

Classi

-
- **Progetto di classi:**
 - Dalla specifica dell'interfaccia, alla definizione dell'implementazione
 - **Metodi**
 - **Costruttori**
 - **Documentazione e commenti**
 - **Variabili di istanza (campi)**

Progetto di una classe

- **Un conto bancario (BankAccount)**
- **Individuiamo il comportamento del conto**
- **Metodi**
 - deposita un importo
 - preleva un importo
 - richiedi il saldo

Metodi

- **Metodi della classe `BankAccount`:**

```
deposita()  
preleva()  
saldo()
```

- **Vogliamo fornire supporto per chiamate di metodo quali:**

```
harrysChecking.deposita(2000);  
harrysChecking.preleva(500);  
System.out.println(harrysChecking.saldo());
```

Metodi

Specifica:

- **livello di accesso** (`public`)
- **tipo del risultato** (`String, ..., void`)
- **nome del metodo** (`deposita`)
- **lista di parametri per ciascun metodo**
(`double importo`)

Continua...

Sintassi: definizione di metodi

```
accessSpecifier returnType methodName(parType parName,...)  
{  
    corpo del metodo  
}
```

Esempio:

```
public void deposita(double importo)  
{  
    . . .  
}
```

Metodi per BankAccount

```
public void deposita(double importo) { . . . }  
public void preleva(double importo) { . . . }  
public double saldo() { . . . }
```

Costruttore

- Un costruttore inizializza i campi dell'oggetto creato
- Nome del costruttore = nome della classe

```
public BankAccount()  
{  
    // corpo . . .  
}
```

Continua...

Costruttore

- **Il corpo del costruttore**
 - eseguito quando l'oggetto viene creato con **new**
 - definisce la struttura dell'oggetto creato
- **Una classe può avere più costruttori, tutti con lo stesso nome (il nome della classe) purché tutte abbiano parametri diversi**
- **Il compilatore distingue le diverse definizioni basandosi sul tipo degli argomenti**

Sintassi: definizione del costruttore

```
accessSpecifier ClassName(parameterType parameterName, . . .)  
{  
    corpo del costruttore  
}
```

Esempio:

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

BankAccount: Interfaccia pubblica

- I costruttori ed i metodi `public` di una classe formano l'interfaccia pubblica della classe

```
class BankAccount
{
    // Costruttori
    public BankAccount()
    {
        // corpo da definire
    }
    public BankAccount(double initialsaldo)
    {
        // corpo da definire
    }

    // Metodi
    public void deposita(double importo) Continua...
```

BankAccount: **Interfaccia pubblica**

```
{  
    // corpo da definire  
}  
  
public void preleva(double importo)  
{  
    // corpo da definire  
}  
  
public double saldo()  
{  
    // corpo da definire  
}  
  
// campi privati / rappresentazione da definire  
}
```

Sintassi: definizione di classe

```
accessSpecifier class ClassName
{
    costruttori
    metodi
    campi (rappresentazione)
}
```

Esempio:

```
public class BankAccount
{
    public BankAccount(double saldoIniziale) { . . . }
    public void deposita(double importo) { . . . }
    . . .
}
```

- A volte introdurremo i campi come prima cosa

Domanda

- **Come possiamo utilizzare il metodi dell'interfaccia pubblica per azzerare il saldo del conto `harrysChecking`?**

Risposta

-

```
harrysChecking.preleva(harrysChecking.saldo())
```

Documentazione

```
/**  
    Un conto bancario ha un saldo che può essere  
    modificato mediante depositi e prelievi.  
*/  
class BankAccount  
{  
    . . .  
}
```

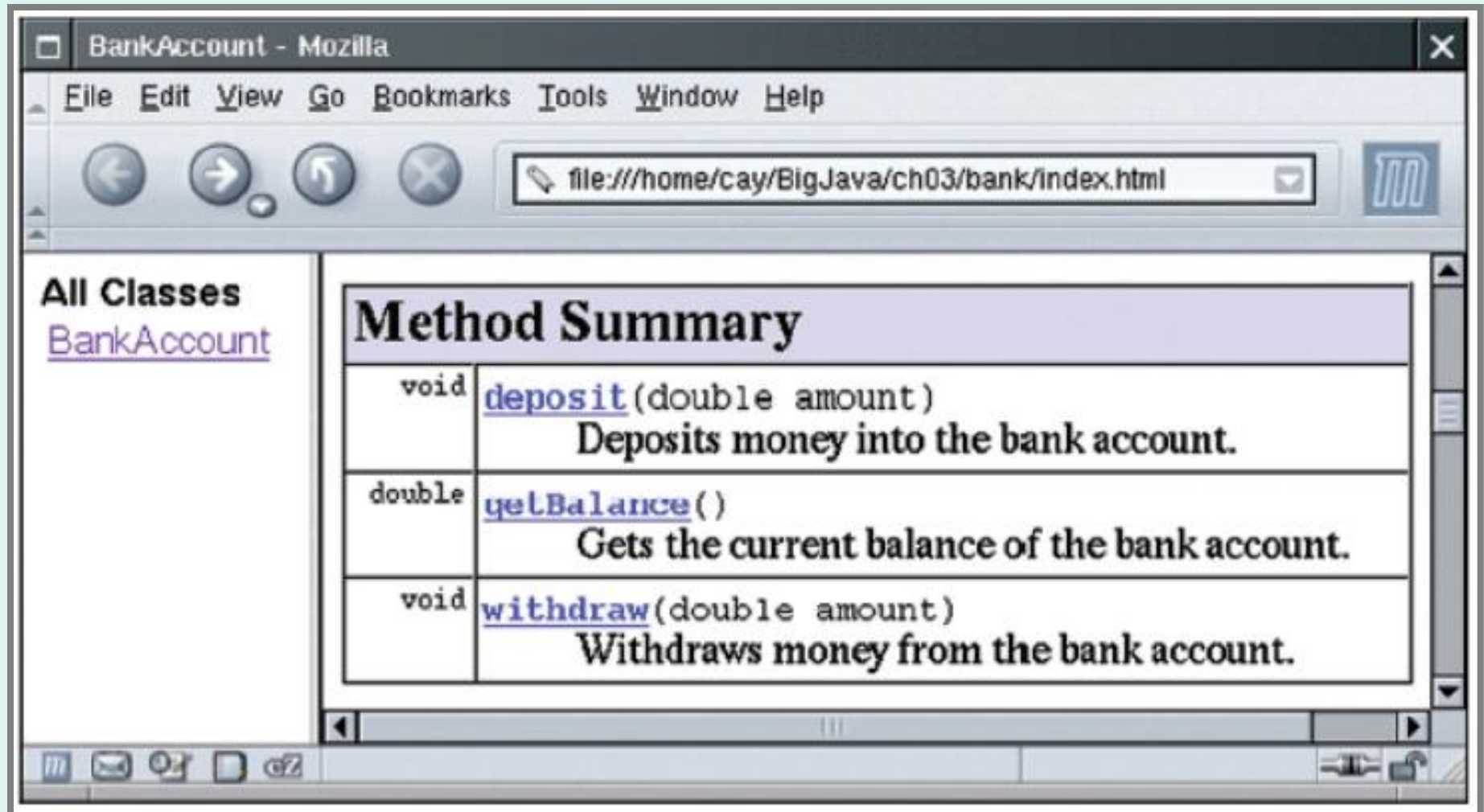
- **Fornire documentazione per**
 - Tutte le classi
 - Tutti i metodi
 - Tutti i parametri
 - Tutti i valori restituiti

Documentazione dell'interfaccia

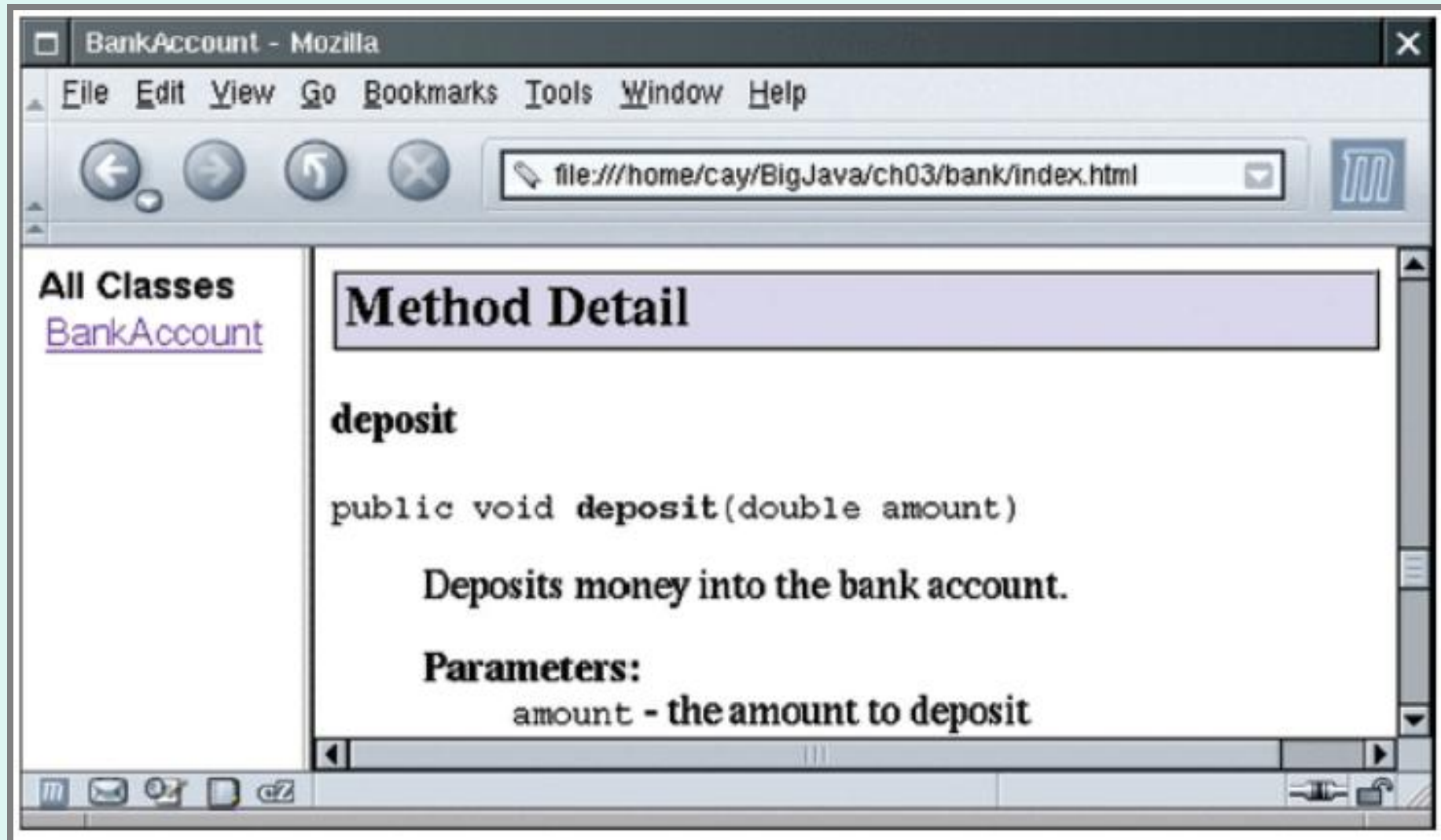
```
/**
    Preleva dal conto.
    @param: importo - l'importo da prelevare
 */
public void preleva(double importo)
{
    // definizione in seguito
}
```

```
/**
    Restituisce il saldo corrente del conto.
    @return il saldo
 */
public double saldo()
{
    // definizione in seguito
}
```

Javadoc



Javadoc



Campi

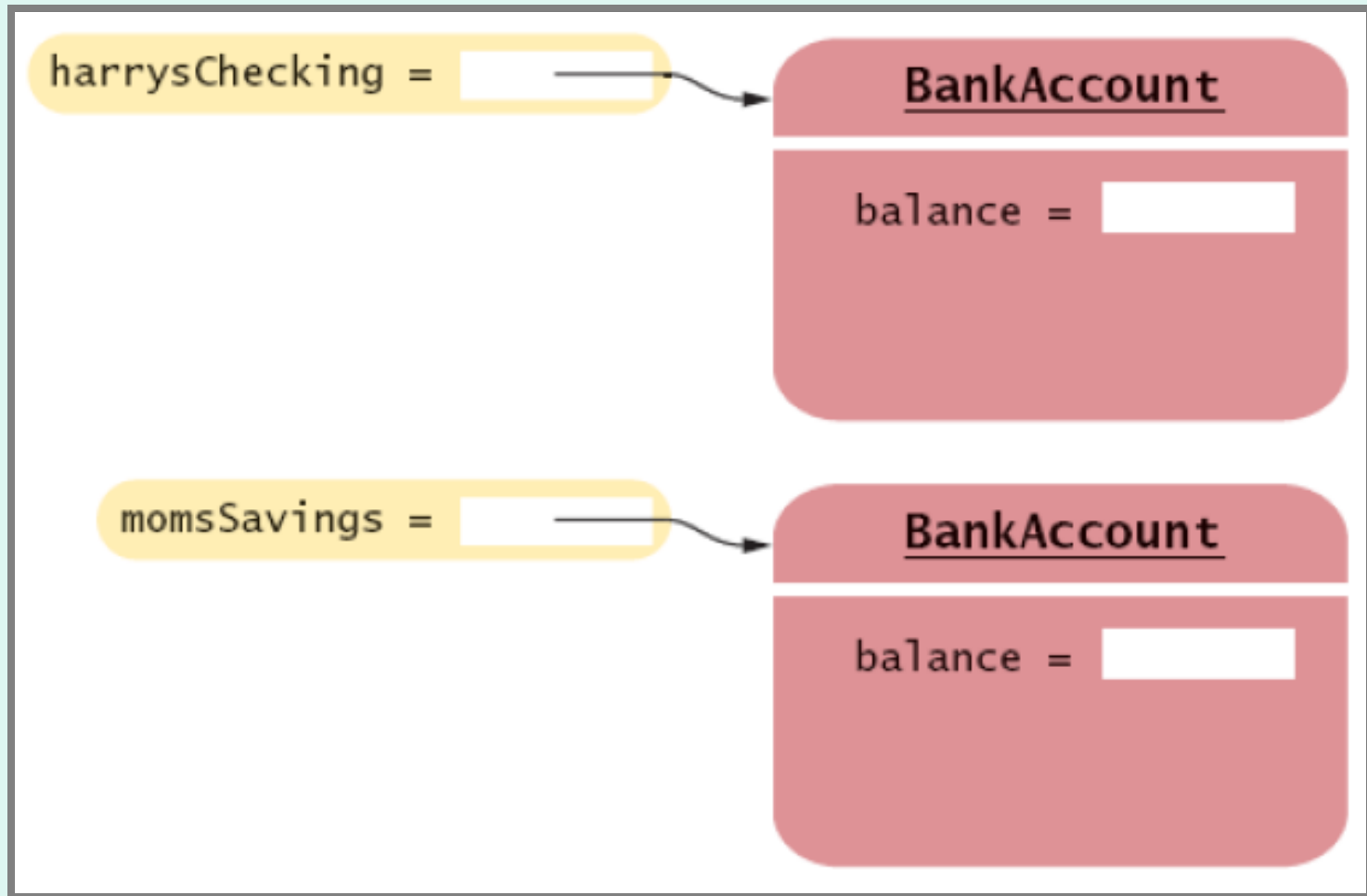
- Gli oggetti memorizzano il proprio stato ed i propri dati nei campi
- La definizione di classe introduce i campi, dichiarandoli

```
class BankAccount
{
    . . .
    private double saldo;
}
```

Campi

- **La dichiarazione di un campo include:**
 - Specifica dei diritti di accesso (**private**)
 - Tipo della variabile (**double**)
 - Nome della variabile (**saldo**)
- **Ciascun oggetto di una classe ha una copia distinta dei propri campi**

Campi



Sintassi: dichiarazione di campi

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

Esempio:

```
class BankAccount
{
    . . .
    private double saldo;
    . . .
}
```

Accesso ai campi

- I campi non sono parte dell'interfaccia
- **private** quindi visibili solo ai metodi della classe, non al codice esterno alla classe
- Il metodo `deposita()` di `BankAccount` può accedere al campo `saldo`:

```
public void deposita(double importo)
{
    double nuovoSaldo = saldo + importo;
    saldo = nuovoSaldo;
}
```

Continua...

Accesso ai campi

- **Metodi di altre classi, all'inverso, non possono accedere ai campi di BankAccount**

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

Accesso ai campi

- **Encapsulation**

- raggruppare i dati e le operazioni in classi non è sufficiente a garantire l'effetto black box per le istanze
- Necessaria l'assistenza del compilatore

- **Garantisce “hiding” di dati**

- Per ogni classe esporta l'interfaccia pubblica (il *contratto* tra la classe e i clienti) e maschera l'implementazione privata che manipola lo stato dell'oggetto
- In questo modo i i clienti dipendono solo dal contratto, e possiamo re-implementare i metodi senza dover modificare i clienti (*codice mantenibile*)

Implementazione di costruttori

- I costruttori definiscono codice per inizializzare i campi

```
public BankAccount()  
{  
    saldo = 0;  
}  
public BankAccount(double saldoIniziale)  
{  
    saldo = saldoIniziale;  
}
```

Chiamata di un costruttore

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

  - Crea un nuovo oggetto di tipo **BankAccount**
  - Chiama il secondo costruttore (visto che la chiamata fornisce un parametro)
  - Inizializza il campo **saldo** dell'oggetto appena creato al valore 1000
  - Restituisce un riferimento all'oggetto appena creato come risultato della valutazione dell'espressione **new**
  - Memorizza il riferimento nella variabile **harrysChecking**

# Implementazione di metodi

- **Un metodo che non restituisce valori**

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

- **Un metodo che restituisce un valore**

```
public double saldo()
{
 return saldo;
}
```

- **Nulla di strano**

# File BankAccount.java

```
01: /**
02: Un conto bancario ha un saldo che può essere
03: modificato mediante depositi e prelievi.
04: */
05: public class BankAccount
06: {
07: /**
08: Costruisce un conto bancario con saldo zero
09: */
10: public BankAccount()
11: {
12: saldo = 0;
13: }
14:
15: /**
16: Costruisce un conto con un dato saldo iniziale.
17: @param initialsaldo il saldo iniziale
18: */
```

*Continua...*

# File BankAccount.java

```
19: public BankAccount(double initialsaldo)
20: {
21: saldo = initialsaldo;
22: }
23:
24: /**
25: deposita un importo sul conto.
26: @param importo l'importo da depositare
27: */
28: public void deposita(double importo)
29: {
30: double newsaldo = saldo + importo;
31: saldo = newsaldo;
32: }
33:
34: /**
35: Preleva un importo dal conto.
36: @param importo l'importo da prelevare
```

*Continua...*

# File BankAccount.java

```
37: */
38: public void preleva(double importo)
39: {
40: double newsaldo = saldo - importo;
41: saldo = newsaldo;
42: }
43:
44: /**
45: Restituisce il saldo corrente.
46: @return saldo corrente
47: */
48: public double saldo()
49: {
50: return saldo;
51: }
52: // rappresentazione
53: private double saldo;
54: }
```



# Domanda

---

- Come modifichereste la definizione della classe `BankAccount.java` per associare a ciascun conto un numero?

# Risposte

- Aggiungendo un nuovo campo

```
private int accountNumber;
```

e definendo come segue l'implementazione del costruttore e del metodo di accesso

```
public BankAccount(double initsaldo, int number)
{
 saldo = initsaldo; accountNumber = number;
}

public int getAccountNumber()
{
 return accountNumber;
};
```

# Classi Test / Applicazione

---

## Ricordiamo

- **Classe applicazione:** una classe con un metodo `main` che contiene codice lanciare la computazione utilizzando le classi che compongono il programma
- **La struttura tipica di una classe test:**
  1. Costruisci uno o più oggetti
  2. Invoca i metodi sulle diverse istanze
  3. Stampa i risultati

*Continua...*

# Classi Test / Applicazione

---

- **Le prassi per costruire applicazioni che consistono di più classi variano a seconda del sistema ospite.**
- **Tipicamente i passi sono:**
  1. Crea una nuova directory / folder
  2. Costruisci un file per classe e includi nella directory
  3. Compila tutti i file della directory
  4. Lancia la classe test

# File BankAccountApp.java

```
01: /**
02: Una classe applicazione per la classe BankAccount.
03: */
04: public class BankAccountApp
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: }
```

# Domanda

---

7. Quando lanciamo la classe `BankAccountApp`, quante istanze della classe `BankAccount` vengono costruite? Quanti oggetti di tipo `BankAccountApp`?

# Risposta

---

7. **Un oggetto di tipo `BankAccount`, nessun oggetto di tipo `BankAccountTester`.**

**Lo scopo della classe applicazione è solo quello di definire il metodo `main`.**

# Esercizio

---

- **Vogliamo estendere la classe `BankAccount` fornendo ai conti bancari le seguenti caratteristiche:**
  - per eseguire le operazioni di prelievo, deposito e richiesta saldo su un conto e' necessario prima autenticarsi
  - Una volta che ci siamo autenticati, inizia una sessione durante la quale possiamo eseguire le operazioni;
  - Al termine delle operazioni possiamo chiudere la sessione.



# Tombola!

---

- Vediamo l'implementazione delle classi che dell'applicazione che simula una tombola
- Utilizziamo la classe di libreria
  - Random
- Definiamo le due classi
  - Banco
  - Giocatore

# Random

---

- **Costruttore:**

- `Random(long seed)`

crea un generatore a partire da seed

- **Metodi:**

- `public int nextInt(int n)`

restituisce un numero pseudocasuale nell'intervallo  $[0..n-1]$  ottenuto dalla sequenza di numeri casuali associati al generatore

# Banco

---

- **Costruttore:**

- `Banco()`

- crea una nuova istanza della classe

- **Metodi:**

- `int estraiNumero()`

- restituisce un numero nell'intervallo [1..90]

- (ad ogni chiamata genera un numero diverso)

# Giocatore

---

- **Costruttore**

- `Giocatore(String nome)`

- crea un giocatore, con il nome indicato e una scheda

- **Metodi:**

- `void controllaNumero(int)`

- se **x** è contenuto nella scheda lo marca

- `boolean tombola()`

- true quando tutti i numeri della scheda sono marcati

- `int[] verifica()`

- restituisce un array con i numeri della scheda

# **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 e 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);
```

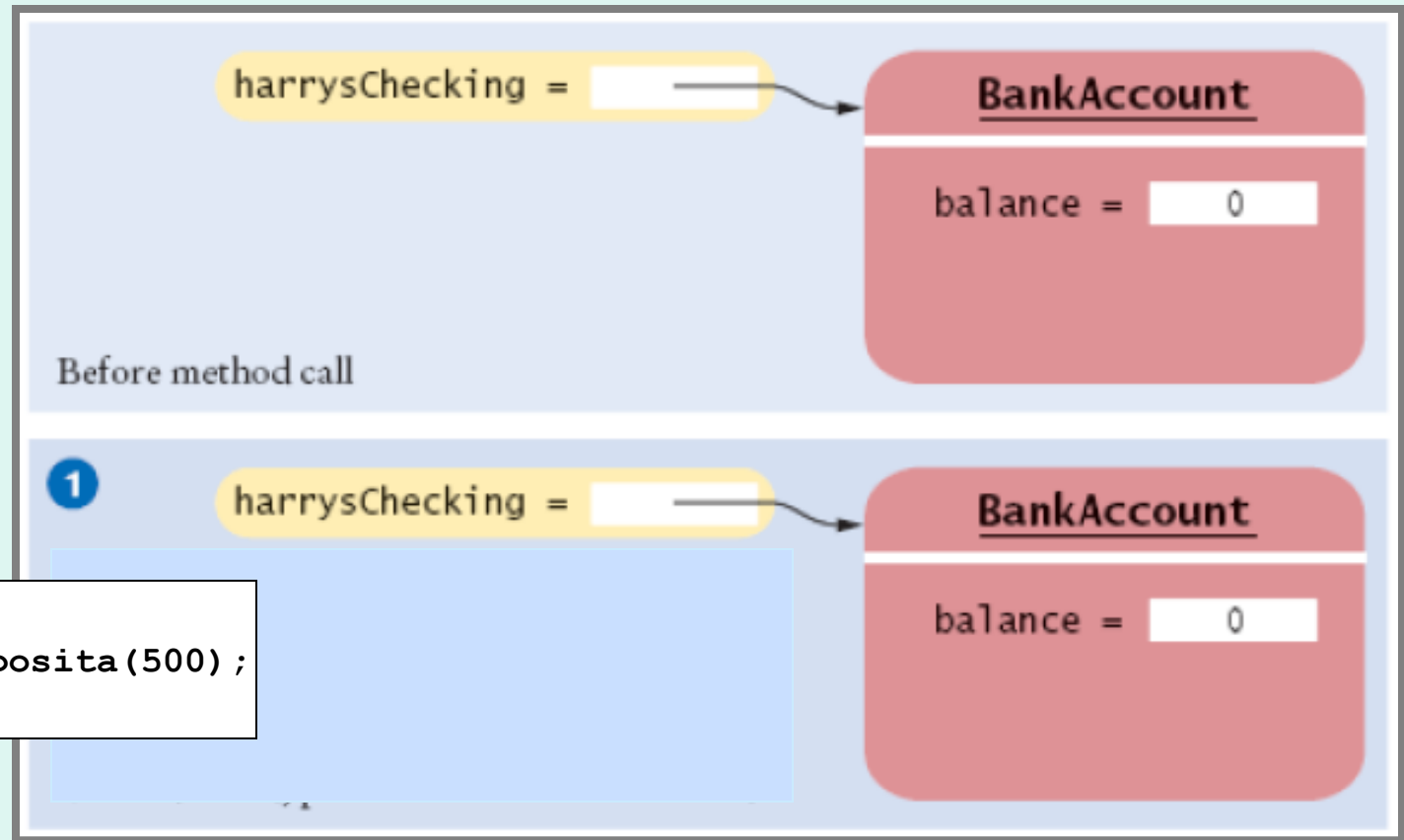
**ricordando**

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

*Continua...*



# Lifetime delle variabili

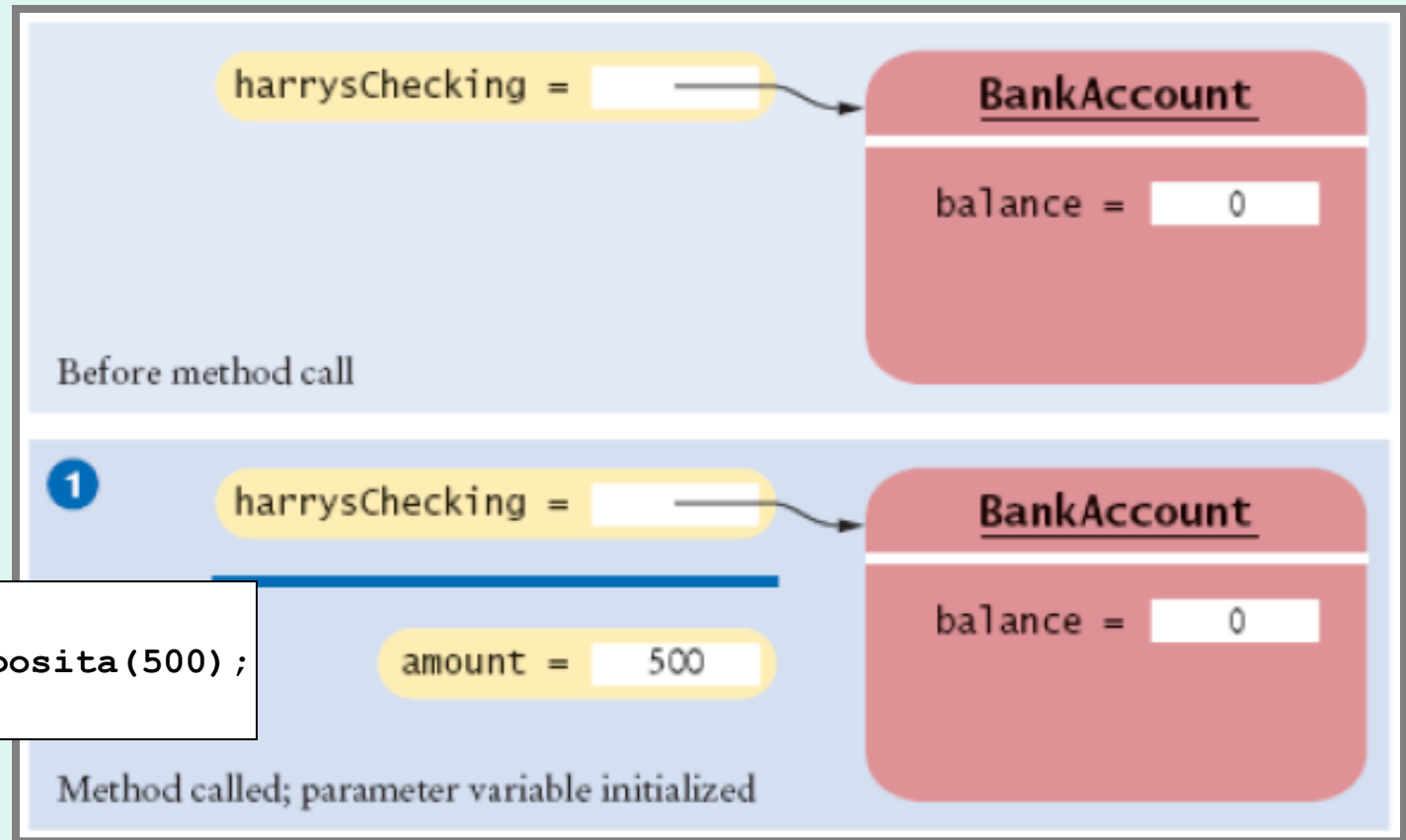


```
harrysChecking.deposita(500);
```

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

*Continua...*

# Lifetime delle variabili

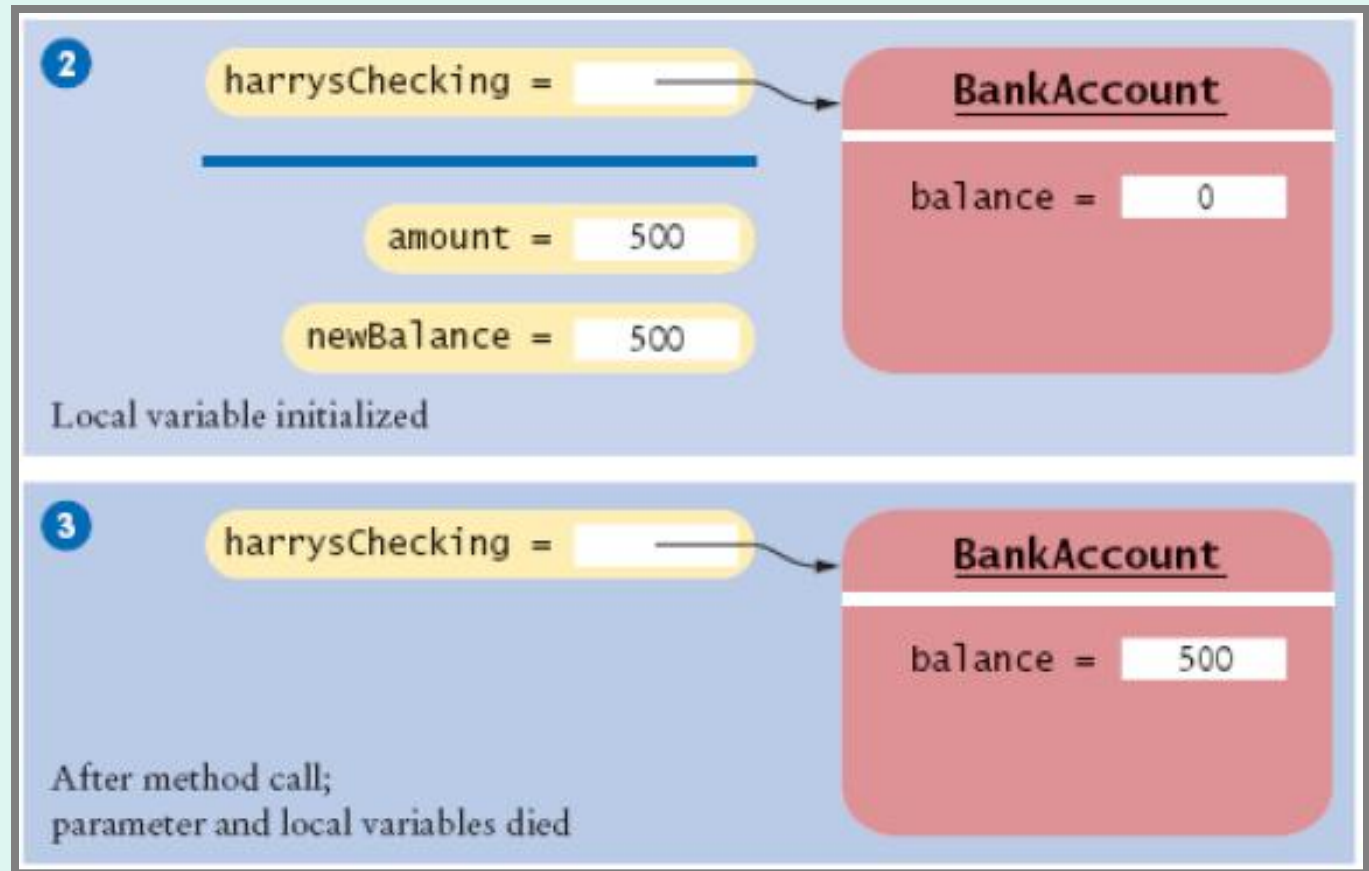


```
harrysChecking.deposita(500);
```

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

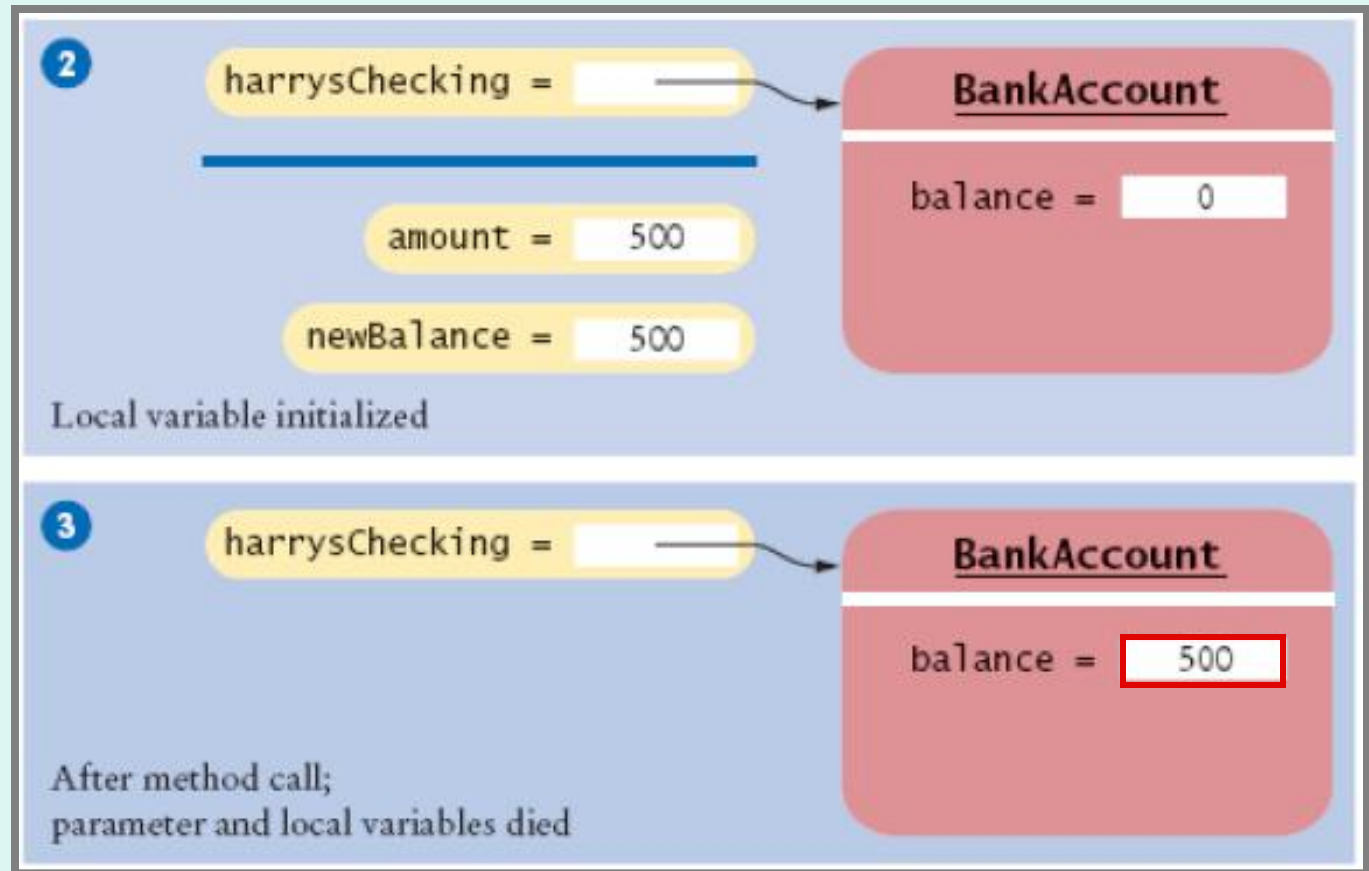
*Continua...*

# 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;
}
```

# 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

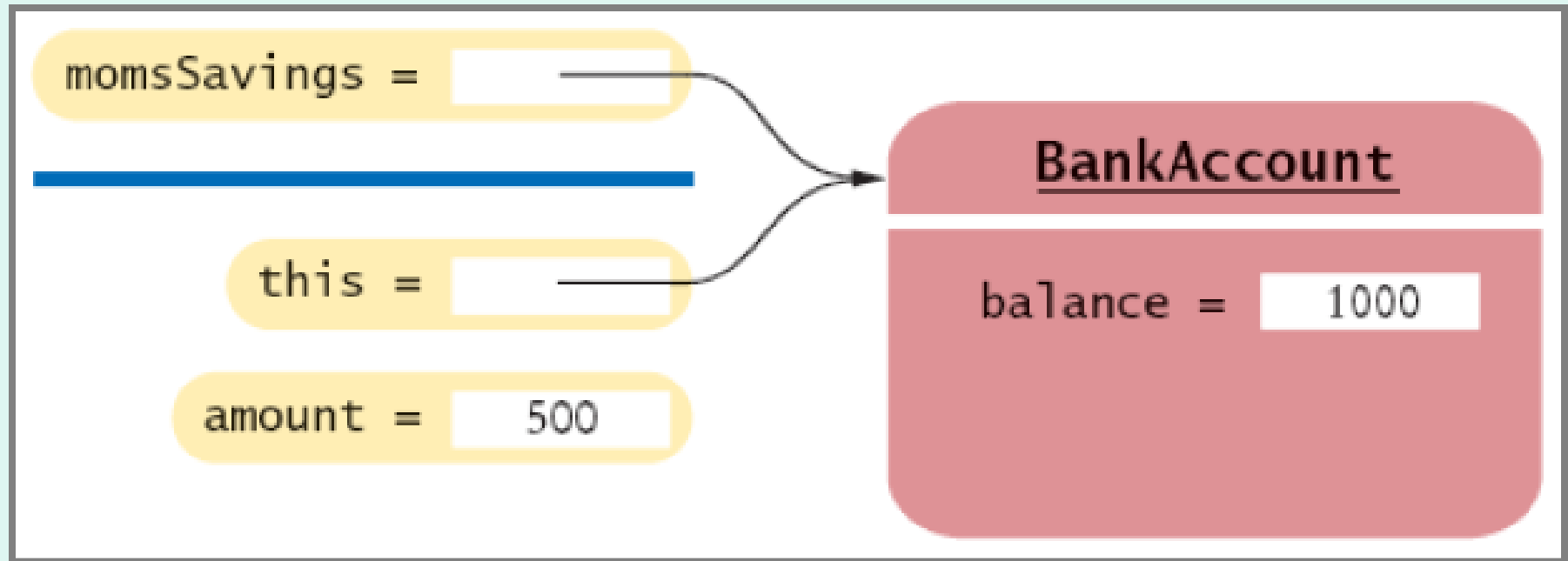
- Quando invochiamo un metodo su un oggetto, questo viene legato a **this**.

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

- **momsSavings.preleva(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...*

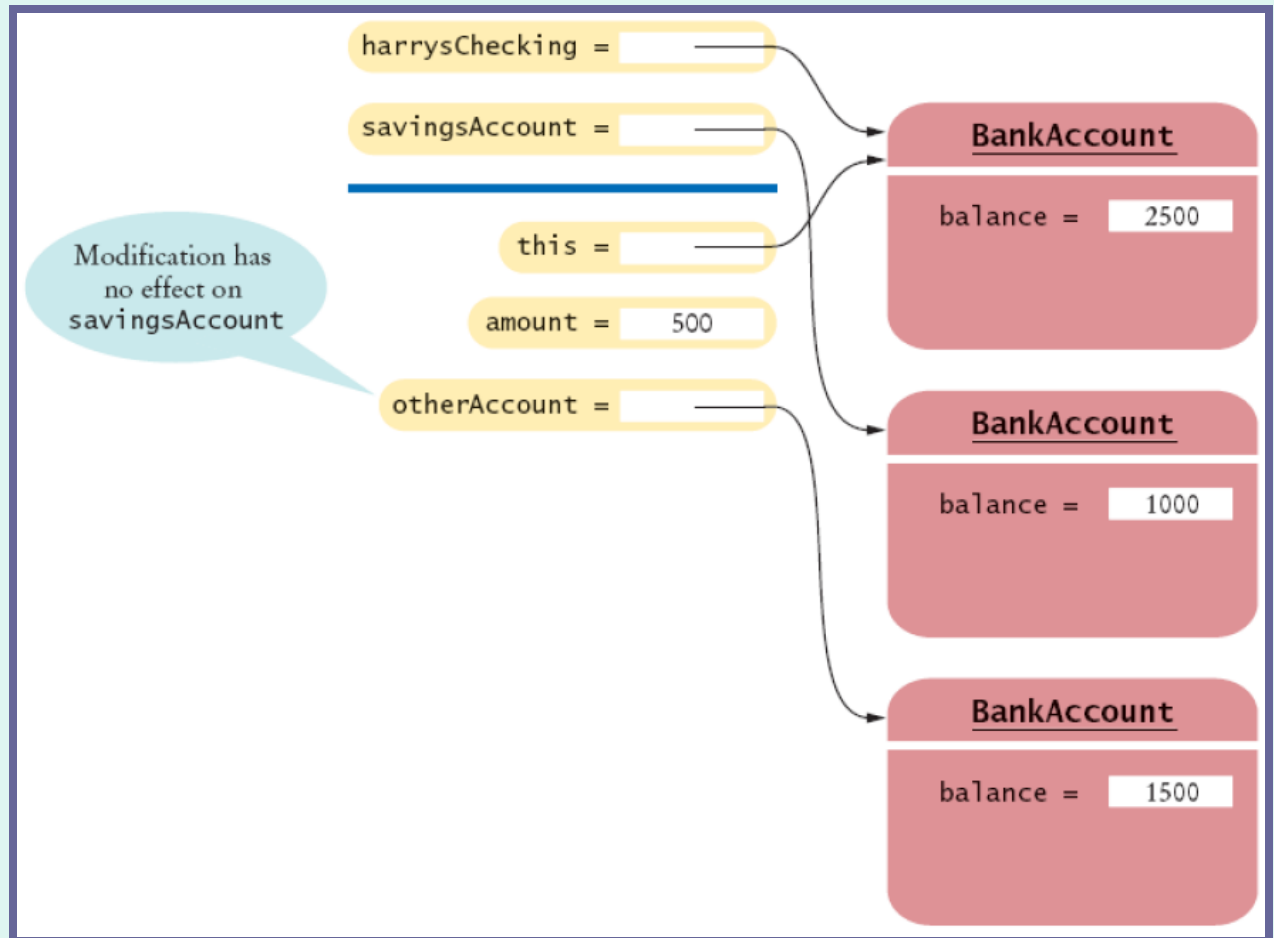
# Call by value

- 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)
{
 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 costruttore

# 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;
 }
}
```

**Static**



# Metodi static

- Esempio

```
public class Financial
{
 public static double percentOf(double p, double a)
 {
 return (p / 100) * a;
 }
 // . . .
}
```

# Metodi static

- Altri esempi ben noti
  - `main()`
  - i metodi della classe `Math`

|                                                                                   |                                             |
|-----------------------------------------------------------------------------------|---------------------------------------------|
| <code>Math.sqrt(x)</code>                                                         | Radice quadrata                             |
| <code>Math.pow(x, y)</code>                                                       | Potenza $x^y$                               |
| <code>Math.exp(x)</code>                                                          | <i>Esponenziate <math>e^x</math></i>        |
| <code>Math.log(x)</code>                                                          | Logaritmo naturale                          |
| <code>Math.sin(x)</code> , <code>Math.cos(x)</code> ,<br><code>Math.tan(x)</code> | Funzioni trigonometriche<br>(x in radianti) |
| <code>Math.round(x)</code>                                                        | Attotondamento                              |
| <code>Math.min(x,y)</code> , <code>Math.max(x,y)</code>                           | Minimo, massimo                             |

# Metodi static

---

- **Sono definiti all'interno di classe, ma ...**
  - non sono associati ad oggetti
  - non sono invocati su oggetti
- **Quando definire un metodo static?**
  - quando il metodo opera solamente sui propri parametri

# Metodi static

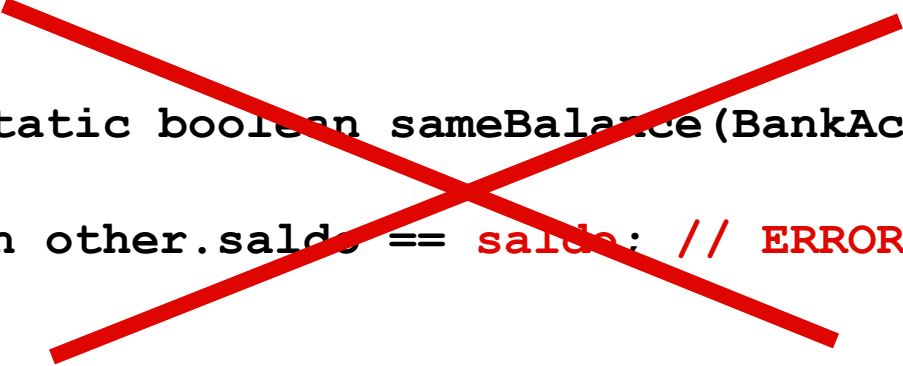
---

- Poichè appartengono alla classe e non sono riferibili ad alcuna istanza, all'interno di questi metodi il parametro implicito `this` è indefinito
- Quindi all'interno di questi metodi non si può far riferimento ad alcun campo dell'oggetto corrente

# Metodi static

- Attenzione agli accessi

```
public class BankAccount
{
 . . .
 public static boolean sameBalance(BankAccount other)
 {
 return other.saldo == saldo; // ERRORE!
 }
 . . .
}
```



# Metodi static

- Correggiamo così

```
public class BankAccount
{
 . . .
 public static boolean sameBalance (BankAccount other)
 {
 return other.saldo == saldo; // OK
 }
 . . .
}
```

# Metodi static

- Correggiamo così

```
public class BankAccount
{
 . . .
 public static boolean sameBalance(BankAccount b1,
 BankAccount b2)
 {
 return b1.saldo == b2.saldo; // OK
 }
 . . .
}
```

# Metodi static

- Oppure così

```
public class BankAccount
{
 . . .
 public static boolean sameBalance(BankAccount b1,
 BankAccount b2)
 {
 return b1.saldo == b2.saldo; // OK
 }
 . . .
}
```



# Chiamata di metodi statici

---

- **All'interno della classe in cui sono definiti**
  - Stessa sintassi utilizzata per i metodi non-static
- **All'esterno della classe di definizione**
  - Utilizziamo **il nome della classe** invece del riferimento ad un oggetto:

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

# Domande

---

- È corretto invocare `x.pow(y)` per calcolare  $x^y$ ?
- Quale delle due istruzioni è più efficiente tra `x*x` e `Math.pow(x, 2)`

# Risposte

---

- No: **x** è un numero, non un oggetto, e non è possibile invocare un metodo su un numero
- **x\*x** la seconda espressione coinvolge una chiamata di metodo, passaggio di parametri ... etc, tutte operazioni costose

# Domanda

---

- **Come utilizzereste la classe `MyMath` qui di seguito per calcolare la radice quadrata di 5?**

```
public class MyMath
{
 public double sqrt(double x)
 {
 return Math.sqrt(x) ;
 }
}
```

# Risposta

---

```
(new MyMath()) .sqrt(5) ;
```

# Campi static (class variables)

- Un campo static appartiene alla classe, non ad alcuna delle istanze della classe

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

- Una sola copia di `lastAssignedNumber`, accessibile solo all'interno della classe

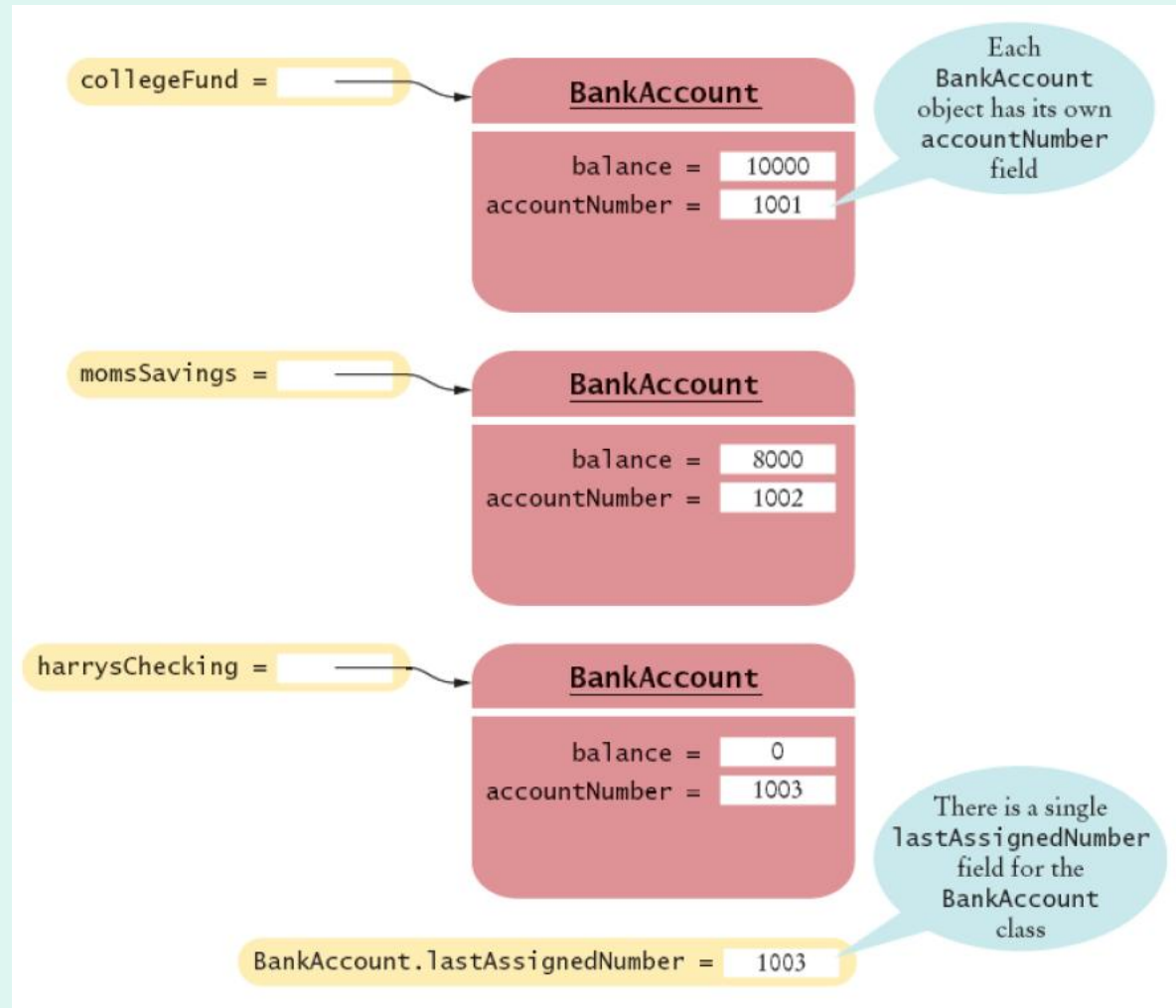
# Campi static

- **La nuova definizione del costruttore**

```
/**
 Genera il prossimo numero di conto da assegnare
 */
public BankAccount()
{
 lastAssignedNumber++;
 accountNumber = lastAssignedNumber;
}
```

- **Questo è un uso tipico dei campi static**
  - Altro caso di utilizzo: definizione di costanti (campi static e final)

# Campi e campi static





# Campi static

## Tre modalità di inizializzazione:

1. Automatica: inizializzati ai valori di default.
2. Mediante un inizializzatore esplicito

```
public class BankAccount
{
 . . .
 private static int lastAssignedNumber = 1000;
 // Eseguito una volta sola,
 // quando la classe viene caricata
}
```

3. Mediante un blocco di inizializzazione static

*Continua...*

# Campi static

---

- **Allocazione/Deallocazione:**
  - vengono creati quando viene caricata la classe,
  - continuano ad esistere finchè esiste la classe, quindi durante tutta l'esecuzione.

# Campi static

- Come tutti i campi, preferibilmente dichiarati `private`
- Eccezioni: costanti

```
public class BankAccount
{
 . . .
 public static final double OVERDRAFT_FEE = 5;
 // riferita mediante il nome qualificato
 // BankAccount.OVERDRAFT_FEE
}
```

# Accesso ai campi static

---

- **All'interno della classe in cui sono definiti**
  - Stessa sintassi utilizzata per i campi non-static
- **Da una classe diversa da quella di definizione**
  - Utilizziamo il nome della classe invece del riferimento ad un oggetto:

```
double fee = BankAccount.OVERDRAFT_FEE;
```

```
Printstream ps = System.out;
```

# Domanda

---

- **L'istruzione `System.out.println(4)` denota una chiamata di metodo static?**

# Risposta

---

- **No: il metodo `println()` è invocato qui sull'oggetto `System.out`**

# Costanti: `final`

---

- Una variabile `final` è una costante
- Una volta inizializzata, non è possibile modificarne il valore
- Convenzione: i nomi delle variabili usano solo caratteri **MAIUSCOLI**

```
final double QUARTER = 0.25;
final double DIME = 0.1;
final double NICKEL = 0.05;
final double PENNY = 0.01;
```

# Costanti: `static final`

- Costanti utilizzate da più di un metodo possono essere dichiarate come campi, e qualificate come `static` e `final`
- Le costanti `static final` possono inoltre essere dichiarate `public` per renderle disponibili ad altre classi

```
public class Math
{
 . . .
 public static final double E = 2.7182818284590452354;
 public static final double PI = 3.14159265358979323846;
}

double circonferenza = Math.PI * diametro;
```



# La classe Pila - 1

```
class Pila {
 private int size;
 private int defaultGrowthSize;
 private int marker;
 private contenuto[];
 final int INITSIZE=3;

 public Pila() {
 size=initialSize;
 defaultGrowthSize=INITSIZE;
 marker=0;
 contenuto=new int[size];
 }
```

# La classe Pila - 2

```
private void cresci(int dim) {
 size+=dim;

 int temp[]=new int[size];
 for (int k=0;k<marker;k++)
 temp[k]=contenuto[k];

 contenuto=temp;
}
```

# La Classe Pila - 3

---

```
public void inserisci(int k) {
 if (marker==size) {
 cresci(defaultGrowthSize;)
 }
 contenuto[marker]=k;
 marker++;
}
```

# La Classe Pila - 4

```
public int estrai() {
 if (marker==0) {
 System.out.println(
 "Non posso estrarre da una pila vuota");
 System.exit(1);
 }
 return contenuto[--marker];
}
```

# Packages

---

- **Package: insieme di classi e interfacce in relazione**
- **Per formare un package basta inserire la direttiva**

```
package packageName;
```

**come prima istruzione nel file sorgente**

- **Una sola direttiva per file**
- **Classi contenute in file che non dichiarano packages vengono incluse in un package “anonimo”**
  - package anonimo OK solo per micro applicazioni, o in fase di sviluppo

*Continua...*

# Packages

| Package     | Finalità                 | Classe Tipica |
|-------------|--------------------------|---------------|
| java.lang   | Supporto al linguaggio   | Math, String  |
| java.util   | Utilities                | Random        |
| java.io     | Input e Output           | PrintStream   |
| Java.awt    | Abstract Window Toolkit  | Color         |
| Java.applet | Applets                  | Applet        |
| Java.net    | Networking               | Socket        |
| Java.sql    | Accesso a database       | ResultSet     |
| Java.swing  | Ingerfaccia utente Swing | JButton       |
| ...         | ...                      | ...           |

# Accesso agli elementi di un package

- Per accedere ai tipi di un package utilizziamo il nome “qualificato”

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

- Uso dei nomi qualificati verboso
- **import** permette sintesi

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

# Import

---

- di una classe

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

- di tutte le classi di un package

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

*Continua...*



# Import

- **Packages non formano gerarchie**

```
// import dei tipi di java.awt.color
import java.awt.color.*;
// import dei tipi di java.awt (non del package color!)
import java.awt.*; // import dei tipi di java.awt.
```

- **Static import**

- delle costanti e metodi statici dei tipi di un package

```
import static java.lang.Math.PI
import static java.lang.Math.*;
```

# Nomi di package

- Packages utili anche come “*namespaces*” per evitare conflitti di nomi (per classi/interfacce)
- Esempio, Java ha due classi **Timer**

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

- **Nomi di package devono essere univoci**
  - Convenzione: utilizziamo come prefissi domini internet, oppure indirizzi e-mail (in ordine inverso)

```
it.unive.dsi
it.unive.dsi.mp
```

*Continua...*

# Localizzazione di package

- Nomi di package devono essere consistenti con i path della directory che li contengono

```
it.unive.dsi.mp.banking
```

- Deve essere contenuto in un folder/directory localizzato nel path corrispondente

```
UNIX: <base directory>/it/unive/dsi/mp/banking
```

```
WINDOWS: <base directory>\it\unive\dsi\mp\banking
```

*Continua...*

# Localizzazione di package

- **CLASSPATH:** definisce le directory base dove localizzare i packages
- **Spesso utili due directory base**
  - per file sorgenti (.java)
  - per file compilati (.class)

**UNIX:**

```
export CLASSPATH=/home/mp/java/src:/home/mp/java/classes:.
```

**WINDOWS:**

```
set CLASSPATH=c:\home\mp\java\src;\home\mp\java\classes;.
```