TELECOM nancy
Ingénieurs du numérique • Inspiring your digital future

UNIVERSITÉ DE LORRAINE

IA – 2014 / 2015

Programmation Orientée Objet

Cours 3

Gérald Oster <oster@loria.fr>

Plan du cours

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

7ème Partie : Introduction à la conception objets

2ème 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
- Encapsulation : Programmer en connaissant le comportement d'un objet et non pas sa structure interne

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 executé quand un objet est crée
- 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

Syntaxe 3.2 Définition d'un constucteur

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

Exemple :

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

Objectif :

Définir le comportement d'un constructeur

`BankAccount` Interface publique

Les construsteurs publiques 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

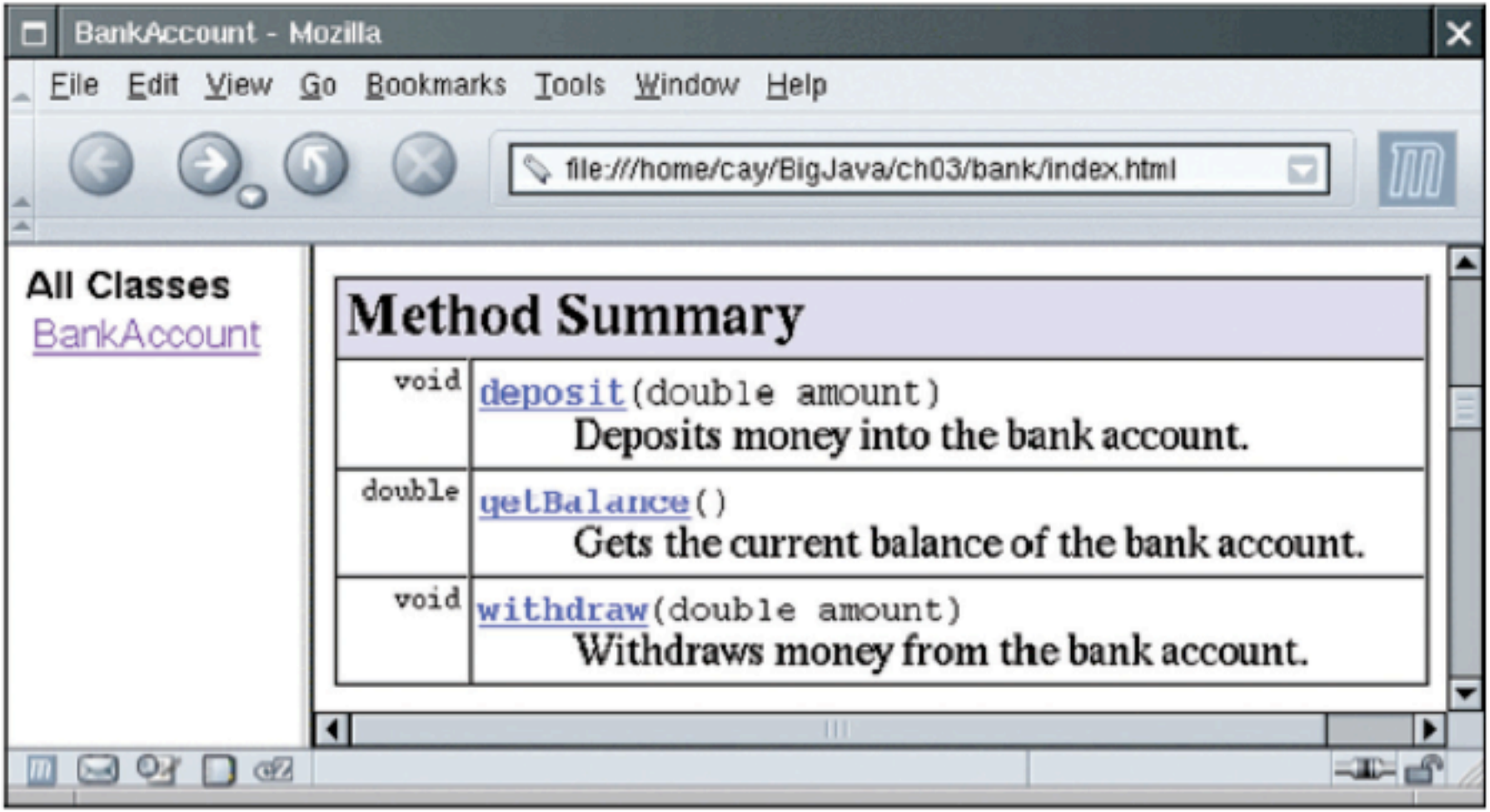
```
/**
   Withdraws money from the bank account.
   @param the amount to withdraw
*/
public void withdraw(double amount)
{
   //implementation filled in later
}
/**
   Gets the current balance of the bank account.
   @return the current balance
*/
public double getBalance()
{
   //implementation filled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

Commenter une classe

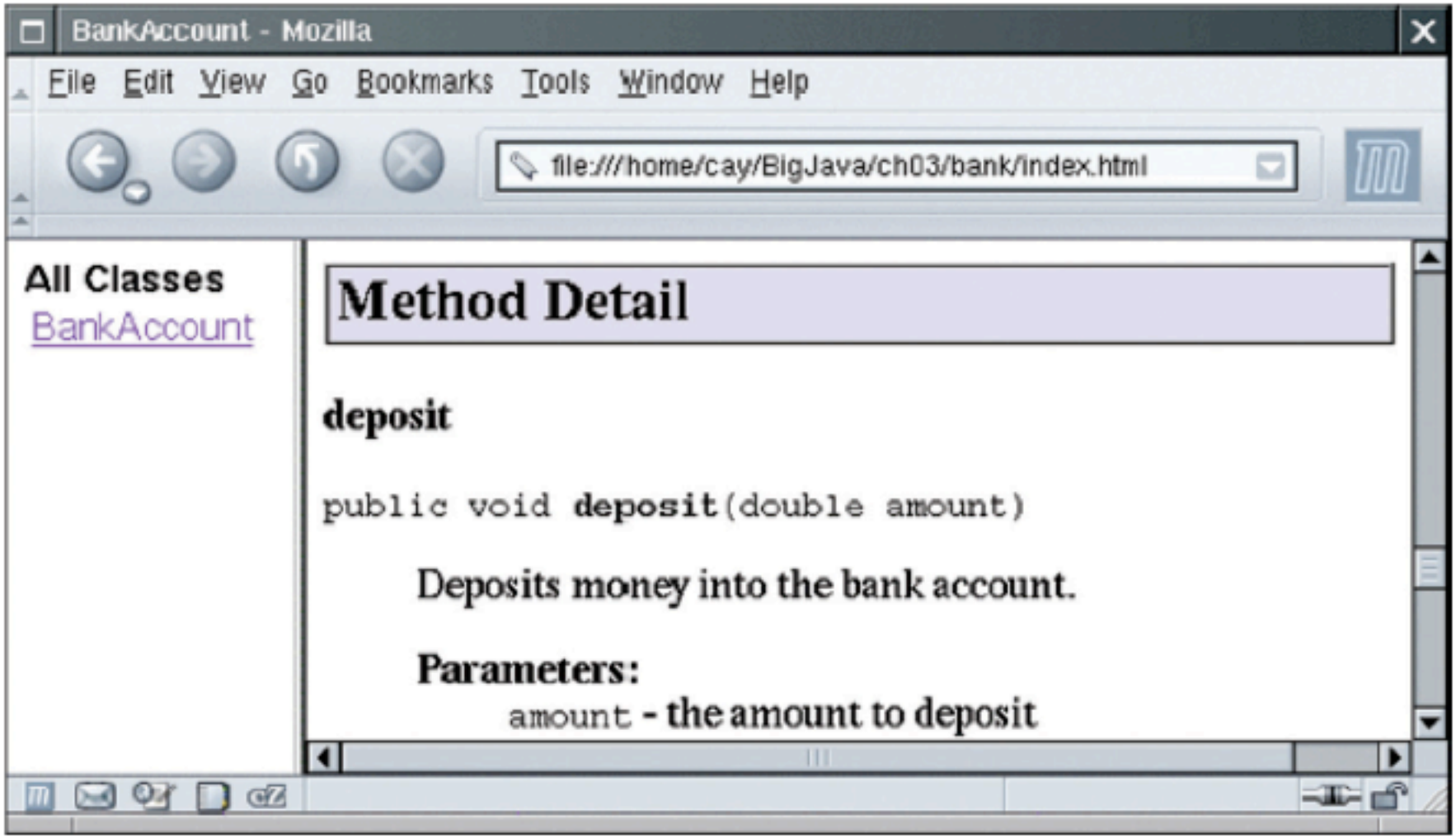
```
/**
   A bank account has a balance that can be changed by
   deposits and withdrawals.
   */
   public class BankAccount
   {
      . . .
   }
```

- 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

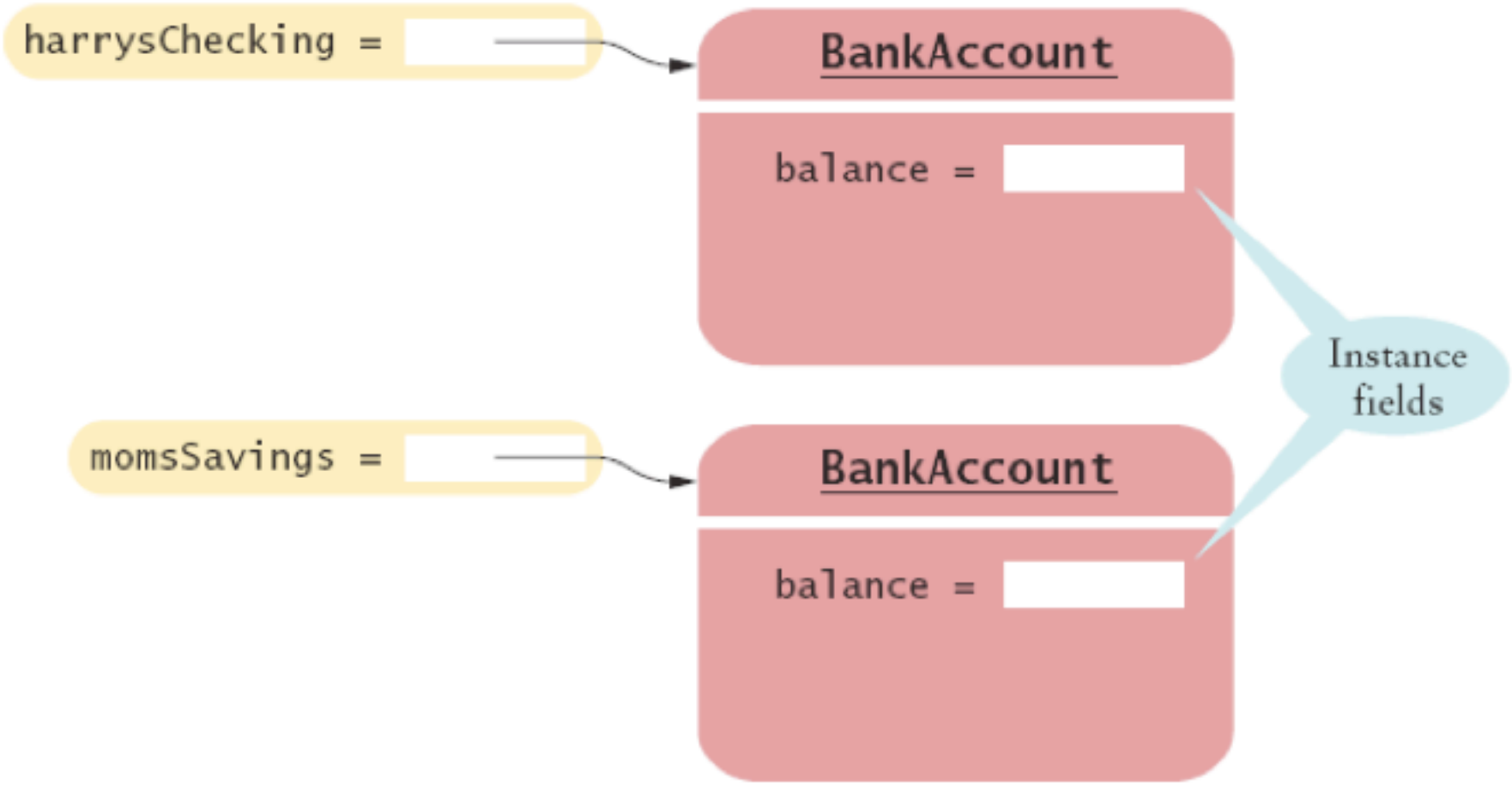
- Un objet stocke ses données dans des champs d'instance
- Champ : un terme technique pour désigner le stockage de la localisation d'un bloc de mémoire
- Instance d'une classe : un objet d'une classe
- La déclaration d'une classe spécifie les variables d'instance publiques

```
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

Accèder aux variables d'instance

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

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

Accèder 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: }
```

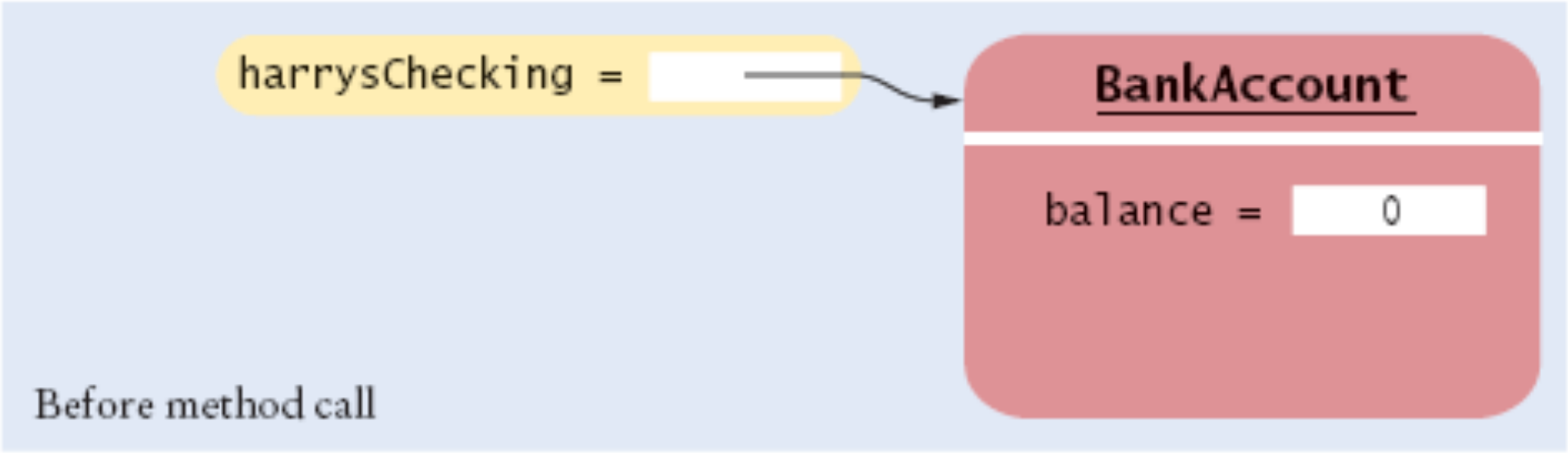
Output:
1500
Expected: 1500

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

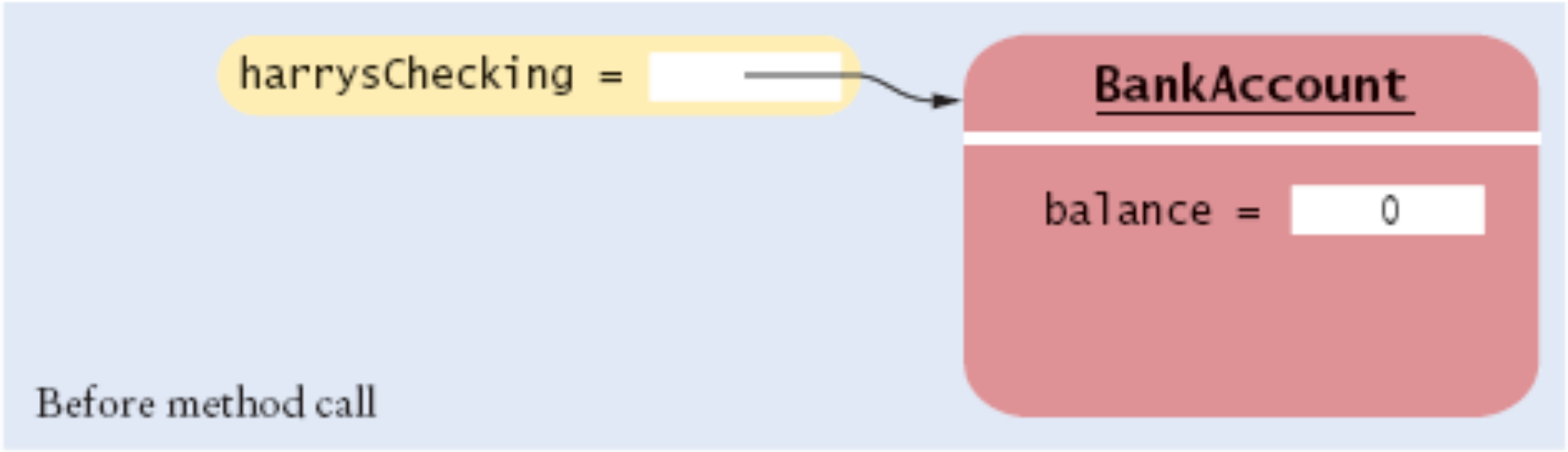
Cycle de vie des Variables – Appel de la méthode `deposit`

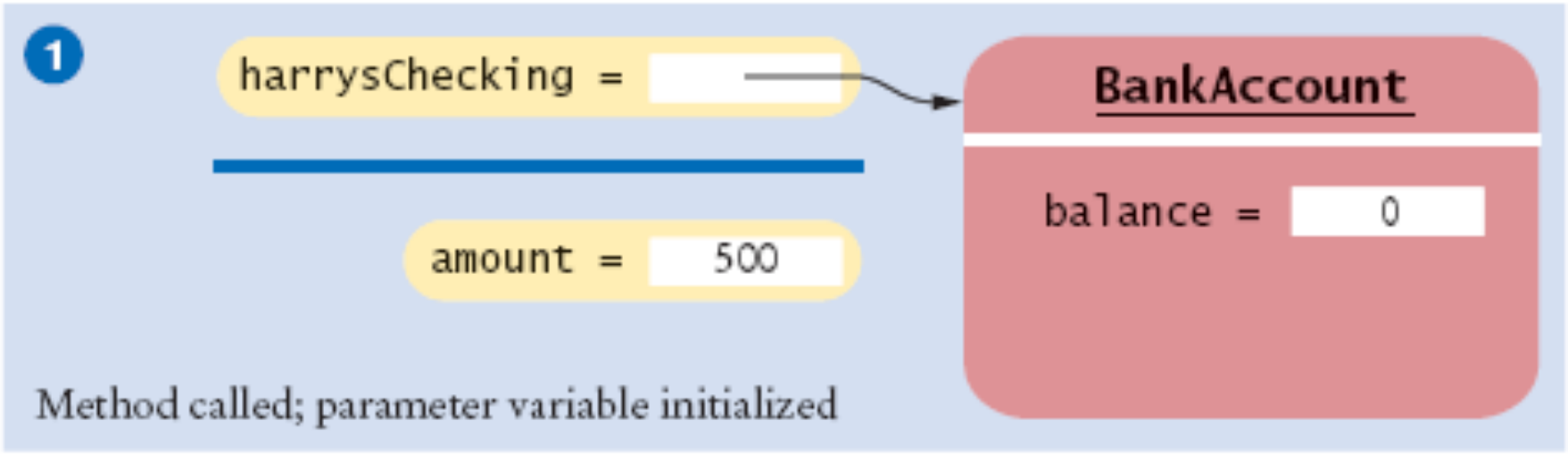
```
harrysChecking.deposit(500);
```



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

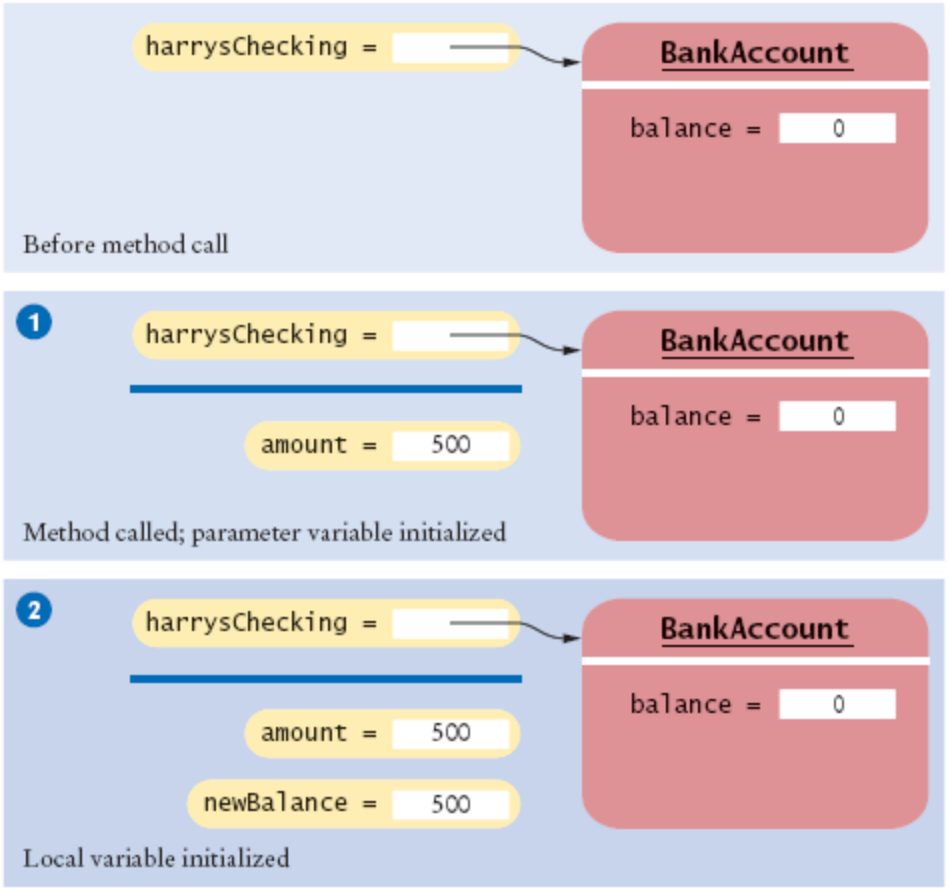
```
harrysChecking.deposit(500);
```
❶





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
```

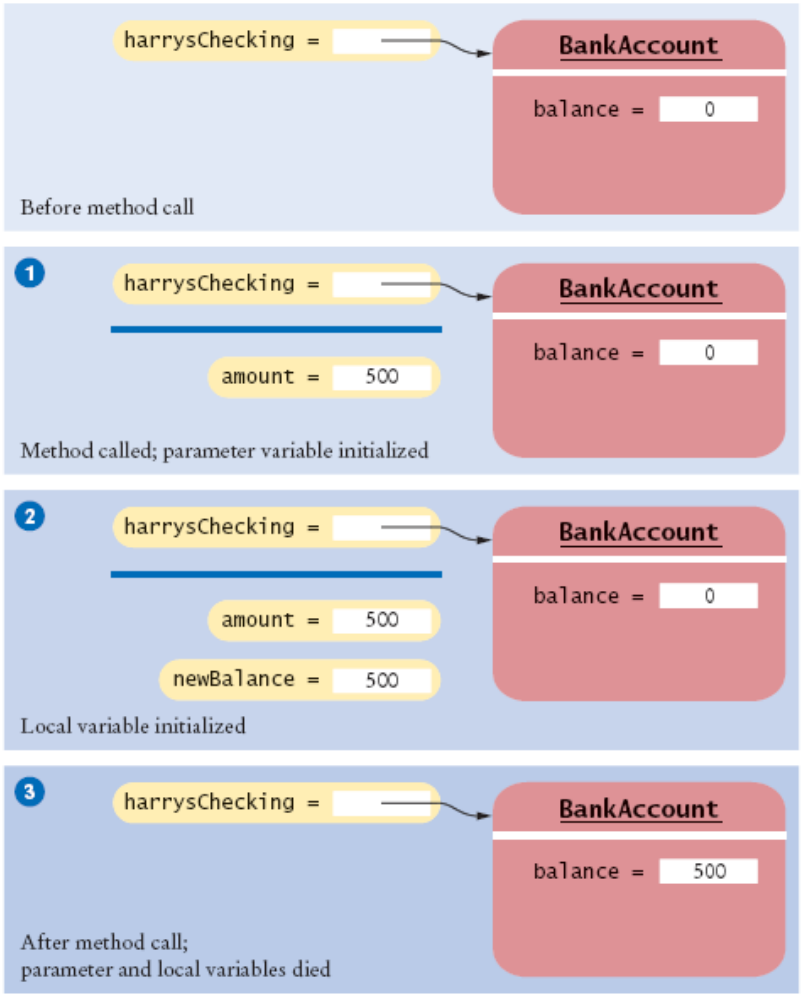


Figure 7 Lifetime of Variables

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 :

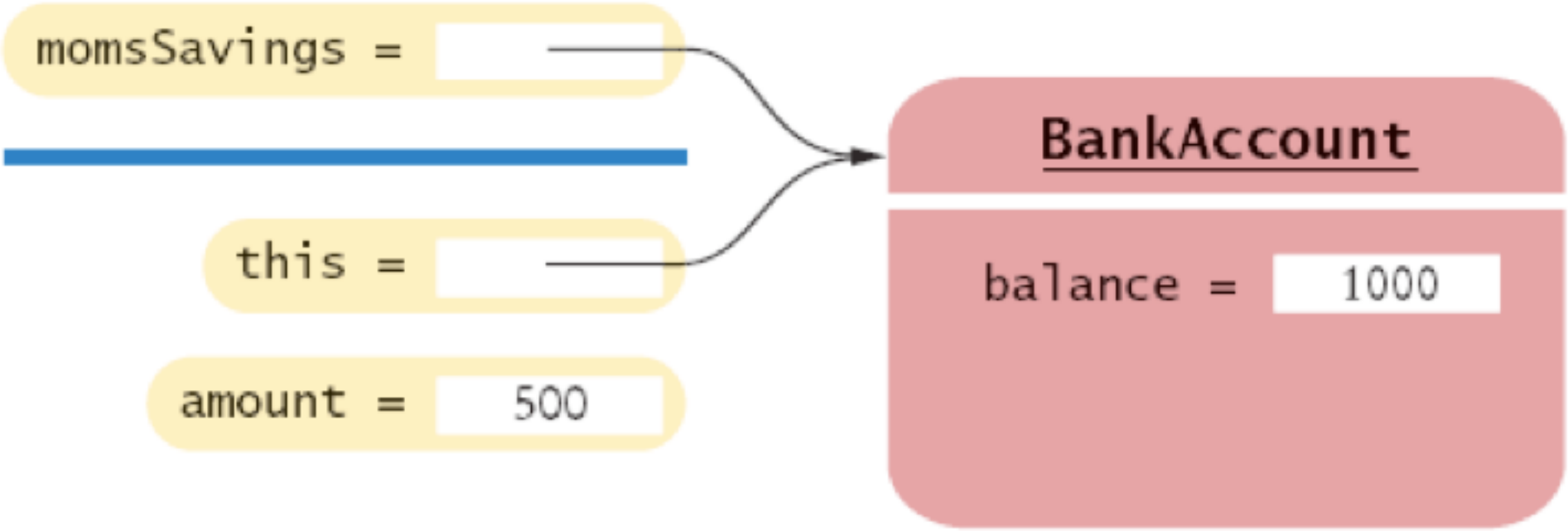
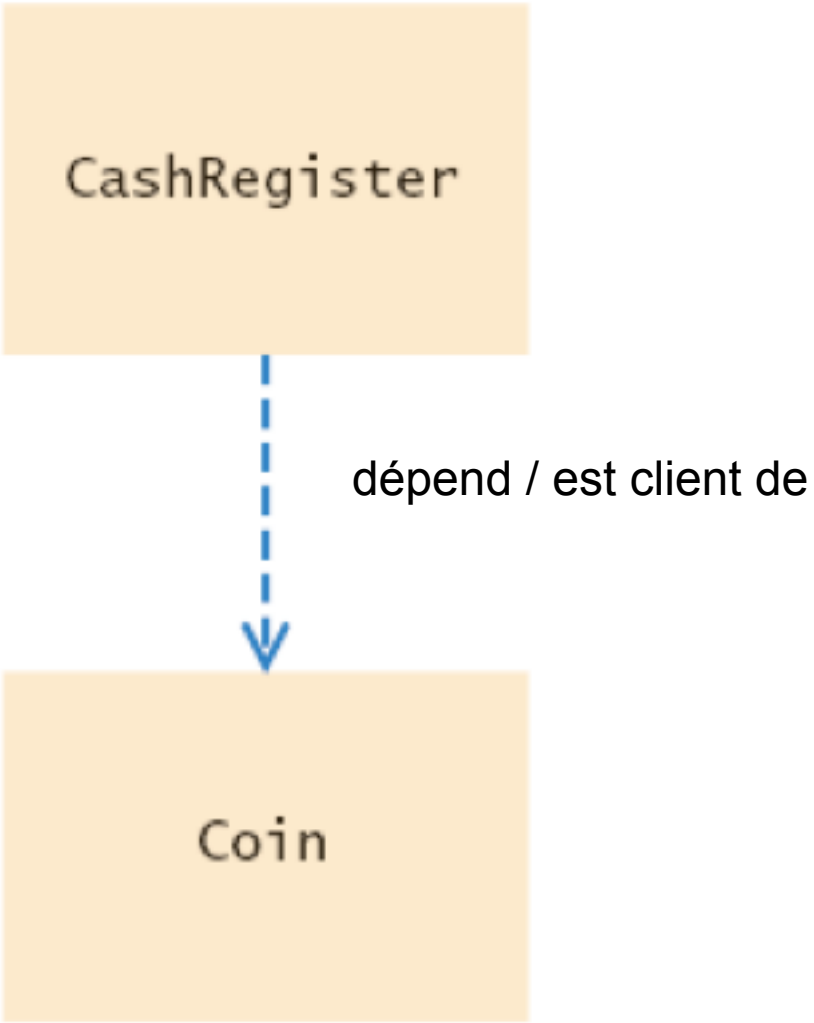
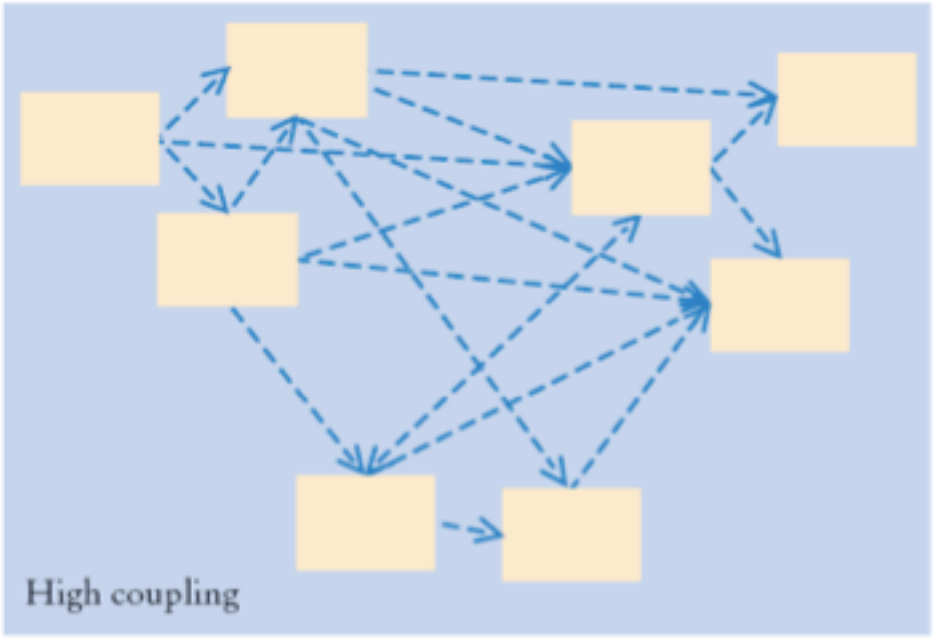
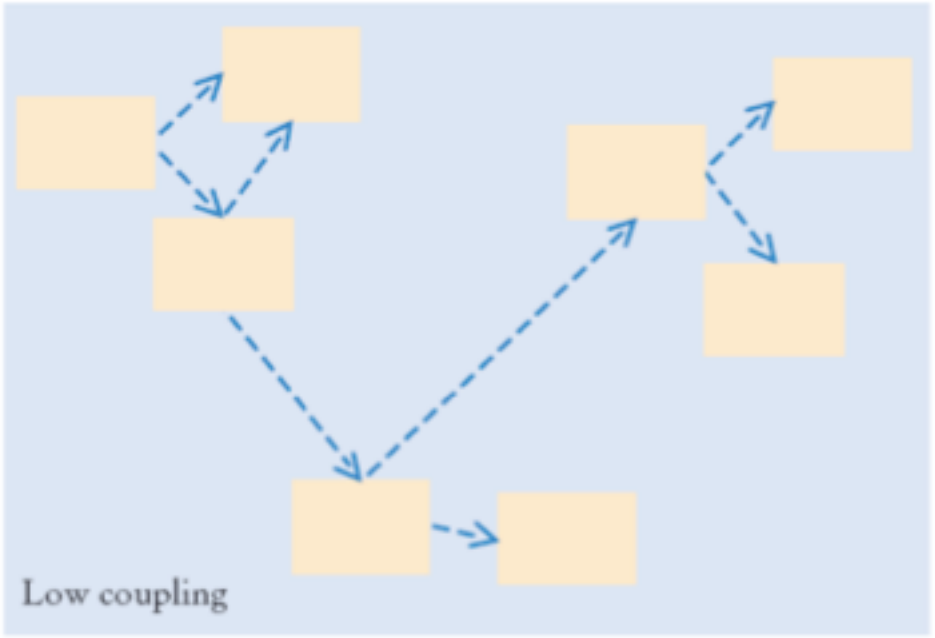
```
momsSavings.withdraw(500)
```

signifie

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

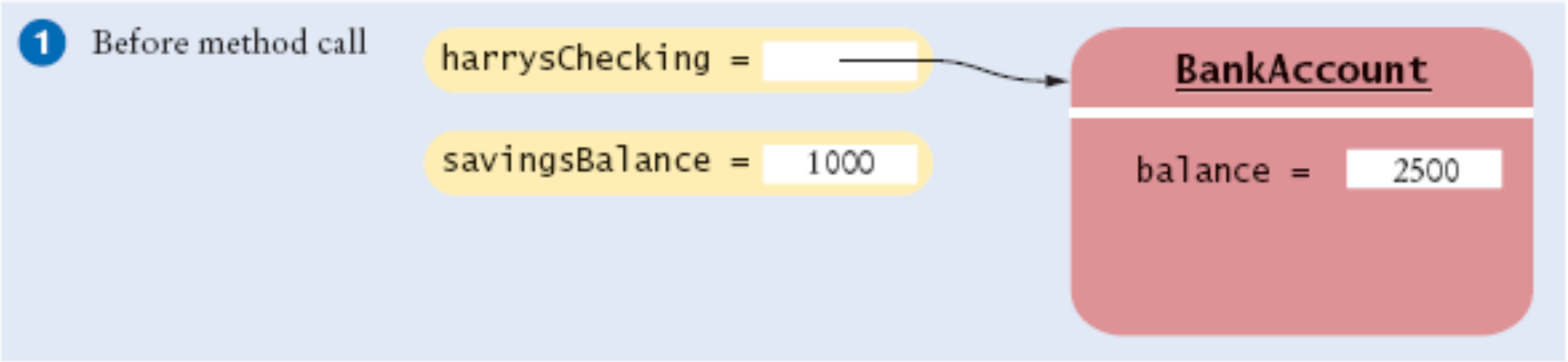
Paramètre implicite et `this`

- 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`



Objectifs de cette partie

- Apprendre à choisir les classes appropriées à implémenter
- Comprendre les notions de *cohésion* et de *couplage*
- Minimiser les effets de bord (*side effects*)
- Documenter les responsabilités de chaque méthodes et leurs appelants avec des pré-conditions et des post-conditions
- Comprendre la différence entre méthodes d'instance et de classe
- Introduire la notion de variable de classe
- Comprendre les règles de portée des variables locales et des variables d'instance
- Découvrir la notion de package

Découvrir et choisir des classes

- Une classe représente un unique concept/notion du monde du problème (chercher les noms dans l'énoncé du problème)
- Le nom d'une classe est généralement un nom qui décrit un concept
- Concepts mathématiques :

```
Point
Rectangle
Ellipse
```

- Concepts de la vie de tous les jours :

```
BankAccount
CashRegister
```

Découvrir et choisir des classes /2

- Acteurs – Objets qui ‘ ‘travaille pour vous’ ’
  `Scanner`
  `Random // meilleur nom: RandomNumberGenerator`
- Classes utilitaires – pas d’objet (instance) seulement des méthodes de classes
  `Math`
- Programme principal : contient uniquement une méthode `main`
- Ne transformer pas les actions en classe :
  `Paycheck` est un meilleur nom que `ComputePaycheck`

Questions

On vous demande d'écrire un programme de jeu d'échec. `ChessBoard` peut-elle être une classe ? Et `MovePiece`?

Réponse : Oui (`ChessBoard`) et Non (`MovePiece`).

Cohésion

- Une classe doit représenter un seul concept
- L'interface publique d'une classe est cohésive si toutes ses fonctionnalités sont en relation avec le concept représenté
- Cette classe manque de cohésion :

```
public class CashRegister
{
   public void enterPayment(int dollars, int quarters,
      int dimes, int nickels, int pennies)
   . . .
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   . . .
}
```

Cohésion

`CashRegister`, comme décrit précédemment, inclu deux concepts: *cash register* et *coin*

Solution : Faire deux classes :

```
public class Coin
{
   public Coin(double aValue, String aName) { . . . }
   public double getValue() { . . . }
   . . .
}
public class CashRegister
{
   public void enterPayment(int coinCount, Coin coinType)
      { . . . }
   . . .
}
```

Couplage

- Une classe *dépend* d'une autre classe si elle utilise des instances de cette seconde classe
- `CashRegister` dépend de `Coin` pour déterminer la valeur du paiement
- `Coin` ne dépend pas de `CashRegister`
- Couplage fort = beaucoup de dépendances entre classes
- Minimiser le couplage pour minimiser l'impact du changement d'une interface
- Pour visualiser les relations entre classes, dessinez des diagrammes
- UML: Unified Modeling Language. Une notation pour l'analyse et la conception orientée objet

Couplage



Couplage fort et faible entre des classes





Questions

Pourquoi la classe `Coin` ne dépend pas de la classe `CashRegister` ?

Réponse :
Aucune utilisation des méthodes de coin n'est requise dans la classe `CashRegister`

Questions /2

Pourquoi doit-on minimiser le couplage entre classes ?

Réponse :
Si une classe ne dépend pas d'une autre, alors elle ne peut être affectée par un changement de l'interface de cette dernière.

Accesseurs, Modificateurs et Classes non-mutables

- Accesseur : ne change pas l'état du paramètre implicite

```
double balance = account.getBalance();
```

- Modificateur : modifie l'objet sur lequel il est invoqué

```
account.deposit(1000);
```

- Classe non-mutable : ne possède pas de modificateurs (exemple `String`)

```
String name = "John Q. Public";
String uppercased = name.toUpperCase();
                          // name is not changed
```

Questions

La méthode `substring` est elle un accesseur ou modificateur de la classe `String` ?

Réponse : c'est un accesseur – appeler `substring` ne modifie pas l'objet. En fait toutes les méthode de la classe `String` sont des accesseurs

Questions /2

La classe `Rectangle` est-elle non mutable ?

Réponse : Non – la méthode `translate` est un modificateur.

Effets de bord (Side Effects)

- Effet de bord d'une méthode : tout modification observable de l'extérieur de la méthode

```
public void transfer(double amount, BankAccount other)
{
   balance = balance - amount;
   other.balance = other.balance + amount; // Modifies
      explicit parameter
}
```

- Mettre à jour les paramètres (explicite) d'une méthode peut engendrer des surprises au programmeur ; il est préférable d'éviter ce genre de manipulation si possible

Effets de bord (Side Effects) /2

- Un autre exemple d'effet de bord : l'affichage

```
public void printBalance() // Not recommended
{
   System.out.println("The balance is now $" + balance);
}
```

  Mauvaise idée : le message est en anglais, et il utilise `System.out`. Il est préférable de découpler entrée/sortie du fonctionnement de votre classe

- Vous devez minimiser les effets de bord qui modifie autre chose que le receveur (paramètre implicite)

Questions

Si a référence un compte bancaire, l'appel `a.deposit(100)` modifie ce compte bancaire. Est-ce un effet de bord ?

Réponse : Non – un effet de bord d'une méthode modifie autre chose que le receveur de l'appel de méthode.

Questions /2

Considérons cette méthode

```
void read(Scanner in)
{
   while (in.hasNextDouble())
      add(in.nextDouble());
}
```

A-t-elle des effets de bord ?

Réponse : Oui – La méthode affecte l'état de l'objet Scanner passé en paramètre

Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre

- ```
  void transfer(double amount, double otherBalance)
  {
     balance = balance - amount;
     otherBalance = otherBalance + amount;
  }
  ```
- Ne fonctionne pas
- Scenario :

  ```
  double savingsBalance = 1000;
  harrysChecking.transfer(500, savingsBalance);
  System.out.println(savingsBalance);
  ```
- En Java, une méthode ne peut jamais modifier la valeur d'une variable de type primitive passée en paramètre
  ```

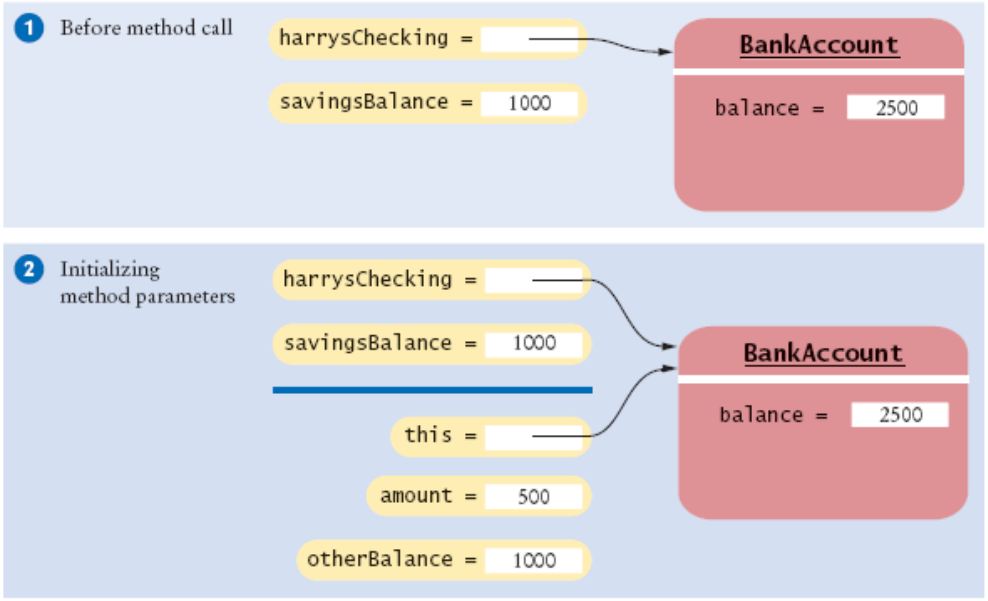
Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /2

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance)
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
}
```



Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /3

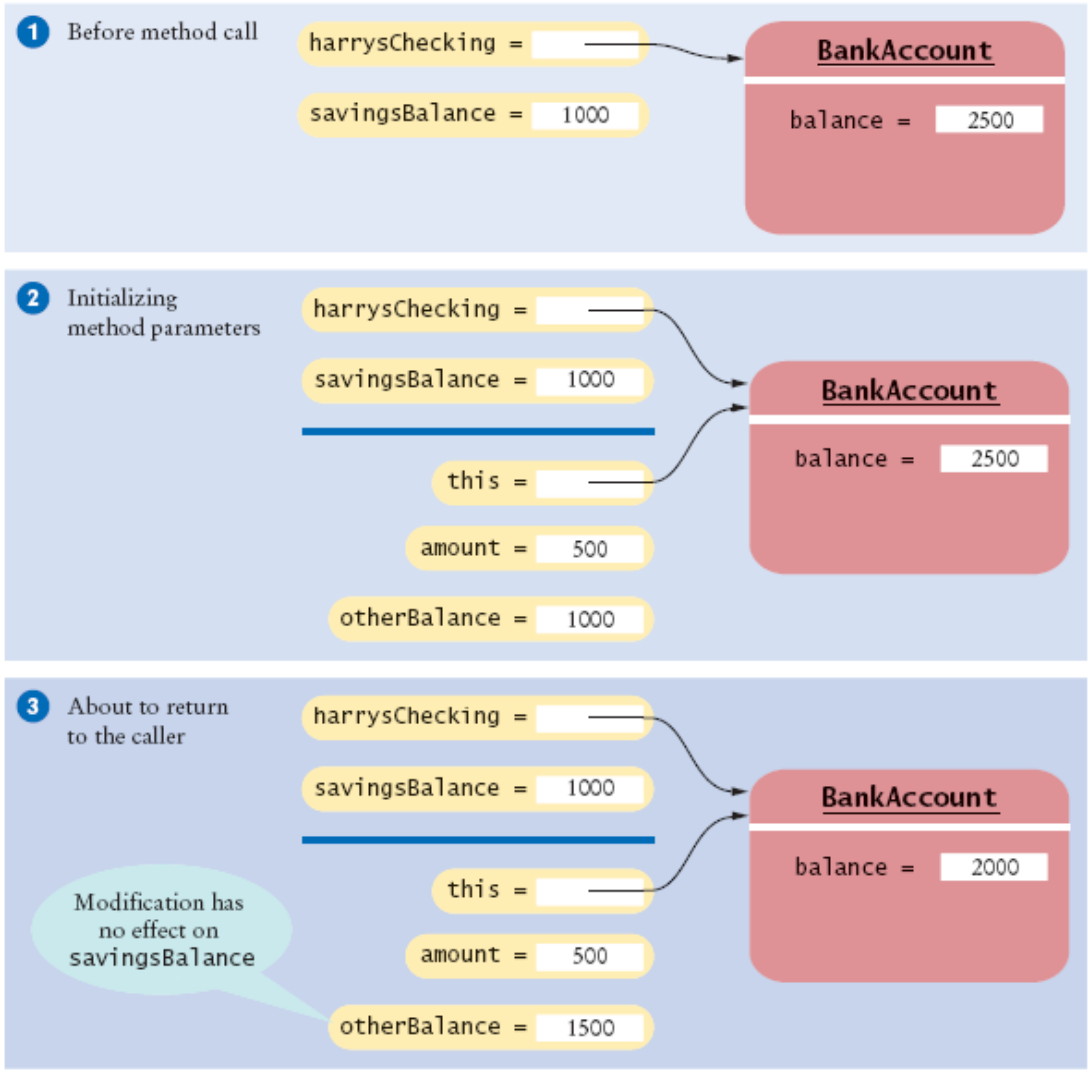
```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);   ❶
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance)   ❷
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
}
```



Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /4

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);  (1)
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance)  (2)
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
}  (3)
```

Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /5



Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /6

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);  ❶
System.out.println(savingsBalance);  ❹
...
void transfer(double amount, double otherBalance)  ❷
{
   balance = balance - amount;
   otherBalance = otherBalance + amount;
}  ❸
```

Erreur commune : Tentative de modification d'une valeur primitive passée en paramètre /7

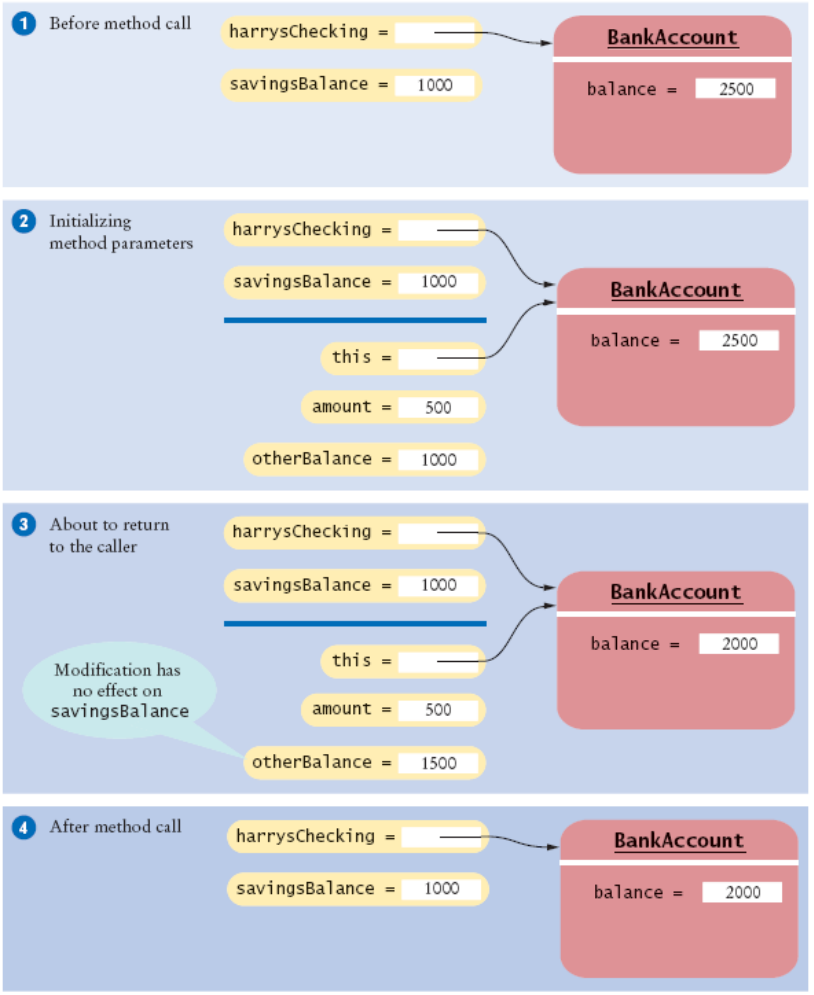


Figure 3 Modifying a Numeric Parameter Has No Effect on Caller

Passage par valeur et passage par référence

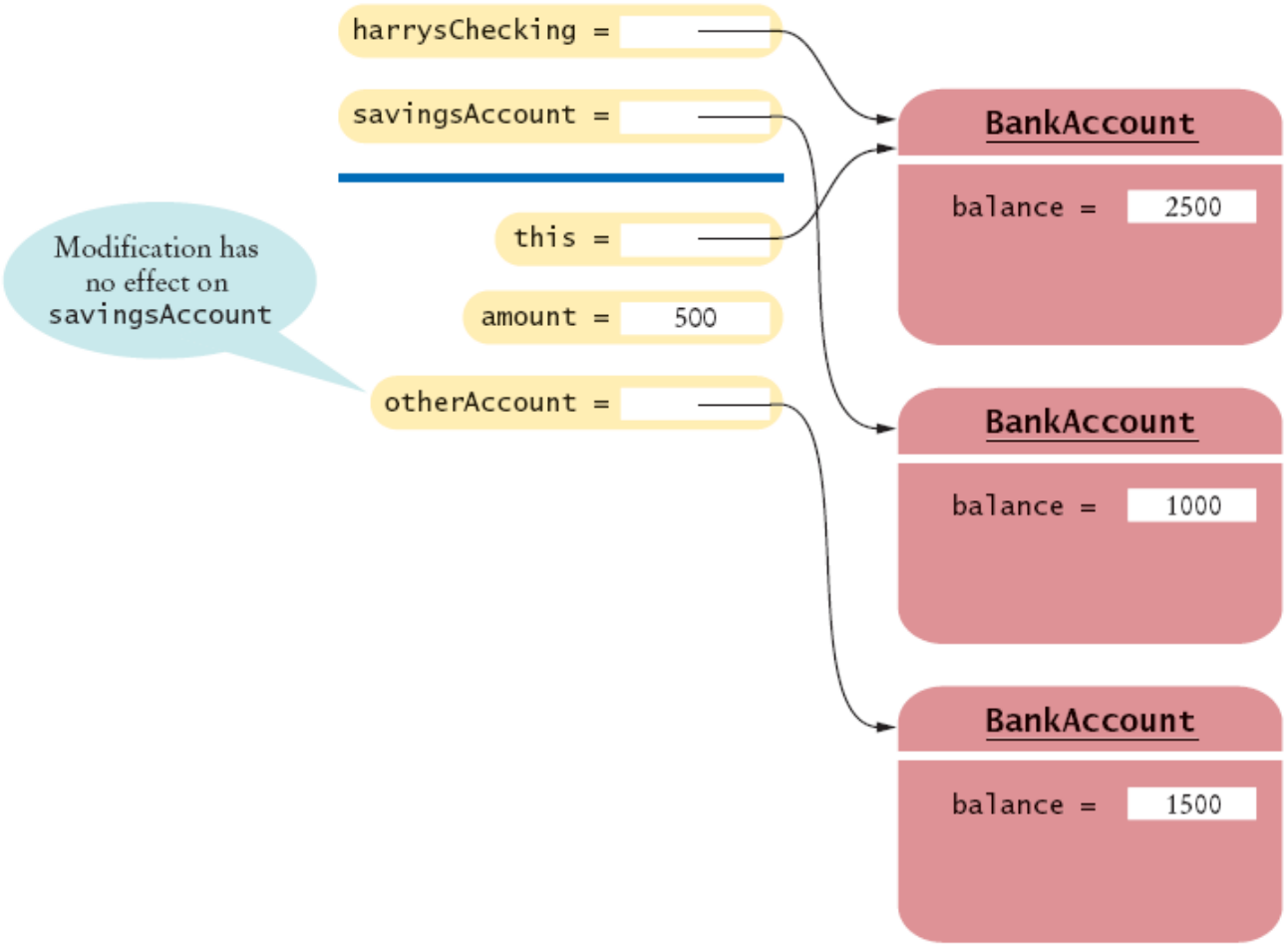
- Passage par valeur : Les paramètres sont copiés dans les variables représentant les paramètres au début de l'exécution de l'appel de méthode
- Passage par référénce : Les méthodes peuvent modifier les paramètres
- En Java uniquement des passage par valeur
- Une méthode peut change l'état d'un objet passé en paramètre (référence passée en paramètre), mais ne peut remplacer la référence par une autre référence

Passage par valeur et passage par référence /2

```
public class BankAccount
  {
     public void transfer(double amount, BankAccount
        otherAccount)
     {
        balance = balance - amount;
        double newBalance = otherAccount.balance + amount;
        otherAccount = new BankAccount(newBalance); //
           Won't work
     }
  }
```

Exemple : Passage par valeur

```
harrysChecking.transfer(500, savingsAccount);
```



Modifying an Object Reference Parameter Has No Effect on the Caller

Préconditions

- Précondition: Contrat que l'appelant d'une méthode doit respecter
- Documenter les préconditions pour que l'appelant n'appel pas avec de mauvaises valeurs en paramètre
- ```
  /**
     Deposits money into this account.
     @param amount the amount of money to deposit
     (Precondition: amount >= 0)
  */
  ```
- Usage courant:
  - *Pour restreindre les valeurs des paramètres d'une méthode*
  - *Pour s'assurer qu'une méthode n'est appelée que lorsque l'objet est dans un état particuliere*
- Si la précondition est violée, la méthode ne garantit pas un résultat correct.
  ```

Préconditions /2

- Méthode peuvent lever des exceptions si la précontions est violée (cf. cours plus tard)

```
if (amount < 0) throw new IllegalArgumentException();
balance = balance + amount;
```

- Méthode n'a pas à tester si une condition est satisfaite (Test peut être coûteux)

```
// if this makes the balance negative, it's the caller's
   fault
balance = balance + amount;
```

Préconditions /3

- Méthode peut effectuer une vérification d'assertion

```
assert amount >= 0;
balance = balance + amount;
```

- Pour activer la vérification des assertions :

```
java -enableassertions MyProg
```

On peut desactiver l'exécution des assertions quand notre programme est complétement testé, cela permet d'améliorer les performances d'exécution

- Tendance des programmeurs débutant (retour silencieux)

```
if (amount < 0)
    return; // Non recommandé car difficile à débugguer
balance = balance + amount;
```

Syntaxe Assertion

```
assert condition;
```

Exemple :

```
assert amount >= 0;
```

Objectif :

S'assurer qu'une condition est bien remplie. Si la vérification des assertions est activée et que la condition est violée alors une erreur est levée lors de l'exécution.

Postconditions

- Condition qui est satisfaite après l'exécution d'une méthode
- Si une méthode est appelée en respectant ses préconditions, elle doit garantir ses postconditions
- Il a deux types de postconditions :
  - *La valeur de retour est correctement calculée*
  - *L'objet est dans un certain état après l'exécution de la méthode*

```
/**
    Deposits money into this account.
    (Postcondition: getBalance() >= 0)
    @param amount the amount of money to deposit
    (Precondition: amount >= 0) */
```

- Ne documenter pas les postconditions triviales qui répète la clause `@return`

Postconditions /2

```
amount <= getBalance() // bonne formulation
amount <= balance // mauvaise formulation
```

- Contrat : Si l'appelant rempli la précondition, la méthode doit assurer la précondition

Questions

Pourquoi souhaiteriez vous ajouter une précondition à une méthode que vous fournissez à un autre programmeur ?

Réponse : Ensuite, vous n'aurez plus à tester les mauvaises valeurs – cela devient la responsabilité de l'appel (l'autre développeur).

Méthodes de classe

- Toute méthode appartient à une classe
- Une méthode de classe n'est pas invoquée sur un objet
- Pourquoi écrire une méthode qui n'opère pas sur objet ?
  Raison habituelle : des calculs sur des nombres qui ne sont pas des objets (ne peuvent donc pas recevoir d'appel de méthode).,
  Exemple : `x.sqrt()` avec `x` de type `double`
- ```
  public class Financial
  {
     public static double percentOf(double p, double a)
     {
        return (p / 100) * a;
     }
     // More financial methods can be added here.
  }
  ```
  ```

Méthodes de classe /2

- Appeler avec le nom de la classe comme receveur au lieu d'une référence d'objet :

```
double tax = Financial.percentOf(taxRate, total);
```

- `main` est une méthode de classe – il n'existe pas encore d'autre objets

Variable de classe

- Une variable de classe appartient à une classe et non pas à un objet particulier de la classe.

- ```
  public class BankAccount
  {
     . . .
     private double balance;
     private int accountNumber;
     private static int lastAssignedNumber = 1000;
  }
  ```

- Si `lastAssignedNumber` n'était pas `static`, chaque instance de `BankAccount` aurait sa propre valeur de `lastAssignedNumber`
  ```

Variable de classe /2

- ```
  public BankAccount()
    {
       // Generates next account number to be assigned
       lastAssignedNumber++; // Updates the static field
       // Assigns field to account number of this bank
          account
       accountNumber = lastAssignedNumber; // Sets the
          instance field }
  ```

- Minimiser l'usage des variables de classe (final static ne sont pas concernés)
  ```

Variable de classe /3

- 3 manières d'initialiser :
  1. *Ne rien faire. La variable est initialisée avec 0 (pour les nombres), false (pour les booléens) ou null (pour les références d'objet)*
  2. *Initialiser explicitement*

```
public class BankAccount
{
    . . .
    private static int lastAssignedNumber = 1000;
        // Executed once,
        // when class is loaded }
```

  3. *Utiliser un bloc de code static*

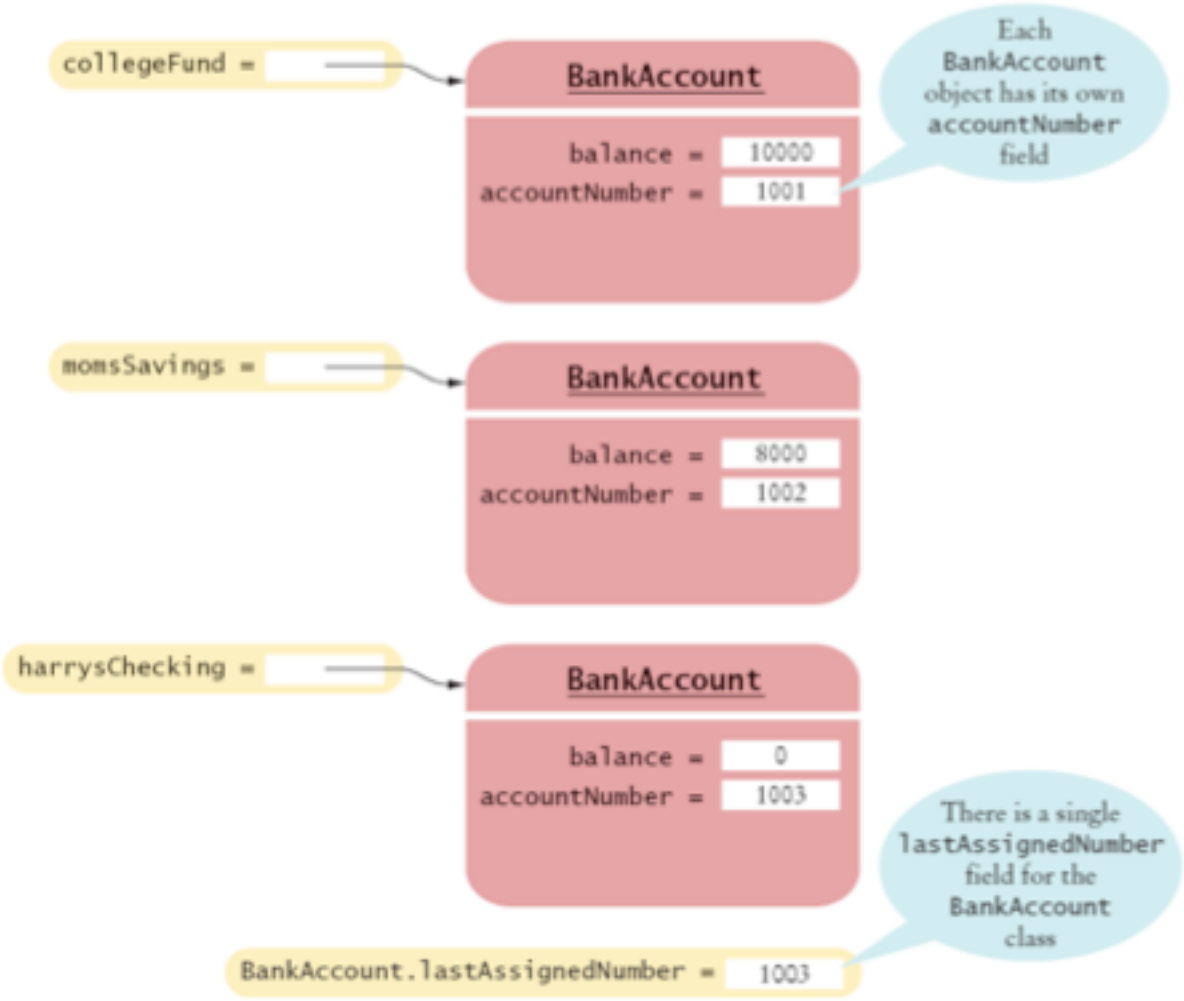
- Les variables de classe devraient toujours être déclarées comme `private`

Variable de classe /4

- Exception : les constantes qui peuvent être publique

```
public class BankAccount
{
    . . .
    public static final double OVERDRAFT_FEE = 5; //
       Refer to it as
       // BankAccount.OVERDRAFT_FEE
}
```

Une variable de classe et une variable d'instance



Portée des variables locales

- Portée des variables : Portion d'un programme où une variable peut être accèdée
- Portée d'une variable locale : de sa position de déclaration à la fin du bloc englobant

Portée des variables locales /2

- Le même nom de variable est utilisée dans plusieurs méthode :

```
public class RectangleTester
{
   public static double area(Rectangle rect)
   {
      double r = rect.getWidth() * rect.getHeight();
      return r;
   }
   public static void main(String[] args)
   {
      Rectangle r = new Rectangle(5, 10, 20, 30);
      double a = area(r);
      System.out.println(r);
   }
}
```

- Ces variables sont indépendantes. Leurs portées sont disjointes

Portée des variables locales /3

- La portée d'une variable locale ne peut contenir la définition d'une variable avec le même nom

```
Rectangle r = new Rectangle(5, 10, 20, 30);
if (x >= 0)
{
   double r = Math.sqrt(x);
   // Error - can't declare another variable called r
      here
   . . .
}
```

Portée des variables locales /4

- Pourtant, deux variables locales peuvent avoir le même nom si leurs portées sont disjointes

```
if (x >= 0)
{
   double r = Math.sqrt(x);
   . . .
   } // Scope of r ends here
else
{
   Rectangle r = new Rectangle(5, 10, 20, 30);
   // OK - it is legal to declare another r here
   . . .
}
```

Portée des variables de classe

- Les variables privées ont la portée de la classe : Vous pouvez accèder toutes les variables dans toutes les méthodes de la classe
- On doit préciser le nom de la classe pour accèder aux variables publiques hors de la classe de définition

```
Math.sqrt
harrysChecking.getBalance
```

- Dans une méthode, il n'est pas nécessaire (mais fortement conseillé) de préciser le nom de la classe lors de l'appel d'une méthode ou d'une variable de la même classe

Portée des variables de classe /2

- Un appel non qualifié (méthode ou instance) référence l'objet courant `this`

```
public class BankAccount
{
   public void transfer(double amount, BankAccount other)
   {
      withdraw(amount); // i.e., this.withdraw(amount);
      other.deposit(amount);
   }
   . . .
}
```

Recouvrement des portées

- Une variable locale peut masquée une variable d'instance qui porte le même nom
- La portée locale gagne sur la portée de classe

```
public class Coin
{
   . . .
   public double getExchangeValue(double exchangeRate)
   {
      double value; // Local variable
      . . .
      return value;
   }
   private String name;
   private double value; // Field with the same name
}
```

Recouvrement des portées /2

- On peut toujours accèder aux variables masquées en les qualifiant avec la référence this

```
value = this.value * exchangeRate;
```

Organisation des classes en paquetage

- Paquetage (*Package*): Ensemble de classe en relation
- Pour placer des classes dans un paquetage, il faut ajouter la ligne :

  `package packageName;`

  comme première instruction dans le code source de la classe
- Noms de paquetage sont formées par une succession d'identifiants séparés par des points

Organisation des classes en paquetage /2

- Par exemple, pour placée la classe `Financial` dans un paquetage nommé `com.horstmann.bigjava`, le fichier `Financial.java` doit débuté par :

```
package com.horstmann.bigjava;

public class Financial
{
   . . .
}
```

- Paquetage par défaut, aucune instruction `package`

Paquetages couramment utilisés dans la librairie Java

Paquetage	Objectif	Classe exemple
java.lang	Fonctionnalité du langage	Math
java.util	Utilitaires	Random
java.io	Entrée / sortie	PrintStream
java.awt	Abstract Windowing Toolkit (interface graphique)	Color
java.applet	Applets	Applet
java.net	Réseau	Socket
java.sql	Accès base de données	ResultSet
javax.swing	Swing (interface graphiqe)	JButton

Syntaxe Paquetage

```
package packageName;
```

Exemple :

```
package com.horstmann.bigjava;
```

Objectif :

Déclarer que toutes les classes de ce fichier appartiennent à un paquetage spécifique.

Importer des paquetages

- On peut toujours utiliser une classe sans importer son paquetage

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

- Fatiguant d'utiliser le nom qualifié
- Importation permet d'utiliser les noms courts des classes

```
import java.util.Scanner; . . .
Scanner in = new Scanner(System.in)
```

- On peut importer toutes les classes d'un paquetage

```
import java.util.*;
```

- Jamais nécessaire d'importer `java.lang`
- Jamais nécessaire d'importer les autres classes d'un même paquetage

Nom de paquetage et localisation des classes

- Utiliser des noms de paquetages pour éviter les clash sur les noms

  java.util.Timer vs. javax.swing.Timer

- Noms de paquetages ne doivent pas être ambigue
- Recommendation : debute par un nom de domaine inversé

  com.horstmann.bigjava
  fr.esial.uhp

- Les chemins doivent concorder avec les noms de paquetages

  `com/horstmann/bigjava/Financial.java`

Nom de paquetage et localisation des classes /2

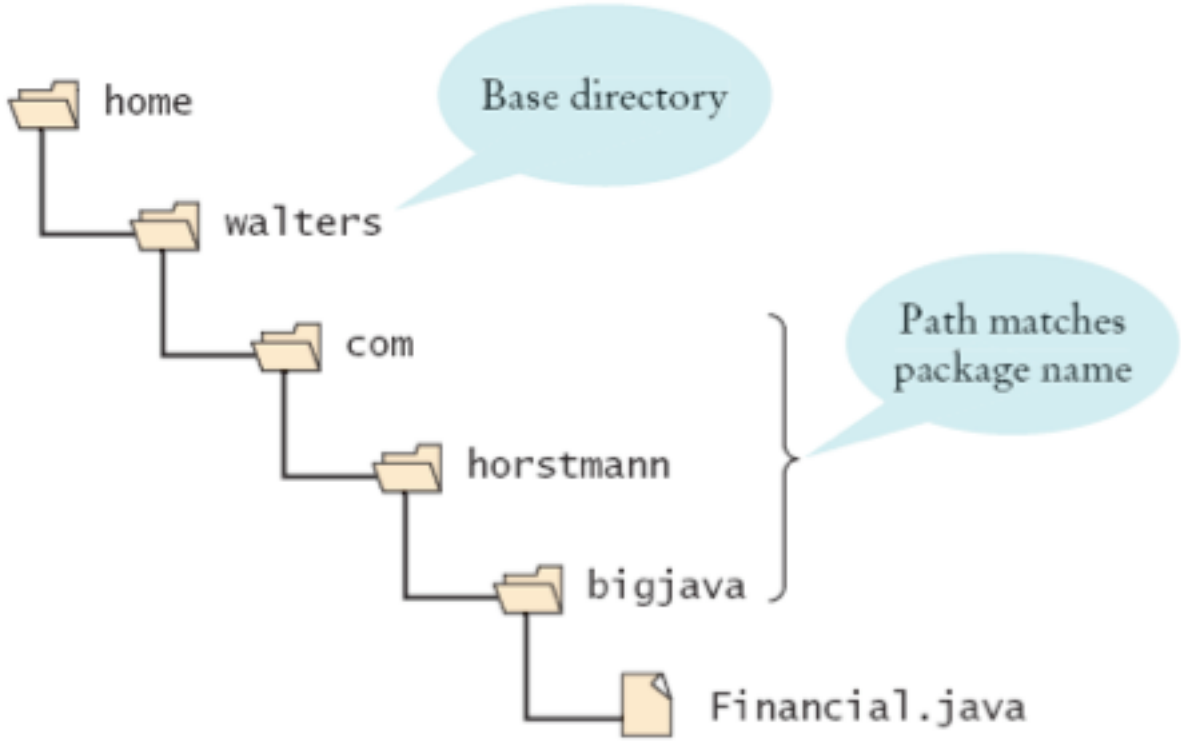
- Les chemins sont recherchés à partir du classpath

  `export CLASSPATH=/home/walters/lib:.`

  `set CLASSPATH=c:\home\walters\lib;.`

- Class path contient les répertoires de base où les répertoires des paquetages sont placés

Répertoires de base et sous répertoires des paquetages



Questions

Parmi les éléments suivants, lesquels sont des paquetages ?

a. `java`
b. `java.lang`
c. `java.util`
d. `java.lang.Math`

Réponse :

a. No
b. Yes
c. Yes
d. No