

Programmation Orientée Objet

1ère Partie : Introduction

Gérald Oster <oster@loria.fr>

Présentation du module

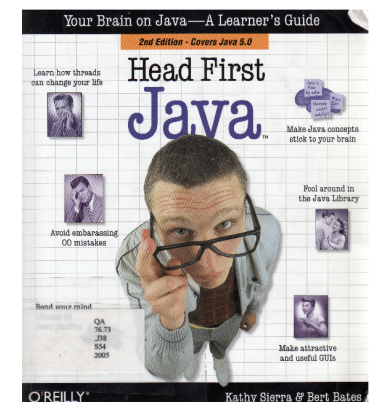
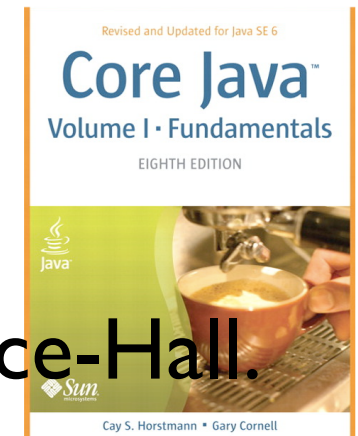
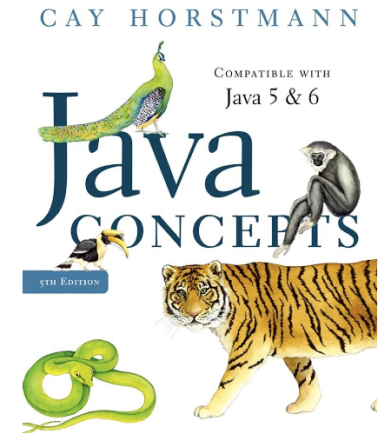
- Objectifs :
 - Apprendre à programmer
 - Connaître les concepts fondamentaux des langages objets
 - Etudier et maîtriser un langage objet : Java

Organisation pratique

- 5 cours magistraux
- 9 séances de travaux dirigés
- 8 séance de travaux pratiques
- Evaluation :
 - 1 examen sur table (09.01.2009)
 - 1 travail pratique (à planifier)
 - 1 projet (évaluation dans le module TOP)

Bibliographie

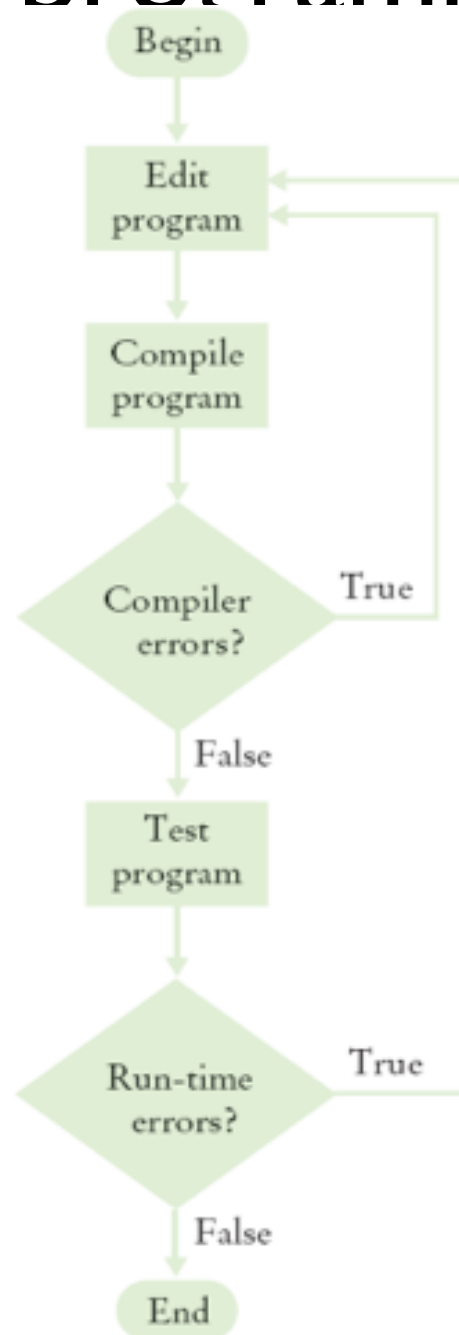
- *Big Java*,
C. Horstmann, John Wiley & Sons.
- *Java Concepts*,
C. Horstmann, John Wiley
- *Core Java*,
C. Horstmann et G. Cornell, Prentice-Hall.
- *Head First Java*,
K. Sierra et B. Bates, O'Reilly.
- API disponible en ligne
(<http://java.sun.com/javase/6/docs/api/>)



Plan du cours

- Introduction
- Programmation orientée objet :
 - Classes, objets, encapsulation, composition
 - 1. Utilisation
 - 2. Définition
- Héritage et polymorphisme :
 - Interface, classe abstraite, liaison dynamique
- Généricité
- Collections

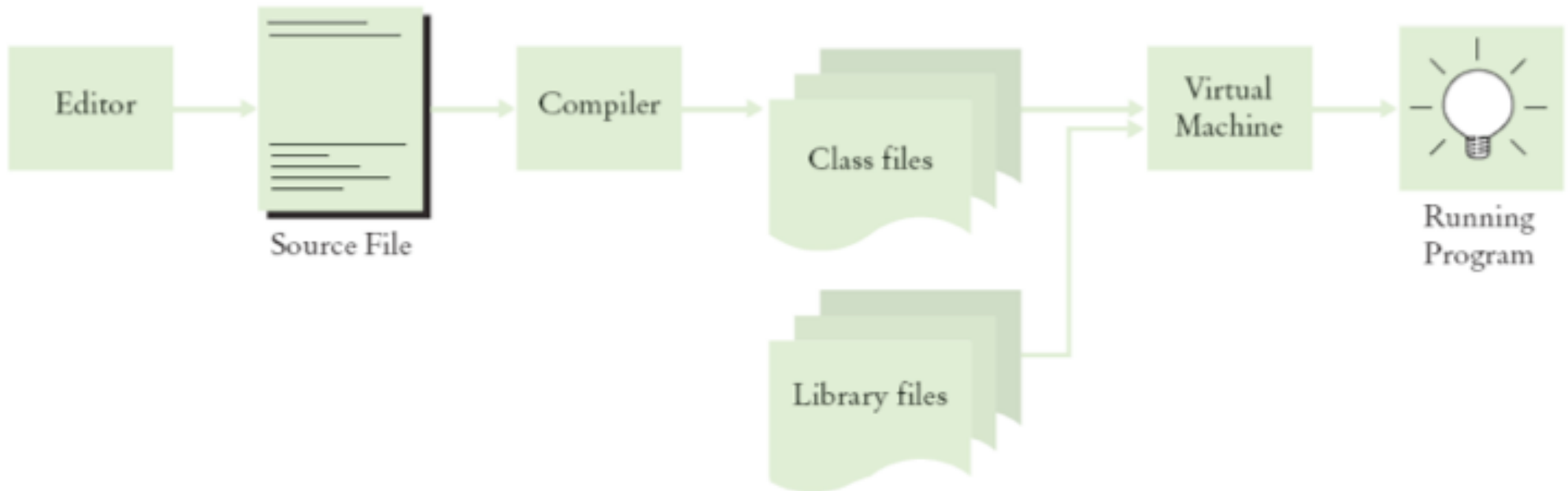
Procédé de programmation



Pourquoi Java ?

- Simplicité
- Correction
- Indépendance vis à vis de la plate-forme
(*write once, run anywhere*)
- Une librairie très riche
- Conçu pour Internet ;-)

Procédé de compilation



Programmation Objet ? Kesako ?

- Programmation dirigée par les données et non par les traitements
- Les procédures/méthodes existent toujours, mais on se concentre :
 - d'abord, sur les entités à manipuler
 - ensuite, comment les manipuler
- Notion d'encapsulation :
 - les données et les procédures liées sont regroupées au sein d'une même entité
 - cacher le fonctionnement interne d'une entité

I^{ère} Partie : Concepts et manipulations

Types

- Chaque valeur/expression a un type
- Exemple :
 - "bonjour" : **type** `String` (chaîne de caractères)
 - 27 : **type** `int` (entier)
 - 'x' : **type** `char` (caractère)
 - true : **type** `boolean` (valeur booléenne)

Variables

- Variable :
 - Stocke une valeur
 - Peut être utilisée à la place de la valeur qu'elle stocke

- Définition d'une variable :

nomDuType nomVariable = valeur;

ou

nomDuType nomVariable;

opérateur d'affectation

- Par exemple :

`String greeting = "Hello, Dave!";`

Variables : affectation

```
int luckyNumber = 13; ①
```

```
luckyNumber = 12; ②
```

①

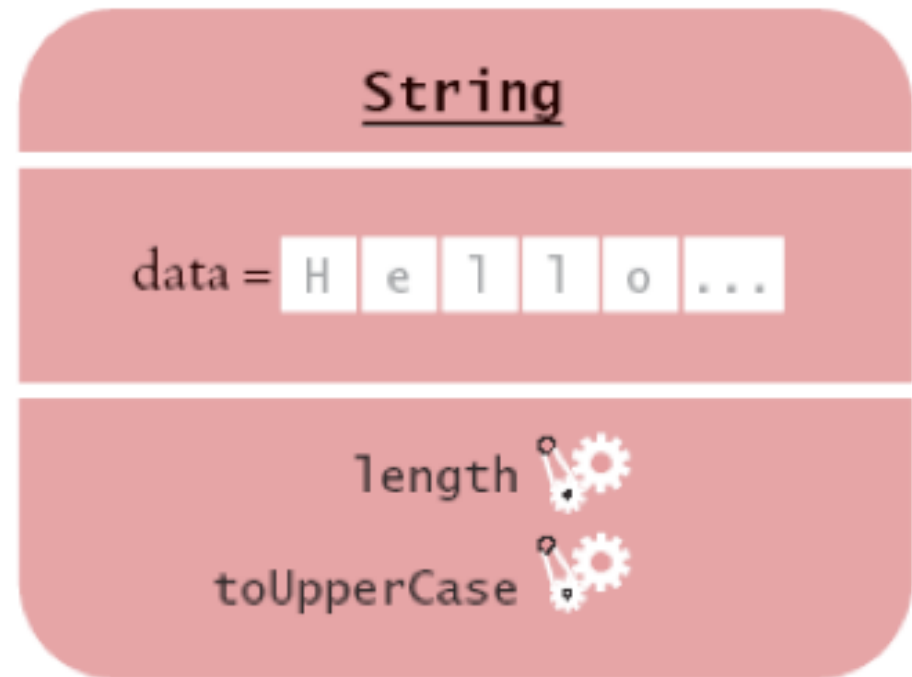
luckyNumber = 13

②

luckyNumber = 12

Objet

- Objet : une entité manipulée dans un programme (en appelant des méthodes)
- Un objet est caractérisé par :
 - Son identité :
 - Unicité
 - Son type
 - Son état :
 - valeurs des attributs à un moment donné
 - Son comportement :
 - ensemble des méthodes (consultation, mise à jour)

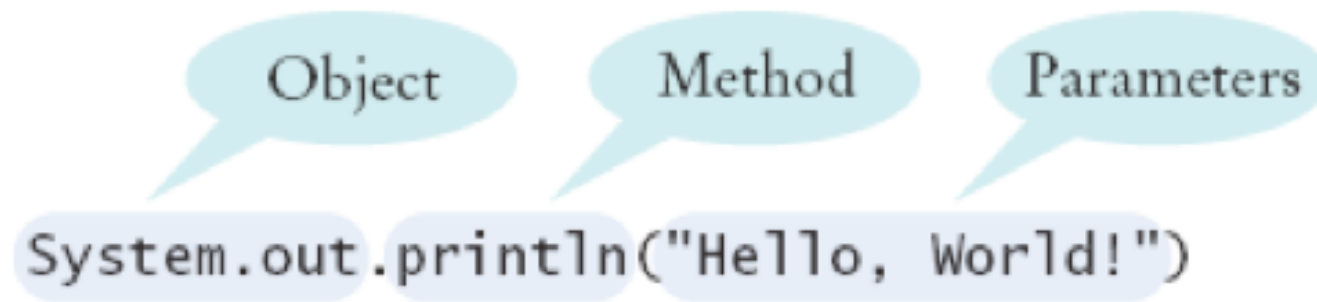


Classe

- Définition d'une famille d'objets ayant une même structure et un même comportement caractérisée par un nom
- Chaque objet appartient à une classe
- Permet d'instancier une multitude d'objets
- Convention d'écriture
 - `objetA : NomDeClasse`

Méthode

- Méthode : séquence d'instructions qui accèdent aux données d'un objet
- On manipule des objets par des appels de ses méthodes



- Interface publique :
 - définie ce l'on peut faire sur un objet

Méthode /2

- La classe d'un objet détermine les méthodes que l'on peut appeler sur un objet
- `String greeting = "Hello, World! "`
- `length()` : **compte le nb de caractères**
- `int n = greeting.length(); // affecte 13 à n`
- `toUpperCase()` : **crée un nouvel objet String dont les caractères sont en majuscules**
- `String river = "Mississippi";`
- `String bigRiver = river.toUpperCase(); // "MISSISSIPPI"`
- **Quand on appelle une méthode sur un objet, toujours vérifier que cette méthode est définie dans la classe appropriée**
- `System.out.length(); // This method call is an error`

Paramètres explicites et receveur

- Paramètre (paramètre explicite) :
 - données en entrée d'une méthode
 - certaines méthodes n'ont pas de paramètres explicites

- `System.out.println(greeting)`
`greeting.length()`

"Hello, World" →

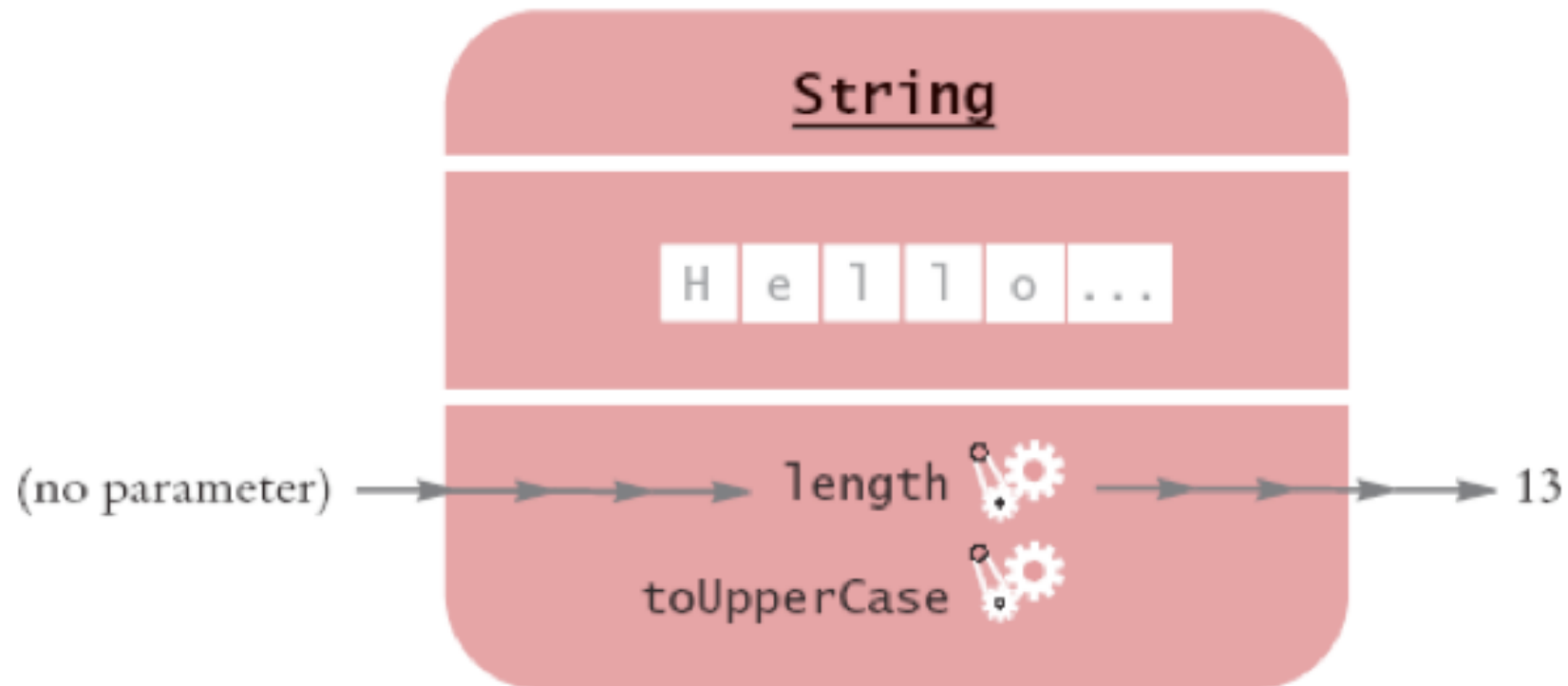


- Receveur (paramètre implicite) :
 - objet sur lequel on invoque la méthode
- `System.out.println(greeting)`

Valeur de retour

- Le résultat calculé par une méthode
- Retournée au code qui a appelé la méthode

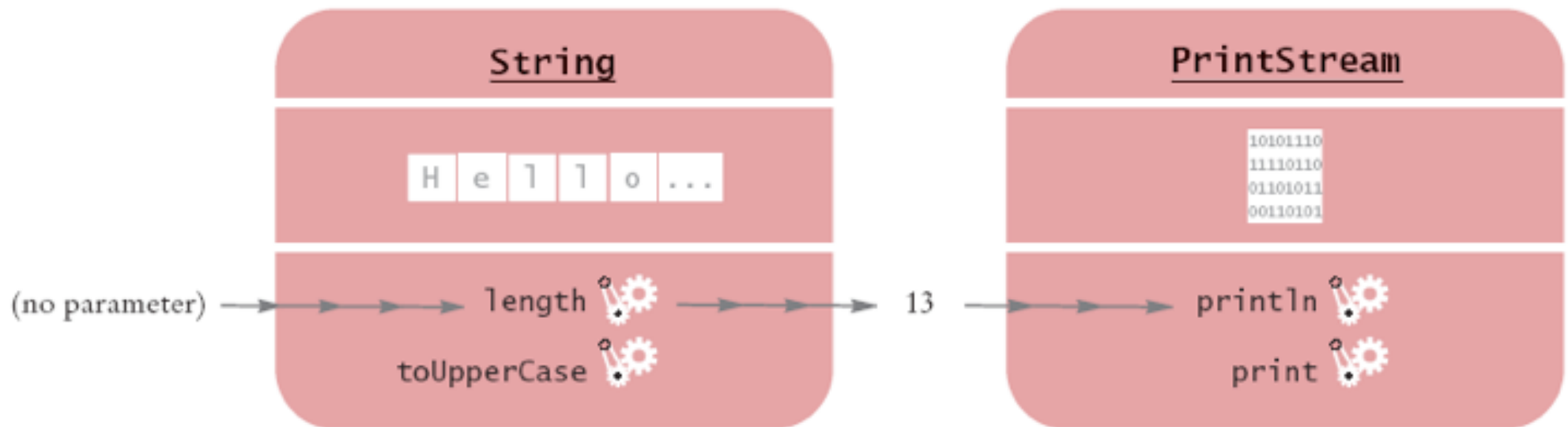
```
int n = greeting.length();  
// retourne une valeur stockée dans n
```



Utilisation d'une valeur de retour

- Une valeur de retour peut être passée en paramètre d'une autre méthode :

```
System.out.println(greeting.length());
```



- Certaines méthodes n'ont pas de valeur de retour : (`println`)

Méthode : définition

- **Spécification :**
 - nom de la méthode
 - type de la valeur de retour
 - types des paramètres explicites
- **Remarque :**
 - le type du receveur n'est pas précisé ; classe courante
- **Exemple :**

```
public int length()  
public String replace(String target,  
                      String replacement)
```

Méthode : définition /2

- Si une méthode ne retourne pas de valeur :
 - type de retour est déclaré comme `void`

```
public void println(String output)  
    //dans la classe PrintStream
```

- Type d'un paramètre explicite ne peut pas être `void`

Méthode : définition : surcharge

- Plusieurs méthodes (ou constructeurs) avec le même nom («*overloading*»)

```
class Point {  
    private int x;  
    private int y;  
  
    public void translate(int xp,int yp) {  
        x= x+xp;  
        y= y+yp;  
    }  
}  
  
    public void translate (Point p) {  
        x += p.x;  
        y += p.y;  
    }  
}
```

- En Java, une méthode est identifiée par :
 - nom, nombre de paramètres et types des paramètres
 - Mais, ne prend pas en compte : type de retour

Appel de méthode

objet.nomDeLaMéthode (paramètres)

- **Exemple :**

```
System.out.println("Hello, Dave!");
```

- **Effets :**

- Invoker une méthode d'un objet et lui passer des paramètres additionnels

Types primitifs : nombre

- Valeur entière :
 - `short, int, long`
 - `13`
- Valeur réelle :
 - `float, double`
 - `1.3`
 - `0.00013`
- Attention : ce ne sont pas des objets
- Classe « *Wrapper* » :
 - `Sort, Integer, Long, Float, Double`

Types primitifs : nombre /2

- Opérateurs arithmétiques :

- $+$, $-$, $*$, $/$, $\%$

- $($, $)$

- Exemples :

- $10 + n$

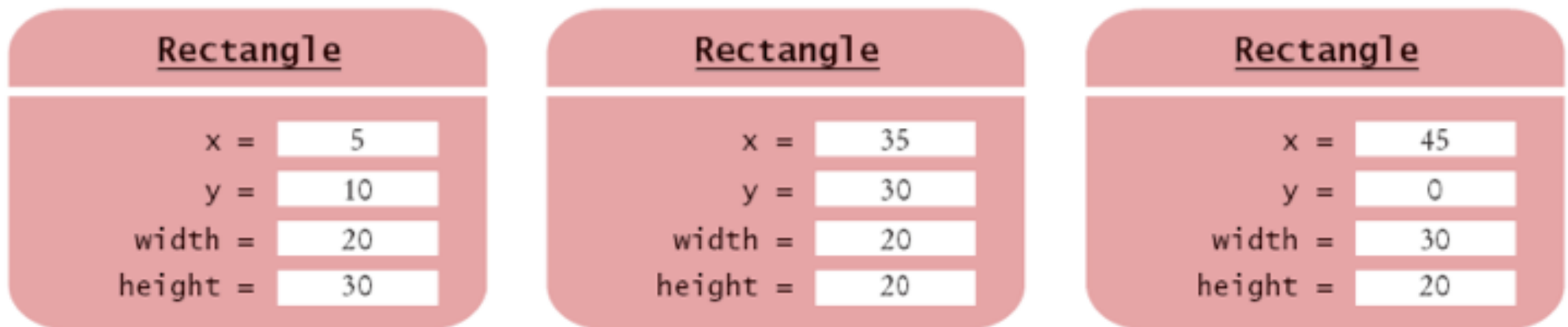
- $n - 1$

- $10 * n$

- Ce ne sont pas des méthodes

Exemple d'objet : Rectangle

- Cette classe représente un `Rectangle` et non pas la figure Rectangle



- 3 objets = 3 instances de la classe `Rectangle`

Constructeurs

- Utilisation :

```
new Rectangle(5, 10, 20, 30)
```

- L'opérateur new :

- construit l'objet de classe Rectangle
- utilise les paramètres pour initialiser les attributs de l'objet
- Retourne le nouvel objet

- Généralement, l'objet est stocké dans une variable :

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

Constructeurs /2

- Créer un objet c'est :
 - Instancier une classe par l'appel d'un constructeur
- Les quatres valeurs 5, 10, 20, et 30 sont les paramètres de construction
- Certaines classes offrent plusieurs constructeurs (*surchage*)

```
new Rectangle()
```

```
new Rectangle(5,10,20,30)
```

Accesseur / Modificateur

- **Accesseur** : ne change pas l'état interne d'un objet (paramètre implicite)

```
double width = box.getWidth();
```

- **Modificateur** : change l'état interne

```
box.translate(15, 25);
```

Exemple

- Ecrire une classe de test (`Test`)
- Fournir une méthode `main`
- Dans cette méthode, construire plusieurs objets
- Appeler des méthodes sur ces objets
- Afficher les résultats que vous escomptez

Exemple : solution

```
01: import java.awt.Rectangle;
02:
03: public class MoveTester
04: {
05:     public static void main(String[] args)
06:     {
07:         Rectangle box = new Rectangle(5, 10, 20, 30);
08:
09:         // Déplacer le rectangle
10:         box.translate(15, 25);
11:
12:         // Afficher les informations concernant le rectangle
13:         System.out.print("x: ");
14:         System.out.println(box.getX());
15:         System.out.println("Expected: 20");
16:
17:         System.out.print("y: ");
18:         System.out.println(box.getY());
19:         System.out.println("Expected: 35");    }
20: }
```

Output:

```
x: 20
Expected: 20
y: 35
Expected: 35
```

Référence (d'un objet)

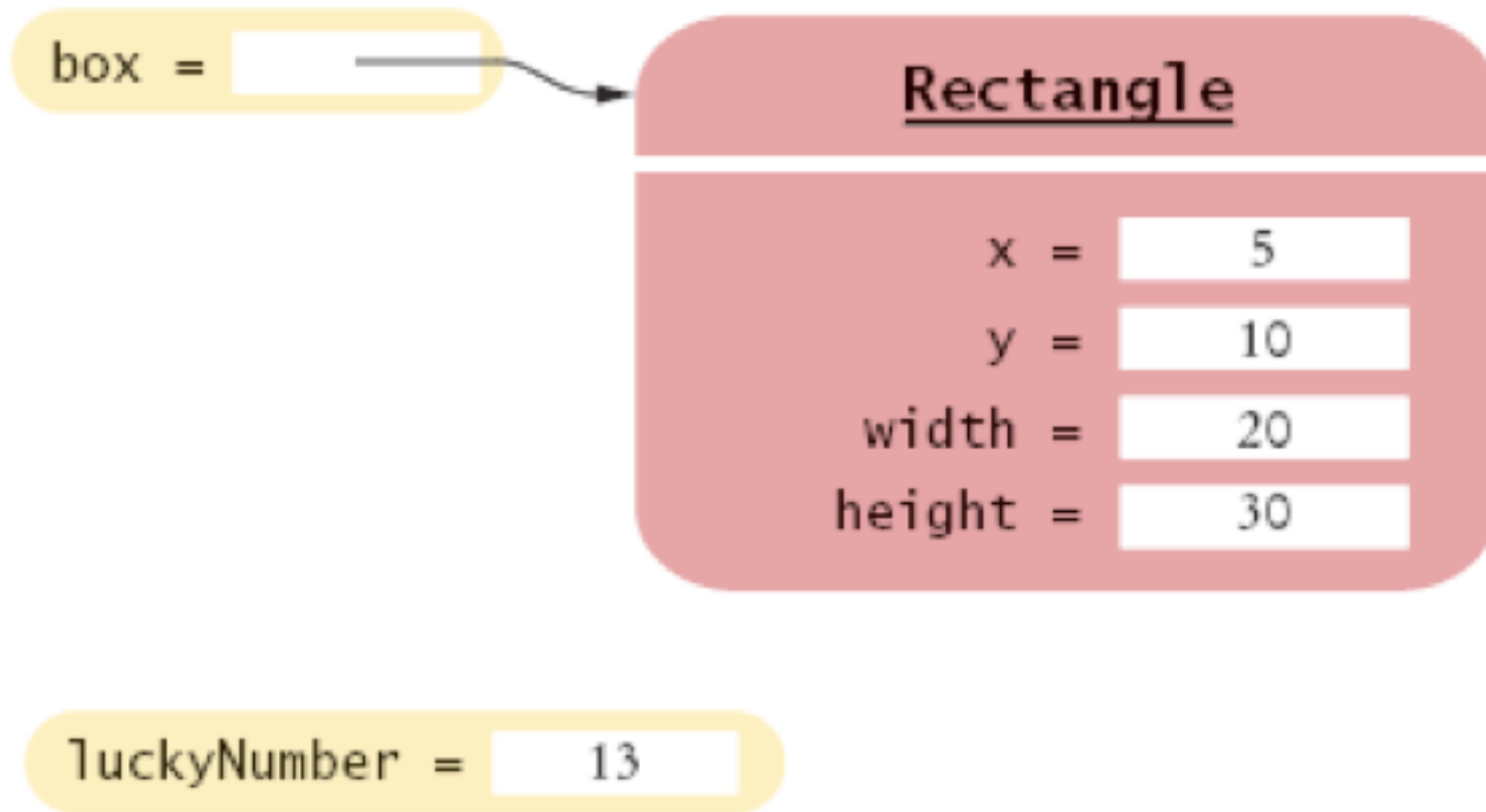
- Référence : décrit la localisation d'un objet
- Opérateur `new` retourne une référence vers un nouvel objet

```
Rectangle box = new Rectangle();
```

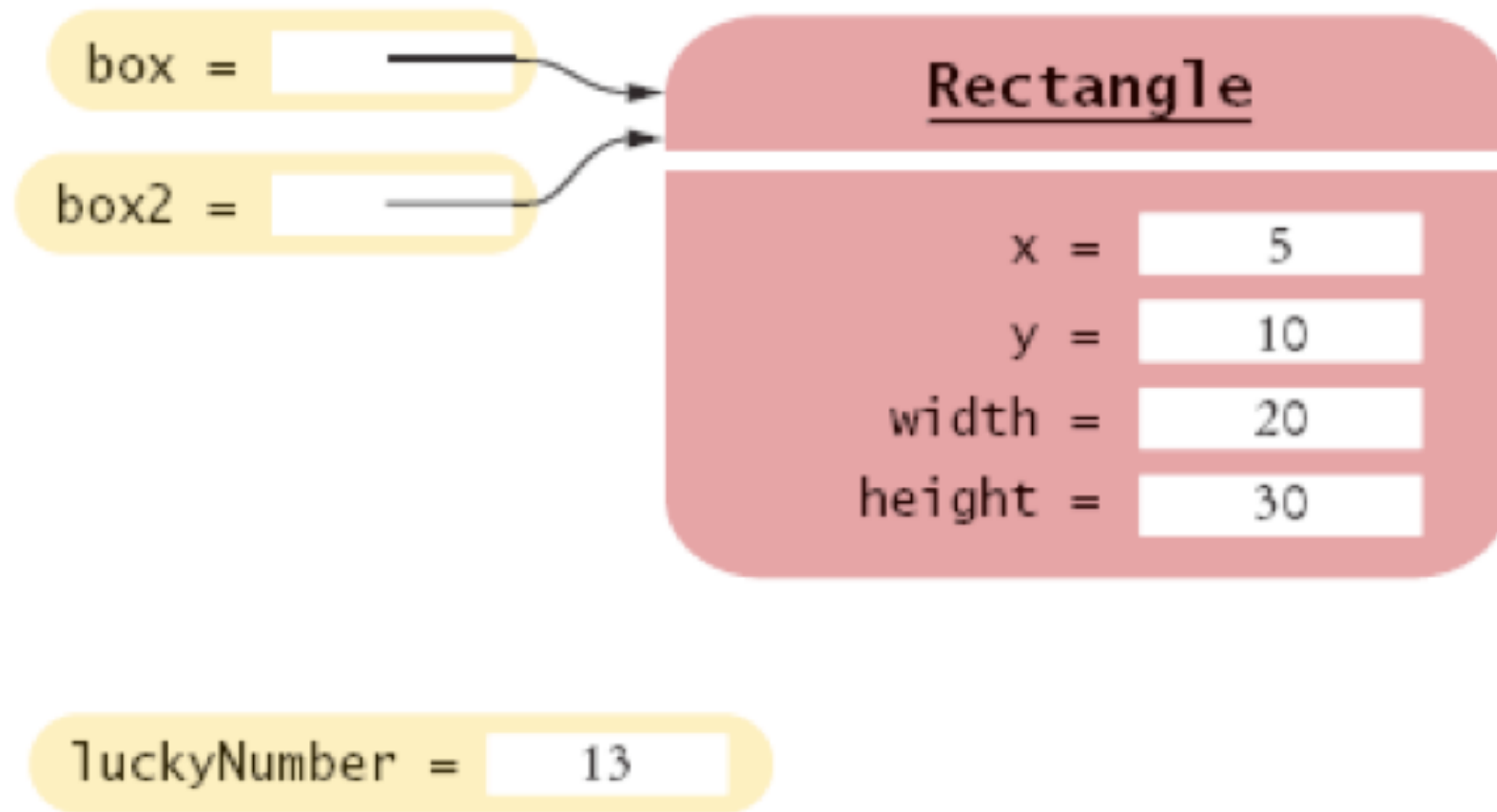
- Plusieurs variables peuvent référencer un même objet

```
Rectangle box = new Rectangle(5, 10, 20, 30);  
Rectangle box2 = box;  
box2.translate(15, 25);
```

Référence /2



Référence /3



Copie d'une valeur

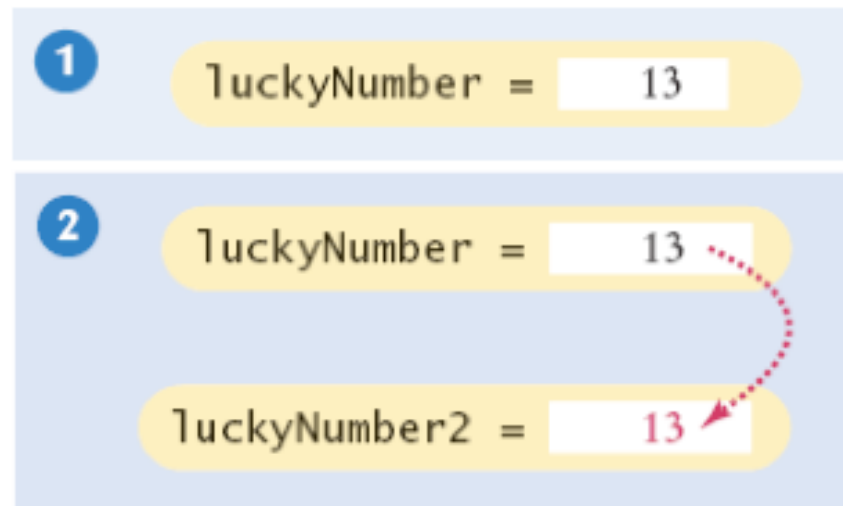
- `int luckyNumber = 13;` ①

①

`luckyNumber = 13`

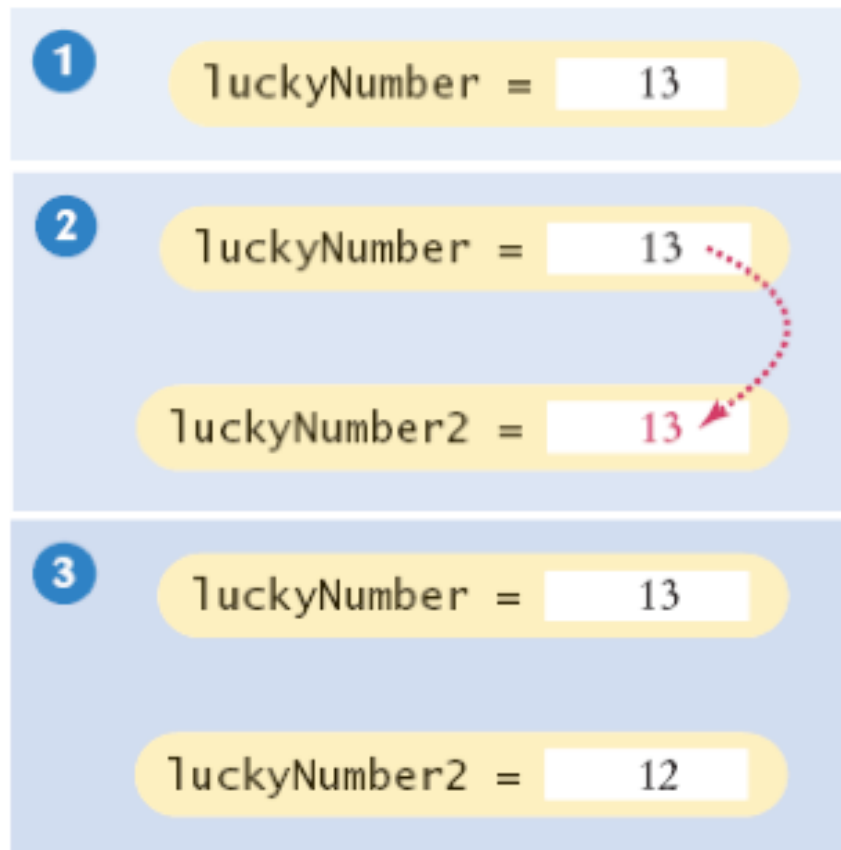
Copie d'une valeur

- `int luckyNumber = 13;` ①
- `int luckyNumber2 = luckyNumber;` ②



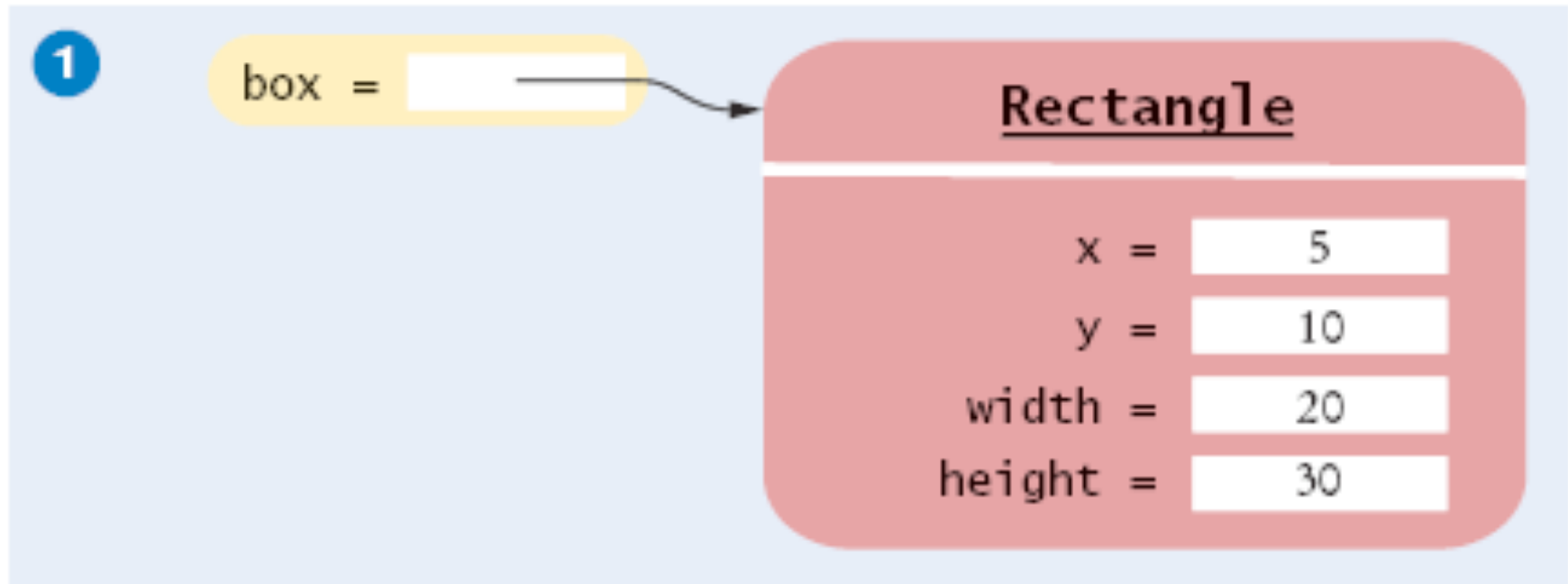
Copie d'une valeur

- `int luckyNumber = 13;` ①
- `int luckyNumber2 = luckyNumber;` ②
- `luckyNumber2 = 12;` ③



Copie d'une référence

```
Rectangle box = new Rectangle(5, 10, 20, 30);
```

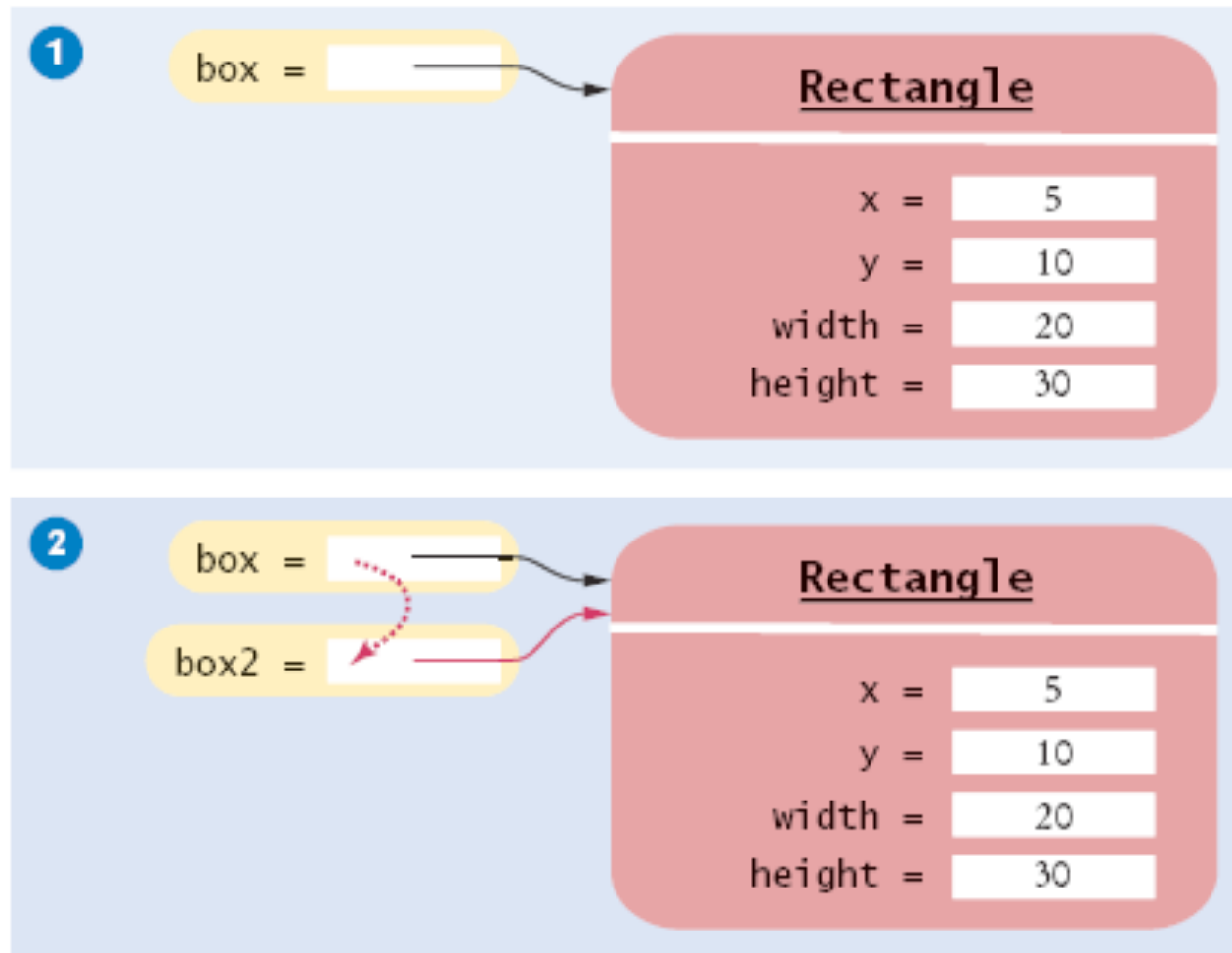


Copie d'une référence

```
Rectangle box = new Rectangle(5, 10, 20, 30);  
Rectangle box2 = box;
```

1

2

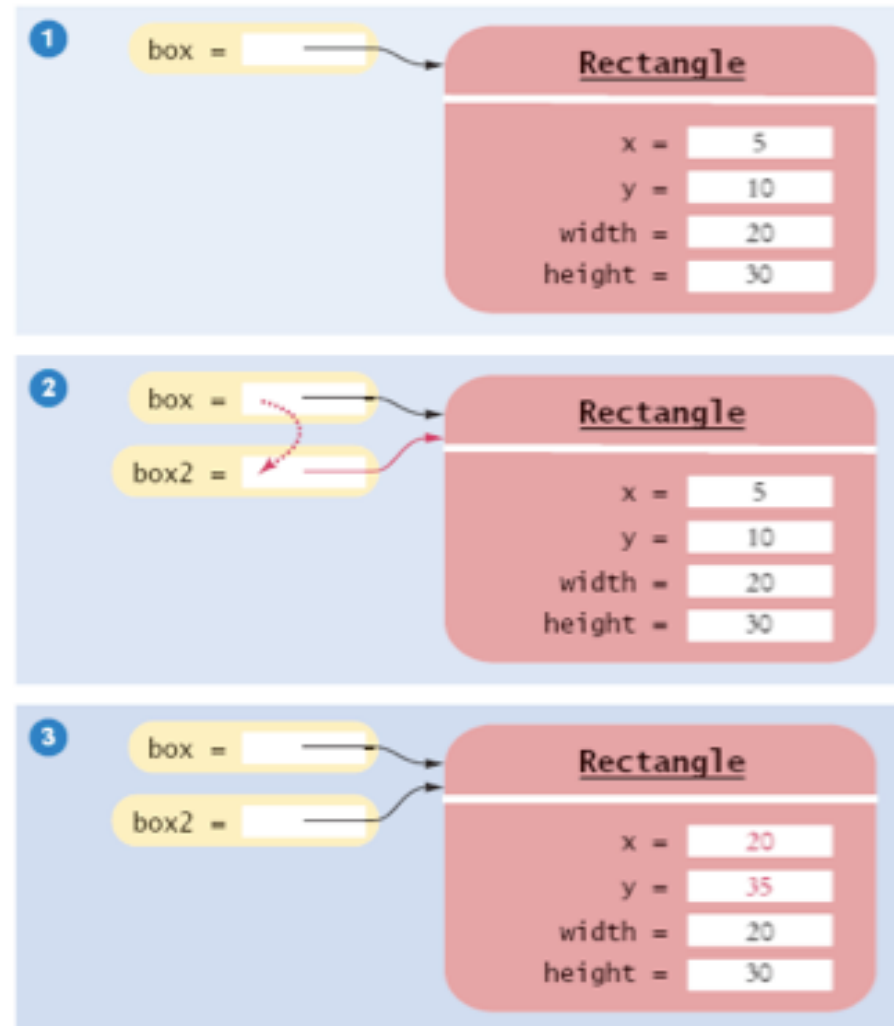


Copie d'une référence

Rectangle box = new Rectangle(5, 10, 20, 30); ①

Rectangle box2 = box; ②

Box2.translate(15, 25); ③



2^{ème} Partie : Conception et Réalisation

Boîtes noires

- Une boîte noire réalise « magiquement » des choses
- Elle cache son fonctionnement interne
- **Encapsulation** : cacher les détails non important
- Quel est le bon *concept* pour chaque boîte noire particulière
- Concepts sont découverts par abstraction
- **Abstraction** : supprimer les fonctions non essentielles tant que l'essence du concept reste présente
- En *programmation orientée objet*, les objets sont les boîtes noires à partir desquels un programme est construit

Niveaux d'abstraction : Génie Logiciel /2

- En génie logiciel, il est possible de concevoir de **bonnes** et de **mauvaises abstractions** offrant des **fonctionnalités identiques** ;
- Comprendre ce qu'est une bonne conception est l'une des enseignements les plus importants qu'un développeur peut apprendre.
- En premier, définir le comportement d'une classe
- Ensuite, implémenter cette classe

Spécifier l'interface publique d'une classe

Comportement d'un compte bancaire (abstraction) :

- déposer de l'argent
- retirer de l'argent
- consulter le solde

Spécifier l'interface publique d'une classe : Méthodes

Méthodes de la classe `BankAccount` :

- `deposit`
- `withdraw`
- `getBalance`

Nous souhaitons utiliser un compte bancaire de la manière suivante :

```
harrysChecking.deposit(2000);  
harrysChecking.withdraw(500);  
System.out.println(harrysChecking.getBalance());
```

Spécifier l'interface publique d'une classe : Définir Méthodes

- Modificateur d'accès (tel que `public`)
- Type de retour (tel que `String` ou `void`)
- Nom de la méthode (tel que `deposit`)
- Liste des paramètres (`double amount` pour `deposit`)
- Corps de la méthode entre `{ }`

Exemples:

- `public void deposit(double amount) { . . . }`
- `public void withdraw(double amount) { . . . }`
- `public double getBalance() { . . . }`

Syntaxe Définition d'une méthode

```
accessSpecifier returnType methodName(parameterType  
parameterName, . . . )  
{  
    method body  
}
```

Exemple :

```
public void deposit(double amount)  
{  
    . . .  
}
```

Objectif :

Définir le comportement d'une méthode.

Spécifier l'interface publique d'une classe : définition constructeurs

- Un constructeur initialise les champs d'une instance
- Nom du constructeur = nom de la classe

```
public BankAccount()  
{  
    // body--filled in later  
}
```

- Le corps du constructeur est exécuté quand un objet est créé
- Les instructions d'un constructeur vont initialiser l'état interne de l'objet en construction
- Tous les constructeurs d'une même classe portent le même nom
- Le compilateur différencie les constructeurs en fonction des paramètres

Syntax 3.2 Définition d'un constructeur

```
accessSpecifier ClassName(parameterType parameterName, . . .)
{
    constructor body
}
```

Exemple :

```
public BankAccount(double initialBalance)
{
    . . .
}
```

Objectif :

Définir le comportement d'un constructeur

BankAccount **Interface publique**

Les construteurs publics et les méthodes publiques d'une classe forme l'*interface publique* d'une classe.

```
public class BankAccount
{
    // Constructors public BankAccount()
    {
        // body--filled in later
    }
    public BankAccount(double initialBalance)
    {
        // body--filled in later
    }
}
```

BankAccount Interface publique /2

```
// Methods
public void deposit(double amount)
{
    // body--filled in later
}
public void withdraw(double amount)
{
    // body--filled in later
}
public double getBalance()
{
    // body--filled in later
}
// private fields--filled in later
}
```

Syntaxe Définition d'une classe

```
accessSpecifier class ClassName
{
    constructors
    methods
    fields
}
```

Exemple :

```
public class BankAccount
{
    public BankAccount(double initialBalance) {. . .}
    public void deposit(double amount) {. . .}
    . . .
}
```

Objectif :

Pour définir une classe, il faut écrire son interface publique et ses détails d'implémentation.

Commenter une interface publique

```
/**
    Withdraws money from the bank account.
    @param the amount to withdraw
 */
public void withdraw(double amount)
{
    //implementation filled in later
}

/**
    Gets the current balance of the bank account.
    @return the current balance
 */
public double getBalance()
{
    //implementation filled in later
}
```

Commenter une classe

```
/**
```

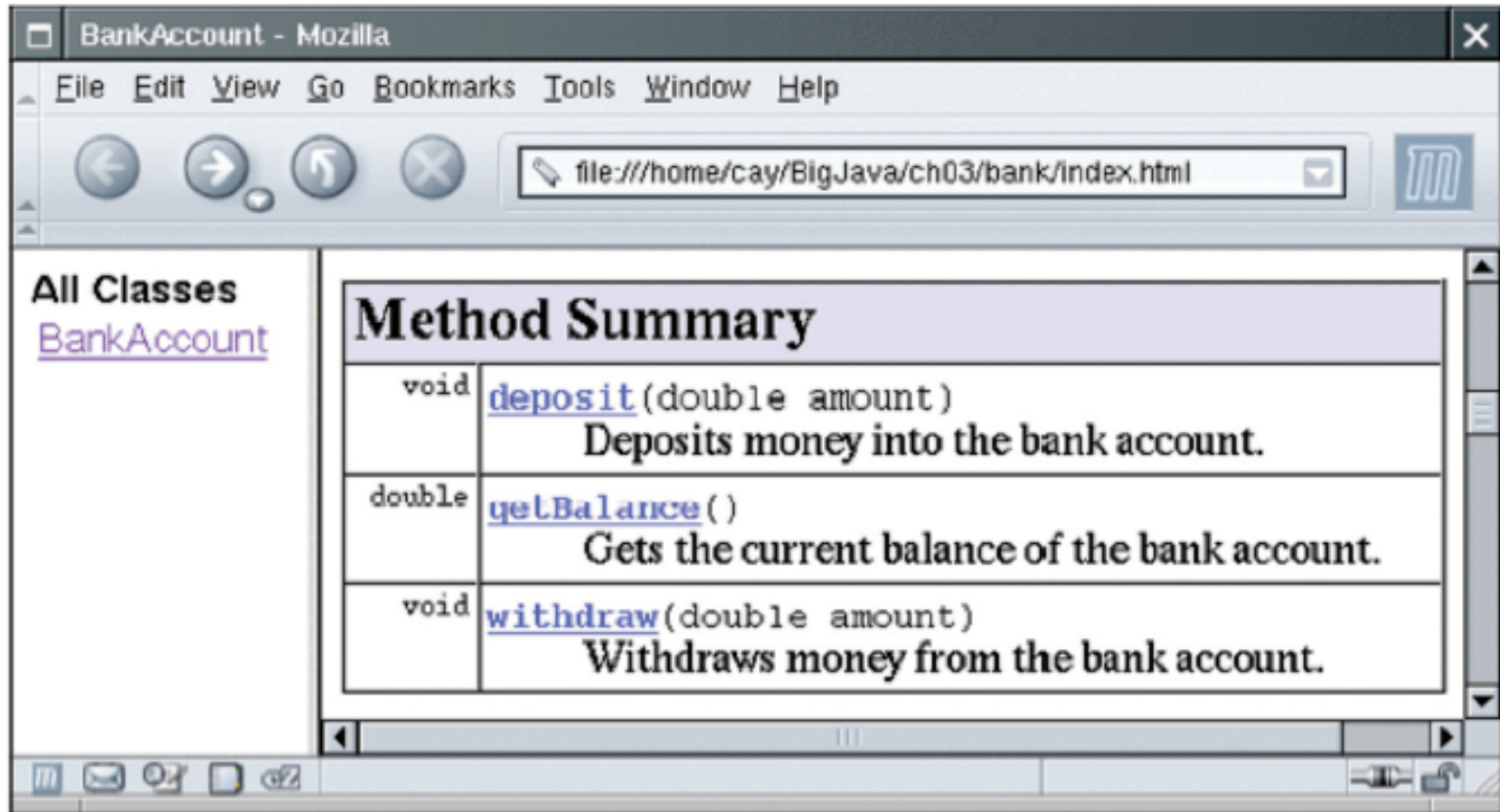
```
    A bank account has a balance that can be changed by  
    deposits and withdrawals.
```

```
*/
```

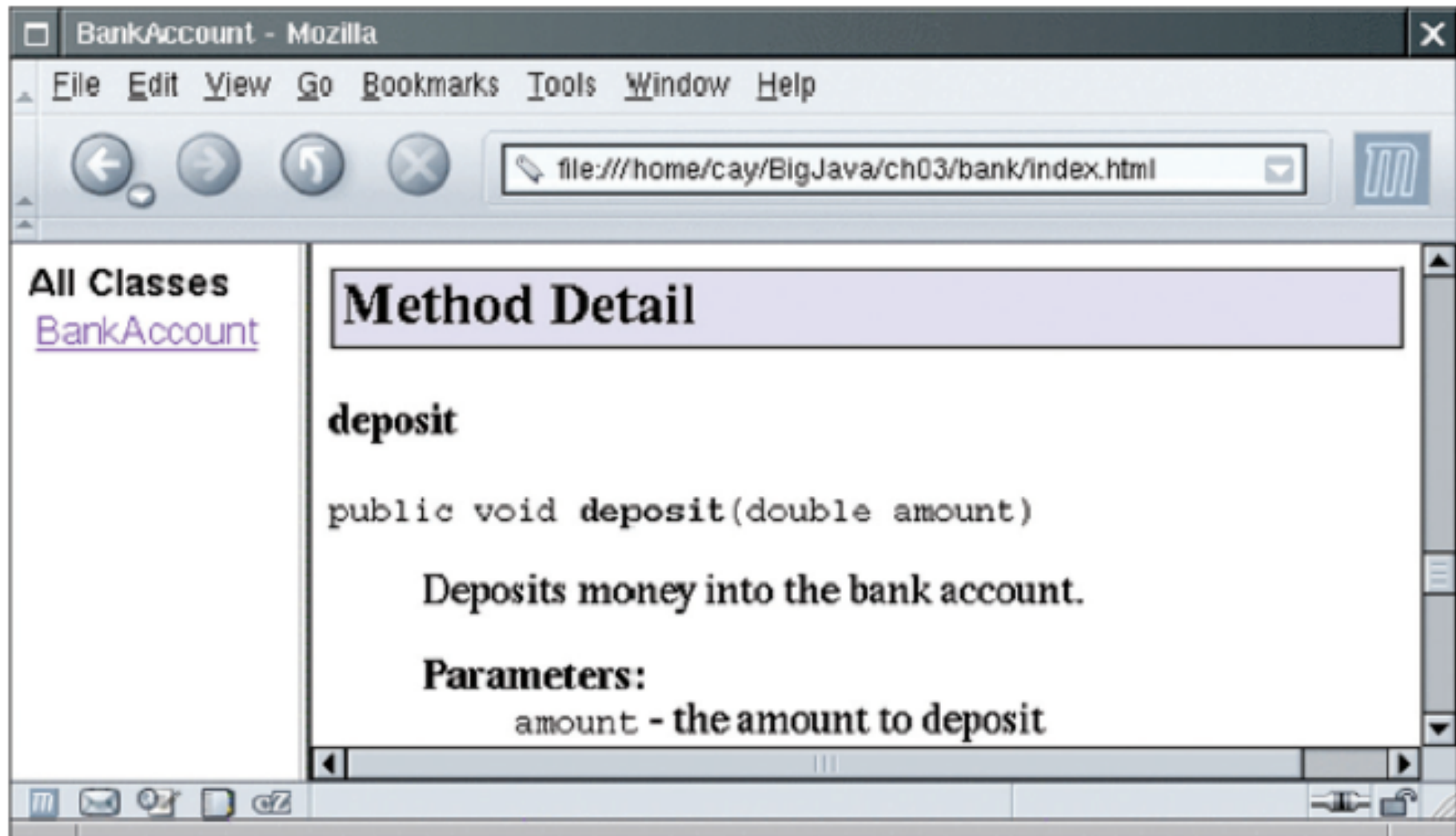
```
public class BankAccount  
{  
    . . .  
}
```

- Il faut documenter :
 - *chaque classe*
 - *chaque méthode*
 - *chaque paramètre*
 - *chaque valeur de retour.*

JavaDoc – Documentation des méthodes



JavaDoc – Documentation d'une méthode



Variable/Champs d'une instance

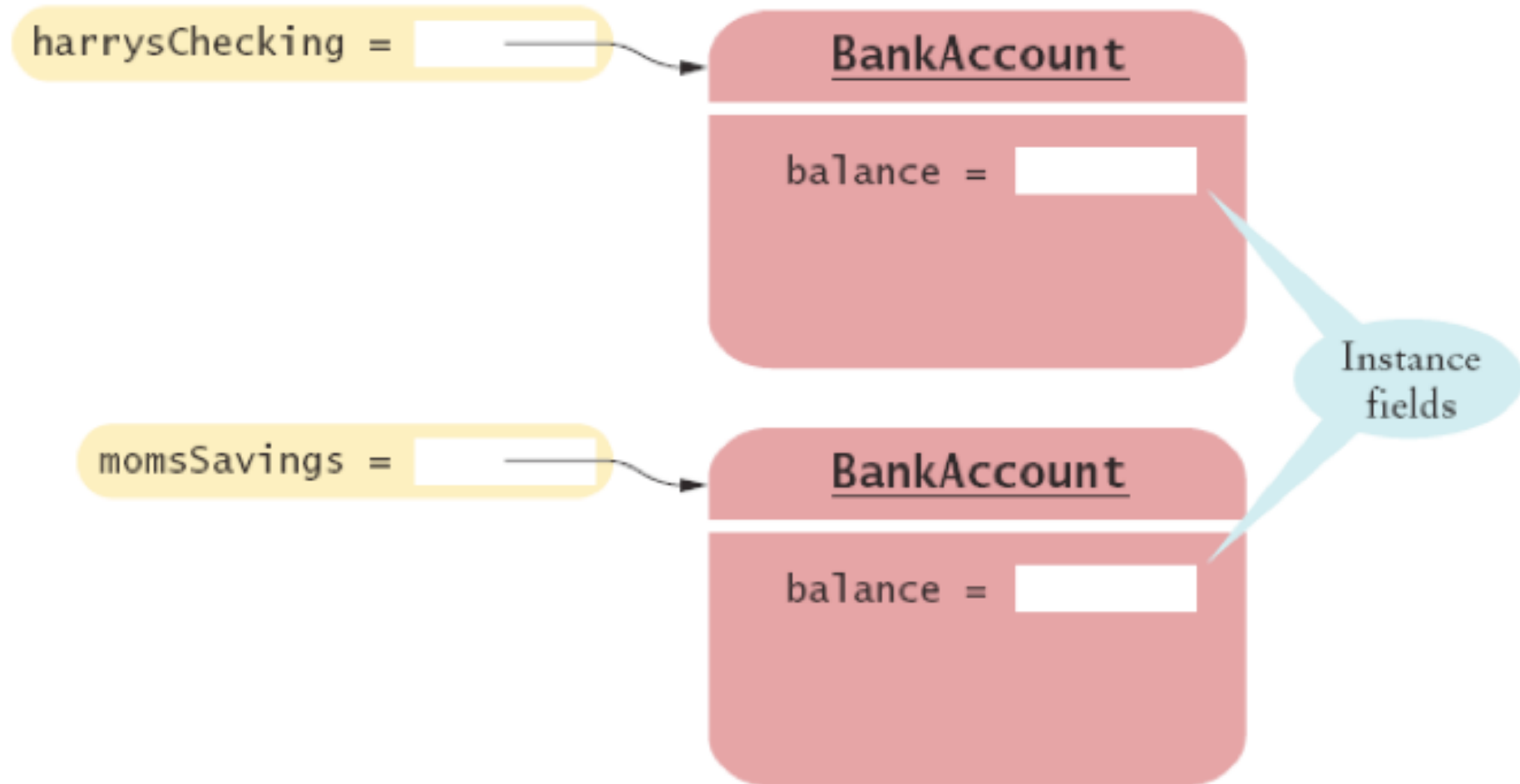
- Un objet stocke ses données dans des champs d'instance
- Champ : un terme technique pour désigner le stockage de la localisation d'un bloc de mémoire
- Instance d'une classe : un objet d'une classe
- La déclaration d'une classe spécifie les variables d'instance publiques

```
BankAccount
{
    . . .
    private double balance;
}
```

Variable/Champs d'une instance /2

- La déclaration d'un champ d'instance comporte :
 - *Un modificateur d'accès (généralement `private`)*
 - *le type de la variable (tel que `double`)*
 - *le nom de la variable (tel que `as balance`)*
- Chaque objet d'une classe possède son propre ensemble de champs d'instance
- Généralement, vous devez déclarer les variables d'instance comme privées

Variable/Champs d'une instance /3



Syntaxe Déclaration des variables d'une instance

```
accessSpecifier class ClassName
{
    . . .
    accessSpecifier fieldType fieldName;
    . . .
}
```

Exemple :

```
public class BankAccount
{
    . . .
    private double balance;
    . . .
}
```

Objectif :

Pour définir un champ qui est présent dans chaque objet d'une classe

Accéder aux variables d'instance

- La méthode `deposit` de la classe `BankAccount` peut accéder aux variables d'instance privées :

```
public void deposit(double amount)
{
    double newBalance = balance + amount;
    balance = newBalance;
}
```

Accéder aux variables d'instance /2

- Les autres ne sont pas autorisées :

```
public class BankRobber
{
    public static void main(String[] args)
    {
        BankAccount momsSavings = new BankAccount(1000);
        . . .
        momsSavings.balance = -1000; // ERROR
    }
}
```

- *Encapsulation* cache les données d'un objet et donne accès au données par des méthodes
- Pour encapsuler des données d'une instance, déclarer les données comme `private` et définir des méthodes publiques qui accèdent à ces données

Implémentation des constructeurs

- Les constructeurs contiennent les instructions pour initialiser les variables d'instance d'un objet

```
public BankAccount()  
{  
    balance = 0;  
}  
public BankAccount(double initialBalance)  
{  
    balance = initialBalance;  
}
```

Exemple d'appel d'un constructeur

- `BankAccount harrysChecking = new BankAccount(1000);`
 - *Crée un nouvel objet de type `BankAccount`*
 - *Appel le second constructeur (puisque un paramètre est fourni)*
 - *Défini le paramètre `initialBalance` à 1000*
 - *Initialise la variable d'instance `balance` du nouvel objet crée égale à la valeur `initialBalance`*
 - *Retourne une référence vers un objet, qui est la localisation de l'objet dans la mémoire , comme la valeur de l'expression `new`*
 - *Stocke la référence dans la variable `harrysChecking`*

Implémentation des méthodes

- Certaines méthodes ne retournent pas de valeur

```
public void withdraw(double amount)
{
    double newBalance = balance - amount;
    balance = newBalance;
}
```

- D'autres retournent une valeur de retour

```
public double getBalance()
{
    return balance;
}
```

Exemple d'appel d'une méthode

- `harrysChecking.deposit(500);`
 - *Défini le paramètre `amount` à 500*
 - *Récupère le champ `balance` de l'objet dont la localisation est stockée dans `harrysChecking`*
 - *Ajoute la valeur de `amount` à la valeur de `balance` et stocke cette valeur dans la variable `newBalance`*
 - *Stocke la valeur `newBalance` dans la variable d'instance `balance` en écrasant l'ancienne valeur*

Syntaxe L'instruction `return`

```
return expression;  
or  
return;
```

Exemple :

```
return balance;
```

Objectif :

Spécifie la valeur qu'une méthode doit retourner. Arrête immédiatement l'exécution de la méthode. La valeur retournée devient la valeur de l'expression d'appel.

BankAccount.java

```
01: /**
02:     A bank account has a balance that can be changed by
03:     deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:     /**
08:         Constructs a bank account with a zero balance.
09:     */
10:     public BankAccount()
11:     {
12:         balance = 0;
13:     }
14:
15:     /**
16:         Constructs a bank account with a given balance.
17:         @param initialBalance the initial balance
18:     */
19:     public BankAccount(double initialBalance)
20:     {
21:         balance = initialBalance;
22:     }
23:
```

BankAccount.java /2

```
24:    /**
25:        Deposits money into the bank account.
26:        @param amount the amount to deposit
27:    */
28:    public void deposit(double amount)
29:    {
30:        double newBalance = balance + amount;
31:        balance = newBalance;
32:    }
33:
34:    /**
35:        Withdraws money from the bank account.
36:        @param amount the amount to withdraw
37:    */
38:    public void withdraw(double amount)
39:    {
40:        double newBalance = balance - amount;
41:        balance = newBalance;
42:    }
43:
44:    /**
45:        Gets the current balance of the bank account.
46:        @return the current balance
47:    */
```

BankAccount.java /3

```
48:     public double getBalance()  
49:     {  
50:         return balance;  
51:     }  
52:  
53:     private double balance;  
54: }
```

Test unitaire

- *Test unitaire* : vérifie que la classe s'exécute correctement en isolation, hors de tout programme.
- Pour tester une classe, utilisez un environnement interactif de test ou écrivez une classe de test.
- *Class de test*: une classe dont la méthode main contient des instructions pour tester une autre classe.
- Généralement, l'exécution d'une classe de test :
 1. *Construire un ou plusieurs objets de la classe à tester*
 2. *Appeler une ou plusieurs méthodes*
 3. *Afficher un ou plusieurs résultats (comparer le résultat attendu)*

BankAccountTester.java

```
01: /**
02:     A class to test the BankAccount class.
03: */
04: public class BankAccountTester
05: {
06:     /**
07:         Tests the methods of the BankAccount class.
08:         @param args not used
09:     */
10:     public static void main(String[] args)
11:     {
12:         BankAccount harrysChecking = new BankAccount();
13:         harrysChecking.deposit(2000);
14:         harrysChecking.withdraw(500);
15:         System.out.println(harrysChecking.getBalance());
16:         System.out.println("Expected: 1500");
17:     }
18: }
```

Output:

1500

Expected: 1500

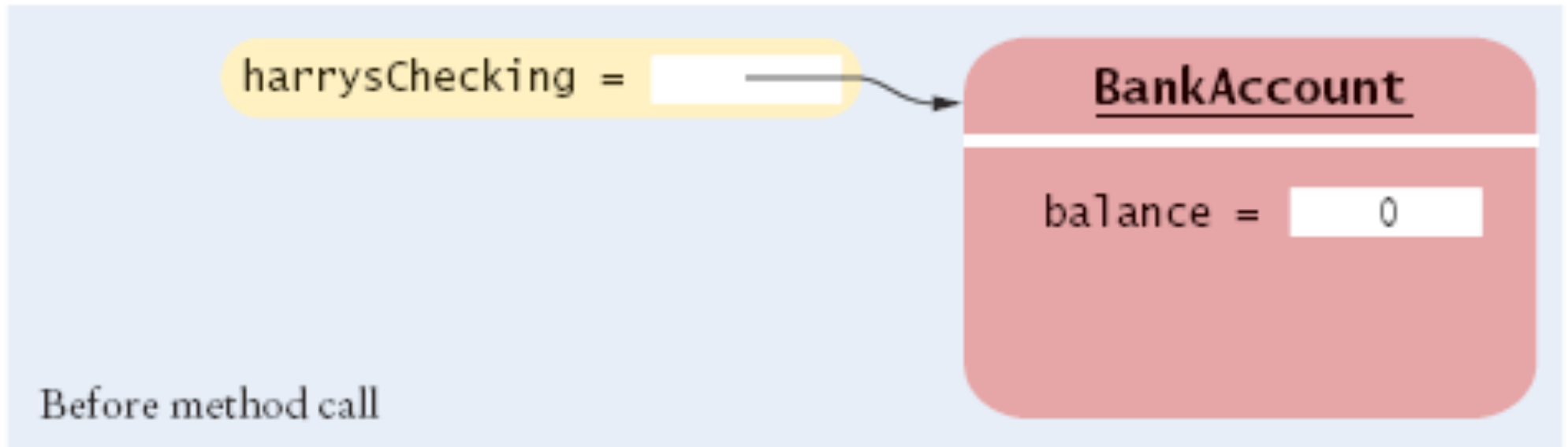
Différentes catégories de variables

- Catégories de variables
 1. Variables d'instance (*balance* dans *BankAccount*)
 2. Variables locales (*newBalance* dans la méthode *deposit*)
 3. Paramètres (*amount* dans la méthode *deposit*)
- Une variable `instance` appartient à un objet
- Les variables d'instance restent en vie jusqu'à ce que plus aucune méthode utilise cet objet
- En Java, le *ramasse miette* (*garbage collector*) collecte les objets qui ne sont plus utilisés
- Variables locales et paramètres appartiennent à une méthode
- Variables d'instance sont initialisées à une valeur par défaut, mais les variables locales doivent être initialisées

Animation 3.1

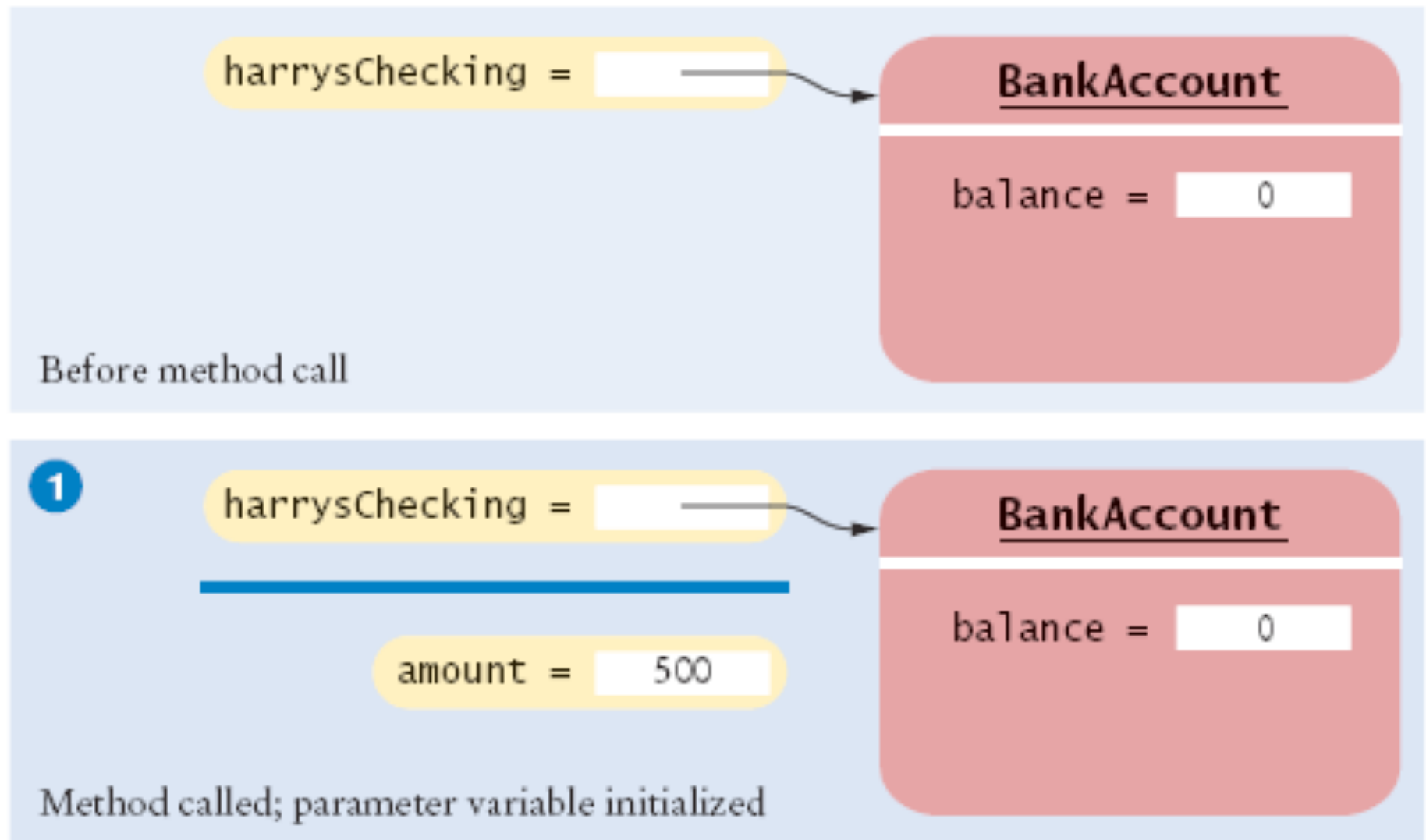
Cycle de vie des Variables – Appel de la méthode `deposit`

```
harrysChecking.deposit(500);
```



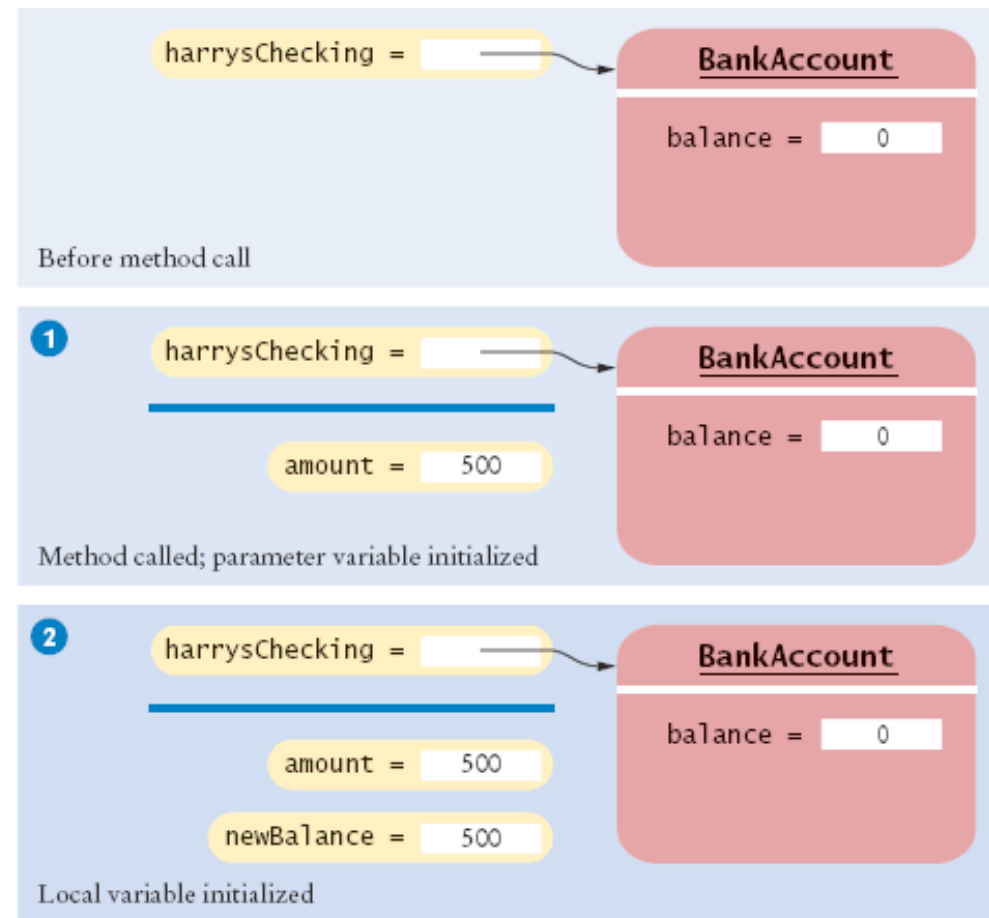
Cycle de vie des Variables – Appel de la méthode `deposit`

`harrysChecking.deposit(500);` ①



Cycle de vie des Variables – Appel de la méthode `deposit`

```
harrysChecking.deposit(500); ①  
double newBalance = balance + amount; ②
```



Cycle de vie des Variables – Appel de la méthode `deposit`

```
harrysChecking.deposit(500); ①  
double newBalance = balance + amount; ②  
balance = newBalance; ③
```

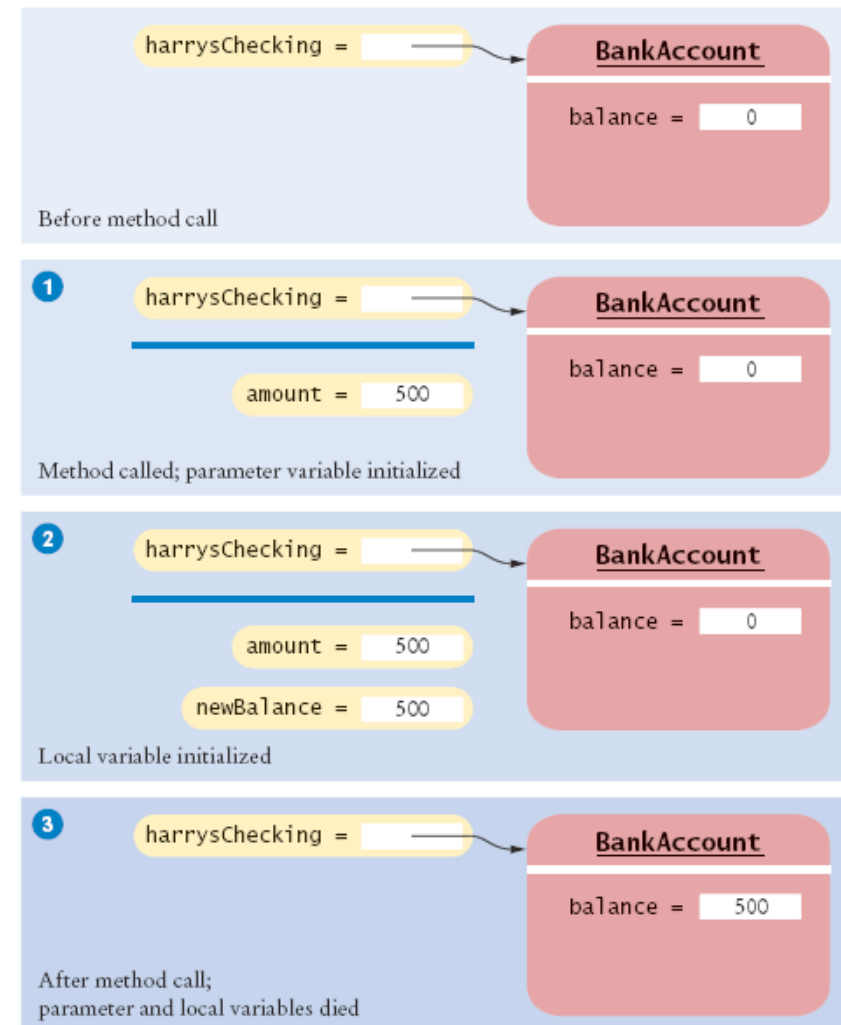


Figure 7 Lifetime of Variables

Paramètres implicite et explicites d'une méthode

- Le paramètre implicite d'une méthode est l'objet sur lequel la méthode est invoquée
- La référence `this` dénote le paramètre implicite (*receveur*)
- L'utilisation d'une variable d'instance dans une méthode dénote la variable d'instance du paramètre implicite

```
public void withdraw(double amount)
{
    double newBalance = balance - amount;
    balance = newBalance;
}
```

Paramètres implicite et explicites d'une méthode /2

- `balance` est le solde de l'objet à gauche du point :

```
momsSavings.withdraw(500)
```

signifie

```
double newBalance = momsSavings.balance - amount;  
>momsSavings.balance = newBalance;
```

Paramètre implicite et `this`

- Chaque méthode à un paramètre implicite (*receveur*)
- Le paramètre implicite est toujours appelé `this`
- Exception : Les méthodes de classes n'ont pas de paramètre implicite (voir suite du cours)
- ```
double newBalance = balance + amount;
// actually means
double newBalance = this.balance + amount;
```
- Quand vous faites référence à une variable d'instance dans une méthode, le compilateur applique automatiquement au paramètre `this`

```
momsSavings.deposit(500);
```

## Paramètre implicite et `this`

---

