



# Un approccio metodologico

---

- **FASE1: Progettazione dell' Interfaccia Pubblica**

- Decidere il comportamento che la classe dovrà fornire
  - Identificare i metodi da fornire
- Stabilire in che modo la classe verrà usata
  - Definire l'interfaccia della classe, i prototipi dei metodi
- Scrivere un programma di esempio che utilizza la classe
- Scrivere lo scheletro della classe
  - Prototipi e corpi vuoti

- **FASE2: Implementazione di una classe**



# Progettazione dell'interfaccia pubblica di un conto corrente

---

- Comportamento di un conto corrente bancario (astrazione):
  - depositare contante
  - prelevare contante
  - leggere il saldo
  - creare un nuovo conto



# Conto corrente (BankAccount): metodi

---

- Metodi della classe **BankAccount**:

```
deposit  
withdraw  
getBalance
```

- Vogliamo poter eseguire le seguenti operazioni:

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



# Definizione di un metodo

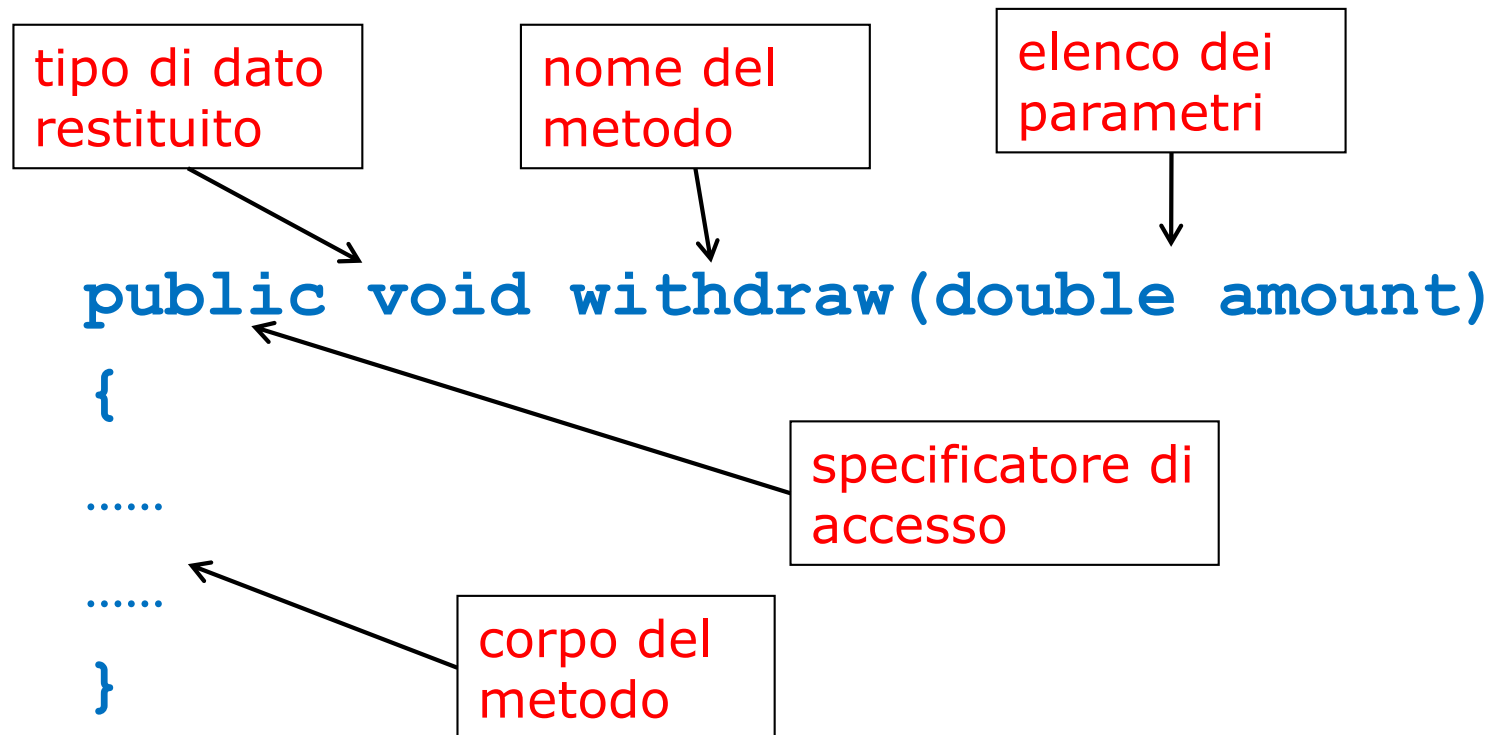
---

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

## ○ Sintassi

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

# Definizione di un metodo



- Lo *specificatore di accesso* indica la visibilità (*scope*) del metodo
  - public** indica che il metodo può essere invocato anche dai metodi esterni alla classe `BankAccount` (e anche da quelli esterni al package a cui appartiene la classe `BankAccount`)



# Costruttore

---

- Un costruttore inizializza le variabili di istanza
- Il nome del costruttore è il nome della classe

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



# Costruttore

---

- Il corpo del costruttore è eseguito quando viene creato un nuovo oggetto
- Le istruzioni del costruttore assegnano valori alle variabili di istanza
- Ci possono essere diversi costruttori ma tutti devono avere lo stesso nome (**overloading**)
  - il compilatore li distingue dalla lista dei parametri espliciti



# Nota su overloading (sovraccarico)

---

- Più metodi con lo stesso nome
  - Consentito se i parametri li distinguono, cioè hanno firme diverse  
(**firma = nome del metodo + lista tipi dei parametri nell'ordine in cui compaiono**)
  - Il tipo restituito non conta
- Frequente con costruttori
  - Devono avere lo stesso nome della classe
  - Es.: aggiungiamo a Rectangle il costruttore

```
public Rectangle(int x_init, int y_init) {  
    x=x_init;  
    y=y_init;  
}
```
- Usato anche quando dobbiamo agire diversamente a seconda del tipo passato
  - Ad es., *println* della classe *PrintStream*





# Sintassi costruttore

---

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

## **Example:**

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

## **Purpose:**

To define the behavior of a constructor



# BankAccount: Interfaccia Pubblica

---

- I costruttori e i metodi public di una classe formano l'*interfaccia pubblica* della classe.

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



# BankAccount: Interfaccia Pubblica

---

```
// 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
}
```



# Self Check

---

- Come possiamo usare i metodi dell'interfaccia pubblica per azzerare il conto `harrysChecking`?
- Supponiamo di voler un conto bancario che tiene traccia di un numero di conto oltre al saldo. Come si deve cambiare l'interfaccia pubblica?



# Risposte

---

- `harrysChecking.withdraw(harrysChecking.getBalance())`
- Aggiungere un parametro `accountNumber` ai costruttori, ed aggiungere un metodo `getAccountNumber()`. Non c'è bisogno di un metodo `setAccountNumber` poichè il numero di conto non cambia dopo la sua costruzione.



# Sintassi definizione di classe

---

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

## Example:

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

# Definizione di una classe

## File BankAccount.java

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

nome del  
classe

specificatore di  
accesso

- *specificatore di accesso* **public** indica che la classe BankAccount è utilizzabile anche al di fuori del *package* di cui fa parte la classe
- una classe pubblica deve essere contenuta in un file avente il suo stesso nome
  - Es.: la classe BankAccount è memorizzata nel file BankAccount.java



# Commenti per documentazione javadoc

---

```
/**
 * 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
}
```





# Commenti alla classe

---

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

- Fornire commenti per
  - ogni classe
  - ogni metodo
  - ogni parametro esplicito
  - ogni valore restituito da una funzione

BankAccount

file:///Users/GC/Desktop/index.html

Google

Pair Programming
Offerte voli
Apple (112)
Amazon
eBay
Yahoo!
Notizie (536)

All Classes

[BankAccount](#)

[Package](#)
[Class](#)
[Use](#)
[Tree](#)
[Deprecated](#)
[Index](#)
[Help](#)

PREV CLASS NEXT CLASS

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#)

[FRAMES](#) [NO FRAMES](#)

DETAIL: FIELD | [CONSTR](#) | [METHOD](#)

Class BankAccount

java.lang.Object

└ BankAccount

```
public class BankAccount
extends java.lang.Object
```

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

Constructor Summary

[BankAccount](#)()

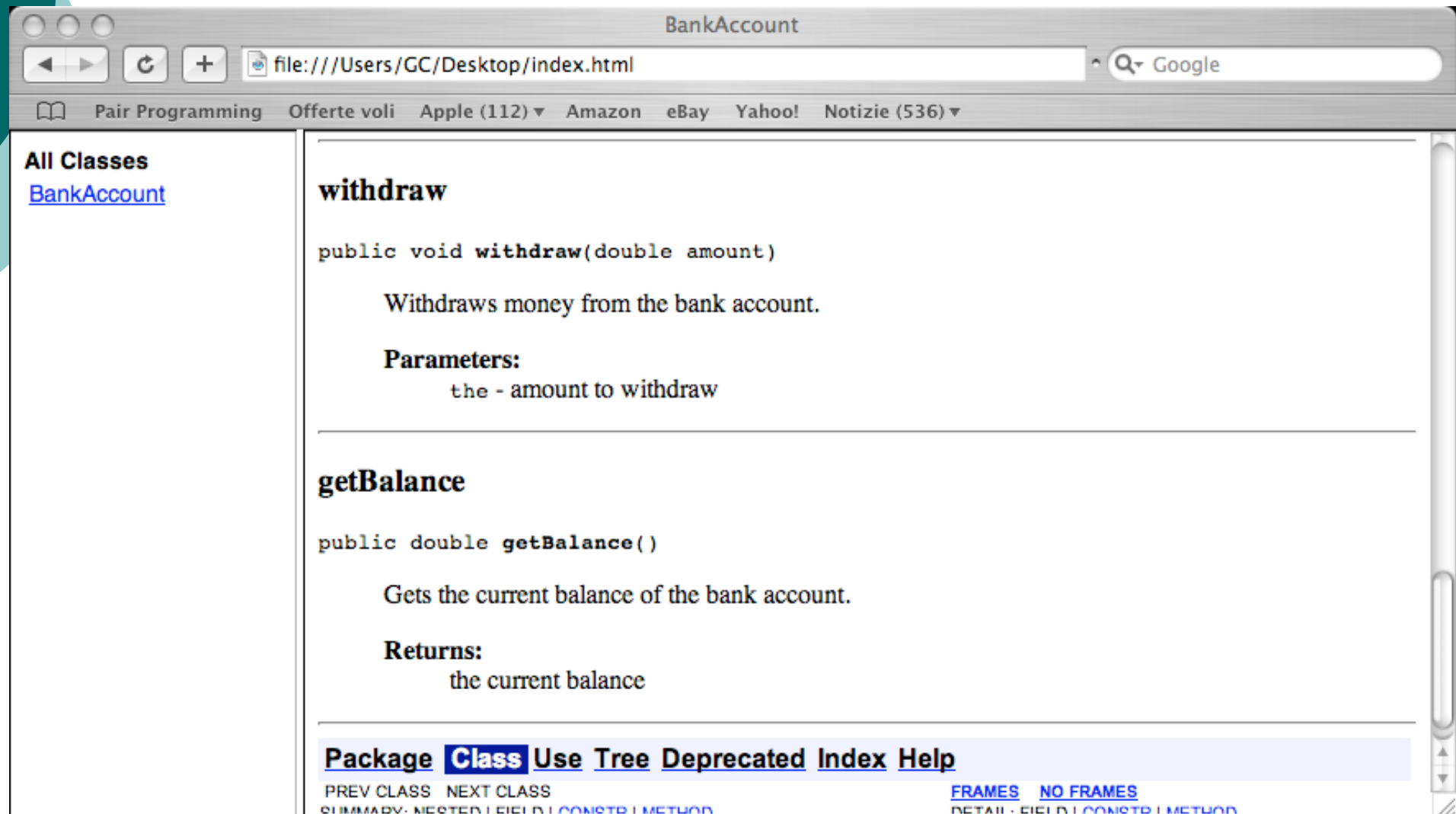
[BankAccount](#)(double initialBalance)

Method Summary

void	<a href="#">deposit</a> (double amount)
double	<a href="#">getBalance</a> () Gets the current balance of the bank account.
void	<a href="#">withdraw</a> (double amount) Withdraws money from the bank account.

Methods inherited from class java.lang.Object

# Documentazione



The screenshot shows a web browser window titled "BankAccount". The address bar displays "file:///Users/GC/Desktop/index.html". The browser's search bar contains "Google". The browser's toolbar includes links for "Pair Programming", "Offerte voli", "Apple (112)", "Amazon", "eBay", "Yahoo!", and "Notizie (536)".

The main content area is divided into two panes. The left pane, titled "All Classes", contains a link to "BankAccount". The right pane displays the documentation for the "BankAccount" class, showing the "withdraw" and "getBalance" methods.

**withdraw**

```
public void withdraw(double amount)
```

Withdraws money from the bank account.

**Parameters:**

the - amount to withdraw

---

**getBalance**

```
public double getBalance()
```

Gets the current balance of the bank account.

**Returns:**

the current balance

---

**Package Class Use Tree Deprecated Index Help**

PREV CLASS NEXT CLASS

SUMMARY: NESTED | FIELD | CONSTR | METHOD

FRAMES NO FRAMES

DETAIL: FIELD | CONSTR | METHOD



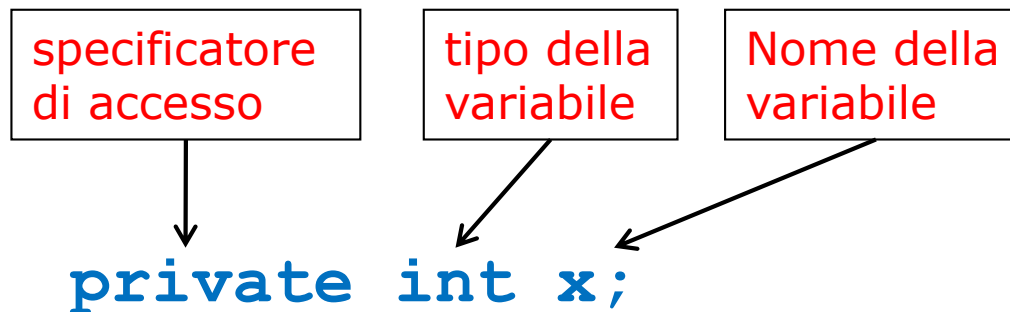
# Variabili di istanza

---

- Contengono il dato memorizzato nell'oggetto
- Istanza di una classe: un oggetto della classe
- La definizione della classe specifica le variabili d'istanza:

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

# Definizione di una variabile di istanza



- Lo *specificatore di accesso* indica la visibilità (scope) della variabile
  - **private** indica che la variabile di istanza può essere letta e modificata solo dai metodi della classe
    - dall'esterno è possibile accedere alle variabili di istanza **private** solo attraverso i metodi pubblici della classe
    - Solo raramente le variabili di istanza sono dichiarate **public**
- Il tipo delle variabili di istanza può essere
  - una classe, Es.: **String**
  - un array
  - un tipo primitivo, Es.: **int**



# Accesso variabili di istanza

---

- Il metodo `deposit` di `BankAccount` può accedere alla variabile di istanza `private`:

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



# Accesso variabili di istanza

---

- Metodi di altre classi non possono:

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

- Incapsulamento = si nasconde dato e si fornisce l'accesso attraverso i metodi



# Self Check

---

- Supponiamo che ogni **BankAccount** ha un numero di conto. Cosa bisogna cambiare nelle variabili di istanza?
- Quali sono le variabili di istanza della classe **Rectangle**?





# Risposte

---

- Una variabile di istanza

```
private int accountNumber;
```

deve essere aggiunta alla classe

- ```
private int x;  
private int y;  
private int width;  
private int height;
```



# Implementazione Costruttori

---

- Contengono istruzioni per inizializzare le variabili di istanza

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



# Implementazione Costruttori

---

```
BankAccount harrysChecking = new BankAccount(1000);
```

- Crea un nuovo oggetto di tipo **BankAccount**
- Chiama il secondo costruttore siccome viene passato un parametro
- Assegna il parametro **initialBalance** a 1000
- Assegna la copia del campo **balance** del nuovo oggetto creato con **initialBalance**
- Restituisce un riferimento ad un oggetto di tipo **BankAccount** (cioè la locazione di memoria dell'oggetto) come valore della **new**-expression
- Salva il riferimento nella variabile **harrysChecking**



# Implementazione Metodi

---

- Alcuni non restituiscono un valore

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

- altri si

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




# Invocazione metodo

---

```
harrysChecking.withdraw(500);
```

- Assegna il parametro **amount** a 500
- Recupera il contenuto del campo **balance** dell'oggetto la cui locazione è salvata in **harrysChecking**
- Sottrae il valore **amount** da **balance** e salva il risultato in **newBalance**
- Salva il valore di **newBalance** in **balance**, sovrascrivendo il vecchio valore



# The return Statement

---

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

## Esempio:

```
return balance;
```

## Scopo:

Specificare il valore che un metodo restituisce ed uscire immediatamente dal metodo.

Il valore di ritorno diventa il valore dell'espressione della chiamata a metodo.

# File 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:
```



# File BankAccount.java

```
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:     */
```





# File BankAccount.java

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



# Self Check

---

- Com'è implementato il metodo `getWidth` della classe `Rectangle`?
- Com'è implementato il metodo `translate` della classe `Rectangle`?



# Risposte

---

- ```
public int getWidth()
{
    return width;
}
```

- Ci sono diverse risposte corrette. Una possibile implementazione è:

```
public void translate(int dx, int dy)
{
    int newX = x + dx;
    x = newX;
    int newY = y + dy;
    y = newY;
}
```



# Testare una classe

---

- Classe Tester: una classe con il metodo **main** che contiene istruzioni per testare un'altra classe
- Solitamente consiste in:
  1. costruire uno o più oggetti della classe da testare
  2. invocare sugli oggetti uno o più metodi
  3. stampare a video i risultati delle computazioni



# File 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: }
```



# Categorie di variabili

---

- Variabili di istanza
  - Appartengono all'oggetto
  - Esistono finché l'oggetto esiste
  - Hanno un valore iniziale di default
- Variabili locali
  - Appartengono al metodo
  - Vengono create all'attivazione del metodo e cessano di esistere con esso
  - Non hanno valore iniziale se non inizializzate
- Parametri formali
  - Appartengono al metodo
  - Vengono create all'attivazione del metodo e cessano di esistere con esso
  - Valore iniziale è il valore del parametro reale al momento dell'invocazione



# Categorie di variabili

---

- L'ordine delle dichiarazioni è irrilevante
  - Convenzione: i metodi prima delle variabili di istanza
- Controllo di accesso
  - **public**: consente l'accesso al di fuori della classe
  - **private**: limita l'accesso ai membri della classe
  - Si applica sia ai metodi che alle variabili di istanza



# Regole Java

---

- Periodo di vita di una variabile
  - Parametri e variabili locali vivono solo durante l'esecuzione di un metodo
  - Le variabili di istanza hanno lo stesso periodo di vita dell'oggetto cui appartengono
- Periodo di vita di un oggetto
  - Un oggetto esiste fino a quando c'è una variabile di riferimento che si riferisce ad esso
  - La distruzione è automatica





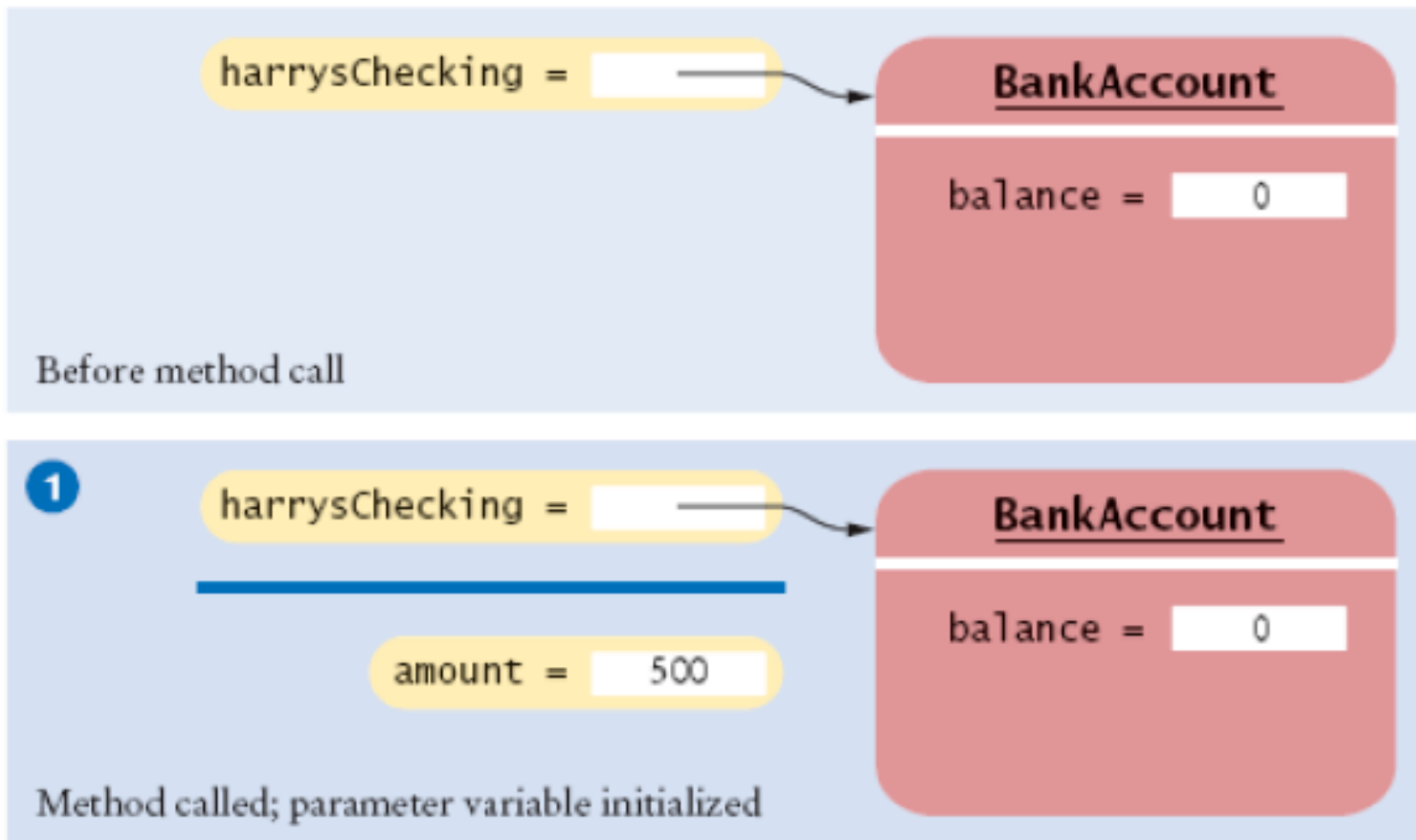
# Garbage collector

---

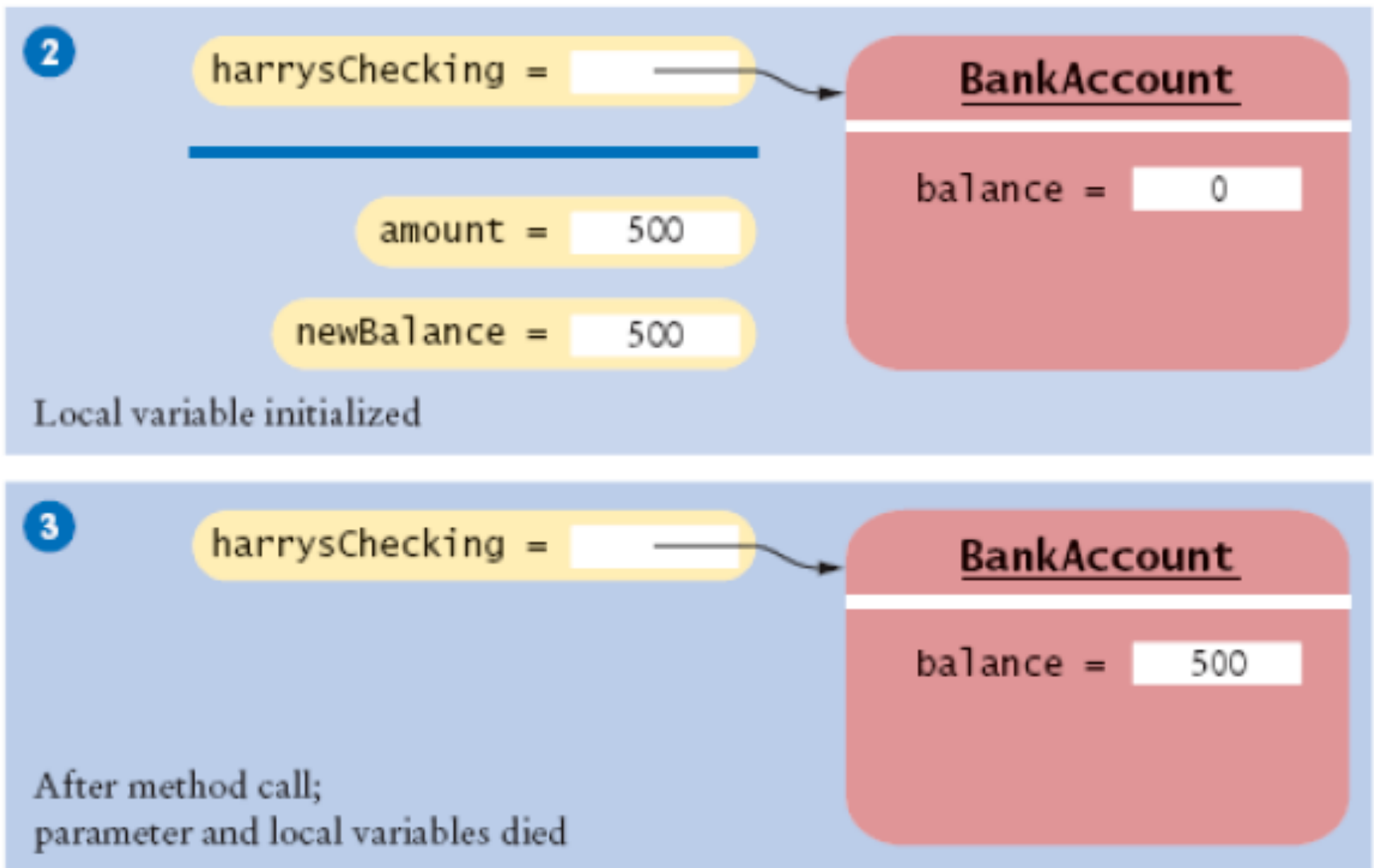
- In java, il garbage collector periodicamente recupera la memoria relativa ad oggetti non più referenziati
- Ciclo di vita

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

# Ciclo di vita delle variabili



# Ciclo di vita delle variabili

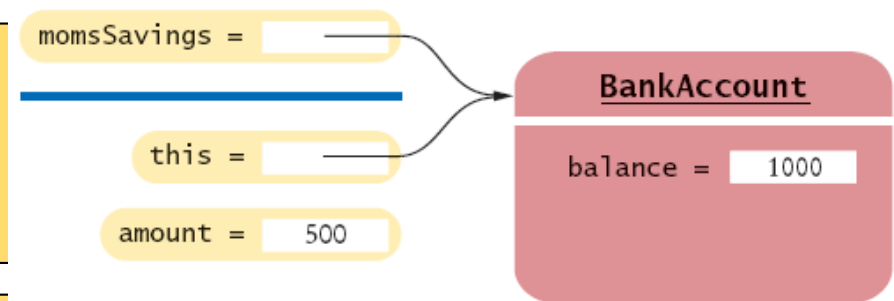


# Parametro espliciti ed impliciti

- Parametro implicito: l'oggetto di invocazione
- Il riferimento **this** denota il parametro implicito

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

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



**Figure 8** The Implicit Parameter of a Method Call



# Progettazione ad oggetti

---

- Caratterizzazione attraverso le **classi** delle entità (oggetti) coinvolte nel problema da risolvere (individuazione classi)
  - identificazione delle classi
  - identificazione delle responsabilità (operazioni) di ogni classe
  - individuazione delle relazioni tra le classi
    - dipendenza (usa oggetti di altre classi)
    - aggregazione (contiene oggetti di altre classi)
    - ereditarietà (relazione sottoclasse/superclasse )
- Realizzazione delle classi



# Realizzazione di una classe

---

1. individuazione dei metodi dell'interfaccia pubblica:
  - determinazione delle operazioni che si vogliono eseguire su ogni oggetto della classe
2. individuazione delle variabili di istanza:
  - determinazione dei dati da mantenere
3. individuazione dei costruttori
4. Codifica dei metodi
5. Collaudo del codice



# Programmi Java

---

- Un programma Java consiste di una o più classi
- Per poter eseguire un programma bisogna definire una classe pubblica che contiene un metodo

```
public static void main(String[] args)
```