Variabili e Parametri

Scope, Lifetime
Inizializzazione

Categorie di Variabili

- **Variabili di istanza / campi**
  - **`saldo`** in **`BankAccount`**
- **Variabili locali**
  - **`newsaldo`** nel metodo **`deposita()`**
- **Parametri**
  - **`importo`** per il metodo **`deposita()`**

Categorie di Variabili

- **Variabili di istanza / campi**
  - appartengono ad uno (ed un solo oggetto)
  - Lifetime = lifetime degli oggetti a cui appartengono
  - Inizializzati a valori di default

- **Variabili locali ed parametri**
  - appartengono ai metodi in cui sono dichiarati
  - Lifetime = esecuzione dei metodi che le dichiarano
  - Parametri inizializzati agli argomenti
  - Variabili locali inizializzate esplicitamente

Lifetime delle variabili

- Consideriamo l'invocazione

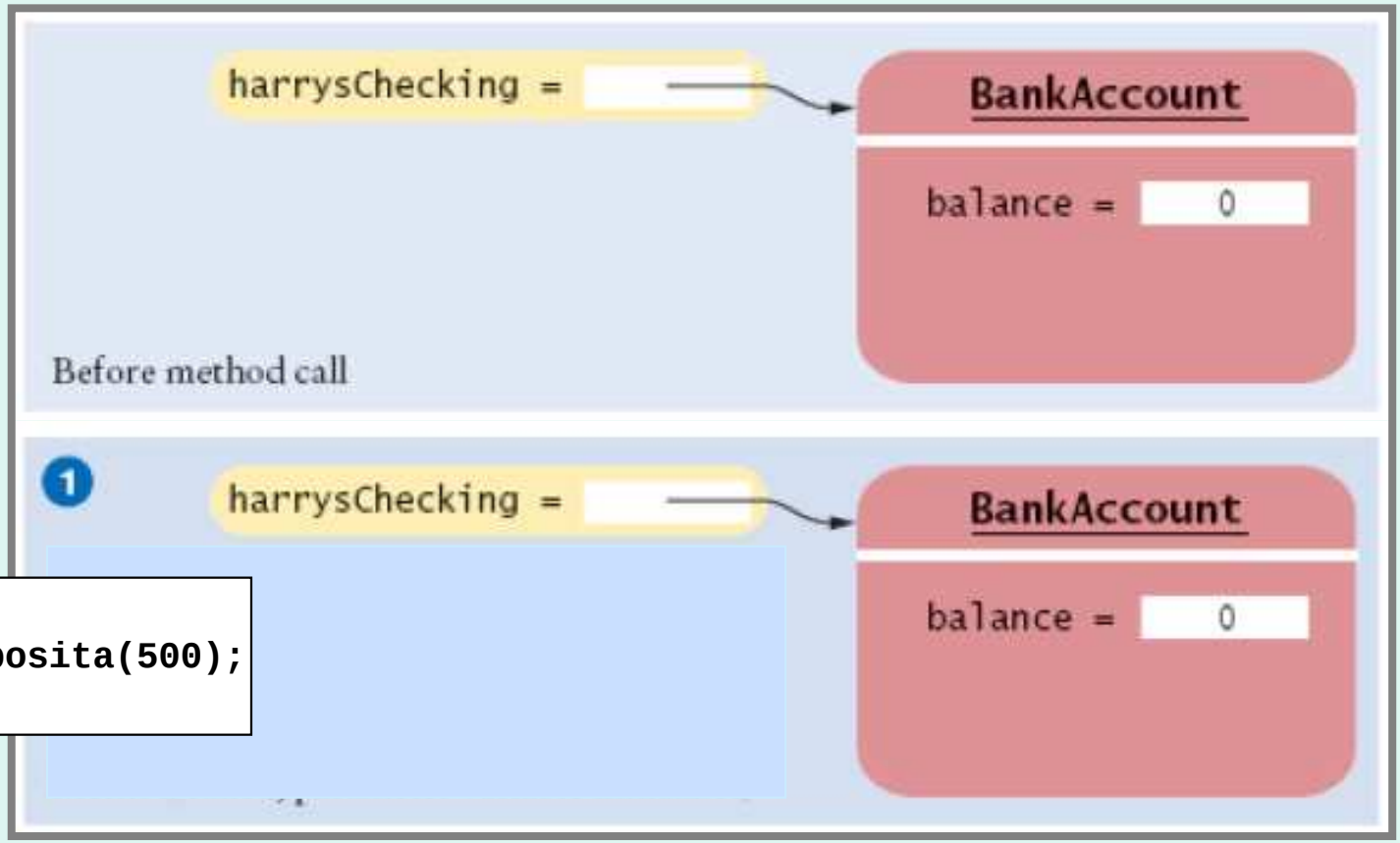
```
harrysChecking.deposita(500);
```

ricordando

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

Continua…

Lifetime delle variabili

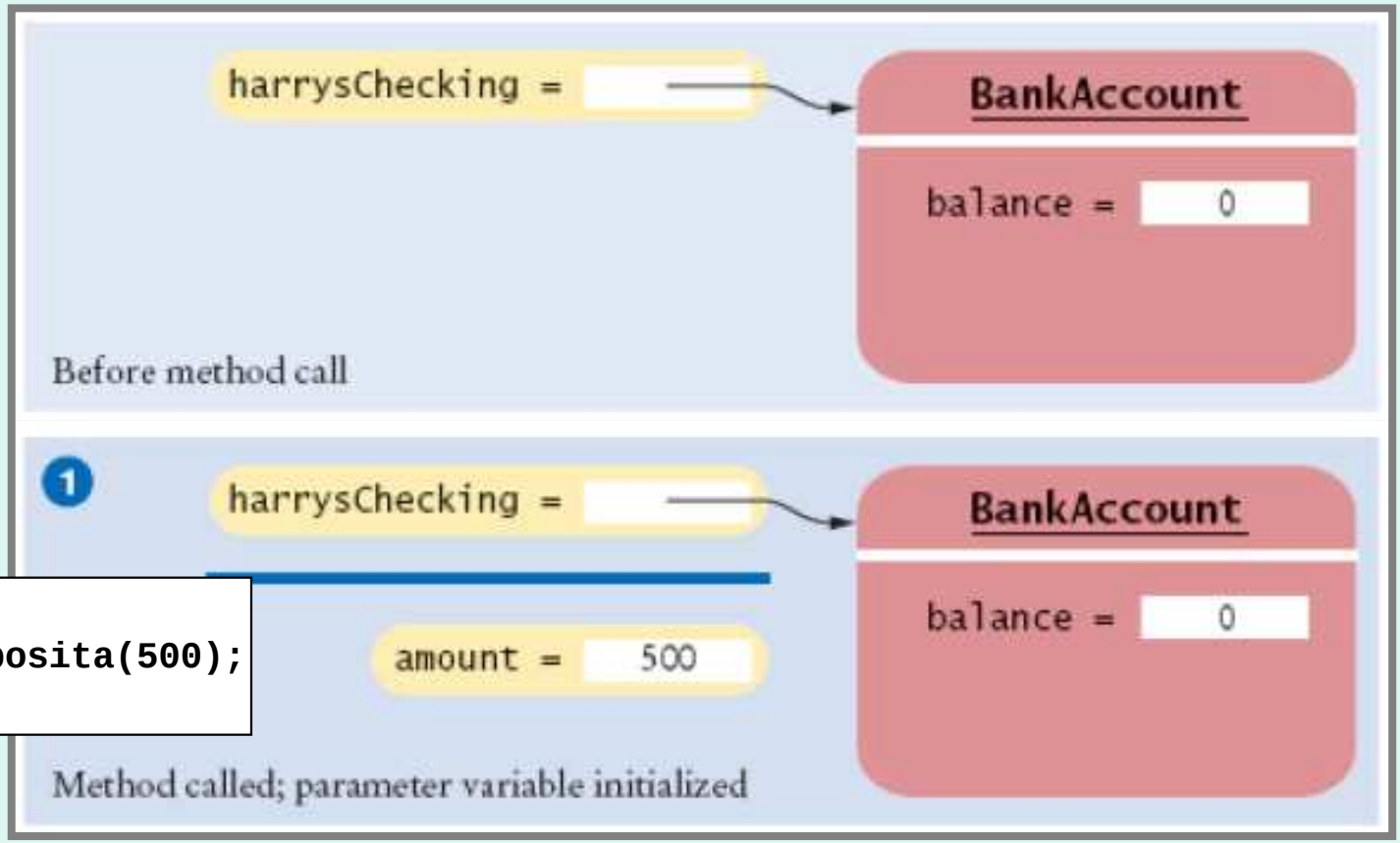



```
harrysChecking.deposita(500);
```

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

Continua…

Lifetime delle variabili

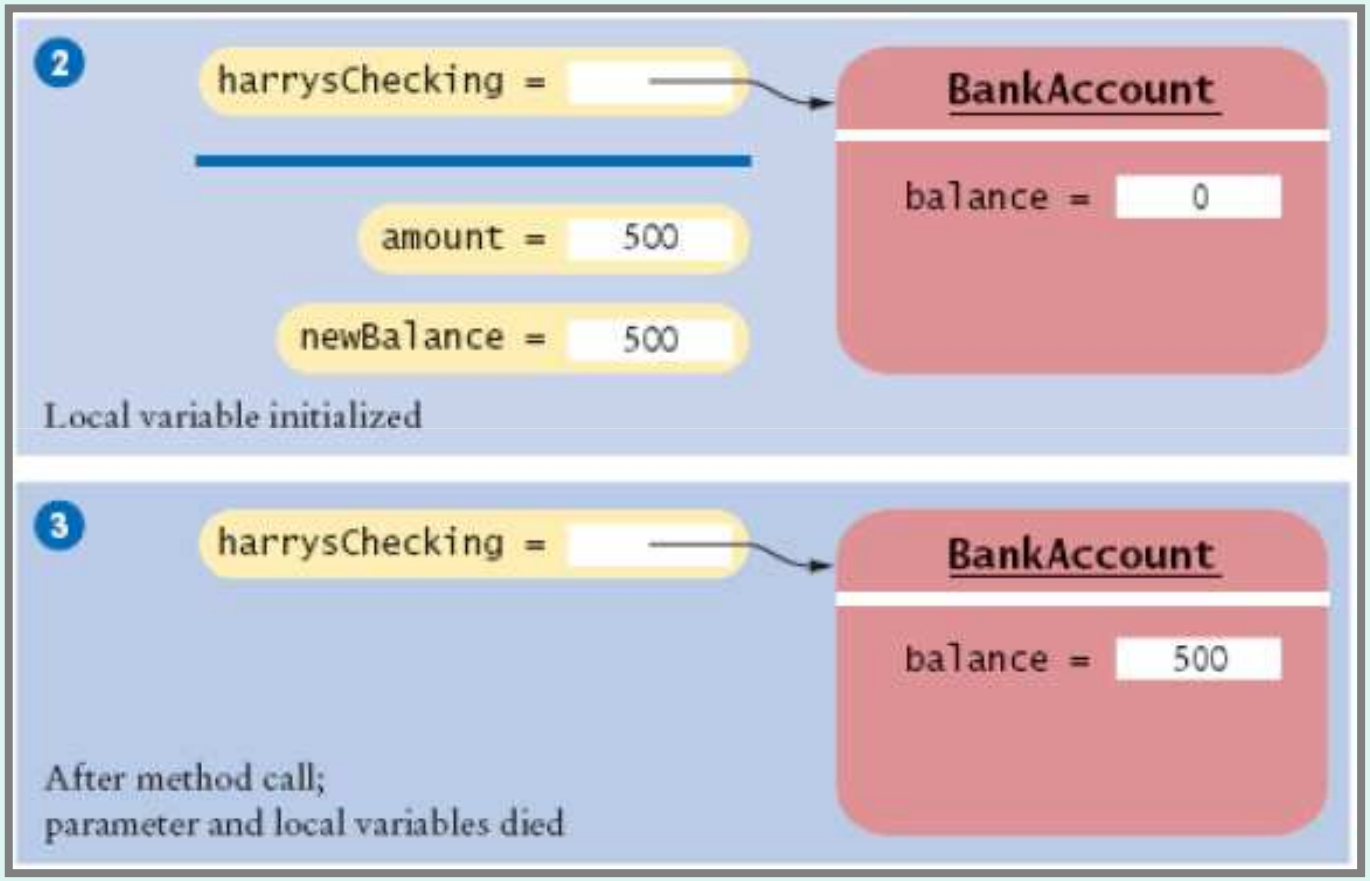



```
harrysChecking.deposita(500);
```

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

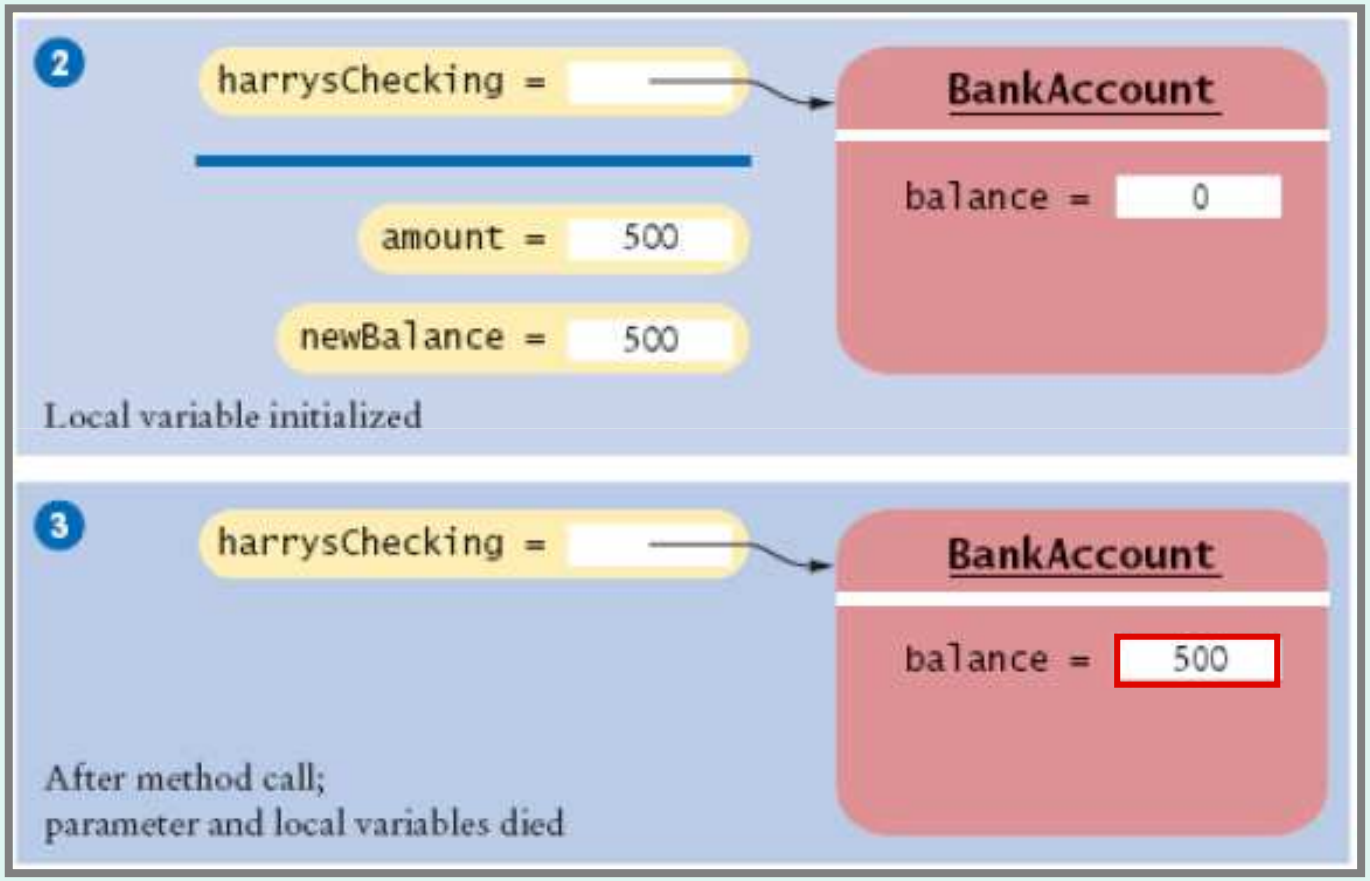
Continua…

Lifetime delle variabili




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

Lifetime delle variabili




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

Domanda

- Quali sono le variabili di istanza e le variabili locali create durante l'esecuzione della applicazione **`BankAccountTester`** ?

```
01: /**
02:    Una classe test per la classe  BankAccount.
03: */
04: public class BankAccountTester
05: {
06:    /**
07:       Testa i metodi della classe BankAccount.
08:       @param args non utilizzato
09:    */
10:    public static void main(String[] args)
11:    {
12:       BankAccount harrysChecking = new BankAccount();
13:       harrysChecking.deposita(2000);
14:       harrysChecking.preleva(500);
15:       System.out.println(harrysChecking.saldo());
16:    }
17: }
```

Risposta

- **Tre variabili locali**
  - **`harrysChecking`**
  - **`newsaldo`** nel metodo **`deposita`**
  - **`newsaldo`** nel metodo **`preleva`**

  Una variabile di istanza, `saldo` all'interno di `harrysChecking`

Parametri impliciti ed espliciti

- Ogni metodo ha un parametro implicito, che rappresenta l'oggetto su cui il metodo viene invocato
- Il parametro implicito è denotato da `this`
- I nomi dei campi nel corpo di un metodo sono riferiti a `this`

this

```
public void preleva(double importo)
{
    double newsaldo = saldo - importo;
    saldo = newsaldo;
}
```

saldo è il campo dell'oggetto su cui il metodo viene invocato

esplicitiamo usando **this**

```
public void preleva(double importo)
{
    double newsaldo = this.saldo - importo;
    this.saldo = newsaldo;
}
```

this

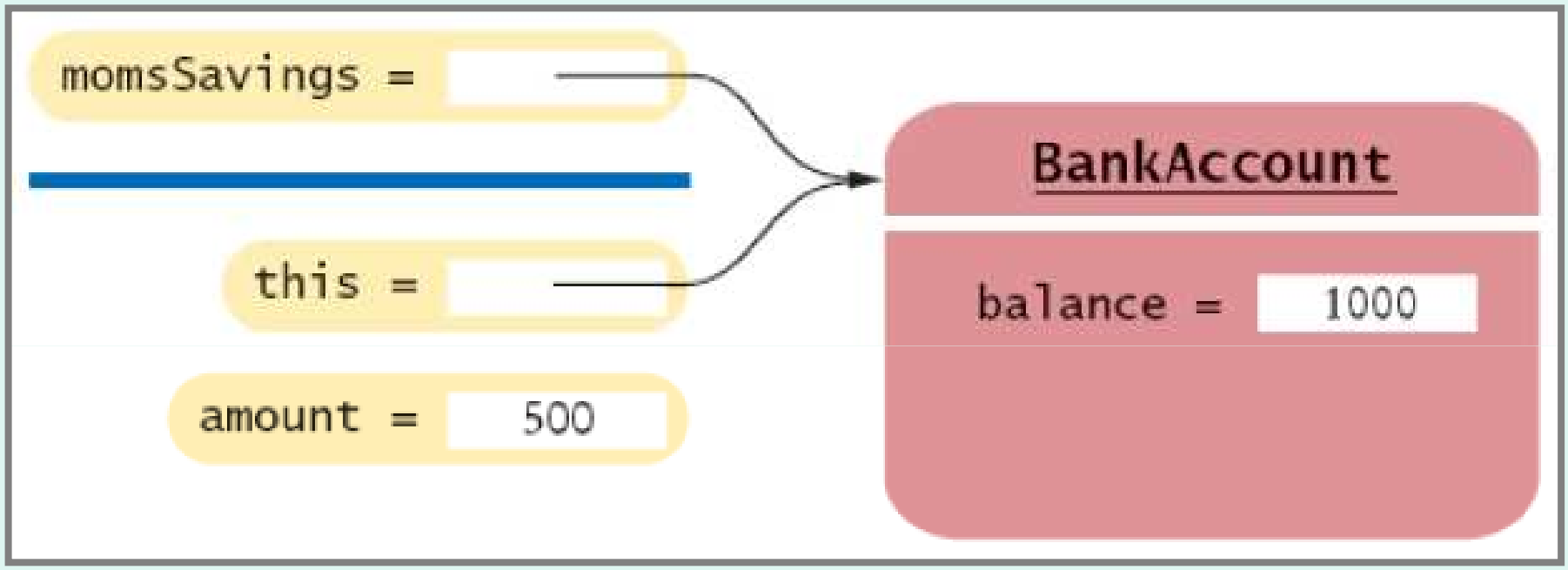
- Quando invochiamo un metodo su un oggetto, questo viene legato a **this.**

```
public void preleva(double importo)
{
   double newsaldo = this.saldo - importo;
   this.saldo = newsaldo;
}
```

- **momsSavings.deposita(500)** esegue il codice seguente

```
   double newsaldo = momsSavings.saldo - importo;
   momsSavings.saldo = newsaldo;
```

this



Domande

- Quale è il tipo del parametro implicito nel metodo `preleva()` della classe `BankAccount`?
- Nel metodo `preleva()`, avrebbe senso utilizzare `this.importo` al posto di `importo`?
- Quali sono i parametri impliciti ed espliciti del metodo `main()` della classe `BankAccountApp`?

Risposte

- `BankAccount`.
- Non è legale: `this` ha tipo `BankAccount` e `BankAccount` non ha un campo `importo`.
- Nessun parametro implicito in quanto il metodo è statico. Un parametro esplicito: `args`.

Domanda

- **Quali sono gli aspetti comuni tra metodi e funzioni? Quali le differenze?**

Risposta

- **I metodi e le funzioni si basano sugli stessi meccanismi di chiamata/ritorno e di passaggio di parametri**

  Differiscono in modo essenziale per il fatto che:

  un metodo è
  - sempre associato ad un oggetto
  - invocato con un messaggio a quell'oggetto

  una funzione è
  - definita in modo indipendente da un oggetto
  - invocata passando un valore come argomento

Passaggio di parametri

- **Call by value:**
  - Copia il valore di ciascun argomento nel corrispondente parametro formale.
  - Non permette di modificare gli argomenti
- **Call by reference:**
  - Associa il riferimento di ciascun argomento al corrispondente parametro formale
  - Permette di modificare gli argomenti
- **In Java: call by value**
  - Ma passando riferimenti “by value” otteniamo l’effetto desiderato

Continua…

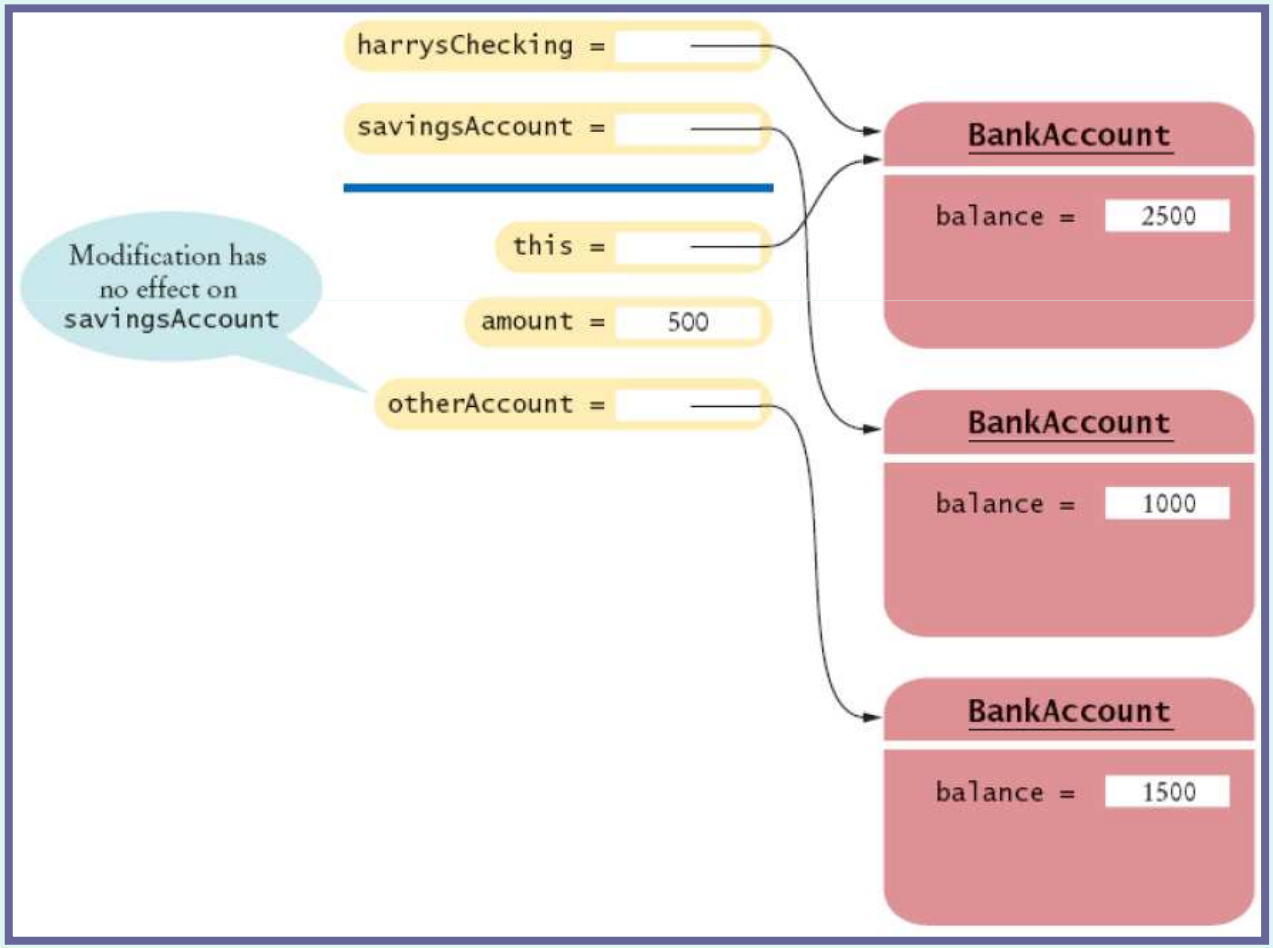
Call by value

- Modifiche dirette dei parametri non hanno effetti sugli argomenti

```
public class BankAccount
{
 public void transfer(double importo, BankAccount otherAccount)
 {
    saldo = saldo - importo;
    double newsaldo = otherAccount.saldo + importo;
    otherAccount = new BankAccount(newsaldo); // ATTENZIONE
 }
}
```

Esempio

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



Call by value

- **Un metodo può comunque modificare lo stato degli argomenti di tipo riferimento (anche se non può modificare direttamente gli argomenti)**

Domanda

- Come modifichereste il codice del metodo `transfer` per ottenere l'effetto desiderato?

Continua…

Risposta

- Due possibili soluzioni

```
public void transfer(double importo, BankAccount other)
 {
    saldo = saldo - importo;
    other.saldo = other.saldo + importo;
 }
```

o meglio ancora

```
public void transfer(double importo, BankAccount other)
 {
    this.preleva(importo);
    other.deposita(importo);
 }
```

Scope delle variabili

- **SCOPE DI UNA VARIABILE: la regione che va dalla sua dichiarazione alla fine del blocco in cui la dichiarazione è inclusa**
- **Al solito, ma esistono diversi tipi di variabili e diversi tipi di blocchi:**
  - Variabili locali: blocco = metodo
  - Campi: blocco dipende dai diritti di accesso definiti nella dichiarazione

Continua…

Scope delle variabili locali

- **Variabili con scope diverso sono diverse indipendentemente dal nome:**

```
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);
   }
}
```

Continua…

Scope delle variabili locali

- **ATTENZIONE: lo scope di una locale non può contenere la dichiarazione di una variabile con lo stesso nome**

```
Rectangle r = new Rectangle(5, 10, 20, 30);
if (x >= 0)
{
   double r = Math.sqrt(x); // ERRORE
   . . .
}
```

… strano ma vero!

Scope dei campi

- **Dipende dalla loro dichiarazione di accesso.**
- **Campi `private:` accessibili da qualunque punto della classe (in particolare in tutti i metodi della classe)**
- **Campi `public:` accessibili da tutti i metodi della classe e da metodi di altre classi**

regole valgono uniformemente tutti gli elementi della classe: campi, metodi, …

Scope e diritti di accesso

```
public class A {
  private int privata;
  public  int pubblica;
  private int privM (){ return privata; }
  public  int pubM () { return privM()+ pubblica; }
  public  int binM (A other){ return privata + other.privata;}
}
```

```
public class Test {
   public void accessi(){
    A a1 = new A();
    a1.privata = 3;
    a1.pubblica = 6;
    a1.privM();
    a1.pubM ();
    A a2 = new A();
    a2.binM(a1);
   }
}
```

Scope e diritti di accesso

```
public class A {
  private int privata;
  public  int pubblica;
  private int privM (){ return privata; }
  public  int pubM () { return privata + pubblica; }
  public  int binM (A other){ return privata + other.privata;}
}
```

```
public class Test {
   public void accessi(){
    A a1 = new A();
    a1.privata = 3; // KO
    a1.pubblica = 6;// OK
    a1.privM();     // KO
    a1.pubM ();     // OK
    A a2 = new A();
    a2.binM(a1);    // OK
   }
}
```

Scope e diritti di accesso

- All'interno di un metodo non è necessario qualificare i campi o metodi dell'oggetto corrente
- Membri non qualificati sono riferiti a `this`

```
public class BankAccount
{
   public void transfer(double importo, BankAccount other)
   {
      preleva(importo);      // this.preleva(importo);
      other.deposita(importo);
   }
   ...
}
```

Scope sovrapposti

- Una variabile locale, o un parametro, possono mascherare un campo con lo stesso nome
-

```
public class Coin
{
   . . .
   public double getExchangeValue(double exchangeRate)
   {
      double value; // Variabile locale
      . . .
      return value;
   }
   private String name;
   private double value; // Campo con lo stesso nome
}
```

Continua…

Scope sovrapposti

- I campi mascherati da locali sono ancora accessibili mediante accessi qualificati da **`this`**

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

Costruttori

- **regole sintattiche:**
  - il nome del costruttore deve essere quello della classe
  - non ha un tipo di ritorno
  - puo' essere dotato di qualsiasi modificatore di accesso
- **ogni classe ha almeno il *default constructor* senza parametri.**
- **Il default constructor ha lo stesso livello di accessibilita' della classe per cui e' definito**

Costruttori annidati

Quando si hanno piu' costruttori, questi possono anche chiamarsi in modo "annidato", usando `this(...)`

```
public class BankAccount {
        private int accountNumber;
        private double saldo;

        public BankAccount(double saldo) {
          this.saldo = saldo;
        }

        public BankAccount() {
          this(0);
        }
}
```

deve necessariamente essere **il primo** statement del costrutture

Costruttori e Inizializzazioni

Per garantire l'inizializzazione dei campi di un oggetto possiamo usare anche inizializzazioni esplicite:

- Assegnando dei valori ai campi dati direttamente al momento della loro dichiarazione
- Scrivendo dei *blocchi di inizializzazione:* blocchi di codice definiti dentro la classe ma fuori da metodi e costruttori
- I blocchi di inizializzazione sono utili per definire *una parte di codice comune a tutti i costruttori*

Costruttori e Inizializzazioni

```
public class Init {
   // inizializzazioni esplicite: 1°
   private int id;
   private int x = 65;
   private String name;
   private Rectangle d = new Rectangle();
   private static int nextId = 1;

    // blocco di inizializzazione: 2°
   {
    id = nextId;
    nextId = nextId +1;
   }
    // costruttori: 3°
   Init(String n){
        name=n;
   }
   Init(String n, Rectangle d){
        name=n;this.d=d;
   }
}
```