Capitolo 2

Utilizzare oggetti

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione

Oggetti e classi

- Gli oggetti sono entità di un programma che si possono manipolare invocando metodi.
- Tali oggetti appartengono a diverse classi. Per esempio l'oggetto *System.out* appartiene alla classe *PrintStream*.

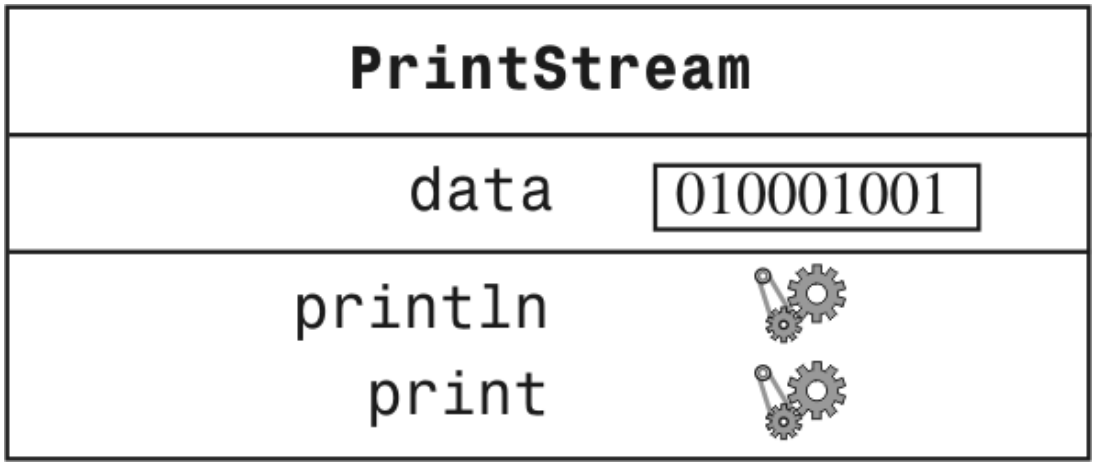

Figura 3:
Rappresentazione dell'oggetto `System.out`

Metodi

- Metodo: sequenza di istruzioni che accede ai dati di un oggetto
- Gli oggetti possono essere manipolati invocando metodi
- Classe: insieme di oggetti con lo stesso comportamento
- Una classe specifica i metodi che possono essere applicati ai suoi oggetti

```
String greeting = "Hello";
greeting.println() // Error
greeting.length() // OK
```

- L'interfaccia pubblica di una classe specifica *cosa* si può fare con i suoi oggetti mentre l'implementazione nascosta descrive *come* si svolgono tali azioni.

Rappresentazione di due oggetti di tipo String

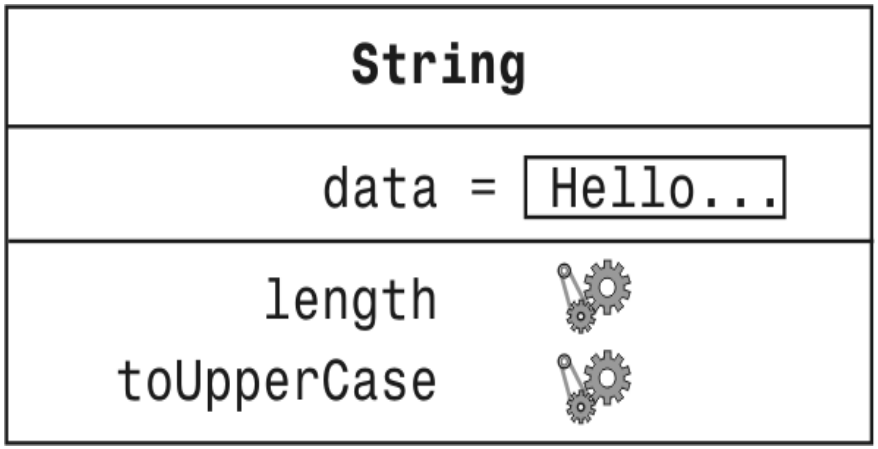

Figura 4
Rappresentazione di due oggetti di tipo `String`

Metodi String

- `length`: conta il numero di caratteri presenti in una stringa.

```
String greeting = "Hello, World!";
int n = greeting.length(); // assegna a n il numero 13
```

Continua…

Metodi String

- `toUpperCase`: crea un nuovo oggetto di tipo String che contiene gli stessi caratteri dell'oggetto originale, con le lettere minuscole convertite in maiuscole.

```
String river = "Mississippi";
String bigRiver = river.toUpperCase();
// assegna a bigRiver l'oggetto "MISSISSIPPI"
```

Continua…

Metodi String

- Quando applicate un metodo a un oggetto, dovete essere certi che il metodo sia definito nella classe corrispondente.

```
System.out.length(); // Questa invocazione di metodo è errata
```

Parametri impliciti ed espliciti

- Parametro (parametro esplicito): dati in ingresso a un metodo. Non tutti i metodi necessitano di parametri.

```
System.out.println(greeting)
greeting.length() // non ha parametri espliciti
```

- Parametro implicito: l'oggetto di cui si invoca un metodo

```
System.out.println(greeting)
```


Continua…

Parametri impliciti ed espliciti

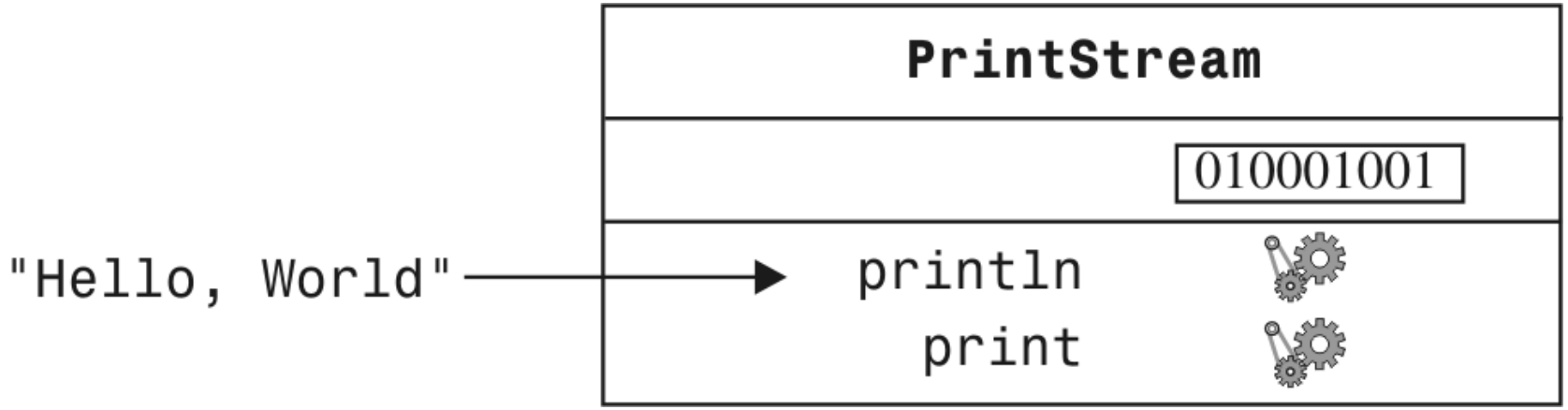

Figura 5
Passaggio di parametro al metodo `println`

Valori restituiti

- Il valore restituito da un metodo è il risultato che il metodo ha calcolato perché questo venga utilizzato nel codice che ha invocato il metodo

```
int n = greeting.length(); // restituisce il valore
                           // memorizzato in n
```

Continua

Valori restituiti

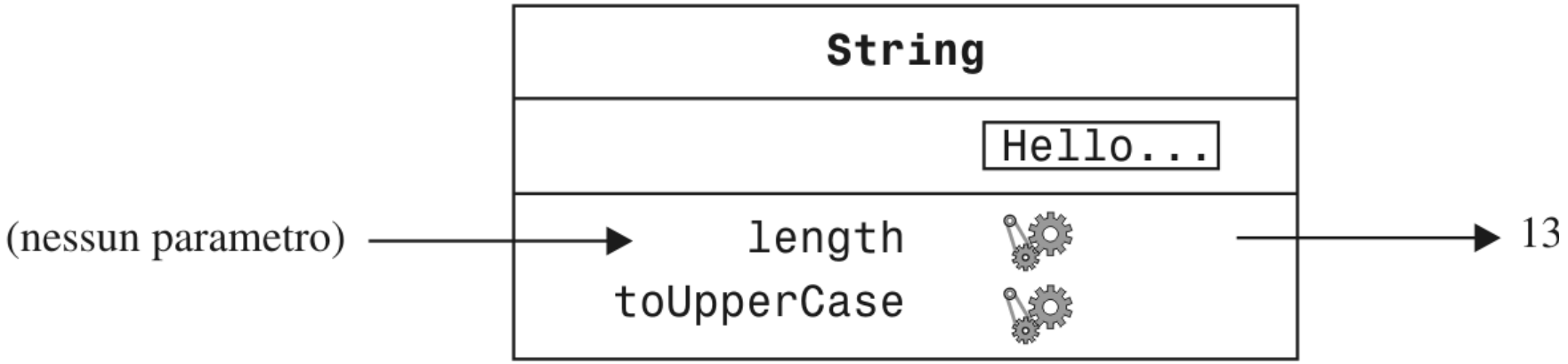

Figura 6 Invocazione del metodo `length` su un oggetto di tipo `String`

Utlizzo dei valori restituiti

- Il valore restituito da un metodo può anche essere utilizzato direttamente come parametro di un altro metodo

```
System.out.println(greeting.length());
```

- Non tutti i metodi restituiscono valori.
  Per esempio:

```
println
```

Continua…

Utilizzo dei valori restituiti

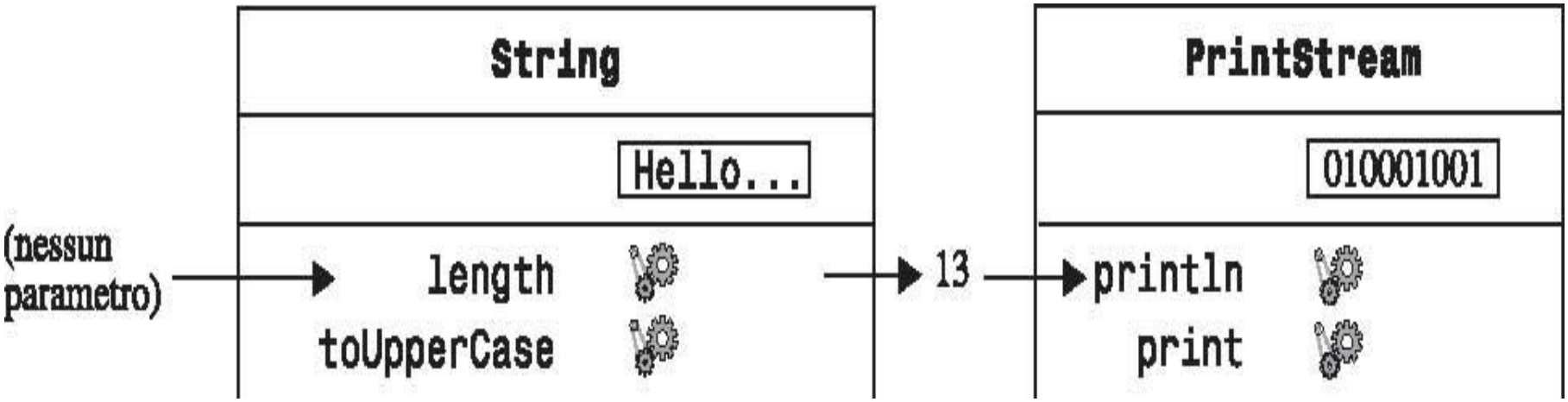

Figura 7
Il valore restituito da un metodo utilizzato come parametro di un altro metodo

Una invocazione più complessa

- Il metodo `replace` esegue metodi di ricerca e sostituzione

```
river.replace("issipp", "our")
  // costruisce una nuova stringa ("Missouri")
```

- Come si vede nella Figura 8, questa invocazione di metodo ha
  - un parametro implicito: la stringa "`Mississippi`"
  - due parametri espliciti: le stringhe "`issipp`" e "`our`"
  - un valore restituito: la stringa "`Missouri`"

Continua…

Una invocazione più complessa

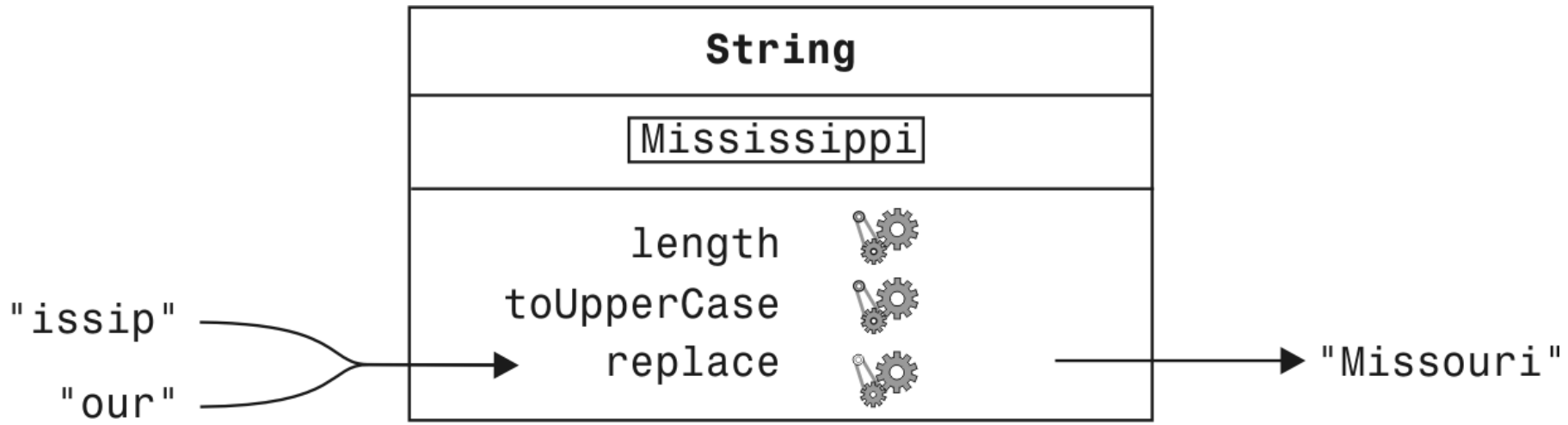

Figura 8 Invocazione del metodo `replace`

Definizioni di metodo

- **Quando in una classe si definisce un metodo, vengono specificati i tipi dei parametri espliciti e del valore restituito.**

- **Il tipo del parametro implicito è la classe in cui è definito il metodo: ciò non viene menzionato nella definizione del metodo, e proprio per questo si parla di parametro "implicito".**

Continua…

Definizioni di metodo

- Esempio: `la classe String` definisce

```
public int length()
   // restituisce un valore di tipo int
   // non ha parametri espliciti

public String replace(String target, String replacement)
   // restituisce un valore di tipo String;
   // due parametri espliciti di tipo String
```

Continua...

Definizioni di metodo

- Se il metodo non restituisce un valore, il tipo di valore restituito viene dichiarato come *void*

```
public void println(String output) // nella classe PrintStream
```

- Il nome di un metodo è sovraccarico se una classe definisce più metodi con lo stesso nome (ma con parametri di tipi diversi).

```
public void println(String output)
public void println(int output)
```

Variabili oggetto

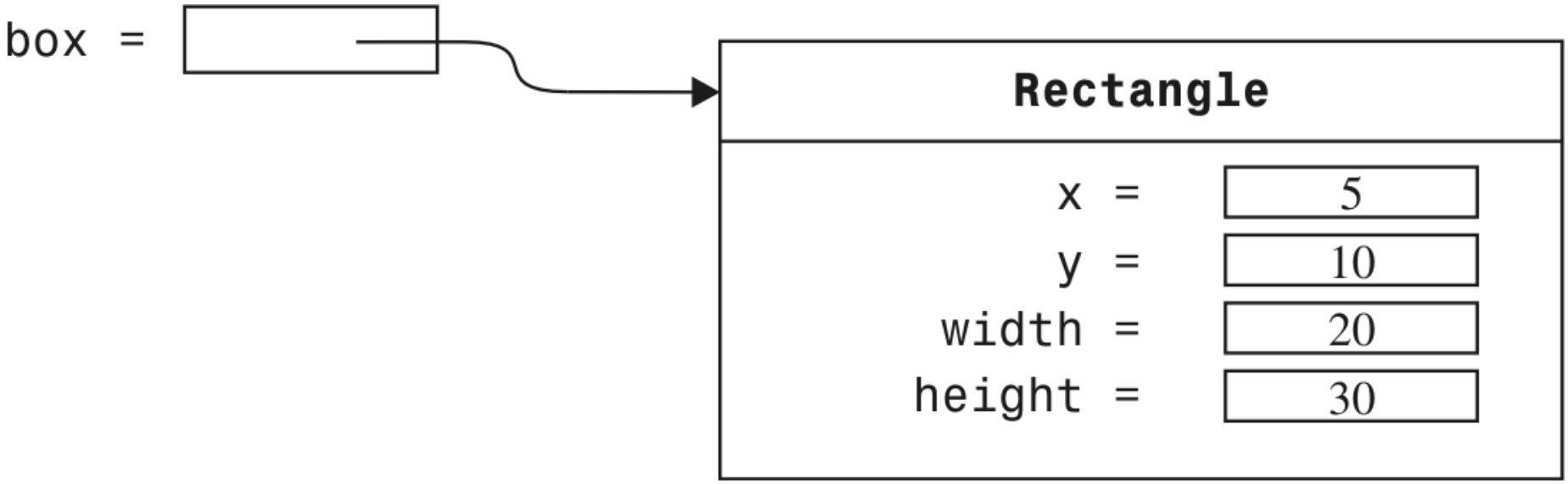

Figure 17
Una variabile oggetto contenente un riferimento a un oggetto

Variabili oggetto

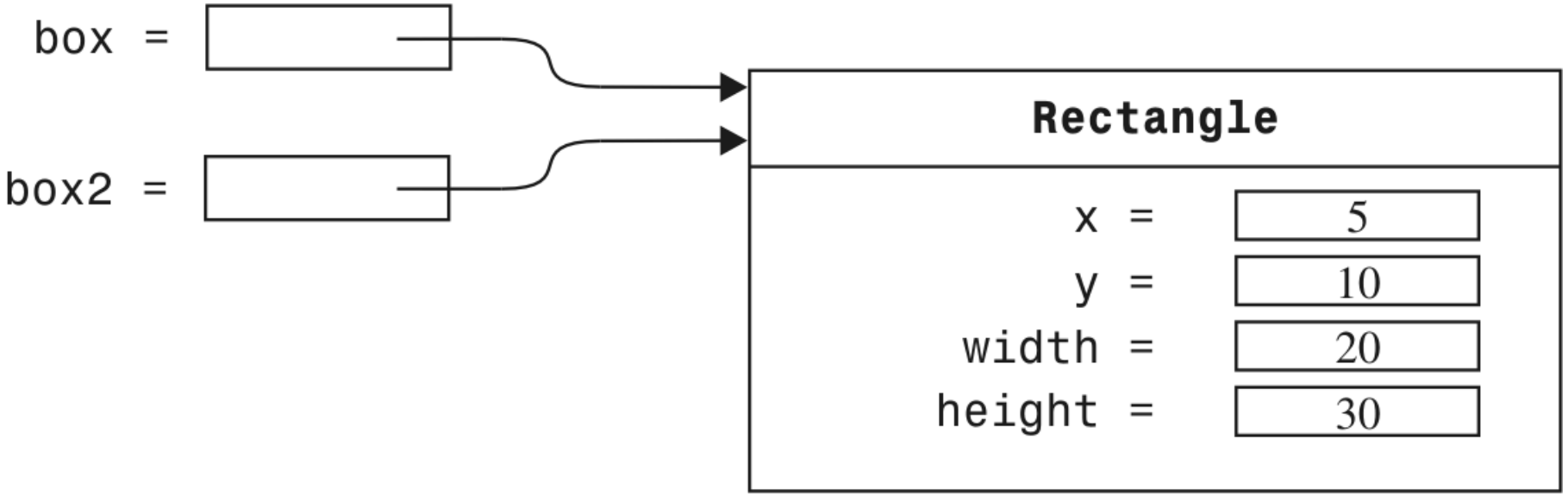

Figura 18
Due variabili oggetto che fanno riferimento al medesimo oggetto

Copiatura di riferimenti a oggetti

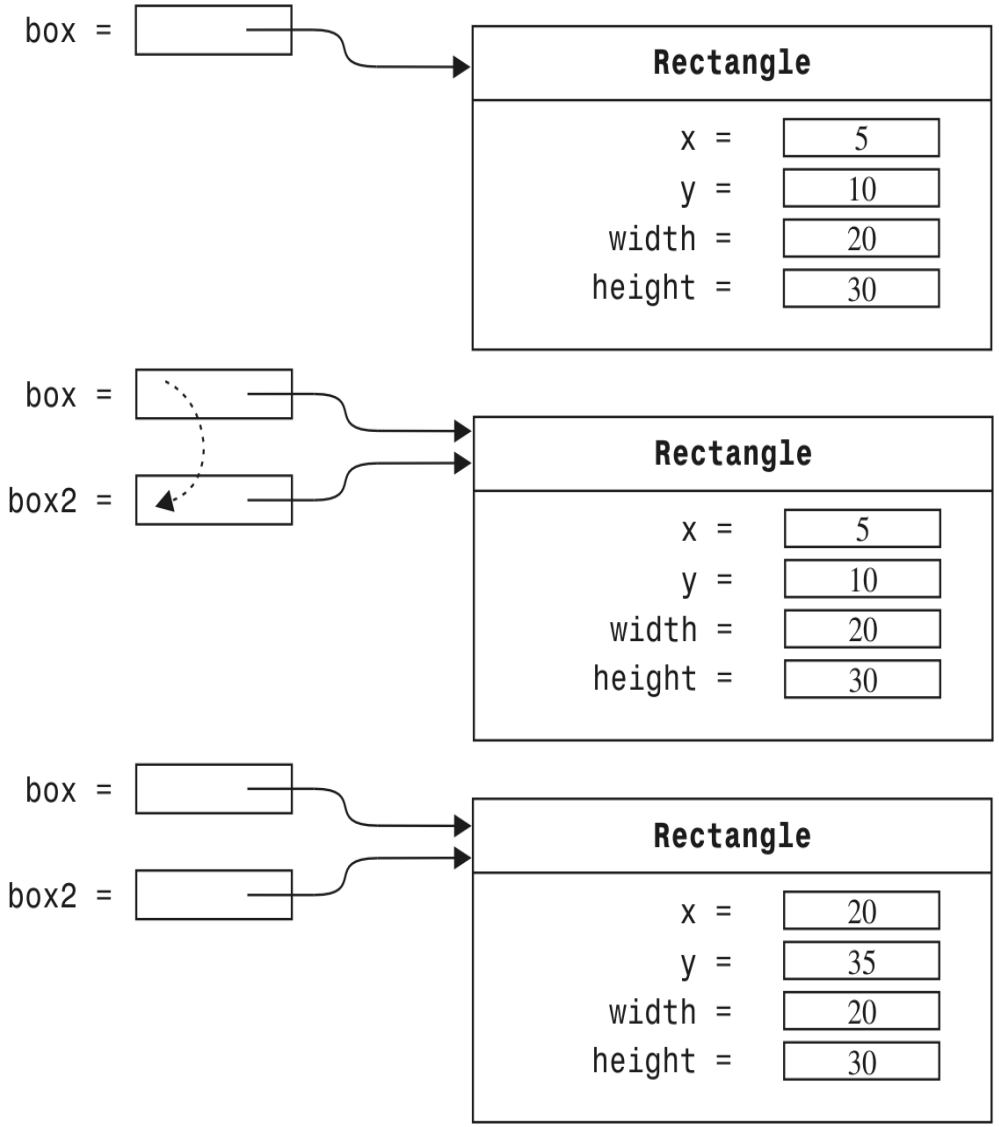

Figura 21
Copiatura di riferimenti a oggetti

Capitolo 3

Realizzare classi

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione

Scatole nere

- I concetti vengono identificati durante il processo di astrazione.
- Astrazione: processo di eliminazione delle caratteristiche inessenziali, finché non rimanga soltanto l'essenza del concetto.
- Nella programmazione *orientata agli oggetti* ("object-oriented") le scatole nere con cui vengono costruiti i programmi vengono chiamate oggetti.

Livelli di astrazione: un esempio dalla vita reale

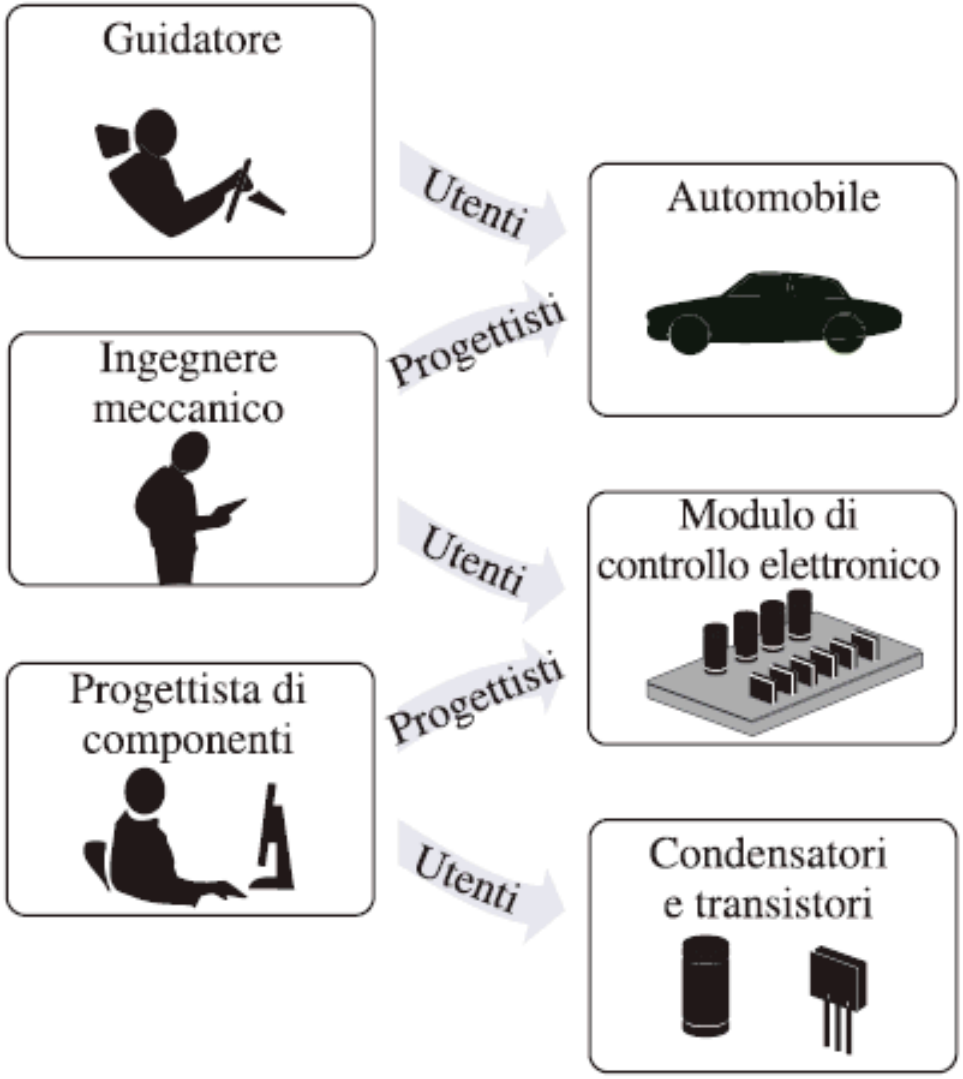

Figura 1
Livelli di astrazione nella progettazione di automobili

Livelli di astrazione: un esempio concreto

- I guidatori non hanno bisogno di capire come funzionano le scatole nere
- L'interazione di una scatola nera con il mondo esterno è ben definita:
  - I guidatori interagiscono con l'auto usando pedali, pulsanti,ecc.
  - I meccanici controllano che i moduli di controllo elettronico mandino il giusto segnale d'accensione alle candele
  - Per i produttori di moduli di controllo elettronico i transistori e i condensatori sono scatole nere magicamente costruite da un produttore di componenti elettronici
- L'incapsulamento porta all'efficienza:
  - Un meccanico si occupa solo di moduli per il controllo elettronico senza preoccuparsi di sensori e transistori
  - I guidatori si preoccupano solo di interagire con l'auto (mettere carburante nel serbatoio) e non dei moduli di controllo elettronico

Livelli di astrazione: progettazione del software

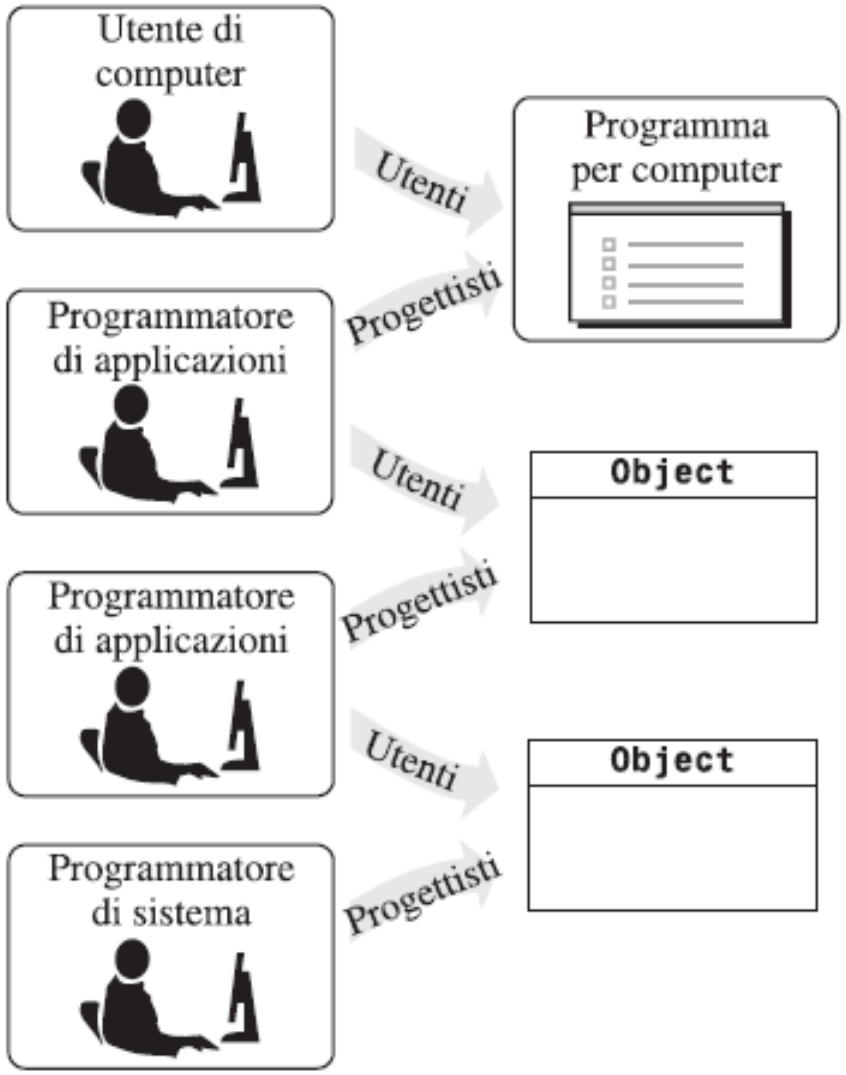

Figura 2
Livelli di astrazione nella progettazione del software

Livelli di astrazione: progettazione del software

- Ai primordi dell'informatica, i programmi per computer erano in grado di manipolare tipi di dati primitivi, come i numeri e i caratteri.
- Con programmi sempre più complessi, ci si trovò a dover manipolare quantità sempre più ingenti di questi dati di tipo primitivo, finché i programmatori non riuscirono più a gestirli.
- Soluzione: incapsularono le elaborazioni più frequenti, generando "scatole nere" software da poter utilizzare senza occuparsi di ciò che avviene all'interno,

Continua…

Livelli di astrazione: progettazione del software

- Fu usato il processo di astrazione per inventare tipi di dati a un livello superiore rispetto a numeri e caratteri.
- Nella programmazione orientata agli oggetti gli oggetti sono scatole nere.
- Incapsulamento: la struttura interna di un oggetto è nascosta al programmatore, che ne conosce però il comportamento.
- Nella progettazione del software si utilizza il processo di astrazione per definire il comportamento di oggetti non ancora esistenti e, dopo averne definito il comportamento, possono essere *realizzati* (o “implementati”)

Progettare l'interfaccia pubblica di una classe

- Progettare la classe `BankAccount`
- Procedimento di astrazione:
  - operazioni irrinunciabili per un conto bancario

Progettare l'interfaccia pubblica di una classe: i metodi

- Metodi della classe `BankAccount`:

```
deposit
withdraw
getBalance
```

- Vogliamo che i metodi possano funzionare nel modo seguente:

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

Progettare l'interfaccia pubblica di una classe: definizione di metodo

Ogni definizione di metodo contiene

- uno *specificatore di accesso* (solitamente `public`)
- il *tipo di dati restituito* (come `void` or `double`)
- il nome del metodo (come `deposit`)
- un elenco dei *parametri* del metodo, racchiusi fra parentesi (come `double amount`)
- il *corpo* del metodo: enunciati racchiusi fra parentesi graffe `{ }`

Esempi:

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

Sintassi 3.1: Definizione di metodo

```
specificatoreDiAccesso tipoRestituito nomeMetodo(tipoParametro
nomeParametro, ...)
{
   corpo del metodo
}
```

Esempio:

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

Serve a:

Definire il comportamento di un metodo.

Progettare l'interfaccia pubblica di una classe: definizione di costruttore

- I costruttori contengono istruzioni per inizializzare gli oggetti.
- Il nome di un costruttore è sempre uguale al nome della classe.

```
public BankAccount()
{
   // corpo, che verrà riempito più tardi
}
```

Continua…

Progettare l'interfaccia pubblica di una classe: definizione di costruttore

- Il corpo del costruttore è una sequenza di enunciati che viene eseguita quando viene costruito un nuovo oggetto.
- Gli enunciati presenti nel corpo del costruttore imposteranno i valori dei dati interni dell'oggetto che è in fase di costruzione.
- Tutti i costruttori di una classe hanno lo stesso nome, che è il nome della classe.
- Il compilatore è in grado di distinguere i costruttori, perché richiedono parametri diversi.

Sintassi 3.2: Definizione di costruttore

```
specificatoreDiAccesso nomeClasse(tipoParametro nomeParametro,
                                                              ...)
{
    corpo del costruttore
}
```

Esempio:

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

Serve a:

Definire il comportamento di un costruttore

Interfaccia pubblica

- I costruttori e i metodi pubblici di una classe costituiscono la sua interfaccia pubblica:

```
public class BankAccount
{
   // Costruttori
   public BankAccount()
   {
      // corpo, che verrà riempito più avanti
   }
   public BankAccount(double initialBalance)
   {
      // corpo, che verrà riempito più avanti
   }

   // Metodi
   public void deposit(double amount)
```

Continua

Interfaccia pubblica

```
   {
      // corpo, che verrà riempito più avanti
   }

   public void withdraw(double amount)
   {
      // corpo, che verrà riempito più avanti
   }
   public double getBalance()
   {
      // corpo, che verrà riempito più avanti
   }
   // campi privati, definiti più avanti
}
```

Sintassi 3.3: Definizione di classe

```
specificatoreDiAccesso class nomeClasse
{
    costruttori
    metodi
    campi
 }
```

Esempio:

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

Serve a:

Definire una classe, la sua interfacia pubblica e i suoi dettagli realizzativi.

Commentare l'interfaccia pubblica

```
/**
   Preleva denaro dal conto bancario.
   @param amount l'importo da prelevare
*/
public void withdraw(double amount)
{
   // realizzazione (completata in seguito)
}
```

```
/**
   Ispeziona il saldo attuale del conto corrente.
   @return il saldo attuale
*/
public double getBalance()
{
   // realizzazione (completata in seguito)
}
```

Riassunto dei metodi generato da javadoc

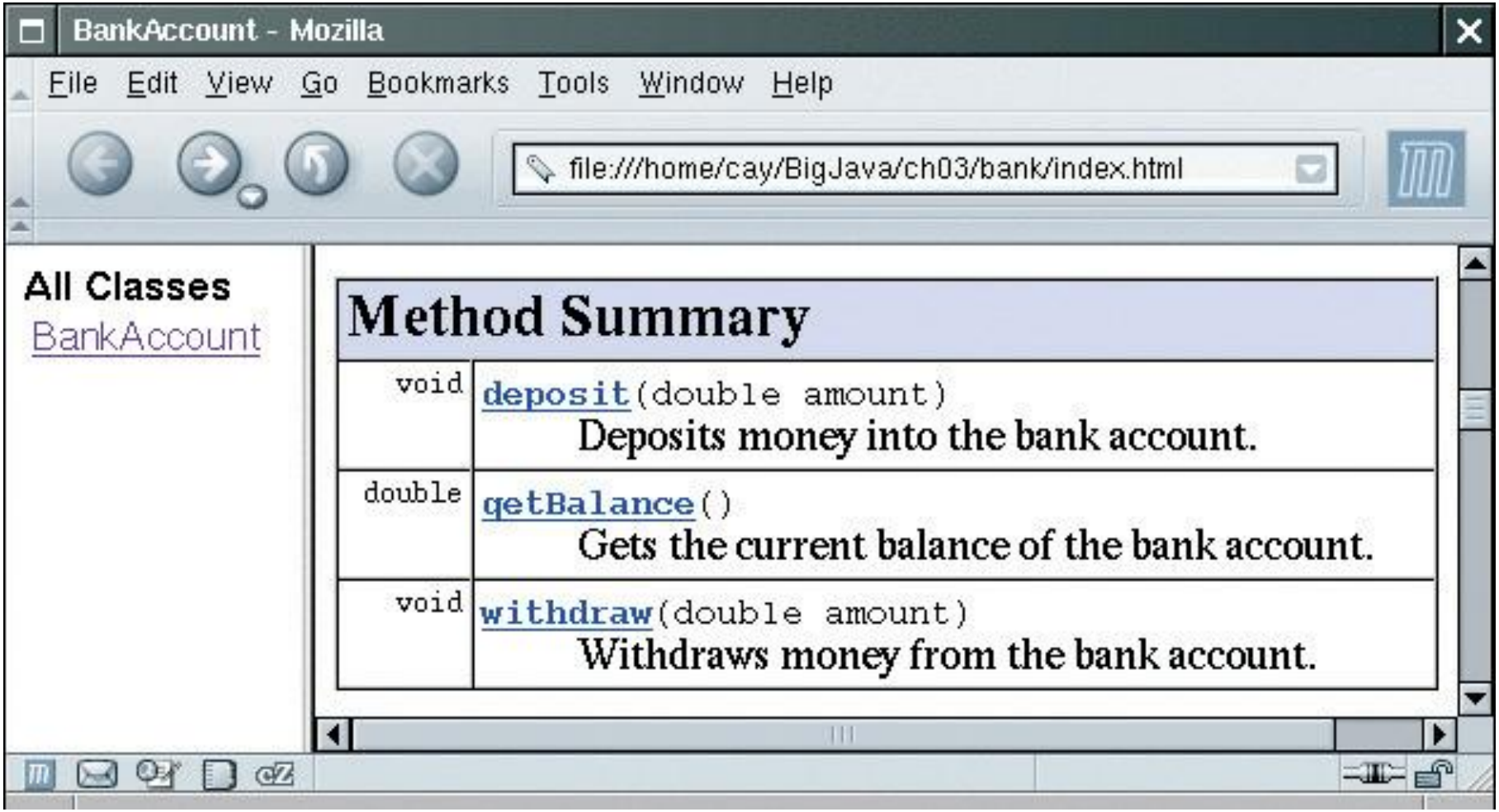

Figura 3 Un riassunto dei metodi generato da `javadoc`

Campi di esemplare

- Un oggetto memorizza i propri dati all'interno di *campi* (o *variabili*) *di esemplare* (o *di istanza*)
- *Campo* è un termine tecnico che identifica una posizione all'interno di un blocco di memoria
- Un *esemplare* (o *istanza*) di una classe è un oggetto creato da quella classe.
- La dichiarazione della classe specifica i suoi campi di esemplare:

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

Campi di esemplare

- La dichiarazione di un campo di esemplare è così composta:
  - Uno *specificatore d'accesso* (solitamente `private`)
  - Il *tipo* del campo di esemplare (come `double`)
  - Il nome del campo di esemplare (come `balance`)
- Ciascun oggetto di una classe ha il proprio insieme di campi di esemplare.
- I campi di esemplare sono generalmente dichiarati con lo specificatore di accesso `private`

Campi di esemplare

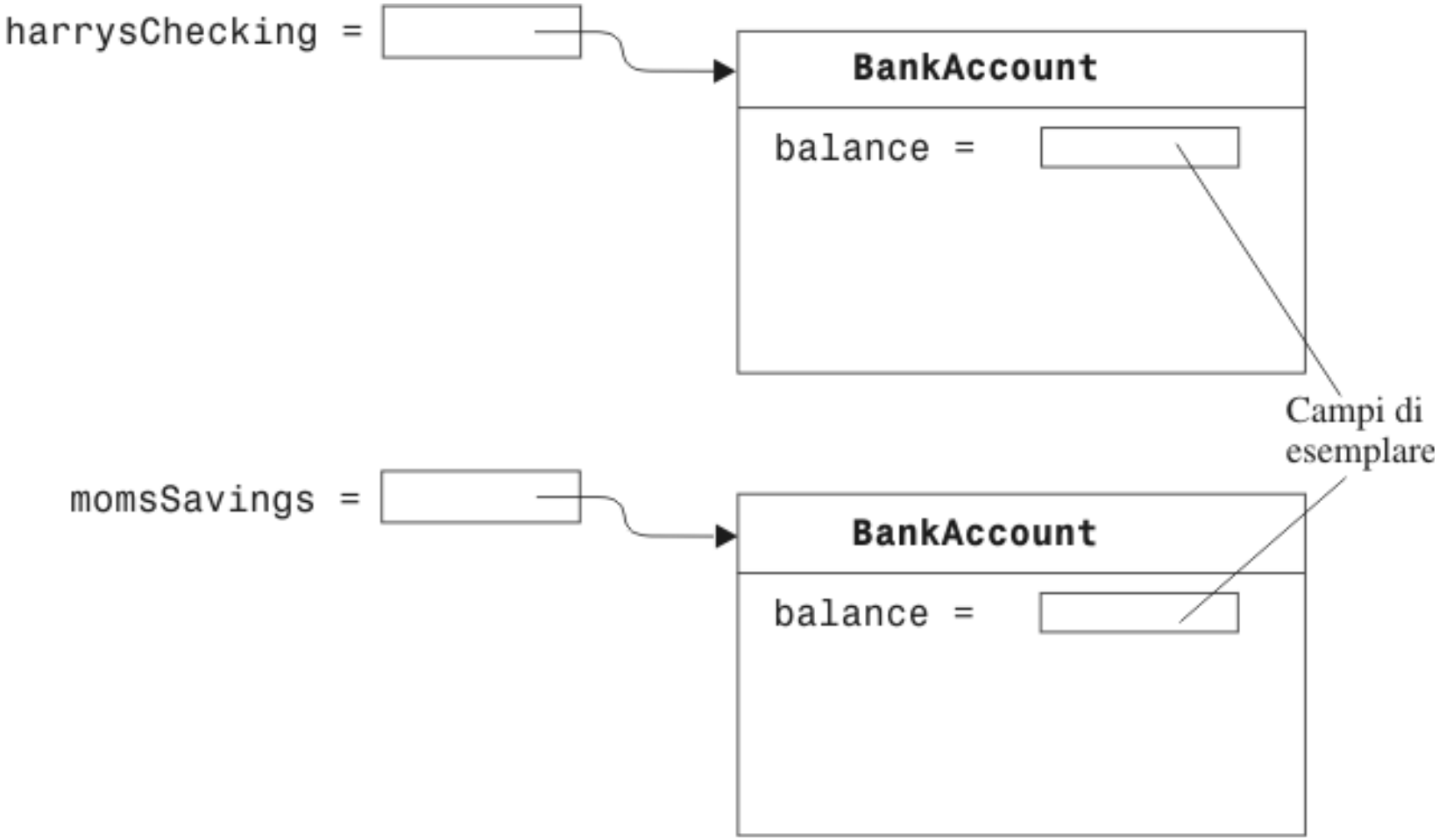

Figura 5
Campi di esemplare

Sintassi 3.4:
Dichiarazione di campo di esemplare

```
specificatoreDiAccesso class NomeClasse
{
   . . .
   specificatoreDiAccesso tipoVariabile nomeVariabile;
   . . .
}
```

Esempio:

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

Serve a:

Definire un campo che sia presente in ciascun oggetto di una classe.

Accedere ai campi di esemplare

- I campi di esemplare sono generalmente dichiarati con lo specificatore di accesso `private`: a essi si può accedere soltanto da metodi della medesima classe e da nessun altro metodo.
- Se i campi di esemplare vengono dichiarati privati, ogni accesso ai dati deve avvenire tramite metodi pubblici.
- L'incapsulamento prevede l'occultamento dei dati degli oggetti, fornendo metodi per accedervi.

Continua

Accedere ai campi di esemplare

- Ad esempio, alla variabile `balance` si può accedere dal metodo `deposit` della classe `BankAccount`, ma non dal metodo `main` di un'altra classe.

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

Realizzare i costruttori

- Un costruttore assegna un valore iniziale ai campi di esemplare di un oggetto.

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

Esempio di invocazione di un costruttore

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

- Creazione di un nuovo oggetto di tipo `BankAccount`.
- Invocazione del secondo costruttore (perché è stato fornito un parametro di costruzione).
- Assegnazione del valore 1000 alla variabile parametro `initialBalance`.
- Assegnazione del valore di initialBalance al campo di esemplare `balance` dell'oggetto appena creato.
- Restituzione, come valore dell'espressione `new`, di un riferimento a un oggetto, che è la posizione in memoria dell'oggetto appena creato.
- Memorizzazione del riferimento all'oggetto nella variabile `harrysChecking`.

Esempio di una invocazione di metodo

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

- Assegnazione del valore 500 alla variabile parametro `amount`.
- Lettura del campo balance dell'oggetto che si trova nella posizione memorizzata nella variabile `harrysChecking`.
- Addizione tra il valore di `amount` e il valore di `balance`, memorizzando il risultato nella variabile `newBalance`.
- Memorizzazione del valore di `newBalance` nel campo di esemplare `balance`, sovrascrivendo il vecchio valore.

File BankAccount.java

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

Continua

File BankAccount.java

```
19:    public BankAccount(double initialBalance)
20:    {
21:       balance = initialBalance;
22:    }
23:
24:    /**
25:       Versa denaro nel conto bancario.
26:       @param amount l'importo da versare
27:    */
28:    public void deposit(double amount)
29:    {
30:       double newBalance = balance + amount;
31:       balance = newBalance;
32:    }
33:
34:    /**
35:       Preleva denaro dal conto bancario.
36:       @param amount l'importo da versare
```

Continua

File BankAccount.java

```
37:     */
38:     public void withdraw(double amount)
39:     {
40:        double newBalance = balance - amount;
41:        balance = newBalance;
42:     }
43:
44:     /**
45:        Ispeziona il valore del saldo attuale del conto bancario.
46:        @return il saldo attuale
47:     */
48:     public double getBalance()
49:     {
50:        return balance;
51:     }
52:
53:     private double balance;
54: }
```

File BankAccountTester.java

```
01: /**
02:    Classe di collaudo per la classe BankAccount.
03: */
04: public class BankAccountTester
05: {
06:    /**
07:       Collauda 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.deposit(2000);
14:       harrysChecking.withdraw(500);
15:       System.out.println(harrysChecking.getBalance());
16:    }
17: }
```

Capitolo 8

Progettazione di classi

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione