LEZIONE 1

String concat(String str)

Int length()

String substring(int beginIndex) oppure String substring(int beginIndex, int endIndex)

String toLowerCase() oppure String toUpperCase()

String trim() ritorna una copia della stringa, omettendo gli spazie.

char <-> int : codice ASCII del carattere

Esempio

char c; int d;

c = (char) 65;

d = (int) 'A';

&& and

|| or

! not

Espressione condizionale:

(espressione booleana)?<valore1>:<valore2>

Esempio

String s = JOptionPane.showInputDialog("Inserisci un numero");

int x = Integer.parseInt(s);

double x = Double.parseDouble(s);

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

LEZIONE 2

Dichiarazione array:

int[] v = new int[100];

Dichiarazione Matrici:

double[][] matrix = new double[][];

matrix.length // numero di righe

matrix[0].length // numero di colonne

**InputStream**: E’ il capostipite degli stream di input per i byte

**OutputStream**: E’ il capostipite degli stream di output per i byte

**Reader**: E’ il capostipite degli stream di input per i caratteri

**Writer**: E’ il capostipite degli stream di output per i caratteri

Gestione tastiera: soluzione completa

• Per gestire correttamente la tastiera useremo quindi una sequenza di istruzioni di questo tipo:

*BufferedReader kbd = new BufferedReader(new InputStreamReader(System.in));*

Gestione del video: soluzione completa

• *PrintWriter video = new PrintWriter(System.out);*

– La classe FileReader (derivata da Reader) ci permette di leggere un file di testo

– La classe FileWriter (derivata da Writer) ci permette di scrivere in un file di testo

FileInputStream/FileOutputStream

– public FileInputStream(File file) throws FileNotFoundException

– public FileInputStream(String name) throws FileNotFoundException

*– public FileReader(File file) throws FileNotFoundException*

*– public FileReader(String name) throws FileNotFoundException*

*– public FileWriter(File file) throws FileNotFoundException*

*– public FileWriter(String name) throws FileNotFoundException*

*– public FileWriter(String name, boolean append) throws FileNotFoundException*

La classe File

*– public File(String pathname) throws NullPointerException*

*– public File(File parent, String child) throws NullPointerException*

*– public File(String parent, String child) throws NullPointerException*

• Campi più importanti:

*– public static final char separatorChar*: restituisce il carattere (dipendente dal sistema operativo) che separa i nomi nelle directory

• Metodi più importanti:

*– public boolean delete() throws SecurityException*

*– public boolean exists() throws SecurityException*

*– public String getAbsolutePath()*

*– public String getName()*

*– public boolean isDirectory()*

*– public boolean isFile()*

*– public long length()*

*– public String[] list()*

*– public File[] listFiles()*

*– public static File[] listRoots()*

*– public boolean mkdir() throws SecurityException*

*– public boolean mkdirs() throws SecurityException*

scrivi(new PrintWriter(System.out), "Ciao"); // stampa su schermo

scrivi(new PrintWriter(new FileWriter(f)), "Ciao"); // scrive su file

Leggere da file:

BufferedReader inFile = new BufferedReader(new FileReader(“inventory.dat”));

Scrivere su file:

PrintWriter outFile = new PrintWriter(new FileWriter("tabelline.txt"));

• Possiamo quindi utilizzare inFile.readLine() per leggere le righe del file

• Quando il file termina readLine() restituisce null

StringTokenizer

...

tokenizer = new StringTokenizer (line); // divide gli elementi delle righe

name = tokenizer.nextToken();

units = Integer.parseInt (tokenizer.nextToken());

price = Float.parseFloat (tokenizer.nextToken());

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

**LEZIONE 3 -CLASSI**

Regola generale

Campi dati (variabili di istanza) non public/tipicamente private

Campi operazioni (metodi) di servizio public

Campi operazioni (metodi) di supporto private

**ERRORI**

Eccezioni controllate

• Tutte le sottoclassi di IOException

– EOFException

– FileNotFoundException

– MalformedURLException

– UnknownHostException

– ClassNotFoundException

– CloneNotSupportedException

Eccezioni non controllate

• Tutte le sottoclassi di RunTimeException

– ArithmeticException

– ClassCastException

– IllegalArgumentException

– IllegalStateException

– IndexOutOfBoundsException

– NoSuchElementException

– NullPointerException

Catturare eccezioni

• Vengono eseguite le istruzioni all’interno del blocco **try**

• Se nessuna eccezione viene lanciata, le clausole **catch** sono ignorate

• Se viene lanciata una eccezione viene eseguita la corrispondente clausola catch

Dettagli su Exception

• Costruttori di oggetti della classe Exception:

– Exception() costruisce un’eccezione senza uno specifico messaggio

– Exception(String msg) costruisce un’eccezione con il messaggio msg

• Metodo ereditati dalla classe Throwable:

– String getMessage() restituisce come stringa il messaggio contenuto nell’eccezione

• Quando l’eccezione viene lanciata:

– può essere già stata creata o venir creata contestualmente

• Garbage collector: distrugge solo le eccezioni che sono state catturate

Per lanciare un’eccezione, usiamo la parola chiave throw (lancia), seguita da un oggetto di tipo eccezione

throw exceptionObject;

• Il metodo termina immediatamente e passa il controllo al gestore delle eccezioni

– Le istruzioni successive non vengono eseguite

Per segnalare le eccezioni controllate che il metodo può lanciare usiamo la parola chiave throws

public void read(BufferedReader in) throws IOException

• Un metodo può lanciare più eccezioni controllate, di tipo diverso

public void read(BufferedReader in) throws IOException, ClassNotFoundException

Per avere un messaggio di errore che stampa lo stack delle chiamate ai metodi in cui si è verificata

l’eccezione usiamo il metodo printStackTrace()

catch(DivisionePerZeroException exception) {

exception.printStackTrace();

}

A volte vogliamo eseguire altre istruzioni prima dell’arresto

• La clausola finally viene usata per indicare un’istruzione/blocco che va eseguita/o sempre

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

LEZIONE 4 EREDITARIETA’ E POLIMORFISMO.

public class Studente extends Persona{

…..

}

super(...) invoca il costruttore della classe base, deve essere la prima istruzione del metodo, se non presente si assume la chiamata super(), cioè al costruttore di default (se non esiste il compilatore segnala errore.

Polimorfismo: presenza di metodi con la stessa segnatura che si comportano in modo diverso all'interno di una gerarchia di classi

In Java, per i metodi statici si usa binding statico, mentre per i metodi di istanza si usa binding dinamico.

public static void main(String[] args) {

Studente s1 = new Studente("Maria Verdi",24,"1234");

Persona p2 = s1;

p2.stampa(); // binding dinamico

}

public static void stampa(Persona p) { … }

public static void stampa(Studente s) { … }

public static void main(String[] args) {

Studente s1 = new Studente("Maria Verdi",24,"1234");

Persona p2 = s1;

stampa(p2); // binding statico

}

UGUAGLIANZA E COPIA DI OGGETTI

protected Object clone()

// Crea e restituisce una copia di questo oggetto.

public boolean equals(Object obj)

// Verifica se il contenuto di questo oggetto è "uguale" a obj.

public Class getClass()

// Restituisce la classe a runtime a cui appartiene l'oggetto.

public String toString()

// Restituisce una stringa che rappresenta l'oggetto

toString() viene invocato automaticamente nelle operazioni di stampa

Tutti gli oggetti sono compatibili con la classe Object.

boolean r = (x.getClass().equals(y.getClass()); // verifica se i due oggetti x e y appartengono alla stessa classe

La classe Class definisce la funzione

boolean isInstance(Object)

che restituisce true se e solo se il suo parametro attuale è un riferimento ad oggetto di una classe

compatibile per l'assegnazione con la stessa classe dell'oggetto di invocazione.

Esempio

A a1=new A();

A.class.isInstance(a1) // vale true

|  |  |  |
| --- | --- | --- |
| **FUNZIONE** | **SVILUPPATORE DELLA CLASSE** | **CLIENT** |
| Uguaglianza superficiale |  | == |
| Uguaglianza profonda | Ridefinizione di equals() | equals() |
| Copia superficiale |  | = |
| Copia Profonda | Ridefinizione di clone() Costruttore di copia | clone() Costruttore di copia |

**CLASSI ASTRATTE E INTERFACCE**

Le classi astratte sono classi particolari, nelle quali una o più funzioni possono essere solo dichiarate (cioè si descrive la segnatura), ma non definite (cioè non si specificano le istruzioni).

public abstract class Persona {

// campi dati

…

// campi operazione

…

public abstract double aliquota();

}

Una classe astratta può contenere uno o più metodi astratti (non definiti).

public class Professore extends Persona {

// campi dati

…

// campi operazione

…

public double aliquota() {

return …;

}

}

Se A è una classe astratta:

• Si possono definire e usare variabili di tipo A

• NON si possono creare oggetti di tipo A

Esempio:

Persona p = new Persona(…); // ERRORE

Persona p = new Professore(…); // OK

p.aliquota(); // OK grazie al polimorfismo

**INTERFACCE**

Una interfaccia è un'astrazione per un insieme di funzioni pubbliche delle quali si definisce solo la segnatura, e non le istruzioni.

Un'interfaccia viene poi implementata da una o più classi (anche astratte). Una classe che implementa un'interfaccia deve definire o dichiarare tutte le funzioni dell’ interfaccia.

Si dichiarano solo funzioni pubbliche (non definite)

Non si possono dichiarare variabili (a meno che non siano final).

Non possiamo:

• creare oggetti di tipo I

• invocare altri metodi degli oggetti riferiti da I diversi da quelli presenti in I

Esempio

interface Tassabile {

double aliquota (); // dichiarazione

}

abstract class Persona implements Tassabile {

….

double aliquota (); // dichiarazione

}

class Professore extends Persona implements Tassabile { double aliquota () { // definizione

return …;

}

}

Ereditarietà:

interface I { void g();

}

interface J { void h();

}

interface M extends I, J { // vale solo per le interfacce

void k();

}

class C implements M { void g() { ... }

void h() { ... }

void k() { ... }

}

Interfacce e classi astratte hanno qualche similarità:

• entrambe hanno funzioni dichiarate e non definite;

• non esistono istanze di interfacce,

• non esistono istanze dirette di classi astratte.

Si tenga però presente che:

• una classe astratta è comunque una classe, ed è quindi un'astrazione di un insieme di oggetti (le sue istanze)

• una interfaccia è un'astrazione di un insieme di funzioni.

\* Solo campi dati final

|  |  |  |  |
| --- | --- | --- | --- |
| **class** | | **abstract** | **interface** |
| Riferimenti | SI | SI | SI |
| Oggetti | SI | (indirettamente) | NO |
| Campi dati | SI | SI | NO\* |
| Funzioni solo dichiarate | NO | SI | SI |
| Funzioni definite | SI | SI | NO |
| extends (abstract) class | 1 | 1 | NO |
| implements interface | >= 0 | >= 0 | NO |
| extends interface | NO | NO | >= 0 |

Interfacce Listener in una GUI

public interface MouseListener {

// Invocato quando si clicca sul componente associato

void mouseClicked(MouseEvent e);

// Invocato quando il puntatore del mouse entra nell'area del componente

void mouseEntered(MouseEvent e);

// Invocato quando il puntatore del mouse esce dall'area del componente

void mouseExited (MouseEvent e);

// Invocato quando un pulsante del mouse viene premuto sul componente

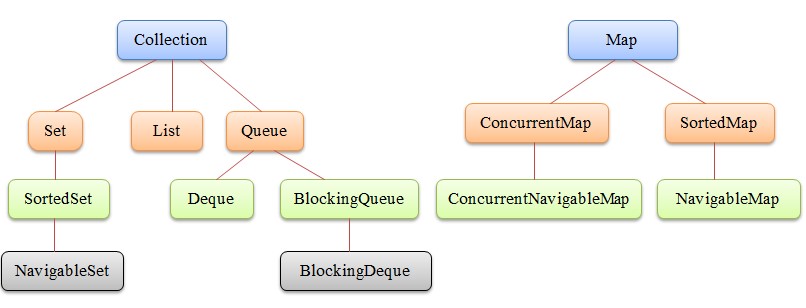
void mousePressed(MouseEvent e);

// Invocato quando un pulsante del mouse viene rilasciato sul componente

void mouseReleased(MouseEvent e);

}

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

LEZIONE 5 (JAVA COLLECTION FRAMEWORK)

Set: insiemi

-SortedSet: insiemi ordinati

List: liste (o sequenze)

Queue: code

-Deque: coda a doppia uscita

Map: mappe <chiave, valore>

-SortedMap: mappe con chiavi ordinate

……



I **tipi generici** sono usati in Java per definire classi, interfacce e metodi parametriche rispetto ad un tipo di dati su cui operano.

La definizione di un tipo generico C rispetto ad un altro tipo T viene indicata usando la notazione <T> nella dichiarazione di C

Esempio

class C<T> {

private T info;

public C(T info) { this.info=info; }

public T getInfo() { return info; }

public void setInfo(T info) { this.info=info; }

}

Un tipo generico C<T> deve essere usato istanziando il tipo di riferimento T con una classe specifica.

Esempio

C<String> c1 = new C<String>("ciao");

C<Persona> c2 = new C<Persona>(new Persona(…));

String s = c1.getInfo();

Persona p = c2.getInfo();

….

**Interfaccia Collection**

public interface Collection<E> {

// Basic Operations

int size();

boolean isEmpty();

boolean contains(Object element);

boolean add(E element); // Optional

boolean remove(Object element); // Optional

Iterator<E> iterator();

boolean equals(Object o);

// Bulk Operations

boolean containsAll(Collection<?> c);

boolean addAll(Collection<? extends E> c); // Optional

boolean removeAll(Collection<?> c); // Optional

boolean retainAll(Collection<?> c); // Optional

void clear(); // Optional

// Array Operations

Object[] toArray();

<T> T[] toArray(T[] a);

}

L'interfaccia Set non definisce alcuna nuova funzione rispetto a Collection.

**Interfaccia List**

public interface List<E> extends Collection<E> {

boolean add(int index, E element); // Optional

E get(int index);

E set(int index, E element); // Optional

int indexOf(Object o);

int lastIndexOf(Object o);

boolean remove(int index);

ListIterator<E> listIterator();

ListIterator<E> listIterator(int index)

boolean addAll(int index, Collection<? extends E> c); // Optional

List<E> subList(int fromIndex, int toIndex);

}

**Interfaccia Iterator**

public interface Iterator<E> {

boolean hasNext();

E next();

void remove(); // Optional

}

public interface ListIterator<E> extends Iterator<E> {

boolean hasPrevious();

E previous();

int nextIndex();

int previousIndex();

void set(E e); // Optional

}

**Iteratori**

Esempio

List<E> c = ... // lista di oggetti di tipo E

...

ListIterator<E> it = c.listIterator(c.size()); // iteratore per la lista c

// inizializzato per puntare all'ultimo elemento della lista

while (it.hasPrevious()) { //finche' il cursore non e' al primo elemento

E e = it.previous(); // poni l'elemento precedente in e

// e sposta il cursore indietro

... // processa l'elemento corrente (denotato da e)

}

**Mappe**

Una **mappa** è una struttura dati che serve a memorizzare coppie **(chiave,valore)**.

public interface Map<K,V> {

int size();

boolean isEmpty();

void clear(); // Optional

boolean equals(Object o);

boolean containsKey(Object key);

boolean containsValue(Object value);

V get(Object key);

V put(K key, V value); // Optional

V remove(Object key); // Optional

Set<K> keySet();

Collection<V> values();

}

**Collezioni ordinate**

Il Java Collections Framework prevede anche la possibilità di definire collezioni ordinate.

• SortedSet: per rappresentare insiemi di oggetti ordinati (non sono ammessi oggetti ripetuti).

• SortedMap: per rappresentare mappe ordinate per chiave.

**Collezioni ordinate: Comparable**

I tipi della collezione implementano l'interfaccia Comparable

public interface Comparable<T> {

int compareTo(T o);

}

compareTo() confronta l'oggetto di invocazione this con l'oggetto passato come parametro o, restituendo un intero negativo (-1) se this è più piccolo di o, 0 se sono uguali, o un intero positivo (1) se this è più grande di o.

**Collezioni ordinate: Comparator**

Si definisce una classe che implementa Comparator.

public interface Comparator<T> {

int compare(T o1, T o2);

}

compare() confronta gli oggetti o1 e o2, restituendo un intero negativo (-1) se o1 è più piccolo di o2, 0 se sono uguali, o un intero positivo (1) se o1 è più grande di o2.

Le classi che implementano SortedSet o SortedMap sono tipicamente dotate di due costruttori:

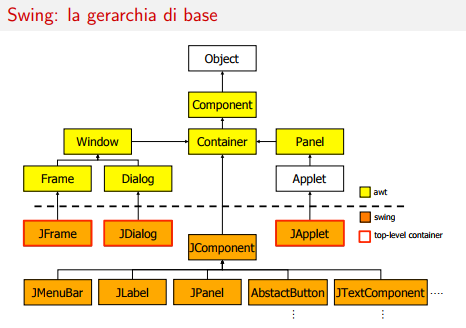
• Costruttore senza argomenti: che basa l'ordinamento degli oggetti nella collezione sul fatto che questi

implementino l'interfaccia Comparable

• Costruttore con un argomento di tipo Comparator: che basa l'ordinamento degli oggetti nella collezione

sull'ordinamento indotto dalla funzione compare del comparatore passato al costruttore

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

**LEZIONE 6 (JAVA.SWING)******

**I top-level container e le gerarchie di contenimento**

Un Container è un Component che può contenere altri componenti

Swing definisce tre top-level container: JFrame, JDialog, e JApplet

Ogni top-level container ha un contentpane che contiene (direttamente o indirettamente) i componenti visibili della GUI.

Il metodo setDefaultCloseOperation() ha 4 possibili operazioni:

JFrame.DO\_NOTHING\_ON\_CLOSE

JFrame.HIDE\_ON\_CLOSE

JFrame.DISPOSE\_ON\_CLOSE: elimina la finestra liberando la memoria allocata

JFrame.EXIT\_ON\_CLOSE

Principali metodi

JFrame(String titolo)

setTitle(String titolo)

void setSize(int lrg, int alt)

void setLocation(int x, int y)

void pack() = ottimizza le dimensioni della finestra

void setVisible(boolean b)

void setDefaultCloseOperation(int op)

Container getContentPane()

setContentPane(Container contentPane)

setJMenuBar(JMenuBar menubar)

Component add(Component comp)

Un Layout Manager è un oggetto che implementa l’interfaccia LayoutManager e determina il dimensionamento e posizionamento dei componenti all’interno di un Container. Ce ne sono diversi tra cui:

**FlowLayout, GridLayout, BorderLayout**

void setLayout(LayoutManager manager)

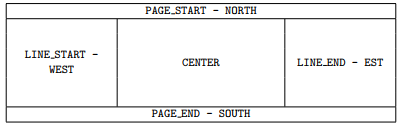
La classe **FlowLayout** implementa un layout manager che dispone i componenti grafici in un container “riga per riga”, tipici **JButton**

FlowLayout(int align): crea un FlowLayout e consente di specificare (parametro align) l’allineamento dei componenti, scegliendo tra FlowLayout.LEADING (allineamento a sinistra), FlowLayout.CENTER (allineamento al centro), o FlowLayout.TRAILING (allineamento a destra)

La classe **GridLayout** implementa un layout manager che dispone i componenti grafici in un container secondo una griglia di celle

il metodo void add(Component c, String pos) consente di aggiungere componenti specificandone il posizionamento.

il parametro pos può assumere i valori: BorderLayout.CENTER, BorderLayout.PAGE\_START (o BorderLayout.NORTH), BorderLayout.PAGE\_END (o BorderLayout.SOUTH), BorderLayout.LINE\_START (o BorderLayout.WEST), BorderLayout.LINE\_END (o BorderLayout.EAST)



Per supportare la definizione di una gerarchia di contenimento tra container usiamo oggetti della classe JPanel, un oggetto di classe JPanel è un JComponent che può essere aggiunto al content pane di un JFrame. Essendo anche un oggetto di tipo Container, un JPanel può a sua volta contenere componenti grafici e altri JPanel ogni oggetto JPanel può avere un proprio LayoutManager.

Esempio contenitori multipli

public final class MyFrame extends JFrame {

private static final String titolo = "Contenitori Multipli";

//Pannello Nord:

private static final JLabel infoLbl = new JLabel("Selezionare");

private static final JPanel nordPnl = new JPanel();

//Pannello Centrale:

private static final JCheckBox opz1Chk =

new JCheckBox("Opz1");

private static final JCheckBox opz2Chk =

new JCheckBox("Opz2");

private static final JPanel centroPnl =

new JPanel(new GridLayout(2,1));

//Pannello Sud:

private static final JButton okBtn=new JButton("OK");

private static final JButton cancBtn=new JButton("Annulla");

private static final JPanel sudPnl = new JPanel();

//Popoliamo i Container "dal basso verso l’alto" (nel costruttore)

public MyFrame() {

super(titolo);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

//Pannello Nord

nordPnl.add(infoLbl);

//Pannello Centro

centroPnl.add(opz1Chk);

centroPnl.add(opz2Chk);

//Pannello Sud

sudPnl.add(okBtn);

sudPnl.add(cancBtn);

//Container Principale

Container frmContentPane = this.getContentPane();

frmContentPane.add(nordPnl, BorderLayout.NORTH);

frmContentPane.add(centroPnl, BorderLayout.CENTER);

frmContentPane.add(sudPnl, BorderLayout.SOUTH);

//Impostiamo le proprieta’ di visualizzazione

// Imposta la dimensione minima

// per visualizzare tutti i componenti

this.pack();

// Posizioniamo la finestra al centro dello schermo

this.setLocationRelativeTo(null);

// Rendiamo visibile la finestra

this.setVisible(true);

}

}

Esempio JMenuBar

JFrame frm = new JFrame("Il mio Frame");

JMenuBar jb = new JMenuBar();

frm.setJMenuBar(jb);

oggetto JMenu

JMenuBar jb = new JMenuBar();

JMenu jm = new JMenu("Edit");

jb.add(jm);

Esempio JMenuBar

public class MyMenu extends JFrame {

private static final String titolo = "Un Menu di Esempio";

//Menu Move

private static final JMenuItem upIt = new JMenuItem("Up");

private static final JMenuItem dwnIt = new JMenuItem("Down");

private static final JMenu moveMenu = new JMenu("Move");

//Menu di Editing

private static final JMenuItem rndmIt = new JMenuItem("Random");

private static final JMenu editMenu = new JMenu("Edit");

//Menu File

private static final JMenuItem newIt = new JMenuItem("New");

private static final JMenuItem openIt = new JMenuItem("Open");

private static final JMenuItem saveIt = new JMenuItem("Save as...");

private static final JMenu fileMenu = new JMenu("File");

//Menu Bar

private final JMenuBar menuBar = new JMenuBar();

// Popoliamo i menu "dal basso verso l’alto" (nel costruttore)

public MyMenu() {

super(titolo);

this.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

// Move Menu

moveMenu.add(upIt);

moveMenu.add(dwnIt);

// Edit Menu

editMenu.add(moveMenu);

editMenu.add(rndmIt);

// File Menu

fileMenu.add(newIt);

fileMenu.add(openIt);

fileMenu.add(saveIt);

// Barra menu

menuBar.add(fileMenu);

menuBar.add(editMenu);

// Imposta la Barra Menu nel frame

this.setJMenuBar(menuBar);

// Impostazioni di visualizzazione

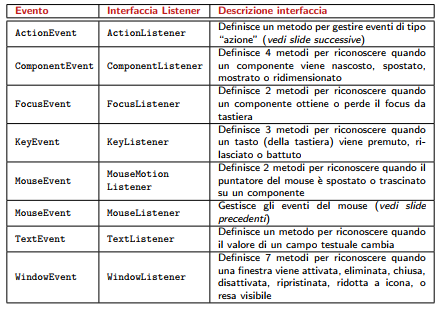
this.pack();

this.setLocationRelativeTo(null);

this.setVisible(true);

}

}



Esempio ActionListener

import java.awt.event.\*;

import javax.swing.\*;

public class MyFrameListener implements ActionListener {

public void actionPerformed(ActionEvent e) {

// gia’ sappiamo che assoceremo il Listener ad un bottone...

// getSource() restituisce un riferimento all’oggetto che ha

// generato l’evento

JButton b = (JButton) e.getSource();

// getText() di JButton restituisce il testo definito come

// etichetta del bottone

JOptionPane.showMessageDialog(null,"Hai premuto il pulsante: "

+ b.getText());

}

}

public final class MyFrame extends JFrame {

// private static final String titolo, ... larghezza, altezza...

private static final JButton uno = new JButton("uno");

// due, ... , cinque

// Creo un’istanza di MyFrameListener

private static final MyFrameListener listener = new MyFrameListener();

public MyFrame() {

super(titolo);

//chiusura, dimensioni, layout, content pane ...

// Associa listener al pulsante uno

uno.addActionListener(listener);

// due, tre, quattro

// Associa listener al pulsante cinque

cinque.addActionListener(listener);

frmContentPane.add(uno);

// due, tre, quattro

frmContentPane.add(cinque);

this.setVisible(true);}}

Implementazione

public class EsempioTextField extends JFrame implements ActionListener {

// Definizione di: centro, sud, campoTesto, button, titolo (come sopra)

private static final JTextField campoTesto = new JTextField(20);

public EsempioTextArea() {

// Creazione e impostazione finestra (come sopra)

// il listener e’ implementato dalla classe stessa

button.addActionListener(this);

this.pack();

this.setVisible(true);

}

}

// Implementazione di actionPerformed per la gestione degli eventi

public void actionPerformed(ActionEvent e) {

// Accede direttamente al componente "campoTesto"

JOptionPane.showMessageDialog(null, campoTesto.getText());

}

}

Esempio con classe esterna per il listener

public class Listener implements ActionListener {

// Costanti usate quando la stringa "comando" viene assegnata

public final String UPOPT = "up";

public final String DOWNOPT = "down";

public final String RANDOMOPT = "random";

public void actionPerformed(ActionEvent e) {

// legge il comando dall’evento

String com = e.getActionCommand();

// L’evento viene gestito in base al comando

// (se non e’ una stringa valida, nessuna azione viene eseguita)

if (com == UPOPT)

upOpt();

else if (src == DOWNOPT)

downOpt();

else if (src == RANDOMOPT)

randomOpt();

}

// Metodi specifici invocati in base alla stringa "comando" ricevuta

private void upOpt() { ... }

private void randomOpt() { ... }

private void downOpt() { ... }

}

public class MyFrame extends JFrame {

// ...

JMenuItem upOpt = new JMenuItem("Up");

JMenuItem downOpt = new JMenuItem("Down");

JMenuItem randomOpt = new JMenuItem("Random");

Listener ascoltatore = new Listener();

public MyFrame() {

// costruzione frame e menu ...

// lo stesso listener viene associato alle diverse voci di menu

upOpt.addActionListener(ascoltatore);

// upOpt scrive nell’ActionEvent generato la stringa Listener.UPOPT

upOpt.setActionCommand(Listener.UPOPT);

downOpt.addActionListener(ascoltatore);

// downOpt scrive nell’ActionEvent

// generato la stringa Listener.DOWNOPT

downOpt.setActionCommand(Listener.DOWNOPT);

randomOpt.addActionListener(ascoltatore);

// randomOpt scrive nell’ActionEvent

// generato la stringa Listener.RANDOMOPT

randomOpt.setActionCommand(Listener.RANDOMOPT);

// ...

}

}

Alcuni JComponent comuni

JButton, JCheckBox, JRadioButton, JSlider, JComboBox, JList, Jspinner

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

**Lezione 7 – Tipi astratti**

La prima scelta che dobbiamo fare riguarda il modo in cui la classe realizza le operazioni del tipo

astratto. In base a questa scelta, classifichiamo gli schemi realizzativi come:

– con side-effect, nel caso in cui i metodi della classe operano modifiche su oggetti di moduli clienti;

– funzionali, nel caso in cui i metodi della classe calcolano opportuni valori, restituendoli come risultati senza eseguire modifiche di oggetti appartenenti a moduli clienti.

Le possibilità a disposizione del progettista sono realizzazioni:

– senza condivisione di memoria, nel caso in cui i metodi della classe operano in modo da assicurare che le

rappresentazioni di oggetti diversi non condividano mai memoria;

– con condivisione di memoria, nel caso in cui i metodi della classe non impediscono che le rappresentazioni di oggetti diversi condividano memoria

In realtà in Java il recupero automatico della memoria dinamica non più referenziata favorisce

l'uso di due schemi realizzativi:

– lo schema **con side-effect senza condivisione di memoria**, per realizzare oggetti mutabili, cioè appartenenti a classi che sono dotate di metodi che effettuano side-effect cambiandone lo stato;

– lo schema **funzionale con condivisione di memoria**, per realizzare oggetti immutabili, cioè appartenenti a classi che non sono dotate di metodi che effettuano side-effect.

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

**Lezione 8 - Complementi**

Un docente insegna in più classi e ogni classe ha più docenti.

class Coppia {

Docente d; Classe c;

}

class DocenteClasse {

Set<Coppia> docenteclasse;

}

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

**Lezione 9 – Thread**

Esempio Thread

public class MiaClasseRunnable implements Runnable {

private String nome;

public MiaClasseRunnable(String n) {

nome = n;

}

public void run() {

for (int i = 0; i < 10; i++) {

System.out.println(nome + ": " + i);

}

System.out.println(nome + ": DONE! ");

}

}

public class Main {

public static void main(String[] args) {

MiaClasseRunnable ja = new MiaClasseRunnable("Jamaica");

Thread mioThread = new Thread(ja);

mioThread.start();

}

}

Il metodo Thread.sleep() lancia eccezioni checked di tipo InterruptedException che vanno gestate o con

try-catch o rilanciandole.

yield() mette in pausa temporaneamente il thread.

public boolean isAlive() in Thread

– Ritorna false se il thread non è mai stato avviato (attraverso start() ) o è TERMINATED

– Ritorna true se il thread è stato avviato e non fermato (quindi è RUNNING o NOT RUNNING)

void join() – Quando il thread corrente t1 invoca t2.join() si mette in attesa che il thread associato all’oggetto t2 termini (passo allo stato dead); quando t2 termina allora t1 riparte proseguendo con la sua istruzione successiva.

void join(long mills) – Attende al più mills millisecondi la morte del thread generato dall’ oggetto di invocazione

**Server Socket**

• Creare un’istanza della classe java.net.ServerSocket specificando il numero di porta su cui rimanere in ascolto

– ServerSocket serverSocket = new ServerSocket(4567);

Chiamare il metodo accept() che fa in modo che il server rimanga in ascolto di una richiesta di connessione (la porta non deve essere già in uso )

– Socket socket = serverSocket.accept();

Aprire un socket specificando indirizzo IP e numero di porta del server

– Socket socket = new Socket(“127.0.0.1”, 4567)

• Se la connessione ha successo si usano (sia dal lato client che dal lato server) gli stream associati al

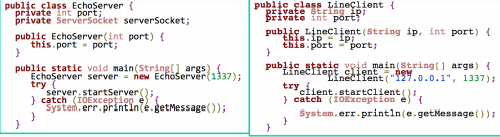
socket per permettere la comunicazione tra client e server (e viceversa)

– Scanner in = new Scanner(socket.getInputStream());

– PrintWriter out = new PrintWriter(socket.getOutputStream());



****



Classe InetAddress

• Rappresenta un indirizzo IP (sia versione 4 che 6).

Per creare un oggetto di tale classe:

– static InetAddress[] getAllByName(String host);

– static InetAddress getByAddress(byte[] addr);

– static InetAddress getByAddress(String host, byte[] addr);

– static InetAddress getByName(String host);

– static InetAddress getLocalHost();

• Metodi interessanti:

– String getHostAddress() ; /\* pretty print \*/

– String getHostName(); /\* eventuale lookup inverso \*/

– boolean isReachable(int timeout); /\* ping \*/

ECHOSERVER MULTITHREAD



Eccezioni controllate

• Tutte le sottoclassi di IOException

– EOFException

– FileNotFoundException

– MalformedURLException

– UnknownHostException

– ClassNotFoundException

– CloneNotSupportedException

Eccezioni non controllate

• Tutte le sottoclassi di RunTimeException

– ArithmeticException

– ClassCastException

– IllegalArgumentException

– IllegalStateException

– IndexOutOfBoundsException

– NoSuchElementException

– NullPointerException

Progettare nuove eccezioni

public class DivisionePerZeroException extends RuntimeException{

public DivisionePerZeroException() {

super("Divisione per zero!");

}

public DivisionePerZeroException(String msg) {

super(msg);

}

}

System.out.println(exception);

– Invoca il metodo toString() della classe DivisioneperZeroException

• Ereditato dalla classe RuntimeException

• Restituisce una stringa che descrive l’oggetto Exception costituita da

– Il nome della classe a cui l’oggetto appartiene seguito da : e dal messaggio di errore associato

all’oggetto

• Per avere un messaggio di errore che stampa lo stack delle chiamate ai metodi in cui si è verificata

l’eccezione usiamo il metodo printStackTrace()

catch(DivisionePerZeroException exception)

{

exception.printStackTrace();

}

\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

**Lezione 10 – Complementi sui Thread**

Monitor è il “Custode” di un oggetto che determina l’accesso a suoi comportamenti (metodi e sezioni di codice).

In Java la parola chiave è **synchronized**

Da ricordare:

– un solo thread alla volta può eseguire metodi synchronized

– un qualsiasi numero di thread può eseguire metodi non synchronized

**Deadlock**: due thread devono accedere a due oggetti ma ciascuno acquisisce il lock su uno dei due e poi aspetta il rilascio dell’altro oggetto da parte dell’altro thread, il che non avverrà mai perché a sua volta l’altro thread è in attesa del rilascio del primo oggetto da parte del primo thread.

Si noti che in JAVA (come in tutti i linguaggi OO moderni) synchronized è “rientrante”.

Cioè il thread che ha bloccato l'oggetto nell'eseguire del codice synchronized può invocare altri metodi synchronized (sullo stesso oggetto) senza creare problemi di deadlock.

Infatti vengono bloccati gli altri thread che invocano metodi synchronized sullo stesso oggetto, non il thread che ha il lock.

primitive aggiuntive:

– wait() ,

– notify(), notifyAll()

**Principali metodi dei Thread**

static Thread currentThread() : Returns a reference to the currently executing thread object.

void destroy() Deprecated.

long getId() : Returns the identifier of this Thread.

String getName() : Returns this thread's name.

int getPriority() : Returns this thread's priority.

ThreadGroup getThreadGroup() : Returns the thread group to which this thread belongs.

void interrupt() : Interrupts this thread.

boolean isAlive() : Tests if this thread is alive.

void join() : Waits for this thread to die.

void join(long millis) : Waits at most millis milliseconds for this thread to die.

void resume()Deprecated.

void run()

static void sleep(long millis)

void start()

void stop() Deprecated.

void stop(Throwable obj) Deprecated.

void suspend() Deprecated.

static void dd() : Causes the currently executing thread object to temporarily pause and allow other threads to execute.

