

## Contenu du module

- Environnement de développement
- Introduction à Java (classes et objets)
- Héritage et polymorphisme
- Documentation en Java: Javadoc
- Classes abstraites, internes et interfaces
- Collections



2



Karima Boudaoud

IUT GTR - Sophia Antipolis

## Bibliographie (1)

- Ce cours a été préparé principalement à partir du support de cours de Peter Sander, que je tiens à remercier très particulièrement pour son aide et ses précieux conseils.

Autres supports d'où a été préparé ce cours:

- Les langages orientés objets, La programmation Java, cours de Frédéric Drouhin, IUT GTR Colmar
- Programmation orientée objet, cours de B. Botella, IUT GTR Chalon en Champagne
- Une introduction au langage Java, cours de Renaud Zigman, Xalto
- Java cours de Eric Lemaitre , CS Institut
- Supports de cours de Richard Grin, UNSA
- Introduction à Java cours de Blaise Madeline
- Une introduction au langage Java, documentation de Jbuilder
- IO Framework et Java distribué, cours de Stéphane Frénot, INSA de Lyon
- Programmation réseau en Java, cours de Patrick Itey, INRIA



Karima Boudaoud

IUT GTR - Sophia Antipolis

3

## Bibliographie (2)

### m Pages Web

- m [www.java.sun.com](http://www.java.sun.com)
- m Api
- m Books
- m Tutorial

### m Livres

- m Les livres sont toujours (pratiquement) une bonne façon de se documenter
- m Préférer les livres d'initiation aux livres de référence plutôt destinés aux « programmeurs confirmés »

### m Livres de référence

- m Java in a Nutshell (existe aussi en français)

### m Livres d'initiation

- m De manière générale tous les livres d'initiation



Karima Boudaoud

IUT GTR - Sophia Antipolis

4



## Environnement de développement

### Environnement de développement (1)

#### • Editions de Java

- ü il existe trois éditions de Java qui sont la **J2SE**, la **J2EE** et la **J2ME**
- ü la **J2SE** (Java 2 Standard Edition) représente l'édition de base de Java
- ü la **J2EE** (Java 2 Enterprise Edition) qui propose des API supplémentaires par rapport à l'édition J2SE pour pouvoir écrire des applications distribuées. On y retrouve les notions d'EJBs, JSP et servlet
- ü la **J2ME** (Java 2 Micro Edition) qui a été défini pour faciliter l'écriture de programmes embarqués sur des téléphones portables, des PDA, des cartes à puce, etc... C'est une version allégée de la J2SE.



Richard Grin Univ.  
Nice Sophia Antipolis

Karima Boudaoud IUT GTR -  
Sophia Antipolis

6

### Environnement de développement (2)

#### • Les principaux outils de base de la JDK

- ü les principaux outils de java sont : **javac**, **java**, **jdb**, **javap**, **javadoc**
- ü **javac**, représente le compilateur de java, c'est ce qui permet de compiler un programme écrit en langage java
- ü **java** est l'interpréteur de java, c'est la commande à utiliser pour exécuter un programme java
- ü **jdb** est le débogueur de java, c'est un outil utilisé pour déboguer un programme écrit en java
- ü **javap** permet de déassembler un fichier compilé
- ü **javadoc** est un générateur de documentation. Il permet de générer de la documentation sur les programmes écrits en java



Eric Lemaître CS  
Institut Copyright

Karima Boudaoud IUT GTR -  
Sophia Antipolis

7

### Environnement de développement (3)

#### • Le compilateur

- ü le compilateur **javac** permet de compiler un programme java (i.e un code source) en bytecodes java.
- ü la commande à utiliser pour compiler un programme est **javac [options] ClassName.java**
- ü à l'issue de cette commande, le compilateur **javac** génère un fichier **ClassName.class** afin qu'il soit ensuite interprété par la JVM (Java Virtual Machine)

#### • L'interpréteur

- ü l'interpréteur **java** permet d'exécuter une application écrite en langage java (autre qu'une applet), plus spécifiquement un fichier **ClassName.class** (i.e le java bytecodes).
- ü par le biais de l'interpréteur java, on peut faire passer des arguments à la fonction main
- ü la commande à utiliser pour exécuter un programme est **java [options] Classname <args>**



Eric Lemaître CS  
Institut Copyright

Karima Boudaoud IUT GTR -  
Sophia Antipolis

8

## Environnement de développement (4)

### • Le débogueur

- ü le débogueur **jdb** permet de déboguer "en ligne" un programme (une classe)
- ü il n'est pas facile à utiliser
- ü pour pouvoir déboguer un programme (i.e une classe) il faut compiler la classe avec l'option -g
- ü la commande à exécuter pour l'utiliser est **jdb Classname**
- ü Il existe une aide pour le débogueur. Pour accéder à cette aide, il faut taper **help** ou ?

### • Le générateur

- ü le générateur de documentation **javadoc** permet de générer des documents HTML équivalents aux documents Java de SUN (i.e ayant la même structure)
- ü la commande à exécuter pour l'utiliser est **javadoc Classname**



Eric Lemaître CS  
Institut Copyright

Karima Boudaoud IUT GTR -  
Sophia Antipolis

9



## Environnement de développement (4)

### • La JDK

- ü on peut télécharger la JDK à partir du site de Sun : [java.sun.com](http://java.sun.com)

### • Autres outils de développement

il existe plusieurs outils de développement dont la plupart sont payants. Néanmoins voici quelques uns

#### ü Eclipse

Windows/Unix, gratuit, site : <http://www.eclipse.org>

#### ü JBuilder

Windows, payant mais il existe une version d'essai de 30 jours

#### ü Java Workshop

Windows/Unix, gratuit

#### ü Visual ++

Windows, payant



Karima Boudaoud

IUT GTR - Sophia Antipolis

10



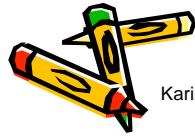
## Environnement de développement (5)

### • Variables d'environnement

- ü pour pouvoir utiliser la JDK, il faut paramétrier le **PATH** pour l'outil Java et le **CLASSPATH** pour les classes de l'utilisateur
- ü **PATH**
  - § il faut spécifier dans le PATH, le répertoire qui contient tous les outils de base de Java c'est-à-dire javac, java,..
- ü **CLASSPATH**
  - § il faut spécifier dans le CLASSPATH, le chemin d'accès aux classes de l'utilisateur
  - § pour Windows, il faut faire

```
set CLASSPATH=.;C:\Jdk\classes.zip;C:\Karima\mesClasses
```
  - § pour Unix, il faut faire

```
CLASSPATH=.:~/home/Karima /mesClasses:/products/java/lib/classes.zip
```



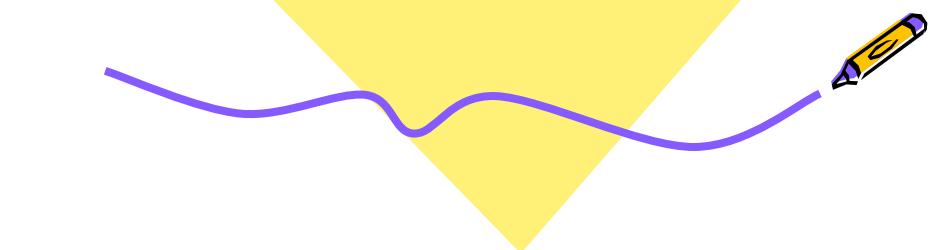
Karima Boudaoud

IUT GTR - Sophia Antipolis

11



Introduction à la  
programmation orientée  
objet



# Programmation Orientée Objet

- Intro
  - Abstraction et encapsulation
  - Objet
    - instance de classe
    - utilisation
  - Classe
    - type d'objet
    - définition



15/09/99

© 1999 Peter Sander

13



# Programmation Orientée Objet

- **Idée clé**
  - Abstraction
    - pour modéliser (représenter) la réalité



15/09/99

© 1999 Peter Sander

14

## Programmation Orientée Objet Langages et Abstraction (1)



Langages plus proches de la machine

Langages plus proches de la réalité



15/09/99

© 1999 Peter Sander

15



## Programmation Orientée Objet Langages et Abstraction (2)

- La programmation structurée
  - C, Pascal, Fortran, ...
  - Programme : procédures – fonctions
  - Que doit faire mon programme ?
    - structure !
- La programmation orienté objet :
  - Java, C++, Eiffel
  - Sur quoi porte mon programme ?
    - les entités manipulées

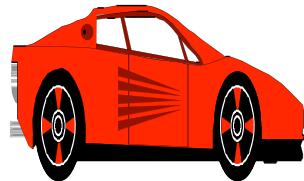


Frederic Drouhin, IUT GTR  
Colmar

16

## Programmation Orientée Objet Abstraction

- Abstraction
  - Représentation d'une réalité en code



abstraction



15/09/99

© 1999 Peter Sander

17



## Programmation Orientée Objet Type Abstrait

...et objets spécifiques

- mon vélo rouge
- au Canada
- Emacs

- Idée de type d'objet
  - une bicyclette
  - des vacances
  - un éditeur de texte
  - ...
- Distinction importante
  - Type d'objet
    - on parle de « classe d'objet »
  - Objet spécifique
    - on parle d' « instance de classe »



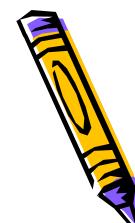
15/09/99

© 1999 Peter Sander

18

## Programmation Orientée Objet Objet

- Caractéristiques de la POO
  - Il n'y a que des objets
    - tout est objet (à part les primitifs en Java !)
  - Un programme est composé d'objets
  - Ces objets communiquent entre eux
    - en s'envoyant des messages
  - Chaque objet a son propre espace mémoire
    - qui peut contenir d'autres objets
  - Chaque objet a un type
    - un objet est une instance d'un type (on dit aussi « classe »)
  - Un message peut s'adresser à n'importe quel objet d'un type donné



19

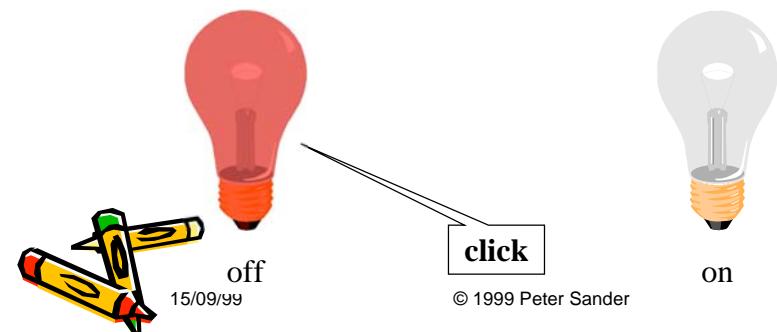


15/09/99

© 1999 Peter Sander

## Programmation Orientée Objet Type Abstrait

- Type abstrait possède
  - État interne
  - Actions
- En réalité
  - Pour changer l'état, il faut appliquer une action



© 1999 Peter Sander

20

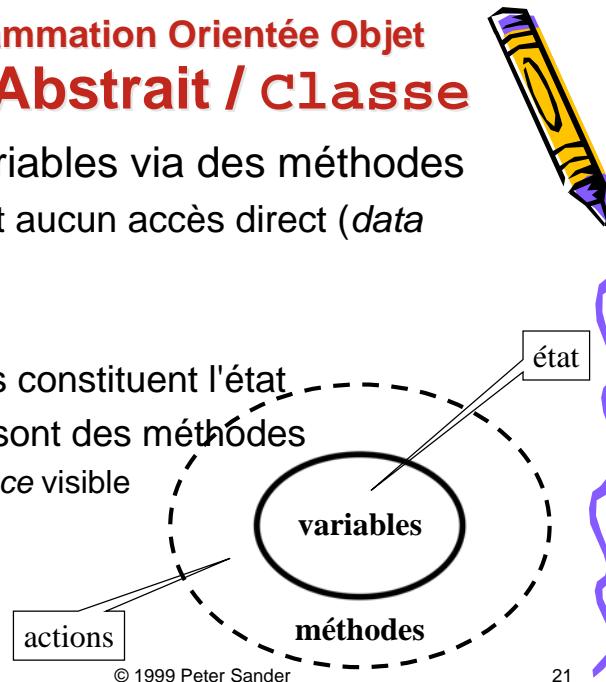


## Programmation Orientée Objet Type Abstrait / Classe

- Accès aux variables via des méthodes
  - Normalement aucun accès direct (*data hiding*)
- En Java
  - Des variables constituent l'état
  - Les actions sont des méthodes
    - c'est *l'interface visible*



15/09/99



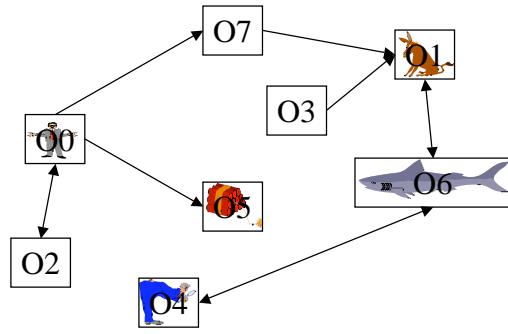
21

## Programmation Orientée Objet Encapsulation

- Un programme
  - Un ensemble d'objets qui communiquent entre eux et avec l'extérieur



15/09/99

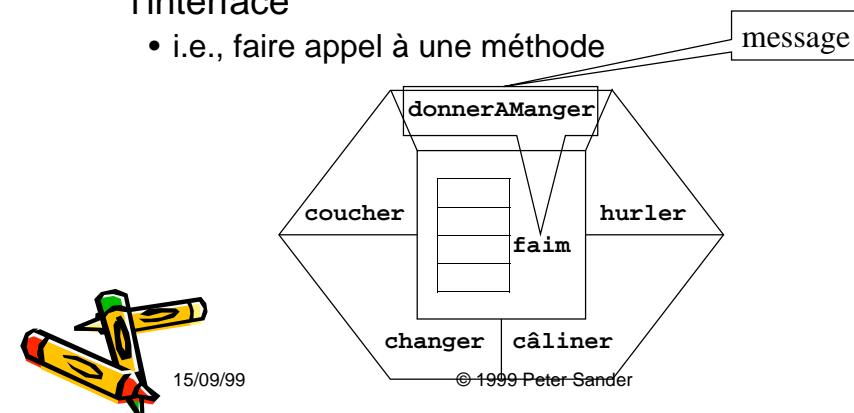


© 1999 Peter Sander

23

## Programmation Orientée Objet Encapsulation

- En abstrait, classe Bébé encapsule état et actions
  - Pour changer l'état, il faut passer par l'interface
    - i.e., faire appel à une méthode



22

## Classes L'Encapsulation

- Une classe comprend des *membres*
  - ses variables d'instance
    - son état interne
  - ses méthodes
    - des services qu'on peut lui demander
- L'encapsulation isole les variables
  - Oblige à y accéder par un service

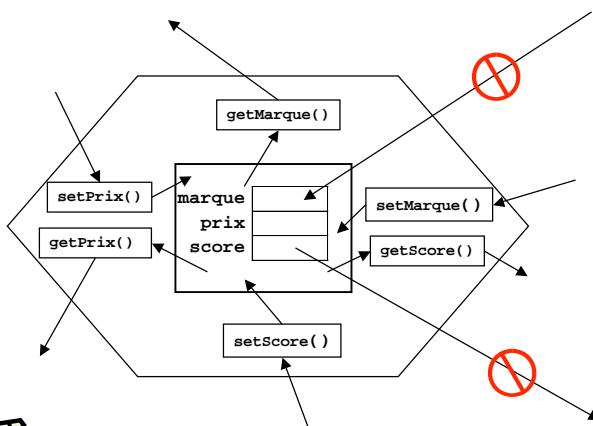


15/09/99

© 1999 Peter Sander

24

# Classes L'Encapsulation



15/09/99

© 1999 Peter Sander

25

# Programmation Orientée Objet Réutilisabilité

- Par composition
  - Réutilisation d'un objet existant
  - J'utilise ma voiture...
    - ...une banale Mercedes-Benz SL500
- Par héritage
  - Définition d'un nouvelle classe d'objet basée sur une classe existant
  - La société AMG propose son modèle SL55...
    - ...une Mercedes-Benz SL500 légèrement modifiée



15/09/99

© 1999 Peter Sander

27

# Classes L'Encapsulation

- Partie visible de l'objet est son *interface*
  - Ses méthodes publiques
    - public String getMarque() { ... }
    - public void setMarque(String marque) { ... }
    - public double getPrix() { ... }
    - public void setPrix(double prix) { ... }
    - public double getScore() { ... }
    - public void setScore(double score) { ... }
- Mais il y aussi... **interface**



15/09/99

© 1999 Peter Sander

26

# Programmation Orientée Objet Classe et objet

- Une classe...
  - est une usine à fabriquer des objets
  - spécifie les méthodes des objets de ce type
- Un objet...
  - est une instance d'une classe



© 1999 Peter Sander

28

# Programmation Orientée Objet

## Classe et objet

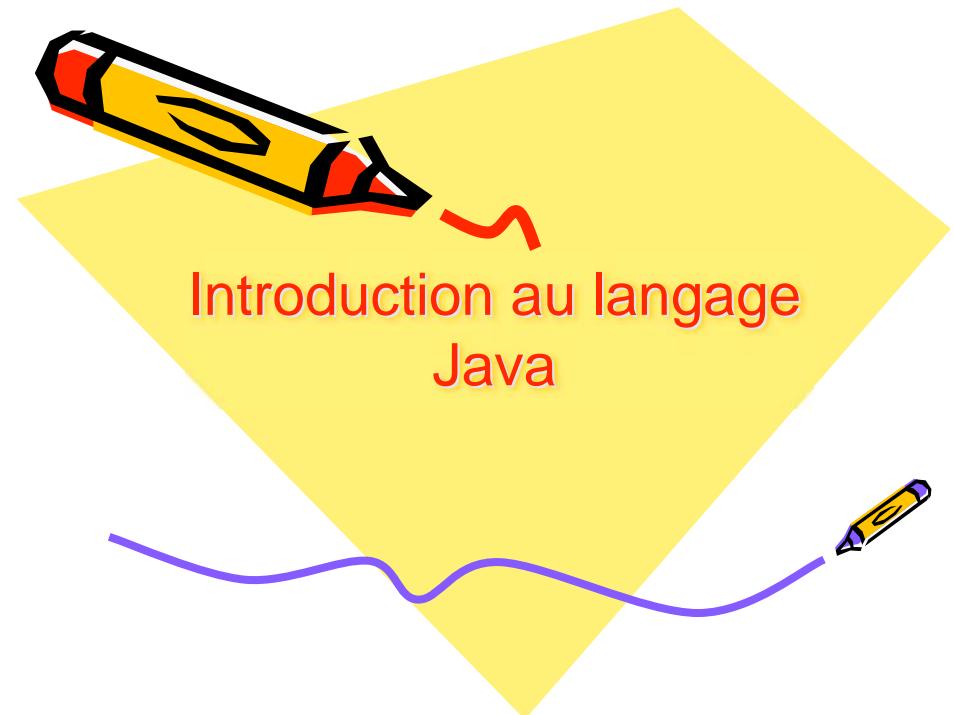
- Classes
  - Des objets vus de l'intérieur
    - Point de vue du développeur
- Objets
  - Des classes vues de l'extérieur
    - Point de vue de l'utilisateur



15/09/99

© 1999 Peter Sander

29



## Hiérarchie (1)

Une ligne de code Java...

Project Hacks

```
package fr.essi.sander.hacks;
public class HelloWorld {
    ...
    public static void main(String[] args) {
        ...
        System.out.println("Ah que hello !");
        ...
    }
}
```



15/09/99

© 1999 Peter Sander

31



## Hiérarchie (2)

...est contenue dans une *méthode*...

Project Hacks

```
package fr.essi.sander.hacks;
public class HelloWorld {
    ...
    public static void main(String[] args) {
        ...
        System.out.println("Ah que hello !");
        ...
    }
}
```



© 1999 Peter Sander

32



## Hiérarchie (4)

...qui est contenue dans une *classe*...

Project Hacks

```
package fr.essi.sander.hacks;
public class HelloWorld {
    ...
    public static void main(String[] args) {
        ...
        System.out.println("Ah que hello !");
        ...
    }
}
```

15/09/99

© 1999 Peter Sander

33

## Hiérarchie (5)

...qui appartient à un *package*...

## Hiérarchie (5)

...qui appartient à un *package*...

Project Hacks

```
package fr.essi.sander.hacks;
public class HelloWorld {
    ...
    public static void main(String[] args) {
        ...
        System.out.println("Ah que hello !");
        ...
    }
}
```

15/09/02

© 1999-2002 Peter Sander

34

Project Hacks

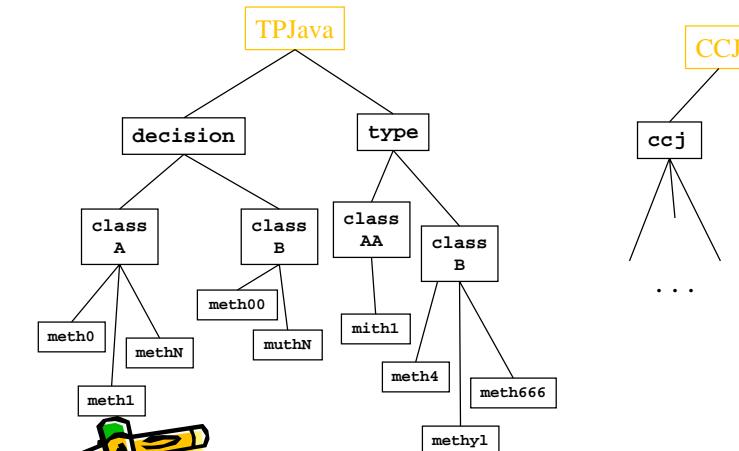
```
package fr.essi.sander.hacks;
public class HelloWorld {
    ...
    public static void main(String[] args) {
        ...
        System.out.println("Ah que hello !");
        ...
    }
}
```

15/09/02

© 1999-2002 Peter Sander

35

## Hiérarchie (7)



15/09/99

© 1999 Peter Sander

36

## Hiérarchie (8)

- **Projet**
  - ü Organise plusieurs package autour d'un thème
- **Package**
  - ü Organise plusieurs classes autour d'une fonctionnalité
    - `java.lang`
    - `java.rmi`
    - `javax.swing`
    - `fr.essi.sander.net.clientserver`
  - ü Nom en minuscules



8/09/00

© 1999-2000 Peter Sander

37



## Hiérarchie (9)

- **Classe**
  - ü Type abstrait
  - ü Contient
    - variables d'instance (données)
      - (aussi variables de classe)
    - méthodes (actions)
  - ü Sert de gabarit pour instantiation (création) d'objets
    - les variables décrivent l'état de l'objet
    - les méthodes présentent l'interface publique de l'objet



15/09/99

© 1999 Peter Sander

38



## Hiérarchie (10)

- **Classe**
  - ü Nom court commence par majuscule
    - `System`
    - `HelloWorld`
  - ü Nom complet comprend le package
    - `java.lang.System`
    - `fr.essi.sander.hacks.HelloWorld`
- **Méthode**
  - ü Nom commence par minuscule
    - `main`



15/09/02

© 1999-2002 Peter Sander

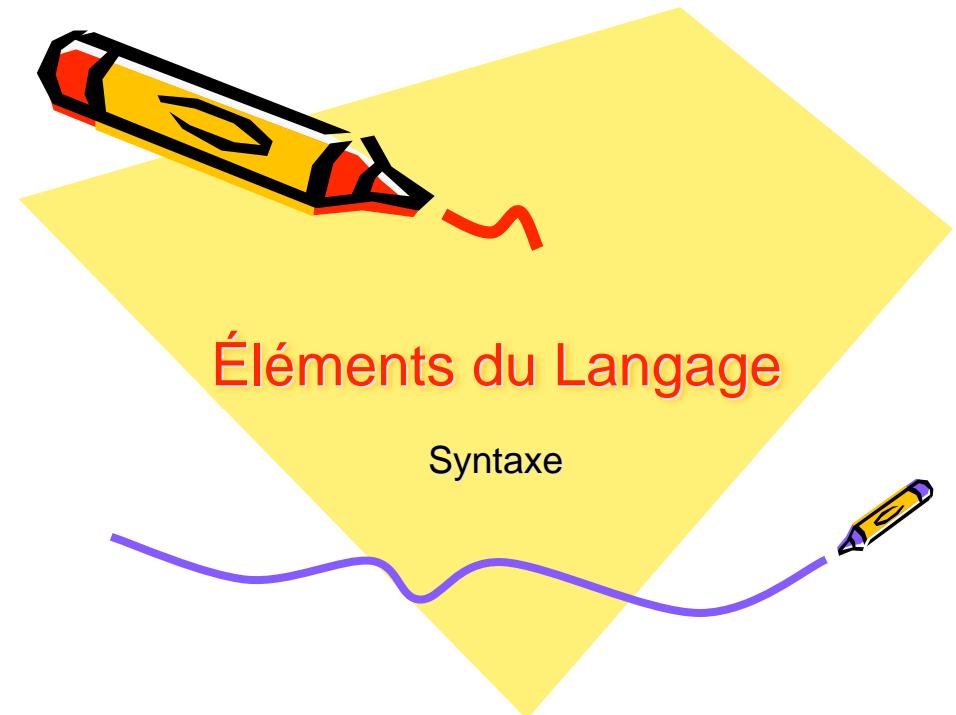
39



39

## Éléments du Langage

Syntaxe



# Éléments du Langage

- **Typage**

- ü Primitif
- ü Référencé
- ü Collections

- **Décisions**

- **Itérations**

- **Méthodes**



15/09/99

© 1999 Peter Sander



41

## Types Primitifs Commentaires

ü bloc

```
/* le code qui suit fait des choses tellement
intéressantes qu'il faut plusieurs lignes rien
que pour le décrire */
```

```
int france = 3;
int brazil = 0;
```

ü fin de ligne

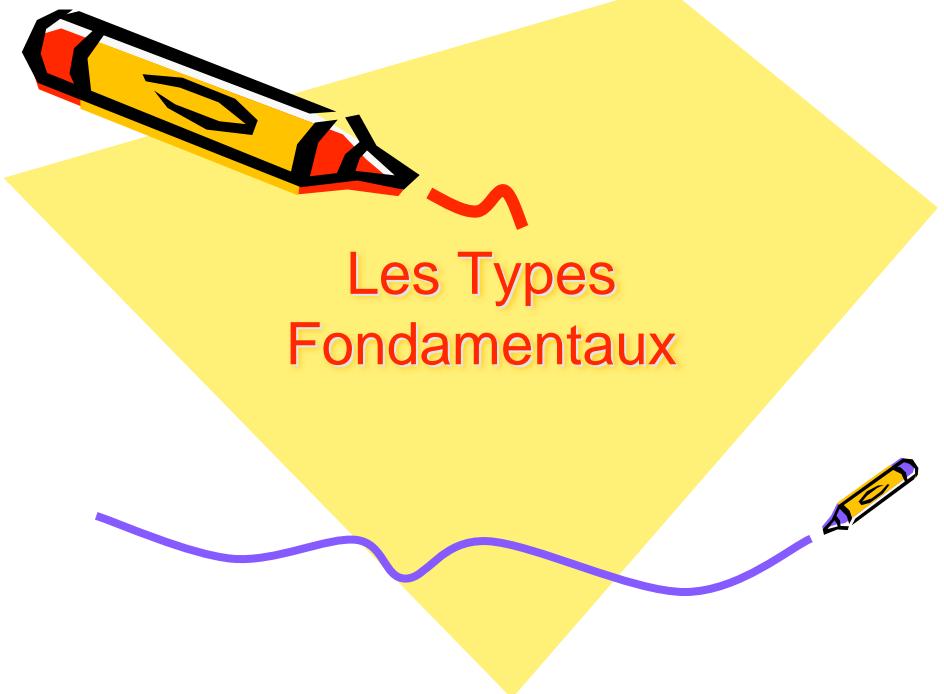
```
int sénégal = 1; // pas de commentaire
int france = 0; // non plus
```



15/09/02

© 1999-2002 Peter Sander

43



## Les Types Fondamentaux

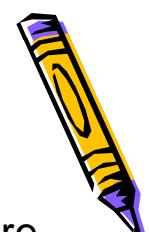
### Variables (1)

- Une variable, c'est une case mémoire
  - Il faut
    - un nom
    - un type
- Java est un langage fortement typé



14/09/01

44



## Types Primitifs Variables (2)

- Nom de variable
  - Constitué de
    - lettres : **a z E R t Y...**
    - nombres : **3 1 4 5...**
    - certains autres caractères : **\_**
  - Commence par lettre (minuscule par convention)



15/09/02

© 1999-2002 Peter Sander

45



## Types Primitifs Variables (4)

- Il faut initialiser les variables...

```
int qFilles = 120;  
int qGarcons; // alors, c'est combien ?  
double qMoyen = (nombF * qFilles + nombG *  
    qGarcons) / (nombF + nombG)
```

- ...sinon (en Java)
  - initialisation par défaut (variables d'instance)
  - erreur de compilation (variables locales)
- ...sinon (en bien d'autres langages)
  - une valeur aléatoire



15/09/99

© 1999 Peter Sander

47



## Types Primitifs Variables (3)

```
int ru496, point3, maVariableAMoi; // ok  
int 123hop, pour%cent; // non !  
int moiMême; // prudence  
int mes_notes, Pi; // oui, mais non-  
    standard  
int mesNotes; // mieux, usage standard
```

- Min- et majuscules sont différents



```
int moiMeme, moimeme, Moimeme;
```



15/09/02

© 1999-2002 Peter Sander

46



## Types Fondamentaux

- Taxonomie

Types « référencés »

- tableaux
- classes

**String**

Types « primaires »

- caractères
- numériques
- int**
- double**
- logique
- boolean**



types fondamentaux

© 1999-2001 Peter Sander

48



## Types Fondamentaux Types Primitifs

- Numérique
  - int
  - double



15/09/99

© 1999 Peter Sander

49

## Types Primitifs Numérique (2)

```
public class RecetteRU {  
    int portionsFrites = 120;  
    int nombreSteackes = 145;  
    ...  
}
```



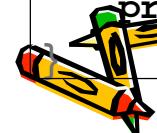
15/09/99

© 1999 Peter Sander

51

## Types Primitifs Numérique (1)

```
public class RecetteRU {  
    int portionsFrites = 120;  
    int nombreSteackes = 145;  
    double prixFrites = 10.50;  
    double prixSteacke = 7.25;  
  
    double totale = (portionsFrites *  
        prixFrites) + (nombreSteackes *  
        prixSteacke);
```



15/09/99

© 1999 Peter Sander

50

## Types Primitifs Numérique (3)

```
public class RecetteRU {  
    int portionsFrites = 120;  
    int nombreSteackes = 145;
```

déclaration de type



15/09/99

© 1999 Peter Sander

52

## Types Primitifs Numérique (4)

```
public class RecetteRU {  
    int portionsFrites = 120;  
    int nombreSteackes = 145;
```

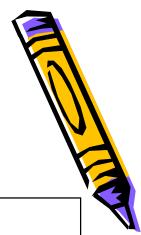
nom de variable



15/09/99

© 1999 Peter Sander

53



## Types Primitifs Numérique (6)

```
public class RecetteRU {  
    int portionsFrites = 120;  
    int nombreSteackes = 145;
```

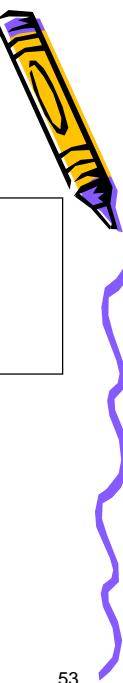
fin d'expression



15/09/99

© 1999 Peter Sander

55



## Types Primitifs Numérique (5)

```
public class RecetteRU {  
    int portionsFrites = 120;  
    int nombreSteackes = 145;
```

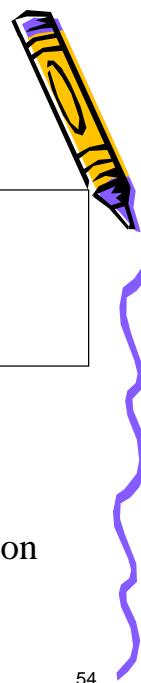
initialisation



15/09/99

© 1999 Peter Sander

54



## Types Primitifs Numérique (7)

```
int pieces10centimes = 5;  
int pieces20centimes = 12;  
  
double valeurJaunes = pieces10centimes * 0.1  
+ pieces20centimes * 0.2;
```

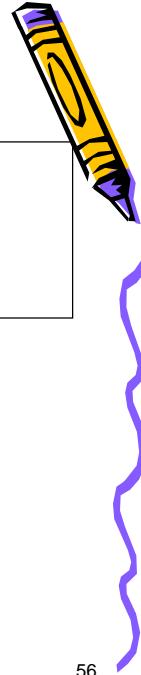
- Les entiers **int**
  - Des entités indivisibles
    - nombres cardinaux ou ordinaux
- Les flottants **double**
  - Des entités d'échelle continue
    - nombres réels



14/09/01

© 1999-2001 Peter Sander

56



## Types Primitifs Numérique (8)

- Tous les types d'entiers

`byte` 8 bit -128 .. 127

`short` 16 bit -32768 .. 32767

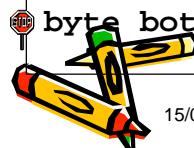
`int` 32 bit -21474483648 .. 21474483647

`long` 64 bit -9223372036854775808 .. 9223372036854775807

- Initialisés à 0 par défaut

- Entiers sont `int` par défaut

```
int i = -42;  
  
int total = i + 17; // 17 est un int  
  
byte botal = i * 10; // faux résultat
```



15/09/99

© 1999 Peter Sander

57



## Types Primitifs Numérique (9)

- Conversion automatique

– permis s'il n'y a pas de perte de précision

```
short s = -39;  
  
int t = s + 356; // ok 16 bits -> 32 bits  
  
s = t - 12; // nonono 32 bits -> 16 bits
```

- Conversion manuelle (cast)

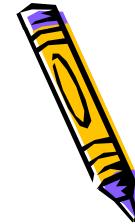
```
int t = 356;  
  
short s = (short) (t - 30); // oui, c'est exprès  
  
t = (int) 365.25; // oui, sous ma responsabilité
```



15/09/99

© 1999 Peter Sander

59



59

## Types Primitifs Numérique (8)

- Tous les types de flottants

`float` 32 bit ±3.40283247E+38 .. ±1.40239846E-45

`double` 64 bit ±1.79769313486231570E+308 ..

±4.94065645841246544E-324

- Initialisés à 0.0

- Flottants sont `double` par défaut

```
double d = 3.14159; // ok
```

```
float f = 3.14159; // nonono !
```

```
float x = 3.14159f; // ok, faut spécifier
```

- `double` est à préférer à `float`



15/09/99

© 1999 Peter Sander

58



## Types Primitifs Logique

- Le type `boolean`

- Valeurs

`true`

`false`

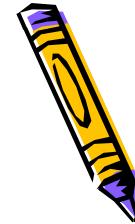
- Traité plus tard (voir Décisions)



15/09/99

© 1999 Peter Sander

60



## Types Primitifs Constantes (1)

```
static final double L_PAR_100KM = 8.2;
static final double PRIX = 5.8;
double km = 437.3;
System.out.println(km / 100.0 * L_PAR_100KM *
    PRIX);
```

- Sont des constantes

L\_PAR\_100KM

PRIX // si seulement c'était vrai...



14/09/01

© 1999-2001 Peter Sander

61



## Types Primitifs Constantes (2)

- Par convention, nom tout en majuscules

```
static final double L_PAR_100KM = 8.2;
static final double PRIX = 5.8;
```

- Mot clé **final** interdit réaffectation de valeur
  - Mot clé **static**, nous le verrons plus tard

```
static final double PRIX = 5.8;
PRIX = 6.2; // nonono
```



© 1999 Peter Sander

62

## Types Primitifs Constantes (3)

- Ne jamais utiliser des nombres magiques (*magic numbers*)

```
if (temp > 41) {    ☺
    panique();
}
```

- Préférez une constante

```
static final int JOURS_PAR_AN = 365;
static final int NB_COLONNES = 4;
static final double TEMP_NORMALE = 37.0;
```



15/09/99

© 1999 Peter Sander

63



## Types Primitifs Arithmétique (1)

- Opérations élémentaires

addition      a + b

soustraction    i - j

multiplication   quant \* prix

division        qi / nbCafes

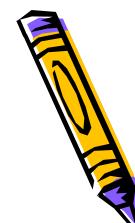
modulo          qi % nbCafes

exponentiation   Math.pow(x, n)



© 1999 Peter Sander

64



## Types Primitifs Arithmétique (2)

- Affectation

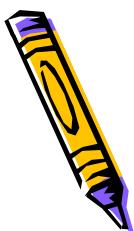
```
t = a + b - c * d / e; // a + b - ((c * d) / e)  
t = a + (b - c) * d / (e - 1);
```



15/09/99

© 1999 Peter Sander

65



## Types Primitifs Arithmétique (3)

- Incrément / décrément
  - l'expression du type...

```
heure = heure + 1;
```

...est si banale qu'il y a le raccourci

```
heure++;
```



15/09/99

© 1999 Peter Sander

66

## Types Primitifs Arithmétique (4)

int i = 0; i++;	
int t = i++;	
int s = ++i; int r = --i;	
r += i;	
s *= i; i /= 2;	

int i = 0; i = i + 1;	
int t = i; i = i + 1;	
i = i + 1; int s = i; i = i - 1; int r = i;	
r = r + i;	
s = s * i; i = i / 2;	

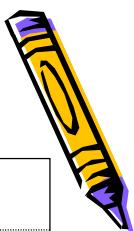
équivalent à



15/09/99

© 1999 Peter Sander

67



## Types Primitifs Arithmétique (5)

- Autres fonctions

Math.sin(x)	sinus x (en radians)
Math.cos(x)	cosinus x (en radians)
Math.tan(x)	tangente x (en radians)
Math.exp(x)	$e^x$
Math.log(x)	$\ln x$ , $x > 0$
Math.ceil(x)	plus petit entier $\geq x$
Math.floor(x)	plus grand entier $\leq x$
Math.abs(x)	valeur absolue $ x $



68

© 1999 Peter Sander



## Types Fondamentaux String (1)

- Chaîne de caractères entourée de ""

```
String nom; // déclaration
nom = "Jacques"; // affectation
String unAutre = "Nicolas"; //décl. & affect.
unAutre = ""; // String vide
```

– Méthode **length()** donne nb. de caractères

```
System.out.println(nom.length());
-> 7
System.out.println(unAutre.length());
-> 0
```

15/09/99

© 1999 Peter Sander

69

## Types Fondamentaux substring()

- **substring(debut, finPlus)** extrait un morceau

```
String salut = "Hello, life!";
String antiSalut = salut.substring(0, 4);
System.out.println("Life is " + antiSalut);
-> Life is Hell
```

- **debut** est le premier caractère à prendre
- ⚠ – **finPlus** est la première caractère à ne pas prendre

15/09/99

© 1999 Peter Sander

71

## Types Fondamentaux String (2)

- Entrée au terminal (`ccj.Console` importée)

```
String nom = Console.in.readWord(); //un seul
                                     mot
<- Fox Mulder
System.out.println(nom);
-> FOX
Console.in.readWord(); // deuxième mot
System.out.println(nom);
-> Mulder
nom = Console.in.readLine(); //une ligne entière
<- Dana Scully
System.out.println(nom);
-> Dana Scully
```

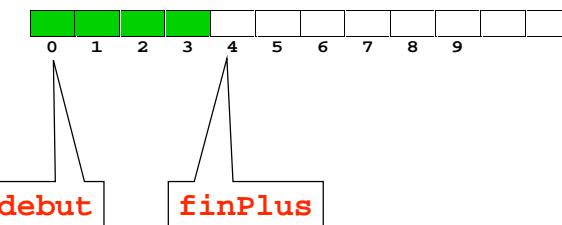
15/09/99

© 1999 Peter Sander

70

## Types Fondamentaux substring()

- Premier élément du **String** est l'élément 0
- **length()** est dernier élément - 1
  - `substring.length() == finPlus - debut`



© 1999 Peter Sander

72

## Types Fondamentaux **String** (3)

- Toute entrée au terminal est un **String**

```
String nom = "Nicolas";
nom = "Jacques";
nom = Console.in.readWord(); // un seul mot
<- Fox Mulder
System.out.println(nom);
-> Fox
nom = Console.in.readLine(); //une ligne
enti鑑re
<- Dana Scully
System.out.println(nom);
-> Dana Scully
```

15/09/99

© 1999 Peter Sander

73

## Types Fondamentaux **String** (4)

- Pour avoir un num閞ique

```
int promo = Console.in.readInt();
double moyenne = Console.in.readDouble();
```



15/09/99

© 1999 Peter Sander

74

## Types Fondamentaux **String** (5)

- L'op閞ateur « + » enchaîne deux **String**

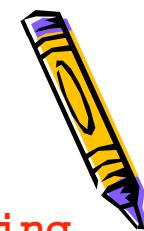
```
String s1 = "Ah que ";
System.out.println(s1 + "coucou");
-> Ah que coucou
```



15/09/99

© 1999 Peter Sander

75



## Types Fondamentaux **String** (6)

- Op閞ateurs

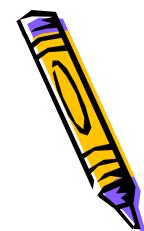
s.length()	Longueur de <b>s</b>
s.substring(i, j)	Sous-chane des de pos. <b>i</b> & <b>j-1</b>
s.toUpperCase()	<b>s</b> toute en majuscules
s.toLowerCase()	<b>s</b> toute en minuscules
"" + x	Le nombre <b>x</b> en caractres
Numeric.parseDouble(s)	<b>double</b> reprsent par <b>s</b>
Integer.parseInt(s)	<b>int</b> reprsent par <b>s</b>



15/09/99

© 1999 Peter Sander

76



# Types Fondamentaux

## Mise en Forme

- Caractères spéciaux

\n	β la ligne
\t	tab

- Sont équivalents

```
System.out.println("Ah que coucou");  
System.out.print("Ah que coucou\n");
```



15/09/99

© 1999 Peter Sander

77

## Décisions Typage

- Type **boolean**
  - Prend des valeurs
    - true
    - false

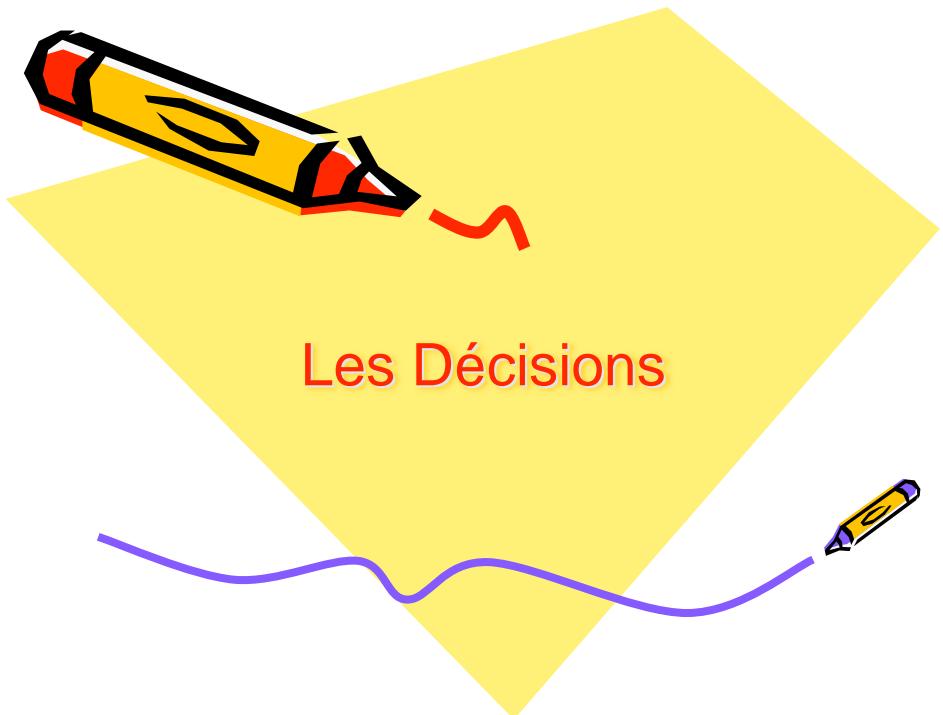
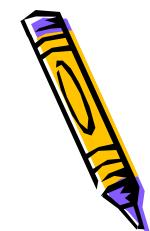
```
boolean javaEstFacile = true;
```



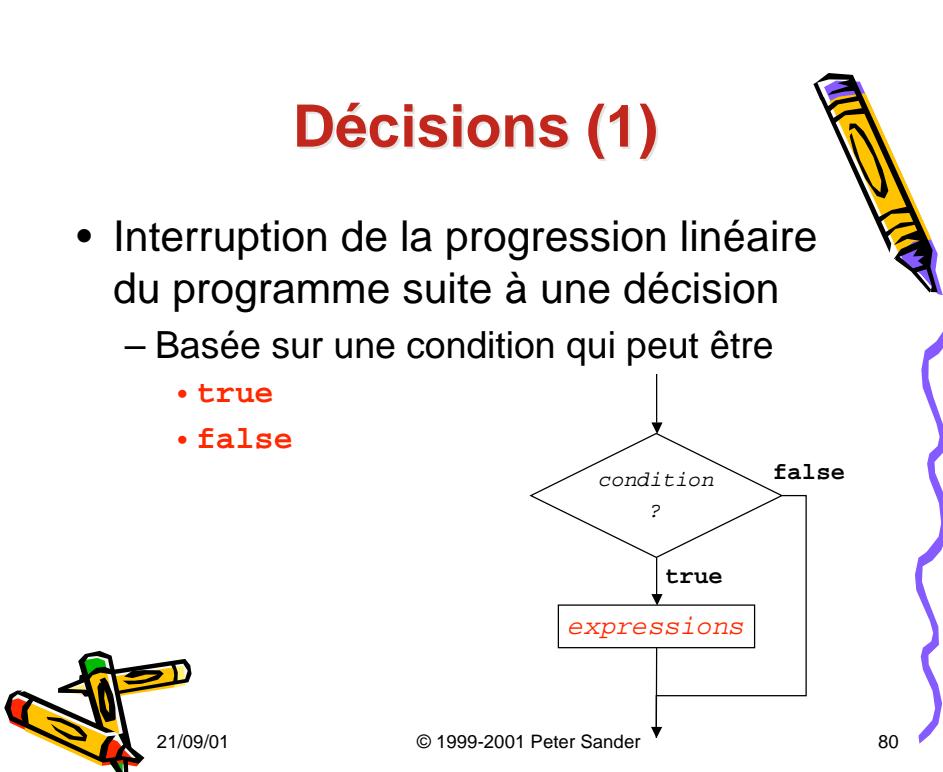
21/09/02

© 1999-2002 Peter Sander

79



## Les Décisions



## Décisions (1)

- Interruption de la progression linéaire du programme suite à une décision
  - Basée sur une condition qui peut être
    - true
    - false



© 1999-2001 Peter Sander

80



## Décisions (2)

- Conditions basées sur des opérateurs relationnels
  - e.g. « égal à », « supérieur à »...
- Opérateurs relationnels combinées par opérations logiques
  - e.g. « et », « ou »...
- Traduction en code par des structures
  - **if**
  - **while**
  - **for**
  - **do**



21/09/01

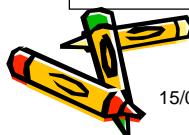
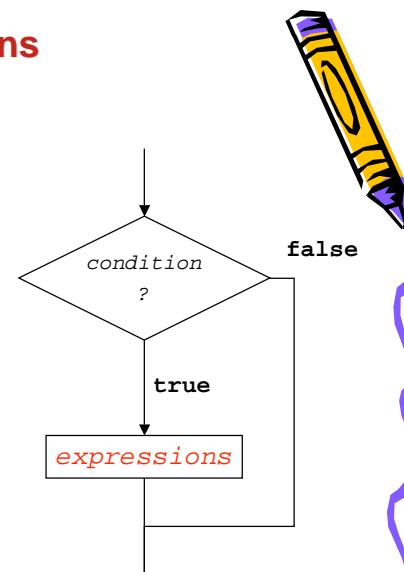
© 1999-2001 Peter Sander

81



### Décisions if

```
if (condition) {
    expressions
}
stopVoiture();
if (piecesJaunes <= 0) {
    baisseVitre();
    System.out.println(
        "Désolé, mec !");
    leveVitre();
}
demarreEnTrombe();
```



15/09/99

© 1999 Peter Sander

83

## Décisions Opérateurs Relationnels

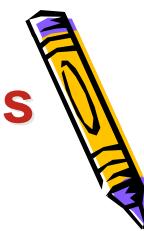
Math	Java	Description
>	>	SupÉrieur β
≥	≥	SupÈrieur ou Ègal β
<	<	InfÈrieur β
□	≤	InfÈrieur ou Ègal β
=	==	EgalitÈ
≠	!=	InÈgalitÈ



15/09/99

© 1999 Peter Sander

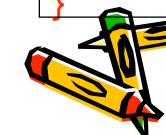
82



### Décisions if...else

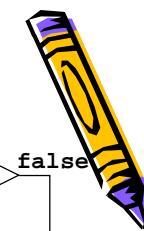
```
if (condition) {
    oui expressions
} else {
    non expressions
}
```

```
if (r >= 0) {
    sqrt = Math.sqrt(r);
} else {
    System.err.println("Erreur");
}
```



15/09/99

© 1999-2000 Peter Sander



84

83



## Décisions

# Egalité – le Piège du Débutant

- Lequel est le bon ?

```
if (vitesse = 160) {
    points == -8;
}
```

```
if (vitesse == 130) {
    points = -6;
}
```

```
if (vitesse = 110) {
    points = -4;
}
```

- Evidemment...

```
if (vitesse == 90) {
    points = 0;
}
```

### Ne pas confondre

- Affectation =
- Égalité ==



21/09/01

© 1999-2001 Peter Sander

85



## Décisions

# Opérateurs Relationnels

- Comparaison des **String**

```
String nom = "Fred";
if (nom.equals("Frod")) {
    System.out.println("Alors, Frod...");
}
```

- C'est la méthode **equals()** appliquée à une variable de type **String**
- Caractère par caractère :



15/09/99

F r e d  
| | |  
F r o d

© 1999 Peter Sander

86



## Décisions

# Opérations Logiques

- Et : **&&**
- Ou : **||**
- Négation : **!**
- Ordre de priorité
  - !** précède **&&** précède **||**
  - a || !b && c** est équivalent à **a || (( !b) && c)**
  - ()** peuvent rendre l'expression plus claire



21/09/01

© 1999-2001 Peter Sander

87



## Décisions

### &&

a	b	a && b
true	true	true
true	false	false
false	?	?
false	false	false

- Fonctionnalité

```
int vent = Console.in.readInt();
String type = "Ouragan";

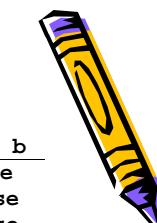
if (vent < 64 && vent >= 56) {
    type = "Violente tempête";
}
if (vent < 56 && vent >= 48) {
    type = "Tempête";
```



15/09/99

© 1999 Peter Sander

88



## Décisions

||

a	b	a    b
true	?	true
false	true	true
false	false	false

- Fonctionnalité

```
int beaufort = Console.in.readInt();

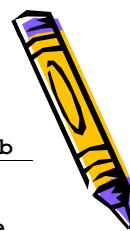
if (beaufort < 0 || beaufort > 12) {
    System.err.println("Entrée erronnée");
} else {
    fait quelquechose
}
```



15/09/99

© 1999 Peter Sander

89



## Décisions

!

a	!a
true	false
false	true

- Fonctionnalité

```
int beaufort = Console.in.readInt();

if (!(beaufort > 9)) {
    System.out.println("Sortons");
} else {
    System.out.println("Au plumard");
}
```



21/09/01

© 1999-2001 Peter Sander

90

## Décisions Variables Logiques

- Type fondamental **boolean**

```
boolean beau = true;
boolean riche = true;

if (beau && riche) {
    System.out.println("Ça vaut mieux
    !");
}
```

- Valeurs possibles : **true false**



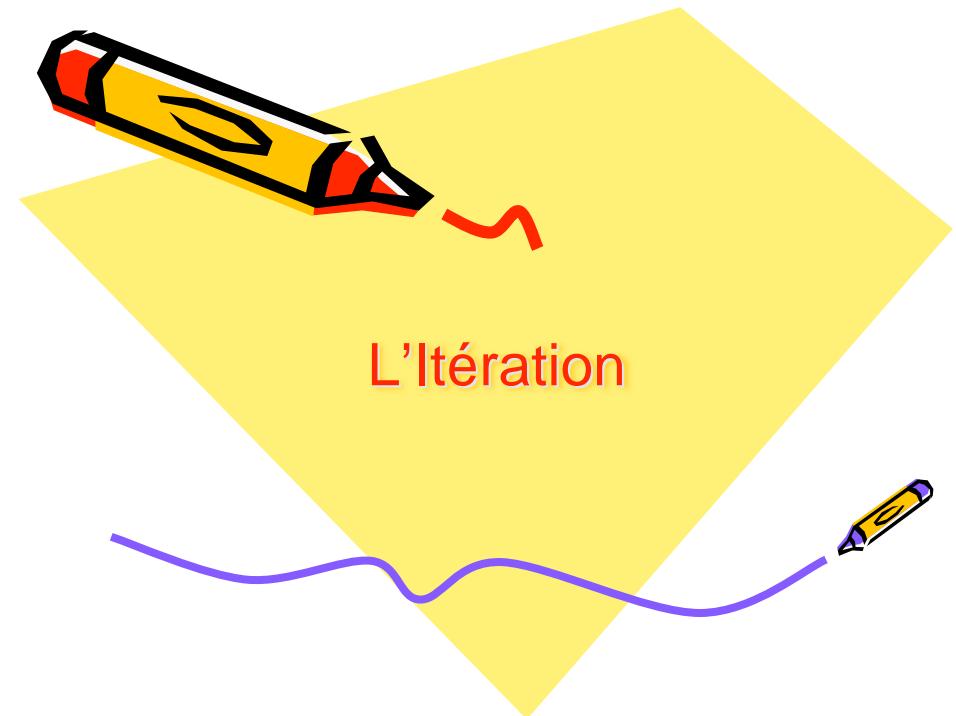
21/09/01

© 1999-2001 Peter Sander

91



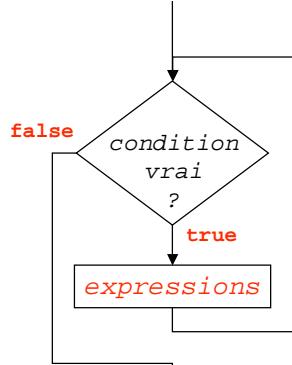
## L'Iteration



## Itération while (1)

- Boucle tant que (while) une condition est vraie

```
while (condition) {  
    expressions  
}
```

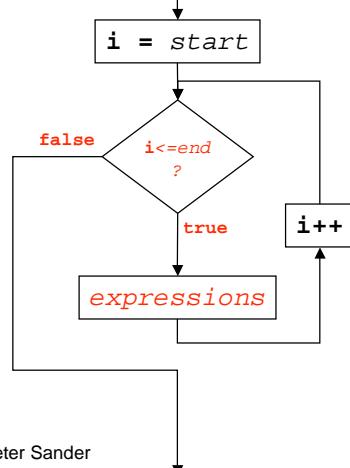


© 1999 Peter Sander

93

## Itération for (1)

```
for (i = start; i <= end;  
     i++) {  
    expressions  
}
```



© 1999 Peter Sander

95

## Itération while (2)

- La boucle la plus souvent rencontrée

```
i = valInit;  
while (i <= valeurFin) {  
    expressions  
    i++;  
}
```

– Si souvent qu'elle s'écrit...



15/09/99

© 1999 Peter Sander

94

## Itération for (2)

- Exemple - pour calculer

$$n! = 1 \times 2 \times 3 \times \dots \times n$$

```
public int factoriel(int n) {  
    int facteur;  
    int produit = 1;  
    for (facteur = n; facteur > 0; facteur--) {  
        produit = produit * facteur;  
    }  
    return produit;  
}
```



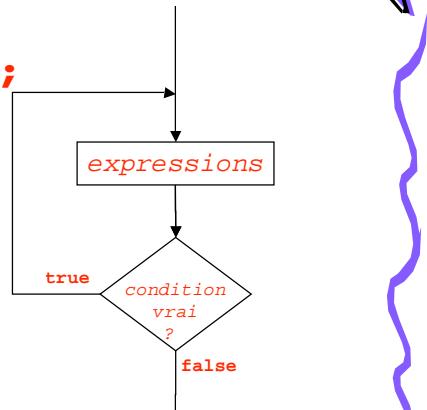
15/09/99

© 1999 Peter Sander

96

## Itération do

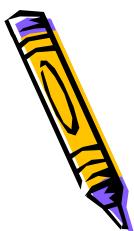
```
do {  
    expressions  
} while (condition);
```



15/09/99

© 1999 Peter Sander

97



## Itération Conditions

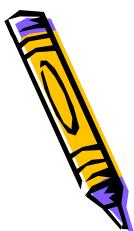
- Conditions de bouclage
  - Compteur
    - `i < 100`
  - Sentinelle
    - `valeurEntree != 0`
  - Flag
    - `fait != true`
  - Borne
    - `montant < 0.5`



21/09/01

© 1999-2001 Peter Sander

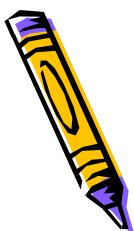
98



## Itération Compteur

- Passer par toutes les valeurs

```
for (int i = 1; i <= 100; i++) {  
    System.out.println(i);  
}
```



21/09/01

© 1999-2001 Peter Sander

99

## Itération Sentinelle

- En attente d'une condition particulière

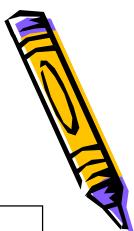
```
do {  
    int valeurEntree =  
        Console.in.readInt();  
    if (valeurEntree != -1) {  
        fait quelquechose  
    }  
} while (valeurEntree != -1)
```



21/09/01

© 1999-2001 Peter Sander

100



## Itération Flag

- Signale une condition

```
boolean fait = false;  
while (!fait) {  
    essaie de faire le nécessaire  
}
```



21/09/02

© 1999-2002 Peter Sander

101



## Itération Borne

- Limite à ne pas dépasser

```
double montant = soldeInitiale;  
while (montant > 0.5) {  
    dépenser (presque) sans compter  
}
```



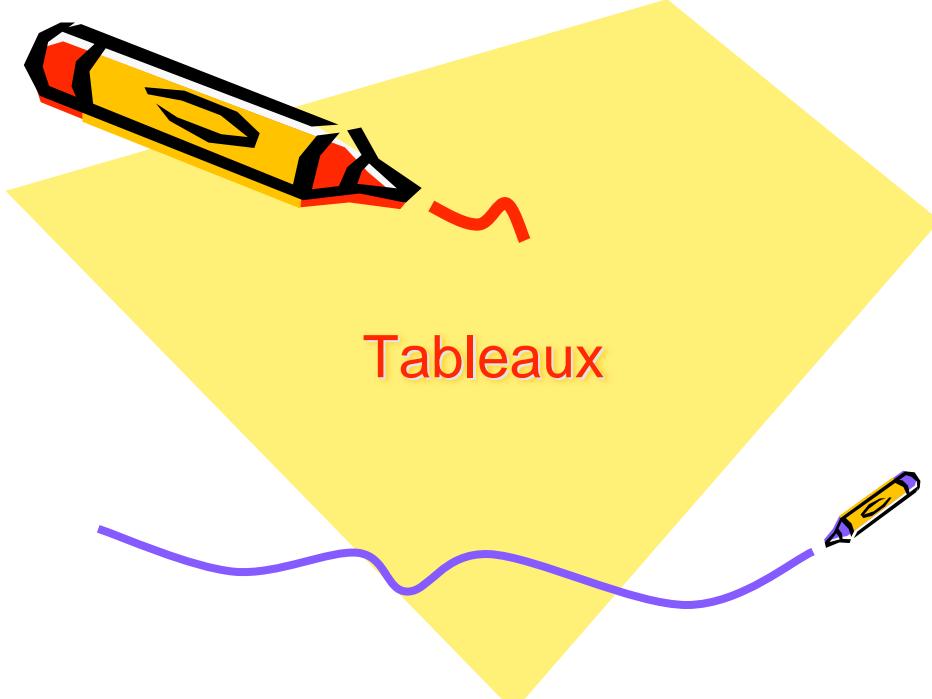
© 1999-2001 Peter Sander

102

21/09/01



## Tableaux



## Structures de Données

- Une variable de type primitif
  - Stockage d'un élément d'un type donné
    - int, double, char, boolean, String...
- Un tableau
  - Stockage d'éléments tous du même type
    - int[], double[], char[], boolean[], String[]...
- Une collection d'objets
  - Stockage divers et varié
    - Vector, ArrayList, HashMap, LinkedList...



© 2002 Peter Sander

104



# Stockage d'Éléments Similaires

- Peu pratique en variables...

```
String dept01 = "Ain";
String dept02 = "Aisne";
...
String dept95 = "Val d'Oise";
```

- Mieux en tableau...

```
String[] dept = {"Québec", "Ain", "Aisne",
                  ..., "Val d'Oise"};
```



23/09/02

© 1999-2002 Peter Sander

105



# Tableaux

- Array

- Stockage d'éléments tous du même type
- Structure à part entière
- Un tableau est un objet référencé
- Assimilable à une classe
- Création en trois étapes
  - 1. déclaration
  - 2. allocation de mémoire
  - 3. initialisation des éléments



23/09/02

© 1999-2002 Peter Sander

106

## Tableaux Déclaration

- Indiqué par []

- Deux possibilités

```
type[] nom;
type nom[];
```

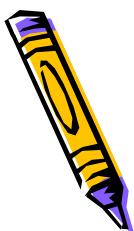
```
int[] tableau1;
int tableau2[];
int[][] matrice; // tableau bidimensionnel
int[] x, y[]; // équivalent à int x[], y[][];
int tab[10]; // ne compile pas
```



15/09/99

© 1999 Peter Sander

107



## Tableaux Allocation

- Alloué dynamiquement

- à l'aide du mot clé new

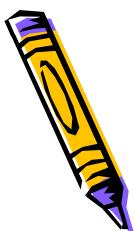
```
int[] tableau1; // déclaration
tableau1 = new int[10]; // allocation
int tableau2[]; // déclaration
tableau2 = new int[35]; // allocation
int[][] matrice; // déclaration
matrice = new int[2][4]; // allocation
int[] x, y[]; // déclaration
x = new int[5]; // allocation
y = new int[3][2]; // allocation
```



24/09/01

© 1999-2001 Peter Sander

108



# Tableaux Déclaration et Allocation

- Peut combiner déclaration et allocation

```
int[] tableau1 = new int[10];
int tableau2[] = new int[35];
int[][] matrice = new int[2][4];
int[] x = new int[5];
int[] y[] = new int[3][2];
```



15/09/99

© 1999 Peter Sander

109



# Tableaux Initialisation (1)

- Chaque élément initialisé séparément

```
int[] tablo = new int[10];
for (int i = 0; i < 10; i++) {
    tablo[i] = i;
}
```



© 1999 Peter Sander

110



# Tableaux Initialisation (2)

- Valeurs initiales peuvent être énumérées

```
int[] joursParMois = {31, 28, 31,
                      30, 31, 30, 31, 31, 30, 31, 30, 31};
```

– Expédie déclaration, allocation,  
initialisation



24/09/01

© 1999-2001 Peter Sander

111



# Tableaux De String

- Pareil que pour tableaux de primitifs

```
String[] jours = {"lundi",
                  "mardi", "mercredi", "jeudi",
                  "vendredi", "samedi",
                  "dimanche"};
```



© 1999 Peter Sander

112



## Tableaux Accès

- Indexage à partir de 0
- Accès aux éléments par [ ]  
`tablo[i] i = 0..tablo.length - 1`
- Nombre d'éléments donné par la variable  
`nom.length`

```
for (int i = 0; i < tablo.length; i++)  
{  
    System.out.print(tablo[i] + " ");  
    tablo[i] = -tablo[i];
```

24/09/01

© 1999-2001 Peter Sander

113



## Tableaux Accès - l'Erreur du Débutant

- Y a-t-il un problème ?

```
for (int i = 0; i <= tablo.length; i++) {  
    System.out.print(tablo[i] + " ");  
}
```

- Le compilateur balance  
`java.lang.ArrayIndexOutOfBoundsException:  
10 at Truc.main(Truc.java:6)`



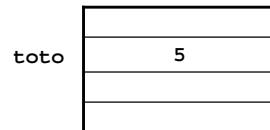
15/09/99

© 1999 Peter Sander

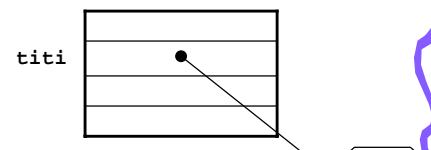
114

## Tableaux En Mémoire

- Variable type primitive  
`int toto = 5;`
- Case mémoire contient la valeur



- Variable type référencée  
`String titi = "cinq";`
- Case mémoire contient référence de la valeur
- Suivre la référence pour trouver la valeur

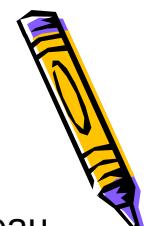


115



23/09/02

© 1999-2002 Peter Sander



## Tableaux Copier un Tableau

- Copie d'un tableau ou une partie de tableau dans un autre tableau
  - `System.arraycopy(...)`
- Il s'agit d'une copie « de surface »
  - Les références aux objets sont copiées
    - pas les objets
    - les objets seront donc partagés

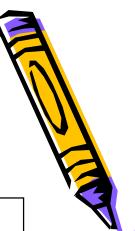


15/09/99

© 1999 Peter Sander

116

## Classes et Objets



## Programmation Orientée Objet Création d'objet

- Cas spécial : `String`

```
String maCaisse = "R5";  
String maVoiture = new String("Z4");  
maVoiture = new String("DB5");
```

- Classe de type `Toto`

```
Toto unToto = new Toto();  
Toto unAutreToto = new Toto(17, "euh");
```



15/09/99

© 1999 Peter Sander

118

## Programmation Orientée Objet Création d'objet

- Création d'objet

- `String`

```
String maCaisse = "R5";  
String maVoiture = new String("Z4");  
maVoiture = new String("DB5");
```

- `Toto`

```
Toto unToto = new Toto();
```

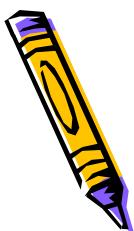
affectation      création d'objet



29/09/02

© 1999-2002 Peter Sander

119



## Programmation Orientée Objet Création d'objet

```
int valeurNutritive = 0;
```

déclaration

```
Produit diner = new Produit(0,"BigMac");
```

création

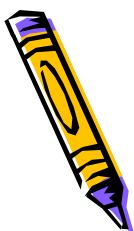
création par constructeur



29/09/02

© 1999-2002 Peter Sander

120



# Programmation Orientée Objet

## Création d'objet

- Signature du constructeur dans la documentation
  - `public Time(int y, int m, int d, int hr, int min, int sec)`
- Donc on peut créer `Time`

```
Time millenium = new Time(2000, 1, 1, 0, 0, 0);
```



29/09/02

© 1999-2002 Peter Sander

121



# Programmation Orientée Objet

## Utilisation d'objet

- Invocation de méthode  
`objet.methode();`
- Méthodes d'accès
  - Accesseur (*getter*)  
`objet.getProp();`
  - Mutateur (*setter*)  
`objet.setProp();`

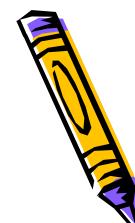
```
Time ahui = new Time(); // création  
int annee = ahui.getYear(); // utilisation
```



15/09/99

© 1999 Peter Sander

122



# Classe et Objet

- Le rêve
  - Une classe
- La réalité
  - Un objet

```
public class Voiture {  
    String marque = "Citroën";  
    String modele = "2CV";  
    Color couleur = Color.red;  
    public void demarre() {  
        ...  
    }  
    public void accelere(int v) {  
        ...  
    }  
}
```

```
public class Toto {  
    ...  
    public static void main(String[] args) {  
        Voiture tire = new Voiture("Porsche","911",  
            Color.black);  
        tire.demarre();  
        tire.accelere(260);  
        ...  
    }  
}
```



15/09/99

© 1999 Peter Sander

123



# Variables

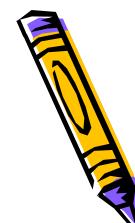
- « d'Instance »
  - Contenues :
    - dans une classe
    - dans aucune méthode
  - Représentent l'état interne d'un objet
- « Locales »
  - Contenues
    - dans une méthode
  - Utilisées par la méthode pendant son exécution



15/09/99

© 1999 Peter Sander

124



## Variables d'Instance

```
public class Voiture {
    String marque = "Citroën";
    String modèle = "2CV";
    Color couleur = Color.red;

    public void démarre() {
        ...
    }

    public void accélère(int v) {
        ...
    }
    ...
}
```

15/09/99

© 1999 Peter Sander

125

## Variables Locales

```
public class Toto {
    ...
    public static void main(String[] args) {
        Voiture tire = new Voiture(
            "Porsche",
            "911",
            Color.black);
        tire.démarre();
        tire.accélère(260);
        ...
    }
}
```

15/09/99

126

## Classe et Objet

JVM exécute

```
public class Toto {
    ...
    public static void
    main(String[] args) {
        Voiture tire = new
        Voiture("Porsche", "911",
        ...
        tire.démarre();
        tire.accélère(260);
        ...
    }
}
```

Toto utilise l'objet

public class Voiture {
 String marque = "Citroën";
 String modèle = "2CV";
 Color couleur = Color.red;

 public Voiture(...) {
 ...
 }
 ...
}

Toto demande une nouvelle instance

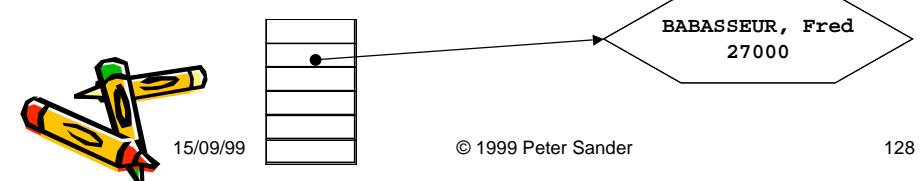
15/09/99

© 1999 Peter Sander

127

## Affectation

```
public class SalarieTest {
    public static void main(String[] args) {
        Salarie fred = new Salarie(
            "BABASSEUR, Fred", 24000);
        double nouvPaie = fred.getPaie() + 3000;
        fred.setPaie(nouvPaie);
        System.out.println("Nom : " + fred.getNom());
        System.out.println ("Paie : " + fred.getPaie());
    }
}
```

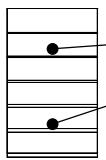


© 1999 Peter Sander

128

## Affectation

```
public class SalarieTest {  
    public static void main(String[] args) {  
        Salarie fred = new Salarie(  
            "BABASSEUR, Fred", 24000);  
        Salarie boss = fred;  
        boss.setPaie(60000);  
    }  
}
```



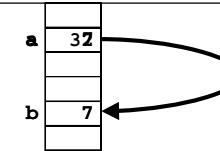
© 1999 Peter Sander

129

## Programmation Orientée Objet Copie de Type Primitif

- Pour types primitifs

```
int a = 7;  
int b = a;  
a = 32;  
System.out.println(a + ", " + b);  
-> 32, 7
```



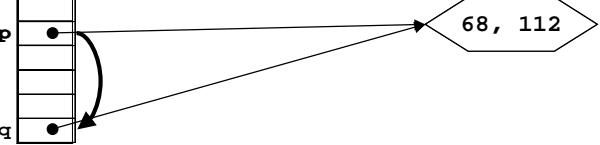
© 1999 Peter Sander

130

## Programmation Orientée Objet Copie de Type Référencé

- Pour types référencés

```
Point p = new Point(12, 34);  
Point q = p;  
p.move(56, 78);  
System.out.println(p + ", " + q);  
-> Point[x=68.0,y=112.0],  
      Point[x=68.0,y=112.0]
```



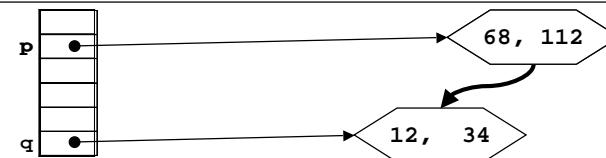
© 1999 Peter Sander

131

## Programmation Orientée Objet clone de Type Référencé

- Pour types référencés

```
Point p = new Point(12, 34);  
Point q = (Point) p.clone();  
p.move(56, 78);  
System.out.println(p + ", " + q);  
-> Point[x=68.0,y=112.0],  
      Point[x=12.0,y=34.0]
```



© 1999 Peter Sander

132

## Les Méthodes

## Méthodes Points de Vue (1)

- Utilisation d'une méthode
- Définition d'une méthode

```
...  
System.out.println("Ah que" + " hello world !");  
...
```

```
...  
public void maMethode(byte q1)  
{  
    ...  
}  
...
```



28/09/02

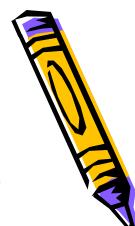
© 1999-2002 Peter Sander

134

## Méthodes Points de Vue (2)

- Une ligne de code Java...est contenue dans une méthode...

```
package fr.essi.sander.hacks;  
  
public class HelloWorld {  
    ...  
    public static void main(String[] args) {  
        ...  
        System.out.println("Ah que" +  
                           " hello world !");  
        ...  
    }  
    ...  
}
```



- Une méthode fournit un service
  - Nous savons ce qu'elle fait
  - Nous ignorons comment elle le fait

```
r2 = b * b - 4 * a * c;  
r = Math.sqrt(r2);  
soln1 = (-b + r) / (2 * a);  
soln2 = (-b - r) / (2 * a);
```

Calcul de la  
racine carrée

Mais selon quel  
algorithme ?

29/09/02

© 1999-2002 Peter Sander

135

## Méthodes Boîte Noir

- Une méthode fournit un service
  - Nous savons ce qu'elle fait
  - Nous ignorons comment elle le fait

```
r2 = b * b - 4 * a * c;  
r = Math.sqrt(r2);  
soln1 = (-b + r) / (2 * a);  
soln2 = (-b - r) / (2 * a);
```

15/09/99

© 1999 Peter Sander

136

# Méthodes

# Entrées / Sorties

- Peut prendre zéro ou plusieurs paramètres en entrée

```
r = Math.sqrt(r2);
r = Math.pow(r2, 2);
System.out.println("Racine = " + r);
```

- Zéro ou une seule valeur de retour



29/09/02

© 1999-2002 Peter Sander

137

# Méthodes

# Signature

- La *signature* décrit une méthode
    - Nom
    - Nombre et type des paramètres
  - D'autres infos n'en font pas partie
    - Niveau d'accès
    - Autres mots clé
    - Type de valeur renvoyée



15/09/99

© 1999 Peter Sander

139

# Méthodes

## Création - Structure

## Signature

```
public static void main(String[] args) {  
    ...  
    System.out.println("Bienvenue au cauchemar");
```

...  
The following table summarizes the results of the experiments.

A simple illustration of two yellow crayons with black outlines and red caps.

29/09/

Corp

© 1999-2001 Peter Sande

138

# Méthodes

# Signature

Diagram illustrating the structure of a Java main method:

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

The diagram shows the following annotations:

- An annotation pointing to the word "main": **Nom**
- An annotation pointing to the parameter list "String[] args": **Nombre et type de paramètres**



29/09/0

© 1999-2001 Peter Sande

140

## Méthodes D'autres Informations

```
public static void main(String[] args){  
}  
Mots clé  
Niveau d'accès  
Type de valeur retournée
```

29/09/01

© 1999-2001 Peter Sander

141

## Méthodes Passage de Paramètres

- Java passe des paramètres par valeur
  - Pass by value*
- D'autres langages permettent le passage par référence
  - Pass by reference*



© 1999-2001 Peter Sander

143

## Méthodes Passage de Paramètres

- Création d'une méthode
- Utilisation d'une méthode

```
...  
boolean bilan =  
    Console.in.readBoolean(  
        );  
fou(bilan);  
fou(false);  
...
```

```
public void fou(boolean  
    dingue) {  
    if (dingue) {  
  
        System.out.println("Apte");  
    } else {  
  
        System.out.println("Inapte");  
    }  
}
```

Comment lier ?

29/09/01

© 1999-2001 Peter Sander

142

## Méthodes Passage par Valeur (1)

```
...  
tauxAnnuel = 6;  
b = soldeFutur(tauxAnnuel);  
...
```

p reçoit la valeur 6  
(la valeur de tauxAnnuel)

```
public double soldeFutur(double p) {  
    ...  
    ...  
}
```



© 1999 Peter Sander

144

## Méthodes

# Passage par Valeur (2)

```
...
tauxAnnuel = 6;
b = soldeFutur(tauxAnnuel);
...
```

b reçoit la valeur  
du résultat

```
public double soldeFutur(double p) {
    double r = 1000 * Math.pow(1 + p / (12*100),
                               12*10);
    return r;
}
```

15/09/99

© 1999 Peter Sander

145



## Méthodes

# Passage par Valeur (3)



Mémoire soldeFutur



15/09/99

© 1999 Peter Sander

146

## Méthodes

# Passage par Valeur (4)

- Modifier un paramètre dans une méthode...

```
public void compteEnBanque(double solde) {
    double solde2 = solde * solde;
    solde = solde2;
}
```

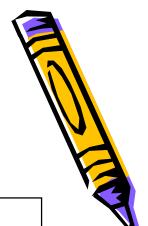
...n'a aucun effet à l'extérieur

```
double monSolde = 12.34;
compteEnBanque(monSolde);
System.out.println(monSolde);
      -> 12.34
```

15/09/99

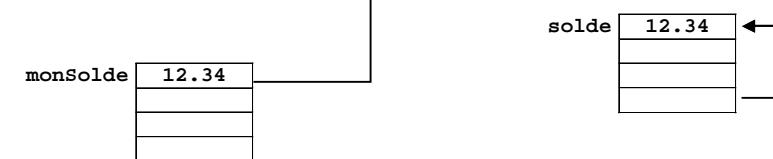
© 1999 Peter Sander

147



## Méthodes

# Passage par Valeur (5)



Mémoire soldeFutur



15/09/99

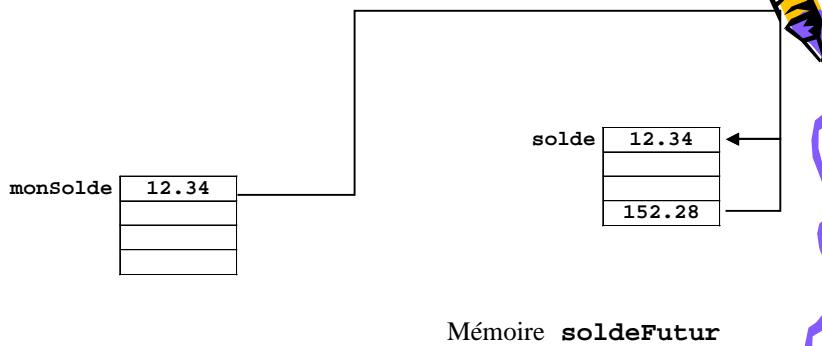
© 1999 Peter Sander

148



## Méthodes

# Passage par Valeur (5)



15/09/99

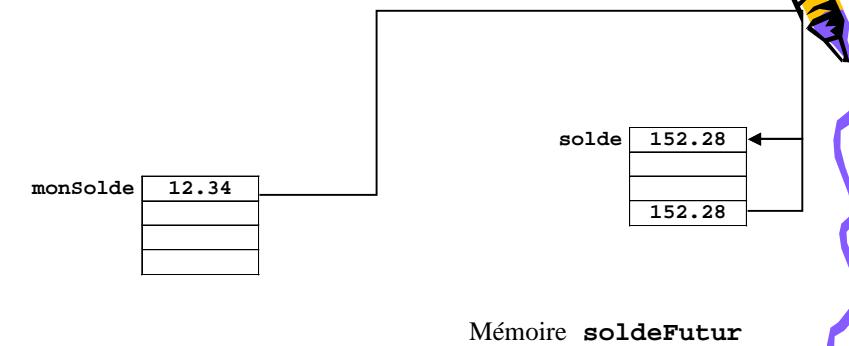
© 1999 Peter Sander

149

Mémoire `soldeFutur`

## Méthodes

# Passage par Valeur (5)



Mémoire `soldeFutur`

15/09/99

© 1999 Peter Sander

150

## Méthodes

# Passage de Type Référencé

- ⚠ Java passe les types référencés par...valeur
- Subtilité - valeur passée est la référence !



15/09/99

© 1999 Peter Sander

151

## Méthodes

# Passage de Type Référencé (1)

```
String[] pablo =  
    {"Málaga", "La  
    Corogne", "Barcelone",  
    "Vallauris", "Cannes"};  
  
public void  
artiste(String[]  
sejour) {  
    ...  
    sejour[4] = "Mougins";  
    ...  
}
```

```
System.out.println("Séjour  
final : " + pablo[4]);  
-> Cannes  
  
artiste(pablo);  
  
System.out.println("Séjour  
final : " + pablo[4]);  
-> Mougins  
• Pourquoi ça change ?
```



29/09/02

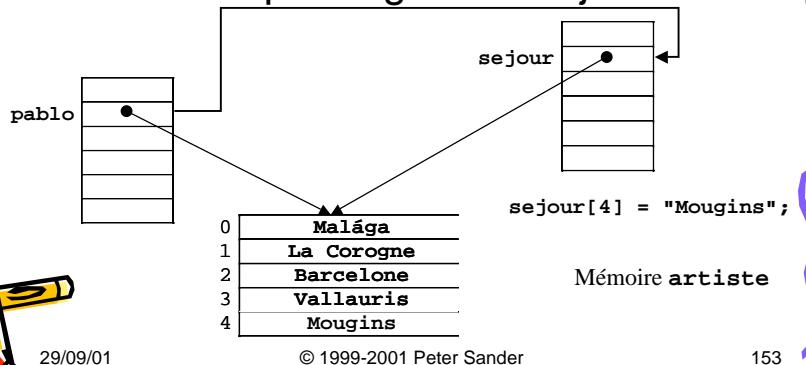
© 1999-2002 Peter Sander

152

## Méthodes

# Passage de Type Référencé (2)

- La méthode reçoit la valeur de la référence
- Une méthode peut agir sur l'objet



## Méthodes Le Retour

- Valeur renournée par mot clé `return`
- `return` termine immédiatement la méthode

```
public int beaufort(int vent) {  
    if (vent >= 64) {  
        return 12;  
    }  
    else if (vent >= 56) {  
        return 11;  
    }  
    else if (vent >= 48) {  
        ...  
    }  
    else {  
        return 0;  
    }  
}
```

29/09/02

© 1999-2002 Peter Sander

155

## Méthodes

# Passage de Paramètres

- Java n'ayant pas de *pointeurs*...
  - ...n'a pas de passage de *primitifs* par référence
- Mais, Java a des *références* sur des types...
  - ...référencés (des objets)
  - Nous les rencontrerons plus tard...
- En Java,
  - on ne dit pas « pointeur »
  - on dit « référence »



15/09/99

© 1999 Peter Sander

154



## Méthodes Le Retour de l'Erreur du Débutant

- `return` est un mot clé et non un appel de méthode

```
public String inelegant() {  
    return("Encore un !");  
}
```



29/09/01

© 1999-2001 Peter Sander

156

## Méthodes Le Retour

- Valeur de retour n'est pas obligatoire

```
public void beaufort(int vent) {  
    if (vent >= 64) {  
        System.out.println("Ouragan");  
        return;  
    }  
    else if (vent >= 56) {  
        ...  
    }  
    System.out.println("Le calme plat");  
}
```

return  
explicite

return  
implicite

15/09/99

© 1999 Peter Sander

157

## Méthodes Le Retour

- Une fois promis, il faut livrer...

```
public boolean dansIntervalle(int val,  
    int min, int max) {  
    return val >= min && val <= max;  
}
```



15/09/99

© 1999 Peter Sander

159

## Méthodes Le Retour

- Une fois promis, il faut livrer...

```
public boolean dansIntervalle(int val,  
    int min, int max) {  
    if (val >= min && val <= max) {  
        return true;  
    }  
}
```



Et si la condition  
était fausse ?



15/09/99

© 1999 Peter Sander

158

## Méthodes Typage

- Paramètres doivent s'accorder avec signature

```
public boolean dansIntervalle(int val,  
    int min, int max) {  
    return val >= min && val <= max;  
}
```

– Sinon, utilisation erronée

```
...  
dansIntervalle(3.5, 3, 4);  
...
```



15/09/99

© 1999 Peter Sander

160

## Méthodes Typage

- Valeur rentrée doit s'accorder avec déclaration

```
...
double malheur = Console.in.readDouble();
int malheurAbs = Math.abs(malheur);
...
```



15/09/99

© 1999 Peter Sander

161



## Méthodes Coercition (Cast)

- Changement explicite du type

```
...
double malheur = Console.in.readDouble();
int malheurAbs = (int) Math.abs(malheur);
...
```



Je, soussigné Machin, m'engage  
sur l'honneur à prendre toute  
responsabilité pour cette affectation

© 1999 Peter Sander

162



## Méthodes Coercition (Cast)

- Traitement spéciale de **String**
  - l'opérateur + enchaîne 2 **String**

```
String msg = Console.in.readString();
System.out.println("J'ai reçu " + msg);
```

– + convertit ses arguments en **String**

```
String racine2 = "" + Math.sqrt(2);
```



15/09/99

© 1999 Peter Sander

163



## Méthodes Variables

- Déclarées dans une méthode
  - Variables locales* à la méthode
  - Faut les initialiser, sinon...
  - Durée de vie limité (*automatic duration*)
    - créées en début d'exécution de la méthode
    - existent le temps d'exécution de la méthode
    - disparaissent au **return** de la méthode
  - Visibilité limitée



29/09/02

© 1999-2002 Peter Sander

164



## Méthodes Variables Locales

```
public double compteEnBanque(double  
    solde) {  
    double gnouveau;  
    if (solde >= 0) {  
        gnouveau = solde * solde;  
    }  
    return gnouveau;  
}
```

Variable locale



15/09/99

© 1999 Peter Sander

165



## Méthodes Variables Locales

```
public double compteEnBanque(double  
    solde) {  
    double gnouveau = solde; Faut  
    if (solde >= 0) {  
        gnouveau = solde * solde; l'initialiser  
    }  
    return gnouveau;  
}
```



15/09/99

© 1999 Peter Sander

167



## Méthodes Variables Locales

```
public double compteEnBanque(double solde) {  
    double gnouveau;  
    if (solde >= 0) {  
        gnouveau = solde * solde;  
    }  
    return gnouveau;  
}
```

- Où est le problème ?
  - Si `solde < 0`, `gnouveau` a quelle valeur ?



15/09/99

© 1999 Peter Sander

166



## Méthodes Variables Locales

```
...  
double solde = 1000;  
solde = comteEnBanque(solde);  
solde = comteEnBanque(solde);  
...
```

Variable  
crée

Variable  
détruite



15/09/99

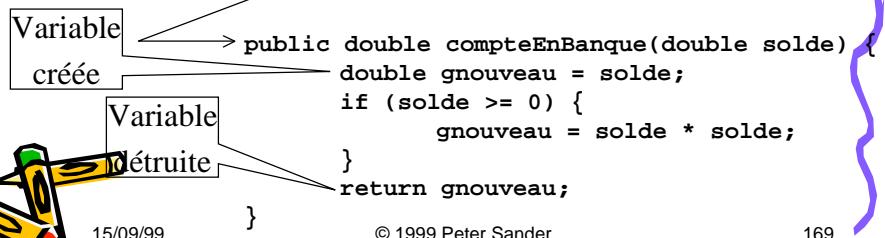
© 1999 Peter Sander

168



## Méthodes Variables Locales

```
...  
double solde = 1000;  
solde = comteEnBanque(solde);  
solde = comteEnBanque(solde);  
...
```



## Méthodes Visibilité des Variables

```
...  
{  
    int solde = Console.in.readInt();  
    solde *= 17; // à l'intérieur -  
    visible  
    System.out.println("Solde = " +  
        solde);  
}  
solde++; //à l'extérieur - invisible
```



15/09/99

© 1999 Peter Sander

171

## Méthodes Visibilité des Variables

- Visibilité limitée (scope) par {}
  - Les {} déterminent un *bloc*
  - Variable déclarée dans un bloc est
    - visible dans le bloc
    - invisible depuis l'extérieur



15/09/99

© 1999 Peter Sander

170

## Méthodes Blocs

- Utilisations multiples et variées

```
public int intRoot(double r2) {  
    int r = 0;  
    if (r2 >= 0) {  
        r = (int) Math.round(Math.sqrt(r2));  
    }  
    return r;  
}
```

– Bloc {} imbriqué dans bloc {}

– r déclarée dans {}

- visible de partout, même depuis {}



© 1999 Peter Sander

172

## Méthodes Blocs - l'Erreur du Débutant

```
public int intRoot(double r2) {  
    if (r2 >= 0) {  
        int r = 0;  
        r = (int) Math.round(Math.sqrt(r2));  
    }  
    return r; // r invisible  
}
```

- Bloc {} imbriqué dans bloc {}
- r déclarée dans {}
  - invisible depuis {}

15/09/99

© 1999 Peter Sander

173



## Méthodes

## Blocs - l'Erreur du Débutant

```
public int intRoot(double r2) {  
    int r = 0;  
    if (r2 >= 0) {  
        int r = 0; // redéclaration de r  
        r = (int) Math.round(Math.sqrt(r2));  
    }  
    return r;  
}
```

- Bloc {} imbriqué dans bloc {}
- r déclarée dans {}
  - visible depuis {}
  - redéclarée dans {}

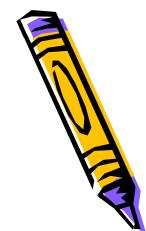
15/09/99



174

## Classes Variables et Méthodes

- Variables
  - locales
    - déclarées à l'intérieur d'une méthode
    - connues uniquement dans la méthode
  - d'instance
    - déclarées à l'extérieur de toute méthode
    - déclarées à l'intérieur d'une classe



15/09/99

© 1999 Peter Sander

175



## Classes Variables et Méthodes

- Quel en sera le résultat
  - à la compilation ?
  - à l'exécution ?

Ça compile !

```
private String moi = "et moi"; //  
d 'instance  
  
public void maMethode() {  
    System.out.println(moi);  
}
```

-> et moi



15/09/99

176

# Variables et Méthodes

- Quel en sera le résultat
  - à la compilation ?
  - à l'exécution ?

```
private String moi = "et moi"; //d'instance
public void maMethode() {
    String moi = "Moi "; // locale
    System.out.println(moi);
}
```

-&gt; Moi

Ça compile !

15/09/99

© 1999 Peter Sander

177



# Variables et Méthodes

- Pas de confusion entre
  - variable local **moi**
  - variable d'instance **moi**

```
private String moi = "et moi"; //d'instance
public void maMethode() {
    String moi = "Moi "; // locale
    System.out.println(moi);
}
```

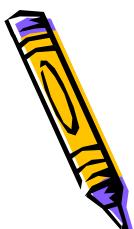
Cherche d'abord  
variable locale

En trouve  
une

15/09/99

© 1999 Peter Sander

178



# Variables et Méthodes

- Pas de confusion entre
  - variable local **moi**
  - variable d'instance **moi**

```
private String moi = "et moi"; //d'instance
public void maMethode() {
    System.out.println(moi);
}
```

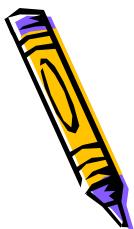
Cherche d'abord  
variable locale  
Cherche ensuite  
variable d'instance

En trouve  
une

15/09/99

© 1999 Peter Sander

179



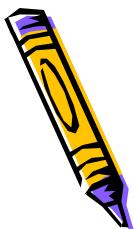
# Variables et Méthodes

- Pas de confusion entre
  - variable local **moi**
  - variable d'instance **moi**
- Comment utiliser les 2 ?
  - Deuxième effet du mot clé **this**
  - Fait référence à l'objet courant



© 1999 Peter Sander

180



## Classes

# Variables et Méthodes

- Pas de confusion entre
  - variable local `moi`
  - variable d'instance `this.moi`

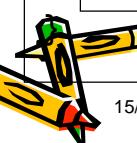
```
private String moi = "et moi"; // d'instance

public void maMethode() {
    String moi = "Moi "; // locale
    System.out.println(moi + " " +
        this.moi);
    -> Moi et moi
```

15/09/99

© 1999 Peter Sander

181



## Classes

# Variables et Méthodes

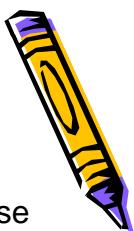
- Règle
  - Toute référence à une variable d'instance passe par `objet.nom`
  - Par exemple `fred.salaire`
  - `qcm.note`
  - `this.moi`
- Exception à la règle
  - Pour `this.nom` on peut écrire `nom`
    - `this` est facultatif



15/09/99

© 1999 Peter Sander

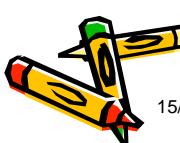
182



## Classes

# Variables et Méthodes

- Règle
  - Toute référence à une méthode d'instance passe par `objet.methode(...)`
  - Par exemple `fred.setSalary(21000)`
  - `qcm.oublieNote()`
  - `this.acheveMoi("vite")`
- Exception à la règle
  - Pour `this.methode()` on peut écrire `methode()`
    - `this` est facultatif



15/09/99

© 1999 Peter Sander

183

## Classes

# Variables et Méthodes

```
public class Boss extends Employee {
    static final double BASE = 24000;
    private double salary = BASE;
    Employee fred = new Employee("Fred", BASE);

    public raise(double raise) {
        fred.setSalary(fred.getSalary() + raise);
        this.setSalary(this.getSalary() + 2 *
            raise);
    }
}
```

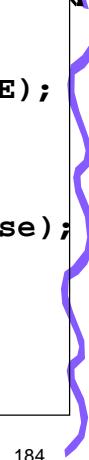
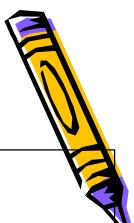
Facultatif



15/09/99

© 1999 Peter Sander

184



## Classes et Objets Construction

- Un constructeur n'est pas obligatoire...
  - ...mais quand y en a pas, y en a toujours...
  - ...sinon impossible d'instancier un objet
- Aucun constructeur déclaré pour **Toto** ?
  - Java invoque automagiquement le constructeur par défaut :

```
public Toto() {  
    super(); // késako ?
```

2/10/02

© 1999-2002 Peter Sander

186

## Construction

under Construction

## Classes Construction

- Le constructeur - une méthode spéciale
  - Même nom que la classe
  - Pas de valeur de retour
  - Sert à initialiser les variables
  - Invoqué par `new`
- Exemple - pour la classe **Clock**

```
public Clock() {  
    time = new Time();  
    clockFace = new Circle();  
    ...  
}
```

Invocation du constructeur de **Time**

Invocation du constructeur de **Circle**

187

15/09/99

© 1999 Peter Sander

## Classes Construction

- Multiples constructeurs possible
  - Méthode *surchargée* (*overloaded*)

```
public Time() { // l'heure actuelle  
    ...  
}  
  
public Time(int y, int m, int d,  
           int hr, int min, int sec) {  
    ...  
}
```

15/09/99

© 1999 Peter Sander

188

# Classes Construction

- Constructeur sans arguments
  - Initialisation aux valeurs par défaut

```
static final String DEF_NAME = "";  
static final double DEF_SALARY = 16000;  
  
String name;  
double salary;  
  
public Employee() {  
    name = DEF_NAME;  
    salary = DEF_SALARY;
```

29/09/02

© 1999-2002 Peter Sander

189

# Classes Construction

```
public Employee() {  
    // 37 lignes de code  
}  
  
public Employee(String nom) {  
    name = nom;  
    // ensuite les mêmes 36 lignes de code  
}  
  
public Employee(String nom, double salaire) {  
    name = nom;  
    salary = salaire;  
    // ensuite les mêmes 35 lignes de code  
}
```



29/09/02

© 1999-2002 Peter Sander

191

# Classes Construction

- Un constructeur peut en appeler un autre en cascade
  - Premier effet du mot clé **this**
    - référence à l'objet courant

```
public Constructeur(...) {  
    this(...); // appelle au constructeur  
  
    ...  
}
```



N.B. : obligatoirement sur la première ligne !

15/09/99

© 1999-2000 Peter Sander

190

# Classes Construction

```
public Employee() {  
    // 37 lignes de code  
}  
  
public Employee(String name) {  
    this(name, DEF_SALARY);  
}  
  
public Employee(String nom, double salaire) {  
    name = nom;  
    salary = salaire;  
    // les mêmes 35 lignes de code  
}
```



15/09/99

© 1999 Peter Sander

192

# Classes Construction

```
public Employee() {  
    this(DEF_NAME);  
}  
  
public Employee(String name) {  
    this(name, DEF_SALARY);  
}  
  
public Employee(String nom, double salaire) {  
    name = nom;  
    salary = salaire;  
    // les mêmes 35 lignes de code  
}
```

15/09/99

© 1999 Peter Sander

193

# Classes et Objets Construction

- **this(...)**
  - invoque le constructeur de l'objet en cours de construction
- **super(...)**
  - invoque le constructeur de sa superclasse



2/10/02

© 1999-2002 Peter Sander

195

# Classes et Objets Construction

- Un constructeur peut en appeler un autre
  - Premier effet du mot clé **super**

```
public Classe(...) {  
    super(...); // constructeur de la  
    superclasse  
    ...  
}
```

- N.B. : obligatoirement sur la première ligne !
- Si l'y en a pas explicitement, Java invoque automagiquement **super()**



2/10/02

© 1999-2002 Peter Sander

194

# Classes et Objets Construction

- Mais avant d'invoquer le constructeur...
  - initialisation des variables

```
static final String DEF_NAME = "";  
static final double DEF_SALARY = 16000;  
String name = DEF_NAME;  
double salary = DEF_SALARY;  
  
public Employee() {  
    super(); // invocation explicite  
}
```



15/09/99

© 1999 Peter Sander

196

# Classes et Objets Construction

- Séquence d'événements

1. Construction de la superclasse

- explicitement dans le code

```
public SousEmployee(String nom, double salaire) {  
    super(nom, salaire);  
}
```

- implicitement par Java

- invocation automatique `super()`
- erreur si constructeur sans arguments pas défini dans superclasse

2. Initialisation de variables

3. Invocation du constructeur



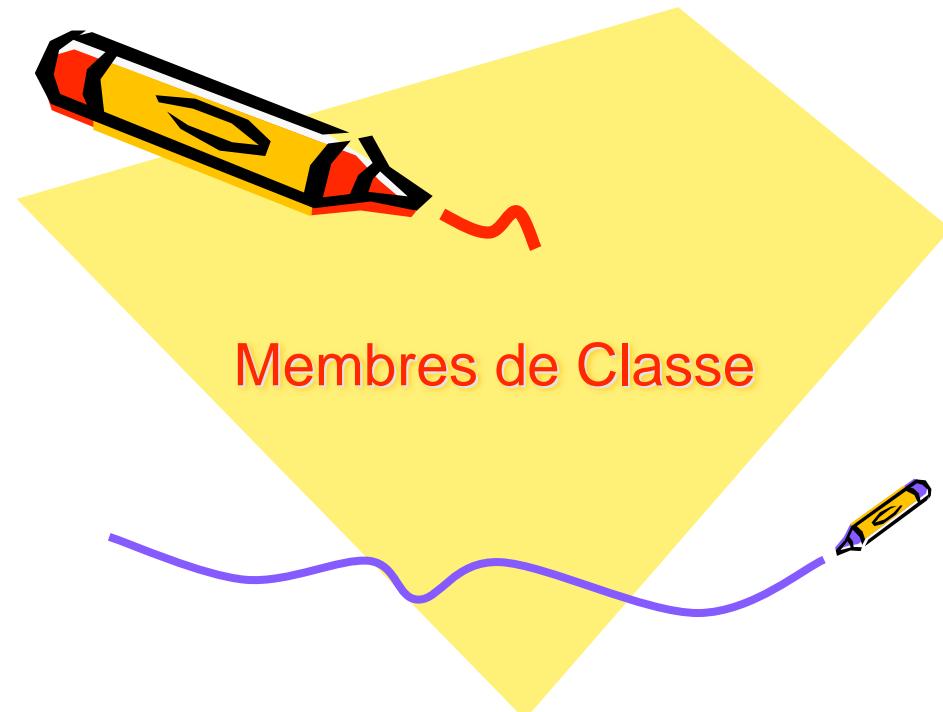
15/09/99

© 1999 Peter Sander

197



## Membres de Classe



# Classes et Objets Membres d'Instance et de Classe

- Variables d'instance

- Déclarées par `accès type nom`
- Référencées par `objet.nom`
- Une copie pour chaque instance (objet)

- Variables de classe

- Déclarées par `accès static type nom`
- Référencées par `classe.nom`
- Une seule copie pour toutes les instances



2/10/02

© 2001-2002 Peter Sander

199



# Classes et Objets Variables de Classe

- Mot clé `static` indique variable de classe

```
public class Boss {  
    public static final double BASE =  
        24000;  
    private Boss() {} // non instanciable  
}  
public class Manager {  
    private double salary = Boss.BASE;  
    ...
```



15/09/99

© 1999 Peter Sander

200



# Variables de Classe

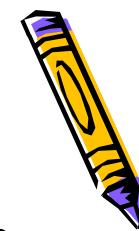
- Niveau d'accès
  - **public static** utilisable depuis une autre classe
    - généralement réservé aux constantes

```
public static final String BLAH =
    "Blah";
```
  - **private static** visible que dans sa propre classe
    - variable *partagée* par toutes les instances de la classe

15/09/99

© 1999 Peter Sander

201



# Variables de Classe

- Exemple :
  - compteur du nombre d'instances d'une classe

```
public class Gadget {
    private static int nombreExistant = 0;
    private int noDeSerie; // un par objet
    public Gadget() { // nouvel objet créé
        noDeSerie = ++Gadget.nombreExistant;
    }
    public int getNoDeSerie() {
        return noDeSerie;
    }
    ...
}
```



© 1999 Peter Sander

202



# Méthodes d'Instance et de Classe

- Méthodes d'instance
  - Déclarées par **accès retour nom(...)**
  - Référencées par **objet.nom(...)**
- Méthodes de classe
  - Déclarées par **accès static retour nom(...)**
  - Référencées par **classe.nom(...)**



2/10/02

© 2001-2002 Peter Sander

203

# Méthodes de Classe

- Pareil que pour variables
  - Méthode *partagée* par toutes les instances de la classe

```
aireCercle = Math.PI *Math.pow(radius, 2);
```

Variable de la classe  
java.lang.Math

Méthode de la classe  
java.lang.Math



15/09/99

© 1999 Peter Sander

204



## Classes et Objets Méthodes de Classe

- Mot clé **static** indique méthode de classe

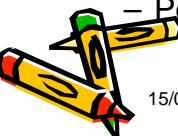
```
public class Boss {  
    ...  
    public static void main(String[] args) {  
        ...  
    }  
}
```

- **main** : point d'entrée de la JVM
- Pourquoi est-ce *obligatoirement static* ?

15/09/99

© 1999 Peter Sander

205



## Classes et Objets Méthodes de Classe

- JVM exécute **main()** de **Toto**

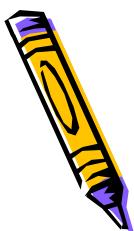
- Aucune instance de classe (objet) n'existe
  - aucune invocation de **new Toto()** nulle part
  - aucune variable d'instance n'existe encore
  - aucune méthode d'instance n'existe encore
- La classe, elle existe : la classe a été chargée
  - variables de classe accessibles
  - méthodes de classe accessibles
    - variables locales des ces méthodes utilisables



15/09/99

© 1999 Peter Sander

206



## Classes et Objets Méthodes de Classe

```
public class PME {  
    ...  
    public static void main(String[] args) {  
        String gerant = "personne";  
        ...  
        System.out.println("Notre gérant : "  
                           + gerant);  
    }  
}
```

Oui :

- aucune instance d'objet de type **PME**
- n'utilise que des variables locales



15/09/99

© 1999 Peter Sander

207



## Classes et Objets Méthodes de Classe

```
public class PME {  
    private static final double BASE = 14000;  
    public static double augmente(double prime) {  
        return BASE + prime;  
    }  
    public static void main(String[] args) {  
        System.out.println("Salaire augmenté : "  
                           + augmente(40000));  
    }  
}
```

– Légale ? Oui :

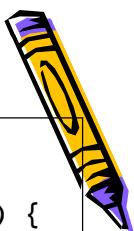
- aucune instance d'objet de type **PME**
- n'utilise que des méthodes et variables statiques (de classe)



15/09/99

© 1999 Peter Sander

208



# Classes et Objets

## Méthodes de Classe

```
public class PME {  
    private double salaire;  
    public static void main(String[] args) {  
        salaire = 36712.49;  
    }  
}
```

– Légale ?

Non :

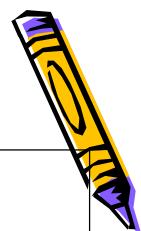
- **salaire** raccourcit **this.salaire**
- **this** désigne un objet de type **PME**
- aucun objet de type **PME** n'existe



15/09/99

© 1999 Peter Sander

209



# Classes et Objets

## Méthodes de Classe

```
public class PME {  
    private double salaire;  
    public static void main(String[] args) {  
        PME pme = new PME();  
        pme.salaire = 36712.49;  
    }  
}
```

– Légale ?

Oui :

- **pme** désigne un objet de type **PME**
- **salaire** est variable d'objet de type **PME**



© 1999 Peter Sander

210

# Classes et Objets

## Méthodes de Classe

```
public class PME {  
    private static final double BASE = 14000;  
    public double augmente(double prime) {  
        return BASE + prime;  
    }  
    public static void main(String[] args) {  
        System.out.println("Salaire augmenté : "  
                           + augmente(40000));  
    }  
}
```

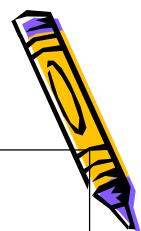
– Légale ? Non :

- **augmente()** raccourcit  
**this.augmente()**
- **this** désigne un objet de type **PME**
- aucun objet de type **PME** n'existe



15/09/99

211



# Classes et Objets

## Méthodes de Classe

```
public class PME {  
    private static final double BASE = 14000;  
    public double augmente(double prime) {  
        return BASE + prime;  
    }  
    public static void main(String[] args) {  
        PME pme = new PME();  
        System.out.println("Salaire augmenté : "  
                           + pme.augmente(40000));  
    }  
}
```

– Légale ? Oui :

- **pme** désigne un objet de type **PME**
- **augmente()** est méthode d'objet de type **PME**



© 1999 Peter Sander

212

