

**Lezione X**  
**Lu 22-Ott-2007**

**Introduzione  
a classi e oggetti  
(continua)**

# Esempio: Usare la classe BankAccount

□ Abbiamo a disposizione una classe che descrive il *comportamento* di un *conto bancario*



□ Tale comportamento consente di

▪ *depositare* denaro nel conto

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

▪ *prelevare* denaro dal conto

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

▪ *conoscere* il valore del saldo del conto

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

□ Le operazioni consentite dal comportamento di un oggetto si effettuano mediante l'*invocazione di metodi*

Variabile riferimento

metodo

# Esempio di metodi d'accesso e modificatori



- *getBalance( )* è un *metodo di accesso*: restituisce informazioni su uno specifico oggetto, in questo caso il saldo, senza modificarlo

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

- *deposit( )* e *withdraw( )* sono *metodi modificatori*, perché cambiano lo stato dell'oggetto, in questo caso il saldo

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

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

# Usare la classe `BankAccount`

- ❑ Trasferiamo denaro da un conto ad un altro

```
double amount = 500;  
account1.withdraw(amount);  
account2.deposit(amount);
```

- ❑ Calcoliamo e accredtiamo gli interessi di un conto

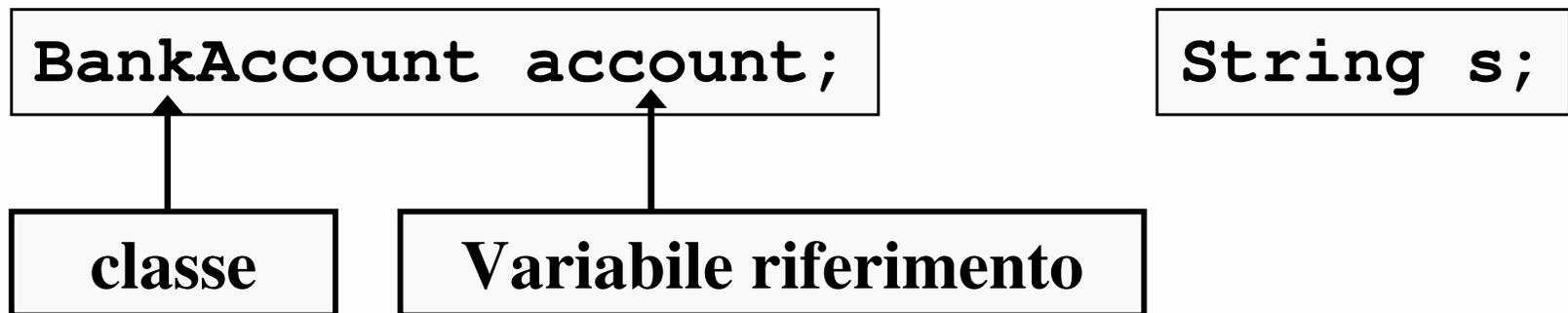
```
double rate = 0.05; // interessi del 5%  
double amount = account.getBalance() * rate;  
account.deposit(amount);
```

# Costruire oggetti

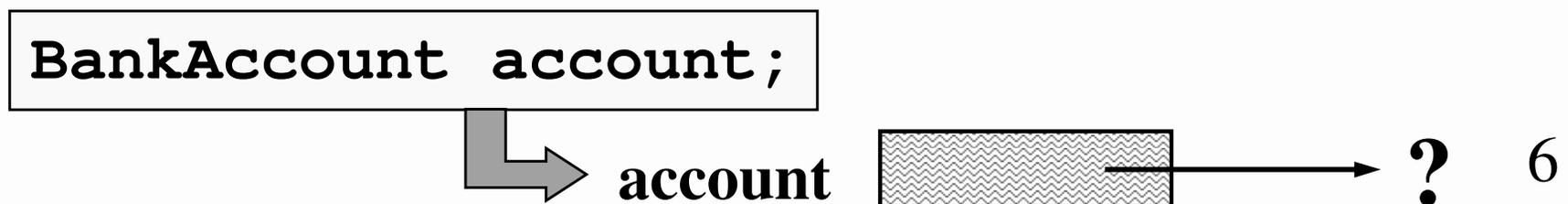
- ❑ Sappiamo quindi come *operare* su un conto bancario ma non sappiamo come “**aprire un nuovo conto bancario**” cioè *creare un nuovo oggetto* di tipo **BankAccount**
- ❑ Se vogliamo creare un oggetto e usarlo più volte abbiamo bisogno di *memorizzarlo* da qualche parte nel programma
- ❑ Per conservare un oggetto si usa una *variabile oggetto* che conserva non l’oggetto stesso ma informazioni sulla sua posizione nella memoria del computer
  - si dice che è un *riferimento* o *puntatore*

# Le variabili oggetto

- ❑ Per definire una variabile oggetto si indica il nome della classe ai cui oggetti farà riferimento la variabile seguito dal nome della variabile stessa



- ❑ In Java *ciascuna variabile oggetto è di un tipo specifico* e potrà fare riferimento *soltanto* a oggetti di quel tipo
- ❑ La definizione di una variabile oggetto crea un riferimento *non inizializzato* cioè la variabile non fa riferimento ad alcun oggetto



# Costruire oggetti BankAccount

- Come già sappiamo, 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* che può essere assegnato ad

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

**account**

0xffff4567

**BankAccount**

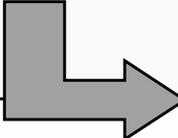
# Costruire oggetti **BankAccount**

- ❑ Che caratteristiche ha l'oggetto appena creato?
  - *qual è il saldo del nuovo conto bancario?*
- ❑ Le proprietà di un oggetto appena creato dipendono da come è realizzata la classe. Quindi la descrizione di tali proprietà deve far parte delle informazioni fornite all'utilizzatore della classe: la *documentazione* della classe
- ❑ Nel caso della classe **BankAccount** un oggetto appena creato dovrà avere un *saldo di valore zero* senza dubbio una scelta di progetto molto ragionevole

# La classe **BankAccount**

- *Senza sapere come sia stata realizzata la classe **BankAccount** siamo in grado di aprire un nuovo conto bancario e di depositarvi un po' di denaro*

```
double initialDeposit = 1000;  
BankAccount account = new BankAccount ();  
System.out.println("Saldo: "  
    + account.getBalance());  
account.deposit(initialDeposit);  
System.out.println("Saldo: "  
    + account.getBalance());
```



```
Saldo: 0  
Saldo: 1000
```

# Definire i metodi

# Il progetto di BankAccount

- ❑ Sapendo già il *comportamento* della classe **BankAccount** il suo progetto consiste nella *definizione della classe*
- ❑ Per definire una classe occorre *realizzarne i metodi*
  - **deposit()**
  - **withdraw()**
  - **getBalance()**

```
public class BankAccount
{
    public void deposit(double amount)
    {
        realizzazione del metodo
    }
    public void withdraw(double amount)
    {
        realizzazione del metodo
    }
    public double getBalance()
    {
        realizzazione del metodo
    }

    dati della classe
}
```

# Le intestazioni dei metodi

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

- La *definizione di un metodo* inizia sempre con la sua *intestazione (firma, 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 e il nome
    - più parametri sono separati da una virgola



# Lo specificatore di accesso

- Lo *specificatore di accesso* di un metodo indica *quali altri metodi possono invocare il metodo*
- Dichiarando un metodo **public** si consente l'accesso a *qualsiasi altro metodo di qualsiasi altra classe*
  - è comodo per programmi semplici e *faremo sempre così* salvo casi eccezionali

# Il tipo di dati restituito

- La dichiarazione di un metodo specifica quale sia il tipo di dati restituito dal metodo al termine della sua invocazione

- ad esempio `getBalance()` restituisce un valore di tipo **double**

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

- Se un metodo *non restituisce alcun valore* si dichiara che restituisce il tipo speciale **void** (assente, non valido...)

```
double b = account.deposit(500); // ERRORE  
account.deposit(500);           // OK
```

# 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 che 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*)

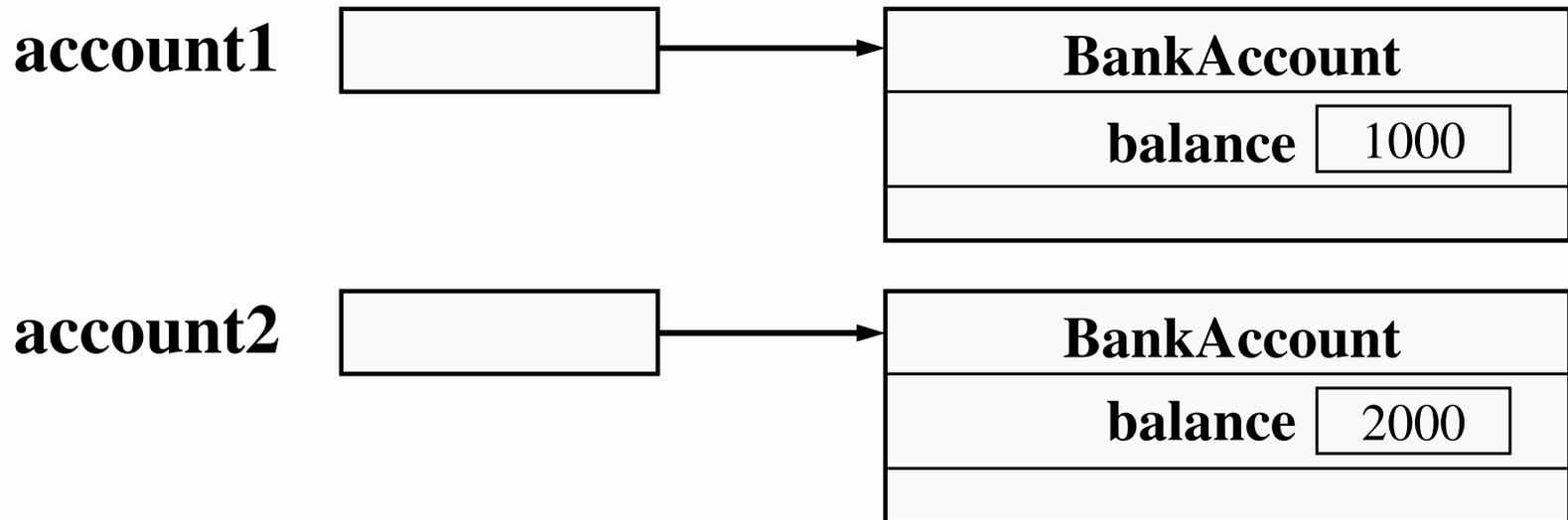
# Variabili di esemplare

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

- ❑ La *dichiarazione* di una *variabile di esemplare* è costituita da
  - uno specificatore di accesso
    - in questo caso **private** altre volte vedremo **public**
  - il tipo di dati della variabile (**double**)
  - il nome della variabile (**balance**)

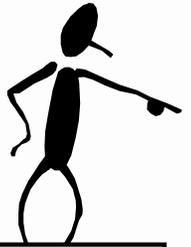
# Variabili di esemplare

- *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

# Dichiarazione di variabili di esemplare



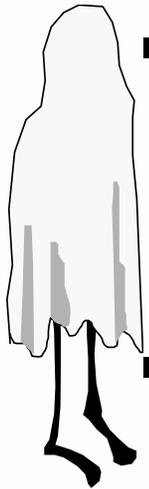
- Sintassi:

```
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*

# 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 a 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*!

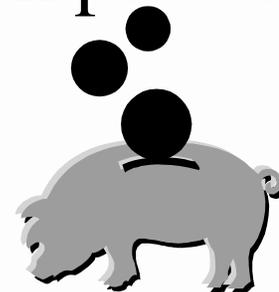
```
double b = account.getBalance(); // OK
```

# 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 a un valore qualsiasi!



# **Il progetto di BankAccount**

# Il progetto di BankAccount

- ❑ Sapendo già il *comportamento* della classe **BankAccount**, il suo progetto consiste nella *definizione della classe*
- ❑ Per definire una classe occorre *realizzarne i metodi*
  - **deposit()**
  - **withdraw()**
  - **getBalance()**

```
public class BankAccount
{
    private double balance;

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

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

    public double getBalance()
    { realizzazione del metodo
    }

}
```

# I metodi di **BankAccount**

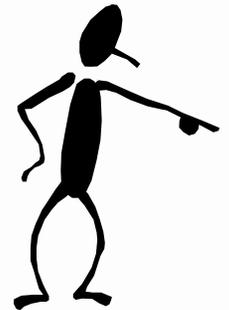
- La realizzazione dei metodi di **BankAccount** è molto semplice
  - lo stato dell'oggetto (il saldo del conto) è memorizzato nella variabile di esemplare **balance**
  - quando si deposita o si preleva una somma di denaro, il saldo del conto si incrementa o si decrementa della stessa somma
  - il metodo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

# I metodi di BankAccount

```
public class BankAccount
{
    private double balance;

    public void deposit(double amount)
    {
        balance = balance + amount;
    }
    public void withdraw(double amount)
    {
        balance = balance - amount;
    }
    public double getBalance()
    {
        return 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 tipo di valore restituito diverso da void

# 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*

# I parametri dei metodi

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

- Nel metodo vengono utilizzate due variabili
  - **amount** è il *parametro esplicito* del metodo
  - **balance** si riferisce alla *variabile di esemplare balance* della classe **BankAccount**, ma sappiamo che di tale variabile esiste *una copia per ogni oggetto*
- Alla variabile **balance di quale oggetto** si riferisce il metodo?
  - si riferisce alla variabile che appartiene all'*oggetto con cui viene invocato il metodo*, ma come fa?

# Invocazione di metodi

- ❑ Se il metodo restituisce un dato, questo è generalmente memorizzato in una variabile dello stesso tipo del dato restituito
- ❑ L'invocazione del metodo è quindi usata, tipicamente, in un enunciato di definizione e inizializzazione

```
double myBalance = myAccount.getBalance();
```

- ❑ o di assegnazione

```
double myBalance;  
...  
myBalance = myAccount.getBalance();
```

- ❑ Se il metodo non ritorna dati, l'invocazione è come segue.:

```
myAccount.deposit(1000);
```

# Il parametro implicito dei metodi

- All'interno di ciascun metodo, il riferimento all'oggetto con il quale è eseguito il metodo si chiama *parametro implicito* e si indica con la parola chiave **this**
  - in questo caso, **this** assume il valore di **account** all'interno del metodo **deposit**

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

- Ogni metodo ha sempre uno e un solo parametro implicito, dello stesso tipo della classe a cui appartiene il metodo
  - eccezione: *i metodi statici non hanno parametro implicito*
- Il parametro implicito *non deve essere dichiarato* e si chiama sempre **this**

# Uso del parametro implicito

□ La *vera sintassi* del metodo dovrebbe essere

```
public void deposit(double amount)
{
    this.balance = this.balance + amount;
}
// this è di tipo BankAccount
```

ma *nessun programmatore Java scriverebbe così*,  
perché Java consente una *comoda scorciatoia*

- quando in un metodo ci si riferisce ad una variabile di esemplare, il compilatore costruisce *automaticamente* un riferimento alla variabile di esemplare dell'oggetto rappresentato dal parametro implicito **this**

# Costruttori

# I costruttori

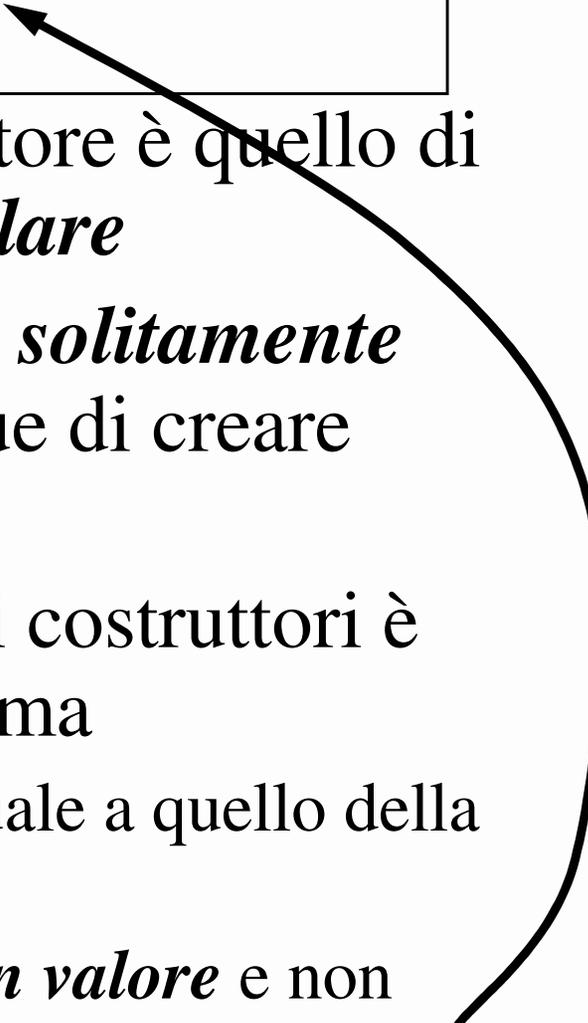
- ❑ La classe **BankAccount** è stata realizzata quasi completamente, manca il codice per *creare un nuovo conto bancario*, con saldo a zero
- ❑ Per consentire la creazione di un nuovo oggetto di una classe, *inizializzandone lo stato*, dobbiamo scrivere un *costruttore* per la classe

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

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

# I costruttori

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



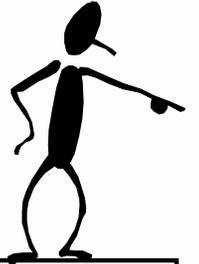
- ❑ Lo *scopo principale* di un costruttore è quello di *inizializzare le variabili di esemplare*
- ❑ I *costruttori*, come i metodi, sono *solitamente pubblici*, per consentire a chiunque di creare oggetti della classe
- ❑ La sintassi utilizzata per definire i costruttori è molto simile a quella dei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

# 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
```

# Definizione di costruttori



## □ Sintassi:

```
public class NomeClasse
{
    ...
    tipoDiAccesso NomeClasse
        (TipoParametro1 nomeParametro1, ...)
    { realizzazione del costruttore
    }
    ...
}
```

- Scopo: definire il comportamento di un costruttore della classe *NomeClasse*
- Nota: i costruttori servono a inizializzare le variabili di esemplare di oggetti appena creati

# 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 costante del linguaggio

```
BankAccount account = null;
```

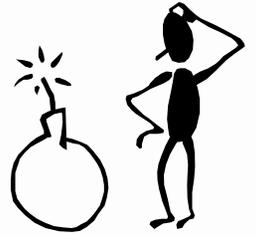
- vedremo in seguito applicazioni utili di questa proprietà
- in questo caso la variabile è comunque inizializzata

# 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 errore di tipo **NullPointerException** (*pointer* è un sinonimo di *reference*, “riferimento”)

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

    public BankAccount()
    {
        balance = 0;
    }

    ...
}
```

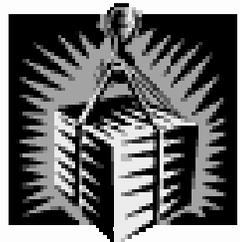


# Una classe con più costruttori

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

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

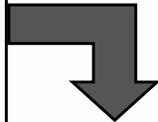
- ❑ Notiamo che, se esistono più costruttori in una classe, *hanno tutti lo stesso nome*, perché devono comunque avere lo stesso nome della classe
  - questo fenomeno (*più metodi o costruttori con lo stesso nome*) è detto *sovraccarico* del nome (*overloading*)
  - *il compilatore* decide quale costruttore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



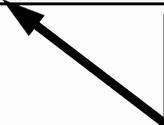
# Una classe con più costruttori

- ❑ 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
```



# 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*
    - al *valore speciale null* le variabili oggetto, in modo che tali variabili non si riferiscano ad alcun oggetto

# La classe BankAccount

```
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;
}
```

# Esempio: utilizzo di BankAccount

- Scriviamo un programma che *usi* la classe **BankAccount** per risolvere un problema specifico
  - viene aperto un conto bancario, con un saldo 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 BankAccountTest
{
    public static void main(String[] args)
    {
        final double RATE = 0.05;
        final double INITIAL_BALANCE = 10000.;

        BankAccount acct = new BankAccount
            (INITIAL_BALANCE);

        // calcola gli interessi dopo il primo anno
        double interest = acct.getBalance() * RATE;

        // somma gli interessi dopo il primo anno
        acct.deposit(interest);

        System.out.println("Saldo dopo un anno: "
            + acct.getBalance() + " euro");
        ...
    }
}
```

# Esempio: utilizzo di BankAccount

```
public class BankAccountTest
{
    public static void main(String[] args)
    {
        ...
        // calcola gli interessi dopo il secondo anno
        interest = acct.getBalance() * RATE;

        // somma gli interessi dopo il secondo anno
        acct.deposit(interest);

        System.out.println("Saldo dopo due anni: "
            + acct.getBalance() + " euro");
    }
}
```

# **Commentare l'interfaccia pubblica**

# Commentare l'interfaccia pubblica

- ❑ In java esiste un formato standard per i commenti di documentazione
- ❑ Se si usa il formato standard si può poi utilizzare il programma *javadoc* che genera automaticamente documentazione in formato HTML
- ❑ Formato standard:
  - Inizio commento: `/**` delimitatore speciale
  - Fine commento: `*/`
- ❑ Per ciascun metodo si scrive un commento per descrivere lo scopo del metodo

# Commentare l'interfaccia pubblica

- ❑ Ecco il commento di documentazione del costruttore della classe BankAccount

```
/**  
    versa una cifra di danaro nel conto bancario  
  
    @param amount la cifra da depositare  
*/  
public void deposit(double amount)  
  
/**  
    preleva una cifra di danaro dal conto  
    bancario  
  
    @param amount la cifra da prelevare  
  
*/  
public void withdraw(double amount)
```

# Commentare l'interfaccia pubblica

- ❑ @param seguito dal nome del parametro e da una breve descrizione del parametro
- ❑ @return seguito da una breve descrizione del valore restituito
- ❑ In genere oltre a commentare ciascun metodo, si scrive anche un breve commento per la classe
- ❑ @author seguito dal nome dell'autore
- ❑ @version seguita dal numero di versione della classe o da una data

# Commentare l'interfaccia pubblica

- ❑ Se si sono usate le convenzioni per la documentazione standard, si può generare la documentazione della classe in modo automatico, lanciando da riga di comando il programma di utilità **javadoc** seguito dal nome del file **NomeClasse.java**

***\$javadoc NomeClasse.java***

- ❑ Questo produce un insieme di file in formato HTML

```
>javadoc BankAccount.java
>Loading source file BankAccount.java...
>Constructing Javadoc information...
>Building tree for all the packages and classes...
>Building index for all the packages and classes...
>Generating overview-tree.html...
>Generating index-all.html...
>...
```

**Lezione XI**  
**Ma 23-Ott-2007**

**Iterazioni e decisioni**

# Problema

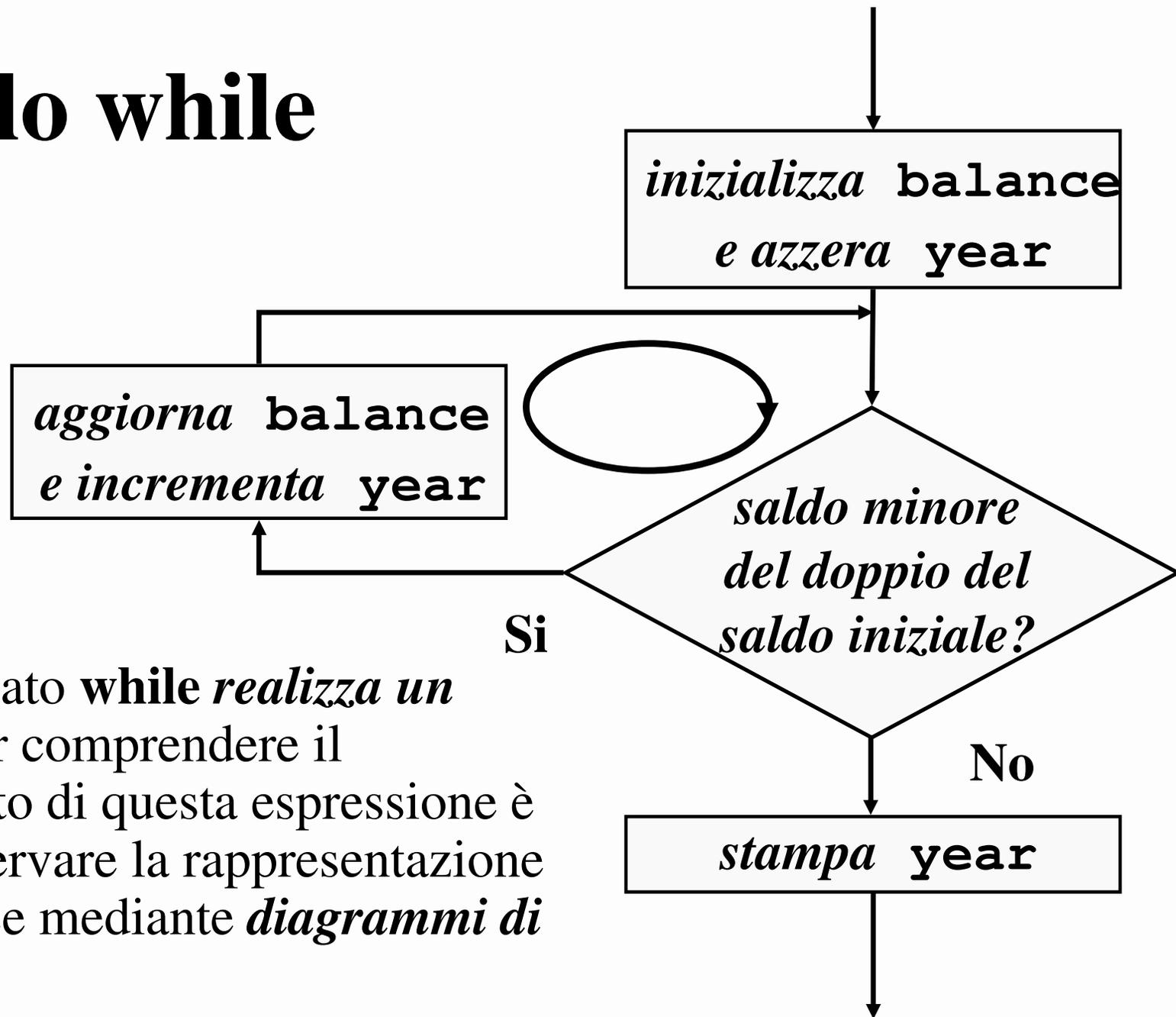
- Riprendiamo un problema visto in precedenza per il quale *abbiamo individuato un algoritmo* senza averlo ancora realizzato

**Problema:** Avendo depositato ventimila euro in un conto bancario che produce il 5% di interessi all'anno capitalizzati annualmente, quanti anni occorrono affinché il saldo del conto arrivi al doppio della cifra iniziale?

# Algoritmo che risolve il problema

- 1 All'anno 0 il saldo è 20000
  - 2 **Ripetere** i passi 3 e 4 **finché** il saldo è minore del doppio di 20000 poi passare al punto 5
    - 3 Aggiungere 1 al valore dell'anno corrente
    - 4 Il nuovo saldo è il valore del saldo precedente moltiplicato per 1.05 (cioè aggiungiamo il 5%)
  - 5 Il risultato è il valore dell'anno corrente
- L'enunciato **while** consente la realizzazione di programmi che devono *eseguire ripetutamente una serie di azioni finché è verificata una condizione*

# Il ciclo while



L'enunciato **while** *realizza un ciclo*: per comprendere il significato di questa espressione è utile osservare la rappresentazione del codice mediante *diagrammi di flusso*

# Codice Java

```
public class DoubleInvestment
{
    public static void main(String[] args)
    {
        final double INITIAL_BALANCE = 20000;
        final double RATE = 0.05;
        double balance = INITIAL_BALANCE;
        int year = 0;
        while (balance < 2 * INITIAL_BALANCE)
        {
            year++;

            double interest = balance * RATE;
            balance = balance + interest;
        }
        System.out.println("L'investimento "
            + "raddoppia in " + year
            + " anni");
    }
}
```

# Tipi di enunciati in Java

## □ Enunciato semplice

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

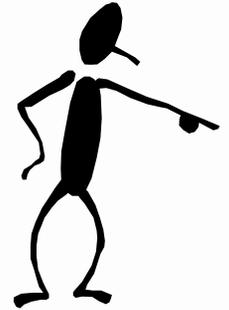
## □ Enunciato composto

```
while (x >= 0) x--;
```

## □ Blocco di enunciati

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

# L'enunciato while



□ Sintassi:

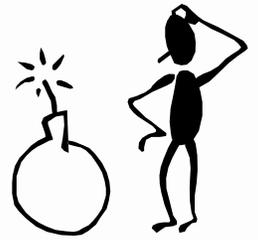
```
while (condizione)  
    enunciato
```

□ Scopo:

eseguire un *enunciato* finché la *condizione* è vera

□ Nota: il *corpo* del ciclo **while** può essere un enunciato qualsiasi quindi anche un *blocco di enunciati*

# Cicli infiniti



- Esistono errori logici che *impediscono la terminazione di un ciclo* generando un *ciclo infinito*
  - **l'esecuzione del programma continua ininterrottamente**
- Bisogna arrestare il programma con un comando del sistema operativo oppure addirittura riavviare il computer

```
int year = 0;
while (year < 20)
{
    double interest = balance * rate;
    balance = balance + interest;
    // qui manca year++;
}
```

# Confrontare valori numerici

# Confrontare valori numerici

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

```
while (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  
  
while (a == 5)  // esegue enunciato  
    enunciato   // finché a è uguale a 5
```

# Espressioni booleane

# Il tipo di dati *booleano*

- Ogni *espressione* ha un *valore*
  - $x + 10$  espressione aritmetica valore numerico
  - $x < 10$  espressione relazionale valore *booleano*
- Un'espressione relazionale ha un valore *vero* o *falso* : in Java **true** o **false** (in algebra **1** o **0**)
- I valori **true** e **false** non sono numeri né oggetti né classi: appartengono a un tipo di dati diverso detto *booleano* dal nome del matematico George Boole (1815-1864) pioniere della logica
  - è un *tipo fondamentale* in Java come quelli numerici

# Le variabili booleane

- ❑ Il tipo di dati **boolean** come tutti gli altri tipi di dati consente la definizione di *variabili*
- ❑ A volte è comodo utilizzare variabili booleane per memorizzare valori di passaggi intermedi in cui è opportuno scomporre verifiche troppo complesse
- ❑ Altre volte l'uso di una variabile booleana rende più leggibile il codice
- ❑ Spesso le variabili booleane vengono chiamate *flags* (bandiere) perché possono assumere soltanto due valori cioè trovarsi in due soli stati possibili: su e giù come una bandiera

# Definizione e assegnazione

## □ Definizione e inizializzazione

```
boolean a = true;
```

```
int x;  
...  
boolean a = (x > 0) && (x < 20);
```

## □ Assegnazione

```
boolean a = true;  
...  
a = (x > 0) && (x < 20);
```

# Operatori booleani

# Gli operatori booleani o logici

- ❑ Gli *operatori* booleani o logici servono a svolgere *operazioni* su valori booleani

```
while (x > 10 && x < 20)
// esegue finché x è maggiore di 10
// e minore di 20
```

- ❑ L'operatore **&&** (*and*) combina due o più condizioni in una sola che risulta *vera se e solo se sono tutte vere*
- ❑ L'operatore **||** (*or* oppure) combina due o più condizioni in una sola che risulta *vera se e solo se almeno una è vera*
- ❑ L'operatore **!** (*not*) *inverte* il valore di un'espressione booleana

# Gli operatori booleani o logici

- Più operatori booleani possono essere usati in un'unica espressione

```
while ( (x > 10 && x < 20) || x > 30 )
```

- La valutazione di un'espressione con operatori booleani viene effettuata con una strategia detta “cortocircuito”

*la valutazione dell'espressione termina appena è possibile decidere il risultato*

nel caso precedente se **x** vale **15** l'ultima condizione non viene valutata perché sicuramente l'intera espressione è **vera**

# Gli operatori booleani o logici

Tabelle di verita'

<i>a</i>	<i>b</i>	<i>a &amp;&amp; b</i>
true	true	true
true	false	false
false	<i>qualsiasi</i>	false

<i>a</i>	<i>b</i>	<i>a    b</i>
true	<i>qualsiasi</i>	true
false	true	true
false	false	false

<i>a</i>	<i>!a</i>
true	false
false	true

**cortocircuito**

# Leggi di De Morgan

In linguaggio Java

**!(a && b)** equivale a **!a || !b**

□ Dimostrazione: tabelle di verita'

<i>a</i>	<i>b</i>	<i>a &amp;&amp; b</i>	<i>!(a &amp;&amp; b)</i>
false	false	false	true
false	true	false	true
true	false	false	true
true	true	true	false

<i>a</i>	<i>b</i>	<i>!a</i>	<i>!b</i>	<i>!a    !b</i>
false	false	true	true	true
false	true	true	false	true
true	false	false	true	true
true	true	false	false	false

# Leggi di De Morgan

In notazione del linguaggio Java

**!(a || b) equivale a !a && !b**

□ Dimostrazione: tabelle di verita'

<i>a</i>	<i>b</i>	<i>a    b</i>	<i>!(a    b)</i>
false	false	false	true
false	true	true	false
true	false	true	false
true	true	true	false

<i>a</i>	<i>b</i>	<i>!a</i>	<i>!b</i>	<i>!a &amp;&amp; !b</i>
false	false	true	true	true
false	true	true	false	false
true	false	false	true	false
true	true	false	false	false

# Gli operatori booleani o logici

- ❑ In un'espressione booleana con più operatori la valutazione viene fatta da *sinistra a destra* dando la precedenza all'operatore *not* poi all'operatore *and* infine all'operatore *or*
- ❑ L'ordine di valutazione può comunque essere alterato dalle parentesi tonde

```
while (!(x < 0 || x > 10))  
// esegue finché x è compreso tra 0 e 10  
// estremi inclusi [0,10]
```

```
while (!x < 0 || x > 10)  
// esegue finché x è maggiore o uguale a 0
```

# Esempio

Tre modi equivalenti di scrivere un'espressione logica sfruttando la seconda legge di De Morgan

```
boolean a = !(x < 0 || x > 10); // x ∈ [0,10]
```

```
boolean a = !(x < 0) && !(x > 10); //x ∈ [0,10]
```

```
boolean a = x >= 0 && x <= 10; // x ∈ [0,10]
```

# Decisioni

# Gestione di un conto corrente

```
import java.util.Scanner;
public class UnlimitedWithdrawal
{
    public static void main(String[] args)
    {
        double init = 10000; // saldo iniziale
        BankAccount account = new BankAccount(init);

        System.out.println("Quanto vuoi prelevare?");

        Scanner in = new Scanner(System.in);
        double amount = in.nextDouble(); //<- 100000

        account.withdraw(amount);

        double balance = account.getBalance();

        System.out.println("Saldo: " + balance);
    }
}
```



Saldo: -90000

# L'enunciato if

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

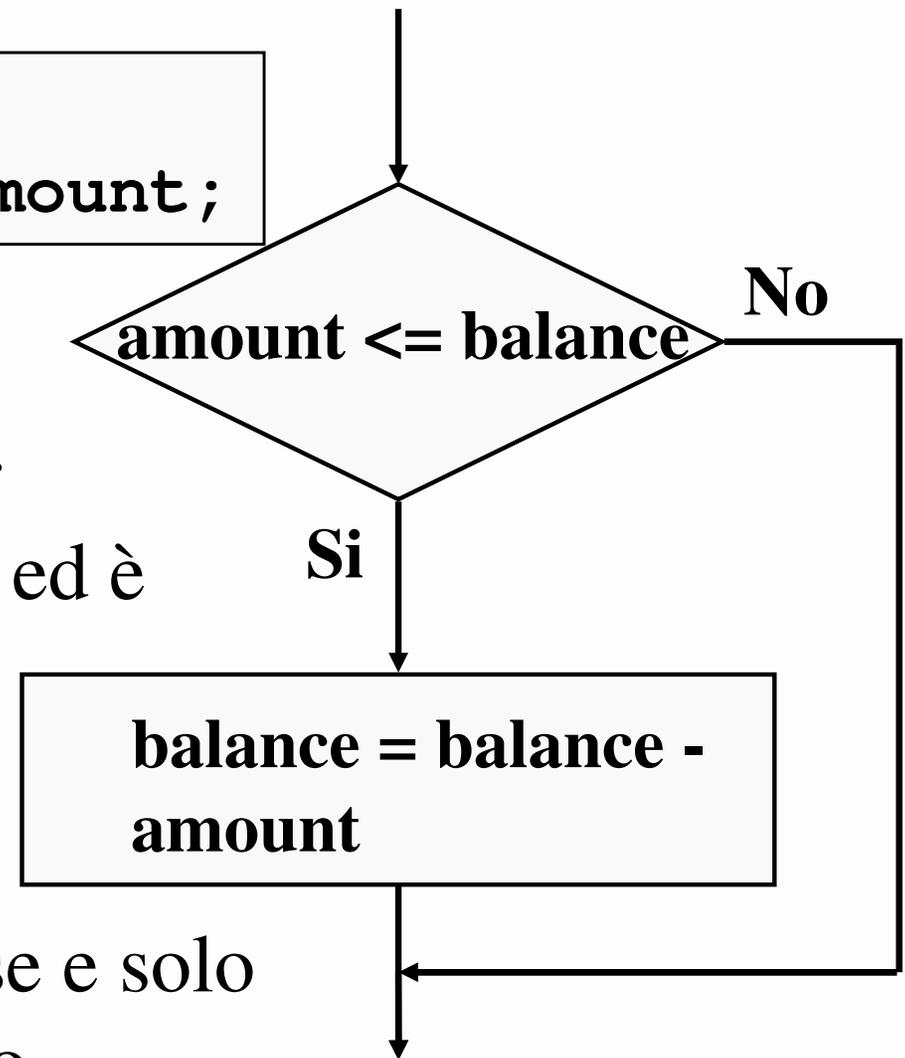
```
public void withdraw(double amount)
{
    balance = balance - amount;
}
```

- ❑ 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;
```

- 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



# 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");
```

- ❑ **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
- ❑ **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”)

# La clausola else

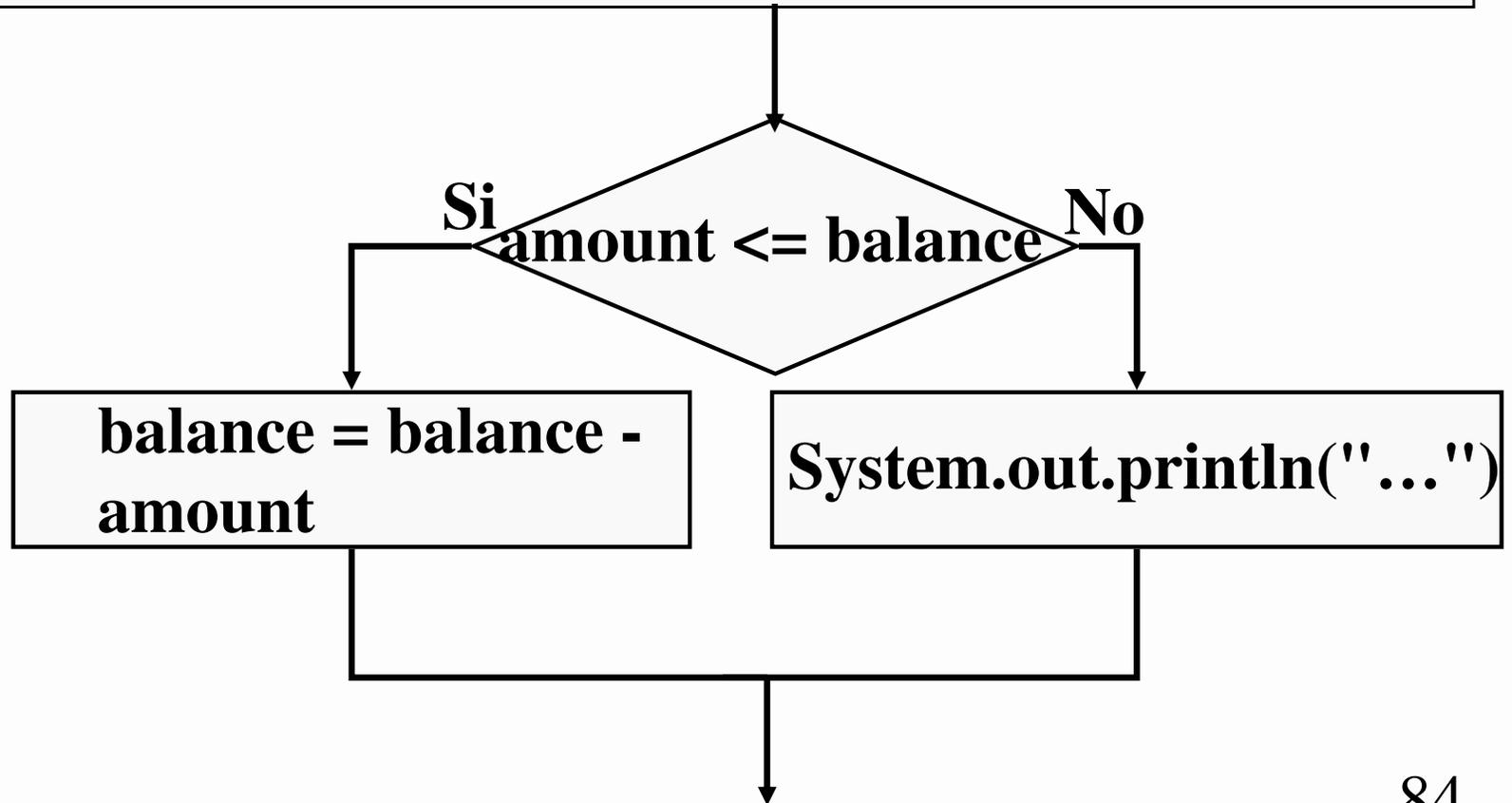
- ❑ Per realizzare un'*alternativa* si utilizza la clausola **else** dell'enunciato **if**

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

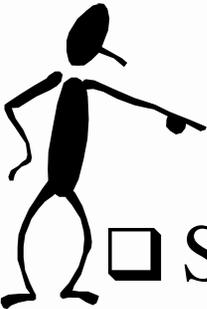
- ❑ **Vantaggio:** ora c'è *una sola verifica*
  - se la verifica ha successo viene eseguito il *primo* corpo dell'enunciato **if/else**
  - *altrimenti* viene eseguito il *secondo* corpo
- ❑ Non è un costrutto sintattico *necessario* ma è utile

# La clausola else

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



# 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");
}
```

# Alternative multiple

# Sequenza di confronti

- Se si hanno più di due alternative si usa una sequenza di confronti

```
if (richter >= 8)
    System.out.println("Terremoto molto forte");
else if (richter >= 6)
    System.out.println("Terremoto forte");
else if (richter >= 4)
    System.out.println("Terremoto medio");
else if (richter >= 2)
    System.out.println("Terremoto debole");
else
    System.out.println("Terremoto molto debole");
```

# Sequenza di confronti

- Se si hanno più di due alternative si usa una sequenza di confronti

```
if (richter >= 8)
    System.out.println("...");
else
    if (richter >= 6)
        System.out.println("...");
    else
        if (richter >= 4)
            System.out.println("...");
        else
            if (richter >= 2)
                System.out.println("...");
            else
                System.out.println("...");
```

Corretto, ma  
non si usa  
questo stile

# Sequenza di confronti

- ❑ Se si fanno confronti di tipo “**maggiore di**” si devono scrivere *prima i valori più alti* e viceversa

```
if (richter >= 0)        // NON FUNZIONA!  
    System.out.println("Terremoto molto debole");  
else if (richter >= 2)  
    System.out.println("Terremoto debole");  
else if (richter >= 4)  
    System.out.println("Terremoto medio");  
else if (richter >= 6)  
    System.out.println("Terremoto forte");  
else  
    System.out.println("Terremoto molto forte");
```

- ❑ Questo non funziona perché stampa **Terremoto molto debole** per qualsiasi valore di **richter**

# Sequenza di confronti

```
if (richter >= 8)      // NON FUNZIONA!  
    System.out.println("Terremoto molto forte");  
if (richter >= 6)  
    System.out.println("Terremoto forte");  
if (richter >= 4)  
    System.out.println("Terremoto medio");  
if (richter >= 2)  
    System.out.println("Terremoto debole");  
if (richter >= 0)  
    System.out.println("Terremoto molto debole");
```

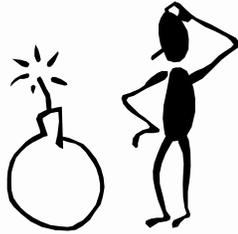
- ❑ Se non si rendono mutuamente esclusive le alternative mediante l'uso della clausole **else**, non funziona: se **richter** vale **3** stampa sia **Terremoto debole** sia **Terremoto molto debole**

# **Lezione XII**

## **Me 24-Ott-2007**

**Consigli Utili**  
**Diramazioni annidate**  
**Di nuovo iterazioni**

# Errori con operatori relazionali



- Alcune espressioni “*naturali*” con operatori relazionali sono **errate** ma per fortuna il compilatore **le rifiuta**

```
if (0 <= x <= 1)          // NON FUNZIONA!
```

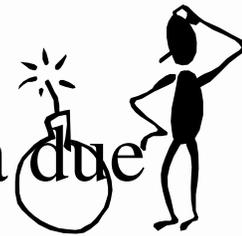
```
if (x >= 0 && x <= 1)    // OK
```

```
if (x && y > 0)          // NON FUNZIONA!
```

```
if (x > 0 && y > 0)     // OK
```

- Perché il compilatore le rifiuta?

# Errori con operatori relazionali



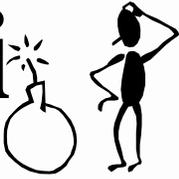
- Il compilatore analizza l'espressione logica e trova due operatori di confronto quindi esegue il primo da sinistra e decide che il risultato sarà un valore booleano

```
if (0 <= x <= 1) x++; // NON FUNZIONA!
```

- Successivamente si trova a dover applicare il secondo operatore relazionale a due operandi il primo dei quali è di tipo **boolean** mentre il secondo è di tipo **int**

```
operator <= cannot be applied to boolean
int
if (0 <= x <= 1) x++;
           ^
1 error
```

# Errori con operatori relazionali



- Il compilatore analizza l'espressione logica e trova un operatore di confronto (che ha la precedenza sull'operatore booleano **&&** ) il cui risultato sarà un valore di tipo **boolean**

```
if (x && y > 0) x++; // NON FUNZIONA!
```

- Successivamente si trova ad applicare l'operatore booleano **&&** a due operandi il primo dei quali è di tipo **int** mentre il secondo è di tipo **boolean**

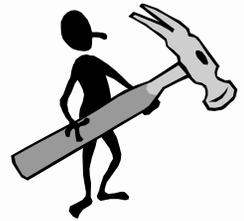
```
operator && cannot be applied to int boolean
```

```
if (x && y > 0) x++;
```

```
      ^
```

```
1 error
```

# Rientri e Tabulazioni



- ❑ Decidere il numero ideale di caratteri bianchi da usare per ogni livello di rientro è molto arduo
- ❑ In questo corso consigliamo di usare **tre** caratteri

```
if (amount <= balance)
{
    balance = balance - amount;
    if (amount > 20000000)
    {
        System.out.println("Esagerato!");
    }
}
```

- ❑ Consigliamo anche di *non usare* i “caratteri di tabulazione” che di solito generano un rientro di otto caratteri eccessivo

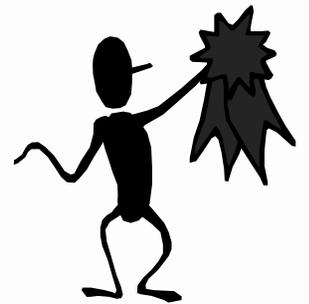
# Disposizione delle graffe

- ❑ Incolonnare le parentesi graffe
- ❑ Eventualmente lasciare su una riga da sola anche la graffa aperta
- ❑ Così invece è più difficile trovare la coppia!

```
if (...)
{
    ...;
    ...;
}
```

```
if (...)
{
    ...;
    ...;
}
```

```
if (...) {
    ...;
    ...;
}
```



# Diramazioni annidate

# Diramazioni annidate

□ Soluzione dell'equazione di secondo grado

$$ax^2 + bx + c = 0 \Rightarrow \begin{cases} x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} & \text{se } b^2 - 4ac \geq 0 \\ x_{1,2} = \frac{-b \pm i\sqrt{b^2 - 4ac}}{2a} & \text{se } b^2 - 4ac < 0 \end{cases}$$

□ Per calcolare le due soluzioni bisogna prima calcolare il valore del discriminante  $b^2 - 4ac$

□ Se il discriminante è nullo vogliamo anche segnalare che le due soluzioni sono coincidenti anche se la formula da usare è sempre la prima

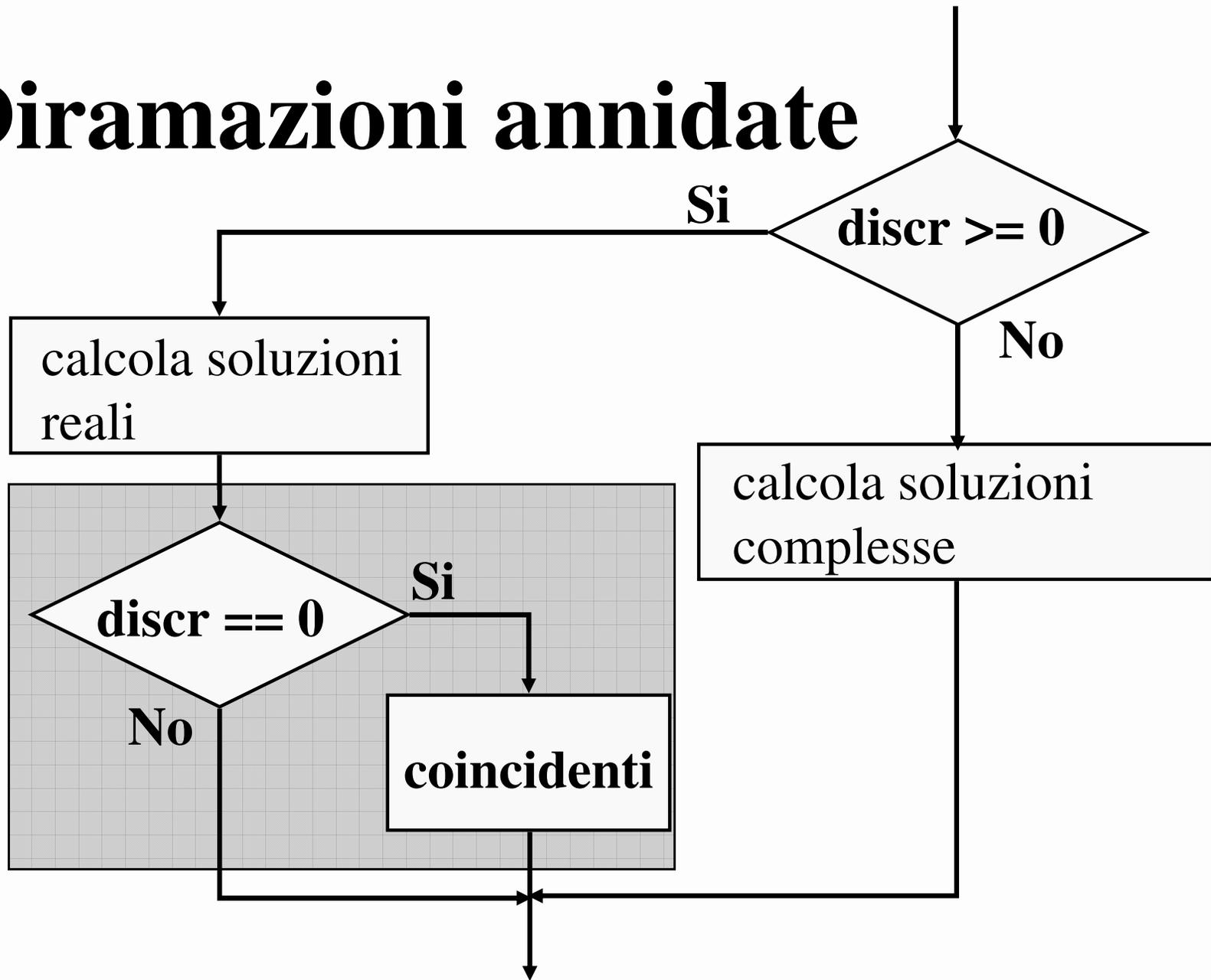
# Diramazioni annidate

```
int discr = b * b - 4 * a * c; // discriminante

if (discr >= 0)
{
    ... // calcola le due soluzioni reali
    if (discr == 0)
        System.out.println("Coincidenti");
}
else
{
    ... // calcola le due soluzioni complesse
}
```

- Per risolvere il problema usiamo due *diramazioni annidate*: un enunciato **if** all'interno del corpo di un altro enunciato **if**; cosa perfettamente lecita perché il corpo di un **if** può essere un enunciato composto cioè un altro **if**

# Diramazioni annidate



# Confronto fra numeri in virgola mobile

# Confronto di 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?");
```

# Confronto di numeri in virgola mobile

- ❑ Per fare in modo che gli errori di arrotondamento non influenzino la logica del programma i confronti tra numeri in virgola mobile devono prevedere una *tolleranza*



$|x - y| \leq \varepsilon$  // non sufficientemente preciso

$|x - y| \leq \varepsilon \cdot \max(|x|, |y|)$  con  $\varepsilon = 10^{-14}$

(questo valore di  $\varepsilon$  è ragionevole per **double**)

- ❑ Il codice per questa verifica di *uguaglianza con tolleranza* è

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

# Confronto fra stringhe

# Confronto di stringhe

- ❑ Per confrontare stringhe si usa il metodo `equals()`

```
if (s1.equals(s2))
```

- ❑ Per confrontare stringhe ignorando la differenza tra maiuscole e minuscole si usa

```
if (s1.equalsIgnoreCase(s2))
```

- ❑ *Non usare mai l'operatore di uguaglianza per confrontare stringhe!* Usare sempre `equals()`

- unica eccezione: `null`

```
if (s1 == null)
```

- ❑ Se si usa l'operatore uguaglianza il successo del confronto sembra essere deciso in maniera "casuale" in realtà dipende da come è stata progettata la Java Virtual Machine e da come sono state costruite le due stringhe

```
if (s1 == s2) // non funziona correttamente!
```



# Ordinamento lessicografico

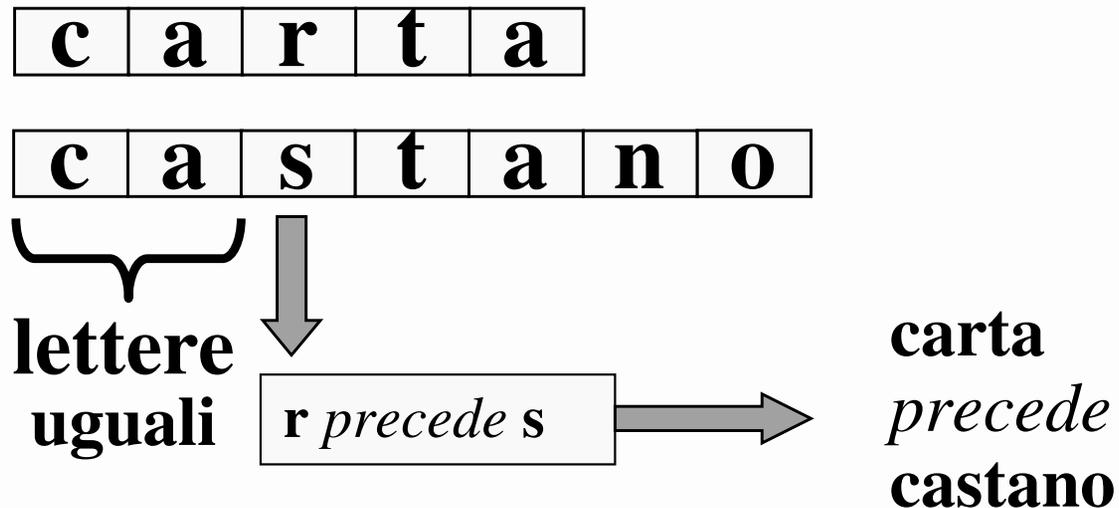
- ❑ Se due stringhe sono diverse è possibile conoscere la relazione che intercorre tra loro secondo l'*ordinamento lessicografico* simile al comune ordinamento alfabetico
- ❑ Il confronto lessicografico tra stringhe si esegue con il metodo **compareTo()**

```
if (s1.compareTo(s2) < 0)
```

- ❑ Il metodo **int compareTo(String s2)** restituisce un valore **int**
  - *negativo* se **s1 precede s2** nell'ordinamento
  - *positivo* se **s1 segue s2** nell'ordinamento
  - *zero* se **s1 e s2** sono *identiche*

# Confronto lessicografico

- Partendo dall'inizio delle stringhe si confrontano a due a due i caratteri in posizioni corrispondenti finché una delle stringhe termina oppure due caratteri sono diversi
  - se una stringa termina essa precede l'altra
    - se terminano entrambe sono uguali
  - altrimenti l'ordinamento tra le due stringhe è uguale all'ordinamento alfabetico tra i due caratteri diversi

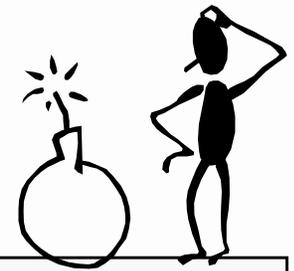


# Confronto lessicografico

- ❑ Il confronto lessicografico genera un ordinamento *simile* a quello di un comune dizionario
- ❑ L'ordinamento tra caratteri in Java è *simile* all'ordinamento alfabetico comune con qualche differenza... anche perché tra i caratteri non ci sono solo le lettere! Ad esempio
  - i numeri precedono le lettere
  - tutte le lettere maiuscole precedono tutte le lettere minuscole
  - il carattere di “spazio bianco” precede tutti gli altri caratteri
- ❑ L'ordinamento lessicografico è definito dallo standard **Unicode** <http://www.unicode.org>

# **Il problema dell'else sospeso**

# Il problema dell'else sospeso

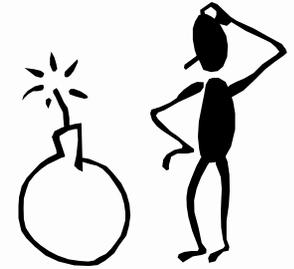


```
double cost = 5; // prezzo per USA
if (country.equals("USA"))
    if (state.equals("HI"))
        cost = 10; // Hawaii più costoso
else // NON FUNZIONA!
    cost = 20; // estero ancora più costoso
```

- ❑ I livelli di rientro suggeriscono che la clausola **else** debba riferirsi al primo enunciato **if**
  - **ma il compilatore ignora i rientri!**
- ❑ La regola sintattica è che una clausola **else** appartiene sempre all'enunciato **if più vicino**

# Il problema dell'else sospeso

- L'esempio precedente svolge in realtà la funzione evidenziata dai rientri seguenti

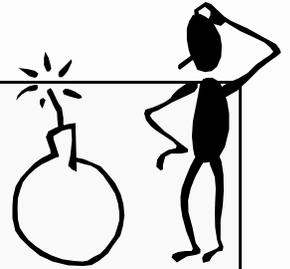


```
double cost = 5; // prezzo per estero
if (country.equals("USA"))
    if (state.equals("HI"))
        cost = 10; // Hawaii più costoso
    else
        cost = 20; // USA ancora più costoso
```

- Il risultato è che gli stati esteri ottengono il prezzo più basso e gli USA continentali il più alto! Il contrario di ciò che si voleva...

# Il problema dell'else sospeso

```
double cost = 5; // prezzo per USA
if (country.equals("USA"))
{
    if (state.equals("HI"))
        cost = 10; // Hawaii più costoso
}
else
    cost = 20; // estero ancora più
                // costoso
```



- Per ottenere il risultato voluto bisogna “nascondere” il secondo enunciato **if** all’interno di un blocco di enunciati inserendo una coppia di parentesi graffe
  - per evitare problemi con l’*else sospeso* è meglio *racchiudere sempre il corpo di un enunciato if tra parentesi graffe* anche quando sono inutili

# Ciclo for

# Ciclo for

- Molti cicli hanno la forma

```
int i = inizio;  
while (i < fine)  
{ enunciati  
  i++;  
}
```

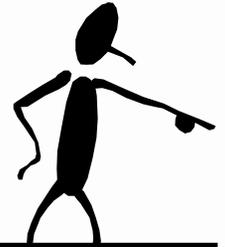
- Per comodità esiste il ciclo *for equivalente*

```
for (int i = inizio; i < fine; i++)  
{ enunciati  
}
```

- Non è necessario che l'incremento sia di una sola unità, né che sia positivo né che sia intero

```
for (double x = 2; x > 0; x = x - 0.3)  
{ enunciati  
}
```

# L'enunciato for



- Sintassi:

```
for (inizializzazione; condizione;  
    aggiornamento)  
    enunciato
```

- Scopo: eseguire un'*inizializzazione* poi ripetere l'esecuzione di un *enunciato* ed effettuare un *aggiornamento* finché la *condizione* è vera
- Nota: l'*inizializzazione* può contenere la *definizione di una variabile* che sarà *visibile soltanto all'interno del corpo del ciclo*

```
for (int y = 1; y <= 10; y++)  
{  
    ...  
}  
// qui y non è più definita
```

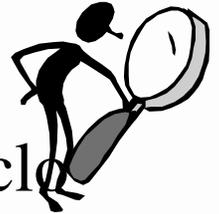
# Tipico uso del ciclo for

- Si usa per realizzare cicli che hanno la seguente struttura:

```
for (imposta contatore all'inizio; verifica se il
contatore e' alla fine; aggiorna il contatore)
{ ...
  // proibito modificare il contatore!!!
}
```

```
//Esempio: invertire i caratteri di una stringa
String s = in.next(); // <- "ABCD"
String r = ""; // Stringa vuota
for (int i = 0; i < s.length(); i++)
{
  // elaborazione
  r = s.charAt(i) + r;
}
System.out.println(r); // -> "DCBA"
```

# Visibilità delle variabili



- ❑ Se il valore finale di una variabile di controllo del ciclo deve essere visibile *al di fuori del corpo del ciclo* bisogna definirla *prima del ciclo*
- ❑ Poiché una variabile definita nell'inizializzazione di un ciclo *non è più definita* dopo il corpo del ciclo è possibile (e comodo) usare di nuovo lo stesso nome in altri cicli

```
double b = ...;
for (int i = 1; (i < 10) && (b < c); i++)
{
    ...
    modifica b
}
// qui b è visibile mentre i non lo è
for (int i = 3; i > 0; i--)
    System.out.println(i);
```

# Ciclo do

- Capita spesso di dover *eseguire il corpo di un ciclo almeno una volta* per poi ripeterne l'esecuzione se una condizione è verificata
- Esempio tipico: leggere un valore in ingresso ed eventualmente rileggerlo finché non viene introdotto un valore “valido”

# Ciclo do

- Si può usare un ciclo **while** “innaturale”

```
// si usa un'inizializzazione "ingiustificata"  
double rate = 0;  
while (rate <= 0)  
{ System.out.println("Inserire il tasso:");  
  rate = in.nextDouble();  
}
```

ma per comodità esiste il ciclo **do**

```
double rate;  
do  
{ System.out.println("Inserire il tasso:");  
  rate = in.nextDouble();  
} while (rate <= 0); // notare ";" !!!
```

# Differenza fra cicli while e do

```
inizializzazione;  
while (condizione)  
{  
    enunciati;  
    aggiornamento;  
}
```

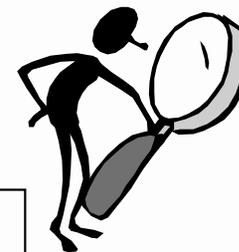
```
inizializzazione;  
do  
{  
    enunciati;  
    aggiornamento;  
} while (condizione);
```

- ❑ La differenza nell'esecuzione e' solo nella prima iterazione del ciclo.
- ❑ Nel ciclo **while** gli **enunciati** del corpo non sono mai eseguiti, se la **condizione** e' valutata falsa alla prima iterazione (prima verifica)
- ❑ Nel ciclo **do** gli **enunciati** del corpo sono eseguiti una volta, se la **condizione** e' valutata falsa alla prima iterazione

**Lezione XIII**  
**Gi 25-Ott-2007**

**Complementi di  
sintassi Java**

# Operatore di selezione



- ❑ Java prevede un operatore di selezione nella forma

```
condizione ? valore1 : valore2
```

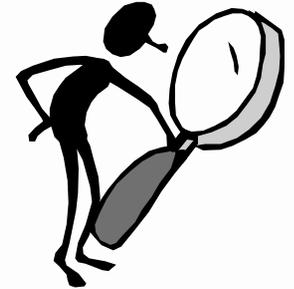
- ❑ Se la condizione è vera, l'espressione vale *valore1*, altrimenti *valore2*
- ❑ Può essere usato nelle espressioni di assegnazione

```
y = x >= 0 ? x : -x; // y = |x| modulo
```

- ❑ L'espressione e' una scorciatoia per

```
if (x >= 0)
    y = x;
else
    y = -x;
```

# L'enunciato switch

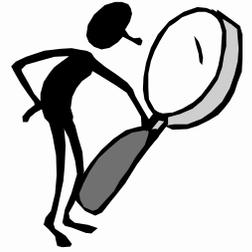


- Una sequenza che confronti *un'unica variabile intera* con diverse *alternative costanti* può essere realizzata con un enunciato **switch**

```
int x;
int y;
...
if (x == 1)
    y = 1;
else if (x == 2)
    y = 4;
else if (x == 4)
    y = 16;
else
    y = 0;
```

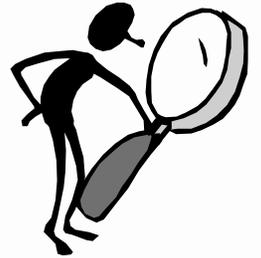
```
int x;
int y;
...
switch (x)
{
    case 1: y = 1; break;
    case 2: y = 4; break;
    case 4: y = 16; break;
    default: y = 0; break;
}
```

# L'enunciato switch



- ❑ **Vantaggio:** non bisogna ripetere il nome della variabile da confrontare
- ❑ **Svantaggio:** non si può usare se la variabile da confrontare non è intera
- ❑ **Svantaggio:** non si può usare se uno dei valori da confrontare non è costante
- ❑ **Svantaggio:** ogni **case** deve terminare con un enunciato **break**, altrimenti viene eseguito anche il corpo del **case** successivo! Questo è fonte di molti errori...

# Verificare la fine di un intervallo



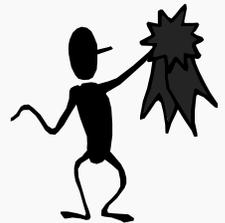
- ❑ Non usate mai la seguente espressione per verificare la fine di un intervallo in un ciclo

```
for (int i = 0; i != n; i++) //se n e' negativo?  
{                          // ciclo infinito  
...  
}
```



- ❑ al suo posto usare

```
for (int i = 0; i < n; i++)  
/* cosi' funziona, anche con valori negativi di n */  
{  
...  
}
```



# Verificare la fine di un intervallo

- ❑ Con i numeri in virgola mobile e' anche peggio!!!

Questa è la costante pi greco

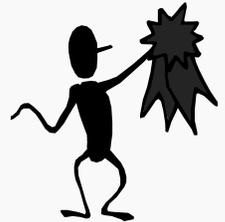


```
for (double alfa=0; alfa != Math.PI; alfa += Math.PI/16)
/* il ciclo risulta infinito per l'approssimazione
dell'aritmetica in virgola mobile!!! */
{
...
}
```



- ❑ al suo posto usare

```
for (double alfa=0; alfa < Math.PI; alfa += math.PI/16)
//cosi' funziona bene!!!
{
...
}
```



# **Valori sentinella e ciclo e mezzo**

# Leggere una sequenza di dati

- Molti problemi di elaborazione richiedono la *lettura di una sequenza di dati in ingresso*
  - ad esempio calcolare la somma di numeri interi
  - ogni numero sia inserito in una riga diversa
- Spesso il programmatore non sa *quanti saranno* i dati forniti in ingresso dall'utente
- **Problema:** leggere una sequenza di dati in ingresso (ad esempio numeri interi) *finché i dati non sono finiti*
- **Soluzione:** metodi della classe `java.util.Scanner`

# Valori sentinella

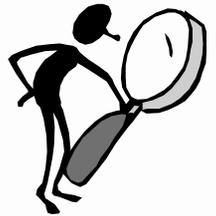
- Nel nostro problema, supponiamo di segnalare la fine dei dati con un valore particolare, ad esempio la lettera **Q** (come quit che in inglese significa “uscire”)
- La lettera **Q** si dice “*valore sentinella*”
- Non possiamo adoperare come sentinella un numero intero, perché qualsiasi numero intero fa parte dell’insieme dei possibili dati da acquisire
- Saremo costretti ad acquisire i dati come stringhe e non come numeri interi
  - Insieme dei dati in ingresso:  $Z \cup \{ "Q", "q" \}$

# Leggere una sequenza di dati

```
Scanner in = new Scanner(System.in);
int sum = 0;
boolean done = false;
while (!done)
{
    String line = in.next();
    if (line.equalsIgnoreCase("Q"))
        done = true;
    else
        sum += Integer.parseInt(line);
}
```

- ❑ Il ciclo usa la variabile “ausiliaria” **done** diversa dalla variabile che contiene i dati **line** perché la verifica della condizione va fatta a metà del ciclo

# Il problema del “ciclo e mezzo”



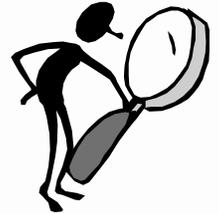
## □ Un ciclo del tipo

- “fai qualcosa, verifica una condizione, fai qualcos’altro e ripeti il ciclo se la condizione era vera”

non ha una struttura di controllo predefinita in Java e deve essere realizzata con un “trucco” come quello appena visto di usare una variabile booleana detta *variabile di controllo* del ciclo

- ## □ Una struttura di questo tipo si chiama anche “ciclo e mezzo” o *ciclo ridondante* (perché c’è qualcosa di “aggiunto” di innaturale...)

# Il problema del “ciclo e mezzo”



❑ Per uscire da un ciclo si può usare anche l'enunciato **break**

❑ Si realizza un ciclo infinito **while** (**true**) dal quale si esce soltanto quando si verifica la condizione all'interno del ciclo

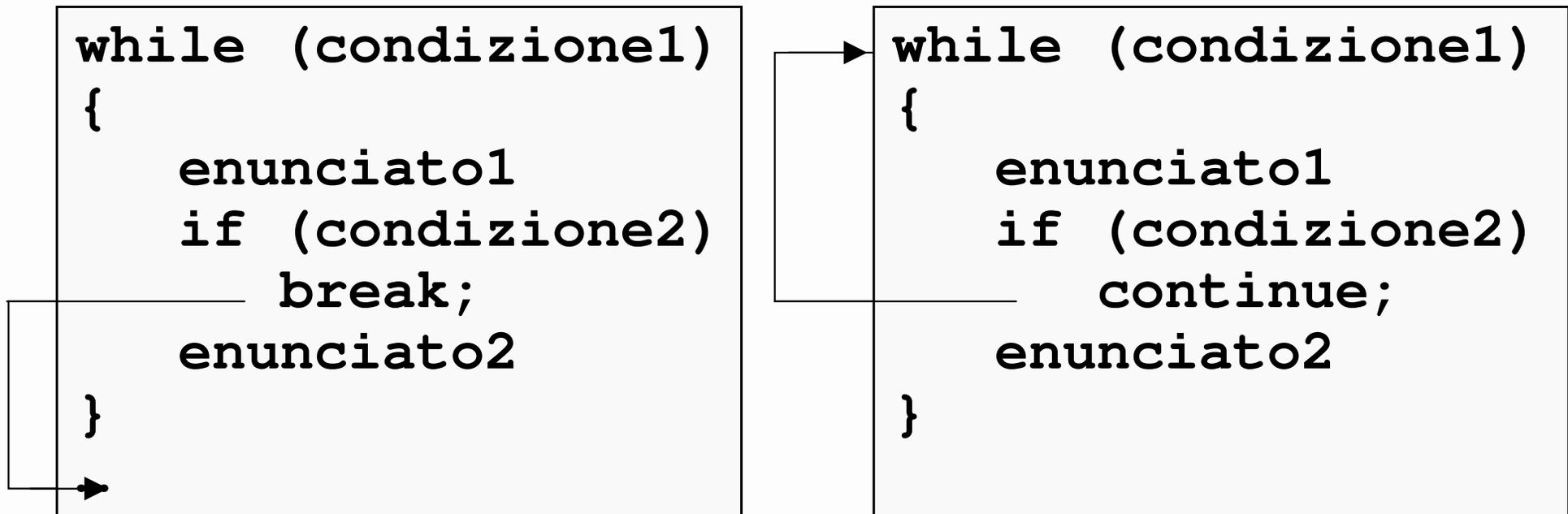
```
while (true)
{
    String line = in.next();
    if (line.equalsIgnoreCase("Q"))
        break;
    else
        ... // elabora line
}
```

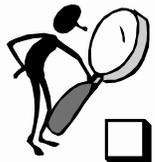
❑ Nel caso di cicli annidati l'enunciato **break** provoca la terminazione del ciclo più interno tra quelli in cui si trova annidato l'enunciato stesso

# Enunciati **break** e **continue**



- ❑ Esiste anche l'enunciato **continue** e puo' essere usato all'interno dei cicli
- ❑ provoca la terminazione dell'iterazione corrente e il passaggio all'iterazione successiva.





# Il problema del “ciclo e mezzo”

- ❑ Per uscire dal ciclo e mezzo si può, in alternativa, usare un enunciato con effetto collaterale

```
String line; // va definita fuori
while ((line = in.next()).equalsIgnoreCase( "Q" ))
{
    ... // elabora line
}
```

- ❑ L'effetto collaterale è che la condizione logica del ciclo while contiene anche un'assegnazione alla variabile riferimento String line.
- ❑ Non è bene, in generale, usare enunciati con effetti collaterali, perché rendono il codice difficile da leggere
- ❑ In questo caso la soluzione è però molto attraente

# Esercizio

- ❑ Scrivere una classe eseguibile che invii a standard output i numeri multipli del numero intero  $p$  compresi nell'intervallo intero  $[\text{min}, \text{max}]$
- ❑ Algoritmo
  1. porre  $i = \text{min}$
  2. finché  $i$  è minore o uguale a  $\text{max}$  ripetere i passi 3 e 4
  3. stampare  $i$ , se risulta divisibile per  $p$
  4. incrementare  $i$
  5. fine
- ❑ Non effettuiamo, per ora, verifiche sui valori dei numeri  $p$ ,  $\text{min}$ ,  $\text{max}$
- ❑ Classe **MultipleLister** (elencatore di multipli)

# Esercizio

- ❑ La stampa ottenuta e' un po' disordinata
- ❑ Si modifichi la classe in modo da stampare n numeri per riga (ad esempio  $n = 10$ )
- ❑ Per fare questo contiamo i multipli trovati e quando il loro numero è multiplo di n inseriamo una nuova riga
- ❑ Classe **NewLineMultipleLister**

# Esercizio

- ❑ Ancora non abbiamo ottenuto una stampa ordinata
- ❑ Si modifichi la classe in modo da stampare i numeri in colonna
- ❑ Per ottenere questo, decidiamo di stampare tutti i multipli con un numero fisso di cifre  $k$  (ad esempio 5)
- ❑ Se un multiplo non ha cinque cifre, inseriamo degli spazi a sinistra finché otteniamo il numero di cifre volute

3	6	9
12	15	18

- ❑ Classe **InColumnMultipleLister**

# Esercizio

- Aggiungete ora i seguenti controlli:
  - Se  $p \leq 0$ , si stampi un avviso e si termini il programma
  - Se  $\text{min} > \text{max}$ , si scambino i valori

# Esercizio: un punto nel piano

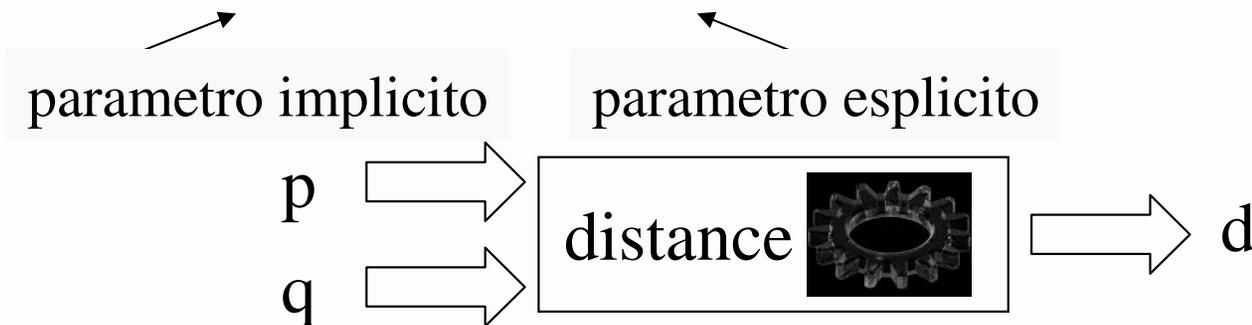
- ❑ Scrivere la classe **MyPoint2D** per rappresentare un punto del piano cartesiano
- ❑ Fra punti del piano si definiscano le seguenti funzioni
  - Distanza fra due punti
  - Punto medio fra due punti
- ❑ Si definisca anche una descrizione testuale di un oggetto di tipo **MyPoint2D**

# Esercizio: MyPoint2D - la distanza

```
/**
 * calcola la distanza fra due punti
 * @param q il secondo punto
 * @return la distanza fra i due punti
 */
public double distance(MyPoint2D q) {...}
```

□ Il metodo si usa nel seguente modo:

```
...
MyPoint2D p = new MyPoint2D(...);
MyPoint2D q = new MyPoint2D(...);
...
double d = p.distance(q);
```



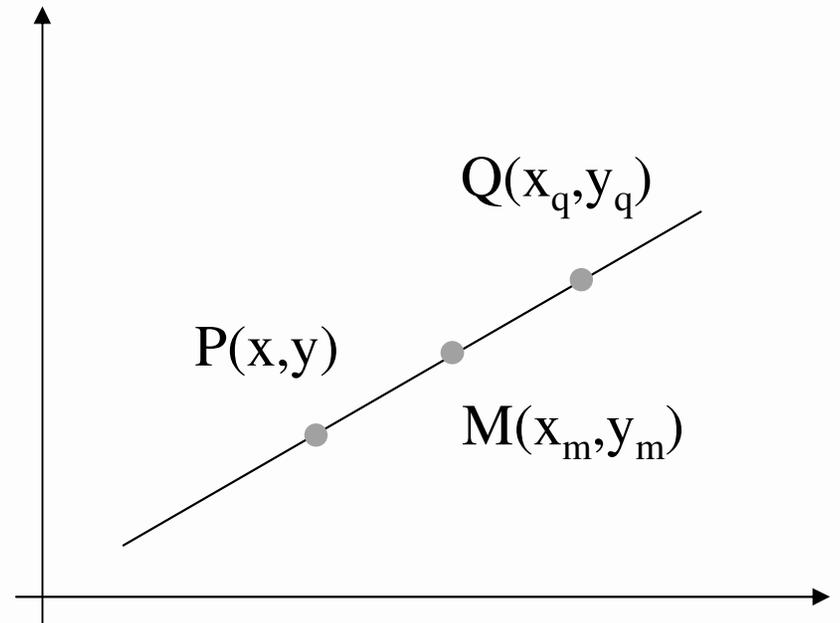
# Esercizio: classe MyPoint2D

□ Distanza

$$|PQ| = \sqrt{(x - x_q)^2 + (y - y_q)^2}$$

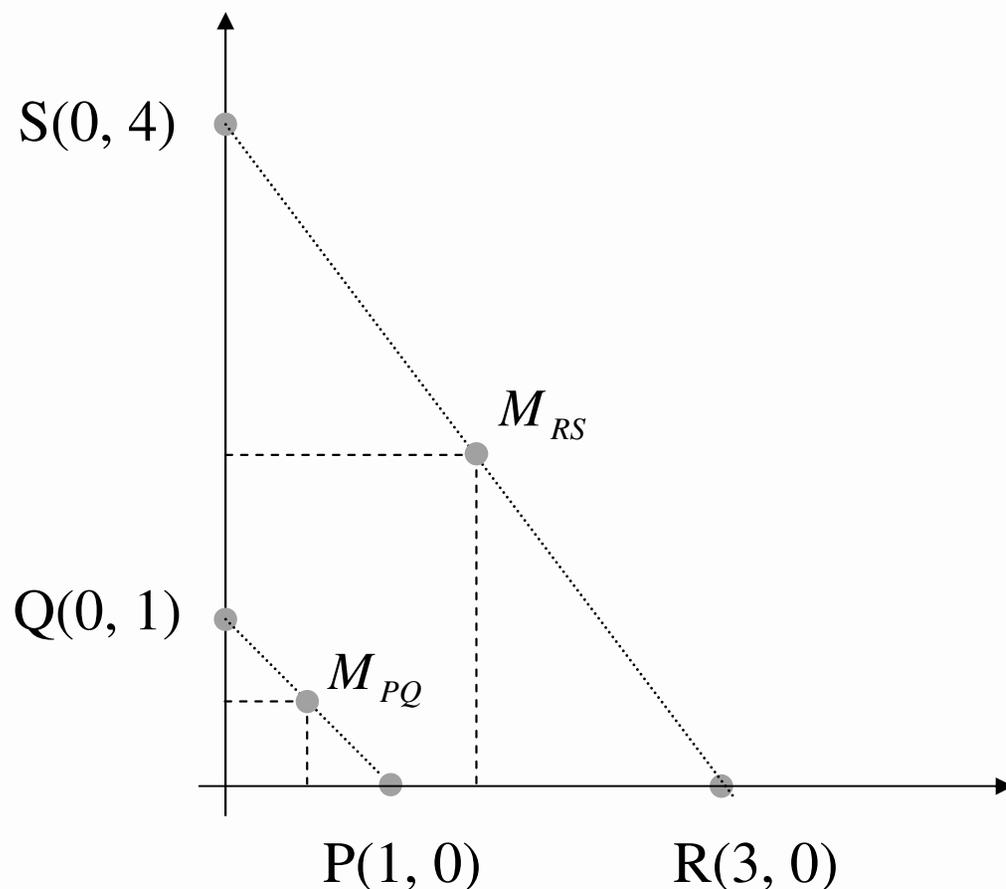
□ Punto medio

$$M = \left( x_m = \frac{x + x_q}{2}, y_m = \frac{y + y_q}{2} \right)$$



# Esercizio: un punto nel piano

- Si scriva poi una classe di prova che richieda l'introduzione da standard input di due punti  $p$  e  $q$  e ne calcoli la distanza e il punto medio, inviando i risultati a standard output
  - Individuare dei casi di prova, ad esempio i seguenti:



$$\overline{RS} = 5$$

$$\overline{PQ} = \sqrt{2}$$

$$M_{PQ} = \left(\frac{1}{2}, \frac{1}{2}\right)$$

$$M_{RS} = \left(\frac{3}{2}, 2\right)$$

# Esercizio: numeri complessi

- ❑ In Java non esiste un tipo di dati fondamentali per elaborare i numeri complessi
- ❑ Non e' fornita neppure una classe nella java platform API per questa funzione
- ❑ Scrivere la classe **MyComplex** per rappresentare i numeri complessi
- ❑ Realizzare le seguenti operazioni
  - Somma, sottrazione, prodotto, divisione
  - Parti reale e immaginaria, modulo, complesso coniugato, reciproco
  - Descrizione testuale  $(a + ib)$