

Paradigmes de Programmation

Johan Brichau

Université catholique de Louvain
Ingénierie Informatique
www.info.ucl.ac.be/~jbrichau



Tom Mens

Université de Mons-Hainaut
Service de Génie Logiciel
w3.umh.ac.be/genlog



L'Orientation Objet

© Johan Brichau, Tom Mens, UMH, 2007

- vs. la programmation 'impérative' & 'fonctionnelle'
 - "Ask not what you can do to your datastructures but ask what your datastructures can do for you"
- Origines dans la simulation du monde réel (Simula): OBJECTS + MESSAGES
- Caractéristiques
 - Abstraction (+ encapsulation)
 - Polymorfisme
 - Delegation / Inheritance
 - Overriding
 - Liaison tardive (late binding)

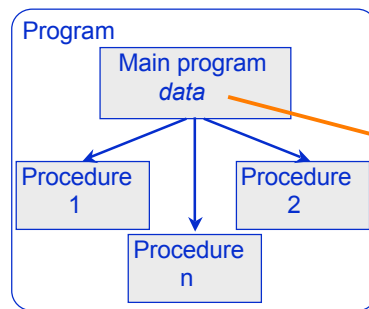
Polymorphisme
structuré

Abstraction

© Johan Bricchau, Tom Mens, UMH, 2007

3

- Cfr. Structures de données abstraites
- Dans les langages non-orienté objet
 - Principalement par convention



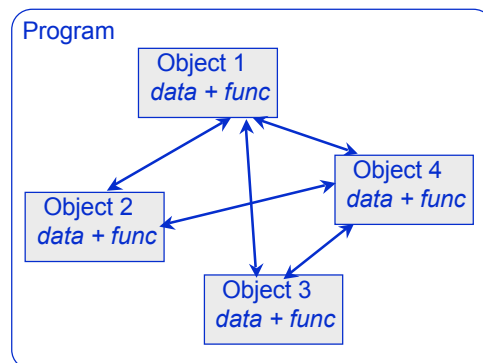
Donnés partagés:
Proc 1 peut être la cause
d'une faute dans proc 2

Encapsulation

© Johan Bricchau, Tom Mens, UMH, 2007

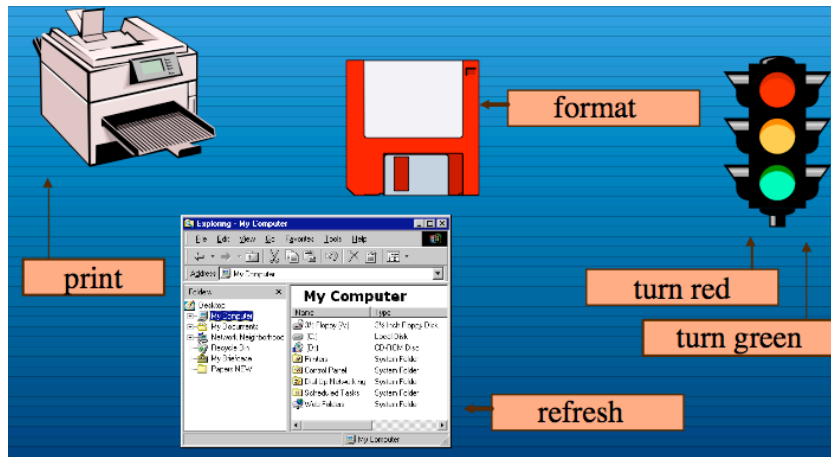
4

- Encapsulation
 - État (State)
 - Comportement (Behaviour)



OBJETS

Polymorfisme

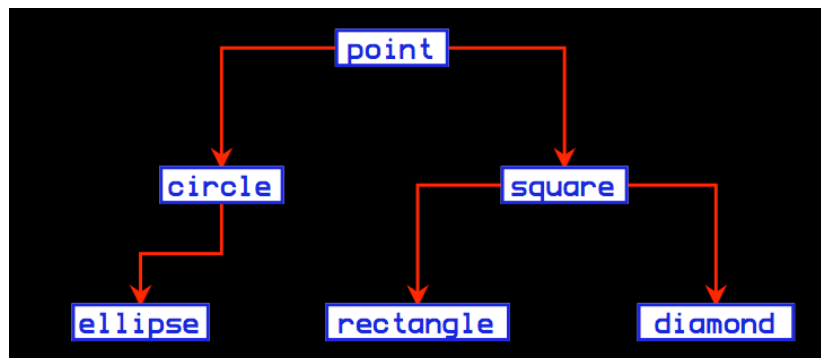


MESSAGES

© Johan Brichau, Tom Mens, UMH, 2007

5

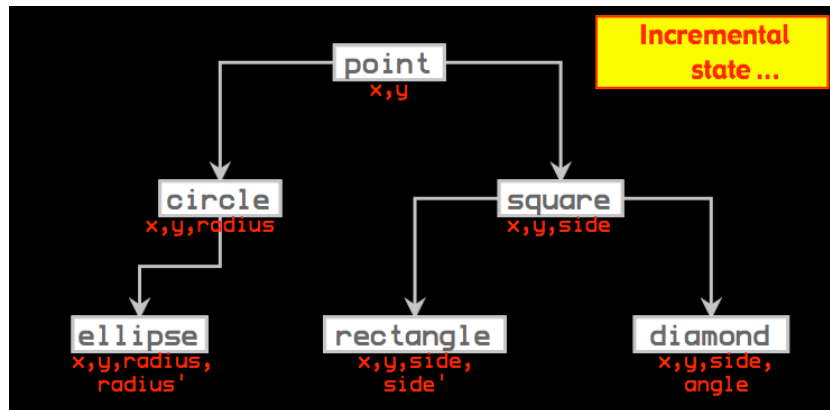
Delegation / Inheritance



© Johan Brichau, Tom Mens, UMH, 2007

6

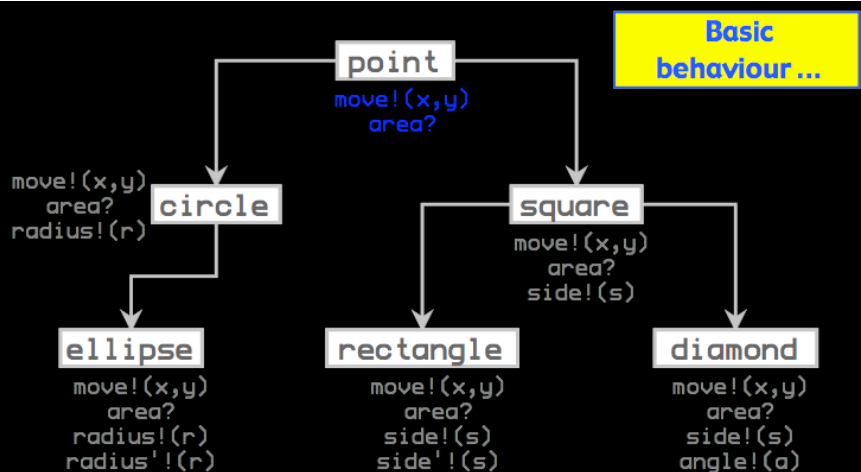
Delegation / Inheritance



© Johan Brichau, Tom Mens, UMH, 2007

7

Delegation / Inheritance / Overriding

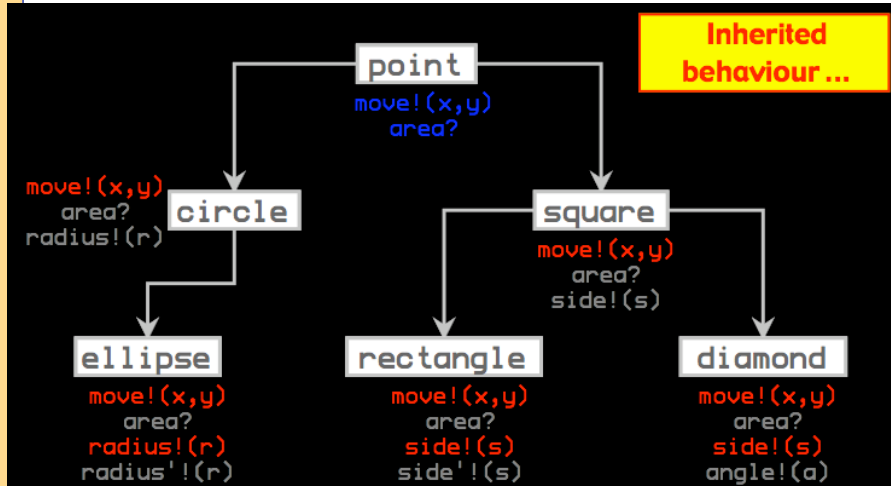


© Johan Brichau, Tom Mens, UMH, 2007

8

Delegation / Inheritance / Overriding

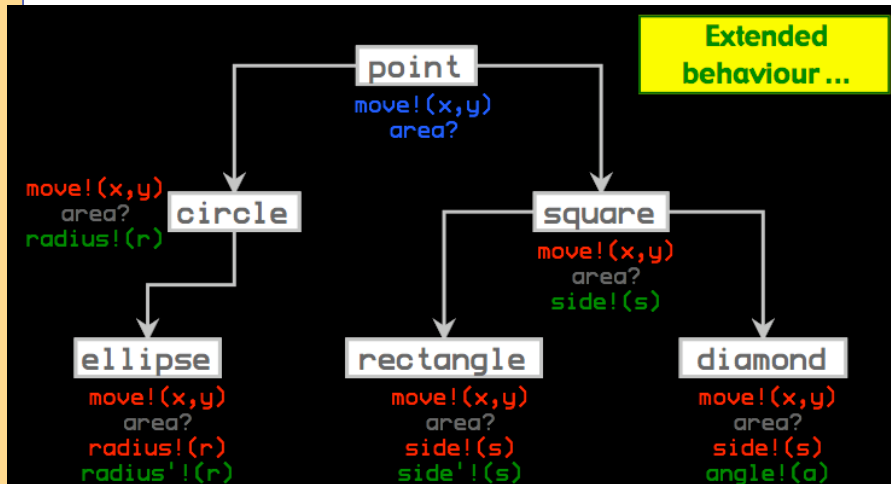
© Johan Brichau, Tom Mens, UMH, 2007



9

Delegation / Inheritance / Overriding

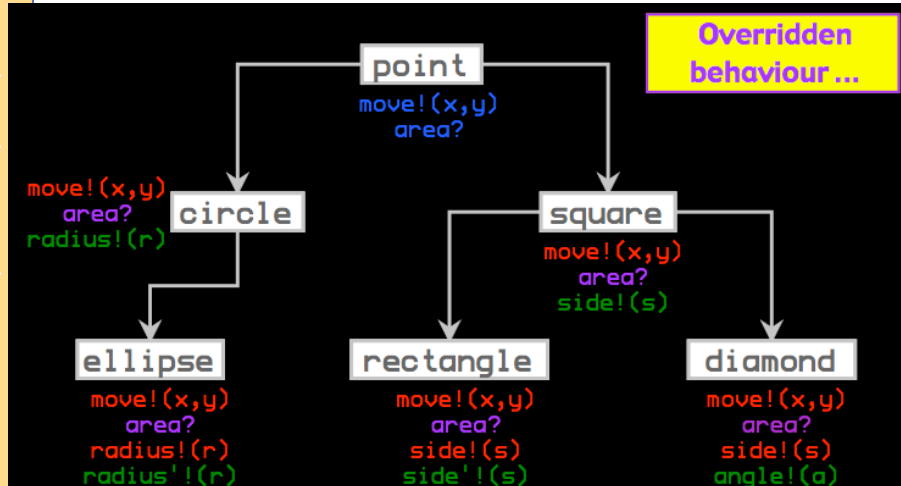
© Johan Brichau, Tom Mens, UMH, 2007



10

Delegation / Inheritance / Overriding

© Johan Bricchau, Tom Mens, UMH, 2007



11

Liaison Tardive

© Johan Bricchau, Tom Mens, UMH, 2007

- La liaison entre un message et son implementation n'est déterminé que pendant l'exécution
 - Essentiel pour le polymorfisme !

12

Scheme

la programmation orientée objet

programmation orientée objet Exemple

- Nous voulons créer un compte bancaire
 - En utilisant les notions orientées objets

BankAccount	← classe
- balance	← variables
+ deposit(amount)	
+ withdraw(amount)	
+ status()	← méthodes

- Problème
 - Scheme n'est pas un langage orienté objet

© Johan Brichau, Tom Mens, UMH, 2007

14

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

15

- Question
 - Comment pouvons nous simuler les classes et les objets en *Scheme*?
 - Quels sont les mécanismes que nous pouvons utiliser pour faire cela?
- Réponse
 - Essayons de trouver une solution pas à pas

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

16

- Retraits successifs

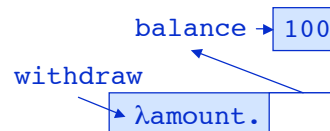
```
> (define balance 100)
balance

> (define (withdraw amount)
  (if (>= balance amount)
      (begin
        (set! balance (- balance amount))
        balance)
      "insufficient funds"))

withdraw

> (withdraw 20)
80
> (withdraw 10)
70
```

Désavantage: `balance` est une variable locale



Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

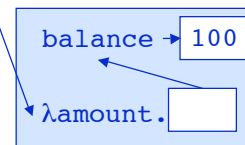
• Retraits successifs avec définition locale

```
> (define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin
            (set! balance (- balance amount))
            balance)
          "insufficient funds"))))
new-withdraw
```

```
> (new-withdraw 20)
80
> (new-withdraw 10)
70
```

Désavantage: la valeur initiale de balance est fixée

new-withdraw



17

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

• Retraits successifs avec initialisation

```
> (define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "insufficient funds")))
```

make-withdraw

```
> (w1 20)
30
> (define w1 (make-withdraw 50))
w1
> (define w2 (make-withdraw 100))
w2
> (w2 50)
50
> (w1 10)
20
```

18

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

- création d'une classe
 - une classe est implémentée par un "dispatch table"

```
(define (number-class x)
  (lambda (message)
    (cond
      ((eq? message 'show) (display x))
      ((eq? message 'inc) (set! x (+ x 1)))
      ((eq? message 'dec) (set! x (- x 1)))
    )))
```

19

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

- création d'un objet à partir de cette classe

```
> (define n (number-class 5))
```

- envoi de messages

```
> (n 'show)
5
> (n 'inc)
> (n 'show)
6
> (n 'dec)
> (n 'show)
5
```

20

Scheme programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

- création d'une classe
 - les méthodes + et - ont un argument

```
(define (number-class x)
  (lambda (message)
    (cond
      ((eq? message 'show) (display x))
      ((eq? message '+) (lambda (arg)
                          (set! x (+ x arg))))
      ((eq? message '-') (lambda (arg)
                          (set! x (- x arg))))
    )))
```

21

Scheme programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

- création d'un objet à partir de cette classe

```
> (define n (number-class 5))
```

- envoi de messages

```
> (n 'show)
5
> (n '+ 5)
procedure expects 1 argument, given 2: + 5
> ((n '+) 5)
> (n 'show)
10
> ((n '-') 4)
> (n 'show)
6
```

22

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

- création d'une classe: simplification

```
(define (number-class x)
  (lambda (message . args)
    (cond
      ((eq? message 'show)
       (display x))
      ((eq? message '+)
       (set! x (+ x (car args))))
      ((eq? message '-')
       (set! x (- x (car args))))
    )))

> (define n (number-class 5))
> (n 'show)
5
> (n '+ 5)
> (n 'show)
10
> (n '- 4)
> (n 'show)
6
```

23

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

- Exemple: Un compte bancaire

```
(define (new-account balance)
  (lambda (msg)
    (cond
      ((eq? msg 'withdraw) (lambda (amount)
                             (if (>= balance amount)
                                 (begin
                                   (set! balance (- balance amount))
                                   balance)
                                 "insufficient funds"))
      ((eq? msg 'deposit) (lambda (amount)
                            (set! balance (+ balance amount))
                            balance))
      ((eq? msg 'status) (lambda ()
                           (display balance)
                           balance))
      (else (error "message" msg "not understood")))
    )))
```

24

Scheme programmation orientée objet

Introduction des fonctions auxiliaires (c.à.d. méthodes privées)

```
(define (new-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (status)
    (display balance)
    balance)
  (lambda (m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'status) status)
          (else (error "message" m "not understood"))
    )))
))
```

© Johan Brichau, Tom Mens, UMH, 2007

25

Scheme programmation orientée objet

Introduction des fonctions auxiliaires

```
(define (new-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (status)
    (display balance)
    balance)
  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'status) status)
          (else (error "message" m "not understood"))
    )))
  dispatch)
```

© Johan Brichau, Tom Mens, UMH, 2007

26

Scheme programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

27

• Exemple: Un compte bancaire

```
> (define acc (new-account 100))
acc

> acc
#<PROCEDURE dispatch>

> ((acc 'withdraw) 20)
80
> ((acc 'deposit) 50)
130
> ((acc 'status))
130
> (acc 'something)
message something "not understood"
```

Java programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

28

• Même exemple en Java

```
public class Account {
    private double balance; // variable

    public Account(double amount) { // constructor
        balance = amount; }
    public double status() { // method
        System.out.println(balance);
        return balance; }
    public double deposit(double amount) { // method
        balance = balance + amount;
        return balance; }
    public double withdraw(double amount) { // method
        if (balance >= amount)
            { balance = balance - amount; }
        else
            { System.out.println("insufficient funds"); }
        return balance;
    }
}
```

Java programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

· Même exemple en Java

```
Account acc = new Account(100);  
System.out.println(acc);  
acc.withdraw(20);  
acc.deposit(50);  
acc.status();
```

```
Account@5ac072  
130.0
```

29

Scheme programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

· faciliter l'envoi des messages - introduction du syntaxe `send`

```
> (define (send object msg)  
  (if (null? (cdr msg))  
      ((object (car msg))  
         ((object (car msg)) (cadr msg))))  
  send
```

```
> (send acc '(withdraw 30))  
90
```

```
> (send acc '(deposit 10))  
100
```

```
> (send acc '(status))  
100
```

30

Scheme programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

• Deux comptes bancaires

```
> (define tom-acc (new-account 1000))
> (define pierre-acc (new-account 1000))

> (send tom-acc '(withdraw 100))
900
> (send tom-acc 'status)
900
> (send pierre-acc 'status)
1000
```

31

Java programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

• Deux comptes bancaires - Même exemple en Java

```
Account tomAcc = new Account(1000);
Account pierreAcc = new Account(1000);
tomAcc.withdraw(100);
tomAcc.status();
pierreAcc.status();

900.0
1000.0
```

32

Scheme programmation orientée objet

- Un compte bancaire partagé (« sharing »)

```
> (define tom-acc (make-account 1000))
> (define pierre-acc tom-acc)

> (send tom-acc '(withdraw 100))
900
> (send tom-acc 'status)
900
> (send pierre-acc 'status)
900
```

© Johan Bricchau, Tom Mens, UMH, 2007

33

Java programmation orienté objet

- Un compte bancaire partagée
- Même exemple en Java

```
Account tomAcc = new Account(1000);
Account pierreAcc = tomAcc;
tomAcc.withdraw(100);
tomAcc.status();
pierreAcc.status();

900.0
900.0
```

© Johan Bricchau, Tom Mens, UMH, 2007

34

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

- Faciliter l'envoi de messages
- simplification du syntaxe `send`

```
> (define account1 (new-account 20))
> (send account1 'withdraw 10)
10
> (send account1 'status)
10
> (send account1 'deposit 5)
15
```

- Tester le nombre d'arguments d'un message
`error: unknown request withdraw or wrong number of arguments 0`

35

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

```
(define (new-account amount)
  (define (withdraw args)
    (set! amount (- amount (car args)))
    amount)
  (define (deposit args)
    (set! amount (+ amount (car args)))
    amount)
  (define (status args)
    (display amount))
  (define (dispatch m noOfArgs)
    (define (checkSignature message number)
      (and (eq? m message) (= noOfArgs number)))
    (cond
      ((checkSignature 'withdraw 1) withdraw)
      ((checkSignature 'deposit 1) deposit)
      ((checkSignature 'status 0) status)
      (else (error "message" m "not understood"
                  "or wrong number of arguments" noOfArgs)
            ))
    ))
  dispatch)

(define (send object msg . args)
  ((object msg (length args)) args))
```

36

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

- Problème avec cette solution
 - Elle ne respecte pas le principe de encapsulation (masquage d'information)
 - Nous pouvons toujours accéder à l'implémentation des méthodes

```
> (define acc (new-account 100))  
> (acc 'withdraw 1)  
#<procedure:withdraw>
```

37

Scheme programmation orientée objet

© Johan Brichau, Tom Mens, UMH, 2007

- Solution

```
(define (new-account amount)  
  (define (withdraw args)  
    (set! amount (- amount (car args)))  
    amount)  
  (define (deposit args)  
    (set! amount (+ amount (car args)))  
    amount)  
  (define (status args)  
    (display amount))  
  (define (dispatch m args)  
    (define (checkSignature message number)  
      (and (eq? m message) (= (length args) number)))  
    (cond  
      ((checkSignature 'withdraw 1) (withdraw args))  
      ((checkSignature 'deposit 1) (deposit args))  
      ((checkSignature 'status 0) (status args))  
      (else (error "message" m "not understood"  
                  "or wrong number of arguments" args))))  
  dispatch)
```

38

Scheme programmation orientée objet

© Johan Brichau, Toim Mens, UMH, 2007

- Question
 - Quelle est la différence entre notre implémentation de classes en Scheme et celle en Java?
- Réponse
 - En Scheme, avec un "dispatch table", chaque objet contient ses propres méthodes
 - En Java, les méthodes sont spécifiées dans la classe, et partagées par toutes ses instances
- Solution
 - C'est aussi possible en Scheme, mais c'est un peu plus compliqué

39

Classes et Objets

package "class.ss" en Scheme

Scheme

programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

- "class.ss" est une module qui facilite la programmation orientée objet
 - en introduisant du sucre syntaxique
- Fait partie de MzScheme
 - une extension orientée objet de Scheme

41

Scheme

classes et méthodes

© Johan Bricchau, Tom Mens, UMH, 2007

- *class syntax*
(class superclass-expr
class-clause
...)
- *where class-clause is one of*
(init init-declaration ...)
(init-field init-declaration ...)
(init-rest variable)
(init-rest)
(field field-declaration ...)
(public optionally-renamed-
variable ...)
(public-final optionally-
renamed-variable ...)
(private variable ...)
- *where class-clause is one of*
method-definition
definition
expr
(begin class-clause ...)
(rename renamed-variable ...)
(override optionally-renamed-
variable ...)
(override-final optionally-
renamed-variable ...)
(inherit optionally-renamed-
variable ...)
(inherit-field optionally-
renamed-variable ...)

42

Java programmation orientée objet

© Johan Bricchau, Tom Mens, UMH, 2007

- Exemple: un compte bancaire en Java

```
public class Account {
    private double balance; // variable

    public Account(double amount) { // constructor
        balance = amount; }
    public double status() { // method
        System.out.println(balance);
        return balance; }
    public double deposit(double amount) { // method
        balance = balance + amount;
        return balance; }
    public double withdraw(double amount) { // method
        if (balance >= amount)
            { balance = balance - amount; }
        else
            { System.out.println("insufficient funds"); }
        return balance;
    }
}
```

43

Scheme classes et méthodes

© Johan Bricchau, Tom Mens, UMH, 2007

- Exemple: un compte bancaire en MzScheme

```
(define account%
  (class object%
    ; Declare public methods:
    (public withdraw! deposit! status?)

    ; Public variables
    (init-field (balance 0))

    ; Public method implementations:
    (define (deposit! v)
      (set! balance (+ balance v))
      (status?))
    (define (withdraw! v)
      (set! balance (- balance v))
      (status?))
    (define (status?) (display balance) (newline))

    ; Call superclass initializer:
    (super-new)))
```

44

Scheme objets et messages

© Johan Bricchau, Tom Mens, UMH, 2007

45

- Creation d'un objet

```
(define account1 (make-object account%))
(define account2 (new account% (balance 100)))
```
- Envoi de messages

```
(send account1 status?)
0
(send account1 deposit! 100)
100
(send account1 withdraw! 30)
70

(send account2 status?)
100
(send account2 deposit! 100)
200
(send account2 withdraw! 50)
150
```

Scheme classes et méthodes

© Johan Bricchau, Tom Mens, UMH, 2007

46

- Exemple: une pile

en utilisant une liste comme
structure de données

```
(define stack%
  (class object%
    ; Declare public methods:
    (public push! pop! none? print-name)

    (define stack null) ; A private field
    (init-field (name 'stack)) ; A public field

    ; Method implementations:
    (define (push! v) (set! stack (cons v stack)))
    (define (pop!)
      (let ((v (car stack)))
        (set! stack (cdr stack))
        v))
    (define (none?) (null? stack))
    (define (print-name) (display name) (newline))

    ; Call superclass initializer:
    (super-new)))
```

Scheme objets et messages

© Johan Bricchau, Tom Mens, UMH, 2007

- Creation d'un objet

```
(define stack1 (make-object stack%))
(define stack2 (new stack% (name 'Fred)))
```
- Envoi de messages

```
> (send stack1 push! 1)
> (send stack1 none?)
#f
> (send stack1 pop!)
1
> (send stack1 none?)
#t
> (send stack1 print-name)
stack
> (send stack2 print-name)
Fred
```

47

Java classes et méthodes

© Johan Bricchau, Tom Mens, UMH, 2007

- Exemple: une pile

```
import java.util.*;
public class Stack {
    private LinkedList stack; // variable
    public String name = "stack";
    Stack() { // constructor
        stack = new LinkedList(); }
    Stack(String n) { // constructor
        this();
        name = n; }
    public void push(Double v) { // method
        stack.add(v); }
    public Double pop() {
        Double d = (Double)(stack.getFirst());
        stack.removeFirst();
        return d; }
    public boolean none() {
        return stack.isEmpty(); }
    public void printName() {
        System.out.println();
        System.out.println(name); }
}
```

48

Java objets et messages

© Johan Bricchau, Tom Mens, UMH, 2007

- Creation d'un objet

```
Stack stack1 = new Stack();  
Stack stack2 = new Stack("Fred");
```

- Envoi de messages

```
stack1.push(new Double(1.0));  
System.out.println(stack1.none());  
System.out.println(stack1.pop());  
System.out.println(stack1.none());  
stack1.printName();  
stack2.printName();
```

- Résultats

```
false  
1.0  
true  
stack  
Fred
```

49

Scheme héritage et extension

© Johan Bricchau, Tom Mens, UMH, 2007

- Creation d'une sous-classe

```
(define double-stack%  
  (class stack%  
    (inherit push!)  
  
    (public double-push!)  
    (define (double-push! v) (push! v) (push! v))  
  
    ; Always supply name  
    (super-new (name 'double-stack))))
```

- Envoi de messages

```
> (define stack2 (make-object double-stack%))  
> (send stack2 print-name)  
double-stack  
> (send stack2 double-push! 1)  
> (send stack2 pop!)  
1  
> (send stack2 pop!)  
1
```

50

Java héritage et extension

© Johan Bricchau, Tom Mens, UMH, 2007

51

- **Creation d'une sous-classe**

```
public class DoubleStack extends Stack {
    DoubleStack() {
        super("double stack");
    }
    public void doublePush(Double v) {
        this.push(v);
        this.push(v);
    }
}
```
- **Envoi de messages**

```
DoubleStack stack2 = new DoubleStack();
stack2.printName();
stack2.doublePush(new Double(1.0));
System.out.println(stack2.pop());
System.out.println(stack2.pop());
```
- **Résultats**

```
double-stack
1.0
1.0
```

Scheme héritage et redéfinition

© Johan Bricchau, Tom Mens, UMH, 2007

52

- **Creation d'une sous-classe**

```
(define extended-stack%
  (class stack%
    ; Declare method overriding
    (override print-name)

    ; Add inherited field to local environment
    (inherit-field name)

    (define (print-name)
      (display "The name of the stack is: ")
      (display name)
      (newline))

    (super-new)))
```
- **Envoi de messages**

```
> (define stack3 (new extended-stack% (name 'Tom)))
> (send stack3 print-name)
The name of the stack is: tom
```

Java

héritage et redéfinition

© Johan Bricchau, Tom Mens, UMH, 2007

- Creation d'une sous-classe en Java

```
public class ExtendedStack extends Stack {
  public ExtendedStack() { // constructor
    super(); }
  public ExtendedStack(String n) { // constructor
    super(n); }
  public void printName() { // redefined method
    System.out.println("The name of the stack is: " + name);
  }
}
```

- Envoi de messages

```
ExtendedStack stack3 = new ExtendedStack("Tom");
stack3.printName();
```

- Résultats

```
The name of the stack is: tom
```

53

Scheme

Classes as first-class values

© Johan Bricchau, Tom Mens, UMH, 2007

- make-safe-stack-class est une fonction qui peut être appliquée à une classe et qui renvoie une nouvelle classe

```
(define-values (make-safe-stack-class is-safe-stack?)
  (let ((safe-stack<%> (interface (stack<%>)))
        (values
         (lambda (super%)
           (class* super% (safe-stack<%>)
             (inherit none?)
             (rename [std-pop! pop!])
             (override pop!)
             (define (pop!) (if (none?) #f (std-pop!)))
             (super-new)))
         (lambda (obj)
           (is-a? obj safe-stack<%>))))))

(define safe-stack% (make-safe-stack-class stack%))
```

54

Scheme Classes as first-class values

© Johan Brichau, Tom Mens, UMH, 2007

- Exemple de l'utilisation de safe-stack

```
(define stack1 (make-object stack%))
> (is-safe-stack? stack1)
#f
> (send stack1 pop!)
car: expects argument of type <pair>; given ()

(define safe-stack1 (new safe-stack% (name 'safe)))
> (is-safe-stack? safe-stack1)
#t
> (send safe-stack1 pop!)
#f
> (send safe-stack1 print-name)
safe
```

55

Scheme Classes as first-class values

© Johan Brichau, Tom Mens, UMH, 2007

- make-safe-stack-class peut également être appliquée dynamiquement à des sous-classes de stack%
- on appelle ça un "mix-in"

```
(define safe-extended% (make-safe-stack-class
  extended-stack%))
(define safe-stack3 (new safe-extended% (name 'safe)))

> (send safe-stack3 print-name)
The name of the stack is: safe
> (is-safe-stack? safe-stack3)
#t
> (send safe-stack3 pop!)
#f
```

56

Classes as first-class values

- En Java
 - les classes ne sont pas "first class"
 - Une classe ne peut pas être créée ou étendue dynamiquement
 - les classes sont créées lors de la phase de compilation
 - pendant la phase d'exécution on peut créer les objets (instance d'une classe), mais pas les classes