JAVA for DUCK

Indice

Introduzione all'uso di Java

Inizialmente nato come un progetto per implementare codici di programmazione dentro apparecchi elettronici di comune utilizzo casalingo, **Java** (nato dal genio di James Gosling che, originariamente, gli diede il nome OAK) è il più popolare linguaggio utilizzato nello sviluppo di applicazioni sia in ambiente desktop che per Internet.

Java è un linguaggio Class-Based (basato su Classi) e Object Oriented (orientato agli oggetti), specificamente disegnato per implementare qualsiasi tipo di dipendenza necessaria.

Il cuore di Java è la **Java Virtual Machine** (JMV) che rende il linguaggio completamente indipendente dalla piattaforma (cioé da hardware e sistema operativo) nonostante sia stato creto dalla **Sun Microsystem** per funzionare nella piattaforma proprietaria Solaris.

Nel sito ufficiale del linguaggio - java.com – si trovano gratuitamente tutte le versioni della Java Virtual Machine; il sito è ottimizzato per distinguere il sistema operativo che si utilizza e selezionare automaticamente la versione adatta alla piattaforma in cui dobbiamo installarla.

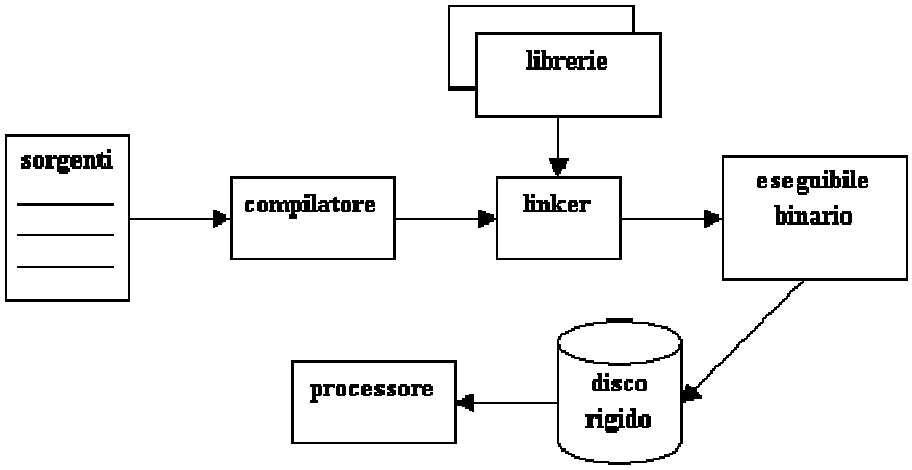
La portabilità di JAVA

Il linguaggio di programmazione Java è stato pensato come linguaggio semplice e familiare utile anche per integrare Internet con piccole applicazioni per la rete, dette **applet**, eseguibili all'interno delle pagine web visibili in Internet.
Java, creato dalla Sun Microsystem ed apparso pubblicamente nel **1995** è un linguaggio **orientato agli oggetti**. Questo significa che utilizza costrutti per trattare i concetti fondamentali di **oggetto**, **classe** ed **ereditarietà**.

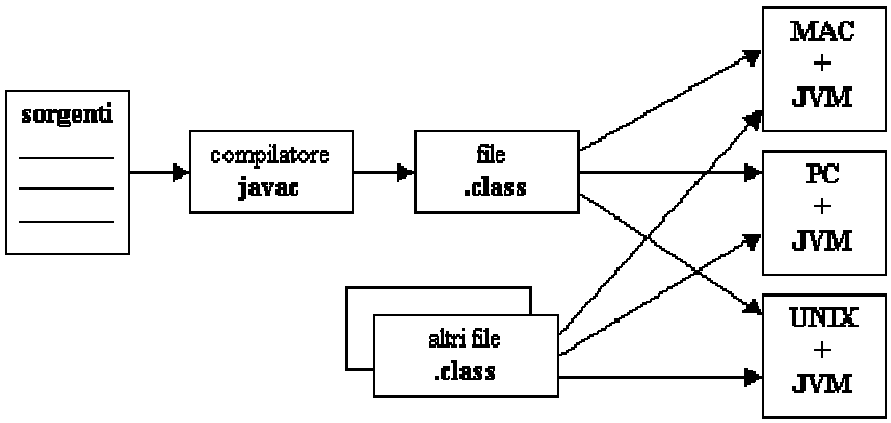
Java è inoltre caratterizzato dalla **portability** (portabilità) su differenti piattaforme.

In generale, quando si scrive un programma lo si scrive e lo si compila su un certo elaboratore con un dato sistema operativo questo fa sì che il nostro programma possa "girare" solo su elaboratori dotati dello stesso sistema operativo.
Al contrario, un'applicazione scritta in Java, una volta scritta e compilata, non ha bisogno di subire l'operazione di "porting" perchè potrà essere eseguita senza modifiche su doversi sistemi operativi ed elaboratori con architetture hardware diverse.

Vediamo come viene realizzata in pratica la portabilità con Java.



Schema standard di compilazione/esecuzione: tutto dipende dalla piattaforma



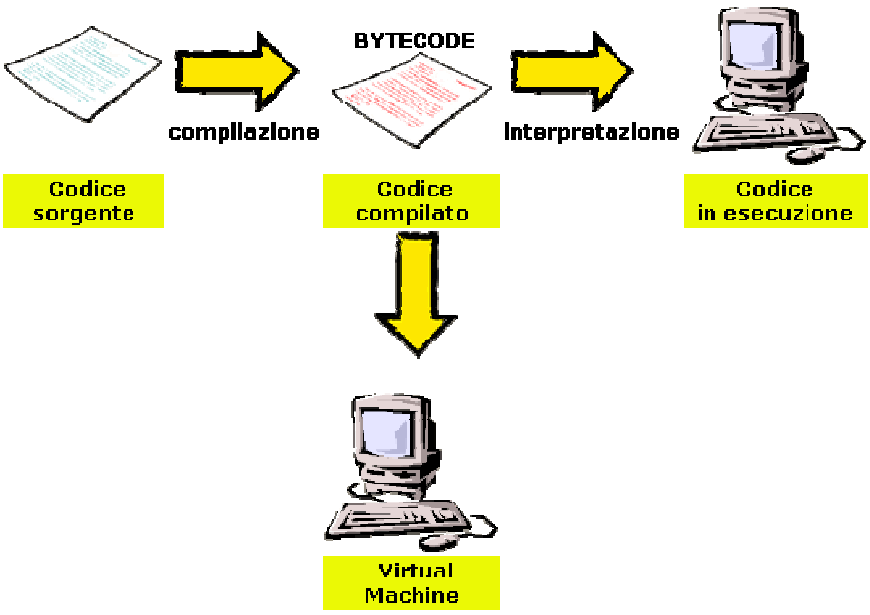
Schema misto di compilazione e interpretazione

- Uno o più file sorgente, **.java,** del programma vengono compilati in **bytecode** (*codice di byte*) producendo file **.class**
- Esecuzione (interpretazione) con la **Java Virtual Machine** (**JVM** - Macchina Virtuale di Java).

Quando si compila il codice sorgente scritto in Java, il compilatore genera il codice compilato, chiamato bytecode. Questo e un formato intermedio indipendente dall'architettura ed e usato per trasportare il codice in modo efficiente tra varie piattaforme hardware e software.
Questo codice non può essere eseguito da una macchina reale. E' un codice generato per una macchina astratta, detta Virtual Machine.

Ci sono **JVM** praticamente per **tutte le piattaforme** (**Solaris, Windows, Linux, Macintosh,...**) perciò i file **.class** possono essere eseguiti su qualunque JVM. Questo permette, ad esempio, di scaricare un applet e eseguirlo direttamente sulla propria JVM.

Per essere eseguito su una macchina reale, il codice compilato (**bytecode**) deve essere interpretato.



Significa che ogni istruzione viene tradotta, della Virtual Machine, nella corrispettiva istruzione della macchina su cui si sta eseguendo l'applicazione in quel momento. A seconda dell'elaboratore su cui il programma viene eseguito, si ha un'interpretazione diversa. E' questa la modalità attraverso la quale le applicazioni Java diventano **portabili**.

Per questi motivi si può dire che **Java è un linguaggio interpretato**, anche se la produzione del bytecode è effettuata con un'operazione di **compilazione**.

Ambiente di sviluppo

Dopo questa breve introduzione prepariamo tutti gli strumenti necessari per poter scrivere ed eseguire del codice Java.

Per cominciare a scrivere **programmi anche complessi** con una certa facilità utilizzeremo **NetBeans**.

Netbeans è un **IDE** (**Integrated Developmet Enviroment**, ambiente integrato di sviluppo) cioè uno strumento che permette di **editare**, **compilare** ed **eseguire** i **progetti JAVA** che scriveremo. L'ambiente IDE consente, in particolare, la **creazione automatica** del **codice** per la **gestione dei componenti grafici** (*vedi Usare l'interfaccia GUI Builder dell'IDE NetBeans*) e dei relativi **eventi** da gestire grazie all'uso di un **wizard**[1].

Per utilizzare NetBeans si può operare come segue:

- installare **JDK** (Java Developer Kit), se non è già stato installato,
- installare **Netbeans** (Java Standard Edition).

per scaricare ed entrambi i software:

- utilizzare il link http://www.oracle.com/technetwork/java/javase/downloads/index.html

avendo cura di prelevare entrambi nell'**ultima versione**[2] disponibile adatti alla vostra piattaforma (Windows XP, 7 ....).

Non considerare le versioni "beta" perché sono ancora in fase di test quindi non necessariamente stabili!

Struttura dei programmi

Le **applicazioni Java sono dei programmi veri e propri** che possono essere eseguiti autonomamente.

In Java, un'applicazione può essere costituita da una o più **classi** (*vedi Programmazione ad oggetti: classi e istanze*).

Tra tutte le **classi** che compongono l'applicazione, una si differenzia dalle altre perchè contiene il **metodo** (*vedi metodi*) **main()**.
Questo metodo è importante perche **l'esecuzione di un'applicazione Java comincia eseguendo questo metodo**.

Per iniziare, ci limiteremo alla costruzione di applicazioni con una sola classe. Un semplice programma Java, formato da una sola classe, assume la seguente struttura:

```
class <nome classe>
{
  public static void main(String args[])
  {
    dichiarazioni di variabili istruzioni
  }
}
```

La classe dichiarata in questo modo contiene il solo metodo main. Questo metodo raggruppa le dichiarazioni di variabili e le istruzioni che compongono l'applicazione Java.
Esso viene dichiarato usando le parole chiave public, static e void che specificano alcune proprietè del metodo:

- **public** indica che metodo a pubblico ed è visibile;
- **void** indica che non ci sono valori di ritorno;
- **static** indica che il metodo è associato alla classe e non può essere richiamato dai singoli oggetti della classe.

[1] *IT: Mago. Indica un procedimento che consente di semplificare attività complesse.*
[2] *La versione di **NetBeans** descritta in questi appunti è la **8.0** con **Java: 1.7.0_05.***

Dopo il nome del metodo, tra parentesi, sono indicati i **parametri**. Il metodo **main** possiede come parametro un array di stringhe (indicato con args[]) che corrisponde ai parametri passati dalla riga di comando quando viene eseguita l'applicazione.

Le parentesi **graffe** sono usate per individuare un **blocco**, che può essere:

- una classe,
- un metodo,
- oppure un insieme di istruzioni.

I **blocchi** sono rappresentati da un gruppo di istruzioni racchiuse tra **parentesi graffe {...}**. All'interno di **un blocco** possono essere presenti **altri blocchi in modo annidato**. Per poter meglio distinguere questi blocchi conviene seguire ed usare una tecnica di **indentazione**. Anche se un programma Java potrebbe essere scritto usando una sola riga, è utile disporre ordinatamente le sue istruzioni e gli elementi sintattici per migliorare la leggibilita del programma stesso.

Ogni istruzione ed ogni dichiarazione deve terminare con il **punto e virgola (;).**

E' importante sottolineare che **Java è case-sensitive**. Questo significa che vi è differenza nello scrivere una lettera maiuscola o minuscola. Per esempio System è diverso da system. Un errore nella digitazione comporta la segnalazione di un errore di compilazione.

Prima di passare ad approfondire i concetti di classe, metodo etc. cominciamo con un semplice esempio

Esercizi svolti

Scriviamo un semplice programma per visualizzare un messaggio.

```
class Esempio_1{
   public static void main(String args(]) {
       System.out.println("Quanti anni hai?");
   }
}
```

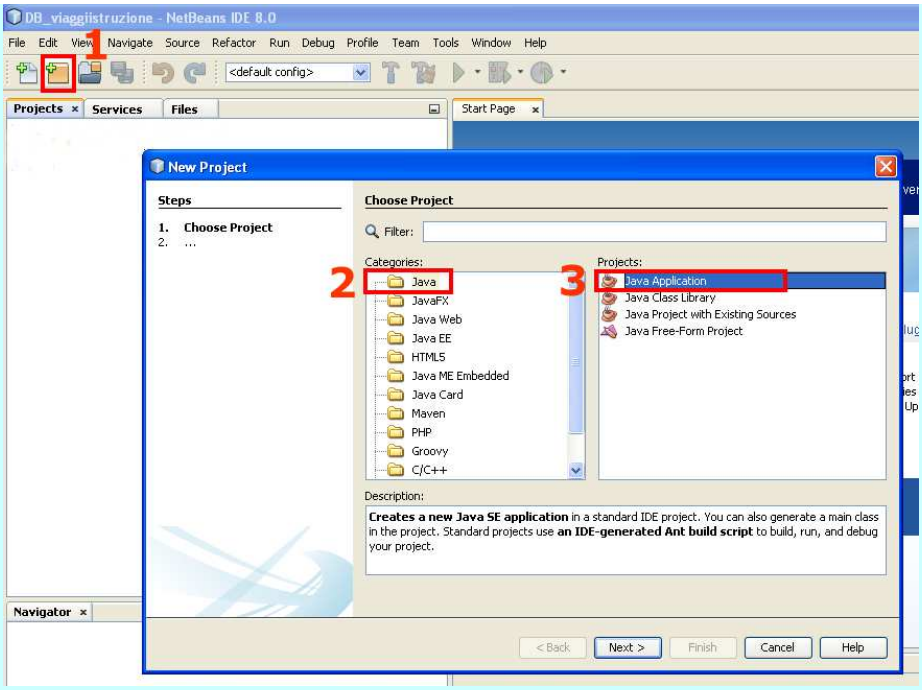
Analizziamo il programma appena scritto:

Il **risultato** dell'esecuzione di questo programma è la visualizzazione a video della scritta "Quanti anni hai?", dopodiche il programma termina.

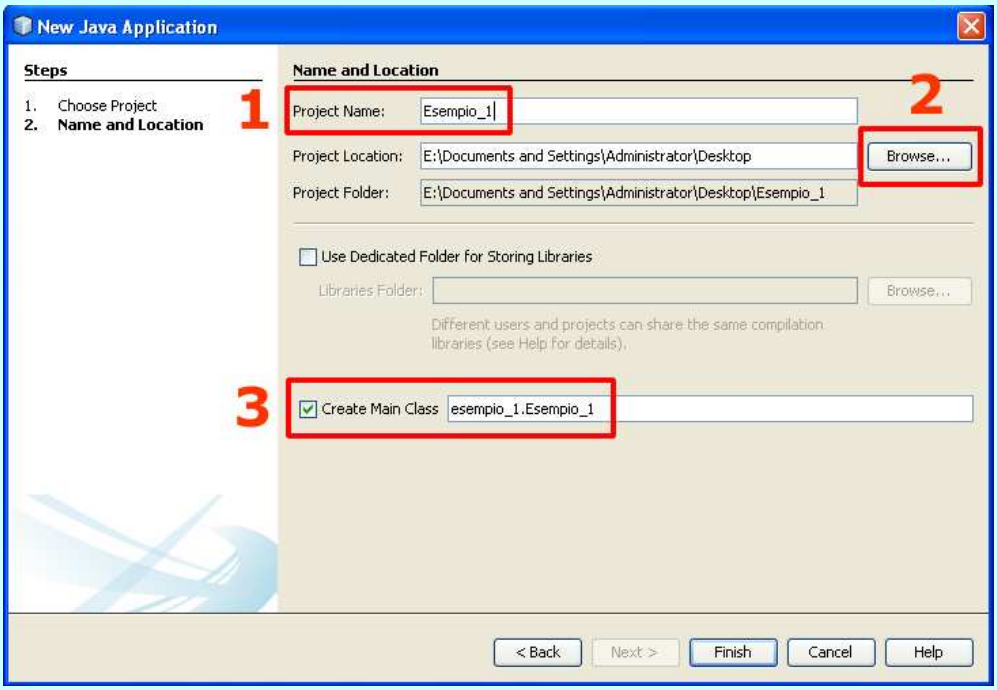
System.out.println(...) è un metodo (*vedi metodi*) che **scrive** sul dispositivo standard di output, solitamente il **video**, i suoi argomenti e torna a capo.

Vediamo quali sono le operazioni necessarie per scrivere e rendere eseguibile questo programma Java usando NetBeans.

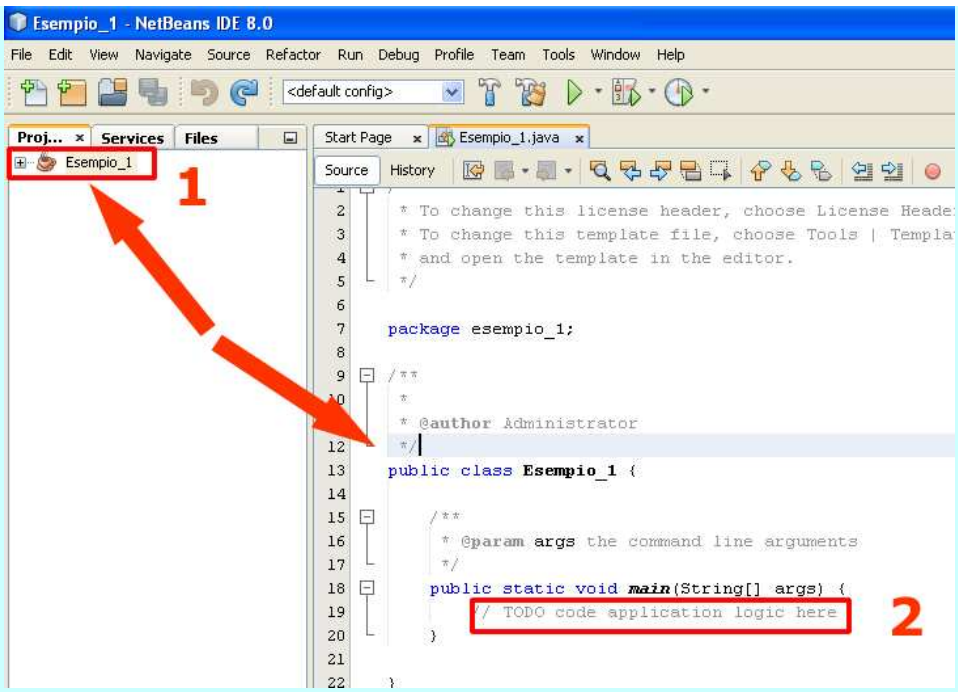
Dopo aver mandato in esecuzione NetBeans creiamo un **nuovo progetto** di tipo **Java Application** operando secondo le indicazioni della ***figura*** seguente:



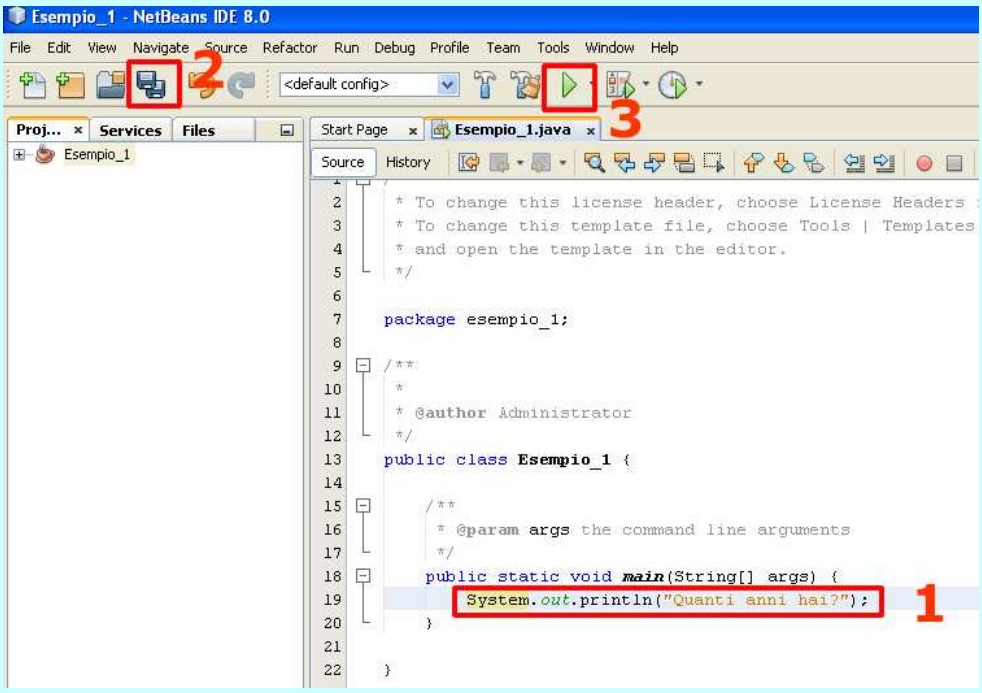
Dalla pagina **New Java Application** assegnamo un nome al progetto, selezioniamo la **collocazione** del progetto ed assicuriamoci di creare una **main class** come indicato in ***figura***:



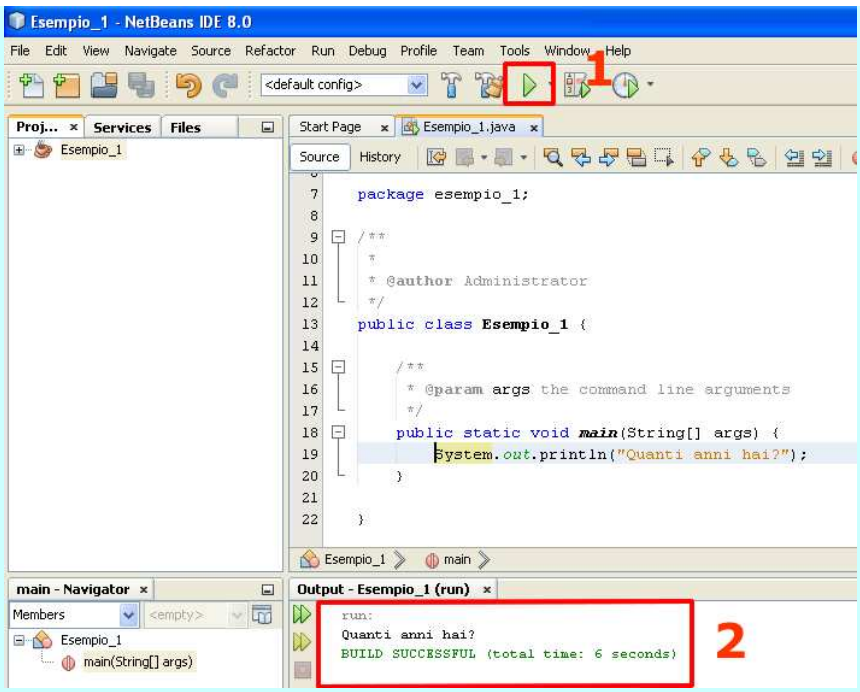
Il codice sorgente che scriveremo verrà salvato in un **file con estensione .java** ed il **nome** del **file** corrisponde al **nome della classe**, cioè il nome inserito dopo la parola chiave class, come si vede in ***figura***:



Nell'area indicata con (2) possiamo scrivere il codice del nostro programma.



Dopo aver scritto il codice (1) dovremo salvare il nostro progetto (2) e quindi potremo mandarlo in esecuzione (3). Il passo (2) verrà eseguito automaticamente cliccando **run project** (3). Il programma dell'esempio verrà essere salvato in un file chiamato Esempio_1.java e quindi eseguito.



Nelle ***figura*** precedente vediamo in (2) il risultato dell'esecuzione ed il tempo in cui si è completata.
Se il programma è composto da più classi **ogni classe viene memorizzata su un file diverso**.

Cenni sulla programmazione ad oggetti

Classi e oggetti

Java è un linguaggio **Object Oriented** basato sui due concetti principali della logica della programmazione ad oggetti:

- Classe
- Oggetto

Una **classe** è un'astrazione che rappresenta un insieme di oggetti che condividono le stesse caratteristiche e le stesse funzionalità.

Le classi definiscono dei tipi di dato e permettono la creazione degli oggetti secondo le caratteristiche definite nella classe stessa.

Un **oggetto** è un'istanza o più precisamente una realizzazione concreta di una classe. Per ottenere un oggetto dovremo quindi **istanziare la classe nell'oggetto**.

In altri termini una **classe** è paragonabile al **progetto di un'infrastruttura** che può poi essere realizzata o meno con l'**istanziazione dei suoi oggetti** tutti con le stesse caratteristiche, ovvero gli attributi (con valori diversi), su cui opereranno i metodi.

Una **classe** può anche essere vista come un **contenitore** all'interno della quale troviamo due elementi principali:

- **attributi** cioè **variabili** che identificano la classe e che vengono usati dai vari metodi per compiere qualche operazione.
- **metodi** cioè delle **funzioni** che svolgono un certo compito.

Quindi se creeremo una classe "Numeri" avremo, ad esempio, due attributi del tipo "numero1" e "numero2" (e potremo definire un metodo per assegnare i valori agli attributi) e due metodi "somma" e "sottrazione" che compiono rispettivamente la somma e la sottrazione del "numero1" e "numero2".

Come detto prima, la classe è un'entità a se stante, quindi adesso avremo necessità di istanziare la classe "Numeri" per poi fare una chiamata al metodo che assegna i valori agli attributi della classe (passando i valori desiderati come parametri del metodo) ed in seguito i metodi di somma e sottrazione.

Visto che ogni istanza della classe è una realizzazione fisica della classe le due istanze sono completamente indipendenti l'una dall'altra. Se creiamo due oggetti di tipo "Numeri" e richimaiamo il metodo di assegnazione passando numeri diversi otterremo due oggetti diversi. Dunque la somma e la sottrazione relative ai due oggetti saranno generalmente diversi.

Classi astratte

Una **classe astratta definisce una interfaccia** senza implementarla completamente. Questo serve come base di partenza per generare una o più classi specializzate aventi tutte la stessa interfaccia di base, le quali potranno poi essere utilizzate indifferentemente da applicazioni che conoscono l'interfaccia base della classe astratta. **La classe astratta da sola non può essere istanziata**, viene progettata soltanto per svolgere la funzione di **classe base** da cui le **classi derivate** possono ereditare i metodi. Le classi astratte sono usate anche per rappresentare concetti ed entità astratte.

Codice JAVA	**Descrizione**
`public abstract class Computer_classe_astratta`	`Classe astratta`
`{`	
`  public abstract void accendi();`	
`}`	
`public class PC extends Computer_classe_astratta`	`Classe concreta`
`{`	
`  public void accendi()`	`Implementazione per il metodo accendi()`
`  {`	
`      System.out.println("PC acceso");`	
`  }`	
`}`	

Le interfacce

Un **caso particolare di classi astratte** sono le cosiddette **interfacce**, dichiarate tramite la parola chiave "**interface**". Una **interfaccia** ha una struttura **simile a una classe**, ma può contenere **SOLO metodi d'istanza astratti e costanti** (quindi non può contenere costruttori, variabili statiche, variabili di istanza e metodi statici).

Ad esempio, questa è la dichiarazione dell'interfaccia **java.lang.Comparable**:

```
public interface Comparable<T>{
    public int compareTo(T  o);
}
```

Spesso l'interfaccia comprende anche una descrizione informale del significato dei metodi (che chiameremo specifica o contratto).

Packages

Le **classi, con i relativi metodi**, di varia utilità per lo sviluppo di applicazioni sono **incluse in liberie** e servono per costituire l''ambiente di programmazione Java.

Queste collezioni di classi, messe insieme in un unico "pacchetto", formano un unico file di estensione **.jar** detto comunemente **package**.

Le principali librerie sono:

- **java.lang:** collezione delle classi di base the e sempre inclusa in tutte le applicazioni.
- **java.io:** libreria per la gestione degli accessi ai file e ai flussi di input e output.
- **java.awt:** libreria di base per componenti **GUI**[3] contenente le classi per la gestione dei componenti grafici (colori. font, bottoni, finestre).
- **javax.swing:** libreria avanzata ed indipendente dalla piattaforma per componenti **GUI**.
- **java.net**: supporto per creare applicazioni che si scambiano dati attraverso la rete.
- **java.util:** classi di utility varie (gestione data, struttura dati stack, ...).
- **java.sql:** libreria Java Database Connectivity (**JDBC**) per facilitare l'accesso a database.

La disponibilità di un vasto insieme di librerie consente di raggiungere uno degli obiettivi della programmazione orientata agli oggetti: la **riusabilità del software**. Si può dire che le versioni piu recenti di Java differiscono dalle precedenti solo per il numero maggiore di librerie incluse piuttosto che per i costrutti del linguaggio.

Le librerie di Java sono quindi costituite da varie **classi** già sviluppate e **raggruppate in base all'area di utilizzo**, e sono molto utili per la creazione di nuovi programmi.

> *In Java non si possono definire tipi composti perché le classi rappresentano già delle definizioni di nuovi tipi e quando serve definire un tipo composto basta creare una classe che contiene tra i suoi attributi i tipi che interessano.*

Esercizi svolti

Creare semplici classi ed oggetti in Java

Entriamo adesso nel vivo della trattazione spiegando **come si creano classi ed oggetti in Java**. Creiamo la nostra classe "Numeri" con due attributi "numeroX" e "numeroY":

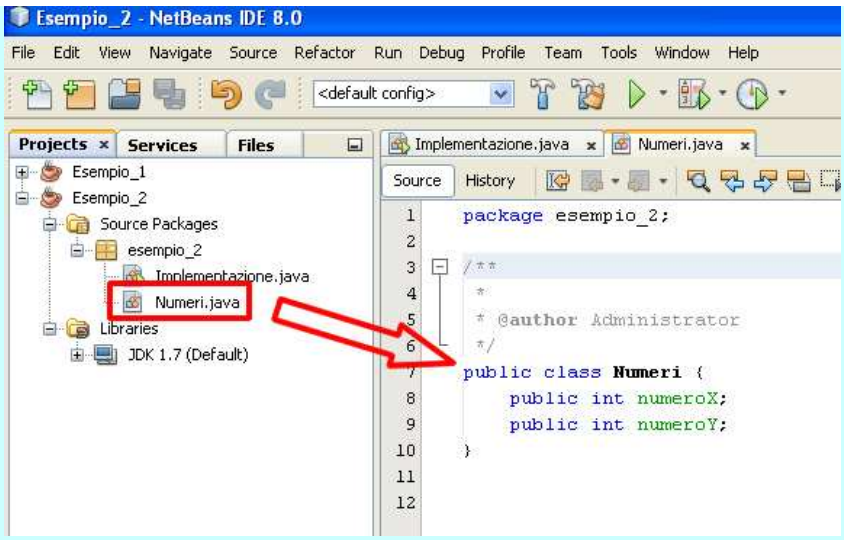
```
public class Numeri {
    public int numeroX;
    public int numeroY;
}
```

Come si può vedere un attributo viene dechiarato secondo la sintassi:

- **Modificatore di visibilità** (nell'esempio public),
- **Tipo della variabile** (nell'esempio int),
- **Nome variabile** (nell'esempio numeroX e numeroY).

Per quanto riguarda la parola chiave **public** rappresenta un modificatore di visibilità, ovvero indica che numeroX e numeroY sono due attributi che possono essere richiamti anche fuori dalla classe.

[3] ***GUI**: Graphical User Interface*



Abbiamo adesso creato la classe "Numeri" e l'abbiamo salvata in un file chiamato "Numeri.java". All'interno della classe che abbiamo scritto non c'è un metodo **main** nel quale viene istanziata la classe quindi dobbiamo definire un'altra classe con il **metodo main.**

La classe si chiamerà "**Implementazione**":

```
public class Implementazione {
      public static void main(String args[]){

      // --- Prima istanza della classe Numeri
      Numeri numeri1;
      numeri1 = new Numeri();
      numeri1.numeroX=2;
      numeri1.numeroY=3;

      // --- Seconda istanza della classe numeri
      Numeri numeri2;
      numeri2 = new Numeri();
      numeri2.numeroX=5;
      numeri2.numeroY=6;

      // --- Visualizziamo i valori numeroX e numeroY delle due istanze
      System.out.println(numeri1.numeroX); //Stampa 2
      System.out.println(numeri1.numeroY); //Stampa 3

      System.out.println(numeri2.numeroX); //Stampa 5
      System.out.println(numeri2.numeroY); //Stampa 6
      }
}
```

La classe "Implementazione" contiene il solo **metodo main** che costituisce il punto d'inizio per l'esecuzione del programma.

Quindi all'interno del main saranno presenti le **istanze della classe e tutti le chiamate ai metodi relativi**.

Analizziamo la classe "**Implementazione**".

Il parametro **String args[]** consente di salvare nell'array **args** eventuali stringhe scritte dall'utente in fase di lancio del programma[4].

Per quanto riguarda la **creazione di un oggetto** questa è eseguita in due passi:

- prima di tutto viene dichiarata una variabile di tipo Numeri (Numeri numeri1;)
- poi avviene la creazione vera e propria con la parola chiave **new** (numeri1 = new Numeri()).

Leggendo il listato completo della classe "**Implementazione**" si nota che vengono creati due oggetti

[4] *Per fare chiarezza, se lanciassimo (dopo la compilazione) la classe Implementazione con il seguente comando **"java Implementazione HELLO WORLD**" avremo che la stringa HELLO verrà salvata nella posizione 0 dell'array args (args[0]=HELLO) e WORLD verrà salvato nella posizione 1 dell'array args (args[1]=WORLD)*

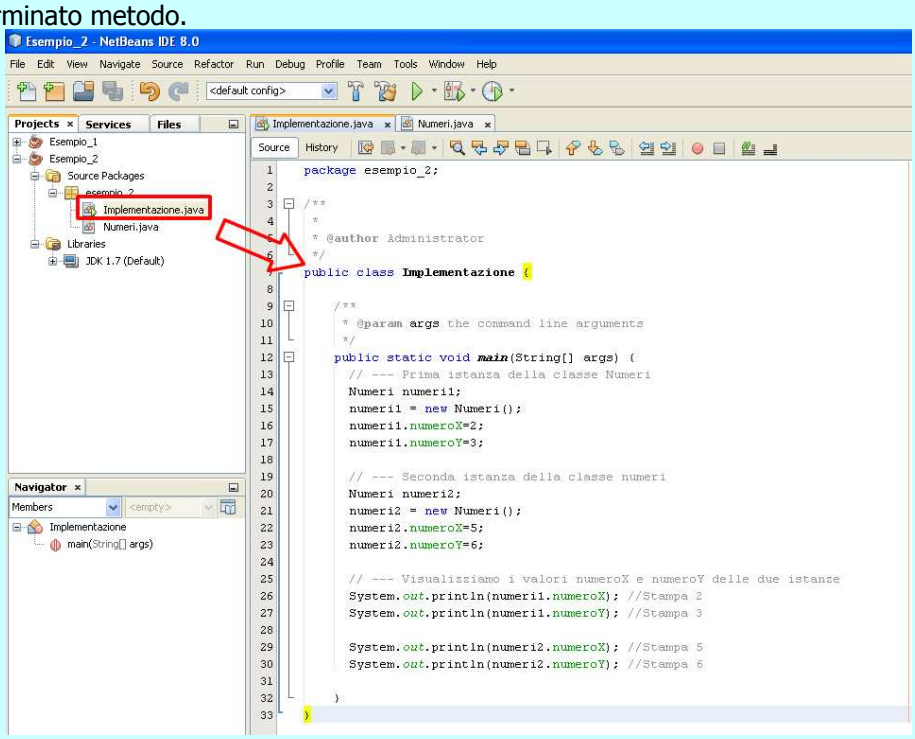
(numeri1 e numeri2) di una stessa classe. I due oggetti sono completamente distinti, come abbiamo precedentemente detto, infatti la visualizzazione degli attributi dei due oggetti mostra i valori corretti precedentemente impostati.

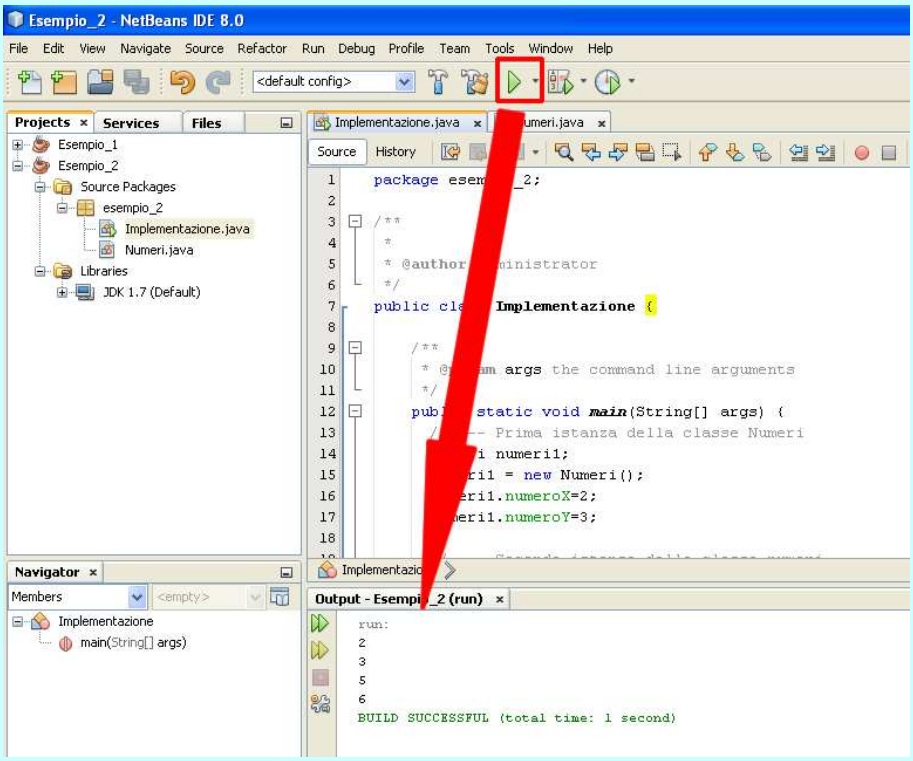
Per quanto riguarda **l'impostazione dei dati** possiamo dire che avviene **tramite il punto "."** che permette l'accesso agli attributi di quell'oggetto.

Quindi il significato della stringa

numeri1.numeroX=2; accedi all'**oggetto** identificato da **numeri1**
inserisci il **valore 2** nell'**attributo numeroX**

Il metodo **System.out.println()**, infine, permette di visualizzare il valore di una variabile o il risultato che ritorna un determinato metodo.





Cliccando su "**run project**" o **F6** si ottiene il risultato indicato in ***figura***

Importare i packages

Per scrivere un progetto a volte devono essere utilizzate numerose classi, ma in generale queste tendono a trovarsi in pochi package. È possibile **importare** un intero package utilizzando l'asterisco al posto del nome della classe. In ogni caso come l'**importazione non inserisce nulla**, anche l'uso della importazione globale non inserisce nulla ma comunica al compilatore di utilizzare il package indicati per ricercare classi che non sono state trovate.

Per usare un package in una classe, **prima della definizione della classe** dobbiamo quindi inserire l'istruzione **import** che indica l'intenzione di usare le classi presenti nel package:

```
import java.awt.*;
```

Alcuni packages vengono automaticamente importati: in particolare è sempre implicito un **import java.lang.*;** in questo modo **non è necessario importare classi di uso molto comune**.

Metodi

Definiamo ora quale sia il significato della parola **metodo,** come sia possibile dichiarare e richiamare un metodo.

Un metodo appartenente ad una certa classe è una sorta di funzione che all'interno contiene un blocco di codice che **esegue alcune operazioni**.

La sintassi per la dichiarazione di un metodo segue le seguenti regole:

```
[modificatori] tipoDiRitornoDelMetodo nomeDelMetodo([parametri]) {
corpoDelMetodo }
```

Esempio: definizione di un metodo

```
public int sottrazione (int x, int y){
     ...
}
```

Analizziamo adesso, in maniera approfondita, ogni singolo elemento che costituisce la dichiarazione di un metodo:

Modificatori: sono delle parole chiave che permettono di modificare il comportamento e le caratteristiche di un certo metodo. I modificatori più comuni sono **public**, **private**, **static**.

Tipo di ritorno del metodo: indica il tipo di ritorno che il metodo restituisce dopo l'esecuzione del "corpo del metodo". Se per esempio un metodo effettua la somma di due numeri ritornerà un **int** oppure se un metodo prenderà in input due string e al termine restituisce un'unica stringa il tipo di ritorno sarà **String**. Spesso alcuni metodi svolgono alcune operazioni che non ritornano nessun valore, in questo caso un metodo di questo tipo avrà come tipo di ritorno **void**.

- **Nome del metodo**: semplicente rappresenta il nome con cui verrà identificato il metodo. E' buona norma scegliere il nome del metodo seguendo le linee guida espresse in precedenza:
  Il nome di ogni metodo di una classe avrà l'**iniziale minuscola** e se è composto da più parole quelle successive avranno iniziale maiauscola come per esempio "restituisciNumeroMetriQuadri"..
- **Parametri**: indica i parametri che il metodo prenderà in ingresso. Se per esempio avremo un metodo che esegue la somma di due numeri avremo come parametri due variabili di tipo int.
- **Corpo del metodo**: insieme di istruzioni che vengono eseguita non appena il metodo viene richiamato.

Per chiarire i concetti esposti sopra vediamo alcuni esempi per la definizione di metodi.

Esempio: definizione di metodi

```
public class OperazioniSuNumeri{
   public int sottrazione (int x, int y){
     int sottrazione;
     sottrazione=x-y;
     return sottrazione;
```

```
   }

   public String comparazione (int x, int y){
     if(x>y){
          String esito="il numero" + x + "è maggiore del numero" + y;
          return esito;
     }else{
          String esito="il numero" + y + "è maggiore del numero" + x;
          return esito;
     }
   }
}
```

Nel codice soprastante sono stati dichiarati due metodi:

- uno che esegue la sottrazione tra due numeri,
- un altro che presi in ingresso due numeri restituisce una stringa che rappresenta l'esito della comparazione tra i due numeri.

La parola chiave **return** è utilizzata per terminare il metodo e far ritornare il risultato delle operazioni. Chiaramente se nella dichiarazione del metodo abbiamo definito come valore di ritorno un int non potremo fare un return di una stringa. Nel metodo "comparazione" abbiamo utilizzato un **operatore di concatenazione** (+) per ritornare un'unica stringa che contiene l'esito del confronto.

Una volta dichiarati i due metodi vediamo adesso come **richiamarli** e, a tale scopo, definiamo un'altra classe che chiameremo nuovamente "Implementazione" dove sarà presente il **metodo main()** che al suo interno **istanzierà la classe "OperazioniSuNumeri" e richiamerà i due metodi**:

```
public class Implementazione{
    public static void main(String args[]){
      // --- Istanza della classe OperazionisuNumeri
      OperazioniSuNumeri operazioni = new OperazioniSuNumeri();
      int sottrazione = operazioni.sottrazione(10,2); //richiamo il metodo
sottrazione
      String esitoComparazione = new String();
      esitoComparazione = operazioni.comparazione(8,3); //richiamo il metodo
comparazione

      // --- Visualizzazione dei risultati ottenuti
      System.out.println(esitoComparazione);
      System.out.println(sottrazione);
    }
}
```

Come possiamo vedere la chiamata ai metodi della classe "OperazioniSuNumeri" viene effettuata utilizzando il punto (**.**) come avevamo precedentemente visto per il settaggio degli attributi. Ovviamente quando viene richiamato un **metodo che richiede dei parametri** di input (come i metodi visti nell'esempio), è **necessario passare i valori nello stesso numero di quanti sono i parametri dichiarati** nel metodo e dello stesso tipo. Per esempio le seguenti chiamate ai metodi produrranno un errore:

```
// --- il metodo dichiarato consta di due parametri.
// --- passando un solo valore si ha un errore
operazioni.sottrazione(10);


// --- il metodo dichiarato consta di due parametri di tipo int.
// --- passando un valore di tipo stringa si verifica un errore.
operazioni.comparazione(9,"ciao");
```

Il metodo costruttore

Parliamo adesso del concetto di **metodo costruttore**. Un costruttore di una classe, prima di tutto, viene definito con lo **stesso nome della classe**, è una caratteristica che lo differenzia da un normale metodo è il fatto di **non avere un tipo di ritorno**.

Il metodo costruttore viene chiamo, solo ed esclusivamente, quando viene istanziata la classe, quindi si accede al costruttore una sola volta per ogni istanza della classe.

Esempio: definizione di un metodo costruttore

```
public class OperazioniSuNumeri {
      //costruttore
      public OperazioniSuNumeri {
          System.out.println("Ho istanziato una classe OperazioniSuNumeri);
      }
      public int sottrazione (int x, int y) {
          //sottrazione
      }
      public String comparazione (int x, int y) {
          //comparazione
      }
}
```

Come si vede nel codice di esempio il **costruttore** scritto rispecchia le caratteristiche sopra elencate ovvero non ha nessun tipo di ritorno ed ha lo **stesso nome della classe**.

Quando istanzieremo la classe "OperazioniSuNumeri" con il seguente codice:

```
OperazioniSuNumeri numero;
numero = new OperazioniSuNumeri();
```

di fatto chiameremo in maniera automatica il costruttore della classe e quindi avremo come output

```
Ho istanziato una classe OperazioniSuNumeri
```

Nel nostro esempio abbiamo inserito il costruttore **della classe come primo metodo**, ma ciò **non è rilevante** in quanto al momento della creazione dell'istanza dell'oggetto in automatico verrà richiamato il costruttore anche se è l'ultimo metodo inserito nella classe.

Da ciò si deduce che il Java non analizza le classi in maniera sequenziale.

Java ha la particolarità che all'atto di compilazione se non è presente un costruttore all'interno della classe viene aggiunto automaticamente un costruttore di default che consentirà di istanziare l'oggetto in maniera corretta ma non compirà nessuna operazione dato che il corpo del metodo di costruttore di default è vuoto.

A cosa serve il metodo costruttore?

Ipotizziamo di avere la nostra classe "OperazioniSuNumeri" che oltre ai metodi scritti negli esempi precedenti implementi la somma, la moltiplicazione, la divisione e molte altre funzionalità. Fino a questo momento, senza il costruttore, passavamo, sia per il metodo "sottrazione" che per il metodo "comparazione", i valori al momento della chiamata del metodo. Nel caso volessimo effettuare un'analisi completa su due numeri (quindi richiamando per esempio un metodo per la somma, il prodotto, la sottrazione, la divisione e la comparazione) senza il costruttore creeremo per ogni chiamata a funzione due variabili per immagazzinare i valori dei numeri "x" e "y" e quindi, in totale, verrebbero allocate dieci variabili con contenuto identico per poter chiamare le cinque funzioni.

Utilizzando il metodo costruttore sarà possibile allocare le due variabili una sola volta e i vari metodi chiamati leggeranno semplicemente il contenuto delle variabili d'istanza senza costruirne una copia.

Esempio:

```
public class OperazioniSuNumeri {
    public numeroX;
    public numeroY;
    public OperazioniSuNumeri(int x, int y) {
      numeroX=x;
      numeroY=y;
    }
    public int sottrazione() {
       int sottrazione;
       sottrazione=numeroX-numeroY;
```

```
        return sottrazione;
    }
    public int somma() {
        int somma;
        somma=numeroX+numeroY;
        return somma;
    }
    // gli altri metodi con la stessa struttura
}
```

e il codice per istanziare l'oggetto e la chiamata dei metodi cambierà nel modo seguente:

`public class Implementazione {`	
`  public static void Main(String args[]) {`	
`    OperazioniSuNumeri operazioni;`	`istanza della classe OperazionisuNumeri`
`    operazioni = new OperazioniSuNumeri(10,2);`	
`    int sottrazione = operazioni.sottrazione();`	`richiamo il metodo sottrazione`
`    int somma = operazioni.somma();`	`richiamo il metodo somma`
`    System.out.println(somma);`	`visualizzo i risultati ottenuti`
`    System.out.println(sottrazione);`	
`  }` `}`	

Regole di scrittura

Prima di iniziare a scrivere del codice Java è bene apprendere delle piccole regole che stanno alla base della scrittura di un codice pulito, manutenibile e di facile comprensione anche da altre persone:

Nome classe:

Indicheremo il nome di una classe sempre con una lettera maiuscola. Quindi se creiamo una classe "casa" il nome della classe sarà "Casa". In caso il nome della classe sia composta da più parole indicheremo ogni iniziale della parola con la lettera maiuscola come per esempio "CasaInVendita".

Struttura del nome dei metodi:

Il nome di ogni metodo di una classe avrà l'iniziale minuscola e se è composto da più parole quelle successive avranno iniziale maiauscola come per esempio "restituisciNumeroMetriQuadri".

Struttura del nome degli attributi:

Valgono le stesse regole espresse per i metodi.

Significato dei nomi:

E' buona regola dare dei nomi per classi, metodi e attributi che abbiano un significato. Questo permette di leggere il codice di una certa applicazione con minore difficoltà e nel minor tempo possibile. Per esempio, se abbiamo un metodo che esegue la somma tra due numeri e lo chiamiamo "somma", una persona che leggerà il nostro codice capirà immediatamente cosa quel metodo esegue e potrà continuare la lettura del codice senza troppi problemi. Nel caso avessimo creato un metodo che esegue la somma e lo avessimo chiamato "sm" ciò avrebbe necessariamente costretto il lettore ad andare a leggersi il codice riguardante il metodo per capirne il funzionamento.

Anche se **Java non analizza le classi in maniera sequenziale** è buona norma strutturare la classe seguendo la lista sottostante:

1. Variabili di istanze ed eventuali costanti
2. Il costruttore
3. Tutti gli altri metodi della classe

Queste buone norme di programmazione sono elencate nella sola ottica di mettere in grado chiunque di leggere o magari rileggere dopo molto tempo e comprendere rapidamente il codice.

Variabili e costanti

Le variabili

Una variabile non è altro che una locazione di memoria nella quale viene salvato un determinato dato. Il nome con cui viene identificata una variabile, per esempio la variabile numeroX, rappresenta l'indirizzo fisico in memoria in quella locazione.

Non tutte le variabile però sono uguali, infatti a seconda del punto di codice in cui viene definita una variabile questa assumerà un preciso significato.

Legato a ciò possiamo collegare il concetto di **scope di una variabile** o **visibilità**.

Prima di tutto una variabile, per essere utilizzata per svolgere qualche operazione deve essere:

- prima **dichiarata**,
- poi **inizializzata**.

Per la **dichiarazione** si fa riferimento al seguente codice:

```
int x;
```

che ha lo scopo di allocare in memoria una variabile referenziata dal nome "x" che in memoria occuperà, in bit, la dimensione di un int.
Di fatto però all'interno della variabile "x" non viene immagazzinato nessun valore, diciamo che la dichiarazione è quel processo che permette di allocare in memoria spazio sufficiante per poi inizializzare "x" con un numero intero.

La fase di **inizializzazione** viene effettuata nel seguente codice:

```
x=10;
```

In questo modo si va a scrivere il valore 10 all'interno della cella di memoria riferenziata da "x".

- In Java sono presenti tre tipi di variabili:
- **variabili di istanza,**
- **variabili locali,**
- **parametri formali.**

Andiamo adesso ad analizzarli singolarmente fornendo qualche esempio per chiarire meglio i concetti.

Variabili di istanza

Le variabili di istanza sono quelle variabile che sono definite all'interno di una classe, ma fuori dai metodi della classe stessa. Le variabili di istanza sono denominate anche attributi della classe.

```
public class Numeri {
     public int numeroX;
     public int numeroY;
     :
}
```

le variabili "numeroX" e "numeroY" sono dunque delle variabili di istanza. Le variabili di istanza verranno deallocate dalla memoria non appena l'oggetto, istanza della classe, terminerà di esistere per esempio non appena il flusso dell'applicazione sarà terminato. Spesso alcune variabili di istanza vengono definite come **costanti** perchè il loro i valore rimarrà costante (non potrà essere modificato). Per definire una costante si usa la sintassi seguente:

```
public final int NUMEROZ = 10; //dichiaro una costante pari a 10
```

Variabili locali

Le variabili locali sono tutte quelle variabili che vengono dichiarate ed utilizzate all'interno dei metodi di una classe. Riprendendo spunto dal codice scritto in precedenza:

```
public int sottrazione (int x, int y) {
       int sottrazione; // "sottrazione" è una variabile locale
```

```
        sottrazione=x-y;
        return sottrazione;
}
```

la variabile "sottrazione" sarà dunque una variabile locale.

E' importante specificare che la visibilità delle variabili locali è relativa al metodo nella quale viene dichiarata. Per esempio se scrivessi un'altro metodo all'interno della classe e facessi riferimento alla variabile "sottrazione" riceverei un messaggio di errore del compilatore che richiede di dichiare la variabile perchè, per quel metodo, di fatto non esiste.
Una variabile locale viene deallocata non appena il metodo effettua il return e quindi ritorna al metodo principale main della classe.

Parametri formali

I parametri formali sono quelle variabili che vengono dichiarate all'interno delle **parentesi** tonde di un determinato **metodo**. Sempre analizzando il codice dell'esempio precedente:

```
public int sottrazione (int x, int y)
```

le variabili "x" ed "y" sono parametri formali. Con questa dichiarazione le variabili "x" e "y" sono state solo dichiarate e quindi non contengono nessun valore.

L'inizializzazione della variabile avverrà non appena verrà richiamato il metodo sottrazione e verranno passati tra parentesi i due numeri su i quali effettuare la somma.
Leggendo attentamente il blocco di codice relativo al metodo sottrazione possiamo vedere che le variabili "x", "y" e "sottrazione" vengono usate nello stesso modo.

All'atto pratico fra un parametro formale ed una variabile locale non c'è nessuna differenza: possiamo dire che un **parametro formale è una sottocategoria delle variabili locali**.

Dato che appartengono alla famiglia delle variabili locali i parametri formali avranno ovviamente identica visibilità e verranno deallocati non appena il metodo effettuerà il **return**.

Le costanti

La sintassi utilizzata per la definizione delle costanti è analoga a quella utilizzata per definire le variabili, a parte l'aggiunta della parola chiave **final** che precede la dichiarazione:

```
final int costante = 31;
```

final indica al compilatore che il valore associato alla variabile di tipo intero chiamata costante non potrà più essere variato durante l'esecuzione del programma. Anche in questo caso è possibile eseguire le operazioni di dichiarazione ed inizializzazione in due passi, ma sempre con il vincolo che, **una volta eseguita l'inizializzazione della costante, il valore di quest'ultima non potrà più essere variato**.

Esempio: definizione di costanti

Codice JAVA	Descrizione
`final int costante;` costante = 3;	`assegna il valore 3 alla costante di tipo intero chiamata "costante"`

`final int costante;`
`costante = 3;`
`costante = 5;`
	`viene rilevato un errore in fase di compilazione perché non è possibile variare il valore di una costante dopo averla inizializzata`

Tipi di dati

Il tipo di dato individua la natura dei dati che saranno memorizzati all'interno delle variabili. Per natura dei dati intendiamo la qualità, la quantità e i limiti dell'informazione che è possibile memorizzare. Ogni tipo di dato possiede uno specifico nome che ne identifica le caratteristiche.

I tipi di dati standard, detti anche tipi fondamentali, definiscono sia la gamma dei valori memorizzati, sia lo

spazio occupato all'interno della memoria.

I dati possono essere classificati in base al tipo come:

1. **tipi primitivi**.
   I tipi primitivi sono i tipi semplici che non possono essere decomposti, come per esempio numeri interi o booleani.
2. **tipi derivati.**
   I tipi derivati si ottengono dai tipi primitivi mediante opportuni operatori forniti dal linguaggio: essi includono i tipi strutturati (record) o gli array, le classi e così via

Dati primitivi

Gli interi

- **byte** = 8 bit = valori da -128 a 127 incluso
- **short** = 16 bit = da -32768 a 32767 incluso
- **int** = 32 bit = da -2147483648 a 2147483647 incluso
- **long** = 64 bit = da -9223372036854775808 a 9223372036854775807 incluso

Esempio: numeri interi

```
byte VarByte = 100;
short VarShort = 30000;
int VarInt = -2000000000;
long VarLong = 9000000000000;
```

I numeri a virgola mobile

- **float** = 32 bit = da 3.4e–038 a 3.4e+038
- **double** = 64 bit = da 1.7e–308 a 1.7e+308

- Il tipo **float** specifica un valore in **precisione singola** che usa 32 bits di memoria. La precizione singola con alcuni processori è più veloce ed occupa la metà dello spazio usato dalla precisione doppia, ma diventa impreciso quando i valori sono molto grandi o molto piccoli. Le variabili di tipo float sono comode quando si ha bisogno di operare con i **numeri decimali**, ma non è richiesto un largo grado di precisione. Per esempio, float può essere usato per rappresentare **monete** e loro **frazioni** (dollari e centesimi, euro e centesimi, …).
- Il tipo **double** specifica un valore in **precisione doppia** che usa 64 bits per memorizzare i numeri. Attualmente, con i moderni processori ottimizzati per i calcoli matematici ad alta velocità, la doppia precisione è più velo ce della singola. Tutte le funzioni matematiche trascendenti, come ad esempio **sin( )**, **cos( )**, e **sqrt( )**, restituiscono valori **double**. Quando c'è la necessità di di mantenere di mantenere l'accuratezza nel corso di molti calcoli iterativi, o di manipolare numeri di grandi dimensioni, **double** è la scelta migliore.

Esempio: numeri in virgola mobile

Codice JAVA	Descrizione
`float VarFloat = 100.5;`	`Precisione singola`
`double VarDouble = 10000000.5;`	`Doppia precisione`
`double d1 = 123.4;`	`Doppia precisione`
`double d2 = 1.234e2;`	`Lo stesso valore di d1, ma in notazione scientifica`
`float f1  = 123.4f;`	`Precisione singola`

```
class Test {
  public static void main(String[] args){
      float f1=3.2f;
      float f2=6.5f;

      if(f1==3.2f){
```

```
        System.out.println("uguale");
      else {
        System.out.println("diverso");
      }
      if(f2==6.5f){
        System.out.println("uguale");
      else {
        System.out.println("diverso");
      }
   }
}
```

Se, in questo esempio, togliamo la "f" (nella condizione indicata dall'**if**) confronteremo floating con diversa precisione il che può causare risultati inattesi.

I dati boolean

Un tipo di dato **boolean** può assumere soltanto due valori:

- **boolean** = true || false = vero o falso

La dichiarazione ed inizializzazione di una variabile booleana avviene tramite un'istruzione analoga a quella utilizzata nel caso dei tipi di dati numerici:

```
boolean nomeVariabile = ValoreBooleano;
```

dove **boolean** indica il tipo della variabile chiamata **nomeVariabile**, mentre **ValoreBooleano** deve essere sostituito con uno dei due valori che la variabile può assumere: true o false. Anche in questo caso le istruzioni di dichiarazione e di inizializzazione possono essere separate:

```
boolean nomeVariabile;
nomeVariabile = ValoreBooleano;
```

sempre con il vincolo che la variabile non venga utilizzata prima di essere inizializzata.

Per quanto semplice, questo tipo di dato riveste un'importanza fondamentale in informatica. E' infatti attraverso il tipo booleano che è possibile valutare un'espressione attribuendole un valore vero o falso, ed è in base a questo genere di valutazioni che si definisce il flusso logico di un programma.

Gli **operatori di confronto** (*vedi operatori si confronto*), messi a disposizione dalla sintassi Java, consentono, ad esempio, di valutare alcune espressioni booleane restituendo come output sempre dei valori booleano a seconda che le espressioni vengano ritenute valide o meno:

Esempi: operatori di confronto

Codice JAVA	output
`int int1 = 7;` `String str1 = new String("Una stringa qualsiasi");`	
`System.out.println(int1 > 10);`	`false`
`System.out.println(int1 <= 7);`	`true`
`System.out.println(str1.equals("Una stringa qualsiasi"));`	`true`

`int x = 5;`
`int y = 10;`
`boolean risultatoBinario;`
`risultatoBinario = (x == y);`
`System.out.println(risultatoBinario);`
	`false`

`risultatoBinario = (x < y);` `System.out.println(risultatoBinario);`	`true`

Il carattere

- **char** = 16 bit, interi senza segno che rappresentano un carattere Unicode = da '\u0000' a '\uffff'

Il tipo char consente di memorizzare un qualsiasi carattere Unicode. Il carattere è delimitato da apice singolo ('). Vediamo un esempio:

```
char VarChar = 'x';
```

Il tipo di dati char può contenere alcuni caratteri che non possono essere specificati tramite la tastiera. tali caratteri prendono il nome di sequenze di escape, possiamo definirle utilizzando il carattere back slash (\) seguito da un carattere come descritto qui di seguito:

\n - Crea una nuova riga e va a capo
\r - Ritorno a capo
\f - Crea una nuova pagina
\' - Inserisce un apice singolo
\" - Inserisce un apice doppio
\\ - Inserisce una barra (back slash)
\b - Cancella i caratteri a sinistra (back space)
\t - Inserisce un carattere id tabulazione

Dati derivati: stringhe

Una **stringa** per definizione è una sequenza di caratteri. Nel linguaggio **Java non** esiste il **tipo dati primitivo stringa**, ma la gestione di tali oggetti è affidata all'apposita **classe String** (inclusa nel package **java.lang**), perciò le stringhe vengono trattate come veri e propri **oggetti**. Ogni oggetto **String** contiene una serie di char quindi viene gestito come array di caratteri, per questo motivo i caratteri delle stringhe sono numerati da 0 ad length()-1.

Questo tipo di dato però è così importante che, in Java, gode di un trattamento speciale pur NON essendo, come detto, un tipo predefinito.

Creare un oggetto di tipo String

Per **creare un oggetto di tipo String** (differentemente dagli oggetti di altre classi) non dobbiamo necessariamente bisogno di invocare il metodo new; la creazione di una oggetto String può infatti essere:

- **Esplicita**: attraverso il costruttore polimorfo String()

```
String s= new String("parola");
String s= new String();

char[] ciaoArray = { 'c', 'i', 'a', 'o', '!' };
String ciaoString = new String(ciaoArray);
System.out.println(ciaoString);  // output: ciao!
```

Come con qualsiasi altro oggetto, è possibile **creare oggetti String** utilizzando la parola chiave "**new**" e un costruttore (modo esplicito).

- **Implicita**: quando il compilatore incontra una serie di caratteri delimitati da virgolette crea un oggetto di tipo String a cui assegna la stringa individuata.

```
String s = "parola"; //equivale a
String s = new String("parola");
String greeting = "Ciao mondo!";
String OggettoStringa = new String("Valore testuale ");
OggettoStringa = OggettoStringa + "dell'OggettoStringa";
System.out.println(OggettoStringa);
```

In questo caso, "Ciao mondo!" è una stringa letterale, una serie di caratteri nel codice che viene racchiuso tra doppi apici. Ogni volta che si incontra una stringa letterale nel codice, il compilatore crea un oggetto String con il suo valore, in questo caso, "Ciao mondo".

In definitiva, si possono dichiarare variabili di classe String semplicemente assegnare loro stringhe arbitrarie. Ogni stringa viene racchiusa tra virgolette, tutte le operazioni sulle stringhe vengono effettuate utilizzando i metodi contenuti nella classe **String**.

Esempio: creare un oggetto String

Codice JAVA	output

`String stringa = "Bau Bau";`
`String nome = "Carlo";`
`String cognome;`
`cognome = "Rossi";`
`nome = "Paolo  ";`	

`System.out.print(nome);  // output: senza andare  a capo`
`System.out.println(cognome); // output: andando a capo`
	`Paolo Rossi`

Qualunque carattere può comparire in una stringa, utilizzando, se necessario, una **sequenza di escape** (*vedi sequenze di escape*).

Esempio: usare una sequenza di tipo escape

Codice JAVA	output
`System.out.println("Ha detto \"Sí\"!!")`	`Ha detto "Sí"!!`
`System.out.println("sopra\nsotto");`	`sopra` `sotto`
`System.out.println("il file C:\\Documenti\\readme.txt");`	`il file C:\Documenti\readme.txt`
`System.out.println("il carattere \u0041 e' la A in UNICODE");` `System.out.println("il carattere \\u0041 e' la A in UNICODE");`	`il carattere A è la A in UNICODE` `il carattere \u0041 è la A in UNICODE`

Estrazione di una sottostringa

```
str.substring(start, pastEnd)
```

Restituisce la sottostringa di str a partire dalla posizione **start** fino a **pastEnd - 1**.

Attenzione: il primo carattere di una stringa ha posizione 0, e l'ultimo ha posizione str.length()-1.

C	i	a	o	,		M	o	n	d	o	!	carattere
0	1	2	3	4	5	6	7	8	9	10	11	posizione

```
String saluto = "Ciao, Mondo!";
System.out.println(saluto.substring(6,10)); //output: Mondo
```

Concatenamento di Stringhe

Esempio: concatenare stringhe

Codice JAVA	output

`String nome = "Mario";`
`String cognome = "Rossi";`
`System.out.println(nome + cognome);`
	`MarioRossi`

`System.out.println(nome + "  " + cognome);`	`Mario  Rossi`

L'operatore **+** è *sovraccaricato* (overloaded) quindi **può essere applicato sia a numeri che a stringhe**.

Esempio: ridurre gli enunciati System.out.print

Codice JAVA	output
`System.out.println("Rai" + 3);`	`Rai3`

`System.out.println(3 + 4 + 5);`	`12`
`System.out.println("" + 3 + 4 + 5);`	`345`
`System.out.println(4 + 5 + "pippo");`	`9pippo`

La concatenazione è molto utile per ridurre il numero degli enunciati **System.out.print(...)** .

Esempio: uso di System.out.print, System.out.println

Codice JAVA	output
`System.out.print("L'interesse dopo un anno e':");`	`L'interesse dopo un anno e':`
`System.out.println(interesse);`	`43.9`
	`(sulla stessa riga e va a capo)`
`System.out.println("L'interesse dopo un anno e': " + interesse);`	`L'interesse dopo un anno e': 43.9`
	`(su una riga e va a capo)`

Confronto tra stringhe

Come per i numeri, anche per le stringhe può essere necessario effettuare dei confronti, considerando o ignorando la differenza fra i caratteri minuscoli e maiuscoli, per verificare se le stringhe sono uguali cioè se sono dalla stessa sequenza di caratteri o ovvero quale delle due stringhe precede l'altra in ordine alfabetico.

La classe String fornisce i **metodi** d'istanza necessari per queste operazioni di confronto: **equals** e **compareTo:**

Codice JAVA	Spiegazione
stringa1.**compareTo**(stringa2)	Confronta due stringhe e restituisce un valore intero che vale: • -1 se la stringa this precede alfanumericamente la stringa stringa2, • 0 se sono uguali, • +1 se la stringa *stringa1* segue alfanumericamente la stringa *stringa2*.
stringa1.**compareToIgnoreCase**(stringa2)	Confronta due stringhe, ignorando le differenze fra maiuscole e minuscole, e restituisce un valore intero che vale: • -1 se la stringa *stringa1* precede alfanumericamente la stringa *stringa2*, • 0 se sono uguali, • +1 se la stringa *stringa1* succede alfanumericamente la stringa *stringa2*.
stringa1.**compareTo**(stringa2)	Restituisce un int -1,0, 1 e seconda che la stringa *stringa1* sia minore, uguale, maggiore rispetto alla la stringa *stringa2*
stringa1.**equals**(stringa2) E' case sensitive	Confronta la stringa *stringa1* con la stringa *stringa2*. Restituisce true se sono uguali false in caso contrario.
stringa1.**equalsIgnoreCase**(stringa2) NON è' case sensitive	Restituisce true se e solo se *stringa2* è un oggetto String che rappresenta la stessa sequenza di caratteri della stringa *stringa1*, ignorando la differenza fra maiuscole e munuscole.

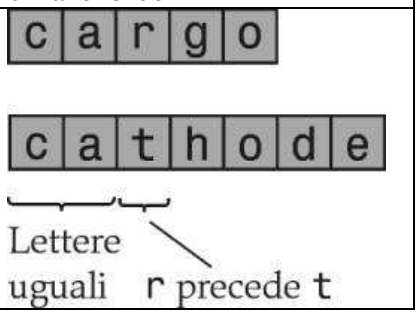
Esempio: uso del metodo equals

Codice JAVA	input /output
`String s1 = "Ciao";`	
`String s2 = Input.readLine();`	`Ciao`
`boolean b = s1.equals(s2);`	`b vale true`

class equalsDemo {
 public static void main(String args[]) {
  String s1 = "Ciao";
  String s2 = "Ciao";
  String s3 = "Arrivederci";
  String s4 = "CIAO";	

System.out.println(s1 + " uguale a " + s2 + " -> " + s1.equals(s2));	Ciao uguale a Ciao -> true
System.out.println(s1 + " uguale a " + s3 + " -> " + s1.equals(s3));	Ciao uguale a Good-bye -> false
System.out.println(s1 + " uguale a " + s4 + " -> " + s1.equals(s4));	Ciao uguale a CIAO -> false
System.out.println(s1 + " uguale a " + s4 + " -> " + s1.equalsIgnoreCase(s4));	Ciao uguale a CIAO -> true
} }	

Esempio: uso del metodo compareTo

Codice JAVA	output
b = "cargo".**compareTo**("cathode") < 0;	b vale true
	

Il metodo d'istanza **compareTo** *[**compareToIgnoreCase**]* può essere utilizzato ad esempio nella scrittura di algoritmi di **ordinamento di sequenze di stringhe**.

Tips and tricks - Quando non usare "=="

Per confrontare sia le stringhe che tutti gli oggetti non usare "==" perché si possono ottenere **risultati inattesi**.

Esempi: risultati inattesi!

Codice JAVA	output
String stringa1 = "Ciao"; String stringa2 = "Ciao"; boolean confronto = (stringa1 == stringa2);	Il valore di “confronto” sarà **true** come atteso

String stringa1 = "Ciao";
String stringa2 = Input.readLine(); // Ciao
boolean confronto = (stringa1 == stringa2);
	Pur avendo immesso “Ciao” in stringa2 il valore di “c

Tips and tricks - Non confondere la stringa vuota con l'oggetto null

Come avviene per tutte le classi, esiste un valore particolare che può essere associato ad una variabile di tipo **String** per indicare che quella variabile non si riferisce alcun oggetto.

Questo valore, indipendentemente dalla classe, è chiamato **null**. Sugli oggetti null non è possibile invocare metodi (si verificherebbe un errore di esecuzione). Per prevenire tali errori, si può testare l'uguaglianza con null usando ==. **Il valore null non deve essere confuso con la stringa vuota ""** (quella di lunghezza 0, che non contiene caratteri), che è un oggetto ben definito:

- è possibile invocare metodi su una stringa vuota;
- il confronto di uguaglianza con la stringa vuota deve essere fatto con il metodo equals(...);.

I principali metodi della classe String

Tipologia	Sintassi / Esempi	Significato
dichiarazione	**String stringa1**	Dichiara una stringa
	String stringa1 = ”sequenza di caratteri”	Dichiara ed inizializza una stringa
estrazione di sottostringhe	char **charAt**(int index);	Restituisce il carattere all'indice specificato. 0 = primo carattere, lenght()-1 = ultimo.
	stringa1.chartAt(int i)	Restituisce il carattere della stringa in posizione i-esima
	String **substring**(int inizio[, int fine])	Restituisce una sottostringa partendo da "inizio" fino a "fine". Se non viene indicato "fine" la stringa viene prelevata fino all'ultimo carattere. **E' importante ricordare che:** □ il primo carattere a sinistra è in **posizione 0**, □ l'ultimo carattere preso in considerazione sarà in posizione "**fine-1**",
	stringa1.**substring**(int inizio[, int fine])	
	split(String simbolo)	Restituisce un'array di stringhe dove ciascun elemento è una sottostringa divisa dal “simbolo” che abbiamo inserito.
	stringa1.**split**(";");	Esempio: il separatore è un punto e virgola.
confronto	int **compareTo**(String stringa2);	Confronta due stringhe e restituisce un valore intero che vale: □ -1 se la stringa this precede alfanumericamente il parametro, □ 0 se sono uguali, □ +1 se la stringa this succede alfanumericamente il parametro.
	stringa1.**compareTo**(stringa2)	Esempio
	int **compareToIgnoreCase**(String stringa2)	Confronta due stringhe, ignorando le differenze fra maiuscole e minuscole, e restituisce un valore intero che vale: □ -1 se la stringa this precede alfanumericamente il parametro, □ 0 se sono uguali, □ +1 se la stringa this succede alfanumericamente il parametro.
	stringa1. **compareToIgnoreCase** (stringa2)	Esempio
	boolean **equals**(String anotherString);	Confronta la stringa this con un'altra. Ritorna true se sono uguali false altrimenti. E' case

		sensitive, ossia Pippo e PiPpO sono diversi.
	stringa1.**equals**(String stringa2)	Esempio
	boolean **equalsIgnoreCase**(String anotherString);	Restituisce true se e solo se l'argomento è un oggetto String che rappresenta la stessa sequenza di caratteri della stringa this, ignorando la differenza fra maiuscole e munuscole. NON è' case sensitive, ossia Pippo e PiPpO sono uguali.
	stringa1.**equalsIgnoreCase**(stringa2)	Esempio
	int **length**();	Restituisce la lunghezza di una stringa
	stringa1.length()	Esempio
concatenamento	String **concat**(String stringa2);	Concatena la stringa parametro alla fine di this string.
	String **+** String	Concatena le due stringhe
ricerca	int **indexOf**(int carattere);	Restituisce l'indice della stringa this della prima occorrenza del carattere "carattere".
	int **indexOf**(int carattere, int inizio);	Restituisce l'indice della stringa this, partendo da "inizio", della prima occorrenza del carattere "carattere".
	int **indexOf**(String stringa2);	Restituisce l'indice della stringa this della prima occorrenza della striga stringa2.
	int **indexOf**(String stringa2, int fromIndex);	Restituisce l'indice della stringa this, partendo da fromIndex, della prima occorrenza della stringa stringa2.
	int **lastIndexOf**(int carattere);	Restituisce l'indice della stringa this dell'ultima occorrenza del carattere carattere.
	int **lastIndexOf**(String stringa2);	Restituisce l'indice della stringa this dell'ultima occorrenza della stringa stringa2.
	startsWith(String stringa2**);** **endWith(String** stringa2**)**	Restituisce true se la stringa inizia/finisce con la sequenza stringa2
	stringa1.startsWith(String stringa2) stringa1.endWith(String stringa2)	Esampio: restituiscono rispettivamente true se la stringa stringa1 inizia/finisce con la sequenza stringa2
conversione	String **toLowerCase**();	Converte la stringa this in soli caratteri minuscoli.
	str.toLowerCase()	Converte una stringa in minuscolo
	String **toUpperCase**();	Converte la stringa this in soli caratteri maiuscoli
	stringa1.toUpperCase()	Esempio
	String **trim**();	Toglie dalla stringa this spazi all'inizio e alla fine.
	stringa1.**trim**()	Eliminare gli spazi iniziali e finali di una stringa
	static String **valueOf**(tipo d);	trasforma in stringa il parametro d di qualsiasi tipo.
sostituzione	stringa1.**replace** (char oldChar, char newChar);	Sostituisce tutti i caratteri oldChar con i caratteri newChar.

	stringa1.**replaceAll**(String simbolo1, String simbolo2)	Sostituisce tutte le occorrenze del "simbolo1" con "simbolo2".
	stringa1.**replaceAll**(" ","")) stringa1.**replaceAll**("/","#"))	Esempio

Esempio: uso dei metodi length, charAt, equals

Codice JAVA	Descrizione
`public class gestioneStringa {` `  public static void main (String [] args){`	
`  String stringa="Questo è un oggetto Java";`	Dichiara una stringa
`  System.out.println("Lunghezza stringa : "+stringa.`**`length`**`());`	Visualizza la lunghezza della stringa

`  System.out.println("Il terzo carattere è : "+stringa.`**`charAt`**`(2));`
`  String stringa2=" seconda stringa";`
`  System.out.println("Stringhe concatenate : "+stringa+stringa2);`
	Visualizza il terzo carattere

`System.out.println(stringa.substring(0,6));`	Visualizza una sottostringa

`  System.out.println("Le due stringhe sono uguali :"+stringa.`**`equals`**`(stringa2));`
`  }`
`}`
	Verifica se le due stringhe sono uguali

Esempio: uso del metodo substring

Codice JAVA	output

`String ciao = "Ciao, Mondo!";`
`int lung = ciao.length();`
`System.out.println(lung);`
	`12`

`String data = "01-02-1995";`
`String anno = data.`**`substring`**`(6, 10);`
`System.out.println(anno);`
	`1995` Il sesto carattere contando da 0 è "1" mentre il nono (10-1) è "5"

Uso della classe StringBuffer

Una notevole limitazione delle stringhe della classe **String** è che sono oggetti immutabili, non possono essere modificati cioè non esistono metodi che modificano una stringa.

Per aumentare l'efficienza del codice è utile prendere in considerazione la classe **java.lang.StringBuffer** (*vedi esempio invertiStr*) che permette la modifica delle stringhe.

Tale classe mette a disposizione una serie di importanti metodi di modifica tra cui:

tipologia	sintassi	Descrizione
modifica	**append**(tipo d)	Concatena in fondo alla stringa data una rappresentazione a stringa del dato d.
	insert(int ind, tipo d)	Inserisce dall'indice ind una rappresentazione a stringa del dato d.
	delete(int inizio, int fine)	Cancella i caratteri da inizio a fine-1.
	deleteCharAt(int k)	Cancella il carattere di indice k
	replace(int inizio, int fine, String s)	Cancella i caratteri da inizio a fine ed inserisci s da inizio.
	reverse()	Inverte la stringa
	setCharAt(int k, char c)	Sostituisce il k-esimo carattere con il carattere c.
	setLength(int lung)	Imposta a lung la nuova lunghezza

conversione	String **toString**()	Trasforma in tipo String la StringBuffer this

Esempio: metodi leggiStr(), invertiStr(String stringa)

Codice JAVA	Descrizione
`import java.io.*;`	Funzioni I/O standard di base
`import java.lang.*;`	**Metodi** sulle stringhe

`class UsareStringhe`
`{`
`  public static void main(String args[])`
`  {`
`  System.out.println("Scrivi un testo:");`	

`  String s1 = leggiStr();`	**Chiamata del metodo leggiStr** per leggere una stringa da tastiera
`  System.out.println("Scrivi un'altro testo:");`	
`  String s2 = leggiStr();`	**Chiamata del metodo leggiStr** per leggere una stringa da tastiera
`  System.out.println("\nLa 1^ stringa e' " + s1);` `  System.out.println("La 2^ stringa e' " + s2);`	Visualizzazione stringhe

`  System.out.print("La 1^ stringa inizia con " + s1.charAt(0));`
`  System.out.println(", ed e' lunga " + s1.length() + " caratteri");`
`  System.out.print("La 2^ stringa inizia con " + s2.charAt(0));`
`  System.out.println(", ed e' lunga " + s2.length() + " caratteri");`
	Uso dei **metodi:**
	• length()
	• charAt(int)

`  if (s1.equals(s2)){`
`  System.out.println("Le stringhe sono uguali");`
`  }else{`
`  System.out.println("Le stringhe sono diverse");`
`  }`
	Verifica se le due stringhe sono uguali

`  System.out.println("La 1^ stringa in maiuscolo: " + s1.toUpperCase());` `  System.out.println("La 2^ stringa in minuscolo: " + s2.toLowerCase());`	Uso dei **metodi**: □ toUpperCase(): conversione in maiuscolo □ toLowerCase(): conversione in minuscolo
`  System.out.println("Stringa 1 invertita: " + invertiStr(s1));`	**Chiamata del metodo invertiStr** per visualizzare la stringa1 invertita
`  }`	Fine del metodo principale Main()
`  public static String leggiStr()` `  {`	**Metodo leggiStr()**, che non ha parametri in entrata e in uscita restituisce una stringa immessa da tastiera

`  InputStreamReader input = new InputStreamReader(System.in);`
`  BufferedReader tastiera = new  BufferedReader(input);`
`  try {`
`  String stringa = tastiera.readLine();`
`  return stringa;`
`  }`
`  catch (Exception e) {`
`  System.out.println("Errore: " + e + " nella lettura da tastiera");`
`  System.exit(0);`
`  return("errore");`
`  }`	

`  }`	Fine del metodo **leggiStr**
`  public static String invertiStr(String stringa) {` `  StringBuffer h = new StringBuffer(stringa);`	**Metodo invertiStr(String stringa)**, che **inverte una**

`int primo=0;`
`int ultimo=s.length()-1;`
`char temp;`
`while (primo<ultimo) {`
`temp=h.charAt(primo);`
`h.setCharAt(primo,h.charAt(ultimo));`
`h.setCharAt(ultimo,temp);`
`primo++;`
`ultimo--;`
`}`
`return h.toString();`
	stringa s passata come parametro

`}`	Fine del **metodo** invertiStr
`}`	Fine della **classe** UsareStringhe

Dati derivati: le date

La classe principale per trattare date ed ore è **java.util.GregorianCalendar** che estende la classe astratta **java.util.Calendar** e *sostituisce la ormai deprecata classe java.util.Date*.

La classe **GregorianCalendar** è utile per rappresentare il calendario standard utilizzato dalla maggior parte del mondo. Per questo motivo, probabilmente è preferibile utilizzarla nella maggior parte dei casi in cui serve fare uso di un calendario.

La classe **Calendar** è una **classe astratta** (*vedi classe astratta*) che fornisce metodi per la conversione tra un istante specifico nel tempo e una serie di campi del calendario, YEAR, MONTH, DAY_OF_MONTH, HOUR, e così via, e per **manipolare i campi del calendario**, come ottenere la data della prossima settimana etc. Un istante di tempo può essere rappresentato da un valore in millisecondi[5].

Il metodo java.util.Calendar.set(int year, int month, int date) imposta i valori per i campi del calendario: YEAR, MONTH6, and DAY_OF_MONTH e viene dichiarazione con:

```
public final void set(int year,int month,int date)
```

I parametri year, month e day corrispondono ai campo YEAR, MONTH e DAY rispettivamente. Questo metodo non restituisce valori.

La classe fornisce inoltre campi e metodi per l'implementazione di un sistema di calendario concreto, tali campi e metodi sono definiti come protetti.
Come altre classi anche Calendar fornisce un **metodo di classe**, **getInstance**, che restituisce un oggetto Calendar in cui i campi del calendario sono inizializzati con la data e l'ora correnti:

```
Calendario DataCorrente = Calendar.getInstance();
```

Un oggetto **Calendar** è in grado di produrre tutti i valori del campo di calendario necessari per attuare la formattazione in una particolare lingua e o in un particolare stile di calendario (per esempio, Japanese-Gregorian, Japanese-Traditional). **Calendar** definisce la gamma di valori restituiti da alcuni campi del calendario, così come il loro significato.

Ad esempio: il primo mese del sistema di calendario ha valore **MONTH == JANUARY** per tutti i calendari, altri valori sono definiti dalla sottoclasse concreta, come **ERA**.

Esempio: creare oggetti di classe GregorianCalendar e Calendar

Codice JAVA	Descrizione / output
import java.util.**Calendar**;	Importare la classe **astratta Calendar**
import java.util.**GregorianCalendar**;	Importare la classe **GregorianCalendar**
public static void main(String args[]){	
`Calendar dataA=new GregorianCalendar();`	Creare un oggetto di **classe GregorianCalendar** che contenga la data e l'ora corrente
`// GregorianCalendar dataA = new GregorianCalendar();`	Equivalente alla precedente

[5] *Millisecondi calcolati a partire dal 1 Gennaio 1970 00:00:00.000 GMT (Gregorian)*
[6] *I mesi iniziano da 0*

	Utilizzare il **metodo get()** passando come paramentro una delle **costanti statiche** definite nella classe:
`int anno1 = dataA.get(GregorianCalendar.YEAR);`	**YEAR**
`int mese1 = dataA.get(GregorianCalendar.MONTH)+1;`	**MONTH:** i mesi partono da 0
`int giorno1 = dataA.get(GregorianCalendar.DATE);`	**DATE**
`int ore1 = dataA.get(GregorianCalendar.HOUR);`	**HOUR**
`int minuti1 = dataA.get(GregorianCalendar.MINUTE);`	**MINUTE**
`int secondi1 = dataA.get(GregorianCalendar.SECOND);`	**SECOND**
`/*`	***Equivalenti a:***
`int anno1 = dataA.get(Calendar.YEAR);`	*GregorianCalendar.**YEAR***
`int mese1 = dataA.get(Calendar.MONTH)+1;`	*GregorianCalendar.**MONTH***
`int giorno1 = dataA.get(Calendar.DATE);`	*GregorianCalendar.**DATE***
`int ore1 = dataA.get(Calendar.HOUR);`	*GregorianCalendar.**HOUR***
`int minuti1 = dataA.get(Calendar.MINUTE);`	*GregorianCalendar.**MINUTE***
`int secondi1 = dataA.get(Calendar.SECOND);`	*GregorianCalendar.**SECOND***
`*/`	
`int dow = dataA.get(Calendar.DAY_OF_WEEK);`	**DAY_OF_WEEK**
`int dom = dataA.get(Calendar.DAY_OF_MONTH);`	**DAY_OF_MONTH**
`int doy = dataA.get(Calendar.DAY_OF_YEAR);`	**DAY_OF_YEAR**
`System.out.println("Giorno della settimana " + dow);`	**Giorno della settimana …**
`System.out.println("Giorno del mese " + dom);`	**Giorno del mese …**
`System.out.println("Giorno dell'anno " + doy);`	**Giorno dell'anno …**
`System.out.println("Data corrente "+giorno1+'/'+mese1+'/'+anno1);`	**Data corrente …**
`System.out.println("Ora corrente "+ore1+':'+minuti1+':'+secondi1);`	**Ora corrente …**
`Calendar data2 = Calendar.getInstance();`	Creare un oggetto della **classe astratta Calendar** che contenga la data e l'ora corrente
	Utilizzare il **metodo get()** passando come paramentro una delle **costanti statiche** definite nella classe:
`int anno2 = data2.get(Calendar.YEAR);`	**YEAR**
`int mese2 = data2.get(Calendar.MONTH) + 1;`	**MONTH:** i mesi partono da 0
`int giorno2 = data2.get(Calendar.DATE);`	**DATE**
`int ore2 = data2.get(Calendar.HOUR);`	**HOUR**
`int minuti2 = data2.get(Calendar.MINUTE);`	**MINUTE**
`int secondi2 = data2.get(Calendar.SECOND`	**SECOND**

`System.out.println("Data corrente " + giorno2 + '/' + mese2 + '/' + anno2);`	**Data corrente …**
`System.out.println("Ora corrente " + ore2 + ':' + minuti2 + ':' + secondi2);`	**Ora corrente …**
`}`	

La classe **GregorianCalendar** implementa dei metodi per effettuare confronti ed operazioni con le date. Di seguito sono riportati alcuni esempi.

Esempio: costruire un metodo per manipolare le date

Codice JAVA	Descrizione/output

`package esempio;`
 	
`import java.util.Calendar;`
`import java.util.GregorianCalendar;`	

`public class Prova_Date {`	
`public static String dataMan(Calendar data){`	**Metodo** per la manipolazione di una data

`int anno = data.get(Calendar.YEAR);`
 	
`int mese = data.get(Calendar.MONTH) + 1;`
 	
`int giorno = data.get(Calendar.DATE);`
 	
`String dataASS = "" + giorno + '/' + mese + '/' + anno;`	

`return dataMan;`
 	
`}`	

`public static void main(String args[]){`
`Calendar data = new GregorianCalendar(2008, 11, 18);`
`System.out.println("Data scelta " + dataMan(data));`
`data.add(Calendar.DATE, +33);`
`System.out.println("Data " + dataMan(data));`
`data.add(Calendar.YEAR, -1);`
`System.out.println("Data " + dataMan(data));`
`data.roll(Calendar.MONTH, +1);`
`System.out.println("Data " + dataMan(data));`
	Data scelta 18/12/2008
	Data 20/1/2009
	Data 20/1/2008
	Data 20/2/2008

`}` `}`	

Esempio: confronto di due date

Codice JAVA	Descrizione / output
`Calendar data1 = new GregorianCalendar(2008, 11, 18);` `Calendar data2 = new GregorianCalendar(2007, 11, 10);`	Creazione ed inizializzazione di due oggetti di **classe GregorianCalendar**

`/*`
`GregorianCalendar data1 = new GregorianCalendar(2008, 11, 18);`
`GregorianCalendar data2 = new GregorianCalendar(2007, 11, 10);`
`*/`
	Equivalente alle precedenti

`if (data1.before(data2))`
`{`
`System.out.println("Data1 precede Data2");`
`}`
	Data1 precede Data2

`if (data1.after(data2))` `{`	Data1 segue Data2

`  System.out.println("Data2 precede Data1");` `}`	

`if (data1.equals(data2))`
`{`
`  System.out.println("Le date sono uguali");`
`}`
	`Le date sono uguali`

Esempio: somma e sottrazione con le date

Codice JAVA	Descrizione
`Calendar data = new GregorianCalendar(2008, 11, 18);`	Creazione ed inizializzazione di un oggettO di **classe GregorianCalendar**
`data.add(GregorianCalendar.DATE, +33);`	Aggiungere 33 giorni
`data.add(GregorianCalendar.YEAR, -1);`	Togliere 1 anno
`data.roll(GregorianCalendar.MONTH, +1);`	Come Add, ma non modifica altri campi come DAY…

Esempio: convertire i millisecondi

Codice JAVA	Descrizione

`long milliseconds1 = data1.getTimeInMillis();`
`long milliseconds2 = data1.getTimeInMillis();`
`long diff = milliseconds2 - milliseconds1;`
	Differenza in millisecondi

`long diffSeconds = diff / 1000;`	Differenza in secondi
`long diffMinutes = diff / (60 * 1000);`	Differenza in minuti
`long diffHours = diff / (60 * 60 * 1000);`	Differenza in ore
`long diffDays = diff / (24 * 60 * 60 * 1000);`	Differenza in giorni

Per semplificare la gestione delle date è possibile utilizzare la classe **SimpleDateFormat** (**java.text.SimpleDateFormat**).

Tabella riepilogativa su come comporre i **pattern**:

Lettere	Componenti		Presentazione	Esempi
	Data	Tempo		
G	**Era designator**		**Text**	**AD**
y	**Year**		**Year**	**1996; 96**
Y	**Week year**		**Year**	**2009; 09**
M	**Month in year**		**Month**	**July; Jul; 07**
w	**Week in year**		**Number**	**27**
W	**Week in month**		**Number**	**2**
D	**Day in year**		**Number**	**189**
d	**Day in month**		**Number**	**10**
F	**Day of week in month**		**Number**	**2**
E	**Day name in week**		**Text**	**Tuesday; Tue**
u	**Day number of week**		**Number**	**1**

	(1 = Monday, ..., 7 = Sunday)			
a		**Am/pm marker**	**Text**	**PM**
H		**Hour in day (0-23)**	**Number**	**0**
k		**Hour in day (1-24)**	**Number**	**24**
K		**Hour in am/pm (0-11)**	**Number**	**0**
h		**Hour in am/pm (1-12)**	**Number**	**12**
m		**Minute in hour**	**Number**	**30**
s		**Second in minute**	**Number**	**55**
S		**Millisecond**	**Number**	**978**
z		**Time zone**	**General time zone**	**Pacific Standard Time; PST; GMT-08:00**
Z		**Time zone**	**RFC 822 time zone**	**-0800**
X		**Time zone**	**ISO 8601 time zone**	**-08; -0800; -08:00**

Esercizi svolti

Formattare e convertire date

```
package esempio;

// Librerie da importare
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Prova_Date
{
 public static void main(String args[])
 {
   // Data odierna
   Calendar cal9 = new GregorianCalendar();

   // Formattare oggetti data con la classe SimpleDateFormat
   SimpleDateFormat formato9 =  new SimpleDateFormat("dd/MM/yyyy");

   // Applicazione del formato alla data odierna
   System.out.println("(1) " + formato9.format(cal9.getTime()));

   // Data in formato String
   String dataS9 = "02/03/2008";
   try {
     // Parsing Date da String a Date
     Date data9 = formato9.parse(dataS9);
     System.out.println("(2) " + data9);
     System.out.println("(3) " + formato9.format(data9));
```

```
      System.out.println("(4) " + formato9.format(data9.getTime()));
    }
    catch (ParseException e) {
      e.printStackTrace();
      System.out.println("Non convertibile " + dataS9);
    }
  }
}
OUTPUT:
(1) 30/06/2014
(2) Sun Mar 02 00:00:00 CET 2008
(3) 02/03/2008
(4) 02/03/2008
```

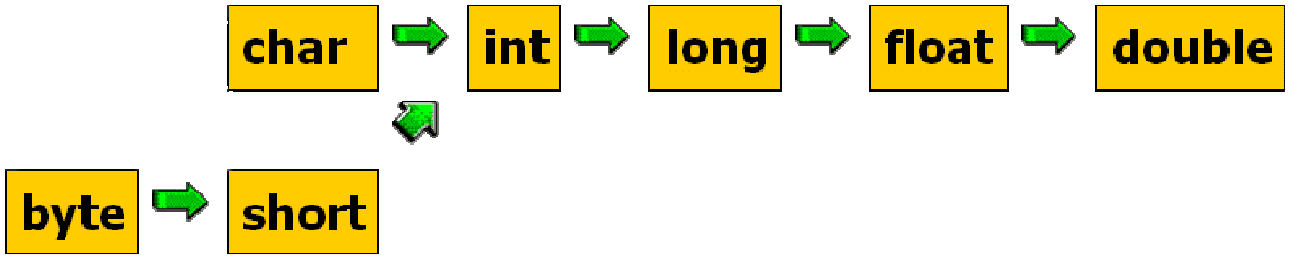
Esempio: convertire una data dal formato americano in quello italiano

```
SimpleDateFormat formatterIT = new SimpleDateFormat("dd/MM/yyyy");
SimpleDateFormat formatterEN = new SimpleDateFormat("yyyy/MM/dd");

Date dataIT;
try{

     dataIT        = formatterEN.parse("2012/04/05″);
     String dataEN = formatterIT.format(dataIT);
     System.out.println(dataIT);
     System.out.println(dataEN);
}
catch   (ParseException e){
     e.printStackTrace();
}
```

Casting e conversioni

Il casting

Le **operazioni di cast** sono fondamentali in ogni linguaggio di programmazione, così come nei linguaggi più diffusi, anche in Java esistono le conversioni di tipo che vengono fatte automaticamente ed altre che richiedono un cast esplicito. Con il termine **cast** si intende l'operazione di **passaggio di una variabile da un tipo di dato ad un altro**. Vediamo lo schema di conversione dei casting che possono avvenire in maniera esplicita:



Se nello schema c'è una **freccia** viene fatto un **casting implicito**, ovvero è possibile evitare di scrivere il cast, lo inserirà il compilatore automaticamente.

Si può notare che il tipo **boolean** non fa parte dello schema, ciò significa che il tipo **boolean non può essere mai convertito**.

Per i **numeri interi**, ovvero i numeri veri e propri, che rientrano nel range di valore è possibile **evitare** di scrivere un **casting esplicito**.

Esempio: casting implicito

Codice JAVA	Descrizione

`byte a = 5;`	`5 rientra nel range di tipo byte, è quindi possible evitare di scrivere il casting esplicito`
`int b = a;`	`Esiste un percorso nello schema da` **`a`** `a` **`b`** `quindi non c'è bisogno di un casting esplicito`
`int c = 5;`	
`long d = c;`	
`float e = 1.45;`	`Possibile conversione con perdita di dati da double a float`
`float e = (float) 1.45;`	`Il casting` **`esplicito`** `è indispensabile!!`
`double f = 1.45;`	

Se non esiste un percorso nello schema proposto, abbiamo bisogno di fare un **casting esplicito** (con probabile perdita di dati).

Esempio: casting esplicito

Codice JAVA	Descrizione
`int a = 5;` `byte b = a;`	**Errore**, `non esiste un percorso nel grafo`
`byte c = (byte) a;`	`Ok, non c'è neanche perdita di dati`
`byte d = (byte) 1000;`	`Ok, ma con perdita di dati (1000 non rientra nel range del tipo byte)` `Il casting` **`esplicito`** `è indispensabile!!`

Cast implicito nelle operazioni aritmetiche

In una espressione aritmetica, **le variabili sono automaticamente elevate ad una forma più estesa** per non causare perdita di informazioni. Se uno degli operandi è byte, float o char il risultato, gli operandi vengono trasformati in tipo int (tranne con l'operatore ++ e –); se uno dei due operanti è double anche l'altro viene trasformato in double; se uno dei due è float anche l'altro viene trasformato in float; stesso discorso per il tipo long; altrimenti vengono trasformati in int.

Porebbe essere utile, durante lo sviluppo di un'applicazione, convertire un determinato dato in un altro tipo per compiere delle altre operazioni. Ciò è possibile utilizzando la tecnica del **casting**; vediamone un esempio:

Esempio: casting implicito

Codice JAVA	Descrizione
`float variabilefloat = 2.5;` `int i = (int) variabilefloat;`	`converte variabilefloat in un intero escludendo i decimali`

Conversioni di stringhe in tipi primitivi

Da String in tipo numerico

La conversione da stringa in tipo primitivo va fatta "a mano". Più precisamente, utilizzando i metodi di conversione **.parseXXX()** che sono metodi statici delle classi che corrispondono ai tipi primitivi.

L'operazione di "**parsing**" [7], consiste nel leggere il valore di un oggetto per convertirlo in un altro tipo.

Ad esempio si può avere una stringa con un valore di "81". Internamente la stringa contenente i caratteri '8 'e '1' e non il numero 81.

Esempio: parsing da String in tipo numerico

[7] L'operazione di "parsing" consiste nel leggere il valore di un oggetto per convertirlo in un altro tipo. Il "parsing" è il processo di rilevamento dei dati (testo) e la determinazione del loro significato sulla base di un insieme di regole.

Codice JAVA	Descrizione

`int i = Integer.parseInt("55");`
`long l = Long.parseLong("924358725");`
`float f = Float.parseFloat("3.14");`
`double d = Double.parseDouble("7.123434534");`
	Parsing esplicito di stringhe in numeri di vario tipo

`boolean b = Boolean.parseBoolean("true");`	**Parsing** di stringa in boolean
`String stringa10 = "10";` `Integer risultato= 20 + stringa10;`	Questo **NON funziona** in quanto non è possibile sommare un numero intero ad una stringa
`Integer risultato= 20 + Integer.parseInt(stringa10);`	30
`String unaStringa= "123";` `int i = Integer.parseInt(unaStringa);`	123
`int riv = 1789;` `String riv_string = riv;`	**Tipi incompatibili**: è richiesto un tipo int
`int riv = 120;` `String riv_string = "" + riv;`	**Parsing implicito**
`double pi = "3.1415926";`	**Tipi incompatibili**: è richiesto un tipo double
`double pi = Double.parseDouble("3.1415926");`	**Parsing esplicito** corretto
`int miliardo = Integer.parseInt("1000000000");`	**Parsing esplicito** corretto

Come appare chiaramente dall'esempio, esistono vari metodi che effettuano parsing numerici, analizziamo i due più usati:

- Il metodo **Integer.parseInt** analizza una stringa per trovare i numeri interi, quindi prende il valore dalla stringa e restituisce un numero (intero) vero e proprio.
- Il metodo **Double.parseDouble** analizza una stringa per trovare i numeri in doppia precisione, quindi prende il valore dalla stringa e restituisce un numero (in dopppia precisione) vero e proprio.

Da String in char

Per ottenere **un caratterere di una stringa** si usa il metodo della classe String **charAt:**

- stringa.**charAt**(int posizione)

Questo metodo restituisce il carattere nella posizione specificata. Attenzione che la prima posizione è 0.

Esempio: charAt

Codice JAVA	Descrizione
`String s = "pippo";` `char c1 = s.charAt(0);`	primo carattere della stringa "s"(posizione 0)
`int lunghezza = s.length();` `char c2 = s.charAt(lunghezza - 1);`	ultimo caratterere della stringa "s" (ultima posizione = lunghezza stringa meno uno)

Conversioni di tipi primitivi in stringhe

Da tipi numerici in String

La conversione da tipo numerico in stringa avviene in automatico, come facilitazione da parte del compilatore, vediamo qui di seguito alcuni esempi.

Esempio: conversione di tipi numerici in String

Codice JAVA	Descrizione
`int a = 7;` `String messaggio = "La variabile a vale "+a;`	**Modo meno efficiente**: la concatenazione delle stringhe fa sì che il codice generato dal compilatore prima crea un StringBuilder, poi fa l'append dei valori, poi ottiene un String dallo StringBuilder.
`int n=737;` `String s = "" + n;`	

`String s = String.valueOf(n);`	**`valueOf di String`** fa nient'altro che chiamare **`Integer.toString()`**
`String s = Integer.toString(n);`	**`Modo più efficace`** ("diretto")

Operatori matematici e di confronto

Per poter effettuare delle operazioni tra le variabili dobbiamo utilizzare dei caratteri "speciali" che prendono il nome di **operatori**.
Gli operatori si dividono in operatori aritmetici, di assegnazione, logici e di confronto. Di seguito elenchiamo gli operatori principali e di più comune utilizzo.

Operatori aritmetici	Descrizione	Esempio	Risultato
+	Addizione	• x = 2 y = 2 x + y	• 4
-	Sottrazione	• x = 5 y = 2 x - y	• 3
*	Moltiplicazione	• x = 5 y = 4 x * y	• 20
/	Divisione	• 15 / 5 5 / 2	• 3 2.5
%	resto divisione tra interi	• 5 % 2 10 % 8 10 % 2	• 1 2 0
++	Incremento (equivalente a: x = x + 1)	• x = 5 x++	• x = 6
--	Decremento (equivalente a: x = x - 1)	• x = 5 x--	• x = 4
Operatori di assegnazione	**Descrizione**	**Esempio**	**Risultato**
=	assegnazione	x = y	x = y
+=	somma e assegnazione	x += y	x = x + y
-=	sottrazione e assegnazione	x -= y	x = x - y
*=	moltiplicazione e assegnazione	x *= y	x = x * y
/=	divisione e assegnazione	x /= y	x = x / y
%=	modulo e assegnazione	x %= y	x = x % y
Operatori logici	**Descrizione**	**Esempio**	**Risultato**
&&	AND logico e	x = 6 y = 3 (x < 10 && y > 1)	true
||	OR logico o	x = 6 y = 3 (x == 5 || y == 5)	false
!	not contrario	x = 6 y = 3	true

	non	!(x == y)	
Operatori di confronto	**Descrizione**	**Esempio**	**Risultato**
==	è uguale a	5 == 8	false
!=	diverso da	5 != 8	true
>	maggiore di	5 > 8	false
<	minore di	5 < 8	true
>=	maggiore o uguale a	5 >= 8	false
<=	minore o uguale a	5 <= 8	true

Gli **operatori matematici** presentati in questa tabella consentono solo **operazioni** di **base**. Nel caso si volesse, ad esempio, **elevare** un numero **a potenza** allora sarebbe necessario utilizzare il metodo **pow(double a, double b)** della classe **java.lang.math**. Il package math ci mette a disposizione **le più importanti funzioni matematiche** come per esempio seno, coseno, tangente e molto altro.

E' possibile concatenare tra loro diversi Operatori per avere risultati diversi come è visibile anche nella tabella esemplificativa.

Esempio: operatori matematici e di confronto

Codice JAVA	Descrizione
`int x = 1;`	`= operatore di assegnazione`
`int y = x + 1;`	`y è uguale a 2`
`y += 10;`	`y diventa 12`

`If (x == y){`
`  System.out.println("Primo risultato FALSO");`
`}`
`else if (y > x){`
`  System.out.println("Secondo risultato VERO");`
`}`	

In questo esempio abbiamo introdotto, per vedere il funzionamento degli operatori di confronto, una sintassi del tipo **if...else** che rappresenta l'esempio più comune di struttura condizionale (*vedi if*) .

Istruzioni condizionali

In questa sezione analizziamo cosa sono e come possiamo scrivere le **istruzioni condizionali**.

La selezione: il costrutto if... else if... else

La struttura tipica del costrutto **if** è la seguente:

IF ad una via	

`if (condizione1) {`
`  ...`
`}`
	`if (condizione1) {`
	`  ...`
	`}`

`if (condizione1) ...;`	`if (condizione1) ...;`
IF a più vie	

`if (condizione1) {`
`  ...`
`  ...`
`}`
	`if (condizione1) {`
	`  ...`
	`  ...`
	`}`

`else if (condizione2) {`
`  ...`
`  ...`
`}`
	`else if (condizione2) {`
	`  ...`
	`  ...`
	`}`

`else{` `  ...`	`else{` `  ...`

... }	... }

Esempio: if ad una via

Codice JAVA	Descrizione
int x = 1;	= operatore di assegnazione
int y = x + 1;	y è uguale a 2
y += 10;	y diventa 12
	if è seguito dalla condizione: "x" uguale **(==)** ad "y". **La condizione è sempre tra parentesi**

if (x == y){
System.out.println("Primo risultato FALSO");
}
	I due esempi di **if ad una sola via** sono **equivalenti;** nel secondo caso sono state **omesse** le **parentesi graffe.**

if (x == y)System.out.println("Primo risultato FALSO");	

Esempio: if a due vie

Codice JAVA	Descrizione
int x = 1;	= operatore di assegnazione
int y = x + 1;	y è uguale a 2
y += 10;	y diventa 12

if (x == y){
 	
System.out.println("Primo risultato FALSO");
}
	if a due via

else if (y > x){	
System.out.println("Secondo risultato VERO"); }	La condizione **else if**, indica un'alternativa determinando un'ulteriore condizione (se "y" è maggiore di "x"). Anche **else if** deve essere seguita da un **blocco di graffe**, nel caso in cui **le istruzioni** da eseguire siano **maggiori di una**.

Quando il confronto avviene fra due stringhe allora le cose cambiano, **NON è possibile utilizzare "=="** ma, come già detto, servono invece alcuni **metodi speciali** della **classe String** (*vedi Confronto fra stringhe*) come **equalsTO**() e **compareTo**():

Esempio: uso del metodo compareTo() :

Codice JAVA	Descrizione
String stringa1 = "Paperino"; String stringa2 = "Pippo";	Definizione ed inzializzazione stringhe

if(stringa1.**compareTo**(stringa2)> 0)
System.out.println("La prima stringa segue la seconda");
}
else if (stringa1.**compareTo**(stringa2)< 0){
System.out.println("La prima stringa precede la seconda");
}
	if a due via

La selezione multipla: il costrutto switch

Per descrivere tante condizioni per una stessa variabile la soluzione migliore risulta l'uso del costrutto **switch** la cui struttura tipica è la seguente:

switch (variabile) { **case** valore1: ... ... **break**;	In questo blocco switch troviamo n casi, determinati dalla parola chiave **case** e seguiti dal valore del caso e dai due punti.

`case valore2:`
`...`
`...`
`break;`
	Dopo tutte le istruzioni prima dell'ultima è indispensabile l'istruzione **break** che indica al compilatore di non tenere presente cosa c'è scritto sotto e permette di andare avanti. L'uso del break permette quindi di uscire dallo switch.

`...` `...`	

`case valoren:`
`...`
`...`
`break;`	

`default:`
`...`
`...`
`}`
	L'ultimo caso non ha bisogno della parola chiave *case* perchè è la scelta predefinita (**default**), nel caso in cui "k" non sia uguale a nessuno dei valori indicati in precedenza dopo la parola **case**.

Esempio: switch

Codice JAVA	Descrizione / output

`class ProvaSwitch{`
`  public static void main(String args[]){`
`  int k = 10;`
`  switch(k){`	

`  case 5:`
`  System.out.println("Il valore scelto è k = 5");`
`  break;`
	Se k è uguale a 5…

`  case 10:`
`  System.out.println("Il valore scelto è k = 10");`
`  break;`
	Se k è uguale a 10…

`  case 15:`
`  System.out.println("Il valore scelto è k = 15");`
`  break;`
	Se k è uguale a 15…

`  default:` `  System.out.println("Il valore scelto è default");`	In nessuno dei casi precedenti…

`  }`
`  }`
`}`	

	Il valore scelto è k = 10

I cicli

In Java esistono due tipi di **cicli** (operazioni ripetute più di una volta): il ciclo definito o ciclo **for** ed il ciclo indefinito o ciclo **while**.

Il ciclo "definito" - FOR

Il ciclo **for** viene utilizzato quando si vuole eseguire delle operazioni un numero determinato (già conosciuto) di volte. La sintassi del ciclo **for** è la seguente:

```
for (assegnazione variabile; limite massimo; incremento){
    :
}
```

Quindi andiamo a determinare un valore iniziale della variabile nell'assegnazione, un valore massimo determinato spesso da una condizione e un incremento o decremento in base alle necessità.

Esempio: ciclo FOR

Codice JAVA	Descrizione / output
`class ProvaFor1 {` `  public static void main(String args[]) {`	

`  for (int i=0; i<10; i++) {`
`  System.out.println(i);`
`  }`
	Primo parametro del ciclo **for**:
	• **int i=0:** dichiarato come intero "i" ed inizializzato con il valore di 0
	• **i<10:** limite massimo impostato in modo

	da eseguire la stessa operazione finché "i" è minore di **10** (cioè fino a 9) • **i++:** il valore di "i" è incrementato ad ogni passaggio di 1.
} }	
	Come risultato avremo dieci righe con i numeri da 0 a 9

class ProvaFor2 {
public static void main(String args[])
{	

for (int i=0, j=2; **i<10**; i++, j++) {
System.out.println("i=" + i + " e j=" + j);
}
	In questo caso utilizziamo le **due variabili locali** "**i**" e "**j**" che si incrementano ma hanno valori differenti: infatti il valore iniziale di "i" è 0 mentre quello di "j" è 2, quindi "i" terminerà a 9 mentre "j" terminerà a 11.

} }	

E' possibile anche inserire un for all'interno di un altro for attenendo così due **for annidati o nidificati**.

Un esempio è mostrato nel blocco di codice sottostante che mostrerà la tavola pitagorica.

Esempio: cicli FOR nidificati

Codice JAVA	Descrizione
class ProvaFor2{ public static void main(String args[]) {	Inserimento di **un ciclo dentro un altro** dove:
for (int i=1; i<11; i++) {	ogni valore di "i" va da 1 a 10
for (int j=1; j<11; j++) {	ogni valore di "j" va da 1 a 10
System.out.print(i + " x " + j + " = " + **i*j** + " ");	ogni valore di "j" viene moltiplicato per il numero "i".
}	Ricomincia il prossimo valore di "j"
System.out.**println**();	Al termine del ciclo di "j", si scrivie un ritorno a capo
}	Ricomincia il prossimo valore di "i"
} }	

Il ciclo "indefinito" - WHILE

I cicli **while** (ovvero **ripeti se la condizione è vera**) sono differenti dai cicli for, non si usano le variabili con la formula dell'incremento, ma si eseguono fin quando una condizione appare falsa. Vediamo subito un piccolo esempio:

Esempio: ciclo WHILE

Codice JAVA	Descrizione

class ProvaWhile1 {
public static void main(String args[]) {
int i = 0;
while (i<10) {
System.out.println(i);
++i;
}
}
	i++ imposta prima il valore di i e poi lo incrementa
	++i prima incrementa e poi imposta il valore

```
}
```

Questo esempio visualizza il valore di "i" fino a quando "i" non diventa 10, in questo caso abbiamo dovuto dichiarare e assegnare prima il valore di "i" e, dentro il blocco **while**, abbiamo incrementato il suo valore. Prima di eseguire le istruzioni contenute nel blocco il compilatore controlla che la condizione sia vera.

Il ciclo "indefinito" e le stringhe

Se il ciclo while prevede la presenza di una condizione che intende verificare l'uguaglianza fra stringhe sarà necesario utilizzare **equals()** o **equalsIgnoreCase()**, **compare()** o **compareIgnoreCase()**. (*vedi controllo stringhe*).

Esempio: uso del ciclo WHILE con equalsIgnoreCase

```
class ProvaWhile3 {
    public static void main(String args[]) {
       :
       :
       String risposta = new String();
       risposta = "S";
       while (risposta.equalsIgnoreCase("S")) {
          :
          :
          System.out.println("Vuoi continuare (S/N) ?  ");
          risposta = MioInput.readLine();
       }
    }
}
```

Il ciclo "indefinito" – DO... WHILE

Utilizzando la sintassi con **do**, è possibile anche definire un **ciclo indefinito** in grado di eseguire **almeno una volta** le istruzioni dentro il blocco.

Esempio: ciclo DO... WHILE

```
class ProvaWhile2 {
   public static void main(String args[]) {
       int i = 0;
       do {
           System.out.println(i);
           ++i;
       }
       while (x<10);
   }
}
```

Le istruzioni break e continue

- Le istruzioni **break** e **continue** servono ad ottimizzare il costrutto di selezione **if... else** ed i cicli iterativi **for** e **while**. In particolare:
- l'istruzione **break** interrompe un blocco di istruzioni e **provoca un salto** alla prima istruzione successiva al blocco
- l'istruzione **continue** interrompe un blocco di istruzioni contenuto in un'iterazione e **determina il salto** dell'esecuzione dell'**iterazione successiva**.

Le linee guida della programmazione strutturatane limitano l'uso a casi di assoluta necessità (per esempio l'uso di break nell'istruzione switch) , perchè introducono nel programma dei salti, l'effetto dei quali, nei casi più complessi, può essere davvero diifficile valutare con sicurezza.

Esercizi svolti

Calcolare il massimo tra i divisori di un numero n inserito da tastiera

```
import java.io.*;
class massimoDivisore
{
  public static void main (String args[])
  {
     int i, n;
     InputStreamReader mioIn = new InputStreamReader(System.in);
     Bufferreader mioTasteiar = new BufferReader(mioIn);
     try
     {
        System.out.print("\ inserisci il numero: ");
        n = miaTastiera.readLine();
        i = n-1;
        while(i>1)
        {
           if (n%i == 0) break;
           i--;
        } // fine while
        System.out.println("il divisore più grande è: " + i);
      } // fine try
      catch (Exception e)
      {
        System.out.println("si è verificata un\' eccezione");
      } // fine catch
  } // fine main
} // fine classe
```

Visualizzare tutti i divisori di un numero n, inserito da tastiera

```
import java.io.*;
class stampaDivisori
{
  public static void main (String args[])
  {
     int i, n;
     InputStreamReader mioIn = new InputStreamReader(System.in);
     Bufferreader mioTasteiar = new BufferReader(mioIn);
     try
     {
        System.out.print("\ inserisci il numero: ");
        n = miaTastiera.readLine();
        i = 2;
        while(i<n)
        {
           if (n%i == 0)
           {
            i++;
            continue;
           }
           System.out.println("il divisore è: " + i);
           i++;
        } // fine while

      } // fine try
      catch (Exception e)
      {
        System.out.println("si è verificata un\' eccezione");
      } // fine catch
  } // fine main
} // fine classe
```

Gli array[8]

Array ad una dimensione

In generale un array è una sequenza di locazioni di memoria, che contengono entità dello stesso tipo, ed a cui si può fare riferimento con un nome comune. Le entità che compongono un array prendono il nome di elementi dell'array, **ogni elemento** di un **array** è identificato dal **nome dell'array** e da un **indice**. Anche in Java un **vettore** permette di utilizzare un elenco di elementi ed è **l'istanza della classe array**.

La notazione più comune, adottata anche da Java, fa uso delle parentesi quadre:

```
tipo nome[indice]
tipo[indice] nome
```

Per utilizzare un array dobbiamo passare attraverso tre fasi, come per gli oggetti:

- dichiarazione,
- creazione,
- inizializzazione.

E'importante ricordare che gli **indici** degli array partono da **zero.**

Array di tipo primitivo

Dichiarazione di array

Prima di tutto **dichiariamo l'array** inserendo una **coppia di parentesi quadre** prima oppure dopo l'identificatore.

```
int num[];
int[] num;
char alfabeto[];
char[] alfabeto;
```

La presenza delle parentesi quadre indica che:

- non stiamo dichiarando una variabile di tipo intero ma un array di interi nè una di tipo carattere ma un array di caratteri,
- non abbiamo creato degli array ma semplicemente dei riferimenti ad array, infatti non è stata specificata la dimensione.

Creazione di array

Per **creare un'array** usiamo l'operatore **new**, specificando la dimensione, questa operazione è **obbligatoria**.

Codice JAVA	Descrizione
`num = new int[50];`	Abbiamo **creato un array di 50 interi** con "num" come riferimento ad esso
`alfabeto = new char[21];`	Abbiamo **creato un array di 21 caratteri** con "alfabeto" come riferimento ad esso

Un array è una classe speciale di Java ed in quanto tale possiede una istanza leggermente diversa da quelle degli altri oggetti.

Tips and tricks – Dimensione di un array

[8] IT: vettori

La dimensione "n" di un array può essere stabilita solo al momento della creazione e non può essere più cambiata!

Inoltre, se si cerca di accedere ad un elemento fuori dall'intervallo definito [0, n-1] o ad un elemento con indice NON intero, come in questi casi:

```
num = new int[50];
num[51];
num[-1];
num[1.5];

alfabeto = new char[21];
alfabeto[23];
```

il programma si blocca perchè si sta infatti cercando di accedere in modo errato alla memoria!

Il runtime Environment di Java segnala un'eccezione di tipo **ArrayIndexOutOfBoundsException**.

Inizializzazione di array

Per inizializzare un array dobbiamo inizializzarne ogni singolo elemento A questo punto possiamo accedere agli elementi dell'array:

```
num[5] = 18;
int n = num[5]; // assegnamo ad una variabile l'elemento del vettore che è stato inizializzato
```

Avendo creato un array di 50 elementi gli indici validi vanno da 0 a 49.

```
alfabeto[0] = 'a';
alfabeto[1] = 'b';
alfabeto[2] = 'c';
:
alfabeto[20] = 'z';
```

Dichiarazione, creazione e inizializzazione di array

Le operazioni sopra descritte non devono necessariamente essere eseguite separatamente ma possono anche essere eseguite "contemporaneamente".

- Dichiarazione e creazione di array
  Dichiarazione e creazione possono essere utilizzate nella stessa istruzione:

```
int num = new int[50];
char alfabeto = new char[21];
```

  Possiamo anche dichiarare due variabili array dello stesso tipo e attribuirgli due dimensioni diverse:

```
int[] a,b;
a = new int[50];
b = new int[30];
```

- Dichiarazione, creazione e inizializzazione di array
  Possiamo eseguire tutti i passi necessari per utilizzare un array tramite una particolare sintassi:

```
char alfabeto[] = {'a', 'b', 'c', ..., 'z'};
double[] a = {2.5,7.8,11.0};
```

L'elenco di numeri tra parentesi graffa prende il nome di inizializzatore e consente in un colpo solo di creare l'array, definendone la dimensione dell'array, e di attribuire i valori iniziali dei suoi elementi. E' solo una scorciatoia, l'effetto è esattamente quello della serie di istruzioni dette in precedenza.

L'attributo length

Ogni oggetto array contiene, oltre alla collezione di elementi, una **variabile di istanza costante pubblica** chiamata **length** che memorizza il numero di elementi in esso contenuti.

```
int n;
n = a.length; // n  vale 50
n = b.length; // n  vale 30
```

Accedendo alla variabile length è possibile risalire al numero di elementi contenuti in un array.

```
double[] v;
v = new double[5];
```

```
System.out.println(v.length); // stampa 5
```

Attenzione: l'attributo length è di sola lettura, non è possibile assegnargli un valore, infatti, come abbiamo detto, la dimensione di un array non può cambiare dopo la creazione.

```
a.length = 60; // Errore di compilazione
```

Vediamo ora un esempio riepilogativo con creazione di un array di 3 numeri reali ed assegnazione di un valore:

```
double[] num;
num = new double[3];
num[0] = 2.5;
num[1] =  7.8;
num[2] = 11.0;
```

Vediamo un esempio da cui è possibile rilevare che dopo la creazione dell'oggetto array associato ad una variabile array è possibile accedere ai singoli elementi della collezione. Questi elementi saranno inizializzati al valore di default del tipo corrispondente (per i numeri è zero).

Codice JAVA	Descrizione
`int[] a;`	a è una variabile di tipo array di interi
`a = new int[5];`	creazione di un oggetto array con 5 elementi di tipo int associato alla variabile array a
	Accesso ad un singolo elemento di un array:
`a[0] = 23;`	assegnazione del primo elemento dell'array
`a[4] = 92;`	assegnazione dell'ultimo elemento dell'array
`a[5] = 16;`	quando l'istruzione viene eseguita si ha la segnalazione di un errore **perché l'indice 5 non è nell'intervallo [0,4]**

Esempio: convertire un array

Codice JAVA	Descrizione
`String[] numbers = {"385","556"};`	Array di stringhe
`int numberList[] = new int[numbers.length];`	Creazione array formato dallo stesso numero di elementi dell'array di stringhe

`for (int i = 0; i < numbers.length; i++)`
`{`
`  numberList[i] = Integer.parseInt(numbers[i]);`
`}`
	Convertire ogni elemento in formato numerico

Esempio: somma degli elementi di un array di interi

Codice JAVA	Descrizione
`public static int sommaValoriArray(int[] v)` `{`	Scriviamo un **metodo sommaValoriArray** che prende come parametro un array di interi e **restituisce la somma dei valori contenuti nell'array**

`int somma = 0;`
`for (int i=0; i < v.length; i++){`
`  somma += v[i];`
`}`
	Il **parametro array v** è dichiarato come variabile array: esso contiene un **riferimento ad un array passato come parametro** al momento dell'invocazione del metodo sommaValoriArray.

`return somma;` `}`	Ogni modifica all'array effettuata usando v sarà visibile anche fuori dal metodo **sommaValoriArray**
`public static void main(String[] args){`	**Metodo main**
`int[] x = new int[100];`	Creazione array di 100 elementi
`for (int i=0; i<100; i++){`	
`x[i] = i;`	L'elemento con indice i di x

	contiene i
System.out.println(i + " - " + **sommaValoriArray**(x)); } }	Visualizza la somma dei primi 100 numeri

Esempio: ricerca sequenziale di un elemento in un array

Scriviamo un metodo **cercaArray** che prende come parametro un array di stringhe ed una stringa e restituisce true se quella stringa è presente nell'array, e false altrimenti:

```
public static boolean cercaArray(String[] v, String e) {
    for (int i=0; i<v.length; i++)
        if (e.equals(v[i])) return true;
    return false;
}
```

Esempio: metodo main

Codice JAVA	Descrizione
public static void main(String[] args){	creazione array x di 3 stringhe
String[] x = { "uno", "due", "tre" };	
if (cercaArray(x, "due"))	ricerca della stringa "due" nell'array x
System.out.println("trovato");	stampa "trovato"
else System.out.println("non trovato");	…questa stampa non viene effettuata
}	

Esempio: ricerca del valore massimo in un array

Scriviamo un predicato massimoArray che prende come parametro un array restituisce il massimo valore presente nell'array (si assuma che l'array contiene almeno un elemento):

```
public static long massimoArray(long[] v) {
    int max = 0;
    for (int i=1; i<v.length; i++)
        if (v[i] > v[max]) max = i;
    return v[max];
}
```

Il metodo mantiene un indice max dell'elemento dell'array con valore più grande trovato: inizialmente è zero, come se il massimo fosse in prima posizione nell'array v, ed è eventualmente aggiornato nel ciclo for se si trova un valore più grande.

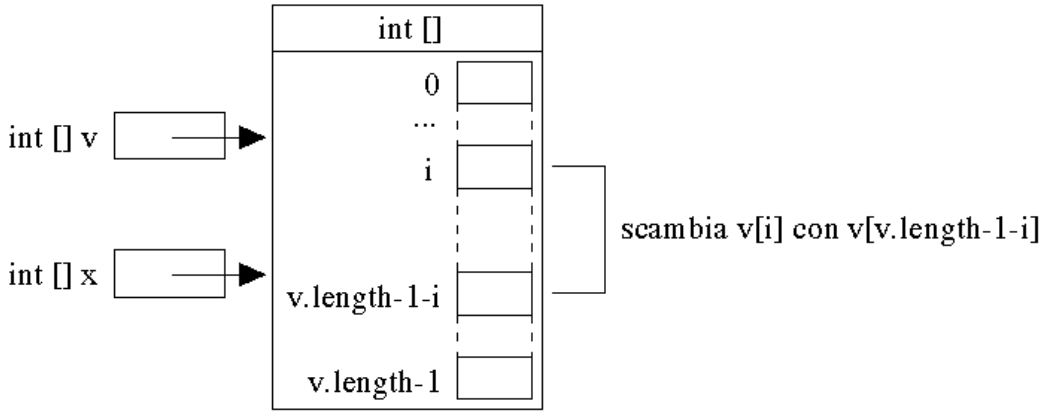
Esempio: crea un array

```
public static void main(String[] args){
    long[] x = { 42, 97, 31 };            // creazione array x di 3 long
    System.out.println(massimoArray(x));  // stampa 97
}
```

Esempio: ordine inverso di un array

Scriviamo un metodo rovesciaArray che prende come parametro un array di interi e lo modifica **invertendone l'ordine degli elementi**:

```
public void rovesciaArray(int[] v) {
    for (int i=0; i < v.length/2; i++) {
        int temp;
        temp = v[i];
        v[i] = v[v.length-1-i];
        v[v.length-1-i] = temp;
    }
}
```



Esempio: metodo main

```
public static void main(String[] args){
    int[] x = { 5, 3, 9, 5, 12};  // creazione array x di 5 elementi
    for (int i=0; i<5; i++){      // stampa 5 3 9 5 12
        System.out.println(x[i]);
    }
    rovesciaArray(x);             // rovescia l'array x
    for (int i=0; i<5; i++) {     // stampa 12 5 9 3 5
        System.out.println(x[i]);
    }
}
```

Si noti che il meccanismo di passaggio dei parametri fornisce come parametro attuale al metodo rovesciaArray il riferimento ad array contenuto in x, che viene copiato nel parametro formale v. Quindi v ed x si riferiscono al medesimo oggetto array (***vedi figura sopra***).

Gli array: ordinamento con le stringhe

Per sapere se due stringhe sono uguali, cioè, sono la stessa sequenza di caratteri oppure quale delle due stringhe viene prima dell'altra, la classe **String** fornisce i metodi d'istanza necessari: **equals** e **compareTo** (*vedi Confronto tra stringhe*)

Qui un semplice programma di ordinamento di un vettore di stringhe. Il programmma usa il metodo **compareTo**( ) per determinare un ordinamento **bubble sort**.

Esempio: boubble sort con compareTo

```
// A bubble sort for Strings.
class SortString {
   static String arr[] = {"Now", "is", "the", "time", "for", "all", "good",
"men",   "to", "come", "to", "the", "aid", "of", "their", "country"};

   public static void main(String args[]) {
       for(int j = 0; j < arr.length; j++) {
           for(int i = j + 1; i < arr.length; i++) {
              if(arr[i].compareTo(arr[j]) < 0) {
                String t = arr[j];
                arr[j] = arr[i];
                arr[i] = t;
              }
           }
           System.out.println(arr[j]);
       }
   }
}
```

L'output di questo programma è questa lista di parole:

Now	men
aid	of
all	the
come	the
country	their
for	time
good	to
is	to

Come si può vedere da questo output, **compareTo** tiene in considerazione le maiuscole e le minuscole. La parola "Now" viene per prima perchè le altre iniziano con una lettera minuscola, questo significa che ha un valore minore nel set di caratteri ASCII.

Esempio: boubble sort

```
for(j=1;j<=n-1;j++)
{
  for(k=j+1;k<=n;k++)
  {
      if(descrizione[j].compareTo(descrizione[k])> 0)
      {
         temp=descrizione[j];
         descrizione[j]=descrizione[k];
         descrizione[k]=temp;
         temp1=q[j];
         q[j]=q[k];
         q[k]=temp1;
         temp1=scorta[j];
         scorta[j]=scorta[k];
         scorta[k]=temp1;
       }
  }
}
```

Per ignorare la differenza fra maiuscole e minuscole durante il confronto fra stringhe, si può usare **compareToIgnoreCase( )**[9], come mostrato qui:

```
int compareToIgnoreCase(String str);
```

Questo metodo restituisce gli stessi risultati di **compareTo**( ), con l'eccezione di **ignorare le differenze fra maiuscole e minuscole**.

Gestire le eccezioni: il blocco try-catch

Si dice eccezione ogni evento inusuale che possa verificarsi durante l'esecuzione di un programma al di fuori del suo comportamento normale o desiderato.

Le eccezioni comprendono sia errori che potrebbero essere fatali al programma, bloccandolo o interrompendolo, sia situazioni insolite ma meno gravi.
Per aumentare la "robustezza" o "tolleranza ai guasti" di un'applicazione, tutti i più recenti linguaggi OOP mettono a disposizione la possibilità di catturare e trattare eccezioni.
In particolare, il programma deve avere la possibilità di "catturare" un'eccezione, riconoscerla e prevedere le necessarie azioni di recupero, almeno per mantenere il programma in uno stato interno consistente.

Il trattamento standard delle eccezioni consiste nel trasmetterle al metodo chiamante, fino eventualmente a raggiungere il metodo avviato per primo, facendolo abortire.

`try`
`{`
`  <codice soggetto ad eccezioni>`
`}`
	Try: codice che potrebbe generare un'eccezione

[9] *Questo metodo è stato aggiunto a partire da Java 2.*

catch (<tipo eccezione> <nome parametro>)
{
  <codice da eseguire se si verifica un eccezione>
}
	Catch: codice di gestione dell'eccezione (visualizzazione di un messaggio d'errore per l'utente, etc.)

[finally *{* *<codice da eseguire comunque>* *}]*	**Finally**: codice **facoltativo** eseguito in ogni caso, anche se, ad esempio, le istruzioni del blocco try o di un blocco catch terminano con un comando **return**; può contenere delle istruzioni che **chiudono dei files** oppure **rilasciano delle risorse**, per garantire la consistenza dello stato.

I costrutti **try-catch-finally** possono essere **annidati** a piacere. Ad esempio, se in un blocco catch o finally venisse generata un'eccezione, si possono racchiudere le corrispondenti istruzioni in un altro blocco try.

Esercizi svolti

Costruiamo ora la classe "**Implementazione**" che contiene il solo **metodo main** che costituisce il punto d'inizio per l'esecuzione del programma.

Quindi all'interno del main saranno presenti le **istanze della classe e tutti le chiamate ai metodi relativi**.

Il **metodo** "**divisione**", che può generare un'eccezione, è stato inserito all'interno del blocco try. Se il metodo non solleva un'eccezione si passerà alla visualizzazione del risultato della divisione, altrimenti si passarà ad eseguire il codice contenuto nel blocco catch.

```
public class Implementazione {
   public static void main(String
args[]) {
      int divisione;
      OperazioniSuNumeri operazioni =
new OperazioniSuNumeri();
      try
      {
         divisione =
operazioni.divisione(5,0);
         System.out.println("Il
risultato della divisione è:" +
divisione);
      }
      catch (Exception e)
      {
         System.out.println(e);
         System.out.println("Non puoi
effettuare una divisione per zero");
      }
      System.out.println("Ho terminato
comunque l'esecuzione del programma");
   }
}

public class OperazioniSuNumeri {
    public numeroX;
    public numeroY;
```

catch (Exception e):
e è un parametro tra parentesi del tipo **Exception** nella quale viene inserito il **messaggio di errore relativo all'eccezione sollevata**

```
java.lang.ArithmeticException: / by zero
```

rappresenta la visualizzazione relativa alla variabile **e** nel caso si faccia una divisione per zero

Visualizzazione del nostro messaggio di errore

Questa riga è la visualizzazione la fine dell'applicazione.

```
    public OperazioniSuNumeri(int x,
int y) {
      numeroX=x;
      numeroY=y;
    }
    public int sottrazione() {
       int sottrazione;
       sottrazione=numeroX-numeroY;
       return sottrazione;
    }
    public int somma() {
       int somma;
       somma=numeroX+numeroY;
       return somma;
    }
    public int moltiplicazione() {
       int moltiplicazione;
       moltiplicazione =numeroX *
numeroY;
       return moltiplicazione;
    }
    public int divisione() {
       int divisione;
       divisione = numeroX / numeroY;
       return divisione;
    }
}
```

Tips and tricks - Exception

Quando si inserisce come parametro della catch il tipo **Exception** si cattura **qualsiasi eccezione** si generi. Questo tipo di eccezione è dunque la **più generale possibile** quindi, nel caso si intenda catturare più eccezioni con la stessa catch (*vedi Multi-catch vs eccezioni separate*) è indispensabile scrivere le eccezioni a partire da quello **meno generale** fino ad arrivare a quello **più generale** che potrebbe appunto essere la **Exception.**

Multi-catch vs eccezioni separate

Le eccezioni possono essere catturare anche utilizzando il costrutto **multi-catch**:

```
public static boolean connessione(){
 boolean connesso = false;
 try {
        Class.forName(driver).newInstance();
        connessione = (Connection) DriverManager.getConnection(url +
dbname,userName,password);
        connesso = true;
 }
 catch (ClassNotFoundException | InstantiationException | IllegalAccessException
| SQLException e){
        connesso = false;
 }
 return connesso;
}
```

Per ottenere risultati migliori è buona norma **gestire le eccezioni catturandole separatamente** e ciò può essere ottenuto utilizzando più blocchi catch uno dopo l'altro. Vediamo adesso come compiere questa miglioria sfruttando sempre la nostra classe Implementazione:

```
try {
   divisione = operazioni.divisione(5,0);
```

```
   System.out.println("Il risultato della divisione è:" + divisione);
}
catch (ArithmeticException exc) {
          System.out.println(exc);
          System.out.println("Non puoi effettuare una divisione per zero");
}
catch (Exception exc) {
         System.out.println("Errore generico");
}
```

In questo modo quando viene generata un'eccezione la JVM andrà a controllare i vari blocchi catch. I blocchi vengono scorsi tutti fino a che non viene trovato quello che gestisce quel particolare tipo di eccezione e quindi viene eseguito il codice all'interno del catch relativo.

Nel nostro esempio dunque se viene generata un'eccezione di tipo AritmeticException verrà eseguito il codice del primo blocco catch. Nel caso in cui avessimo avuto un altro tipo di eccezione dunque la JVM sarebbe passata oltre al catch relativo all'AritmeticException ed avrebbe eseguito il codice all'interno il secondo catch.

L'**ordine** con il quale devono essere scritti i **catch** è importante in quanto devono essere scritti **da quello meno generale a quello più generale**. Infatti nel caso avessimo invertito l'ordine dei due catch nel codice d'esempio l'eccezione sarebbe stata catturata automaticamente dal blocco più generale Exception senza mai raggiungere il catch relativo alla gestione specifica dell'AritmeticException.

Il fatto di poter gestire in maniera separata i vari tipi di eccezione, permette al programmatore di capire immediatamente quale punto del codice ha generato l'eccezione e cosa di preciso è andato storto.

Un altro esempio

```
try
{
 str = MioInput.readLine();
 numero2 = Integer.parseInt(str);
}
catch (IOException e) {
 System.out.println ("Si è verificato un errore: " + e);
 System.exit(-1);
}
```

In questo caso il nostro programma termina, in caso di errore, emettendo il messaggio "Si è verificato un errore..." ed un exit code pari a "-1".

Exit code o codice di errore

Quando un programma termina è previsto, in tutti i sistemi operativi, che tale programma lasci una traccia di sé indicando un numero, che per la sua natura era denominato come "**codice di errore**". Il parametro di exit dovrebbe indicare se l'esecuzione del programma è andata a buon fine o no ed in particolare:

- 0 indica che l'esecuzione è andata a **buon fine**. Questo exit code equivale a **true**;
- 1, -1, qualunque cosa != 0 in dica che è accaduto qualche **errore**, si possono usare diversi valori per diversi tipi di errori. Un exit code diverso da 0 equivale a **false**.

In questo modo, script o altri programmi che avviano un certo programma o l'utente finale stesso, possono sapere se tutto è andato bene o se qualcosa è andato storto. Ovviamente **possiamo gestire questo numero come vogliamo**, a patto di limitare l'eventuale prevista interoperabilità con script e programmi vari[10].

Ci sono molti tipi di eccezioni che si possno verificare durante l'esecuzione di un programma nella tabella seguente ne vediamo alcune:

[10] *Su sistemi Windows a partire da XP è possibile verificare il codice di errore (o di uscita) lasciato da un programma che si è appena chiuso e che è stato avviato in un prompt dei comandi e che rimanga aperto in tale prompt dei comandi (non come le applicazioni solo GUI), direttamente nel Prompt dei comandi stesso, tramite la variabile %ERRORLEVEL%.*

Eccezione	Descrizione
IOException	eccezioni legate all'input/output dei dati
ArrayIndexOutofBounds	eccezioni generate quando si tenta di accedere ad un elemento dell'array andando oltre la dimensione massima
FileNotFound	eccezioni generate quando si tenta di leggere un file che non esiste
EOFException	eccezioni generate quando si tenta di leggere un dato da un file di cui si è già raggiunta la fine
NumberFormatException	eccezioni generate quando si tenta di convertire una stringa in numero Esempio: int Integer.parseInt(String s) L'operazione è critica, perché può avvenire solo se la stringa data contiene la rappresentazione di un intero. Se ciò non accade, parseInt solleva una NumberFormatException
SQLException	Un'eccezione che fornisce informazioni su un errore di accesso al database o altri errori. Ogni SQLException fornisce diversi tipi di informazioni fra cui una stringa che descrive l'errore. Questo viene utilizzato come il messaggio di eccezione Java, disponibile tramite il metodo getMesasge.

Eccezioni controllate

Un'eccezione controllata deve essere raccolta da un **metodo in una clausola catch** o deve essere nella **lista delle clausole throws** di ciascun metodo che possa lanciare l'eccezione o propagarla.

La clausola **throws** deve essere dichiarata nell'intestazione del metodo, es:

```
public static int leggiFile(BufferedReader br) throws java.io.IOException {
```

Il compilatore segnala se un'eccezione controllata non viene gestita correttamente

Esempio: eccezione controllata

```
try {
     sql = "Insert into studenti (cognome, nome, indirizzo, classe, sezione)
values ('"
     + txtcognome.getText()+ "', '" + txtnome.getText()+ "', '" +
txtindirizzo.getText() + "', "
     + Integer.parseInt(txtclasse.getText())+ ", '" + txtsezione.getText()+
"')";
     Utility.dichiarazione.executeUpdate(sql);
     lblMsg.setText("Inserimento riuscito");
     cancella();
 }
 catch(NumberFormatException e1){
     lblMsg.setText("Classe errata " + sql);
 }
 catch (SQLException e){

     lblMsg.setText("Inserimento NON riuscito " + sql);
}
```

Generare eccezioni: l'istruzione throw

Un codice può segnalare una anomalia sollevando una specifica eccezione tramite l'istruzione **throw.**

Esempio: throw

```
throw new IllegalArgumentException("Errore nei parametri");
```

Solitamente un'istruzione throw è inclusa in un'istruzione if che valuta una condizione per verificare se deve essere sollevata l'eccezione

Esempio: throws Exception

```
public static boolean connessione()throws Exception {
    boolean connesso = false;
    try {
        Class.forName(driver).newInstance();
        connessione = (Connection) DriverManager.getConnection(url + dbname);
        dichiarazione = (Statement) connessione.createStatement();
        connesso = true;
    }
    catch (Exception e){
        connesso = false;
    }
    return connesso;
}
:
:
public  FrmMenu() {
    initComponents();
    // --- Configurazione finestra -------------------
    this.setSize(700,350);
    this.setExtendedState(JFrame.NORMAL);
    // --- Posizionare finestra -------------------
    this.setLocationRelativeTo(null);

    try {
      if (!connessione()){
         optMessaggio.showMessageDialog(null, "Connessione NON riuscita ");
         System.exit(0);
      }
    } catch (Exception e){
      optMessaggio.showMessageDialog(null, "Connessione NON riuscita " + e);
      System.exit(0);
    }
}
:
:
```

Input e Output

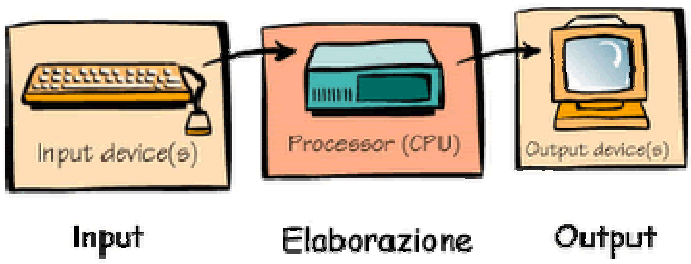
La **classe System**, presente nel package **java.lang** (*vedi input, vedi oggetti*), è il punto di accesso a diverse funzioni strettamente legate alle funzionalità di base di Java, come la JVM, i meccanismi di sicurezza e le proprietà del sistema. Oltre a questo, la classe System dispone di tre campi che forniscono l'accesso all'I/O di sistema:

- **In**: che consente di accedere all'input,
- **Out**: che permette l'accesso all'output,
- **Err**: che rappresenta un canale dedicato agli errori.

Il **primo** di questi campi è di tipo **InputStream** mentre gli **altri** sono di tipo **PrintWriter**; queste classi appartengono al package **java.io** e dispongono di funzionalità di lettura e scrittura, **in** e **out** consentono di accedere all'**input/output** di sistema, mentre **err** viene utilizzato per la **gestione degli errori**.

- Quando un programma ha la necessità di segnalare un evento critico, lo si può inviare ad err; quando la comunicazione è un normale output, viene inviata ad out.

Tutti i programmi si basano sul seguente schema:



Output

La presentazione dei **risultati a video** viene definita **standard output**, a cui si accede tramite l'oggetto **System.out**. Vediamo un semplice esempio:

```
class CiaoMondo  {
    public static void main(String[] args)
    {
      String nome = “Grazia”;
      String cognome = “Cesarini”;
      System.out.print ("Ciao mondo, sono il primo programma in Java ");
      System.out.println ("di " + nome + " " + cognome);
    }
}
```

Per visualizzare una qualsiasi frase dovremmo procedere con la seguente riga di codice:

```
System.out.print("Qua puoi scrivere ciò che vuoi,");
System.out.print("ricorda che verrà visualizzato");
```

le due frasi verranno visualizzate **una di seguito all'altra**. Se volessimo **andare a capo**, al posto di print, dovremmo utlizzare **println**, quindi verrà fuori una cosa del genere:

```
System.out.println("Qua puoi scrivere ciò che vuoi,");
System.out.print("ricorda che verrà visualizzato");
```

In questo caso, il programma scriverà la frase "Qua puoi scrivere ciò che vuoi," , andrà a capo, poi scriverà la frase "ricorda che verrà visualizzato".

Input

InputStreamReader

Un InputStreamReader è un ponte fra flussi di byte a flussi di caratteri:

> legge i byte e li trasforma in caratteri utilizzando un set di caratteri specificato. Il set di caratteri che usa può essere specificata in base al nome o può essere dato esplicitamente, o può essere accettato charset di default della piattaforma.

Ogni chiamata di uno dei **metodi read()** di un **InputStreamReader** potrebbe causare la lettura di **uno o più byte** dal sottostante flusso di byte di ingresso. Per attivare una conversione efficiente di byte in caratteri, **più byte possono essere letti in anticipo** dal flusso sottostante in modo da soddisfare l'operazione in corso di lettura.

BufferedReader

Per la **massimizzare l'efficienza** si può ad esempio impostare la seguente struttura:

```
BufferedReader in = New BufferedReader (new InputStreamReader (System.in));
```

La classe **BufferedReader** legge il testo da un **flusso di caratteri in input**, trasferisce in un "buffer"[11] i caratteri in modo da rendere efficiente la lettura di caratteri, matrici, e linee.
La dimensione del buffer può essere specificato o può essere utilizzato la dimensione predefinita. Il valore di default è abbastanza grande per la maggior parte degli scopi.
In generale, ogni richiesta di lettura fatta da un Reader provoca una corrispondente richiesta di lettura del carattere sottostante o del flusso di byte.

E 'quindi consigliabile che un **BufferedReader** abbia come argomento un qualsiasi **Reader** la cui operazione di **read()** può essere onerosa, come ad esempio **FileReaders** e **InputStreamReaders**. Per esempio,

```
BufferedReader in  = new BufferedReader (new FileReader("pippo.in"));
```

consente l'input dal file specificato.

I programmi che utilizzano DataInputStreams per l'input testuale possono sostituire ogni DataInputStream con un BufferedReader appropriato.

Tips and tricks – Senza Buffer

Senza buffer, ogni invocazione di read() o readline() potrebbe causare la lettura di byte direttamente dal file, la conversione in caratteri, ed una successiva ripetizione dell'operazione. Questo modo di procedere risulterà **molto inefficiente**.

Files di caratteri

Java utilizza il concetto astratto di stream per implementare le funzioni relative alla gestione dei file.

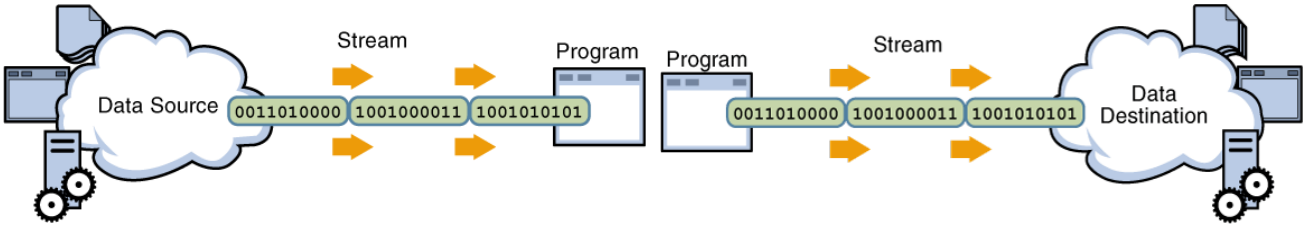
Lo stream può essere immaginato come un canale tra la sorgente di una certa informazione e la sua destinazione. Gli stream vengono classificati in:

- **stream di input**, se il punto di vista è quello di chi riceve l'informazione (produttore) attraverso il flusso,
- **stream di output**, se il punto di vista è quello di chi invia l'informazione (consumatore) nel flusso.

La potenza dell'astrazione del concetto di stream consiste in questo: non è necessaruio conoscere la sorgente delle informazioni quando si legge un flusso di input, così come non è necessario conoscere la destinazione quando si scrive un flusso di output.

Uno stream può rappresentare i **più disparati tipi di sorgenti e destinazioni**, tra cui **files**, **dispositivi periferici** come il monitor o la tastiera, o **altri programmi**.

Gli streams supportano **vari tipi di dati** che vanno **dal semplice byte ad oggetti complessi**. Indipendentemente dal tipo di sorgente, un programma usa un **input stream** per leggere dati, uno alla volta, da una sorgente e viceversa un programma usa un **output stream** per scrivere dati, uno alla volta, su una destinazione.



Esempio: Leggere informazioni byte per byte in un programma.

Esempio: Scrivere informazioni byte per byte da un programma.

[11] *area di transito della RAM*

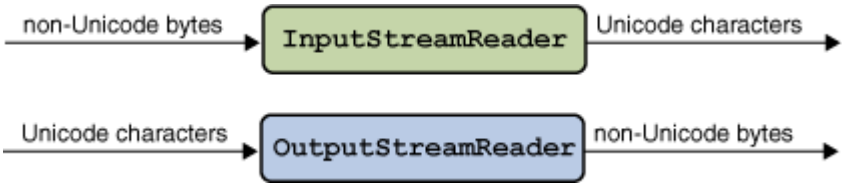
Stream e file

Come detto, la sorgente e la destinazione dei dati di uno stream possono essere qualsiasi, in questo caso considereremo uno **stream** in cui **sorgente** o **destinazione** siano **files**. Nel caso siano coinvolti files potremmo parlare di:

- **flusso di caratteri**, per file di **testo**;
- **flusso di byte**, per file **binari**;
- flusso di tipi primitivi, per file di tipi primitivi;
- flusso di oggetti, per file di oggetti.

Il tipo di **accesso** a questi file è unicamente di **tipo sequenziale**: per accedere a uno specifico oggetto memorizzato in un file, è necessario accedere prima a tutti gli oggetti che lo precedono in quel file.

- **Stream di caratteri**
  Nel formato **testo**, i dati sono rappresentati in forma **leggibile dall'uomo**. Ciascun carattere (la cui codifica richiede 4 byte) è rappresentato tramite il suo valore UNICODE. Ad esempio, il numero 234 è rappresentato da una sequenza di tre caratteri: "2", "3", "4"; aprendo il file con un nor male word processor, si vede il numero 234, poichè il word processor traduce i valori UNICODE nei simboli corrispondenti: il numero risulta, cosi, immediatamente leggibile.
  I valori **numerici non** sono rappresentati in **UNICODE**: sono invece rappresentati con insiemi di byte di lunghezza che varia col tipo di dato; ricordiamo infatti che la codifica di un numero è diversa a seconda che si tratti di un numero naturale di un numero reale (con la virgola), che abbia un segno o sia senza segno ecc.
  Gli **stream di caratteri** vengono gestiti da una gerarchia di classi che si basa sulla codifica standard UNICODE. In cima alla gerarchia ci sono le classi astratte Reader e Writer, da cui si derivano sottoclassi concrete per l'IO dei caratteri.
  E' inoltre possibile **convertire stream di byte codificanti come caratteri non UNICODE in stream di caratteri UNICODE** mediante le classi **InputStreamReader** e **OutputStreamWriter** che operano come in ***figura***.



- **Byte Stream**
  Diversamente da quanto avviene nel formato testo, nel **formato binario** i dati **non** sono **leggibili** per **l'uomo con un generico word processor**. Un editor di testo UNICODE, infatti, cerca di associare i byte a quattro a quattro, visualizzando per ogni quartetto il corrispondente carattere UNICODE. Ma i dati contenuti in un file binario non devono essere associati 4 byte alla volta. Pertanto, **aprendo un file binario con un word processor, si vedranno simboli privi di senso**.
  I programmi usano byte stream per eseguire I/O di bytes da 8-bit. Tutti i byte stream discendono da due classi astratte: InputStream ed OutputStream. Un IO sui files può essere realizzato tramite le sottoclassi concrete **FileInputStream** e **FileOutputStream** che consentono di **copiare byte per byte** il contenuto di un file di input in un file di output.

Per gestire questi differenti tipi di file in Java utilizziamo differenti classi, che permettono di controllare le operazioni di lettura e scrittura su file tramite stream.

Le classi per gestire i file

Alla base della gestione degli stream si trovano due **classi astratte** (*vedi classi astratte*): **InputStream** e **OutputStream**, dedicate rispettivamente agli **stream di input** e agli **stream di output**.

Essendo classi astratte, **non e possibile creare direttamente oggetti di queste classi**. Per creare oggetti stream aventi file come sorgente o come destinazione, si utilizzano le sottoclassi:

1. FileInputStream e FileOutputStream: per leggere da un file di byte attraverso uno stream di input e per

scrivere su un file di byte attraverso uno stream di output;

2. **FileReader** e **File Writer**: per **leggere da un file di caratteri** attraverso uno stream di input e per **scrivere su un file di caratteri** attraverso uno stream di output;
3. DataInputStream e DataOutputStream: per leggere da un file di tipi primitivi attraverso uno stream di input e per scrivere su un file di tipi primitivi attraverso uno stream di output;
4. ObjectInputStream e ObjectOutputStream: per leggere da un file di oggetti attraverso uno stream di input e scrivere su un file di oggetti attraverso uno stream di output.

Per utilizzare tutte queste classi occorre importare il package **java.io.**
Per utilizzare i file con i flussi che ci interessano (di byte, caratteri, tipi primitivi, oggetti), il procedimento è comune:

- si considerano le **classi InputStream** e **OutputStream,**
- si ridefiniscono i loro **metodi principali**: **read**(), **write**() e **close**().

D'ora in poi ci occuperemo in particolare di **stream di caratteri** per file di **testo** prendendo quindi in considerazione solo il **caso n.2.** Come detto, per la **lettura** e la **scrittura di files di caratteri** si usano le sottoclassi: **FileReader** e **File Writer** delle classi astratte **InputStream** e **OutputStream.**

Leggere e scrivere files di caratteri

I flussi di caratteri vengono utilizzati per elaborare qualunque testo composto di caratteri appartenenti al set **UNICODE** (che comprende il set **ASCII**)[12].

Per leggere e scrivere da un flusso di caratteri occorre utilizzare **le classi**, rispettivamente **Reader** e **Writer**. Per leggere e scrivere su flussi che coinvolgono file su disco si utilizzano **FileReader e FileWriter** e le loro **sottoclassi**.

- **FileWriter**(String fileName)
  Dato il nome di un file, costruisce un oggetto FileWriter.
- **FileWriter**(String fileName, boolean append).
  Dato il nome di un file, costruisce un oggetto FileWriter, con una variabile booleana indica il file può essere scritto o no in **modalità append** (true).

Una volta creati due oggetti, uno per l'input ed uno per l'output, delle **classi FileReader** e **FileWriter**, si può utilizzare il **metodo read()** per leggere da un file ed il **metodo write()** per scrivere su un file. Quando la lettura o la scrittura sono terminate si chiama il **metodo close()** per chiudere il flusso.

Per assicurarsi che il buffer sia stato svuotato e, nel caso di **BufferWriter**, che la scrittura su file sia stata effettiavmente eseguita, è necessario usare il **metodo flush()**.

Esempio: FileWriter e BufferedWriter

Codice JAVA	Descrizione
`private void btnAggiungiActionPerformed(java.awt.event.ActionEvent evt) {`	
`  try  {`	
`    FileWriter file = new FileWriter("pullman.txt", true);`	Aprire un file per la scrittura
`    BufferedWriter output = new BufferedWriter(file);`	Aprire un buffer per trasferire I record in output
`    riga = txtPartenza.getText() + ";" + txtDestinazione.getText() + ";" + txtArrivo.getText() + "\r\n";`	Il record è formato dai: campi immessi da tastiera separati da ";". La riga si conclude con \r\n (Formato Windows)

[12] *Esempi di file di caratteri sono i normali file generati con word processor e salvati in formato testo o file HTML.*

```
    output.write(riga);                          Scrivere un record
    cancella();
    output.flush();                              Salvare su disco il file
    output.close();                              Chiudere il file
  }
  catch(IOException e)                           Controllo dell'eccezione
  {                                              IOException
    lblER1.setText("Errore:"+ e.toString());
  }                                              fine "btnAggiungi…"
}
```

Java riconosce la piattaforma in cui è attualmente in esecuzione, in modo da fornire output dipendi dalla piattaforma (Linux, Windows, MacOS). Quindi, se è noto che il file viene sempre aperto su una particolare macchina, si dovrà usare come:

- **Windows: carriage return + line feed \r\n**
- Linux, Android: line feed \n
- MacOS: carriage return \r

Esercizi svolti

Visualizzare un elenco, cognome e nome, contenuto in un file.

```
package my_leggi;
import java.nio.charset.Charset;
import java.io.*;                              import java.io.BufferedReader;
                                               import java.io.EOFException;
                                               import java.io.File;
                                               import java.io.IOException;
import java.nio.file.*;                        import java.nio.file.Files;
                                               import java.nio.file.Path;
                                               import java.nio.file.Paths;
public class leggi
{
 public static void main(String[] args)
 {
   String riga;
   File temp = new File("studenti.csv");
   Path archivio =                             utilizzare l'istanza Path come un'ancora
Paths.get(temp.getAbsolutePath());
   Charset charset = Charset.forName("US-      Il file è codificato in "US-ASCII"
ASCII");
   try
   {
     BufferedReader buffer =
Files.newBufferedReader(archivio,charset);
     while ((riga = buffer.readLine()) !=      Il record è formato dai campi immessi da
null)                                          tastiera separati da ";".
     {                                         La riga si conclude con \r\n
                                               readline() legge l'intera riga senza il
                                               carattere "a capo" che nel nostro caso è
                                               \r\n
        String[] campi = riga.split(";");      la riga viene suddivisa ("split") nei campi
                                               che la compongono
        System.out.println(campi[0] + ' '
+ campi[1]);
     }
     buffer.close();
   }
   catch(EOFException e1)                      Visualizzazione messaggio di EOF nella
   {                                           label lblER2
     System.err.println("Lettura file
completata");
   }
   catch(IOException e)                        Visualizzazione messaggio di errore di IO
   {                                           nella label lblER2
```

```
     System.err.println("Error: " +
e.toString());
   }
 }
}
```

Output

```
run:
Rossi Mario
Bianchi Andrea
Verdi Pino
Gialli Ugo
Rossi Ada
Bianchi Tino
Verdi Lino
Gialli Gino
BUILD SUCCESSFUL (total time: 4 seconds)
```

Usare l'interfaccia GUI Builder dell'IDE NetBeans

Vediamo come è possibile creare l'interfaccia grafica utente (**GUI**) per alcune semplici applicazioni utilizzando il **GUI Builder** dell'**IDE NetBeans**.

In particolare vediamo come utilizzare l'interfaccia GUI Builder per creare un contenitore GUI, aggiungere, ridimensionare, allineare i componenti, modificare le proprietà dei componenti ed infine come rispondere a un evento semplice associato ad un pulsante.

Programmazione ad eventi con il package grafico Swing

Prima di cominciare è il caso di spiegare brevemente il funzionamento della **programmazione ad eventi**

Nella **programmazione ad eventi** è necessario specificare gli oggetti (**sorgenti**) con cui si può avere un'interazione (**evento**), e, cioè un insieme di classi che, in modo concorrente, controllano se le sorgenti hanno riscontrato l'evento (**ascoltatori**).

Ogni componente **grafico** è una potenziale **sorgente**, mentre gli **ascoltatori** sono oggetti che implementano specifiche interfacce, dette appunto **listener**.

L'architettura Java è divenuta **graphics-ready** grazie alla presenza del **package AWT**[13] (java.awt), il primo package grafico (Java 1.0) indipendente dalla piattaforma... o quasi!
A partire da Java 2; versione preliminare da Java 1.1.6) il package è stato sotituito dal nuovo **package** grafico **Swing** (javax.swing) scritto in Java e realmente indipendente dalla piattaforma.
Il **package Swing** definisce una gerarchia di classi che forniscono ogni tipo di componente grafico (finestre, pannelli, frame, bottoni, aree di testo, checkbox, liste a discesa, etc). Grazie alla presenza di **Swing** è possibile operare con la programmazione **event-driven**. Grazie a questo tipo di programmazione non è necessario utilizzare algoritmi stile input/elaborazione/output… ma è sufficiente operare sulle reazione agli eventi che l'utente, in modo interattivo, genera sui componenti grafici. Questa programmazione è basata sui concetti di **evento** e di **ascoltatore degli eventi**

Gli eventi: sorgenti ed ascoltatori

Quando si crea l'interfaccia di un software si definisce un **insieme di oggetti** con i quali l'**utente** potra **interagire**.
Per sviluppare un software dotato di un'interfaccia GUI innanzitutto sarà opportuno:

- predisporre la rappresentazione grafica inserendo uno o più componenti grafiche (**sorgenti**) nell'interfaccia;
- decidere quali **eventi** (azioni compiute dall'utente) gestire e predisporre le conseguenti reazioni del sistema;
- definire un **ascoltatore** di uno o piu eventi fra quelli individuati;
- associare l'ascoltatore alla sorgente.

[13] AWT: Abstract Window Toolkit

E' importante notare che **ogni componente può essere sorgente di diversi eventi**: per esempio, un pulsante prevede non solo la pressione, ma anche altre azioni legate al mouse (singolo o doppio clic, passaggio del cursore, pulsante destro ecc.), alla modifica di stato delle sue proprietà ecc.

In qualunque momento, l'utente può cliccare su una componente o premere un tasto della tastiera, e la GUI delle applicazioni deve intercettare tali eventi e gestirli quindi deve essere 'sempre all'erta'. Oltre agli eventi generati dall'utente esistono anche **eventi time-driven**, generati da thread eseguiti in parallelo (o... pseudo-parallelo), che non dipendono dall'utente ma che implicano un aggiornamento della GUI.

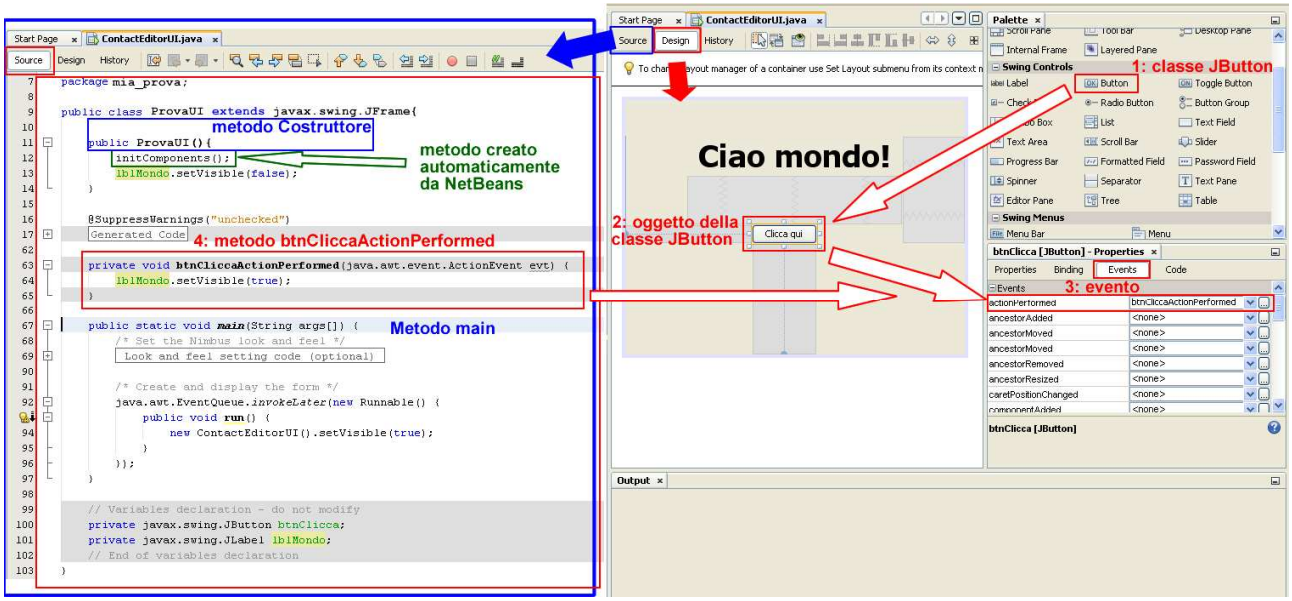
Tutti queste considerazioni stanno alla base della **programmazione orientata agli eventi**.

Java prevede particolari oggetti, i **listener** (**ascoltatori**); questi vengono 'avvisati' dal sistema quando si verifica un evento che li riguarda e che hanno il compito di gestire tali eventi: l'implementazione del metodo o dei metodi di gestione spetta al programmatore. Ad esempio: il listener degli eventi da tastiera verrà invocato quando l'utente premerà, appunto, un tasto della tastiera.

La relazione componente-listener è di tipo molti-a-molti: **un componente può avere più listener** che, a loro volta, possono essere **'registrati' presso più componenti** (è possibile, ad esempio, utilizzare un oggetto contenitore come listener e registrarvi tutti gli oggetti GUI in esso contenuti, per gestirne gli eventi).

I **listener sono interfacce da implementare**, il che significa che spesso siamo 'obbligati' a riscriverne alcuni metodi.

Fortunatamente per noi, una volta selezionato l'evento da associare ad un oggetto, **NetBeans** inserirà **automaticamente un ascoltatore di eventi**.



Nelle ***figure*** vediamo il costruttore che richiama il **metodo initComponents()**. Questo metodo viene generato automaticamente da **NetBeans** e contiene tutto il codice dell'interfaccia grafica, ascoltatori compresi, costruita nell'area disegno. Il **codice non può essere modificato** (*vedi Tips and tricks - Guarded Blocks*) direttamente dal programmatore ma solo attraverso l'area delle proprietà dei singoli oggetti.

Tips and tricks – Guarded Blocks

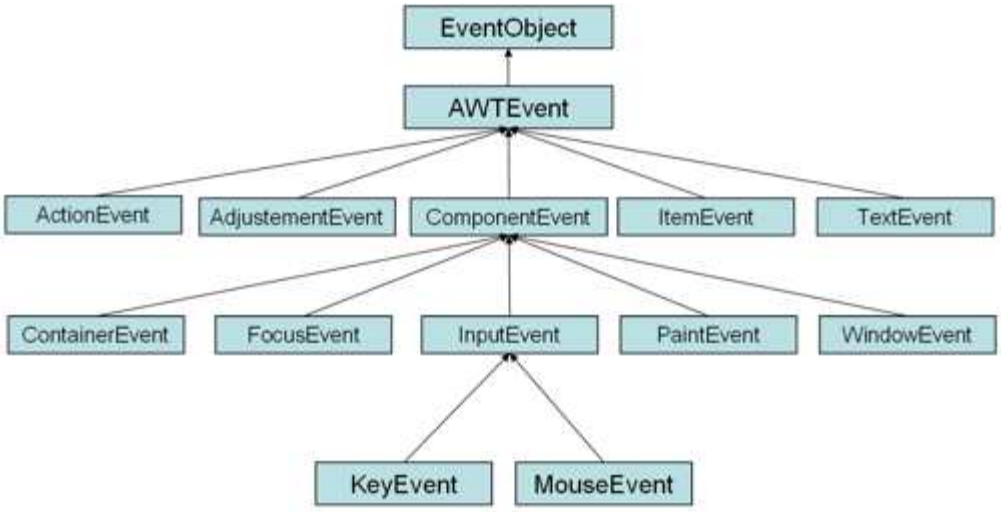
Facendo clic sul **pulsante Source**, l'IDE visualizza l'applicazione del codice sorgente Java nell'editor con tutte le **sezioni di codice** che vengono **generati automaticamente dal GUI Builder**, non solo il metodo **initComponents().** Tutte queste sezioni vengono indicate con **zone grigie** (diventano blu quando selezionate), chiamate **Guarded Blocks**[14]. I "Guarded Blocks" sono aree protette non modificabili dal programmatore in visualizzazione Source. In visualizzazione Source è possibile modificare solo il codice che appare nelle aree **bianche** del Editor cioè quello scritto dal programmatore stesso.

Analisi del package Swing

Come detto l'architettura Java è divenuta **graphics-ready** grazie alla presenza del package AWT. Il **package AWT** infatti contiene le classi e le interfacce fondamentali per il rendering grafico. Queste classi consentono di realizzare **interfacce utente** (**GUI**) complesse e di definire l'**interazione** attraverso la specifica di elementi e di **gestori** degli stessi consentendo quindi di operare con la programmazione **event-driven**.

Riassumiamo qui di seguito la **gerarchia di eventi di AWT**:

[14] *Guarded Blocks: blocchi custoditi*

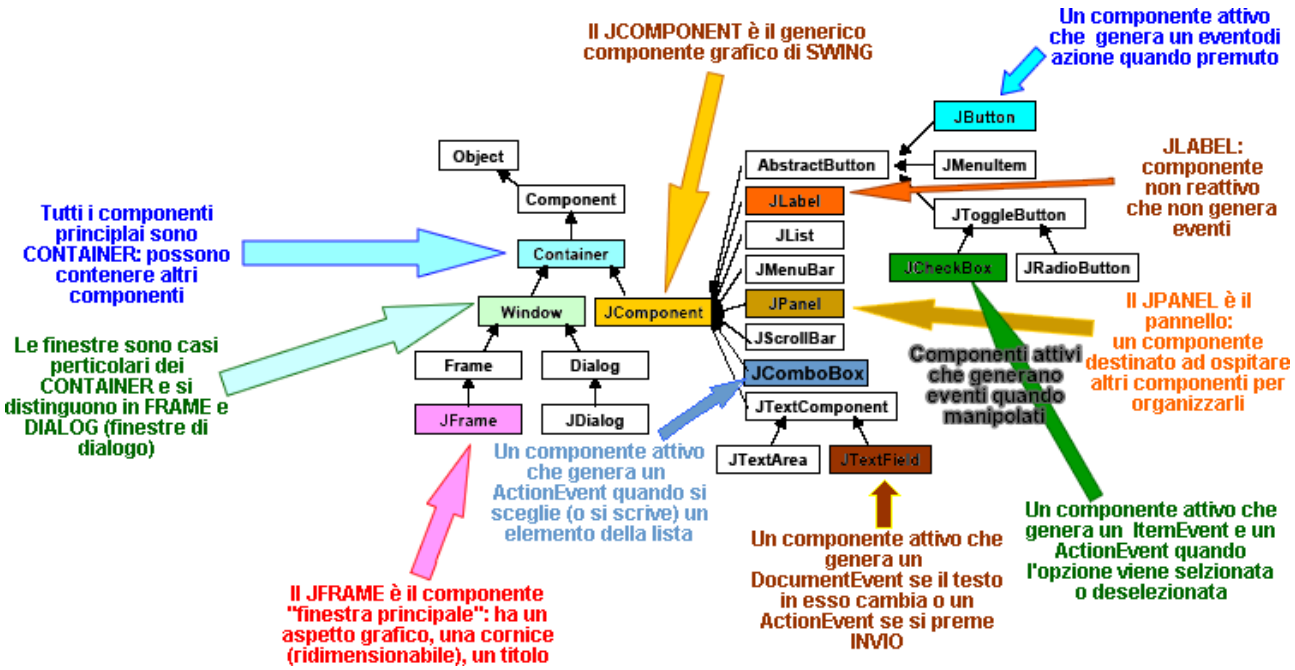


La tabella seguente, invece, riassume le associazioni **event-listener** presenti nel **package AWT**:

Event	Listener
ActionEvent	ActionListener
AdjustementEvent	AdjustementListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener
KeyEvent	KeyListener
MouseEvent	MouseListener, MouseMotionListener
TextEvent	TextListener
WindowEvent	WindowListener

A partire da Java 2 e gli elementi AWT furono in gran parte superati da quelli dal nuovo **package** grafico **Swing**. Oltre a fornire un set più ricco di elementi, Swing crea i suoi elementi personali. Swing offre la possibilità di usare sia il **sistema "Look and feel"** che usa gli elementi del sistema operativo, sia il **"Java Look and Feel"**, che fornisce agli elementi lo stesso aspetto su ogni piattaforma. Anche grazie alla presenza di **Swing** è possibile operare con la programmazione **event-driven**.

Riassumiamo la gerarchia di classi del **package SWING**:



Catturare gli eventi da mouse e tastiera

L'IDE può aiutarci a trovare l'elenco degli eventi disponibili per i componenti GUI da gestire (*vedi Tips and tricks - Trovare metodi, proprietà ed eventi*)

Per gestire gli **eventi da tastiera** abbiamo a disposizione i seguenti strumenti:

- **KeyEvent**, classe utilizzata per definire gli eventi.
  Nel set di costanti[15] di KeyEvent tali costanti sono:
  - VK_A, ..., VK_Z[16] - per le lettere;
  - VK_0, ..., VK_9 - per i tasti numerici;
  - VK_PAGE_UP, VK_PAGE_DOWN, ... - per gli altri tasti hanno nomi specifici;
  - più altri valori che consentono di verificare se sono stati attivati dei modificatori (shift, ctrl, alt, ...)[17].

  KeyEvent ci permette di individuare **quale tasto** è stato premuto mediante i **metodi**:
- **getKeyCode**() : int restituisce il valore (vd. costanti predefinite sopra) associato al tasto premuto.
- **getKeyChar**() : char restituisce il valore letterale (char) del tasto premuto.
- **getKeyText**(int code) : String (metodo statico) restituisce il valore sotto forma di String del codice di tasto passato come parametro.
- **KeyListener**, listener (interfaccia) degli eventi, che ci 'obbliga' a riscrivere tre metodi;
  I metodi di KeyListener da implementare sono i seguenti:
- **keyPressed**(keyEvent e) : void Pressione di un tasto
- **keyReleased**(keyEvent e) : void Tasto rilasciato
- **keyTyped**(keyEvent e) : void Tasto 'digitato'
- **KeyAdapter**, classe Adapter che ci consente di riscrivere solo i metodi che ci servono dei tre 'obbligatori' di Listener.

In maniera del tutto analoga a quanto visto per la tastiera, è possibile gestire gli **eventi** generati mediante **mouse** con i seguenti strumenti:

- **MouseEvent**; classe utilizzata per definire gli eventi;

La classe MouseEvent fornisce i seguenti metodi per recuperare alcune informazioni utili:

- getButton() : int restituisce il codice del bottone premuto
- getClickCount() : int restituisce il numero di click 'consecutivi' effettuati
- getPoint() : Point restituisce le coordinate del pixel cliccato su schermo organizzate sotto forma di oggetto Point
- getX() : int restituisce la coordinata X del pixel cliccato su schermo
- getY() : int restituisce la coordinata Y del pixel cliccato su schermo
- **MouseListener**, MouseMotionListener, MouseInputListener; **vari listener** (interfacce) delle azioni effettuate dal mouse (il più utilizzato, comunque, è il primo: MouseListener);
- MouseAdapter, MouseMotionAdapter, MouseInputAdapter; **classi adapter** che implementano i listener appena elencati.
  Implementando MouseListener, occorrerà riscrivere i seguenti metodi:
  - mouseClicked(MouseEvent e) : void
  - mouseEntered(MouseEvent e) : void
  - mouseExited(MouseEvent e) : void
  - mousePressed(MouseEvent e) : void
  - mouseReleased(MouseEvent e): void

Tips and tricks - Impostare l'aspetto del cursore del mouse

In Java è possibile impostare l'aspetto che dovrà assumere il cursore del mouse al passaggio sulle varie

[15] *Java 'mappa' i tasti della tastiera su di un insieme di costanti predefinite*
[16] *VK sta per 'virtual key', 'tasto virtuale', ad indicare che si tratta della rappresentazione interna di Java*
[17] *Per un elenco completo, si rimanda alla documentazione ufficiale*

componenti.

La classe che gestisce l'aspetto del cursore è Cursor, con costruttore principale:

- **Cursor**(int type):void

mentre, per impostare il cursore di un componente, bisogna utilizzare il metodo di Component:

- **setCursor**(Cursor c) : void

Java mette a disposizione diversi cursori predefiniti, tutti identificati mediante costanti intere definite nella classe Cursor (è a tali costanti che fa riferimento il **parametro intero type** presente **nel costruttore di Cursor**); è comunque possibile creare aspetti personalizzati per il cursore del mouse mediante il metodo di **Toolkit**:

- **createCustomCursor**(Image cursor, Point hotSpot, String name) : Cursor

che permette di impostare l'**Image cursor** passata come parametro come **aspetto visivo** del cursore, **hotSpot** come **punto sensibile** e name come nome.

Concetti chiave

L'**IDE di GUI Builder** risolve il problema centrale della creazione di GUI Java semplificando il flusso di lavoro di creare interfacce grafiche, **liberando gli sviluppatori dalle complessità dei gestori di layout Swing**. Lo fa **GUI Builder**, estendendo l'IDE NetBeans, per consentire un semplice "Disegno libero" con regole di layout semplici facili da capire e da usare.

Quando si predispone il proprio modulo, il GUI Builder fornisce le linee guida visive che suggeriscono le spaziatura ottimali e l'allineamento dei componenti e traduce le decisioni di progettazione in un'interfaccia utente funzionale che viene realizzato con il nuovo gestore di layout GroupLayout e altri costrutti Swing.

Perché utilizza un modello dinamico di layout grafico, GUI Builder si comporta come ci si aspetterebbe in fase di esecuzione, adeguandosi per accogliere le modifiche apportate senza alterare le relazioni definite tra i componenti. Ogni volta che si ridimensiona un form, degli switch locali oppure si specifica un aspetto diverso, l'interfaccia grafica si adatta automaticamente rispettando l'aspetto di destinazione.

Disegno libero

Nella IDE di GUI Builder, è possibile creare i moduli semplicemente mettendo componenti dove si vuole, come se si stesse utilizzando il posizionamento assoluto, GUI Builder capisce quali sono gli attributi richiesti dal layout disegnato e quindi genera il codice automaticamente non c'è quindi bisogno di preoccuparsi di inserti, ancore, riempimenti e così via.

Posizionamento automatico delle componenti (snapping immediato)

Quando si aggiungono i componenti di un form, GUI Builder fornisce un feedback visivo che consente ai componenti il posizionamento sulla base dell'aspetto grafico ed utili suggerimenti in linea ed altri feedback visivi che indicano come i componenti devono essere immessi sul modulo e posizionati automaticamente sulla base dell'orientamenti. Questi suggerimenti vengono forniti sulla base delle posizioni dei componenti che sono già stati messi sul form, pur consentendo agli spazi vuoti di rimanere flessibili in fase di runtime.

Feedback visivo

GUI Builder fornisce anche un feedback visivo per quanto riguarda l'ancoraggio ed i concatenamento; questi indicatori consentono di identificare rapidamente le diverse relazioni di posizionamento ed i comportamenti dei componente "pinning" che influenzano il modo in cui i GUI appaiono e si comportano in fase di esecuzione. Questo accelera il processo di progettazione grafica, che consente di creare rapidamente interfacce visuali funzionanti e dall'aspetto professionale.

Best practice

Anche se l'IDE GUI Builder semplifica il processo di creazione di interfaccia GUI, è spesso utile **delinearlo** con un semplice disegno in modo da vedere **l'interfaccia prima di iniziare a realizzarlo**, Molti progettisti di interfacce considerano questa una tecnica una "best practice".

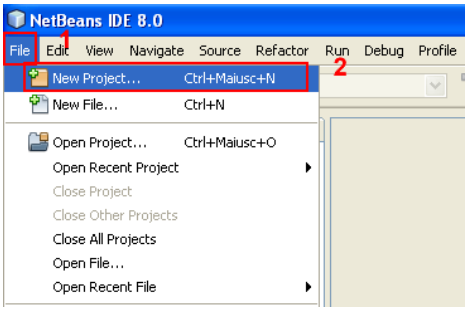
Tuttavia, **ai fini delle esercitazioni qui proposte**, si può semplicemente realizzare l'interfaccia e quindi dare un'occhiata a come appare il nostro modulo compilato passando alla sezione anteprima (*vedi Anteprima del GUI*).

Creazione del 1° GUI con Swing

Perché tutto lo sviluppo Java nell'**IDE** si svolge nell'ambito di **progetti**, abbiamo prima bisogno di creare un **New Project**[18] in cui **memorizzare le fonti e file di progetto**.

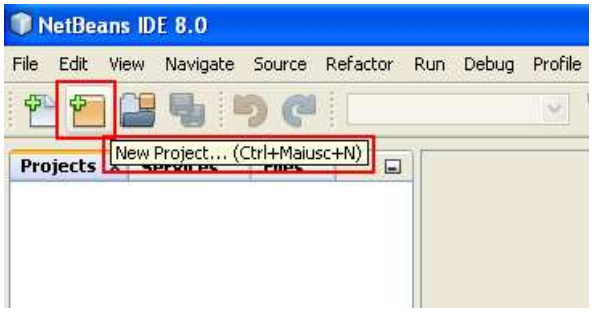
In questo esempio costruiremo una semplice applicazione memorizzata interamente in un unico progetto.

Per creare l'applicazione **ContactEditor** con **New Project**:



Scegliete **File** > **New Project**.

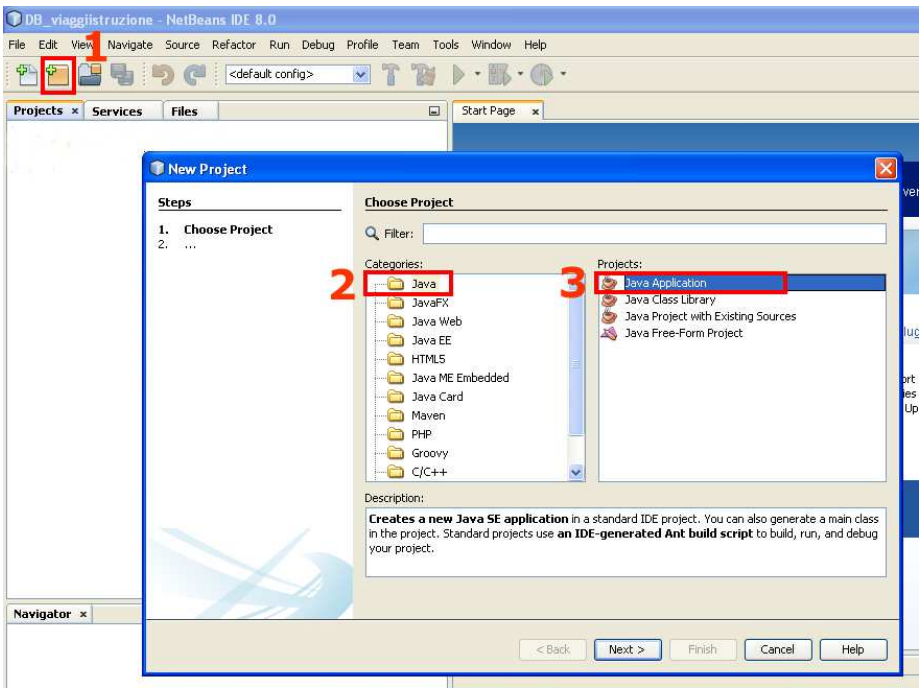
← ovvero →



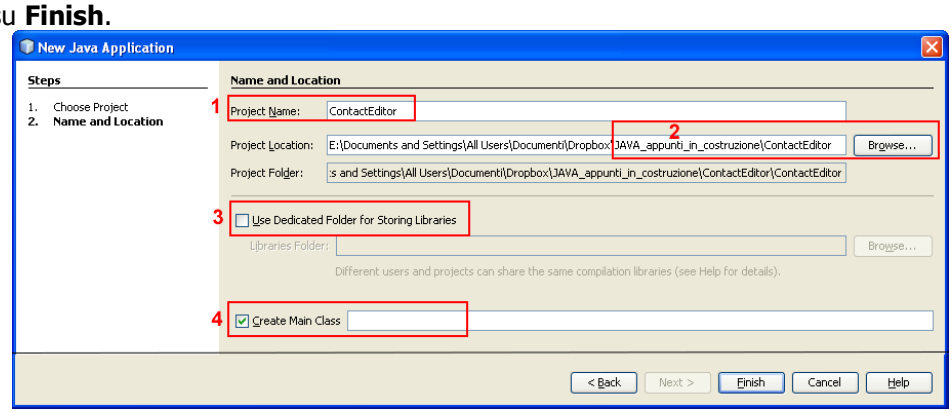
Fare clic sull'icona **New Project** nella **barra degli strumenti IDE**.

Nel riquadro **Categories**, selezionare **Java** nel pannello **Choose Projects**, scegliere **Java Application**. Fare clic su **Next**.

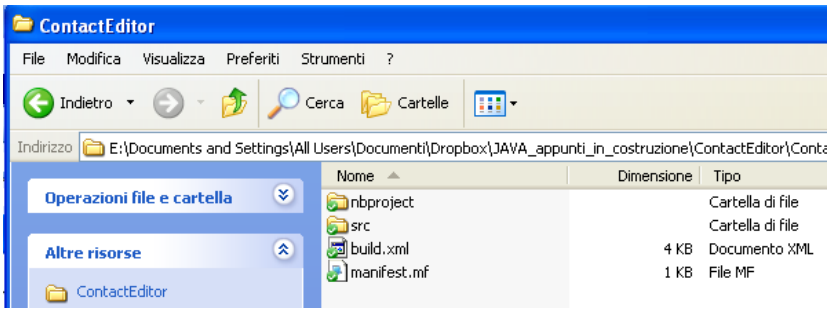
[18] Un progetto IDE è **un gruppo di file sorgenti** Java, più i suoi **metadati** associati, tra cui progetti specifici di file di proprietà, un script di build Ant per controllare la generazione ed eseguire le impostazioni, e un file che mappa obiettiviproject.xml Ant ai comandi IDE.



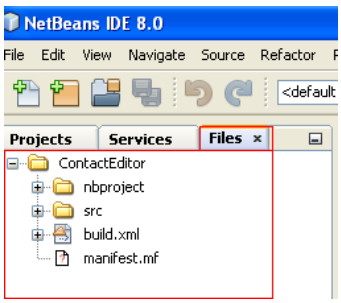
- Inserisci **ContactEditor** nel campo **Project Name** e specificare il **percorso** del progetto.
- Lasciare deselezionata la casella della cartella per l'uso dell'archiviazione di librerie.
- Assicurarsi che la casella di controllo MAIN project sia selezionata e cancellare il campo Create Main Class.
- Fare clic su **Finish**.



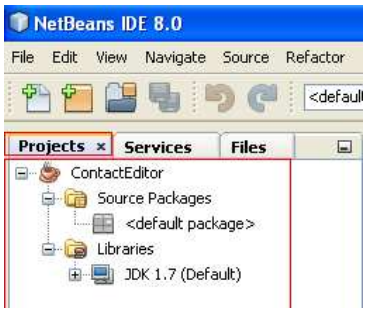
L'IDE crea la **cartella ContactEditor** nella posizione indicata. Questa cartella contiene tutti i file associati al progetto:



Per visualizzare la struttura del progetto, utilizzare la finestra dei file dell'IDE:



1. Creare un Container JFrame

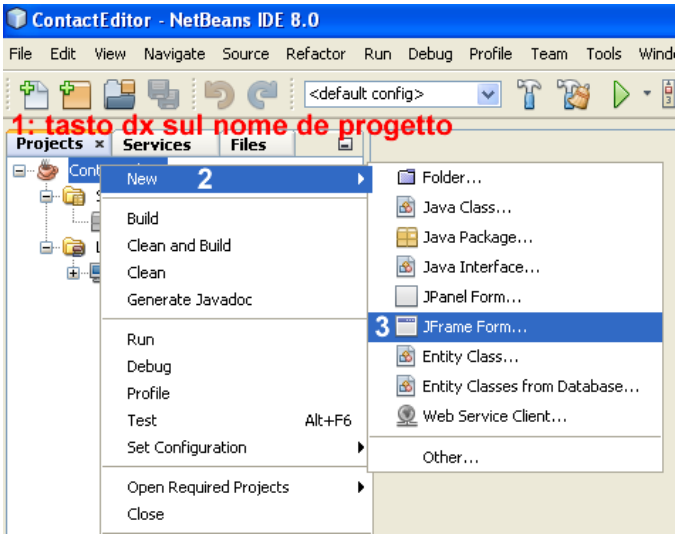


Dopo aver creato la nuova applicazione, la cartella di origine dei pacchetti nella **Finestra Project** conterrà un nodo vuoto **<default package>**.

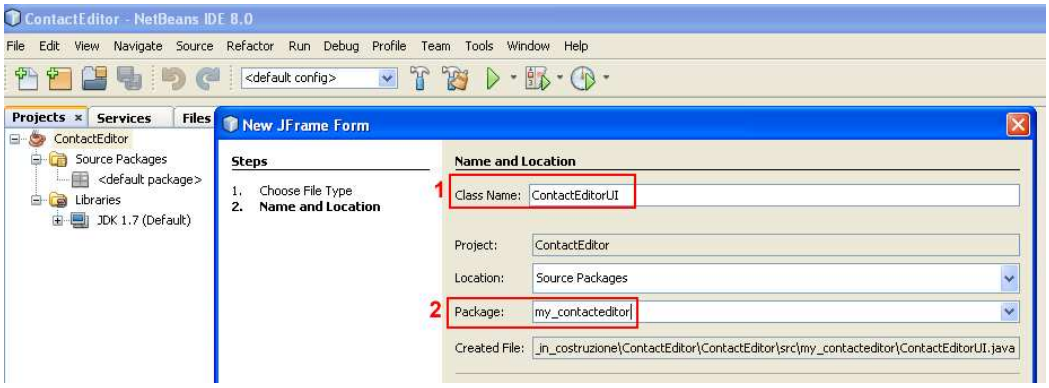
Per procedere con la costruzione dell'interfaccia, è necessario creare un **contenitore di Java all'interno del quale metteremo gli altri componenti GUI richiesti**.

Per creare il contenitore si utilizza il componente **JFrame** e lo si posiziona in un nuovo package.

- Per aggiungere un contenitore JFrame:
- nella Finestra Projects, fare clic su ContactEditor e scegliere il modulo New> JFrame Form.



- In alternativa, è possibile trovare un modulo JFrame scegliendo New > Other > Swing GUI Forms > JFrame Form.
- Inserisci **ContactEditorUI** come nome della classe.
- Inserisci **my_contacteditor** come nome del pacchetto.
- Fare clic su **Finish**.

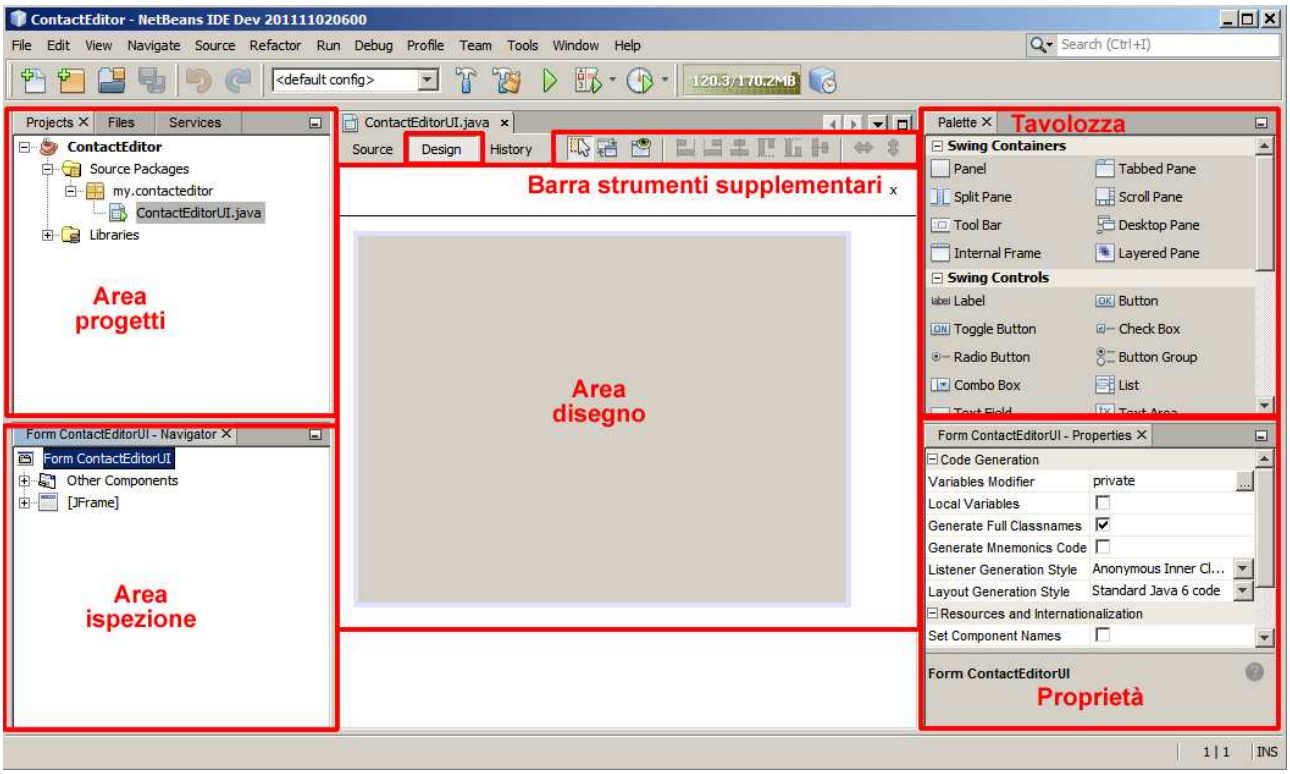


L'**IDE** crea il **form** e la **classe ContactEditorUI** all'interno dell'**applicazione ContactEditorUI.java** e apre il **modulo ContactEditorUI nella GUI Builder**. Si noti che il pacchetto **my_contacteditor sostituisce il pacchetto di default**.

Ora abbiamo creato un **New Project** per la nostra applicazione[19].

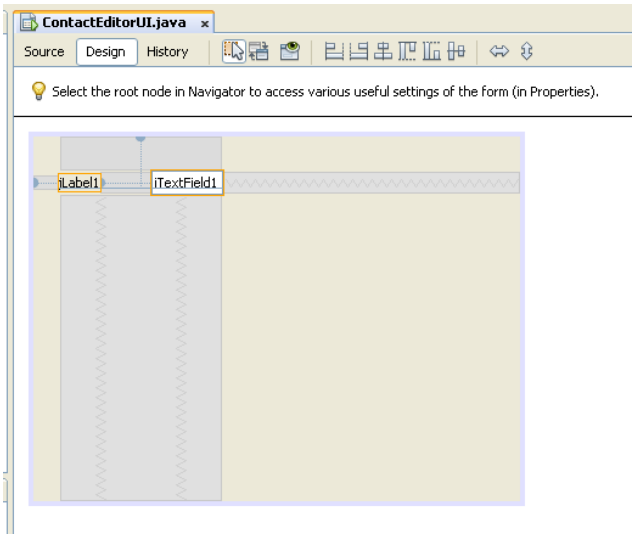
2. Le finestre del GUI Builder

Quando abbiamo aggiunto il **contenitore JFrame**, l'IDE ha aperto il modulo appena creato **ContactEditorUI** in una **scheda Editor** con una **barra degli strumenti** che contiene diversi pulsanti, come mostrato in ***figura***:



- il **form ContactEditor** aperto in visualizzazione Progettazione di GUI Builder
- **tre finestre** aggiuntive, lungo i bordi del IDE, che consentono di navigare, organizzare, costruire e modificare i form GUI.

Le **diverse finestre** del GUI Builder sono:



- **Design Area**. Finestra principale del GUI Builder per la creazione e la modifica dei form GUI.
- Il pulsante **Source** (**Codice sorgente**) della barra degli strumenti consente di visualizzare il codice sorgente di una classe che presenta aree protette, scritte dal GUI Builder (*vedi Tips and tricks – Guarded Blocks*), ed aree modificabili scritte dal programmatore.
- il pulsante **Design** (**Progettazione**) consente di visualizzare una rappresentazione grafica dei componenti GUI.

  Se è necessario **apportare modifiche al**

[19] *Per esplorare l'interfaccia GUI Builder con una demo interattiva, visualizzare l' esplorazione del GUI Builder (swf.) screencast.*

codice all'interno di un blocco custodito, facendo clic sul **pulsante Design Editor** restituisce l'IDE al GUI Builder in cui è possibile apportare le necessarie modifiche al modulo. Quando si salvano le modifiche, l'IDE aggiorna i file sorgente.

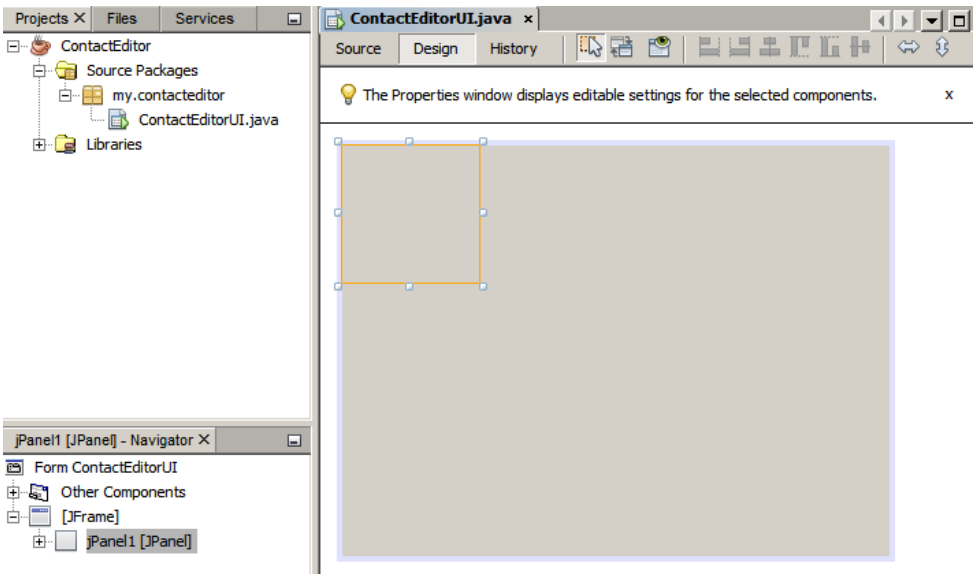
- il pulsante **History** (**Cronologia)** consente di accedere alla storia locale delle modifiche del file.
- I pulsanti della **barra degli strumenti supplementari** forniscono un comodo accesso ai comandi più comuni, come:
  - la scelta tra la Selection mode (Selezione),
  - la Connection mode (Connessione),
  - la Preview design (anteprima del form),
  - l'**allineamento dei componenti** (Align left in column, Align right in column, Center horizontally, Align top in row, Align bottom in row, Center vertically),
  - l'impostazione del componente di **auto-ridimensionamento** (Change horizontal reisaizability, Change vertical reisaizability).
- **Navigator**. Fornisce una rappresentazione di tutte le componenti, sia visivi e non visivi, in l'applicazione come una gerarchia ad albero. Il navigatore fornisce anche un **feedback visivo** dei componenti della struttura attualmente in corso di modifica nel GUI Builder e come consente di organizzare i componenti dei pannelli disponibili.
- **Palette** (tavolozza). Un elenco personalizzabile di componenti disponibili contenenti le schede per JFC / Swing, AWT e componenti JavaBeans, così come i gestori di layout. Inoltre, è possibile creare, rimuovere e riordinare le categorie visualizzate nel riquadro con il customizer.
  Per gli sviluppatori esperti, è disponibile il **Gestore palette** che consente di aggiungere alla palette **componenti personalizzati da JAR, library, o altri progetti** . Per aggiungere componenti personalizzati attraverso il **Gestore palette**, scegliere **Tools > Palette> Swing / AWT Components**.
- **Finestra Proprietà**. Consente di visualizzare le proprietà del componente attualmente selezionati nella GUI Builder, Finestra Navigator, Finestra Project, o la Finestra dei file.

3. Aggiungere componenti

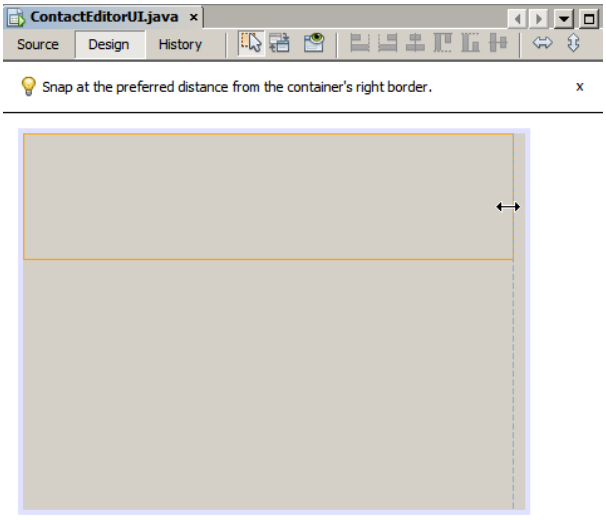
Ora iniziamo a sviluppare l'interfaccia utente della nostra applicazione **ContactEditor** utilizzando le **Palette** dell'IDE per aggiungere i vari componenti GUI di cui abbiamo bisogno per il nostro modulo.

Grazie al "**Disegno libero**" dell'IDE, tutto quello che dobbiamo fare è **trascinare** e **rilasciare** i **componenti** (drag and drop) necessari all'interfaccia grafica, come mostrato nelle ***figure*** che seguono.

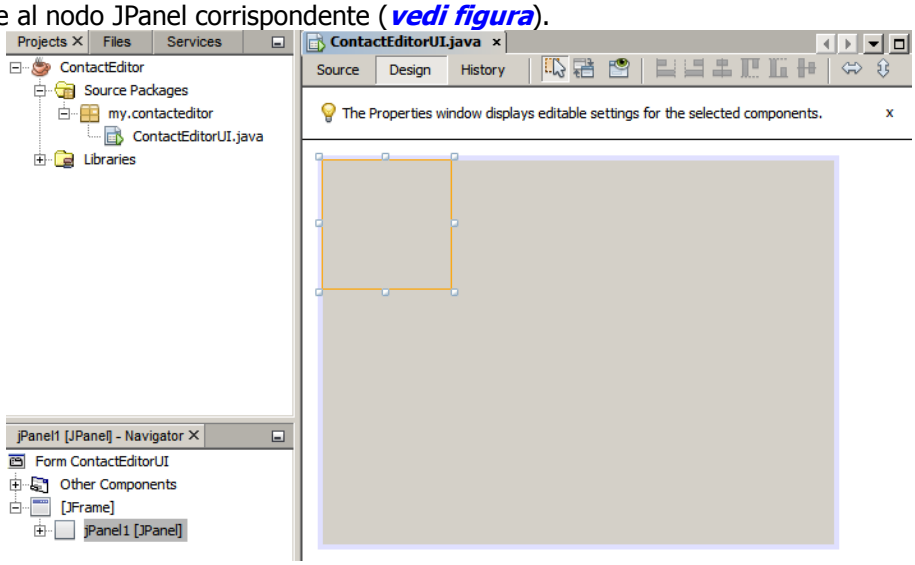
Dal momento che abbiamo già aggiunto un **JFrame** come form **contenitore** di livello superiore, il passo successivo è quello di aggiungere **un paio di JPanels** che ci permetteranno di **raggruppare** i **componenti** del nostro interfaccia.



Per **aggiungere** un **JPanel**:



- Nella finestra **Palette** (tavolozza), selezionare il componente **Panel** della categoria **Swing Containers** premendo e rilasciando il pulsante del mouse.
- Spostare il **cursore** verso l'angolo superiore sinistro del form nel GUI Builder (***vedi figura***) .
- Quando il componente si troverà in prossimità dei bordi superiore e sinistro del contenitore, **indicazioni di allineamento orizzontale e verticale** apparirà indicando i margini preferenziali.
- Cliccare il modulo per inserire il JPanel in questa posizione. Il componente JPanel appare sul form ContactEditorUI con **l'evidenziazione arancio ad indicare che è stato selezionato**.
- Dopo aver **rilasciato** il pulsante del mouse, intorno al perimetro del componente appaiono delle piccole "**maniglie**" di forma quadrata.
- Nella finestra **Navigator** viene visualizzato l'oggetto JPanel per mostrare le relazioni di ancoraggio del componente al nodo JPanel corrispondente (***vedi figura***).



- Ora dobbiamo ridimensionare il JPanel per fare spazio ai componenti che successivamente metteremo al suo interno.
- Il pannello, una volta **deselezionato**, **scompare** perché **non** abbiamo ancora aggiunto **bordo** e **titolo**, tuttavia, quando si passa il **cursore sul JPanel**, appaiono dei **bordi in grigio chiaro** in modo che la sua posizione possa essere vista chiaramente. È sufficiente fare clic in qualsiasi punto all'interno del componente per riselezionare e mostrare le maniglie di ridimensionamento e gli indicatori di ancoraggio.

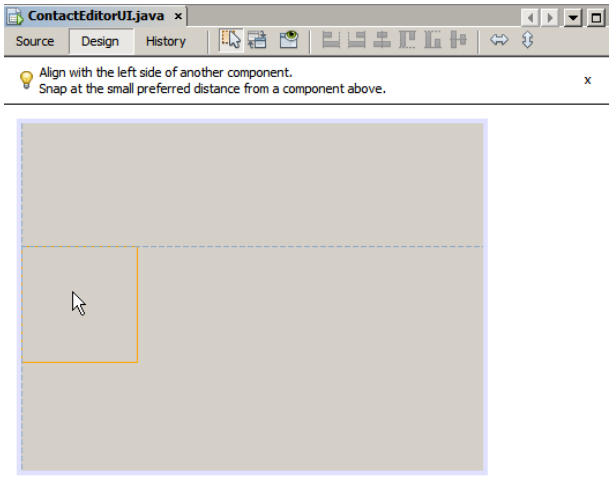
Per **ridimensionare** il JPanel:

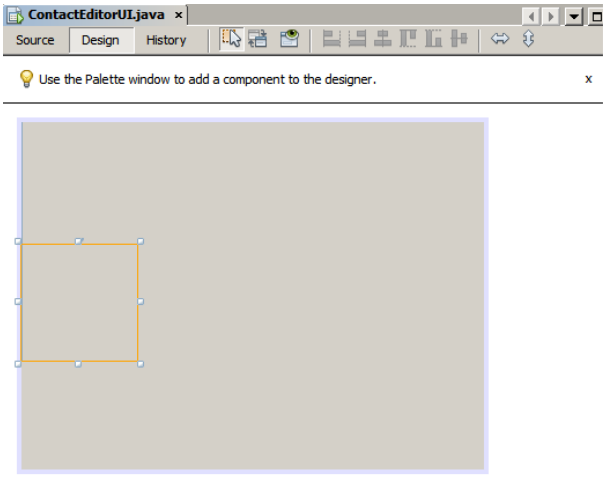
- **Selezionare** il JPanel appena aggiunto: le maniglie per il ridimensionamento riappariranno intorno al perimetro del componente.
- Fare clic e tenere premuto il **quadratino di ridimensionamento** sul bordo destro del JPanel e **trascinare** fino a quando la linea guida di allineamento tratteggiata appare vicino al bordo del form.
- Dopo aver ridimensionato il JPanel, **rilasciare** la maniglia di ridimensionamento.
- Il JPanel ora è esteso tra i margini sinistro e destro del contenitore secondo l'offset raccomandato (***vedi figura***).

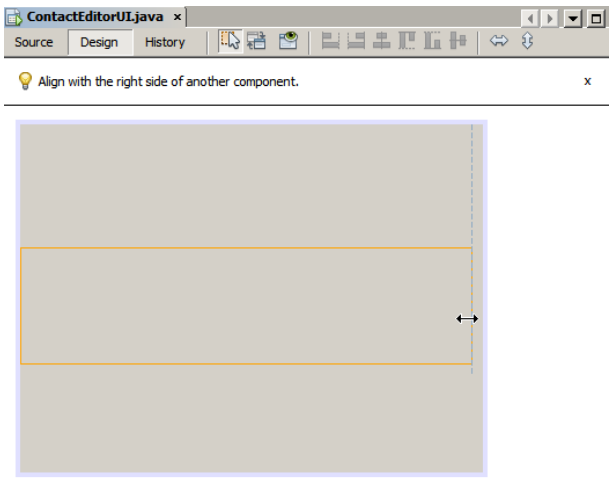
Ora che abbiamo aggiunto un pannello per contenere le informazioni della nostra UI, è necessario ripetere la procedura per aggiungere un altro direttamente sotto la prima. Con riferimento alle figure, ripetere le precedenti operazioni, facendo attenzione al posizionamento suggerito dal GUI Builder.

Si noti che la spaziatura verticale suggerita tra i due JPanels è più stretto di quello ai bordi.

Dopo aver **aggiunto il secondo JPanel** va ridimensionato in modo da riempire lo spazio verticale rimanente nel form.







Perché vogliamo distinguere visivamente le funzioni nelle sezioni superiore e inferiore della nostra interfaccia grafica, abbiamo bisogno di aggiungere un bordo ed altre proprietà ad ogni JPanel. Per prima cosa eseguire questa operazione utilizzando la finestra Proprietà ed il menù a comparsa.

Per aggiungere bordi ed il titolo al JPanels:

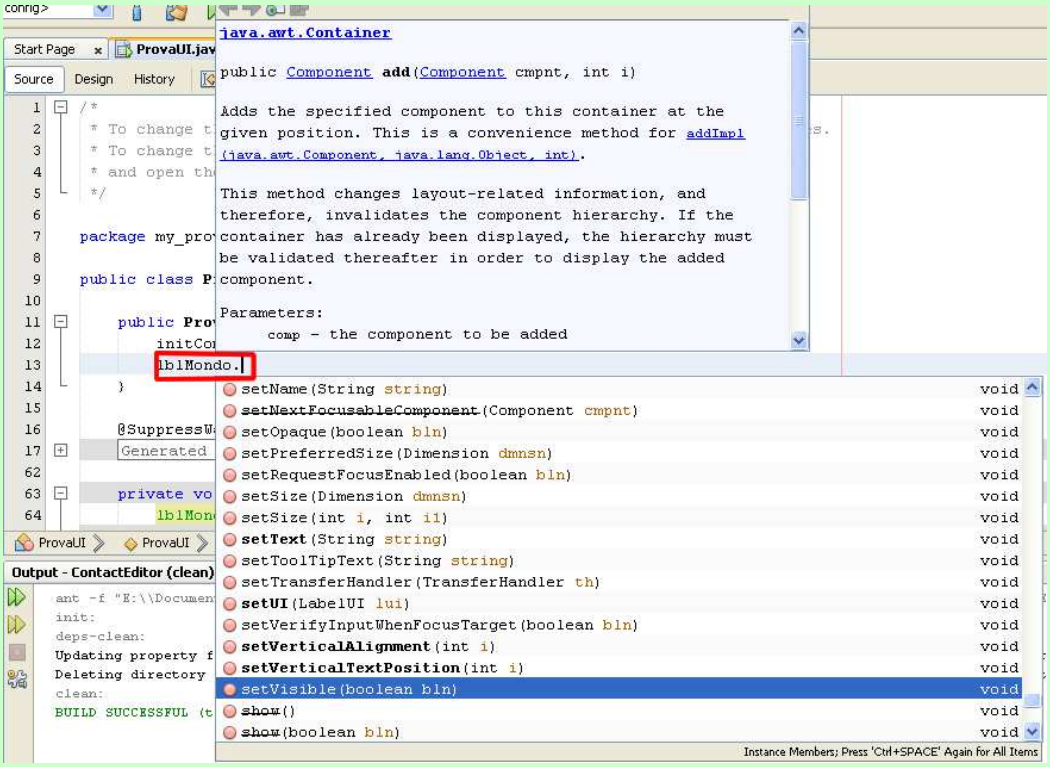
- Selezionare il JPanel superiore nella GUI Builder.
- Nella finestra Proprietà, fare clic sul pulsante **puntini di sospensione (...)** accanto alla **proprietà Border**.
- Nell'editor del Border del JPanel che viene visualizzato, selezionare il nodo **TitledBorder** nel riquadro **Available borders**.
- Nel riquadro **Proprietà** sottostante, specificare "**Nome**" per la proprietà **Title**.
- Fare clic sui **puntini di sospensione (...)** accanto alla proprietà **Font**, selezionare **Grassetto** per lo **stile** del carattere, e immettere **12** per le **dimensioni**. Fare clic su **OK** per chiudere le Finestre di dialogo.

Selezionare il **JPanel inferiore** e **ripetere** i passaggi da 2 a 5, ma questa volta fare clic con il JPanel e accedere alla Finestra delle proprietà utilizzando il menu a comparsa. Inserisci "**E-mail**" per la proprietà **Title**. **TitledBorder** vengono aggiunti ad entrambe le componenti JPanel.

Tips and tricks – Trovare metodi, proprietà ed eventi

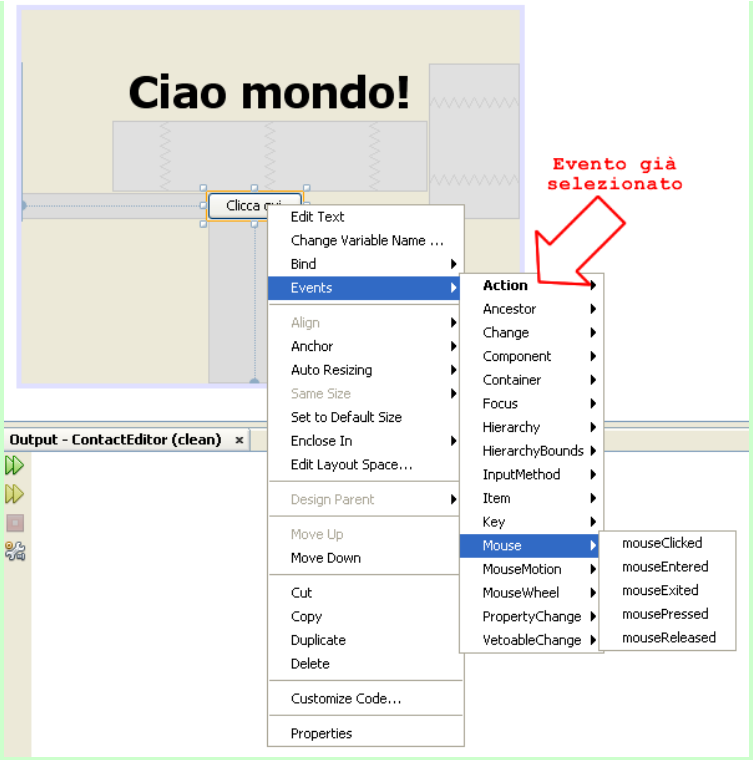
Lavorando ad un progetto potremmo aver bisogno di conoscerne **metodi**, **proprietà** ed **eventi** degli oggetti che utilizziamo; NetBeans ci viene in aiuto con alcuni semplici "trucchi"!

- Per trovare **metodi** e **proprietà** di ogni oggetto si può utilizzare l'**intellisense**[20]**.** Dopo aver digitato il nome dell'oggetto seguito dal **punto** automaticamente apparirà un **elenco di metodi e di proprietà** dai quali selezionare la voce interessata con un click.

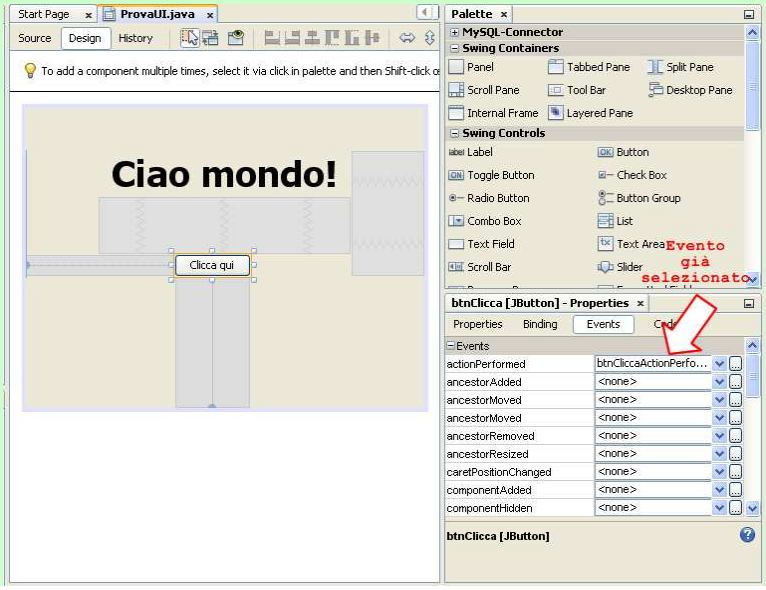


- Per trovare gli eventi è sufficiente fare clic, con il **tasto destro del mouse,** su qualsiasi componente GUI e selezionare **Eventi** dal menu a comparsa e sfogliare il menu per vedere cosa abbiamo a disposione anche senza effettuare selezioni.

[20] ***IntelliSense*** *è una forma di completamento automatico resa popolare da* ***Visual Studio Itegrated Development Environment****. Serve inoltre come documentazione per i nomi delle* ***variabili****, delle* ***funzioni*** *e dei* ***metodi*** *usando* ***metadati*** *e* ***reflection****. La* ***reflection*** *è la capacità di un* ***programma di eseguire elaborazioni che hanno per oggetto il programma stesso****, e in particolare la struttura del suo codice sorgente.*



Per trovare gli **eventi** è possibile anche fare clic sulla **scheda Eventi** della finestra **Proprietà** associata ad un particolare **oggetto**. Nella scheda **Eventi**, è possibile visualizzare e modificare i gestori degli eventi associati al componente GUI attivo.

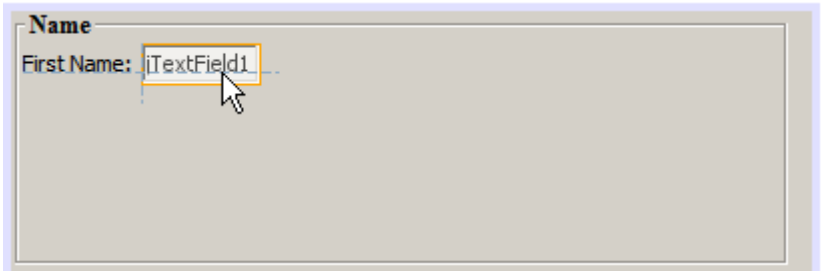


Componenti individuali

Ora abbiamo bisogno di iniziare ad aggiungere i componenti che presenteranno le informazioni effettive del "contatto" nella nostra lista contatti. In questo esempio aggiungeremo **quattro JTextFields** per visualizzare le informazioni di contatto e le **JLabels che li descrivono**.

Mentre facciamo queste operazioni possiamo notare le linee orizzontali e verticali che mostra il GUI Builder suggerendo la spaziatura consigliata fra le componenti. Questo garantisce che il GUI venga prodotto automaticamente rispettando l'aspetto che dovrà avere in fase di runtime.

Per aggiungere un **JLabel** al form:

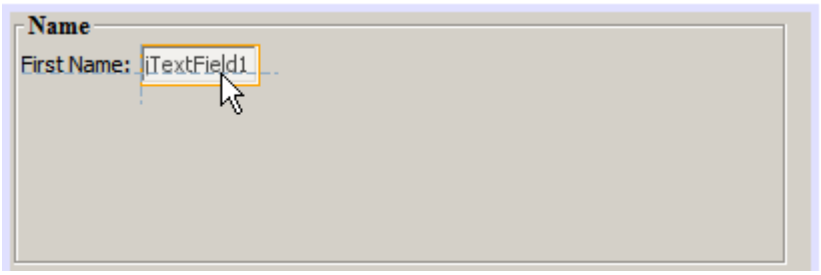


- Nella finestra Palette, selezionare il componente **Label** della categoria **swing controls**.
- Spostare il cursore sul JPanel "**Name**" aggiunto in precedenza.
- Quando appaiono le linee guida, che indicano che la JLabel è posizionata nell'angolo in alto a sinistra del **JPanel,** con un piccolo margine ai bordi superiore e sinistro, fare clic per posizionare l'etichetta. La JLabel verrà aggiunta sul form ed un nodo corrispondente che rappresenta il componente viene aggiunto alla finestra **Navigator**.

Prima di andare avanti, abbiamo bisogno di modificare il **testo visualizzato della JLabel** che abbiamo appena aggiunto. Anche se è possibile modificare il testo visualizzato nel componente in qualsiasi momento, il modo più semplice è quello di farlo quando vengono aggiunti.

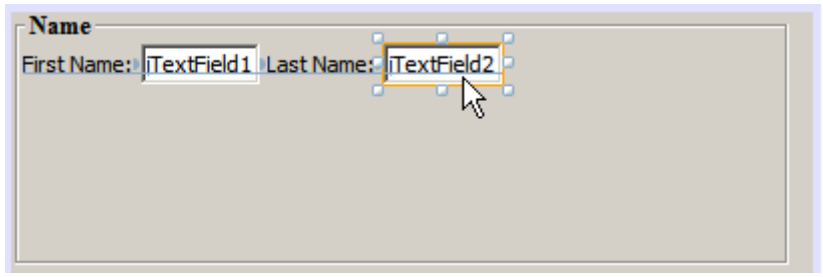
Per modificare il testo visualizzato in una **JLabel**:

- Fare **doppio clic** sulla **JLabel** per selezionare il testo che appare sulla label digitare **"First Name"** e premere Invio. Il nuovo nome JLabel viene visualizzato e la larghezza del componente si regola come risultato della modifica.
- Ora aggiungeremo un **JTextField** in modo da avere un'idea delle funzionalità di base dell'allineamento .

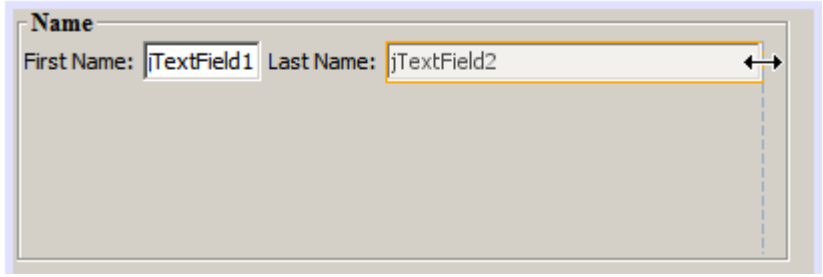


Per aggiungere un **JTextField** al form:

- Nella finestra palette, selezionare il componente **JTextfield**.
- Spostare il cursore immediatamente a destra di **"First Name"**, la JLabel che abbiamo appena aggiunto. Quando l'orientamento orizzontale che indica che la linea di base JTextField è allineato con quello della JLabel e la distanza tra i due componenti è suggerita con un orientamento verticale, fare clic per posizionare il JTextField. Il JTextField si posiziona sul form allineato con la linea di base JLabel, come mostrato in ***figura***. Si noti che la JLabel spostata leggermente verso il basso, al fine di allinearsi con il campo di testo di base più alto di. Come di consueto, un nodo che rappresenta il componente viene aggiunto alla finestra Navigator.



Prima di procedere oltre, è necessario aggiungere un **ulteriore JLabel** e **JTextField** immediatamente a destra di quelli che abbiamo appena aggiunto, come mostrato nella ***figura seguente***. Questa volta dovremo digitare "Last name" come testo da mostrare con la **JLabel.** Il testo segnaposto dei **JTextFields** non va per ora modificato.

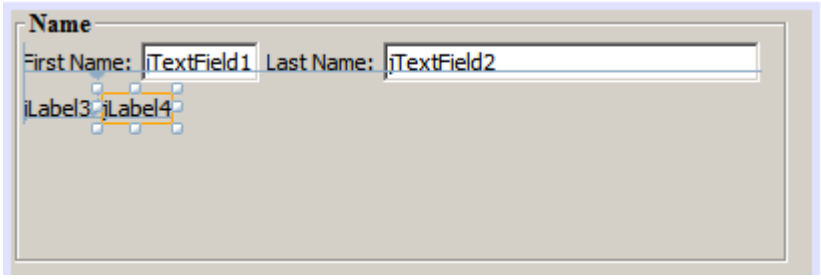


Per **ridimensionare** una **JTextField**:

- Selezionare la JTextField che abbiamo appena aggiunto alla destra della JLabel "Last name".
- Trascinare il bordo della JTextField verso il bordo destro del JPanel per ridimensionarlo.
- Quando le linee guida dell'allineamento **verticale** suggeriscono il margine tra il campo di testo e il bordo destro del JPanel, rilasciare il pulsante del mouse per ridimensionare la JTextField. Il bordo destro della JTextField è allineato con il margine sul lato consigliato del JPanel, come ***mostrato nella figura***.

Componenti multipli

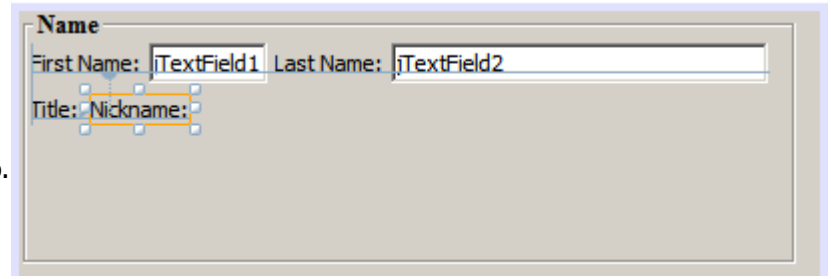
Ora **aggiungeremo le JLabels "Title" e "Nickname"** che **descriveranno le due JTextFields** che aggiungeremo. Trascinare e rilasciare i componenti tenendo premuto il **tasto Shift**, per aggiungere rapidamente al form. Anche in questo caso il GUI Builder mostrerà le linee guida orizzontali e verticali che suggeriscono la spaziatura preferito fra le componenti.
Per aggiungere **JLabels** più al form:



- Nella finestra palette, selezionare il componente **Label** della categoria swing controls premendo e rilasciando il pulsante del mouse.
- Spostare il cursore direttamente sul form sotto la JLabel "First Name" che abbiamo aggiunto in precedenza. Quando le linee guida che indica che il bordo sinistro del nuovo JLabel mostreranno che è allineato con quello della JLabel sopra ed esiste un piccolo margine tra di loro, premere **shift-clic** per **posizionare** la JLabel aggiunta prima.

Continuando a tenere premuto il tasto **shift**, posizionare un altra JLabel immediatamente a destra della prima. Assicurarsi di rilasciare il tasto **shift** prima di posizionare la seconda JLabel. Se si dimentica di rilasciare il tasto **shift** prima di posizionare l'ultima JLabel è sufficiente premere il tasto **ESC**. Le JLabels vengono aggiunti al modulo creando una seconda fila, come ***mostrato nella figura seguente***. I nodi che rappresentano ogni componente vengono aggiunti alla finestra del Navigator.

Prima di procedere, è necessario **modificare il nome delle JLabels** in modo da essere in grado di vedere l'effetto degli allineamenti stabiliti.

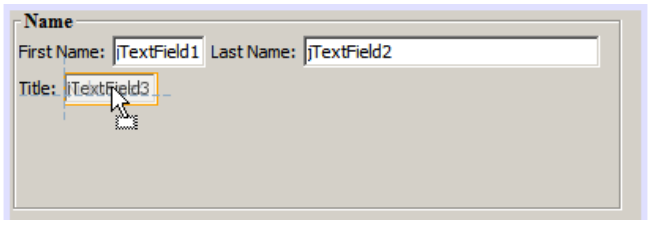


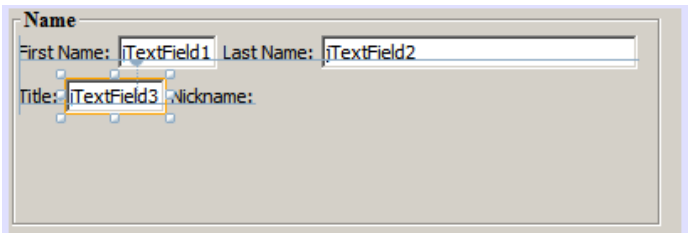
Per modificare il testo visualizzato di JLabels:

- Fare doppio clic sulla prima JLabel per selezionare il testo visualizzato.
- Digitare "Title" e premere Invio.
- Ripetere i passaggi precedenti, inserendo "Nickname" come proprietà "name" per la seconda JLabel. I nuovi nomi delle JLabels sono visualizzati sul form e vengono spostati a causa della loro larghezza, come ***mostrato in figura***.

4. Inserire componenti tra altri componenti

Spesso è necessario aggiungere un componente tra componenti già messi sul form. Ogni volta che si aggiunge un componente tra due componenti esistenti, il GUI Builder farà spazio per il nuovo componente. Per dimostrare questo, si inserirà una JTextField tra le JLabels abbiamo aggiunto in precedenza, come mostrato nelle due seguenti ***figure***.





Per inserire una **JTextField tra due JLabels**:

- Nella finestra palette, selezionare una componente **JTextField** dalla categoria swing controls.
- Spostare il cursore sopra la **JLabels** "Nickname", in seconda fila, in modo tale da sovrapporsi alla JTextField ed allinearsi alle loro linee di base.
- Se si incontrano difficoltà nel posizionare il nuovo campo di testo, è possibile scattare la linea guida sinistra del JLabel "Nickname" come mostrato nella prima figura.

- Fare clic per posizionare la **JTextField** tra le JLabels "Title" e "Nickname". La JTextField si posiziona tra le due JLabels. La JLabel più a destra sposta alla destra della JTextField.

Abbiamo ancora bisogno di aggiungere una JTextField aggiuntivo al modulo che venga visualizzato il "Nickname" di ogni contatto sul lato destro del modulo.

Per aggiungere una **JTextField**:

- Nella palette, selezionare una componente JTextField dalla categoria Swing controls.
- Spostare il cursore a destra dell'etichetta Nickname e fare clic per posizionare il campo di testo. La JTextField si posiziona accanto alla JLabel alla sua sinistra.

Per ridimensionare una **JTextField**:

- Trascinare i quadratini di ridimensionamento dell'etichetta del "Nickname".
- Quando le linee guida di allineamento verticale suggeriscono il margine giusto tra il campo di testo e i bordi del JPanel rilasciare il pulsante del mouse per ridimensionare la JTextField.
- Il bordo destro della JTextField è allineato con il margine sul lato opportuno del JPanel e la GUI Builder suggerisce il comportamento più appropriato per il ridimensionamento.

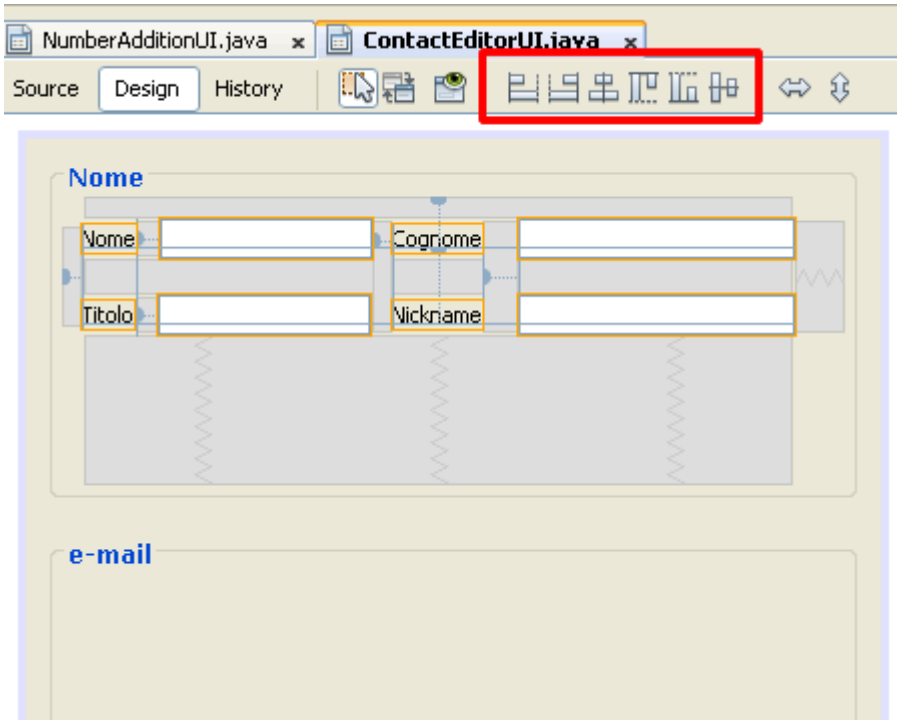
Premere Ctrl + S per salvare il file.

5. Componenti per l'allineamento

Come abbiamo visto, l'allineamento è uno degli aspetti fondamentali della creazione di un GUI professionale, per questo motivo, ogni volta che si aggiunge un componente a un modulo, il GUI Builder allinea efficacemente, come dimostrano le linee guida di allineamento che appaiono. Talvolta è necessario, tuttavia, **precisare diverse relazioni tra gruppi di componenti**. In precedenza abbiamo aggiunto quattro JLabels di cui avevamo bisogno per il GUI del nostro ContactEditor, ma non le abbiamo allineate. Ora allineiamo le due colonne di JLabels in modo che la loro linea destra bordi fino.

Per allineare i componenti:

- Tenere premuto il tasto **Ctrl** e fare clic per **selezionare** le JLabels "Firs name" (Nome) e **Titolo** sul lato sinistro del modulo.
- Fare clic sul pulsante **Allinea a destra** nella colonna nella barra degli strumenti. In alternativa, è possibile fare **clic destro** e scegliere uno dei due **Allinea > tasto destro** nella colonna dal menu a comparsa.



Ripetere l'operazione per le JLabels "Last name" (Cognome) e "Nickname".

Posizionare le JLabels in modo che i bordi destro del loro testo siano allineati. I rapporti di ancoraggio vengono aggiornati, indicando che i componenti sono stati raggruppati.

Ora dobbiamo fare in modo che le due JTextFields inserite tra le JLabels vengono ridimensionate correttamente. A differenza delle due JTextFields che si estendevano fino al bordo destro del form, inserendo il comportamento **resizeability** "non viene impostato automaticamente" ai vari componenti.

Per impostare **resizeability** come comportamento dei componenti:

- Fare **Control-clic** sulle due JTextField inseriti.

- Con entrambe le JTextFields selezionate, fare **clic** su uno di loro e scegliere il **ridimensionamento automatico > orizzontale dal menu a comparsa**. Le JTextFields vengono ridimensionate in orizzontale in fase di esecuzione. Le linee guida di allineamento e gli indicatori di ancoraggio vengono aggiornati, fornendo un feedback visivo delle relazioni tra i componenti.

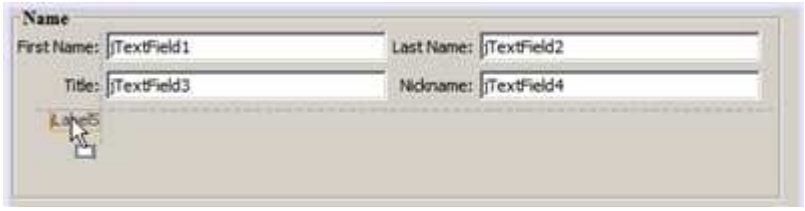
Per impostare la **stessa dimensione** per tutte le componenti:

- Fare **Control-clic** su tutte e quattro le JTextFields sul form per selezionarle.
- Con le JTextFields selezionate, fare **clic** con uno di essi e scegliere **Same size> same width dal menu a comparsa**. Le JTextFields sono tutti impostati per **la stessa larghezza** e sono aggiunti indicatori al bordo superiore di ciascuna, fornendo feedback visivo delle relazioni dei componenti.

Ora abbiamo bisogno di **aggiungere un'altra JLabel che descrive il JComboBox** che permetterà agli utenti di selezionare il formato delle informazioni visualizzate nella nostra applicazione ContactEditor.

Per allineare una **JLabel** ad un **gruppo di componenti**:

- Nella finestra **palette**, selezionare il componente Label della categoria Swing.
- Spostare il cursore sotto le JLabels "First name" e "Title" sul lato sinistro del JPanel. Quando la linea guida che indica che il bordo destro del JLabel nuovo è allineato con i bordi a destra del gruppo di componenti di cui sopra (le due JLabels), fare clic per posizionare il componente.
- La JLabel si posiziona allineata a destra con la colonna di JLabels sopra, come mostrato nella ***figura*** seguente. La GUI Builder aggiorna le linee di allineamento di stato che indicano la spaziatura del componente e le relazioni di ancoraggio.



Come negli esempi precedenti, fare **doppio clic sulla JLabel** per selezionare il testo visualizzato e quindi digitare il nome da visualizzare. Si noti che quando le JLabels si posizionano, gli altri componenti si spostano per accogliere il testo da visualizzare.

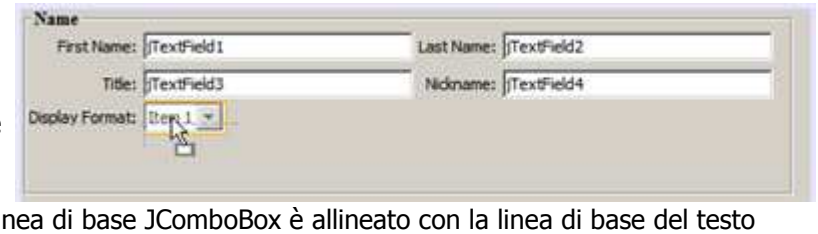
6. Allineamento alla linea di base

Ogni volta che si aggiungono o spostano i componenti che includono testo (JLabels, JTextFields, e così via), l'IDE propone allineamenti che si basano su linee di base del testo nei componenti. Quando si inserisce il JTextField precedenza, ad esempio, la linea di base è stata allineata automaticamente alle JLabels adiacenti.

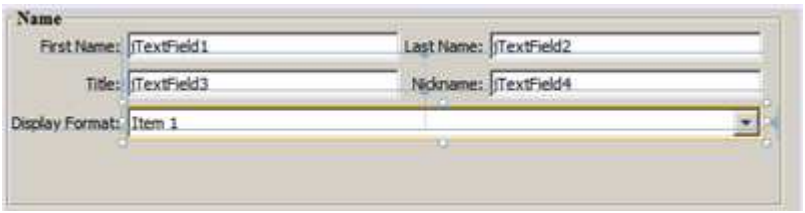
Ora aggiungeremo una JComboBox che permetterà agli utenti di selezionare il formato delle informazioni che l'applicazione ContactEditor visualizzerà. Come si aggiunge la JComboBox, si allinea la linea di base a quella del testo della JLabel di. Si notino ancora una volta le linee guida di base di allineamento che appaiono per aiutarci con il posizionamento.

Per allineare le componenti alle linee di base:

- Nella finestra palette, selezionare il componente JComboBox della categoria swing controls.
- Spostare il cursore immediatamente a destra del JLabel che abbiamo appena aggiunto. Quando la linea guida orizzontale che indica che la linea di base JComboBox è allineato con la linea di base del testo nella JLabel e la distanza tra i due componenti si consiglia con un orientamento verticale, fare clic per posizionare la casella combinata.



Il componente si posiziona allineato con la linea di base del testo della JLabel alla sua sinistra, come mostrato nella ***figura*** seguente. La GUI Builder mostra linee di stato che indicano la spaziatura del componente e le relazioni di

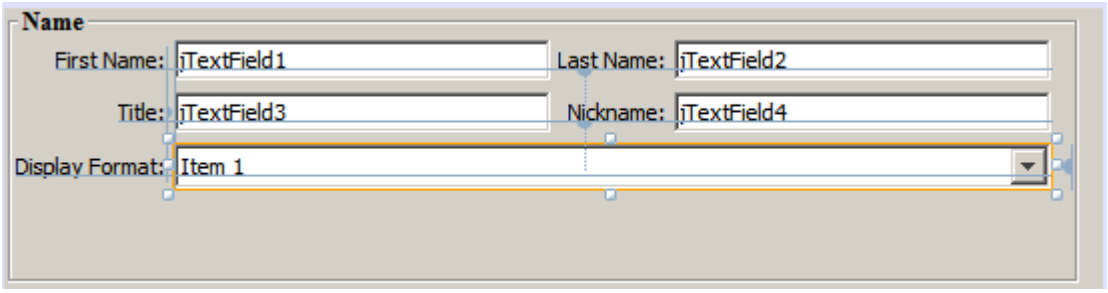


ancoraggio.

Per ridimensionare la JComboBox:

- Selezionare la ComboBox nella GUI Builder.
- Trascinare il quadratino di ridimensionamento sul bordo destro della JComboBox verso destra fino a quando le linee guida di allineamento appaiono suggerendo lo spazio consigliato tra il JComboBox e bordi del JPanel.
- Come mostrato nella ***figura*** seguente, il bordo destro della **JComboBox** si allinea con il margine sul lato consigliato del JPanel e la larghezza del componente viene impostata automaticamente in base alle dimensioni del form.
- Per il momento lasceremo la JComboBox così com'è.

Premere Ctrl + S per salvare il file.



Tips and tricks – Dimensionamento dei gap fra gli oggetti

Quando abbiamo **più oggetti "appoggiati"** sullo stesso **JPane**, **JInternalFrame** etc. può essere complicato impostare un GAP fra loro che venga poi mantenuto intatto in fase di esecuzione perchè alcuni oggetti si ridimensionano per riempire l'intera finestra e questo rende l'applicazione abbastanza brutta.

È possibile evitare questo facendo **clic destro** su ciascun componente **JPanel**, scegliendo **"Space Around Component..." / "Edit layout space..."** dal menù a comparsa e impostare gli spazi corrispondenti manualmente.

7. Aggiunta, allineamento e ancoraggio

Fino ad ora ci siamo concentrati su l'aggiunta di componenti al nostro GUI ContactEditor utilizzando le indicazioni di allineamento che l'IDE fornisce per aiutarci con il posizionamento.

È importante comprendere, tuttavia, che un'altra parte integrante del posizionamento dei componenti è l'**ancoraggio** che abbiamo già utilizzato anche senza rendercene conto.

Come accennato in precedenza, ogni volta che si aggiunge un componente ad una maschera, l'IDE propone il look di destinazione e il posizionamento preferito con le linee guida.

Una volta posizionati i **nuovi componenti** vengono **ancorati al bordo più vicino del contenitore** o componente che garantisce che le relazioni fra i componenti verranno mantenuti in fase di esecuzione.

Ogni volta che si aggiunge un componente a un form, il GUI Builder individua automaticamente le posizioni preferite e imposta i rapporti di concatenamento necessari in modo da potersi concentrare sulla progettazione dei form piuttosto che lottare con complicati dettagli di implementazione.

Per **aggiungere**, **allineare** e **modificare** il testo visualizzato da un **JLabel**:

- Nella finestra **palette**, selezionare il componente **JLabel** della categoria swing controls.
- Spostare il cursore sul form immediatamente al di sotto del titolo del JPanel "e-mail. Quando le linee guida, indicando che è posizionato nell'angolo in alto a sinistra del JPanel con un piccolo margine ai bordi

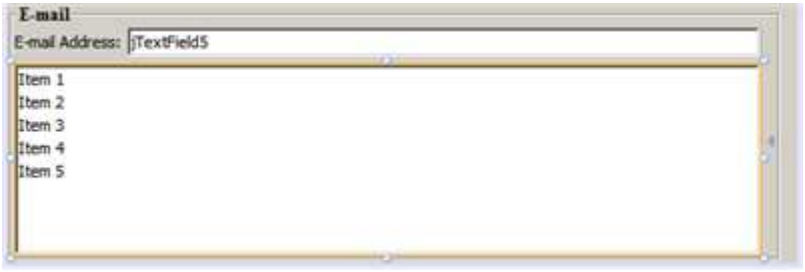
superiore e sinistro, fare **clic** per posizionare la JLabel.
- Fare **doppio clic** sul **JLabel** per selezionare il **testo** visualizzato. Quindi digitare **Indirizzo e-mail:** e premere Invio.
- La **JLabel** andrà nella posizione giusta sul form, ancorato ai bordi superiore e sinistro del JPanel. Come prima, un nodo corrispondente che rappresenta il componente viene aggiunto alla finestra Navigator.

Per **aggiungere un JTextField**:

- Nella finestra **palette**, selezionare il componente **JTextField** dalla categoria swing controls.
- Spostare il cursore immediatamente a destra della etichetta **Indirizzo e-mail** che abbiamo appena aggiunto. Quando le linee guida che indicano che la **linea di base JTextField** è **allineato** con la linea di **base** del testo della **JLabel** ed il **margine** tra le due componenti si adatta ad un orientamento verticale, fare clic per posizionare il campo di testo.
- Il **JTextField** si posiziona sulla destra della JLabel **Indirizzo e-mail** ed si collega al JLabel. Il nodo corrispondente viene aggiunto alla finestra Impostazioni.
- Trascinare il **quadratino** di **ridimensionamento** della **JTextField** verso la destra del JPanel che contenitore fino a quando le linee guida di allineamento visualizzate indicheranno la differenza tra il JTextField e bordi JPanel.
- Il **bordo destro** del **JTextField** spezza la linea guida che indica l'allineamento dei margini preferenziali.

Ora abbiamo bisogno di aggiungere la JList in cui verrà visualizzato l'intero elenco dei contatti del nostro ContactEditor.

Per **aggiungere e ridimensionare una JList**:



- Nella finestra **palette**, selezionare il componente **JList** della categoria swing controls.
- Spostare il cursore immediatamente sotto la JLabel **Indirizzo e-mail** aggiunto in precedenza. Quando le linee guida che indicano che i bordi superiore e sinistro della JList sono allineati con i margini preferenziali lungo il bordo sinistro del JPanel e della JLabel sopra, fare clic per posizionare la **JList**.
- Trascinare la **maniglia destra** di ridimensionamento della JList verso la destra del JPanel finché le indicazioni di allineamento indicheranno che è la stessa larghezza della JTextField sopra.
- La JList si posiziona nel modo designato dalle linee guida di allineamento e il nodo corrispondente viene visualizzato nella finestra Navigator. Si noti inoltre che il form si espande per accogliere la JList appena aggiunta

Poiché **JLists** vengono utilizzati per **visualizzare lunghi elenchi** di dati, che in genere richiedono l'aggiunta di un **JScrollPane**. Ogni volta che si aggiunge un componente che richiede un JScrollPane, **GUI Builder lo aggiunge automaticamente**.
Perché **JScrollPanes non** sono componenti **visuali**, è necessario utilizzare la finestra **Impostazioni** per visualizzare o modificare qualsiasi JScrollPane creato dal GUI Builder:

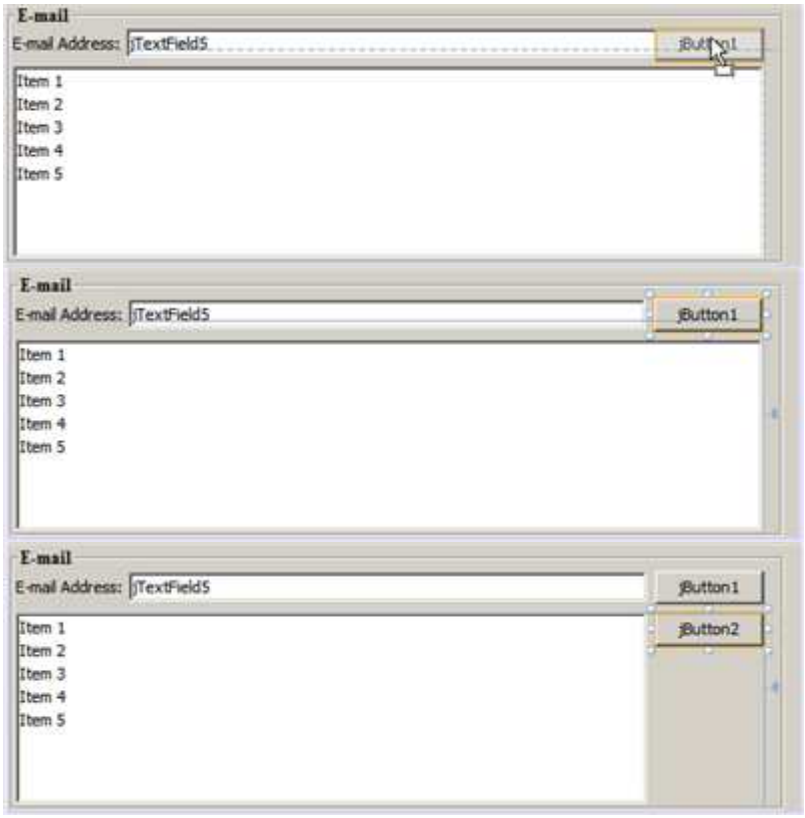
8. Dimensionamento dei componenti

Spesso è utile per impostare diverse componenti correlati, come i bottoni, con le stesse dimensioneiper la coerenza visiva. Per dimostrare questo, aggiungiamo **quattro JButtons** al nostro modulo ContactEditor che ci permetteranno di aggiungere, modificare e rimuovere le singole voci della nostra lista di contatti, come mostrato nelle ***figure*** seguenti.
In seguito, per i quattro pulsanti, verranno impostate le stesse dimensione in modo che si possa facilmente riconosciuto come offrire funzionalità correlate.

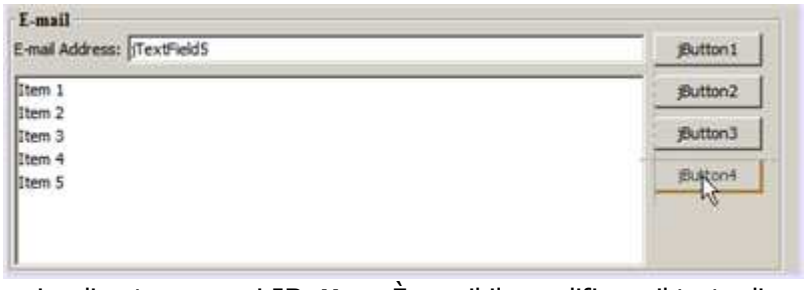
Per aggiungere, allineare, e modificare su più tasti il testo visualizzato:

- Nella finestra palette, selezionare il componente **JButton** dalla categoria swing controls.
- Spostare il JButton sul bordo destro della JTextField "e-mail address" nel JPanel inferiore. Quando le linee guida che indicano che la linea di base del **JButton** ed il bordo destro sono in linea con quella del **JTextField**, **tenere premuto shift e fare clic** per posizionare il primo pulsante lungo il bordo destro del JFrame. La larghezza della JTextField di riduce per adattarsi ai JButton quando si rilascia il pulsante

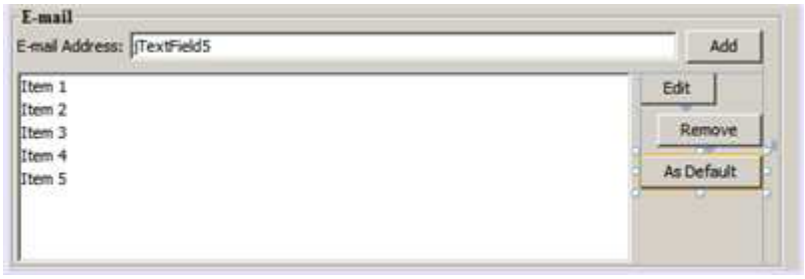
del mouse.



- Spostare il cursore in alto a destra della **JList** nel **JPanel** inferiore. Quando le linee guida sembrano indicare che i bordi superiore e sinistro del JButton sono in linea con quella del **JList**, **tenere premuto shift e fare clic** per posizionare il secondo pulsante lungo il bordo destro del JFrame.
- Aggiungere **due JButtons** aggiuntivi che seguono i due che abbiamo già aggiunto per creare una colonna. Assicurarsi di posizionare i JButtons in modo tale che la distanza consigliata sia rispettata e coerente. Se si dimentica di rilasciare il tasto **shift** prima di posizionare l'ultimo JButton, è sufficiente premere il tasto **ESC**.



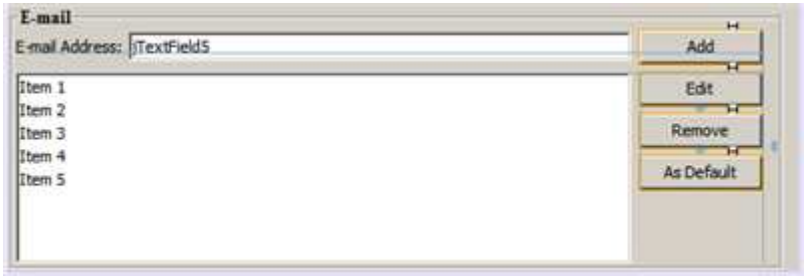
- Impostare il testo visualizzato per ogni **JButton**. È possibile modificare il testo di un pulsante con il tasto destro del mouse sul pulsante e scegliendo **Modifica testo**. In alternativa, è possibile fare clic sul pulsante di **pausa**, e quindi fare **clic** di nuovo. Inserire **Add** per il pulsante in alto, **Edit** per il secondo, **Remove** per il terzo, e **As Default** per il quarto.
- I componenti JButton scattano nelle posizioni indicate dalle linee guida di allineamento. La larghezza dei pulsanti cambia per accogliere i nuovi nomi.



Ora che i pulsanti sono posizionati dove vogliamo, impostiamo i quattro pulsanti in modo che abbiano la stessa dimensione in modo da avere la coerenza visiva e chiarire che sono collegate funzionalmente.

Per impostare componenti con la **stessa dimensione**:

- **Selezionare tutti e quattro i JButtons** premendo il tasto **Ctrl** durante la selezione.
- Il **pulsante destro del mouse** su **uno** di esse e scegliere **Same Size > Same Width** dal menu a comparsa.



I JButtons sono hanno ora la stessa dimensione che corrisponde a quella del pulsante con il nome più lungo.

9. Componenti multipli

Spesso è necessario avere componenti multipli in un altro componente in modo tale che sia chiaro che svolgeranno un gruppo di funzioni correlate.

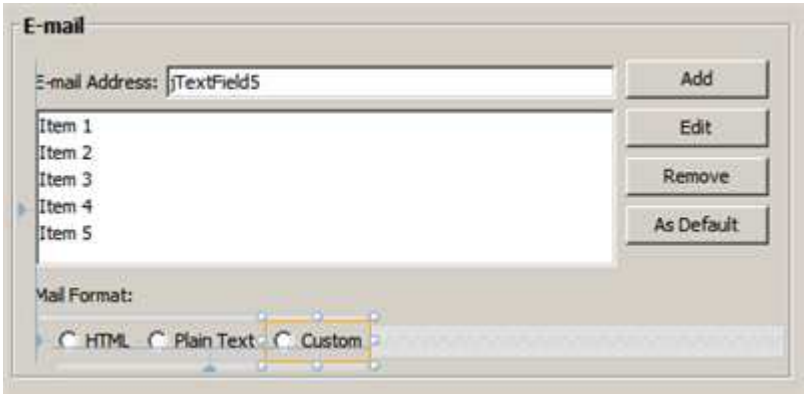
Un caso tipico, per esempio, è quello di **porre diversi checkbox sotto un'etichetta comune**. Il GUI Builder permette di realizzare facilmente il raggruppamento, fornendo specifiche linee guida che suggeriscono l'offset preferito per il look dell'interfaccia.

In questa sezione aggiungeremo **un paio di JRadioButtons al di sotto di una JLabel** in modo da permettere agli utenti di personalizzare il modo in cui l'applicazione visualizzi i dati.

Per **rientrare** JRadioButtons al di sotto di una JLabel, come si vede dalle ***figure*** seguenti:

- Aggiungere una JLabel nel form di sotto del JList. Assicurarsi che l'etichetta sia allineata a sinistra con la JList sopra.
- Nella **finestra palette**, selezionare il componente **JRadioButtons** della categoria Swing.
- Spostare il cursore sotto la JLabel che abbiamo appena aggiunto. Quando le linee guida sembrano indicare che il bordo sinistro del JRadioButton siano allineata con quello della JLabel, spostare il JRadioButton leggermente verso destra fino a quando le linee guida di rientro secondarie siano visualizzate. **Tenere premuto shift e fare clic** per posizionare il primo pulsante di opzione.

- Spostare il cursore a destra del primo JRadioButton, **tenere premuto shift e fare clic** per posizionare il secondo ed il terzo JRadioButtons, facendo attenzione a rispettare la spaziatura suggerita per le componenti. Assicurarsi di **rilasciare** il tasto **shift** prima di posizionare l'ultimo JRadioButton.
- Impostare il testo visualizzato per ogni JRadioButton. (È possibile modificare il testo di un pulsante con il tasto destro del mouse sul pulsante e scegliendo Modifica testo. In alternativa, è possibile fare clic sul pulsante di pausa, e quindi fare clic su di nuovo.) Inserire HTML per il pulsante di opzione a sinistra, Plain Text per il secondo, e Custom per il terzo.
- Tre JRadioButtons vengono aggiunti al form in modo rientrato al di sotto del JLabel "mail format".

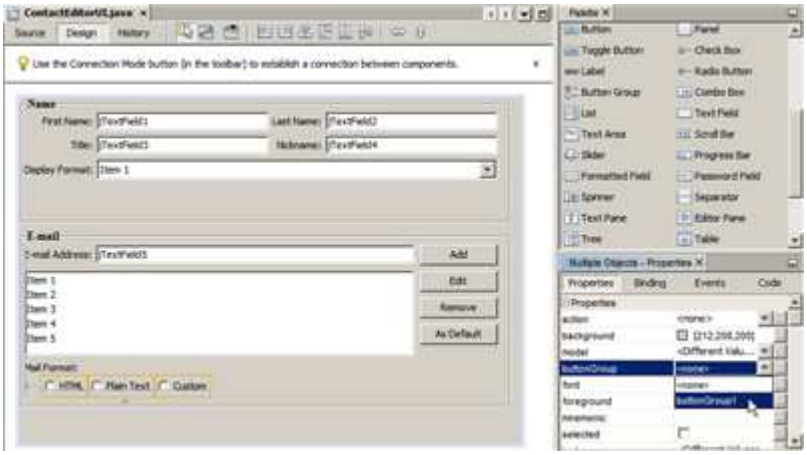


Ora abbiamo bisogno di aggiungere i **tre JRadioButtons** ad un **ButtonGroup** per attivare il comportamento previsto in cui **può essere selezionato un solo pulsante di opzione alla volta**.

Questo, a sua volta, consentirà alle informazioni di contatto della nostra applicazione ContactEditor di essere visualizzate nel formato mail scelto.

Per aggiungere un **JRadioButtons** ad un **ButtonGroup**:

- Nella finestra palette, selezionare il componente **ButtonGroup** della categoria swing controls.
- Fare **clic su un punto qualsiasi** dell'area di progettazione GUI Builder per aggiungere il componente **ButtonGroup** al form. Si noti che il ButtonGroup **non appare nel form** stesso, tuttavia, è visibile nella **zona Other** delle componenti del Navigator.
- **Seleziona tutti e tre i JRadioButtons** sul form.
- Nella casella combinata della proprietà ButtonGroup della finestra Proprietà selezionare **buttonGroup1**.
- I te JRadioButtons vengono ora aggiunti al gruppo di pulsanti.



Premere Ctrl + S per salvare il file.

9. Conclusione

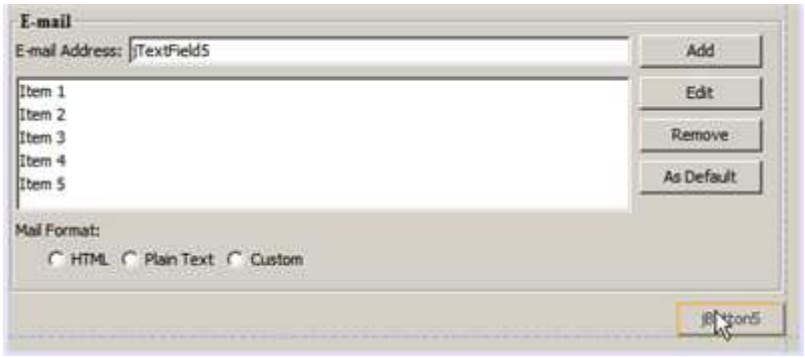
Ora abbiamo bisogno di aggiungere i pulsanti che consentono agli utenti di confermare le informazioni in input per un singolo contatto e aggiungerlo alla lista dei contatti o cancellare, lasciando invariato il database. In questo passo, aggiungeremo i due pulsanti necessari e poi modificarli in modo che appaiano le stesse dimensioni del nostro modulo, anche se il loro testo del display sono di lunghezza diversa.

Per aggiungere e modificare il testo di visualizzazione dei pulsanti:

Se il JPanel inferiore è esteso al bordo inferiore del form JFrame, trascinare verso il basso il bordo inferiore del JFrame. Questo vi dà spazio tra il bordo del JFrame e il bordo del JPanel per il vostro pulsanti OK e Annulla.

Nella finestra palette, selezionare il componente Button dalla categoria swing controls.

Spostare il cursore sul modulo sottostante JPanel e-mail . Quando le linee guida che indica che il bordo destro del JButton è allineato con l'angolo in basso a destra del JFrame, fare clic per posizionare il pulsante.



Aggiungere un'altra JButton alla sinistra del primo, facendo attenzione a posizionarlo utilizzando la spaziatura suggerito lungo il bordo inferiore del JFrame.

Impostare il testo visualizzato per ogni JButton. Inserisci OK per il tasto sinistro e Annulla per giusta. Si noti che la larghezza dei pulsanti cambia per accogliere i nuovi nomi.

Impostare i due JButton essere la stessa dimensione selezionando sia, tasto destro del mouse o, e la scelta dello stesso formato > Stessa larghezza dal menu a comparsa.

I componenti JButton appaiono sul form e nel loro nodi corrispondenti vengono visualizzati nella finestra di navigazione. Il codice componenti JButton 'si aggiunge anche file sorgente del modulo che è visibile nella visualizzazione Source della redazione. Ciascuno dei JButtons sono impostate le stesse dimensioni del pulsante con il nome più lungo.

Premere Ctrl + S per salvare il file.

L'ultima cosa che dobbiamo fare è eliminare il testo segnaposto nei vari componenti. Si noti che mentre la rimozione di testo segnaposto dopo la sgrossatura di un modulo può essere una tecnica utile per evitare problemi con allineamenti di componenti e le relazioni di ancoraggio, maggior parte degli sviluppatori di solito rimuovere questo testo mentre posa i componenti sul form. Come si passa attraverso il form, selezionare e cancellare il testo segnaposto per ciascuna delle JTextFields.

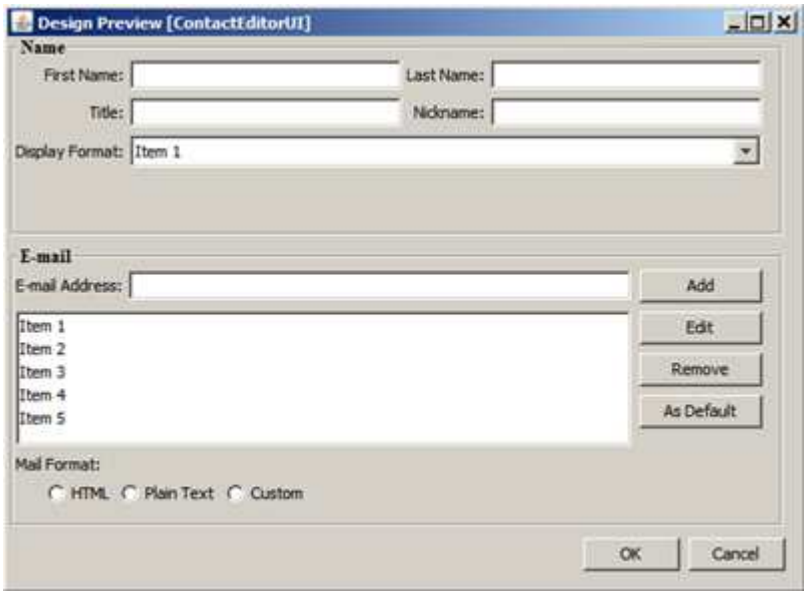
Lasceremo le voci segnaposto sia nel JComboBox che nel JList per un altro tutorial.

10. Anteprima del GUI

Dopo aver compilato correttamente la GUI di ContactEditor, si può provare l'interfaccia per vedere i risultati.

È possibile visualizzare l'anteprima del modulo, facendo clic sul pulsante **Preview Form** ( ) nella **barra degli strumenti del GUI Builder**.

Il modulo si apre in una finestra separata, che consente di provarlo prima di completare la costruzione e la gestione.



Affinché le interfacce create con il GUI Builder funzioni al di fuori dell'IDE, l'applicazione deve essere **compilata con le classi per la gestione di layout GroupLayout**[21] in modo che siano disponibili anche in fase di **runtime** (esecuzione).

Creazione del 2° GUI con Swing e gesione eventi

Il primo passo è quello di creare un GUI per l'applicazione che andremo a sviluppare. Chiameremo il nostro progetto **Addizione**.

- Scegliere **File> New Project** o, in alternativa, fare clic sull'icona New Project nella barra degli strumenti IDE.
- Nel riquadro **Categorie**, selezionare il nodo **Java**. Nel pannello **Project**, scegliere **Application Java**.
- Fare clic su **Avanti**.
- Digitare **Addizione** nel campo New Project e specificare un percorso, ad esempio, nella vostra home directory, come percorso del progetto.
- (Facoltativo) Selezionare la casella **Use Dedicated Folder for Storing Libraries** e specificare il percorso per la cartella librerie.
- **Deselezionare** la casella di controllo **Crea Main class** se è selezionata.
- Fare clic su **Finish**.

1. Creare l'interfaccia: contenitore JFrame

Per procedere con la costruzione della nostra interfaccia, abbiamo bisogno di **creare un contenitore** all'interno del quale **metteremo gli altri componenti GUI richiesti**. In questo passo creeremo un contenitore con il componente JFrame. Noi metteremo il contenitore in un nuovo pacchetto, che apparirà all'interno del nodo sorgente dei pacchetti.

- Nella Finestra **Projects**, fai **clic** sul nodo *Addizione* e scegliere **File > New file > Other**.
- Nella Finestra di dialogo **New file**, scegliere la categoria **swing GUI Forms** e il tipo di file **JFrame Form**.
- Fare clic su **Avanti**.

ovvero

- Nella Finestra Projects, fai clic con il tasto destro del mouse sul nodo *Addizione* e scegliere New > JFrame Form. Fare clic su Avanti.

poi:

---

[21] *Queste classi sono incluse in Java SE 6 , ma non in Java SE 5. Se si esegue l'IDE su JDK 6, l'IDE genera il codice dell'applicazione per utilizzare le classi GroupLayout che sono in Java SE 6. Ciò significa che è possibile distribuire l'applicazione per funzionare su sistemi in cui è installato Java SE 6.*

- Inserire *AddizioneUI* come nome della **classe**.
- Inserire *my_addizione* come nome del **pacchetto**.
- Fare clic su **Finish**.

L'IDE crea:

- il **form *Addizione*UI**
- la **classe *Addizione*UI** all'interno dell'**applicazione *Addizione***,
- apre il **modulo** *Addizione*UI nella GUI Builder.

Il **pacchetto** my_*addizione* sostituisce il pacchetto di default.

2. Creare l'interfaccia: aggiunta di componenti

Si utilizza la palette per popolare l'interfaccia dell'applicazione con:

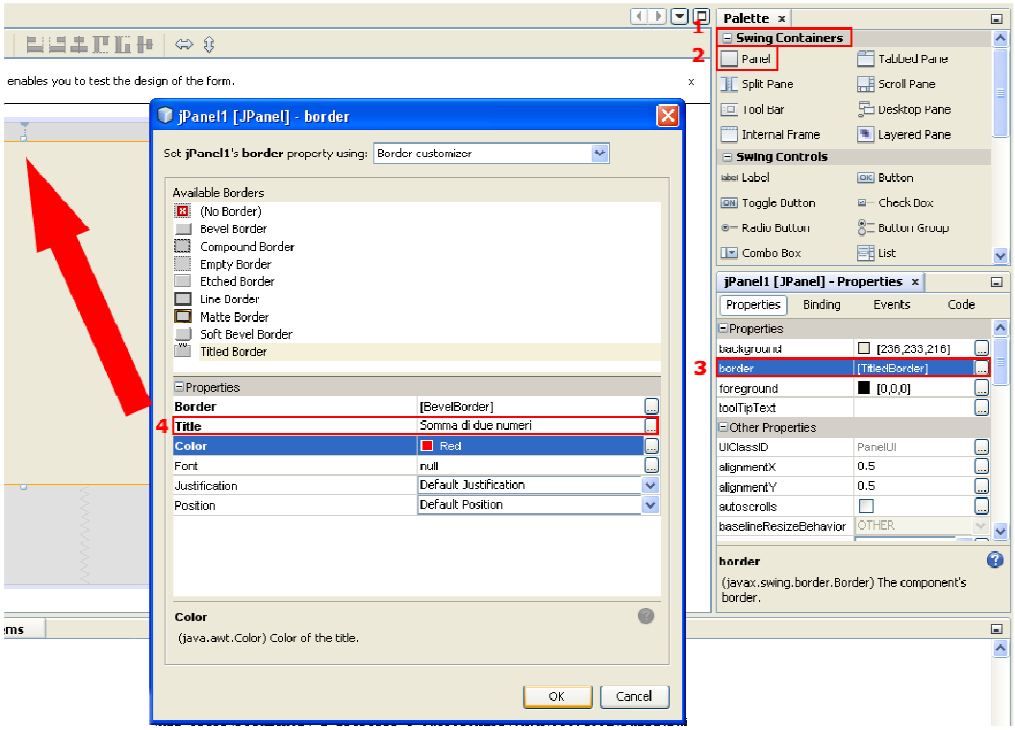
- un JPanel,
- tre JLabels,
- tre JTextFields,
- tre JButtons.

Dopo aver trascinato e posizionato i componenti di cui sopra, il JFrame dovrebbe essere qualcosa di simile alla schermata riportata di seguito.

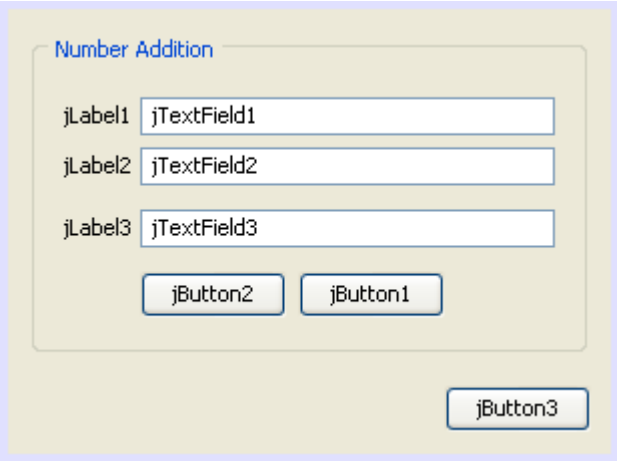
Se non viene visualizzata la **Finestra palette** nell'angolo in alto a destra dell'IDE, scegliete **Window > Palette**.

- Per iniziare, selezionare un gruppo di **Panel** della **categoria swing Containers** sulla palette e **rilasciarlo sul JFrame**.
- Mentre il **JPanel** è evidenziato, andare alla **finestra Proprietà** e fare clic sui **puntini di sospensione (...)** accanto a bordo e scegliere uno **stile di bordo**.
- Nella Finestra di dialogo **Border**, selezionare **TitledBorder** dalla lista, e il tipo in **Somma di due numeri** nel campo **Titolo**.
- Fare clic su OK per salvare le modifiche e chiudere la Finestra di dialogo.

Ora si dovrebbe vedere un JFrame vuoto con il titolo "**Somma di due numeri"** come nella ***figura***.



A questo punto aggiungiamo tre JLabels, tre JTextFields e tre JButtons ed otteniamo quello che si vede nella figure seguente.



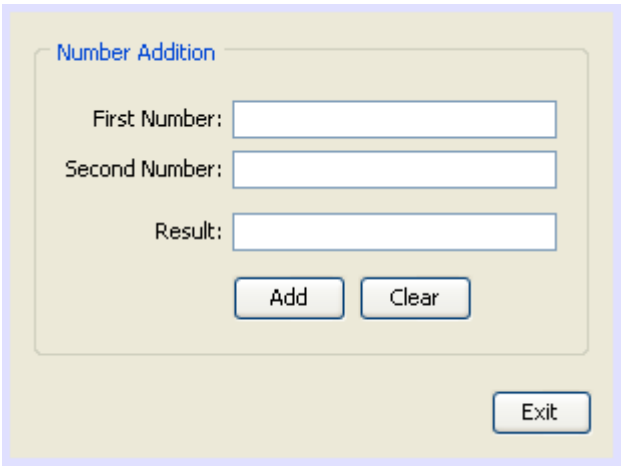
3. Creare l'interfaccia: rinominare i componenti

In questa fase andremo a rinominare il testo di visualizzazione dei componenti che sono stati appena aggiunti al JFrame.

- Fare **doppio clic** su **jLabel1** e modificare la proprietà di testo per **Primo Numero**
- Fare **doppio clic** su **jLabel2** e modificare il testo in **Secondo Numero**
- Fare **doppio clic** su **jLabel3** e modificare il testo di **Risultato**
- Eliminare il testo di esempio da **txtNumero1**. È possibile rendere il testo sul display modificabile facendo **clic destro del campo di testo** e scegliendo **Edit text** dal menu a comparsa. Potrebbe essere necessario **ridimensionare** la txtNumero1 alla sua dimensione originale.

ovvero

- Fare clic con il **tasto destro** sull'oggetto e scegliere **proprietà**.
- Fra le proprietà potrà essere opprtuno intervenire su:
  - **text**, per modificare il testo che dovrà apparire all'interno dell'oggetto;
  - **VerticalSize** ed **HorizontalSize**, per impostare le dimensioni dell'oggetto;
- Per assegnare un nuovo nome all'oggetto, ad esempio **txtNumero1,** fare **clic destro del campo di testo**, scegliendo **Change variable text...** dal menu a comparsa e digitare il nuovo nome del testo.
- Ripetere questo passaggio per txtNumero2 e txtRisultato.
- Rinominare il testo visualizzato di **jButton1** a **Cancella** ed assegnare il nome **btnCancella**. (È possibile modificare il testo di un pulsante con il tasto destro del mouse sul pulsante e scegliendo Modifica testo. In alternativa, è possibile fare clic sul pulsante di pausa, e quindi fare clic su di nuovo.)
- Rinominare il testo visualizzato di **jButton2** a **Somma ed assegnare il nome btnSomma**.
- Rinominare il testo visualizzato di **jButton3** a **Esci ed assegnare il nome btnEsci**.



Il tuo GUI finito dovrebbe apparire come in ***figura.***

4. Aggiungere funzionalità: l'evento ActionPerformed

Ora ci accingiamo a **dare funzionalità ai pulsanti** Somma, Cancella e Somma di due numeri.

Le caselle txtNumero1 e txtNumero2 saranno utilizzati per l'input dell'utente e txtRisultato per l'uscita programma - quello che stiamo creando è **un calcolatore molto semplice**.

Gestione evento per il pulsante Esci: gestione evento

Per dare ai tasti una funzione, è **necessario assegnare un gestore eventi** per ciascuna **per rispondere agli eventi**. Nel nostro caso vogliamo sapere quando il pulsante viene premuto, o con un clic del mouse o tramite tastiera. Quindi useremo un ActionListener in grado di rispondere all'**ActionEvent**.

Cliccare con il **tasto destro** del mouse sul pulsante **btnEsci** e, dal menu a comparsa, scegliere **Event > Action > ActionPerformed**. Si noti che il menu contiene molti eventi cui si può rispondere!

Quando si seleziona l'evento **ActionPerformed**, l'IDE aggiungerà **automaticamente un ActionListener** al pulsante **btnEsci** e generare un metodo per la gestione dell'ascoltatore actionPerformed.

L'IDE aprirà la **finestra del codice sorgente** e scorrendo fino al punto in cui si implementa l'azione che si desidera che il pulsante svolga quando si preme il pulsante (o con un clic del mouse o tramite tastiera). La finestra del codice sorgente deve contenere le seguenti righe:

```
private void btnEsciActionPerformed (java.awt.event.ActionEvent evt) {
     / / TODO aggiungere il codice di gestione qui:
}
```

Ora aggiungiamo il codice per ciò che vogliamo che faccia il tasto Esci. Sostituire la linea con la scritta "TODO…." con System.exit(0);

Il codice del pulsante di uscita dovrebbe essere simile a questo:

```
private void btnEsciActionPerformed (java.awt.event.ActionEvent evt) {
           System.exit (0);
}
```

Gestione evento per il pulsante Cancella

Fare clic sulla scheda **Design** nella parte superiore dell'area di lavoro per tornare alla **Struttura maschera**.

Tasto destro del mouse sul pulsante Cancella (btnCancella). Dal pop-up Eventi Selezionare **Menu> Azione> ActionPerformed**.

Il pulsante Cancella svolgerà ora la funzione di cancellare tutto il testo dalle jTextFields sovrascrivendo il testo esistente con uno spazio vuoto. Per fare questo, è necessario aggiungere del codice come quello che segue:

```
private void btnSommaActionPerformed (java.awt.event.ActionEvent evt) {
        txtNumero1.setText("");
        txtNumero2.setText("");
        txtRisultato.setText("");
}
```

Gestione evento per il pulsante Aggiungi

Con il pulsante Aggiungi si dovranno eseguire tre operazioni:

- accettare l'input dell'utente per i textFields (txtNumero1, txtNumero2),
- convertire l'input da tipo String a un flot,
- eseguire la summa dei due numeri,
- convertire la somma in tipo String e si mette in txtRisultato .

Per ottenere questo risultato:

- Fare clic su "**Design**" nella parte superiore dell'area di lavoro per tornare all'**area Design**.
- Fare clic con il tasto destro sul pulsante **Aggiungi** (btnCancella). Dal menu a comparsa, selezionare **Event > Action> ActionPerformed**.
- Inserire il codice di risposta all'evento **ActionPerformed** sul pulsante Aggiungi. Il codice sorgente sarà:

Codice JAVA	Descrizione
`private void btnCancellaActionPerformed (java.awt.event.ActionEvent evt) {` `  float num1, num2, risultato;`	`Definire le variabili float`

```
    num1 = Float.parseFloat (txtNumero1.getText ());
    num2 = Float.parseFloat (txtNumero2.getText ());
    risultato = num1 + num2;


    txtRisultato.setText (String.valueOf (risultato));

}
```

Convertire il testo in tipo float
Calcolo del risultato della somma
Passare il valore di **risultato** txtRisultato
Cambiare il valore di **risultato** da float in string

Il nostro programma è ora terminato e possiamo compilarlo ed eseguirlo per vederlo in azione.

5. Eseguire il programma

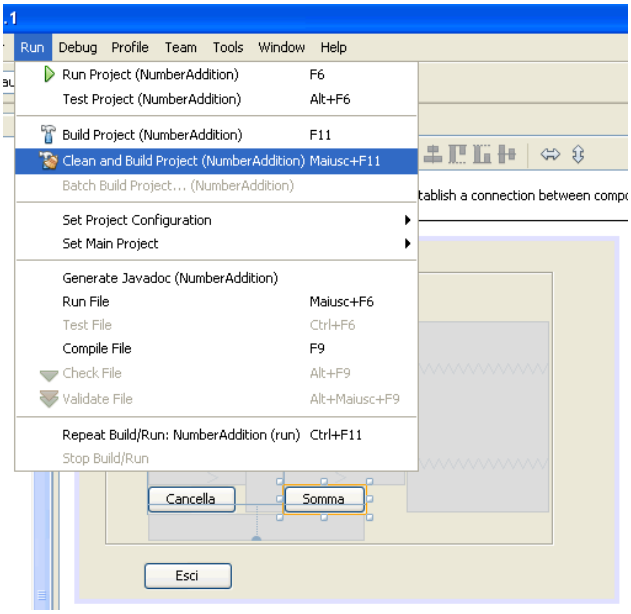
Per eseguire il programma nell'IDE:

- Scegliere **Run > Esegui** progetto principale (in alternativa, premere il tasto **F6**).

  Nota - Se si ottiene una Finestra che informa che il **progetto Addizione non ha una Main class** è necessario selezionare **my_addizione.AddizioneUI** come **classe principale** nella stessa Finestra e fare clic sul pulsante OK.

Per eseguire il programma al di fuori dell'IDE:

- Scegliere **Run > Clean and Build Main Project** (Maiusc + F11) per creare il file JAR dell'applicazione.
- Uso di Esplora file del sistema o file manager, accedere alla Addizione/dist directory.
  Nota - La posizione della directory del progetto Addizione dipende dal percorso specificato durante la creazione del progetto.
- Fare doppio clic sul file **Addizione.jar**.



Avviare l'applicazione dalla riga di comando.

- Aprire una Finestra del **prompt dei comandi** o terminale.
- Nel prompt dei comandi, **passare alla directory Addizione/dist**.
- Nella **riga di comando**, digitare la seguente istruzione: **java-jar Addizione.jar**

  Nota - Assicurarsi che **my_addizione.AddizioneUI** sia impostato come **classe principale** prima di eseguire l'applicazione. È possibile controllare questo pulsante destro del mouse sul nodo del progetto Addizione nel riquadro Project, scegliere Proprietà dal menu a comparsa, e selezionando la categoria Esegui nella Finestra di dialogo Proprietà. Il campo Classe principale dovrebbe visualizzare my_addizione.AddizioneUI.

Dopo alcuni secondi, l'applicazione dovrebbe iniziare.

> **Nota** - Se facendo doppio clic sul file JAR non si avvia l'applicazione controllare come sono impostate le associazioni di file JAR nel sistema operativo.

La classe JTable JOptionPane

Per implementare semplici finestre di pop-up per comunicare **messaggi** agli utenti, richiedere un **valore in input** o **chiedere conferma** (sì/no/annulla), non è necessario implementare una struttura basata su **frames, pannelli, componenti e listener**: ci si può infatti avvalere degli **strumenti** forniti dalla **classe JOptionPane**

JOptionPane è una **classe** che **estende** la **JComponent** (dunque fa parte del package **Swing**) e che

permette di visualizzare **finestre di pop-up modali** (ossia: che bloccano il flusso dell'esecuzione del **thread** che le ha invocate finchè l'utente non le chiude).

La **classe** mette a disposizione una gran quantità di **metodi** (per informazioni dettagliate ed un **elenco completo**, vedi API ufficiali), ma la maggior parte di questi è riconducibile alla visualizzazione di **quattro tipi di finestre**:

- **finestre di conferma**, dove l'utente deve cliccare su pulsanti sì-no-annulla;
- **finestre di input**, dove è richiesto l'inserimento di un valore;
- **finestre di messaggio**, che informano l'utente riguardo qualcosa (messaggio da impostare dal programmatore);
- **finestre di opzione**, che racchiudono tutti i tipi precedentemente elencati.

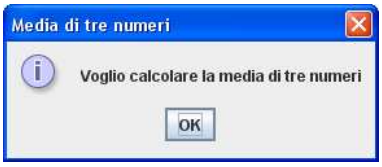
A queste **funzionalità** corrispondono i **metodi**:

- **showConfirmDialog;**
- **showInputDialog;**
- **showMessageDialog;**
- **showOptionDialog.**

Nella maggior parte dei casi, i **parametri** hanno significati ovvi, mentre per altri è necessaria una piccola trattazione:

- **parentComponent** identifica il **componente 'padre'** della **dialog box**; è possibile utilizzare anche il valore **null**, che utilizzerà un **frame** di default come padre;
- ***messageType*** definisce lo **stile** del messaggio: se si tratta di una informazione generica, una domanda, un avviso di '**pericolo**'.... a seconda dei casi, verranno mostrati dei **simboli** nella **dialog**. I valori possibili, come indicato in ***figura***, sono i seguenti:
  - ERROR_MESSAGE,
  - INFORMATION_MESSAGE,
  - WARNING_MESSAGE,
  - QUESTION_MESSAGE,
  - PLAIN_MESSAGE;

ERROR_MESSAGE



INFORMATION_MESSAGE

WARNING_MESSAGE

QUESTION_MESSAGE

- ***optionType*** definisce il set di **pulsanti** per le opzioni che verranno mostrati sul fondo della **dialog box**; valori possibili (ma non gli unici possibili):
  - DEFAULT_OPTION,
  - YES_NO_OPTION,
  - YES_NO_CANCEL_OPTION,
  - OK_CANCEL_OPTION.
- Quando una **finestra di dialogo** restituisce un **intero**, il **valore** di quest'ultimo può essere uno tra i seguenti (**costanti predefinite**):
  - YES_OPTION,
  - NO_OPTION,
  - CANCEL_OPTION,

- OK_OPTION,
- CLOSED_OPTION.

La seguente porzione di **codice** definisce una **classe** eseguibile:

Codice JAVA	Descrizione
`import javax.swing.JOptionPane;`	importare la classe "**JOptionPane**" che fa parte del package "swing" per utilizzare le finestre di input
`// import javax.swing.*;`	importare un intero package utilizzando l'asterisco al posto del nome della classe
`public class ProvaJOptionPane{` `  public static void main(String[] args){`	

`    String messaggio = null;`
`    int conferma = 0;`
`    String output = null;`
	definizione variabili

`    messaggio = JOptionPane.showInputDialog(null, "Inserisci un messaggio", "Inserisci una stringa", JOptionPane.PLAIN_MESSAGE);`	finestra input dialog
`    conferma = JOptionPane.showConfirmDialog(null, "Seleziona sì, no o cancella", "Scegli un'opzione", JOptionPane.YES_NO_CANCEL_OPTION, JOptionPane.PLAIN_MESSAGE);` `    output = "Hai scritto: \"" + messaggio + "\" ed hai cliccato sul pulsante: " + conferma;`	finestra confirm dialog
`    JOptionPane.showMessageDialog(null, output);`	finestra message dialog
`    System.exit(0);`	termina l'**esecuzione**
`  }` `}`	

Tips and tricks – Importare una classe

Per importare una classe, ad esempio la classe "JOptionPane" che fa parte del package "swing", si può **importare direttamente la classe**, ad esempio: "javax.swing.JOptionPane", o l'intero package che la contiene utilizzando l'**asterisco** al posto del nome della classe, ad esempio: "javax.swing.*".

In ogni caso con **l'importazione non si inserisce nulla** quindi anche l'uso della **importazione globale non inserisce nulla** ma comunica al compilatore di utilizzare il package indicati per ricercare classi che non sono state trovate.

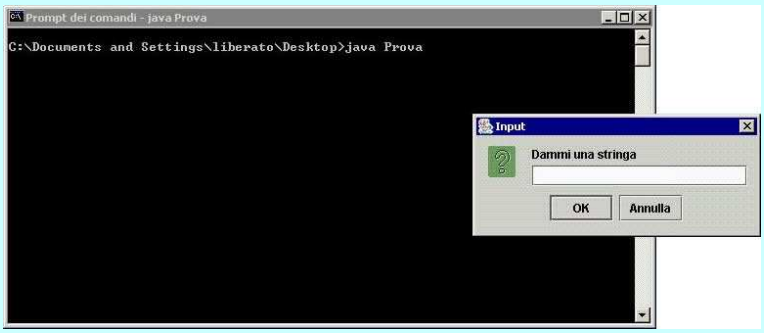
Esercizi svolti

Vediamo alcuni esempi di input da tastiera con l'uso della **classe JOptionPane.**

1. Aprire una finestra in cui si possa immettere una **stringa**, visualizzare in otput la stringa inserita.

```
import javax.swing.*;                          import per utilizzare una finestra di input

class Prova {                                  esecuzione del programma si ferma alla
  public static void main(String args[]) {     invocazione del metodo

    String s;                                  Definizione variabili da utilizzare
```

```
    s=JOptionPane.showInputDialog("Dammi una stringa");   finestra di input con una stringa fra parentesi che dice all'utente cosa deve inserire

    // s=JOptionPane.showInputDialog("");                 finestra di input senza messaggio per l'utente

    System.out.println(s);                                Visualizzazione della risposta alla domanda fatta con il metodo showInputDialog.
                                                          La risposta è sempre una stringa. Se serve un numero la variabile deve essere convertita.

    System.exit(0);                                       chiusura del programma
  }
}
```



2. Aprire una finestra in cui si possa immettere un **numero**, visualizzare in otput il numero inserito.

```
import javax.swing.*;                                 import per utilizzare una finestra di input

class InputInt {                                      esecuzione del programma si ferma alla
  public static void main(String args[]) {            invocazione del metodo

    String s;                                         Definizione variabili da utilizzare
    int x;
    s=JOptionPane.showInputDialog("Dammi un numero");  finestra di input con una stringa fra parentesi che dice all'utente cosa deve inserire

    x=Integer.parseInt(s);                            conversione della stringa “s” in numero “x” con il metodo Integer.parseInt

    System.out.println(x+20);                         Visualizzazione della risposta alla domanda fatta con il metodo showInputDialog

    System.exit(0);                                   chiusura del programma
  }
}
```

Vediamo un esempio di input da tastiera con la classe **showInputDialog**, questa classe restituisce un oggetto e questo oggetto è generalmente **una stringa** che riflette la scelta dell'utente

3. Aprire una finestra in cui si possa immettere due **numeri interi e fare la somma** da restituire in output.

```
import javax.swing.*;                                 import per utilizzare una finestra di input
class Somma {
  public static void main(String args[]) {
    int x, y;                                         Definizione variabili da utilizzare
    String s;
    s=JOptionPane.showInputDialog("Dammi il primo numero");   Input del primo numero
    x=Integer.parseInt(s);                            Conversione del primo dato in intero
    s=JOptionPane.showInputDialog("Dammi il secondo numero");  Input del secondo numero
    y=Integer.parseInt(s);                            Conversione del secondo dato in intero
```

```
    System.out.println("Ecco il risultato");          Messaggo per l'utente
    System.out.println(x+y);                          Calcolo della somma e visualizzazione
    System.exit(0);
  }
}
```

La classe JTable

Le liste permettono di avere una sequenza di righe. In alcune applicazioni è però necessario visualizzare le informazioni anche in colonne

Nome	Cognome	Indirizzo	Telefono
Mario	Bianchi	Via Roma, 12	059/1111111
Franco	Rossi	Via Milano, 33	059/2222222

La **libreria swing** mette a disposizione un componente per la rappresentazione di informazioni in forma tabellare, implementato dalla classe JTable
Ogni tabella implementata con JTable recupera i dati da rappresentare tramite un modello, istanza di una classe che implementa l'interfaccia **DefaultTableModel.** Questa interfaccia mette a disposizione metodi per sapere quante righe/colonne servono, qual è il valore di ogni singola cella, se le celle sono editabili, ecc.
È disponibile anche una classe astratta, AbstractTableModel, che implementa la maggior parte dei metodi di **DefaultTableModel**, e lascia al programmatore da implementare solo i tre metodi:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

Esercizi svolti

Vediamo alcuni esempi su come inserire righe in una **JTable** in posizioni o campi specificati.

1. Questo programma crea una tabella che contiene 3 righe e 2 colonne.

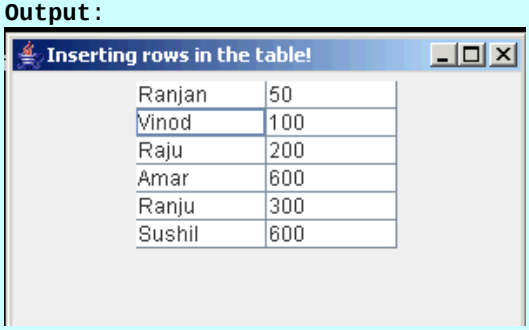
```
import javax.swing.*;
import javax.swing.table.DefaultTableModel;
public class InsertRows{
   public static void main(String[] args) {
     JFrame frame = new JFrame("Inserting rows in the table!");
     JPanel panel = new JPanel();
     String data[][] = {{"Vinod","100"},{"Raju","200"},{"Ranju","300"}};   Dati da inserire nella JTable
     String col[] = {"Name","code"};                                       Intestazione della JTable
     DefaultTableModel model = new DefaultTableModel(data,col);            Questo metodo crea un DefaultTableModel ed inizializza la tabella che lo conterrà.
                                                                           Ha i seguenti argomenti:
                                                                           data: Questo è un oggetto da aggiungere ad una tabella.
                                                                           col: Questo è un oggetto colonna da aggiungere alla tabella.
     JTable table = new JTable(model);
     // insertRow(int row_index, Object data[])                           Questo metodo è usato per inserire una riga in una specifica posizione.
                                                                           Ha bisogno dei seguenti parametri:
                                                                           row_index: Questo è l'indice della riga da aggiungere.
                                                                           data: Questo è il dato che deve essere aggiunto alla tabella.
     model.insertRow(0,new Object[]{"Ranjan","50"});                       Insert first position
```

```
     model.insertRow(3,new
Object[]{"Amar","600"});

     model.insertRow(table.getRowCount(),new
Object[]{"Sushil","600"});
     panel.add(table);
     frame.add(panel);
     frame.setSize(300,300);
     frame.setVisible(true);

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLO
SE);
      }
}
```

Insert 4 position

Insert last position

Output:



2. Questo programma crea una tabella che contiene 3 righe e 2 colonne.

```
package my_Jtable2;
import javax.swing.table.DefaultTableModel;
public class JTable2 extends javax.swing.JFrame {
  public static void main(String[] args) {
    :
  }
  private void
btnFineActionPerformed(java.awt.event.ActionEvent
evt) {
       System.exit(0);
  }
  private void visualizza(){
    String intestazione []= {"Name","code"};
    DefaultTableModel elenco = new
DefaultTableModel();
    elenco.setColumnIdentifiers(intestazione);

    elenco.addRow(new Object[]{"Ranjan","50"});
    elenco.addRow(new Object[]{"Vinod","100"});
    elenco.addRow(new Object[]{"Raju","200"});
    elenco.addRow(new Object[]{"Ranju","300"});
    elenco.addRow(new Object[]{"Ranjan","50"});
    elenco.addRow(new Object[]{"Amar","600"});
    elenco.addRow(new Object[]{"Sushil","600"});
    elenco.addRow(new Object[]{"Sushil","600"});
    tblElenco.setModel(elenco);

  }
  public JTable2() {
     initComponents();

     visualizza();

  }
```

Creazione intestazione
Creazione della **DefaultTableModel**

Inserimento intestazione nella **DefaultTableModel**
Inserimento dati nella **DefaultTableModel**

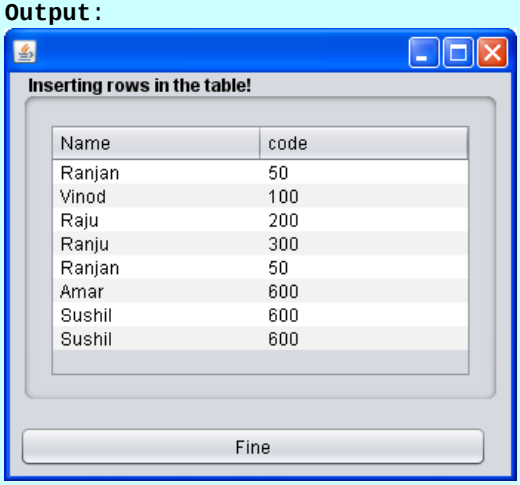
Inserimento del **DefaultTableModel** nella **JTable**

Il costruttore richiama:

- il **metodo initComponents()**. Questo metodo viene generato automaticamente da NetBeans e contiene tutto il codice dell'interfaccia grafica, ascoltatori compresi, costruita nell'area disegno
- il **metodo per visualizzare** i dati nella tabella

```
  private void initComponents() {
    :
  }
}
```

Output:



3. Visualizzare, in una **tabella**, l'orario dei pullman contenuto in un file: orario di arrivo, orario di partenza e destinazione.

```
import javax.swing.table.DefaultTableModel;
:
private void
pnlElencoComponentShown(java.awt.event.ComponentEvent evt)
{
   String intestazione[]= {"OrarioP","Destinazione",
"OrarioA"};                                          costruzione riga di
                                                     intestazione della Jtable
   DefaultTableModel elenco1 = new DefaultTableModel();

   elenco1.setColumnIdentifiers(intestazione);

   try
   {
      FileReader file = new FileReader("pullman.txt");

      BufferedReader input = new BufferedReader(file);

      while ((riga = input.readLine())!=null){

                                                     Il record è formato dai
                                                     campi immessi da tastiera
                                                     separati da ";".
                                                     La riga si conclude con
                                                     \r\n
                                                     readline() legge l'intera
                                                     riga senza il carattere
                                                     "a capo" che nel nostro
                                                     caso è \r\n
        String campi[]= riga.split(";");             la riga viene suddivisa
                                                     ("split") nei campi che
                                                     la compongono

        elenco1.addRow(new Object[]{campo[0], campo[1],
campo[2]});
   }
        input.close();
        tblElenco.setModel(elenco1);                 Visualizzazione dati
                                                     nella tabella tblElenco
   }                                                 Visualizzazione messaggio
   catch(EOFException e1){                           di EOF nella label lblER2
        lblER2.setText("Lettura file completata");
   }
   catch(IOException e2){                            Visualizzazione messaggio
        lblER2.setText("File errato");               di errore di IO nella
   }                                                 label lblER2
```

```
}                                                    fine "pnlElenco..."
```

Multiple Document Interface

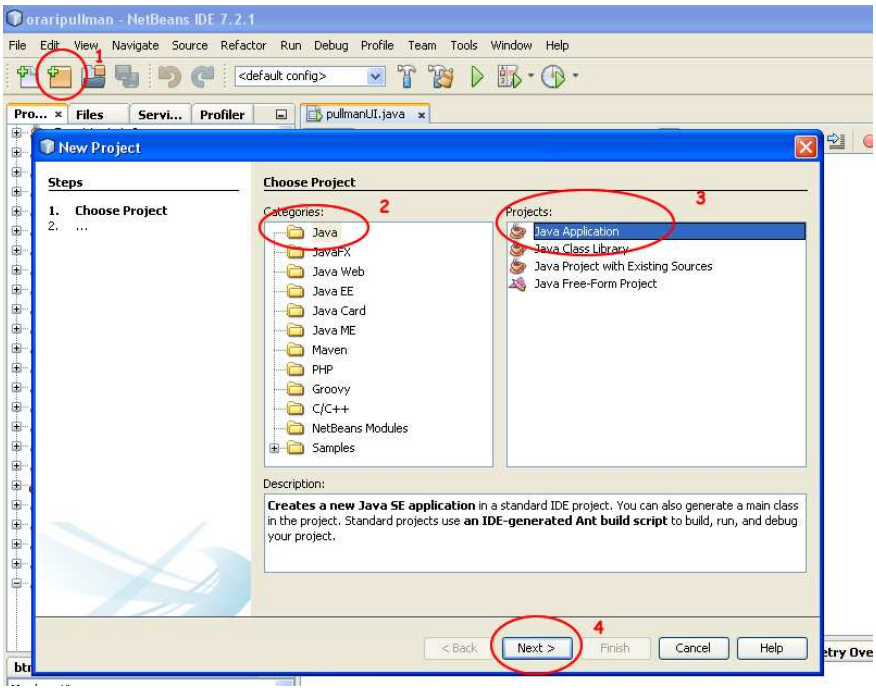
Un form è una finestra "singola", cioè può contenere i controlli della Casella degli strumenti, ma **non può contenere altri form**.

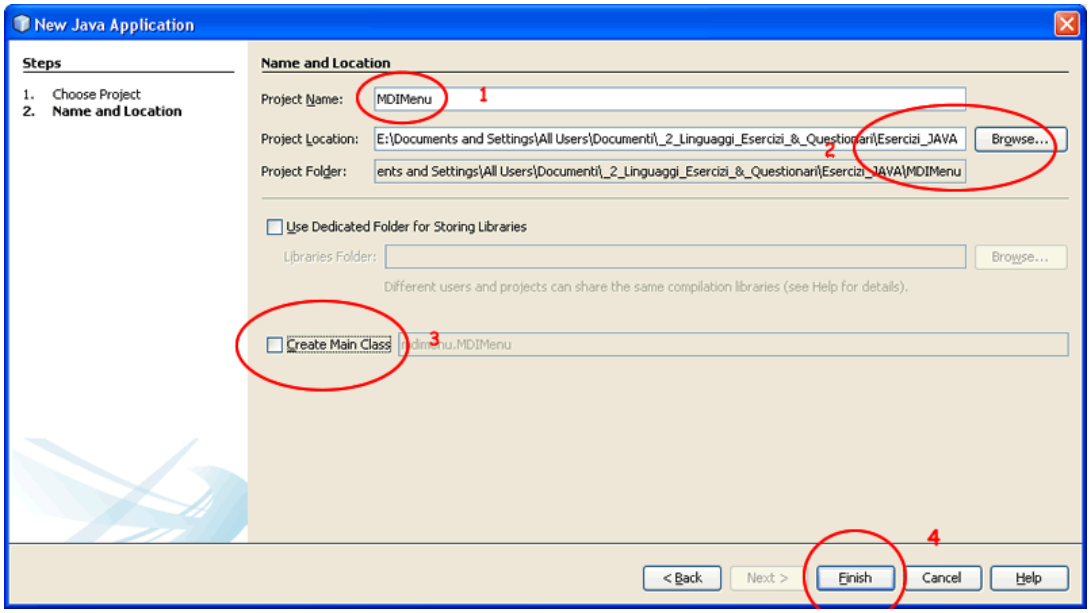
Un form MDI (Multiple Document Interface), invece, è una finestra utilizzata come sfondo di un'applicazione con interfaccia a documenti multipli. Un form MDI è il contenitore dei form MDI secondari dell'applicazione.

Ad esempio, Microsoft Word è un'applicazione MDI, Blocco Note è un programma SDI (Single document interface, interfaccia a documenti singoli).

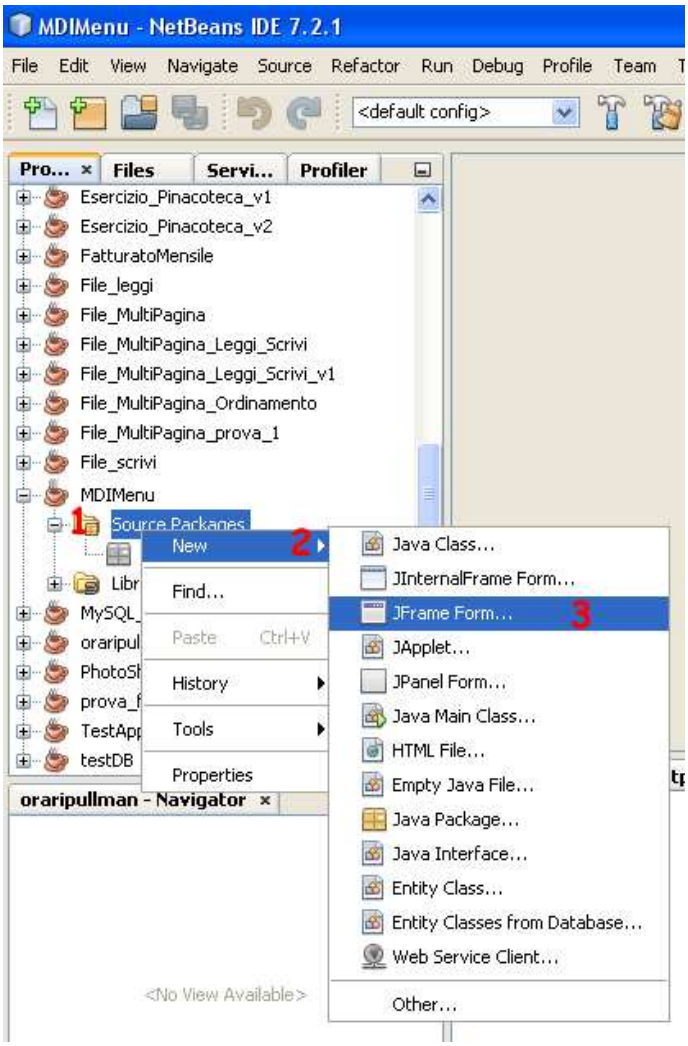
Vediamo ora come realizzare un form MDI.

1. Creare un nuovo progetto

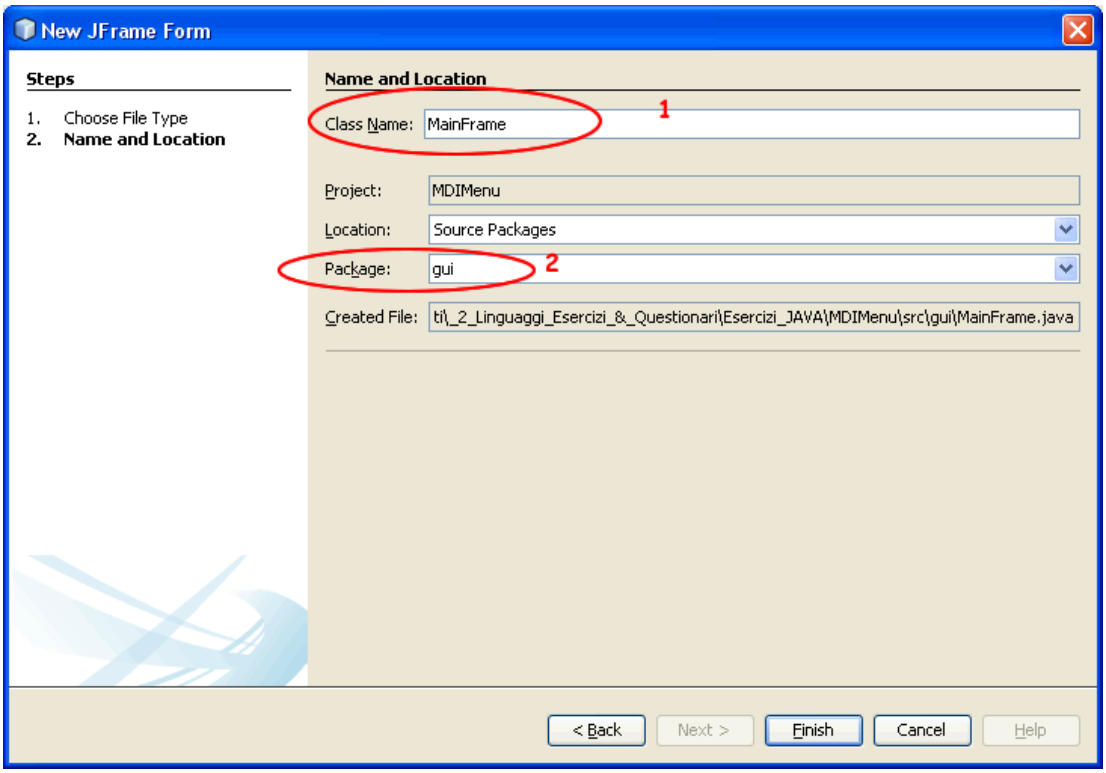




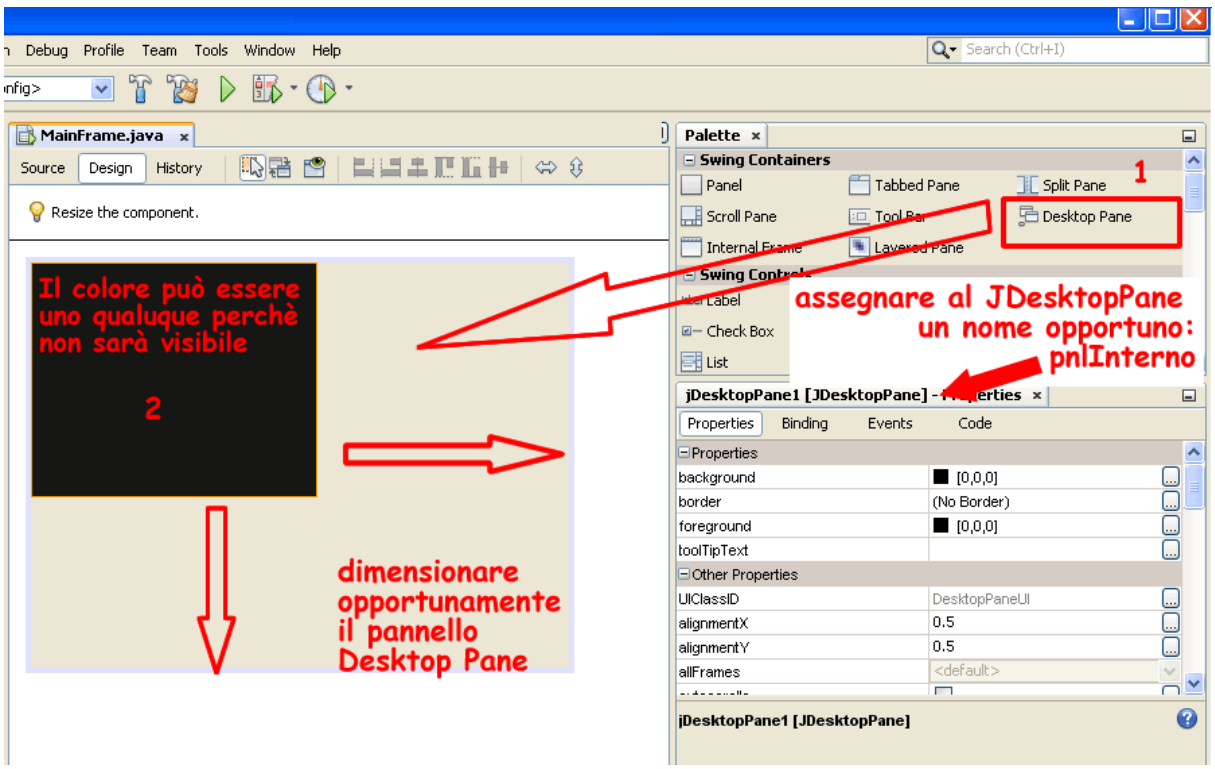
2. Inserire un JFrame Form



3. Assegnare un nome alla classe ed al package

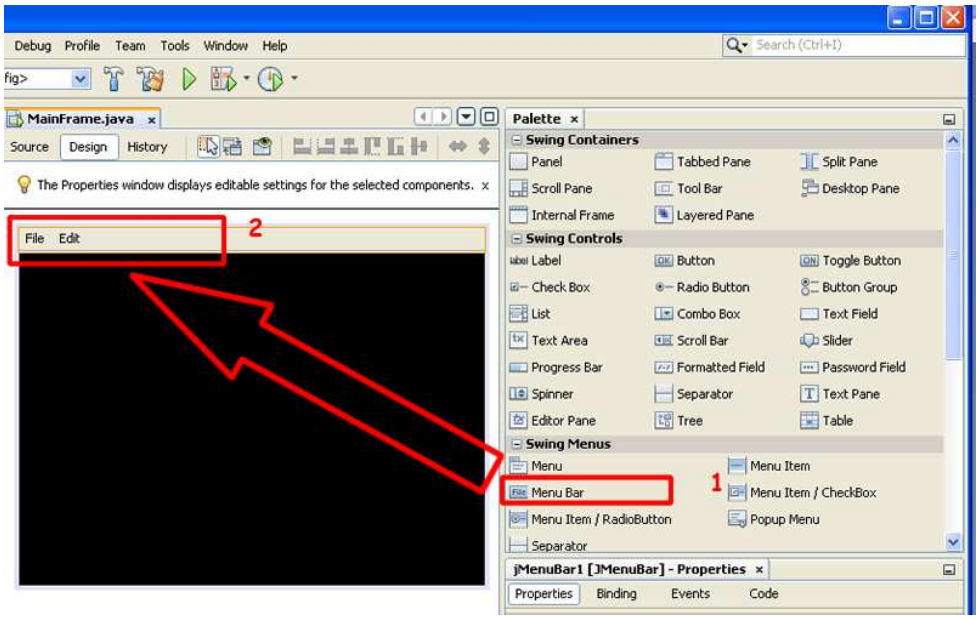


4. Inserire un Desktop Pane

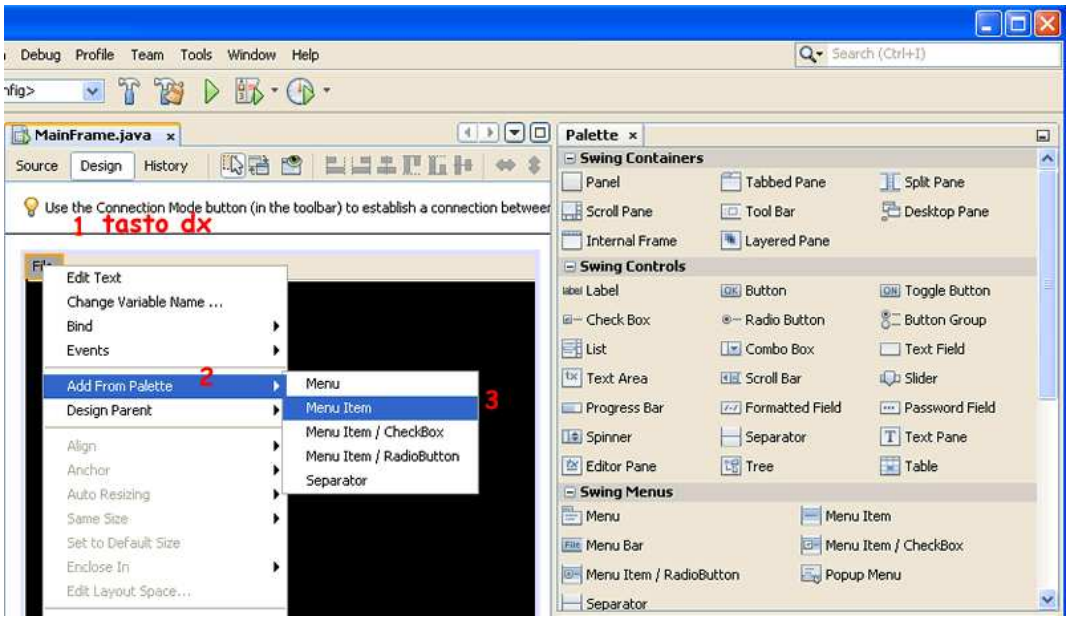


Un **JDesktopPane** è una sottoclasse della JLayerdPane con cui è possibile utilizzare adeguatamente i JInternalFrames.

5. Inserire una Menu Bar

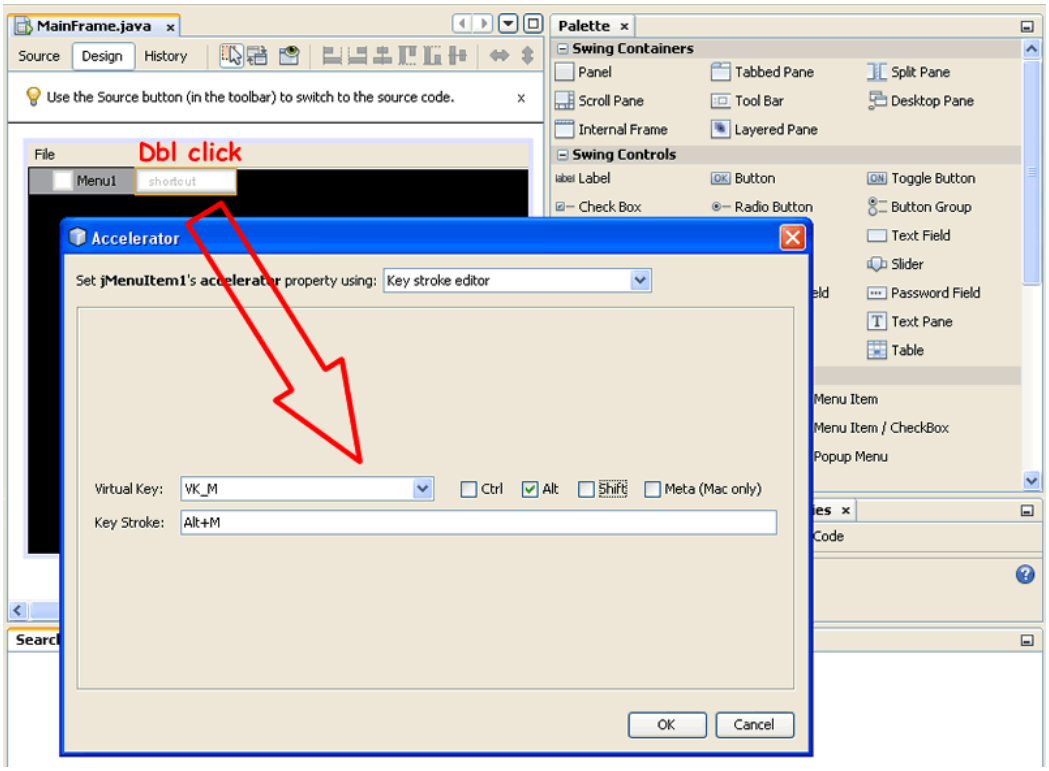


6. Aggiungere alla Menu Bar uno o più Menu Item

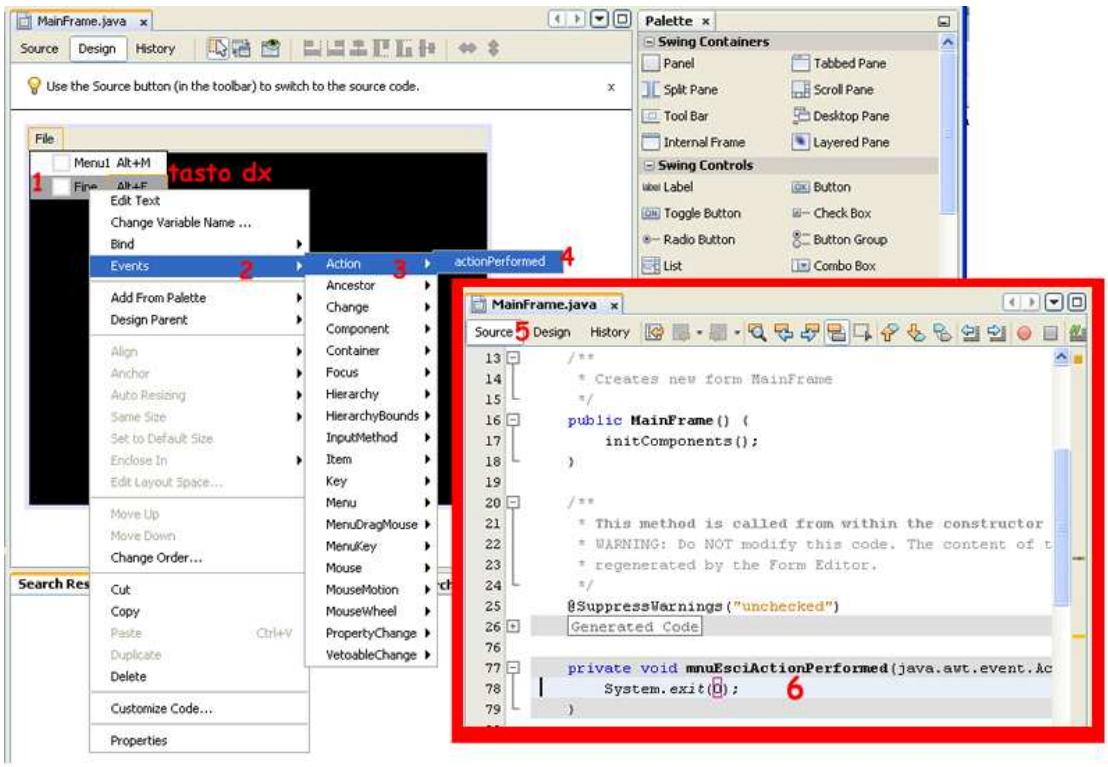


7. Aggiungere al Menu Item l'opzione di scelta rapida

Per assegnare un nome ed un'etichetta operare con tasto destro e **Change Variable name** ed **Edit Text**, rispettivamente.



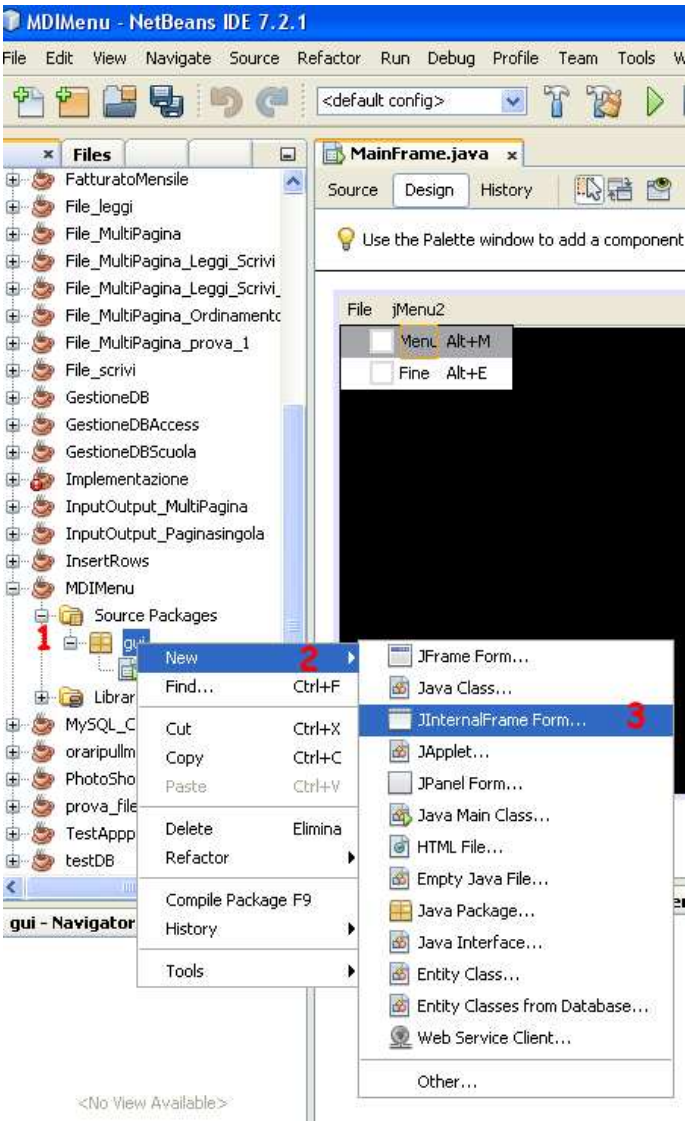
8. Associare un evento ad ogni Menu Item



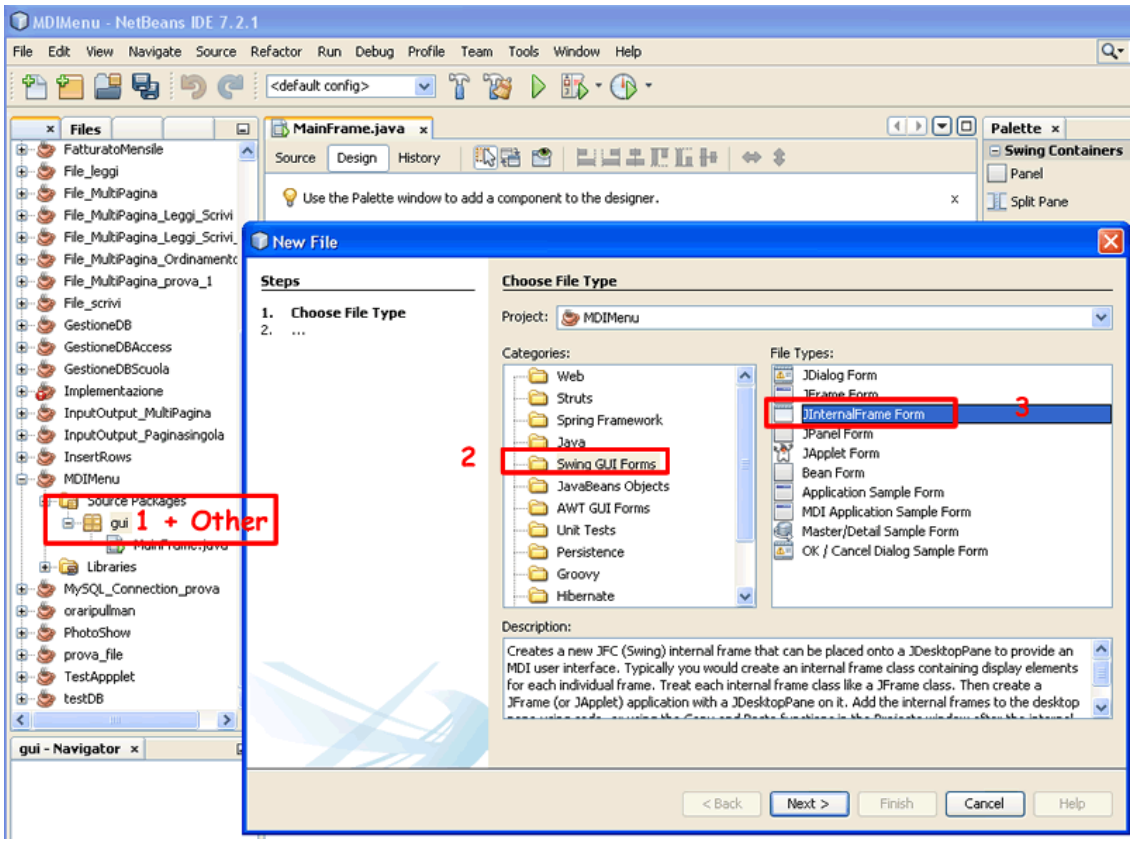
Nella Menu Bar, usando **tasto destro "Add Menu"** è possibile aggiungere altre opzioni ed a questa è necessario aggiungere almeno un Menu Item. Ripetere i punti: 6, 7, 8.

9. Inserire un nuovo JInternalFrame Form (vedi punto 2)

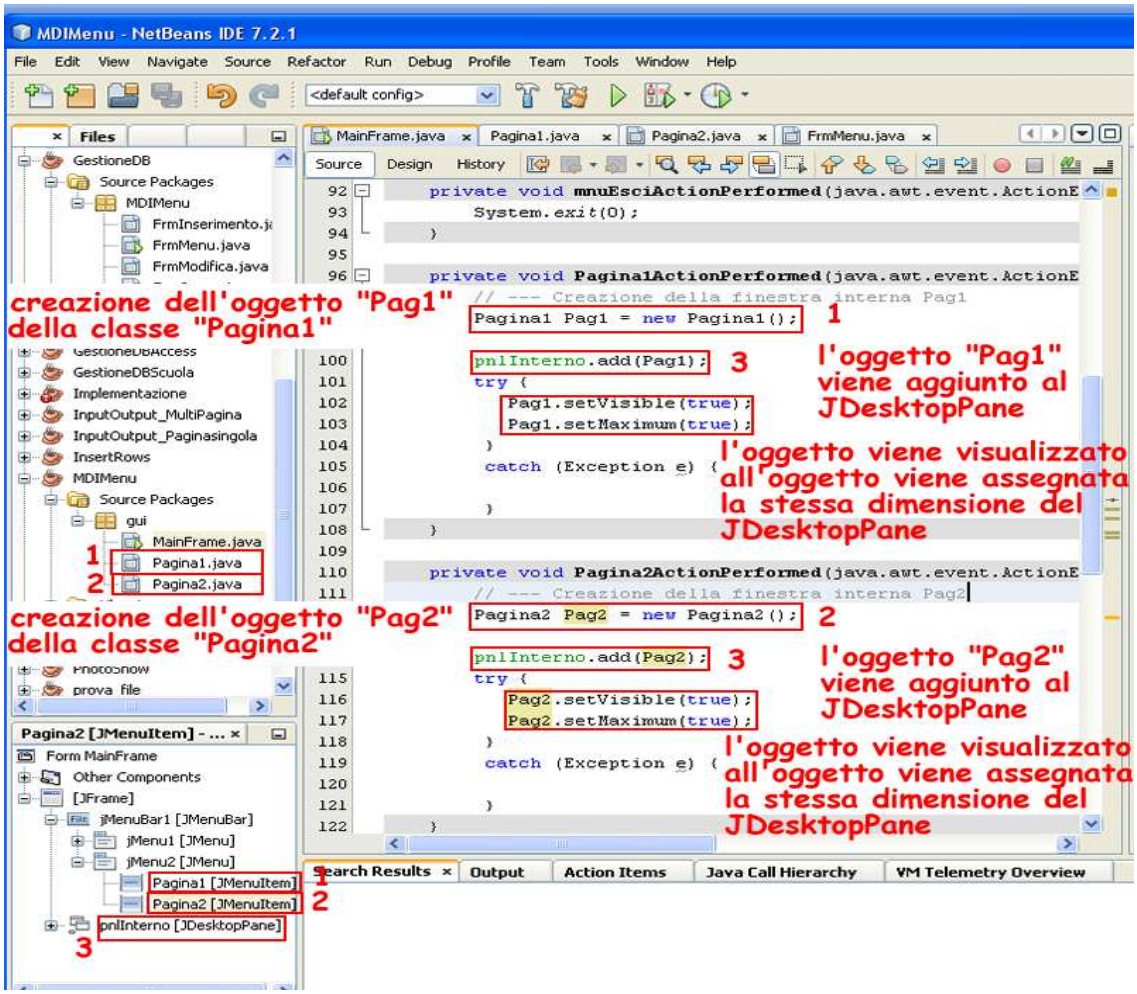
Il nuovo **JInternalFrame Form** dovrà essere visualizzabile all'interno del JFrame Form inserito in precedenza e dotato di DesktopPane e Menu Bar.



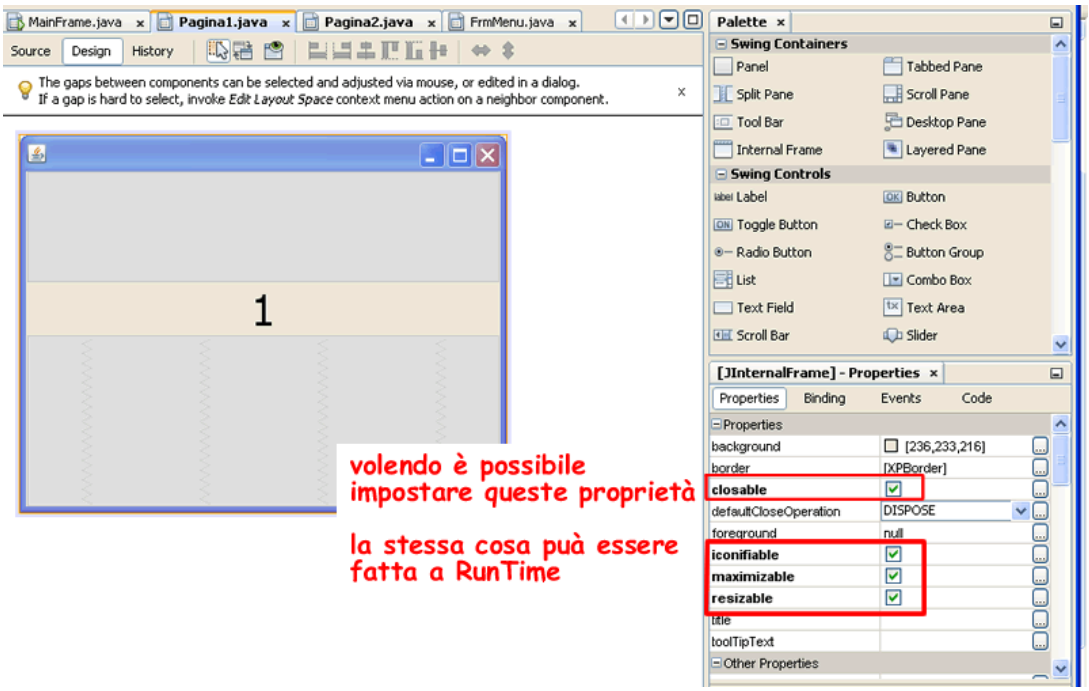
Equivalente al metodo utilizzato in ***figura***: **tasto destro > New > JInternalFrame Form** è la scelta: **tasto destro > New > Other > Swimg GUI Forms > JInternalFrame Form**



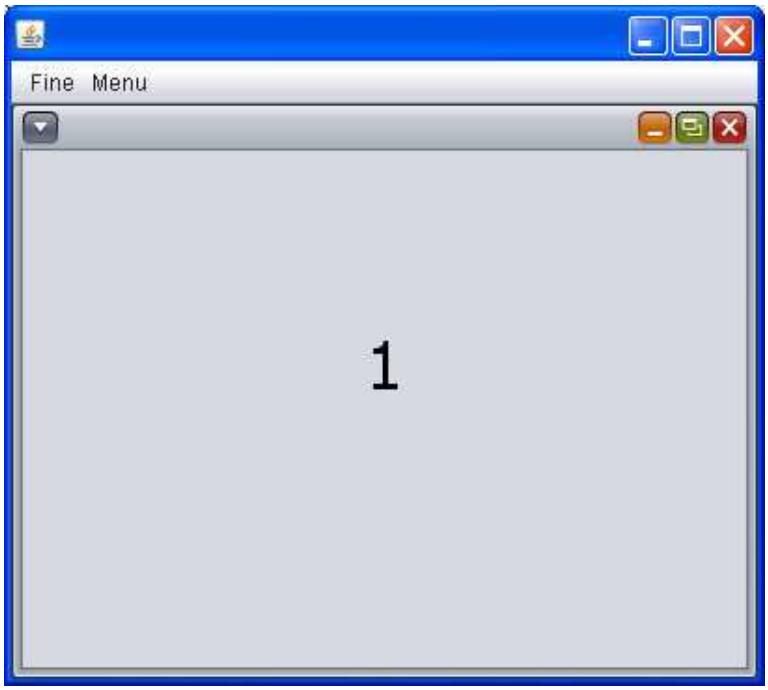
10. Scrivere il codice per visualizzare le "pagine interne"



11. Alcune proprietà che possono essere utili...

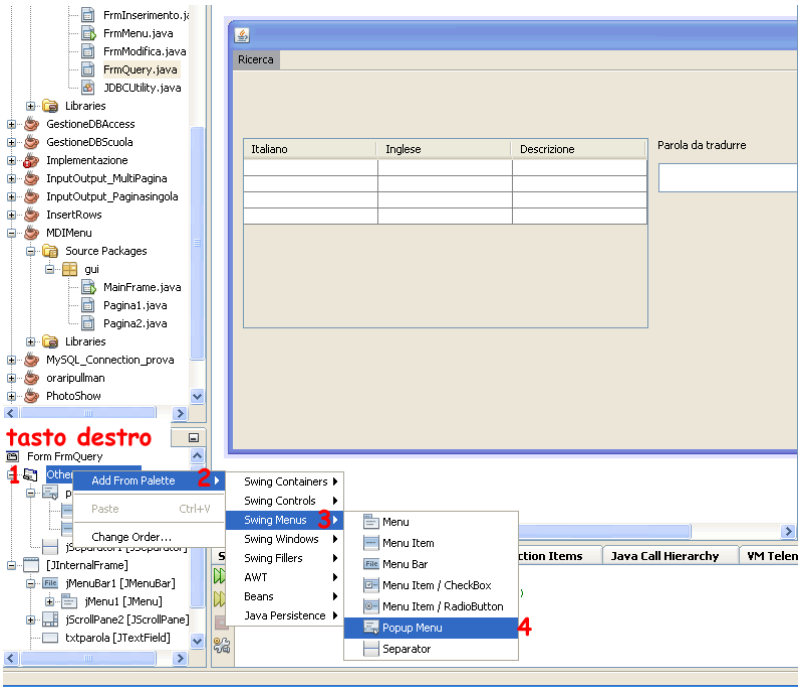


12. Output

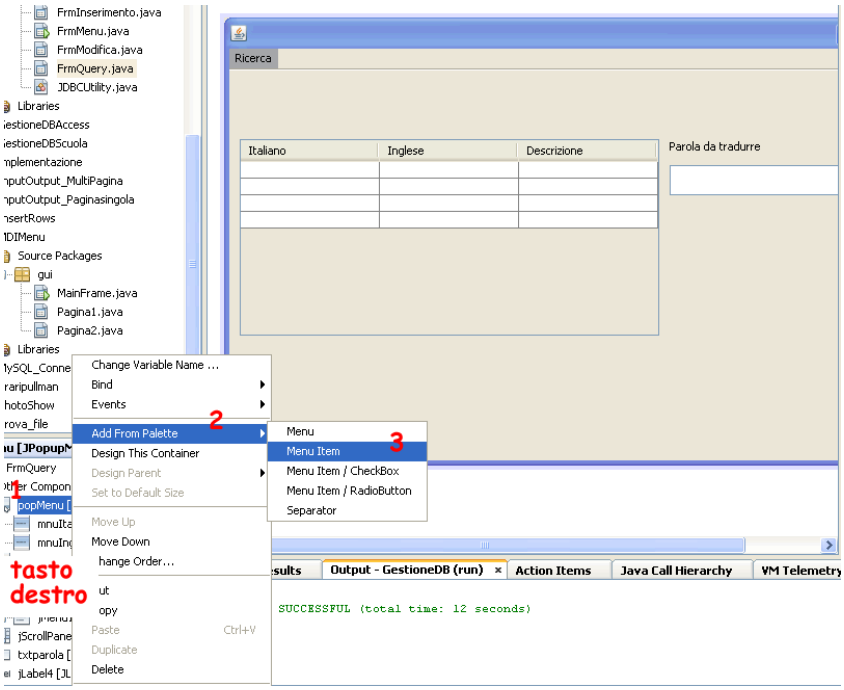


Usare il JpopupMenu

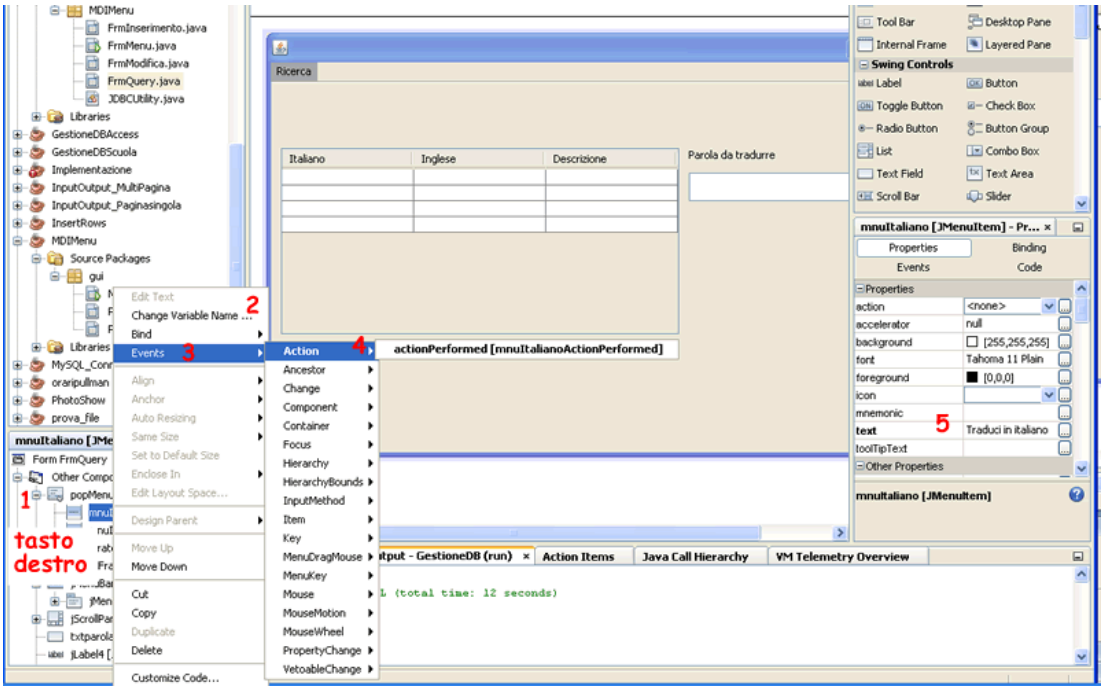
1. Inserire un Menu PopUp



2. Aggiungere al Menu PopUp uno o più Menu Item

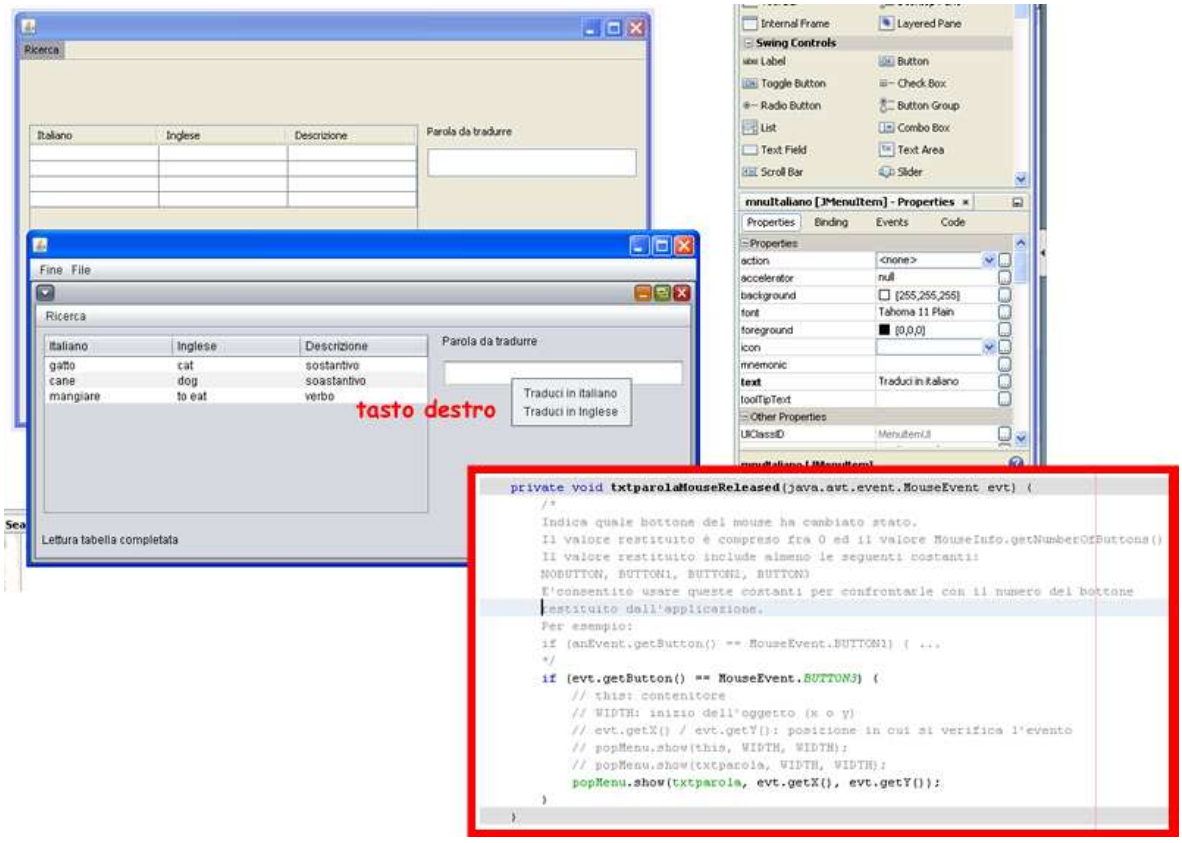


3. Associare un nome, una descrizione ed un evento ad ogni Menu Item



Dopo aver aggiunto uno o più Menu Item (1 - Tasto destro sul Menu Item) a ciascuno è necessario: assegnare un nome (2 - Change variable Name...), associare un evento (3 - Events, 4 - Action) ed una descrizione (5 - Text).

4. Scrivere il codice per usare il "Menu PopUp"



Codice JAVA	Descrizione
private void **txtparolaMouseReleased**(java.awt.event.MouseEvent **evt**) {	evt: parametro del metodo L'evento MouseReleased sull'oggetto txtparola indica quale bottone del mouse ha cambiato stato dell'oggetto. Il **valore** restituito, **evt.getButton()**, è compreso fra [0, MouseInfo.getNumberOfButtons()] Il valore restituito include almeno le seguenti **costanti**: **NOBUTTON**, **BUTTON1**, **BUTTON2**, **BUTTON3** E'consentito usare queste costanti per confrontarle con il numero del bottone restituito dall'applicazione.
if (evt.getButton() == MouseEvent.BUTTON3) {	Controllo se il pulsante del mouse che ha cambiato stato è il tasto destro (**BUTTON3**)

popMenu.show(**txtparola**, evt.**getX**(), evt.**getY**());	Se l'evento si è verificato come previsto allora si può **mostrare il menù popup** seguendo **una delle modalità**:
	• **oggetto** cui viene associato il menù:
	  o this: contenitore
	  o **txtparola**: oggetto prescelto in questo esempio
	• **dimensione del menù**: WIDTH: inizio dell'oggetto (x o y)
	posizione in cui si verifica l'evento: evt.getX() / evt.getY()

}
}
	Esempio:
	popMenu.show(this, WIDTH, WIDTH);
	popMenu.show(txtparola, WIDTH, WIDTH);

Gestire i database

Installare XAMPP

Il primo passo da affrontare per utilizzare un database è l'installazione dell'ambiente di sviluppo.
Per semplificare questa operazione conviene scaricare un pacchetto software chiamato **XAMPP** (o WAMPP) che comprende:

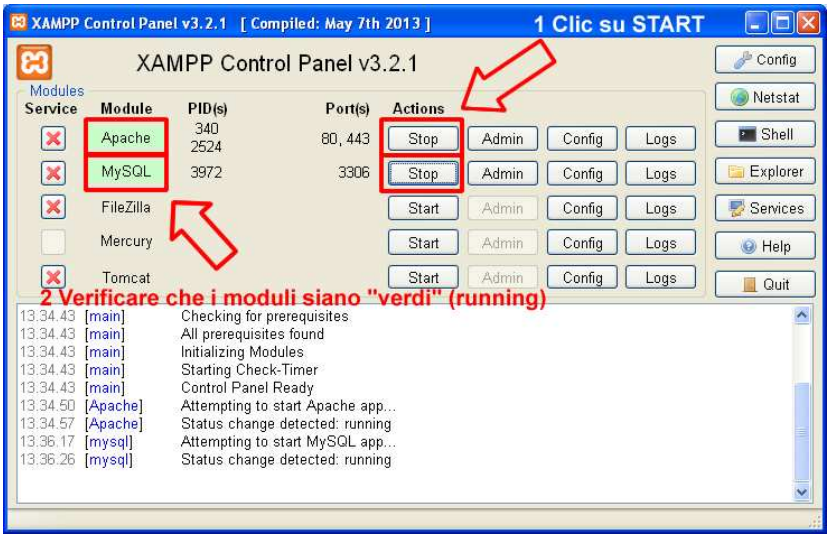
- il web server Apache, MySQL,
- l'interprete PHP,
- il software phpMyAdmin per la gestione dei database,
- Filezilla FTP Server
- Tomcat.

La nostra attenzione sarà posta esclusivamente su **MySQL e phpMyAdmin**, anche se tutti i componenti della suite sono particolarmente interessanti.
Se al termine dell'installazione avremo mantenuto i **settaggi di default**

- la **username** per l'accesso ai database sarà **root**,
- mentre **non** sarà **presente** alcun valore per la **password**.

Usare phpMyAdmin XAMPP

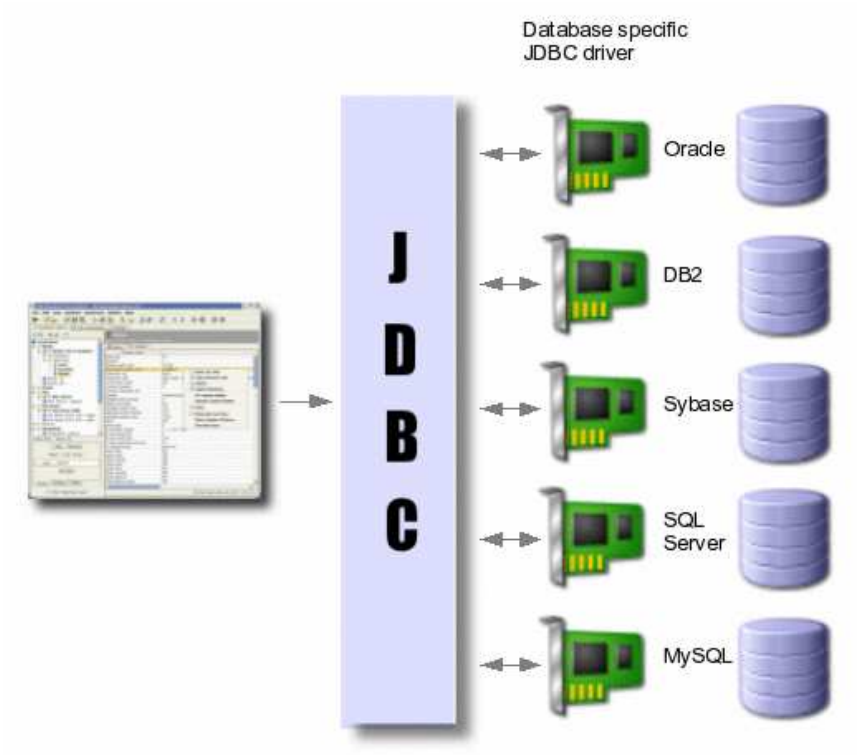
Dopo aver riscontrato come tutto sia regolarmente funzionante, occupiamoci della gestione dei database con phpMyAdmin. PhpMyAdmin si attiva immediatamnente dopo aver "lanciato" XAMPP ed aver attivato "Apache" e "MySQL":



Tecnologia Java per la gestione dei database

Java DB

Java DB è la distribuzione supportata da Oracle del database open source Apache Derby. La sua **facilità d'uso**, gli standard di conformità, serie completa di funzioni, e **ingombro ridotto** rendono il database ideale per gli sviluppatori Java.
Java DB è scritto nel linguaggio di programmazione **Java**, fornendo una portabilità **"write once, run anywhere"**. Può essere incorporato in applicazioni Java, senza bisogno di amministrazione da parte dallo sviluppatore o dell'utente. Può essere utilizzato anche in **modalità client-server**. Il DB Java è completamente transazionale e fornisce un'interfaccia SQL standard così come un driver compatibile JDBC 4.1.



Java Data Objects (JDO)

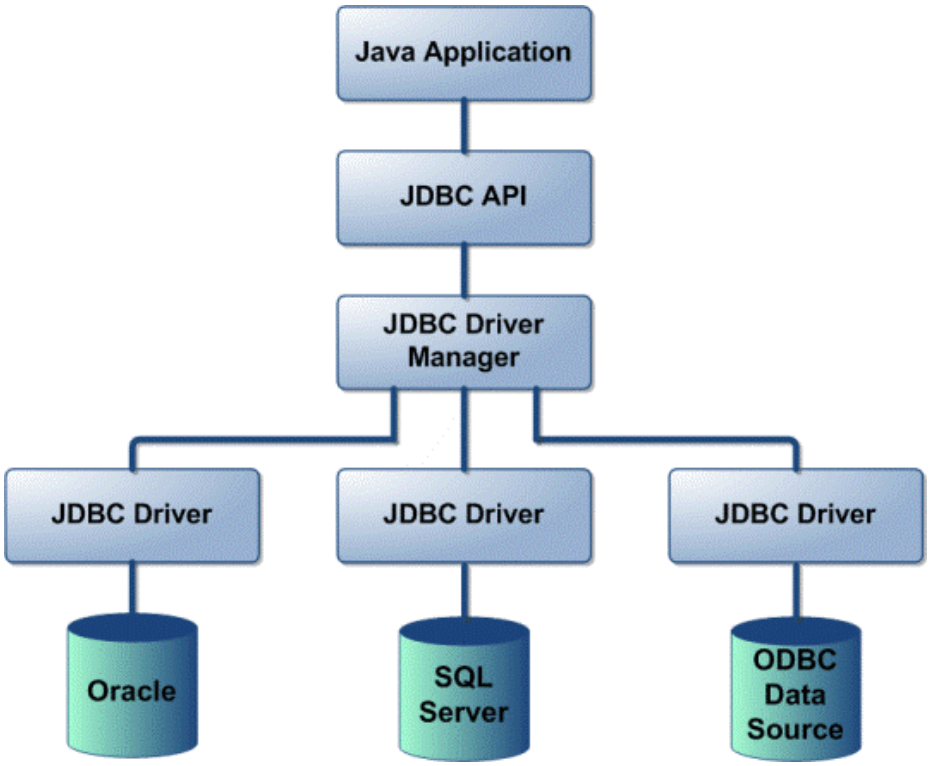
L'API (Application programming interface) Data Objects (JDO[22]) è un'interfaccia basata sul

[22] *JDO è stato sviluppato in Java Specification Request 12 (JSR 12) e Java Specification Request 243 (JSR 243). Cominciando con JDO 2.0, lo sviluppo delle API e il Compatibility Kit Technology (TCK) si svolge*

modello di astrazione persistente standard di Java. I programmatori di applicazioni possono utilizzare la tecnologia JDO per memorizzare direttamente le istanze del modello di dominio Java nella memoria permanente (database). I vantaggi sono la **facilità di programmazione**, la portabilità delle applicazioni, **l'indipendenza del database**, ad alte prestazioni, e l'integrazione opzionale con Enterprise JavaBeans (EJB).

Il Java Database Connectivity (JDBC)

L'API Java Database Connectivity (**JDBC**) è lo standard industriale per la connettività di database indipendente dal linguaggio di programmazione Java ed una vasta gamma di database - database SQL e altre fonti di dati tabulari, come fogli di calcolo o file flat. **L'API JDBC fornisce una API a livello di chiamata per l'accesso al database basato su SQL**. La tecnologia JDBC consente di utilizzare il linguaggio di programmazione Java e di sfruttare la capacità "write once, run anywhere" per le applicazioni che richiedono l'accesso ai dati aziendali. Con un driver abilitato alla tecnologia JDBC, è possibile collegare tutti i dati aziendali, anche in un ambiente eterogeneo.



Accedere ai database MySQL da Java

Perché usare MySQL

Il database MySQL® è diventato il database open source più conosciuto al mondo grazie a prestazioni veloci, elevata affidabilità e facilità d'uso. È utilizzato in ogni continente, da singoli sviluppatori Web e da molte delle aziende più grandi e in più rapida espansione del mondo, tra cui leader di settore come Yahoo!, Google, Alcatel-Lucent, Nokia, YouTube e Zappos.com., per risparmiare tempo e denaro potenziando siti Web con volumi d'accesso elevatissimi, sistemi business-critical e pacchetti di soluzioni software.

MySQL non è soltanto il database open source più famoso al mondo, è diventato anche il database utilizzato per una nuova generazione di applicazioni basate sullo stack **LAMP**[23].

Come usare MySQL

Prima di utilizzare JDBC con MySQL dobbiamo **scaricare il driver** (il connector di Java) corrispondente dal sito http://www.mysql.com/products/connector.

Dal file .zip scaricato estraiamo il **.jar**. Il file scaricato rappresenta il connettore : **mysql-connector-java-5.1.25-bin.jar**.

---

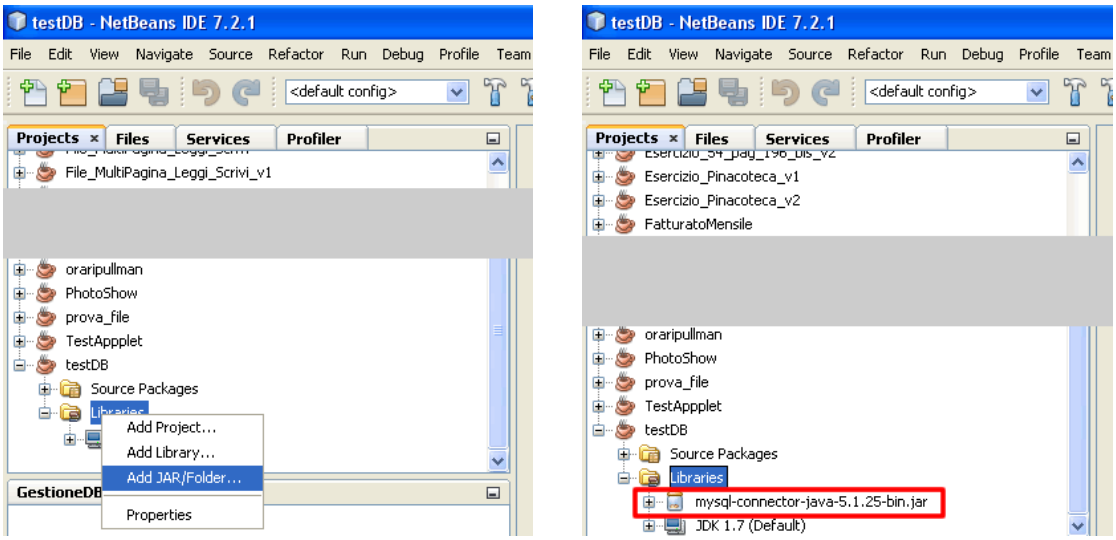
nell'ambito del progetto open-source Apache JDO.

[23] *LAMP: **L**inux, **A**pache, **M**ySQL, **P**HP / **P**erl / **P**ython*

Questo file **va salvato in una cartella ben riconoscibile** come ad esempio quella di NetBeans ovvero quella in cui vengono salvati i programmi Java.

Creazione del 1° progetto con JDBC

Creaiamo un nuovo progetto denominato **testDB**, come **Java Application**. Nella sezione **Projects di NetBeans**, espandiamo la cartella del progetto e clicchiamo con il tasto destro sulla cartella Library e scegliamo **Add JAR/Folder** per aggiungere **mysql-connector-java-5.1.25-bin.jar** prelevandolo dalla cartella in cui è stato precedentemente salvato.



MySQL connector

Il connector è ora aggiunto alla libreria del progetto.

Vediamo ora i passi da compiere e gli oggetti necessari per sviluppare il nostro programma:

1. Attivare il Driver

Prima di effettuare la connessione al database deve essere **attivato il driver di MySQL** attraverso l'istruzione:

```
static final String driver = "com.mysql.jdbc.Driver";
  :
Class.forName(driver); ;
```

Il **metodo Class.forName consente di attivare il driver passato come parametro**. A seconda del database da utilizzare, si inserirà un parametro differente e si specificheranno gli opportuni parametri di connessione, ma la parte rimanente del codice potrà restare praticamente invariata, dunque indipendente dal tipo di database associato.

2. Stabilire la connessione al database

L'**oggetto Connection** viene usato per la connessione ai database ed è in grado di fornire le informazioni che ne descrivono:

- le tabelle,
- la grammatica dal linguaggio SQL supportato,
- le procedure,
- le capacità di questo collegamento,

e così via.

Queste informazioni si ottengono con il **metodo getMetaData**.

L'oggetto Connection, per la gestione del processo di connessione al database, stabilirà la "connessione" ed avrà il compito di mantenere tutte le informazioni di accesso al database. All'interno di questo oggetto, infatti, viene assegnata l'effettiva connessione mediante l'uso del metodo statico **getConnection** della **classe DriverManager**.

3. Eseguire le istruzioni SQL

Per **l'esecuzione di un'istruzione SQL** statica viene utilizzato **un oggetto Statement** che restituisce i risultati che questa produce.

Come impostazione predefinita, per ogni oggetto **Statement** può essere aperto **solo un oggetto ResultSet**.

Pertanto, se la lettura di **un oggetto ResultSet è intercalata con la lettura di un altro**, ognuno deve essere stato generato da **diversi oggetti Statement**. Tutti i metodi di esecuzione nell'interfaccia Statement implicitamente chiudono oggetto ResultSet corrente di uno Statment, se ne esiste uno aperto.

Per poter eseguire le istruzioni SQL sul database si utilizza quindi la **classe Statement**, che prevede quattro **metodi principali per l'invio dei comandi**:

- **executeQuery()** : consente di eseguire istruzioni per la **lettura** dei dati (es: SELECT...)
- **executeUpdate()** : consente di **modificare** i dati del database (es: INSERT..., UPDATE..., o DELETE...) ovvero la sua struttura, per esempio, creando e modificando le tabelle (es: CREATE..., ALTER...)
- **execute()** : esegue istruzioni SQL che possono restituire più risultati.
- **executeBatch()** : presenta al database una serie di comandi da eseguire e se tutti i comandi vengono eseguiti con successo, restituisce un array di conteggi di aggiornamento.

4. Recuperare le informazioni da una query

Per recuperare le informazioni del database, oltre a definire la query di interrogazione, abbiamo bisogno di un nuovo oggetto chiamato **ResultSet**. L'oggetto ResultSet è un contenitore da cui è possibile acquisire i dati risultanti dalla query. L'oggetto può essere letto in maniera sequenziale, scorrendo le righe che sono state restituite dall'interrogazione e prelevando uno specifico dato mediante l'indicazionedel nome del campo corrispondente o della posizione all'interno della riga di output.

Codice JAVA	Descrizione
`package my_testDB;`	
`import com.mysql.jdbc.Connection;` `import com.mysql.jdbc.Statement;`	

`import java.sql.DriverManager;`
`import java.sql.ResultSet;`
`import javax.swing.table.DefaultTableModel;`	

`public class TestDBUI extends javax.swing.JFrame {`	
`  Connection connessione = null;` `  Statement statement = null;`	
`  static final String driver   = "com.mysql.jdbc.Driver";`	Individuare il DRIVER
`  static final String dbname   = "vocabolario";`	Indicare il DATABASE
`  static final String userName = "root";`	Indicare USERNAME
`  static final String password = "";`	Indicare la PASSWORD
`  static final String url  = "jdbc:mysql://localhost:3306/";`	Individuare i parametri di connessione

`  public boolean connessione(){`
`    boolean connesso = false;`
`    try {`	

`        Class.forName(driver);`	Attivare il Driver
`        connessione = (Connection) DriverManager.getConnection(url + dbname, userName, password);`	Stabilire la Connessione
`        lblMsg1.setText("Database connesso");`	

connesso = true;
optMessaggio.showMessageDialog(null, "Connessione riuscita");
}
catch (Exception e){
connesso = false;
optMessaggio.showMessageDialog(null, "Connessione NON riuscita");
}
return connesso;
}	

public boolean **disconnessione**() {
boolean disconnesso = false;
try{
connessione.close();
disconnesso = true;
optMessaggio.showMessageDialog(null, "Disconnessione riuscita ");
}
catch (Exception e) {
disconnesso = false;
optMessaggio.showMessageDialog(null, "Disconnessione NON riuscita ");
}
return disconnesso;
}	

private void cancella(){
txtUserId.setText("");
txtPassword.setText("");
lblMsg1.setText("");
}
private void cancella2(){
txtUserId2.setText("");
txtPassword2.setText("");
lblMsg2.setText("");
}	

private void **visualizza**(){
try{
String intestazione []= {"ID", "User ID","Password"};
DefaultTableModel elenco = new DefaultTableModel();
elenco.setColumnIdentifiers(intestazione);
	Visualizzazione di tutti i dati della tabella utenti nella **JTable tblElenco**

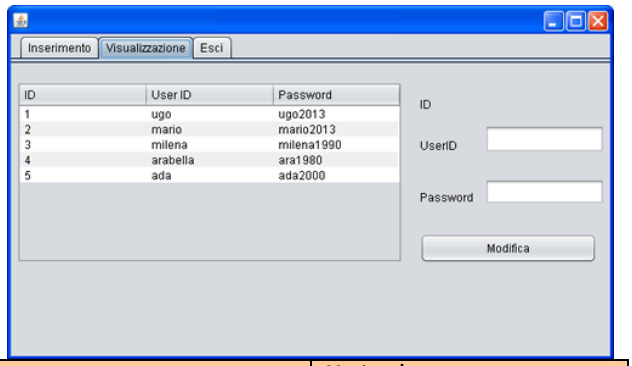
statement = (Statement) connessione.createStatement();	Recuperare le informazioni del database e definire la query di interrogazione
String query = "select * from utenti";	Query di interrogazione
ResultSet risultato = statement.**executeQuery**(query);	Recuperare le informazioni da una **query di interrogazione** eseguita

while(risultato.next())
{
elenco.addRow(new Object[]{ risultato.getString("id"), risultato.getString("userid"), risultato.getString("password")});
}
tblElenco.setModel(elenco);	

}
catch(Exception e){
lblMsg2.setText("Lettura tabella completata");	

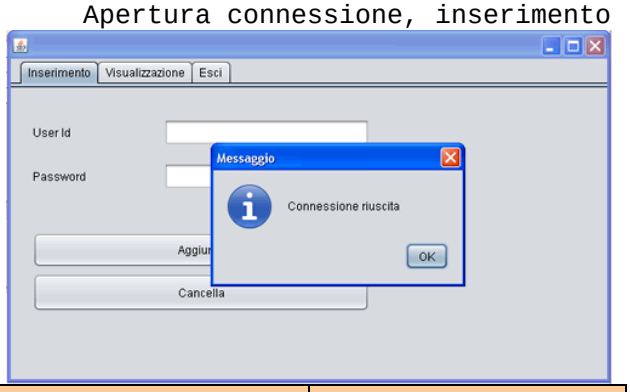
} }	

Output:
Visualizzazione contenuto della tabella



public **TestDBUI**() { initComponents();	Metodo costruttore[24]
this.setLocationRelativeTo(null);	**Posizionare finestra** al centro dello schermo
if (**!connessione()**) System.exit(0); }	**Apertura della connessione.** Chiusura del programma per esito negativo

Output:
Apertura connessione, inserimento



private void **btnAggiungiActionPerformed**(java.awt.event.ActionEvent evt) { try {	**Inserimento**
String sql = "Insert into utenti (userid, password) values ('" + txtUserId.getText()+ "', '" + txtPassword.getText()+ "')";	Query di comando
statement.**executeUpdate**(sql);	Recuperare le informazioni da una **query di comando** eseguita

lblMsg1.setText("Inserimento riuscito");
cancella();
}
catch (Exception e){
lblMsg1.setText("Inserimento NON riuscito " + e.toString());
}	

[24] ***Metodo costruttore***. *Un costruttore di una classe viene definito con lo stesso nome della classe, è una caratteristica che lo differenzia da un normale metodo è il fatto di non avere un tipo di ritorno.*
Il metodo costruttore viene chiamo, solo ed esclusivamente, quando viene istanziata la classe, quindi si accede al costruttore una sola volta per ogni istanza della classe.

}
private void **pnlVisualizzazioneComponentShown**(java.awt.event.ComponentEvent evt) {
visualizza();
}	

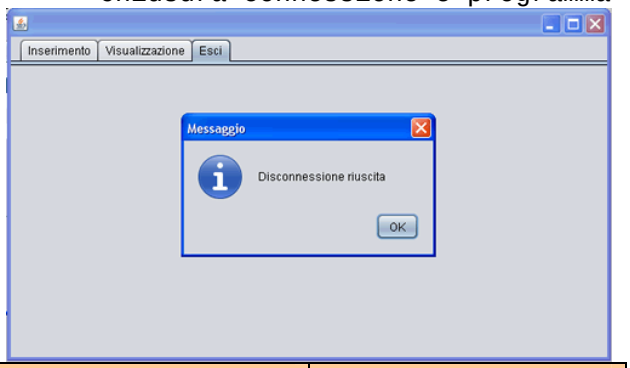
private void **btnModificaActionPerformed**(java.awt.event.ActionEvent evt) { try {	**Modifica**
String sql = "**Update** utenti " + " set userid = '" + txtUserId2.getText()+ "', " + " password = '" + txtPassword2.getText()+ "'" + " where id = " + Integer.parseInt(lblID.getText());	**Query di comando**
statement.**executeUpdate**(sql);	Recuperare le informazioni da una **query di comando** che viene eseguita

lblMsg2.setText("Modifica riuscita");
cancella2();
visualizza();
}
catch (Exception e){
lblMsg2.setText("Modifica NON riuscita " + e.toString());
}	

private void **pnlEsciComponentShown**(java.awt.event.ComponentEvent evt) {
disconnessione();
System.exit(0);
}
	Chiusura della connessione e del programma

Output:

Chiusura connessione e programma



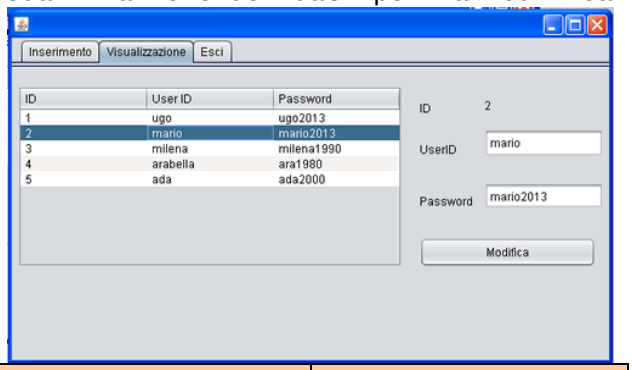
private void btnCancellaActionPerformed(java.awt.event.ActionEvent evt) {
cancella();
}	

private void **tblElencoMouseClicked**(java.awt.event.MouseEvent evt) { Object dato = null;	**Scelta di una riga** della tabella
int row = tblElenco.getSelectedRow(); int col = tblElenco.getSelectedColumn();	**Posizione** della cella selezionata

dato = (Object)tblElenco.**getValueAt**(row,0);
lblID.setText(dato.toString());
dato = (Object)tblElenco.**getValueAt**(row,1);
txtUserId2.setText(dato.toString());
dato = (Object)tblElenco.**getValueAt**(row,2);
txtPassword2.setText(dato.toString());
}
	Cella selezionata: **dato** = (Object)tblElenco .getValueAt(row, col);
	Contenuto di una cella: **getValueAt**

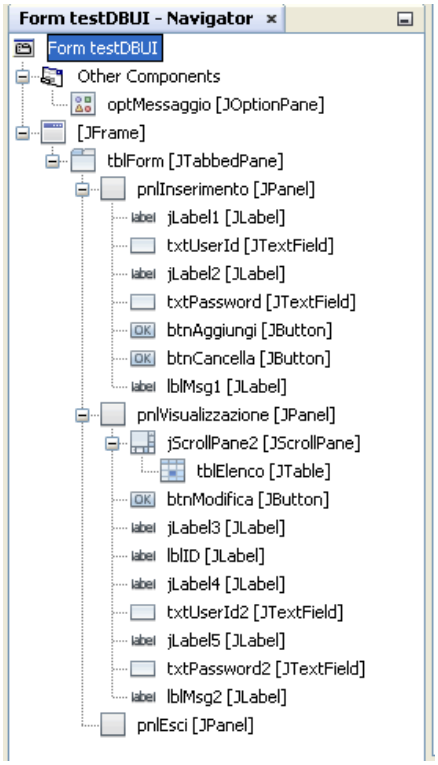
Output:

Selezione di una riga e visualizzazione dei dati per la modifica




```
  public static void main(String args[]) {     :
     :
     :
  }
}
```

Oggetti utilizzati nel progetto:



Query di comando

```
Le stringhe "sql":
String sql = "Insert into utenti (userid, password) values ('" +
txtUserId.getText()+ "', '" + txtPassword.getText()+ "')";
```

```
String sql = "Update utenti  set  userid = '" + txtUserId2.getText()+ "',
password = '" + txtPassword2.getText()+ "'"
           + " where id = " + Integer.parseInt(lblID.getText());
```

contengono istruzioni da eseguire e vengono passate come parametro al metodo **executeUpdate** di **statement**. Il **valore** restituito da tale comando corrisponde al **numero di righe inserite o modificate**. Se l'istruzione SQL eseguita fosse, ad esempio, una **CREATE** il **valore restituito sarebbe 0**. Il valore restituito, volendo, si può assegnare ad una variabile int:

```
statement.executeUpdate(sql);
int numero = statement.executeUpdate(sql);
```

Query di interrogazione

La stringa "query":

```
String query = "select * from utenti";
```

viene quindi **eseguita** grazie all'oggetto ResultSet:

```
ResultSet risultato = statement.executeQuery(query);
```

l'oggetto può essere letto in maniera **sequenziale**, scorrendo le righe che sono state restituite dall'interrogazione:

```
while(risultato.next())
{
     elenco.addRow(new Object[]{ risultato.getString("id"),
risultato.getString("userid"), risultato.getString("password")});
}
ovvero
while(risultato.next())
{
     elenco.addRow(new Object[]{ risultato.getString(0), risultato.getString(1),
risultato.getString(2)});
}
```

Accedere ai database MS-Access da Java

Introduzione

In questa sezione vedremo come utilizzare il bridge JDBC ODBC per accedere ad un **database MS-Access** da applicazioni Java.

Driver ODBC

In Java, abbiamo bisogno di un driver da caricare in fase di esecuzione della connessione a qualsiasi sorgente di dati. Lo stesso vale anche per le origini dati ODBC (Open DataBase Connectivity). Il driver è implementato come una classe che viene rintracciata e caricata in fase di esecuzione. Il **driver ODBC** per le connessioni JDBC è chiamato **sun.java.odbc.JdbcOdbcDriver**.

Stringa di connessione ODBC

Per connetterci all'origine dati abbiamo bisogno di una **stringa di connessione ODBC**. Consideriamo, ad esempio, di scrivere un programma che si colleghi ad un database di Access denominato mydb.mdb presente nella directory dell'applicazione. Utilizzeremo una stringa di connessione ODBC come segue:

```
Driver={Microsoft Access Driver (*.mdb)};DBQ=myDB.mdb;
```

seguita da una specifica aggiuntiva che punterà al driver necessario per la connessione, cioè jdbc: odbc: seguito dalla stringa di connessione. Quindi la stringa di connessione completa sarà:

```
jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)}; DBQ=myDB.mdb;
```

Così, per generalizzare quanto sopra, per essere in grado di connettersi con un DSN (Database Source Name) ODBC, abbiamo bisogno di una **stringa di connessione del modulo**:

```
jdbc:odbc:<stringa DSN ODBC>
```

Importare le classi per stabilire la connessione al database

Il pacchetto contenente le relative classi di database è contenuta in java.sql. Quindi, facciamo l'importazione come segue:

```
import java.sql.*;
```

Caricare il driver JDBC ODBC

Caricare dinamicamente la classe sun.java.odbc.JdbcOdbcDriver come segue:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

La classe **sun.jdbc.odbc.JdbcOdbcDriver** non deve essere scaricata ed installata **perchè è già inclusa nel Sun Java SDK**, deve essere solo dichiarata come detto sopra.

Aprire il database MS-Access nell'applicazione

Per fare questo, usiamo il DSN ODBC come specificato di seguito:

```
String database = "jdbc:odbc:Driver={Microsoft Access Driver
(*.mdb)};DBQ=myDB.mdb;";
Connection connessione = DriverManager.getConnection(database, "", "");
```

Creare un oggetto Statement per eseguire una query SQL

La dichiarazione (Statement) deve essere creata per eseguire una query SQL sul database aperto. E' fatta con il seguente codice:

```
Statement dichiarazione = connessione.createStatement();
```

Ripulire dopo aver completato il lavoro

Per pulire dopo aver eseguito la query SQL, chiamiamo **dichiarazione.close ()** per smaltire l'oggetto Statement oggetto. Poi, prima della fine del nostro programma o dopo il punto in cui decidiamo che il database non è più richiesto, chiudiamo il database con una chiamata a connessione.close (). Il codice seguente fa la pulizia dopo che abbiamo finito:

```
dichiarazione.close(); // Close the statement
connessione.close();   // Close the database. Its no more required
```

Eseguire un'istruzione SQL con un oggetto Statement

Chiamare dichiarazione.execute ("istruzione SQL") per eseguire una query SQL. La query restituisce il numero di righe prodotte dalla query. Se l'ultima query deve restituire un ResultSet cosa che si verifica in genere con una query di tipo SELECT ... , quindi chiamano **dichiarazione.getResultSet ()**, che restituisce l'oggetto **ResultSet**. Il codice seguente mostra come utilizzare una query di selezionare e visualizzare il valore contenuto nelle prime due colonne della tabella.

```
String selTable = "SELECT * FROM NOMETABELLA";
dichiarazione.execute(selTable);
ResultSet risultato  = dichiarazione.getResultSet();
while((risultato!=null) && (risultato.next()))
{
   System.out.println(risultato.getString(1) + " : " + risultato.getString(2));
}
```

Creazione del 2° progetto con JDBC:ODBC

Codice JAVA	Descrizione
`package my_testDBAccess;`	
`import java.sql.Connection;` `import java.sql.Statement;`	

`import java.sql.DriverManager;`
`import java.sql.ResultSet;`
`import javax.swing.table.DefaultTableModel;`	

`public class TestDBAccessUI extends javax.swing.JFrame {`	
`  Connection connessione = null;` `  Statement statement = null;`	
`  static final String driver = "sun.jdbc.odbc.JdbcOdbcDriver";`	Caricamento Database MSAccess
`  static final String dbname = "vocabolario.mdb";`	Indicare il DATABASE
`  static final String url = "jdbc:odbc:DRIVER={Microsoft Access Driver (*.mdb)};DBQ="+ dbname +";";`	

`  public boolean connessione(){`
`    boolean connesso = false;`
`    try {`	

`        Class.forName(driver);`	Attivare il Driver
`        connessione = (Connection) DriverManager.getConnection(url + dbname, userName, password);`	Stabilire la Connessione

`        lblMsg1.setText("Database connesso");`
`        connesso = true;`
`        optMessaggio.showMessageDialog(null, "Connessione riuscita");`
`    }`
`    catch (Exception e){`
`        connesso = false;`
`        optMessaggio.showMessageDialog(null, "Connessione NON riuscita");`
`    }`
`    return connesso;`
`  }`	

`  public boolean disconnessione() {`
`     boolean disconnesso = false;`
`     try{`
`       connessione.close();`
`       disconnesso = true;`
`       optMessaggio.showMessageDialog(null, "Disconnessione riuscita ");`
`     }`
`     catch (Exception e) {`
`       disconnesso = false;`
`       optMessaggio.showMessageDialog(null, "Disconnessione NON riuscita ");`
`     }`
`      return disconnesso;`
` }`	

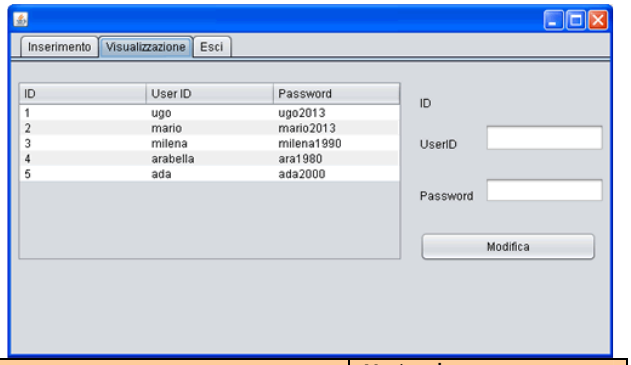
`  private void cancella(){`
`       txtUserId.setText("");`
`       txtPassword.setText("");`
`       lblMsg1.setText("");`
`  }`
`  private void cancella2(){`
`       txtUserId2.setText("");`
`       txtPassword2.setText("");`
`       lblMsg2.setText("");`
`  }`	

`  private void visualizza(){`
`    try{`
`       String intestazione []= {"ID", "User ID","Password"};`
`       DefaultTableModel elenco = new DefaultTableModel();`
`       elenco.setColumnIdentifiers(intestazione);`
	Visualizzazione di tutti i dati della tabella utenti nella **JTable tblElenco**

`       statement = (Statement)`	Recuperare le

`connessione.createStatement();`	informazioni del database e definire la query di interrogazione
`String query = "select * from utenti";`	Query di interrogazione
`ResultSet risultato = statement.executeQuery(query);`	Recuperare le informazioni da una **query di interrogazione** eseguita
`while(risultato.next()) { elenco.addRow(new Object[]{ risultato.getString("id"), risultato.getString("userid"), risultato.getString("password")}); } tblElenco.setModel(elenco);`	
`} catch(Exception e){ lblMsg2.setText("Lettura tabella completata"); } }`	

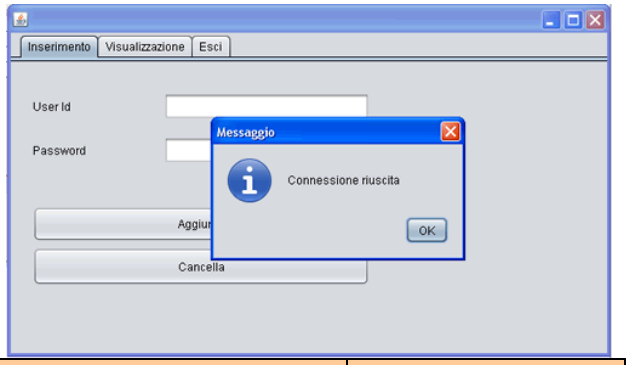
Output:
Visualizzazione contenuto della tabella



`public TestDBAccessUI() { initComponents();`	Metodo costruttore[25]
`this.setLocationRelativeTo(null);`	**Posizionare finestra** al centro dello schermo
`if (!connessione()) System.exit(0); }`	**Apertura della connessione**. Chiusura del programma per esito negativo

Output:
Apertura connessione, inserimento

[25] ***Metodo costruttore***. *Un costruttore di una classe viene definito con lo stesso nome della classe, è una caratteristica che lo differenzia da un normale metodo è il fatto di non avere un tipo di ritorno.*
Il metodo costruttore viene chiamo, solo ed esclusivamente, quando viene istanziata la classe, quindi si accede al costruttore una sola volta per ogni istanza della classe.



private void **btnAggiungiActionPerformed**(java.awt.event.ActionEvent evt) { try {	**Inserimento**
String sql = "Insert into utenti (userid, password) values ('" + txtUserId.getText()+ "', '" + txtPassword.getText()+ "')";	Query di comando
statement.**executeUpdate**(sql);	Recuperare le informazioni da una **query di comando** eseguita

lblMsg1.setText("Inserimento riuscito");
cancella();
}
catch (Exception e){
lblMsg1.setText("Inserimento NON riuscito " + e.toString());
}
}	

private void **pnlVisualizzazioneComponentShown**(java.awt.event.ComponentEvent evt) {
visualizza();
}	

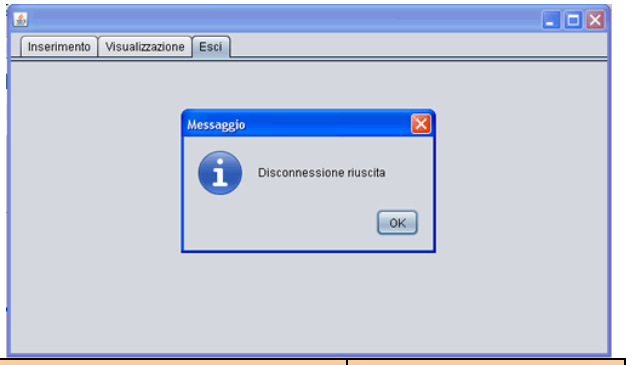
private void **btnModificaActionPerformed**(java.awt.event.ActionEvent evt) { try {	**Modifica**
String sql = "**Update** utenti " + " set userid = '" + txtUserId2.getText()+ "', " + " password = '" + txtPassword2.getText()+ "'" + " where id = " + Integer.parseInt(lblID.getText());	**Query di comando**
statement.**executeUpdate**(sql);	Recuperare le informazioni da una **query di comando** che viene eseguita

lblMsg2.setText("Modifica riuscita");
cancella2();
visualizza();
}
catch (Exception e){
lblMsg2.setText("Modifica NON riuscita " + e.toString());
}
}	

private void **pnlEsciComponentShown**(java.awt.event.ComponentEvent evt) {
disconnessione();
System.exit(0);
}
	Chiusura della connessione e del programma

Output:

```
Chiusura connessione e programma
```



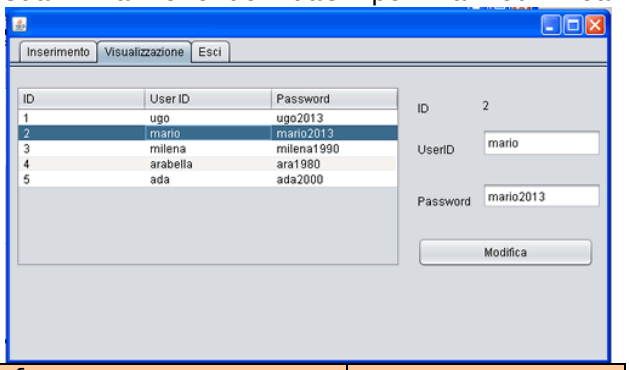
private void btnCancellaActionPerformed(java.awt.event.ActionEvent evt) {
    cancella();
}	

private void **tblElencoMouseClicked**(java.awt.event.MouseEvent evt) { Object dato = null;	**Scelta di una riga** della tabella
int row = tblElenco.getSelectedRow(); int col = tblElenco.getSelectedColumn();	**Posizione** della cella selezionata

dato = (Object)tblElenco.**getValueAt**(row,0);
lblID.setText(dato.toString());
dato = (Object)tblElenco.**getValueAt**(row,1);
txtUserId2.setText(dato.toString());
dato = (Object)tblElenco.**getValueAt**(row,2);
txtPassword2.setText(dato.toString());
}
	Cella selezionata: **dato** = (Object)tblElenco.getValueAt(row, col);
	Contenuto di una cella: **getValueAt**

Output:

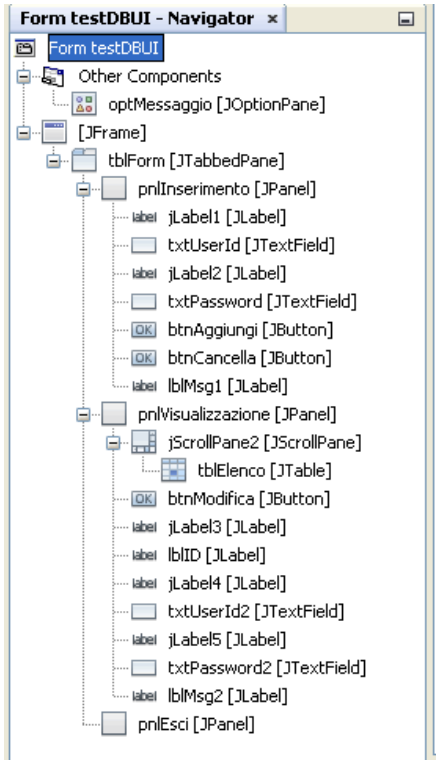
Selezione di una riga e visualizzazione dei dati per la modifica



public static void **main**(String args[]) {   :
    :
    :
}	

}	

Oggetti utilizzati nel progetto:



Query di comando

Le stringhe "sql":

```
String sql = "Insert into utenti (userid, password) values ('" +
txtUserId.getText()+ "', '" + txtPassword.getText()+ "')";
String sql = "Update utenti  set  userid = '" + txtUserId2.getText()+ "',
password = '" + txtPassword2.getText()+ "'"
           + " where id = " + Integer.parseInt(lblID.getText());
```

contengono istruzioni da eseguire e vengono passate come parametro al metodo executeUpdate di dichiarazione. Il valore restituito da tale comando corrisponde al numero di righe inserite o modificate. Se l'istruzione SQL eseguita fosse, ad esempio, una CREATE il valore restituito sarebbe 0. Il valore restituito, volendo, si può assegnare ad una variabile int:

```
dichiarazione.executeUpdate(sql);
int numero = dichiarazione.executeUpdate(sql);
```

Query di interrogazione

La stringa "query":

```
String query = "select * from utenti";
```

viene quindi eseguita grazie all'oggetto ResultSet:

```
ResultSet risultato = dichiarazione.executeQuery(query);
```

l'oggetto può essere letto in maniera sequenziale, scorrendo le righe che sono state restituite dall'interrogazione:

```
while(risultato.next())
{
```

```
  elenco.addRow(new Object[]{ risultato.getString("id"),
risultato.getString("userid"), risultato.getString("password")});
}
```

ovvero

```
while(risultato.next())
{
   elenco.addRow(new Object[]{ risultato.getString(0), risultato.getString(1),
risultato.getString(2)});
}
```

Esercizi svolti

La applicazione esegue le seguenti operazioni:

1. carica il diver **JDBC ODBC**.
2. Apre una fonte dei dati ODBC che apre il file **myDB.mdb (MS-Access)** presente nella direcory di lavoro dell'applicazione.
3. Ottiene l'oggetto Statement per l'esecuzione del codice SQL.
4. Genera il nome di una tabella con un generatore di numeri casuali.
5. Crea una tabella.
6. Immette 25 voci casuali nella tabella.
7. Visualizza il contenuto della tabella.
8. Cancella o svuota la tabella creata.
9. Chiude l'oggetto Statement poi chiude la connessione al database.

```
/* Program:
 * Setup database driver manager to understand and use ODBC MS-
ACCESS data source.
 * Written by Arnav Mukhopadhyay (ARNAV.MUKHOPADHYAY@smude.edu.in)
 * Compile as: javac dbAccess.java
 */

import java.sql.*;

public class dbAccess
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String database =
              "jdbc:odbc:Driver={Microsoft Access Driver
(*.mdb)};DBQ=myDB.mdb;";
            Connection connessione =
DriverManager.getConnection(database, "", "");
            Statement dichiarazione = connessione.createStatement();
            String tableName = "myTable" +                          Creare una table
String.valueOf((int)(Math.random() * 1000.0));
            String createTable = "CREATE TABLE " + tableName +
                                 " (id Integer, name Text(32))";
            dichiarazione.execute(createTable);
            for(int i=0; i<25; i++)                                 Inserire valori
            {                                                       nella table
              String addRow = "INSERT INTO " + tableName + " VALUES
( " +
                    String.valueOf((int) (Math.random() * 32767)) +
", 'Text Value " +
                    String.valueOf(Math.random()) + "')";
              dichiarazione.execute(addRow);
            }
            String selTable = "SELECT * FROM " + tableName;         Leggere la table
            dichiarazione.execute(selTable);
            ResultSet risultato = dichiarazione.getResultSet();
```

```
            while((risultato!=null) && (risultato.next()))
            {
                System.out.println(risultato.getString(1) + " : " +
risultato.getString(2));
            }
            String dropTable = "DROP TABLE " + tableName;              Svuotare la
            dichiarazione.execute(dropTable);                          table
            dichiarazione.close();                                     Chiudere il
            connessione.close();                                       database e
                                                                       disconnetetre
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
```

Le Applet Java

Che cos'è un' Applet Java?

Un'Applet Java non è altro che una comune applicazione grafica scritta in java, con la particolare caratteristica di essere destinata alle pagine web. Infatti essa non viene eseguita all'interno di una propria finestra, ma utilizza quella del browser.
Come vedremo più avanti, le applet vengono create estendendo la classe predefinita JApplet la quale eredita tutte le caratteristiche di un contenitore e quindi **l'applicazione può essere sviluppata come si crea una semplice interfaccia grafica con NetBeans**.

> ***Nota*** *- La classe JApplet deriva dalla classe pura Applet dove quest'ultima fu progettata per utilizzare i componenti AWT, infatti, i componenti Swing, subentrati successivamente, non vengono disegnati correttamente ed ecco perché si è pervenuti alla nuova classe JApplet che consente di utilizzare entrambe le tipologie.*

La figura seguente mostra il ciclo di vita di un'applet e i relativi metodi, da implementare, per gestire la sua esecuzione da parte del browser:

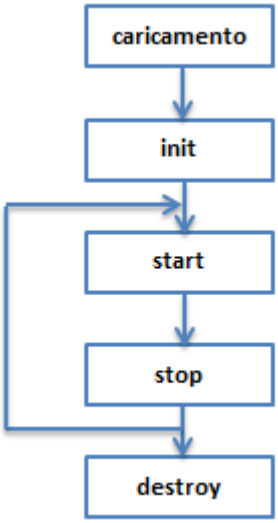


Figura 1

Appena viene caricata l'applet dal sito sul proprio PC il browser inizierà a chiamare i metodi come segue:

1. **init**: questo metodo ha la funzione di costruttore e quindi contiene tutta la fase di inizializzazione dell'applicazione, compresa la costruzione dell'interfaccia grafica. Esso viene eseguito una sola volta.
2. Il metodo **start** viene eseguito subito dopo il metodo "init". Ogni volta che ci si sposta su una pagina web diversa viene chiamato il metodo **stop** e viene richiamato di nuovo il metodo "start" ogni qualvolta si ritorna sulla stessa pagina.
3. **destroy**: viene chiamato dal browser una volta chiusa la pagina in cui vi è l'applet, questo per rilasciare le risorse fino ad allora impiegate.

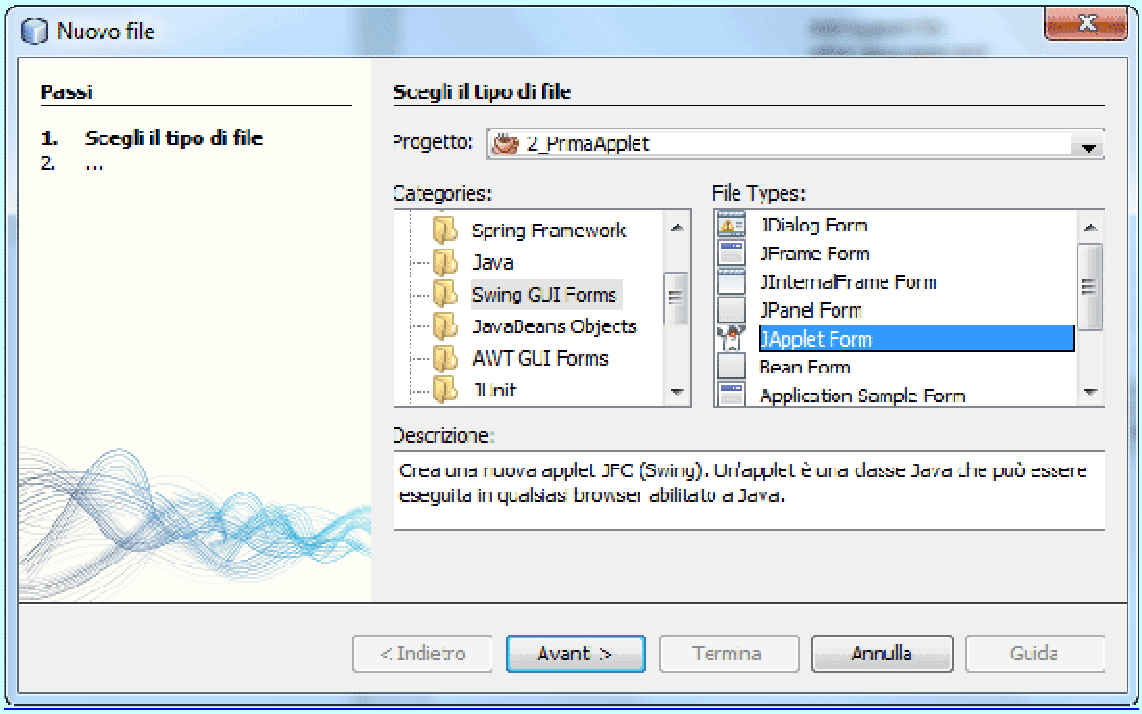
Come si può capire i metodi "**start**" e "**stop**" vengono **richiamati più volte**; un esempio di utilizzo possono essere le animazioni le quali vengono fermate quando l'utente non le sta guardando per poi riprendere l'esecuzione.
In sostanza, in termini molto più generali e tecnici, **i metodi "start" e "stop" servono per evitare di tenere il processore occupato quando l'applet non è più visibile**. Non tutti i metodi devono essere obbligatoriamente implementati.

Creazione dell'applet tramite NetBeans

Esercizi svolti

Dapprima creiamo un **nuovo progetto Java Application** e **togliamo il segno di spunta a Create Main Class**.
Clicchiamo col tasto **destro sul progetto e dal menu a tendina** cliccare su **New>Other....**
Dalla finestra **New file**,

1. selezioniamo **JAppletForm** dalla **categoria Swing GUI Forms**, come mostra la ***figura***:



2. scegliamo un **nome da dare alla classe** e cliccare su **Finsh**.
3. Trasciniamo **nell'area disegno** i componenti **JLabel** e i due pulsanti **JButton** e diamo alle rispettive variabili (Variable Name) un nome significativo, per esempio: **JLabel** > **msg_lbl**, **JButton1 > salutaPippo_btn**, **JButton2 > salutaPluto_btn**. Ovviamente, **cambiamo anche la proprietà text dei due pulsanti**, e l'eventuale **dimensione del font** dell'etichetta: insomma, divertitevi a scoprire gli effetti delle varie proprietà associate ai vari componenti!
4. Ora non ci resta che associare all'evento **actionPerformed** di **ogni singolo pulsante** il rispettivo codice che visualizzerà il messaggio di benvenuto(vedi articolo Creare una semplice interfaccia grafica con NetBeans):

```
private void salutaPippo_btnActionPerformed(java.awt.event.ActionEvent evt) {
   msg_lbl.setText("Benvenuto Pippo!");
}
private void salutaPluto_btnActionPerformed(java.awt.event.ActionEvent evt) {
   msg_lbl.setText("Benvenuto Pluto!");
}
```

5. Bisogna sottolineare che il **programma appena creato è privo di main,** pertanto non può essere eseguito utilizzando il comando java, richiamato da NetBeans attraverso il menu **Esegui>Run Main Project**, bensì deve essere **eseguito** attraverso il **comando appletviewer**, che simula il browser, il quale può essere richiamato **attraverso il menu Esegui>Run File**.

6. L'ultimo passo sarebbe quello di inserire l'**applet** all'interno di una **pagina web**, ma in questo tutorial non vedremo i dettagli del codice html
7. Nelle cartelle del progetto generate da NetBeans, una volta eseguita l'applet attraverso Esegui>Run File **NetBeans ha generato in automatico, tra le varie cartelle presenti nella cartella di destinazione**, specificata al momento della creazione del progetto la **cartella build** nella quale sono presenti sia il file html che una sottocartella, chiamata **classes**, contenente i **file .class dell'applet**. A questo punto potremmo sbirciare e manipolare il codice html per adattarlo alle nostre esigenze. Da tener presente che la **cartella classes** deve trovarsi **nella stessa cartella** in cui è presente il file **html**.

Sitografia

- **NetBeans IDE - The Smarter and Faster Way to Code:**
  - https://netbeans.org/features/index.html
  - https://netbeans.org/kb/docs/java/gui-builder-screencast.html
  - https://netbeans.org/kb/docs/java/quickstart-gui.html - Tutorial - Designing a Swing GUI in NetBeans IDE
  - https://netbeans.org/kb/docs/java/gui-functionality.html - Tutorial - Introduction to GUI Building
  - https://netbeans.org/kb/trails/matisse.html
- **The Java Tutorials - Using Swing Components**
  - http://docs.oracle.com/javase/tutorial/uiswing/components/index.html
  - http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html#features
  - http://docs.oracle.com/javase/6/docs/api/overview-summary.html

http://www.mrwebmaster.it/java/guide/guida-java/ - Guida scritta da Matteo Petrioli
http://www.datrevo.com/lezione-di-java-14-casting-implicito-ed-esplicito/
http://www.fnidea.it/mini-tutorial/java.html
http://www.html.it/articoli/gestione-delle-date-in-java-1/
http://www.java-samples.com/showtutorial.php?tutorialid=220
http://www.kodejava.org/examples/21.html
http://www.onlinetutorial.it/1373/le-stringhe-in-java.html
http://www.redbaron85.com/guide-di-programmazione/
http://www.redbaron85.com/guide-di-programmazione/51/
http://www.webmasterpoint.org/programmazione/java/java/definizione-costanti.html
http://www.webmasterpoint.org/programmazione/java/java/variabili-tipo-boolean.html
POLITO - http://elite.polito.it/files/courses/01KPS/laboratorio/Java_Date.pdf
POLITO - http://elite.polito.it/teaching-mainmenu-69/laurea-i-livello-mainmenu-82/96-01kpsbf
UNIFE - http://www.unife.it/ing/informazione/fond-info-modulo-b/materiale-didattico/
UNIMI - http://prog.di.unimi.it/laboratorio/lezioni/lez12-plain.pdf
UNIPI - http://projects.cli.di.unipi.it/LIP/LIP-07/index.html - Corso di Laurea in Informatica - Laboratorio di Introduzione alla Programmazione
UNIPI - http://www.di.unipi.it/~romani/DIDATTICA/LSD/LSD/Introduzione/Interfacce/main.html
UNIROMA - http://www.dis.uniroma1.it/~platania/documents/istruzioni_jdk.pdf
UNITO - http://www.di.unito.it/~nunnarif/informatica_applicata/dispense/InfApplicata-03-Utilities.pdf
http://www.liceocopernico.it/docenti/java/corsohtml/index.htm - Appunti sul linguaggio Java
http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html
http://java.sun.com/j2se/1.5.0/docs/api/java/util/GregorianCalendar.html
http://marcobonati.wordpress.com/2009/08/25/operazioni-aritmetiche-con-date-e-formattazione-in-java/
http://antoniotancredi.altervista.org/2010/04/07/java-la-classe-java-util-scanner/
http://tips4java.wordpress.com/2013/02/18/combo-box-with-hidden-data/ - Caselle combinate con dati nascosti
http://www.codeproject.com/Articles/35018/Access-MS-Access-Databases-from-Java

Bibliografia

Corso di Java - P.Camagni, R.Nikolassy - Edizioni HOEPLI

Java - P.Gallo, M.L.Pietramala - Edizioni Minerva scuola

Java 7 - Guida allo sviluppo - Datrio Guadagno - Edizioni FAG s.r.l.