

Altri Complementi di Java

Massimiliano de Leoni

L'intento di questo documento è di descrivere alcune caratteristiche di Java non descritte sul testo. Le funzionalità descritte sono le seguenti:

La classe File.....	1
JFileChooser	3
JOptionPane	5
La visualizzazione di tabelle in Java.....	7
La classe JTextArea	9
La classe Timer	11
La gestione degli eventi del mouse: MouseListener.....	12
Organizzazione dei file .class di un progetto: i file JAR	13
Ottenimento della date e dell'ora locale	14
Note riguardo JDK 1.5	15

La classe File

Questa classe è parte del package `java.io`. Sebbene la maggior parte delle classi definite da questo package agisce sui flussi, la classe `File` non lo fa; tratta direttamente con i file e con il sistema di cartelle: non specifica quali informazioni vengono recuperate dai file ma descrive le proprietà del file stesso.

Un oggetto `File` viene visualizzato per ottenere o manipolare le informazioni associate ad un file sul disco, quali i permessi, l'ora, la data e il percorso della directory e per gestire le gerarchie delle sottodirectory. Infatti, una directory viene tratta come `File`, con un'ulteriore proprietà: un elenco di nomi di file che può essere esaminati con un metodo.

Per creare oggetti `File` possono essere utilizzati i seguenti costruttori:

```
File(String pathDir)
File(String pathDir,String nomeFile)
File(File dirObj,String nomeFile)
```

dove `pathDir` è il nome di percorso del file, `nomeFile` è il nome del file e `dirObj` è un oggetto `File` che specifica una directory. Per quanto riguarda i separatori nel percorso Java accetta sia lo standard Unix/Internet con la barra (/) sia la convenzione Windows/DOS di barra rovesciata (\) con l'accortezza di dover utilizzare la sua sequenza di escape (\\) in una stringa. Ad esempio per creare un oggetto file che referenzi la radice di un disco si può fare indistintamente nei due seguenti modi

```
File f1=new File("/");
File f2=new File("\\");
```

A questo punto, ad esempio, per creare un riferimento al file `autoexec.bat` presente nella radice del disco si può fare:

```
File f3=new File(f1,"autoexec.bat");
File f3=new File("/autoexec.bat");
File f3=new File("/", "autoexec.bat");
```

Le tre operazioni sono identiche; il programmatore può scegliere il costruttore di volta in volta più comodo.

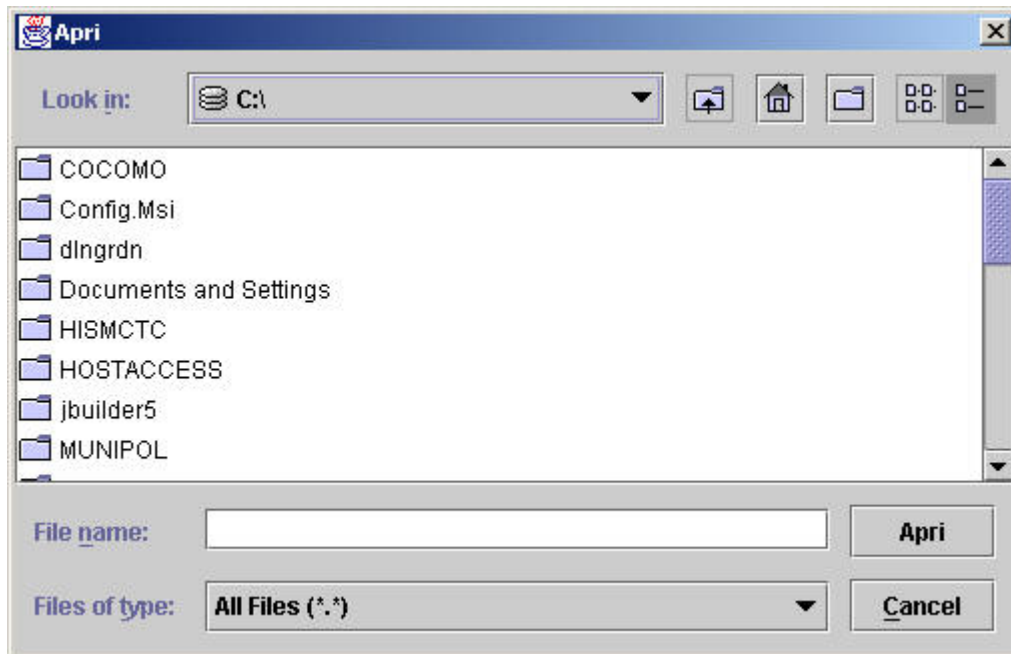
Di seguito alcuni dei metodi più utili messi a disposizione dalla classe:

<code>boolean exists()</code>	Restituisce true se l'oggetto File si riferisce ad un file che esiste
<code>boolean isDirectory()</code>	Restituisce true se l'oggetto File si riferisce ad una directory
<code>boolean isFile()</code>	Restituisce true se l'oggetto File si riferisce ad un file regolare
<code>String getName()</code>	Restituisce il nome del file referenziato dall'oggetto File
<code>String getAbsolutePath()</code>	Restituisce il percorso completo del file ref. dall'oggetto File
<code>int length()</code>	Restituisce la dimensione del file
<code>boolean renameTo(File f)</code>	Il nome del file specificato dal parametro diventa il nuovo nome dell'oggetto file. Restituisce true in caso di successo.
<code>boolean delete()</code>	Cancella il file referenziato dall'oggetto File
<code>String[] list()</code> <code>File[] listFiles()</code>	Se l'oggetto File si riferisce ad una directory, i due metodi restituiscono la lista dei file e delle directory presenti sottoforma di un array di stringhe o di oggetti File

Un'ultima osservazione: le classi che agiscono sui flussi come *FileInputStream* o *FileWriter*, ammettono un costruttore che prende in ingresso direttamente un oggetto file. Ad esempio se *f* è un oggetto *File* si può scrivere

```
FileInputStream in = new FileInputStream(f);  
FileWriter out = new FileWriter(f);
```

JFileChooser



Un file chooser è un oggetto grafico che permette di navigare il file system e di selezionare uno o più file su cui eseguire una determinata operazione. Qualunque applicazione grafica ne utilizza uno per facilitare le operazioni sul disco. Il componente `javax.swing.JFileChooser` offre questa funzionalità attraverso una accessoriata finestra modale.

Si può creare una istanza di `JFileChooser` utilizzando i seguenti costruttori:

```
JFileChooser()  
JFileChooser(File dirCorrente)
```

Il primo crea un `JFileChooser` che punta alla home directory dell'utente, il secondo punta alla directory specificata dal parametro.

Per visualizzare un `JFileChooser` è possibile ricorrere alla seguente coppia di metodi che restituiscono un intero:

```
int showOpenDialog(Component parent)  
int showSaveDialog(Component parent)
```

Il primo visualizza un `JFileChooser` per l'apertura di file; il secondo per il salvataggio. Entrambi i metodi prendono come parametro un componente: tipicamente il `JFrame` principale che verrà bloccato per tutta la durata dell'operazione. Passando `null` come parametro la finestra verrà centrata e risulterà indipendente dalle altre.

L'intero che viene restituito può assumere tre valori:

JFileChooser.APPROVE_OPTION	Se è stato premuto il bottone di conferma
JFileChooser.CANCEL_OPTION	Se è stato premuto il bottone Cancel
JFileChooser.ERROR_OPTION	Se si è verificato un errore.

Per conoscere il risultato dell'interrogazione è possibile usare i seguenti metodi:

File getCurrentDirectory()

File getSelectedFile()

che restituiscono rispettivamente la directory corrente e il file selezionato come oggetto File. Altri metodi permettono un uso più avanzato:

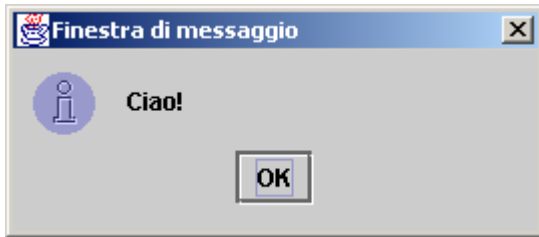
void setDialogTitle(String titolo)	Imposta il titolo della finestra del JFileChooser utilizzando come parametro una stringa
void setApproveButtonText(String titolo)	Imposta il testo del bottone di conferma del JFileChooser utilizzando il parametro stringa
void setFileSelectionMode(int modo)	Permette di selezionare solo file, directory o entrambi, utilizzando come parametro uno dei seguenti valori: JFileChooser.FILES_ONLY Solo i file JFileChooser.DIRECTORY_ONLY Solo le directory JFileChooser.FILES_AND_DIRECTORIES Entrambi
void setMultiSelectionEnabled(boolean a)	Abilita o disabilita la possibilità di selezionare più di un file alla volta
File[] getSelectedFiles()	Se è possibile selezionare più di un file alla volta, restituisce un vettore contenente i file selezionati dall'utente

Esempio

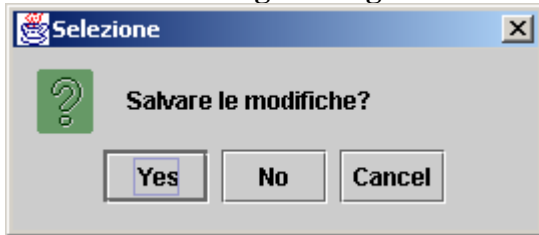
```
import java.io.File;
import javax.swing.JFileChooser;

public class Application
{
    public static void main(String args[])
    {
        JFileChooser fc=new JFileChooser();
        fc.setApproveButtonText("Apri");
        int value=fc.showOpenDialog(null);
        while(value==JFileChooser.APPROVE_OPTION)
        {
            File f=x.getSelectedFile();
            System.out.println("È stato selezionato il file "+f);
            value=fc.showOpenDialog(null);
        }
        System.exit(0);
    }
}
```

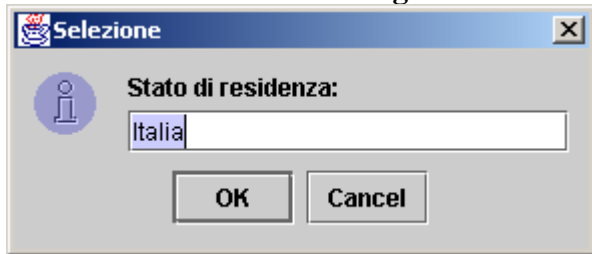
JOptionPane



Message Dialog



Confirm Dialog



Input Dialog

La classe `javax.swing.JOptionPane` permette di realizzare facilmente finestra modali di input, di allarme o di conferma. Le API di `JOptionPane` mettono a disposizione tre tipi di pannelli: Confirm, Input e Message Dialog. Il primo viene usato quando si deve chiedere all'utente di effettuare una scelta tra un gruppo di possibilità, il secondo torna utile quando si deve richiedere l'inserimento di una stringa di testo mentre il terzo viene usato per informare l'utente. La classe `JOptionPane` fornisce un gruppo di metodi statici che permettono di creare facilmente queste finestre ricorrendo ad una sola riga di codice. In questo paragrafo ci si concentrerà sull'uso di un sottoinsieme di tali metodi, nella convinzione che essi permettano di risolvere la stragrande maggioranza delle situazioni in modo compatto ed elegante.

Una finestra di conferma personalizzata si costruisce utilizzando il seguente metodo statico che ammette due forme:

```
int showConfirmDialog(Component parent, Object Message)
int showConfirmDialog(Component parent, Object Message, String title, int
    optionType, int messageType)
```

Il metodo restituisce uno dei seguenti valori a seconda se il bottone premuto è rispettivamente **Si**, **No**, **Annulla**, **Ok**:

```
YES_OPTION, NO_OPTION, CANCEL_OPTION, OK_OPTION
```

che sono delle costanti definite nella classe `JOptionPane`.

Una finestra di informazione personalizzata si costruisce con il seguente metodo statico che ammette due forme:

```
void showMessageDialog(Component parent, Object Message)
void showMessageDialog(Component parent, Object Message, String title, int
    messageType)
```

Una finestra di input personalizzata si ottiene con il seguente metodo statico che ammette tre forme:

```
Object showInputDialog(Component parent, Object Message)
Object showInputDialog(Component parent, Object Message, String title, int
    messageType)
Object showInputDialog(Component parent, Object Message, String title, int
    messageType, Icon icon, Object[] SelectedValues, Object initialValue)
```

Il metodo restituisce la stringa (come Object e quindi su cui è necessario fare il cast) inserita nell'InputDialog oppure null se è stato premuto il bottone annulla.

I metodi appena descritti richiedono di specificare i seguenti parametri

parent	Questo parametro serve a specificare il frame principale; esso verrà bloccato fino al termine dell'interazione. Ponendo a null questo parametro la finestra verrà visualizzata al centro dello schermo e risulterà indipendente dal resto dell'applicazione.
Message	Questo campo permette di specificare una stringa da visualizzare come messaggio oppure, in alternativa, una qualunque sottoclasse di Component
title	Questo parametro è utilizzato per specificare il titolo della finestra modale.
messageType	Attraverso questo parametro è possibile influenzare l'aspetto complessivo della finestra, per quanto riguarda il tipo di icona e il layout. Il parametro può assumere uno dei seguenti valori: <ul style="list-style-type: none"> • JOptionPane.ERROR_MESSAGE • JOptionPane.INFORMATION_MESSAGE • JOptionPane.WARNING_MESSAGE • JOptionPane.QUESTION_MESSAGE • JOptionPane.PLAIN_MESSAGE
optionType	I Confirm Dialog possono presentare diversi gruppi di opzioni per scegliere i bottoni da visualizzare nella finestra modale: <ul style="list-style-type: none"> • JOptionPane.YES_NO_OPTION • JOptionPane.YES_NO_CANCEL_OPTION • JOptionPane.OK_CANCEL_OPTION
icon	Ogni istanza della classe Icon rappresenta un'icona. In questo caso l'oggetto passato sarà visualizzato come icona di un Input Dialog. Nella maggior parte dei casi questo parametro può essere messo a null in modo tale che sia scelta l'icona di default.
SelectedValues	Questa parametro permette di impostare i possibili valori, tipicamente stringhe, che possono selezionati attraverso l'Input Dialog. I valori selezionabili sono inseriti in un ComboBox in modo tale che nessun altro valore è ammissibile. Se questo parametro viene messo a null allora il ComboBox è sostituito da un campo di testo e qualsiasi valore è ammesso.
initialValue	Questo parametro permette di stabilire il valore, tipicamente una stringa, che di default viene visualizzato.

Esempio

```
import javax.swing.JOptionPane;

public class Application
{
    public static void main(String args[])
    {
        JOptionPane.showMessageDialog(null, "Ciao!", "Finestra di "+
            "messaggio", JOptionPane.INFORMATION_MESSAGE);
        int yn=JOptionPane.showConfirmDialog(null, "Salvare le "+
            "modifiche?", "Selezione", JOptionPane.YES_NO_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE);
        if (yn==JOptionPane.NO_OPTION) System.exit(1);
        String prova;
        prova=(String) JOptionPane.showInputDialog(null,
```

```
        Stato di residenza:", "Selezione",  
        JOptionPane.INFORMATION_MESSAGE, null, null, "Italia");  
System.out.println("Selezionato: "+prova);  
String x[]={"a","b","c","d","e","f","g"};  
prova=(String)JOptionPane.showInputDialog(null,  
        "Scegli l'opzione:", "Selezione",  
        JOptionPane.INFORMATION_MESSAGE, null, x, x[3]);  
System.exit(0);  
    }  
}
```

La visualizzazione di tabelle in Java



Numero di Matri...	Nome	Cognome
09109923	Diego Armando	Maradona
09090232	Jose	Altafini
09119923	Alex	Del Piero
09110232	Filippo	Inzaghi
09109923	Diego Armando	Maradona
09090232	Jose	Altafini
09109923	Diego Armando	Maradona

Una *tabella* è un componente che visualizza righe e colonne di dati. È possibile trascinare il cursore sui bordi delle colonne per ridimensionarle; è anche possibile trascinare una colonna in una nuova posizione. Le tabelle sono implementate dalla classe `JTable`.

Uno dei suoi costruttori è mostrato di seguito:

```
JTable(Object dati[][], Object intCol[])
```

dove `dati` è un array bidimensionale delle informazioni da presentare e `intCol` è un array monodimensionale con le intestazioni delle colonne.

Ecco i passi per utilizzare una tabella in un frame:

1. Creare un oggetto `JTable`
2. Creare un oggetto `JScrollPane` dove l'argomento del costruttore specifica la `JTable` appena creata. In questo modo la tabella verrà aggiunta al pannello
3. Aggiungere il pannello di scorrimento al pannello dei contenuti del `JFrame` (per intendere quello ottenuto tramite il metodo `getContentPane()` della classe)

Un `JScrollPane` è un pannello di scorrimento che presenta un'area rettangolare nella quale si può vedere un componente. Se il componente ha dimensione maggiore del pannello vengono fornite barre di scorrimento orizzontali e/o verticali.

Esempio

```
import java.awt.*;
import javax.swing.*;

public class Tabella extends JFrame
{
    private JTable tab;
    private JScrollPane jsp;

    public Tabella(Object dati[][][],Object intCol[])
    {
        this.getContentPane().setLayout(new BorderLayout());
        tab=new JTable(dati,intCol);
        jsp=new JScrollPane(tab);
        this.getContentPane().add(jsp,BorderLayout.CENTER);
    }

    public static void main(String args[])
    {
        Object data[][][]={
            {"09109022","Paolo","Rossi"},
            {"09109923","Diego Armando","Maradona"},
            {"09090232","Jose","Altafini"},
            {"09119923","Alex","Del Piero"},
            {"09110232","Filippo","Inzaghi"},
            {"09109923","Diego Armando","Maradona"},
            {"09090232","Jose","Altafini"},
            {"09109923","Diego Armando","Maradona"},
            {"09090232","Jose","Altafini"}
        };
        String intestazioni[]={"Numero di Matricola","Nome","Cognome"};
        Tabella t=new Tabella(data,intestazioni);
        t.setSize(300,300);
        t.setVisible(true);
    }
}
```

La classe JTextArea

Questa classe costruisce un oggetto grafico da utilizzare quando si intende lavorare su testi di lunghezza media, con l'unica limitazione di permettere l'utilizzo di un unico carattere, stile e colore in tutto il documento. La via più breve prevede l'uso del seguente costruttore:

```
JTextArea(String text,int rows,int cols)
```

che crea una JTextArea con il testo specificato dalla stringa text e le dimensioni specificate dai parametri rows e cols.

È importante ricordare che questo componente non dispone di barre laterali di scorrimento; per questa ragione è indispensabile, all'atto della sua creazione, inserirlo dentro uno JScrollPane, concettualmente un pannello simile a JPanel ma che dispone di scrollbar. I passaggi necessari per fare questo sono i seguenti:

1. Creare un oggetto JTextArea
2. Creare un oggetto JScrollPane dove l'argomento del costruttore specifica la TextArea appena creata da aggiungere al pannello di scorrimento.
3. Aggiungere il pannello di scorrimento al pannello dei contenuti del JFrame (per intendere quello ottenuto tramite il metodo getContentPane() della classe)

Ad esempio, supponendo che il pannello dei contenuti della finestra sia in BorderLayout si può aggiungere una TextArea (con pannello di scorrimento) con tre semplicissime istruzioni:

```
...  
JTextArea ta=new JTextArea();  
JScrollPane scroll=new JScrollPane(ta);  
this.getContentPane().add(scroll, BorderLayout.CENTER);  
...
```

dove in questo caso si suppone che le prime due istruzioni siano due variabili di istanza e la terza un metodo invocato nel costruttore di una classe che estende JFrame.

Di seguito alcuni metodi che permettono di agire sul testo di una TextArea:

<code>void setText(String s)</code>	Cancella il contenuto del pannello e lo rimpizza con <code>s</code>
<code>String getText()</code> <code>String getText(int offs,int len)</code>	Restituisce il testo contenuto all'interno del componente. Se sono specificati i due parametri ne restituisce un sottoinsieme a partire dalla posizione <code>offs</code> e per <code>len</code> caratteri.
<code>void insert(String str,int pos)</code>	Inserisce la stringa <code>str</code> nella posizione specificata da <code>pos</code>
<code>void append(String str)</code>	Inserisce il contenuto della stringa in coda al documento.
<code>Void replaceRange(String str, int start,int end)</code>	Rimpiazza la sezione di testo compresa tra <code>start</code> e <code>end</code> con la stringa specificata

Un gruppo di metodi permette di conoscere la posizione attuale del cursore ed eventualmente modificarla; tra questi metodi ci sono anche quelli che permettono di operare direttamente sulla clipboard:

<code>int getCaretPosition()</code>	Restituisce la posizione del cursore
<code>void setCaretPosition(int pos)</code>	Sposta il cursore alla posizione specificata
<code>void cut()</code> <code>void copy()</code> <code>void paste()</code> <code>void selectAll()</code>	Questi quattro metodi effettuano le operazioni sugli appunti. Il primo e il secondo effettuano le operazioni di taglia e copia sulla selezione effettua all'interno degli appunti. Il terzo inserisce nella posizione corrente del cursore all'interno della TextArea il contenuto degli appunti. L'ultimo metodo seleziona <u>tutto</u> il contenuto della TextArea

La classe Timer

A partire dal JDK 1.3, è possibile programmare attraverso le classi `Timer` e `TimerTask` un compito per un'esecuzione in un tempo futuro ad ogni intervallo di tempo specificato. Quando scade l'intervallo viene eseguito il compito. La classe `Timer` si utilizza per programmare l'esecuzione di un compito il quale è rappresentato da un'istanza di `TimerTask`. Quindi per programmare un compito è prima necessario creare un oggetto `TimerTask` e poi schedarne l'esecuzione futura utilizzando un'istanza di `Timer`.

Tipicamente un compito si costruisce estendendo la classe `TimerTask` che definisce i seguenti due metodi:

<code>void run()</code>	Contiene il codice per il timer task
<code>boolean cancel()</code>	Termina l'esecuzione del compito. Restituisce true se la terminazione dell'esecuzione del task viene impedita.

Una volta che il task è stato creato ne viene programmato l'esecuzione futura attraverso un oggetto di tipo `Timer`. Questa classe definisce i seguenti metodi:

<code>void cancel()</code>	Cancella la programmazione del compito.
<code>void schedule(TimerTask T, long attesa)</code> <code>void schedule(TimerTask T, long attesa, int rip)</code>	T viene programmato per essere eseguito dopo che è trascorso il tempo <i>attesa</i> e poi ripetuto (se specificato) a intervalli di durata <i>rip</i> . I parametri <i>attesa</i> e <i>rip</i> sono specificati in millisecondi.

Esempio

```
import java.util.*;

public class TTest extends TimerTask
{
    String messaggio;
    int n;
    //rip è il numero di ripetizioni nella scrittura della data prima della
    //terminazione del programma
    int rip;
    public TTest(int aRip)
    {
        super();
        n=1;
        rip=aRip;
    }
    public void run()
    {
        GregorianCalendar gc=new GregorianCalendar();
        int hh=gc.get(Calendar.HOUR);
        int mm=gc.get(Calendar.MINUTE);
        int ss=gc.get(Calendar.SECOND);
        System.out.println("L'ora attuale è "+hh+":"+mm+":"+ss);
        if((n++)>rip)
        {
            this.cancel();
            System.exit(0);
        }
    }
}
```

```
    }  
}  
  
public class Application  
{  
    public static void main(String args[])  
    {  
        TTest x=new TTest(100);  
        Timer myTimer=new Timer();  
        myTimer.schedule(x,500,1000);  
        while(true); //Blocca l'esecuzione del programma  
    }  
}
```

La gestione degli eventi del mouse: MouseListener

Questa interfaccia definisce cinque metodi per gestire gli eventi del mouse. Questi metodi vengono idealmente invocati dai componenti quando l'utente clicca all'interno di questi con il mouse:

<code>void mouseClicked(MouseEvent me)</code>	Invocato quando il mouse viene premuto e rilasciato nello stesso punto.
<code>void mouseEntered(MouseEvent me)</code>	Quando il mouse entra in un componente
<code>void mouseExited(MouseEvent me)</code>	Quando il mouse esce da un componente
<code>void mousePressed(MouseEvent me)</code>	Quando il pulsante viene premuto
<code>void mouseReleased(MouseEvent me)</code>	Quando il mouse viene rilasciato

La classe `MouseEvent`, istanziata in maniera trasparente al programmatore, specifica alcune proprietà aggiuntive sull'evento scatenato. I due metodi più usati sono `int getX()` e `int getY()` che restituiscono le coordinate (x,y) del punto in cui si trovava il mouse quando si è scatenato l'evento.

Supponiamo che una certa applicazione sia stata progettata con una `Label` che cambia colore quando il mouse vi passa sopra e torna grigia all'uscita del mouse. Questo ovviamente può essere fatto implementando l'interfaccia `MouseListener`: dei 5 metodi a disposizione solo due sono necessari (in particolare il 2 e il 3 in tabella). Tuttavia, dovendo scrivere una classe che implementa tale interfaccia, è comunque necessario implementare anche gli altri 3 metodi, sebbene con corpo vuoto. Questa cosa può essere in alcuni casi noiosa, soprattutto se viene utilizzato un solo metodo; così Java ha pensato bene di costruire un'*adaptor* il `MouseAdapter`, una classe che implementa l'interfaccia `MouseListener` con 5 metodi vuoti. In questo modo nella gestione dei 2 metodi associati ai due eventi è possibile estendere l'*adaptor* definendo il corpo dei due metodi e trascurando gli altri.

Esempio: Un ascoltatore degli eventi del mouse che colora di diverso colore una `JLabel` quando il mouse vi passa sopra

```
import java.awt.event.*;  
import javax.swing.*;  
  
public class PassaListener extends MouseAdapter  
{  
    JLabel etichetta;  
    Color originale;
```

```
public PassaListener(JLabel aEtichetta)
{
    etichetta=aEtichetta;
    originale=aEtichetta.getBackground();
}

public void MouseEntered(MouseEvent me)
{
    etichetta.setBackground(Color.red);
}

public void MouseExited(MouseEvent me)
{
    etichetta.setBackground(originale);
}
}
```

Organizzazione dei file .class di un progetto: i file JAR

La convenzione JAVA nella nomenclatura delle classi prevede un file Java per ogni classe pubblica dell'applicazione (in particolare il file dovrà avere lo stesso nome della classe che definisce). Inoltre, qualora si definiscano package, ognuno di questi corrisponde nel File System ad una directory separata. Tipicamente una applicazione complessa può arrivare ad avere anche decine di classi che possono essere suddivise in package per semplificarne la ricerca; senza contare che le classi stesse potrebbero utilizzare file musicali, file immagine ed altri tipi di risorse che ovviamente devono essere distribuiti unitamente ai file .class. Ciò vuol dire che un progetto potrebbe contenere moltissimi file e ciò può complicare non poco l'invio e l'installazione di software.

Proprio a questo scopo, il linguaggio Java ha introdotto il concetto di **Java Archiver**(JAR). Un file JAR permette di disporre in modo efficace un set di classi e le risorse ad esse associate in un unico file. La tecnologia JAR può quindi semplificare non poco la distribuzione di una applicazione.

Dal punto di vista puramente tecnico, un file JAR non è altro che un file ZIP contenente i file dell'applicazione. All'interno dei file JAR chiaramente i file devono essere organizzati secondo la stessa struttura di file e directory che hanno quando sono decompressi sul disco. Ad esempio se un progetto consta di 5 file: a.class, b.class, c.class, d.class e e.gif raggruppati all'interno di una stessa directory, il file JAR(cioè il file ZIP) dovrà contenere compressi gli stessi file disposti nell'ipotetica radice del file JAR.

Per creare un file JAR, quindi, si può tranquillamente utilizzare un prodotto per la gestione dei file compressi (tipo **Winzip**) e poi successivamente rinominare il file .zip in file .jar. In alternativa, se non si è in possesso di nessun compressore ZIP si può utilizzare l'utilità jar messa a disposizione dal JDK.

Una volta che è stato creato un JAR è possibile far "girare" l'applicazione semplicemente nel modo seguente:

```
java -cp [nome_jar] [class_main]
```

dove **nome_jar** è il nome del file JAR (compreso l'estensione .jar) e **class_main** è la classe che ha definito il metodo main.

Sebbene questo metodo è sicuramente migliore di avere moltissimi file sparsi, sarebbe meglio che un ipotetico utente non debba sapere nulla sulla classe che definisce il main.

Ciò si può realizzare definendo il **file manifesto** che specifica informazioni aggiuntive. Tale file deve necessariamente chiamarsi MANIFEST.MF e si deve trovare nella sottodirectory META-INF del file JAR.

Un tipico esempio di file manifesto per specificare la classe con il main è il seguente:

```
Manifest-Version: 1.0
Created-By: 1.4.0_01 (Sun Microsystems Inc.)
Main-Class: [class_main]
```

A questo punto per far “girare” l’applicazione è sufficiente scrivere

```
java -jar [nome_jar]
```

Ottenimento della date e dell'ora locale

Il Java permette di conoscere l’ora e la data corrente attraverso la classe `GregorianCalendar` del package `java.util` costruendone un’istanza a partire dal costruttore senza parametri.

Il metodo più comunemente usato di questa classe è

```
int get(int campoCalendar)
```

che restituisce il valore di uno dei componente dell’oggetto chiamante. Il componente è indicato come parametro `campoCalendar`. Alcuni dei componenti che possono essere richiesti sono i seguenti:

<code>Calendar.YEAR</code>	Restituisce l’anno (corrente).
<code>Calendar.MONTH</code>	Restituisce il mese (corrente). Tale valore va da 0 a 11.
<code>Calendar.DAY_OF_MONTH</code>	Restituisce il giorno del mese (corrente).
<code>Calendar.HOUR_OF_DAY</code>	Restituisce l’ora (corrente).
<code>Calendar.MINUTE</code>	Restituisce il minuto (corrente).
<code>Calendar.SECOND</code>	Restituisce il secondo (corrente).

Esempio

```
import java.util.*;

public class Application
{
    public static void main(String args[])
    {
        GregorianCalendar cal=new GregorianCalendar();
        int aa=cal.get(Calendar.YEAR);
        int mm=cal.get(Calendar.MONTH)+1;
        int gg=cal.get(Calendar.DAY_OF_MONTH);
        System.out.print("Oggi è il giorno "+gg);
        System.out.println(" del mese "+mm+" anno "+aa);
    }
}
```

Note riguardo JDK 1.5

Il Java Development Kit 1.5 introduce molte estensioni al linguaggio Java, estensioni che ovviamente non sono compatibili con i JDK precedenti. Maggiori informazioni sulle peculiarità aggiunte a partire da questa versione sono disponibili al seguente URL nel sito della Sun: <http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html>. Il risultato di queste aggiunte è che molte classi, le quali nelle versioni precedenti compilavano correttamente senza nessuna segnalazione, a partire dalla versione 1.5 mostrano in compilazione molti *warning*¹, segnalati attraverso messaggi simili al seguente:

```
Note: Esame.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.
```

È possibile ottenere informazioni più dettagliate circa i messaggi di warning specificando l'opzione `-Xlint` in compilazione.

Ad esempio:

```
javac *.java -Xlint.
```

Ad esempio, nel caso di due warning, uno di tipo *unchecked* e l'altro *serial*, l'output dettagliato potrebbe essere il seguente:

```
Esame.java:109: warning: [unchecked] unchecked call to add(E) as a member  
of the raw type java.util.LinkedList  
    prenotazioni.add(unoStudente);  
Esame.java:9: warning: [serial] serializable class Esame has no definition  
of serialVersionUID  
    public class Esame implements java.io.Serializable  
           ^
```

Nel resto del paragrafo verranno illustrate le condizioni sotto le quali questi warning possono essere generati.

A partire dalla versione 1.5 il seguente (inutile!!!) frammento di codice genererebbe un *unchecked warning*:

```
LinkedList myIntList = new LinkedList(); //1  
For(int i=0;i<100;i++) //2  
    myIntList.add(new Integer(i)); //3  
Iterator iter=myIntList.iterator(); //4  
while(iter.hasNext()) //5  
{ //6  
    Integer x = (Integer) iter.next(); //7  
    System.out.println(x); //8  
} //9
```

Il motivo di questo warning è che la lista creata nel codice può essere poco sicura (unsafe) perché potenzialmente, senza un opportuno controllo da parte del programmatore, sarebbe possibile aggiungere alla lista oggetti di tipo diverso (ad esempio oggetti **Studente** e oggetti **Integer**). Una lista di oggetti incompatibili possono causare il lancio di eccezioni di tipo `ClassCastException` quando si tenta, dopo avere ottenuti gli elementi della lista attraverso il

¹ Si ricordi che un *warning* è un messaggio restituito dal compilatore per segnalare la presenza in un file sorgente di una operazione “potenzialmente pericolosa” o “sospettata errata” che comunque non rappresenta un errore semantico o sintattico e quindi non impedisce la compilazione.

metodo `get` o gli iteratori, si realizzare il necessario successivo cast (come alla riga 7). Questo perché i metodi di `LinkedList` prendono in ingresso e restituiscono `Object`. La Sun ha pensato bene a tal proposito di introdurre a partire dalla versione 1.5 il concetto di *generics*. I *generics* sfruttano il fatto che il programmatore nella maggior parte dei casi sa quale tipo di oggetti sono inseriti in una particolare lista. Nel codice questa conoscenza è evidente dalla presenza di un cast specifico. Tuttavia, quando questa conoscenza non è presente, come già detto, è possibile che venga generato un errore a run-time sul cast. I *generics* sono dei parametri speciali racchiusi tra un minore e un maggiore che possono seguire i riferimenti e i costruttori delle classi del Collection Framework (e quindi anche `LinkedList`) e che sono usati per specificare il tipo di oggetti che quell'oggetto del Collection Framework può collezionare.

Ecco il frammento del codice di sopra che sfrutta i *generics*:

```
LinkedList<Integer> myIntList =  
    new LinkedList<Integer>(); //1  
For(int i=0;i<100;i++) //2  
    myIntList.add(new Integer(i)); //3  
Iterator iter=myIntList.iterator(); //4  
while(iter.hasNext()) //5  
{ //6  
    Integer x = iter.next(); //7  
    System.out.println(x); //8  
} //9
```

Si noti la definizione della lista `myIntList`: viene specificato che non solo è una `LinkedList` ma in particolare che è una `LinkedList` di `Integer`.

In questo frammento alla riga 7 il cast non è più presente perché non necessario dato che il compilatore già sa che l'oggetto restituito è di tipo `Integer` poiché la lista, usando in *generics*, è dichiarata essere di oggetti `Integer`. Si potrebbe pensare che l'unica differenza è che il cast alla riga 7 è come se fosse stato spostato alla riga 1 ma non è così: la differenza è grande. Il compilatore in questo modo è in grado di verificare e garantire la correttezza degli oggetti aggiunti alla lista nella riga 3 a tempo di compilazione. In altre parole si ha la certezza che la lista manterrà sempre oggetti di tipo `Integer`. Il cast, infatti, dice solo al compilatore qualcosa che il programmatore pensa sia vero ad un certo momento dell'esecuzione.

L'effetto, specialmente nei grandi programmi, è una maggiore leggibilità e robustezza. Un altro tipo di *warning* introdotto a partire dalla versione 1.5 riguarda le classi che implementano l'interfaccia `Serializable`. Il compilatore associa ad ogni classe serializzabile un numero di versione, chiamato `serialVersionUID`, che è usato durante la deserializzazione per verificare che l'oggetto letto da file è compatibile con quello definito all'interno dell'applicazione. Se la classe caricata per quello oggetto ha un differente `serialVersionUID` rispetto a quello che ci si aspettava, significa che l'oggetto letto è incompatibile e quindi verrà lanciata una `InvalidClassException`. Una classe serializzabile può definire esplicitamente il proprio `serialVersionUID` esplicitamente definendo un campo statico di nome "`serialVersionUID`" di tipo `long`:

```
modificatore_di_accesso static final long serialVersionUID = valore;
```

Se una classe serializzabile non dichiara esplicitamente un `serialVersionUID` allora il compilatore calcolerà un valore di default per quella classe basandosi su vari aspetti della classe su cui in questa sede non entreremo in dettaglio. È comunque *fortemente raccomandato* che tutte le classi serializzabili esplicitamente dichiarino un `serialVersionUID` poiché il valore di default è altamente sensibile ai dettagli della classe e può variare con la versione del compilatore. Il risultato è di ottenere inaspettate `InvalidClassExceptions` durante le deserializzazioni. Per questa

ragione, per garantire consistenti valori di `serialVersionUID` tra differenti compilatori java su diverse piattaforme, occorre dichiarare esplicitamente un valore. A partire dal JDK 1.5, se `serialVersionUID` non viene esplicitamente definito, il compilatore restituisce un *Serializable warning*.

La Sun suggerisce di dichiarare privato il modificatore della variabile definita esplicitamente quando è possibile perché altrimenti si applicherebbe non solo alla classe dichiarata nell'immediato ma anche in futuro ad eventuali classi derivate.