



Realizzare Classi



Percorso formativo

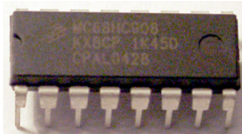
- Programmare in Java:
 - Definire classi
 - Istanziare oggetti
- Imparare ad usare oggetti e classi predefiniti
- Imparare a definire nuove classi



In questa Lezione

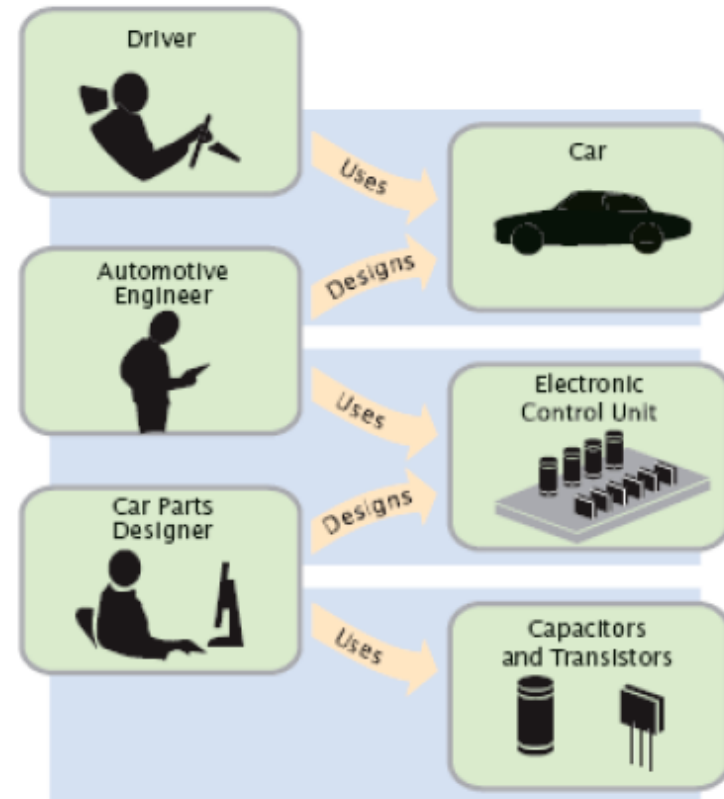
Astrazione

- Utile nella descrizione, progettazione, implementazione e utilizzo di sistemi complessi
- Dettagli trascurabili vengono *incapsulati* in sottosistemi più semplici che vengono quindi utilizzati come delle scatole nere
 - non occorre conoscere il loro funzionamento interno
 - basta conoscere l'essenza del concetto che rappresentano e l'interazione con il resto del sistema
- Ad esempio, un autista per usare un'auto non necessita di conoscerne i dettagli ingegneristici, deve solo sapere a cosa serve e ad interagire con essa



Un esempio di astrazione: Progettazione di automobili

- Automobile
 - trasmissione
 - centralina
 - ...



- L'astrazione apporta semplificazione e specializzazione
 - Migliora l'efficienza

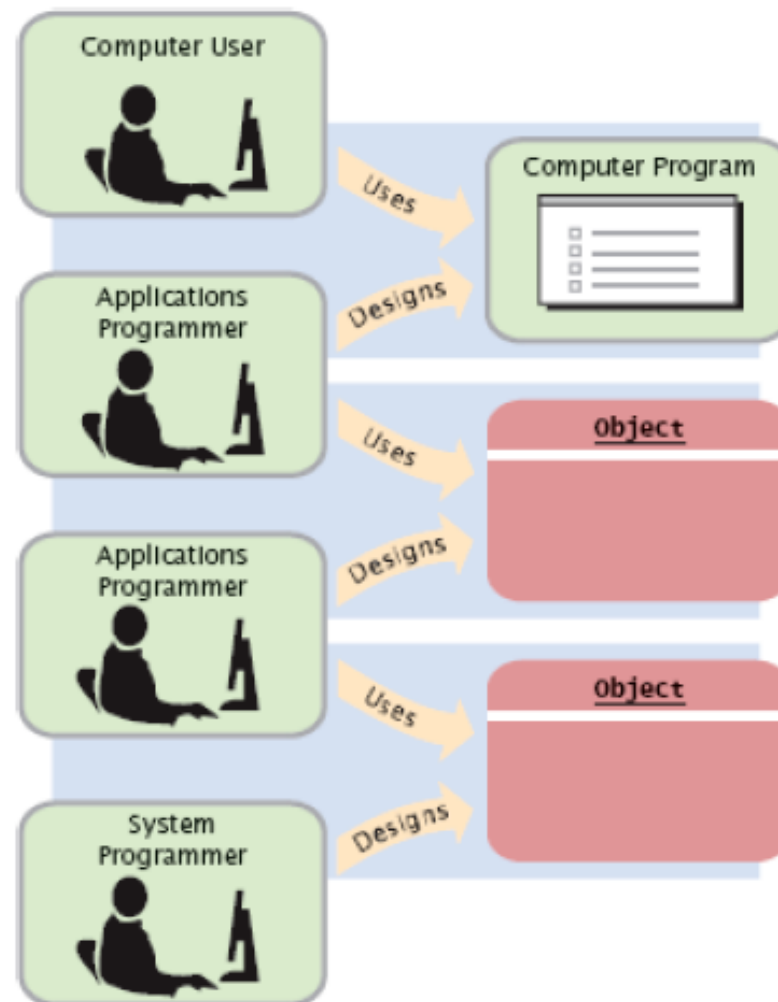


Astrazione in Software Design

- In precedenza: i programmi manipolavano tipi di dati primitivi come numeri e caratteri
- Per il programmatore risulta essere molto complicato manipolare molte quantità primitive (si possono facilmente introdurre errori)
- Soluzione: Incapsulare le computazioni di routine in scatole nere (software)
- L'astrazione è usata per inventare tipi di dati di più alto livello (**utilizzato nel processo di identificazione**)
 - Nella programmazione OO gli oggetti sono scatole nere
- Incapsulamento: i programmatori usano un oggetto conoscendo il suo comportamento, ma non la sua struttura interna (**utilizzato nell'implementazione**)

Un esempio di astrazione

- Porta la stessa visione nel software
 - Non solo tipi primitivi e routines
- Basta conoscere il comportamento esterno, non la struttura interna



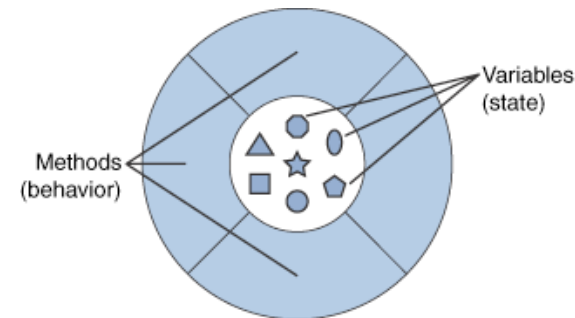


Programmazione orientata agli oggetti

- Gi oggetti forniscono la base per le astrazioni
 - Gli elementi che descrivono la soluzione ad un problema sono implementati da oggetti equivalenti
- *Incapsulamento*: un programmatore che usa un oggetto conosce il suo comportamento ma non necessita conoscere la sua struttura interna
- Definire buone astrazioni non è semplice
 - È sicuramente possibile progettare cattivi programmi OO
- In una corretta progettazione (ad oggetti) prima si definiscono le classi e poi si implementano

Classi

- Il comportamento di un oggetto è descritto da una *classe*



- Ogni classe ha
 - Un'interfaccia pubblica
 - Insieme di metodi (funzioni) che si possono invocare per manipolare l'oggetto
 - Es.: `Rectangle(x_init,y_init,width_init,height_init)` metodo dell'interfaccia che crea un rettangolo (`costruttore`)
 - Un'implementazione nascosta
 - codice e variabili usati per implementare i metodi dell'interfaccia e non accessibili all'esterno della classe
 - Es.: `x`, `y`, `width`, `height`



La definizione di una classe

```
public class NomeDellaClasse {  
    definizione di metodo  
    definizione di metodo  
    ...  
}
```

- Una classe contiene i suoi metodi (descrivono il comportamento comune alle istanze della classe)
- Le parentesi graffe aperte e chiuse fungono da delimitatori del contenuto di una classe

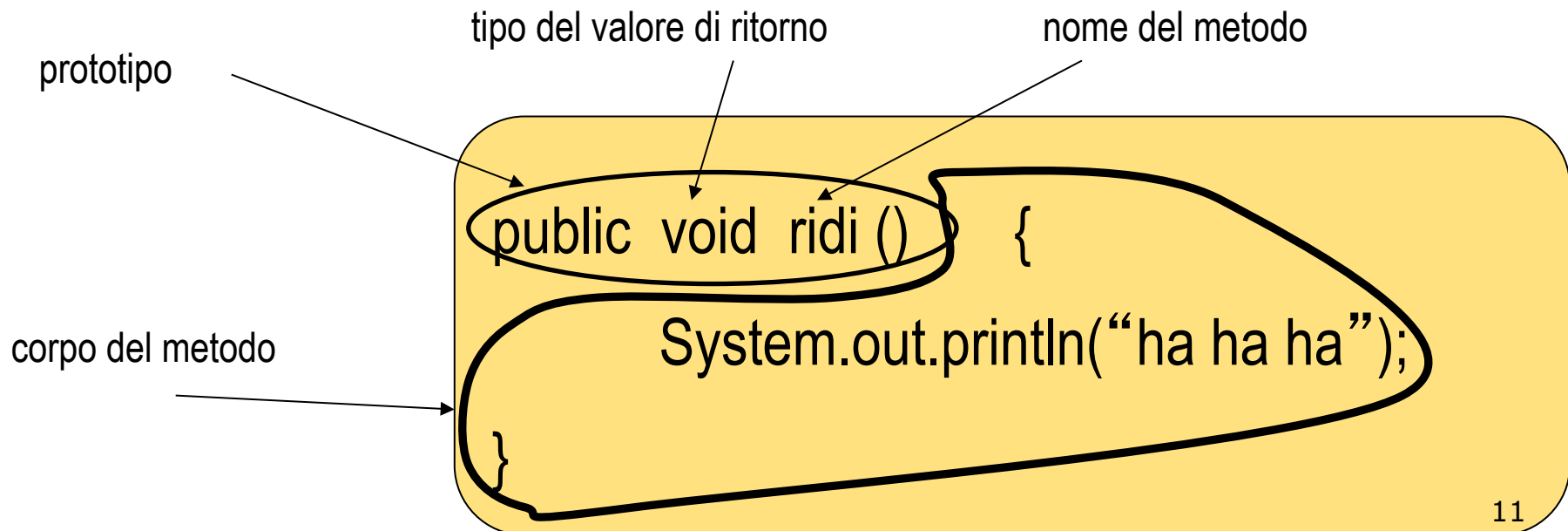


Un esempio

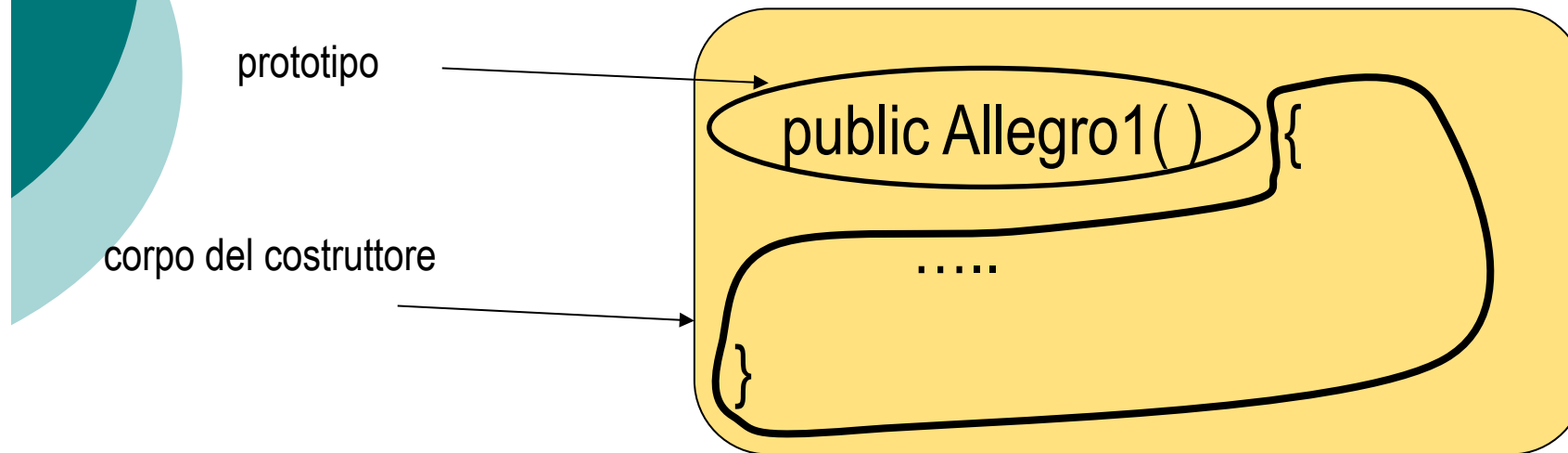
```
public class Allegro1 {  
    public Allegro1() {  
    }  
    public void ridi() {  
        System.out.println("haha");  
    }  
}
```

La definizione di un metodo

- Prototipo e corpo (**body**)
- Le parentesi graffe aperte e chiuse fungono da delimitatori del corpo del metodo



La definizione di un costruttore



- Nessun valore di restituzione
- Il nome coincide con quello della classe
- Invocato insieme a **new** per restituire un riferimento ad un oggetto appena creato



Utilizzare la classe **Allegro1**

1. Salvare la classe in un file (**Allegro1.java**)
2. Compilare la classe (**javac Allegro1.java**)
3. Scrivere un programma che utilizzi la classe **Allegro1**

```
public class UsaAllegro1 {  
    public static void main(String[] args) {  
        Allegro1 x;  
        x = new Allegro1();  
        x.ridi();  
        x.ridi();  
    }  
}
```



Metodi con argomenti

- Supponiamo che il mittente del messaggio **ridi** debba poter specificare la sillaba della risata
 - ha, ho, hee ...

```
Allegro2 x;  
x = new Allegro2();  
x.ridi("ho");  
x.ridi("hee");
```

Metodi con argomenti

dichiarazione del parametro

```
public void ridi(String sillaba) {
```

```
    System.out.print(sillaba);
```

```
    System.out.println(sillaba);
```

usi del parametro

- Il numero e tipo degli argomenti nel messaggio devono coincidere con quelli nel prototipo



Esempio con Overloading

```
class Allegro2 {  
    public Allegro2() {  
    }  
    public void ridi() {  
        System.out.println("haha");  
    }  
    public void ridi(String syl) {  
        System.out.print(syl);  
        System.out.println(syl);  
    }  
}
```

- Notare che il metodo **ridi** è overloaded



Oggetti con memoria

- Supponiamo di voler fornire al metodo costruttore un argomento String che rimpiazzì “ha” come sillaba di default per la risata
- `Allegro3 x;`
`x = new Allegro3 (“ho”);`
`x.ridi();` // hoho
`x.ridi (“hee”);` // heehee
`x.ridi();` // hoho



Oggetti con memoria

- Costruttore: `public Allegro3(String syl) ...`
- Problemi:
 - Gli argomenti di un metodo esistono solo durante l'esecuzione del metodo
 - Gli argomenti di un metodo sono visibili solo al metodo
- Come fare in modo che il metodo `ridi` possa conoscere il valore della `syl` ?
- Fornire agli oggetti la capacità di memorizzazione

Variabili di istanza (instance variables)



Variabili di istanza

- Una variabile dichiarata all'interno di una classe ma al di fuori di qualsiasi metodo
 - Accessibile a tutti i metodi dell'oggetto
 - Lo scopo è di memorizzare le informazioni necessarie ai metodi che devono essere preservate tra diverse invocazioni
 - Ciascun oggetto ha le proprie variabili di istanza le quali hanno i propri valori
 - **Stato di un oggetto**: i valori delle variabili di istanza
 - Tipicamente sono inizializzate dal costruttore

Esempio

```
public class Allegro3 {  
    public Allegro3(String syl) {  
        defaultSyl = syl;  
    }  
    public void ridi() {  
        System.out.print (defaultSyl);  
        System.out.println (defaultSyl);  
    }  
    public void ridi(String syl) {  
        System.out.print(syl);  
        System.out.println(syl);  
    }  
    private String defaultSyl;  
}
```

Le variabili di istanza
possono essere usate
da tutti i metodi

Dichiarazione di una
variabile di istanza



...miglioriamo ancora ...

```
public class Allegro4 {  
    public Allegro4() {  
        defaultSyl = "ha";  
    }  
    public Allegro4(String syl) {  
        defaultSyl = syl;  
    }  
    public void ridi() {  
        System.out.print (defaultSyl);  
        System.out.println(defaultSyl);  
    }  
    public void ridi(String syl) {  
        System.out.print(syl);  
        System.out.println(syl);  
    }  
    public void ridi(String s1, String s2) {  
        System.out.print(s1);  
        System.out.println(s2);  
    }  
    private String defaultSyl;  
}
```




... utilizzatore ...

```
public class RidiamoUnPoco {  
    public static void main(String[] a) {  
        System.out.println("Vivi allegramente!");  
        Allegro4 x,y,z;  
        x = new Allegro4("yuk");  
        y = new Allegro4("harr");  
        z = new Allegro4();  
        x.ridi();  
        x.ridi("hee");  
        y.ridi();  
        z.ridi();  
    }  
}
```



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()) ;
```



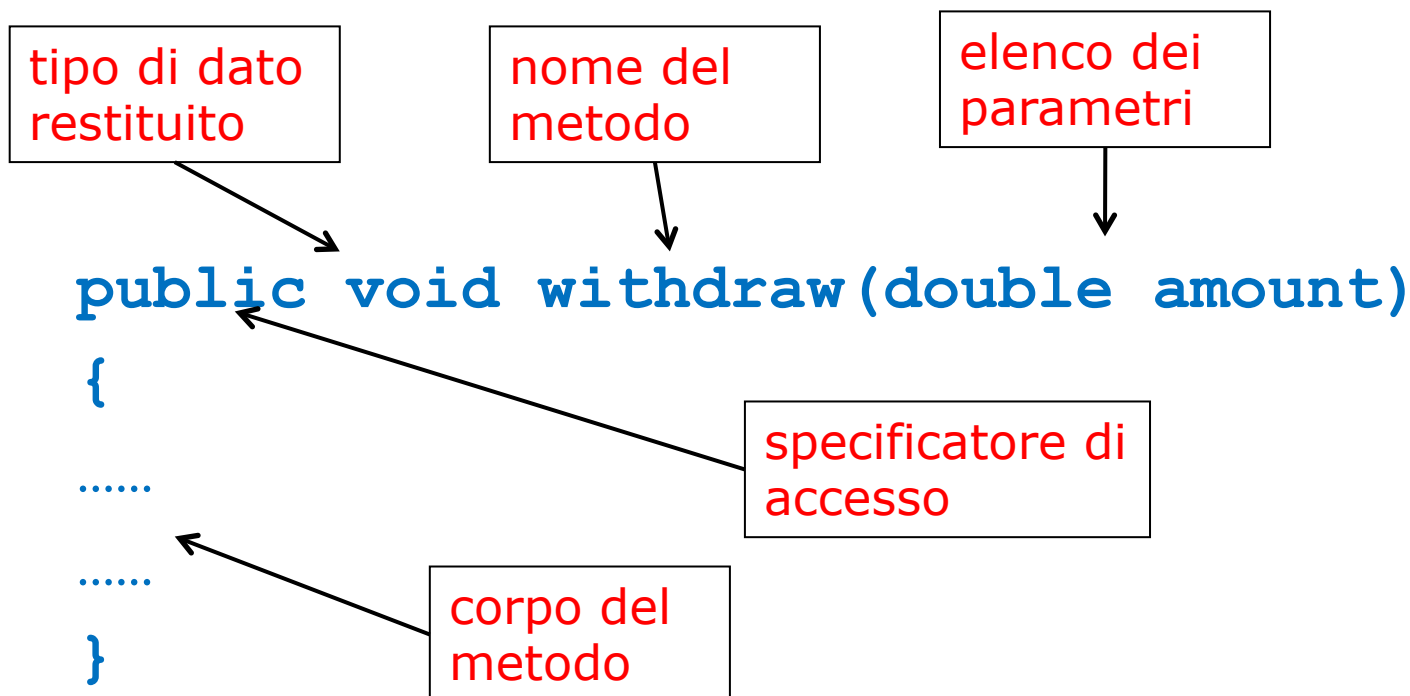
Definizione di un metodo

```
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](#)

[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

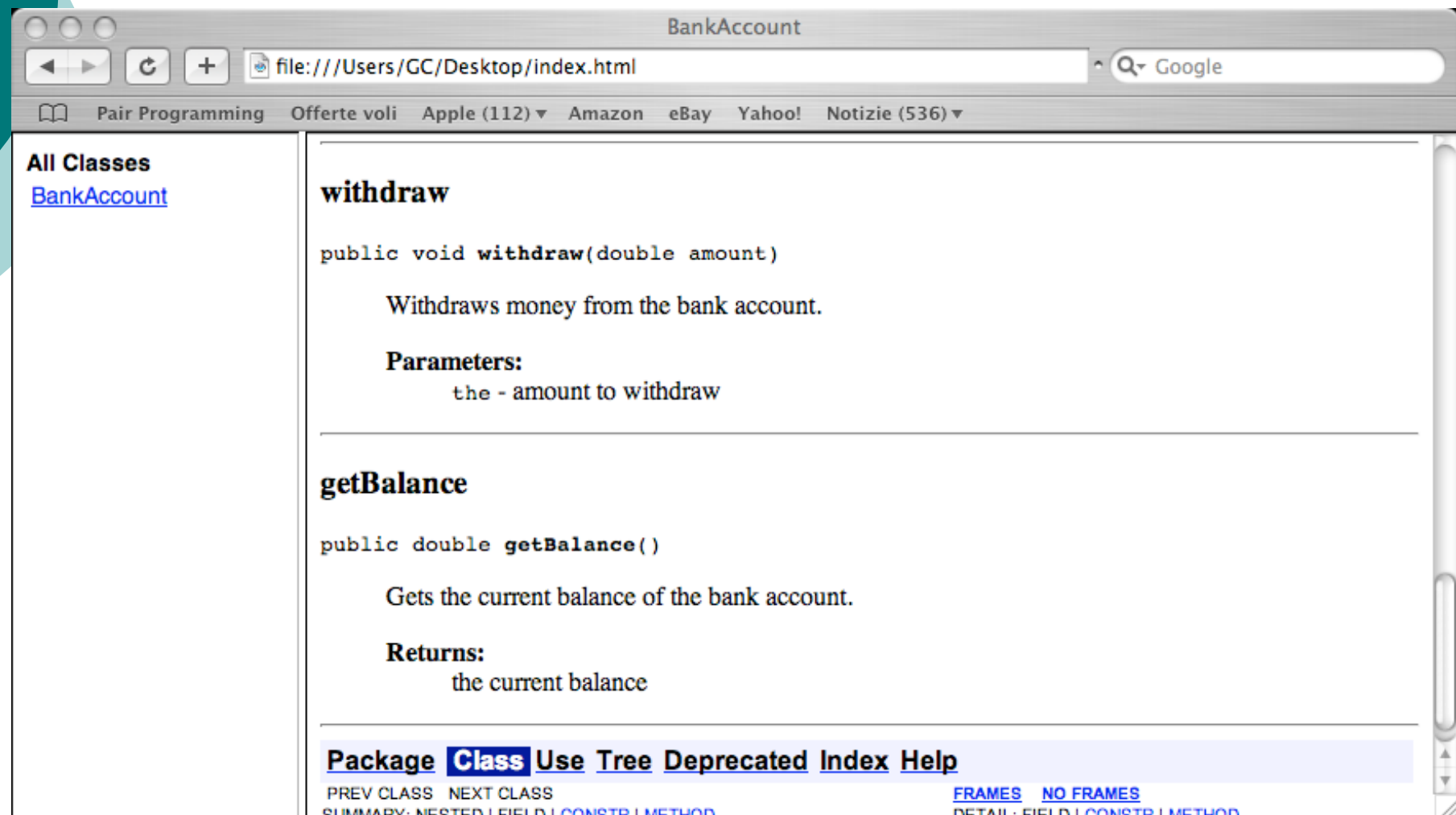
BankAccount ()
BankAccount (double initialBalance)

Method Summary

void	deposit (double amount)
double	getBalance () Gets the current balance of the bank account.
void	withdraw (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" and the search bar contains "Google". The browser's bookmark bar includes "Pair Programming", "Offerte voli", "Apple (112)", "Amazon", "eBay", "Yahoo!", and "Notizie (536)".

On the left side, under "All Classes", the link "[BankAccount](#)" is visible.

The main content area displays the documentation for the **withdraw** method:

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

Withdraws money from the bank account.

Parameters:
the - amount to withdraw

The **getBalance** method is also shown:

```
public double getBalance()
```

Gets the current balance of the bank account.

Returns:
the current balance

At the bottom, a navigation bar contains the following links: **Package**, **Class**, [Use](#), [Tree](#), [Deprecated](#), [Index](#), and [Help](#). Below this bar, there are additional links: "PREV CLASS", "NEXT CLASS", "FRAMES", "NO FRAMES", "SUMMARY: NESTED | FIELD | CONSTR | METHOD", and "DETAIL: FIELD | CONSTR | METHOD".

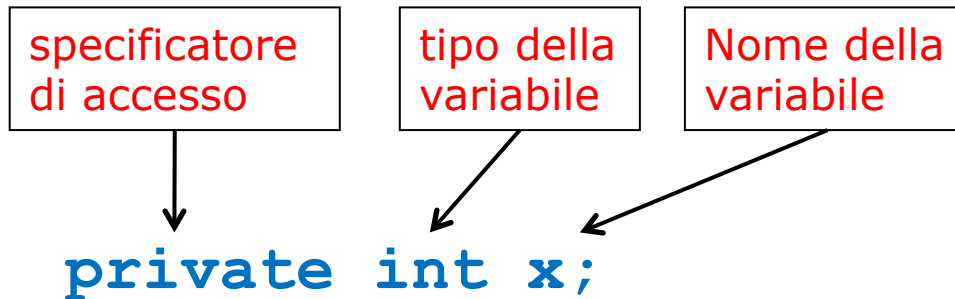


Variabili di istanza

- 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



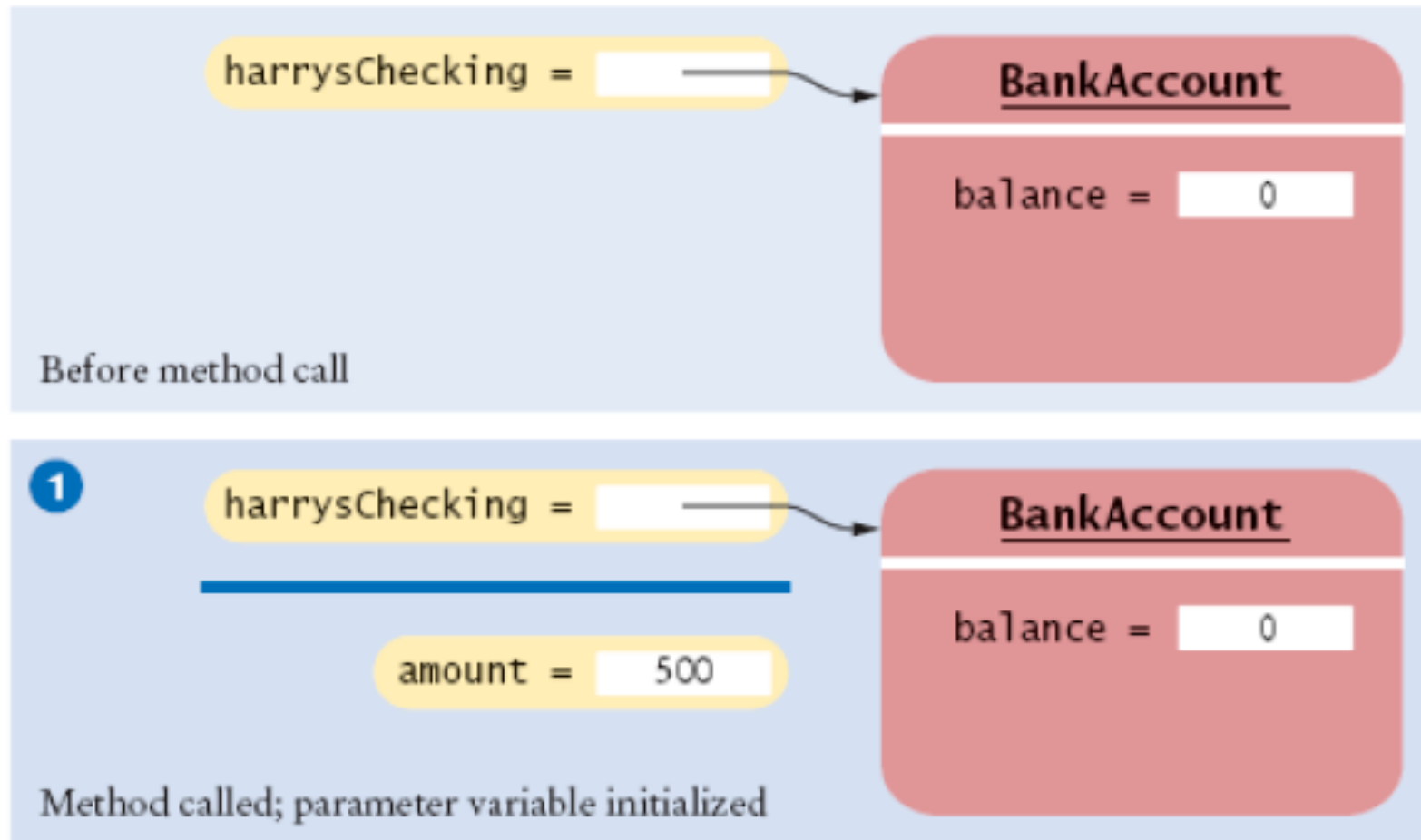
# Garbage collector

---

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

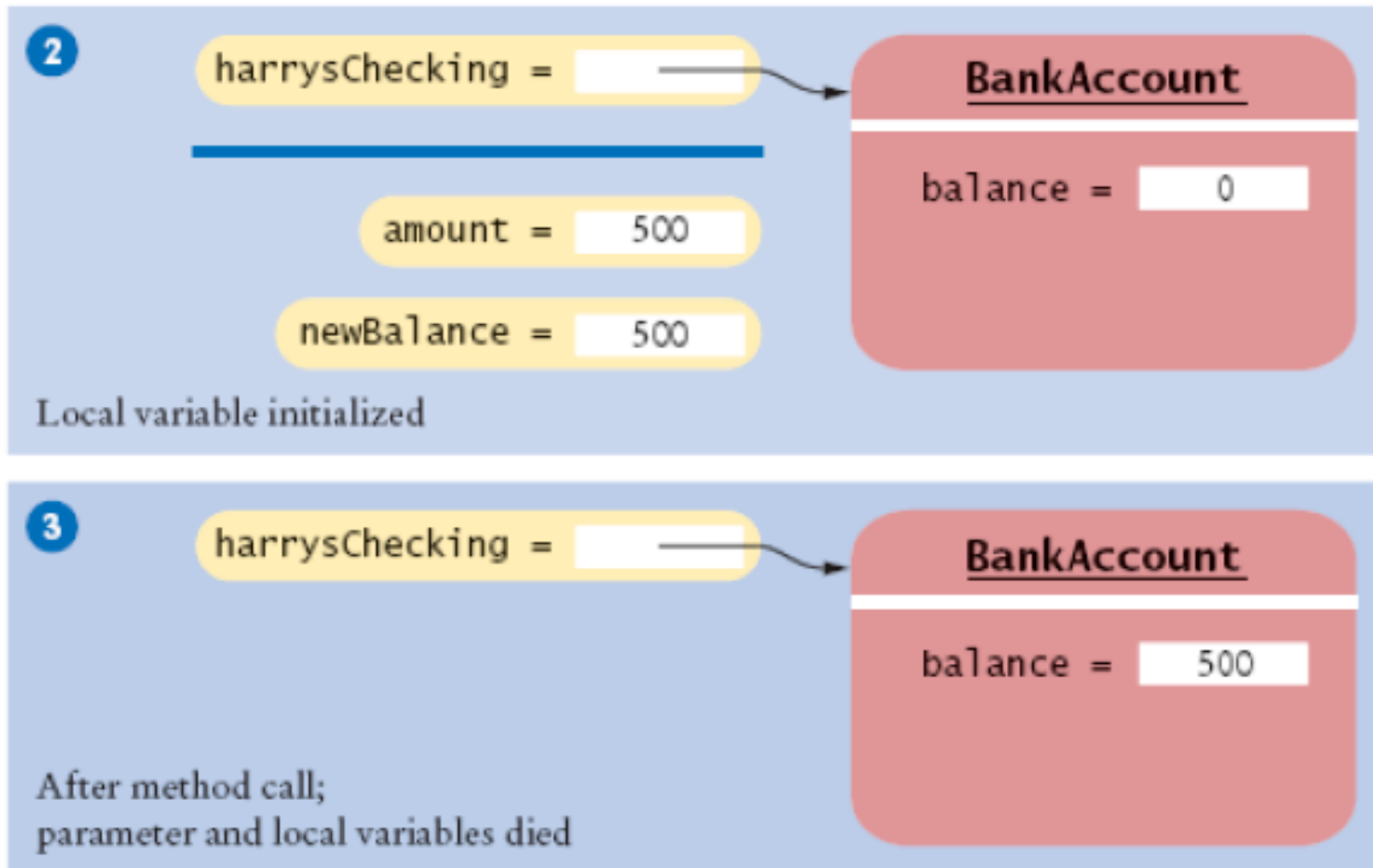
```
harrysChecking.deposit(500);
double newBalance = balance + amount;
balance = newBalance;
```

# Ciclo di vita delle variabili





# Ciclo di vita delle variabili

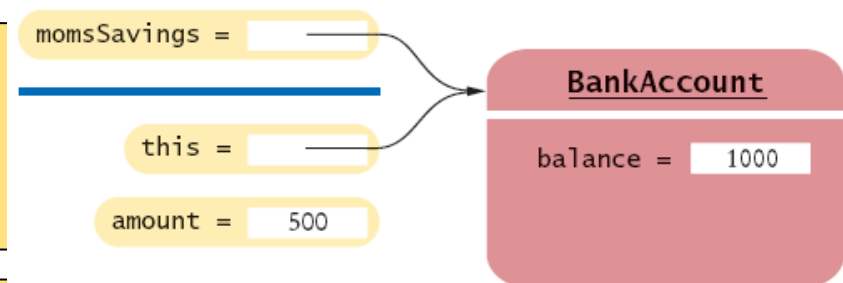


# Parametro espliciti ed impliciti

- 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)
```



## Esercizio

---

- Supponiamo di voler gestire i dati relativi ai modelli in vendita presso un concessionario d'auto. Per ogni modello occorre tener traccia della marca, del nome, della targa, della capacità del serbatoio e del numero dei chilometri che il modello è in grado di percorrere con un litro di carburante.
- Il titolare del concessionario potrebbe essere interessato a calcolare l'autonomia di ogni modello (in chilometri).
- Si definisca inoltre una classe TestAuto che permetta di creare 2 automobili e di calcolarne l'autonomia.