Introduction à la programmation avec Java

UFR Sciences de Nice

Licence Math-Info 2006-2007

Module L1I1

Frédéric MALLET

Jean-Paul ROY

Les interactions entre objets

Où en sommes nous ?

- Les objets Java modélisent les objets d'un problème.

  Machine à tickets de parking, tickets de concert
- Les objets sont créés à partir de classes.

  La classe des machines à tickets
- Le bloc de déclaration de classe déclare le nom de la classe, ses champs, ses méthodes, son constructeur.
- Les champs stockent les données utilisées par les objets.

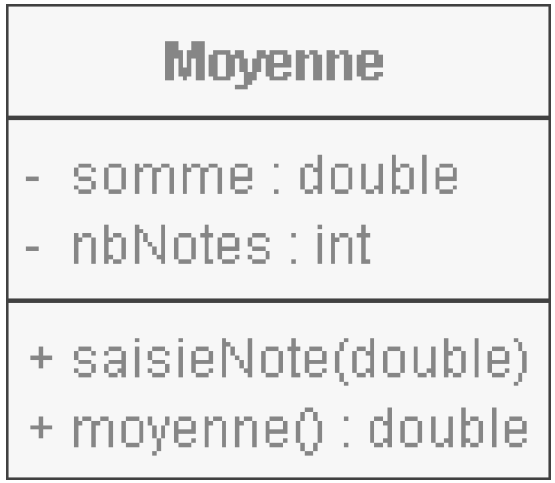
  Prix d'un ticket, montant inséré, total encaissé
- Le constructeur permet de donner des valeurs initiales différentes aux différents objets d'une classe.

  1€ pour le parking, 30€ pour le concert
- Les méthodes sont les messages acceptés par les objets.

  `getPrice(), insertMoney(int), printTicket()`

Correction du TP - Moyenne

```
class Moyenne {
   double somme;
   int nbNotes;
   Moyenne(double premiereNote) {
      somme = premiereNote;
      nbNotes = 1;
   }
   void saisieNote(double nouvelleNote) {
      somme += nouvelleNote;
      nbNotes++;
   }
   double moyenne() {
      return somme/nbNotes;        // car nbNotes ≠ 0
   }
}
```



Correction du TP – **Poly2**

On veut représenter un polynôme du second degré :

a*x² + b*x + c

Un polynôme est la donnée de ses coefficients :

- Trois champs **a**, **b**, **c** de type **double** que l'on précisera au moment de construire l'objet polynôme.

x est un paramètre : on n'en a pas besoin pour calculer les racines, mais seulement pour évaluer le polynôme !

Quelles sont les méthodes d'un polynôme ?

- **double eval(double x)** // évaluer le polynôme en x

  N.B. x peut changer à chaque évaluation !
- **double racine()** // calculer une racine
- **double delta()** // le discriminant $b^2 - 4ac$
- **public String toString()** // la représentation externe

```
class Poly2 {
   double a, b,c;
   Poly2(double va, double vb, double vc) {      // va ≠ 0
     this.a = va;
     this.b = vb;
     this.c = vc;
   }
   double eval(double x) {
     return this.a * x * x + this.b * x + this.c;
   }
   double delta() {
     return this.b * this.b - 4 * this.a * this.c;
   }
   double racine() {
     double d = this.delta();
     if (d >= 0)
       return (-this.b - Math.sqrt(d)) / (2 * this.a);
     return Double.NaN;
   }
   public String toString () {
     return this.a + "*x^2+" + this.b + "*x+" + this.c;
   }
}
```

Abstraction et Modularité

> « Le second, de diviser chacune des difficultés que j'examinerais, en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les mieux résoudre. »
> Descartes, « Discours de la méthode », 1668

L'abstraction correspond à la capacité d'ignorer les détails pour se concentrer sur un niveau supérieur du problème.

- Une voiture sert à transporter des personnes de A à B

La modularité est le processus de division d'un tout en parties bien définies.

- Une voiture a des sièges (pour transporter), un volant (pour diriger), un moteur et des roues.

Diviser pour régner !

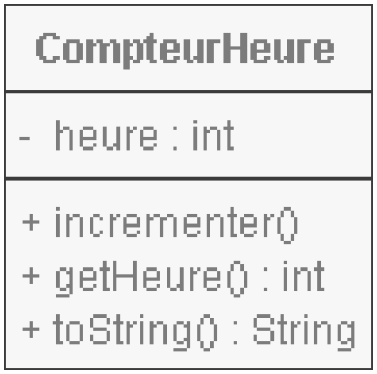
Exemple : une horloge numérique

09:25

- Abstraction
  - Une horloge numérique compte les heures et les minutes (peut importe la couleur du boîtier, l'électronique nécessaire, le coût).
- Modularité
  - L'horloge peut se décomposer en un compteur d'heures (de 0 à 23) et un compteur de minutes (de 0 à 59).
  - Si on sait faire ces deux compteurs, on sait faire une horloge.

Le compteur des heures

```
class CompteurHeure {
  int heure = 0;              // l'état : valeur de l'heure

  void incrementer() {        // de 1 en 1, jusqu'à 23
    if (this.heure == 23)
      this.heure = 0 ;        // remise à zéro
    else
      this.heure = this.heure + 1 ;
   }
   int getHeure() {           // renvoie l'heure courante
    return this.heure ;
   }

   public String toString() {      // affichage sur deux chiffres
    if (this.heure < 10)
      return "0" + this.heure ;
    else
      return "" + this.heure ;
   }
} }
```

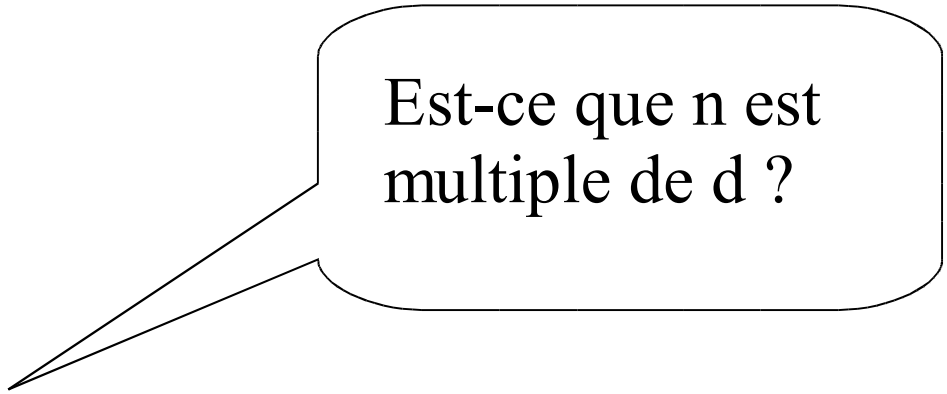


L'opérateur %

- L'opérateur % calcule le reste de la division entière.
- Par exemple :

  `27 / 4` vaut **`6`**

  `27 % 4` vaut **`3`**




```
boolean estMultipleDe(int n, int d) {
  return (n % d == 0);
}
```

La concaténation des chaînes (rappel)

Les opérateurs ont différentes significations suivant le type des opérandes ! Exemples :

`+ : int x int → int`

`42 + 12 → 54`

`+ : String x String → String`

`"Ja" + "va" → "Java"`

ici + est l'opérateur de concaténation sur les chaînes.

`+ : String x int → String`

`"Java" + 2 → "Java2"`

ici + est l'opérateur de concaténation mixte.

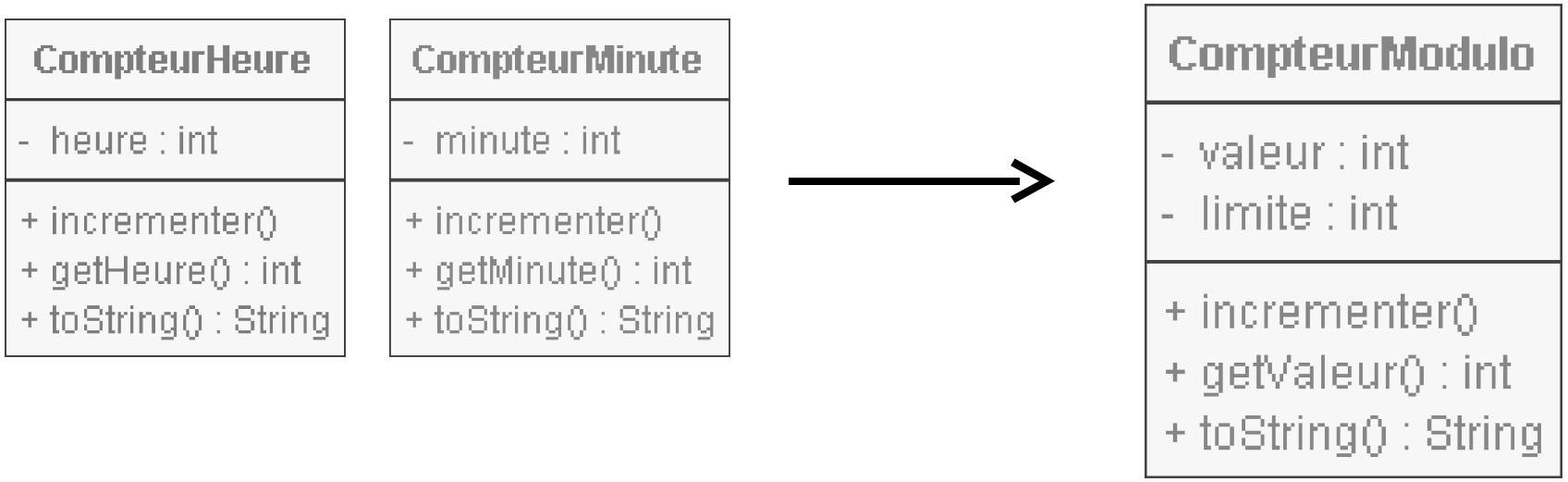
Astuce

- pour convertir l'entier heure en une chaîne de caractères, on lui concatène la chaîne vide : **`"" + this.heure`**

Attention : comparer **`2 + 3 + ""`** et **`"" + 2 + 3`**

Abstraction : un `CompteurModulo`

- Le compteur des minutes est similaire au compteur des heures, mais il compte jusqu'à 59 au lieu de 23…



- Abstraction
  - · On remplace la valeur limite par une variable à initialiser (attribut/variable locale/paramètre ?)
  - · Mais on perd la spécificité (**`heure`** devient **`valeur`**)

Le constructeur

- Comment/quand initialiser la valeur limite ?
  - dès le début de la vie de l'objet.
  - en fonction, d'une valeur extérieure.

- Constructeur avec paramètre :

```
CompteurModulo(int valeurLimite) {
   this.valeur = 0 ;
   this.limite = valeurLimite ;
}
```

Même nom que la classe !

champs

paramètre

- Instanciation (création d'un nouvel objet de la classe) :
  - pour les heures : **`new CompteurModulo(24)`**
  - pour les minutes : **`new CompteurModulo(60)`**

Le sens courant du mot clé « `this` »

- **`this`** seul ou suivi d'un point, représente **l'objet courant**, qui est sous-entendu, dans une méthode ou un constructeur.
  - ***`this.limite`*** *est donc l'attribut* ***`limite`*** *de l'objet courant*
- ***`this`*** est obligatoire lorsqu'il y a **ambiguïté**. Une variable paramètre peut en effet avoir le même nom qu'un champ, (c'est une *surcharge*) : il faut alors pouvoir les distinguer :

```
CompteurModulo(int limite) {
  this.valeur = 0;
  this.limite = limite;
}
```

- **`this`** peut être omis en général, il est sous-entendu :
  - **`this.limite`** s'écrira **`limite`** sauf dans le cas précédent !
  - **`this.incrementer()`** s'écrira **`incrementer()`**, c'est un message envoyé à l'objet **`this`** !

La surcharge du constructeur

- Elle permet d'initialiser un objet de deux façons distinctes :
  - **`new CompteurModulo(24, 9)`**
  - **`new CompteurModulo(24)`**

```
CompteurModulo(int limite, int valeur) {
   this.valeur = valeur ;
   this.limite = limite ;
}

CompteurModulo(int limite) {
   this.valeur = 0 ;
   this.limite = limite ;
}
```

Signature :
`CompteurModulo(int,int)`

Signature :
`CompteurModulo(int)`

- La surcharge est résolue grâce au nombre et aux types des arguments d'appels (c-à-d, la signature du constructeur)

Un autre sens pour le mot « `this` »

- Le codage du constructeur **`CompteurModulo(int)`** est maladroit, puisqu'il répète celui du précédent dans un cas particulier !
- *Dans un constructeur uniquement*, le mot « **`this`** » suivi d'une parenthèse, ET utilisé en 1ère ligne uniquement, fait référence à un autre constructeur de la même classe, souvent le constructeur le plus général, qui a le plus de paramètres :

```
CompteurModulo(int limite) {
   this(limite, 0);
}
```

On passe la main à l'autre constructeur **`CompteurModulo(int, int)`**

- Ne pas confondre **`this.`** et **`this(`**

```
class CompteurModulo {
  int valeur;              // valeur courante
  int limite;              // limite maximum du comptage

  // constructeurs non reproduits ici ...

  void incrementer () { // de 1 en 1, jusqu'à limite-1
    this.valeur = (this.valeur + 1) % this.limite;
  }
  int getValeur() {              // renvoie la valeur courante
    return this.valeur;
  }
  public String toString() {  // affichage sur deux chiffres
    if (this.valeur < 10)
      return "0" + this.valeur;
    else
      return "" + this.valeur;
  }
}
```

Le mot clé assert – JDK 1.4

- Le JDK 1.4 a introduit le mot clé **`assert`** en Java
- **`assert`** permet d'affirmer qu'une propriété est vraie
  - *En fait, si la propriété est fausse, une erreur se produit et le programme est arrêté*
- Syntaxe
  - **`assert`** ***`propriété`*** **`;`**
    - La propriété est une expression booléenne qui doit être évaluée à true pour passer à l'instruction suivante
  - **`assert`** ***`propriété`*** **`: "`*`message d'erreur`*`" ;`**
    - On peut également choisir un message d'erreur si la propriété est false.
    - Sinon un message d'erreur par défaut est proposé.

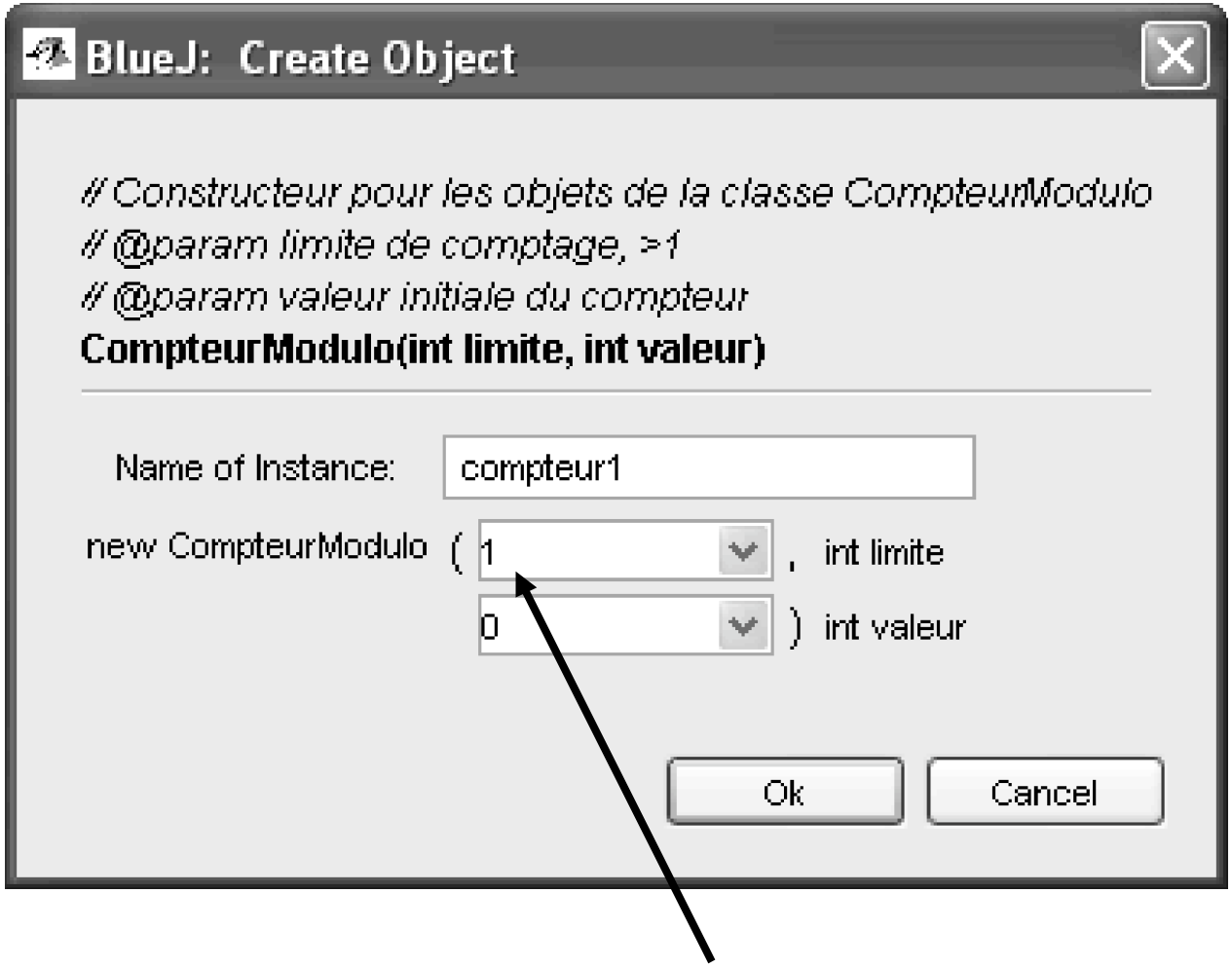
Exemple – le **`CompteurModulo`**

```
void incrementer () {    // de 1 en 1, jusqu'à limite-1
    this.valeur = (this.valeur + 1) % this.limite;
}
```

- Le code proposé peut provoquer plusieurs erreurs difficiles à détecter :
  - Une erreur se produit lorsque `limite` vaut 0 ;
  - Le comportement est « étrange » pour certaines valeurs de `limite` (1, valeurs négatives).
- Le problème aurait dû être détecté dans le constructeur :

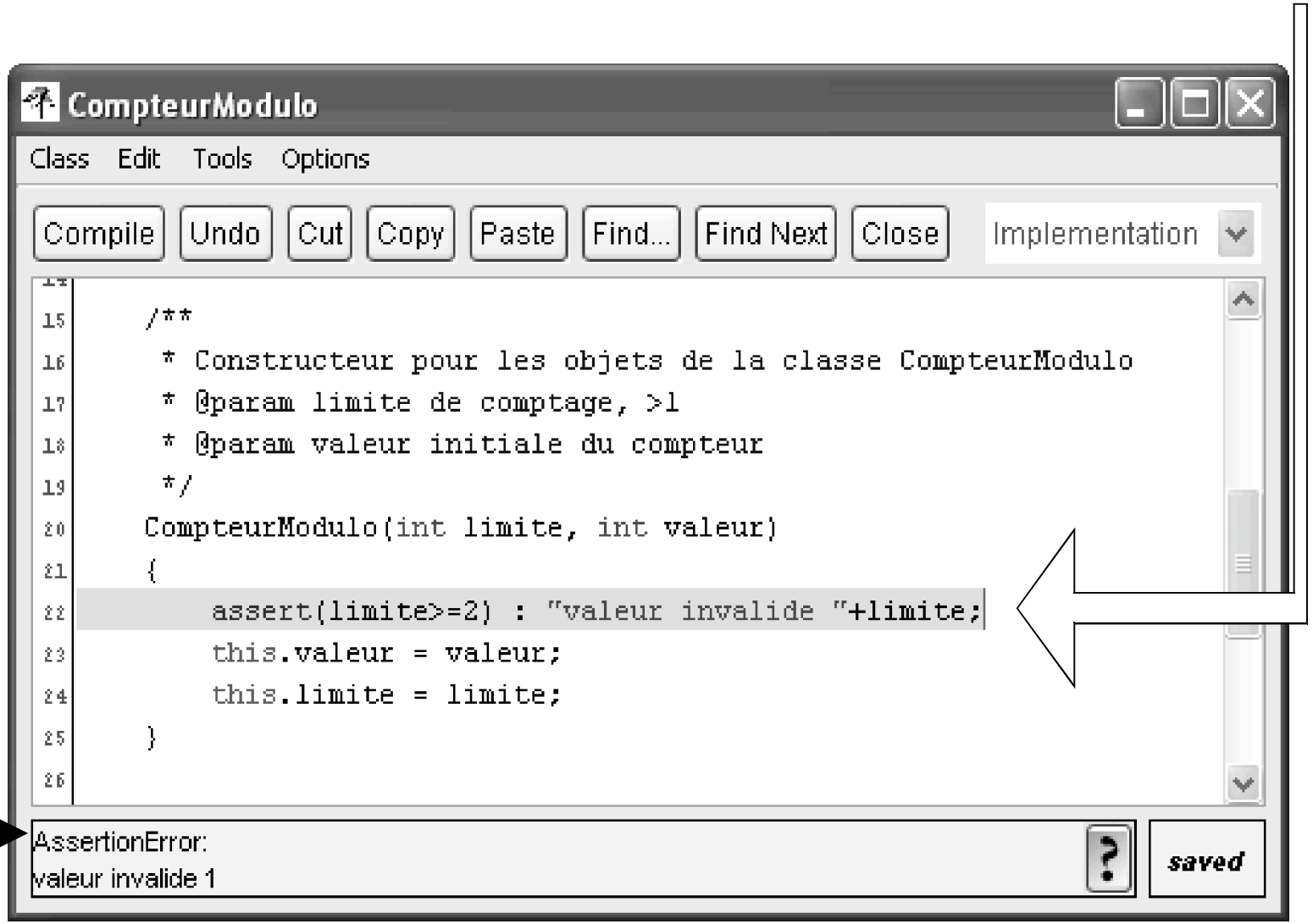
```
CompteurModulo(int limite, int valeur) {
    assert(limite >= 2) : "valeur limite invalide " + limite;
    this.valeur = valeur ;
    this.limite = limite ;
}
```

Dans BlueJ



Malgré la documentation je choisis limite = 1

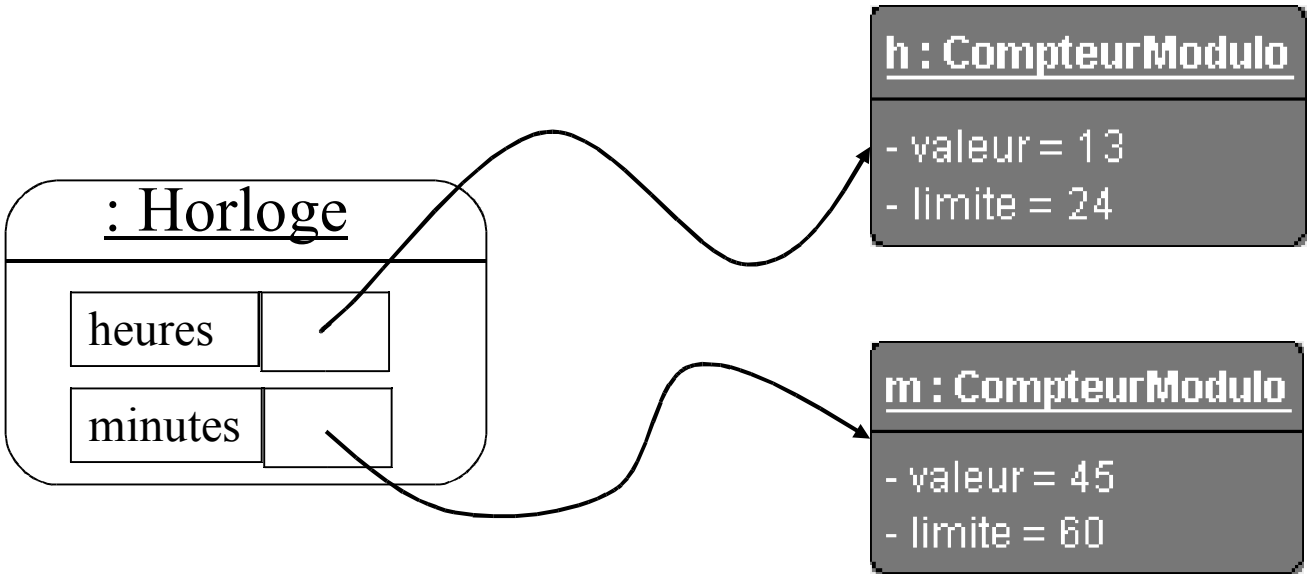
- Une erreur se produit : *AssertionError*
- La construction est annulée
- La ligne où l'assertion est **violée** est mise en évidence



Composer des objets

Diagrammes d'objets

- Une horloge est composée de deux **`CompteurModulo`**
  - un pour les heures et un pour les minutes.



La « composition »

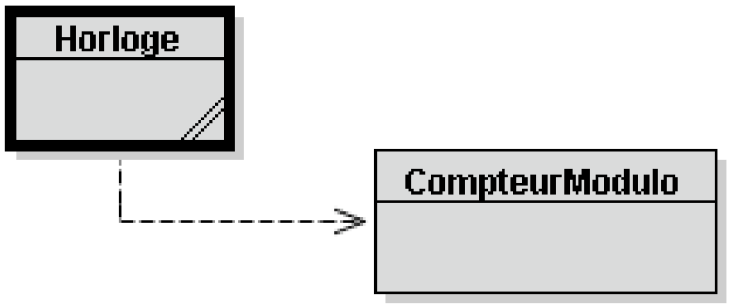
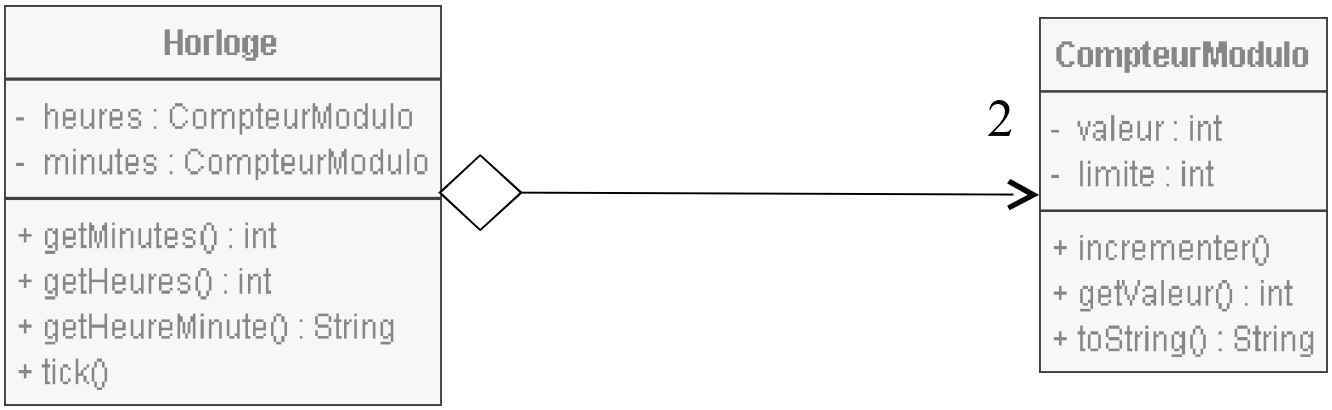


Diagramme de classes en BlueJ : la classe **`Horloge`** utilise la classe **`CompteurModulo`**

- Une horloge est composée de deux **`CompteurModulo`**
  - un pour les heures et un pour les minutes.

Diagramme de classes UML

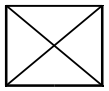


Classe et type

- Les classes définissent des types.
  - Un nom de classe peut être utilisé comme type pour une variable (attribut, paramètre, variable locale) :

```
CompteurModulo heures;
```

  - Attention, l'objet n'est pas construit : **`null`**



  - **`heures`** pointe sur un objet seulement après une initialisation :

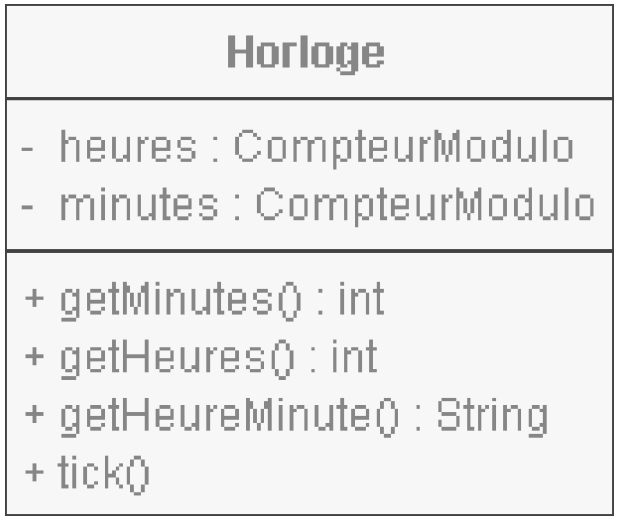
```
heures = new CompteurModulo(24) ;
```

  - Les variables de type objet stockent des références aux objets.

La classe **Horloge**

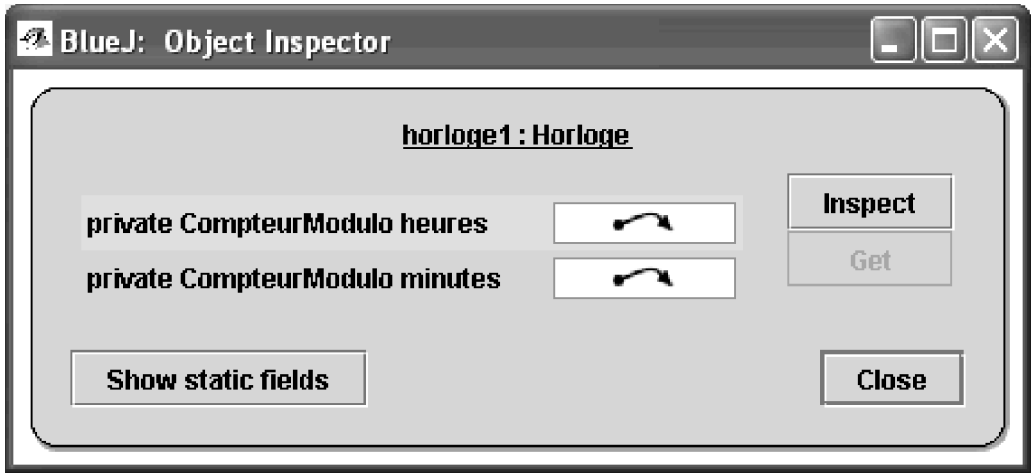
```
/**
 * Une horloge est composée de deux compteurs
 */
class Horloge {
  /** mémorise les heures */
  CompteurModulo heures;
  /** mémorise les minutes */
  CompteurModulo minutes;

  /** Constructeur de la classe Horloge
   * @param heures l'heure courante (de 0 à 23)
   * @param minutes les minutes courantes (de 0 à 59)
   */
  Horloge(int heures, int minutes) {
    this.heures = new CompteurModulo(24, heures);
    this.minutes = new CompteurModulo(60, minutes);
  }
  // méthodes sur la page suivante…
```

```
/** @return les minutes courantes observées par l'horloge */
int getMinutes() {
  return minutes.getValeur();          // délègue à l'objet minutes
}

/** @return l'heure courante observée par l'horloge */
int getHeures() {
  return heures.getValeur();           // délègue à l'objet heures
}

/** @return heure:minutes */
String getHeureMinute() {
  return heures + ":" + minutes ;      // concaténation
}

/** augmente l'heure courante d'une minute */
void tick() {
  minutes.incrementer();
  if (minutes.getValeur() == 0)
      heures.incrementer();
}
```

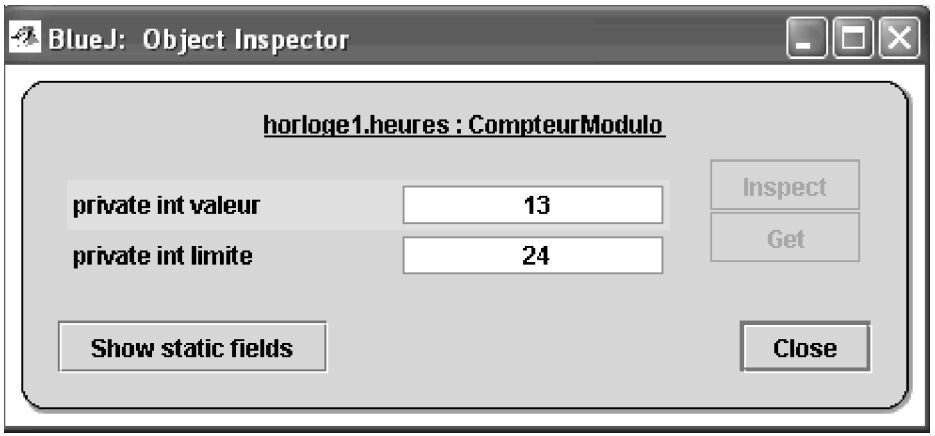


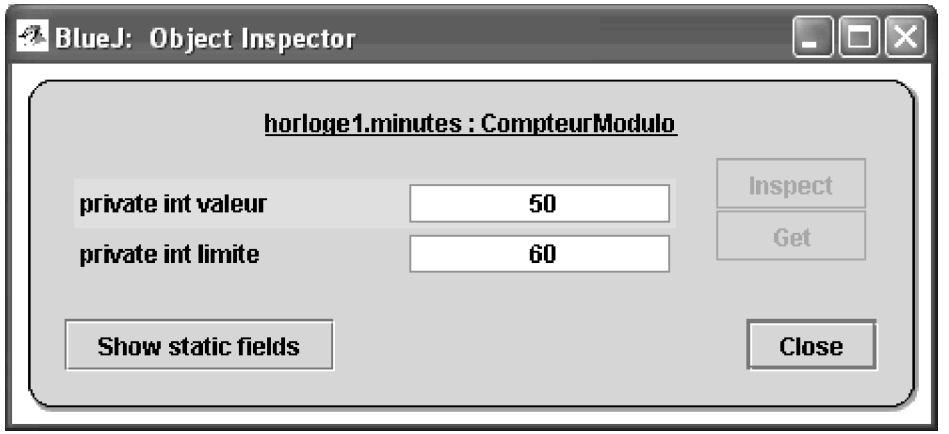
Avec BlueJ

- BlueJ montre les attributs de type objet sous forme de flèches (références)



- Le bouton Inspect permet de dé-référencer (plonger dans l'objet référencé) :





La classe **`Math`** : une classe sans objets !

- Vous avez implémenté la méthode abs en TD dans le contexte de la classe **`Calculette`**.

```
class Calculette {
  double abs(double valeur) {
    if (valeur >= 0)
      return valeur ;
    else
      return -valeur;
  }
}
```

- Il FAUT créer un objet **`calc`** de la classe **`Calculette`** pour invoquer cette méthode sous la forme **`calc.abs(-5)`**…
- Mais quel serait donc l'état de cet objet sans champ ?

Méthodes « statiques »

- La plupart des méthodes que l'on a vues jusqu'à présent utilisent un objet (ayant un état) pour être invoquées.

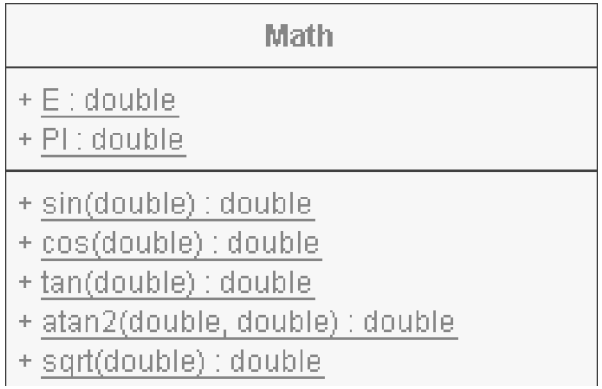
```
ticketMa1.insertMoney(12);
this.insertMoney(12);
insertMoney(12);                    // this est sous-
```

  entendu !

- Les méthodes statiques ne dépendent pas d'un état !
  - Elles ne nécessitent pas d'objet pour être exécutées
  - Elles sont quand même associées à une classe
  - En UML, elles sont représentées soulignées pour être distinguées
  - C'est le cas de la méthode **`abs`** vue en TD

Quand une méthode est-elle statique ?

- C'est précisé par l'énoncé !
- La méthode est soulignée dans les diagrammes UML



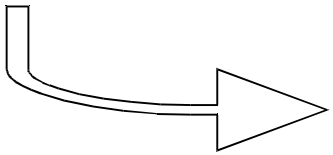
- C'est indiqué dans la javadoc :

Method Summary

static double	**abs**(double a) Returns the absolute value of a double value.
static float	**abs**(float a) Returns the absolute value of a float value.
static int	**abs**(int a) Returns the absolute value of an int value.

Comment utiliser les méthodes statiques ?

- C'est le SEUL cas où il n'y a pas besoin d'objet pour envoyer un message !
- On s'adresse à la classe plutôt qu'à l'objet.
  - Exemple : **`Math.sqrt(12)`** (cf. TD3)
  - **`Math`** n'est pas un objet mais une classe.
- Il faut quand même utiliser la notation pointée.
- Une méthode statique est aussi nommée méthode de classe.
- Une méthode « normale » (non statique) est aussi nommée méthode d'instance.



elle s'adresse à un objet,
instance d'une classe !

La classe **`Math`**

- La classe **`Math`** regroupe un certain nombre de méthodes de classe (donc statiques) utiles pour les calculs mathématiques (log, trigo, etc)
  - Exemple : **`static double cos(double x)`**
- Elle a également deux attributs statiques : **`PI`** et **`E`**
  - Tout comme les méthodes de classe, les *attributs de classe* ne nécessitent pas d'objet pour être utilisés
- On s'adresse à la classe plutôt qu'à un objet :
  - **`Math.PI`**
  - **`Math.E`**
  - **`double x = Math.log(2);`**

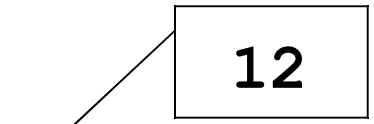
La méthode d'instance **abs**

• de la classe **Calculette** (TD3/Exo2)… Elle modifiait le champ **screen** :

```
void abs() {
  if (this.screen < 0)
    this.screen = -this.screen;
}
```

• C'est une méthode d'instance (pas de mot « **static** »). Pour l'utiliser, il faut disposer d'un objet de type **Calculette** :

```
Calculette calc = new Calculette();
calc.addToScreen(-12);
calc.abs();
System.out.println(calc.getScreen());
```



• Comparer avec la calculette du transparent 4-25

La méthode de classe **abs**

• Supposons que la véritable fonction mathématique $x \mapsto |x|$ **Math.abs** n'existe pas… et que nous ayons des calculs de valeurs absolues à faire sur plusieurs nombres ! Ecrivons une méthode statique, par exemple dans une classe **MathUtil** :

```
class MathUtil {
  static int abs(int x) {
    if (x < 0) return x; else return -x;
  }
  static double abs(double x) {
    if (x < 0) return x; else return -x;
  }
}
```

Surcharge !

• Dans la classe **MathUtil**, elle s'utilisera par **abs**, mais dans une autre classe, elle s'utilisera par **MathUtil.abs**

Visibilité : `private` ou `public` ?

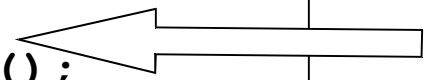
- Java est plutôt paranoïaque sur la sécurité, à la fois pour des raisons liées à la propreté du logiciel, et pour des raisons liées aux dangers d'Internet.
- Une classe est en général publique.
- Un champ ou une méthode peut être privée ou publique :
- **`public`**
  - Toute classe a le droit de les utiliser (à condition de posséder une référence sur un objet).
  - Les méthodes sont presque toujours publiques. On peut restreindre leur accès si elles n'ont d'utilité qu'à l'intérieur de la classe.
- **`private`**
  - Seulement utilisable dans le texte de la classe qui le déclare.
  - Donc inaccessibles à partir d'une autre classe !
  - Les champs sont presque toujours privés, contrairement à ce que nous avons fait jusqu'ici pour simplifier une première approche…

Dans la classe **`TicketMachine`**

```
public class TicketMachine {
 private int price ;

 public int getPrice() {
   return this.price;
 }

  ...
 public void foo()
 { // méthode ou constructeur:
   // tout ceci est autorisé :

   int x = this.price+5;
   this.price = 0;
    y = this.getPrice() + 5;
    TicketMachine mach1 = new …;
    x = mach1.price;
    x = mach1.getPrice();
 }
}
```

Dans une autre classe

```
public class Foo {
   ...
   public void foo () {
     TicketMachine mach2 ;
    // ceci est INTERDIT:
     int p = mach2.price ;
     // ceci est autorisé :
     int p = mach2.getPrice();
    }
}
```



ATTENTION : on voit qu'un objet a donc accès aux champs private d'un autre objet de la même classe !…

Le mot ***`private`*** concerne donc la classe et non l'objet !

`private` et `public` avec BlueJ