accès si besoin, le type de retour, le nom et les éventuels arguments. Le corps est laissé vide. Le prototype est suivi d'un point-virgule.
- Ex:

```
public int maMethode (String nom, int cœfficient);
```

- En tant que telles, ces méthodes abstraites ne peuvent être utilisées. Il faudra donc les redéfinir autre part en codant leur réalisation.

47

Les Interfaces

49

- L'implémentation d'une interface par une classe obligera donc à coder au sein de cette dernière les réalisations de TOUTES les méthodes abstraites de l'interface. On utilisera donc de préférence des interfaces de tailles réduites.
- Il apparaît donc que ce n'est pas réellement des fonctionnalités qui sont transmises à la classe (les réalisations des méthodes ne sont pas fixées), mais plutôt un comportement.
- Par exemple, une classe A doit forcément réaliser plusieurs fonctionnalités précises: on pourra lui faire implémenter une interface donnée décrivant les prototypes de ces méthodes. Les noms de ces dernières, leurs types de retour, etc. sont fixées et l'utilisateur ne pourra pas modifier ces éléments ce qui lui impose des règles dans sa programmation.

48

Les Interfaces

50

- Dans le fichier essaiInterface.java (à compiler):

```
public interface essaiInterface { // déclaration d'une interface
    public void MethodeDeLinterface ();
}
```

- Dans le fichier MaClasse.java:

```
//Exemple d'implémentation de l'interface  essaiInterface par une classe

public class MaClasse implements essaiInterface {
        public void Methode1() { System.out.println ("Méthode 1");}
        public void MethodeDeLinterface() {
                System.out.println ("Méthode de l'interface"); }
        public void Methode2(){ System.out.println(Methode 2"); }
}
```

49

Les Interfaces

51

- Une classe peut implémenter plusieurs interfaces: elle cumule alors leurs méthodes. Ceci permet de simuler l'héritage multiple du C++.
- Plusieurs classes peuvent Implémenter la même interface. C'est même là l'un des Intérêts principaux du mécanisme. De ce fait, ces classes partagent des fonctionnalités identiques, dont les noms, les arguments et les types de retour sont identiques, ce qui assure une cohérence d'utilisation entre ces classes. Mais ces dernières restent bien distinctes (le reste de leurs méthodes leur est propre). Elles ne sont pas liées par une relation comme dans le cas d'un héritage.
- Dans ce dernier cas, les réalisations des classes abstraites étant codées au sein des classes implémentant l'interface, il est possible d'adapter ces réalisations à la classe concernée.

50

Les Interfaces

52

- Exemple: plusieurs Classes représentent des formes géométriques:
  classe Disque, classe Triangle, classe Carré, etc. Chacune doit avoir une méthode de calcul d'aire.
- Le risque est de définir des méthodes variées dans chacune des classes (nom différent, par exemple: calculAire () dans l'une, CalculerAire(), dans une autre, etc.).
- Pour uniformiser cela, il est possible de définir par exemple une interface Forme avec un prototype de méthode
  public double Aire();
- En faisant implémenter cette interface par toutes les classes, on assure que les méthodes de calcul d'aire suivent la même logique.
- Bien entendu, les aires ne sont pas calculées avec les mêmes formules pour toutes les formes et donc la méthode ne sera pas réalisées (Codée) de la même manière dans toutes les classes. Mais on assure que chaque méthode renverra un double et que l'utilisateur pourra l'appeler de la même manière pour tous les objets provenant des différentes classes (objet.Aire()).

51

Les Interfaces

53

Exemple

```
public interface Forme {
    public static final double Pi=3.14;
    public double Aire();
 }
public class Disque implements Forme {
    …// codes des attributs et méthodes
    public double Aire() {
            return (Pi * R * R);
     }
}
public class Triangle implements Forme {
    …
    public double Aire() {
            return (B* h/ 2.0);
     }
}
```

52

Les Interfaces

54

- Une interface peut coder des constantes en plus de méthodes (cf. Pi dans l'exemple précédent) .
- Les constantes sont forcément *static* et *final*. On peut le préciser dans l'interface, mais dans le cas contraire cela est implicite.

  public static final double Pi=3.14*;*
- Les méthodes sont forcément *abstract*. De la même manière, ce mot-clé peut être codé ou non, cela ne changera rien:

  public abstract *double Aire ();*
- Une interface ne peut pas implémenter elle-même une autre interface.
- Une interface peut hériter non seulement d'une autre interface, mais également de plusieurs en même temps (attention aux collisions de noms: deux méthodes de prototypes identiques ne posent pas problème, mais deux méthodes ne se distinguant que par des types de retour différents provoquent un conflit):

```
public interface inter1  extends inter2, inter3, inter4{
                ….}
```

53

Les Interfaces

56

- **Soit :**

```
1. interface Base {
2.         boolean m1 ();
3.         int m2(int s);
4. }
```

- Quel est le fragment de code qui compilera?

A. interface Base2 implements Base { }

B. abstract class Class2 extends Base {
   public boolean m1() { return true; } }

C. abstract class Class2 implements Base { }

D. abstract class Class2 implements Base {
   public boolean m1() { return (true); } }

E. class Class2 implements Base {
   public boolean m1() { return false; }
   public int m2(int s) { return 42; } }

54

Question

57

A. Une classe peut implémenter plusieurs interfaces mais doit étendre une seule classe
B. Une classe peut implémenter plusieurs classes mais doit étendre une seule interface
C. Une classe peut implémenter plusieurs classes et peut étendre plusieurs interfaces
D. Une classe doit implémenter une seule interface et étendre une seule classe

58

Design Patterns

- MVC Pattern : *Model-View-Controller Pattern*
- DAO Pattern : *Data Access Object Pattern*

MVC Pattern

59

- Modèle MVC signifie Model-View-Controller. Ce modèle est utilisé pour séparer les modules de l'application.
  - **Modèle** - représente un objet transportant des données. Il peut aussi avoir une logique permettant la mise à jour du contrôleur si les données changent.
  - **Vue** - représente la visualisation des données qui contient le modèle.
  - **Controller** - agit à la fois sur le modèle et la vue. Il contrôle le flux de données en modèle objet et met à jour la vue chaque fois que les données changent. Il garde la vue et le modèle séparé.

57

MVC Pattern

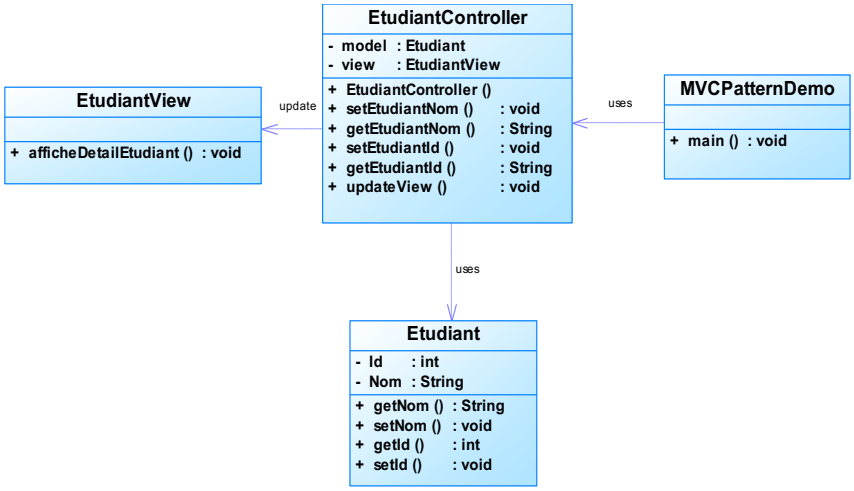
60

- **Implémentation :**
  - Nous allons créer
    - Classe **Etudiant** agissant comme un modèle.
    - **EtudiantVue** sera une classe d'affichage qui permet d'afficher les détails des étudiants sur la console.
    - **EtudiantControleur** est la classe responsable de stocker les données dans le contrôleur de l'objet Etudiant et la mise à jour de la vue **EtudiantVue** en conséquence.
  - MVCPatternDemo : notre classe de démonstration, utilisera **EtudiantControleur** pour démontrer l'utilisation du modèle MVC.

58

MVC Pattern

61



59

DAO Pattern

62

- Le modèle Data Access Object (DAO) est utilisé pour séparer les composants logiciels (API) de faible niveau permettant l'accès aux données ou les opérations des couches métiers de haut niveau.
- Voici les participants à Data Access Object Pattern.
  - Data Access Object Interface - Cette interface définit les opérations standard à effectuer sur un objet modèle.
  - Data Access Object concrète classe - Cette classe implémente l'interface ci-dessus. Cette classe est responsable d'obtenir des données à partir d'une source de données qui peut être une base de données ou tout autre mécanisme de stockage.
  - Modèle Objet- Cet objet contenant des méthodes get / set pour stocker les données récupérées à l'aide de la classe DAO.

60

DAO *Pattern*

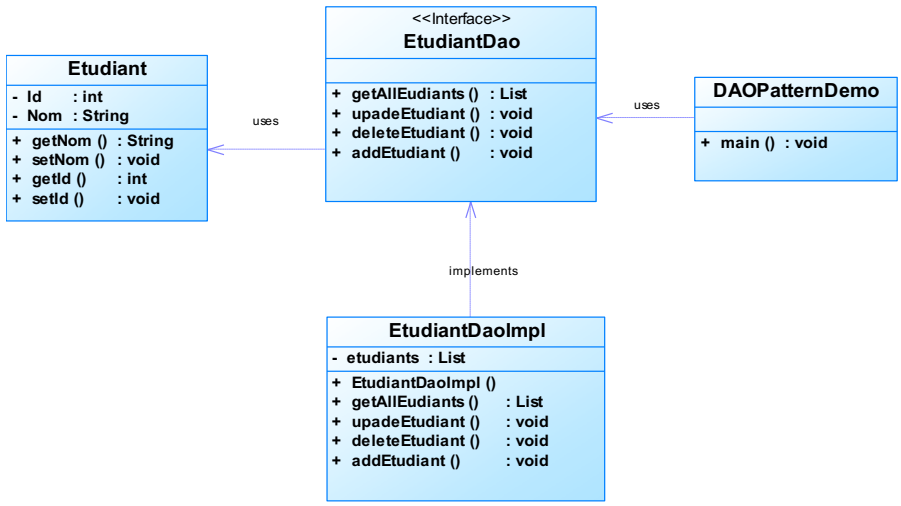
63

- Implémentation :
  - Nous allons créer un objet étudiant agissant comme un modèle d'objet.
  - EtudiantDao est Data Access Object Interface.
  - EtudiantDaoImpl est une classe concrète implémentation de Data Access Object Interface.
  - DaoPatternDemo, notre classe de démonstration, utilisera EtudiantDao pour démontrer l'utilisation du modèle Data Access Object.

61

DAO *Pattern*

64



62

Question

15

- Qu'affichera le programme suivant ?

```
for(int i=0; i<7; i++){
   switch(i) {
      case 5: System.out.println("cinq");
              system.out.println("deuxième message 5");
              break;
      case 4: System.out.println("quatre");
              break;
      case 3:
      case 2: System.out.println("deux");
              break;
      case 1 : System.out.println("un");
      case 0 : System.out.println("zero");
              break;
      default : System.out.println("kesako?");
              break;
      }
   }
```

Le résultat affiché sera:
Zéro
Un
Zéro
Deux
Deux
Quatre
Cinq
Deuxième message 5
Kesako?

63

Question

16

- **Soit:**

```
1. class A {
2. public static void main(String [] args) {
3. int x = 0;
4. // ici
5. do { } while (x++ < y);
6. System.out.println(x);
7. }
8. }
```

- **Qu'elle instruction, ajoutée à la ligne 4, produit le résultat 12 ?**
  - A. int y = x;
  - B. int y = 10;
  - C. int y = 11;
  - D. int y = 12;
  - E. int y = 13;
  - F. aucune instruction.

64

QUESTIONS !!!

65