Introduction à la programmation avec Java

UFR Sciences de Nice

Licence Math-Info 2006-2007

Module L1I1

Frédéric MALLET

Jean-Paul ROY

Structure d'une classe

Les méthodes

Où en sommes-nous ?

• Les méthodes permettent d'envoyer des **messages** aux objets :

```
triangle3.changeTaille(20,14) ;
circle1.changeCouleur(″yellow″) ;
circle1.bougeVertical() ;
```

• Une méthode peut recevoir des **paramètres typés**.
• La **signature d'une méthode** indique le type des paramètres qu'elle attend et le type de valeur qu'elle renvoie :

```
void changeTaille(int hauteur, int largeur)
void changeCouleur(String couleur)
void bougeVertical()
```

Quel rapport avec les maths ?

• Les **méthodes** ne sont (au fond) que des **fonctions** appliquées sur un contexte implicite (l'état de l'objet) :

```
void changeTaille(int hauteur, int largeur)

     changeTaille  : Triangle . int x int → ∅
```

```
void changeCouleur(String couleur)

     changeCouleur  : Cercle . String → ∅
```

```
void bougeVertical()

     bougeVertical  : Cercle . ∅ → ∅
```

Des fonctions sans résultat ?…

- D'abord il faut passer par un objet pour activer une telle fonction (envoi de message) !
- Ensuite, pourquoi diable une fonction $E \rightarrow \varnothing$?
- Oui, l'informatique utilise ce genre de fonction, rare en maths ! Cet $\varnothing$ signifie qu'elle ne renvoie **aucun résultat** !

`cercle1`, change immédiatement ta couleur en jaune !

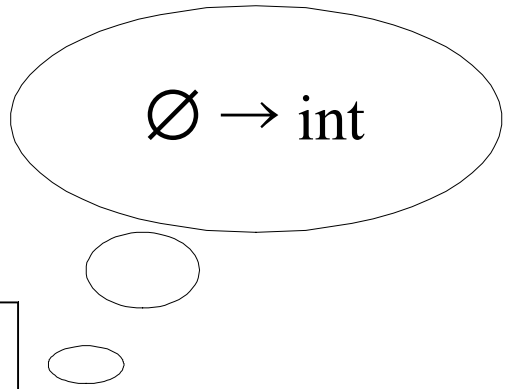
`void`

Le cercle obéit, ***modifie son état***, mais ne répond pas !

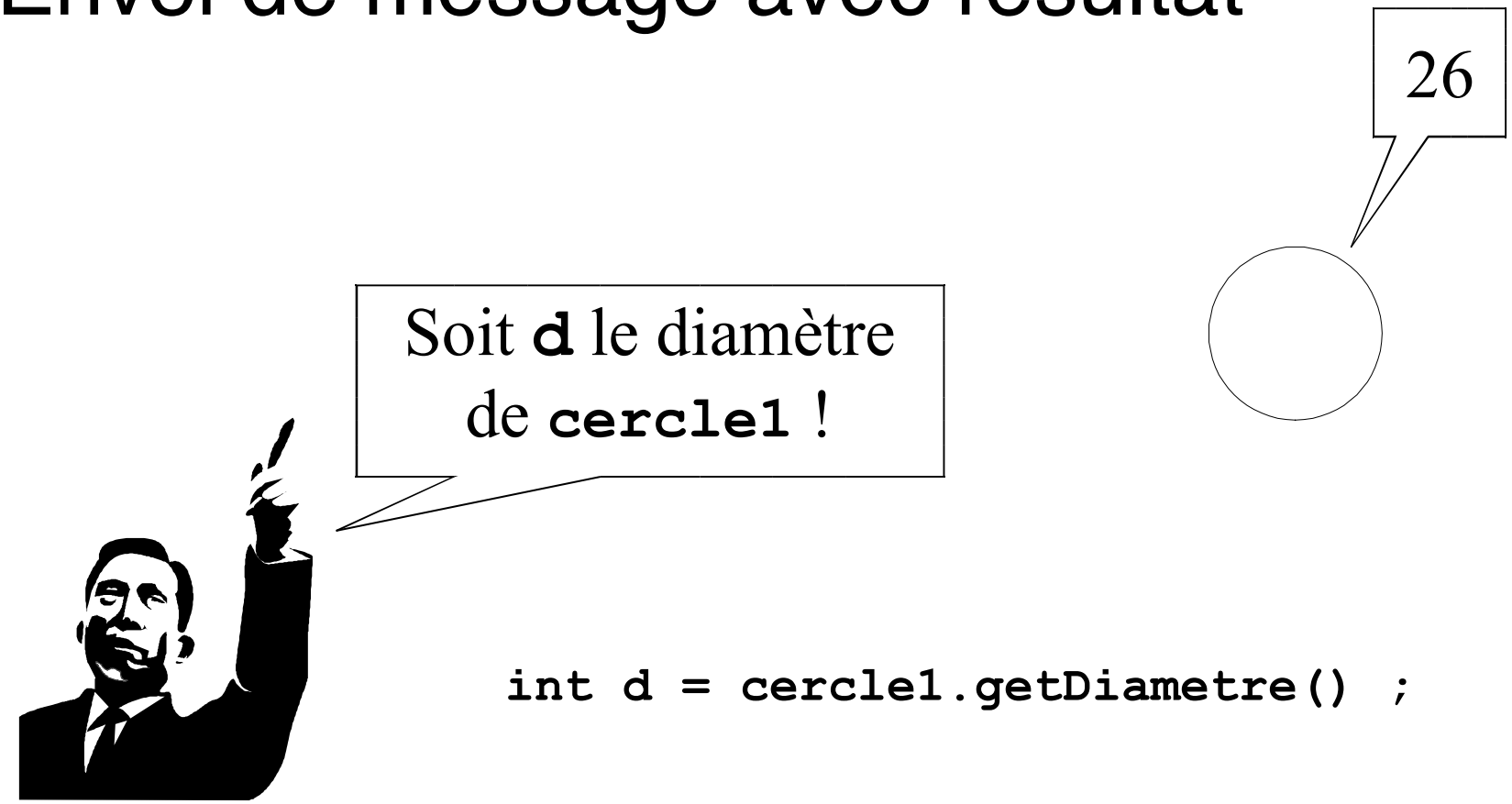
Une méthode peut rendre un résultat !

- Ex: rajoutons à la classe **`Cercle`** une méthode qui permettra de demander à un cercle de nous donner son diamètre (qui est l'un de ses champs, de type entier) :

```
class Cercle {

    int diametre;

     ......

    int getDiametre() {

      return this.diametre;

    }

    ......

}
```



Envoi de message avec résultat




```
int d = cercle1.getDiametre() ;
```

- **`getDiametre()`** est une méthode d'accès : un **accesseur**.
- Elle doit parler à un objet, ici **`cercle1`**
- Le nom des accesseurs commence en général par **`get`**

Une mini classe Calculette

```
class Calculette {

  int screen;

  void clear() {
   this.screen = 0 ;
  }

  void addToScreen(int n) {
   this.screen = this.screen + n;
  }

  int getScreen() {
    return this.screen;
  }

}
```

1

2

3

1 Bloc de déclaration de classe = l'en-tête de la Classe.

2 Les champs d'un objet de la classe : ils représentent l'état de l'objet.

3 Les blocs de déclaration de méthodes. Elles décrivent les messages auxquels un objet de la classe saura répondre, en rendant ou non une valeur de retour.

RETENEZ BIEN CETTE ARCHITECTURE DE CLASSE !

NB :

- le début des blocs de déclaration est marqué par {
- les blocs terminent par }
- Un en-tête précède l'accolade de début (classe, méthode, etc.)

Les différentes variables de cette classe

- La classe **`Calculette`** comporte deux sortes de variables :
  - **`screen`** est le nom d'un champ : **variable d'instance**.
    `screen` est initialisé à zéro par défaut (car de type int)
  - **`n`** est un **paramètre** de la méthode **`addToScreen`**.

- Toutes les instances d'une classe possèdent le même jeu de variables d'instance mais avec des valeurs distinctes propres à chaque instance. Elles représentent **l'état de l'objet**.

- Les variables paramètres représentent de l'information supplémentaire qui existe seulement lors de l'envoi d'un message à un objet.

```
calc1.addToScreen(15);
```

Les expressions

· Les **expressions** représentent une valeur typée :

  · `0` est une expression constante (de type **`int`**)

  · **`2 + 3 - 6`** également : elle représente la valeur -1

  · **`this.screen + n`** est une expression variable, sa valeur dépend du champ **`screen`** et du paramètre **`n`**

  · **`cercle1.getDiametre() * 2`** est une expression ; son évaluation consiste en l'appel de la méthode **`getDiametre`** sur l'objet **`cercle1`** suivie d'une multiplication par 2.

  · **`cercle1.changeCouleur("jaune")`** n'est pas une expression puisque son résultat est **`void`** !

La priorité des opérateurs

- Certaines expressions sont ambiguës :
  `2 + 4 * 5` vaut `(2 + 4) * 5` ou `2 + (4 * 5)`
- Les parenthèses permettent toujours de lever l'ambiguïté.
- Les **règles de priorité** (subjectives) aussi :

Les plus prioritaires	`* / %`
Les moins prioritaires	`- +`

- Si même priorité, **dans l'ordre du texte (gauche à droite)**
  `2 + 4 * 5` vaut donc `22` (d'abord `*`, puis `+`)
  `2 + 4 - 5` vaut donc `1` (d'abord `+`, puis `-`)

Modification de variable : l'opérateur =

Ex:
```
this.screen = 0;
```

La valeur de la variable **`this.screen`** **devient égale à** `0`.

Ex:
```
this.screen = this.screen + n;
```

La valeur de la variable **`this.screen`** **devient égale à** l'ancienne valeur de la variable **`this.screen`** augmentée de la valeur de la variable **`n`** .

Abréviation :
```
this.screen += n;
```

```
<variable> = <expression>;
```

L'instruction d'affectation

- **L'instruction d'affectation** permet ici de modifier la valeur d'un champ : `this.screen = 0;`
  - La nouvelle valeur est représentée par une expression.
  - Les instructions se terminent par un point-virgule ;
  - Le type de l'expression (membre de droite) DOIT être compatible avec le type de la variable affectée (à gauche) : **`this.screen + n`** est de type **`int`** , car **`int + int`** → **`int`**
- L'opérateur **`+=`** est l'**opérateur d'affectation combiné** :
  - **`this.screen += n`** est équivalent à : **`this.screen = this.screen + n`** avec les types **`int`** et **`double`**.

Ne confondez pas les signes = et ==

- Le signe **=** a bien deux sens en mathématique :

Posons x = y+1. Alors…	C'est le signe **=** de Java : **l'affectation**
Si x = y+1, alors…	C'est le signe **==** de Java : **le test d'égalité**

```
if (x == 0) {
  y = x;
} else {
  y = x + 1;
}
```

```
// si x est égal à 0,
// alors y devient égal à x
// sinon
// y devient égal à x + 1
```

Portée et durée de vie d'une variable

• La **portée** (visibilité) d'une variable est la partie du code source d'où l'on peut accéder à sa valeur.

  • La portée d'un champ est la totalité du bloc de la classe.

  • La portée d'un paramètre est son bloc de déclaration (e.g. méthode, etc.)

• La **durée de vie** d'une variable correspond à la période pendant laquelle elle subsiste avant sa destruction.

  • Un champ a la même durée de vie que l'objet auquel il appartient.

  • Un paramètre a une durée de vie limitée au seul appel de son bloc (durée de l'appel de la méthode).

Etude d'une classe : `TicketMachine`

- Vous avez le source de la classe sous les yeux !
- Modélisation naïve d'un distributeur de billets de train à prix unique.
- Les clients insèrent de l'argent et demandent l'impression d'un ticket.
- On ne vérifie pas que le client a inséré assez d'argent pour avoir son ticket !

```
class TicketMachine
{
    <variables d'instance ?>

    ......
```

- Une machine à tickets doit connaître :
  - le prix d'un billet : **`price`**
  - la somme insérée par un client jusqu'à présent : **`balance`**
  - le total collecté par cette billetterie : **`total`**

- Donc trois champs pour chaque machine à tickets :

```
class TicketMachine {
    int price;
    int balance;
    int total;

    <le constructeur ?>

     ......
```

Initialisation des champs : le constructeur

- **`price`**, **`balance`**, **`total`** sont de type **`int`**. La valeur initiale par défaut d'un champ numérique est **`0`** :
  - Or on veut pouvoir choisir la valeur initiale de **`price`**
  - et cette valeur peut être différente pour chaque machine.
- Le **constructeur** permet de choisir ou de calculer la valeur initiale des champs d'un objet :
  - le constructeur peut avoir des paramètres.
  - le constructeur est appelé (précédé du mot clé **`new`**) lors de la construction d'un objet.
    - le constructeur n'est pas une méthode !

• Le **constructeur** d'une classe permet (en liaison avec **`new`**) de créer une nouvelle instance de cette classe et d'initialiser ses champs :

```
class TicketMachine {

    int price;
    int balance;
    int total;

    TicketMachine(int ticketCost) {
       this.price = ticketCost;
       this.balance = 0;
       this.total = 0;
    }

    <les méthodes>
    ......
```

• Soit **`TicketMa1`** une nouvelle machine à tickets distribuant des billets à 6 euros :

```
TicketMachine ticketMa1 = new TicketMachine(6);
```

Appel du constructeur : TicketMachine(int), ticketCost vaut 6

• Il faut maintenant décrire les messages susceptibles d'être envoyés à une machine donnée : ses méthodes. Ici quatre :

· connaître le prix d'un billet de cette machine

`getPrice() : int`

· connaître la somme déjà introduite par le client

`getBalance() : int`

· connaître le contenu total de la machine

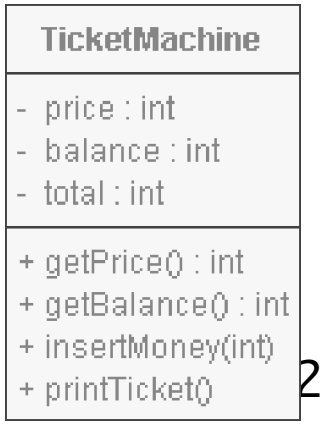
?

· insérer une somme d'argent

`insertMoney(int) : void`

· imprimer le billet

`printTicket() : void`



Les méthodes « accesseurs »

- Fonctions permettant d'accéder aux champs d'un objet :

```
int getPrice() {
  return price;
}
```

⇔ `return this.price;`

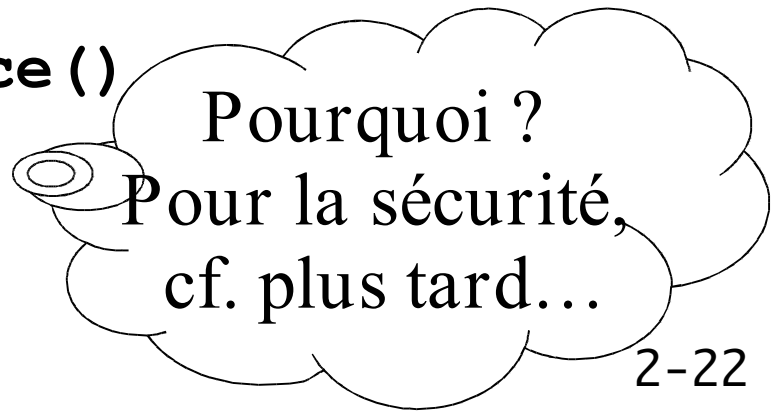
```
int getBalance() {
  return balance;
}
```

⇔ `return this.balance;`

- Il est considéré comme sain de passer par des sélecteurs pour accéder à la valeur du champ d'un objet plutôt que d'accéder directement à cette valeur avec la notation pointée :

Très bien → `ticketMa1.getPrice()`

Bien → `ticketMa1.price`



Les méthodes « modificateurs »

- Elles permettent de modifier les champs d'un objet :

```
void insertMoney(int amount) {
   balance = balance + amount;
}
```

Pas d'ambiguïté sur le nom balance :

```
this.balance = this.balance + amount;
```

N.B.

a) Aucun résultat (**`void`**), seulement une action : modifier la valeur du champ **`balance`** de l'objet courant qui est **`this`**.

b) Autre écriture : **`balance += amount;`**

RETENIR : Une fonction à valeur **`void`** fait une action, modifie quelque chose, mais n'a aucun résultat !

Les méthodes mixtes

• Elles font plusieurs choses, avec ou sans résultat.
Ici imprimer un billet et remettre le solde client à zéro :

```
void printTicket() {
    System.out.println(”############”);
    System.out.println(”La ligne Valrose”);
    System.out.print(”BILLET ”);
    System.out.println(”(” + price + ” euros)”);
    System.out.println(”############”);
    this.total += this.balance;
    this.balance = 0;
}
```

Remarque : l’ordre des instructions est important !

• ***System.out*** est un objet : l’organe de sortie.

void println(String texte)

- La méthode **`println()`** invoquée sur l'objet **`System.out`** affiche un texte (type **`String`**)

  Ex: *`System.out.println("Bonjour");`* affiche Bonjour

- Pour afficher une variable on la **concatène** (avec l'opérateur +) à une chaîne de caractères

  Ex: *`System.out.println(this.price + " euros");`*

  `this.price` est remplacé par sa valeur.

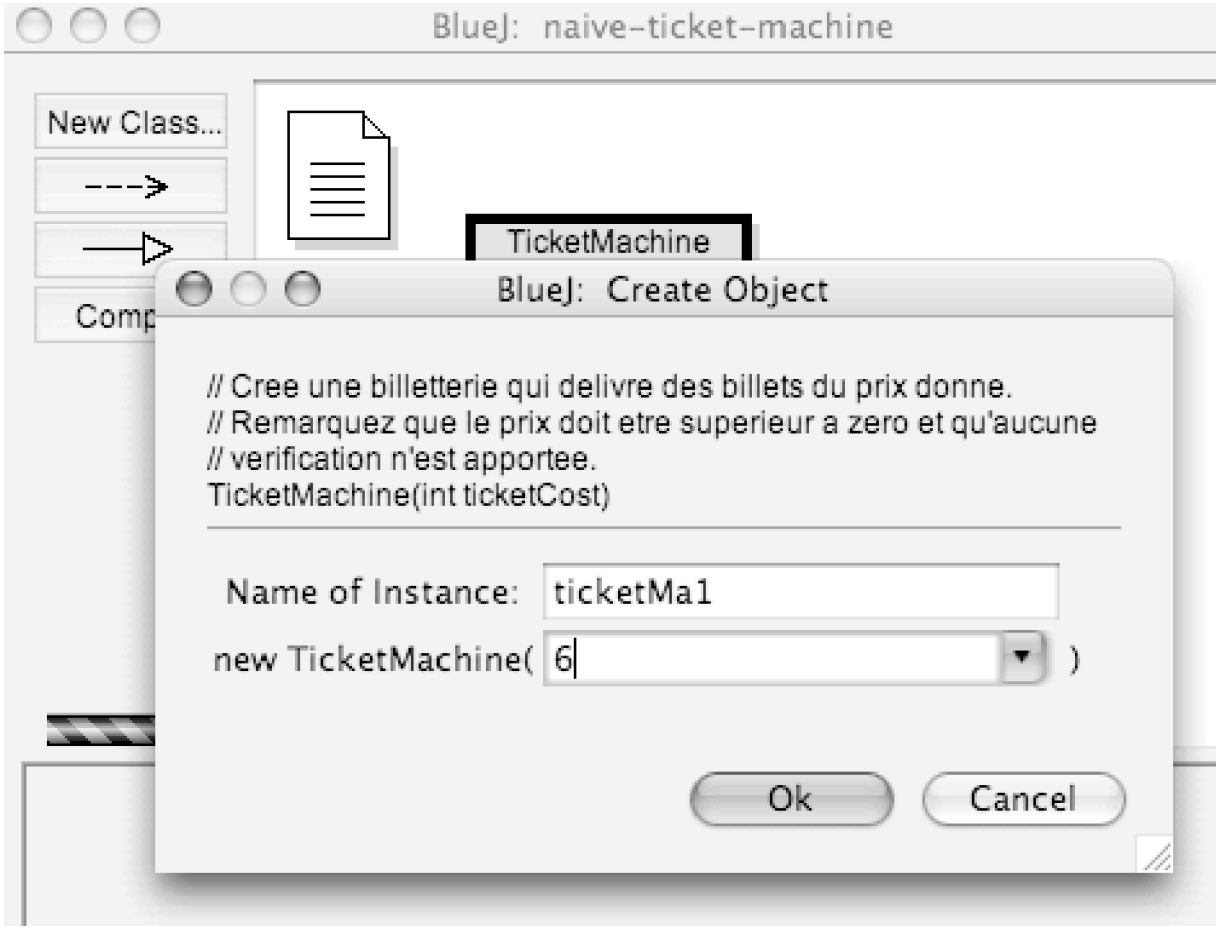
- Attention à ne pas confondre l'addition et la concaténation, toutes deux notées +

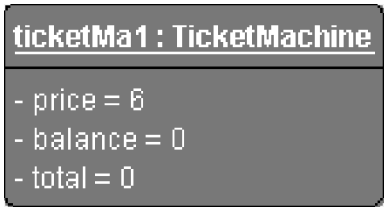
  Si `price` vaut 8 et `total` vaut 2 alors :

  `price + total + " "` vaut `"10 "`

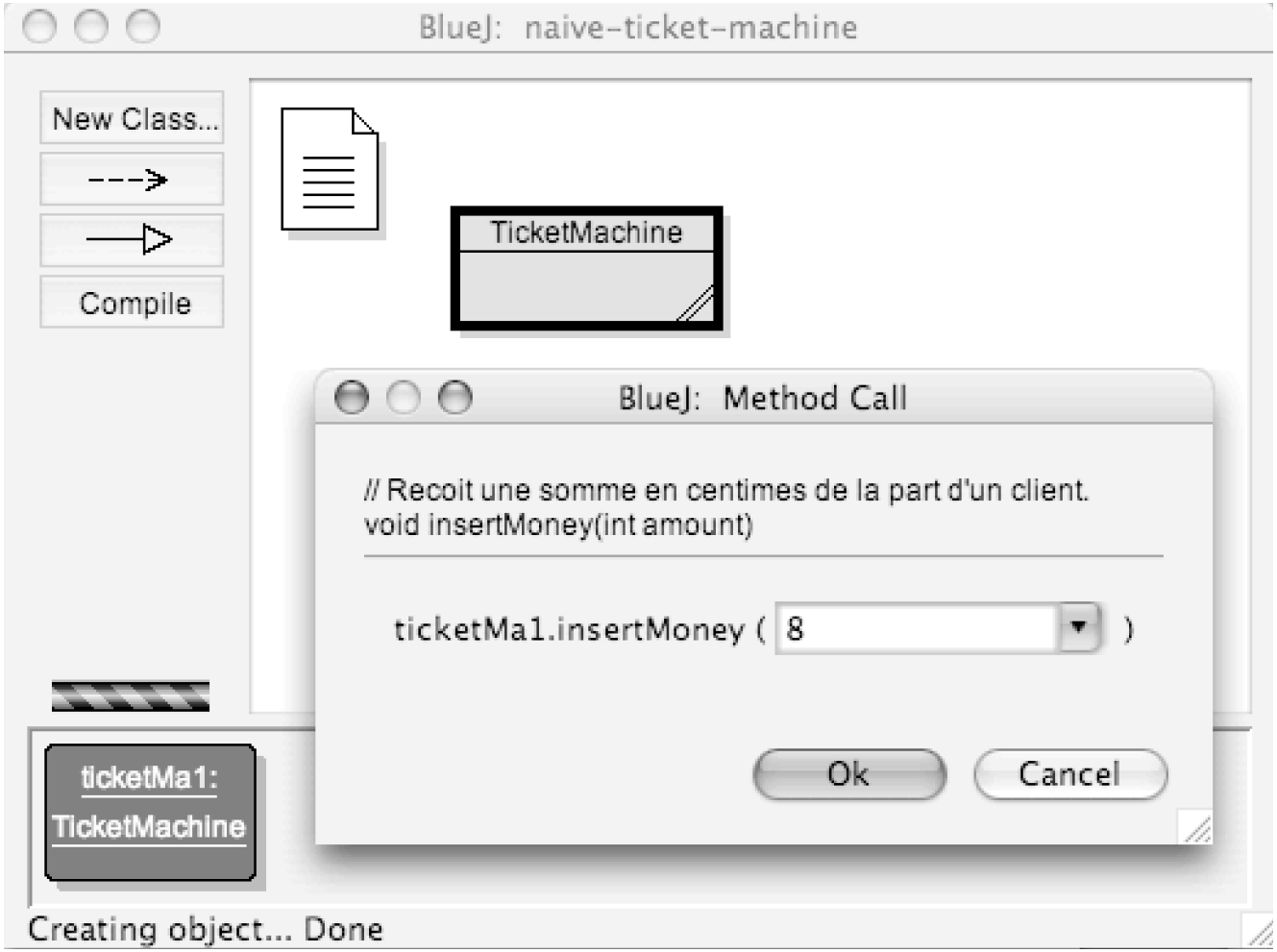
  `" " + price + total` vaut `" 82"`

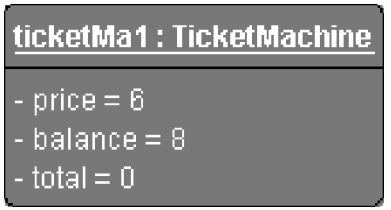
Construction d'une instance de `TicketMachine`



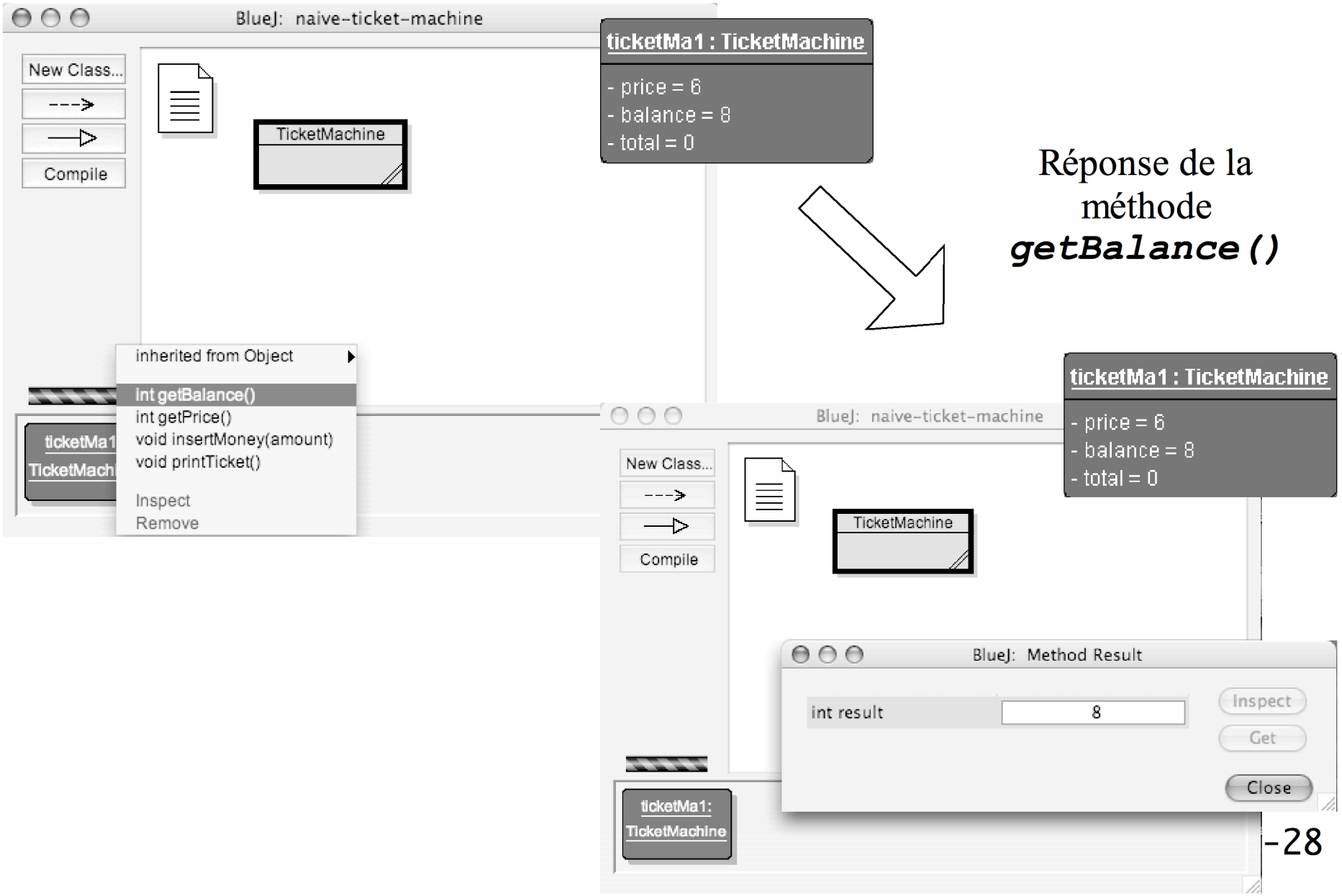


Insertion d'argent (8 euros) dans la machine

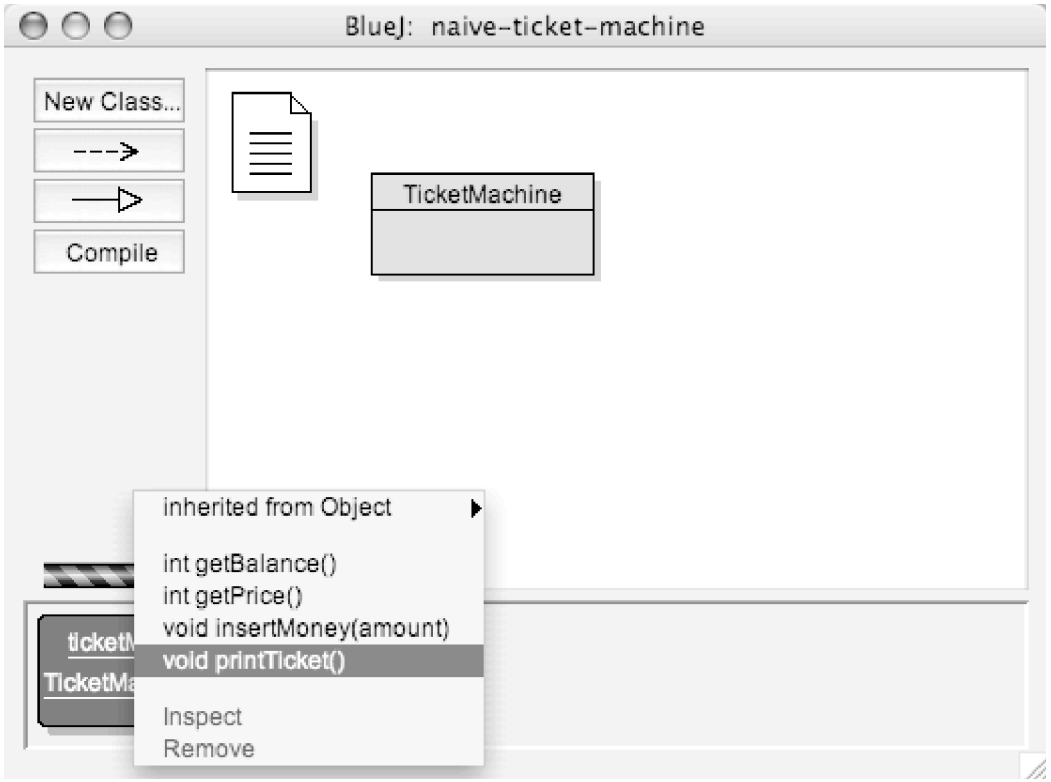


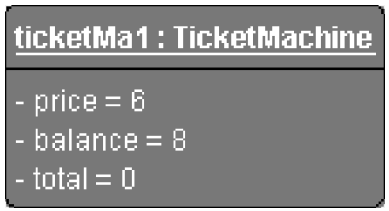


Message : quel est le solde ?

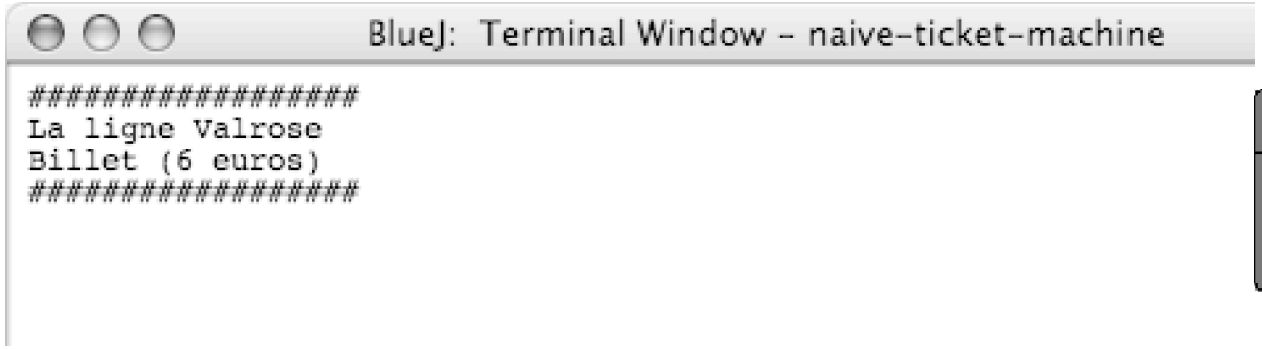


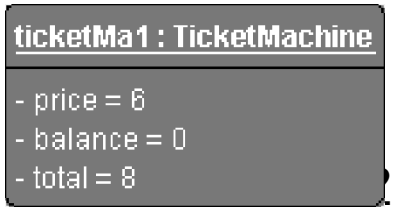
Message : imprimer le billet !



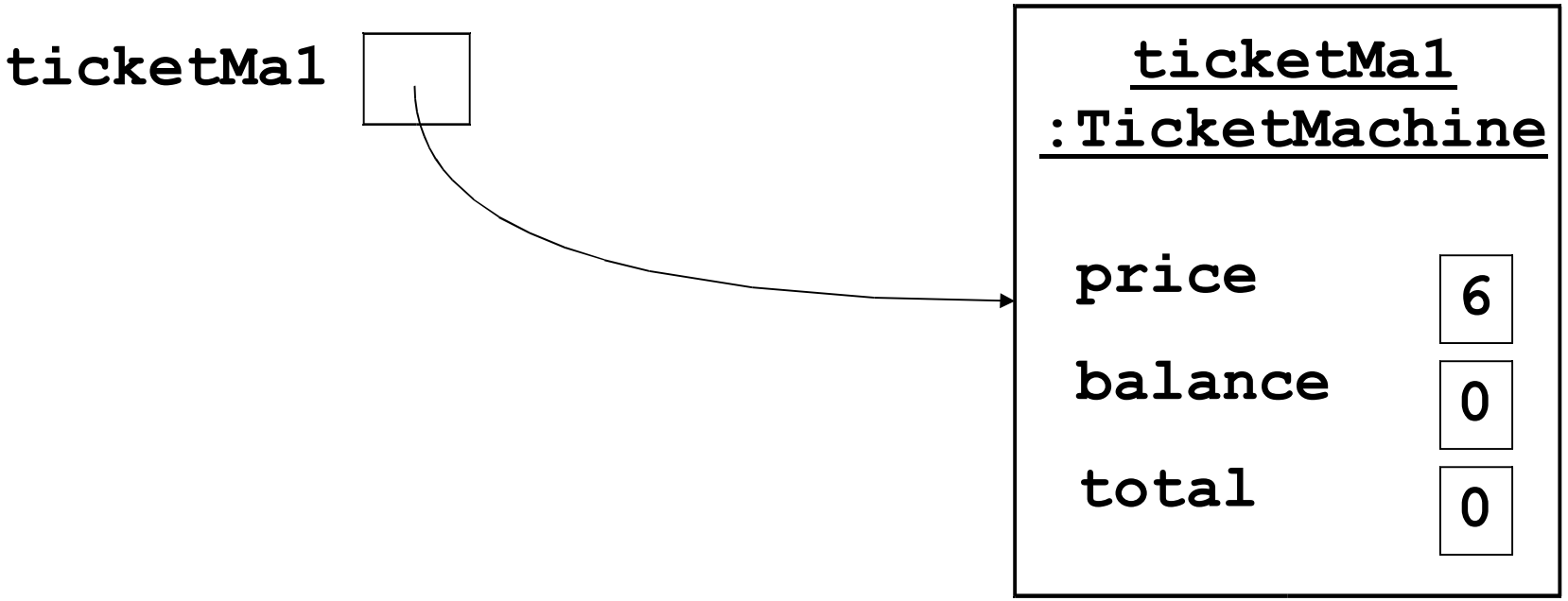


Une action est effectuée par **`printTicket()`** : aucun résultat !





Qu’est-ce qu’un objet ?



• La variable **`ticketMa1`** contient l’adresse en mémoire du bloc contenant les informations sur l’objet. On dit que c’est une **référence** (ou un **pointeur**) vers l’objet !

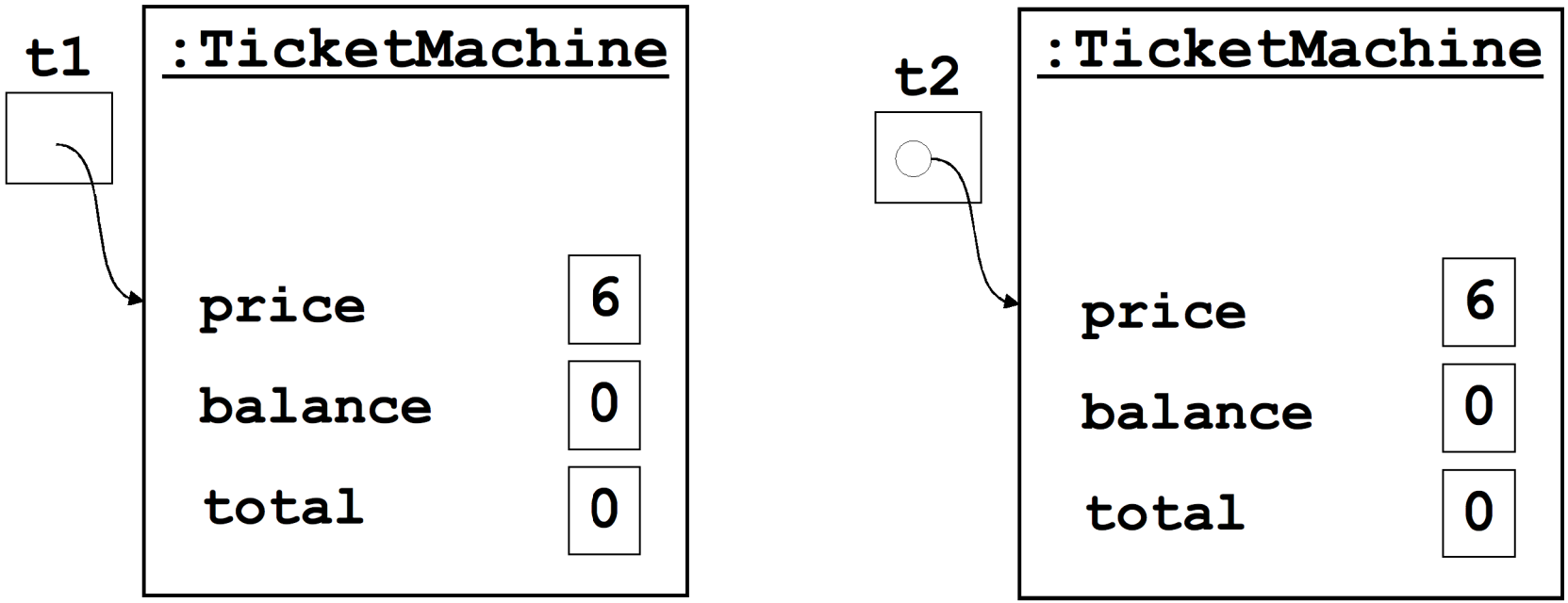
• L’objet lui-même est **pointé** par la référence.

L'égalité == entre objets

• Attention : le signe **==** entre objets est une **comparaison entre références** et non entre champs !

```
TicketMachine t1 = new TicketMachine(6);
TicketMachine t2 = new TicketMachine(6);
```

`t1 == t2` ⟹ `false`

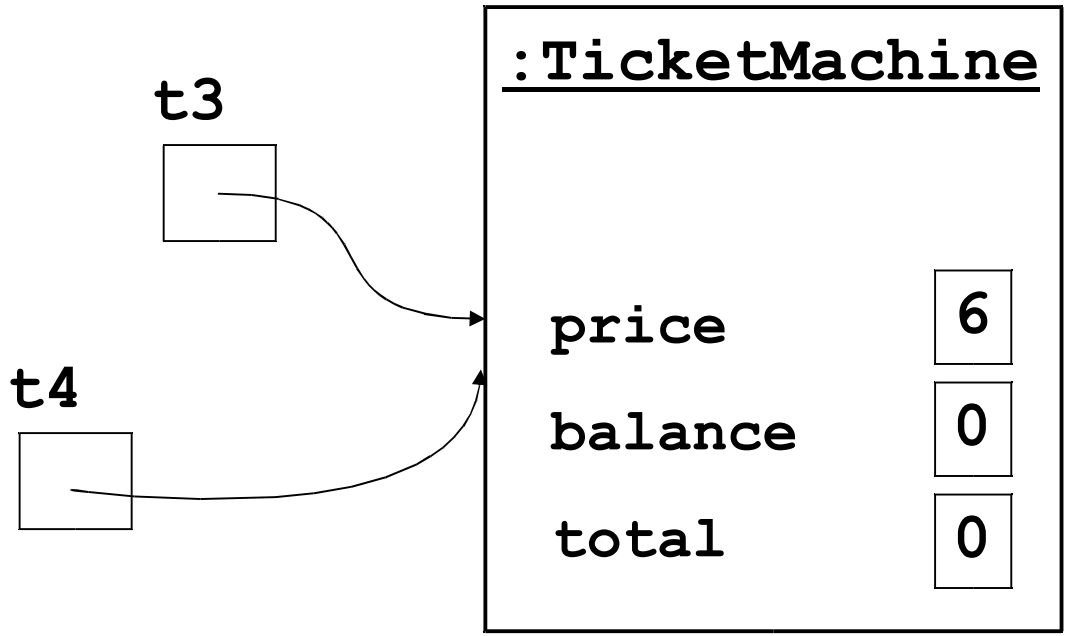


L’affectation = entre objets

- ATTENTION : le signe **=** entre objets est une **affectation entre références** et non entre champs !

```
TicketMachine t3 = new TicketMachine(6);
TicketMachine t4 = t3;
```

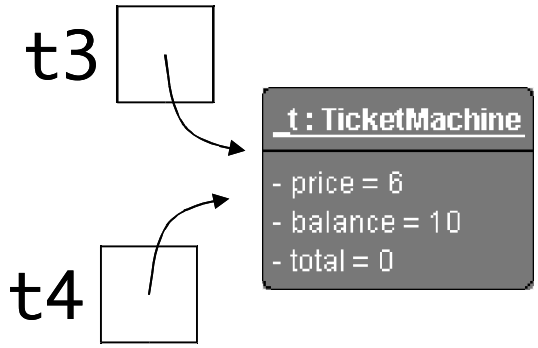
t3 == t4 ⟹ **true**



• On dit que les références **`t3`** et **`t4`** **pointent vers le même objet** en mémoire !

• Donc si je modifie l'un des champs de **`t3`**, cela induira un effet sur **`t4`** qui pointe sur la même zone :

```
t3.insertMoney(5);
t4.insertMoney(5);
```




```
t4.getBalance()  ⟹  10
```

```
t3.getBalance() == t4.getBalance()  ⟹  true
          t3 == t4                  ⟹  true
```

« *Seul le sot confond la lune et le doigt qui pointe la lune !* » [Proverbe zen]

• Comment comparer d'un seul coup tous les champs ?

EN PRINCIPE ON NE PEUT PAS !!!

• La solution usuelle consiste à rajouter à la classe **`TicketMachine`** une méthode permettant à un distributeur de savoir s'il est « égal » à un autre distributeur, au sens où ses champs contiendraient les mêmes valeurs :

```
boolean egalA(TicketMachine t) {
    return « tous mes champs sont égaux
             à ceux de t » ;
}
```

WAIT AND SEE !

Ne pas confondre :

```
if (t1.egalA(t2)) ...
if (t1 == t2) ...
```