Realizzare Classi

Astrazione

- Utile nella descrizione, progettazione, implementazione e utilizzo di sistemi complessi
- Procedimento con il quale si sostituiscono composizioni di elementi noti con nuovi elementi astraendo i dettagli della composizione
  - dettagli trascurabili vengono *incapsulati* in sottosistemi 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 interagire con essa

Livelli di astrazione

- Nei sistemi complessi si hanno diversi livelli di astrazione
- Es. automobile:
  - Il motore è una scatola nera per l'autista
  - I pistoni, le ventole, le centraline, etc. sono scatole nere per il meccanico
- L'astrazione
  - apporta semplificazione e specializzazione
  - migliora l'efficienza

Astrazione in Software Design

- I linguaggi di programmazione definiscono attraverso i tipi di dati primitivi e le istruzioni un dominio su cui definire le soluzioni ai problemi
- Solitamente il dominio dato da un linguaggio di programmazione e il dominio del problema sono sensibilmente differenti
- Il meccanismo dell'astrazione fornisce al dominio delle soluzioni concetti analoghi a quelli del dominio dei problemi
- Linguaggi di programmazione che facilitano le astrazioni sono auspicabili

Programmazione orientata agli oggetti

- Le classi e gli oggetti sono gli strumenti per realizzare le astrazioni
  - la classe definisce/implementa un nuovo concetto (di più alto livello rispetto a quelli esistenti)
  - gli oggetti consentono l'utilizzo dell'astrazione implementata dalla classe (in un programma)
- Incapsulamento: per utilizzare un oggetto non è necessario conoscere la sua struttura interna

Programmazione orientata agli oggetti

- Definire buone astrazioni non è semplice
  - è l'aspetto più importante di una buona progettazione O.O.

- In una corretta progettazione (ad oggetti) prima si definiscono le classi e poi si implementano

Esempio: contapersone

- Un “tally counter” è un semplice dispositivo che mantiene un conteggio che può essere
  - incrementato
  - azzerato
  - visualizzato

Figure 1 A Tally Counter

- Simulazione di un conteggio:

```
Counter tally = new Counter();
tally.count();
tally.count();
int result = tally.getValue(); // Sets result to 2
```

Realizzazione di un tally counter

- Ogni oggetto di tipo Counter deve mantenere un conteggio autonomo dagli altri
  - necessita di una propria variabile value (variabile di istanza)
  - la variabile è un dettaglio di implementazione che può essere nascosto (incapsulamento)

- Per garantirne la funzionalità devono poter essere invocati i seguenti metodi:
  - count(): incrementa value
  - reset(): azzera value
  - getValue(): restituisce il valore di value

In Java:

```
public class Counter{
        private int value;


        public void count(){
                value = value + 1;
        }

        public int getValue(){
                return value;
        }

        public void reset(){
                value = 0;
        }
}
```

Classi

- Il comportamento di un oggetto è descritto da una *classe*
- Ogni classe ha
  - Interfaccia pubblica
    - Un 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)
  - Implementazione nascosta
    - Codice e variabili usati per implementare i medodi dell'interfaccia e non accessibili all'esterno della classe
    - Es.: x,y,width,height

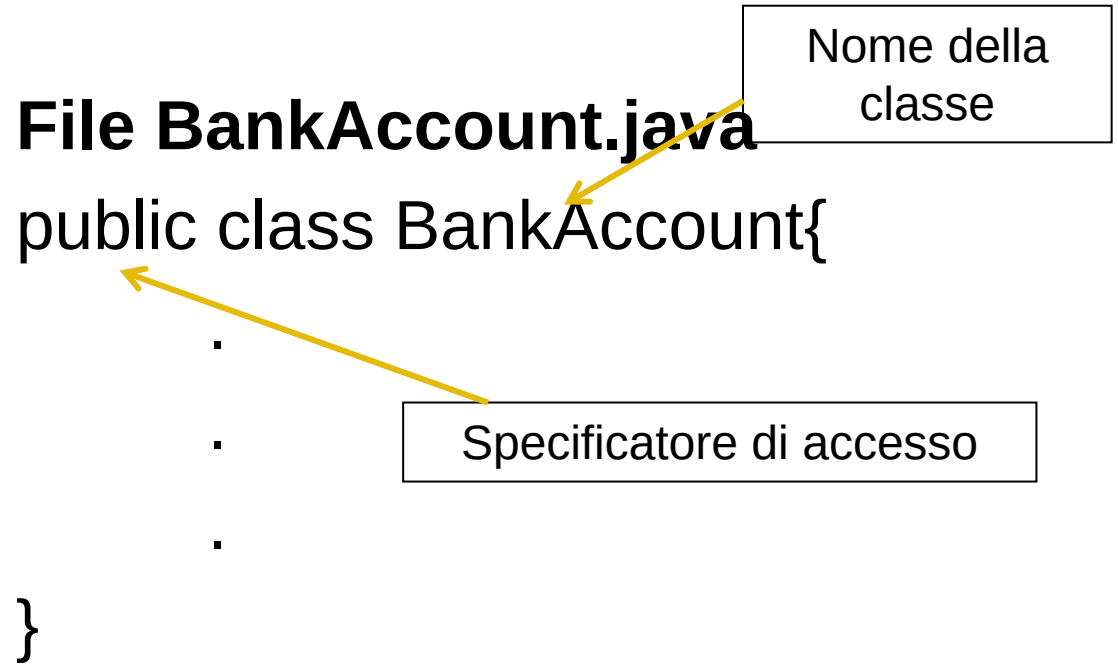
Sintassi definizione di classe

```
accessSpecifier class ClassName
 {
    constructors
    methods
    fields
 }
```

Example:

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

Definizione di una classe



- *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 **public** deve essere contenuta in un file avente il suo stesso nome
  - Es.: la classe BankAccount è memorizzata nel file BankAccount.java

Progettazione dell'interfaccia pubblica

- Comportamento di un conto corrente bancario (astrazione):
  - deposita contante
  - preleva contante
  - legge saldo
- Necessaria la memorizzazione del saldo (variabile di istanza)

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());
```

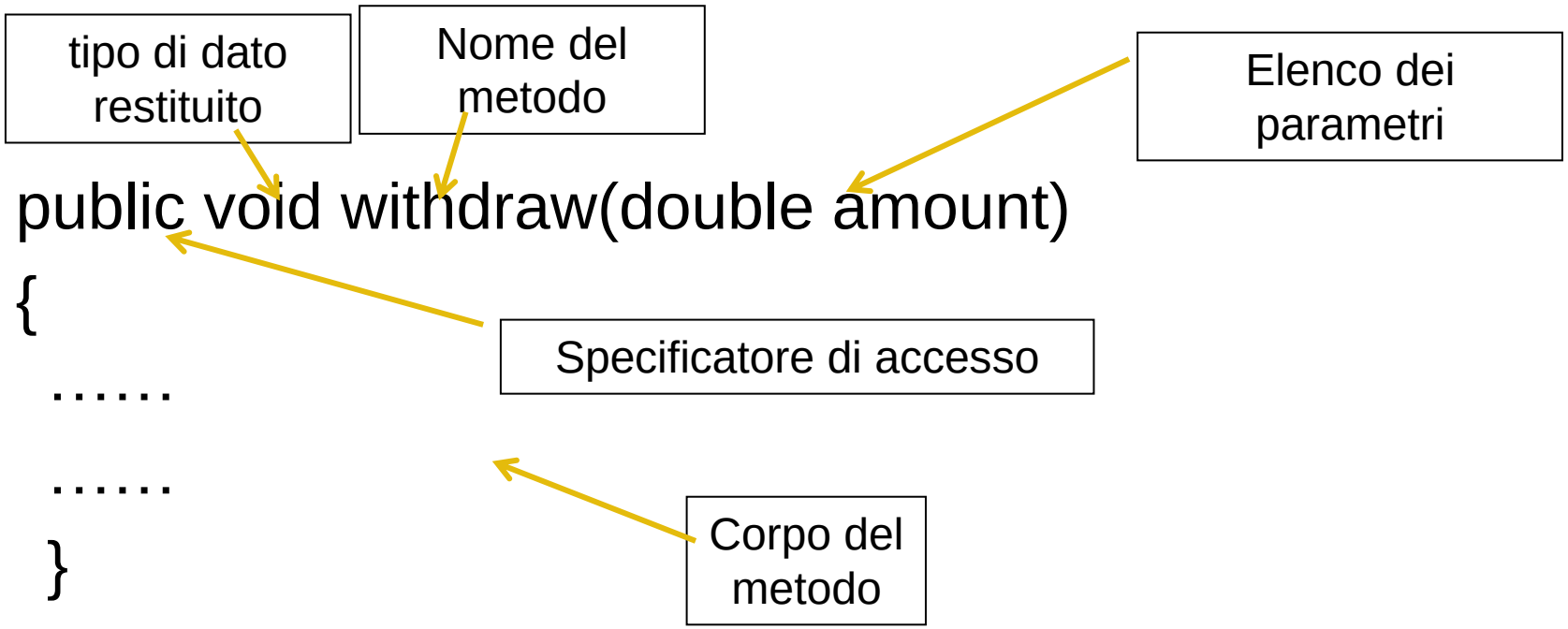
Definizione di un metodo

```
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 nei metodi esterni alla classe BankAccount (e anche in quelli esterni al package a cui appartiene la classe BankAccount)

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
   }

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

BankAccount: Interfaccia Pubblica

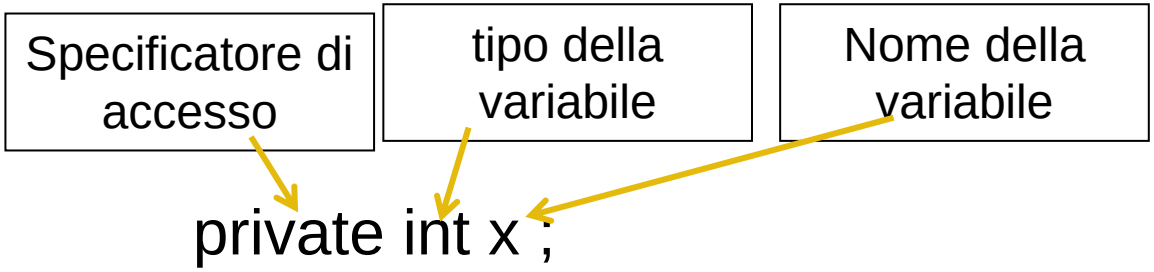
```
    {
       // body--filled in later
    }

    public void withdraw(double amount)
    {
       // body--filled in later
    }
    public double getBalance()
    {
       // body--filled in later
    }
    // private fields--filled in later
}
```

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 **public** della classe
    - 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

} Tipi riferimento

Uso variabili di istanza nei metodi

- Il metodo deposit di `BankAccount` può accedere alla variabile di istanza balance (dichiarata `private`):

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

Accesso variabili di istanza

- Le variabili dichiarate private non sono visibili in altre classi:

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

- Realizzazione incapsulamento:
  - si nasconde il dato contenuto nelle variabili di istanza usando lo specificatore private e
  - se ne fornisce l'accesso solo attraverso i metodi dell'interfaccia pubblica

Metodo withdraw

- Il metodo withdraw di `BankAccount` sottrae dal saldo il valore prelevato dal conto:

```
public void withdraw(double amount){
      double newBalance = balance - amount;
      balance = newBalance;
}
```

Esecuzione metodo

```
harrysChecking.withdraw(500);
```

- Si valuta la variabile harrysChecking

(riferimento ad un oggetto BankAccount in memoria)

- Dal valore di harrysChecking e l'identificatore withdraw si determina l'indirizzo del metodo

(il codice di withdraw è contenuto nello spazio di memoria occupato dalla classe BankAccount)

Esecuzione:

```
harrysChecking.withdraw(500);
```

```
public void withdraw(double amount){
      double newBalance = balance - amount;
      balance = newBalance;
}
```

1. Si assegna il parametro esplicito `amount` a 500
2. Si recupera il contenuto del campo `balance` dell'oggetto la cui locazione è salvata in `harrysChecking`
3. Si sottrae il valore `amount` da `balance` e si salva il risultato in `newBalance`
4. Si salva il valore di `newBalance` in `balance`, sovrascrivendo il vecchio valore

Significato di this (1)

- All'interno di un metodo per riferirsi esplicitamente al parametro implicito si può usare la parola chiave this
- Il nome di una variabile di istanza all'interno di un metodo di una classe si riferisce alla variabile di istanza del parametro implicito

```
public void trasferisci(double somma, Conto conto)
    {
        saldo = saldo - somma;
        conto.saldo = conto.saldo + somma;
    }
```

Significato di this (2)

- A volte per chiarezza si usa this.nomeMetodo o this.nomeVariabile

```
public void trasferisci(double somma, Conto conto)
        {
                this.saldo = this.saldo - somma;
                conto.saldo = conto.saldo + somma;
        }
```

- Altre volte è necessario

```
public void saldoNuovo(double saldo)
        {
                this.saldo = saldo;
        }
```

Significato di this (3)

- All'interno di un metodo A, se un metodo B viene usato senza parametro implicito allora si assume che il parametro implicito di B è quello con cui abbiamo invocato A
  - B deve essere un metodo invocabile sul parametro implicito di A

```
public void trasferisci(double somma, Conto conto)
        {
                preleva(somma);
                conto.deposita(somma);
        }
```

- Anche in questo caso si può usare this:

```
this.preleva(somma);
```

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 in genere 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

Implementazione Costruttori

- Contengono istruzioni per inizializzare le variabili di istanza

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

Invocazione Costruttore

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

- Crea un nuovo oggetto di tipo `BankAccount`
- Chiama il secondo costruttore (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`

Documentazione del software

- Consiste nell'insieme delle informazioni fornite insieme al software per facilitarne l'uso e descriverne la struttura e le caratteristiche tecniche
  - Buona parte della documentazione è costituita dal manuale utente
- In fase di sviluppo, è un indispensabile ausilio per l'interazione tra i programmatori
- In fase di manutenzione, fornisce rapidamente le informazioni necessarie a chi dovrà operare modifiche o aggiunte al prodotto iniziale
- E' un aspetto centrale degli standard di progettazione del software

Documentazione del software in Java

- Si genera automaticamente dai commenti nel codice utilizzando il tool `javadoc`
- Lanciando `javadoc` si ottiene una documentazione in forma di ipertesto HTML
  - La documentazione di Java, detta Java API (Java Application Programming Interface), è generata automaticamente con `javadoc`
- Vantaggi:
  - facile da consultare (ipertesto)
  - facile da realizzare (si forniscono i commenti mentre si struttura l'interfaccia delle classi)

Commenti per documentazione javadoc

```
/**
   Withdraws money from the bank account.
   @param amount 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

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:    */
```

File BankAccount.java

```
19:    public BankAccount(double initialBalance)
20:    {
21:       balance = initialBalance;
22:    }
23:
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
```

File BankAccount.java

```
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:     */
48:     public double getBalance()
49:     {
50:        return balance;
51:     }
52:
53:     private double balance;
54: }
```

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

Testare una classe

- Importante:
  - Copertura: ogni istruzione dei metodi da testare devono essere eseguite almeno una volta
  - Rieseguibilità: il test deve poter essere rieseguito sotto le stesse condizioni (nessun input da operatore, dati presi da file o da codice)
  - Tracciamento: visualizzazione degli effetti delle operazioni eseguite (stato oggetti prima e dopo le operazioni, indicazione dell'operazione eseguita)
- NOTA: per motivi di spazio negli esempi si riporteranno classi tester minimali

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:    }
17: }
```

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

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 public che contiene un metodo main(String[ ] args)