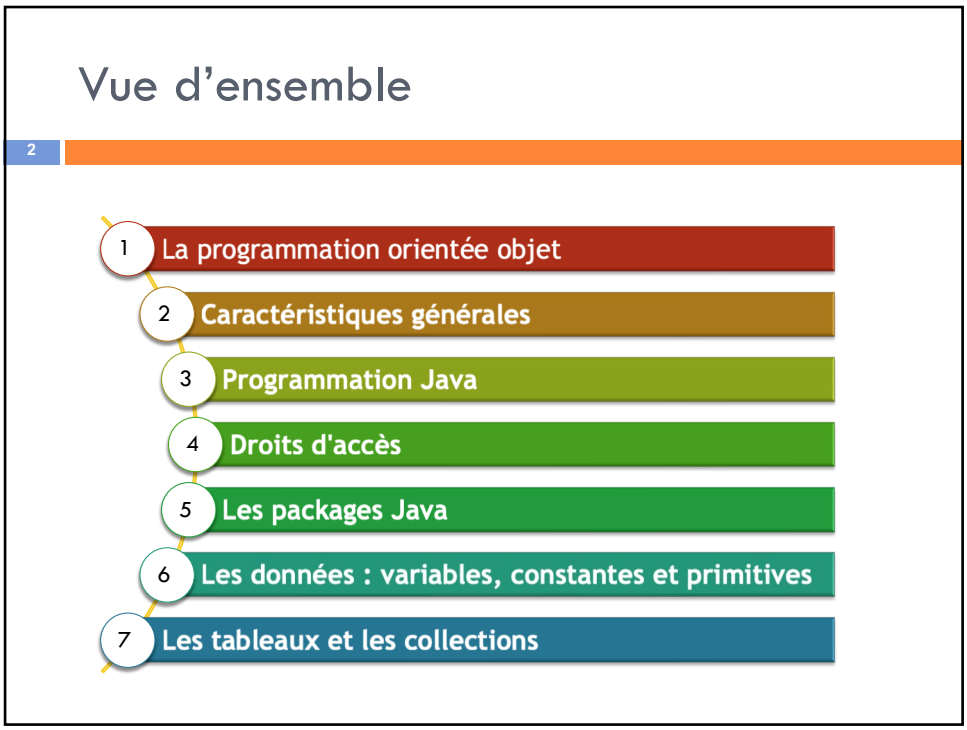

Java
INTRODUCTION À LA POO JAVA
PARTIE 1
I.AZAIEZ
Janvier 2024

1


Vue d'ensemble
2
1 La programmation orientée objet
2 Caractéristiques générales
3 Programmation Java
4 Droits d'accès
5 Les packages Java
6 Les données : variables, constantes et primitives
7 Les tableaux et les collections

2

Chapitre 1

3

LA PROGRAMMATION OBJET JAVA

3

Le langage java c'est quoi?

4

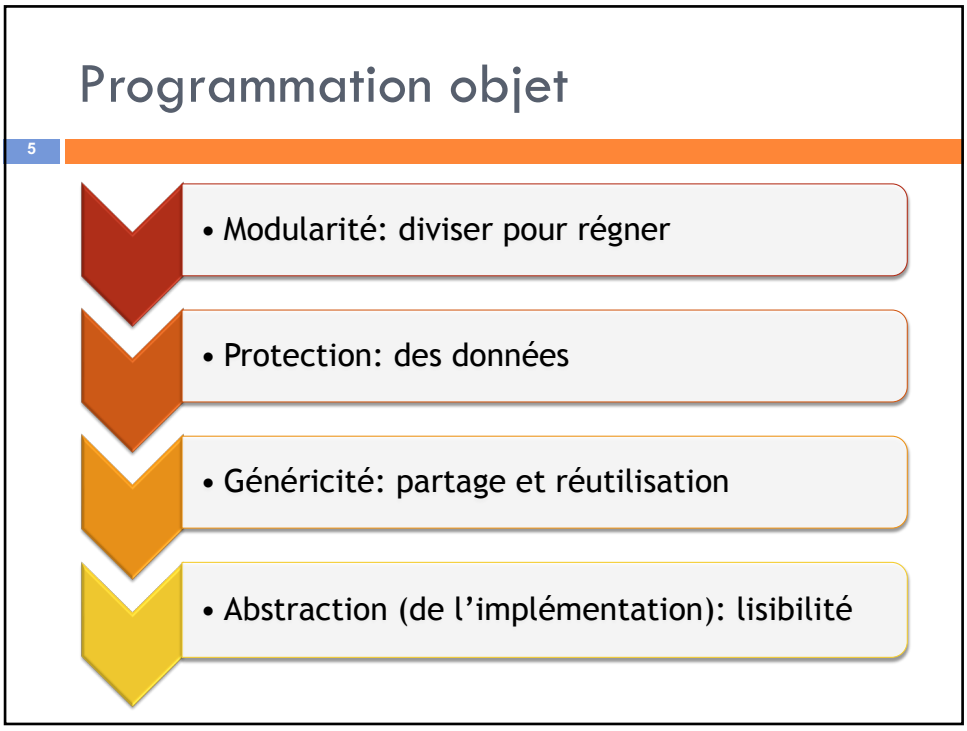
Java est utilisé pour créer :

- Des applications Desktop
- Des applications pour les Smartphones
- Des applications embarquées dans des cartes à puces
- Des application JEE (Java Entreprise Edition)

Pour créer une application java, il faut installer un kit de développement java

- JSDK : Java Standard Développement Kit, pour développer les applications DeskTop
- JEE : Java Entreprise Edition, pour développer les applications qui vont s'exécuter dans un serveur d'application JEE (Appache tomcat, Jboss, …).
- …

4



5

Programmation objet

6

- La méthode orientée objet permet de concevoir une application sous la forme d'un ensemble d'objets reliés entre eux.
- Lorsque que l'on programme avec cette méthode, la première question que l'on se pose plus souvent est :
  - **" qu'est-ce que je manipule ? ",** Au lieu de **" qu'est-ce que je fais ? ".**
- L'une des caractéristiques de cette méthode permet de concevoir de nouveaux objets à partir d'objets existants.
- On peut donc réutiliser les objets dans plusieurs applications.
- Pour faire la programmation orientée objet il faut maitriser les fondamentaux de l'oriente objet a savoir:
  - **Objet et classe**
  - **Héritage**
  - **Encapsulation (Accessibilité)**
  - **Polymorphisme**

6

Programmation objet : les objets

7

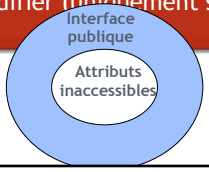
Chaque objet à sa propre identité, ses caractéristiques ("propriétés ou "attributs") et son comportement (actions qu'il peut réaliser: "méthodes").

Un objet est constitué d'attributs qui définissent son état à un instant donné (ses caractéristiques principales).

Ex : pour un vélo, la vitesse sélectionnée, les plateaux, …

Un utilisateur externe ne peut y accéder directement

Des méthodes publiques définissent une interface vis-à-vis de l'extérieur : Par elle, un utilisateur externe peut éventuellement connaître les valeurs des attributs, voir les modifier (uniquement si cela lui est permis)



7

Programmation objet : les objets

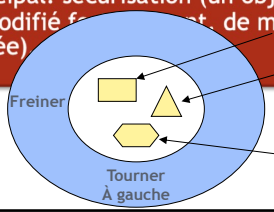
8

Ceci constitue le mécanisme d'encapsulation propre aux langages objets

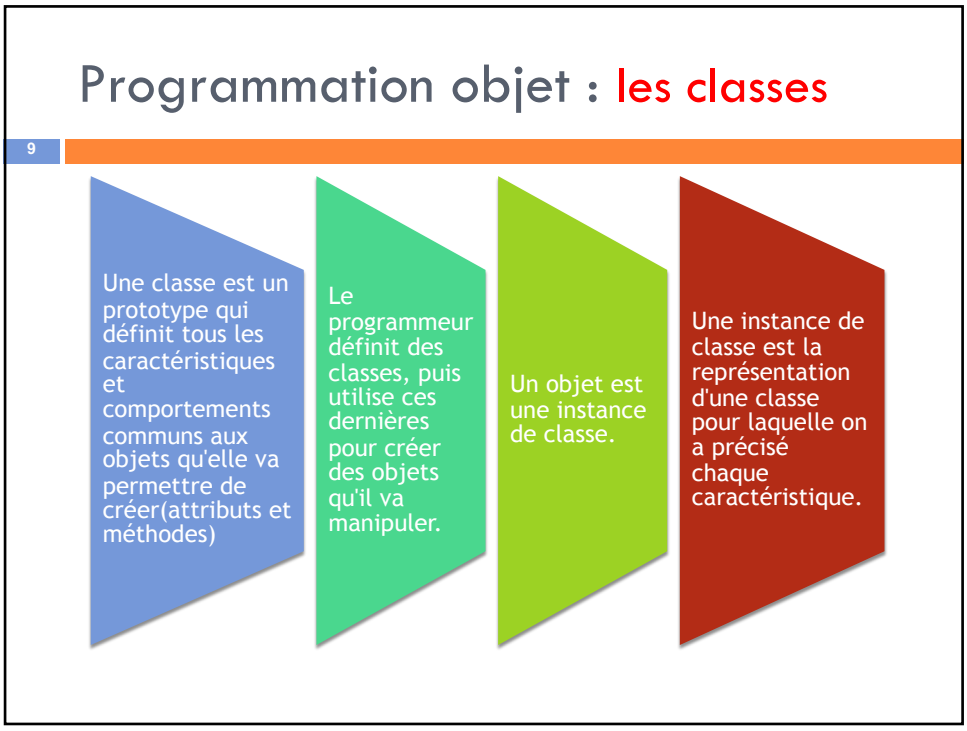
les caractéristiques propres à un objet ne sont accessibles qu'à cet objet. Pour qu'un autre objet puisse interagir avec celles-ci, il est nécessaire que l'interface publique du premier objet contienne des méthodes autorisant ces opérations.

Intérêt principal: sécurisation (un objet ne peut être modifié [illegible] de manière non contrôlée)

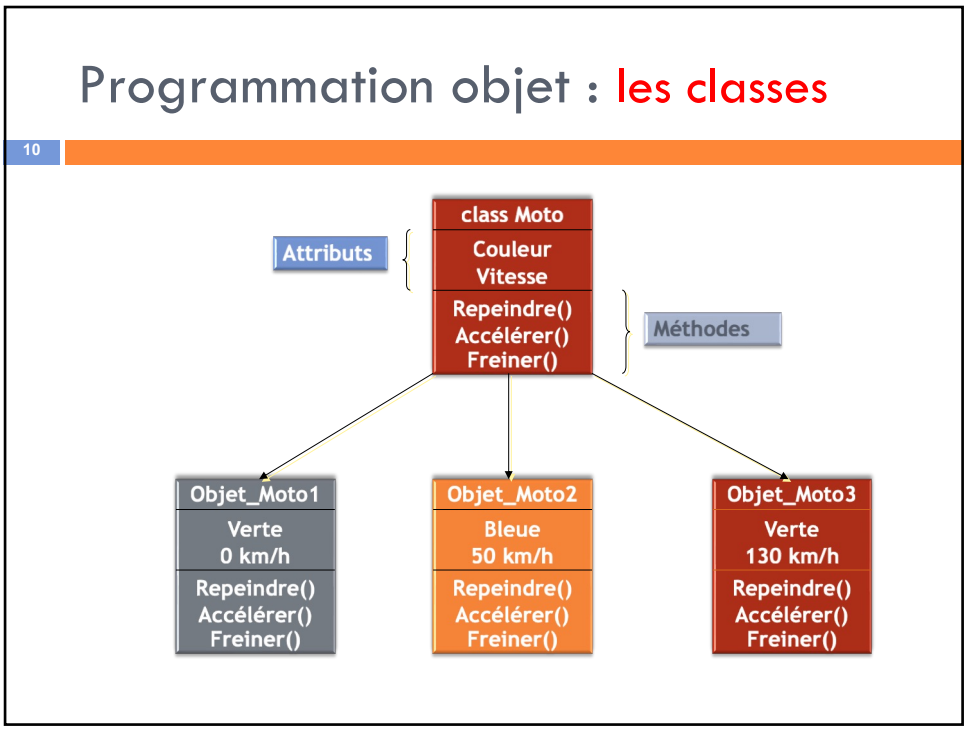
Ex : Moto



8


Programmation objet : les classes
9
Une classe est un prototype qui définit tous les caractéristiques et comportements communs aux objets qu'elle va permettre de créer(attributs et méthodes)
Le programmeur définit des classes, puis utilise ces dernières pour créer des objets qu'il va manipuler.
Un objet est une instance de classe.
Une instance de classe est la représentation d'une classe pour laquelle on a précisé chaque caractéristique.

9


Programmation objet : les classes
10
class Moto
Couleur
Vitesse
Repeindre()
Accélérer()
Freiner()
Attributs
Méthodes
Objet_Moto1
Verte
0 km/h
Repeindre()
Accélérer()
Freiner()
Objet_Moto2
Bleue
50 km/h
Repeindre()
Accélérer()
Freiner()
Objet_Moto3
Verte
130 km/h
Repeindre()
Accélérer()
Freiner()

10

Programmation objet

11

- La notion d'objet permet la modularité et l'encapsulation. Isoler les erreurs et facilité la maintenance.
- La notion de classe fournit un mécanisme de réutilisabilité.
- Java est un langage totalement objet : un programme est constitué lui-même d'une classe comportant (entre autres) une méthode « main » dans laquelle se trouve le code.
- Lors de l'exécution des programmes, des instances de classes sont créées et détruites au fur et à mesure des besoins.
- L'environnement java offre un ensemble standard de classe (bibliothèques ou packages) remplissant les fonctions de base : fonctions mathématiques, vecteurs, chaînes de caractères, etc.

11

Programmation objet : les attributs

12

Les variables d'instance définissent les attributs d'un objet. Leurs types et noms sont précisés au sien de la classe qui permet de créer cet objet. Leurs valeurs sont fixées et modifiées dans l'objet.

- Ex : dans la classe Moto, deux variables d'instance sont définies («couleur» et «vitesse»). Les valeurs de «couleur» fixées pour chacun des objets créés sont vertes, bleu et rouge. Ces dernières peuvent être modifiées au cours de l'exécution du programme si une méthode l'autorise.

Il peut exister des variables de classe (static) qui s'appliquent à la classe elle-même et à tous ses objets. Leurs valeurs sont stockées dans la classe.

- Ex : un compteur d'objets créés à partir de la classe. Ce compteur est général à cette dernière et non pas propre à chacune des instances. Ici : 3 motos.

12

Programmation objet : les méthodes

13

Les méthodes d'instances définissent les opérations réalisables sur les objets (fonctions propres aux objets). Elles sont codées dans les classes et leur portée se limite à celles-ci.

- Ex : dans la classe MOTO sont définies les méthodes Repeindre ( ), Accélérer ( ) et Freiner ( ).

Java ne possède pas de fonction définie en-dehors de classes.

- On utilise alors des méthodes « de classe ».

Les méthodes de classes (static) agissent sur une classe ou sur d'autres objets et n'ont pas forcément besoin qu'un objet soit créé pourvoir être utilisées.

- Ex : la méthode « main »

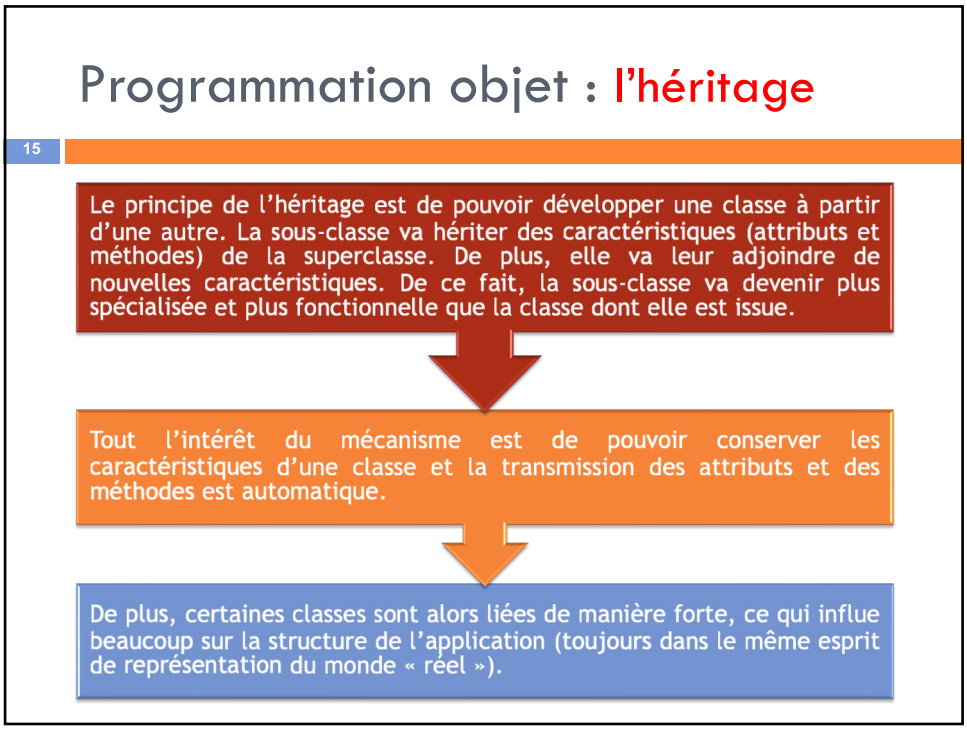
13

Programmation objet : synthèse

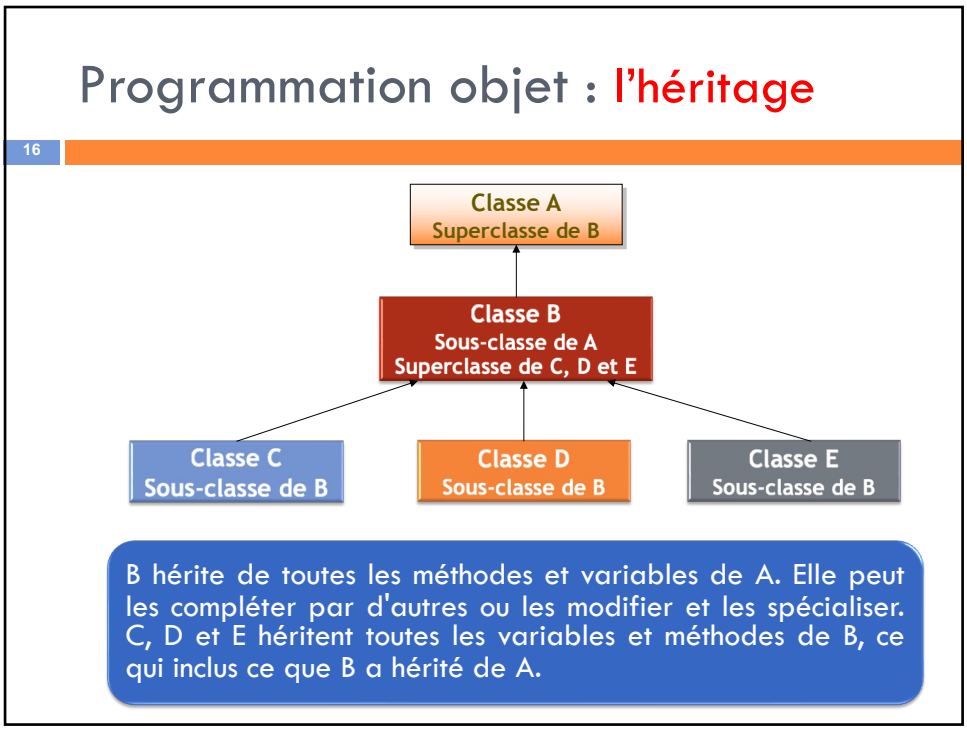
14

- Une classe est définit par :
  - Les attributs
  - Les méthodes
- Les attributs permettent de décrire l'état des objets de cette classe.
  - Chaque attribut est défini par :
    - Son nom
    - Son type
    - Eventuellement sa valeur initiale
- Les méthodes permettent de décrire le comportement des objets de cette classe. Parmi les méthodes d'une classe, existe une méthode particulière :
  - Est appelée au moment de la création d'un objet de cette classe. Cette méthode est appelée CONSTRUCTEUR

14


Programmation objet : l'héritage
15
Le principe de l'héritage est de pouvoir développer une classe à partir d'une autre. La sous-classe va hériter des caractéristiques (attributs et méthodes) de la superclasse. De plus, elle va leur adjoindre de nouvelles caractéristiques. De ce fait, la sous-classe va devenir plus spécialisée et plus fonctionnelle que la classe dont elle est issue.
Tout l'intérêt du mécanisme est de pouvoir conserver les caractéristiques d'une classe et la transmission des attributs et des méthodes est automatique.
De plus, certaines classes sont alors liées de manière forte, ce qui influe beaucoup sur la structure de l'application (toujours dans le même esprit de représentation du monde « réel »).

15

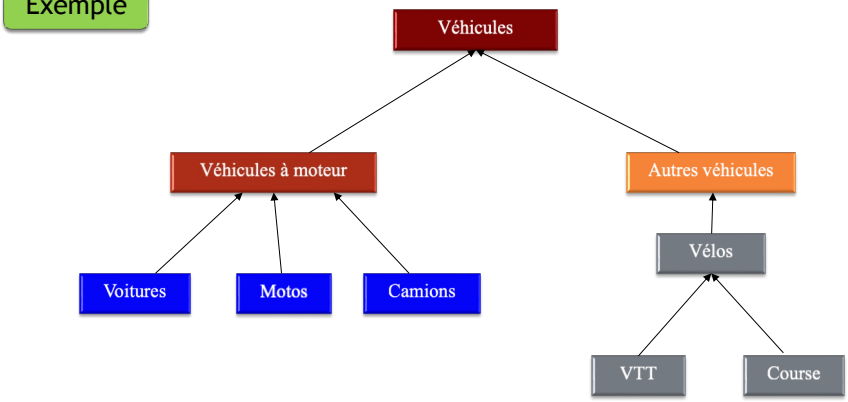

Programmation objet : l'héritage
16
Classe A
Superclasse de B
Classe B
Sous-classe de A
Superclasse de C, D et E
Classe C
Sous-classe de B
Classe D
Sous-classe de B
Classe E
Sous-classe de B
B hérite de toutes les méthodes et variables de A. Elle peut les compléter par d'autres ou les modifier et les spécialiser. C, D et E héritent toutes les variables et méthodes de B, ce qui inclus ce que B a hérité de A.

16

Programmation objet : l'héritage

17

Exemple



17

Programmation objet : l'héritage

18

Héritage = spécialisation. Un VTT est un vélo spécialisé. Le premier possède plus de caractéristiques que le second.

- Mot-clé : *extends* si une classe B hérite d'une classe C, on a :
- class B extends C (…)

Toutes les classes suivent une hiérarchie stricte.

- Chaque classe possède une super-classe et peut avoir une ou plusieurs sous-classes.

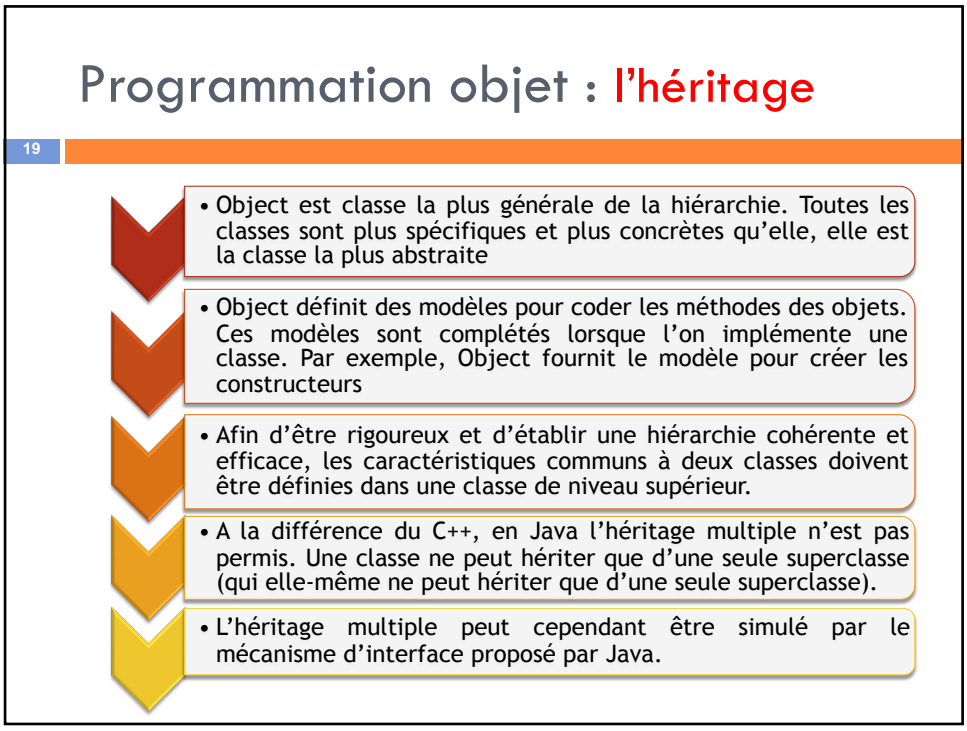
En Java, toutes les classes suivant une héritent de classe Object, classe de base du langage Java.

- class Moto (…) est équivalent à class Moto extends Object(…)

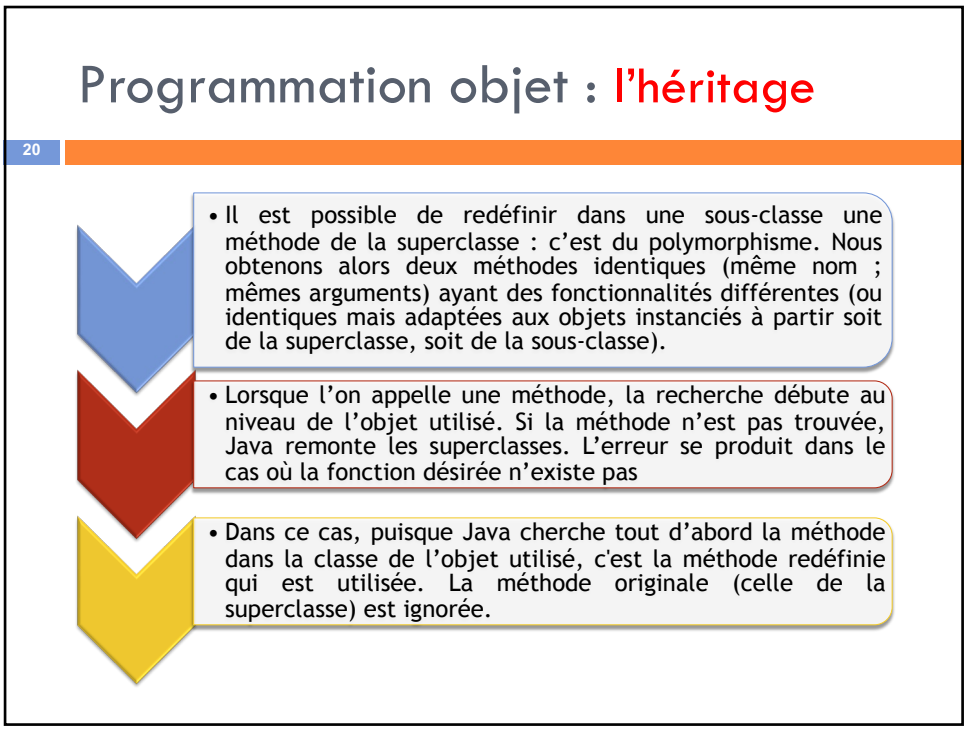
En l'absence du mot extends

- Une classe est considérée par défaut comme héritant de la classe Object.

18


Programmation objet : l'héritage
19
• Object est classe la plus générale de la hiérarchie. Toutes les classes sont plus spécifiques et plus concrètes qu'elle, elle est la classe la plus abstraite
• Object définit des modèles pour coder les méthodes des objets. Ces modèles sont complétés lorsque l'on implémente une classe. Par exemple, Object fournit le modèle pour créer les constructeurs
• Afin d'être rigoureux et d'établir une hiérarchie cohérente et efficace, les caractéristiques communs à deux classes doivent être définies dans une classe de niveau supérieur.
• A la différence du C++, en Java l'héritage multiple n'est pas permis. Une classe ne peut hériter que d'une seule superclasse (qui elle-même ne peut hériter que d'une seule superclasse).
• L'héritage multiple peut cependant être simulé par le mécanisme d'interface proposé par Java.

19


Programmation objet : l'héritage
20
• Il est possible de redéfinir dans une sous-classe une méthode de la superclasse : c'est du polymorphisme. Nous obtenons alors deux méthodes identiques (même nom ; mêmes arguments) ayant des fonctionnalités différentes (ou identiques mais adaptées aux objets instanciés à partir soit de la superclasse, soit de la sous-classe).
• Lorsque l'on appelle une méthode, la recherche débute au niveau de l'objet utilisé. Si la méthode n'est pas trouvée, Java remonte les superclasses. L'erreur se produit dans le cas où la fonction désirée n'existe pas
• Dans ce cas, puisque Java cherche tout d'abord la méthode dans la classe de l'objet utilisé, c'est la méthode redéfinie qui est utilisée. La méthode originale (celle de la superclasse) est ignorée.

20

Programmation objet : polymorphisme

21

Si le besoin s'en fait sentir, il est tout de même possible d'accéder à méthode de la superclasse au lieu de la méthode redéfinie dans la sous-classe.

Pour cela, on utilise le mot-clé "super" pour désigner la superclasse (de la même manière que "this" désigne l'objet courant).

- Exp : super.methode();

Note complémentaire : dans la terminologie objet « surcharge de méthode » et « redéfinition de méthode » représentent deux mécanismes de polymorphisme.

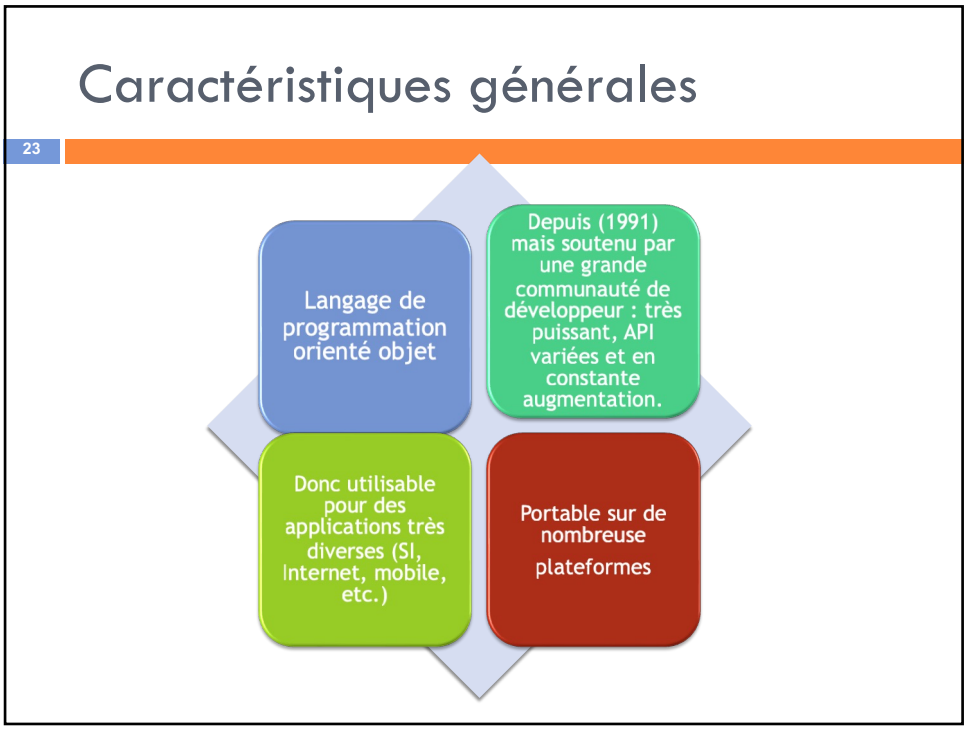
21

Chapitre II

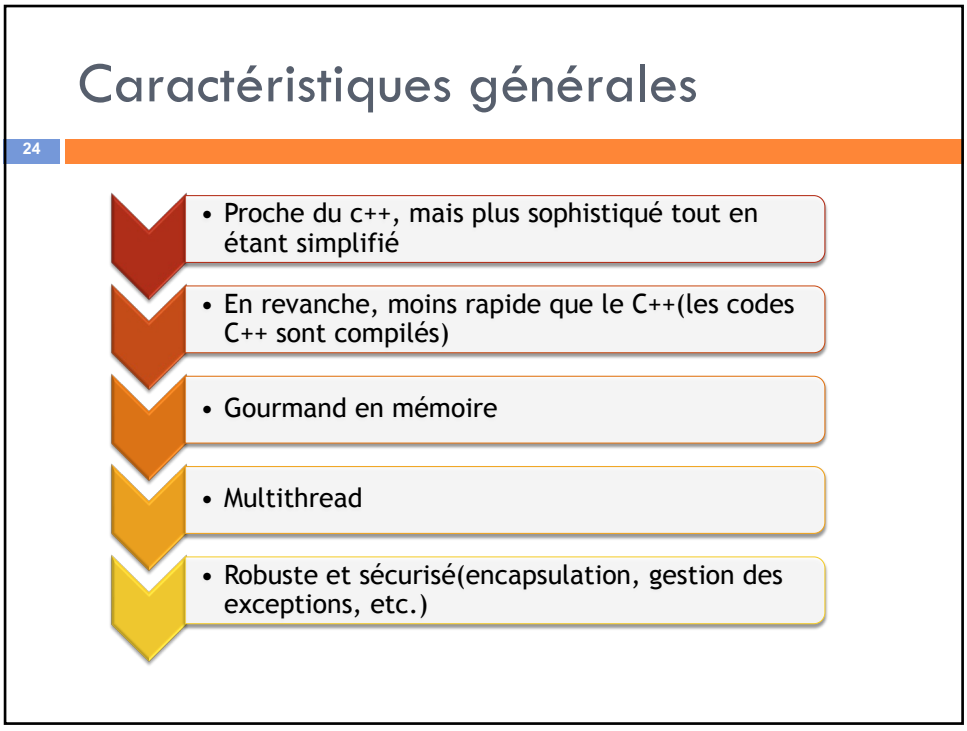
22

CARACTÉRISTIQUES GÉNÉRALES

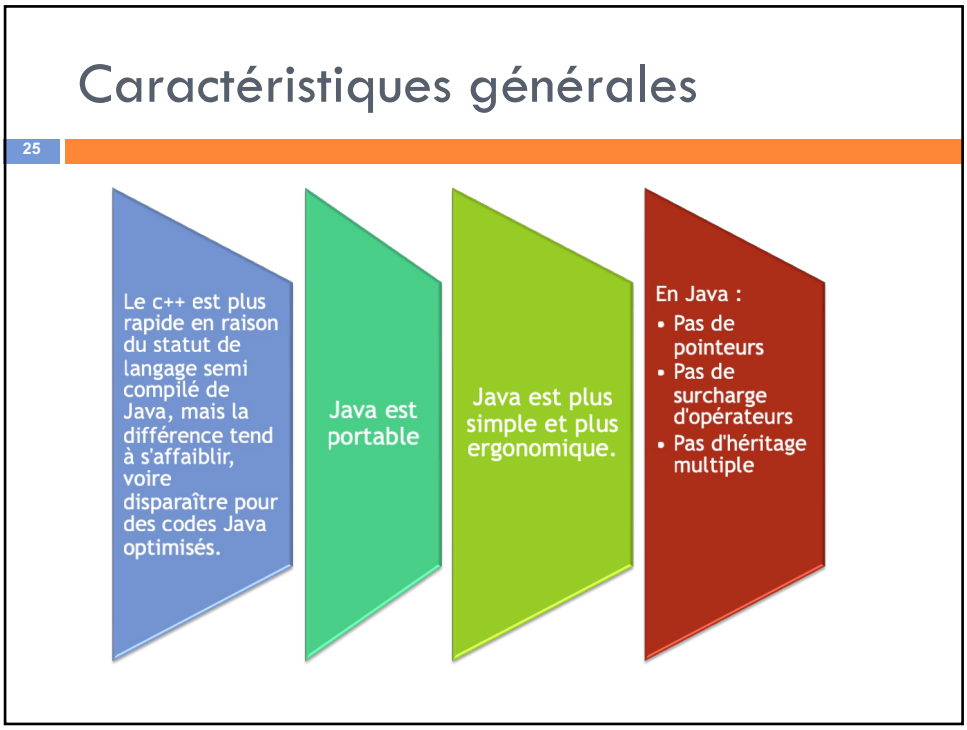
22


Caractéristiques générales
23
Langage de programmation orienté objet
Depuis (1991) mais soutenu par une grande communauté de développeur : très puissant, API variées et en constante augmentation.
Donc utilisable pour des applications très diverses (SI, Internet, mobile, etc.)
Portable sur de nombreuse plateformes

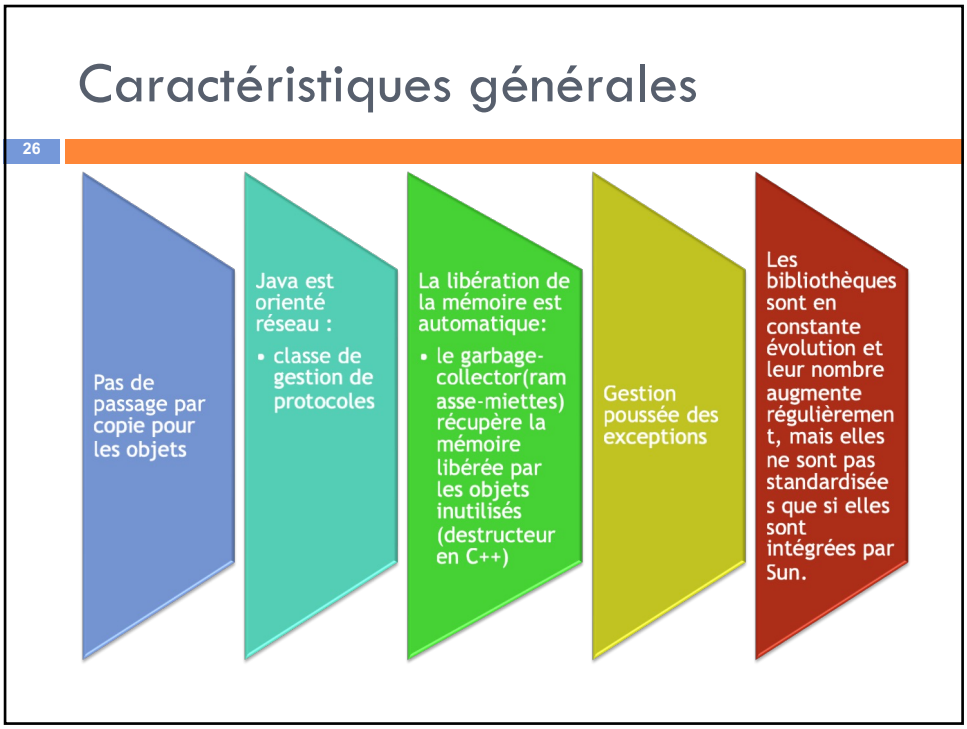
23


Caractéristiques générales
24
• Proche du c++, mais plus sophistiqué tout en étant simplifié
• En revanche, moins rapide que le C++(les codes C++ sont compilés)
• Gourmand en mémoire
• Multithread
• Robuste et sécurisé(encapsulation, gestion des exceptions, etc.)

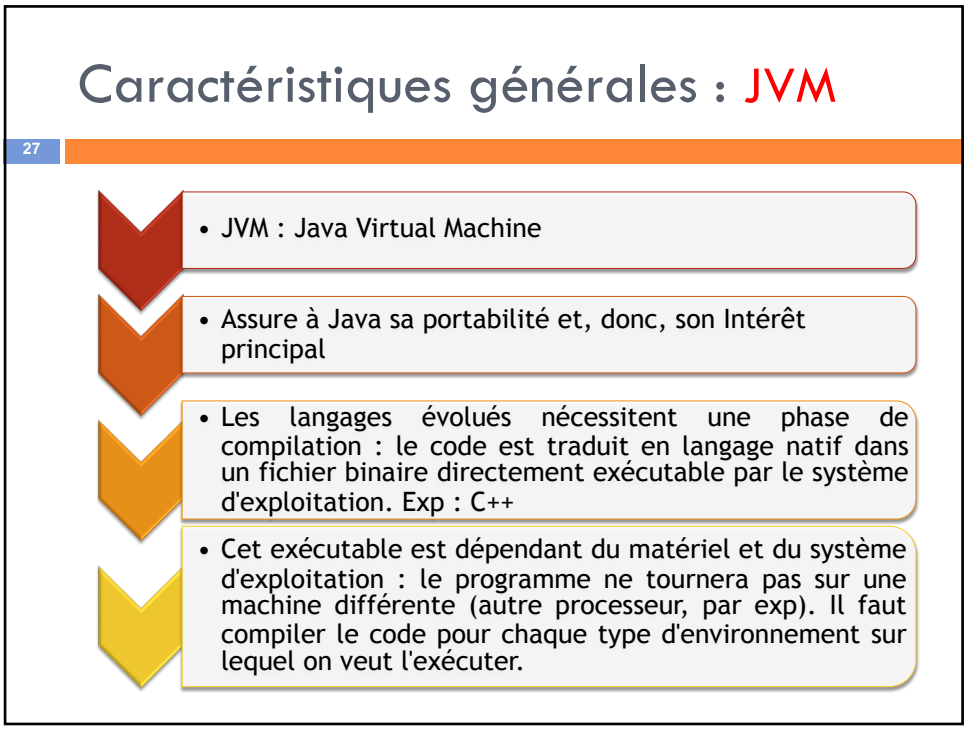
24


Caractéristiques générales
25
Le c++ est plus rapide en raison du statut de langage semi compilé de Java, mais la différence tend à s'affaiblir, voire disparaître pour des codes Java optimisés.
Java est portable
Java est plus simple et plus ergonomique.
En Java :
• Pas de pointeurs
• Pas de surcharge d'opérateurs
• Pas d'héritage multiple

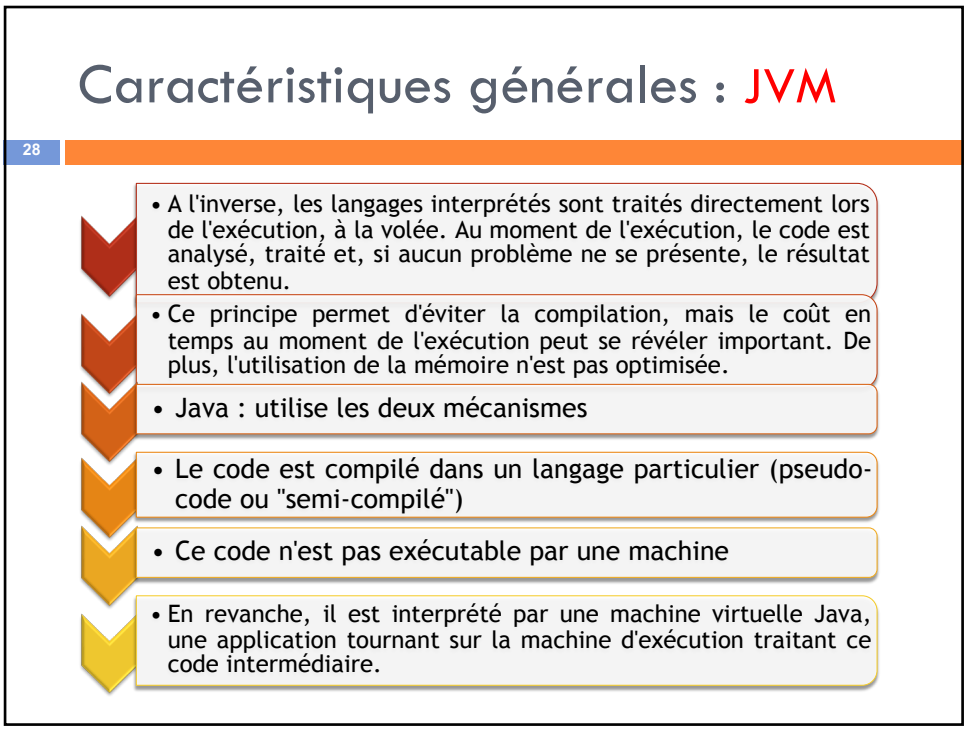
25


Caractéristiques générales
26
Pas de passage par copie pour les objets
Java est orienté réseau :
• classe de gestion de protocoles
La libération de la mémoire est automatique:
• le garbage-collector(ramasse-miettes) récupère la mémoire libérée par les objets inutilisés (destructeur en C++)
Gestion poussée des exceptions
Les bibliothèques sont en constante évolution et leur nombre augmente régulièrement, mais elles ne sont pas standardisées que si elles sont intégrées par Sun.

26


Caractéristiques générales : JVM
27
• JVM : Java Virtual Machine
• Assure à Java sa portabilité et, donc, son Intérêt principal
• Les langages évolués nécessitent une phase de compilation : le code est traduit en langage natif dans un fichier binaire directement exécutable par le système d'exploitation. Exp : C++
• Cet exécutable est dépendant du matériel et du système d'exploitation : le programme ne tournera pas sur une machine différente (autre processeur, par exp). Il faut compiler le code pour chaque type d'environnement sur lequel on veut l'exécuter.

27


Caractéristiques générales : JVM
28
• A l'inverse, les langages interprétés sont traités directement lors de l'exécution, à la volée. Au moment de l'exécution, le code est analysé, traité et, si aucun problème ne se présente, le résultat est obtenu.
• Ce principe permet d'éviter la compilation, mais le coût en temps au moment de l'exécution peut se révéler important. De plus, l'utilisation de la mémoire n'est pas optimisée.
• Java : utilise les deux mécanismes
• Le code est compilé dans un langage particulier (pseudo-code ou "semi-compilé")
• Ce code n'est pas exécutable par une machine
• En revanche, il est interprété par une machine virtuelle Java, une application tournant sur la machine d'exécution traitant ce code intermédiaire.

28

Caractéristiques générales : JVM

29

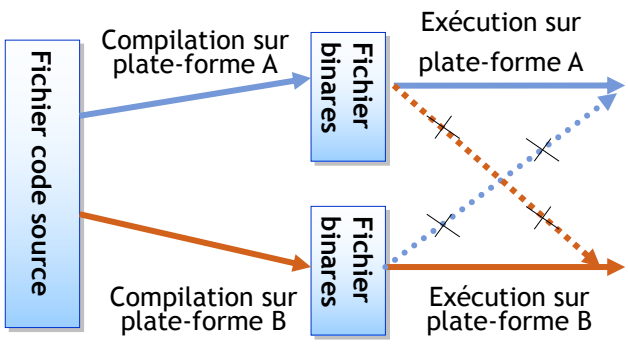
- Ceci explique la "lenteur" relative de java par rapport à C++…
- … mais procure à Java sa portabilité
- La JVM est propre à la plateforme (matériel, OS,…), mais le code devient indépendant de la machine.
- Donc, un programme tapé en Java est exécutable sur toute plate-forme comportant une machine virtuelle. Il n'est pas nécessaire de réécrire le code, ni de le recompiler pour changer de machine. La compilation est réalisée une fois et le code intermédiaire est interprété par la JVM.

29

Caractéristiques générales

30

- Cas d'un langage compilé

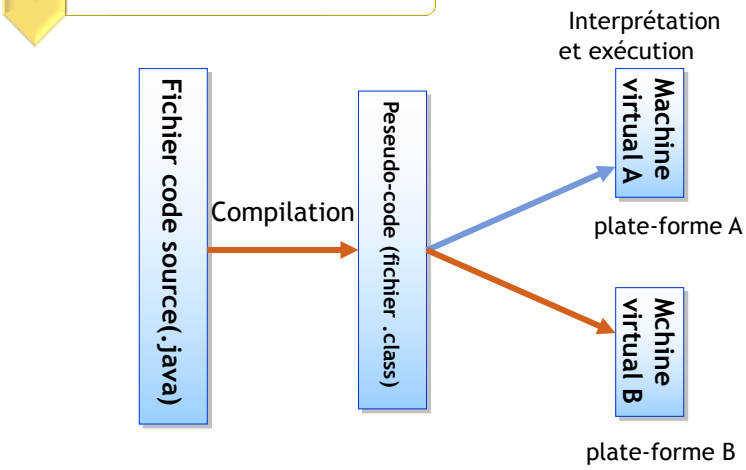


30

Caractéristiques générales : JVM

31

- Cas du langage java

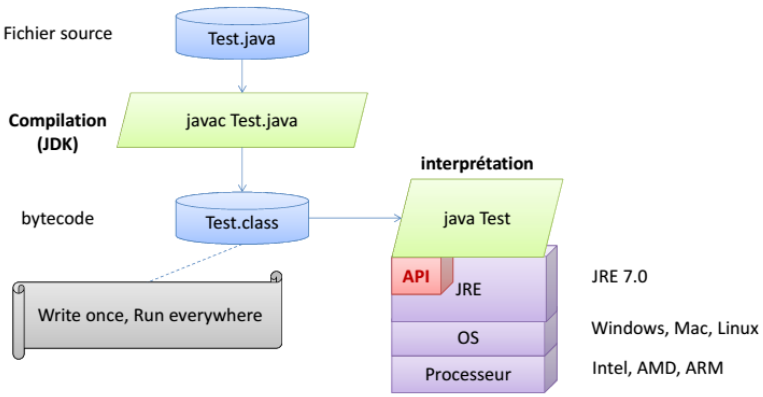


31

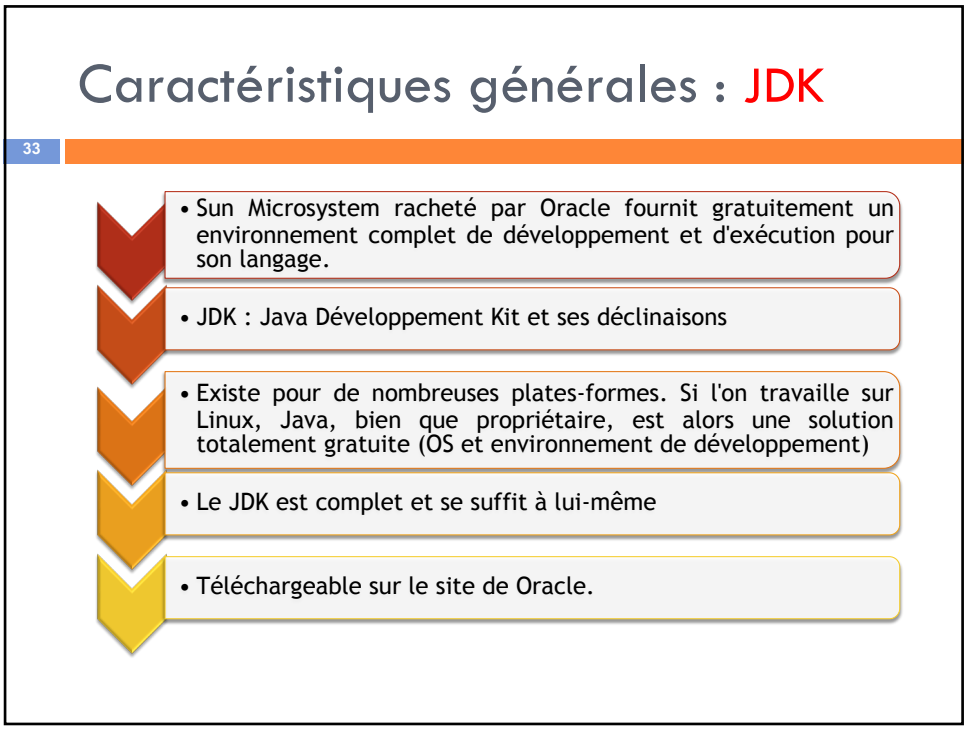
Caractéristiques générales

32

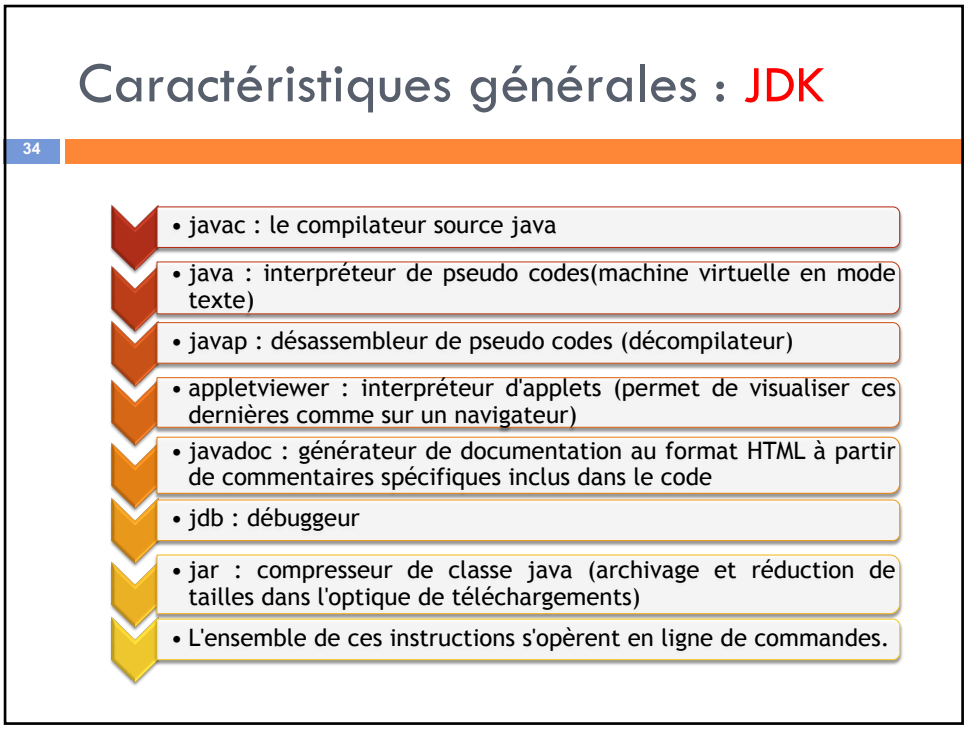
- Processus de compilation en java



32


Caractéristiques générales : JDK
33
• Sun Microsystem racheté par Oracle fournit gratuitement un environnement complet de développement et d'exécution pour son langage.
• JDK : Java Développement Kit et ses déclinaisons
• Existe pour de nombreuses plates-formes. Si l'on travaille sur Linux, Java, bien que propriétaire, est alors une solution totalement gratuite (OS et environnement de développement)
• Le JDK est complet et se suffit à lui-même
• Téléchargeable sur le site de Oracle.

33


Caractéristiques générales : JDK
34
• javac : le compilateur source java
• java : interpréteur de pseudo codes(machine virtuelle en mode texte)
• javap : désassembleur de pseudo codes (décompilateur)
• appletviewer : interpréteur d'applets (permet de visualiser ces dernières comme sur un navigateur)
• javadoc : générateur de documentation au format HTML à partir de commentaires spécifiques inclus dans le code
• jdb : débuggeur
• jar : compresseur de classe java (archivage et réduction de tailles dans l'optique de téléchargements)
• L'ensemble de ces instructions s'opèrent en ligne de commandes.

34

Caractéristiques générales : API

35

Les bibliothèques (packages) constituent des ensembles de classes adaptés à des fonctionnalités précises :

- java.lang : type de base, exceptions, …
- java.util : vector, date, …
- java.awt : composants pour constituer une interface graphique
- java.io : accès aux entrées/sorties
- java.net : fonctionnalités réseau (Sockets, url,..)
- java.beans : beans =composants logiciels réutilisables
- java.sql : API d'accès aux bases de données(via JDBC)
- java.security : signature, cryptographie, authentification
- java.rmi : Remote Method Invocation : appels des objets distants
- …

35

Chapitre III

36

LA PROGRAMMATION JAVA

36

Programmation Java

37

Java étant un langage totalement orienté objet, une application est constitué d'une classe intégrant une méthode publique « main ».

- Exemple d'application provoquant l'affichage d'un message à l'écran :

```
public class Appli {
 public static void main(String [] args) {
        system.out.println("Hello world");
 }
}
```

37

Programmation Java

38

«class»

- permet de définir une classe. L'instruction est suivie du nom donné à celle-ci. Le code de la classe est inclus entre accolades (en règle générale, les accolades délimitent des ensembles d'instructions fonctionnant ensemble).

«public»

- est optionnel et définit les droits d'accès relatifs à cette classe, c'est-à-dire la manière dont les autres éléments de l'application peuvent « accéder » à la classe.

- Attention : « accéder » ne veut pas dire modifier, mais simplement pouvoir invoquer cette classe (créer un objet, par exemple). Cette notion prend toute son importance dans le cas où l'application gère plusieurs package. « public » donne la possibilité à tout élément de l'application d'utiliser la classe (mais pas d'accéder aux éléments internes de cette dernière). Si les droits ne sont pas précisés, ils sont attribués par défaut (*friendly*).

38

Programmation Java

39

« **main** » est la méthode principale de l'application et c'est elle qui sera appelée lors de l'exécution du programme (point d'entré du programme). Plusieurs éléments sont à préciser lors du codage de cette fonction :

- « public » précise que main est une fonction publique accessible sans restriction. Cette condition est nécessaire pour que la machine virtuelle puisse traiter le code.
- « static » définit main comme une méthode de classe. En effet, lors de l'exécution de l'application, aucun objet n'est encore créé, il faut donc que main soit une méthode de classe afin d'être appelée même en l'absence d'objets.
- « void » précise, comme le terme en C++, ce que va retourner la méthode main lors de l'achèvement de son exécution. En l'occurrence, rien dans le cas présent.
- Entre les parenthèse un tableau de Strings nommé args. Dans le cas particulier de main, ce tableau reçoit les éventuels paramètres précisés par l'utilisateur dans la ligne de commande lançant l'exécution de l'application.

39

Programmation Java : Exécution

40

Le code doit être placé dans un fichier de nom IDENTIQUE au nom donné à la classe principale de l'application (attention, Java est sensible à la casse).	• Ici, le fichier sera nommé « Appli.java » (aucune autre possibilité).
Le programme doit être ensuite compilé en pseudo-code.	• Pour cela, utiliser le JDK ou l'application RAD désirée.
Sous Dos, en utilisant le compilateur du JDK :	• javac Appli.java • Le JDK va créer un fichier Appli.class

40

Programmation Java : Exécution

41

Le pseudo-code de ce fichier doit ensuite être interprété par la machine virtuelle. Toujours sous Dos et en utilisant le JDK :

- java Appli

Attention, l'extension .class ne doit pas être précisée.

Ceci va lancer l'appel de la méthode main. Dans le cas présent, le résultat sera l'affichage à l'écran du message Hello World

Pour pouvoir créer un objet autre que l'application elle-même, un constructeur doit être codé (cf. exemple suivant).

41

Programmation Java

42

Exemple

```
public class Moto {
// déclaration des attributs
   private String marque ;
   private String couleur,
   private boolean moteur_en marche = false ;
// déclaration du constructeur
   public Moto (String societe, String couleur) {
         marque = societe ;
         this.couleur = couleur;
   }// fin du constructeur
```

42

Programmation Java

43

```
// déclaration des méthodes autres que le(s) constructeurs
   public void demarrer_moteur(){
        if (moteur_en_marche==false) {
                moteur_en_marche=true ;
                System.out.println("Le moteur tourne. ") ;
        }
        else
        System.out.println("Le moteur est déjà en marche.") ;
   }  // fin de la méthode demarrer_mteur()
} // fin de la classe
```

Exemple

43

Programmation Java

44

- Note : ici, pas de méthode main. Le code ne peut être lancé mais pourrait être inclus dans une application fonctionnelle.
- De la même manière qu'en C++, // et/* */ délimitent des commentaires, non traités lors de l'exécution du programme (// pour des commentaires d'une seule ligne/* */ pour des commentaires occupant éventuellement plusieurs lignes).
- Toute instruction est suivie d'un point-virgule. Indique au compilateur la fin de chacune d'elles.
- Comme en C++, *this* représente l'objet courant
- Création d'une classe (cf. plus haut) :
  - Mot-clé : class
  - Contient les déclarations des attributs et méthodes

44

Programmation Java

45

La classe Moto précédente n'était pas fonctionnelle en tant que telle. Aucun objet Moto n'était créé par cette classe car elle ne comportait pas de méthode main, nécessaire pour coder le fonctionnement d'une application

En revanche, une autre classe comportant une méthode main aurait pu créer de tels objets pour peu qu'on lui ait précisé le chemin d'accès à la classe Moto (instruction « import »)

45

Programmation Java

46

Exemple

```
public class Moto {
    private String marque, couleur;
    private boolean moteur_ en_marche= false ;
    public Moto (String societe, String couleur){
            … // constructeur cf. plus haut
            }
    public void demarrer_moteur (){
            … //cf. plus haut
            }
    public static void main (String [ ] args) {
    Moto moto_1 = new Moto ("honda", "verte") ; // création d'un objet
                                                  //moto_1
    Moto moto_2 = new moto ("honda", "bleue") ; // création d'un objet
                                                  //moto_2
    moto_1.demarrer_ moteur();
            /* appel de la méthode demarrer _moteur () de l'objet moto_1*/
    } // fin de la méthode main
}   // fin de la classe moto
```

46

Programmation Java : utilisation des objets

47

Pour créer un objet à partir d'une classe, il faut :

- Qu'un constructeur soit utilisé (par défaut, il y a un)
- Utiliser la syntaxe suivante :
  - nom_ de_la classe nom_de_l_objet ;
  - nom_de_l_objet = new nom_de_la_classe(arg) ;
- Ex : Moto moto_1 ;
- **/* déclaration de la variable objet mot_1 de type Moto. Aucune réservation mémoire à ce stade. */**
  - moto_1=new Moto ("honda", "verte") ;
- **/*Instanciation de moto_1 par appel du constructeur Moto (String marque, String couleur) */**

Comme dans l'exemple, l'ensemble peut être regroupé en une instruction unique :

- Moto moto_1 = new Moto ("honda", "verte") ;

47

Programmation Java : utilisation des objets

48

- Un objet est autonome , identifiable et possède des caractéristiques précises (attributs et leurs valeurs, méthodes) .
- Pour utiliser un objet , il faut donc préciser son nom. Si l'on souhaite accéder à l'objet courant , on utilise "this" (ce dernier peut être sous-entendu).
  - Accès aux caractéristiques d'un objet : se fait par l'intermédiaire du point "."

    nom_de_objet.attribut

    Ex : "moto_1.couleur" désigne la valeur de l'attribut couleur de l'objet moto_1. Ainsi :

    System.out.println(moto_1.couleur) ;

    Peut permettre d'afficher sur la sortie standard (l'écran) cette valeur (ici , Verte) , si l'attribut est accessible .

48

Programmation Java : utilisation des objets

49

- On accède aux méthodes d'un objet de la même manière :

  nom_de_objet.méthode

  Ex : mot_1.demarrer_moteur() ;

- Ici, on appelle la méthode demarrer_moteur() de l'objet moto_1. Cette méthode teste la valeur du booléen moteur_en_marche de l'objet moto_1 et lui affecte la valeur "true" si ce n'est pas déjà le cas .

49

Programmation Java : utilisation des objets

50

Plusieurs constructeurs peuvent être définis avec des conditions d'exécution différentes (arguments différents) :

```
public Moto (String societe, String couleur){
   marque =societe;
   this.couleur = couleur ;}
```

NB : La méthode initialise les attributs couleur et marque de "this" (l'objet créé par le constructeur, donc l'objet courant) avec les valeurs passées en arguments.

```
public Moto (String couleur) {
   this.colueur = couleur ; }
```

NB : si par défaut l'attribut marque possède déjà une valeur fixée lors de sa déclaration, on peut définir ce constructeur qui ne gère que l'attribut couleur.

50

Programmation Java : utilisation des objets

51

public Moto () { }

NB : ce constructeur construit un objet moto si les attributs possèdent de valeurs par défaut.

Ces différents constructeurs peuvent coexister et selon les cas (en l'occurrence le nombre d'argument passées ou leurs type), la méthode adéquate sera appelée.

Ce mécanisme est nommé "surcharge" et est valable pour toute méthode : on peut surcharger une méthode existence, c'est-à-dire définir une seconde méthode de même nom mais avec un membre ou de type d'argument différent. Les deux méthodes sont parfaitement distinctes.

51

Programmation Java : utilisation des objets

52

Exemple

```
class Moyenne { // bloc de déclaration de classe
  /** Somme des notes obtenues */
  int notes = 0 ;
  /** Nombre de notes obtenues */
  int nombreDeNotes = 0 ;

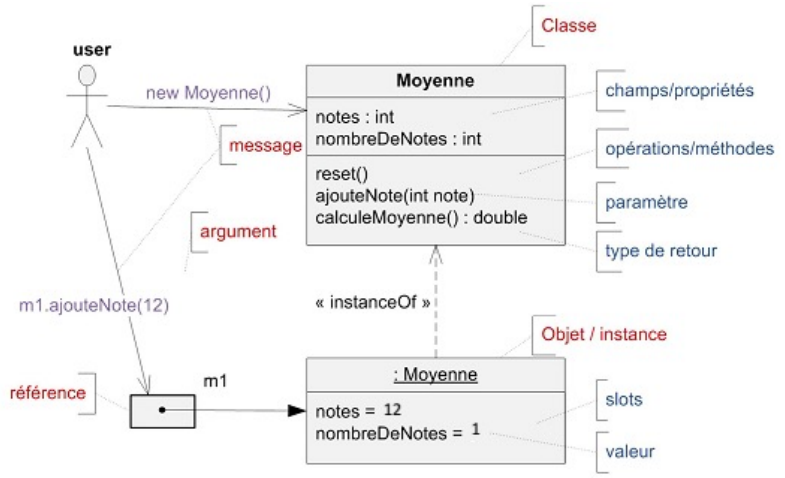
  /** Ajoute une note à la moyenne
   *  @param note nouvelle note obtenue */
  void ajouteNote (int note) {
     notes += note;
     nombreDeNotes += 1;
  }

  /** @return moyenne des notes */
  double calculMoyenne() {
    return ((double)notes) / nombreDeNotes ;
  }
}
```

52

Programmation Java : utilisation des objets

53



53

Getters et Setters

54

- Les attributs privés d'une classe ne sont accessibles qu'à l'intérieur de la classe.
- Pour donner la possibilité à d'autres classes d'accéder aux membres privés, il faut définir dans la classe des méthodes publiques qui permettent de :
  - lire les variables privées. Ce genre de méthodes s'appellent les **accesseurs** ou **getters**
  - modifier les variables privées. Ce genre de méthodes s'appellent les **mutateurs** ou **setters**

54

Sur-casting & sous-casting

55

- Le polymorphisme offre aux objets la possibilité d'appartenir à plusieurs catégories à la fois.
- En effet, nous avons certainement tous appris à l'école qu'il était impossible d'additionner des pommes et des oranges
- Mais, on peut écrire l'expression suivante:

  3 pommes + 5 oranges = 8 fruits

55

Sur-casting des objets

56

- Une façon de décrire l'exemple consistant à additionner des pommes et des oranges serait d'imaginer que nous disons pommes et oranges mais que nous manipulons en fait des fruits.
  - Nous pourrions écrire alors la formule correcte :

```
3 (fruits) pommes
+ 5 (fruits) oranges
-------------------------------
= 8 fruits
```

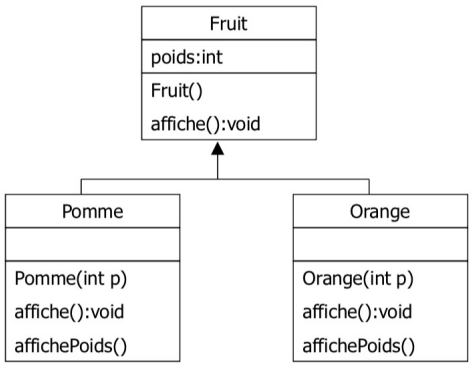
- Cette façon de voir les choses implique que les pommes et les oranges soient "transformés" en fruits. Cette transformation est appelée *sur-casting* (Upcasting)

56

Instanciation et héritage

57

- Considérons l'exemple suivant :



57

Instanciation et héritage

58

```
public class Fruit {
 int poids;
 public Fruit(){
   System.out.println("création d'un
fruit");
}
public void affiche(){
System.out.println("c'est un fruits");
}}
```

```
public class Pomme extends Fruit{
public Pomme(int p) {
 poids=p;
 System.out.println("création d'une pomme
de "+ poids+" gr");}
 public void affiche(){
  System.out.println("c'est une pomme");
 }
public void affichePoids(){
 System.out.println("le poids de la pomme
est "+ poids+" gr");}
}
```

```
public class Orange extends
Fruit{
 public Orange(int p) {
  poids=p;
  System.out.println("création
d'une orange de "+ poids+" gr"); }
 public void affiche(){
  System.out.println("c'est une
orange"); }
  public void affichePoids(){
    System.out.println("le poids
de l'orange est "+ poids+" gr"); }
}
```

```
public class MonAppli {
 public static void main(String[]
args)
 {
    Pomme p = new Pomme(40);
    Orange o = new Orange(50);
} }
```

58

Instanciation et héritage

59

- Le résultat affiché par le programme est :

```
création d'un fruit
création d'une pomme de 40 gr
création d'un fruit
création d'une orange de 50 gr
```

- Nous constatons qu'avant de créer une Pomme, le programme crée un Fruit, comme le montre l'exécution du constructeur de cette classe. La même chose se passe lorsque nous créons une Orange.

59

Sur-casting des objets

60

- Considérons l'exemple 2 suivant :

```
public class MonAppli {
   public static void main(String[] args) {
        // sur-casting implicite
        Fruit f1 = new Orange (40);
        // sur-casting explicite
        Fruit f2 = (Fruit) new Pomme (60);
        // sur-casting implicite
        f2= new Orange (40); }
}
```

Résultat

```
création d'un fruit
création d'une orange de 40 gr
création d'un fruit
création d'une pomme de 60 gr
création d'un fruit
création d'une orange de 40 gr
```

60

Sur-casting des objets

61

- Un objet de type Pomme peut être affecté à un objet de type fruit sans aucun problème :
  - Fruit f1;
  - f1=new Pomme(40);
- Dans ce cas l'objet Pomme est converti automatiquement en Fruit.
- On dit que l'objet Pomme est sur-casté en Fruit.
- Dans java, le sur-casting peut se faire implicitement.
- Toutefois, on peut faire le sur-casting explicitement sans qu'il soit nécessaire.
- Le casting explicite se fait en précisant la classe vers laquelle on convertit l'objet entre parenthèse. Exemple :
  - f2=**(Fruit)** new Orange(40);

61

Sous-casting des objets

62

- Considérons l'exemple suivant :

```
public class MonAppli {

 public static void main(String[] args)
{
        Fruit f1;
        Fruit f2;
        f1= new Pomme(60);
        f2= new Orange(40);

        f1.affichePoids();

        ((Pomme)f1).affichePoids(); }
}
```

- Erreur de compilation

LA méthode affichePoids() n'est pas définie pour la classe Fruit

- Solution sous-casting explicite

62

Sous-casting des objets

63

- Le message d'erreur indique que l'objet f1 qui est de type Fruit ne possède pas la méthode `affichePoids()`.
- Cela est tout à fait vrai car cette méthode est définie dans les classes Pomme et Orange et non dans la classe Fruit.
- En fait, même si l'objet f1 pointe un objet Pomme, le compilateur ne tient pas en considération cette affectation, et pour lui f1 est un Fruit.
- Il faudra donc convertir explicitement l'objet f1 qui de type Fruit en Pomme.
- Cette conversion s'appelle Sous-casting (down-casting) qui indique la conversion d'un objet d'une classe mère vers un autre objet d'une classe dérivée.
- Dans ce cas de figure, le sous-casting doit se faire explicitement.
- L'erreur de compilation peut être évité en écrivant la syntaxe suivante :
  - `((Pomme)f1).affichePoids();`
- Cette instruction indique que l'objet f1 , de type Fruit, est converti en Pomme, ensuite la méthode `affichePoids()` de l'objet Pomme est appelé ce qui est correcte.

63

Question

64

- Soit la définition de classe suivante :

```
1. public class Test extends Base {
2.     public Test (int j) {
3.     }
4.     public Test (int i, int j) {
5.       super(i,j);
6.     }
7. }
```

- Quelle forme de constructeur parmi les suivantes doivent-elles être explicitement présentes dans la définition de la classe *Base*
  - A. **Base() {}**
  - B. **Base (int j) {}**
  - C. **Base (int j, int k) {}**
  - D. **Base (int j, int k, int l) {}**

64

Question

65

- Quel est le résultat du code suivant :

```
public class Pass{
    static int j = 20;
    public static void main (String args[]){
            int i = 10;
            Pass p = new Pass ();
            p.maMethod ( i );
            System.out.println ( i );
            System.out.println ( j );
    }
    public void maMethod ( int x ){
            x = x * 2;
            j = j * 2;
    }
}
```

Résultat :

10

40

65

Chapitre IV

66

DROITS D'ACCÈS

66

Droit d'accès

67

- L'un de principaux avantages liés à la POO réside dans la sécurité inhérente au mécanisme d'encapsulation.
- En règle générale, on souhaite que les attributs d'un objet ne soient pas modifiables (et donc accessibles) par un autre élément que cet objet lui-même.
  - Ex 1: on ne souhaite pas que la valeur d'un attribut puisse être connu par un utilisateur externe.
  - Ex 2: un attribut ne doit prendre certaines valeurs critique pour le programme.

• Il faut donc pouvoir contrôler cela et ne pas fournir d'accès systématique à l'attribut.

67

Droit d'accès

68

- L'accès aux attributs est alors fourni uniquement si le programmeur le souhaite par l'intermédiaire de méthodes spécifiques. Ces dernières constituent l'interface publique de la classe (et des objets créés).
- Ces méthodes étant les seuls points d'accès aux attributs, on peut y coder tous les contrôles éventuellement souhaités.
  - Ex : dans une méthode permettant de change la valeur d'un attribut, on vérifie s'il ne s'agit pas d'une valeur critique. Auquel cas, la modification n'est pas autorisée.
- D'autres méthodes peuvent également ne pas être publiques si elles ne doivent être appelées que par l'objet lui-même.

68

Droit d'accès : public

69

Ces règles générales ne sont pas obligatoires, mais elles donnent à l'encapsulation tout son intérêt.

Plusieurs portées sont applicables aux attributs et méthodes.

« public » rend l'attribut ou la méthode accessible par n'importe quel élément externe.

```
class Moto {
   public String couleur = "rouge";
   }
```

Si un objet Moto_1 est créé, tout autre objet quel qu'il soit peut modifier l'attribut couleur de Moto_1

```
Moto Moto_1 = new Moto ( );
Moto_1.couleur = "Bleue"; // toujours possible
```

69

Droit d'accès : private

70

« private » rend l'attribut inaccessible à tout élément sauf aux méthodes de la classe. Même les classes filles (sous-classes) ne peuvent y accéder. Seule une méthodes publique définie pour ce rôle dans la classe peut permettre d'accéder à l'attribut.

```
class Moto {
   private String couleur = "rouge";
   }
```

Si un objet Moto_1 est créé, seule une méthode de la classe Moto peut modifier l'attribut couleur de Moto_1

```
Moto Moto_1 = new Moto() ;
Moto_1.couleur="bleue"; /* impossible si l'on ne se trouve pas dans la classe Moto */
```

70

Droit d'accès : private

71

Dans le cas d'attributs privés, l'accès ne peut se faire que par l'intermédiaire d'une méthode publique de la classe dédiée à cette tâche : elle peut accéder aux attributs et, comme elle est elle-même publique, un élément externe à la classe peut l'invoquer.

```
class Moto {
        private String couleur= "rouge";
 public void repeindre (String couleur){
        this.couleur = couleur; }
```

Si un objet Moto_1 est créé, un élément externe à la classe peut modifier l'attribut couleur de Moto_1 via repeindre ( )

```
Moto moto_1 =new Moto()
Moto_1.couleur= "Bleue"; /* impossible si l'on ne se
        trouve pas dans la classe Moto */
Moto_1.repeindre ("bleue") ; //toujours possible
```

71

Droit d'accès : protected

72

L'accès « protected » est un hybride des deux accès précédents. Suivant les cas, l'attribut ou méthode « protected » est considéré comme publique ou comme privé.

- Un champ « protected » sera considéré comme publique (donc accessibles sans restriction) pour :
  - Toutes les méthodes de la classe et celles du même package.
  - Toutes les méthodes des sous-classes de celle considérée (quel que soit les packages auxquels elles appartiennent)

Dans tous les autres cas, le champ est considéré comme privé.

72

Droit d'accès : friendly

73

« friendly » est l'accès par défaut.

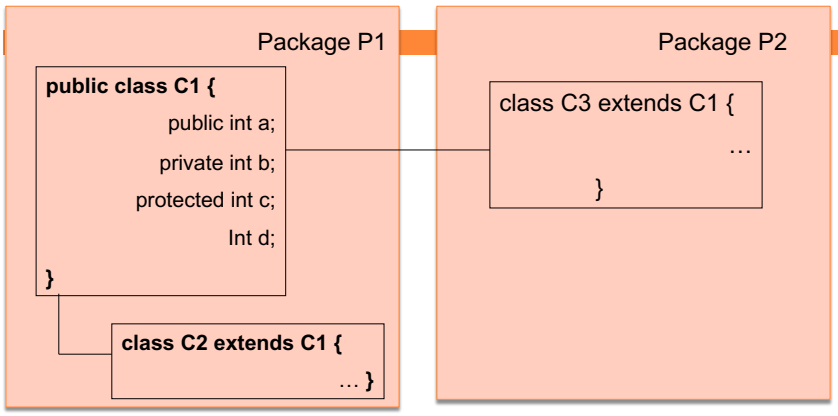
Le champ « friendly » est publique vis-à-vis de l'ensemble des méthodes du package contenant la classe considérée.

Il est privé pour toutes les méthodes des classes appartenant aux autres packages (même si ces classes sont des sous-classes de la classe considérée)

73

Droit d'accès

74



	a	b	c	d
Accessible par C1	X	X	X	X
Accessible par C2	X	-	X	X
Accessible par C3	X	-	X	-

74

Droit d'accès

75

Package P1

```
public class C1 {
        public int a;
        private int b;
        protected int c;
        Int d;
}
```

```
class C2 {
        … }
```

Package P2

```
class C3 {
        …
        }
```

	a	b	c	d
Accessible par C1	X	X	X	X
Accessible par C2	X	-	X	X
Accessible par C3	X	-	-	-

75

Question

76

Package P1

```
public class A{
        int a;
        public int b;
        private int c;
        protected int d;
        }
```

Package P2

```
public class C extends A{
        …
        }
```

Package P3 : Sous Package P1

```
public class B{
        …
        }
```

Pour la classe B : b

Pour la classe C : b,d

76

Question

77

- Quel est le droit d'accès le plus restrictive pour donner l'accès aux membres d'une autre classe de même package.

A. public
B. abstract
C. protected
D. Synchronized
E. default access

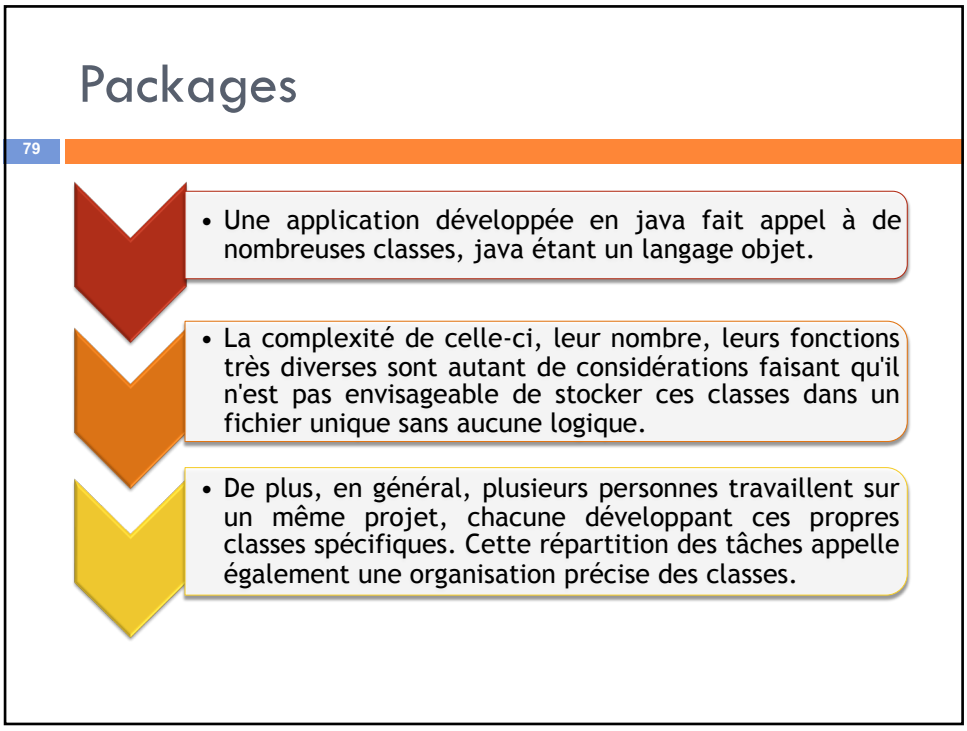
77

Chapitre V

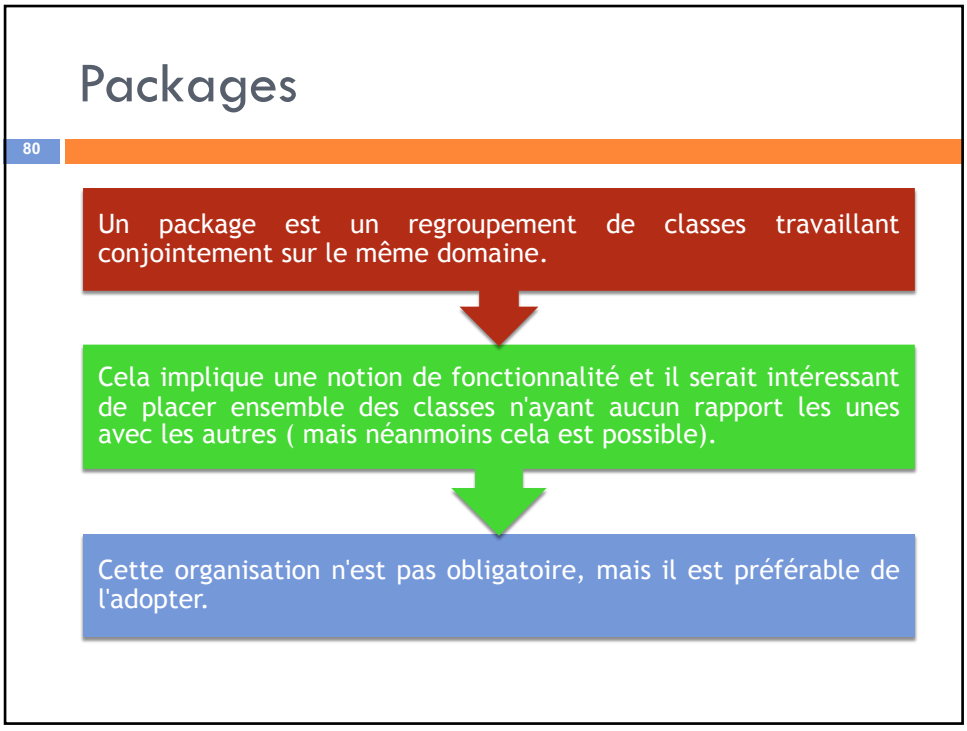
78

LES PACKAGES

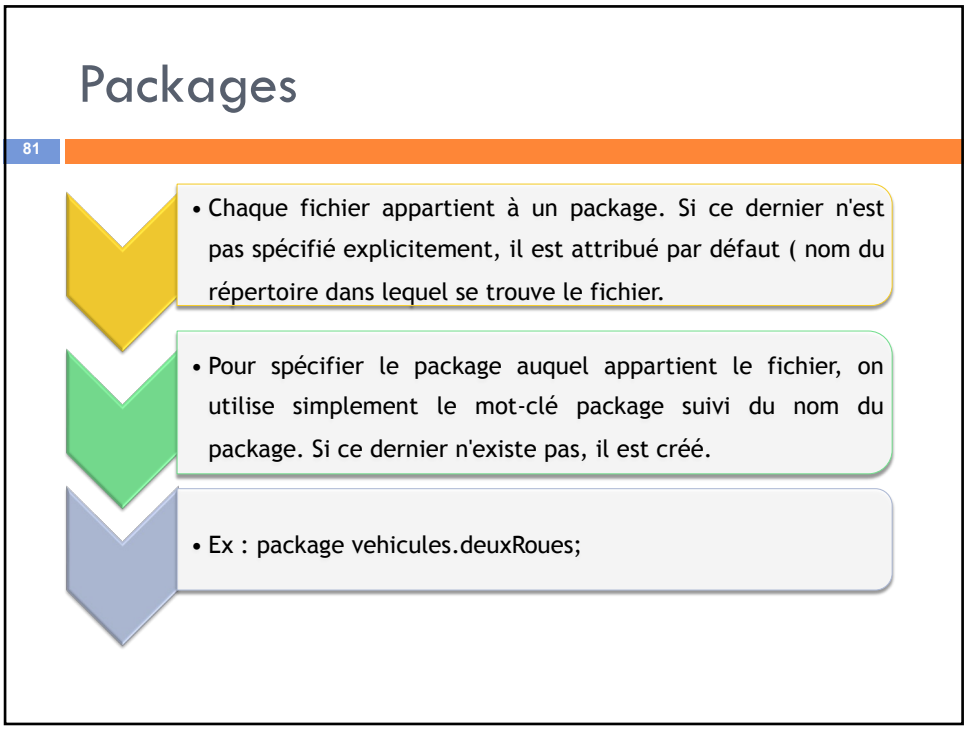
78


Packages
79
• Une application développée en java fait appel à de nombreuses classes, java étant un langage objet.
• La complexité de celle-ci, leur nombre, leurs fonctions très diverses sont autant de considérations faisant qu'il n'est pas envisageable de stocker ces classes dans un fichier unique sans aucune logique.
• De plus, en général, plusieurs personnes travaillent sur un même projet, chacune développant ces propres classes spécifiques. Cette répartition des tâches appelle également une organisation précise des classes.

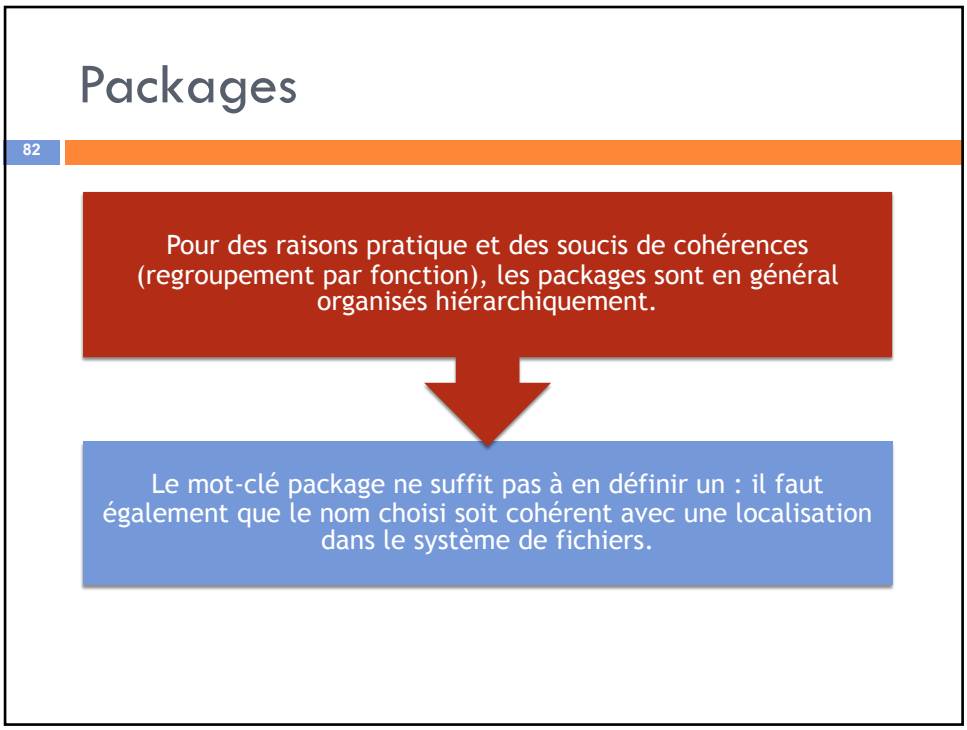
79


Packages
80
Un package est un regroupement de classes travaillant conjointement sur le même domaine.
Cela implique une notion de fonctionnalité et il serait intéressant de placer ensemble des classes n'ayant aucun rapport les unes avec les autres (mais néanmoins cela est possible).
Cette organisation n'est pas obligatoire, mais il est préférable de l'adopter.

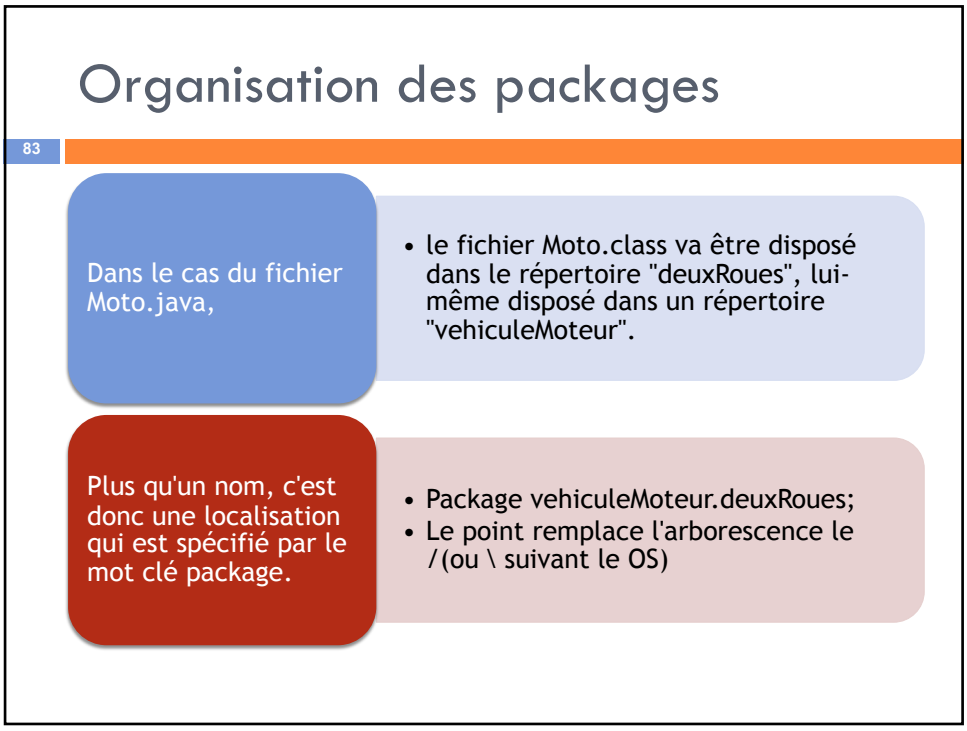
80


Packages
81
• Chaque fichier appartient à un package. Si ce dernier n'est pas spécifié explicitement, il est attribué par défaut (nom du répertoire dans lequel se trouve le fichier.
• Pour spécifier le package auquel appartient le fichier, on utilise simplement le mot-clé package suivi du nom du package. Si ce dernier n'existe pas, il est créé.
• Ex : package vehicules.deuxRoues;

81


Packages
82
Pour des raisons pratique et des soucis de cohérences (regroupement par fonction), les packages sont en général organisés hiérarchiquement.
Le mot-clé package ne suffit pas à en définir un : il faut également que le nom choisi soit cohérent avec une localisation dans le système de fichiers.

82


Organisation des packages
83
Dans le cas du fichier Moto.java,
• le fichier Moto.class va être disposé dans le répertoire "deuxRoues", lui-même disposé dans un répertoire "vehiculeMoteur".
Plus qu'un nom, c'est donc une localisation qui est spécifié par le mot clé package.
• Package vehiculeMoteur.deuxRoues;
• Le point remplace l'arborescence le /(ou \ suivant le OS)

83


Organisation des packages
84
Comment fait la machine virtuelle pour localiser les (.class) nécessaires à l'application.
Une forte corrélation existe entre la dénomination des packages et leur localisation, mais également entre cette localisation et l'environnement (l'OS)
Ainsi, il est nécessaire d'indiquer au système des informations nécessaires à la machine virtuelle lors de son fonctionnement : en l'occurrence, des emplacements des classes.

84

Organisation des packages

85

Pour cela, on utilise la variable d'environnement CLASSPATH

CLASSPATH va indiquer à la JVM le chemin vers les fichiers .class

Il est possible que cette variable n'existe pas, il faut alors la créer.

- Ex : on définit le CLASSPATH suivant :
- C:\...\classes; d:\...\lib;

85

Organisation des packages

86

Les points virgules délimitent les différentes localisations

Ces répertoires représentent les racines à partir desquelles la JVM va tenter d'accéder aux classes. Elle ne va pas chercher ailleurs.

Au moment de l'exécution, si des classes externes à l'application sont nécessaires, la JVM va les chercher dans ces localisations (exemple) :

- Le répertoire c:\...\classes
- Le répertoire d:\...\lib
- Le répertoire courant désigné par point.

86

Organisation des packages

87

Dans le cas d'un fichier Moto.java, nous avions définit un package "vehiculesMoteur.deuxRoues" dans lequel se positionnait le fichier Moto.class.

Au moment de l'exécution d'une l'application nécessitant une classe de ce fichier, la JVM va chercher le répertoire "vehiculesMoteur" dans chacun des emplacements définis dans le CLASSPATH (dont le répertoire courant). Si elle le trouve, elle cherche le répertoire "deuxRoues" du répertoire "vehiculesMoteur" et si elle le trouve elle accède à Moto.class.

87

Organisation des packages

88

Il est possible de définir le CLASSPATH de manière temporaire, au cours de l'exécution

- java -classpath c:\...\lib monAppli
- Ou de manière permanente :
- Dans le panneau de configuration de Windows et via une commande export "CLASSPATH=..." sous Linux

88

Organisation des packages

89

Java étant destiné à être employé fortement sur le web, il ne faut pas que les packages qui transitent aient éventuellement des noms identiques.

Le principe est d'allonger le nom du package par celui de la société, du site et/ou du concepteur inversé :

- Ex: tn.monSite.vehicule.Moto
- Bien sûr, ce nom est en relation avec une arborescence précise des répertoires : un répertoire "Moto" dans un répertoire "vehicule" dans un répertoire "monSite" dans un répertoire "tn"…

89

Utilisation des packages

90

Pour pouvoir utiliser les classes d'un package :

Il convient de le préciser au début du fichier .java

Syntaxe :

import monPackage.maClasse;

Ex: import java.awt.Button;

Ici, on importe la classe Button, contenue dans le package java.awt et, donc, localisé dans le répertoire CLASSPATH\java\awt.
→CLASSPATH désigne le répertoire racine.

90

Utilisation des packages

91

Il est possible d'importer l'ensemble des classes d'un package :

- import monPackage.*;

Il est également permis d'accéder directement à une classe d'un package dans le code d'une application (import est alors inutile).

- Ex:
- java.shape2D.Circle c1 =new java.shape2D.Circle( );

91

Question

92

- **La syntaxe Java permettant l'association d'une classe à un package est :**

A. package <nomDePackage> ;
B. import <nomCompletDeClasse> ;
C. <nomDePackage> contains <nomDeClasse> ;
D. package <nomDePackage>.<nomDeClasse> ;

92

Chapitre VI

93

LES DONNÉES : VARIABLES, CONSTANTES ET PRIMITIVES

93

Les Variables

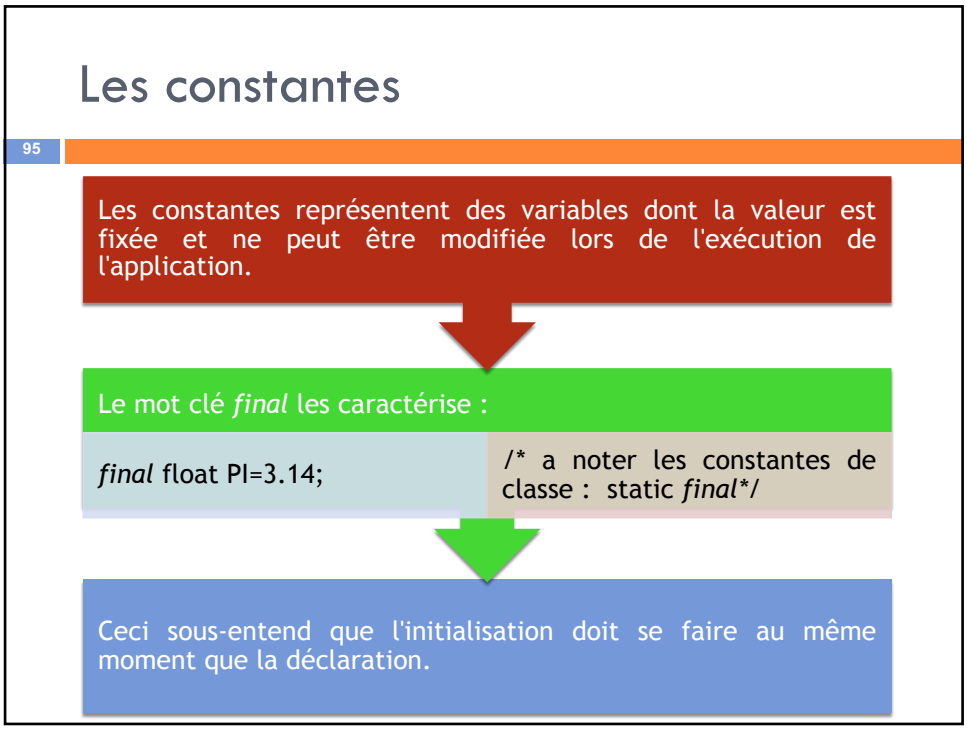
94

On distingue trois sortes de variables :

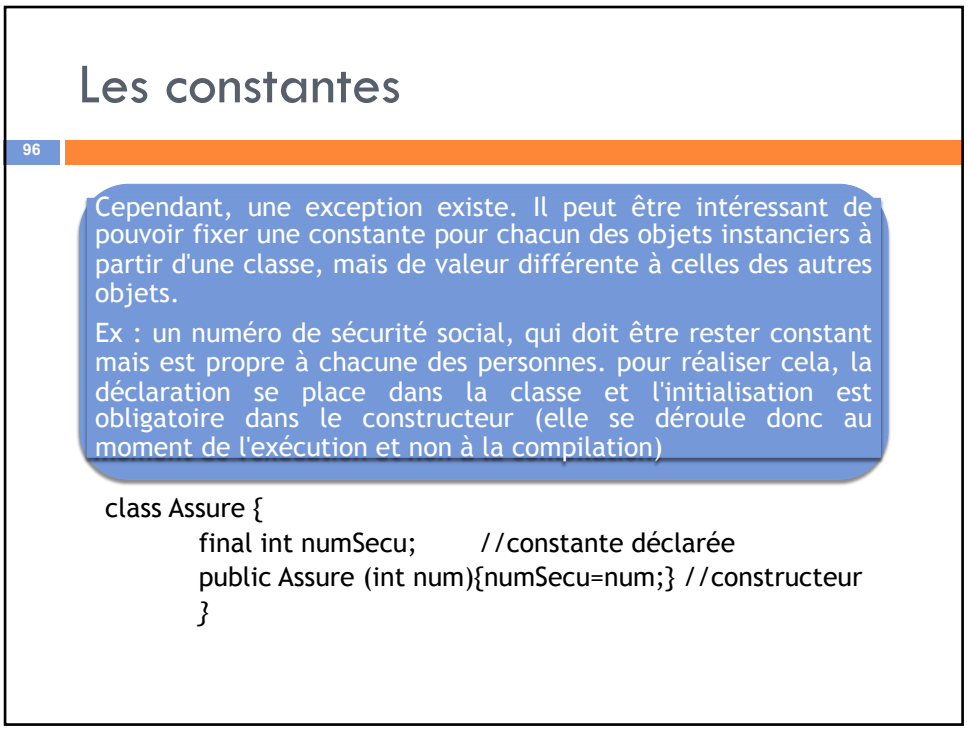
- Les variables d'instances associées aux attributs des objets
- Les variables de classes (static) dont les valeurs s'appliquent à la classe et à tous les objets qui en découles (ex: compteur d'objet)
- Les variables locales déclarées et utilisées ponctuellement: au sein d'une méthode, d'un bloc d'instruction, d'une boucle, etc. une variable locale a une durée de vie limitée à la portion de code qui l'implémente. Si elle est codée en dehors de tout bloc, elle persiste le temps de l'application. Si c'est dans un bloc, elle disparaît à la fin de ce dernier. Si c'est dans une méthode, elle n'existe que le temps de l'exécution de la méthode. Etc.

En java, les variables globales n'existent pas. Les variables d'instance et de classe les remplacent et permettent des échanges d'informations dans et entre les objets.

94


Les constantes
95
Les constantes représentent des variables dont la valeur est fixée et ne peut être modifiée lors de l'exécution de l'application.
Le mot clé final les caractérise :
final float PI=3.14;
/* a noter les constantes de classe : static final*/
Ceci sous-entend que l'initialisation doit se faire au même moment que la déclaration.

95


Les constantes
96
Cependant, une exception existe. Il peut être intéressant de pouvoir fixer une constante pour chacun des objets instanciers à partir d'une classe, mais de valeur différente à celles des autres objets.
Ex : un numéro de sécurité social, qui doit être rester constant mais est propre à chacune des personnes. pour réaliser cela, la déclaration se place dans la classe et l'initialisation est obligatoire dans le constructeur (elle se déroule donc au moment de l'exécution et non à la compilation)
class Assure {
final int numSecu; //constante déclarée
public Assure (int num){numSecu=num;} //constructeur
}

96

Les constantes

97

- Une méthode déclarée final ne peut pas être redéfinie dans une sous classe.
- Lorsque le modificateur final est ajouté à une classe, il est interdit de créer une classe qui en hérite.

```
public  class Point {
priavte int x,y;
//…
public final void affiche() { …}
```

```
public class Pixel extends Point {

public void affiche() {  } // erreur

}
```

```
public  final class Point {
priavte int x,y;
//…
public void affiche() { …}
```

```
public class Pixel extends Point { // erreur

public void affiche() {  }

}
```

97

Les types primitives

98

- Les variables et les constantes utilisées en java représentent des emplacements mémoires dans lesquels peuvent être stockées des données (primitives)
- Une primitive est notamment caractérisée par un "type" représentant sa nature et renseignant le compilateur sur l'espace mémoire occupée ainsi que sur la représentation de la primitive.
- Un type codé sur n bits peux prendre 2^n valeurs.
- Le type peut être un type de base, un objet, …

98

Les types primitifs

99

En java, il est possible de déclarer des données de type :

- Numérique (entiers et décimaux)
- Caractère
- Booléen

A ceux-ci s'ajoutent des structures plus élaborées pouvant être utilisées dans le typage de données:

- Les String (objets)
- Les tableaux
- Les vecteurs
- Les wrappers (objets)
- …

99

Les types primitives

100

Type	Description	Taille en octet (bits)	Plage de valeurs
byte	Entier très court	1 (8)	-128 à 127
short	Entier court	2 (16)	-32 768 à 32 767
int	Entier	4 (32)	-2 147 483 648 à 2 147 483 647
long	Entier long	8 (64)	Environ -2^{18} à 2^{18}
float	Réel (décimal flottant)	4 (32)	$+/-10^{38}$
double	Flottant double	8 (64)	$+/-10^{308}$
char	Caractère	2 (16)	Caractère Unicode (65000 possibilités)
boolean	Booléen	1 (8)	0 ou 1 (false ou true)

100

Les types primitifs

101

Les entiers sont signés par défaut. A la différence de c++, le mot unsigned n'existe pas

void est également un type de base

Le type char utilisait précédemment le codage ASCII (128 caractère maximum) mais implémente maintenant le codage Unicode, ce qui autorise l'utilisation des caractères accentués et facilite la portabilité (notamment).

- Le caractère B peut ainsi être codé simplement 'B' (les ' ' sig code ASCII de …) \t code une tabulation, \n retour à la ligne, \b un retour en arrière, \f saut de page, \' un quotte,…

Un flottant est un nombre à virgule dans lequel la position de cette dernière varie (flotte).

- Une partie des bits servent à coder la mantisse (le nombre sans virgule), une autre l'exposant (repère la position de la virgule) et un bit permet de signer (idem pour les entiers)

101

Les wrappers

102

- Wrapper = enveloppeur
- Un wrapper est un objet pouvant envelopper, encapsuler un type de base. Ce dernier peut alors être manipulé comme un objet.
- Chaque wrapper contient un type primitif, des méthodes pour notamment faire les correspondances type/objet et avec les String, ainsi que des données complémentaires (telles que des constantes MIN_VALUE et MAX_VALUE représentant les valeurs max. et min. possibles).

102

Les wrappers

103

Chaque type de base à sa classe wrapper associée:

- byte➔Byte
- short➔ Short
- int ➔ Integer
- long➔ Long
- float➔ Float
- double➔ Double
- char➔ Character
- boolean➔ Boolean
- void➔ Void

Les classes BigDecimal et BigInteger sont des classes wrapper sans primitives associées permettant de manipuler des nombres plus importants que ceux permis par les types courants.

103

Les wrappers

104

Quelques exemples

- Double chiffreAffaire=new Double(27.43);

Dans ce cas précis, la méthode doubelValue() permet d'accéder à la valeur initiale de l'élément enveloppé :

- double chiffre=chiffreAffaire.doubleValue();

La methode valueOf() permet d'obtenir un wrapper à partir d'une String :

- String s = "39.53";
- Double chiffreAffaire = Double.valueOf(s);

De la même manière, il est possible d'avoir le type fondamentale à partir d'une String :

- double chiffre = Double.parseDouble("109.22");

Les wrappers permettent de rendre compatible les types fondamentaux avec des structures n'acceptant que des objets

- les vecteurs, par exemple

104

Les wrappers

105

La classe Character possède de nombreuses méthodes static permettant d'analyser un char, de connaître sa "nature" (chiffre ou lettre, espace, minuscule ou majuscule) et de récupérer son codage Unicode :

- static int getNumericValue(char) : retourne le code Unicode
- static boolean isLetter(char) : true si le char est une lettre
- static boolean isDigit(char) : true si le char est un chiffre
- static boolean isLetterOrDigit(char) : true si lettre ou chiffre
- static boolean isWhitespace(char) : true si le char est un espace
- static boolean isUpperCase(char) : true si majuscule
- static boolean isLowerCase(char) : true si minuscule
- static int getType(char) : retourne le type de char (selon une série de constantes définies dans la classe Character:
  - 1pour une "UPPERCASE_LETTER", 2 pour une "LOWERCASE_LETTER", 9 pour un DECIMAL_DEGIT_NUMBER", etc.)

105

Le cast

106

Le cast, ou transtypage, est un mécanisme permettant de convertir le type d'une donnée en un autre (par exemple un entier en float ou vice-versa)

Si le passage int/float est simple, il est possible d'effectuer des conversions plus spécifiques (type/objet).

Le procédé est simple et couramment employé, notamment dans le cas de récupérations de valeurs dans une base de données, un vecteur, ou lorsque l'on veut faire correspondre le résultat d'une méthode au type de retour qu'elle implémente (une méthode donne un résultat entier, or elle doit renvoyer un float : on convertit donc le résultat en float).

106

Le cast

107

- Deux types de cast:
- **Implicite :** le compilateur peut se trouver en présence d'affectation d'une valeur d'un type qui ne correspond pas à celui de la variable (ex: le programme tente de placer un float dans une variable de type int). Si les types sont compatibles, aucune erreur ne sera signalée et le compilateur va convertir la donnée pour qu'elle corresponde au type de la variable à laquelle elle doit être affectée. Ceci peut être source de dysfonctionnements et est totalement invisible pour le développeur ou utilisateur de l'application. Il devient très dur de localiser la source du dysfonctionnement.
- Il est possible de coder explicitement un cast.
  - Ex: int x;
    - x=(int) 5.2344; // x reçoit la valeur 5
    - x= int (5.2344); // autre écriture, résultat identique

107

String

108

En c++, les String sont des tableaux de caractères

En Java, ce sont des objets, avec leurs attributs et leurs méthodes. Ainsi,

- String s = "Bonjour";
  - Est équivaut à
- String s = new String("bonjour"); /* appel du constructeur de la classe String*/

Un objet autre peut être converti en String via la méthode toString()

108

Chapitre VII

109

LES TABLEAUX ET LES COLLECTIONS

109

Les tableaux

110

Un tableau est une structure "stockant" une succession d'éléments de même type.

Un tableau est lui-même de type "tableau de..."

- Tableau de int
- Tableau de float
- Tableau de String, …

Les tableaux étant typés, il est donc possible de réaliser des "tableau de tableaux de ...", c.-à-d. des tableaux contenant des tableaux d'un élément donné voir plus. On simule ainsi des tableaux multidimensionnels (de formes variables).

- Ex: trois tableaux contiennent chacun dix valeurs entières. Ces trois structures peuvent être elle-même disposées dans un tableau.

À tout moment, il est possible de connaître la taille d'un tableau en utilisant la syntaxe monTableau.length (à affecter à un entier).

110

Les tableaux

111

La taille d'un tableau est fixée dès sa création. C'est donc une structure relativement rigide.

- Ex: int[ ] monTableau = new int[10];

La création repose sur la même syntaxe que pour les objets car il y a allocation de mémoire: un type (ici, tableau d'entier), un nom, =, new, le type avec les paramètres (ici, la taille: 10).

La taille est ici fixée à la compilation. C'est ce type de fonctionnement qui suivent la plupart des langages. Java permet de s'affranchir en partie de cette contrainte et de fixer la taille au moment de l'exécution:

- Imaginons que nous avons un entier n recevant une valeur différente à chaque exécution de l'application (paramètre passé par l'utilisateur en ligne de commande, valeur recueillie dans une base de données, etc.).
- Il est alors possible de créer un tableau dont la taille varie pour chaque exécution :
- int[ ] monTableau = new int [n+3];
- Le compilateur accepte puisque la taille est fixée (c'est n+3).

111

Les tableaux

112

Comme pour un objet, il faut déclarer un tableau, puis le définir (création, initialisation). new permet d'allouer de la mémoire :

- "int[ ] monTableau;" puis "monTableau=new int[10];"
- Ou bien "int[ ] monTableau = new int[10];"

Un tableau est une variable, il est donc possible d'en déclarer plusieurs en même temps:

- int[ ] monTableau, monAutreTableau;
- "int[ ] monTableau" et "int monTabelau[ ]" comportent néanmoins des différences.
- Si l'opérateurs [ ] est situé juste après le type (int,…), il agira sur toutes les variables de la déclaration:
- int[ ] a,b; // a et b sont deux tableaux d'entiers.
- Si l'opérateur [ ] est situé juste après le nom, il n'agira que sur la variable impliquée :
- int a, b[ ]; // a est un entier et b un tableau d'entiers

Il reste préférable de se limiter à des écritures simples

112

Tableaux des tableaux

113

int[ ][ ] monTabelau = new int[10][4];

- monTableau est un tableau contenant 10 tableaux de 4 entiers.

"int[ ] a, b[ ];" et int a[ ], b[ ][ ];" sont deux écritures équivalentes (a est un tableau d'entiers et b un tableau de tableau d'entiers).

Accès au contenu d'un tableau:

- "monTableau[0]" représente le premier élément du tableau monTableau. Si ce dernier contient des int :
- "int a =monTableau[0];" affecte a l'entier a la valeur monTableau[0]
- De même, "monTableau[1];" représente le second élément, etc.

"monTableau[0][1];" ici, nous sommes en présence d'un tableau de tableau et nous accédons au second élément (noté 1) du premier tableau (noté 0) stocké dans monTableau.

113

Les tableaux : initialisation

114

Plusieurs manières pour "remplir" un tableau sont utilisables:

- Au moment de la création
  - double[ ] monTableau = {4.3, 4.5, 6.7, 3.5};
  - Déclaration, définition (création) et initialisation sont réalisées en une étape. La dimension du tableau est fixée à cet instant.
  - int[ ][ ] tab={{1,7},{23,22},{44,55}};
  - Cas d'un tableau de trois tableaux de deux entiers
- monTableau[2]=22.0; // monTableau supposé contenir des //float ou double
- tab[1][4]=7; // 7 affecter à la 4e position du second tableau //d'entiers contenu dans le tableau tab.

114

Les tableaux : initialisation

115

- Via des boucles

```
1.  for(int i=0; i<4;i++){
2.                      monTaleau[i]=i*2;
3.  }
4.  /* Ici, les valeurs 0,2,4 et 6 vont être placées dans
    les positions d'indices 0 à 3 du tableau. A chaque
    itération de la boucle, une valeur i*2 est affectée à
    monTableau[i]*/

5.  for(int i=0; i<3; i++){
6.          for(int j=0; j<4; j++)
7.                  tab[i][j]=(i*j);
8.   }
9.  }
10. // Cas d'un tableau de 3 tableaux et 4 entiers
```

115

Copies des tableaux

116

La taille d'un tableau est fixe. Il peut alors être nécessaire de construire un tableau plus grand afin d'augmenter le nombre des données pouvant être stockés. Dans ce second tableau, il suffit de recopier les valeurs contenues dans le premier tableau devenu trop petit.

```
int n= x.length;  //récupération de la taille du tableau à copier
  int[ ] y= new int[n*1.5];  /* nouveau tableau de taille
                                identique au premier ou plus
                                grand si nécessaire (ici, de 50%)*/
  for (int i=0; i< n;i++) {  // pour chaque indice entre 0 et…
    y[i]=x[i];     //…la taille du tableau x, on…
  }                  //copie la valeur entre les tableaux
```

Autre possibilité: utiliser la méthode statique arraycopy de la classe java.lang.system en lui précisant le nom de l'ancien tableau, l'index à partir duquel on le parcourt, le nom du nouveau tableau, l'index à partir duquel on écrit dedans et enfin le nombre d'éléments copiés:

```
Sytem.arraycopy(x,0,y,0,n);
```

116

Les tableaux

117

Une autre manière de construire des tableaux multidimensionnels:

```
int[ ][ ]tab =new int[2][ ]; /*création d'un tableau contenant 2
                              tableaux d'entiers. Il faut maintenant créer les
                              tableaux de ce premier tableau*/
• tab[0]=new int[4]; // création du premier tableau d'entiers (taille 4)
• tab[1]=new int[2]; // création du second tableau d'entiers (taille 2)
/* il ne reste plus qu'à initialiser les différents éléments des tableaux:
tab[0][0]=3; etc.*/
```

Ceci permet de construire des matrices non rectangulaire (ici triangulaire)

```
int DIM=4;
double[ ][ ] tab = new double[DIM][ ];
for(int i=0;i<DIM; i++){
  tab[i] = new double[i+1];
}
for (int i=0; i<DIM ;i++){
  for(int j=0; j<i+1; j++) (// remplissage des tableaux)
}
```

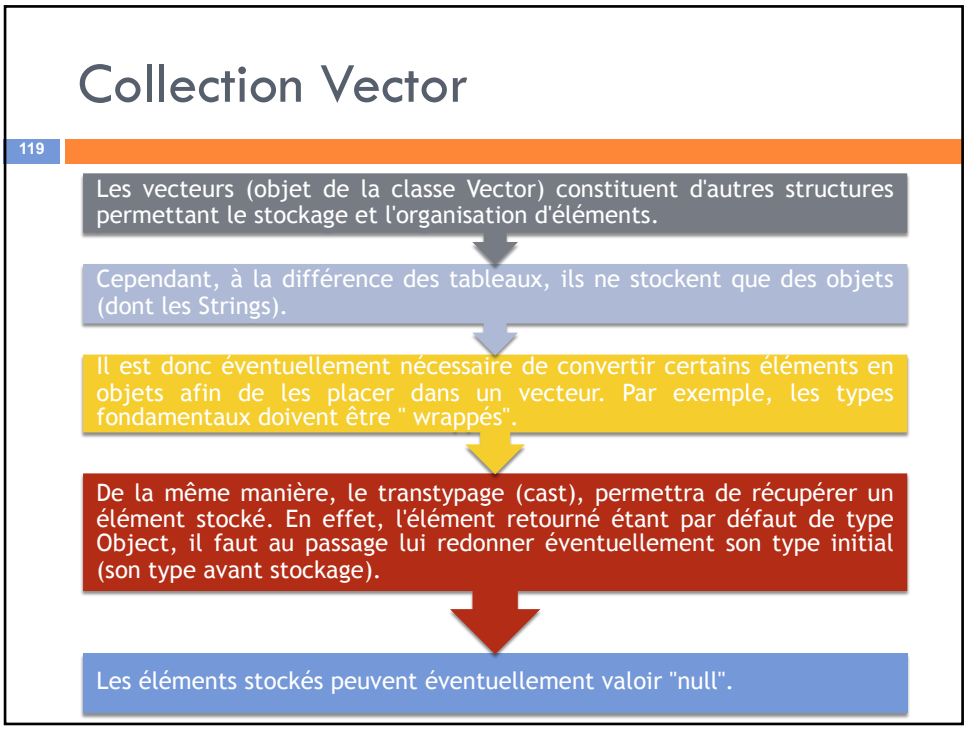
117

Les collections

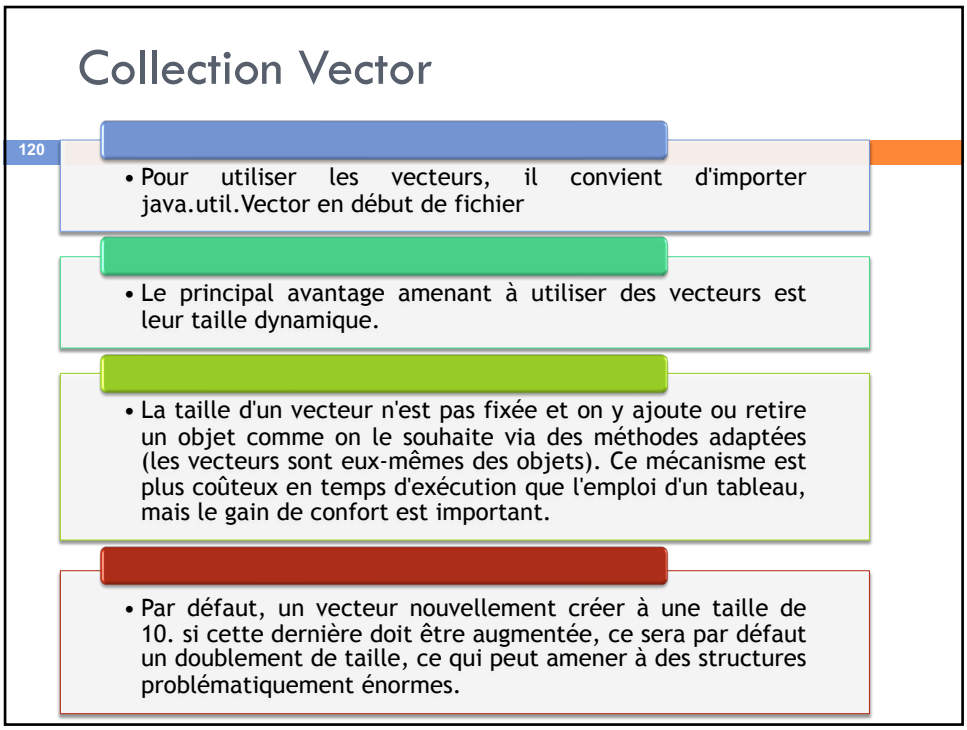
118

- Une collection est un tableau dynamique d'objets de type Object.
- Une collection fournit un ensemble de méthodes qui permettent :
  - D'ajouter un nouvel objet dans le tableau
  - Supprimer un objet du tableau
  - Rechercher des objets selon des critères.
  - Trier le tableau d'objets
  - Contrôler les objets du tableau
  - Etc.
- Les tableaux peuvent être utilisés quand la dimension du tableau est fixe. Dans le cas contraire, il faut utiliser les collections
- Java fournit plusieurs types de collections :
  - ArrayList
  - Vector
  - Iterator
  - HashMap, …
- Dans cette partie du cours, nous allons présenter uniquement comment utiliser les collections Vector et ArrayList.

118


Collection Vector
119
Les vecteurs (objet de la classe Vector) constituent d'autres structures permettant le stockage et l'organisation d'éléments.
Cependant, à la différence des tableaux, ils ne stockent que des objets (dont les Strings).
Il est donc éventuellement nécessaire de convertir certains éléments en objets afin de les placer dans un vecteur. Par exemple, les types fondamentaux doivent être " wrappés".
De la même manière, le transtypage (cast), permettra de récupérer un élément stocké. En effet, l'élément retourné étant par défaut de type Object, il faut au passage lui redonner éventuellement son type initial (son type avant stockage).
Les éléments stockés peuvent éventuellement valoir "null".

119


Collection Vector
120
• Pour utiliser les vecteurs, il convient d'importer java.util.Vector en début de fichier
• Le principal avantage amenant à utiliser des vecteurs est leur taille dynamique.
• La taille d'un vecteur n'est pas fixée et on y ajoute ou retire un objet comme on le souhaite via des méthodes adaptées (les vecteurs sont eux-mêmes des objets). Ce mécanisme est plus coûteux en temps d'exécution que l'emploi d'un tableau, mais le gain de confort est important.
• Par défaut, un vecteur nouvellement créer à une taille de 10. si cette dernière doit être augmentée, ce sera par défaut un doublement de taille, ce qui peut amener à des structures problématiquement énormes.

120

Collection Vector

121

Quelques méthodes applicables aux vecteurs (de la même manière que pour n'importe quel objet: monVecteur.methode()):

- **void** addElement(**Object**) : ajoute l'objet à la fin du vecteur.
- **Object** elementAt(**int**): retourne l'élément qui se trouve à l'indice indiqué (int).
- **Enumeration** elements( ) : retourne un java.util.Enumeration énumérant les objets contenus dans le tableau
- **void** setElementAt(**Object**, **int**) : met à l'indice indiqué (int) l'instance appliquée. L'indice doit être au moins égale à 0 et être strictement inférieur à la taille actuelle du vecteur.
- **boolean** isEmpty() : teste si le Vector est vide.
- **void** removeAllElements( ) : vide le vecteur
- **boolean** removeElement(**Object**) : retire la première occurrence de l'argument s'il figure et retourne true, sinon retourne false.
- **void** removeElementAt(**int**) : retire l'objet qui se trouve à l'indice indiqué (int).
- **int** size( ) : retourne le nombre d'objets contenus par le vecteur.

121

Collection Vector

122

Exemples de manipulations:

- String s="Bonjour";
- Vector monVecteur=new Vector( );

par défaut, taille=10 et incrémentation de taille=taille*2

- monVecteur.addElement(s);
- int taille=monVecteur.size( ); // ici, la taille est de 1
- String a =monVecteur.elementAt(0);

retourne l'élément stocké en position 0 (premier)

- Vector monAutreVecteur = new Vector(40);

construction d'un vecteur de taille 40

- Vector encoreUnAutreVecteur = new Vector(26,15);

vecteur de taille 26 et d'augmentation de taille 15

- Voiture auto_1 = (Voiture) vectVoiture.elementAt(3);

récupération d'un objet de type Voiture dans un vecteur vectVoiture.

122

Collection ArrayList

123

- ArrayList est une classe du package java.util, qui implémente l'interface List.
- Déclaration d'une collection de type List qui devrait stocker des objets de type Fruit :
  - `List <Fruit> fruits;`
- Création de la liste :
  - `fruits = new ArrayList<Fruit>();`
- Ajouter des objets de type Fruit à la liste
  - `fruits.add(new Pomme(30));`
  - `fruits.add(new Orange(40));`

123

Collection ArrayList

124

- Faire appel à la méthode affiche() de tous les objets de la liste:
  - En utilisant la boucle classique for

```
for(int i=0; i<fruits.size();i++){
        fruits.get(i).affiche();
}
```

  - En utilisant la boucle for each

```
for(Fruit f:fruits){
        f.affiche();
}
```

- Supprimer le deuxième Objet de la liste

```
fruits.remove(1);
```

124

Question

125

- En quoi le code ci-dessous est-il incorrect?

```
int [ ] tab; // tableau d'entiers
for (int i=0; i < 10 ; i++) {
    tab[i]=0;
}
```

int [ ] tab = new int[10];

125

Question

126

- **Lors de l'extraction d'un objet à partir d'un vecteur, un cast est nécessaire :**

A. Pour pouvoir appeler les méthodes définies dans la classe la plus spécialisée de l'objet
B. Pour pouvoir stoker des éléments de types primitifs dans le vecteur
C. Car tous les objets doivent pouvoir recevoir les messages pour appeler les méthodes de toutes les classes
D. Pour pouvoir appeler des méthodes de la classe Object à partir de l'objet existant

126

Question

127

- **Méthodes d'instances et - méthodes de classes permettent d'utiliser respectivement :**

A. Variables d'instances ou de classe, et - Variables d'instances ou de classe.
B. Uniquement variables d'instances, et - Variables d'instances ou de classe.
C. Variables d'instances ou de classe, et - Uniquement variables de classe.
D. Uniquement variables d'instances, et - Uniquement variables de classe.

127