

Definizione di classi

Unità didattica 3
Armando Tacchella
Fondamenti di Informatica

Introduzione alla definizione di classi

- ◆ Per utilizzare oggetti all'interno di un programma Java è necessario definire il loro contenuto informativo (stato) e le loro capacità di interazione (metodi)
- ◆ A tal scopo utilizziamo in Java la definizione di **classe**, ossia un **insieme omogeneo di oggetti** caratterizzati dallo stesso contenuto informativo e dalle stesse capacità di interazione
- ◆ Vi sono due tipi di classi in Java:
 - classi **istanziabili** (ad es. `MainWindow`, `InputDialog`, e `OutputBox`) ossia classi di cui possono essere creati oggetti
 - classi **non istanziabili** (ad es. `Math` e `PrimoProgramma`) ossia classi di cui non possono essere creati oggetti e i cui metodi sono utilizzati facendo riferimento direttamente alla classe (ad es. `Math.sin()`)

Esempio: ContoCorrente

- ◆ Un esempio per coprire gli elementi basilari della definizione di classi istanziabili
- ◆ Nel progettare una classe istanziabile, si comincia dalla **specificità** della classe, ossia dalla descrizione del comportamento desiderato per gli oggetti della classe
- ◆ Un oggetto **ContoCorrente** deve tenere traccia del saldo disponibile e consentire di eseguire operazioni di versamento e prelievo
- ◆ Immaginiamo di aver già definito la classe `ContoCorrente` e di utilizzare un oggetto di tale classe...

Esempio: utilizzo di ContoCorrente (1)

- ◆ Almeno due modalità di interazione (metodi) sembrano necessarie: **deposito** e **prelievo** di somme in euro

```
ContoCorrente mioConto;  
mioConto = new ContoCorrente();  
  
mioConto.deposita(250.0);  
  
double prelievo =  
mioConto.prelieva(100.50);
```

Dichiarazione e creazione dell'oggetto mioConto

Metodo per depositare somme di denaro

Metodo per prelevare somme di denaro

Esempio: utilizzo di ContoCorrente (2)

- ◆ In aggiunta a deposito e prelievo, desideriamo un metodo che ci consenta di ottenere il **saldo**

```
ContoCorrente mioConto;  
mioConto = new ContoCorrente();  
  
mioConto.deposita(250.0);  
double prelievo =  
mioConto.prelieva(100.50);  
  
double saldo = mioConto.ottieniSaldo();  
  
saldo contiene 149.50
```

Dichiarazione e creazione dell'oggetto mioConto

Operazioni sul conto

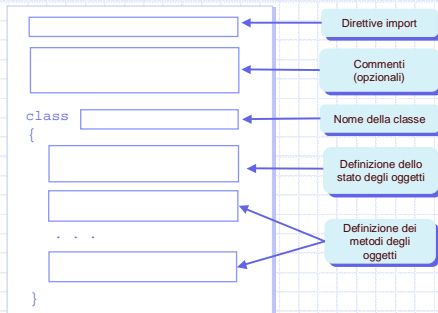
Esempio: utilizzo di ContoCorrente (3)

- ◆ Una volta definita la classe `ContoCorrente` è possibile utilizzarne diverse istanze (diversi oggetti)
- ◆ Nell'esempio ho due conti correnti (**mioConto** e **tuoConto**) ed eseguo varie operazioni

```
ContoCorrente mioConto, tuoConto;  
mioConto = new ContoCorrente();  
tuoConto = new ContoCorrente();  
  
mioConto.deposita(150.0);  
tuoConto.deposita(25.50);  
mioConto.deposita(tuoConto.prelieva(10.75));  
  
double mioSaldo = mioConto.ottieniSaldo();  
double tuoSaldo = tuoConto.ottieniSaldo();
```

Giroconto

Schema per la definizione di classi



Definizione della classe ContoCorrente

```
class ContoCorrente {
    // Lo stato è rappresentato solo dal saldo
    private double saldo;

    // Metodo per il deposito
    public void deposita( double somma ) {
        saldo = saldo + somma;
    }

    // Metodo per il prelievo
    public double preleva( double somma ) {
        saldo = saldo - somma;
        return somma;
    }

    // Metodo per la richiesta del saldo
    public double ottieniSaldo( ) {
        return saldo;
    }
}
```

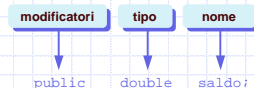
Definizione dello stato

- ◆ Lo stato di un oggetto si esprime attraverso variabili detti **attributi** o **proprietà** dell'oggetto
- ◆ Nella definizione di classe è necessario specificare le proprietà dichiarando le variabili corrispondenti
- ◆ Nell'esempio **ContoCorrente** vi è una sola proprietà **saldo** di tipo **double** che rappresenta il saldo disponibile
- ◆ Ogni proprietà è associata ad una caratteristica distintiva dell'oggetto
- ◆ Nell'esempio **ContoCorrente** gli oggetti **mioConto** e **tuoConto** si differenziano per il valore del saldo
- ◆ Una definizione di classe può essere priva delle dichiarazioni di attributi (ad es. **PrimoProgramma**)

Definizione delle proprietà

```
class <nome classe> {
    <modificatori> <tipo> <nome attributo>;
    ...
    <modificatori> <tipo> <nome attributo>;

    // Definizione dei metodi
}
```



Modificatori public e private

- ◆ I modificatori **public** e **private** impostano l'accessibilità degli attributi e dei metodi
- ◆ Se il componente di una classe (metodo o attributo) viene dichiarato **private**, nessun metodo definito esternamente alla classe può accedervi
- ◆ Se il componente di una classe viene dichiarato **public**, qualsiasi metodo esterno può accedervi

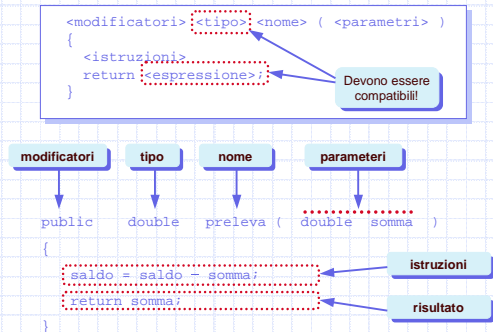
```
class Test
{
    public int memberOne;
    private int memberTwo;
}
```

```
Test myTest = new Test();
myTest.memberOne = 10; ✓
myTest.memberTwo = 20; ✗
```

Definizione delle modalità di interazione

- ◆ Un oggetto interagisce con l'esterno tramite i **metodi**
- ◆ Nell'esempio **ContoCorrente** per depositare, prelevare e per ottenere il saldo usiamo gli appositi metodi
 - La proprietà **saldo** è dichiarata **private**, quindi non è accessibile direttamente (incapsulazione)
 - I metodi realizzano modalità di accesso al saldo identiche per tutti gli oggetti della classe (omogeneità)
- ◆ I metodi dichiarati **public** definiscono l'interfaccia dell'oggetto con l'esterno
- ◆ I metodi dichiarati **private** sono utilizzabili solo dai metodi dell'oggetto e servono, ad esempio, per scomporre altri metodi in componenti più semplici

Definizione dei metodi



Funzioni e procedure

- La definizione presuppone che il metodo sia una **funzione**, ossia che restituisca un valore calcolato sulla base del valore dei parametri e dello stato dell'oggetto
- Vi sono metodi (ad es., **deposita**) che sono **procedure**, ossia eseguono un insieme di operazioni e non restituiscono alcun valore
- Tali metodi hanno tipo **void** (letteralmente, "vano") e sono **privi dell'istruzione return**

```

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

Aggiunta di commissioni a ContoCorrente (1)

- Supponiamo di applicare una commissione percentuale sull'importo di ogni operazione eseguita sul conto
- Tale commissione è variabile e soggetta a negoziazione tra la banca e il cliente
- ContoCorrente necessita di almeno un nuovo attributo:

```
private double commissione;
```

- e di un nuovo metodo:

```

public void impostaCommissione ( double tasso ) {
    commissione = tasso;
}
    
```

Aggiunta di commissioni a ContoCorrente (2)

- Per tenere conto della commissione dobbiamo inoltre modificare i metodi preleva e deposita:

```

public void deposita ( double somma ) {
    saldo = saldo + somma;
    saldo = saldo - (somma * commissione);
}

public double preleva ( double somma ) {
    saldo = saldo - (somma + somma * commissione);
    return somma;
}
    
```

Aggiunta di commissioni a ContoCorrente (3)

- Il seguente frammento di codice applica correttamente le commissioni negoziate dello 0.5% all'operazione di deposito:

```

ContoCorrente mioConto = new ContoCorrente();
mioConto.impostaCommissione(0.005);
mioConto.deposita(250.00);
    
```

- Questo codice invece è problematico:

```

ContoCorrente mioConto = new ContoCorrente();
mioConto.deposita(250.00);
    
```

- Nessuno ci garantisce che la **commissione sia impostata correttamente** prima di eseguire delle operazioni!

Costruttori

- Un **costruttore** è un particolare metodo che viene eseguito ogni volta che si crea un oggetto della classe
- Lo scopo del costruttore è di inizializzare gli attributi di un oggetto in uno stato valido
- Il nome del costruttore è lo stesso del nome della classe
- Si possono definire più costruttori per una stessa classe
- Se non viene definito alcun costruttore allora il compilatore Java includerà un **costruttore di default**
- Se viene definito almeno un costruttore il compilatore non include il costruttore di default

Aggiunta di commissioni a ContoCorrente (4)

- ◆ Definiamo un costruttore per ContoCorrente:

```
public ContoCorrente ( double tasso ) {  
    saldo = 0;  
    commissione = tasso;  
}
```

- ◆ L'utilizzo di ContoCorrente diventa:

```
ContoCorrente mioConto = new ContoCorrente(0.005);  
mioConto.deposita(250.00);
```

- ◆ **Attenzione!** A questo punto non è più consentito scrivere:

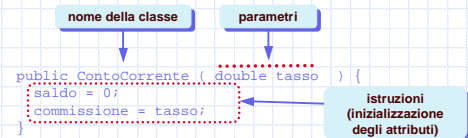
```
ContoCorrente mioConto = new ContoCorrente();
```

Definizione di costruttore

- ◆ Sintassi:

```
public <nome classe> ( <parametri> ) {  
    <istruzioni>  
}
```

- ◆ I costruttori sono usualmente **public** in quanto un costruttore **private** impedirebbe di creare oggetti



Costruttore di default

- ◆ Il costruttore di default equivale al seguente codice:

```
public <nome classe> ( ) {  
}
```

Un costruttore di default non ha alcuna istruzione nel suo corpo.

- ◆ Nel caso di ContoCorrente, non definire alcun costruttore equivale a definire come unico costruttore:

```
public ContoCorrente ( ) {  
}
```

Costruttori multipli

- ◆ Una classe può includere più costruttori a patto che
 - abbiano un diverso numero di parametri, oppure
 - abbiano parametri di tipo diverso
- ◆ Ad es, nel caso di ContoCorrente potresti definire:

```
public ContoCorrente( ) {  
    saldo = 0;  
    commissione = 0;  
}  
public ContoCorrente( double tasso ) {  
    saldo = 0;  
    commissione = tasso;  
}  
public ContoCorrente( double somma, double tasso ) {  
    saldo = somma;  
    commissione = tasso;  
}
```

Variabili locali

- ◆ Gli attributi sono accessibili a tutti i metodi definiti in una classe (vedi deposita, preleva, ... di ContoCorrente)
- ◆ Una variabile è locale quando è dichiarata all'interno della definizione di un metodo
- ◆ Le variabili locali sono accessibili solo all'interno del metodo in cui sono dichiarate
- ◆ Lo spazio in memoria per le variabili locali è allocato solo durante l'esecuzione del metodo: terminata l'esecuzione, lo spazio in memoria viene liberato
- ◆ I parametri di un metodo sono variabili locali al metodo

Esempio di metodo con variabili locali

- ◆ Metodo deposita di ContoCorrente modificato per restituire l'importo versato al netto delle commissioni

```
public double deposita( double somma )  
{  
    double importo, tasso;  
    tasso = somma * commissione;  
    importo = somma - commissione;  
    saldo = saldo + importo;  
    return importo;  
}
```

Utilizzo di variabili locali (1)

Sorgente

```

importoEffettivo
= mioConto.deposita( 200.00 );

public double deposita( double somma )
{
    double importo, tasso;
    tasso = somma * commissione;
    importo = somma - commissione;
    saldo = saldo + importo;
    return importo;
}
    
```

Ad **A** prima di **deposita**

Memoria

A. Le variabili locali non esistono prima dell'esecuzione del metodo.

Utilizzo di variabili locali (2)

Sorgente

```

importoEffettivo
= mioConto.deposita( 200.00 );

public double deposita( double somma )
{
    double importo, tasso;
    tasso = somma * commissione;
    importo = somma - commissione;
    saldo = saldo + importo;
    return importo;
}
    
```

Dopo **B**

Memoria

somma
 importo
 tasso

B. Viene allocato spazio in memoria per le variabili locali (importo e tasso) e il parametro (somma) in cui viene copiato l'argomento (200.00).

Utilizzo di variabili locali (3)

Sorgente

```

importoEffettivo
= mioConto.deposita( 200.00 );

public double deposita( double somma )
{
    double importo, tasso;
    tasso = somma * commissione;
    importo = somma - commissione;
    saldo = saldo + importo;
    return importo;
}
    
```

Dopo **C**

Memoria

somma
 importo
 tasso

C. I valori calcolati sono assegnati alle variabili.

Utilizzo di variabili locali (4)

Sorgente

```

importoEffettivo
= mioConto.deposita( 200.00 );

public double deposita( double somma )
{
    double importo, tasso;
    tasso = somma * commissione;
    importo = somma - commissione;
    saldo = saldo + importo;
    return importo;
}
    
```

A **D** dopo **deposita**

Memoria

D. La memoria assegnata alle variabili locali viene deallocata all'uscita di **deposita**.

Metodi e passaggio argomenti (1)

- ◆ Nella chiamate ai metodi
 - il numero di argomenti è pari al numero di parametri
 - il tipo degli argomenti è compatibile con il tipo dei parametri
- ◆ Ad esempio, le chiamate

```

ContoCorrente mioConto =
    new ContoCorrente(300.00, 0.005);
mioConto.deposita(250.00);
    
```

corrispondono alle definizioni

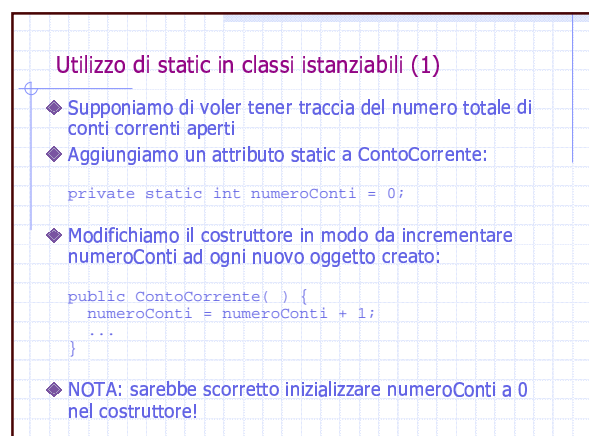
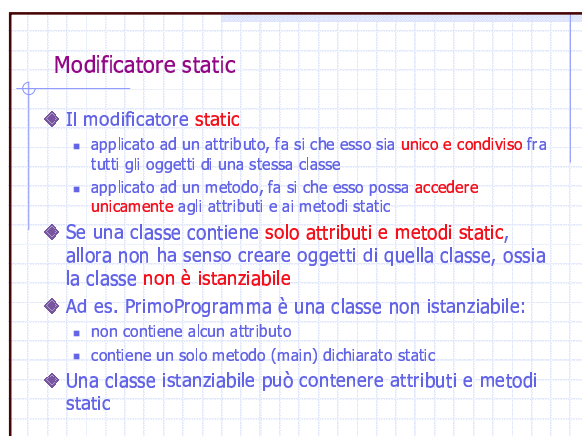
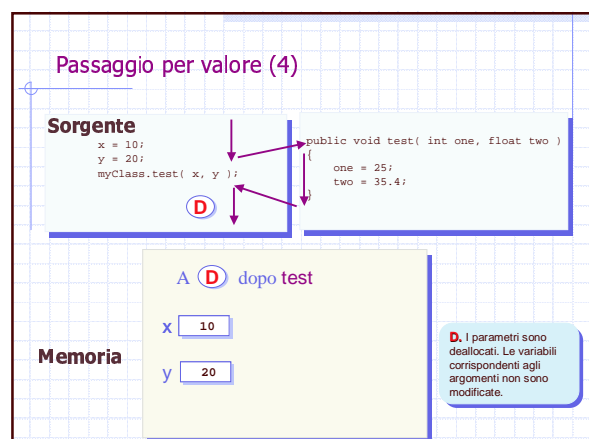
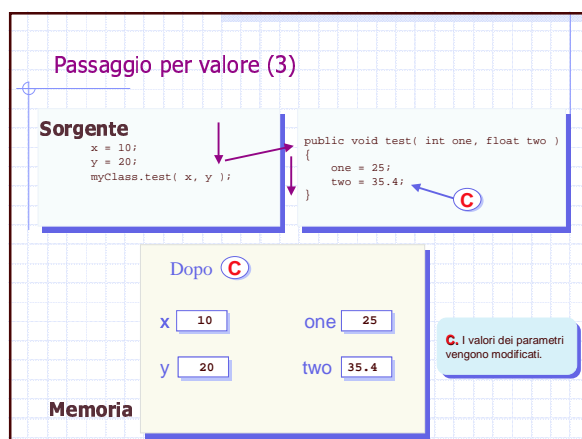
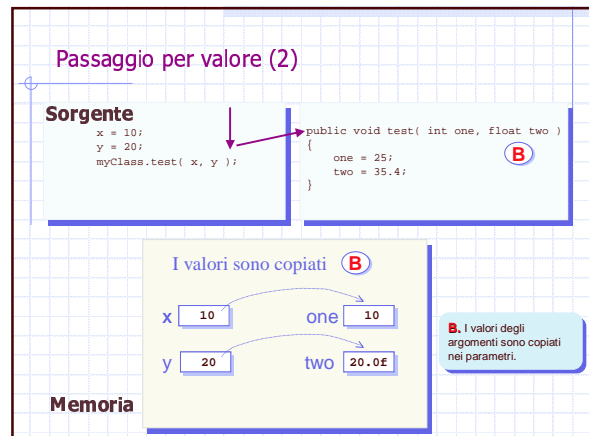
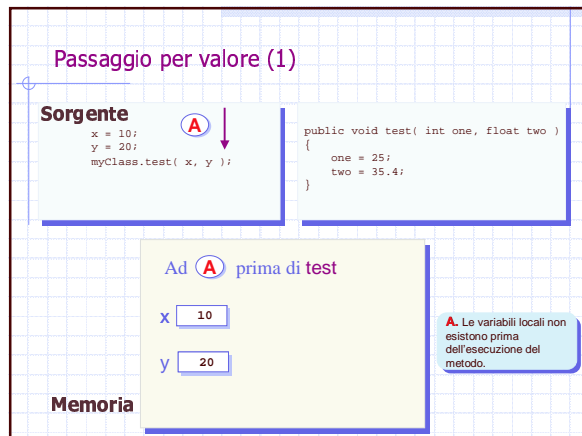
```

public ContoCorrente( double somma, double tasso )

public void deposita( double somma )
    
```

Metodi e passaggio argomenti (2)

- ◆ Quando un metodo viene richiamato, il valore degli argomenti viene **copiato** all'interno dei parametri
- ◆ Questo meccanismo è noto come **call-by-value** (**passaggio per valore**)
- ◆ Le eventuali modifiche ai valori dei parametri all'interno del metodo non si ripercuotono sul valore degli argomenti prima della chiamata al metodo
- ◆ In Java **tutti i parametri** che hanno **tipo primitivo** seguono il meccanismo di passaggio per valore
- ◆ Considerazioni analoghe valgono per la restituzione del risultato al termine del metodo (con l'istruzione **return**)



Utilizzo di static in classi istanziabili (2)

- ◆ Aggiungiamo un metodo per interrogare il numero di conti aperti (sarebbe appropriato anche un metodo static):

```
public int ottieniNumeroConti( ) {
    return numeroConti;
}
```

- ◆ Il seguente frammento di codice illustra il funzionamento:

```
ContoCorrente mioConto = new ContoCorrente();
ContoCorrente tuoConto = new ContoCorrente();
int conti = mioConto.ottieniNumeroConti( );
```

- ◆ Stesso risultato con `tuoConto.ottieniNumeroConti()`

Flashback sul metodo main

- ◆ Un programma Java deve contenere almeno una classe (classe principale) con un metodo main dichiarato come

```
public static void main( ... ) {
    ...
}
```

- ◆ Quindi main è una **procedura, disponibile all'esterno, uguale per tutti gli oggetti della classe principale**
- ◆ L'esecuzione di un programma Java inizia sempre con una chiamata al metodo main
- ◆ La classe principale può contenere attributi e metodi che, per essere utilizzati dal main, devono essere necessariamente dichiarati static

Attributi costanti

- ◆ Gli attributi sono **normalmente** definiti come **variabili** in quanto lo stato degli oggetti evolve nel tempo
- ◆ Se una porzione dello stato degli oggetti è **immutabile nel tempo** allora
 - è uguale per tutti gli oggetti della classe (static)
 - è definito tramite una **costante** (final)
- ◆ Supponiamo che la commissione applicata ad ogni operazione sul conto sia fissata dalla banca allo 0.6%
- ◆ La dichiarazione dell'attributo commissione diventa

```
public static final double commissione = 0.006;
```

- ◆ L'utilizzo della commissione non cambia nei metodi preleva e deposita, ma non compare più nel costruttore

Riassunto (1) – public vs. private

	Proprietà	Metodi
public	Per gli attributi costanti che devono essere visibili all'esterno (ad es. la commissione fissata per tutti i conti correnti)	Per i metodi che definiscono le modalità di interazione degli oggetti (ad es. deposita, preleva, ...)
private	Per tutti gli attributi variabili (ad es. il saldo o la commissione negoziata per ogni conto corrente) e per le costanti interne	Per i metodi che risultano dalla scomposizione di altri metodi della classe

Riassunto (2) – static vs. non static

	Proprietà	Metodi
static	Per tutti gli attributi costanti e per gli attributi variabili che devono essere unici e condivisi fra tutti gli oggetti (ad es. il numero dei conti)	Per i metodi che devono utilizzare esclusivamente attributi e metodi static e per il metodo main (ad es. ottieniNumeroConti)
non static	Per tutti gli attributi variabili il cui valore cambia a seconda dell'oggetto considerato	Per i metodi che devono utilizzare attributi e metodi non static

Riassunto (3) – final vs. non final

	Proprietà	Metodi
final	Per tutti gli attributi costanti (ad es. la commissione non negoziabile)	Non utilizzato (serve nel contesto del meccanismo di ereditarietà fra classi che non abbiamo affrontato)
non final	Per tutti gli attributi variabili	Per tutti i metodi

Esercitazione – Classi istanziabili

- ◆ Realizzare una variante della classe ContoCorrente con:
 - commissione percentuale sulle operazioni negoziabile
 - conteggio del numero di conti aperti
 - diritto fisso calcolato in percentuale sull'importo di ogni operazione di prelievo eseguita se il saldo è minore di 0
- ◆ Realizzare una classe per rappresentare equazioni di secondo grado, con metodi per impostare i coefficienti e per calcolare le soluzioni
- ◆ Realizzare la classe DistributoreAutomatico con le seguenti ipotesi:
 - Vi sono solo tre bibite: chinotto, aranciata, the freddo
 - Il costo delle bibite è fissato a 55 centesimi
 - l'utente può inserire monetine da 5, 10 e 50 centesimi
 - si desidera tenere traccia della somma complessiva presente nelle casse dei distributori

Unità didattica 3 – Argomenti svolti

- Definizione di una classe: stato (**attributi**) e modalità di interazione (**metodi**)
- Modificatori di visibilità **public** e **private**
- Funzioni, procedure e il tipo **void**
- Inizializzazione dello stato tramite **costruttori**
- Variabili locali e passaggio parametri
- Modificatore **static**, attributi e metodi propri della classe e classi non istanziabili
- Modificatore **final** e attributi costanti
- Scelta dei modificatori **public**, **private**, **static**, e **final** nella definizione della classe