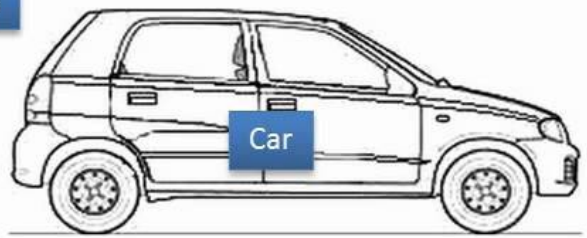
Realizzare classi (Capitolo 3)

What is a Class?

A *class* is the blueprint from which individual objects are created.

Class




```
1  public class Car
2  {
3      private String brand = null;
4      private String model = null;
5      private String color = null;
6
7      public String getBrand()
8      {
9          return brand;
10     }
11
12     public void setBrand(String brand)
13     {
14         this.brand = brand;
15     }
16
17     public String getModel()
18     {
19         return model;
20     }
21
22     public void setModel(String model)
23     {
24         this.model = model;
25     }
26
27     public String getColor()
28     {
29         return color;
30     }
31
32     public void setColor(String color)
33     {
34         this.color = color;
35     }
36
37 }
```

Objects

```
Car maruthiAltok10 = new Car();

maruthiAltok10.setBrand("Maruthi Alto");
maruthiAltok10.setModel("k10");
maruthiAltok10.setColor("Orange");
```

brand = Maruthi Alto
model = K10
color = Orange

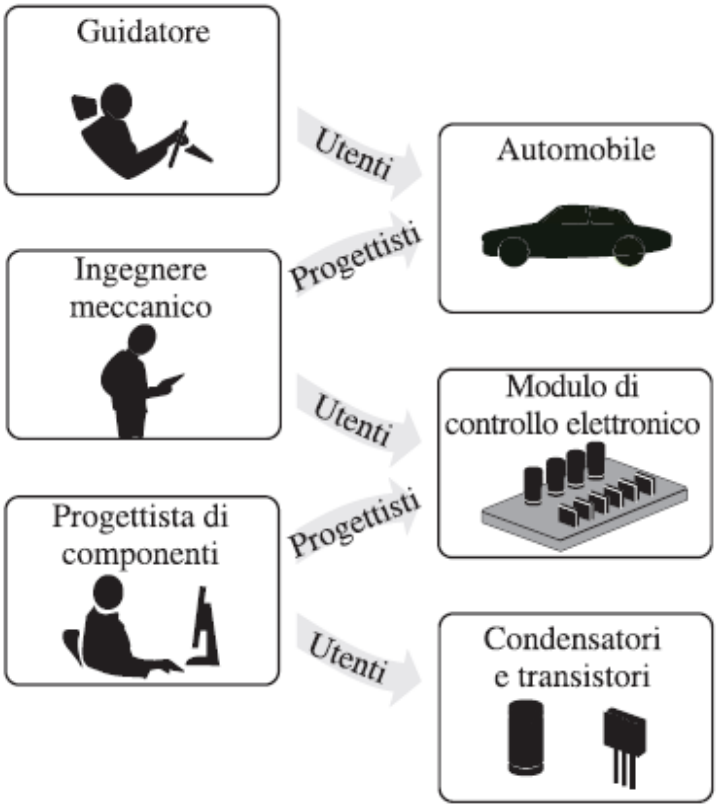
```
Car swift = new Car();

swift.setBrand("Swift");
swift.setModel("ZDI");
swift.setColor("Red");
```

brand = Swift
model = ZDI
color = Red

```
Car maruthiAlto800 = new Car();

maruthiAlto800.setBrand("Maruthi Alto");
maruthiAlto800.setModel("800");
maruthiAlto800.setColor("Blue");
```

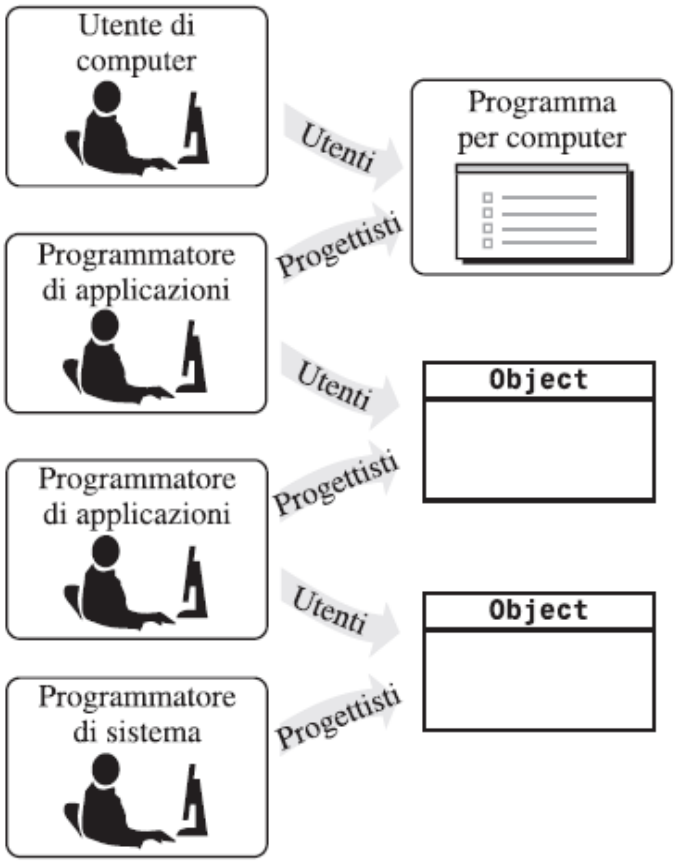
brand = Maruthi Alto
model = 800
color = Blue

Approccio black-box



- Per molti guidatori
  - Un'automobile è una **scatola nera**
  - Non sanno **come funziona**
  - Ma sanno **come usarla**
- Un ingegnere meccanico sa progettare una automobile ma non i moduli di controllo elettronico
  - Non ne conosce il **funzionamento interno**
  - Ma sa **come usarli**
- Scatole nere danno luogo a **incapsulamento**
  - I dettagli non importanti vengono nascosti
  - Ma va identificato il **concetto** che meglio rappresenta una scatola nera: questo comporta **astrazione**

Approccio black-box



- Approccio alla programmazione **orientato agli oggetti**
- Il software viene incapsulato in scatole nere (**oggetti**)

- Diversi livelli di astrazione
  - Un **utente** di computer usa un programma costruito tramite oggetti senza sapere nulla di programmazione
  - Un **programmatore** può usare un oggetto senza conoscerne i dettagli di funzionamento
  - Gli oggetti usati da un programmatore sono programmati da... **altri programmatori**
  - Un oggetto contiene... **altri oggetti**

Il progetto di una classe: BankAccount

Progettare la classe BankAccount

- Vogliamo progettare la classe **BankAccount**, che descriva il comportamento di un conto corrente bancario
- Quali caratteristiche sono essenziali al concetto di conto corrente bancario?
  - Possibilità di **versare** denaro
  - Possibilità di **prelevare** denaro
  - Possibilità di **conoscere** il saldo attuale
- Queste caratteristiche definiscono l'**interfaccia pubblica** della classe

Progettare la classe BankAccount

- Le operazioni consentite dal comportamento di un oggetto si effettuano mediante invocazione di metodi
- Dobbiamo definire tre metodi

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

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

```
public double getBalance();
```

- Tali metodi consentono di
  - depositare denaro nel conto
  - prelevare denaro dal conto
  - conoscere il saldo

```
account.deposit(1000);
```

```
account.withdraw(500);
```

```
double balance = account.getBalance();
```

Definizioni di metodi

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

- La definizione di un metodo inizia sempre con la sua **intestazione** (o **firma**, o **signature**), composta da
  - uno specificatore di accesso
    - In questo caso public, altre volte vedremo private
  - il tipo di dati restituito dal metodo (double, void...)
  - il nome del metodo (deposit, withdraw, getBalance)
  - un elenco di parametri, eventualmente vuoto, racchiuso tra parentesi tonde
    - di ogni parametro si indica il tipo ed il nome
    - più parametri sono separati da una virgola

Definizioni di metodi

- L'intestazione di un metodo è seguita dal **corpo** del metodo stesso, composto da
  - un insieme di **enunciati** che specificano le azioni svolte dal metodo stesso
  - racchiusi tra parentesi graffe

```
public void deposit(double amount)
{
   //corpo del metodo
}
```

```
public double getBalance()
{
   //corpo del metodo
}
```

Metodi di accesso e modificatori

- **Metodo d'accesso**: accede a un oggetto e restituisce informazioni senza modificarlo
  - **getBalance** è metodo di accesso
  - **length** della classe **String** è metodo di accesso
  - **getX, getY, getWidth, getHeight** della classe **Rectangle** sono metodi di accesso
- **Metodo modificatore**: altera lo stato di un oggetto
  - **deposit** e **withdraw** sono metodi modificatori
  - **translate** della classe **Rectangle** è un metodo modificatore

Costruttori

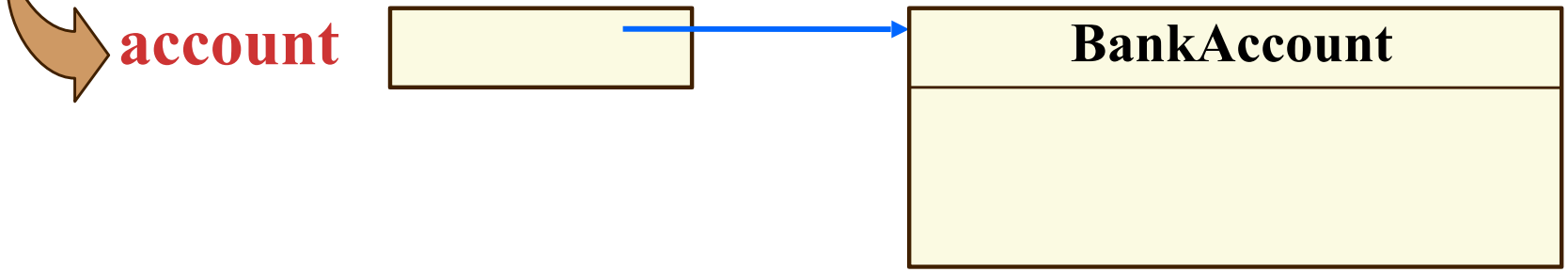
Ripasso: costruire oggetti

- Abbiamo detto che per creare un nuovo oggetto di una classe si usa l'operatore **new** seguito dal nome della classe e da una coppia di parentesi tonde

```
new BankAccount();
```

- L'operatore **new** crea un nuovo oggetto e ne restituisce un riferimento, assegnabile a una variabile oggetto del tipo appropriato

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



I costruttori

- Nella realizzazione della classe **BankAccount** bisogna includere il codice per **creare** un nuovo conto bancario, per esempio con **saldo iniziale a zero**
  - Una scelta di progetto molto ragionevole
- Per consentire la creazione di un nuovo oggetto di una classe, inizializzandone lo stato, dobbiamo scrivere un nuovo metodo, il **costruttore della classe**

```
public BankAccount()
   {
      //corpo del costruttore
   }
```

- ***I costruttori hanno sempre lo stesso nome della classe***

I costruttori

- Sintassi:

```
tipoAccesso NomeClasse(TipoParametro nomeParametro,...)
   {  //realizzazione del costruttore
   }
```

- Lo scopo principale di un costruttore è quello di **inizializzare** un oggetto della classe
- I costruttori, come i metodi, sono solitamente **pubblici**, per consentire a chiunque di creare oggetti della classe
- La sintassi utilizzata per definire i costruttori è molto simile a quella dei metodi, ma
  - il **nome** dei costruttori è uguale a quello della classe
  - i costruttori **non** restituiscono alcun valore e **non bisogna dichiarare** che restituiscono **void**

Molto importante!
Errore frequente!

Invocazione di costruttori

- I costruttori si invocano soltanto con l'operatore new

```
new BankAccount();
```

- L'operatore **new** riserva la memoria per l'oggetto, mentre il costruttore definisce il suo stato iniziale
- Il valore restituito dall'operatore **new** è il riferimento all'oggetto appena creato e inizializzato
  - quasi sempre il valore dell'operatore **new** viene memorizzato in una variabile oggetto

```
BankAccount account = new BankAccount();
// ora account ha saldo zero
```

Una classe con più costruttori

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

```
public BankAccount()
   {  // corpo del costruttore
      // inizializza il saldo a 0
   }
public BankAccount(double initialBalance)
   {  // corpo del costruttore
      // inizializza il saldo a initialBalance
   }
```

Una classe con più costruttori

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

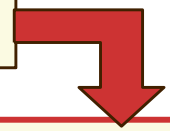
```
BankAccount account = new BankAccount(500);
```

- 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 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 resolve 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 a indicare che viene compiuta la **stessa elaborazione su tipi di dati diversi**

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

Usare la classe BankAccount

- Se la classe **BankAccount** esistesse, saremmo già in grado di **usarla** senza sapere come sia stata realizzata, solo conoscendo la sua **interfaccia pubblica**

```
//aprire un nuovo conto e depositare denaro
double initialDeposit = 1000;
BankAccount account = new BankAccount();
account.deposit(initialDeposit);
System.out.println("Saldo: " + account.getBalance());
```

```
double amount = 500; //fare un bonifico
account1.withdraw(amount);
account2.deposit(amount);
```

```
double rate = 0.05; // accreditare interessi
double amount = account.getBalance() * rate;
account.deposit(amount);
```

- Questa volta, però, la classe **BankAccount** non esiste ancora: la dobbiamo **realizzare** noi

È tutto chiaro? …

1. Come si può svuotare il conto bancario **`account`** usando i metodi dell'interfaccia pubblica della classe?
2. Supponete di voler realizzare una più potente astrazione di conto bancario che tenga traccia anche di un *numero di conto*. Come va modificata l'interfaccia pubblica?

Definire una classe

Definizione di classe

- Sintassi:

```
tipoAccesso class nomeClasse
   {  costruttori (intestazione e corpo)
      metodi (intestazione e corpo)
      variabili (campi) di esemplare
   }
```

- Le variabili di esemplare memorizzano lo stato di un oggetto
  - La classe **BankAccount** deve avere un campo di esemplare che permetta di memorizzare il saldo di un oggetto di tipo **BankAccount**

Definire la classe BankAccount

```
public class BankAccount
{
    //Costruttori
   public BankAccount()
   {  corpo del costruttore }

    public BankAccount(double initialBalance)
    {  corpo del costruttore }

    //Metodi
    public void deposit(double amount)
    {  realizzazione del metodo }

   public void withdraw(double amount)
   {  realizzazione del metodo }

   public double getBalance()
   {  realizzazione del metodo }

    //Variabili di esemplare
    ...
}
```

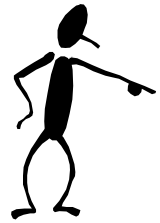
Variabili di esemplare

Lo stato di un oggetto

- Gli oggetti (quasi tutti…) hanno bisogno di **memorizzare il proprio stato attuale**, cioè l'insieme dei valori che
  - **descrivono** l'oggetto e
  - **influenzano** (anche se non necessariamente) il risultato dell'invocazione dei metodi dell'oggetto

- Gli oggetti della classe **BankAccount** hanno bisogno di memorizzare il valore del **saldo** del conto bancario che rappresentano
- Lo stato di un oggetto viene memorizzato mediante **variabili di esemplare** (o "variabili di istanza", instance variables)

Dichiarare variabili di esemplare



- Sintassi:

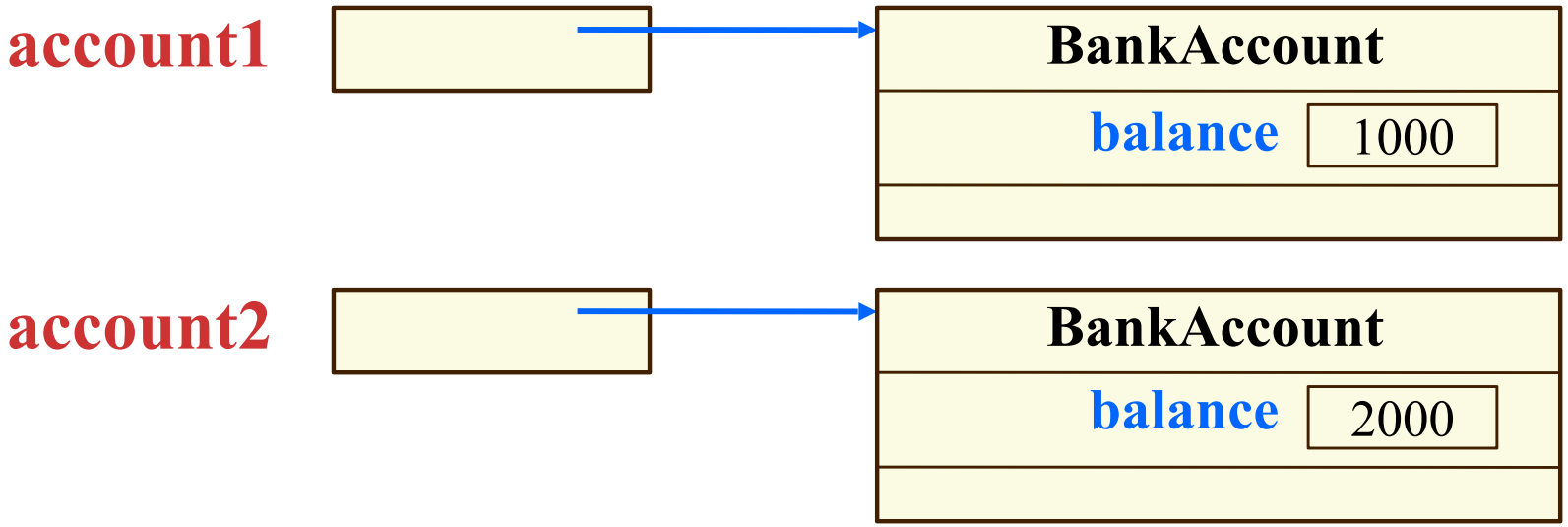
```
public class NomeClasse
{   ...
    tipoDiAccesso TipoVariabile nomeVariabile;
    ...
}
```

- **Scopo**: definire una variabile **nomeVariabile** di tipo **TipoVariabile**, una cui copia sia presente in ogni oggetto della classe **NomeClasse**
- Esempio:

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

Variabili di esemplare

- Ciascun oggetto (“esemplare”) della classe ha una propria copia delle variabili di esemplare



- tra le quali non esiste nessuna relazione: possono essere modificate **indipendentemente** l’una dall’altra

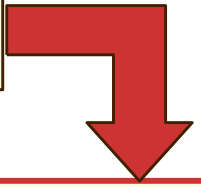
Variabili di esemplare

- Così come i metodi sono di solito “pubblici” (public), le variabili di esemplare sono di solito “private” (private)
- Le variabili di esemplare private possono essere lette o modificate **soltanto** da metodi della classe a cui appartengono
  - le variabili di esemplare private sono nascoste (hidden) al programmatore che utilizza la classe, e possono essere lette o modificate soltanto mediante l’invocazione di metodi pubblici della classe
  - questa caratteristica dei linguaggi di programmazione orientati agli oggetti si chiama **incapsulamento** o information hiding

Incapsulamento

- Poiché la variabile balance di **BankAccount** è private, non vi si può accedere da metodi che non siano della classe (errore **semantico** segnalato dal compilatore)

```
/* codice interno ad un metodo che
   non appartiene a BankAccount */
double b = account.balance; // ERRORE
```



balance has private access in BankAccount

- Si possono usare solo i metodi pubblici!

```
double b = account.getBalance(); // OK
```

Incapsulamento

- L'incapsulamento ha molti **vantaggi**, soltanto pochi dei quali potranno essere evidenziati in questo corso di base
- Il vantaggio fondamentale è quello di **impedire l'accesso incontrollato** allo stato di un oggetto, impedendo così anche che l'oggetto venga (accidentalmente o deliberatamente) posto in uno stato inconsistente
  - Il progettista della classe **BankAccount** potrebbe definire (ragionevolmente) che soltanto un saldo non negativo rappresenti uno stato consistente per un conto bancario

Incapsulamento

- Dato che il valore di **balance** può essere modificato soltanto invocando i metodi **deposit** o **withdraw**, il progettista può impedire che diventi negativo, magari segnalando una **condizione d'errore**
- Se invece fosse possibile assegnare direttamente un valore a balance **dall'esterno**, ogni sforzo del progettista di **BankAccount** sarebbe vano
  - Si noti che, per lo stesso motivo e anche per realismo, **non** esiste un metodo **setBalance**, dato che il saldo di un conto bancario non può essere impostato ad un valore qualsiasi!

È tutto chiaro? …

1. Si supponga di modificare la classe BankAccount in modo che ogni conto bancario abbia anche un numero di conto. Come si modificano i campi di esemplare?
2. Quali sono i campi di esemplare della classe Rectangle?

Realizzare costruttori e metodi

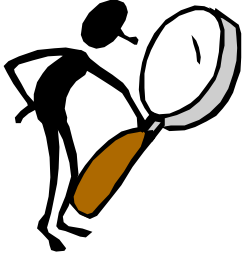
I metodi di BankAccount

- La realizzazione dei costruttori e dei metodi di **BankAccount** è molto semplice
  - lo stato dell'oggetto (il saldo del conto) è memorizzato nella variabile di esemplare **balance**
  - i costruttori devono inizializzare la variabile balance
  - quando si deposita o si preleva una somma di denaro, il saldo del conto si incrementa o si decrementa della somma specificata
  - il metodo **getBalance** restituisce il valore del saldo corrente memorizzato nella variabile **balance**
- Per semplicità, questa realizzazione **non** impedisce che un conto assuma saldo negativo

I costruttori di BankAccount

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

   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }
   ...

   private double balance;
}
```

Il costruttore predefinito



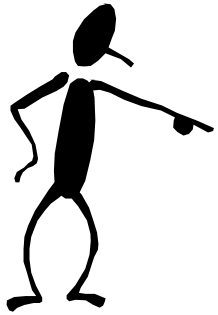
- Cosa succede se non definiamo un costruttore per una classe?
  - il compilatore genera un **costruttore predefinito**
  - (senza alcuna segnalazione d'errore)

- Il costruttore predefinito di una classe
  - è **pubblico** e non richiede parametri
  - **inizializza** tutte le variabili di esemplare
    - a **zero** le variabili di tipo **numerico**
    - a **false** le variabili di tipo **boolean**
    - al valore speciale **null** le variabili **oggetto**, in modo che tali variabili non si riferiscano ad alcun oggetto

I metodi di BankAccount

```
public class BankAccount
{   ...

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

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

   public double getBalance()
   {  return balance;
   }

   private double balance;
}
```

L'enunciato return



- Sintassi:

```
return espressione;
```

```
return;
```

- **Scopo**: terminare l'esecuzione di un metodo, ritornando all'esecuzione sospesa del metodo invocante
  - se è presente una espressione, questa definisce il valore restituito dal metodo e deve essere del tipo dichiarato nella firma del metodo
- Al termine di un metodo con valore restituito di tipo **void**, viene eseguito un **return implicito**
  - il compilatore segnala un **errore** se si termina senza un enunciato return un metodo con un diverso tipo di valore restituito

La classe BankAccount completa

```
public class BankAccount
{  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;
   }

   private double balance;
}
```

È tutto chiaro? …

1. Come è stato realizzato il metodo getWidth() della classe Rectangle?
2. Come è stato realizzato il metodo translate della classe Rectangle?

Parametri espliciti/impliciti

I parametri dei metodi

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

- Cosa succede quando invochiamo il metodo?

```
account.deposit(500);
```

- L'esecuzione del metodo dipende da due valori
  - il **riferimento** all'oggetto account
  - il **valore** 500
- Quando viene eseguito il metodo, il suo **parametro esplicito** **amount** assume il valore 500
  - esplicito perché compare nella firma del metodo
- A quale variabile **balance** si riferisce il metodo?
  - si riferisce alla variabile che appartiene all'oggetto **account** con cui viene invocato il metodo
  - **account** è il **parametro implicito** del metodo

Il riferimento null

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)
- 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

```
BankAccount account = null;
```

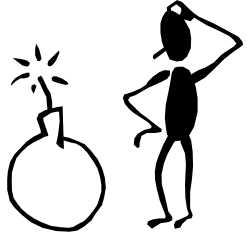
- vedremo in seguito applicazioni utili di questa proprietà
- in questo caso la variabile è comunque inizializzata

Il riferimento null

- Diversamente dai valori numerici, che in Java non sono oggetti, le stringhe sono oggetti
  - una variabile oggetto di tipo **String** può, quindi, contenere un riferimento null

```
String greeting = "Hello";
String emptyString = ""; // stringa vuota
String nullString = null; // riferimento null
int x1 = greeting.length(); // vale 5
int x2 = emptyString.length(); // vale 0
// nel caso seguente l'esecuzione del programma
// termina con un errore
int x3 = nullString.length(); // errore
```

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 eccezione di tipo **NullPointerException** (pointer è un sinonimo di reference, "riferimento")

```
Exception in thread "main"
java.lang.NullPointerException
   at NomeClasse.main(NomeClasse.java:12)
```

Collaudare una classe

Classi di collaudo

- Usiamo la classe **BankAccount** per risolvere un problema specifico
  - apriamo un conto bancario, saldo iniziale 10000 euro
  - sul conto viene accreditato un interesse annuo del 5% del valore del saldo, senza fare prelievi né depositi
  - qual è il saldo del conto dopo due anni?
- **BankAccount** non contiene un metodo **main**
  - Compilando BankAccount.java si ottiene BankAccount.class
  - Ma **non** possiamo eseguire BankAccount.class
- Dobbiamo scrivere una **classe di collaudo** (o di test) che contenga un metodo main nel quale
  - Costruiamo uno o più oggetti della classe da collaudare
  - Invochiamo i metodi della classe per questi oggetti
  - Visualizziamo i valori restituiti

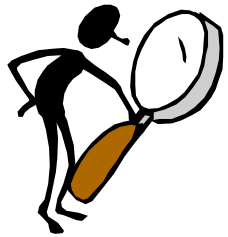
Esempio: utilizzo di BankAccount

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

È tutto chiaro? …

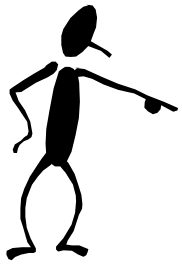
1. Quando eseguo BankAccountTester, quanti oggetti di tipo BankAccount vengono costruiti? E quanti di tipo BankAccountTester?

Un programma con più classi



- Per scrivere semplici programmi **con più classi**, si possono usare due strategie (equivalenti)
  - Scrivere **ciascuna classe in un file diverso**, ciascuno avente il nome della classe con estensione .java
    - Tutti i file vanno tenuti nella stessa cartella
    - Tutti i file vanno di norma compilati separatamente
    - Solo la classe di collaudo (contenente il metodo main) va eseguita
  - Scrivere **tutte le classi in un unico file**
    - un file .java può contenere una sola classe public
    - la classe contenente il metodo main deve essere public
    - le altre non devono essere public (non serve scrivere private, semplicemente non si indica l'attributo public)
    - il file .java deve avere il nome della classe public

Riassunto: progettare una classe



1. **Capire** cosa deve fare un oggetto della classe
   - Elenco in linguaggio naturale delle operazioni possibili
2. **Specificare** l'interfaccia pubblica
   - Ovvero, definire i metodi tramite le loro intestazioni
3. **Documentare** l'interfaccia pubblica
4. **Identificare** i **campi di esemplare** a partire dalle intestazioni dei metodi
5. **Realizzare** costruttori e metodi
   - Se avete problemi a realizzare un metodo forse dovete riesaminare i passi precedenti
6. **Collaudare** la classe con un programma di collaudo

Materiale di complemento (capitolo 3)

Categorie e cicli di vita delle variabili

- Per ora saltiamo questa sezione (3.6)
- Ne parliamo più avanti

Parametri impliciti e il riferimento this

- Per ora saltiamo questa sezione (3.7)
- Ne parliamo più avanti

Commenti di documentazione

Commentare l'interfaccia pubblica

- I commenti ai metodi sono **importantissimi** per rendere il codice comprensibile a voi e agli altri!
- Java ha **delimitatori speciali** per commenti di documentazione

```
/**
   Preleva denaro dal conto
   @param amount importo da prelevare
*/
public void withdraw(double amount)
{
    //corpo del metodo
}
```

- `@param nomeparametro` per descrivere un parametro esplicito
- `@return` per descrivere il valore

```
/**
   Ispeziona saldo attuale
   @return saldo attuale
*/
public double getBalance()
{
    //corpo del metodo
}
```

Commentare l'interfaccia pubblica

- Inserire brevi commenti anche alla classe, per illustrarne lo scopo

```
/**
   Un conto bancario ha un saldo
   modificabile tramite depositi e prelievi
*/
public class BankAccount
{
   ...
}
```

- Usando commenti di documentazione in questo formato si può generare in maniera automatica documentazione in html

```
javadoc NomeClasse.java
```

  - Genera un documento **NomeClasse.html** ben formattato e con collegamenti ipertestuali, contenente i commenti a **NomeClasse**

Class BankAccount

java.lang.Object
    BankAccount

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

Un conto bancario che ha un saldo modificabile tramite depositi e prelievi

Constructor Summary

Constructors

Constructor	Description
`BankAccount()`	
`BankAccount(double initialBalance)`	

Method Summary

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

Modifier and Type	Method	Description
`void`	`deposit(double amount)`	Deposita denaro sul conto
`double`	`getBalance()`	Ispeziona saldo attuale
`void`	`withdraw(double amount)`	Preleva denaro dal conto

Methods inherited from class java.lang.Object

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

Constructor Details

BankAccount

```
public BankAccount()
```

BankAccount

```
public BankAccount(double initialBalance)
```

Method Details

deposit

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

Deposita denaro sul conto

Parameters:
amount - importo da depositare

withdraw

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

Preleva denaro dal conto

Parameters:
amount - importo da prelevare

getBalance

```
public double getBalance()
```

Ispeziona saldo attuale

Returns:
saldo attuale

Decisioni (capitolo 5)

L'enunciato if

- Il programma precedente consente di prelevare tutto il denaro che si vuole
  - il saldo balance può diventare **negativo**

```
balance = balance - amount;
```

- È una situazione assai poco realistica!
- Il programma deve **controllare** il saldo e agire di conseguenza, consentendo il prelievo oppure no

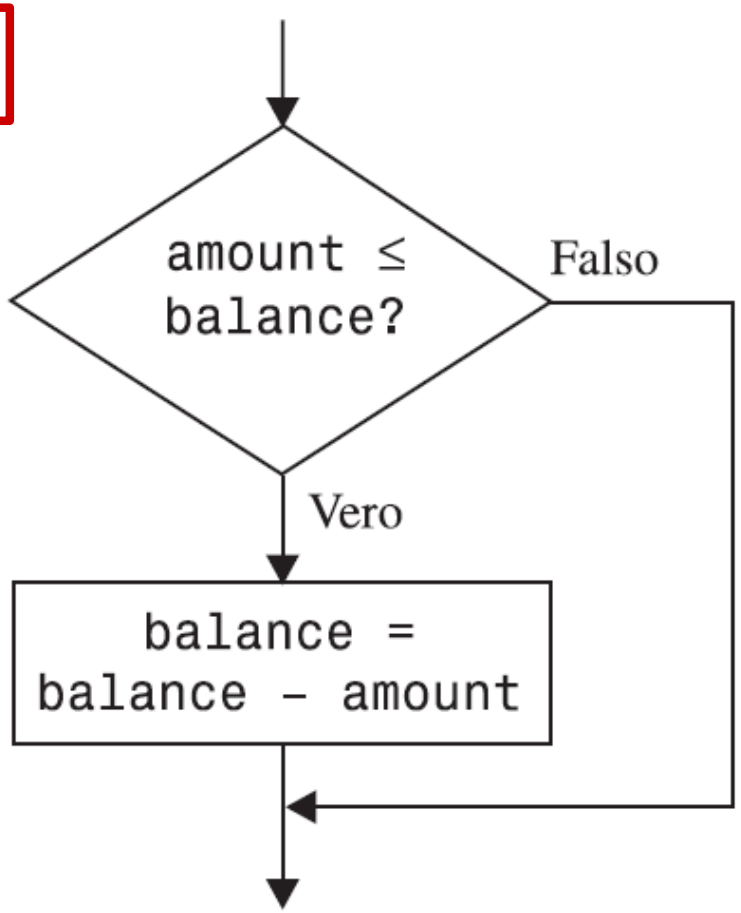
L'enunciato if

```
if (amount <= balance)
   balance = balance - amount;
```

verifica

corpo

- L'enunciato if si usa per realizzare una decisione ed è diviso in due parti
  - una **verifica**
  - un **corpo**
- Il corpo viene eseguito **se e solo se** la verifica ha successo



Tipi di enunciato in Java

- Enunciato **semplice**

```
balance = balance - amount;
```

- Enunciato **composto**

```
if (x >= 0) x=0;
```

- **Blocco** di enunciati

```
{ zero o più enunciati di qualsiasi tipo }
```

Un nuovo problema

- Proviamo ora a emettere un messaggio d'errore in caso di prelievo non consentito

```
if (amount <= balance)
   balance = balance - amount;
if (amount > balance)
   System.out.println("Conto scoperto");
```

- **Primo problema**: se si modifica la prima verifica, bisogna ricordarsi di modificare anche la seconda (es. viene concesso un fido sul conto, che può "andare in rosso")
- **Secondo problema**: se il corpo del primo **if** viene eseguito, la verifica del secondo if usa il nuovo valore di balance, introducendo un **errore logico**
  - quando si preleva più della metà del saldo disponibile

La clausola else

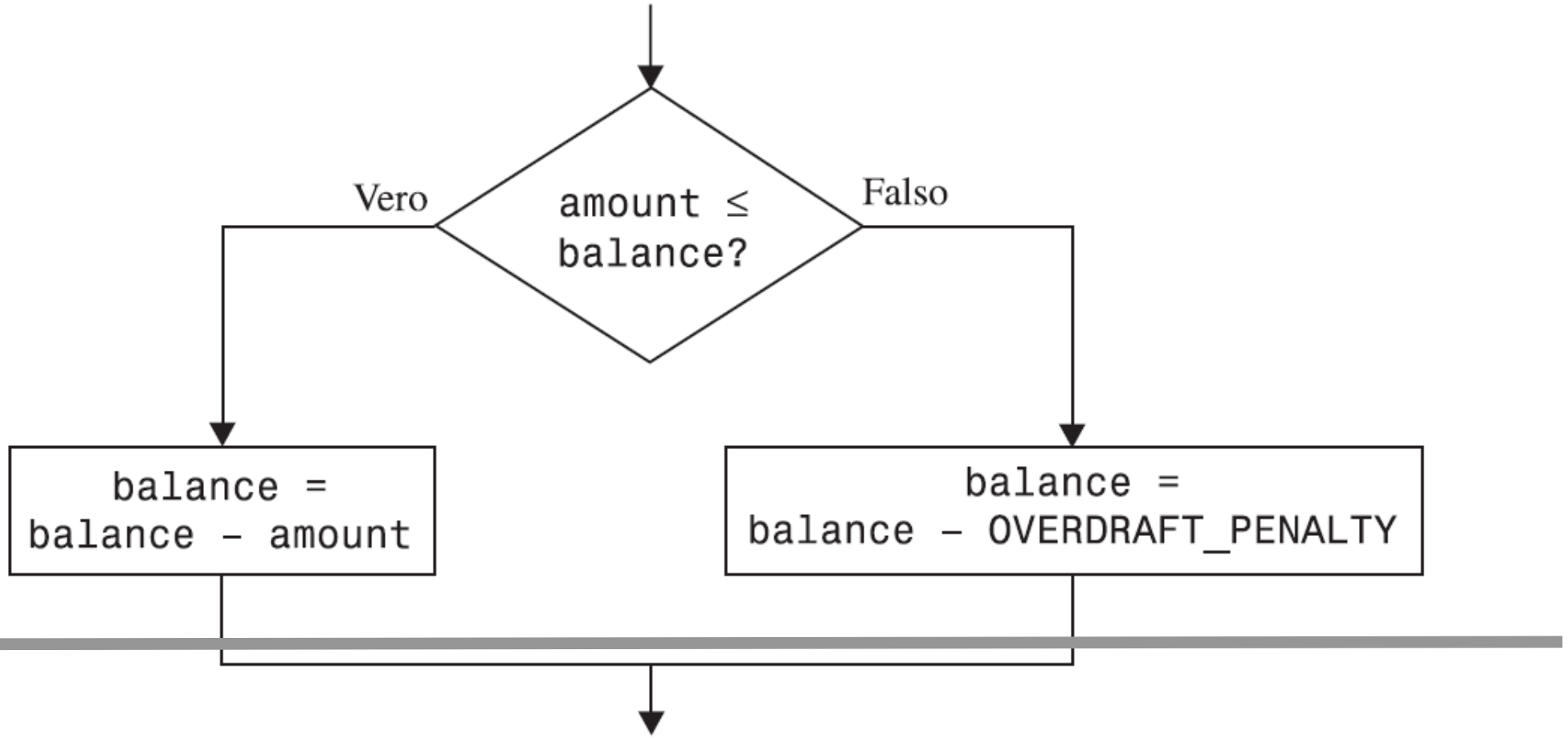
- Per realizzare un'alternativa, si utilizza la clausola **else** dell'enunciato if

```
if (amount <= balance)
   balance = balance - amount;
else
   System.out.println("Conto scoperto");
```

- **Vantaggio**: ora c'è una sola verifica
  - se la verifica ha successo, viene eseguito il **primo** corpo dell'enunciato **if/else**
  - altrimenti, viene eseguito il **secondo** corpo

La clausola else

```
if (amount <= balance)
   balance = balance - amount;
else
   { System.out.println("Conto scoperto");
     balance = balance - OVERDRAFT_PENALTY;
   }
```

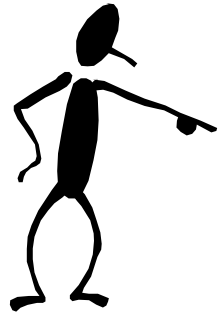


L'enunciato if

- Sintassi:

```
if (condizione)
    enunciato1
```

```
if (condizione)
    enunciato1
else
    enunciato2
```



- Scopo:
  - eseguire **enunciato1** **se e solo se** la condizione è vera;
  - se è presente la clausola **else**, eseguire **enunciato2** **se e solo se** la condizione è falsa
- Spesso il corpo di un enunciato **if** è costituito da più enunciati da eseguire in sequenza; racchiudendo tali enunciati tra una coppia di parentesi graffe { } si crea un **blocco di enunciati**, che può essere usato come corpo

```
if (amount <= balance)
{   balance = balance - amount;
    System.out.println("Prelievo accordato");
}
```

È tutto chiaro? …

1. Perché nell'esempio precedente abbiamo usato la condizione amount <= balance e non la condizione amount < balance?
2. Qual è l'errore logico nell'enunciato seguente, e come lo si può correggere?

```
if (amount <= balance)
   newBalance = balance - amount;
   balance = newBalance;
```

Confrontare valori numerici

Confrontare valori

- Le condizioni dell'enunciato if sono molto spesso dei confronti tra due valori

```
if (x >= 0)
```

- Gli operatori di confronto si chiamano **operatori relazionali**

Attenzione:

negli operatori costituiti da due caratteri ***non*** vanno inseriti spazi intermedi

>	Maggiore
>=	Maggiore o uguale
<	Minore
<=	Minore o uguale
==	Uguale
!=	Diverso

Operatori relazionali

- Fare **molta** attenzione alla differenza tra l'operatore relazionale == e l'operatore di assegnazione =

```
a = 5;              // assegna 5 ad a

if (a == 5)         // esegue enunciato
   enunciato        // se a è uguale a 5
```

Confronto di numeri in virgola mobile

Confrontare numeri in virgola mobile

- I numeri in virgola mobile hanno una precisione limitata e i calcoli possono introdurre **errori** di **arrotondamento** e **troncamento**
- Tali errori sono inevitabili e bisogna fare molta attenzione nella formulazione di verifiche che coinvolgono numeri in virgola mobile

```
double r = Math.sqrt(2);
double x = r * r;
if (x == 2)
   System.out.println("OK");
else
   System.out.println("Non ci credevi?");
```

Confrontare numeri in virgola mobile

- Affinché gli errori di arrotondamento non influenzino la logica del programma, i confronti tra numeri in virgola mobile devono prevedere una **tolleranza**

  - Verifica di uguaglianza tra x e y (di tipo double):

    $| x - y | < \varepsilon$ con $\varepsilon = 1E\text{-}14$

  - Scelta migliore se x,y sono molto grandi o molto piccoli

    $| x - y | < \varepsilon \max(|x|,|y|)$ con $\varepsilon = 1E\text{-}14$

(questo valore di ε è ragionevole per numeri double)

- Il codice per questa verifica è

```
final double EPSILON = 1E-14;
if ( Math.abs(x - y) <=
            EPSILON*Math.max(Math.abs(x),Math.abs(y)) )
   ...
```

Confrontare numeri in virgola mobile

- Possiamo definire un metodo statico che verifichi l'uguaglianza con tolleranza

```
public class Numeric
{  public static boolean approxEqual(double x, double y)
   {  final double EPSILON = 1E-14;
      double xyMax = Math.max(Math.abs(x),Math.abs(y));
      return Math.abs(x - y) <= EPSILON * xyMax;
   }
}
```

```
double r = Math.sqrt(2);
if (Numeric.approxEqual(r * r, 2))
   System.out.println("Tutto bene...");
else
   System.out.println("Questo non succede...");
```