DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Il progetto di BankAccount
I costruttori

Prof. Luca Trevisan

Corso di Fondamenti di Informatica

A.A. 2025/2026

I costruttori

- La classe **BankAccount** è stata realizzata quasi completamente, manca il codice per ***creare un nuovo conto bancario***, con saldo a zero

- Per consentire la creazione di un nuovo esemplare di una classe, ***inizializzandone lo stato***, dobbiamo scrivere un ***costruttore*** per la classe

```
public class BankAccount
{  private double balance;
   public BankAccount()
   {  balance = 0;
   }
   ...
}
```

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

I costruttori

- La sintassi utilizzata per definire i costruttori è molto simile a quella dei metodi, ma
  - il ***nome*** dei costruttori è sempre uguale a quello della classe (quindi iniziale maiuscola…)
  - ***i costruttori non restituiscono alcun valore***
    e **non bisogna neppure dichiarare che restituiscono void**
- Lo ***scopo principale*** di un costruttore è quello di ***inizializzare le variabili di esemplare*** di un oggetto
- I ***costruttori***, come i metodi, sono ***solitamente pubblici***, per consentire a chiunque di creare esemplari della classe

Invocazione di costruttori

- I costruttori si invocano ***soltanto*** con l'operatore **new**

```
new BankAccount();
```

- L'operatore **new** ***predispone la memoria*** per l'oggetto sulla base del progetto (cioè delle variabili di esemplare definite nella classe), mentre (generalmente) il costruttore definisce il suo stato iniziale
- Il ***valore*** restituito dall'operatore **new** è il ***riferimento*** all'oggetto appena creato e inizializzato (cioè il suo indirizzo)
  - Di solito il valore restituito dall'operatore **new** viene memorizzato in una ***variabile oggetto***

```
BankAccount account = new BankAccount();
// ora account fa riferimento a un nuovo
// conto avente saldo uguale a zero
```

Definizione di costruttori

- Sintassi:

```
public class NomeClasse
{   ...
    tipoDiAccesso NomeClasse(TipoParametro nomeParametro,...)
    {  realizzazione del costruttore
    }
}
```

- Scopo: definire il comportamento di un costruttore della classe ***NomeClasse***
- Possono essere presenti ***parametri di costruzione***, con sintassi identica ai parametri espliciti dei metodi, sia nella definizione sia nell'invocazione
- Nota: i costruttori servono (di norma) a inizializzare le variabili di esemplare di oggetti appena creati
- Nota: il ***tipoDiAccesso*** è solitamente **public**

Sintassi alternativa (sconsigliata)

- Invece di definire un costruttore, si possono inizializzare le variabili di esemplare contestualmente alla loro definizione

```
public class BankAccount
{
    private double balance = 0;
    . . .
}
```

- Soluzione meno flessibile:
  - in un costruttore, i valori iniziali delle variabili possono anche essere frutto di una elaborazione dei parametri di costruzione.
  - **MEGLIO USARE IL COSTRUTTORE**

Sintassi alternativa

- Le variabili di esemplare eventualmente non inizializzate vengono **comunque inizializzate** a un valore predefinito

- Se poi viene loro associato un valore all'interno di un costruttore, quest'ultimo **prevale** (perché tale assegnazione avviene DOPO)

Valori predefiniti per le variabili di esemplare

- Le variabili di esemplare che non vengono inizializzate assumono automaticamente i seguenti valori
  - ***zero*** se sono di tipo ***numerico***
    - comprese le variabili di tipo **char**
  - **false** se sono di tipo **boolean**
  - Il ***valore speciale*** **null** se sono variabili oggetto, in modo che non si riferiscano ad alcun oggetto

È meglio non programmare in questo modo!
Impariamo a definire i costruttori

Il riferimento null

- Una variabile di un tipo numerico fondamentale contiene sempre un valore valido (eventualmente casuale, se non è stata inizializzata in alcun modo)

```
BankAccount account = null;
```

- Una variabile oggetto può invece contenere esplicitamente un riferimento a nessun oggetto valido assegnando alla variabile il valore **null**, che è una parola chiave del linguaggio
  - vedremo in seguito applicazioni utili di questa proprietà
  - in questo caso la variabile risulta comunque inizializzata

Usare un riferimento null

- Una variabile oggetto che contiene un riferimento **null** non si riferisce ad alcun oggetto
  - non può essere usata per invocare metodi

- Se viene usata per invocare metodi, l'interprete termina l'esecuzione del programma, segnalando un errore di tipo **NullPointerException** (perché puntiamo a Null)

Il costruttore predefinito

- E se ***non definiamo un costruttore*** per una classe?
  - ***il compilatore si comporta come se fosse presente il "costruttore predefinito"***
    (senza alcuna segnalazione d'errore)

Il costruttore predefinito

- Il costruttore predefinito (e implicito) di una classe
  - ***è pubblico e non richiede parametri***
  - ***esiste solo se NON sono definiti costruttori espliciti***
  - ***non fa niente!*** Di conseguenza, le variabili di esemplare mantengono i valori iniziali assunti in base alle regole viste in precedenza

- È sintatticamente necessaria la presenza di un costruttore (esplicito o implicito) per il funzionamento dell'operatore **new**

Riassumendo

- **Se non definisco il costruttore**
  - Viene utilizzato il costruttore implicito. Le variabili di esemplare assumono il loro valore di default a seconda del tipo
  - Se ho assegnato esplicitamente un valore a qualche variabile, verrà utilizzato questo valore
- **Se definisco un costruttore**
  - Anche se ho assegnato esplicitamente un valore a qualche variabile, il valore verrà sovrascritto secondo le indicazioni del costruttore

- Domanda: si possono avere più costruttori?

Una classe con più costruttori

- Una classe può avere più di un costruttore!
- Ad esempio, definiamo un costruttore per creare un nuovo conto bancario con un saldo iniziale diverso da zero

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

   public BankAccount()
   {  balance = 0;
   }
   ...
}
```

Una classe con più costruttori

- Per usare il nuovo costruttore di BankAccount, bisogna fornire il parametro initialBalance

```
BankAccount account = new BankAccount(500);
```

- Notiamo che, se esistono più costruttori in una classe, **hanno tutti lo stesso nome**, perché devono comunque avere lo stesso nome della classe
  - questo fenomeno (più metodi o costruttori con lo stesso nome) è detto sovraccarico del nome (**overloading**)
  - il **compilatore decide** quale costruttore invocare per la creazione di un oggetto, **basandosi sul numero e sul tipo dei parametri** forniti nell'invocazione

Una classe con più costruttori

- Il compilatore effettua la risoluzione dell'ambiguità nell'invocazione di costruttori o metodi sovraccarichi
- Se non trova un costruttore che corrisponda ai parametri forniti nell'invocazione, segnala un errore semantico

```
// NON FUNZIONA!
BankAccount a = new BankAccount("tanti soldi");
```

```
cannot find symbol
symbol   : constructor BankAccount (java.lang.String)
location : class BankAccount
```

Sovraccarico del nome

- Se si usa lo stesso nome per metodi diversi, il nome diventa sovraccarico (nel senso di carico di significati diversi…)
  - questo accade spesso con i costruttori, dato che se una classe ha più di un **costruttore**, essi devono avere lo stesso nome
  - accade più di rado con i **metodi**, ma c'è un motivo ben preciso per farlo (ed è bene farlo in questi casi)
    - usare lo stesso nome per metodi diversi (che richiedono parametri di tipo diverso) sta ad indicare che viene compiuta la stessa elaborazione su tipi di dati diversi (esempio: metodi statici della classe Math)

Sovraccarico del nome

- La libreria standard di Java contiene numerosi esempi di metodi sovraccarichi

```
public class PrintStream
{  public void println(int n) {...}
   public void println(double d) {...}
   ...
}
```

- Quando si invoca un metodo sovraccarico, il compilatore **risolve** l'invocazione individuando quale sia il metodo richiesto **sulla base dei parametri espliciti** che vengono forniti

A proposito di stampa di informazioni…

- Abbiamo visto che stampando un oggetto BankAccount ci appare a video una scritta tipo
  - **`BankAccount@7852e922`**

- Se vogliamo avere informazioni più specifiche dobbiamo realizzare il metodo
  - **`public String toString()`**

```
public String toString() {
    return "BankAccount: saldo
corrente
        pari a " + balance;
}
```

Considerazioni sui diversi tipi di variabili nei metodi

- In un metodo non statico possiamo gestire diversi tipi di variabili
  - variabili d'istanza: appartengono all'oggetto ed esistono fino a quando esiste l'oggetto
    - Hanno un'inizializzazione automatica (ma sempre meglio usare il costruttore)
  - variabili parametro esplicito: esistono quando viene invocato il metodo
    - Le uso dentro al metodo con il nome definito nella firma
    - Vengono inizializzate con il valore specificato tra parentesi quando il metodo viene invocato
  - variabili locali: definite dentro al metodo, esistono solo dentro al metodo
    - Devono essere inizializzate
  - Attenzione: se parametri espliciti o variabili locali hanno lo stesso nome di una variabile d'istanza la "nascondono"!

La classe BankAccount completa

```
public class BankAccount
{   private double balance;

    public BankAccount()
    {   balance = 0;
    }
    public BankAccount(double initialBalance)
    {   balance = initialBalance;
    }
    public void deposit(double amount)
    {   balance = balance + amount;
    }
    public void withdraw(double amount)
    {   balance = balance - amount;
    }
    public double getBalance()
    {   return balance;
    }
    public String toString()
    {   return "BankAccount: saldo corrente pari a " + balance;
    }
}
```

`new BankAccount()`

Riassumendo...

- L'operatore **new** assegna all'oggetto una zona di memoria di dimensioni opportune (dipende dalla JVM)
- Nell'oggetto vengono memorizzate informazioni utili alla JVM per far funzionare l'oggetto (es: il nome della classe di cui è esemplare)
- Nell'oggetto vengono create le (eventuali) variabili di esemplare, ciascuna delle quali viene inizializzata
  - al valore eventualmente presente nella sua definizione oppure al valore predefinito (zero, **false** o **null**, in relazione al tipo di variabile)
- Viene eseguito il costruttore (eventualmente quello predefinito), che solitamente (ma non necessariamente) assegna un (nuovo) valore alle variabili di esemplare!
- L'operatore **new** restituisce l'indirizzo dell'oggetto creato

Esempio: utilizzo di BankAccount

- Attenzione: la classe **BankAccount** non è eseguibile!
  - Non ha il metodo **main**...

```
C:\java BankAccount
Exception in thread "main"
java.lang.NoSuchMethodError: main
```

- È una classe destinata a essere utilizzata da altre classi, dotate di **main**
- A volte, a soli scopi di collaudo, le classi vengono rese eseguibili anche se l'algoritmo non lo richiede, aggiungendo un metodo **main** che crei alcuni esemplari della classe stessa e li manipoli

Esempio: utilizzo di BankAccount

- Scriviamo un programma BankAccountTester2.java che ***usi*** la classe **BankAccount** per risolvere un problema specifico
  - viene aperto un conto bancario,
  - viene effettuato un versamento iniziale di 10000 euro
  - ogni anno viene automaticamente accreditato nel conto un importo (interesse annuo) pari al 5 per cento del valore del saldo, senza fare prelievi né depositi
  - qual è il saldo del conto dopo due anni?

Esempio: utilizzo di BankAccount

```
public class BankAccountTester2
{  public static void main(String[] args)
   {  BankAccount account = new BankAccount();
      account.deposit(10000);
      final double RATE = 5;
      // calcola gli interessi dopo il primo anno
      double interest = account.getBalance() * RATE / 100;
      // somma gli interessi dopo il primo anno
      account.deposit(interest);
      System.out.println("Saldo dopo un anno: "
         + account.getBalance() + " euro");
      // calcola gli interessi dopo il secondo anno
      interest = account.getBalance() * RATE / 100;
      // somma gli interessi dopo il secondo anno
      account.deposit(interest);
      System.out.println("Saldo dopo due anni: "
         + account.getBalance() + " euro");
   }
}
```

Riassumendo

- Passi per costruire un programma di collaudo
  - Definire una nuova classe
  - Definire in essa il metodo main
  - Costruire oggetti all'interno di main
  - Applicare metodi agli oggetti
  - Visualizzare risultati delle invocazioni dei metodi

Un programma con piu' classi

- Nota: per scrivere semplici programmi con più classi, si possono usare due strategie (equivalenti)

- Strategia 1
  - scrivere **ciascuna classe in un file diverso**, ciascuno avente il nome della classe con estensione .java
  - **BankAccount.java e BankAccountTester.java**, si compilano entrambe e si esegue solo quella che ha il **main**

Strategia 1

BankAccountTester.java

```
public class BankAccountTester
{
   public static void main(String[] args)
   {
      //definisco oggetto BankAccount
      BankAccount account =
            new BankAccount(500);

      //invoco metodi di BankAccount
      account.deposit(300);

      double currentBalance =
          account.getBalance();
   }
}
```

BankAccount.java

```
public class BankAccount
{
   private double balance;

   public BankAccount()
   { balance = 0;}
   public BankAccount(double initBalance)
   { balance = initBalance; }
   public deposit (double amount)
   { balance +=amount; }
   public widthdraw(double amount)
   { balance -=amount; }
   public getBalance()
   { return balance;}
}
```

Un programma con più classi

- Nota: per scrivere semplici programmi con più classi, si possono usare due strategie (equivalenti)

- Strategia 2
  - scrivere tutte le classi in un unico file
    - un file .java può contenere una sola classe public
    - la classe che contiene il metodo **main** deve essere public
      - quindi le altre classi non devono essere public (semplicemente non si indica l'attributo public)
    - il file deve avere il nome dell'unica classe public, sempre con estensione .java (qui, BankAccountTester.java)

Strategia 2

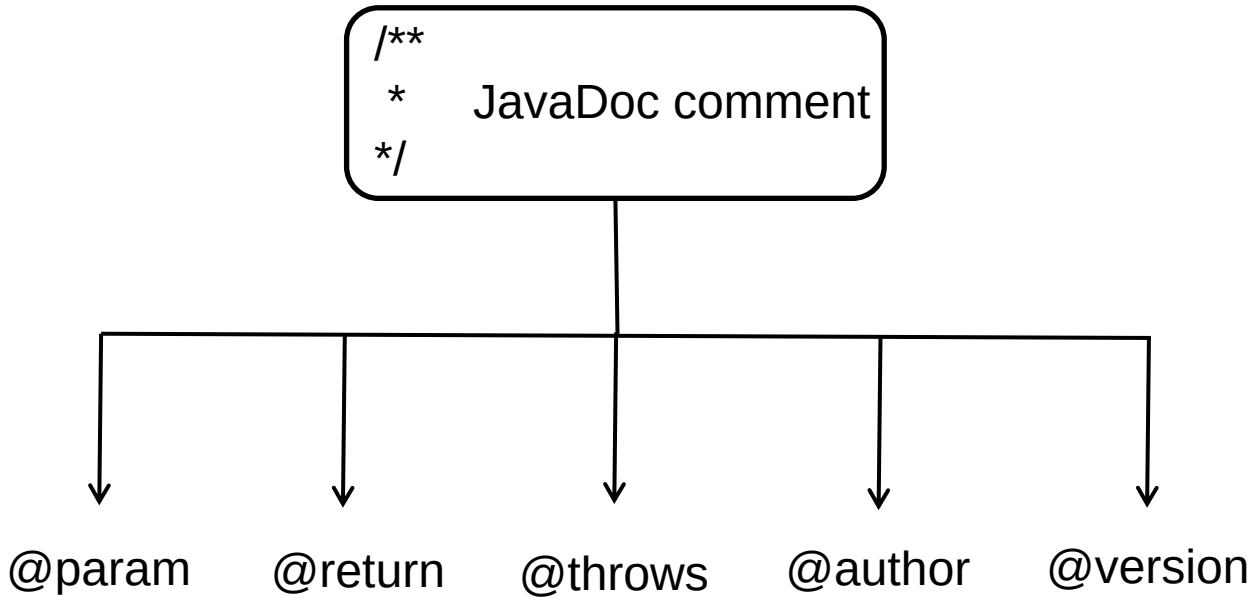
BankAccountTester.java

```
public class BankAccountTester
{
   public static void main(String[] args)
   {
     BankAccount account =
           new BankAccount(500);
     //invoco metodi di BankAccount
     account.deposit(300);
     double currentBalance =
                  account.getBalance();
   }
}
class BankAccount{
   private double balance;
   public BankAccount()
   { balance = 0;}
   public BankAccount(double initBalance)
   { balance = initBalance; }
   public void deposit (double amount)
   { balance +=amount; }
   public void widthdraw(double amount)
   { balance -=amount; }
   public double getBalance()
   { return balance;}
}
```

Commentare l'interfaccia pubblica

Commentare l'interfaccia pubblica

- Formato standard per i commenti di documentazione
  - si può poi utilizzare il programma **javadoc che genera automaticamente documentazione in formato HTML**



Commentare l'interfaccia pubblica

- Formato standard:
  - Inizio commento: /** delimitatore speciale
  - Fine commento: */
- Per ciascun metodo/costruttore si scrive un commento per descriverne lo scopo
- Se il commento si estende su più frasi (ogni frase è terminata da un punto), allora la prima frase deve essere un riassunto delle funzionalità del metodo

Commentare l'interfaccia pubblica

- **@param** seguito dal nome del parametro e da una breve descrizione del parametro
  - Un @param per ogni parametro del metodo/costruttore
- **@return** seguito da una breve descrizione del valore restituito dal metodo

Commentare l'interfaccia pubblica

- **@throws** seguito dal tipo di eccezione lanciata e da una breve descrizione del motivo per cui viene lanciata
  - Un @throws per ogni eccezione lanciata dal metodo/costruttore
- Breve commento, prima della definizione della classe
  - **@author** seguito dal nome dell'autore della classe
  - **@version** seguita dal numero di versione della classe o da una data

Commentare l'interfaccia pubblica

Ecco il commento di documentazione di alcuni metodi della classe **BankAccount**

```
/**
   Versa una quantità di denaro nel conto bancario.

   @param amount la quantità da depositare
   @throws IllegalArgumentException se amount negativo
*/
public void deposit(double amount)
{ ... } // normale corpo del metodo

/**
   Restituisce il saldo del conto bancario.
   @return il saldo
*/
public double getBalance()
{ ... } // normale corpo del metodo
```

Commentare l'interfaccia pubblica

javadoc NomeClasse.java  index.html

```
MacBook-Pro-di-Cinzia:bank cinzia$ javadoc BankAccount.java
Loading source file BankAccount.java...
Constructing Javadoc information...
Standard Doclet version 1.8.0_121
Building tree for all the packages and classes...
Generating ./BankAccount.html...
Generating ./package-frame.html...
Generating ./package-summary.html...
Generating ./package-tree.html...
Generating ./constant-values.html...
Building index for all the packages and classes...
Generating ./overview-tree.html...
Generating ./index-all.html...
Generating ./deprecated-list.html...
Building index for all classes...
Generating ./allclasses-frame.html...
Generating ./allclasses-noframe.html...
Generating ./index.html...
Generating ./help-doc.html...
```

PACKAGE CLASS TREE DEPRECATED INDEX HELP

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class BankAccount

java.lang.Object
    BankAccount

```
public class BankAccount
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor	Description
`BankAccount()`	Crea un conto bancario con saldo iniziale uguale a 0
`BankAccount(double amount)`	Crea un conto bancario con saldo iniziale specificato

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method	Description
void	`deposit(double amount)`	Versa una quantità di denaro nel conto bancario.
double	`getBalance()`	Restituisce il saldo del conto bancario.
void	`withdraw(double amount)`	Preleva una quantità di denaro nel conto bancario.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Details

BankAccount

```
public BankAccount(double amount)
```

Constructor Details

BankAccount

```
public BankAccount(double amount)
```

Crea un conto bancario con saldo iniziale specificato

Parameters:

`amount` - il valore del saldo iniziale

BankAccount

```
public BankAccount()
```

Crea un conto bancario con saldo iniziale uguale a 0

Crea un conto bancario con saldo iniziale uguale a 0

Method Details

getBalance

```
public double getBalance()
```

Restituisce il saldo del conto bancario.

Returns:

il saldo

withdraw

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

Preleva una quantità di denaro nel conto bancario.

Parameters:

`amount` - la quantità da depositare

Throws:

`java.lang.IllegalArgumentException` - se amount maggiore di balance

deposit

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

Versa una quantità di denaro nel conto bancario.

Parameters:

`amount` - la quantità da depositare

Throws:

`java.lang.IllegalArgumentException` - se amount negativo

Take home message

- Per progettare una classe
  - Devo decidere con quali variabili rappresentare lo stato di un oggetto della classe
    - Proteggo tali variabili dichiarandole private
  - Definisco uno o più costruttori per inizializzare lo stato dell'oggetto

Take home message

- Definisco metodi di accesso e metodo modificatori
  - Li definisco public in modo da poterli invocare da altre classi per poter manipolare lo stato dell'oggetto (ma solo secondo le mie scelte progettuali "sicure" e nascoste all'utente)
  - Li invoco con nomeOggetto.nomeMetodo(...)
- Posso definire anche metodi statici (o di classe)
  - per gestire informazioni che non riguardano lo stato dell'oggetto. Li invoco con NomeClasse.nomeMetodo(...)
- Commento opportunamente il codice per poter generare la documentazione dell'interfaccia pubblica della classe