Università degli Studi di Napoli Federico II
Facoltà di Ingegneria

Introduzione al linguaggio Java

Dispensa didattica

Prof. Stefano Russo

Dipartimento di Informatica e Sistemistica

Aprile 1997

INDICE

IL LINGUAGGIO JAVA

Java è un linguaggio per la programmazione orientata agli oggetti di tipo interpretato o compilato "Just-in-time" che offre meccanismi di supporto allo sviluppo di applicazioni:

- portabili
- sicure (eg: controllo accessi in memoria)
- robuste (eg: controllo delle eccezioni)
- estensibili (collegamento dinamico)
- concorrenti (multithreading)
- distribuite (a scambio messaggi; integrazione con web browsers)

Java è simile al C++, ma rimuove molte caratteristiche del C e C++ come:

- Aritmetica dei puntatori
- Structs
- Typedefs
- Preprocessor
- New e delete

ORIENTAMENTO AGLI OGGETTI

A differenza del C++, che può essere considerato un linguaggio ibrido in quanto si possono scrivere programmi anche senza classi e oggetti, Java supporta pienamente la programmazione orientata agli oggetti

Fornisce meccanismi di:

- Incapsulamento
- Ereditarietà
- Polimorfismo
- Overloading

JAVA, INTERNET E IL WWW

Java offre meccanismi per lo sviluppo di applicazioni distribuite su rete

Esistono infatti librerie standard di classi predefinite per accedere e interagire con protocolli di rete (TCP, UDP, HTTP, FTP)

Questo rende l'accesso alle informazioni di rete facile quanto l'accesso ai dati locali

Le applicazioni Java inoltre sono facilmente integrabili in applicazioni basate sul WWW e su browser come Netscape

Infatti il linguaggio HTML per gli ipertesti sul Web prevede la possibilità di trasferire via rete ed eseguire applicazioni Java

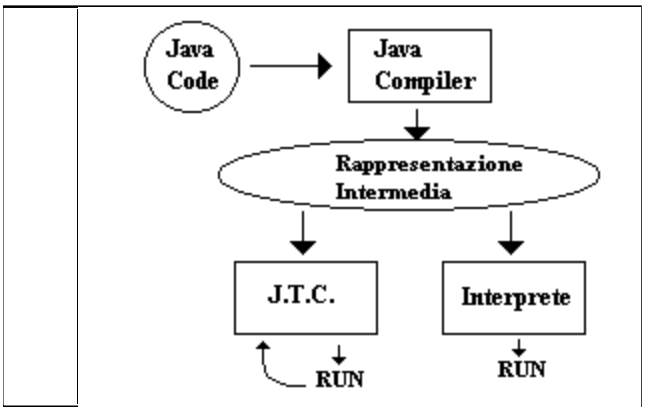
MODELLI DI ESECUZIONE

INTERPRETAZIONE E COMPILAZIONE JUST-IN-TIME

Il codice sorgente di un programma Java viene compilato e tradotto in una rappresentazione intermedia chiamata ***bytecode Java***

Il codice *bytecode* è indipendente dall'architettura hardware e può essere interpretato da qualsiasi sistema dotato del ***run-time Java***

Alternativamente, un programma Java, dopo la compilazione in codice *bytecode*, può essere sottoposto ad una compilazione finale per la generazione del codice in linguaggio macchina immediatamente prima dell'esecuzione



ROBUSTEZZA

Java fornisce meccanismi per:

- Controllo dei *null pointer*
- Controllo degli *array bounds*
- Gestione delle eccezioni
- Verifica dei *bytecode*
- *Garbage collection* automatica

Questi controlli riducono l'onere per il programmatore e consentono di evitare molti problemi a tempo di compilazione e malfunzionamenti del programma in esecuzione - ovviamente a scapito di una maggiore complessità del supporto a tempo d'esecuzione (RTS) del linguaggio e di un certo sovraccarico *run time*.

SICUREZZA

Java è più sicuro di linguaggi come C e C++ nel senso che, non prevedendo l'uso di puntatori, si evitano accessi incontrollati in memoria

I programmi Java:

- non possono richiamare funzioni globali
- non possono accedere arbitrariamente alle risorse del sistema

MULTITHREADING

Java supporta la programmazione concorrente

Un programma può essere costituito da più flussi di controllo o *thread* simultaneamente attivi

In questo modo Java supporta lo sviluppo di applicazioni di tipo interattivo e/o real time

Esempio:

> Si può caricare un immagine con un thread separato, permettendo l'accesso alle informazioni HTML della pagina

Non bisogna confondere multithreading con multiprocessing

Il **multiprocessing** è un meccanismo di sistema operativo che simula l'esecuzione simultanea di più processi (in un sistema monoprocessore), ai quali in un dato intervallo di tempo viene assegnata una percentuale d'uso della CPU

Il **multithreading** permette l'esecuzione concorrente di parti diverse di uno stesso programma, secondo una schedulazione definibile dal programmatore

Il primo quindi serve per eseguire più programmi alla volta, mentre il secondo per eseguire più sezioni di un programma alla volta

ESTENSIBILITA’

Un’applicazione Java è strutturata in moduli detti ***package***

Un *package* contiene un insieme di classi

Il collegamento dei moduli in Java viene fatto a tempo di esecuzione: un modulo viene caricato solo quando è richiesta l’esecuzione di una sua parte di codice

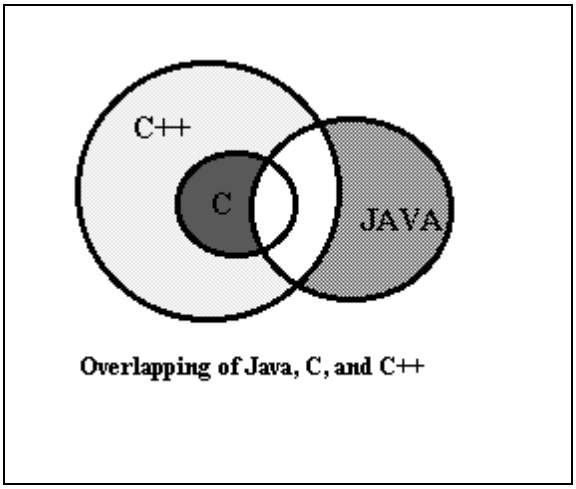
Né il codice eseguibile, né il *bytecode* contengono tutti i segmenti di programma già collegati

È possibile gestire la situazione in cui un modulo da eseguire non esiste e non viene rinvenuto nelle directory opportune
In altri termini l’applicazione non termina, ma gestisce le situazioni di eccezioni

CONFRONTO C/C++ e JAVA

Un programma codificato con il linguaggio di programmazione C++ non vuol dire che è stato progettato con la metodologia orientata agli oggetti - si possono scrivere programmi C++ senza classi né oggetti

Se non si considerano le parti di C++ che erano già presenti nel linguaggio C, Java non è poi così diverso dal C++



Overlapping of Java, C, and C++

I tre linguaggi di programmazione hanno delle caratteristiche in comune

Alcune delle caratteristiche che Java non ha più in comune con il C/C++ sono:

- variabili globali
- puntatori
- allocazione della memoria
- file di intestazione (*header file*)

CONFRONTO C/C++ VS. JAVA (segue)

Parole chiave comuni tra i linguaggi Java e C++

break	case	catch	char
class	const	continue	default
do	double	else	float
for	goto	if	int
long	new	private	protected
public	return	short	static
switch	this	throw	try
void	while	false	true
volatile			

Parole chiave del linguaggio Java

abstract	boolean	byte
byvalue	extends	final
finally	implements	import
instanceof	interface	native
null	package	super
synchronized	throws	

Parole chiave del linguaggio C++ che non sono presenti in Java

asm	auto	bool	delete
enum	extern	friend	inline
operator	namespace	register	signed
sizeof	struct	using	template
typedef	typeid	union	unsigned
virtual			

VARIABILI GLOBALI

La modifica di una variabile globale può avere effetti collaterali su tutto il sistema

In Java l'unico name space globale è la gerarchia di classi

PUNTATORI

Gli svantaggi legati all'utilizzo dei puntatori sono:

- una scorretta aritmetica dei puntatori è la causa di molti malfunzionamenti dei programmi
- gli indirizzi di memoria per la loro natura sono specifici di ogni macchina quindi puntatori a indirizzi specifici di memoria possono causare malfunzionamenti
- non possono esserci dati veramente privati

Il linguaggio Java non ha la capacità di manipolare direttamente i puntatori quindi, non è possibile:

- convertire un numero intero in un puntatore
- togliere i riferimenti ad un indirizzo arbitrario in memoria
- gli array sono oggetti realmente definiti quindi non è possibile scrivere oltre il termine allocato

ALLOCAZIONE DELLA MEMORIA

In Java la gestione della memoria heap viene affidata al garbage collector

L'utente non deve più utilizzare azione composte di malloc/free o new/delete per la gestione degli oggetti dinamici

FILE DI INTESTAZIONE

In Java non ci sono *file header*

Sia il tipo che la visibilità vengono compilati all'interno dei file della classe di Java

Il preprocessore C/C++ è stato eliminato.

Le costanti che venivano create precedentemente in C/C++ con *#define* vengono create con l'apposita parola chiave ***final***

JDK - Java Developer's Kit

L'ambiente di sviluppo JDK mette a disposizione i seguenti strumenti per creare ed eseguire applicazioni e applet Java:

- ***javac:*** compila i programmi Java in bytecode
- ***java:*** esegue i bytecode
- ***javadoc:*** genera documentazione in formato HTML da un codice sorgente java
- ***appletviewer:*** permette di eseguire le applet senza un browser Web
- ***jdb:*** aiuta a trovare e correggere i bug dei programmi
- ***javap:*** disassembla file compilati Java fornendo una rappresentazione dei bytecode
- ***jvah:*** crea file header C e file stub C per una classe Java permettendo al codice di interagire con codice scritto in altri linguaggi

Esempio: HELLO, WORLD!

Il primo programma Java:

```
1: class HelloWorld  {
2:    public static void main(String args[ ])  {
3:     System.out.println("Hello World!");
4:    }
5: }
```

Scrivere in un file di testo il codice sopra elencato

Registrare il file con il nome: *HelloWorld.java*

Compilare il file con il comando: *c:\java\bin\javac HelloWorld.java*

Eseguire con il comando: *c:\java\bin\java HelloWorld*

N.B.: Il nome della classe deve esser anche il nome del file

Riga 1: Viene utilizzata la parola chiave ***class*** per identificare una nuova classe

Riga 2: La parola chiave ***public***, indica che ogni classe può vedere il metodo main

La parola ***static***, permette di chiamare il metodo senza dover creare una particolare istanza della classe

Il metodo main dichiarato *static* viene richiamato dall'interprete dei *bytecode* Java prima di ogni istanza

La parola chiave ***void***, indica che il metodo non restituisce alcun valore

L'interprete Java cerca un metodo ***main*** ogni volta che deve interpretare una classe

Riga 3: Il metodo ***System.out.printl*** serve per visualizzare un messaggio sullo *standard output*

COSTRUTTI BASE DI JAVA

Commenti:

I commenti possono essere inseriti nel codice Java in vari modi:

- /* commento su una o più righe */
- // commento sulla riga
- /** commenti di documentazione */

Istruzioni:

Le istruzioni sono le più piccole unità eseguibili in un programma Java e normalmente terminano con il seguente carattere di punteggiatura ";"

Esempio:

```
int x;
x = 32 * 54;
```

Identificatori:

Gli identificatori possono iniziare con una:

- lettera
- underscore (_)
- dollaro($)

Una sequenza di caratteri può contenere dei numeri
Maiuscole e minuscole sono differenziate (*case sensitivity*)

Esempio:

```
int wonder_lust;
char _filelist;
float $money;
```

VISIBILITA' DELLE VARIABILI

Le variabili in Java sono valide soltanto dal punto in cui vengono dichiarate fino alla fine del blocco di istruzioni che le racchiude

Non è possibile dichiarare una variabile con lo stesso nome di un' altra contenuta in un blocco più esterno

Esempio:

```
class Scope {
    public static void main (String args[])  {
        int bar = 100;
        int ind = 0;
        for ( int i=1; i<bar; i++) {
            System.out.println("Valore : " +ind );
                                //creates a new scope
            int bar = 2;        //Compile time error...
        } // end for
    } // end main
} // end class
```

TIPI DI DATI SEMPLICI

Il linguaggio Java utilizza cinque tipi di dati semplici:

integer

floating point

character

boolean

string

Per utilizzare una variabile di un tipo predefinito non strutturato occorre:

- dichiararla
- inizializzarla

INTEGER

Integer Length	Name or Type
8 bits	*byte*
16 bits	*short*
32 bits	*int*
64 bits	*long*

CHARACTER

Un carattere è un intero senza segno lungo 16-bit

Esempio:

```
char key = 'a';
char tab = '\t';
```

BOOLEAN

Il tipo boolean ha due valori:

- true
- false

Non è possibile effettuare il casting tra i tipi interi e booleani

FLOATING POINT

Float Length	Name or Type
32 bits	*float*
64 bits	*double*

Esempio:

```
float pi = 3.14;
float a  = 3.1E12;
float b  = 2.3e12;
```

STRING

Esempio:

```
String s= "This is a string literal";
```

Le variabili membro vengono inizializzate con i seguenti valori quando vengono dichiarate (N.B.: questo non vale per le variabili automatiche):

Name of type	**Default Value**
byte	*0*
short	*0*
int	*0*
long	*0L*
float	*0.0f*
double	*0.0d*
char	*'\u0000'(NULL)*
boolean	*false*
tipi di riferimento	*null*

VARIABILI STRUTTURATE E OGGETTI

Le variabili non semplici (cioé strutturate) o di un tipo non predefinito sono considerate oggetti

Gli oggetti vanno:

- dichiarati
- istanziati
- inizializzati

Per l'istanziazione si usa la parola chiave *new*

L'istanziazione produce l'effettiva allocazione in memoria dell'oggetto

Esempio:

```
class Aharddisk {
    // dichiarazione delle var. membro e iniz. ai valori di default
    int size_of_disk;          // dichiarata e inizializ. per default
    int seek_time;             // dichiarata e inizializ. per default

public Aharddisk (int insize, int inseek) {
    size_of_disk =insize;
    seek_time= inseek;
    }

public void format () {
       int sector[];                 // dichiarazione della var
       sector[]= new int[200];       // istanziazione
       sector[0]= 0;                 // inizializzazione
       // code to format the disk
    }
```

ARRAY

In Java un *array* è un oggetto

Si possono dichiarare *array* di qualsiasi tipo

Esempio:

```
char s[ ];              // dichiarazione di array
int [ ] array;
int [ ] x, y[ ];
int x[ ], y[ ];
```

Gli *array* sono creati mediante la parola chiave ***new***, occorre indicare un tipo e un numero intero, non negativo, di elementi da allocare

Si possono creare array di *array*

Il run time Java controllerà che tutti gli indici dell'*array* si trovano nell'intervallo corretto

Java controlla in modo rigoroso che il programmatore non tenti accidentalmente di memorizzare o far riferimento ad elementi dell'*array* tramite un indice non valido

ARRAY *(segue)*

Gli *array* hanno un metodo chiamato ***length*** che restuisce la dimensione di un *array*

Esempio:

```
int table [] [] = new int [4] [5];        // dichiaraz. e istanz.
i=table.length;          // 4
table[0].length;         // 5
String names[] = { "marc" , "tom", "pete" };        //dichiaraz.,
                                                    //istanz. e inizializzaz.
```

Non si possono creare *array* statici a tempo di compilazione.

Esempio:

```
int list[50];       // Genera un errore in compilazione
```

OPERATORI

Gli operatori di Java sono caratteri speciali per istruire il compilatore sull'operazione che deve compiere con alcuni operandi

Le istruzioni dell'operazione vengono indicate dagli operatori

Gli operandi possono essere variabili, espressioni o valori costanti

Java ha quarantaquattro operatori differenti suddivisi in quattro gruppi:

- ***aritmetici***
- ***bitwise***
- ***relazionali***
- ***logici***

Gli operatori sono :

- **unari** - agiscono su un singolo operando

  prefissi - posti davanti all'operando (eg :!true, ~95, ++5, --4)

  postfissi - posti dopo l'operando (eg: 5++, 4--, 5<<l, 6>>l, >>>l)

- **binari** - posti tra due operandi (+, -, *, /, %, <, >, <=, >=, ==, !=, &, ^, |, &&, ||, ?:, =)

Operatori Aritmetici

Gli operatori sono usati per operazioni matematiche

Gli operatori devono essere di tipo numerico

Questi operatori non si possono usare con i tipi boolean, mentre è possibile con i tipi char

Il tipo char in Java è un sottoinsieme del tipo int

OPERATORE	RISULTATO	OPERATORE	RISULTATO
+	*somma*	+=	*somma e assegnamento*
-	*sottrazione*	-=	*sottrazione “*
*	*moltiplicazione*	*=	*moltiplicazione “ “*
/	*divisione*	/=	*divisione “ “*
%	*modulo*	%=	*modulo “ “*
++	*incrementa*	--	*deincrementa “ “*

Operatori Bitwise

OPERATORE	RISULTATO	OPERATORE	RISULTATO
~	*bitwise monadico NOT*		
&	*bitwise AND*	&=	*assegnazione*
|	*bitwise OR*	|=	*assegnazione*
^^	*bitwise esclusivo OR*	^=	*assegnazione*
>>	*scorrimento a destra*	>>=	*assegnazione*
>>>	*scorrimento a destra con riempimento di zeri*	>>>=	*assegnazione con riempimento di zeri*
<<	*scorrimento a sinistra*	=<<	*assegnazione*

Operatori Relazionali

Ogni tipo di dato in Java, inclusi gli interi, i numeri a virgola mobile, i caratteri e i booleani può essere confrontato utilizzando il test di uguaglianza

OPERATORE	RISULTATO
==	*uguale a*
!=	*diverso da*
>	*maggiore*
<	*minore*
>=	*maggiore o uguale*
<=	*minore o ugual*

Operatori Booleani

Gli operatori logici agiscono solo con operandi di tipo booleano.
Tutti gli operatori binari logici che combinano due valori boolean hanno come risultato un valore boolean

OPERATORE	RISULTATO	OPERATORE	RISULTATO
&	*logico AND*	**&=**	*AND assegnazione*
|	*logico OR*	|=	*OR assegnazione*
^^	*logico XOR*	^=	*XOR assegnazione*
||	*short-circuit OR*	==	*uguale a*
&&	*short-circuit AND*	**!=**	*non uguale a*
!	*logico monadico NOT*	**?:**	*ternario if-then-else*

ISTRUZIONI DI CONTROLLO FLUSSO

I test che vengono eseguiti dalle istruzioni di controllo flusso possono essere solo espressioni a valore booleano

IF - ELSE:

```
if (boolean) {
        istruzioni;
}
else {
        istruzioni;
}
```

Esempio:

```
int count = 5;
if ( count < 0 ) {
        System.out.println("Error: count value is negative"); // deve essere
                                                    // un'espressione a valore booleano
}
else {
      System.out.println("There will be "+ count +" people for lunch today");
}
```

CASE:

```
switch ( expr1 ) {
        case expr2: istruzioni;
                        break;

        case expr3: istruzioni;
                        break;

        default: istruzioni;
                    break;
}
```

ISTRUZIONI DI CONTROLLO FLUSSO (segue)

FOR:

```
for ( valore_iniziale; valore_finale; incremento) {
        istruzioni;
}
```

Esempio:

```
int count = 0;
for ( count = 0; count < 100; count++ ) {
        System.out.println("Valore -> ", count );
}
```

WHILE:

```
while ( boolean ) {
        istruzioni;
}
```

Esempio:

```
int count = 5;
while ( count < 0 ) {
        count --;
        System.out.println("Error: count value is negative");
}
```

DO - WHILE:

```
do {
        istruzione;
} while ( boolean );
```

Esempio:

```
int count = 0;
do {
        count++;
} while (count > 10)
```

LE CLASSI

In Java l'elemento fondamentale della programmazione orientata agli oggetti è la classe

La classe definisce il formato ed il comportamento di un oggetto

Una classe Java descrive lo stato di un oggetto e le operazioni che si possono effettuare sull'oggetto

Qualsiasi entità in un programma Java è incapsulata in una classe

Esempio:

```
class Car {
        // dichiaraz. delle var. e inizializzaz. a valori di default
        private int speed;
        private float fuel, avmpg;
        private int direction;
        private String colour;

        // costruttore
        public Car (String carColour) {
        colour= carColour;
        }

        // metodi della classe
        public void drive(int newSpeed)  {
                fuel-= (float)speed/avmpg;
                if (fuel<=0)
                        stop();
                else
                        speed= newSpeed;
        }

        public void stop() {
                applyBrakes();
                speed= 0;
        }
}
```

GENERAZIONE DI UN OGGETTO

Esempio:

```
Car Ferrari;                // dichiarazione di un oggetto
Ferrari= new Car("red");    // istanziazione e inizializzazione

Ferrari.drive(100);
Ferrari.stop();
Ferrari.fuel++;  // Compiler error....
```

L'accesso alle variabili private è consentito solo attraverso i metodi della classe

COSTRUTTORI

È un metodo che inizializza immediatamente un oggetto al momento della creazione

Ha lo stesso nome della classe in cui risiede e non ha un tipo di ritorno

Esempio6:

```
Car Ferrari= new Car("red");
```

Il metodo costruttore viene chiamato subito dopo che l'istanza e' stata creata, prima che new ritorni al chiamante

EREDITARIETÀ

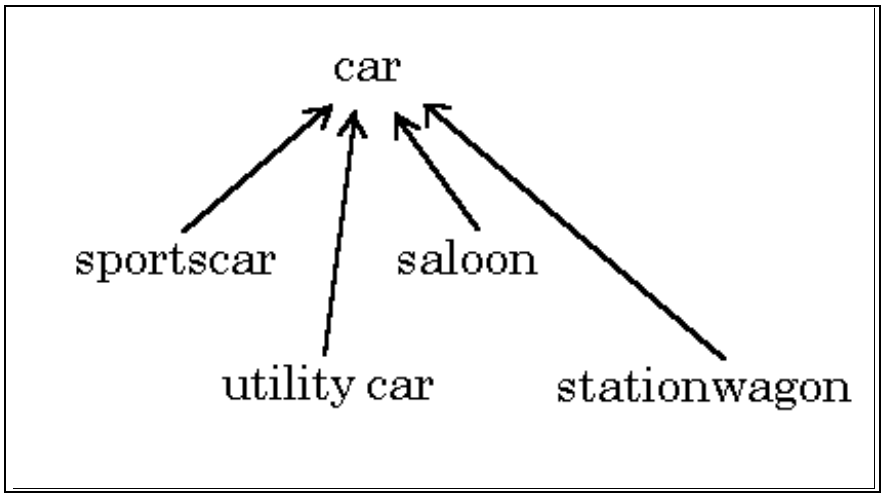
Una entità del mondo reale presenta generalmente numerose relazioni con altre entità

Esistono vari tipi di legami: di **uso** (A usa B), di **appartenenza** (A è una componente di B), di **specializzazione** (A è un caso particolare di B), etc.

Nelle tecniche object-oriented, tipicamente un oggetto corrisponde ad una entità reale (es: BankAccount), una classe descrive gli aspetti comuni ad un insieme di oggetti (es.: tutti gli oggetti di tipo BankAccount).
È importante poter rappresentare le relazioni tra le classi

L'ereditarietà è un meccanismo per descrivere le relazioni gerarchiche tra le classi => A eredita da B se è una particolarizzazione di B

Esempio:



L'ereditarietà offre vantaggi in termini di:

- **riuso**: consentendo di riusare la definizione di una classe nel definire nuove sottoclassi

- **estensibilità**: agevola l'aggiunta di nuove funzionalità, minimizzando le modifiche necessarie al sistema esistente quando si vuole estenderlo

EREDITARIETÀ IN JAVA

I discendenti di una classe ereditano tutte le variabili e i metodi dai loro antenati

Questi discendenti sono noti con il termine di sottoclasse e il genitore di una classe è chiamato superclasse

Java prevede la parola chiave **extends** per creare una sottoclasse

Esempio:

```
class SportsCar extends Car {
        public void control();
        public void applyBreak();
        public void stop();
        ....
}
```

Si può fare automaticamente riferimento alle variabili della superclasse

Java non supporta l'ereditarietà multipla

OVERLOADING DEI METODI

Possono essere definiti più metodi con lo stesso nome, ma con un diverso numero e/o tipo di parametri formali

Esempio:

```
public Car (String carColour);
public Car ();
public Car (String carColour, int maxSpeed);
```

PROTEZIONE DEGLI ACCESSI

Java fornisce molti livelli di protezione per permettere un controllo sulla visibilità di variabili e metodi

Classi e package sono due mezzi per incapsulare e contenere lo spazio dei nomi e lo scope di variabili e metodi

Java per implementare il meccanismo di protezione utilizza le seguenti parole chiave che definiscono il livello di accesso alle variabili e ai metodi definiti nella classe:

- ***public:*** può essere acceduto da qualsiasi classe da qualsiasi package

- ***protected:*** può essere acceduto dai membri della classe, da qualsiaisi sottoclasse e dalle altre classi del package

- ***private:*** può essere acceduto solo dall'interno della propria classe

- ***private protected:*** può essere acceduto dalla classe e dalle sottoclassi del package

Per *default* una classe può essere acceduta da qualsiasi altra classe all'interno del proprio package

PROTEZIONE DEGLI ACCESSI (segue)

Esempio:

FILE DAD.JAVA

```
package brothers;

class Dad {
    private boolean sportscar;
    protected boolean house;
    boolean fishingboat;                    // default
    private protectd boolean familycar;
    public boolean lawnmower;

    public Dad () {
        sportscar= true;            // Dad owns this stuff, he can use
        house= true;                // all of it.
        fishingboat= true;
        familycar= true;
        lawnmower= true;
    }
}


class UncleFrank {   // non sottoclasse ma risiede nello stesso package
    Dad mybrother= new(Dad);     // dichiaraz e istanziaz

    public UncleFrank() {
    //  mybrother.sportscar= true;          //ERROR! Dad only!
        mybrother.house= true;          //OK
        mybrother.fishingboat= true;    //Sure, no problem
    //  mybrother.familycar= true;      //ERROR! Only subclasses
        mybrother.lawnmower= true;      //Uncle Frank can borrow it
    }
}
```

PROTEZIONE DEGLI ACCESSI (segue)

FILE SON.JAVA

```
package kids;
import brothers.Dad;

class Son extends Dad {  //sottoclasse risedente in un package diverso

    public Son() {
    //  sportscar= true;          //ERROR! Son can't use sportscar!
        house= true;              //Yeah, fine
    //  fishingboat= true;        //No, only Dad and Uncle Frank
        familycar= true;          //OK, but be careful
        lawnmower= true;          //Son can use it (unfortunately)
    }
}
```

FILE NEIGHBOR.JAVA

```
package aliens;
import kids.Son;
import brothers.Dad;
import brothers.UncleFrank;

class nosy_neighbor {    //non sottoclasse, package diverso
    static public void main (String args[]) {
        Dad a_dad= new Dad();
        Son a_son= new Son();

    //  a_dad.sportscar= true;    //ERROR! Can't use sportscar!
    //  a_dad.house= true;        //ERROR! Sorry.
    //  a_dad.fishingboat= true;  //ERROR! Forget it.
    //  a_dad.familycar= true;    //ERROR! No way.
        a_dad.lawnmower= true;    //Even the neighbor can access.
        a_son.lawnmower= true;    //Son says it's OK too.
    }
}
```

THIS

In Java la parola chiave *this* viene utilizzata all'interno di un metodo per riferirsi all'oggetto corrente

Si può utilizzare *this* ogni volta che è richiesto un riferimento a un oggetto di un tipo della classe corrente

Esempio:

```
public Car (String tonality) {
        this.colour= tonality;
        colour= tonality;
}
```

Le due istruzioni in grassetto sono funzionalmente identiche

This è spesso usato:

- da un oggetto per mandare se stesso a altri metodi,

eg: un_oggetto.un_metodo(this) passa se stesso al metodo un_metodo chiamato sullóggetto un_oggetto

- oppure per distinguere una variabile membro da una automatica

Esempio:

```
public Car (String colour) {
        this.colour= colour;
 }
```

SUPER

Super riferisce in modo diretto i costruttori o le variabili membro della superclasse

È possibile chiamare un metodo della superclasse dalla sottoclasse

Esempio:

```
super.stop();
// chiama il metodo della superclasse sull'istanza this
Lamborghini.stop();
// chiama il metodo della sottoclasse sull'oggetto
   Lamborghini

class Car {
    private String colour;
    ....
    public Car (String carColour) {
        colour= carColour;
    }

    public void driveCar() {
        // code to drive Car
    }
}

class SportsCar extends Car {

    public SportsCar (String carColour) {
        super (carColour);
    }
    public void driveCar(String today) {
        if (today!= “Sunday”) super.driveCar();
        else        // code to drive Car
    }
}
```

BINDING DINAMICO DEI METODI

I metodi vengono selezionati sulla base del tipo di istanza in esecuzione, e non sulla classe in cui si sta eseguendo il metodo corrente

Esempio11:

```
class A {
        void callme() {
                System.out.printl(" Inside A's callme method");
        }
}

class B extends A {
        void callme() {
                System.out.printl(" Inside B's callme method");
        }
}

class Dispatch {
        public static void main( String args[] ) {
        A a = new B();
        a.callme();
        }
}
```

```
c:\>java\bin\ java Dispatch
Inside B's callme method
```

FINAL

Tutti i metodi e le istanze delle variabili possono essere sovrascitti

Se si desidera dichiarare che non si vuole più permettere alle sottoclassi di sovrascrivere metodi o variabili allora si dichiarano di tipo final

Questo meccanismo viene spesso usato per creare l'equivalente di const in C++

Final può essere utilizzato con:

- variabili/oggetti: rimangono costanti per tutta la loro esistenza
- classi: non possono diventare superclassi, pertanto sono le ultime della gerarchia
- metodi: non sono ridefinibili da nessuna sottoclasse

Esempi:

final float pi = 3.1415;

FINALIZATION

Il linguaggio Java durante l'esecuzione è caratterizzato dal meccanismo di ***garbage collection***

Il meccanismo di *garbage collection* svincola il programmatore dal liberare la memoria allocata per gli oggetti

Se una classe sta allocando alcune risorse per oggetti non appartenenti a Java allora si dovrebbe utilizzare *finalization* in modo da assicurare che queste risorse vengano liberate o deallocate

Bisogna quindi includere un metodo *finalize* a una determinata classe

Java durante l'esecuzione chiama quel particolare metodo ogni volta che viene reclamato dello spazio per quell'oggetto

Appena prima che una risorsa venga deallocata, Java chiama il metodo **void finalize()** sull'oggetto

STATIC

È il modo con cui Java implementa le funzioni globali e variabili globali

Static può essere utilizzato con:

- variabili: - appartengono alla classe e non alle sue istanze;
  - esiste una sola copia delle var. indipendentemente dal numero di oggetti istanziati, creata al caricamento della classe;
  - possono essere lette e/o scritte da oggetti della classe;
  - possono essere usate per la comunicazione allínterno di una classe;

- metodi: - appartengono alla classe e non alle sue istanze;
  - possono chiamare direttamente solo altri metodi static e non possono far riferimento a ***this*** o ***super***
  - le variabili su cui agiscono devono essere *static*, (variabili globali)
  - se il main non fosse dichiarato static e esistessero più istanze di una classe, il sistema non potrebbe sapere a quale oggetto appartenga il main da eseguire prima;

STATIC (segue)

Esempio:

```
class Testrun {

    int x;              // instance var
    static int y;       // static instance var

    static public void main (String args[]) {
       y= 10;
       x= 10;           // ERROR! x isn't static
       printer(10);     // ERROR! printer isn't static
    }

    void printer (int x) {
       System.out.println(x);
    }
}
```

Le due istruzioni in grassetto generano un errore in compilazione

Se fossero presenti diverse istanze di *Testrun* il main non avrebbe alcun criterio per decidere a quale oggetto appartenga la variabile *x* o il metodo *printer*

ABSTRACT

Permette di definire una classe che non fornisce l'implementazione di ogni metodo

Ogni sottoclasse di una classe abstract deve implementare tutti i metodi abstract nella superclasse o essere dichiarata essa stessa abstract

Non si possono dichiarare costruttori di tipo *abstract*

Una classe abstract non puo'essere istanziata

Un metodo abstract è solo definito, non è implementato e non può essere né static né private

Le classi abstract in Java sono leggermente diverse da quelle in C++

Una classe absrtract Java non può essere istanziata da sola, deve essere sottoclassata per essere usata

Una classe abstract C++ contiene una o più funzioni virtuali pure, cioé inizializzate a 0 (zero)

Esempio:

```
abstract class A {
        abstract void callme();
        void metoo()  {
                System.out.println("Inside A's metoo method");
        }
}

class B extends A {
        void callme()  {
        System.out.println("Inside B's callme method");
        }
}

class Abstract  {
        public static void main(String args[])  {
        A a = new B();
        a.callme();
        a.metoo();
        }
}
```

```
c:\> java\bin\java Abstract
Inside B's callme method
Inside A's metoo method
```

Esempio:

C++

```
class A {
        // ...
public
        virtual rotate(int)= 0;
};
```

Java

```
abstract class A {
        // ...

        abstract rotate(int);
}
```

MODULI (*PACKAGES*)

Si tratta del meccanismo per strutturare una applicazione in moduli, ***package*** ***≡ modulo***

I linguaggi C e C++ hanno le unità di traduzione, che non coincidono esattamente con i moduli

Sono un metodo per imporre restrizioni di visibilità

Si possono porre classi all'interno dei propri ***package*** rendendole completamente chiuse al mondo esterno

Il comando ***package*** dice al compilatore in che package devono essere definite le classi ivi incluse

Se viene omesso questo comando, le classi vengono poste nel *package* di default

Il nome della directory deve corrispondere esattamente al nome del *package*

Esempio:

package pkg1[.pkg2[.pkg3]];

package java.awt.MyPackage

CREAZIONE DI UN PACKAGE

Supponiamo di aver definito due classi *Rectangle* e *Triangle* nei file *Rectangle.java* e *Triangle.java*

Per creare un *package* chiamato *Shapes* composto da queste due classi bisogna aggiungere come prima riga ***package Shapes*** in ognuno dei due file

Compilare ognuno i file con **l'opzione -d** per indicare dove posizionare il *package*

Esempio:

c:\java\bin\ javac -d HOME/classes Rectangle.java
c:\java\bin\javac -d HOME/classes Triangle.java

Dopo la compilazione, *HOME/classes* ha una subdirectory chiamata *shapes* che contiene *Rectangle.java* e *Triangle.java*

Assicurarsi che *HOME/classes* sia aggiunta alla lista delle directory di *CLASSPATH*

CLASSPATH

La locazione specifica che il compilatore Java considera come radice e' memorizzata nella variabile d'ambiente CLASSPATH

Si può aggiungere il path della propria gerarchia di classi a CLASSPATH

IMPORT

Ciascuna classe è memorizzata in uno specifico *package*

Il comando import rende visibile determinate classi o interi package

Esempio:

```
import package1.[package2].(classname | *);
 // non esiste limite per la profondita' di una gerarchia
```

Tutte le classi predefinite di Java sono poste in un package chiamato java

Le funzioni basilari sono memorizzate nel *package java.lang*, che è implicitamente importato

Se una classe con lo stesso nome si trova in due diversi package importati usando l'asterisco, il compilatore genera un errore quando si prova ad usare una delle classi

In questo caso bisogna ricorre ad una denominazione esplicita della classe

CLASSI DI UTILITA'

La libreria di Java contiene un insieme di classi di utilità

Queste classi si trovano nei *package* ***java.lang*** e ***java.util***

Vengono utilizzate per memorizzare insiemi di oggetti e per interfacciare funzioni di sistema di basso livello

Come utility più interessanti da analizzare sono:

- **Runtime**
- **System**

LA CLASSE RUNTIME

La classe ***Runtime*** incapsula il processo in esecuzione dell'interprete Java

Non è possibile istanziare la classe *Runtime*, è possibile ottenere un riferimento all'oggetto *Runtime*

Nel momento in cui si ha una gestione a runtime, è possibile chiamare diversi metodi che controllino lo stato ed il comportamento della macchina virtuale Java

Alcuni metodi della classe:
- ***totalMemory:*** restituisce la dimensione della memoria heap
- ***freeMemory:*** indica quanta memoria heap è disponibile
- ***gc:*** esegue la raccolta della spazzatura automaticamente

È possibile anche eseguire altri processi su sistemi operativi multitasking

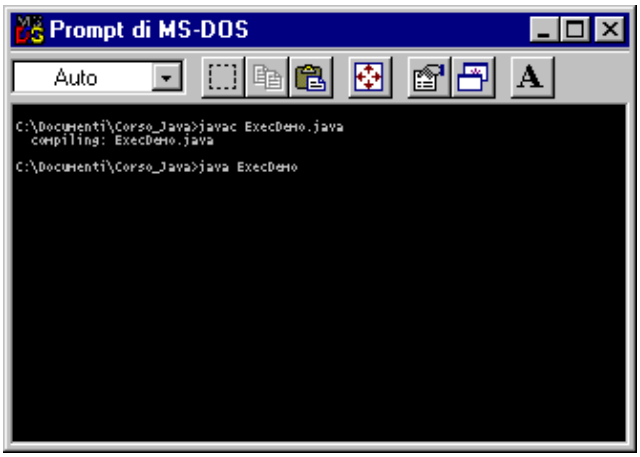
Ciò è possibile tramite il metodo ***exec***:
- ***exec("/usr/lib/ls"):*** sul sistema operativo Unix
- ***exec("notepad"):*** su Windows-95

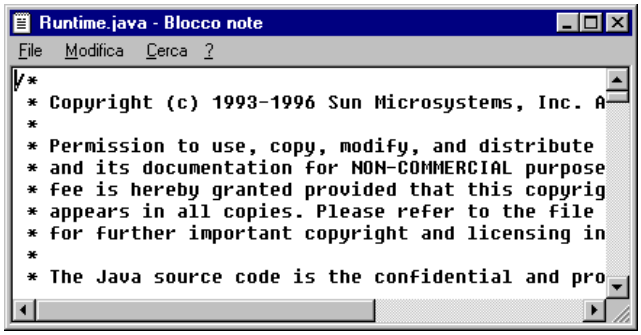
La forma del parametro di exec varia con il sistema operativo

Inoltre il metodo exec restituisce un oggetto di tipo ***Process*** che può essere utilizzato per controllare come il proprio programma Java interagisce con il nuovo processo in esecuzione

Esempio:

```
class ExecDemo {
        public static void main(String args[]) {
                Runtime r = Runtime.getRuntime();
                Process p = null;
                String cmd[] = { “notepad” , “/java/src/java/lang/Runtime.java”);
                try {
                        p = r.exec(cmd);
                } catch (Exception e) {
                        System.out.println(“Error :”+ cmd[0]);
                }
        }
}
```





È possibile eliminare il sottoprocesso tramite il metodo ***destroy***

Il metodo **waitFor** fa in modo che il programma attenda che il sottoprocesso termini, dopo di che il metodo exitValue restituisce il valore a sua volta restituito dal sottoprocesso nel momento in cui termina

LA CLASSE SYSTEM

Gli standard di input e output del run time Java vengono memorizzati nelle variabili:

- ***in***
- ***out***
- ***error***

Queste variabili sono lo standard input, output e error

Sytem.in: è la variabile che provvede a creare un canale di comunicazione con lo standard input, di default è la tastiera
c = Sytem.in.read()

Sytem.out: è la variabile che provvede a creare un canale di comunicazione con lo standard output, di default è il video
Sytem.out.print(“ hello “);
Sytem.out.println(a + b);
Sytem.out.println(“Somma” + a “ +” + b);

Sytem.err: è la variabile che provvede a creare un canale di comunicazione con lo standard error, di default è il video, ma è possibile anche ridirigerlo su un file
FileOutputStream diskfile = neew FileOutputStream(“error.log”);
System.err = new PrintStream(diskfile, true);
Sytem.err.println(“This is now to the disk file”);

INPUT & OUTPUT

Java implementa un meccanismo di astrazione dell' input/output

Tale astrazione si chiama ***stream***, serve per svincolare il programmatore dal tipo di periferica che vuole indirizzare

Lo *streaming* rappresenta tutte le sorgenti e le destinazioni di dati che stanno dietro a un interfaccia uniforme

Lo *stream* è implementato nel ***package java.io***

L'input appartiene alla classe InputStream mentre l'output appartiene alla classe OutputStream

FILE

Il *package* ***java.io*** include una classe ***File*** per comunicare direttamente con il file system

Questa classe permette di modificare data e ora dei file e di eseguire iterazioni su gerarchie di sottodirectory

La classe file non indica come un informazione viene ricercata oppure salvata in file

Una directory in Java può essere esaminata tramite il metodo ***list***

Gli oggetti file possono essere creati tramite tre costruttori:

- ***File f1 = new File("/"):*** l'oggetto file è costruito usando il percorso di un direttorio

- ***File f2 = new File("/","autoexec.bat"):*** l'oggetto file è costruito usando il percorso e il nome del file

- ***File f3 = new File(f1,"autoexec.bat"):*** l'oggetto file è costruito usando il percorso assegnato a *f1* e il nome del file

Java inoltre risolve correttamente le differenze tra i separatori di percorso convenzionalmente utilizzati dai sistemi Unix e Dos

Esistono metodi per modificare i parametri dei file come:

- ***renameTo(File dest):*** rinomina un File col nome del file passato come argomento

- ***delete:*** cancella il file sul disco rappresentato dal percorso dell'oggetto File

e altri per leggerli soltanto:

- ***getName():*** fornisce il nome del file
- ***getPath():*** fornisce la directory di appartenenza dell'oggetto File
- ***getAbsolutePath():*** fornisce il path assoluto dell'oggetto File
- ***getParent():***
- ***canWrite():***controlla se l'oggetto File può essere sovrascritto
- ***canRead():*** controlla se l'oggetto File può essere letto
- ***isDirectory():*** controlla se l'oggetto File è una directory
- ***isFile():*** controlla se l'oggetto File è un file
- ***length():*** fornisce la dimensione in byte dell'oggetto File
- ***lastModified():*** fornisce la data dell'ultima modifica dell'oggetto File

INPUTSTREAM

È una classe astratta che definisce il modello di streaming di input

Tutti i metodi di questa classe generano un IOException in condizioni di errore

Metodi della classe:

- ***read():*** restituisce il successivo byte di input disponibile
- ***close():*** chiude l'input

OUTPUTSTREAM

È una classe astratta che definisce il modello di streaming di output

Tutti i metodi di questa classe generano un ***IOException*** in condizioni di errore

Metodi della classe:

- ***write(int b):*** scrive un byte su uno stream di output
- ***flush():*** svuota il contenuto del buffer
- ***close():*** chiude l'outputstream

GESTIONE DELLE ECCEZIONI

Un' eccezione è una condizione anormale che si genera in una sequenza di codice durante l'esecuzione

La gestione delle eccezioni in Java è affrontata in modo *object-oriented*

Vengono utilizzate cinque parole chiave per gestire le eccezioni:

try

catch

throw

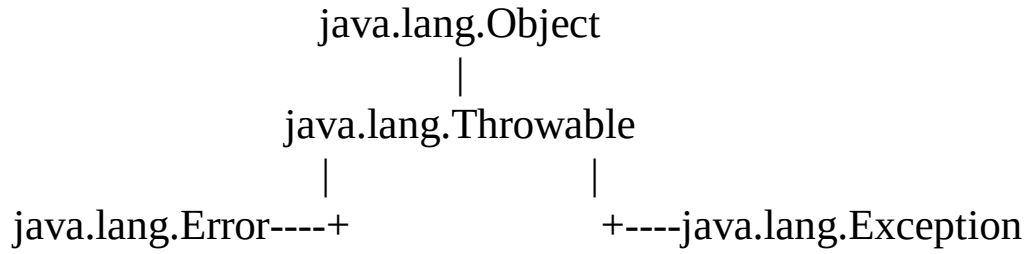
throws

finally

Esempio:

```
try {
        // block of code
}  catch ( ExceptionType1 e) {
        // exception handler for ExceptionType1
}  catch ( ExceptionType2 e) {
        // exception handler for ExceptionType2
throw(e);    // re-throw the exception...
}  finally {
}
```

Lo schema segunete indica quali *packages* vengono utilizzati per la gestione delle eccezioni:



- ***Exception:*** e' usata per le condizioni di eccezioni che i programmi utente dovrebbero raccogliere

- ***Error:*** definisce le condizioni che non ci aspettiamo siano raccolte in circostanze normali

Esempio:

```
class Exc0 {
    public static void main( String args[ ] ) {
        int d = 0;
        int a = 42 / d;
    }
}

c:\>jav\bin\java Exc0
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Java si accorge che il denominatore è nullo

Costruisce un nuovo oggetto di eccezione per fermare il codice e trattare la condizione di errore

Viene lanciata l'eccezione

Il flusso di codice viene interrotto all'operatore di divisione

Lo stack di chiamate viene ispezionato per trovare un gestore di eccezioni

Non trovando niente viene avviato il gestore di default

TRY & CATCH

Esempio:

```
class Exec2 {
    public static void main(String args[ ] {
        try {
            int d = 0;
            int a = 42 / d;
        } catch (ArithmeticException e) {
        }
    } // end_main
} // end_class
```

Lo scope delle istruzioni ***catch*** è ristretto alle istruzioni specificate nel comando ***try***

catch dovrebbe risolvere le condizioni di eccezione ponendo ogni variabile in uno stato ragionevole, continuando poi l'esecuzione

THROW

Viene utilizzato per lanciare esplicitamente un'eccezione

Esempio:

```
class ThrowDemo {
    static void demoproc() {
        try {
        throw new NullPointerException("demo");
        } catch (NullPointerException e) {
        System.out.println(“caught inside demoproc”);
        throw e;
        }
    }

    public static void main( String args[] ) {
        try {
        demoproc();
        } catch(NullPointerException e)  {
            System.out.println("re-caught: " + e);
        }
    }// end_main
} // end_class
```

```
c:\>jav\bin\ java ThrowDemo
caught inside demoproc
recaught: java.lang.NullPointerException: demo
```

THROWS

Identifica la lista di possibili eccezioni che un metodo potrebbe lanciare con throw

Il metodo descritto precedentemente deve esser modificato

Esempio:

static void demoproc() throws NullPointerException {

FINALLY

Ogni volta che un metodo sta per tornare al chiamante a causa di un'eccezione non raccolta o per un esplicito comando di return, l'istruzione finally viene eseguita appena prima che il metodo effettui il ritorno.

Esempio:

```
class FinallyDemo  {
    static void procA()  {
        try {
            System.out.println("Inside procA");
            throw new RuntimeException("Demo");
        }  finally  {
        System.out.println("procA's finally");
        }
    } // end_procA

    static void procB()  {
        try {
            System.out.println("Inside procB");
            return
        }  finally  {
            System.out.println("procB's finally");
        }
    } // end_procB

    public static void main( String args[])  {
        try {
            procA();
        }  catch(Exception e);
        procB();
    } // end_main
} // end_class

c:\> java\bin\java FinallyDemo
Inside procA
procA's finally
Inside procB
procB's finally
```

THREADS

Un *thread* è un flusso di controllo all'interno di un programma.
In un programma sequenziale c'è un songolo flussodi controllo (sistema *single threaded*)

Per gestire diverse risorse e/o dispositivi simultaneamente, i sistemi *single threaded* utilizzano un approccio detto ciclo di eventi con l'uso del *polling*

Un sistema multi-threaded sfrutta il fatto che molti thread di elaborazione spendono la maggior parte del loro tempo aspettando che una certa risorsa diventi disponibile

E' possibile associare ad ogni task il proprio thread essendoci poche probabilita' che piu' task contemporaneamente richiedano l'utilizzo della CPU

Java usa i thread per permettere all'intero ambiente di essere asincrono

Non c'e' un ciclo principale in un programma Java

PRIORITA'

Sono utilizzate per determinare il comportamento di ciascun thread rispetto agli altri

Le priorita' dei thread sono dei semplici numeri interi da 1-10

Un thread puo' cedere volontariamente il controllo oppure puo' essere forzato dall'esterno

SINCRONIZZAZIONE

Viene realizzata con i monitor

Ogni oggetto ha il suo monitor

Un oggetto entra in un monitor richiamando uno dei metodi sinchronized dell'oggetto

A questo punto nessun altro thread puo' chiamare un altro metodo synchronized sullo stesso oggetto

LA CLASSE THREAD

Incapsula tutto il sistema di controllo sui thread

Bisogna distinguere un oggetto Thread e un thread in esecuzione

Esempio:

```
class CurrentThreadDemo {
    public static void main(String args[]) {
        Thread t= Thread.currentThread();
        t.setName("My Thread");
        System.out.println("current thread: " + t):
        try {
            for (int n=5; n>0; n--) {
            System.out.println("" +n);
            Thread.sleep(1000);
        }
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    } end_m,main
} // end_class
```

```
c:\> java\bin\java CurrentThread
current thread:   Thread[My Thread,5,main]
5
4
3
2
1
```

RUNNABLE

Esempio:

```
class ThreadDemo implements Runnable {
        public void ThreadDemo() {
                Thread ct = Thread.currentThread();
                System.out.println("currentthread: " +ct);
                Thread t = new Thread( this, " DEMO THREAD");
                System.out.println(" THREAD CREATED: " +t);
                t.start();
                try {
                        Thread.sleep(3000);
                } catch (InterruptedException e ) {
                        System.out.println("interrupted");
                }
                System.out.println("exiting main thread");
        } // end_threadDemo

        public void run() {
                try {
                        for(int i=5;i>0; i--)  {
                        System.out.println("" +i);
                        Thread.sleep(1000);
                }
                } catch (InterruptedException e)  {
                        System.out.println("child interrupted");
                }
                System.out.println("exiting child thread");
        } // end_run

        public static void main(String args[])  {
                Thread tt= new ThreadDemo();
        } // end_main
} // end_class
```

```
c:\> java\bin\java ThreadDemo
currentthread:   Thread[main,5,main]
THREAD CREATED:     Thread[DEMO THREAD,5,main]
5
4
3
exiting main thread
2
1
exiting child thread
```

SYNCHRONIZED

I thread se non vengono sincronizzati gareggiano uno contro l'altro per completare un metodo

Per forzare gli altri thread ad aspettare che il primo abbia terminato si usa synchronyzed

COMUNICAZIONE TRA I THREAD

All'interno dei metodi synchronized si possono richiamare i metodi:

- ***wait:*** pone un thread in pausa finquando non entra un nuovo thread nel monitor e chiama notify

- ***notify:*** richiama dallo stato di pausa il primo thread che ha chiamato wait nello stesso oggetto

- ***notifyAll:*** richiama dalla pausa tutti i thread che hanno chiamato wait nello stesso oggetto, viene eseguito quello a piu' alta priorita'

Esempio:

```
class Q {
        int n;
        boolean valueSet = false;

        synchronized int get() {
            if ( !valueSet )
            try wait();
            catch(InterruptedException e);
            System.out.println("Got : " + n);
            valueSet = false;
            notify();
            return n;
        }

        synchronized void put( int n ) {
            if (valueSet)
            try wait();
            catch(InterruptedException e);
            this.n= n;
            valueSet = true;
            System.out.println("Put : " + n);
            notify();
        }
} // end_classQ

class Producer implements Runnable {
    Q q:
    Producer(Q q); {
        this.q = q;
        new Thread ( this, "Producer").start();
    }

    public void run () {
        int i = 0;
        while (true) {
            q.put(i++);
        }
    }
} // end_classProducer
```

```
class Consumer implements Runnable {
Q q:
        Consumer(Q q); {
                this.q = q;
                new Thread ( this, "Consumer").start();
        }

        public void run () {
                int i = 0;
                while (true) {
                q.get(i++);
                }
        }
} // end_classConsumer

class PcSynch {
        public static void main( String args[] ) {
                Q q = new Q();
                new Producer(q);
                new Consumer(q);
        }
}
```

```
c:\>java\bin\ java PcSynch
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

JAVA NETWORKING

Java supporta i protocolli TCP/UDP di TCP/IP

I meccanismi di rete vengono forniti dal ***package java.net***

LA CLASSE INETADDRESS

Supporta il meccanismo di naming di Internet

I metodi della classe sono:

- ***getLocalHost:*** restituisce l'oggetto *InetAdddress* che rappresenta la macchina locale
- ***getByName:*** restituisce un oggetto *InetAdddress* accettando come parametro un oggetto hostname
- ***getAllByName:*** restituisce un' array di *InetAdddress* contenente gli indirizzi determinati da un certo nome

Questi metodi restituiscono un'istanza della classe in cui risiedono

Esempio:

```
InetAddress address = InetAddress.getLocalHost();
System.out.printl(address);
address = InetAddress.getByName("mail host");
System.out.println(address);
InetAddress SW[ ];
SW = InetAddress.getAllByName("www.starwave.com");
System.out.println(SW);
```

Vi sono dei metodi che devono essere utilizzati sugli ogggetti della classe InetAddress:

- ***getHostName():*** restituisce una stringa (nome della macchina associato all'oggetto InetAddress)

- ***getAdrress():*** restituisce un array di byte di quattro elementi (indirizzo internet dell'oggetto);

- ***toString():*** restituisce una stringa (nome della machhina e indirizzo IP);

DATAGRAMMI

Sono implementati da Java tramite l'uso di due classi

L'oggetto DatagramPacket rappresenta il contenitore dei dati

DatagramSocket e' il meccanismo usato per spedire o ricevre gli oggetti *DatagramPacket*

DATAGRAMPACKET

Possono essere creati usando i costruttori:

DatagramPacket(byte ibuf[ ] , int ilength)

DatagramPacket(byte ibuf[ ] , int ilength , InetAdddress iaddr , int iport)

I due parametri InetAdddress iaddr , int iport vengono utilizzati da DatagramSocket per determinare dove spedire i dati contenuti nel pacchetto

I metodi di accesso a un DatagramPacket sono:

- ***getAddress():*** restituisce la destinazione InetAddress

- ***getPort():*** restituisce un intero che rappresenta il numero di porta relativo alla destinazione

- ***getData():*** restituisce un'array di byte che rappresentano i dati contenuti nel pacchetto Datagram

- ***getLength():*** restituisce la lunghezza dei dati validi contenuti nell'array di byte a loro volta restituiti dal metodo getData

COMUNICAZIONE CLIENT/SERVER (esempio)

```
import java.net.*;
class WriteServer {
        public static int serverPort = 666;
        public static int clientPort = 999;
        public static int buffer_size = 1024;
        public static DatagramScket ds;
        public static byte buffer[] = new byte[buffer_size];

        public static void TheServer() throws Exception {
                int pos = 0;
                while (true) {
                        int c = System.in.read();
                        switch (c) {
                                case -1:System.out.println("Server Quits.");
                                        return;
                                case '\r' : break;
                                case '\n' : ds.send( new
DatagramPAcket(buffer,pos,InetAddress.getLocalHost(),clientPOrt0));
                                            pos = 0;
                                            break;
                                default   : buffer[pos++] = (byte) c;
                                }
                        }
       }

      public static void TheClient() throws Exception {
       while (true) {
                DatagramPacket p = new DatagramPacket(buffer, buffer.length);
                ds.receive(p);
                System.out.println(new String(p.getData(), 0, 0, p.getLength()));
                    }
      }


       public static void main(String args[]) throws Exception {
        if (args.length == 1) {
                        ds = new DatagramSocket(serverPort);
                        TheServer();
                                }
            else {
                ds = new DatagramSocket(serverPort);
                TheClient();
                }
```

SOCKET

Differentemente dai DaragramSocket, le Socket implementano una connessione persistente altamente attendibile tra Client/Server

Il ServerSocket attende che un Client si connetta ad esso

Una Socket tratta la non disponibilita' di qualcuno a connettersi come una condizione di errore

SOCKET PER CLIENT

Costruttori per il ClientSocket sono:

- ***Socket(String host, int port):*** connette la macchina locale alla macchina e alla porta passate come parametri

- ***Socket(InetAddress address, int port):*** crea una socket a partire da un oggetto preesistente di tipo InetAddress e da una porta

Meto della classe:

- ***getInetAddress():*** restituisce l'InetAddress associato alla socket

- ***getPort():*** restituisce la porta remota su cui e' connessa la Socket

- ***getLocalPort():*** restituisce la porta locale su cui e' connessa la Socket

SOCKET PER SERVER

I server non sono necessariamente macchine, ma sono di fatto programmi in ascolto di qualche programma client locale o remoto per connetterlo sulle porte indicate

I server Socket possiedono un metodo accept, chiamata bloccante, che aspettal'inizio di una comunicazione da parte di un client per poi tornare con un normale Socket

Costruttori per il ServerSocket sono:

- ***ServerSocket(int port):*** crea una socket per un server sulla porta specificata

ServerSocket(int port, int count): crea una socket per un server sulla porta specificata aspettando un tempo dato da count, nel caso in cui la porta sia in uso

LE APPLET

Le applet sono applicazioni messe a disposizione da un internet server;

Vengono trasferite sulla rete, installate automaticamente ed eseguite localmente come parte di un documento Web;

Sul sistema client, l'applet è sottoposta a limitazioni sugli accessi alle risorse di sistema.

ESEMPIO DI APPLET

1) Creare il file con nome ***HelloWorldApplet.java*** con il segegunte codice:

```
import java.awt.*;
import java.applet.*;
public class HelloWorldApplet extends Applet {
        public void main( Graphics g ) {
                g.drawString(" Hello World !!!", 20,20);
        }
}
```

2) Creare il file con nome ***HelloWorldApplet.html*** con il segegunte codice:

```
<applet code="HelloWordApplet" width = 200 height = 40 >
</applet>
```

3) Da linea di comando esguire i seguneti comandi:

c:\>javca\bin\javac HelloWorldApplet.java

c:\>java\bin\appletviewer HelloWorldApplet.html

4) Sullo schermo verrà visulaizzata la seguente schermata:

L'ETICHETTA <APPLET> IN HTML

Per eseguire un applet sia da un doumeno HTML che dal JDK appletviewer viene utilizzata l'etichetta <applet>;

La sintassi dell'etichetta:

```
< APPLET
        [ CODEBASE = codebaseURL ]

        CODE = appletFile

        [ ALT = alternateText ]
        [ NAME = appletInstanceName ]

        WIDTH = pixels    HEIGHT = pixels

        [ ALIGN = alignment ]
        [ VSPACE = pixels ] [ HSPACE = pixels ]
>
        [ <PARAM NAME = AttributeName1 VALUE = AttribureValue1 > ]

        [ <PARAM NAME = AttributeName2 VALUE = AttribureValue2 > ]
        ...

        [ HTML Displayed in the absence of Java ]

</APPLET>
```

Le etichette che sono racchiuse tra le parentesi quadre sono opzionali.

INIZIALIZZAZIONE DI UNA APPLET

Quando si scrive un'applet un programmatore deve ridefinire i metodi nella classe Applet;

Questo richiede la conoscenza dell'ordine di chiamata dei metodi

La lista che segue è l'ordine di chiamata dei metodi della classe Applet:

- ***init:*** e' il primo metodo che viene richiamato e serve per inizializzare le variabili
- ***start:*** viene chiamato dopo init, e come punto di partenza dopo che e' stata bloccata l'esecuzione di un'applet
- ***paint:*** viene chiamato ogni volta che la finestra e' danneggiata, per rispristinare l'applet
- ***update:*** riempe l'applet con il colore di background e dopo chiama paint
- ***stop:*** viene chiamato quando un brower web abbandona la pagina HTML che lo contiene
- ***destroy:*** viene chiamato quando l'applet viene rimossa dalla memoria

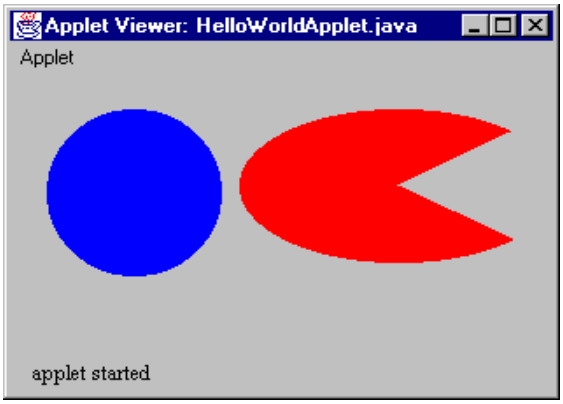
METODI GRAFICI

Gran parte dei metodi grafici fanno parte della classe Graphics

I metodi grafici più utilizzati sono:

- ***drawRect(x1, y1, x2, y2):*** disegna un rettangolo;
- ***fillRect(x1, y1, x2, ,y2):*** colora il rettangolo internamente;
- ***drawoval(x1, y1, x2, y2):*** disegna un cerchio;
- ***fillOval(x1, y1, x2, y2):*** colora il cerchio internamente;
- ***drawLine(x1, y1, x2, y2):*** disegna una linea;
- ***drawArc(x1, y1, x2, y2, startAngle, endAngle):*** disegna un arco;
- ***fillaArc(x1, y1, x2, y2, startAngle, endAngle):*** colora l'arco internamente;
- ***drawPoligon(x[ ], y[ ], pointNum):*** disegna un poligono;
- ***FillPoligon(x[ ], y[ ], pointNum):*** colora poligono internamente;

Esempio:



I COLORI

Java è un ambiente portabile progettato per funzionare su molti tipi di periferiche grafiche

Java cerca la migliore corrispondenza tra il colore richiesto e i limiti imposti del display che si sta utilizzando

Java utilizza la classe ***Color*** per la gestione dei colori

Color possiede una serie di strumenti molto utili per interagire con le informazione relative al colore

È possibile specificare un colore con due semantiche:

- RGB: rosso, verde, blue;
- HSB: tonalitaà, saturazione, luminosità;

Sono definiti una serire di colori statici:

black	white	red	green
blue	cyan	yellow	magenta
orange	pink	gray	darkGray
lightGray			

Per definire un nuovo colore si può utilizzare uno dei tre costruttori seguneti:

- ***Color(int, int, int)***

Questo costruttore di colore considera rosso, verde e blu come interi con valori compresi tra 0 e 255;

Esempio:

```
Color colore1;
colore = new Color( 255, 100, 100);
```

- ***Color(int)***

Questo costruttore di colore accetta un solo parametro intero che contiene informazioni relative ai livelli di rosso, verde, blu. Ogni valore di colore si riferisce ad un intero.

Esempio:

```
Color colore2;
int num = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
colore2 = new Color( num );
```

- ***Color(float, float, float)***

Questo costruttore di colore accetta tre parametri di tipo *float* con valori compreisi tra 0.0 e 1.0 per rosso, verde, blu.

Esempio:

```
Color colore3;
colore2 = new Color( 0.0, 0.4,  1.0 );
```

I FONT

Java fornisce cinque tipi di font:

- Dialog;
- Helvetica;
- TimesRoman;
- Courier;
- Symbol;

Essi fanno parte del set di base dei font utilizzati con PostScript;

L'eccezione è Dialog che è un font utilizzato dal sistema per le scritture sui menù delle finestre;

Dialog è il font di default;

Il costruttore Font crea un nuovo font con specificato il nome, lo stile e l adimensione;

Esempio:

```
Font stringFont;
stringfont = new Font("Helvetica", Font.BOLD | Font.ITALIC, 24 );
```

Vi sono tre variabili statiche per specificare lo stile di un font all'interno di una famiglia:

- Font.PLAIN
- Font.BOLD
- Font.ITALIC

Ecco alcuni metodi da utlizzare con i font:

- ***drawString(String, int x, int y):*** scrive il contenuto di ***String*** sulla finestra di output alla posizione individuata dalle cordinate di ***x,y***;
- ***setFont(font):*** setta il font per u oggetto grafico;
- ***getFont():*** estrae il font da un oggetto grafico;
- ***getfamily():*** restituisce il nome della famiglia del font;
- ***getName():*** restituisce il nome logico del font;
- ***getSize():*** fornisce un intero per la dimensione del font;
- ***getStyle():*** fornisce un intero che rappresenta lo stile del font;

Esempio:

Una procedura che fornisce informazioni circa il font corrente di un oggetto Graphics.

```
public void getfontInfo( Graphics g ) {
        Font f = g.getFont();
        String fontName = f.getFamily();
        int fontSize = f.getSize();
        int fontStyyle = f.getStyle();
        boolean isBold = ((fontStyle & Font.Bold) ==1 )?true:false;
}
```

- Java possiede un certo numero di classi per permettere di mostrare il testo all'interno di un applet;

- La classe FontMetrics permette di ottenere informazioni relative ai font;

- Alcuni dei metodi della classe sono:

- ***stringWidth(string):*** restituisce la lunghezza della stringa specificata con un font;
- ***bytesWidth(string):*** restituisce la larghezza dell'array di byte specificato con il font corrente dell'oggetto graphics;
- ***getAscent():*** distanza dalla *baseline* al margine superiore di un carattere;
- ***getDescent():*** distanza dalla *baseline* al margine inferiore di un carattere;
- ***getHeight():*** dimensione dall'alto al basso del carattere più grosso nei font;
- ***getMaxAscent():*** valore massimo dei bit disegnati per tutti i caratteri;
- ***getMaxDescent():*** valore massimo dei bit disegnati per tutti i caratteri;

(baseline: linea inferiore sulla quale i caratteri sono allineati)

INTERFACCE UTENTE CON AWT

ABSTRACT WINDOW TOOLKIT

Con la sigla AWT si indica l'insieme delle classi che forniscono i meccanismi che permettono di utilizzare il linguaggio java per lo sviluppo di applicazioni multimediali

I metodi implementati nel package sono più di 100.

I metodi servono per generare componenti GUI (Graphical User Interfaces).

COMPONENT

Component è una classe *abstract*, contiene tutti gli attributi di un componente visuale

Tutti gli elementi dell'interfaccia utente che vengono visualizzati sullo schermo e interagiscono con l'utente sono sottoclassi di *Component*

CONTAINER

- La classe *Container* è una sottoclasse abstract di *Component*;

- Ha dei metodi addizionali che permettono ad altri componenti di essere nidificati in essa;

PANEL

- La classe *Panel* è una sottoclasse di *Container*;

- Non è una classe di tipo abstract;

- Si possono aggiungere nuovi componenti a un *Panel* usando il metodo add;

- I nuovi oggetti possono essere posizionati e ridimensionati.

CANVAS

- La classe *Canvas* è una sottoclasse di *Component*;

- Un oggetto canvas è libero da qualsiasi restrizione semantica;

- Si può produrre qualunque aspetto si desideri.

LABEL

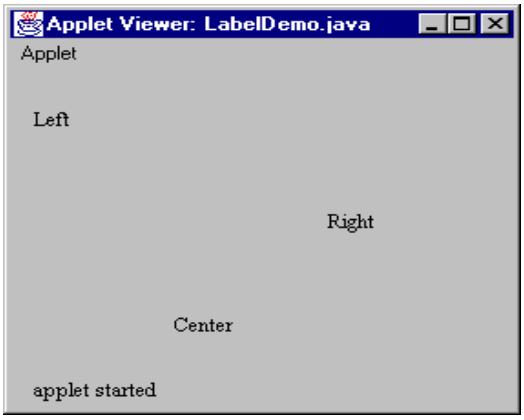
La classe *Label* è una sottoclasse di Canvas

Offre la funzionalità di scegliere come visualizzzare una particolare stringa

Nella classe *label* sono definite tre costanti: LEFT, RIGHT e CENTER

Esempio:

```
import java.awt.*;
import java.applet.*;
public class LabelDemo extends Applet {
        public void init() {
                setLayout(null);
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                Label left = new Label("Left", Label.LEFT);
                Label  right = new Label("Right", Label.RIGHT);
                Label  center = new Label("Center", Label.CENTER);
                add(left);
                add(right);
                add(center);
                left.reshape(0, 0, width, height /3 );
                right.reshape(0, height /3, width, height /3 );
                center.reshape(0, 2 * height /3, width, height /3 );
        }
}
```



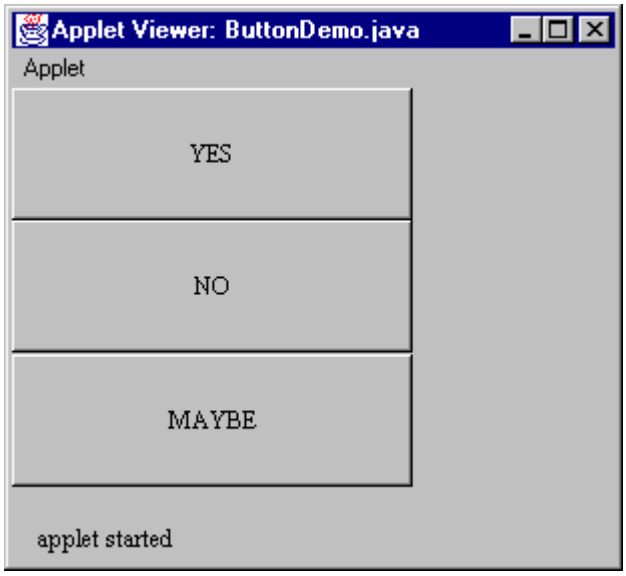
BUTTON

Un *Button* è un componente utilizzabile per invocare qualche azione quando viene premuto;

Un *Button* è identificato da una stringa

Esempio:

```
import java.awt.*;
import java.applet.*;
public class ButtonDemo extends Applet {
        public void init (Graphics g) {
                setLayout( null );
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                Button yes = new Button("YES");
                Button no = new Button("NO");
                Button maybe = new Button("MAYBE");
                add(yes);
                add(no);
                add(maybe);
                yes.reshape(0, 0, width, height /3 );
                no.reshape(0, height /3, width, height /3 );
                maybe.reshape(0, 2 * height /3, width, height /3 );
        }
}
```



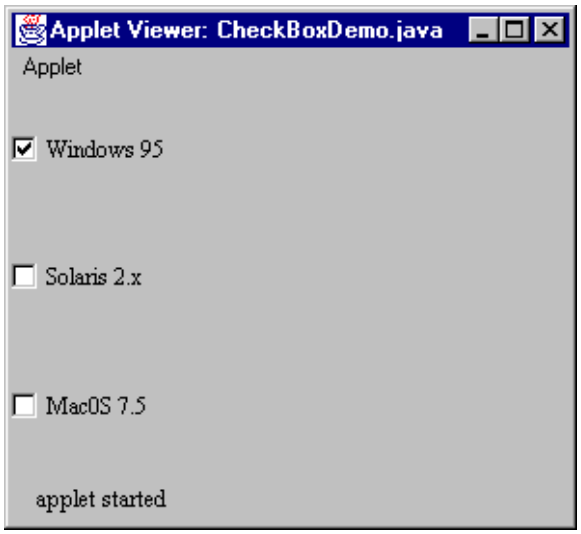
CHECKBOX

La classe *Checkbox* è come una *Label*
Viene utilizzata per selezionare una condizione booleana
Un *Checkbox* è identificato da una stringa
Metodi della classe:

- ***getState:*** per conoscere lo stato corrente del contrassegno
- ***setState:*** per stabilire lo stato del contrassegno

Esempio:

```
import java.awt.*;
import java.applet.*;
public class CheckBoxDemo extends Applet {
        public void init (Graphics g) {
                setLayout( null );
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                Checkbox win95 = new Checkbox("Windows 95", null, true);
                Checkbox solaris = new Checkbox("Solaris 2.x");
                Checkbox mac = new Checkbox("Mac0S 7.5");
                add(win95);
                add(solaris);
                add(mac);
                win95.reshape(0, 0, width, height /3 );
                solaris.reshape(0, height /3, width, height /3 );
                mac.reshape(0, 2 * height /3, width, height /3 );
        }
}
```



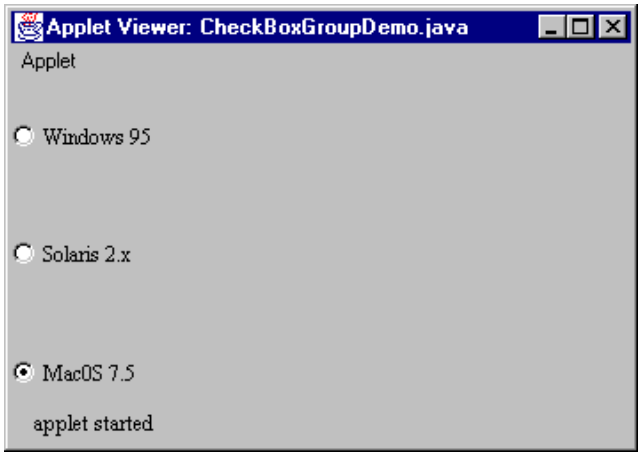
CHECKBOXGROUP

La classe *CheckboxGroup* serve per raggruppare insieme più oggetti checkbox.
Metodi della classe:

- ***getCheckboxGroup:*** per sapere il gruppo a cui un particolare *Checkbox* appartiene
- ***setCheckBoxGroup:*** per stabilire il gruppo a cui fare appartenere un particolare Checkbx
- ***getCurrent:*** per ottenere il *Checkbox* corrente
- ***setCurrent:*** stabilire il *Checkbox* di partenza;

Esempio:

```
import java.awt.*;
import java.applet.*;
public class CheckboxGroupDemo extends Applet {
        public void init () {
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                CheckboxGroup go = new CheckboxGroup();
                Checkbox win95 = new Checkbox("Windows 95", go, true);
                Checkbox solaris = new Checkbox("Solaris 2.x", go, false);
                Checkbox mac = new Checkbox("Mac0S 7.5", go,  false);
                add(win95);
                add(solaris);
                add(mac);
                win95.reshape(0, 0, width, height /3 );
                solaris.reshape(0, height /3, width, height /3 );
                mac.reshape(0, 2 * height /3, width, height /3 );
        }
}
```



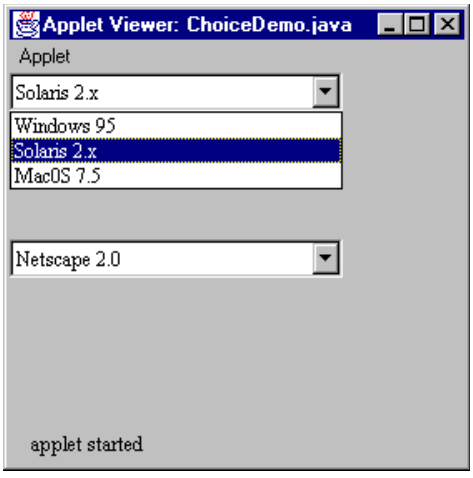
CHOICE

La classe *Choice* viene usata per un menù di selezione di tipo pop-up.
Ogni elemento del menù è una stringa.
Metodi della classe:

- ***addItem:*** per aggiungere un nuove stringhe all'elemento choice
- ***countItems:*** restituisce il numero di elementi di un choice
- ***select:*** per stabiliree l'elemento da selezionare correntemente
- ***getSelectedItem:*** per ottenere l'etichetta dell'elemento attualmente selezionato
- ***getSelectedIndex:*** per ottenere il numero dell'elemento attualmente selezionato

Esempio:

```
import java.awt.*;
import java.applet.*;
public class ChoiceDemo extends Applet {
        public void init () {
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                Choice os = new Choice ();
                Choice browser = new Choice ();
                os.addItem("Windows 95");
                os.addItem ("Solaris 2.x");
                os.addItem("Mac0S 7.5");
                browser.addItem("Netscape 1.1");
                browser.addItem("Netscape 2.0");
                add(os);    add(browser);
                os.reshape(0, 0, width, height /2 );
                browser.reshape(0, height /2, width, height /2 ); }
}
```



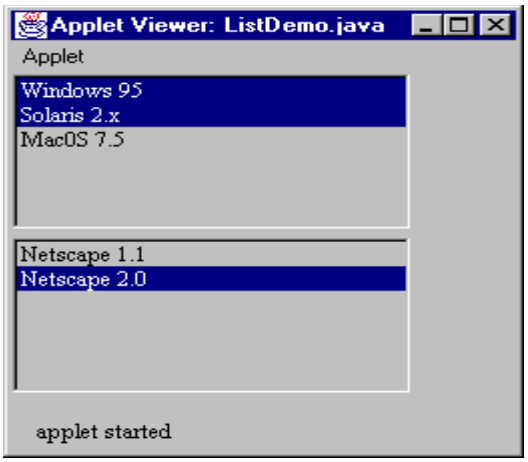
LIST

La classe *List* fornisce una casella di riepilogo a scorrimento, compatta e con possibiltà di scelta multipla

Permette di stabilire più selezioni

Esempio:

```
import java.awt.*;
import java.applet.*;
public class ListDemo extends Applet {
        public void init () {
                setLayout(null);
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                List os = new List(0,true);
                List browser = new List(0,false);
                os.addItem("Windows 95");
                os.addItem ("Solaris 2.x");
                os.addItem("Mac0S 7.5");
                os.select(1);
                browser.addItem("Netscape 1.1");
                browser.addItem("Netscape 2.0");
                add(os);    add(browser);
                os.reshape(0, 0, width, height /2 );
                browser.reshape(0, height /2, width, height /2 );
        }
}
```



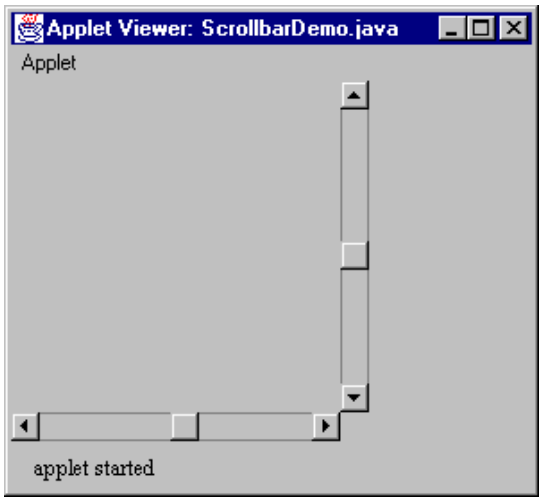
SCROLLBAR

La classe *Scrollabar* viene utilizzata per scegliere valori continui tra un numero minimo e massimo specificati.
La barra di scorrimento ha elementi di controllo sia nell'orientazione verticale che in quela orizzontale.
Metodi della classe:

- ***getMinimum:*** valore minimo di scorrimento
- ***getMaximun****:* valore massimo di scorrimento;
- ***getValue****:* usato per ottenere il valore corrente di scrollbar;
- ***setValue****:* usato per stabilire il valore corrente di scrollbar;

Esempio:

```
import java.awt.*;
import java.applet.*;
public class ScrollbarDemo extends Applet {
        public void init() {
                setLayout(null);
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                Scrollbar hs;
                hs  = new Scrollbar(Scrollbar.HORIZONTAL,50,width/ 10, 0, 100);
                Scrollbar vs;
                vs = new Scrollbar(Scrollbar.VERTICAL,50,  height/ 10, 0, 100);
                add(hs);
                add(vs);
                int thickness = 16;
                hs.reshape(0, height - thickness, width - thickness, thickness);
                vs.reshape(width  - thickness,  0,  thickness, height - thickness);
        }
}
```



TEXTFIELD

La classe *TextField* implementa un'area di ricevimento di testo

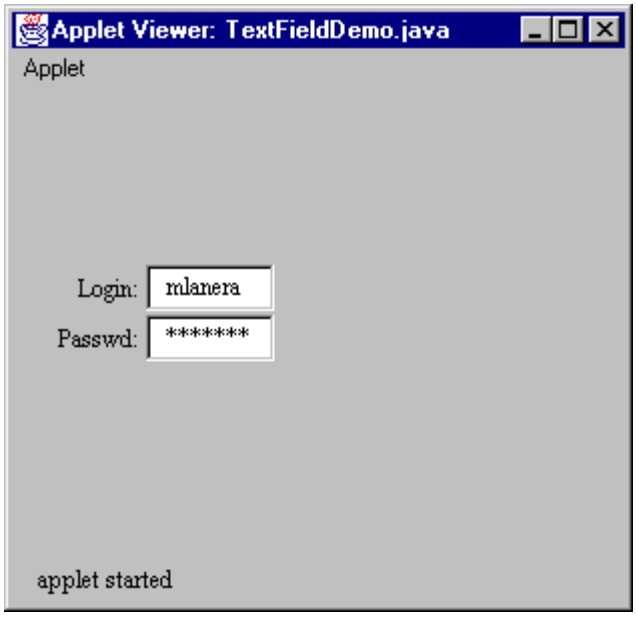
Il testo può essere modificato in diversi modi

Metodi della classe:

- ***setEditable**:* per bloccare il contenuto di un TextField
- ***isEditable**:* per verificare se può essere modificato
- ***getText**:* per conoscere il valore corrente
- ***setText**:* per stabilire il valore di default
- ***select**:* per impostare la selezione di default
- ***selectAll**:* per selezionare l'intero valore
- ***setEchoCharacter**:* stabilisce il carrattere che TextFiled utilizza per la chittografazione
- ***echoCharIsSet**:* per verificare la modalità di funzionamento di un TextField
- ***getEchoChar**:* per conoscere il carattere di chittografazione

Esempio:

```
import java.awt.*;
import java.applet.*;
public class TextFieldDemo extends Applet {
        public void init() {
                setLayout(null);
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                Label namep = new Label("Login: ", Label.RIGHT);
                Label passp = new Label("Passwd: ", Label.RIGHT);
                TextField name = new TextField(8);
                TextField pass = new TextField(8);
                pass.setEchoCharactert('*');
                add(namep);
                add(name);
                add(passp);
                add(pass);
                int space = 25;
                int w1 = width / 3;
                namep.reshape(0, ( height - space) / 2, w1, space);
                name.reshape(w1, ( height - space) / 2, w1, space);
                passp.reshape(0, ( height + space) / 2, w1, space);
                passp.reshape(w1, ( height + space) / 2, w1, space);
        }
}
```



TEXTAREA

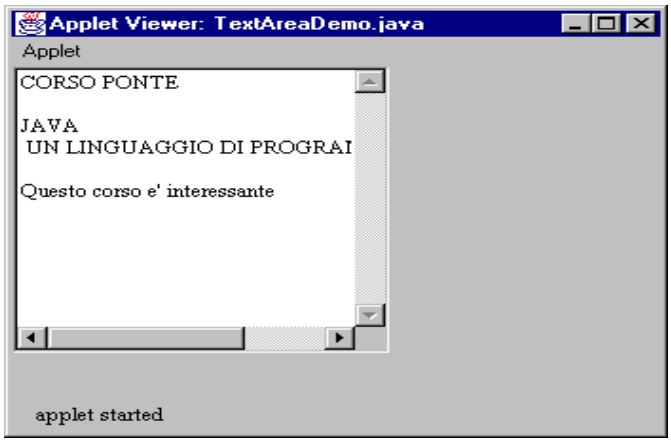
La classe *TextArea* implementa in Java un editor di testo multi-linea

Esistono tre metodi che permettono di modificare il contenuto di una TextArea:

- ***appendText***: aggiungie il parametro *String* alla fine del buffer di testo
- ***insertText***: inserisce il parametro *String* a partire da un dato offset
- ***replaceText***: copia i caratteri del paramtro *String* nel buffer a partire dal primo offset, continuando fino al secondo

Esempio:

```
import java.awt.*;
import java.applet.*;
public class TextAreaDemo extends Applet {
        public void init() {
                setLayout(null);
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                String testo = ' Questo è un esempio !!!';
                TextArea text = new TextArea(testo, 80, 80 );
                add(text);
                text.reshape(0, 0, width , height); }
}
```



EVENT

Ogni componente precedentemente creato può gestire eventi con l'utilizzo di alcuni comodi metodi

Cio viene realizzato attraverso il metodo ***handleEvent***

Il metodo viene chiamato con un'istanza della classe ***Event***

Gli eventi che bisogna sono relativi alla tastiera e al mouse

Gli eventi relativi al mouse sono:

- ***mouseEnter****:* viene invocato quando la prima volta il mouse entra in un componente
- ***mouseMove****:* viene invocato quando il mouse si muovbe in un componente
- ***mouseDown****:* viene invocato quando è premuto un qualunque pulsante del mouse
- ***mouseDrag****:* viene invocato quando il mouse si muove con un pulsante premuto
- ***mouseUp****:* viene invocato quando è rilasciato il pulsante del mouse

Alcuni eventi relativi alla tastiera sono:

- ***keyDown**:* viene invocato quando un tasto è premuto
- ***keyUp**:* viene invocato quando un tasto è rilasciato
- ***shiftDown**:* viene invocato quando è premuto il tasto shift
- ***controlDown**:* viene invocato quando è premuto il tasto control

Per trattare eventi particolari come Button, Scrollbar e Menu si deve ricorrere al metodo *action*

Esempio:

```
import java.awt.*;
import java.applet.*;
public class EventButton extends Applet {
        public void init (Graphics g) {
                setLayout( null );
                int width = Integer.parseInt( getParameter("width"));
                int height = Integer.parseInt( getParameter("height"));
                Button yes = new Button("YES");
                Button no = new Button("NO");
                Button maybe = new Button("MAYBE");
                add(yes);
                add(no);
                add(maybe);
                yes.reshape(0, 0, width, height /3 );
                no.reshape(0, height /3, width, height /3 );
                maybe.reshape(0, 2 * height /3, width, height /3 );
        }

        public boolean action(Event e, Object o) {
                if ( e.target.equals(yes) )
                        drawString("Bottone 1");
        return true;
        }
}
```