Programmazione

Elementi di programmazione in Java

Rosario Culmone

rosario.culmone@unicam.it

UNICAM

Presentazione del modulo Java

Il modulo Java fa parte del corso "Programmazione" [ST0851]

- 6 CFU
- 42 ore di lezione (7 ore per CFU). Rapporto ore di lezione frontale su ore di studio individuale 7/18 ovvero almeno due ore di studio a casa per ogni ora di lezione!
- uso di computer per svolgimento di esercizi ed esempi
- libri di testo: qualsiasi manuale di Java. Uno dei piú semplici é "Introduzione alla programmazione con il linguaggio Java" di Stefano Mizzaro, Franco Angeli, ISBN: 88-464-1696-1
- Un libro di esercizi: P. Coppola e S. Mizzaro. "Laboratorio di programmazione in Java". Apogeo, ISBN: 88-503-2145-7.
- dispense: lucidi delle lezioni e materiale disponibile in rete `http://docenti.unicam.it/pdett.aspxUteId=204&IDPADRE=593&tv=m&ru=RU`

Lessico, sintassi e semantica

Ogni linguaggio, sia esso naturale o artificiale, ha i seguenti aspetti:

lessico il dizionario dei termini

sintassi la giusta sequenza dei termini in una frase

semantica il significato di un testo

Ambiguitá

Una importante caratteristica dei linguaggi é l'ambiguitá semantica ovvero diverso significato per la stessa frase. I linguaggi si possono classificare in:

Linguaggi naturali ad ogni frase sintatticamente corretta possono corrispondere piú significati.

Linguaggi di programmazione ad ogni frase sintatticamente corretta corrisponde un solo significato semantico.

Linguaggi dichiarativi e procedurali

Linguaggi dichiarativi , esprimono cosa si vuole ma non come si ottiene

Linguaggi procedurali , esprimo cosa si vuole mediante passi elementari. Un programma é espresso come istanza di un algoritmo

Per algoritmo intendiamo un metodo per la soluzione di un problema adatto a essere codificato sotto forma di programma. Piú precisamente si puó dire che un algoritmo é una *"sequenza logica di istruzioni elementari (univocamente interpretabili) che, eseguite in un ordine stabilito, permettono la soluzione di un problema in un numero finito di passi*.

Dalla definizione si puó desumere che:

- la sequenza di istruzioni deve essere finita;
- essa deve portare ad un risultato;
- le istruzioni devono essere eseguibili materialmente;
- le istruzioni devono essere espresse in modo non ambiguo.

Realtá e modello

Per sapere dopo quanto tempo una una mela lasciata cadere da 30 metri di altezza tocca il suolo si possono seguire due strade:

- trovare una mela, salire su un palazzo di 10 piani e misurare il tempo da quando la lasciamo a quando tocca il suole con un cronometro (sono inevitabili errori!)
- realizzare un modello (matematico) con i dati salienti (non serve il colore della mela) e applicarlo al caso specifico.

I programmi sono modelli (talvolta estremamente complessi) della realtá o della fantasia.

Problem solving

la Soluzione del problema o come viene piu frequentemente definito Problem solving (dall'inglese), termine che indica l'insieme dei *processi* per *analizzare*, *affrontare* e *risolvere* positivamente situazioni problematiche

é un'attivitá del *pensiero* che un *organismo* o un *dispositivo di intelligenza artificiale* mette in atto per raggiungere una condizione desiderata a partire da una condizione data

Approccio intuitivo

L'approccio scientifico alla risoluzione dei problemi inizialmente era sviluppata secondo uno schema puramente intuitivo:

- percezione dell'esistenza di un problema
- definizione del problema
- analisi del problema e divisione in sottoproblemi
- formulazione di ipotesi per la risoluzione del problema
- verifica della validitá delle ipotesi
- valutazione delle soluzioni
- applicazione della soluzione migliore

Tecniche

Focalizzare	Creare un elenco di problemi	Selezionare il problema
	Verificare e definire il problema	Descrizione scritta del problema
Analizzare	Decidere cosa é necessario sapere	Raccogliere i dati di riferimento
	Determinare i fattori rilevanti	Valori di riferimento
	Elenco dei fattori critici	
Risolvere	Generare soluzioni alternative	Selezionare una soluzione
	Sviluppare un piano di attuazione	Scelta della soluzione del problema
	Piano di attuazione	
Eseguire	Impegnarsi al risultato aspettato	Eseguire il piano
	Controllare le conseguenze	Impegno organizzativo
	Completare il Piano	Valutazione finale

Rasoio di Occam

Il *rasoio di Occam* é il nome con cui viene contraddistinto un principio metodologico espresso nel XIV secolo dal filosofo e frate francescano inglese William of Ockham.

"A paritá di fattori la spiegazione piú semplice tende ad essere quella esatta"

La formula utilizzabile nell'ambito del problem solving puó essere:

"Non moltiplicare gli elementi piú del necessario" oppure

"Non considerare la pluralitá se non é necessario" oppure

"inutile fare con piú ció che si puó fare con meno"

Applicato al comportamento umano diventa la legge di Murphy di Robert J. Hanlon:

"Non attribuire a cattiveria ció che puoi facilmente spiegare con la stupiditá"

Caratteristiche dei linguaggi di programmazione

- *Espressivitá*: la semplicitá con cui si puó scrivere un algoritmo.
- *Didattica*: la rapiditá con cui lo si puó imparare.
- *Leggibilitá*: la facilitá con cui, leggendo un codice, si puó capire cosa fa.
- *Robustezza*: é la capacitá del linguaggio di prevenire, nei limiti del possibile, gli errori di programmazione.
- *Modularitá*: la possibilitá di produrre pezzi autonomi di programma.
- *Flessibilitá*: la possibilitá di adattare il linguaggio, estendendolo con la definizione di nuovi comandi e nuovi operatori.
- *Generalitá*: la facilitá con cui il linguaggio si presta a codificare algoritmi e soluzioni di problemi in campi diversi.
- *Efficienza*: la velocitá di esecuzione e l'uso oculato delle risorse del sistema su cui il programma é eseguito.
- *Coerenza*: l'applicazione dei principi base di un linguaggio in modo uniforme in tutte le sue parti.

Quanti sono i linguaggi di programmazione

Ad oggi (2011) sono stati creati 8500 linguaggi di programmazione, circa 2400 sono stati sviluppati negli Stati Uniti, 600 nel Regno Unito, 160 in Canada e 75 in Australia.

Linguaggi di programmazione

Uso dei linguaggi di programmazione aggiornato a settembre 2014 secondo

TIOBE Software http://www.tiobe.com

N	Linguaggio	%	Variazione in un anno
1	C	16.721%	-0.25%
2	Java	14.140%	-2.01%
3	Objective-C	9.935%	+1.37%
4	C++	4.674%	-3.99%
5	c#	4.352%	-1.21%
6	Basic	3.547%	-1.29%
13	Ruby	1.281%	-0.10%

Albero dei linguaggi di programmazione

Storia di Java

- Smaltalk sviluppato negli anni ’70 principalmente da Alan Kay é il primo linguaggio ad oggetti.
- Java é un linguaggio linguaggio di programmazione orientato agli oggetti creato da James Gosling nel 1991. Fu annunciato da Sun Microsystems ufficialmente il 23 maggio 1995.
- Il 13 novembre 2006 la Sun Microsystems ha rilasciato la sua implementazione del compilatore Java e della macchina virtuale sotto licenza GPL.
- L’8 maggio 2007 SUN ha rilasciato anche le librerie sotto licenza GPL rendendo Java un linguaggio di programmazione la cui implementazione di riferimento é libera.
- Ad oggi l’ultima versione indicata con Java 8 é stata rilasciata il 18 marzo 2014.

Dialetti Java

- Yukihiro Matsumoto nel 1993 annuncia Ruby. E' un linguaggio fortemente ispirato a Smalltalk e non ha la "verbositá" di Java. Esiste Jruby per JVM. La versione piú recente 1.9.2 é del 18 agosto 2010 http://www.ruby-lang.org
- Nel 2003 James Strachan pensó ad un linguaggio che potesse direttamente interagire con il bytecode di Java. La versione piú recente 1.8 di Groovy é del 27 aprile 2011 http://groovy.codehaus.org
- Linguaggio di script progettato per Netscape, uno dei primi browser, nel 1996. Con il nome ECMAScript é standard ISO/IEC 16262.

Caratteristiche di Java

- é un linguaggio che si presta ad essere come primo linguaggio di programmazione da insegnare
- la sintassi é simile al linguaggio C, linguaggio molto diffuso in ambito industriale
- linguaggio diffuso per la realizzazione di applicazioni di rete (lato server, lato client).
- Linguaggio orientato agli oggetti non "puro". Il progenitore di Java é Smaltalk (puro)
- Compilazione con generazione di codice intermedio ByteCode e successiva interpretazione mediante macchina virtuale (Java Virtual Machine).
- Dymamic binding. Recupero delle librerie in fase di esecuzione con conseguente ByteCode compatto (ma possibili errori in esecuzione)
- Multipiattaforma. Il ByteCode puó essere interpretato su hardware e sistemi operativi diversi (basta avere per ogni hardware una JVM)

Traduzione, compilazione e interpretazione

Traduttore, strumento che traduce un testo in una lingua in un’altra presenvandone la semantica.

Per i linguaggi di programmazione, il traduttore si chiama compilatore e il processo di traduzione si chiama compilazione.

Lo strumento che esegue le azioni semantiche si chiama interprete. L’esecuzione di un programma puó essere svolta da un interprete o direttamente dall’hardware (microprocessore)

```
> edit Hello.java
> javac Hello.java
> java Hello
```

Strumenti

BlueJ Strumento per scrivere, compilare ed eseguire programmi Java. Comprende un descrittore grafico della struttura. http://www.bluej.org/

Eclipse Le stesse unzionalitá di bluej ma piú completo ed con possibilitá di funzionalitá aggiuntive. http://www.eclipse.org/

NetBeans Prodotto distribuito gratuitamente da Sun Microsystems e basato sulle librerie NetBeans. http://netbeans.org/

JBuilder Prodotto commerciale prodotto dalla Borland. http://www.embarcadero.com/products/jbuilder

EJE Editore Open Source Italiano pensato per chi deve imparare Java http://sourceforge.net/projects/eje/

BlueJ

Lo strumento che utilizziamo per scrivere ed eseguire i programma si chiama

BlueJ

Da dove scaricarlo http://www.bluej.org/

Note per l'installazione

http://www.javaportal.it/rw/19651/13381/25849/24607/editorial.html

Tutorial anche in italiano su http://www.bluej.org/doc/tutorial.html

Manuale aggiornato all'ultima versione in inglese:

"Objects First with Java. A Practical Introduction using BlueJ" di David J. Barnes e Michael Kölling, Fifth edition, Prentice Hall / Pearson Education, 2012, ISBN 978-013-283554-1

Manuale in italiano:

"Programmare in Java con Bluej. Introduzione alla programmazione a oggetti" di David J. Barnes, Michael Kölling, Pearson Education Italia, 2003, ISBN: 9-78-88719218-77

Primo programma

```
class Hello {
  public static void main(String args []) {
    System.out.println("Hello, world!");
  }
}
```

L'impaginazione del programma non ha alcun effetto sul significato

```
class             Hello {
  public
         static
void main ( String args [      ]   )     {
    System.out.println("Hello, world!");             }           }
```

I precedenti programmi sono identici, il primo é umanamente piú leggibile. Per facilitare la lettura, le parole in blu sono *parole chiave* quelle in arancione sono scelte dal *programmatore*.

Errori

Quando si scrive un programma si possono introdurre errori che impediscono la successiva fase di interpretazione (poiché il testo é incomprensibile). Gli errori possono essere:

- Sintattici. Sono errori che riguardano l'ortografia del testo, ad esempio la mancanza di un `;`, l'uso errato di costrutti.
- Errori a *run time*. Sono errori che si presentano in fase di esecuzione del programma e che sono legati ai dati forniti (o non forniti) dall'utilizzatore del programma. Sono errori insidiosi perché possono presentarsi saltuariamente e apparentemente senza una logica.
- Errori logici. Sono gli errori piú difficili da scoprire perché presentano una apparente dicotomia tra la sintassi (quello che ci sembra di aver scritto) e la semantica (quello che vorremo che il programma facesse). Questi errori richiedono per essere scoperti una conoscenza approfondita del linguaggio di programmazione.

Descrivere un problema

Un problema prima deve essere descritto poi si cerca la soluzione. Esistono linguaggi per descrivere il problema e linguaggi per la soluzione del problema.

- Descrivere risponde alla domanda cosa fa
- Risolvere risponde alla domanda come fa

Descrizione funzionale

Quando si vuole realizzare un programma si dovrebbe prima specificare cosa si vuole ottenere ed i vincoli. Ad esempio si vuole realizzare una programma per il calcolo della radice quadrata, indipendentemente dall'algoritmo che si realizzerá mediante codice si puó subito dire che:

- il parametro su cui si applica la radice quadrata deve essere positivo
- il valore prodotto moltiplicato per se stesso deve essere uguale al parametro

La prima é un precondizione, la seconda é una postcondizione

Ogni dichiarazione di classe potrebbe essere anticipata dalla *specifica funzionale*, ovvero di cosa si vuole realizzare senza dire come deve essere realizzata. La specifica funzionale puó essere "tratteggiata" con l'uso degli `assert`.

pre, post e invarianti

In java esiste una parola chiave `assert` che permette di verificare condizioni

```
class TestAssert {
  public static void main(String [] args) {
      double a = 3.14, b = -3.14;
      System.out.println(sqrt(a)); System.out.println(sqrt(b));
  }
  private double sqrt(double p) {
    double r; // variabile su cui verr\'a posto il risultato
    assert p >= 0 : "Parametro negativo"; // pre
    /* ... */
    assert r * r == p : "Calcolo errato"; // post
    return r;
  }
}
```

Java assert

La sinatassi del comando `asset` é la seguente

`assert espressione_boolena;` oppure

`assert espressione_booleana : stringa;`

```
assert 0 <= valore;
assert 0 <= valore : "valore deve essere positivo " + valore;
assert ref != null;
assert ref != null : "il riferimento deve essere diverso da null";
assert newCount == (oldCount + 1);
assert oggetto.funzione(parametro1, parametro2);
  // funzione deve produrre un valore booleano
```

Uso di assert

Per un corretto uso di `assert` bisogna che:

- Tutti i metodi devono avere *pre*, *post* e *inv* solo su parametri e attributi
- Le precondizioni devono essere localizzate subito dopo la dichiarazione del metodo
- Le postcondizioni subito prima del valore prodotto
- Le invarianti in qualsiasi posto all'interno del metodo
- Le asserzioni possono essere sostituite da eccezioni solo se é previsto un uso corretto. Ricordarsi che la rottura di una asserzione implica la terminazione del programma
- Le asserzioni devono specificare il corretto stato di un oggetto
- le asserzioni possono essere definite prima della codifica degli algoritmi (programmazione per contratto).
- Le asserzioni sono uno strumento da utilizzare nella fase di sviluppo del programma. DEVONO essere rimosse o disattivate a fine sviluppo.

Altro esempio

Se si modella una persona con

- *nome* Nome e cognome devono essere stabiliti al momento della creazione e non possono essere modificati nel corso della vita
- *cognome*
- *sesso* Stabilito al momento della nascita e non piú modificabile (?)
- *data di nascita* La data di nascita deve essere non puó essere precedente alla dato odierna ovvero non é possibile far nascere nel passato e non puó essere modificata successivamente alla creazione
- *stato civile* Si nasce sempre nubili o celibi a seconda se si é femmine o maschi e successivamente le modifiche devono tenere conto della morte del coniuge del divorzio dei matrimoni ecc.
- *etá* E' una invariante poiché deve essere sempre un valore positivo ed é calato sottraendo alla data odierna la data di nascita. Qualsiasi calcolo sull'etá non deve mai produrre valori negativi.

attivazione e disattivazione degli assert

- `java -ea programma` attiva le asserzioni
- `java -da programma` disattiva le asserzioni
- E’ possibile attivare o disattivare le asserzioni su specifiche classi o pacchetti. Ad esempio `java -ea X -da Y Z` esegue `Z` attivando le asserzioni di `X` e disattivando quelle di `Y`

Macchina Virtuale Java

Per conoscere le caratteristiche della JVM (Java Virtual Machine) si puó scrivere ed eseguire il seguente programma

```
import java.util.*;
public class SysProperties {
   public static void main(String[] a) {
      Properties sysProps = System.getProperties();
      sysProps.list(System.out);
   }
}
```

A video sono visualizzate molte informazioni tra le quali molto importante é:
`java.version=1.5.0_19` ad essa sono legate alcune caratteristiche del linguaggio.

Tipi primitivi

I programmi manipolano dati. I dati per essere trattati devono essere caratterizzati.

La caratterizzazione nei linguaggi di programmazione si indica con *tipizzazione*.

Ogni data ha un tipo. In Java vi sono due gruppi di tipi:

- Tipi primitivi. I tipi primitivi sono tipi per cui il linguaggio fornisce gli strumenti per la loro gestione.
- Tipi definiti dall'utente. I tipi definiti dall'utente (classi) sono tipi per cui é l'utente a definire le modalitá con devono essere gestiti (operazioni).

Range dei tipi primitivi

Tipi numerici *integrali* `byte`, `short`, `int`, `long`, *reali* `float`, `double` e *enumerativi* `char`, `boolean`

tipo	byte	MIN_VALUE	MAX_VALUE	costanti
`byte`	1	-128	127	`0, 123, \xF1`
`short`	2	-32768	32767	`13, 18456, -23898`
`int`	4	-2147483648	2147483647	`142567128`
`long`	8	-9 $\cdot 10^{18}$	9 $\cdot 10^{18}$	`123456789123456L`
`float`	4	-3.4 $\cdot 10^{38}$	3.4 $\cdot 10^{38}$	`3.1415F`
`double`	8	-1.7 $\cdot 10^{308}$	1.7 $\cdot 10^{308}$	`3.1415, 0.5e+2, -16.3e-5`
`char`	2	0	65536	`'a', '\n', '\u1F87'`
`boolean`	1			`true, false`

Limiti

Poiché la matematica dei computer é *finita* i calcoli possono produrre valori che non hanno rappresentazione interna ovvero:

- molto grandi `Float.MAX_VALUE * Float.MAX_VALUE` $\longrightarrow$ `Float.POSITIVE_INFINITY`
- molto piccoli `-1.0 / Double.MAX_VALUE` $\longrightarrow$ `Double.NEGATIVE_INFINITY`
- indefiniti `0.0 / 0.0` $\longrightarrow$ `Double.NaN`, `Math.sqrt(-2.0)` $\longrightarrow$ `Double.NaN`

Variabili

E' possibile utilizzare etichette con un nome associate a valori (come alias). Le etichette devono essere associate ad un tipo ed é possibile modificare l'associazione a valori (purché dello stesso tipo, o "quasi"). Tali etichette si chiamano *variabili* poiché é possibile cambiare l'associazione etichetta-valore

```
int numero;
float altezza;
boolean coniugato;
```

Le etichette `numero`, `coniugato` e `coniugato` non hanno associato, per adesso, alcun valore. Tale operazione di chiama *dichiarazione* di variabile.

I nomi delle etichette hanno significato per chi scrive il programma e difficilmente hanno semantica universale. L'etichetta, come altri nomi che sono definiti dal programmatore, si chiamano identificatori.

Operatore di assegnamento

L’operatore di *assegnamento* permette di cambiare l’associazione dell’etichetta al suo valore.

```
int numero;
float altezza;
boolean coniugato;
numero = 10;
altezza = 1.85;
coniugato = true;
```

L’assegnamento ovvero il simbolo = (da non confondere con l’operatore di uguaglianza ==). A sinistra dell’operatore di assegnamento c’é sempre una variabile. A destra dell’operatore di assegnamento vi puó essere o una espressione o una variabile.

Dichiarazione con inizializzazione

E' possibile scrivere una frase che comprenda la dichiarazione della variabile e il valore iniziale associato

```
int numero = 10;
float altezza = 1.85;
boolean coniugato = true;
```

Operazioni di assegnamento;

```
numero = 20;
altezza = 1.90;
coniugato = false;
```

Il valore associato a `numero` da `10` diventa `20`.

Dereferenziazione

Per utilizzare il valore associato ad una variabile basta menzionarla;

```
float farina = 3.4;
float pesoTotale;
pesoTotale = farina + 0.5;
```

Alla variabile `pesoTotale` é associato il valore associato alla variabile `farina` sommato a `0.5`.

Costanti

Talvolta é utile usare costanti, ovvero etichette a cui é assegnato un valore che non puó essere successivamente modificato.

```
final double  piGreco = 3.1415;
double raggio = 15.6;
double area = raggio * raggio * piGreco;
piGreco = 6.28; // errore
```

La parola chiave `final` davanti ad una dichiarazione di variabile obbliga ad una inizializzazione ed impedisce successivamente di utilizzare la variabile a sinistra dell'operatore di assegnamento.

Commenti

Alcune frasi introdotte in un programma possono essere ignorate da parte dell'interprete. Queste frasi si chiamano *commenti*

```
float farina = 3.4; // peso della farina prima del trattamento
float pesoTotale = farina + 0.5; /* peso della farina dopo
                                    aver aggiunto l'acqua */
```

Vi sono diversi modi di introdurre commenti all'interno di un programma:

- commenti di linea. Iniziano con `//e terminano a fine riga`
- commenti multilinea. Inizia con `/*e terminano con */`
- commenti documentabili. Iniziano con `/**` e terminano con `*/`. All'interno del commento si usano delle etichette speciali per estrapolare con uno strumento `javadoc`, documentazione.

Operatori sui tipi numerici

tipo	simbolo	operazione	esempio
`float, double`	+	somma	`4.5e01 + 5.3e0`
	-	sottrazione	`6.56e02 - 5.7e01`
	*	moltiplicazione	`7.03 * 3.0e0`
	/	divisione	`3.9 / 2.8`
`byte, short, int, long`	+	somma	`45 + 5`
	-	sottrazione	`657 - 57`
	*	moltiplicazione	`70 * 3`
	/	divisione	`10 / 3`
	%	resto	`10 % 3`

Operatori sui tipi enumerativi

tipo	simbolo	operazione	esempio
`boolean`	`&&`	and	`true && false`
	||	or	`true || false`
	!	not	`!false`
	^	xor	`false ^ true`

Operatori di confronto

tipo	simbolo	operazione	esempio
`byte, short, int, long`	`<`	minore	`12 < 6`
`float, double, char`	`>`	maggiore	`2.0 > 2.7`
	`==`	uguale	`'a' == 'b'`
	`!=`	diverso	`5 != 6`
	`>=`	maggiore o uguale	`3.6 >= 2.6`
	`<=`	minore o uguale	`3.6 <= 2.6`

L'unico operatore ternario `exp_B ? exp_T : exp_T` dove `exp_B` é un'espressione booleana e `exp_T` sono espressioni dello stesso tipo. Ad esempio l'espressione `3 > 2 ? 3.14 : 6.28` produce `3.14`.

Operatori numerici compatti

tipo	simbolo	operazione	esempio	equivalente
byte, short, int, long	+=	incremento	x+=6	x=x+6
float, double	-=	decremento	x-=2	x=x-2
	/=	diviso per	x/=2	x=x/2
	*=	moltiplicato per	x*=3	x=x*3
	%=	diviso per	x%=3	x=x%3
byte, short, int, long	var++	auto incremento	x++	x=x+1
	var--	auto decremento	x--	x=x-1
	++var	auto incremento	++x	x=x+1
	--var	auto decremento	--x	x=x-1

Operatori orientati al byte

Operatori orientati al byte e nella forma compatta

tipo	simbolo	operazione	esempio
`byte, short, int, long`	`<<`	scorrimento a sinistra	`x << 3`
	`>>`	scorrimento a destra	`x >> 3`
	`>>>`	scorrimento a destra senza segno	`x >>> 3`
`byte, short, int, long`	`=<<`	scorrimento a sinistra	`x =<< 3`
	`=>>`	scorrimento a destra	`x =>> 3`
	`=>>>`	scorrimento a destra senza segno	`x =>>> 3`

Operatore di assegnamento ed espressioni

L'operatore = (operatore di assegnamento) puó essere utilizzato in due contesti:

- Comando di assegnamento. Quando é usato seguito da `;`. Ad esempio `a = 3;` l'effetto é l'assegnamento del valore 3 alla variabile `a`.
- Espressione di assegnamento. Quando é usato all'interno di un'espressione. Ad esempio `(a=3)+2` l'effetto é la modifica della variabile `a` con il valore 3 e il valore di 3 é sommato a 2 producendo come valore finale 5

L'uso dell'operatore di assegnamento in espressioni é spesso usato in concomitanza al comando di assegnamento :

```
int a = 3, b, c;
b = c = a + 2; /* da interpretare come (b = (c = (a + 2));
                  b, c hanno ambedue assegnato il valore di 5 */
```

AreaCerchio

- Dichiarazione di variabile reale con inizializzazione a valore costante.
- Dichiarazione e assegnamento ad una variabile del calore di una espressione reale.

```
class AreaCerchio {
  public static void main(String args []) {
    double raggio = 3.5;
    double area = raggio * raggio * 3.14; // 3.14 \UTF{00E8} una costante
    System.out.println(area);
  }
}
```

Media

- Dichiarazione e inizializzazione di quattro variabili intere.
- Dichiarazione e inizializzazione di una variabile intera.
- Assegnamento ad una variabile del calore di una espressione intera.
- Stampa del valore di una variabile intera.

```
class Media {
  public static void main(String args []) {
    int a = 1, b = 3, c = 9, d = 20;
    int media = 0; /* Inizialmente media \UTF{00E8} posta a 0 */
    media = (a + b + c + d) / 4;
    System.out.println(media);
  }
}
```

Confronti

- Dichiarazione e inizializzazione di quattro variabili intere.
- Dichiarazione e inizializzazione di una variabile booleana con il valore di una espressione booleana.
- Assegnamento ad una variabile booleana del valore di una espressione boolena.
- Stampa di due variabili booleane.

```
class Confronti {
  public static void main(String args []) {
    int a = -20, b = 30, c = 0, d = 17;
    boolean primo = a > b;
    boolean secondo = c < d;
    System.out.println(primo);
    System.out.println(secondo);
  }
}
```

Costanti

- Dichiarazione e inizializzazione di una variabile reale e dichiarazione di una variabile reale.
- Dichiarazione di una costante intera.
- Assegnamento ad una variabile reale del valore di una espressione reale.
- Stampa del valore di una variabile reale.

```
class Costanti {
  public static void main(String args []) {
    double celsius = 18.0, fahrenheit; //inizializzare sempre!
    final int costante = 32;
    fahrenheit = (celsius * 9/5.0) + costante; //divisione reale
    System.out.println(fahrenheit);
  }
}
```

Lettura

- Creazione di un canale di lettura da tastiera.
- Scrittura di una stringa costante .
- Lettura da tastiera di una stringa e di un intero.
- Scrittura della stringa e dell'intero letto.

```
import java.util.*;
class Lettura{
  public static void main(String args []) {
    Scanner tastiera = new Scanner(System.in);
    System.out.print("Nome: ");
    String nome = tastiera.next();
    System.out.print("Et\UTF{00E0}: ");
    int et\UTF{00E0} = tastiera.nextInt();
    System.out.print(nome); System.out.println(et\UTF{00E0});
  }
}
```

Blocco di comandi

Spesso si vuole che vengano eseguiti un blocco di comandi la dove é prevista l'esecuzione di un comando. In questi casi é sempre possibile utilizzare il costrutto di *blocco*, ovvero

{ *comandi* }

```
int a = 10, b = 20, c;
{ c = a + b; b = c - a; }
```

Tutte le iscruzioni contenute in un blocco sono eseguite come se fosse un'unica istruzione (o tutte o nessuna)

Blocchi e dichiarazioni

Se un blocco contiene dichiarazioni, la visibilitá delle variabili é permessa a tutti i blocchi interni ma non viceversa.

```
{
  int a = 10, b = 20;
 {
    int c = 30, d = 40;
    c = a + d;
  }
  a = b;
  c = a; // errato
}
```

Controllo del flusso

I costrutti per il controllo del flusso modificano la sequenza dei comandi da eseguire. I costrutti Java per questo scopo sono: (dove `Eb - espressione booleana, C - comando, D - dichiarazione, Ei - espressione intera, Co - costante)`

- `if (Eb) C; permette di eseguire o no un comando`
- `if (Eb) C; else C; seleziona l'esecuzione tra due comandi`
- `while (Eb) C; ripete l'esecuzione di un comando`
- `do C while(Eb); ripete l'esecuzione di un comando`
- `for(D;Eb;C) C; ripete un certo numero di volte l'esecuzione di un comando`
- `switch(Ei) C; sceglie tra un insieme di comandi`
- `case Co:C; comando etichettato`
- `default:C; comando da eseguire se altrimenti non selezionato`
- `break; interrompe l'esecuzione ciclica di comandi`
- `continue; salta i comandi che lo seguono in un ciclo`

if then

```
class IfThen{
  public static void main(String args [])  throws IOException {
    System.out.println("m - maschio, f - femmina");
    char mf = (char) System.in.read();
    if (mf == 'm') System.out.println("Maschio");
    System.out.println("Non maschio");
    /* Non \UTF{00E8} detto che si sia impostato
        f per femmina */
  }
}
```

L'uso generico é il seguente:

if (espressione booleana) comando

Come funziona if

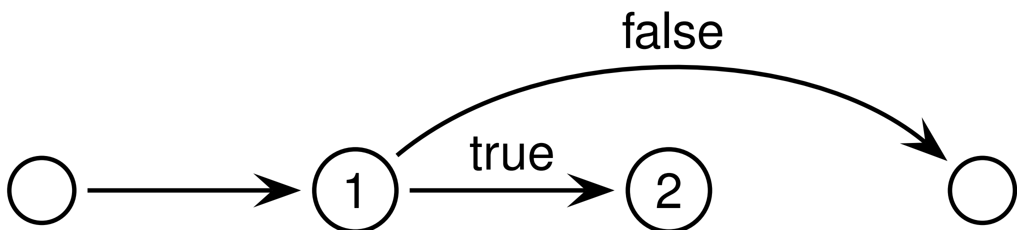
Il costrutto `if` puó essere sintetizzato in:

```
if ( /*1*/ ) /*2*/
```

ad esempio

```
int i = 10, j = 20;
if ( i < j ) System.out.println(i);
```

dove



if then else

```
class IfThenElse{
  public static void main(String args []) {
    int a = 10;
    final int b = 5;
    if (a > b) System.out.println("a > b");
    else System.out.println("a <= b");
    System.out.println("controllo effettuato");
  }
}
```

L’uso generico é il seguente:

if (espressione booleana) comando else comando

Come funziona if else

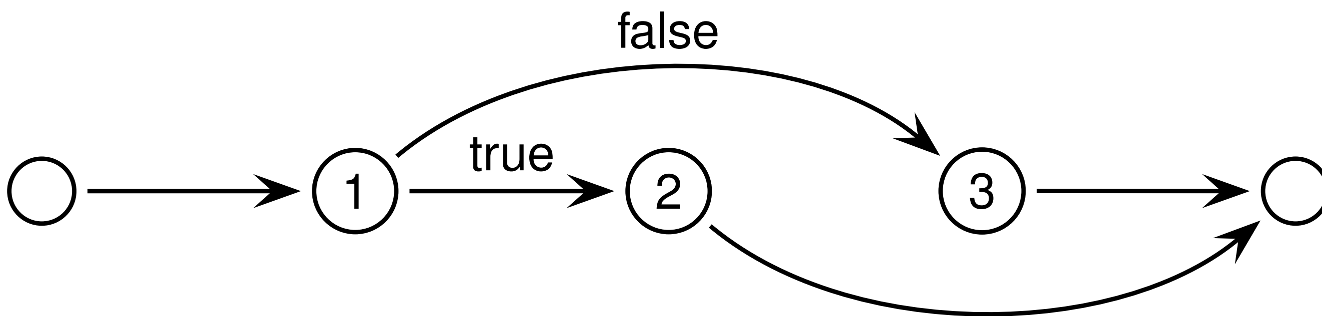
Il costrutto `if else` puó essere sintetizzato in:

```
if ( /*1*/ ) /*2*/ else /*3*/
```

ad esempio

```
int i = 10, j = 20;
if ( i > j ) System.out.println(i); else System.out.println(j);
```

dove



if then else annidati

Annidamento sul ramo `else`

```
class IfThenElseAnnidati{
  public static void main(String args []) {
    int a = 10;
    final int b = 5;
    if (a > b) System.out.println("a > b");
    else if (a < b) System.out.println("a < b");
         else System.out.println("a = b");
  }
}
```

if then else annidati

Annidamento sul ramo `then`

```
class IfThenElseAnnidati{
  public static void main(String args []) {
    int a = 10;
    final int b = 5;
    if (a >= b) System.out.println("a >= b");
      if (a== b) System.out.println("a = b");
      else System.out.println("a < b");
  }
}
```

Il ramo `else` si lega sempre al costrutto `if` piú vicino

for

```
class For{
  public static void main(String args []) {
    final int limite = 10;
    for (int conta = 1; conta < limite; conta++)
      System.out.println(conta);
  }
}
```

L'uso generico é il seguente:

for (inizializzazione ; condizione per continuare ; incremento) comando

Come funziona il for

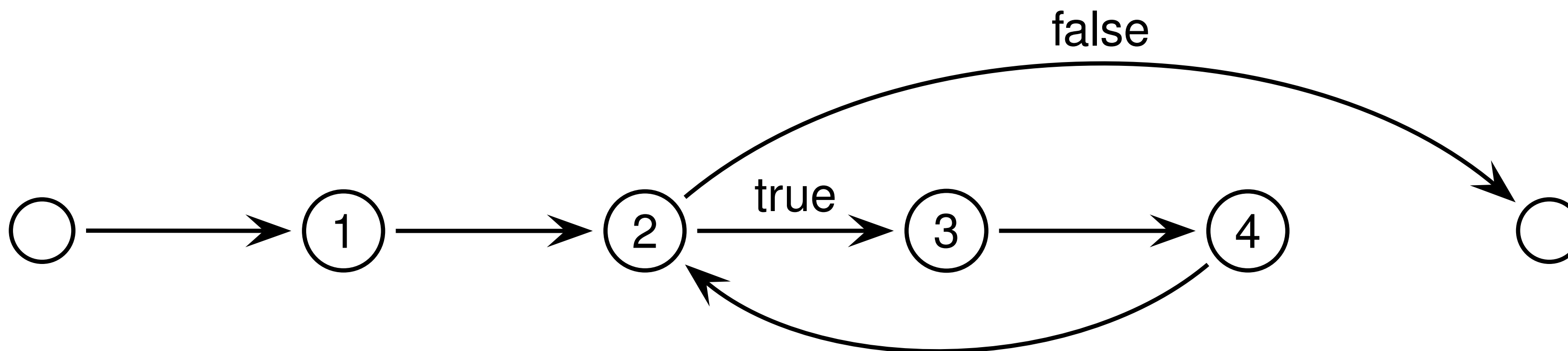
Il costrutto `for` puó essere sintetizzato in:

```
for ( /*1*/ ; /*2*/ ; /*4*/ ) /*3*/
```

ad esempio

```
for ( int i = 0 ; i < 10 ; i++ ) System.out.println(i);
```

dove



while

```
class While{
  public static void main(String args []) {
    int conta = 0;
    final int limite = 10; // \UTF{00E8} bene utilizzare sempre costanti
    while(conta < limite) {
      System.out.println(conta);
      conta = conta + 1; // oppure conta++;
    }
  }
}
```

L'uso generico é il seguente:

while (espressione booleana) comando

Come funziona il while

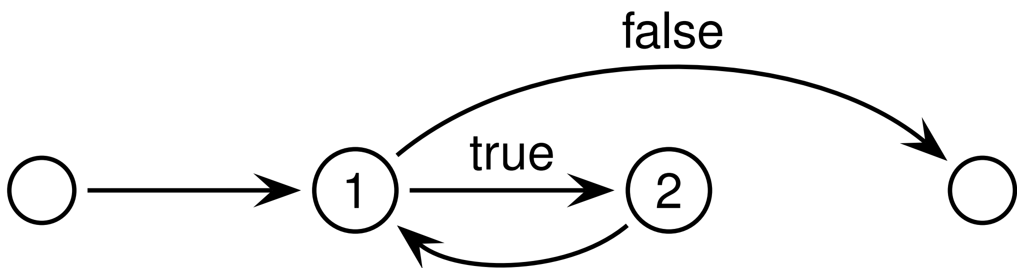
Il costrutto `while` puó essere sintetizzato in:

```
while ( /*1*/ ) /*2*/
```

ad esempio

```
int i = 0;
while (i < 10 ) System.out.println(i++);
```

dove



do while

```
class DoWhile{
  public static void main(String args []) throws IOException {
    char c;
    do {
      System.out.println("Nuovo carattere letto");
      c = (char) System.in.read();
    } while ((c != 's') || (c != 'n'));
  }
}
```

L'uso generico é il seguente:

do comando while (espressione booleana);

Come funziona il do while

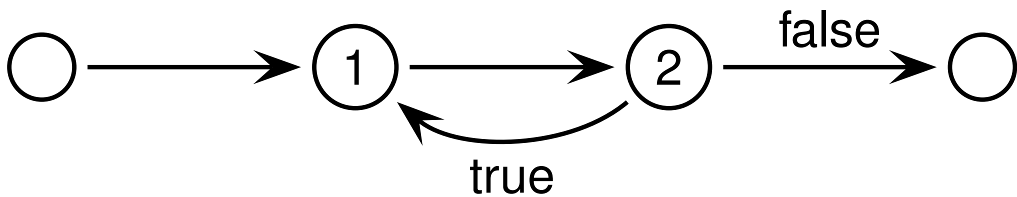
Il costrutto `do while` puó essere sintetizzato in:

```
do { /*1*/ } while ( /*2*/ );
```

ad esempio

```
int i = 0;
do { System.out.println(i++); } while ( i < 10 );
```

dove



switch

```
class Switch{
  public static void main(String args []) {
    int m = 3, g = 0;
    switch(m) {
      case 1: g += 31; case 2: g += 28;
      case 3: g += 31; case 4: g += 30;
      case 5: g += 31; case 6: g += 30;
      case 7: g += 31; case 8: g += 31;
      case 9: g += 30; case 10: g += 31;
      case 11: g += 30; case 12: g += 31;
      default: System.out.println("Errore");
    }
    System.out.println("Giorni "+m);
  }
}
```

Come funziona lo switch

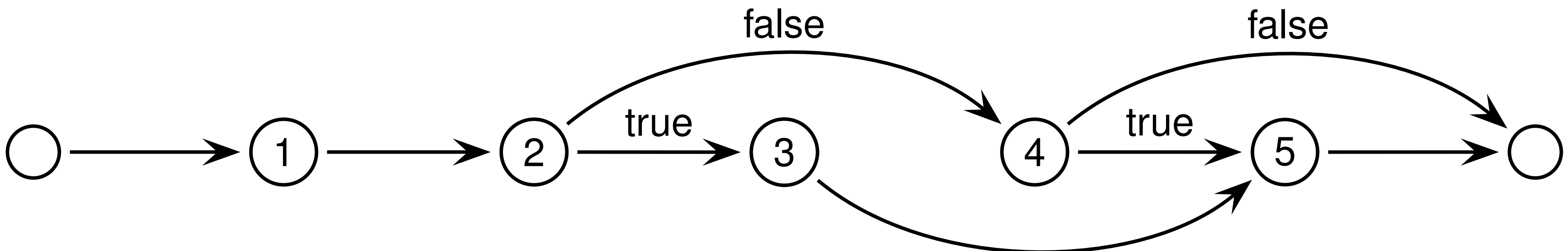
Il costrutto `switch` puó essere sintetizzato in:

```
switch ( /*1*/ ) { case /*2*/ : /*3*/ case /*4*/ : /*5*/ }
```

ad esempio

```
int i = 1;
switch (i) {
  case 1 : System.out.println("A");
  case 2 : System.out.println("B");
}
```

dove



break

```
class Break {
  public static void main(String args []) {
    int m = 3, g = 0;
    switch(m) {
      case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        System.out.println(31);
        break;
      case 2: System.out.println(28); break;
      case 4: case 6: case 9: case 11:
        System.out.println(30);
        break;
      default: System.out.println("Errore");
    }
  }
}
```

continue

```
class Continue{
  public static void main(String args []) {
    final int limite = 10;
    for (int conta = 1; conta < limite; conta++) {
      if (conta == 5) continue;
      System.out.println(conta);
    }
  }
}
```

Sintassi semplificata del comando switch

```
switch (espressione integrale) {
    case costante integrale: comando;
    ....
    case costante integrale: comando;
    default:comando;
}
```

Il comando associato ad un `case` é generalmente un blocco e puó contenere il comando `break` per saltare tutti i comandi successivi. Il costrutto `default` é facoltativo.

Lettura con Scanner

```
import java.util.Scanner;
public class ClasseScanner {
  public static void main (String args[]){
    Scanner in = new Scanner (System.in);
    int i = 0; double  d = 0.0; int ok = 0;
    while(ok == 0) {
      System.out.println("Impostare un intero o un reale");
      if (in.hasNextInt()) { i = in.nextInt(); ok = 1; }
      else if (in.hasNextDouble()) { d = in.nextDouble(); ok = 2; }
           else { System.out.println("Errore"); in.nextLine(); }
    }
    if (ok == 1) System.out.println(i); else System.out.println(d);
  }
}
```

Espressioni regolari

Nella libreria `java.util.regex` vi sono le classi`Pattern` per definire le espressioni regolari e `Matcher` per riconoscere elementi del linguaggio definito con `Pattern`.

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

Sempre su `java.util` vi é la classe `StringTokenizer` per spezzare stringhe in *token*.

```
StringTokenizer st = new StringTokenizer("this is a test");
     while (st.hasMoreTokens())
         System.out.println(st.nextToken());
```

Array

```
class Media{
  public static void main(String args []) {
    int [] esame = new int[5]; double somma = 0.0;
    esame[0] = 18;
    esame[1] = 24;
    esame[2] = 30;
    esame[3] = 25;
    esame[4] = 27;
    for (int i = 0; i < esame.length; i++) somma += esame[i];
    System.out.println(somma/esame.length);
  }
}
```

Giorni per mese

```
class GiorniMesi{
  public static void main(String args []) {
    int [] giorniMese = {31,29,31,30,31,30,31,31,30,31,30,31};
    int giorni = 0, da = 4;
    for ( ; da < giorniMese.length; da++) giorni+=giorniMese[da];
    System.out.println(giorni);
  }
}
```

For su insiemi

```
class Vocali{
  public static void main(String args []) {
    char [] vocale = {'a','e','i','o','u'};
    for ( char v : vocale) System.out.println(v+"->"+(v));
  }
}
```

Lettura e memorizzazione di caratteri

```
import java.io.*;
class LetturaCaratteri {
  public static void main(String args [])  throws IOException {
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    System.out.println("Imposta una sequenza di caratteri");
    final char FINE = '0'; final int DIM = 10000;
    char [] caratteri = new char[DIM]; int i = 0;
    do {
        char c = in.readLine().charAt(0);
        if (c == FINE) break; else { caratteri[i] = c; i++; }
    } while (i < DIM);
    System.out.print("Letti "); System.out.print(i);
    System.out.println("caratteri");
  }
}
```

Conta caratteri

```
import java.io.*;
class ContaCaratteri {
  public static void main(String args []) throws IOException {
    final int alfabeto = 26; int [] caratteri = new int[alfabeto];
    char carattere;
    while (true) {
      carattere = (char) in.read();
      if (carattere == '0') break;
      if (carattere < 'a' || carattere > 'z') continue;
      caratteri[carattere-'a']++;
    }
    for (int i = 0; i < alfabeto; i++) {
      System.out.print('a'+i); System.out.println(caratteri[i]);
    }
  }
}
```

Come misurare il tempo di esecuzione

Se volessimo misurare approssimativamente il tempo di esecuzione di un programma possiamo usare il metodo statico `nanoTime()` della classe `System` che produce il tempo in nanosecondi al momento dell'invocazione oppure `currentTimeMillis()` in millisecondi.

```
class QuantoTempo {
  public static void main(String args []) {
    long start = System.currentTimeMillis(); double n = 0.0;
    for (int i = 2; i < Long.MAX_VALUE/2; i*=2) n += 1.0/i;
    long stop = System.currentTimeMillis();
    System.out.println(n+" in "+(stop-start)+" ms");
  }
}
```

La misura non é accurata perché dipende fa fattori estranei che possono influenzare anche del 10% l'esecuzione dello stesso programma in momenti diversi.

Classi e oggetti

Oltre ai tipi primitivi, i linguaggi orientati agli oggetti offrono la possibilitá di definire nuovi tipi. La definizione dei tipi implica la definizione della struttura e delle operazioni che possono essere applicate agli elementi di tale tipo. Gli elementi di un tipo definito dall'utente si chiamano *oggetti* mentre i tipi definita dall'utente si chiamano *classi*. La possibilitá di definire nuovi tipi permette di:

- Dichiarare oggetti di un ben specifico tipo
- Manipolare gli oggetti con le sole operazioni definite per tale tipo
- Mantenere gli oggetti in uno stato sempre consistente con il significato che associamo al tipo degli oggetti.

Conseguentemente avremmo:

- minori errori causati da uso improprio di elementi
- possibilitá di suddividere in moduli in programma
- distinzione tra fase di definizione dei tipi e uso dei tipi mediante oggetti.

Ora

```
class Ora {
  private int ore, minuti; // attributi
  Ora(int o, int m) {
    if (o >= 0 && o < 24) ore = o;     // si vedr\'a in eccezioni
    if (m >= 0 && m < 60) minuti = m; // si vedr\'a in eccezioni
  }
  public void sommaMinuti(int m) { // operazione
    minuti = (minuti + m) % 60; // minuti che eccedono 60
    ore += (minuti + m) / 60;
    ore = ore % 24; // si \'e superata la mezzanotte
  }
  public void sommaOre(int o) { ore = (ore + o) % 24; }
  public String toString() {return new String(ore+":"+minuti); }
}
```

Uso di Ora

```
class UsoDiOra {
  public static void main(String args []) {
    Ora x = new Ora(10,30), y = new Ora(17,45);
    x.sommaMinuti(45); y.sommaOre(3);
    System.out.println(x);
    System.out.println(y);
  }
}
```

Data

```
class Data {
  private int giorno, mese, anno;
  Data(int g, int m, int a) {
    if (g > 0 && g <= 31) giorno = g;     // si vedr\'a in eccezioni
    if (m > 0 && m <= 12) mese = m;       // si vedr\'a in eccezioni
    if (a >= 0) anno = a; }               // solo DC
  private int daDataAIntero() { return anno*365+mese*31+giorno; }
  private Date daInteroAData(int n) {
    int a = n/365; int m = (n%365)/12; int g = (n%365)%12;
    return new Data(g,m,a); }
  public Data differenza(Data n) {
    return DaInteroAData(daDataAIntero()-n.daDataAIntero());}
  public String toString() {
    return new String(giorno+"/"+mese+"/"+anno); }
}
```

Uso di Data

```
class UsoDiData {
  public static void main(String args []) {
    Data x = new Data(17,1,395); // fine impero romano
    Data y = new Data(5,4,1453); // fine impero bizantino
    Data z = x.differenza(y);    // Quanto tempo tra le fine
    System.out.println(x);       // dell'impero bizantino
    System.out.println(y);       // e quello romano
    System.out.println(z);
  }
}
```

Dado

```
class GiocoDeiDadi {
  public static void main(String args []) {
    Dado uno = new Dado(), due = new Dado();
    System.out.println(uno.lancio());
    System.out.println(due.lancio());
    System.out.println(uno.lancio()+due.lancio());
  }
}
class Dado {
  private final int facce = 6;
  private Random r = new Random();
  int lancio() { return r.nextInt(facce)+1; }
  }
}
```

Classi e linguaggi classici

Per comprendere quale differenza vi sia tra tipi primitivi e tipi definiti dall'utente si consideri il tipo intero `int` e il tipo definito dall'utente `Int`.

```
class Int {
  private int n;
  Int(int i) { n = i; }
  public void put(int i){ n = i; }
  public int get() { return n; }
}
```

`int x; Int x = new Int();` Dichiarazione della variabile x

`x = 10; x.put(10);` Assegnazione del valore 10

`println(x); System.out.println(x.get());` Stampa del valore

Figura

```
class Figura { // nessun controllo
  private final int massimoNumeroDiLati = 20;
  private Segmeno [] lati = new Segmento[massimoNumeroDiLati];
  private int numeroSegmenti = 0;
  public add(Segmento lato) { lati[numeroSegmenti++] = lato; }
}
class Segmento { // nessun controllo
  private Punto a, b;
  Segmento(Punto a, Punto b) { this.a = a; this.b = b; }
}
class Punto { // nessun controllo
  private int x, y;
  Punto(int x, int y) { this.x = x; this.y = y; }
}
```

Lettura di interi, double e caratteri

```
import java.io.*;
public class Lettura {
  public static void main(String [] args) throws Exception {
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    int i = 0; double r = 0.0; char c = '\0';
    System.out.print("Intero: ");
    i = Integer.parseInt(in.readLine());
    System.out.print("Double: ");
    r = Double.parseDouble(in.readLine());
    System.out.print("Char: ");
    c = in.readLine().charAt(0);
    System.out.println(i); System.out.println(r);
    System.out.println(c);
  }
}
```

this

La parola chiave `this` é usata all'interno della definizione dei metodi per riferire all'oggetto in oggetto.

```
class Punto {
  private int x, y;
  Punto(int x, int y) { this.x = x, this.y = y; } // ambiguit\UTF{00E0}%
  public String toString() { return new String("("+x+","+y+")"); }
}
```

`this` all'interno dei metodi e dei costruttori permette di riferire gli attributi disambiguando i riferimenti omomini. Equivalente senza `this`

```
class Punto {
  private int x, y;
  Punto(int a, int b) { x = a; y = b; } // non ambiguit\UTF{00E0}%
  public String toString() { return new String("("+x+","+y+")"); }
}
```

Catena di punti

Esempio di costruzione di una catena di oggetti:

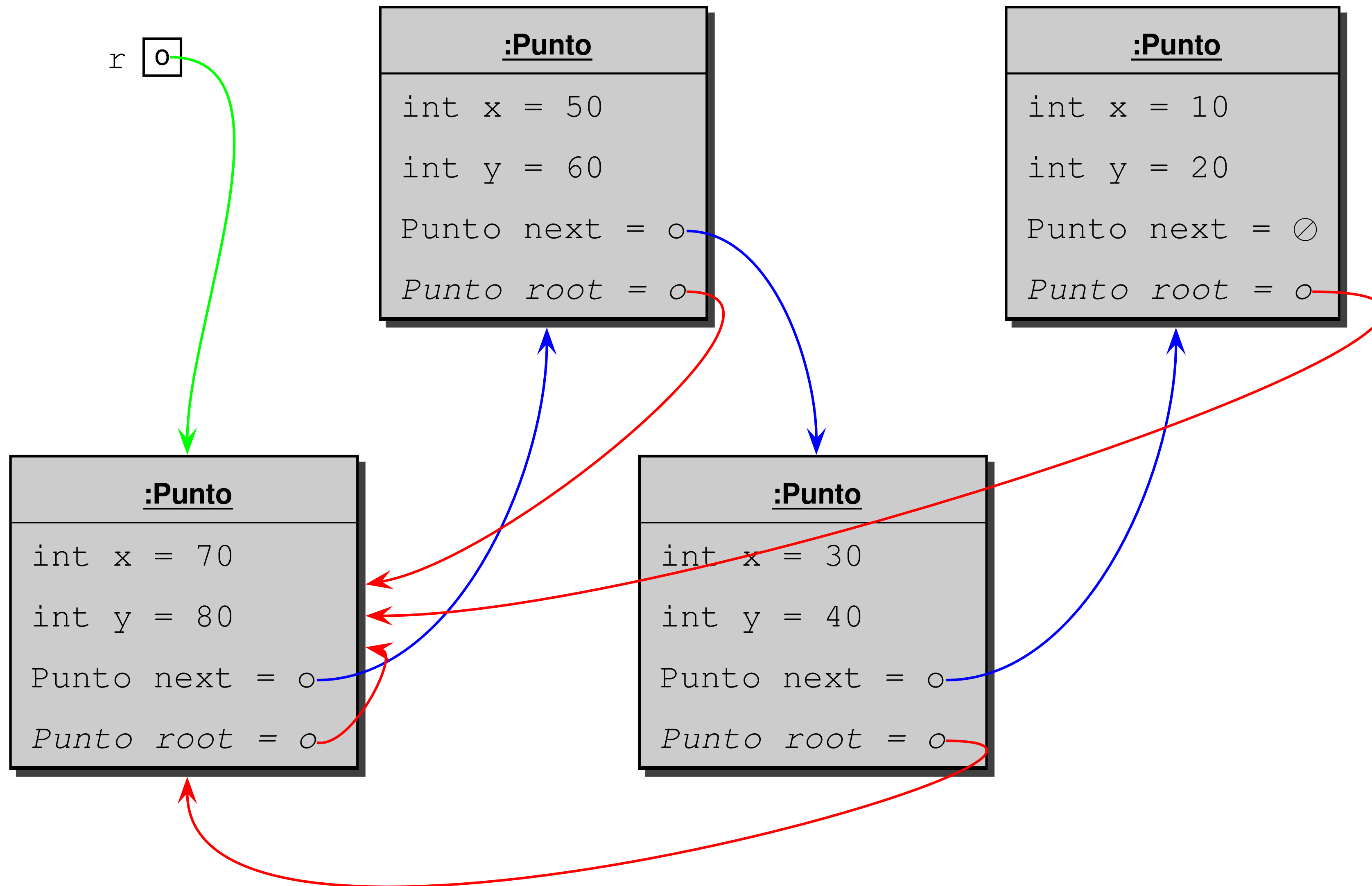
```
class Punto {
private static Punto root;
private int x, y;
private Punto next;
  Punto(int a, int b) { x = a; y = b; next = root; root = this; }
  public String toString() {
    return new String("("+x+","+y+") "
    + (next!=null ? next.toString() : " "));
  }
}
```

Uso di una catena di punti

```
class CostruzionePunti {
  public static void main(String [] args) {
    new Punto(10,20);
    new Punto(30,40);
    new Punto(50,60);
    Punto r = new Punto(70,80);
    System.out.println(r);
  }
}
```

Da qualsiasi punto é possibile stampare l'intera catena poiché tutti condividono la visibilitá di `root`

Attributi di classe e di istanza



Punto

```
class Punto { // un po' di controlli
  private int x, y;
  Punto(int x, int y) { this.x = x; this.y = y; }
  public int getX() { return x; }
  public int getY() { return y; }
  public boolean equals(Punto a) {
    return x == a.getX() && y == a.getY();
  }
  public Punto clone() { return new Punto(x,y); }
  public String toString() {
    return new String("("+x+","+y+")");
  }
}
```

Segmento

```
class Segmento { // un po' di controlli
  private Punto a, b;
  Segmento(Punto a, Punto b) { // eccezione!
    if (!a.equals(b)) { this.a = a; this.b = b; }
  }
  public Punto getA() { return a.clone(); }
  public Punto getB() { return b.clone(); }
  public String toString() {
    return new String(a+"-"+b);
  }
}
```

Figura

```
class Figura { // un po' di controlli
  private final int massimoNumeroDiLati = 20;
  private Segmento [] lati = new Segmento[massimoNumeroDiLati];
  private int numeroSegmenti = 0;
  public add(Segmento lato) {
    for (int i = 0; i < numeroSegmenti; i++) {
      if (lato.getA()==lati[i].getA()||lato.getA()==lati[i].getB()
       ||lato.getB()==lati[i].getA()||lato.getB()==lati[i].getB())
           lati[numeroSegmenti++] = lato;
    }
  }
  public String toString() {
    String r;
    for (int i = 0; i < numeroSegmenti; i++) r = r + lati[i];
    return r;
  }
}
```

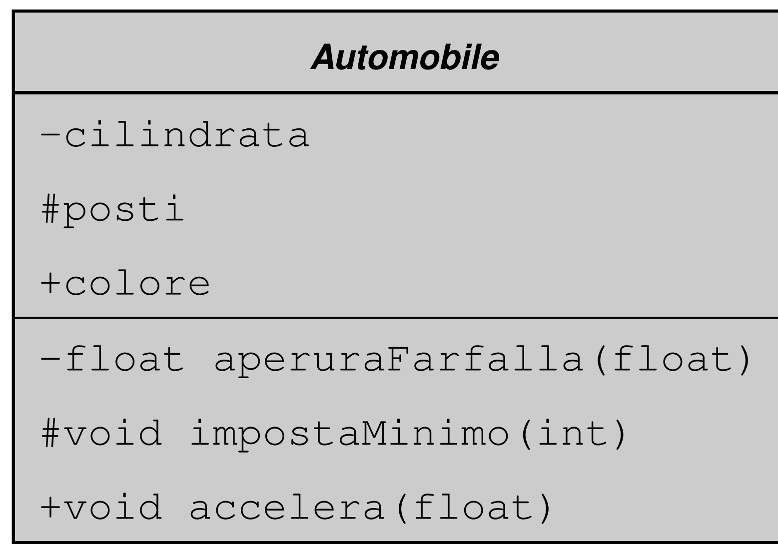
Uso di Figura

```
class UsoDiFigura {
  public static void main(String args []) {
    Figura triangolo = new Figura(), quadrato = new Figura();
    triangolo.add(new Segmento(new Punto(0,0), new Punto(1,3)));
    triangolo.add(new Segmento(new Punto(1,3), new Punto(2,0)));
    triangolo.add(new Segmento(new Punto(2,0), new Punto(0,0)));
    quadrato.add(new Segmento(new Punto(4,4), new Punto(4,8)));
    quadrato.add(new Segmento(new Punto(4,8), new Punto(8,8)));
    quadrato.add(new Segmento(new Punto(8,8), new Punto(8,4)));
    quadrato.add(new Segmento(new Punto(8,4), new Punto(4,4)));
    System.out.println(triangolo);
    System.out.println(quadrato);
  }
}
```

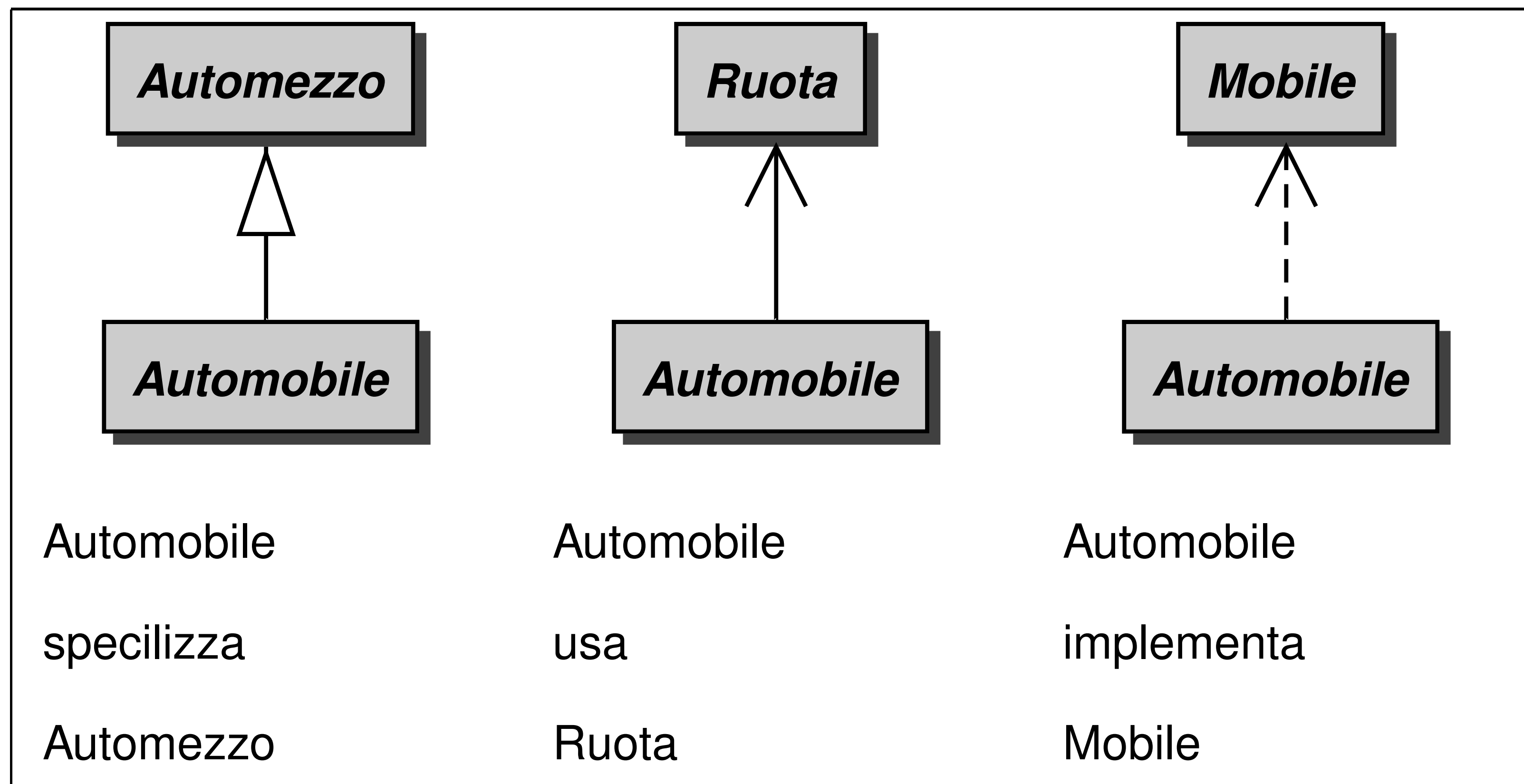
Rappresentazione grafica delle classi

La rappresentazione grafica UML evidenzia:

nome della classe	
attributi	
metodi	nome parametri valore prodotto
visibilitá di attributi e metodi	+ publico # protetto - privato



Grafica di uso e specializzazione



La rappresentazione grafica semplifica la comprensione se il numero degli elementi é elevato

Classi o istanze

I linguaggi orientati agli oggetti hanno strutture sintattiche per esprimere la classificazione e gli oggetti appartenenti a tali classi.

La classe descrive le caratteristiche comuni di oggetti. Gli oggetti appartengono ad una classe.

Invocazione di attributi e metodi

Una volta creato un oggetto (istanza) di una classe, é possibile interagire con esso invocando i metodi definiti per tale oggetto. L'invocazione ha la seguente forma:

```
nomeOggetto.nomeMetodo()
```

Nel caso si voglia accedere agli attributi:

```
nomeOggetto.nomeAttributo
```

Il `.` ha il significato di "applicazione" del metodo o dell'attributo all'oggetto indicato nella parte sinistra del punto.

Attributo static

```
class Static{
  public static void main(String args []) {
    Serial a = new Serial(), b = new Serial();
    System.out.println(a.get());
    System.out.println(b.get());
    System.out.println(a.get());
  }
}
class Serial {
  private static int conta;
  int get() { return conta++; }
}
```

Metodo static

```
class Static{
  public static void main(String args []) {
    MetodoStatic a = new MetodoStatic();
    System.out.println(a.inc());
    System.out.println(a.sub());
    System.out.println(MetodoStatic.add(10,20)); // Classe.add
    System.out.println(a.add(10,20));
  }
}
class MetodoStatic {
  private int conta; // inzializzato automaticamente a 0
  public int inc() { return conta++; }
  public int sub() { return conta--; }
  public static int add(int a, int b) { return a+b; }
}
```

Numero di serie

Un esempio di uso di `static` come attributo é la definizione di una classe che genera (oggetti) numeri di serie

```
class Serial {
  private static int count = 0;
  private int serial;
  Serial() { serial = count++; }
  public int get() { return serial; }
}
class UsoSerial {
  public static void main(String [] args) {
    Serial a = new Serial(), b = new Serial();
    System.out.print(a.get()); System.out.println(b.get());
  }
}
```

Parametri primitivi

```
class ParametriPrimitivi {
  public static void main(String args []) {
    Box a = new Box(10);
    int b = 20; System.out.println(a.op(b));
    System.out.println(b);
  }
}
class Box {
  private int x;
  Box(int x) { this.x = x; }
  public int op(int n) { n +=x; return n; }
}
```

Parametri non primitivi

```
class ParametriNonPrimitivi {
  public static void main(String args []) {
    Integer a = new Integer(10), b = new Integer(20);
    Box c = new Box(a);
    c.op(b);
    System.out.println(c);
  }
}
class Box {
  private int x;
  Box(Integer x) { this.x = x.intValue(); }
  public int op(Integer n) {
    n = new Integer(n.getValue()+x);
    return n.intVaue(); }
}
```

Intero non protetto

```
class InteroNonProtetto {
  public static void main(String args []) {
    Numero x = new Numero(3);
    x.inc();
    Integer y = x.get(); y = new Integer(20);
  }
}
class Numero {
  private Integer x;
  Numero(int n) {if (n < 10) x = new Integer(n); }
  public Integer get() { return x; } // si produce il riferimento
  public void inc() {
    if (x.intValue() < 10) x = new Integer(x.intValue()+1);
  }
}
```

Intero protetto

```
class InteroProtetto {
  public static void main(String args []) {
    Numero x = new Numero(3);
    x.inc();
    Integer y = x.get(); y = new Integer(20);
  }
}
class Numero {
  private Integer x;
  Numero(int n) {if (n < 10) x = new Integer(n); }
  public Integer get() { return new Integer(x.getValue()); } /**/
  public void inc() {
    if (x.intValue() < 10) x = new Integer(x.intValue()+1);
  }
}
```

Gauss iterativo

```
class GaussIterativo {
  public static void main(String args []) {
    System.out.println(Gauss(10));
    System.out.println(Gauss(15));
  }
  private int Gauss(int n) {
    int s = 0;
    for (int i=1; i <= n; i++) s+=i;
    return s;
  }
}
```

Gauss ricorsivo

```
class GaussRicorsivo {
  public static void main(String args []) {
    System.out.println(Gauss(10));
    System.out.println(Gauss(15));
  }
  private int Gauss(int n) {
    if (n == 0) return 0;
    else return n + Gauss(n-1);
  }
}
```

Fattoriale ricorsivo

```
class FattorialeRicorsivo {
  public static void main(String args []) {
    System.out.println(Fattoriale(10));
    System.out.println(Fatoriale(15));
  }
  private int Fattoriale(int n) {
    return n == 0 ? 1 : n * Fattoriale(n-1);
  }
}
```

Fibonacci ricorsivo

```
class FibonacciRicorsivo {
  public static void main(String args []) {
    System.out.println(Fibonacci(10));
    System.out.println(Fibonacci(15));
  }
  private int Fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return Fibonacci(n-1) + Fibonacci(n-2);
  }
}
```

Stringhe

Per stringhe si intende una sequenza di caratteri. Le stringhe in Java sono un tipo e quindi possono essere dichiarate variabili ed esistono costanti:

```
String nome; // dichiarazione di variabile di tipo stringa
String cognome = "Rossi"; // variabile e costante
```

Le stringhe in Java NON sono tipo primitivo anche se "sembrano" primitivi:

```
String x;
x = "Paolo "+"Rossi"; // il + concatena stringhe
```

Operazioni su stringhe

La classe `String` é dichiarata nel pacchetto `java.lang` e inserita automaticamente dal compilatore. I metodi piú comuni sono:

```
String x = new String("Paolo");
String y = x.concat(" Rossi"); // crea nuova stringa "Paolo Rossi"
int z = y.indexOf('o'); // produce 2, l'indice della prima 'o'
int k = y.length(); // produce 11, la lunghezza della stringa
String l = y.toUpperCase(); // produce "PAOLO ROSSI"
String m = y.trim(); // produce "PaoloRossi"
```

`String` non permette di modificare un oggetto

```
String x = new String("Paolo");
x = x.toUpperCase(); /
```

Non modifica "Paolo" ma crea un nuovo oggetto "PAOLO". Il riferimento a "Paolo" é perso (l'oggetto non é piú raggiungibile).

Astrazione

- Capacitá di classificare gli elementi come appartenenti ad una classe. Ad esempio *mela* e *pera* sono *frutti*. *Panda* e *Golf* sono *auto*.
- Capacitá di identificare le caratteristiche che distinguono tra di loro gli elementi che appartengono alla stessa classe. Ad esempio *colore* e *periodo* per *frutti* o *cilindrata* e *posti* per *auto*.
- Capacitá di riconoscere una gerarchia di classi, ovvero di avere classi legate tra di loro con caratteristiche piú specifiche o piú generali. Ad esempio nel caso di *auto* puó specializzarsi in *auto da corsa* e *auto da rally* oppure *auto* é una specializzazione di *automezzo*

Array

Un array é una collezione di elementi tutti dello stesso tipo identificabili con un intero positivo

```
class Elenco {
  public static void main(String [] argv) {
    int [] numeri; /* dichiarazione di una variabile di tipo
                      array di interi */
    numeri = new int[100]; // creazione di un array di interi
    for (int i = 0; i < 100; i++) numeri[i] = i;
    for (int i = 99; i >= 0; i--) System.out.println(numeri[i]);
  }
}
```

Un elenco di 100 interi é creato e quindi stampato in ordine inverso

Array di tipo non primitivo

Il tipo di un array puó essere di tipo dichiarato dall'utente;

```
class ElencoTemperature {
  public static void main(String [] argv) {
    Temperatura [] temperature; /* dichiarazione di una variabile
                                   di tipo array di Temperatura */
    temperature = new Temperatura[100]; // creazione
    for (int i = 0; i < 100; i++)
      temperture[i] = new Temperatura(i);
    for (int i = 99; i >= 0; i--)
      System.out.println(temperature[i]);
  }
}
```

Notare i due `new`, uno per creare l'array e l'altro per creare i singoli oggetti

Matrici

Possono essere dichiarati Matrici o vettori multidimensionali;

```
class Scacchi {
  public static void main(String [] argv) {
      long scacchiera[8][8], long riso = 2;
      for (int i = 0; i < 8; i++)
        for (j=0; j < 8; j++) {
          scacchiera[i][j] = riso; riso *= riso;
        }
      System.out.println(cacchiera[8][8]);
  }
}
```

Un vettore multidimensionale é un vettore di vettori.

```
int a[][] = {{1},{1,2},{1,2,3},{1,2,3,4},{1,2,3,4,5}};
```

Array polimorfo

E’ possibile creare un array di tipo `Object` ovvero del tipo progenitore di qualsiasi altro tipo definito dall’utente. Si potrá quindi scrivere

```
class ElencoPolimorfo {
  public static void main(String [] argv) {
    Object [] oggetti; // dichiarazione
    oggetti = new Object[100]; // creazione
    oggetti[0] = new Temperatura(10.9);
    oggetti[1] = new String("Paolo");
    oggetti[2] = new Dado();
    oggetti[3] = new Double(3.1425);
  }
}
```

Gli elementi di `oggetti` sono tutti oggetti (non é possibile scrivere `oggetti[4] = 6;`).

Limiti dell'array polimorfo

Gli array polimorfi di tipo `Object` hanno principalmente limiti dovuti al fatto che:

- le caratteristiche del tipo specifico si perdono nel momento in cui vengono assegnate al vettore. E' effettuato un cast implicito che toglie tutte le specificitá dell'oggetto inserito nell'array

```
oggetti[0] = (Object)(new Temperatura(10.9));
```

- per recuperare le specificitá degli oggetti presenti in un array polimorfo bisogna effettuare un cast esplicito

```
((Temperatura)(oggetti[0])).set(12.7);
System.out.println(((String)(oggetti[1])).toUpperCase());
```

istanceof

Esiste un operatore che dato un oggetto e un tipo produce `true` o `false` se l'oggetto é istanza del tipo.

```
class ElencoPolimorfoConStampa {
  public static void main(String [] argv) {
    Object [] oggetti; // dichiarazione
    oggetti = new Object[100]; // creazione
    oggetti[0] = new Temperatura(10.9);
    oggetti[1] = new String("Paolo");
    oggetti[2] = new Dado();
    oggetti[3] = new Double(3.1425);
    for (int i = 0; i < oggetti.length; i++)
      if (oggetti[i] istanceof String)
        System.out.println(oggetti[i]);
  }
}
```

Incapsulamento

Gli elementi necessari per la descrizione e il funzionamento dei tipi devono essere nascosti all'esterno per evitare anomalie nel funzionamento degli oggetti creati con tale tipo. I metodi e il costruttore filtrano le modiche a tali elementi.

```
class Punto {
  private int x, y;
  Punto(int a, int b) {
    if (a >= 0 && b >= 0) {x = a; y = b;}  // altrimenti eccezione
  }
  Punto() { x = y = 0; }
  public int getX() { return x; }
  public int getY() { return y; }
  public void putX(int a) {if (a >= 0) x = a; }
  public void putY(int a) {if (a >= 0) y = a; }
  public String toString() { return "X:"+x+" Y:"+y; }
}
```

Uso della classe Punto

Gli elementi caratterizzanti (`x` e `y`) di dei punti `origine`, `alfa` e `beta` una volta creati, possono essere alterati solo mediante i metodi `putX(int)`, `putY(int)` che controllano la congruitá dell'alterazione.

```
class UsoDiPunto {
  public static void main(String args []) {
    Punto origine = new Punto();
    Punto alfa = new punto(10,20);
    Punto beta = new Punto(2,3);
    alfa.putX(30); beta.putY(50);
    System.out.println(alfa);
    System.out.println(beta);
  }
}
```

Dichiarazione errata di Punto

Una piccola modifica nella dichiarazione di `Punto` puó causare anomalie di funzionamento

```
class Punto {
  public int x, y;
  Punto(int a, int b) {
    if (a >= 0 && b >= 0) {x = a; y = b;}  // altrimenti eccezione
  }
  Punto() { x = y = 0; }
  public int getX() { return x; }
  public int getY() { return y; }
  public void putX(int a) {if (a >= 0) x = a; }
  public void putY(int a) {if (a >= 0) y = a; }
  public String toString() { return "X:"+x+" Y:"+y; }
}
```

Uso della classe Punto errata

Gli elementi caratterizzanti (`x` e `y`) di dei punti `origine`, `alfa` e `beta` una volta creati, possono essere alterati oltre che mediante i metodi `putX(int)`, `putY(int)` che controllano la congruitá anche direttamente.

```
class UsoDiPuntoErrato {
  public static void main(String args []) {
    Punto origine = new Punto();
    Punto alfa = new punto(10,20);
    Punto beta = new Punto(2,3);
    alfa.putX(30); beta.putY(50);
    alfa.x = -40; beta.y = -3; // corretto sintatticamente ma ...
    System.out.println(alfa);
    System.out.println(beta);
  }
}
```

Ereditarietá

E’ possibile specializzare classi create da noi o da qualcun altro per adattarle a usi specifici. La specializzazione puó essere per:

- aggiunta di attributi e metodi
- modifica del comportamento di metodi giá presenti

Bisogna tenere presente anche del comportamento dei costruttori delle classi interessate.

Aggiunta di attributi e metodi

```
class Temperatura {
  private float t;
  Temperatura(float t) { if (t > 0 && t < 100) { this.t = t;}} //e
  public void putT(float a) {if (a > 0 && a < 100) t = a; } // e
  public String toString() { return "Temperatura:"+t; }
}
class TemperaturaConTempo extends Temperatura {
  private int s;
  TemperaturaConTempo(float t, int s) {
    if (s > 0) {super(t); this.s = s; // e
  }
  public void putS(int s) { if (s > 0) this.s = s; } // e
}
```

Modifica del comportamento

```
class Temperatura {
  private float t;
  Temperatura(float t) { if (t > 0 && t < 100) { this.t = t;}}
  public void putT(float a) {if (a > 0 && a < 100) t = a; }
  public String toString() { return "Temperatura:"+x; }
}
class TemperaturaCorpore extends Temperatura {
  TemperaturaCorporea(float t) { if (t > 30 && t < 50) super(t); }
  public void putT(float a) {if (a > 30 && a < 50) super.putT(a); }
}
```

L’uso di super

La parola chiave `super` in una dichiarazione di classe che specializza permette di invocare:

- il costruttore della classe padre
- gli attributi della classe padre, secondo i limiti di visibilitá
- i metodi della classe padre, secondo i limiti di visibilitá

Modificatori della visibilitá

Esistono tre tipi di modificatori (piú uno implicito), che possono essere usati nella dichiarazioni di:

- di classi e interfacce
- di attributi
- di metodi

```
class A {
  public int a;            // attributo visibile (modificabile) da tutti
  protected int b;         // attributo visibile da chi specializza
  private int c;           // visibile solo dai metodi di questa classe
  A() {}                   // il costruttore non ha modificatori (pubblico)
  public void a1(){}       // metodo invocabile da tutti
  protected void b1() {}   // invocabile anche da chi specializza
  private void c1(){}      /* metodo invocabile solo da altri
                              metodi di questa classe */
}
```

Significato di visibilitá

Per classe, interfaccia, metodo o attributo visibile di intende la possibilitá di:

- accedere in lettura, per attributi; invocazione per metodi; dichiarazione per classi e interfacce
- accedere in scrittura, per attributi; riscrittura per metodi

Attributi privati

```
class Temperatura {
  private float t;
  Temperatura(float t) { if (t > 0 && t < 100) { this.t = t;}}
  public void putT(float a) {if (a > 0 && a < 100) t = a; }
  public String toString() { return "Temperatura:"+x; }
}
class TemperaturaCorporea extends Temperatura {
  TemperaturaCorporea(float t) { if (t > 30 && t < 50) super(t); }
  public putT(float a) {if (a > 30 && a < 50) super.putT(a); }
  public stampa() { System.out.println(t); }
    // l'attributo t in Temperatura non \'e visibile da qui
}
```

Attributi protetti

```
class Temperatura {
  protected float t;
  Temperatura(float t) { if (t > 0 && t < 100) { this.t = t;}}
  public void putT(float a) {if (a > 0 && a < 100) t = a; }
  public String toString() { return "Temperatura:"+x; }
}
class TemperaturaCorporea extends Temperatura {
  TemperaturaCorporea(float t) { if (t > 30 && t < 50) super(t); }
  public void putT(float a) {if (a > 30 && a < 50) super.putT(a); }
  public void incrementa() { t = t + 1.0; } // corretto
}
class UsoDiProtected {
  public static void main(String args []) {
    TemperaturaCorporea paolo = new TemperaturaCorporea(36.5);
    paolo.t = 38.0; // t non \'e visibile da qui
  }}
```

Attributi pubblici

```
class Temperatura {
  public float t;
  Temperatura(float t) { if (t > 0 && t < 100) { this.t = t;}}
  public void putT(float a) {if (a > 0 && a < 100) t = a; }
  public String toString() { return "Temperatura:"+x; }
}
class TemperaturaCorporea extends Temperatura {
  TemperaturaCorporea(float t) { if (t > 30 && t < 50) super(t); }
  public void putT(float a) {if (a > 30 && a < 50) super.putT(a); }
  public String incrementa() { t = t + 1.0; } // corretto
}
class UsoDiProtected {
  public static void main(String args []) {
    TemperaturaCorporea paolo = new TemperaturaCorporea(36.5);
    paolo.t = -38.0; // corretto
  }}
```

Metodi privati

```
class Temperatura {
  private float t;
  Temperatura(float t) { if (t > 0 && t < 100) { this.t = t;}}
  private void incrementa() { t = t + 1.0; }
  public String toString() { incrementa(); return "Temperatura:"+t; }
}
class TemperaturaCorporea extends Temperatura {
  TemperaturaCorporea(float t) { if (t > 30 && t < 50) super(t); }
  public void febbre() {if  (a < 49) incrementa(); } // errore
}
class UsoDiProtected {
  public static void main(String args []) {
    TemperaturaCorporea paolo = new TemperaturaCorporea(36.5);
    paolo.incrementa();  // errore
  }
}
```

Metodi protected

```
class Temperatura {
  private float t;
  Temperatura(float t) { if (t > 0 && t < 100) { this.t = t;}}
  protected void incrementa() { t = t + 1.0; }
  public String toString() { incrementa(); return "Temperatura:"+t; }
}
class TemperaturaCorporea extends Temperatura {
  TemperaturaCorporea(float t) { if (t > 30 && t < 50) super(t); }
  public void febbre() {if  (a < 49) incrementa(); } // corretto
}
class UsoDiProtected {
  public static void main(String args []) {
    TemperaturaCorporea paolo = new TemperaturaCorporea(36.5);
    paolo.incrementa();  // errore
  }
}
```

Metodi pubblici

```
class Temperatura {
  private float t;
  Temperatura(float t) { if (t > 0 && t < 100) { this.t = t;}}
  public void incrementa() { t = t + 1.0; }
  public String toString() { incrementa(); return "Temperatura:"+t; }
}
class TemperaturaCorporea extends Temperatura {
  TemperaturaCorporea(float t) { if (t > 30 && t < 50) super(t); }
  public void febbre() {if  (a < 49) incrementa(); } // corretto
}
class UsoDiProtected {
  public static void main(String args []) {
    TemperaturaCorporea paolo = new TemperaturaCorporea(36.5);
    paolo.incrementa();  // corretto
  }
}
```

Visibilitá implicita

Se non é specificata alcun tipo di visibilitá nelle dichiarazioni di classi, interfacce, attributi e metodi, allora la visibilitá é a livello di package. E' comunque consigliato la dichiarazione esplicita della visibilitá.

```
package Temperature
class Temperatura {...}
class TemperaturaCorporea extends Temperatura {...}
class UsoDiTempeerature {...}
```

Le classi `Temperatura`, `TemperaturaCorporea` e `UsoDiTemperature` sono reciprocamente visibili

```
public class Temperatura {...}
public class TemperaturaCorporea extends Temperatura {...}
public class UsoDiTempeerature {...}
```

Con `public` davanti al nome di classe la visibilitá é globale

Classi astratte

Talvolta si ha la necessitá di avere classi "incomplete", ovvero classi non utilizzabili per creare oggetti ma utili per "astrarre" comportamenti generali per un gruppo di classi specializzate

```
class FiguraPiana {
  public float area();
}
```

Oppure

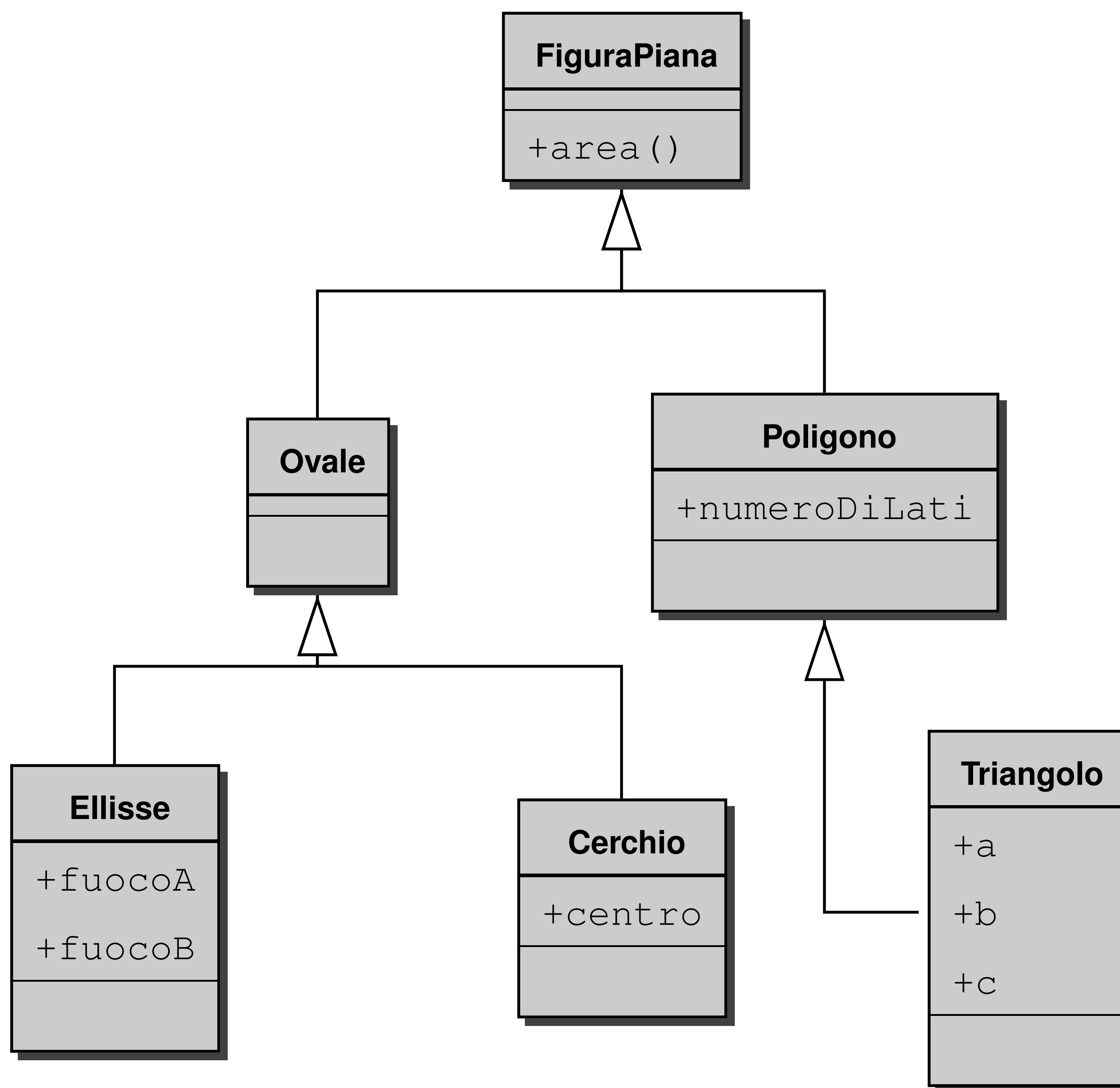
```
abstract class FiguraPiana {
  public float area();
}
```

La classe `FiguraPiana` possiede solo il prototipo del metodo `area()`, ovvero nome, parametri e valore prodotto ma senza corpo {}. La parola chiave `abstract` puó essere omessa se nella dichiarazione della classe é presente almeno un metodo senza corpo.

Gerarchia di classi

```
abstract class Ovale extends FiguraPiana {
}
class Cerchio extends Ovale {
  private Punto centro;
}
abstract class Ellisse extends Ovale {
  private Punto fuocoA, FuocoB;
}
abstract class Poligono extends FiguraPiana {
  private int numeroDiLati;
}
class Triangolo extends Poligono {
  private Punto a, b, c;
}
```

Rappresentazione grafica UML



Classi astratte e concrete

Vi sono due tipi di classi:

- Astratta. Sono le classi che hanno la parola `abstract` prima di `class` o che hanno almeno un metodo senza corpo (signature).
- Concreta. Sono le classi che hanno tutti i metodi con corpo.

Le classi astratte esistono perché prima o poi qualcuno li utilizzerá per creare una specializzazione. Le classi concrete servono per creare istanze, ovvero oggetti di quel tipo. Le classi concrete che specializzano classi astratte hanno l'obbligo di riscrivere e dare corpo ai metodi dichiarati (signature) nelle classi astratte.

Variabili e oggetti

I nomi delle classi sono usati in due contesti:

- Dichiarazione di variabili. Si creano dei nomi che possono etichettare oggetti

```
Cerchio x;
Triangolo y;
```

  `x` e `y` sono etichette che possono riferire ad oggetti rispettivamente di tipo

  `Cerchio` e `Triangolo`

- Creazione di oggetti. Si creano degli elementi di un particolare tipo

```
new Cerchio(c,12.5);
new Triangolo(a,b,c);
```

Polimorfismo

In Java é possibile avere discrepanza tra il tipo delle variabili e oggetti:

```
Ovale x;
Punto fuoco1 = new Punto(1.0,2.6), fuoco2 = new Punto(1.2,3.5);
x = new Cerchio(c,12.5);          // 1
System.out.println(x.getCentro()); // 2
System.out.println(x.area());      // 3
x = new Ellisse(fuoco1, fuoco2);   // 4
System.out.println(x.getFuochi);   // 5
System.out.println(x.area());      // 6
```

Alla variabile `x` si assegna in `1` un oggetto `Cerchio` e successivamente alla stessa variabile `x` in `4` un oggetto `Ellisse`. Ció é possibile poiché sia `Cerchio` che `Ellisse` sono "figli" di `Ovale`.

Invocazione di metodi nel polimorfismo

Quando la variabile `x` ha assegnato un oggetto `Cerchio` su `x` possono essere applicati tutti i metodi definiti in `Cerchio` e i metodi ereditati dagli antenati (come `area`).

Cosí che quando ad `x` é assegnato un oggetto `Ellisse` su `x` possono essere applicati i tutti i metodi definiti in `Ellisse` e i metodi ereditati dagli antenati (come `area`).

new

Per creare oggetti bisogna usare la parola chiave `new`.

```
Cerchio x;
Cerchio y = new Cerchio(c,12.5);
x = y;
```

`x` é solo un'etichetta mentre `new Cerchio(c,12.5)` crea un oggetto con `=` si lega l'oggetto creato all'etichetta `y`. Con `x=y` si lega ció che é legato a `y` a `x`

Una volta creato un oggetto con `new` esiste sino alla fine dell'esecuzione del programma (teorico). La raggiungibilitá di un oggetto é garantita dall'assegnamento ad una variabile:

```
Cerchio x = new Cerchio(c,12.5);
```

La dove é visibile `x` é possibile accedere all'oggetto associato

Costruttore

La parola chiave `new` é sempre seguita dal nome di una classe. `new` invoca il costruttore della classe. Vi sono due tipi di costruttori:

- Inplicito. Se non é definito almeno un "metodo" speciale senza tipo come valore prodotto e con il nome della classe
- Esplicito. Se é definito uno o piú "metodi" senza tipo come valore prodotto e con lo stesso nome della classe

Costruttore esplicito

```
class Punto {
  private int x, y;
  Punto(int a, int b) { x = a; y = b; }
  public String toString() { return new String("("+x+","+y+")"); }
}
class ProvaPunto {
  public static void main(String [] argv) {
    Punto a = new Punto(10,20);
    Punto b = new Punto(30,40);
    System.out.println(a); System.out.prinln(b);
  }
}
```

Costruttore implicito

```
class Punto {
  private int x, y;
  public String toString() { return new String("("+x+","+y+")"); }
  public void setX(int a) { x = a; }
  public void setY(int a) { y = a; }
}
class ProvaPunto {
  public static void main(String [] argv) {
    Punto a = new Punto(); a.setX(10); a.setY(20);
    Punto b = new Punto(); b.setX(30); b.setY(40);
    System.out.println(a); System.out.prinln(b);
  }
}
```

Overloading del costruttore

```
class Punto {
  private int x, y;
  Punto() { x = 0; y = 0; }
  Punto(int a) { x = y = a; }
  Punto(int a, int b) { x = a; y = b; }
  public String toString() { return new String("("+x+","+y+")"); }
}
class ProvaPunto {
  public static void main(String [] argv) {
    Punto a = new Punto(10,20);
    Punto b = new Punto(30);
    Punto c = new Punto();
    System.out.println(a); System.out.prinln(b);
    System.out.println(c);
  }
}
```

Tipi di ereditarietá

Da una classe si possono avere molte specializzazioni. Ad esempio da `FiguraPiana` si puó specializzare in `Ovale`, `Poligono`. Ma ogni classe puó avere solo un padre (singola ereditarietá). Se si vuole impedire ad una classe o ad un metodo di essere *overloading* o *overwriting* (si veda dopo) si usa la patola chiave `final`

```
final class Cerchio extends Ovale {
  private Punto centro;
}
```

Non sará possibile

```
class DoppioCerchio extends Cerchio {
}
```

La parola chiave final

Se la parola `final` é usata nella dichiarazione di un metodo non sará possibile riscrivere il metodo nelle classi che specializzano

```
class Cerchio extends Ovale {
  private Punto centro;
  final public circonferenza() { ... }
}
```

Non sará possibile

```
class DoppioCerchio extend Cerchio {
  public circonferenza() { ... }
  public anello() { ... }
}
```

Riscrittura e sovrascittura di metodi

```
class Cerchio extends Ovale {
  private Punto centro;
  private float raggio;
  public circonferenza() { ... }
}
class DoppioCerchio extends Cerchio {
  private float raggioInterno;
  public circonferenza() { ... }
  public area() { ... }
}
```

La specializzazione `DoppioCerchio` introduce un secondo raggio e il calcolo della circonferenza come media delle due circonferenze riscrivendo il metodo `circonferenza` di `Cerchio` e riscivendo `area` dichiarato in `FiguraPiana` Differenze tra overloading e overwriting. Casi di uso.

Enumerazioni

Java nelle versioni successive alla 1.4 ha introdotto il concetto di insieme tramite le enumerazioni.

```
enum PuntiCardinali {nord, sud, est, ovest}
enum Stagioni {autunno, inverno, primavera, estate}
```

L'uso

```
for (Stagioni g : Stagioni.values())
System.out.println(g);
```

Le enumerazioni sono essenzialmente classi semplificate estensioni di una classe che mette a disposizione metodi per la gestione di elenchi.

Classi annidate

```
class Contiene {
  class Contenuta { // start inner class
    private String n;
    Contenuta(String p) { n = p; }
    public String concatena(String s) { n += s; return n; }
  } // stop inner class
  public void tutto(String s) {
    Contenuta c = new Contenuta("Pluto");
    System.out.println(c.concatena(" mangia ")+"osso");
  }
}
```

Classi annidate anonime

```
import java.awt.*;
import java.awt.event.*;
public class AwtControlDemo extends Frame {
   public static void main(String[] args){
     AwtControlDemo d = new AwtControlDemo();
     d.setSize(400,400);
     d.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent windowEvent){
            System.exit(0);
        }
     });
     d.setVisible(true);
   }
}
```

Cast

E’ possibile far "sembrare" un oggetto o un tipo primitivo "leggermente" diverso da quello che é realmente. Questa operazione di "travestimento" viene chiamata *cast* e puó essere applicata sia tipi primitivi che a tipi definiti dall’utente (oggetti).

Cast su tipi primitivi

```
double altezza = 6.46;
int altezzaApprossimata = (int) altezza; // troncamento;
char c = (char) 64;
float f = 54;
float piGreco = (float) 3.1415 // le costanti reali sono double
```

Vi sono due tipi di casting:

- per estensione. Quando un tipo é convertito ad un tipo piú ampio. Ad esempio `int` convertito a `long`. Sempre applicabile anche sotto forma implicita.
- per restrizione. Quando un tipo piú ampio é convertito ad un tipo piú stretto, é possibile perdita di informazione. Ad esempio un reale convertito ad intero.

Cast su tipi definiti dall'utente

Se si vuole "camuffare" un oggetto specializzato con un oggetto meno specializzato (non é possibile il viceversa) si puó usare il cast applicato ad oggetti.

```
class A { int a() { ... } }            // definzione di a()
class B extends A { int a() { ... } } // overwriting di a()
class C extends A { int a() { ... } } // overwriting di a()
class Test {
  public static void main(String [] argv) {
    A x = new A(); B y = new B(); C z = new C();
    z.a();       // invocazione del metodo a() di C
    ((A)z).a(); // invocazione del metodo a() di A
    ((B)z).a(); // non \'e possibile conoscere a() di B
    ((C)x).a(); // non \'e possibile applicare a() di C
  }
}
```

Come si comporta il polimorfismo

Cose che si possono e che non si possono fare. Consideriamo le seguente due classi:

```
public class A {
  public String m() { return "m in A"; }
  public String n() { return "n in A"; }
}
```

e

```
public class B extends A {
  public String m() { return "m in B"; }
  public String k() { return "k in B"; }
}
```

Come si comporta il polimorfismo

```
import java.io.*;
public class Test {
  public static void main() {
    PrintStream p = System.out;
    A a = new A();            // m e n
    B b = new B();            // m e k
    A ab = new B();           // m e n senza cast
    // B ba = new A(); errore in compilazione
    p.println(a.m());         // m in A
    p.println(b.m());         // m in B
    p.println(ab.m());        // m in B
    // p.println(ab.k()); errore in compilazione
    p.println(((B)ab).k()); // k in B
    p.println(ab.n());        // n in A
  }
}
```

Interfacce

Java non permette l'ereditarietá multipla, ovvero specializzare piú di una classe.

Tuttavia esiste un meccanismo che permette di avere una forma molto simile, l'implementazione di interfacce.

```
interface Disegnabile {
  final int DIMX = 800, DIMY = 600;
  public plot();
}
```

Le interfacce hanno per lo piú il significato di specificare delle proprietá.

```
class Cerchio extends Ovale implements Disegnabile {
  // attributi e metodi
}
```

La classe `Cerchio` ha l'obbligo di dare corpo al metodo `plot()`.

Gerarchie di interfacce

Le interfacce possono essere specializzate come le classi. Si possono quindi definire gerarchie di interfacce e definire interfacce che non possono essere ulteriormente specializzate mediante l'uso di `final`

Come di dichiara una interfaccia

La dichiarazione di una interfaccia é molto simile alla dichiarazione di una classe astratta. Si possono dichiarare solo *signature* di metodi (dichiarazione senza corpo) e costanti.

```
class Cerchio extends Ovale implements Disegnabile, Stampabile {
  // attributi e metodi
}
```

La classe `Cerchio` puó "implementare" piú di una interfaccia

Uso delle interfacce

Non possono essere creati oggetti di tipo interfaccia:

```
new Disegnabile() // errato
```

Possono essere dichiarate variabili di tipo interfaccia:

```
Disegnabile x, y; // corretto
Disegnabile z = new Disegnabile(); // errato
```

Possono essere assegnate a variabili di tipo interfaccia qualsiasi classe che implementa l'interfaccia:

```
Punto centro = new Punto(2.3,3.4);
Disegnabile x = new Cerchio(centro, 10.0);
```

Uso delle interfacce

Possono essere invocati solo i metodi definiti nell'interfaccia:

```
Punto centro = new Punto(2.3,3.4);
Disegnabile x = new Cerchio(centro, 10.0);
x.plot(); // corretto
x.circonferenza(); // errato
```

Esempio 1

Si vuole realizzare un meccanismo di ordinamento che possa essere applicato, senza modifiche, ad oggetti di tipo diverso. Utilizziamo l'interfaccia `Comparable` dichiarata in `java.lang` su due classi da noi definite `Persona` e `Auto`.

```
class Persona implements Comparable {
  private String nome, cognome;
  Persona(String n, String c) { nome = n; cognome = c; }
  public String getNome() { return nome; }
  public String getCognome() { return cognome; }
  public String toString() { return nome + " " + cognome; }
  public int compareTo(Object o) { // in Comparable
    return cognome.compareTo(((Persona)o).getCognome());
  }
}
```

Esempio 2

```
class Auto implements Comparable {
  private int cavalli;
  private String nome;
  Auto(String n, int c) { nome = n; cavalli = c; }
  public String getNome() { return nome; }
  public int getCavalli() { return cavalli; }
  public String toString() { return nome + " " + cavalli; }
  public int compareTo(Object o) { // in Comparable
    return cavalli - ((Auto)o).getCavalli();
  }
}
```

Esempio 3

```
class Esempio {
  public static void main(String [] args) {(new Esempio()).run();}
  private void run() {
    Comparable [] x = new Comparable[4];
    x[0] = new Persona("Paolo","Rossi");
    x[1] = new Persona("Giuseppe","Garibaldi");
    x[2] = new Persona("Lorella","Cuccarini");
    x[3] = new Persona("Margherita","Hack");
    ordina(x); stampa(x);
    x[0] = new Auto("Panda",60);
    x[1] = new Auto("Mercedes C63",457);
    x[2] = new Auto("Ferrari 458",570);
    x[3] = new Auto("Fiesta",75);
    ordina(x); stampa(x);
  }
```

Esempio 3

```
  private void ordina(Comparable [] v) {
    for (int i = 0; i < v.length - 1; i++)
      for (int j = i + 1; j < v.length; j++) {
         if (v[i].compareTo(v[j]) > 0) {
           Comparable t = v[i]; v[i] = v[j]; v[j] = t;
         }
      }
  }
  private void stampa(Comparable [] v) {
    for (int i = 0; i < v.length; i++) System.out.println(v[i]);
  }
}
```

Eccezioni

- Cosa sono le eccezioni. Sono flussi alternativi di esecuzione del programma in presenza di particolari condizioni (previste o impreviste).
- Come usare le eccezioni. Le eccezioni devono essere usate per rendere il programma piú robusto e affidabile.
- Gerarchia delle eccezioni. Le eccezioni sono normali oggetti appartenenti ad una gerarchia.
- I costrutti per la gestione: `try`, `catch`, `finally`, `throw`, `throws`.
- Creazione di eccezioni. L'utente puó creare proprie eccezioni per rendere piú chiaro il programma.

Eccezioni

```
import java.io.*;
public class Lettura {
  public static void main(String argv[]) {
    float f;
    try {
    BufferedReader d
          = new BufferedReader(new InputStreamReader(System.in));
    String s = new String(d.readLine()); // IOException
    f = Float.parseFloat(s);              // NumberFormatException
    } catch (Exception e) { System.out.println("Errore!"); }
    System.out.println(f);
  }
}
```

Aritmetic Exception

```
import java.io.*;
class CalcoloConDivisione {
  public static void main(String argv[]) {
    Calcolo x = new Calcolo(10), y = new Calcolo(20);
    try {
      x.add(10); y.divisione(10); x.divisione(21);
    } catch (AritmetichException e) { System.out.println("E!"); }
  }
}
class Calcolo {
  private int a;
  Calcolo(int x) { a = x+1; }
  public void add(int n) { a += n; }
  int divisione(int d) throws AritmetichException {
    return a/(a-d); // se d \'e 0 l'eccezione \'e propagata
}
```

Eccezioni diverse

```
import java.io.*;
public class LetturaReale {
  public static void main(String argv[]) {
    float f;
    try {
    BufferedReader d
          = new BufferedReader(new InputStreamReader(System.in));
    String s = new String(d.readLine()); // IOException
    f = Float.parseFloat(s);             // NumberFormatException
    } catch (IOException e) { System.out.println(e);
    } catch (NumberFormatException e) { System.out.println(e); }
    System.out.println(f);
  }
}
```

Programmare con eccezioni

Un buon programma dovrebbe avere eccezioni specifiche.

```
class Temperatura throw TemperaturaBassa, TemperaturaAlta {
  private float t;
  Temperatura(float n) {
    if (n < 34) throw new TemperaturaBassa(n);
    else if (n > 43) throw new TemperaturaAlta(n);
         else t = n;
  }
  public int get() { return t; }
  public void inc() {
    float x = t + 1;
    if (x < 34) throw new TemperaturaBassa(x);
    else if (x > 43) throw new TemperaturaAlta(x);
         else t = x;
   }
}
```

Creare eccezioni

```
class TemperaturaBassa extends Exception {
  TemperaturaBassa(float n) {super(n); }
}
class TemperaturaAlta extends Exception {
  TemperaturaAlta(float n) { super(n); }
}
```

Usare eccezioni

```
class UsoDiTemperatura {
  public static void main(String [] argv) {
    Temperatura a, b;
    try {
      a = new Temperatura(36.5);
      b = new Temperatura(45.0);
    } catch (TemperaturaAlta e) {
        System.out.println("Bagno freddo per "+e); }
      catch (TemperaturaBassa e) {
        System.out.println("Coperta per "+e); }
  }
}
```

Lettura e scrittura su files

Vi sono diversi modi di gestire i files. A seconda delle necessitá o delle caratteristiche si usano librerie opporture. La forma piú semplice é (il programma é memorizzato in un file di nome "LetturaFile.java"):

```
import java.io.FileReader;
import java.util.Scanner;
public class LetturaFile {
  public static void main(){
    try {
      Scanner in = new Scanner(new FileReader("LetturaFile.java"));
      while (in.hasNextLine()) System.out.println(in.nextLine());
      in.close();
    } catch (FileNotFoundException e)
      { System.out.println("Il file non esiste");}
  }
}
```

Lettura con espressioni regolari

Scanner include la possibilitá di utilizzare espressioni regolari per filtrare il flusso da leggere

```
import java.util.Scanner;
public class LetturaConFiltro {
  public static void main(String[] argv) {
    String input = "Sono il 33-esimo gatto del 77-esimo piano";
    Scanner in = new Scanner(input);
    in.useDelimiter("[- ]");
    while(in.hasNext())
      if (in.hasNextInt()) System.out.println(in.nextInt());
      else in.next();
    in.close();
  }
}
```

Lettura da file

```
import java.io.*;
public class LetturaDaFile {
  public static void main(String argv[]) {
    BufferedReader in;
    try {
        in = new BufferedReader(new FileReader("text.in"));
    } catch (FileNotFoundException e)
      { System.out.println("Il file non esiste"); }
    String line;
    while (true) {
      try { line = in.readLine(); }
      catch (IOException e) { System.out.println("Errore"); }
      System.out.println(line);
  }
}
```

Scrittura su file

```
import java.io.*;
public class ScritturaSuFile {
  public static void main(String argv[]) {
    PrintWriter out;
    try {
       out = new PrintWriter("text.out");
    } catch (IOException e)
      { System.out.println("Il file non pu\'o essere creato");
    try {
      out.println("Paolo"); out.println("Rossi");
      out.println(1960); out.println(3.14);
    }
      catch (IOException e) { System.out.println("Errore"); }
  }
}
```

Tipi generici

Nella versione 5 di Java (29 settembre 2004) sono introdotti i tipi generici che permettono di creare collezioni di un tipo permettendo il controllo sui tipi a tempo di compilazione ed evitare di utilizzare l'operatore di cast:

```
List lista = new ArrayList();
lista.add("stringa1"); lista.add("stringa2");
String s1 = (String)lista.get(0);
StringBuffer s2 = (StringBuffer)lista.get(1);
```

Con Java 1.5 lo stesso codice puó essere scritto nel modo seguente:

```
List<String> lista = new ArrayList<String>();
lista.add("stringa1"); lista.add("stringa2");
String s1 = lista.get(0);
StringBuffer s2 = lista.get(1);
```

Numero variabili di argomenti

Sempre dalla versione 5 in poi é possibile definire un numero arbitrario di argomenti dello steso tipo nella definizione di un metodo

```
public static void stampa(String ... args) {
    for (int i=0; i <args.length; i++) System.out.println(args[i]);
}
//...
stampa("Questo", "e", "quello");
stampa("Questo", "e", "quello", "e", "quell'altro");
```

Output formattato

Sempre dalla versione 5 é stata aggiunta alla classe `System` la funzione `printf` che permette una formattazione dell' output simile all'omonima funzione del linguaggio C. La segnature é

```
public static void printf(String s, Object ... objs)
```

Ad esempio:

```
int i = 10; double d = 3.14; String s = "ciao";
System.out.printf("Intero %i, reale %f e stringa %s",i,d,s);
```

Commenti documentali

Per agevolare l'uso di decine di miglia di classi prodotte da diversi programmatori, sin dall'inizio é stato definito un linguaggio per la generazione automatica di documentazione mediante l'uso di uno strumento *JavaDoc*. Tal linguaggio é utilizzato all'interno di commenti e quindi ignorato dal compilatore. *JavaDoc* legge del file del codice solo i commenti e genera pagine HTML di documentazione

I commenti documentali sono posti subito prima della dichiarazione di una classe/interfaccia/enumerazione o subito prima della dichiarazione di un metodo o costruttore

Tag di commento [1]

Tag	*Descrizione*
`@author`	Nome dello sviluppatore
`@deprecated`	indica che l'elemento potrá essere eliminato da una versione successiva del software
`@exception`	Indica eccezioni lanciate da un metodo (vedi @throws)
`@throws`	Indica eccezioni lanciate da un metodo. Sinonimo di @exception introdotto in Javadoc 1.2.
`@link`	Crea un collegamento ipertestuale alla documentazione locale o a risorse esterne (tipicamente internet)

Tag di commento [2]

`@param`	Definisce i parametri di un metodo. Richiesto per ogni parametro
`@return`	Indica i valori di ritorno di un metodo. Questo tag non va usato per metodi o costruttori che producono void
`@see`	Indica un'associazione a un altro metodo o classe
`@since`	Indica quando un metodo é stato aggiunto a una classe
`@version`	Indica il numero di versione di una classe o un metodo

Esempio di commento

```
/** Questa classe esegue semplici calcoli aritmetici
      @author Rosario Culmone
      @version 1.0 15/10/2011 */
class Elemento throws Exception {
  private int n; // commento non documentale
  /** Il costruttore pu\'o fallire se il parametro \'e errato
         @param a Il valore a cui si vuol inizializzarare
         @exception Exception Produce una eccezione se il
         parametro passato \'e negativo */
  Elemento(int a) throws Exception {
    if (a > 0) n = a; else throw new Exception();
  }
}
```

Commento di metodo

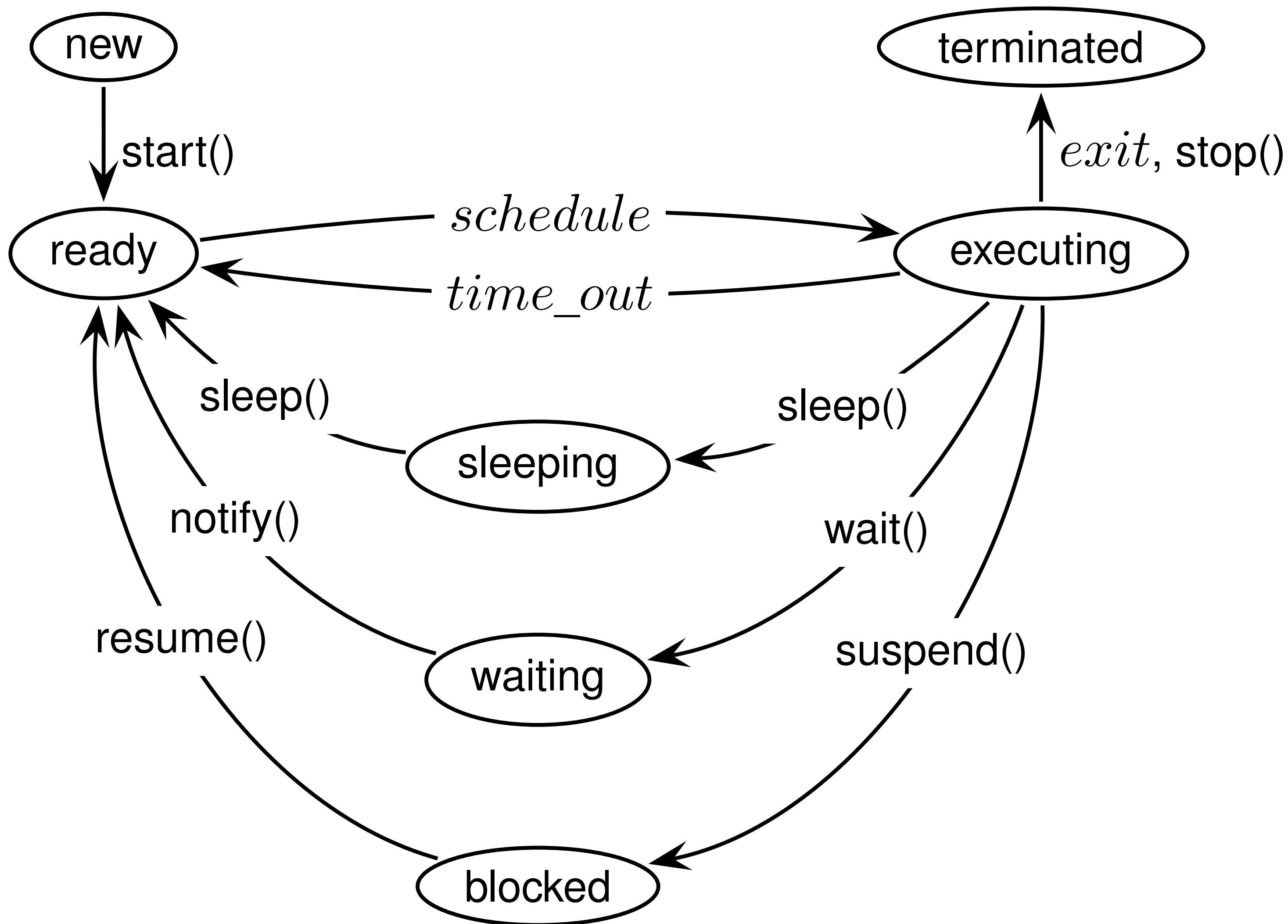
```
/** L'operazione diff effettua la differenza tra due @see Elemento
    @param a L'Elemento da sottrarre
    @return Produce un nuovo elemento con la differenza
    @exception Exception Se la differenza \'e negativa \'e
    lanciata un'eccezione
    @version 1.1
    @since E' presente sin dalla verione 1.0 */
  public Elemento(Elemento a) throws Exception {
    int x = n - a.getValue();
    if (x > 0) return new Elemento(x) else throw new Exception();
  }
}
```

Concorrenza in Java

Java fin dalle origini é stato accompagnato da librerie che permettono la gestione della concorrenza di flussi di esecuzione. In particolare nella libreria `java.lang` é presente da sempre la classe `Thread` e l'interfaccia `Runnable`.

In Java é possibile realizzare programmi che eseguono "simultaneamente" flussi distinti. Per un corretto uso di tali librerie é necessario possedere conoscenze di concetti di concorrenze e sincronizzazione che esulano da questo corso ma che sono trattati nei corsi di Sistemi operativi, Basi di dati e reti di calcolatori.

Stati



Threads

```
class PingPong extends Thread {
  private String nome;
  private int ritardo;
  public static void main(String [] argv) {
    new PingPong("Ping",33).start();
    new PingPong("Pong",45).start();
  {
  PingPong(String chi, int quanto) {
    nome = chi; ritardo = quanto;
  }
  public void run() {
    try {
      for (;;) { System.out.println(nome); sleep(ritardo); }
    } catch (InterruptedException e) { return; }
  }
}
```

syncronized

```
import java.util.*;
class Volo {
  private int posti;
  private String nome;
  public Volo(String n, int p) { nome = n; posti = p; }
  public int disponibili() { return posti; }
  public boolean syncronized prenota() {
    Scanner tastiera = new Scanner(System.in);
    System.out.println(nome+" disponibili "+posti+" prenota: ");
    int p = tastiera.nextInt();
    if (posti-p >= 0) { posti-=p; return true; }
    else return false;
  }
}
```

wait e notify

```
class Conta {
  private int numero;
  public syncronized void pi\'u() {
    numero++; System.out.println(conta+" pi\'u uno");
    if (conta == 1) notify();
  }
  public syncronized void meno() {
    try {
      while (conta == 0) wait();
      conta--;
      System.out.println(conta+" meno uno");
    } (InterruptedException e) { return; }
  }
}
```

Runnable

```
import java.awt.*;  import java.awt.event.*; import javax.swing.*;
public class G extends JFrame implements ActionListener, Runnable {
  public G() {
    super("Grafica"); setSize(400,300);
    JButton b = new JButton("Start"); JLabel l = new JLabel();
    getContentPane().add(b); getContentPane().add(l);
    b.addActionListener(this);
  }
  public run() {
    int c = 0; for (;;) {l.setText(String.valueOf(c)); delay(100); }
  }
  public void actionPerformed(ActionEvent e){new Thread(this).start();}
  public static void main(String [] argv) {
    G e = new G(); e.setVisible(true);
  }
}
```

Files di properties

I dati possono essere memorizzati secondo dei formati standard di modo che programmi diversi possano gestire gli stessi files. I formati piú comuni sono:

- ASCII, ovvero files che contengono byte che codificano caratteri
- RTF, ovvero Rich Text Format. Sono files che memorizzano testi formattati
- CSV, ovvero Comma Separated Value. Sono files che memorizzano elenchi
- XML, ovvero Extensible Markup Language. Sono file per la gestione di dati generici.
- PROPERTIES. Sono file per la gestione di dati generici non strutturati.

Tutti questi formati hanno la caratteristica di essere "human-readable" ovvero leggibili e stampabili. In particolare i file di properties sono coppie di *chiave=valore*. Ad esempio

```
# Questo \'e un file di properties
web = http://www.unicam.it
auto = berlina
```

Gestione dei files di properties

Spesso i programmi usano files per memorizzare configurazioni o dati temporanei.

```
import java.util.Properties;
import java.io.*;
class FileDiProperties {
   public static void main(String [] args) {
      Properties p = new Properties();
      InputStream f = new FileInputStream(propertyFileName);
      p.load(f);
      String modello = p.getProperty("Auto");
      if (modello.equals("berlina")) p.setProperty("Auto","coupe");
      p.store(f,"Versione 2.3");
      f.close();
   }
}
```

Package

Tra le librerie fornite da SUN vi sono

- `java.lang`, é importata automaticamente, il compilatore aggiunge automaticamente `import java.util.*`. Contiene classi fondamentali per il funzionamento di ogni programma come `String`. In particolare le classi wrapper come `Integer, Char, Float, Double, Boolean`
- `java.util`. Contiene classi di uso generale come `Vector, HashTable`
- `java.math`. Contiene le funzioni matematica come `sin, cos, tan, log`
- `java.io`. Contiene le classi per la gestione dei files
- `java.applet`. Contiene le classi per la realizzazione dei programmi Applet
- `java.awt`. Contiene le classi per la realizzazione di programmi con interfaccia grafica.

Object

E' la classe progenitrice di ogni classe. Alcuni metodi forniti da `Object`

- `Object clone()`, crea una copia esatta dell'oggetto a cui é applicata.
- `boolean equals(Object obj)`, confronta due oggetti. E' spesso sovrascritta.
- `String toString()`, produce una stringa di commento. E' spesso sovrascritta.

Uso di Object

```
public class VettoreDiObject {
  public static void main(String argv[]) {
    final int DIM = 3;
    Object [] vettore = new Object[DIM];
    vettore[0] = new Integer(10);
    vettore[1] = new String("Ciao mamma");
    vettore[2] = new Double(3.14);
    for (int i = 0; i < DIM; i++) System.out.println(vettore[i]);
  }
}
```

Uso di toString

```
public class UsoDiToString {
  public static void main(String argv[]) {
    temperaturaCorporea maria = new temperaturaCorporea(36.5);
    temperaturaCorporea paolo = new temperaturaCorporea(39);
    System.out.println(maria);
    System.out.println(paolo);
  }
}
class temperaturaCorporea{
  private float t;
  temperaturaCorporea(float t) {
    if (t > 30 && t < 45) this.t = t; // else errore
  }
  String toString() { return (t > 37 ? "febbre "+t : t; }
}
```

Uso di StringTokenizer

```
public class UsoDiStringTokenizer {
  public static void main(String argv[]) {
  StringTokenizer st =
    new StringTokenizer("Paolo;Rossi;via Roma;37",";");
     while (st.hasMoreTokens()) {
         System.out.println(st.nextToken());
     }
   }
}
```

Uso di equals

```
public class UsoDiEquals {
  public static void main(String argv[]) {
    Persona paolo1 = new Persona("Paolo","Rossi");
    Persona paolo2 = new Persona("Paolo","Bianchi");
    System.out.println(paolo1.equals(paolo2));
  }
}
class Persona{
  private String nome, cognome;
  Persona(String n, String c) {n = nome; c = cognome; }
  public boolean equals(Persona p) {
    return n.equals(p.getNome()) && c.equals(p.getCognome());
  }
  public String getNome() { return new String(nome); }
  public String getCognome() { return new String(cognome); }
}
```

Vector

Si trova nel package `java.util`. Ha le stesse funzionalitá degli array ma senza il limite delle dimensioni predefinite. I principali metodi sono:

- `void add(Object)`, aggiunge un elemento in fondo al vettore.
- `boolean contains(Object)`, verifica nel vettore é presente un elemento dato. Il confronto é effettuato invocando il metodo `equals`.
- `Object elementAt(int)`, produce l'elemento nella posizione fornita.
- `int indexOf(Object)`, produce la posizione dell'elemento fornito, se presente nel vettore. Il confronto é effettuato invocando il metodo `equals`.
- `int size()`, produce il numero di elementi presenti nel vettore.
- `void removeElementAt(int)`, cancella il riferimento alll'elemento della posizione fornita (non l'oggetto!)

Classi wrapper

Sono le classi corrispondenti ai tipi di dato primitivi

(`int`, `float`, `double`, `char`, `boolean`, `long`, `short`). Servono per utilizzare i tipi di dato primitivi la dove é richiesto un oggetto (per esempio in `Vector`)

- `Short`, corrispondente al tipo primitivo `short`.
- `Integer`, corrispondente al tipo primitivo `int`.
- `Long`, corrispondente al tipo primitivo `long`.
- `Boolean`, corrispondente al tipo primitivo `boolean`.
- `Character`, corrispondente al tipo primitivo `char`.
- `Float`, corrispondente al tipo primitivo `float`.
- `Double`, corrispondente al tipo primitivo `double`.

Uso di Vector

```
import java.util.Vector;
import java.io.IOException;
class UsoDiVector{
  public static void main(String args []) throws IOException {
    Vector e = new Vector();
    System.out.println("Lettere e numeri, \".\" per finire");
    char i = (char) System.in.read();
    while (i != '.') {
      if (Character.isDigit(i)) e.add(new Integer(i));
      if (Character.isLetter(i)) e.add(new Character(i));
      i = (char) System.in.read();
    }
    for (int j = 0; j < e.size(); j++)
      System.out.println(e.elementAt(j));
  }
}
```

Interfacce grafiche

Java é conosciuto per l'ampia disponibilitá di oggetti grafici. Nel pacchetto `java.awt` sono presenti molti oggetti grafici di uso comune:

```
Frame finestra = new Frame(); // crea una finestra grafica
finestra.setTitle("Programma esempio"); // imposta il titolo
finestra.Location(100,100); // posiziona la finestra
```

Oggetti grafici diffusi sono:

```
Button bottone = new Button("OK"); // bottone
Choice scelta = new Choice(); // men\'u di scelta
TextField testo = new TextField(); // campo di testo
Dialog messaggio = new Dialog(); // messaggio
```

Componenti grafiche

Ogni oggetto grafico ha i propri metodi per la gestione. Ad esempio il menú a tendina l'operazione per impostare le scelte:

```
Choice scelta = new Choice(); // men\'u di scelta
scelta.add("Scelta 1");
scelta.add("Scelta 2");
scelta.add("Scelta 3");
```

Gli oggetti grafici sono nel pacchetto:

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

Gestione ad eventi con delega

Le operazioni che vengono effettuate sugli oggetti grafici vengono intercettate e gestite con un meccanismo che si chiamo "gestione ad eventi con delega"

```
class AscoltaItemListener implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
      System.out.println("Selezionato " + e.getItem());
    }
}
```

Si definisce una classe che implementa i metodi definiti in una interfaccia specifica per un determinato tipo di evento. Successivamente all'oggetto grafico chi é il suo gestore

```
scelta.addItemListener(new AscoltaItemListener());
```

Le interfacce e altro per la gestione degli eventi si trova su:

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

Esempio di grafica con AWT [1]

Si vuole realizzare un programma che conti il numero di volte che é premuto un bottone.

```
import java.awt.*;
import java.awt.event.*;
public class EsempioAWT
       extends Frame
       implements ActionListener, WindowListener {
  private int clicks = 0;
  private Label label;
  private Button button;
  private String text;
```

Esempio di grafica con AWT [2]

```
EsempioAWT() {
  text = new String("Numero di click ");
  label = new Label(text);
  button = new Button("Click");
  button.addActionListener(this);
  addWindowListener(this);
  setLayout(new GridLayout(2,1));
  add(button);
  add(label);
  pack();
  setVisible(true);
}
```

Esempio di grafica con AWT [3]

```
  public void actionPerformed(ActionEvent e) { // ActionListener
    clicks++; label.setText(text + clicks);
  }
  public void windowActivated(WindowEvent e) {} // WindowListener
  public void windowClosed(WindowEvent e) {}
  public void windowClosing(WindowEvent e) {System.exit(0); }
  public void windowDeactivated(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowOpened(WindowEvent e) {}

  public static void main(String[] args) { new EsempioAWT (); }
}
```

Correttezza di programmi

Non si puó dimostrare che un programma é corretto ma si puó fare molto per renderlo affidabile. Un metodo é adottare la progettazione per contratto. Si definisce *prima* quello che si realizzerá *dopo*. Ad esempio:

Radice quadrata [1]

```
import java.util.*;
class DesignByContract {
  public static void main(String [] args) {
    boolean next; double x = 0.0;
    DesignByContract p = new DesignByContract();
    Scanner in = new Scanner(System.in);
    do { try {
         next = false;
         System.out.print("Imposta numero: ");
         x = in.nextDouble();
         if (x == 0.0) break;
         System.out.println(p.sqrt(x));
       } catch (Exception e) {in.nextLine(); next = true;}
    } while(next);
  }
```

Radice quadrata [2]

```
  public double sqrt(double x) {
    double r = 0.0;
    /* pre condizioni */
    assert x > 0.0;
    /* algoritmo */
    assert x <= r * r;
    /* post condizioni */
  return r;
  }
}
```

Lambda espressioni

xxx