Initiation Java

Table des matières

Introduction

1

A. Principes des langages orientés objet

Besoin général en développement : améliorer la productivité

- Réutiliser du code existant sans avoir à l'examiner
- Produire du code de qualité
  - Limiter les bugs
  - Corriger un bug sans en ajouter de nouveaux
  - Optimiser le code

Programmation Orientée Objet

- Amélioration de l'écriture et de la productivité
- Encapsulation des données et du code qui s'y rapporte
- Masquage des données internes
- Découplage entre différentes parties du code
- Adaptation du code sans risque sur les parties qui l'utilisent déjà

Objet

- Entité regroupant des données et des fonctions associées à ces données (encapsulation)

  Propriétés => données

  Méthodes => fonctions
- Visibilité : propriétés et méthodes privées et publiques

Classes

- Définition de type de variable, qui permet de créer des objets
- Hiérarchie de classes obtenues par héritage
- Modèle de classe (classes abstraites) et modèle de comportement (Interface)
- Bibliothèques de classes utilitaires
- Généricité

Outils

- Tests automatisés (tests unitaires)

- Langage de modélisation (Unified Modeling Language)
- Modèle de conception (Design Pattern)

B. Présentation de Java

Grandes caractéristiques

Un langage de programmation orienté objets
Une architecture de Virtual Machine
Un ensemble d'API variées
Un ensemble d'outils (le JDK)
Syntaxe proche du C
Code source Unicode
Multi-thread

Langage orienté objets

Tout est classe (pas de fonctions) sauf les types primitifs (int, float, double, ...) et les tableaux
Héritage simple pour les classes
Héritage multiple pour les interfaces
Les objets se manipulent via des références
Une API objet standard est fournie

Robuste

Gestion automatique de la mémoire,
Pas d'accès direct à la mémoire,
Tableaux : objets, avec contrôle de taille,
Mécanisme d'exception,
Compilateur contraignant : erreur si exception non gérée, si utilisation d'une variable non affectée,
Contrôle des conversions et des références à l'exécution

Sécurisé

Pris en charge dans l'interpréteur.
Trois couches de sécurité :
Verifier : vérifie le byte code.
Class Loader : responsable du chargement des classes.
Security Manager : accès aux ressources.
Code certifié par une clé.

Portable

Le compilateur Java génère du byte code.
La Java Virtual Machine (JVM) est présente sur Unix, Linux, Windows, MacOS, ...
La taille des types primitifs est indépendante de la plate-forme.
Java est accompagné d'une librairie standard.

Types d'applications

applications console

applications graphique
applet
servlet
android
...

Définition : JRE : Java Runtime Environment

Machine virtuelle
Bibliothèques standards

Définition : JDK : Java Development Toolkit

javac : compilateur de sources java
java : interpréteur de byte code
jar : création d'une archive de classes
appletviewer : interpréteur d'applet
javadoc : générateur de documentation (HTML, MIF)
jdb : debugger

Historique

1993 : projet Oak (langage pour l'électronique grand public)
1995 : Java / HotJava à WWW3
Sept. 95 : JDK 1.0
Fév. 97 : JDK 1.1
JDK 1.2 => Java 2
JDK 1.3
JDK 1.4
JDK 1.5 => Java SE5
JDK 1.6 => Java SE6
JDK 7 => Java SE7
JDK 8 => Java SE8
JDK 9 => Java SE9

Liens

Oracle : *https://www.oracle.com/java/index.html*[1]
http://java.developpez.com/[2]

1 - https://www.oracle.com/java/index.html
2 - http://java.developpez.com/

II

Première approche du langage

A. Structure d'un programme

1. Instructions

Instruction

- réalisation d'opérations sur des variables à l'aide d'opérateurs
- appel de méthodes
- une instruction est terminée par ;
- une instruction peut être écrite sur plusieurs lignes
- on peut placer plusieurs instructions sur une même ligne
- une instruction peut contenir des espaces, des sauts de ligne et des tabulations

Bloc d'instructions

- On peut grouper des instructions avec { }
- nécessaire pour certaines constructions du langage (test, boucle, corps de méthode)
- utilisable pour améliorer la maintenabilité du code

2. Commentaires

Commentaires jusqu'à la fin de la ligne avec //

```
instruction // commentaire valable jusqu'à la fin de la ligne
```

Commentaires en bloc avec / et */*

```
/* début du commentaire
ligne de texte
```

ligne de texte
ligne de texte
fin du commentaire */

3. Annotations

texte donnant des informations au compilateur

@Deprecated
@Override
@SuppressWarnings("deprecation")

4. Javadoc

Commentaires avec une mise en forme spécifique, analysable par l'outil javadoc

/**
* @param args the command line arguments
* @return
* @throws
*/

5. Classes

Structuration du code

Tout le code est écrit au sein de classes

Une instruction écrite à l'extérieur d'une classe provoque une erreur lors de la compilation

Contenu d'une classe

définition de variables

définition de méthode

Les unités de compilation

Le code source d'une classe est appelé unité de compilation. (extension .java)

Il est recommandé (mais pas imposé) de ne mettre qu'une classe par fichier.

Un fichier ne peut contenir qu'une seule classe publique

Le fichier doit avoir le même nom que la classe publique qu'il contient.

Fichiers

.java : code source java

.class : byte code (résultat de la compilation)

.jar : archive contenant des fichiers .class (format zip)

6. Identificateurs

Les différents éléments créés : classes, variables, méthodes sont désignés par un identificateur.

Un identificateur a les caractéristiques suivantes :

- lettres, chiffres, _, $
- sensible à la casse : ident et Ident sont deux identificateurs différents
- 1er caractère doit être une lettre

7. Packages

Un package regroupe un ensemble de classes sous un même espace de 'nom'.

Les noms des packages suivent le schéma : name.subname ...

Les API sont organisées en package (java.lang, java.io, ...)

Une classe Watch appartenant au package time.clock doit se trouver dans le fichier time/clock/Watch.class

Les packages permettent au compilateur et à la JVM de localiser les fichiers contenant les classes à charger.

L'instruction package indique à quel package appartient la ou les classe(s) du fichier

Les répertoires contenant les packages doivent être définis avec CLASSPATH

En dehors du package, les noms des classes sont : packageName.className

L'instruction import packageName permet d'utiliser des classes sans les préfixer par leur nom de package.

B. Point d'entrée du programme : méthode main

méthode statique d'une classe publique

```
1  public class essai {
2    public static void main(String args[]) {
3      System.out.println("bonjour");
4    }
5  }
```

argv : tableau de chaînes contenant les arguments

argv[0] : premier argument

argv[1] : second argument

...

Exécution "à la main"

compilation : javac essai.java

lancement : java essai

Types de base

III

A. Déclaration de variables

1. Définition d'une variable

zone mémoire contenant une valeur, codée suivant un type donné, identifiée par un nom

toutes les variables doivent être déclarées, sous la forme :

[attribut] type nom [= valeur initiale] ;

une variable ne peut être déclarée qu'une seule fois dans un bloc.

la définition peut être placée n'importe où dans une méthode, avant la première utilisation de la variable

2. Type de données

a) booléen

boolean

variable logique (true ou false)

b) octet

byte

un octet

c) caractère

char

caractère (unicode), stocké sur deux octets

d) entier

int

nombre entier, sur 4 octets

short

nombre entier sur 2 octets

long

nombre entier sur 8 octets

e) flottant

float

simple précision (4 octets)
IEEE-754 32 bit floating point type

double

double (8 octets)
IEEE-754 64 bit floating point type

f) Conversion de type

nom du type entre parenthèses

(int)
(float)

le compilateur sait faire des conversions implicites, s'il n'y a pas de perte de précision

```
1 int i;
2 float f;
3 f = i;
```

il est obligatoire de convertir explicitement s'il y a perte de précision

```
1 int i;
2 float f;
3 i = (int)f;
```

B. Initialisation de variable

L'initialisation est optionnelle.
Une initialisation de variable se produit avant l'exécution du bloc dans lequel la variable est définie.
Si la variable n'est pas initialisée, elle prend une valeur par défaut

Attention

Il ne faut pas confondre une initialisation avec une affectation.
Une initialisation n'est effectuée qu'une seule fois, lorsque la variable est créée.

La valeur initiale peut être une autre variable ou une constante.

- caractère, placé entre apostrophe
  'a' , 'F'
- entier : type entier par défaut, suffixe l ou L pour indiquer une constante longue
  - décimal
  - octal (0755)
  - hexadécimal (0xFA15)
- flottant : type double par défaut, suffixe f ou F pour indiquer une constante simple précision
  0.123 ou -0.4e5 ou .67E-20
- booléen
  true, false

C. Portée des variables locales

Visibilité

une variable est accessible uniquement dans la méthode ou le bloc d'instruction dans laquelle elle est définie

une variable est visible dans le bloc où elle est définie

```
1  {
2    int j1 /* visible dans le bloc {} uniquement */
3    int j2 /* visible dans le bloc {} uniquement */
4    {
5      // j1 est aussi visible dans ce bloc
6      int j2; // erreur de compilation car j2 est déjà définie
7    }
8  ....
9  }
10 j1 = 0 ; // erreur de compilation, extérieur du bloc
```

D. Durée de vie des variables

Durée de vie

- automatique : la variable est créée au début d'un bloc d'instruction et détruite à la fin
- statique : la variable est créée au début de l'exécution du programme et existe pendant toute la durée du programme

Attention

Durée de vie et portée des variables sont deux notions complétement différentes, qu'il ne faut pas confondre.

E. Attributs des variables

Attributs

- static : variable de classe, durée de vie de la variable est celle du programme

  attention : pas de variable static dans les variables locales d'une méthode
- final : la variable a une valeur constante
- private : la variable n'est accessible que depuis la classe
- protected : la variable n'est accessible que depuis la classe ou une classe dérivée
- public : la variable est accessible partout

F. Classes associées aux types primitifs

besoin de classes représentant un type primitif

Nécessaire dans certains cas pour les passages de paramètre.
Fournit des informations et des méthodes utilitaires
classes : Byte, Double, Float, Integer, Long, Short
methodes : parseInt, valueOf, ...

Utilisation de tableaux et de classes standards

IV

A. Lecture au clavier

class scanner

```
1 Scanner sc = new Scanner(System.in);
2 int k = sc.nextInt();
```

B. Tableau

Définition

C'est un ensemble ordonné de variables d'un même type, accessible par un nom symbolique.

Déclaration d'un tableau à une dimension

type nom[]

ou

type[] nom

type peut être un type de base ou n'importe quelle classe

initialisation

type nom[] = new type[taille];

type[] nom = {val1,val2,val3} ;

Utilisation

Un élément du tableau s'utilise comme une variable du type défini, en spécifiant un

indice
a = table[23] ;
table[10] = 0 ;

Attention

Le premier élément est à l'indice 0
Il y a un contrôle sur la validité des indices, avec la levée d'une exception en cas de dépassement.
La taille du tableau est fixe, et est disponible avec la propriété length.

Copie d'un tableau

System.arraycopy()

Méthodes utilitaires

java.util.Arrays

Tableaux multidimensionnels

Un tableau peut avoir plusieurs indices.
définition d'un tableau à deux dimensions :
type nom[][]

C. Conteneurs

Présentation

Classes conçues pour contenir des ensembles d'objets

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMa

Conteneurs

3

Exemple simple : Vector, ArrayList

Nombre variable d'éléments

```
1  public class MainVector {
2    public static void main(String[] args ) {
3      Vector<String> v1 = new Vector<>();
4      for (String a : args) {
5        v1.addElement(a);
6      }
7      ArrayList<String> v2 = new ArrayList<>();
8      boolean addAll = v2.addAll(Arrays.asList(args));
9    }
10 }
```

3 - http://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html

Remarque : pas d'utilisation des types primitifs

Vector<int> // : erreur
Vector<Integer> // ok

Il peut exister des conteneurs sans généricité

ArrayDequeue

D. Chaînes de caractères

1. String

classe String

La classe String gère des chaînes de caractères (char).
Caractères unicodes
L'initialisation est effectuée avec un texte entre guillemets

```
1 String s = "\u00catre ou ne pas \u00eaetre"; // s = "Être ou ne pas
  être"
2 int lg = s.length(); // lg = 19
```

Attention : Une String n'est pas modifiable

Toute modification entraîne la création d'une nouvelle String
Utiliser StringBuffer lorsque les chaînes doivent être modifiables.

Opérations sur les chaînes

L'opérateur + permet la concaténation de 2 String.

```
1 String s = "Java" + "Soft"; // s = "JavaSoft"
```

Comment obtenir une String ?

Constructeurs

```
1 char[] data = {'J', 'a', 'v', 'a'};
2 String name = new String(data);
```

Conversion

```
1 String s = (String) new URL("http://server/big.txt").getContent();
```

valueOf de la classe String

```
1 String s = String.valueOf(2 * 3.14159); // s = "6.28318"
2 String s = String.valueOf(new Date()); // s = "Sat Jan 18 12:10:36
  GMT+0100 1997"
```

méthode toString()

Comparaison de String

pas de ==, mais méthode equals

```
1 String s1 = "bonjour";
2 String s2 = "bon";
3 s2 += "jour";
4 System.out.println("Comparaison avec == : " + (s1 == s2));
5 System.out.println("Comparaison avec equals : " +s1.equals(s2));
6 String s3 = "bonjour";
7 System.out.println("Comparaison avec == de constante dans le pool : "
  + (s1 == s3));
8 s3 = new String("bonjour");
9 System.out.println("Comparaison avec == d'une nouvelle chaine : " +
  (s1 == s3));
```

2. StringBuffer,StringBuilder

StringBuffer gère des chaînes de caractères modifiables

initialisation à partir d'une String par le constructeur

```
1 StringBuffer sb = "abc"; // Error: can't convert String to
  StringBuffer
2 StringBuffer sb = new StringBuffer("abc");
```

Remarque : StringBuffer /StringBuilder

StringBuffer : synchronized
StringBuilder : not synchronized

Méthodes

setCharAt(int pos, char c) : modifie un caractère

```
1 sb.setCharAt(1, 'B'); // sb= "aBc"
```

insert(int pos, String s) : insère une chaîne

```
1 sb.insert(1, "1234); // sb = "a1234Bc"
```

append(String s) : ajoute une chaîne

```
1 sb.append("defg"); // sb = "a1234Bcdefg"
```

toString() : fournit une String

```
1 String s = sb.toString(); // s = "a1234Bcdefg"
```

Syntaxe de base

V

A. Opérateurs

1. Priorité et associativité

priorité

Les opérateurs ont des priorités d'évaluation

2 + 3 * 5 // valeur 17 car la multiplication a priorité sur l'addition

associativité

- Si deux opérateurs ont la même priorité, l'associativité définit l'ordre dans lequel ils sont évalués
- généralement de gauche à droite

Tableau des priorités et associativités

opérateurs	associativité
++ -- -(unaire) ~ ! (type)	←
* / %	→
+ -(arithmétiques) +(String)	→
<< >> >>>	→
< <= > >= instanceof	→
== !=	→
&(et bit-à-bit)	→
^	→
|	→
&&	→
||	→
? :	
= *= /= %= += -= <<= >>= >>>= &= ^= |=	←

Priorité opérateurs

Utilisation des parenthèses

On peut modifier les priorités et les associations en groupant les expressions avec des parenthèses

Ne pas hésiter à les utiliser

- dans des expressions complexes même si les expressions seraient correctement évaluées, pour favoriser la lisibilité
- en cas de doute dans l'ordre d'évaluation

2. Opérateurs arithmétiques

+

- (soustraction et changement de signe)

*

/

% (reste de la division entière)

Attention

La valeur calculée dépend des opérandes.

Une division de deux entiers fournit un résultat entier, même si on affecte le résultat à un flottant.

Si plusieurs types sont mélangés dans une expression, il y a élévation des types vers celui qui a la meilleure précision.

3. Opérateurs d'affectation

=
+=
-=
*=
/=
%=
&=
|=
<<=
>>=

a += b est strictement identique à a = a +b

Remarque

l'écriture condensé évite des erreurs de frappe et facilite l'écriture du code

somme = some + valeur ; // erreur détectable uniquement en testant le programme

somme += valeur ; // écriture plus rapide, sans risque de faute de frappe

4. Opérateurs d'incrémentation / décrémentation

++ : augmente la variable de 1

-- : décrémente la variable de 1

l'emplacement de l'opérateur est important

- postfixé : la variable est incrémentée et la valeur de retour est la valeur avant l'incrémentation

  b = 3;

  a = b++; // a vaut 3 et b vaut 4
- préfixé : la variable est incrémentée et la valeur de retour est la valeur après l'incrémentation

  b = 3;

  a = ++b; // a vaut 4 et b vaut 4

5. Opérateurs de comparaison

==
!=
<=
>=
<
>

6. Opérateurs logiques

&& : et

|| : ou
! : négation

Attention : évaluation des opérandes

si le premier opérande permet de connaître le résultat, le second n'est pas évalué

```
1  public class SecondOperandeIgnore {
2    public static void main(String[] args) {
3      int a = 5;
4      if ((a == 5) && montest())
5      {
6        System.out.println(a);
7      }
8      System.out.println(a);
9    }
10   static boolean montest() {
11     System.out.println("Dans la fonction montest");
12     return true;
13   }
14 }
```

7. Opérateur ternaire : ?

(expression)?valeur_si_vrai:valeur_si_faux
Utilisé pour alléger des écritures :

```
1 if (i == 0 || i == 1) {
2    a = 10;
3 }
4 else {
5   a = 11;
6 }
```

peut être écrit

```
1 a = (i == 0 || i == 1)?10:11;
```

8. Opérateurs binaires

~ : complément à 1
& : et logique
| : ou logique
^ : ou exclusif
<< : décalage à gauche
>> : décalage à droite

B. Tests : if, switch

1. if

Test simple

```
if (expression) instruction ;


                        if (expression) {

  instruction1 ;

                           instruction2 ;

                           instruction3 ;

                        }
```

Si expression est true, on exécute l'instruction ou le bloc d'instruction.

Tests successifs

```
if (expression1) {

  instruction1;

                                          instruction2;

                                          instruction3;

                                        }

                                       else if (expression2) {

                                          instruction4;

                                          instruction5;

                                          instruction6;

                                        }
```

```
                                                else if (expression3) {

                                                  instruction7;

                                                  instruction8;

                                                  instruction9;

                                                }

                                                else {

                                                  instruction10;

                                                  instruction11;

                                                  instruction12;

                                                }
```

On peut placer autant de `else if` que l'on veut, éventuellement aucun.
Le `else` final est optionnel.

2. switch

```
switch (expression) {

                                                  case exp1:

                                                    instruction1;

                                                  case exp2:

                                                    instruction2;

                                                  default:

                                                    instruction3;

                                                }
```

Cette structure remplace une succession de `if /else if`
exp1, exp2 sont des constantes
l'option `default` est optionnelle

Attention

Une fois le traitement d'un case terminé, le traitement se poursuit en séquence avec les instructions des case suivants !

Il faut souvent terminer le traitement d'un case par une instruction break pour continuer l'exécution après le bloc switch.

C. Boucles : for, while, do while

1. for

boucles que l'on veut exécuter un nombre de fois "connu"

`etiquette:`

`for (initialisation;test;increment) instruction;`

`initialisation`, `test` et `increment` sont des expressions.

La séquence de traitement est

1. exécution de `initialisation`
2. si `test` est true, exécution de `instruction`, sinon passage à l'instruction suivante
3. exécution de `increment`
4. si `test` est true, exécution de `instruction`, sinon passage à l'instruction suivante
5. etc ...

```
1  for (i=0;i<10;i++) {
2      System.out.println(i);
3  }
```

Conseil : Utiliser un bloc d'instruction même s'il n'y a qu'une seule instruction.

Même s'il n'y qu'une seule instruction, c'est une bonne pratique de créer un bloc d'instruction pour faciliter la lecture et la maintenabilité.

parcours d'une collection

for (Object o : collection) instruction;

o prends successivement comme valeur tous les membres de collection

collection peut être un tableau ou une classe implémentant l'interface collection

2. while

boucles que l'on veut exécuter tant qu'une condition est vraie

`etiquette:`

`while (expression) instruction;`

Tant que la valeur de `expression` est vraie, on exécute l'instruction.

Attention : Il faut que la valeur de l'expression évolue

Si `expression` a une valeur fixe, on ne quitte jamais l'exécution de la boucle.

Conseil : Utiliser un bloc d'instruction même s'il n'y a qu'une seule instruction.

Même s'il n'y qu'une seule instruction, c'est une bonne pratique de créer un bloc d'instruction pour faciliter la lecture et la maintenabilité.

3. do while

boucles que l'on veut exécuter au moins une fois, puis tant qu'une condition est vraie

```
etiquette:
do instruction;
while (expression) ;
```

On exécute une première fois l'instruction, puis on continue tant que la valeur de `expression` est vraie.

Cette structure est utile pour certains algorithmes et évite de répéter deux fois les mêmes instructions.

Attention : Il faut que la valeur de l'expression évolue

Si `expression` a une valeur fixe, on ne quitte jamais l'exécution de la boucle.

Conseil : Utiliser un bloc d'instruction même s'il n'y a qu'une seule instruction.

Même s'il n'y qu'une seule instruction, c'est une bonne pratique de créer un bloc d'instruction pour faciliter la lecture et la maintenabilité.

D. Rupture de boucle : break et continue

break

`break` permet de quitter la boucle en cours de traitement, sans exécuter les instructions suivantes de la boucle.

break etiquette

`break` permet de quitter la boucle indiquée par etiquette, sans exécuter les instructions suivantes de la boucle.

continue

`continue` permet de sauter l'exécution des instructions jusqu'à la fin de la boucle, puis de reprendre le traitement de la boucle en cours.

continue etiquette

`continue` permet de sauter l'exécution des instructions jusqu'à la fin de la boucle indiquée par étiquette, puis de reprendre le traitement de la boucle en cours.

Classes

VI

A. Définition d'une classe

1. Définition

Une classe est une définition d'un nouveau type de variable.
Ce type de variable peut regrouper des variables et des méthodes.
Une classe permet de créer des objets (des instances de la classe) qui sont des variables en mémoire, représentant des entités réelles.

a) Déclaration d'une classe

```
1  class nomClasse {
2          // déclarations des variables
3         // déclarations des méthodes
4  }
```

Définition : Membres, champs, attributs, méthodes

Les variables et les méthodes d'une classe sont appelées les membres de la classe.
Les variables peuvent être appelées des attributs ou des champs.

Visibilité des membres

Les membres peuvent être déclarés avec un attribut de visibilité : private, protected ou public.
Un membre public peut être utilisé dans n'importe quelle portion de code.
Un membre private ne peut être utilisé que dans la classe où il est déclaré.
Un membre protected ne peut être utilisé que dans l'héritage de la classe où il est déclaré.
Si aucun attribut n'est spécifié, le membre est accessible dans le package.

b) Variables

Définition des variables

Une variable peut être de n'importe quel type, y compris une classe.

c) Définition des méthodes

Définitions du corps des méthodes

```
1  class point {
2  private int x, y;
3  public void set (int i, int j) { x=i; y=j;};
4  public void move (int i , int j) {
5    x += i;
6    y += j;
7  }
```

2. Création, utilisation et destruction d'une instance

Création d'un objet (instanciation)

La création d'un objet à partir de la classe s'effectue en utilisant l'opérateur new

```
1  NomClasse monInstance; // déclaration d'une reference sur un objet de
     type NomClasse
2  monInstance = new NomClasse; // création de l'instance
```

Remarque

Lors de la création d'une instance, on peut fournir des paramètres, à condition que le constructeur correspondant existe.

Utilisation d'un objet

Accès aux membres publiques :

- cl1.var_publique
- cl1.methode_publique

```
1  class Personne {
2      Adresse a;
3  };
4  class Adresse {
5      String ville;
6  };
7  Personne p = new Personne();
8  p.a = new Adresse();
```

Destruction d'un objet

La destruction des objets est prise en charge par le garbage collector (GC).

Le GC détruit les objets pour lesquels il n'existe plus de référence.

Les destructions sont asynchrones (le GC est géré dans une thread de basse priorité).

Aucune garantie n'est apportée quant à la destruction d'un objet.

Si l'objet possède la méthode finalize, celle-ci est appelée lorsque l'objet est détruit.

3. Variable this

La variable this existe automatiquement.
C'est une référence vers l'instance courante.

```
1  class Maclasse {
2    private int i;
3    public int affiche()
4    {
5      return this.i; // equivalent à return i
6    }
7  }
```

Son utilisation est superflue dans la majorité des cas, mais les normes de programmation adoptées dans l'équipe peuvent l'imposer.
Il y a des cas où son utilisation est obligatoire, par exemple lorsqu'un paramètre d'une méthode a le même nom qu'une variable membre.

B. Constructeurs et destructeur

1. Définitions

Il y un appel automatique d'une méthode à la création (constructeur) ou à la fin de vie (destructeur) d'un objet.
Java crée automatiquement un constructeur par défaut et un destructeur, qui ne font qu'allouer ou désallouer la mémoire pour les variables.

2. Constructeurs

a) Définition d'un constructeur

Un constructeur est une méthode qui a le nom de la classe et ne renvoie rien (même pas un void).
La méthode peut être surchargée, ce qui permet différents mode d'initialisation de la classe.

```
1  class Point {
2    private int x, y;
3    public Point (int x, int y) {this.x=x; this.y=y;};
4    public  Point() {x=0;y=0;};
5  };
```

b) Constructeur par défaut

Le constructeur par défaut est le constructeur qui ne prend pas de paramètres.
Il est souvent nécessaire d'en définir un.

Attention

Java ne crée un constructeur par défaut que s'il n'existe pas d'autre constructeur !
Dès que l'on écrit un constructeur, il faut aussi écrire un constructeur par défaut.

3. Destructeur

finalize

Méthode appelée automatiquement par le garbage collector lorsque l'instance est détruite

C. Méthode d'une classe

1. Définition

C'est un groupe d'instructions, identifié par un nom symbolique
Elle peut avoir des paramètres, placés sur la ligne de définition, entre parenthèses

```
type mamethode(type1 param1,type2 param2) {
}
```

S'il n'y a pas de paramètres, on indique juste les parenthèses.

```
float msans() {
}
```

Une méthode a accès à toutes les variables membres de la classe.

2. Utilisation d'une méthode

Une méthode est exécutée en lui fournissant des valeurs, indiqués entre parenthèses

```
mamethode(3,4);
```

S'il n'y a pas de paramètres, on indique juste les parenthèses

```
fsans();
```

3. Valeur de retour

Instruction return

Une méthode peut renvoyer une valeur, spécifiée à l'aide de l'instruction return

```
int mretour(param1,param2) {
return param1+param2/2;
}
```

Cette valeur peut être affectée à une variable

```
i = mretour(3,4);
```

S'il n'y a pas de valeur de retour, on utilise le type void.

```
void m1(int i) {
}
```

Remarque : Utilisation de la valeur de retour

Même si une méthode renvoie une valeur de retour, il n'est pas obligatoire de l'utiliser

```
mretour(3,4); // instruction légale, même si une valeur de retour est définie
```

Remarque : Instructions return

Une méthode peut très bien comporter plusieurs instructions return

```
1  int m2return (int i, int j) {
2    if (i > j) {
3      return i;
4    }
5    somme = 0
6    for (k=i;k<j;k++) {
7      somme += k;
8    }
9    return somme;
10 }
```

4. Passage de paramètres

Passage par valeur

Les paramètres sont passés par valeur.

Si on modifie la valeur d'un paramètre dans la méthode, il n'y a pas de répercussion dans le programme d'appel

```
1  int multi(i) {
2    i *= 2;
3    return i;
4  }
5  i = 5;
6  multi(i);
7  System.out.println(i) // affiche 5
```

Attention : passage d'un objet en paramètre

Quand on passe un objet en paramètre, on utilise une référence.

Le passage par valeur concerne donc la référence, qui sera conservée dans le code appelant la méthode.

Mais l'objet auquel on fait référence peut être modifié dans la méthode !

```
1  essai e1 = new essai(2);
2  e1.m1(2);
3  int i = 0;
4  e1.m2(i);
5  System.out.println("i inchangé dans le code appelant m2 :" + i);
6  essai e2 = new essai(-100);
7  System.out.println("état de l'objet e2 après sa création :" + e2.i);
8  e1.m3(e2);
9  System.out.println("état de l'objet e2 après passage dans m3 :" +
   e2.i);
10 class essai {
11   public int i;
12   public essai(int i) {
13     this.i = i;
14   }
15   public void m1(int i) {
16     System.out.println("Methode m1 : " + i);
17   }
18   public void m2(int i) {
19     i = -10;
20     System.out.println("Methode m2: " + i);
21   }
22   public void m3(essai e){
```

```
23      e.i = 0;
24    }
25  }
```

```
1  Methode m1 : 2
2  Methode m2: -10
3  i inchangé dans le code appelant m2 :0
4  état de l'objet e2 après sa création :-100
5  état de l'objet e2 après passage dans m3 :0
```

Nombre variables de paramètres

type... variable
on obtient un tableau type[] variable

Remarque

type... var et type[] var ne peuvent pas être employés en même temps

5. surcharge de méthodes

On peut définir plusieurs méthodes avec le même nom, mais des arguments différents (en nombre ou en type)

Java détermine la bonne méthode à appeler en utilisant les paramètres de l'appel (nombre et type)

```
1 float moyenne (int tab[], int nb){ ... }
2 float moyenne (float tab[], int nb){ ... }
3 int tab[] = new int[10];
4 System.out.println(moyenne(tab, nb)); // appel de la première
  fonction
```

Attention

La valeur de retour n'est pas suffisante pour la surcharge de fonctions.

```
1 float moyenne (int tab[], int nb){ ... }
2 int moyenne (int tab[], int nb){ ... } // erreur à la compilation
```

6. Méthodes get et set

Il est courant qu'une classe comporte des méthodes permettant d'accéder à un champ privé.

Elles permettent :

- de fournir une valeur stockée dans un champ privé
- d'effectuer un calcul à partir de valeurs stockées (par exemple de calculer un age à partir de la date de naissance)
- de contrôler la valeur d'une variable avant de la stocker dans le champ privé.

Définition

Ces méthodes peuvent être appelées :

- des méthodes get et set
- accesseur et mutateur
- getteur et setteur

D. Membres statiques

1. Variables statiques d'une classe

Ce sont des variables qui sont définies pour la classe et non pour les instances de la classe.

Elles sont définies avec l'attribut static.

2. Méthodes statiques

Ce sont des méthodes qui doivent être appelées en utilisant la classe et pas une instance.

Elles sont définies avec l'attribut static.

Elles servent à accéder aux variables statiques ou à définir des "fonctions"

E. Classes internes

Classes définies à l'intérieur d'une classe

```
1  class A {
2  ... // définitions de la classe A
3      class B {
4          ... // définitions de la classe B
5      }
6  ... // poursuite définitions de la classe A
```

la classe interne peut être utilisée pour définir des objets dans la classe externe

intérêts d'une classe interne

un objet de la classe interne est associé à l'objet de la classe externe qui l'a instancié

un objet de la classe interne a accès à tous les champs (mêmes privés) de l'objet externe

un objet externe a accès à tous les champs (mêmes privés) de l'objet interne

Remarque : Limitations

une méthode statique d'une classe externe ne peut pas créer d'objet d'une classe interne

une classe interne ne peut pas avoir de membres statiques

Énumérations

VII

définition d'une liste de constante

```
1  public enum WeekDays
2  {
3    LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE
4  }
5  WeekDays jour;
6  jour = WeekDays.LUNDI;
```

définition de méthode possible

Gestion des exceptions

VIII

A. Traitement d'erreur

En Programmation Orientée Objet, il serait impossible d'utiliser des méthodes de traitement d'erreur classiques

- test de la valeur de retour d'une fonction
- test d'une variable globale contenant un numéro d'erreur

Exception

principe : en cas d'erreur, le code génère une exception, qui sera traitée par une autre partie du code

Mise en œuvre

throw : génération d'une exception lorsqu'une erreur est détectée

try : entête d'un bloc d'instruction où une exception peut se produire

catch : traitement d'une exception

finally : code exécuté quel que soit le chemin suivi (déroulement normal ou exception)

Mécanisme

La levée d'une exception provoque une remontée dans l'appel des méthodes jusqu'à ce qu'un bloc catch acceptant cette exception soit trouvé.

Si aucun bloc catch n'est trouvé, l'exception est capturée par la machine virtuelle et le programme s'arrête.

B. Bloc try / catch / finally

On peut encapsuler le code provoquant éventuellement une erreur dans un bloc

try/catch

```
1  try {
2     // instructions éventuellement en erreur
3
4  }
5  catch (IOException e) {
6     // instructions de traitement d'erreur IOExeption et ses sous
   classes
7  }
8  catch (Exception e) {
9   // instructions de traitement d'erreur des autres exceptions
10 }
11 finally {
12 // instructions de "nettoyage" : libération de ressources, de
   mémoire, ...
13 }
```

Exceptions

Ce sont des instances de classes dérivant de java.lang.Exception

finally

Un bloc (optionnel) finally peut-être posé à la suite des catch. Son contenu est exécuté après un catch ou après un break, un continue ou un return dans le bloc try

Remarque : Méthodes

L'appel à une méthode pouvant lever une exception doit :

soit être contenu dans un bloc try/catch

soit être situé dans une méthode propageant (throws) cette classe d'exception

Remarque : Imbrication des try

On peut placer autant de blocs try que l'on veut et les encapsuler les uns dans les autres.

Attention : Ordre des catch dans le cas d'une hiérarchie de classe

Il faut mettre le catch des classes dérivées AVANT le catch de la classe de base.

try avec ressource

Utilisable sur des objets implémentant AutoCloseable

Remplace l'écriture d'un bloc finally avec appel de close

```
1  static String readFirstLineFromFile(String path) throws IOException {
2      try (BufferedReader br = new BufferedReader(new
   FileReader(path))) {
3              return br.readLine();
4      }
5  }
```

C. Génération d'une exception

La génération d'une exception se fait avec l'instruction throw

`throw MonException;`

throw variable; // lance une exception dont le type est celui de variable

throw ; // relance la dernière exception

D. Indication sur les exceptions d'une méthode

il faut spécifier les exceptions qu'une méthode pourra lever

int m1(int a) throws T1 // m1 enverra des exceptions de type T1

int m1(int a) throws T1, T2 // m1 enverra des exceptions de type T1 ou T2

int m1(int a) // m1 n'enverra aucune exception

E. Bonnes pratiques

Traiter localement ce que l'on peut traiter

Si on met en place un bloc try, il faut traiter les exceptions que l'on peut traiter à ce niveau du code.

Par exemple, si l'accès à un fichier sur le réseau n'est pas possible, on peut retenter plusieurs fois l'accès avant de s'arrêter.

Si on ne sait pas traiter à notre niveau de code, on transmet l'exception (ou une autre exception). Un autre bloc try/catch saura peut être gérer le problème.

Par exemple, si l'accès à un fichier sur le réseau n'est pas possible, on écrit dans un fichier local.

Utiliser les exceptions à bon escient

Lever et traiter une exception est coûteux : instanciation d'un objet, déroutement de l'exécution, ...

Si on peut tester la condition et résoudre la difficulté sans faire appel à une exception, il faut le faire.

Par exemple, même si on a une gestion des exceptions de calcul, tester si on va faire une division par zéro.

Héritage

IX

A. Principe

Besoin

Pour éviter d'avoir à réécrire ou copier plusieurs fois le même code, on veut mettre en commun des structures de données et des traitements,

On veut également pouvoir les enrichir sans remettre en cause ce qui a déjà été écrit et validé.

Mise en œuvre

- Création d'une classe, avec ses variables et ses méthodes membres, appelée classe de base ou super classe
- Création de nouvelles classes à partir de la classe de base. Ces classes peuvent ajouter des champs propres, des méthodes propres ou surcharger les méthodes de la classe de base. Ces classes sont appelées classes dérivées ou sous classes.

B. Syntaxe

```
1  class ClasseBase {
2  //
```

```
3  }
4  class ClasseDerivee extends  ClasseBase {
5  //
6  }
```

C. Attribut protected

Nouvel attribut de visibilité

L'attribut protected donne une visibilité différente suivant que l'on se situe dans une classe dérivée de la classe de base ou non.

Dans une classe dérivée de la classe de base, protected est identique à public.

Dans une classe qui n'est pas dérivée de la classe de base, protected est identique à private.

D. Blocage de l'héritage

On peut interdire la dérivation d'une classe en utilisant l'attribut final lors de la définition de la classe.

E. Redéfinition de méthodes et de champ

Redéfinition de méthode

Une classe dérivée peut redéfinir une méthode qui a été définie dans la classe de base., sauf si elle contient l'attribut final.

Il est conseillé d'ajouter l'annotation @Override pour indiquer au compilateur que l'on est conscient que l'on surcharge une méthode.

Cela permet également d'être averti lorsqu'on fait une erreur de signature dans la méthode surchargée.

Redéfinition de champ

Une classe dérivée peut redéfinir un champ qui a été défini dans la classe de base.

Accès aux membres de la classe mère ; super

Une classe peut accéder aux attributs ou méthodes de sa classe mère en utilisant super ou par conversion.

F. Constructeurs

Attention

Les constructeurs ne sont pas hérités

Enchaînement des constructeurs

Un constructeur de la classe de base est appelé avant le constructeur de la classe dérivée.

Spécification du constructeur de la classe de base

On peut spécifier le constructeur de la classe de base à appeler et définir les paramètres à utiliser

```
1  class clderiv : public clbase
2  clderiv(int i, int j, float f) :
3  {
4     super(i,j) ;
5    z = f;
6  }
```

G. Méthodes de la classe Object

java.lang.Object

Toute classe dérive de la classe Object.

Toute classe dispose donc des méthodes de cette classe, qu'il peut être nécessaire (ou obligatoire) de redéfinir pour les adapter à la classe considérée.

clone()

equals()

toString()

hashcode()

...

H. Référence et héritage

Une référence vers une classe de base peut contenir une référence vers n'importe quelle classe dérivée

```
1  class base {}
2  class deriv extends base {}
3  base pt;
4  pt = new base();
5  pt = new deriv();
```

instanceof

L'opérateur instanceof permet de déterminer la classe d'une instance.

```
1  ...
2  if (l instanceof Empruntable)
3  ...
```

I. Polymorphisme

1. Présentation

Problème

Puisqu'une référence vers la classe de base peut référencer une classe dérivée, quel est le comportement lorsqu'on appelle une méthode surchargée ?

```
1  public class Polymorphisme {
2    static public void main(String... args) {
3      Cbase p = new Cderive(); // p référence un objet de classe
   Cderive
4      p.m1();
5    }
6  }
7  class Cbase {
8    public void m1()
9    {
10     System.out.println("Methode de Cbase");
11   }
12 }
13 class Cderive extends Cbase {
14   public void m1() {
15     System.out.println("Methode de Cderive");
16   }
17 }
```

Réponse : "Methode de Cderive"

La méthode appelée est celle de l'objet réel.

Remarque

Il est préférable d'utiliser l'annotation @Overrride pour éviter de se tromper de signature

2. Classe abstraite

On peut créer des classes qui ne servent qu'à être dérivées.

Ces classes sont appelées classes abstraites et on ne peut pas créer d'instance avec elles.

Pour rendre une classe abstraite il faut ajouter l'attribut abstract dans la définition de la classe.

Une classe abstraite peut comporter une ou plusieurs méthodes abstraites, ne contenant pas de corps.

```
1  class abstract Shape {
2    public abstract double perimeter();
3  }
4  class Circle extends Shape {
5    ...
6    public double perimeter() { return 2 * Math.PI * r ; }
7  }
8  class Rectangle extends Shape {
9    ...
10   public double perimeter() { return 2 * (height + width); }
11 }
12 ...
13 Shape[] shapes = {new Circle(2), new Rectangle(2,3), new Circle(5)};
14 double sum_of_perimeters = 0;
15 for(int i=0; i<shapes.length; i++) sum_of_perimeters =
   shapes[i].perimeter();
```

Une classe dérivée d'une classe abstraite est elle aussi abstraite si elle ne surcharge pas toutes les méthodes abstraites de la classe de base.

J. Interface

Classe dont toutes les méthodes sont abstraites.

Définition avec l'instruction interface au lieu de class

```
1  abstract class Shape { public abstract double perimeter(); }
2  interface Drawable { public void draw(); }
3  class Circle extends Shape implements Drawable, Serializable {
4    public double perimeter() { return 2 * Math.PI * r ; }
5    public void draw() {...}
6  }
7  class Rectangle extends Shape implements Drawable, Serializable {
8  ...
9    public double perimeter() { return 2 * (height + width); }
10   public void draw() {...}
11 }
12 ...
13 Drawable[] drawables = {new Circle(2), new Rectangle(2,3), new
   Circle(5)};
14 for(int i=0; i<drawables.length; i++)
15 drawables[i].draw();
```

Une classe peut implémenter (implements) une ou plusieurs interfaces tout en héritant (extends) d'une classe.

Une interface peut hériter (extends) de plusieurs interfaces.

Utilisation d'une interface

instanceof peut s'appliquer aux interfaces

utilisation d'une boucle sur un ensemble d'objet, avec un test utilisant instanceof, et appel d'une méthode si le test est vrai. (par exemple : serialisation)

K. Programmation fonctionnelle

1. classe anonyme

définition d'une classe "à la volée"

```
1  class A { ...}
2  A a;
3  a = new A() { public void affiche()
4  {
5    System.out.println("anonyme derive de A"); }
6  }
7  a.affiche();
```

Utilisations possibles

surcharge d'une définition de méthode

implantation d'une méthode pour une interface

2. expression lambda (à partir de Java 8)

Définition d'une méthode à la volée

liste de paramètre entre parenthèses (qui peuvent être omises s'il y a un seul paramètre)

->

valeur de retour ou corps de méthode

```
1  p -> p.getGender() == Person.Sex.MALE
2  && p.getAge() >= 18
3  && p.getAge() <= 25
```

```
1  p -> {
2  return p.getGender() == Person.Sex.MALE
3  && p.getAge() >= 18
4  && p.getAge() <= 25;
5  }
```

Référence

http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html[4]

4 - http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Généricité

X

A. Généricité

Besoin en conception

Pour certaines conceptions, on s'aperçoit qu'il faut écrire les mêmes classes en modifiant uniquement le type de certaines données.

Exemple : les classes servant de conteneurs pour différentes variables

Solution

Utiliser la généricité afin de paramétrer des classes à l'aide de types de données.

B. Classe générique

Définition

Description normale d'une classe en définissant un ou plusieurs types de données en tant que paramètres

```
1  class MaClasse <T> {
2   private T obj;
3   public T m() { }
4  };
```

Utilisation

Lors de l'instanciation, il faut spécifier le ou les types paramétrés dans le modèle.

```
1  MaClasse<int> C1 = new MaClasse<int>();
2  ou
3  MaClasse<int> C1 = new MaClasse<>();
```

Type variable

Par défaut, le type est figé

n2 = new nom<Integer>() ; // erreur

Utilisation avec un type non figé

nom <?> n2 = new nom<Integer>();

n2 = new nom<String>()

Héritage

class deriv extends base

nom<deriv> ne dérive pas de nom<base> !

nom<? super deriv> : hérite de toutes les classes nom<d>, où d est un ancêtre de deriv (y compris Object)

nom<? extends base> : parent de nom<d>, où d dérive de base

Classes utilitaires

XI

A. java.lang.Class

Class

Class représente une classe java.

Elle n'est pas instanciable

utilisation

création dynamique des nouvelles instances (mais seul le constructeur par défaut est appelé)

examen de la classe d'un objet inconnu

```
1  Class classname = Class.forName("java.util.date");
2  Date d = (Date)classname.newInstance();
3  System.out.println("Date : " + d);
4  Integer i = classname.getMethod("getMinutes", null).invoke(d, null);
```

B. java.lang.Thread

Lancement d'un nouveau thread

2 possibilités :

- hériter de Thread
- implémenter Runnable

```
1  class C1 extends Thread
2  {
3     public C1() { this.start(); }
4     public void run() {...}
5  }
```

```
1  class C2 implements Runnable
```

```
2  {
3        public C2() {Thread t = new Thread(this); t.start(); }
4        public void run() {...}
5  }
```

Méthodes

void start()
void stop()
void suspend()
void resume()
static void sleep()

Synchronisation

Le mot réservé synchronized permet de synchroniser l'accès à une partie de code ou à une méthode

```
1  class Banque {
2      synchronized void ajouter(int montant) {...}
3      synchronized void retirer(int montant) {...}
4  }
5  class Client implements Runnable {
6      Banque b;
7      public Client(Banque b) {
8         this.b = b;
9         Thread t = new Thread(this);
10        t.start();
11    }
12    public void run() {
13    ...
14    b.ajouter(100);
15    ...
16    b.retirer(10); ...}
17    }
18 Banque b = new Banque();
19 Client c1 = new Client (b);
20 Client c2 = new Client(b);
```

Chaque objet possède les méthodes wait(), notify() et notifyAll()
Dans une partie synchronized d'un objet O :
wait() relâche le verrou et se met en attente.
notify() réveille un thread en attente (fifo)
notifyAll() réveille tous les threads en attente

```
1  class MyThing {
2     synchronized void waiterMethod() {...; wait(); ...}
3     synchronized void notifyMethod() {...; notify(); ...}
4     synchronized void anOtherMethod() {...}
5  }
```

Scheduling et priorité

Le scheduling est en partie dépendant des implémentations
setPriority() permet de fixer la priorité d'un thread
Pour 2 threads de même priorité, par défaut : round robin
T1 cède la place à T2 quand sleep(), wait(), bloque sur un synchronized, yield(), stop()
Certaines JVM implémentent un time slicing (Win32, IE, ...)

C. java.io.*

1. java.io.File

Gestion des fichiers et répertoires

Méthodes
exists()
canRead()
length()
is Directory()
mkdir
list() : liste des fichiers d'un répertoire
...

```
1  import java.io.File;
2  public class MaFile {
3    public static void main(String args[]) {
4      File d = new File("c:\\java");
5      if (d.isDirectory()) {
6        String[] files = d.list();
7        for(int i=0; i < files.length; i++)
   System.out.println(files[i]);
8      }
9    }
10 }
```

2. java.io.FileInputStream, java.io.FileOutputStream

Accès en lecture et en écriture à un fichier

```
1  FileInputStream fis = new FileInputStream("source.txt");
2  byte[] data = new byte[fis.available()];
3  fis.read(data);
4  fis.close();
5  FileOutputStream fos = new FileOutputStream("cible.txt");
6  fos.write(data);
7  fos.close();
```

3. java.io.DataInputStream, java.io.DataOutputStream

Accès en lecture et en écriture à des données

```
1  FileInputStream fis = new FileInputStream("source.txt");
2  DataInputStream dis = new DataInputStream(fis);
3  int i = dis.readInt();
4  double d = dis.readDouble();
5  String s = dis.readLine();
6  FileOutputStream fos = new FileOutputStream("cible.txt");
7  DataOutputStream dos = new DataOutputStream(fos);
8  fos.writeInt(123);
9  fos.writeDouble(123.456);
10 fos.writeChars("Une chaine");
```

4. java.io.PrintStream

manipulation d'un OutputStream au travers des méthode print() et println()

```
1  PrintStream ps = new PrintStream(new FileOutputStream("cible.txt"));
2  ps.println("Une ligne");
3  ps.println(123);
4  ps.print("Une autre ");
5  ps.print("ligne");
6  ps.flush();
7  ps.close();
```

5. java.io.ObjectInputStream, java.io.ObjectOutputStream

lire et écrire des objets, implémentant java.io.serializable, sur des flux

```
1  // Ecriture
2  FileOutputStream fos = new FileOutputStream("tmp");
3  ObjectOutput oos = new ObjectOutputStream(fos);
4  oos.writeObject("Today");
5  oos.writeObject(new Date());
6  oos.flush();
7  // Lecture
8  FileInputStream fis = new FileInputStream("tmp");
9  ObjectInputStream ois = new ObjectInputStream(fis);
10 String today = (String)ois.readObject();
11 Date date = (Date)ois.readObject();
```

Contrôle

Par défaut, tous les champs sont sérialisés (y compris private)

Cela peut poser des problèmes de sécurité

3 solutions :

- Ne pas implémenter Serializable
- Réécrire les méthodes writeObjet() et readObject()
- Le mot clé transcient permet d'indiquer qu'un champ ne doit pas être sérialisé.

D. java.net.*

1. java.net.Socket

Implémentation socket TCP coté client

```
1  String serveur = "www.webformation.fr";
2  int port = 80;
3  Socket s = new Socket(serveur, port);
4  PrintStream ps = new PrintStream(s.getOutputStream());
5  ps.println("GET /");
6  DataInputStream dis = new DataInputStream(s.getInputStream());
7  String line;
```

```
8  while((line = dis.readLine()) != null)
9  System.out.println(line);
```

2. java.net.ServerSocket

Implémentation socket TCP coté serveur

```
1  int port_d_ecoute = 1234;
2  ServerSocket serveur = new ServerSocket(port_d_ecoute);
3  while(true)
4  {
5  Socket socket_de_travail = serveur.accept();
6  new ClasseQuiFaitLeTraitement(socket_travail);
7  }
```

3. java.net.DatagramSocket

implémentation socket UDP

```
1  // Client
2  Byte[] data = "un message".getBytes();
3  InetAddress addr = InetAddress.getByName("www.afsic.fr");
4  DatagramPacket packet = new DatagramPacket(data, data.length, addr,
   1234);
5  DatagramSocket ds = new DatagramSocket();
6  ds.send(packet);
7  ds.close();
8  // Serveur
9  DatagramSocket ds = new DatagramSocket(1234);
10 while(true)
11 {
12 DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);
13 s.receive(packet);
14 System.out.println("Message: " + packet.getData());
15 }
```

4. java.net.MulticastSocket

implémentation socket multicast (UDP)

```
1  // Client
2  Byte[] data = "un message".getBytes();
3  InetAddress addr = InetAddress.getByName("www.afsic.fr");
4  DatagramPacket packet = new DatagramPacket(data, data.length, addr,
   1234);
5  MulticastSocket s = new MulticastSocket();
6  s.send(packet,(byte)1);
7  s.close();
8  // Serveur
9  MulticastSocket s = new MulticastSocket(1234);
10 System.out.println("I listen on port " + s.getLocalPort());
11 s.joinGroup(InetAddress.getByName("www.webformation.fr"));
12 DatagramPacket packet = new DatagramPacket(new byte[1024], 1024);
13 s.receive(packet);
14 System.out.println("from: " + packet.getAddress());
15 System.out.println("Message: " + packet.getData());
16 s.leaveGroup(InetAddress.getByName("www.webformation.fr"));
17 s.close();
```

5. java.net.URL

```
1  URL url = new URL("http://www.webformation.fr/");
2  DataInputStream dis = new DataInputStream(url.openStream());
3  String line;
4  while ((line = dis.readLine()) != null)
5  System.out.println(line);
```

Développements graphiques

XII

A. AWT et swing

Bibliothèques graphiques

Deux bibliothèques graphiques de base : AWT, Swing

Définition : AWT

AWT (Abstract Window Toolkit) : API indépendante du système.
Chaque composant AWT est contrôlé par un composant natif du système.
(composants lourds

Définition : Swing

API standard, pour créer des interfaces graphiques identiques quel que soit le système d'exploitation sous-jacent

Performances moindres que AWT

Principe Modèle-Vue-Contrôleur (MVC) : composants Swing = contrôleur MVC, avec plusieurs choix d'apparence (de vue).

1. AWT

voir awt dans Applet

2. Swing

```
1  import javax.swing.JFrame;
2  import javax.swing.JLabel;
3  public class BonjourSwing {
4  public static void main(String[] args) {
5  // on crée une fenêtre dont le titre est "Bonjour !"
6  JFrame frame = new JFrame("Bonjour !");
7  // la fenêtre doit se fermer quand on clique sur la croix rouge
8  frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
9  // on ajoute le texte "Bonjour !" dans la fenêtre
10 frame.getContentPane().add((new JLabel("Bonjour !")));
11 // on demande d'attribuer une taille minimale à la fenêtre
12 // (juste assez pour voir tous les composants)
13 frame.pack();
14 // on centre la fenêtre
15 frame.setLocationRelativeTo(null);
16 // on rend la fenêtre visible
17 frame.setVisible(true);
18 }
19 }
```

B. Applet

Présentation

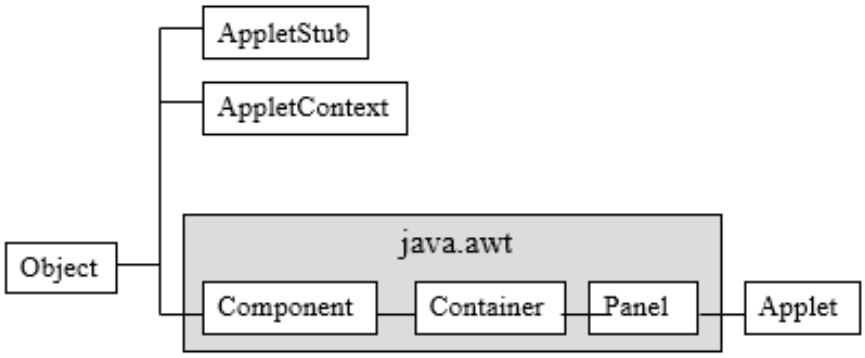
Programme exécuté par une VM Java dans un navigateur

Elle est fournie par un serveur Web dans une page HTML

```
1  <APPLET code='TiffViewer.class' width=50 height=50>
2  <PARAM name='imagesource' value='mon_image.tiff'>
3  </APPLET
```

Elle est soumise au Security Manager du navigateur :

- pas d'accès en lecture ni en écriture sur le disque du navigateur.
- connexion réseau uniquement sur le serveur d'origine.
- pas de chargement de librairie native.
- pas de lancement de processus
- ...



package Applet

Structure

```
1  public class MyApplet extends java.applet.Applet
2  {
3  public void init() {...}
4  public void start() {...}
5  public void paint(java.awt.graphics g) {...}
6  public void stop() {...}
7  public void destroy() {...}
8  }
```

4 méthodes de gestion de l'exécution

init : appelée au chargement de l'applet
start : appelée chaque fois que l'applet est relancée (=> la page est revisitée)
stop : appelée quand l'applet est arrêtée (=> la page est quittée)
destroy : appelée quand on ferme le navigateur

méthodes de tracé

paint : doit réaliser le tracé effectif du contenu de la fenêtre
=> Il est obligatoire de la définir.
update : méthode prédéfinie qui efface l'écran et appelle paint()
=> il faut parfois la redéfinir.

Exemple

```
1  <HTML>
2  <BODY>
3  <APPLET code="MonApplet" codebase="/applets/" width=300 height=200>
4  <PARAM name="message" value="Premier essai">
5  </APPLET>
6  </BODY>
7  </HTML>
```

```
1  String msg = this.getParameter("message");
2  this.showStatus("Applet en cours");
3  Image img = this.getImage(new URL("http://falconet/image.gif"));
```

```
4  AppletContext ctxt = this.getAppletContext();
5  ctxt.showDocument(new URL("http://www.sun.com/java"), "frame");
```

Méthodes

utilisation de méthodes de la classe java.applet.Applet
utilisation de méthodes des packages java.awt ou javax.swing

C. Awt

Contenu d'AWT

un mécanisme de gestion d'événements
des widgets
des LayoutManager

Gestion d'événements

évènements souris
boolean mouseDown(Event ev , int x, int y)
boolean mouseUp(Event ev , int x, int y)
boolean mouseMove(Event ev , int x, int y)
boolean mouseDrag(Event ev , int x, int y)
boolean mouseEnter(Event ev , int x, int y)
boolean mouseExit(Event ev , int x, int y)
Event ev : permet de disposer d'information supplémentaires sur l'événement réel
la valeur de retour permet de spécifier si on a effectivement traité l'événement
true : événement traité
false : événement pas traité => il faut qu'une autre fonction traite cet événement
événements clavier
boolean keyDown(Event ev , int key)
boolean keyUp(Event ev , int key)
il faut que le contexte graphique ait le focus pour gérer un événement clavier => requestFocus()

```
1  import java.awt.* ;
2  import java.awt.event.* ;
3  import javax.swing.event.* ;
4  import javax.swing.* ;
5  public class App2Bout extends JApplet implements ActionListener
6  { public void init ()
7  { pan = new JPanel ();
8  panCom = new JPanel() ;
9  Container contenu = getContentPane() ;
10 contenu.add(pan) ;
11 contenu.add(panCom, "South") ;
12 rouge = new JButton ("rouge") ;
13 jaune = new JButton ("jaune") ;
14 rouge.addActionListener(this) ;
15 jaune.addActionListener(this) ;
16 panCom.add(rouge) ;
17 panCom.add(jaune) ;
18 }
19 public void actionPerformed (ActionEvent e)
20 {
```

```
21  if (e.getSource() == rouge) pan.setBackground (Color.red) ;
22  if (e.getSource() == jaune) pan.setBackground (Color.yellow) ;
23  }
24  private JPanel pan, panCom ;
25  private JButton rouge, jaune ;
26  }
```

Base de données

XIII

A. Présentation de JDBC

Objectifs

Fournir un accès homogène aux SGBDR
Abstraction des SGBDR cibles
Requêtes SQL
Simple à mettre en œuvre

versions

1.0 : paquetage additionnel du JDK 1.0
1.1 : partie intégrante du JDK 1.1
2.0 : ajout de fonctionnalités importantes
3.0 : ajout de quelques fonctionnalités
4.0 : java SE6, simplification de l'utilisation

B. Architecture

Drivers JDBC

JDBC interagit avec le SGBDR par un driver

4 types de drivers :

1. Bridge ODBC (fourni avec JDBC)
2. Native-API partly-Java driver
3. JDBC-Net all-Java driver
4. Native-protocol all-Java driver

Fonctionnement

Charger le driver
Connexion à la base
Création d'un statement
Exécution de la requête
Lecture des résultats

C. Chargement du driver

Utiliser la méthode forName de la classe Class

```
1 Class.forName("sun.jdbc.odbc.JdbcOrdbDriver");
2 Class.forName("postgres95.pgDriver");
```

D. Connexion à la base

Accès via un URL spécifiant

l'utilisation de JDBC
le driver ou le type du SGBDR
l'identification de la base

```
1 jdbc:odbc:ma_base
2 jdbc:pg95:mabase?username=toto:password=titi
```

Ouverture de la connexion

```
1 Connection conn = DriverManager.getConnection(url, user, password);
```

E. Requêtes

3 types de statement

statement : requêtes simples
prepared statement : requêtes précompilées
callable statement : procédures stockées

1. Statement

createStatement

ResultSet sera en lecture seule, avec déplacement vers l'avant

```
1 Statement stmt = conn.createStatement();
```

Spécification du type d'accès et du type de modification du recordset

```
1  Statement stmt = con.createStatement(type de parcours, type de
   modification);
```

3 types d'exécutions

executeQuery : pour les requêtes qui retournent un ResultSet

executeUpdate : pour les requêtes INSERT, UPDATE, DELETE, CREATE TABLE et DROP TABLE

execute : pour les procédures stockées

a) Requêtes fournissant des résultats

Exécution de la requête

```
1  String myQuery = "SELECT prenom, nom, email " +
2  "FROM employe " +
3  "WHERE (nom='Dupont') AND (email IS NOT NULL) " +
4  "ORDER BY nom";
5  ResultSet rs = stmt.executeQuery(myQuery);
```

ResultSet

- trois types de ResultSet pour le parcours
  - déplacement vers l'avant
  - scroll-insensitive : pas de visualisation des modifications éventuelles de la base
  - scroll-sensitive : visualisation des modifications éventuelles
- deux types de ResultSet pour le traitement
  - lecture seule
  - modifiable

Accès aux données d'unResultSet

Les colonnes sont référencées par leur numéro ou par leur nom

L'accès aux valeurs des colonnes se fait par les méthodes getXXX() où XXX représente le type de l'objet

Pas de retour d'une valeur normalisée lorsqu'une colonne est vide. Il faut utiliser la méthode wasNull()

```
1  java.sql.Statement stmt = conn.createStatement();
2  ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
3  while (rs.next())
4  {
5  // print the values for the current row.
6  int i = rs.getInt("a");
7  String s = rs.getString("b");
8  byte b[] = rs.getBytes("c");
9  System.out.println("ROW = " + i + " " + s + " " + b[0]);
10 }
```

Pour les très grosses quantités de données, on peut utiliser des streams

Si la requête fournit plusieurs jeux de résultats : getmoreResult()

parcours d'un ResultSet

isFirst(), isLast()

isBeforeFirst(), isAfterLast()

first(), last(), next(), previous()

absolute()

relative()

modification

updateXXX()

updateRow() à appeler après les autres update

suppression

deleteRow()

ajout

moveToInsertRow()

updateXXX()

insertRow()

movetoCurrentRow() : retourne à la ligne courant avant l'appel à moveToInsertRow()

2. PreparedStatement

compilation d'une instruction SQL

PrepareStatement pstmt = con.preparedStatement("INSERT INTO EMPLOYEE (NOM,TEL) VALUES (?,?)");

définition des paramètres

clearParameter()

setString()

setXXX

setObject

setNull

exécution

executeUpdate()

3. CallableStatement

utilisation de procédures stockées

procédures stockées sans jeu de résultat

CallableStatement cstmt = con.prepareCall("{ call spcalcul(?,?) }");

procédures stockées avec jeu de résultat

CallableStatement cstmt = con.prepareCall("{ ? = call spcalcul(?,?) }");

enregistrement de paramètre de sortie

registerOutParameter(int pos, type)

définition de valeur d'appel

setXXX()

exécution

execute()

accès à un résultat

getXXX()

F. Métadonnées

accès aux informations sur la base : DatabaseMetaData

accès à des informations
getURL(), getMaxConnections()
accès à des ResultSet
GetColumns()

```
1 DatabaseMetaData dbmeta = con.getMetaData()
```

accès aux informations sur un ResultSet : ResultSetMetaData

accès aux :
– nombre de colonne : getColumnCount()
– nom d'une colonne : getColumnName(int col)
– type d'une colonne : getColumnType(int col)

```
1 ResultSetMetaData rsmd = rs.getMetaData()
```

G. Transactions

les connexions assurent la gestion transactionnelle

par défaut, une nouvelle connexion est en mode auto-commit
par défaut, toute opération est une transaction

Gestion d'une transaction

création d'une transaction : setAutoCommit(false);
validation d'une transaction : commit();
annulation d'une transaction : rollback();
l'appel à commit ou rollback fait revenir à l'état par défaut
connaître l'état : getAutoCommit()

Isolation des transactions

TRANSACTION_NONE transaction interdite
TRANSACTION_READ_UNCOMMITTED lecture possible par d'autres transactions avant la validation
TRANSACTION_READ_COMMITTED pas de lecture par d'autres transaction avant la validation
TRANSACTION_REPEATABLE_READ une transaction obtiendra toujours les mêmes informations, même si elles ont été modifiées dans une autre transaction
TRANSACTION_SERIALIZABLE une transaction obtiendra toujours les mêmes informations, même si de nouvelles lignes ont été ajoutées dans une autre transaction

définition du niveau d'isolation : setTransactionIsolation()
information sur les niveaux transactionnels supportés : classe DatabaseMetaData
getDefaultTransactionIsolation()
supportsTransactions()
supportsTransactionIsolationLevel()
supportsDataDefinitionAndDataManipulationTransactions()

H. Traitement par lots

addBatch()
executeBatch() ou clearBatch()

Remarque

on ne peut pas récupérer de résultats dans un traitement batch (pas de SELECT)