JAVA

CAPITOLO 2 – USO DEGLI OGGETTI

Un oggetto in java è caratterizzato da:

- STATO: insieme delle proprietà che caratterizzano un oggetto in un determinato momento
- COMPORTAMENTO: insieme delle azioni possibili all'oggetto

Ogni azione possibile ad un oggetto può essere "iniziata" inviandogli un MESSAGGIO corredato dalle informazioni necessarie, dette ARGOMENTI.

L'insieme di tutti i messaggi che un oggetto è in grado di interpretare è detto INTERFACCIA.

Per ottenere un oggetto di una classe bisogna richiederlo alla classe stessa invocando il servizio COSTRUTTORE.

LA CLASSE "Frazione"

Frazione f = new Frazione(2,3)

F conterrà IL RIFERIMENTO ad un oggetto di tipo frazione, conterrà cioè l'indirizzo al quale potremo recapitare i messaggi per l'oggetto

TIPO	METODO	DESCRIZIONE
int	.compareTo(frazione)	Neg se minore, pos se maggiore, 0 se uguale
frazione	.diviso(frazione)	
frazione	.piu(frazione)	
frazione	.meno(frazione)	
frazione	.per(frazione)	
boolean	.equals(frazione)	True se uguali, se no false
int	.getNumeratore	Restituisce numeratore
int	.getDenominatore	Restituisce denominatore
boolean	.isMaggiore(frazione)	True se vero, se no false
boolean	.isMinore(frazione)	True se vero, se no false
string	.toString(frazione)	Restituisce stringa che rappresenta la frazione inserita

LA CLASSE “ConsoleOutputManager”

ConsoleOutputManager out = new ConsoleOutputManager();

out.println();

È un costruttore che non chiede argomenti, crea solo un canale di comunicazione con il monitor, tramite una variabile che chiamiamo “out”.

LA CLASSE “ConsoleInputManager”

ConsoleInputManager in = new ConsoleInputManager();

Ricordiamo che String è un “tipo di riferimento”, cioè contiene l’indirizzo alla cella di memoria contenente l’oggetto a cui si riferisce, al contrario dei “tipi primitivi” come ad esempio int, i quali contengono direttamente il valore

TIPO	METODO	DESCRIZIONE
char	.readChar()	Legge un carattere
double	.readDouble()	Legge reale in doppia precisione
Int	.readInt()	Legge un intero
String	.readLine()	Legge stringa di testo
long	.readLong()	Legge intero di tipo long
boolean	.readSiNo()	Legge valore di tipo Si/No

LA CLASSE “String”

String s = new String(“ciao”) NO! Basta fare String s = “ciao”

TIPO	METODO	DESCRIZIONE
String	.toUpperCase	Restituisce stessa stringa ma in maiuscolo
String	.toLowerCase	Restituisce stessa stringa ma in minuscolo
Int	.length	Restituisce lunghezza stringa
String	.concat(string)	Restituisce stringa formata dall’unione delle due stringhe (o usa “+”)
String	.substring(int1, int2)	Restituisce parte della stringa (caratteri da int1 a int2)
String	.substring(int)	Restituisce parte della stringa(caratteri da int a fine)
int	.compareTo(string)	Restituisce 0 se uguali, >0 se chi esegue è prima in ord.alf., <0 se dopo

CAPITOLO 3 – SELEZIONE E ITERAZIONE

L'ISTRUZIONE "if – else"

```
if(condizione)
  istruzione1;
else
 istruzione2;
```

Il CONFRONTO in java si esegue con

== , > , >= , < , <= , !=

Quindi scriverò ad esempio: if(x == 2)

OPERATORI BOOLEANI

- AND: &&
- OR: ||
- NOT: !

LEGGI DI DE MORGAN

! (x && y) = !x || !y

! (x || y) = !x && !y

LAZY EVALUATION

In java gli operatori || e && usano una modalità di valutazione detta "lazy evaluation": se avendo già valutato la prima parte di un equazione booleana si può già determinare il risultato dell'espressione, la parte restante non viene valutata

CICLI "While" e "do – While"

```
While(condizione)
  Istruzione;
```

```
do{istruzione  }
while(condizione)
```

Se si vuole terminare un ciclo si può utilizzare l'istruzione "**break**"

L'istruzione "**continue**" invece blocca l'esecuzione del blocco di istruzioni interne al ciclo e passa all'esecuzione del ciclo successiva

CICLO “For”

```
for(int i= 0;  i < 10;  i++, i+2,…) {
istruzioni;  }
```

LA CLASSE “Random”

Contiene generatori di numeri con distribuzione casuale

```
Random = new Random();
int a = random.nextInt();
```

TIPO	METODO	DESCRIZIONE
int	.nextInt()	Restituisce intero casuale
int	.nextInt(int)	Restituisce intero casuale tra 0 e int
int	.nextInt(int1)+int2 [prima dichiaro int1=int-int2	Restituisce intero casuale tra int1 e int2
boolean	.nextBoolean()	Restituisce true o false
float	.nextFloat()	Restituisce float casuale tra 0.0 e 1.0
long	.nextLong()	Resituisce intero di tipo long casuale
double	.nextDouble()	Restituisce reale in doppia precisione casuale
byte	byte[] b = new byte[10]; random.nextByte(b);	riempie l’array di byte casuali

CAPITOLO 4 – TIPI PRIMITIVI E TIPI ENUMERATIVI

OPERATORI:

- ARITMETICI: **+** , **-** , **/** , **%** , *****
- INCREMENTO/DECREMENTO: **++** , **--** , **+=** , **=+** , **-=** , **=-**
- RELAZIONALI: **>** , **<** , **>=** , **<=** , **==** , **!=** (== , != possono confrontare anche riferimenti ad oggetti)
- LOGICI: **&**(AND logico), **|**(OR logico), **^**(OR esclusivo), **!**(NOT), **&&**(AND condizionale Lazy Evaluation), **||**(OR condizionale Lazy Evaluation)
- CONDIZIONALE: **?** (esempio: max = x < y ? y : x)
- ASSEGNAMENTO: **=**
- CONCATENAZIONE: **+** (quando almeno uno dei due operandi è di tipo String)
- ISTANZIAZIONE: **new *nomecostruttore*** (da come risultato il riferimento all'oggetto)

TIPI NUMERICI INTERI:

- **byte** : rappresentazione su 8 bit, valori da -128 (-2^7) a +127 ($2^7 - 1$)
- **short**: rappresentazione su 16 bit, valori da -32768 (-2^{15}) a 32767 ($2^{15} - 1$)
- **int**: rappresentazione su 32 bit, valori da -2147483648 (-2^{31}) a 2147483647 ($2^{31} - 1$)
- **long**: rappresentazione su 64 bit, valori da -2^{63} a $2^{63}-1$

TIPI NUMERICI IN VIRGOLA MOBILE:

- **float**: in singola precisione. Usa rappresentazione 32bit: $(-1)^{segno} * mantissa * 2^{esponente}$
- **double**: in doppia precisione. Usa rappresentazione 64bit. La precisione di rappresentazione aumenta in base al numero di bit utilizzati

TIPI BOOLEANI:

- **boolean**: serve a rappresentare due valori: vero o falso (true o false)

TIPI CARATTERE:

- **char**: utilizza 16bit

Sono consentite conversioni esplicite, cioè **promozioni**, di un tipo ad un altro più ampio

Per passare invece da un tipo più ampio ad uno più ristretto dobbiamo effettuare il **casting esplicito**

IL TIPO “Char”

TIPO	METODO	DESCRIZIONE
char	.charAt(int)	Restituisce il carattere nella posizione specificata

LE CLASSI INVOLUCRO

Nel package della libreria standard java.lang esiste una classe associata a ogni tipo primitivo, che permette di **rappresentare i dati di quel tipo come oggetti**. Queste sono dette “classi involucro”; in esse sono inoltre contenuti campi e metodi utili a trattare gli oggetti del tipo o anche i dati primitivi.

CLASSE INVOLUCRO “Integer”

Costruttori:

- **public Integer(int)** costruisce un nuovo oggetto che rappresenta il numero intero fornito
- **public Integer(String)** costruisce un nuovo oggetto che rappresenta l’int, fornito sotto forma di stringa (se la stringa inserita non è un numero intero da errore)

TIPO	METODO	DESCRIZIONE
int	.compareTo(integer)	Confronta: 0 se uguali, <0 se minore del fornito, >0 se maggiore
int	.intValue()	Restituisce int uguale al valore rappresentato dall’oggetto
int	.parseInt(string)	Restituisce int uguale al valore rappresentato nella stringa

CLASSE INVOLUCRO “Character”

Costruttori:

- public Character(char)

TIPO	METODO	DESCRIZIONE
boolean	.isDigit(char)	Restituisce true se char è una cifra
boolean	.isLetter(char)	Restituisce true se char è una lettera
boolean	.isLowerCase(char)	Restituisce true se char è lettera minuscola
boolean	.isUpperCase(char)	Restituisce true se char è lettera maiuscola
boolean	.isLetterOrDigit(char)	Restituisce true se char è una lettera o una cifra
Char	.toLowerCase(char)	Restituisce char in minuscolo
Char	.toUpperCase(char)	Restituisce char in maiuscolo

CLASSE “StringTokenizer”

E’ un oggetto che permette di estrarre sottostringhe (dette Token, delimitate da un simbolo) dalla stringa fornita come argomento.

Costruttore: **StringTokenizer a = new StringTokenizer(string, “char”)**

Ex:
```
StringTokenizer sequenza = new Sringtokenizer(s , “ “);
Int somma = 0;
while(sequenza.hasMoreTokens() ) {
String token = sequenza.nextToken();
}
```

TIPO	METODO	DESCRIZIONE
boolean	.hasMoreTokens()	Controlla se nella stringa ci sono ancora token disponibili
string	.nextToken()	Restituisce il prossimo token della stringa associata

I TIPI ENUMERATIVI

Sono particolari classi(che si aprono con la parola chiave enum al posto di class). Non ci sono costruttori per questa classe, l’utente può solo utilizzare i riferimenti rappresentati dai valori di tipo enumerativo, senza modificarli.

Ad esempio MeseDellAnno può assumere i seguenti valori: GENNARIO , FEBBRAIO, MARZO...

È possibile accedere a valori di tipo enumerativo con la sintassi tipoenumerativo.valore

Ex: mese = MeseDellAnno.APRILE

TIPO	METODO	DESCRIZIONE
string	.name()	Restituisce il nome della costante enumerativa
int	.ordinal()	Restituisce la posizione del valore rispetto alla lista di quelli del tipo

TIPO “MeseDellAnno”

TIPO	METODO	DESCRIZIONE
string	.successivo()	Restituisce mese successivo a quello che esegue il metodo
string	.precedente()	Restituisce mese precedente a quello che esegue il metodo
Int	.numeroGiorni()	Restituisce numero dei giorni del mese (28 se febbraio)
Int	.numeroGiorni(int anno)	Restituisce numero dei giorni del mese (tiene conto dei bisest.)
int	.numeroGiorni(boolean bisestile)	Restituisce numero giorni del mese (nel caso di febbraio, se argomento=true retituisce 29, altrimenti 28)

LA CLASSE “Data”

Costruttori:

- **public Data()** costruisce una nuova data che rappresenta quella corrente
- **public Data(int g, int m, int a)** costruisce nuova data con gli argomenti specificati

TIPO	METODO	DESCRIZIONE
int	.getAnno()	Restituisce l’intero che rappresenta l’anno della data che esegue il metodo
int	.getGiorno()	
Int	.getMese()	
GiornoDellaSettimana	.getGiornoDellaSettimana()	
MeseDellAnno	.getMeseDellAnno()	
boolean	.isAnnoBisestile()	Restituisce true se l’anno della data è bisestile
int	.QuantoManca(altra Data)	Restituisce numero di giorni mancanti
string	.toString()	Rest. String che rapp. la data nel formato “gg.mm.aaaa”

L’ISTRUZIONE “Switch”

Permette di selezionare l’esecuzione di un’istruzione. Tra più possibili, in base al valore do un’espressione detta selettore

```
switch (selettore) {
   default: ……..;
    break;
    case a: ……..; }
    break;
```

in fase di compilazione valgono le seguenti regole:

- (selettore) deve essere di uno dei tipi char, byte, short, int, Character, Byte, Short, Integer, String oppure di un tipo enumerativo.
- Le espressioni usate come selettore devono essere costanti (cioè il loro valore deve essere determinabile in fase di compilazione)
- Non possono esserci due casi in cui si utilizzano costanti con lo stesso valore
- METTERE SEMPRE **BREAK** DOPO OGNI CASO, ALTRIMENTI ESEGUIRA’ ANCHE I SEGUENTI

Esempio:

```
switch(x) {
        default: out.println(“nulla”);
          break;
        case 0: out.println(“uno”);
          break;
        case1: out.println(“uno”);
          break;    }
```

OPERATORI

Operatore	Tipo degli operandi	Operazione
.	oggetto, membro	accesso a membro dell'oggetto
[ ]	array, int	accesso a elemento di array
(args)	argomenti	invocazione di metodo
++, --	variabile	incremento e decremento postfissi
++, --	variabile	incremento e decremento prefissi
+, -	numeri	più e meno unari
!	boolean	NOT
new	classe, elenco args	creazione di oggetti
(type)	tipo, qualsiasi	cast o conversione di tipo
*, /, %	numero, numero	moltiplicazione, divisione e resto
+, -	numero, numero	addizione, sottrazione
+	stringa, qualsiasi	concatenazione di stringhe
<<, >>	intero, intero	shift di bit
<, <=, >, >=	numero, numero	confronto
instanceof	riferimento, classe o interfaccia	comparazione del tipo
==, !=	primitivo, primitivo	confronto (sui valori)
==, !=	riferimento, riferimento	confronto (sui riferimenti)
&	boolean, boolean intero, intero	AND
^	boolean, boolean intero, intero	OR esclusivo
|	boolean, boolean intero, intero	OR
&&	boolean, boolean	AND (lazy)
||	boolean, boolean	OR (lazy)
?:	boolean, qualsiasi, qualsiasi	condizione
=	variabile, qualsiasi	assegnamento

CAPITOLO 5 – ARRAY E COLLEZIONI

ARRAY DI OGGETTI

Un'array è un insieme ordinato di variabili dello stesso tipo, ognuna delle quali accessibile specificando la posizione in cui si trova. Il tipo base di un array può essere sia un tipo primitivo sia un tipo riferimento (array di oggetti).

Una volta creato un array non è più possibile modificare la sua dimensione!!!

Gli array sono oggetti e vanno quindi creati con l'operatore new

```
Int[] vettore1 = new int[lunghezza vettore]
frazioni = new Frazione[4]
```

ciascun elemento dell'array è accessibile tramite un SELETTORE. Ad esempio frazioni[0] indica il primo elemento dell'array, frazioni[3] l'ultimo. Se invece uso frazioni[4] otterrò un errore in esecuzione.

È possibile inizializzare un array già in fase di dichiarazione specificando tra parentesi graffe gli elementi (la dimensione dell'array viene dedotta in seguito dal compilatore):

```
Frazione[] frazioni = {new Frazione(1,4), new Frazione(2,4), new Frazione(3,2)}
```

Per fare un ciclo for che percorra tutti gli elementi di un array si può utilizzare la forma abbreviata

```
for(Frazione f: a)
   Istruzione;
```

IL PARAMETRO DEL METODO main

public static void main(String[] args)

static indica che si tratta di un metodo statico, cioè di un servizio fornito direttamente dalla classe a cui appartiene. L'argomento del metodo, di nome **args**, è un array di **String**; il metodo non restituisce alcun valore in quanto il tipo restituito è **void**. Infine **public** è un modificatore di visibilità che indica che il metodo può essere invocato al di fuori della classe.

ARRAY DI TIPO PRIMITIVO

Mentre le posizioni di un array di oggetti contengono il riferimento ad un valore, gli array di tipo primitivo contengono direttamente il valore.

Anche per gli array primitivi è possibile specificare i valori iniziali in fase di dichiarazione:

```
char[] iniziali = {‘G’ , ‘P’, ‘N’, ‘O’};
```

ex:
```
final int MAX = 10;
int[] tabella = new int[MAX];
int pos = 0;
int x = in.readInt();
while (x != 0) {
    tabella[pos] = x;
    pos++;
    x = in.readInt()   }
```

Sottoproblemi e Sottoprogrammi

Se abbiamo bisogno di creare dei metodi ad hoc per una nostra classe, possiamo definirli all’interno della classe, come metodi statici.

I metodi statici vanno definiti come privati (con il modificatore “private”), perché il loro uso è riservato alla classe nella quale vengono definiti.

Per definire un metodo dobbiamo scriverne intestazione e corpo, cioè il codice, tra parentesi graffe. Il metodo va collocato ALL’INTERNO del corpo della classe. Il metodo poi restituirà il risultato attraverso l’istruzione “return” seguita dal risultato da restituire.

Ogni esecuzione del metodo è indipendente dalle altre: le variabili create all’interno del metodo (dette variabili locali) una volta conclusa l’esecuzione dello stesso, vengono cancellate.

Per invocare un metodo statico occorre indicare il nome della classe che offre il metodo; tuttavia se il metodo è definito nella stessa classe in cui si effettua l’invocazione, il nome della classe può essere omesso. Ad esempio possiamo scrivere *leggiVettore(lunghezza)* al posto di *SommaVettori.leggiVettore(lunghezza).*

Facciamo un esempio con un programma che:

- legge due vettori di numeri interi della stessa lunghezza
- calcola il vettore somma (dato dalla somma componente per componente)
- visualizza sul monitor i due vettori letti e il vettore somma

```
import prog.io*;

class SommaVettori {
    public static void main(String[] args) {
        ConsoleInputManager in = new ConsoleInputManager();
        ConsoleOutputManager out = new ConsoleOutputManager();

        int lunghezza = in.readInt("inserisci lunghezza vettori: ");
        out.println("lettura primo vettore: ");
        int[] vett1 = leggiVettore(in, lunghezza);
        out.println("lettura secondo vettore: ");
        int[] vett2 = leggiVettore(in, lunghezza);
        int[] somma = new int[lunghezza];

        for (int i = 0; i < somma.lenght; i++)
                somma[i] = vett1[i] + vett2[i];
        String strVett1 = generaStringa(vett1);
        String strVett2 = generaStringa(vett2);
        String strSomma = generaStringa(somma);
        out.println("Vettore 1: [" + strVett1 + "]");
        out.println("Vettore 2: [" + strVett2 + "]");
        out.println("Vettore 3: [" + strSomma + "]");
}



Private static int[] leggiVettore(ConsoleInputManager input, int lung) {
   Int[] vettore = new int[lung];
   for(int i =0; i < vettore.length; i++)
         vettore[i] = input.readInt("Elemento " + i + "? ");
   return vettore;   }


private static String generaStringa(int[] vettore) {
  String risultato  = " ":
  for (int i=0; i < vettore.length; i++) {
     risultato += vettore[i]  + (i < vettore.length – 1 ? "   "   :  " ");
 return risultato; }
}
```

LA CLASSE “Sequenza” - tipi generici

La classe Sequenza offre un costruttore privo di argomenti che costruisce una sequenza vuota, insieme ad un metodo “add” per aggiungere un elemento alla fine della sequenza.

La classe Sequenza è generica: al momento della creazione dell’oggetto sequenza è necessario specificare il tipo degli oggetti che dovrà contenere.

```
Sequenza<String> prima = new Sequenza<String>()
Sequenza<Frazione> seconda = new Sequenza<Frazione>()
```

TIPO	METODO	DESCRIZIONE
boolean	.add(E o)	Aggiunge l’oggetto in argomento alla sequenza e restituisce true
int	.size()	Rest. Numero elementi presenti nella sequenza
boolean	.isEmpty()	Rest. True se la sequenza è vuota
boolean	.contains(E o)	Rest. True se sequenza contiene un oggetto uguale all’argom.
generico	.find(E o)	Rest. Il riferimento al primo oggetto nella sequenza uguale a quello specificato, null se non è presente
boolean	.remove(E o)	Elimina dalla sequenza il primo oggetto uguale a quello specificato e rest. True; se non c’è restituisce False

Scriviamo un’applicazione in grado di leggere una sequenza di stringhe e visualizzarle

```
Import prog.io*; import prog.utili.Sequenza;

class StampaSequenza {
  public static void main(String[] args) {
     ConsoleInputManager in  = new ConsoleInputManager();
     ConsoleOutputManager  out = new ConsoleOutputManager();

     Sequenza<String> memo = new Sequenza<String>();
     String s = in.readLine();
     while (!s.equals(“ “)) {
          memo.add(s);
          s = in.readLine();   }
     for (String x: memo)
         out.println(x);
  }
}
```

LA CLASSE “SequenzaOrdinata”

Formano sequenza ordinate di oggetti. Ad’esempio se E è di tipo String, le sequenze saranno ordinate per ordine alfabetico, se sono numeri o frazioni saranno in ordine crescente, se Data allora cronologico.

LE CLASSI “Vector e Arraylist”

Vector<E> e ArrayList<E> permettono di collezionare riferimenti ad oggetti di tipo generico, accessibili in base alla posizione. Entrambe le classi forniscono un costruttore privo di argomento.

A differenza degli array, la capacità degli oggetti Vector e ArrayList viene modificata automaticamente in base alla necessità (ad esempio quando si esegue il metodo “add”).

È possibile utilizzare un ciclo “for-each” per scandire gli elementi dal primo all’ultimo.

Esempio: programma che legge una sequenza di stringhe che termina con la stringa vuota e la riscrive, prima seguendo l’ordine di inserimento, poi partendo dalla fine

```
Import prog.io*;
import java.util.ArrayList;

class ArrayListCheRovescia {
      public static void main(String[] args)  {
         ConsoleInputManager in = new ConsoleInputManager();
         ConsoleOutputManager out = new ConsoleOutputManager();

          ArrayList<String> memo = new ArrayList<String>();
          String s = in.readLine();
          while (!s.equals(“ “)) {
                 memo.add(s);
                 s = in.readLine();   }
          out.println(“Sequenza inserita: “);
          for (String x: memo)
                 out.println(x);
         out.println(“Sequenza al rovescio: “);
         for (int i = memo.size() – 1;  i >= 0;  i--)
                out.println(memo.get(i));
      }
}
```

CAPITOLO 6 – USO DELLA GERARCHIA

LA CLASSE “Rettangolo”

Rettangolo primo = new Rettangolo(double x, double y)
Dove double x sarà la base e double y l’altezza.

TIPO	METODO	DESCRIZIONE
double	.getArea()	Restituisce area rettangolo
double	.getPerimetro()	Restituisce perimetro
boolean	.equals(Rettangolo r)	Restituisce true se sono uguali
boolean	haAreaMaggiore(rettangolo r)	True se rettangolo che esegue il metodo ha area maggiore
boolean	haPerimetroMaggiore(Rett. r)	True se rettangolo che esegue il metodo ha perimetro maggiore
String	.toString()	Restituisce stringa “base = x, altezza = y”
double	.getBase()	Restituisce base del rettangolo
double	.getAltezza()	Restituisce altezza del rettangolo

Esempio inserimento base con condizione

```
While ((b = in.readDouble("base? ")) < 0)
    out.println("attenzione: la base di un rettangolo non può essere negativa);
```

abbiamo poi la classe "Quadrato" molto simile a quella Rettangolo. In particolare, un quadrato è un particolare tipo di rettangolo in cui la base e l'altezza hanno la stessa lunghezza. L'insieme dei quadrati è pertanto sottoinsieme dell'insieme dei rettangoli. Questo legame può essere evidenziato definendo la classe Quadrato come SOTTOCLASSE della classe Rettangolo. Per farlo, chi definsice la classe quadrato dovrà dichiarare nella sua intestazione che essa ESTENDE la classe Rettangolo:

```
class Quadrato extends Rettangolo;
```

In questo modo la classeQuadrato EREDITA i metodi e i campi definiti nella classe rettangolo. La classe rettangolo è detta SUPERCLASSE di Quadrato.

Il riferiemento ad un oggetto di una sottoclasse può essere sempre assegnato ad una variabile il cui tipo sia una superclasse della classe alla quale appartiene l'oggetto.

Il riferimento ad un oggetto della classe Quadrato (sottoclasse) può essere assegnato ad una variabile dichiarata di tipo Rettangolo (superclasse): poiché un quadrato è un rettangolo (così come un valore int è anche un valore long), una variabile di tipo Rettangolo può riferirsi ad un oggetto di tipo Quadrato (così come una variabile di tipo long può contenere anche valori di tipo int).

Duenque Rettangolo è un SUPERTIPO di Quadrato: i riferimenti a oggetti della classe Quadrato possono essere PROMOSSI al tipo Rettangolo.

```
Rettangolo r;
Quadrato q = new Quadrato(6);
r = q;      #promozione a Rettangolo
```

LA GERARCHIA DELLE CLASSI

Tutte le classi definibili nel linguaggio Java si trovano all'interno di una gerarchia, in testa alla quale vi è la classe Object. Ogni classe Java quindi estende la classe Object; in particolare, quando nell'intestazione della classe non vi è la parola chiave "extends" è sottointeso che la classe estende direttamente la classe Object.

Nel linguaggio Java una classe può ereditare da una sola classe. Questo significa che ogni classe può avere al massimo una superclasse diretta. Al contrario, una classe può essere estesa da molte classi.

La classe object contiene dei metodi (che quindi vengono ereditati da tutte le classi):

- **.equals** che riceve un parametro Object e restituisce un boolean (true solo se l'oggetto che esegue il metodo è lo stesso oggetto fornito tramite il parametro)
- **.toString**

CAPITOLO 7 – ECCEZIONI

In Java gli eventi anomali vengono chiamati eccezioni. Al fine di evitare che, in caso di anomalia, l'esecuzione si interrompa forndendo messaggi poco comprensibili all'utente, il linguaggio fornisce la possibilità di trattare tutti gli eventi anomali in maniera opportuna mediante il meccanismo di intercettazione delle eccezioni.

Nel caso ad esempio di una divisione per 0, l'esecuzione viene interrotta segnalando che si è verificata l'eccezione descritta dalla classe java.lang.ArithmeticException

INTERCETTARE LE ECCEZIONI: L'ISTRUZIONE "try-catch"

Per tentare l'esecuzione di un blocco di codice e intercettare eventuali eccezioni, Java fornisce l'istruzione try-catch, che ha il seguente schema:

```
try{
     blocco_try;   //codice che potrebbe generare eccezioni
}
catch (Tipo_Eccezione1 id1) {
   blocco_gestore1 ;   //gestisce le eccezioni di Tipo_Eccezione1
}
catch (Tipo_Eccezione2 id2) {
    blocco_gestore2 ;
} …
```

Il codice in blocco_try è il blocco di codicr che si vorrebbe eseguire e che può sollevare eccezioni; ciascuna clausola catch costituisce invece il gestore di un determinato tipo di eccezione (specificato come argomento).

Quando esegue un'istruzione try-catch la Java Virtual Machine si comporta nel modo seguente:

- Inizia a eseguire il codice presente nel blocco try; se durante l'esecuzione di questo codice non viene sollevata alcuna eccezione, l'esecuzione prosegue
- Se durante l'esecuzione del codice del blocco try viene sollevata un'eccezione, la Java Virtual Machine scorre la lista degli argomenti dei blocchi catch bell'ordine in cui sono scritti, alla ricerca di un tipo cui possa essere ricondotto quello dell'eccezione sollevata. Quando lo trova, esegue il codice del corrispondente blocco catch
- Se nessuno dei blocchi catch è preposto ad intercettare l'eccezione trovata, la Java Virtual Machine continua l'esecuzione come se il blocco try-catch non esistesse. In particolare, se ciò avviene nel main, questo implica l'interruzione dell'esecuzione e la segnalazione dell'anomalia all'utente.

Quando è sollevata un'eccezione viene crato un oggetto che descrive quanto è accaduto. Nel blocco catch che gestisce l'eccezione è possibile utilizzare tale oggetto, ad esempio invocandone i metodi.

Esempio: classe Somma: legge dei numeri forniti in input e ne visualizza la somma

```
class Somma{
  public static void main(String[] args ) {
     int somma = 0;
     for (String s: args)
       try{
            Integer i = new Integer(s);
            somma = somma + i;
        } catch (NumberFormatException e) {
         System.out.println("La stringa \"" + s + "\" " + "non rappresenta un numero:  ignoro");
        }
    System.out.println(somma);
  }
}
```

ALCUNE ECCEZIONI

Tutte queste classi estendono, direttamente o indirettamente, la classe Exception definita nel package java.lang.

- **ArithmeticException**: viene sollevata quando si verificano condizioni eccezionali, ad esempio divisioni per 0
- **ArrayIndexOutOfBoundsException**: quando si tenta di accedere ad una posizione insesistente di un array
- **ClassCastException**: quando si forza un tipo riferimento a un sottotipo di cui l'oggetto non è istanza (ad esempio fare il cast di un oggetto Rettangolo con lati diversi tra loro a Quadrato)
- **NegativeArraySizeException**: quando si tenta di creare un array di lunghezza negativa
- **NullPointerException**: quando si tenta di accedere a un oggetto tramite un riferimento null
- **NumberFormatException**: quandi si tenta di convertire una string ain un valore numerico ma questa non ha il formato appropriato
- **StringIndexOutOfBoundsException**: quando si tenta di accedere ad una posizione inesistente di una stringa

Le prossime due sono invece nel package java.util:

- **EmptyStackException**: quando si tenta di accedere ad un elemento di uno stack vuoto
- **NoSuchElementException**: quando si richiama il metodo nextToken di un istanza di StringTokenizer e i token sono esauriti, o quando si chiama il metodo next di un oggetto che implementa l'interfaccia Iterator e non ci sono più elementi nell'iteratore

CLASSIFICAZIONE DELLE ECCEZIONI

Le eccezioni vengono definite estendendo la sottoclase Exception di una classe di nome Throwable, a sua volta sottoclasse di Object (quindi Exception ← Throwable ← Object). Le eccezioni sono suddivise in due gruppi fondamentali:

- **Eccezioni non controllate**: sono le istante di RuntimeException, cioè tutte le eccezioni definite dalla classe RuntimeExceptione dalle sue sottoclassi (dirette o indirette)
- **Eccezioni controllate**: tutte le altre, cioè tutte le istante di Exception che non sono istanze di RuntimeException. In altre parole sono tutte le eccezioni definite direttamente dalla classe Exception e da tuttte le sue sottoclassi (dirette o indirette) che non sono sottoclassi anche di RuntimeException

I due gruppi di eccezioni si differenziano per il modo in cui vengono trattate dal compilatore.

L'aggettivo "controllata" si riferisce al fatto che il compilatore controlla che ogni metodo o costruttore che possa sollevare un'eccezione ne fornisca un trattamento specifico. Se ciò non avviene il compilatore fornisce un messaggio di errore. Il trattamento esplicito dell'eccezione deve avvenire in uno dei seguenti modi:

1 - intercettandola tramite un'istruzione try-catch

2 - delegandola esplicitamente al chiamante mediante un'opportuna dichiarazione introdotta dalla parola chiave "throws" nell'intestazione del metodo (o costruttore) stesso. Ad esempio, per dichiarare che il metodo main delega al proprio chiamante tutte le eccezioni di tipo IOException e FileNotFoundException si scriverà nell'intestazione:

```
public Static void main(String[] args)
                    throws IOException, FileNotFoundException;
```

Nel caso delle eccezioni non controllate invece, il compilatore non effettua controlli: quando non sono intercettate, esse vengono vengono automaticamente delgate al chiamante senza bisogno di alcuna indicazione esplicita del codice. Le eccezioni controllate sono utilizzate per descrivere eventi importanti di cui è bene che il programmatore non si dimentichi. D'altra parte, se tutte le eccezioni fossero controllate, la scrittura del codice diventerebbe estremamente pesante.

Un criterio di massima utilizzato per la collocazione di una nuova eccezione nella gerarchia e, in particolare, per la scelta tra eccezioni controllate e non controllate è il seguente:

- Le anomalie legate a eventi esterni al programma vengono descritte da eccezioni controllate. Ad'esempio si consideri un'applicazione che deve accedere ad una pagina web remota e alla mancanza di connessione di rete
- Le anomalie legate a eventi interni al programma, cioè le anomalie che potrebbero essere evitate scrivendo il programma in maniera diversa, vengono descritte da eccezioni non controllate. Ad'esempio la divisione per 0 può essere evitata inserendo nel codice opportuni controlli

CAPITOLO 9 – IMPLEMENTAZIONE DELLE CLASSI

Inizieremo ora a studiare come implementare una classe, in modo che possa essere poi utilizzata da altre classi.

CLASSI E OGGETTI

L'esempio principale di questo capitolo è la classe Frazione, di cui abbiamo già visto il comportamento. L'implementazione della classe viene scitta all'interno del corpo della classe. Cominciamo a scriverne la struttura:

```
public class Frazione {
…
public Frazione(int x, int y) {…   }
public Frazione(int x) {…   }
public Frazione piu(Frazione f) {…   }
public Frazione meno(Frazione f) {…   }
public Frazione per(Frazione f) {…   }
public Frazione diviso(Frazione f) {…   }
public boolean equals(Frazione f) {…   }
public boolean isMinore(Frazione f) {…   }
public boolean isMaggiore(Frazione f) {…   }
public int getNumeratore() {…   }
public int getDenominatore() {…   }
public String toString() {…   }
}
```

Oltre ai COSTRUTTORI e ai METODI di cui abbiamo evidenziato le intestazioni, verranno scritti i CAMPI, cioè i dati che definiscono lo stato del singolo oggetto della classe. Una frazione è definita da due numeri interi, il numeratore e il denominatore; pertanto nella classe Frazione definiamo due campi di tipo int, denominati "num" e "den". Poiché la rappresentazione dell'oggetto riguarda solochi progetta la classe, ma non chi la utilizza, è opportuno nascondere questi campi al codice esterno alla classe: per farlo dichiariamo i due campi scrivendo la parola riservata "private" all'inizio della dichiarazione.

I campi possono essere scritti in qualsiasi punto all'interno del corpo della classe. In ogni caso, per rendere più leggibile il codice, li scriveremo sempre all'inizio, seguiti dai costruttori e infine dai metodi.

La dichiarazione dei campi è quindi: `private int num, den;`

Scriviamo ora il COSTRUTTORE con due argomenti: deve ricevere due numeri interi e deve costruire una nuova frazione:

```
public Frazione(int x, int y) {
    num = x;
    den = y;
}
```

E il costruttore con un solo argomento:

```
public Frazione(int x) {
    num = x;
    den = 1;
}
```

Possiamo osservare che il secondo costruttore può essere simulato dal primo fornendo il valore 1 al denominatore. Il secondo costruttore potrebbe delegare il proprio compito al primo fornendogli come argomenti x e 1.

Quando in una classe c'è più di un costruttore, è possibile che un costruttore ne chiami un altro utilizzando il riferimento "this", seguito da una lista si argomenti da fornire al costruttore chiamato. In sostanza , un costruttore può chiedere ad un altro costruttore della stessa classe di fabbricare come prima cosa il nuovo oggetto (chiamata con il riferimento "this" nella prima istruzione), e poi adattare l'oggetto alle proprie esigenze (nelle istruzioni successive).

Nel nostro caso, il costruttore con un parametro può chiedere a quello con due parametri di costruire l'oggetto. Una volta che l'oggetto è stato fabbricato, non c'è bisogno di alcuna modifica. Pertanto possiamo scrivere il costruttore con un parametro come:

```
public Frazione(int x) {
    This(x, 1);
}
```

Codifichiamo ora i METODI della classe. Esaminiamo il metodo "per". Ricordiamo che questo metodo è eseguito da un oggetto, cioè da una frazione. Il metodo deve ricevere come argomento un'altra frazione e restituire come risultato il riferimento ad una nuova frazione. Il numeratore della nuova funzione è dato dal prodotto dei numeratori delle due frazioni. Per ottenere i numeratori è sufficiente accedere ai campi "num" delle due frazioni. Tali campi sono accessibili perché il metodo si trova all'interno della classe Frazione (essendo dichiarati "private", essi non sono accessibili a tutto il codice scritto esternamente alla classe). Per accedere ad un campo si utilizza la notazione: *riferimento_a_oggetto.nome_campo*

Ad esempio, se f è il nome di un riferimento all'altra frazione, per accedere al campo num è sufficiente scrivere: f.num

Quando all'interno di un metodo vogliamo riferirci all'oggetto stesso che esegue il metodo, possiamo usare il riferimento "this". In pratica la scrittura "this.num" indica il campo num di "questo oggetto", cioè dell'oggetto che esegue il metodo.

Il numeratore e il denominatore della nuova funzione possono essere memorizzati in due variabili n e d di tipo int dichiarate localmente al metodo per:

```
n = this.num * f.num;
d = this.den * f.den;
```

A questo punto costruiamo la nuova frazione richiamando il costruttore con due argomenti e memorizziamo il risultato, cioè il riferimento al nuovo oggetto, in una variabile locale g di tipo Frazione:

```
g = new Frazione(n, d);
```

infine il metodo deve restituire il risultato, ossia il riferimento alla nuova frazione, appena memorizzato nella variabile g. per restituire il risultato si utilizza l'istruzione "return" seguita dal risultato da restituire; in questo caso: return g;

le variabili n,d,g sono definite localmente al metodo, per questo sono dette "variabili locali". Quando il metodo termina vengono distrutte.

Con questo procedimento creiamo tutti gli altri metodi, fino ad ottenere la classe Frazione completa.

```
public class Frazione {
//CAMPI
Private int num, den;

//COSTRUTTORI
public Frazione(int x, int y) {
   num = x;
   den = y;   }
public Frazione(int x) {
   this(x, 1);   }

//METODI
public Frazione piu(Frazione f) {
   int n = this.num * f.den + this.den * f.num;
   int d = this.den * f.den;
   return new Frazione(n, d);   }

public Frazione meno(Frazione f) {
   int n = this.num * f.den - this.den *f.num;
   int d = this.den * f.den;
   return new Frazione(n, d);   }

public Frazione per(Frazione f) {
   int n = this.num * f.num;
   int d = this.den * f.den;
   return new Frazione(n, d);   }

public Frazione diviso(Frazione f) {
   int n = this.num * f.num;
   int d = this.den *f.num;
   return new Frazione(n, d);   }

public boolean equals(Frazione f) {
   Frazione g = this.meno(f);
   return g.num == 0;   }

public boolean isMinore(Frazione f) {
   Frazione g = this.meno(f);
   return (g.num < 0 && g.den > 0)  ||  (g.num > 0 && g.den < 0);   }

public boolean isMaggiore(Frazione f) {
   Frazione g = this.meno(f);
   return (g.num < 0 && g.den < 0)  ||  (g.num > 0 && g.den > 0);   }
```

```
public int getNumeratore() {
   return num;    }

public int getDenominatore() {
   return den;    }

public String toString() {
   if (den == 1)
      return String.valueOf(num);
   else
      return num + "/" * den;    }
}
```

Notiamo che, per come è definita la classe, non vengono mai effettuate semplificazioni. Per semplificare una frazione è necessario dividere numeratore e denominatore per il loro massimo comune divisore. Possiamo semplificare le frazioni al momento della costruzione.

Per il calcolo del massimo comun divisore introduciamo nella classe Frazione un metodo di nome "mcd". Questo metodo non svolge azioni legate a un singolo oggetto, ma esegue un calcolo utile alla classe stessa durante la costruzione di nuovi oggetti; è un serivzio che la classe offre a se stessa, è pertanto opportuno che sia dichiarato STATICO (perché non legato allo specifico oggetto) e PRIVATO (perché svolge solo un servizio interno alla classe). A tale scopo nell'intestazione del metodo useremo i modificatori "private" e "static": `private static int mcd(int a, int b);`

Il caso di denominatore uguale a 0 rappresenta una situaziona anomala. Il costruttore di Frazione, quando riceve 0 come secondo parametro, può sollevare un'eccezione. Per sollevare un'eccezione basta utilizzare l'istruzione "throw" seguita dall'espressione che costruisce l'opportuna istanza dell'eccezione. Il costruttore "ArithmeticException" riceve come argomento una stringa di caratteri che verrà visualizzata nel messaggio di errore:

```
public Frazione(int x, int y) {
        If (y == 0)
            Throw new ArithmeticException("Frazione non valida: den=0");
        else
            //se negativo, metti segno al numeratore; fai semplificazioni con mcd
```

COME SOLLEVARE LE ECCEZIONI: L'ISTRUZIONE "throw"

Nel momento in cui viene sollevata l'eccezione, viene creato un oggetto che descrive l'anomalia e che contiene alcune informazioni realtive allo stato dell'esecuzione

`throw new ArithmeticException("…");`

l'istruzione throw solleva un'eccezione provocando la terminazione anomala del metodo o del costruttore in esecuzione e rinviando l'eccezione al chiamante (o intercettando immediatamente l'eccezione se inserita in un blocco try-catch)

IMPLEMENTAZIONE DI UN INTERFACCIA

Le interfacce Java permettono di specificare un insieme di comportamenti. Le interfacce specificano cioè il prototipo di alcuni metodi, che sono a tutti gli effetti metodi astratti, e il loro contratto, ma non ne forniscono l'implementazione, rimandandola alle classi che implementano l'interfaccia.

Vediamo come implementare un'interfaccia in una classe mostrando l'implementazione dell'interfaccia generica Comparable da parte della classe Frazione. L'interfaccia Comparable prevede un tipo parametro T e specifica solo un metodo:

public in compareTo(T o) : confronta con l'oggetto in argomento, restituisce zero, intero positivo o intero negativo a seconda che l'oggetto che esegue il metodo sia uguale, maggiore o minore di quello tra parentesi.

Affinchè una classe IMPLEMENTI un'interfaccia è necessario:

- Dichiarare nell'intestazione della classe (usando "implements") che implementa l'interfaccia
- Implementare, cioè definire, all'interno della classe i metodi dichiarati nell'interfaccia

```
public class Frazione implements Comparable<Frazione> {
    . . .
}
```

Si noti che, essendo Comparable generica, viene fornito un tipo argomento, in questo caso Frazione.

Si aggiunge poi il metodo:

```
public int compareTo(Frazione altra) {
  if (this.equals(altra))
    return 0;
  else
    if (this.isMinore(altra))
      return -1;
    else
      return 1;
}
```

È possibile fare implementare a una classe più interfacce. La struttura generale dell'intestazione di una classe è quindi la seguente:

```
public class NOME extends CLASSE_BASE implements INTERFACCIA1,INTERFACCIA2,…{…}
```

STRUTTURA DELLE CLASSI

All'interno di una classe si possono trovare vari elementi:

CAMPI

Detti anche "variabili di istanza", sono associati a ciascun oggetto della classe e ne definiscono lo stato. Al momento della creazione di un oggetto si riserva all'interno di esso uno spazio di memoria riservato ai campi; i dati memorizzati in questo spazio esistono finchè l'oggetto esiste.

Per accedere ad un campo si usa la sintassi: riferimento_a_oggetto.nome_campo

Per riferirsi all'oggetto in esecuzione all'interno del codice della classe si può utilizzare il riferimento "this".

I campi vengono inizializzati automaticamente al momento della creazione dell'oggetto; il valore utilizzato per l'inizializzazione dipende dal tipo del campo: int 0, float/double/ 0.0, boolean false, char '\u000', riferimento null.

CAMPI STATICI

Contengono informazioni comuni a tutta la classe, esistono indipendentemente dagli oggetti della classe. Sono detti anche "variabili di classe" (al contrario delle "variabili di istanza" che appartengono ai singoli oggetti).

I campi statici sono definiti facendo precedere la dichiarazione della variabile dal modificatore "static". Come i campi non statici, vengono inizializzati automaticamente al valore di default dalla JVM, oopure si può assegnare il valore durante la dichiarazione:

```
public class B {
    public static int contaOggetti = 0;
    private int x;
    public B (int i) {
        contaOggetti++;
        x = i;
    } …
}
```

Mentre x è un campo, contaOggetti è un campo statico.

All'interno della classe è possibuili accedere al campo stastico per mezzo del nome (contaOggetti), mentre all'esterno di essa bisogna aggiungere il nome della classe (B.contaOggetti).

COSTRUTTORI

Un cotruttore è una porzione di codice con lo stesso nome della classe, viene utilizzato per creare un nuovo oggetto della classe.

Il costruttore ha una propria lista di parametri (come i metodi) ed è invocato dall'espressione chiave "new", il cui risultato è un riferimento all'oggetto creato.

METODI E METODI STATICI

I metodi statici appartengono alla classe: per invocarli è necessario far precedere il loro nome da quello della classe.

I metodi non statici appartengono ai singoli oggetti: per invocarli è necessario far precedere il loro nome da un riferimento all'oggetto

VARIABILI LOCALI E PARAMETRI

Le variabili dichiarate all'interno di un metodo o di un costruttore prendono il nome di VARIABILI LOCALI. Le variabili dichiarate nell'intestazione del metodo o del costruttore prendono invece il nome di PARAMETRI FORMALI. Ad esempio nel metodo

```
public boolean equals(Frazione f) {
    Frazione g = this.meno(f);
    return g.num == 0;   }
```

della classe Frazione, f è un parametro formale, mentre g è una variabile locale.

Queste variabili vengono distrutte alla conclusione del metodo.

A differenza di quantoo accade per i campi, le variabili locali dei metodi non hanno un'inizializzazione automatica; i parametri sono invece inizializzati al monento della chiamata, assegnando ad essi i valori dei rispettivi argomenti indicati nell'invocazione del metodo.

USO DI this

la parola chiave "this" ha due possibili significati in base al contesto:

- Se compare all'interno di un costruttore seguita da una lista di argomenti tra parentesi tonde, è interpretata dal compilatore come l'invocazione di un altro costruttore della stessa classe.
- Può essere usata come variabile di riferimento all'interno di un metodo o di un costruttore in relazione all'oggetto corrente , cioè all'oggetto che sta eseguendo il metodo.

IL GARBAGE COLLECTOR

Quando viene creato un oggetto, la JVM gli riserva uno spazio di memoria. Essendo la memoria limitata è bene che quando un oggetto non esiste più Java recuperi quello spazio di memoria. Java non ha un istruzione per "distruggere" un oggetto, ma considera un oggetto "eliminabile" quando non esistono più riferimenti ad esso all'interno dell'applicazione in esecuzione.

PACKAGE

Permettono di raggruppare in unità logiche classi e interfacce correlate tra loro.

Per aggiungere una classe ad un package è necessario dichiararne l'appartenenza mediante l'istruzione

```
package nome_package
```

tale istruzione deve essere obbligatoriamente la prima a comparire nella classe (preceduta al massimo solo da commenti).

I MODIFICATORI DI VISIBILITA' "public" e "amichevole"

- Abbiamo già detto che "private", applicato ai membri e ai costruttori di una classe, ne limita la visibilità all'interno della classe stessa.
- "public" definisce la protezione più debole. Le risorse dichiarate public (classi, interfacce, metodi, costruttori, campi) all'interno di un package sono accessibili a chiunque, pertanto sono utilizzabili all'interno e all'esterno del package.
- Il livello di visibilità detto "amichevole" è implicitamente dichiarato quando non si mettono modificatori; esso delimtà l'accessibilità solo all'interno del package in cui si trova la classe

CAPITOLO 10 – ESTENSIONE DELLE CLASSI

EREDITARIETA' E IMPLEMENTAZIONE DI SOTTOCLASSI

Illustriamo l'esempio della classe Quadrato come estensione della classe Rettangolo. Il meccanismo di estensione delle classi consente di definire la classe Quadrato recuperando quanto possibile dell'implementazione fornita nella classe Rettangolo. In particolare implementeremo Quadrato senza conoscere nulla dell'implementazione di Rettangolo, ma basandoci sul suo contratto. In generale è possibile estendere classi senz aconoscerne l'implementazione, ma solo il comportamento.

Per prima cosa si specifica nell'intestazione della classe Quadrato che essa estende Rettangolo:

```
class Quadrato extends Rettangolo {

.....

}
```

La parola chiave "extends" indica che la classe Quadrato è ottenuta da Rettangolo estendendone stato e comportamento, cioè aggiungendo campi e metodi.

Dunque gli oggetti di Quadrato sono oggetti di Rettangolo; ogni Quadrato è sempre anche un Rettangolo, ma non è vero il contrario.

Quadrato è quindi SOTTOCLASSE di Rettangolo, e Rettangolo è SUPERCLASSE si Quadrato.

La classe Quadrato EREDITA i membri della classe Rettangolo (campi e metodi), ciò non implica che questi membri siano ACCESSIBILI nel codice di Quadrato. I costruttori NON vengono ereditati.

Definiamo il costruttore della classe Quadrato. In generale il costruttore della sottoclasse avviene costruendo un oggetto della superclasse per poi adattarlo alle esigenze della sottoclasse. Per invocare il costruttore della superclasse si utilizza la parola chiave "super". Come "this" anch'essa assume significati diversi in base al contesto. Nel contesto del costruttore di una sottoclasse, super fa riferimento al costruttore della superclasse.

```
Public Quadrato(double x) {
      Super(x, x);
}
```

Tutti i metodi della classe Rettangolo sono evocabili in Quadrato. Implementiamo ad esempio un metodo getLato. Grazie all'ereditarietà possiamo usare uno tra getBase e getAltezza (metodi della classe Rettangolo).

```
Public double getLato() {

     Return getBase();

}
```

Implementiamo ora il metodo toString in modo che fornisca la sola indicazione del lato; il metodo richiama a sua volta getLato.

```
Public String toString() {
   return "lato =  " + getLato();
   }
}
```

Ecco il codice di questa prima versione della classe Quadrato:

```
public class Quadrato extends Reattangolo {
   //COSTRUTTORI
   public Quadrato(double x) {
      super(x, x);
   }
}
   //METODI
   public double getLato() {
      return getBase();
   }
   Public String toString() {
      return "lato =  " + getLato();
   }
}
```

I metodi eseguibili da oggetti della classe Quadrato possono essere divisi in tre gruppi:

- Metodi DEFINITI per la prima volta nella classe Quadrato, come getLato
- Metodi RIDEFINITI nella classe Quadrato, come toString
- Metodi EREDITATI dalla superclasse che non vengono ridefiniti, come getArea e getPerimetro

COSTRUTTORI E GERARCHIA DELLE CLASSI

Generalmente un costruttore non opera mai da solo, ma si avvale sempre dell'ausilio di una'altro costruttore che può essere della stessa classe o della superclasse.

Per scrivere e utilizzare correttamente i costruttori bisogna tenere conto delle seguenti regole:

- Se in una classe non si definiscono costruttori, allora viene automaticamente aggiunto un costruttore privo di argomenti (e il cui corpo è vuoto). OGNI CLASSE POSSIEDE ALMENO UN COSTRUTTORE.
- Se un costruttore non invoca esplicitamente un costruttore della sua superclasse (utilizzando super) o un costruttore della stessa classe (utilizzando this), all'inizio del costruttore viene automaticamente invocato il costruttore privo di argomenti della superclasse (fa eccezione il costruttore della classe Object)
- In un costruttore, l'invocazione di un costruttore della superclasse o di un altro costruttore della stessa classe può trovarsi solo come prima istruzione.

Il senso delle regole precedenti può essere compreso meglio avendo presente la gerarchia delle classi:

Object ← Figura ← Rettangolo ← Quadrato

In pratica ogni costruttore delega parte del suo compito a un costruttore della superclasse o talvolta della classe stessa. In ogni caso ogni costruttore imprime sull'oggetto il proprio marchio di fabbrica, cioè il nome della classe a cui appartiene. Questa informazione è fondamentale per java durante l'esecuzione per riconoscere il tipo effettivo dell'oggetto, ad esempio per calcolare il valore dell'operatore instanceof o per selezionare il metodo da eseguire in caso di polimorfismo.

Facciamo una considerazione sulle classi astratte. Ricordiamo che esse non possono essere istanziate , eppure ognuna ha almeno un costruttore. A cosa gli serve? Vengono usati per la costruzione degli oggetti delle sottoclassi concrete.

I L RIFERIMENTO "super"

Come "this", anche "super" ha più utilizzi: il primo inerente al richiamo del costruttore della superclasse nel codice della sottoclasse, il secondo lo vediamo ora.

Consideriamo getLato della classe Quadrato:

```
public double getLato() {
   return getbase(); }
```

in mancanza di un riferimento esplicito è sempre sottointeso il rifeirmento this. Pertanto la semplice invocazione getBase() è equivalente a this.getBase(), il cui significato è il messaggio che l'oggetto in

esecuzione invia a se stesso “esegui il tuo metodo getBase”. Il significato invece di super.getBase() è invece il messaggio inviato sempre dall’oggetto in esecuzione “esegui il metodo getBase della superclasse”.

OVERLOADING DEI METODI

In Java si possono definire più metodi con il medesimo nome, ma con segnature differenti (overloading).

Si consideri ad esempio il seguente metodo per il calcolo del valore assoluto di un numero double:

```
public static double valoreAssoluto(double x) {
   if(x > 0)
      return x;
   else
      return -x;
}
```

Questo metodo, applicato ad un valore double, restituisce un risultato double. Applicando il metodo a un valore in, il risultato sarà sempre double. Volendo disporre di un metodo per clalcolare il valore assoluto di numeri int (con risultato di tipo int), possiamo utilizzare l’overloading definendo nella stessa classe:

```
public static int valoreAssoluto(int x) {
   if(x > 0)
      return x;
   else
      return -x;
}
```

In base al tipo dell’argomento utilizzato nella chiamata,il comilatore riconosce quale metodo dovrà essere eseguito. Il secondo metodo potrebbe anche essere scritto:

```
public static int valoreAssoluto(int x) {
   return (int) valoreAssoluto((double) x);
}
```

L’overloading viene risolto in fase di compilazione: viene scelta la segnatura del metodo da eseguire sulla base del tipo di riferimento usato per invocare il metodo e degli argomenti indicati nella chiamata

OVERRIDING, OVERLOADING E SCELTA DEL METODO DA ESEGUIRE

Con l’OVERRIDING si riscrive in una sottoclasse un metodo della superclasse con la stessa segnatura, mentre con l’OVERLOADING è possibile definire metodi con lo stesso nome ma con segnature differenti.

Il compilatore quindi risolve l’overloading stabilendo quale metodo si debba eseguire sulla base degli argomfenti utilizzati nella chiamata.

Se per uno stesso metodo c'è sia overloading che overriding il compilatore può stabilire solo la segnatura del metodo da eseguire (early binding), mentre la decisione relativa al metodo effettivo, tra quelli con la segnatura selezionata, viene rimandata all'esecuzione (late binding).

IMPLEMENTAZIONEDELLA CLASSE "Figura"

A differenza delle classi che abbiamo visto finore, si tratta di una classe ASTRATTA, dunque non può essere istanziata. Ricordiamo che una classe astratta può contenere METODI ASTRATTI, cioè metodi di cui viene fornita solo l'intestazione, ma non l'implementazione, che dovrà essere data nelle sottoclassi concrete.

Nella classe Figura vogliamo fornire i metodi getArea e getPerimetro e i metodi haAreaMaggiore e haPerimetroMaggiore. Gli ultimi due possono essere codificati richiamando i primi due.

Vediamo l'implementazione di haAreaMaggiore:

```
public boolean haAreaMaggiore(Figura altra) {
   double area1 = this.getArea();
   double area2 = altra.getArea();
   if (area1 > area2)
      return true;
   else
      return false;
}
```

Che può essere riscritto in maniera più compatta:

```
public boolean haAreaMaggiore(Figura altra) {
   return this.getArea() > altra.getArea();
}
```

IMPLEMENTAZIONE DELLA CLASSE "rettangolo"

Vediamo una possibile implementazione di Rettangolo, sottoclasse concreta di Figura.

```
public class Rettangolo extends Figura
```

vediamo ora come definire lo stato di un oggetto della classe. Gli oggetti che vogliamo rappresentare sono rettangoli: mi servirà un double per la base e uno per l'altezza.

```
Private double base, altezza;
```

quindi Rettangolo possiede due campi: base e altezza. Scriviamo ora il costruttore (che richiamerà implicitamente quello senza argomenti della superlcasse Figura):

```
public Rettangolo(double x, double y) {
   base = x;
   altezza = y;
}
```

Implementiamo infine i metodi, ottenendo la classe completa:

```
public class Rettangolo extends Figura {
   private double base, altezza;
   public Rettangolo(double x, double y) {
      base = x;
      altezza = y;
   }
   Public double getArea() {
      return base*altezza; }
   public double getPerimetro() {
      return 2*(base + altezza); }
   public double getAltezza() {
      return altezza; }
   public double getBase() {
      return base; }
   public String toString() {
      return "base = " + base + ", altezza = " + altezza; }
   public boolean equals(Rettangolo altro) {
      return altro != null && thius.base == altro.base && this.altezza == altro.altezza; }
   public boolean equals(Object o) {
      if (o instanceof Rettangolo)
         return equals((Rettangolo) o);
      else
         return false; }
}
```

IL MODIFICATORE DI VISIBILITA' "protected"

La sua applicazione è legata all'ereditarietà.

Un membro di una classe dichiarato protected è visibile all'interno del codice di tutte le classi contenute nel medesimo package (come amichevole). All'esterno del package è invece visibile solo alle classi che la estendono direttamente o indirettamente, e solo a carte condizioni:

- Costruttori: è possibile invocare un costruttore protected di una classe esclusivamente nel codici delle classi che la estendono direttamente tramite l'invocazione SUPER
- Campi e Metodi: i campi e i metodi protected di una classe A sono accessibili solo nel corpo delle classi B che la estendono (direttamente o indirettamente) ed esclusivamente tramite variabili riferimento o espressioni il cui tipo sia quello della sottoclasse B

IL MODIFICATORE “final”

Final può essere applicato a variabili, metodi e classi. Assume significati diversi in base al contesto, ma in generale stabilisce che l’identificatore a cui è applicato non può essere modificato.

Applicato alle variabili

Una variabile final è una variabile alla quale si può assegnare un valore una sola volta.

Se si tenta di cambiarne il valore una seconda volta viene generato un errore di compilazione.

Applicato ai metodi

Stabilisce che il metodo non può essere ridefinito dall estensioni della classe.

La ragione principale per utilizzarlo è impedire alle sottoclassi di cambiare il significato del metodo.

Applicato alle classi

Specifica che esse non possono essere estese.

Se si prova ad estendere una classe final viene generato un errore di compilazione.

I motivi per applicarlo sono legati a questioni di correttezza e efficienza. Esempi di classi final in Java sono String e le classi involucro.

COME DEFINIRE UN’ECCEZIONE

Implementando la classe Frazione, abbianmo utilozzato ArithmeticException per rappresentare il tentativo di costruire una frazione con denominatore 0, da richiamare nel costruttore di Frazione al posto di ArithmeticException.

Vogliamo ora definire un’appostia classe che rappresenti questa particolare anomalia. Ricordiamo che gli oggertti che descrivono le eccezioni sono istanze della sottoclasse Exception di Throwable. Dunque per definire una nuova eccezione dobbiamo estendere Exception o una delle sue sottoclassi.

L’eccezione che vogliamo definire è legata all’aritmetica delle frazioni, quindi può essere pensata come una particolare eccezione di tipo aritmetico. Per questa ragione decidiamo di estendere ArithmeticException

```
public class FrazioneException extends ArithmeticException
```

Dev’essere possibile associare all’oggetto eccezione un messaggio che sia poi visualizzato dal metodo String (la maggior parte delle classi che rappresentano eccezioni si limitano a comportamenti di questo tipo, oltre naturalmente a quelli ereditati dalle superclassi). Dunque la classe dovrà avere un costruttore che riceva tramite il parametro il messaggio associato all’errore. Dove memorizziamo il messaggio? Ricordiamo che la classe ArithmeticException è in grado di fornire lo stesso comportamento, quindi non è necessario aggiungere altri campi: la costruzione di un’istanza di FrazioneException verrà delegata al costruttore di ArithmeticException cui sarà fornito il messaggio:

```
public FrazioneException(String msg) {
   super(msg);   }
```

La classe eredita inoltre i metodi della superclasse, tra cui toString. Pertanto non è necessario aggiungere nulla. La classe completa è quindi:

```
public class FrazioneException extends ArithmeticException {
   public FrazioneException(String msg) {
      super(msg);
   }
}
```

RIDEFINIZIONE DI METODI ED ECCEZIONI

Per ridefinire un metodo di una superclasse occorre implementare nella sottoclasse un metodo con lo stesso prototipo. È necessario tenere conto anche delle eccezioni controllate che esso può delegare con la clausola throws: l'insieme di eccezioni controllate che il metodo della sottoclasse dichiara di delegare non può essere più ampio dell'insieme di eccezioni controllate delegate dal metodo della superlcasse che viene ridefinito. Consideriamo ad'esempio le eccezioni:

```
class Eccezione0 extends Exception {
}
Class SottoEccezione2 extends Eccezione0 {
}
```

E le classi:

```
class A {
   public void f(int i) throws Eccezione0 {
      throw new Eccezion0();
   }
}
class D extends A {
   public void f(int i) {
      …
   }
}
class B exrends A {
   public void f(int i) throws SottoEccezione1, SottoEccezione2 {
      if(i < 0)
         throw new SottoEccezione1();
      else
         throw new SottoEccezione2();
   }
}
```

Il metodo f della classe A viene ridefinito nelle sottoclassi B,D ed E. nei tre casi la ridefinizione del metodo è corretta, in quanto l'insieme di eccezioni del metodo della sottoclasse non è più ampio dell'insieme di eccezioni del metodo ridefinito della superclasse. In particolare.

- Il metodo f di E delega Eccezione0 come il metodo f della superclasse A
- Il metodo f di D non delega eccezioni
- Il metodo f di B delega eccezioni di tipo SottoEccezione1 e SottoEccezion2, sottoclassi di Eccezion0

Consideriamo ora la seguente classe C, estensione di B:

```
class C extends B {
   public void f(int i) throws Eccezione0 {
      throw new Eccezione0();
   }
}
```

Il tipo delle eccezioni delegate dal metodo f di C è più ampio di quello delle eccezioni delegate dal metodo della superclasse B → il codice della classe non viene compilato.

METODI ASTRATTI ED ECCEZIONI

Sono metodi di cui viene fornito il prototipo ma non l'implementazione. Specificando l'intestazione di un metodo astratto è possibile indicare le eccezioni che tale metodo potrà delegare. Implementando un metodo astratto dovremo rispettare quanto dichiarato nel metodo: non potremo far delegare ai metodi effettivamente implementati tipi di eccezioni più ampi rispetto a quanto indicato nel metodo astratto.

Considerate le eccezioni Ecceziuone0, SottoEccezione1, SottoEccezione2 definite nel paragrafo precedente, quello che segue è un esempio di implementazione di un metodo astratto di una classe astratta:

```
abstact class A {
   public void f(int i) throws SottoEccezione1, SottoEccezion2 {
      if(i<0)
         throw new SottoEccezione1();
      else
         throw new SottoEccezione2();
   }
}
```

Il seguente è invece un esempio di implementazione di un'interfaccia con un metodo astratto che delega alcune eccezioni:

```
abstract interface I {
   public void f(int i) throws Eccezione0;
}
class A implements I {
   public void f(int i) throws SottoEccezione1, SottoEccezione2 {
```

```
      if(i<0)
        throw new SottoEccezione1();
      else
        throw new SottoEccezione2();
   }
}
```

CAPITOLO 11 – TIPI ENUMERATIVI, GENERICI E INTERFACCE

DEFINIZIONE DI TIPI ENUMERATIVI

Sono definiti tramite particolari classi. La loro dichirazione si apre con la parola riservata “enum” e all’interno del corpo sono elencati, innanzitutto, gli identificatori delle costanti del tipo enumerativo, separati da una virgola. Presentiamo l’esemio del tipo enumerativo MeseDellAnno:

```
public enum MeseDellAnno {
      GENNAIO, FEBBRAIO, MARZO, APRILE, MAGGIO, GIUGNO, LUGLIO, AGOSTO, SETTEMBRE, OTTOBRE,
      NOVEMBRE, DICEMBRE;
}
```

In fase di esecuzione, la JVM crea un oggetto per ogni identificatore indicato e ne memorizza il riferimento nella costante corrispondente.

Ogni tipo enumerativo è una classe che estende una particolare classe generica Enum definita nel package java.lang. in questo modo i metodi definiti nella classe Enum, e ovviamente quelli definiti in Object, sono disponibili per tutti gli oggetti del tipo enumerativo: i metodi name, ordinal, toString vengono ereditati dalla classe Enum. Inoltre tale classe implementa l’interfaccia Comparable<E> e fornisce la seguente implementazione del metodo compareTo:

- Public int compareTo(E o) : confronta l’oggetto del tipo enumerativo che esegue il metodo con quello fornito in argomento. Restituisce valore negativo, zerto o valore positivo a seconda che l’oggetto che esegue il metodo preceda, sia uguale o segua quello in argomento, sulla base dell’ordine con cui sono state dichiarate le costanti enumerative

Un’altra caratteristica dei tipi enumerativi è quella di fornire un metodo statico values che restituisce un array contenente le costanti del tipo enumerativo.

Come per le classi, è possibile aggiungere ad un itpo enumerativo nuovi metodi o sovrascrivere metodi esistenti (a eccezione di compareTo che è final). Ridefiniamo ad esempio il metodo toString di MeseDellAnno:

```
public String toString() {
   switch (this) {
   case GENNAIO:
      return “Gennaio”;
```

```
    case FEBBRAIO:
        return “Febbraio”;
    …così via…
    case DICEMBRE:
        return “Dicembre”;
    default:
        return “”;
    }
}
```

In questo modo il metodo toString ridefinito è disponibile per ogni oggetto del tipo enumerativo. Al momento dell’invocazione del metodo, il caso selezionato in switch sarà quello corrispondente all’istanza che esegue il metodo (defaul non viene mai utilizzato, serve solo per la compilazione del metodo).

Per definire nuovi comportamenti non dobbiamo fare altro che implementare nuovi metodi nel corpo del tipo enumerativo. Pensiamo di voler implementare il metodo Successivo che restituisce il riferimento all’oggetto che rappresenta il mese successivo a quello che esegue il metodo:

```
public MeseDellAnno successivo() {
    return MeseDellAnno.values() [ (this.ordinal() + 1) % 12];
}
```

DEFINIZIONE DI CLASSI GENERICHE

Al fine di mostrare come possono essere definite le classi generiche, sviluppiamo una classe Coppia con due tipi paramentri E e F.

Le istanze di coppia sono coppie di oggetti, il primo di tipo E e il secondo di tipo F. Ad’esempio un’indicazione di tempo può essere modellata come una coppia formata da un oggetto di tipo Datat e da un oggetto di tipo Orario, xioè un’istanza di Coppia<Data, Orario>.

La classe avrà un costruttore con due argomenti, rispettivamente dei tipi argomenti corrispondenti ai tipi paramentri E e F; tale costruttore fabbrica un oggetto che rappresenta la coppia degli oggetti forniti.

Nella classe Coppia forniamo alcuni semplici metodi:

- public E getSinistro() : restituisce primo elemento della coppia
- public F getDestro() : restituisce secondo elemento della coppia
- public String toString() : restituisce stringa contenente le stringhe associate ai due oggetti
- public static int numeroCoppie() : restituisce numero totale di istanze della classe costruite

la classe viene implementata con due campi privati: “sinistro” di tipo E e “destro” di tipo F, che memorizzano i riferimenti ai due oggetti che costituiscono la coppia. Vi è inoltre un campo statico “nCoppie” di tipo int, inizializzato a zero e incrementato a ogni invocazione del costruttore.

Ecco l'implementazione della classe Coppia:

```
public class Coppia<E, F> {
   private E sinistro;
   private F destro;
   priv h  ate static int nCoppie = 0;
   public Coppia(E e, F f) {
      sinistro = e;
      destro = f;
      nCoppie++;
   }
   public E getSinistro() {
      return sinistro;
   }
   public F getDestro() {
      return destro;
   }
   public String toString() {
      return "(" + sinistro + " ," + destro + ")";
   }
   public static int numeroCoppie() {
      return nCoppie;
   }
}
```

Sottolineiamo che una classe generica non viene duplicata per ogni suo tipo parametrizzato, ma rimane unica. In uno specifico tipo parametrizzato, come Coppia<Data, Orario>, i tipi argomento vengono sostituiti ai tipi parametro, senza però creare una nuova classe. Di conseguenza, le parti statiche della classe, non essendo legate a una specifica invocazione della classe, non possono fare riferimento ai tipi parametro.

Osserviamo ad'esempio che il campo statico nCoppie della classe Coppia è unico.

METODI GENERICI

È possibile anche definire metodi generici, cioè metodi dove uno o più tipi sono parametrizzati.

Supponiamo di voler aggiungere alla classe Coppia un costruttore che riceva un oggetto e costruisca la coppia identica, cioè la coppia in cui il primo e secondo elemento coincidono con l'oggetto ricevuto.

Al posto del costruttore, scriviamo un metodo statico da porre nella classe Coppia che svolga la stessa funzione. Cominciamo con lo scrivere un metodo che riceve un riferimento ad un oggetto genereico e crea la coppia identica a partire da esso:

```
public static Coppia creaIdentica(Object o) {
   return new Coppia(o, o);
}
```

Con questa scrittura, il compilatore fornirà degli avvertimenti relativi a operazioni che potrebbero provocare problemi di esecuzione. Infatti non abbiamo specificato i tipi argomento di Coppia. Potremmo scrivere il metodo indicando come argomenti Object:

```
public static Coppia<Object, Object> creaIdentica(Object o) {
   return new Coppia<Object, Object>(o, o);
}
```

In questo modo il compilatore non indicherà nessun problema, però così facendo non abbiamo raggiunto l'obiettivo prefissato.

Affinchè il metodo sia utilizzabile per ogni tipo riferimento, e non solo per uno specifico, possiamo definirlo come un metodo generico, indicando che T è un parametro:

```
public static <T>  Coppia<T, T> creaIdentica(T t) {
   return new Coppia<T, T>(t, t);
}
```

Il compilatore sostituirà al tipo parametro il tipo argomento corrispondente, stabilito analizzando la chiamata. Si considerino ad'esempio le chiamate:

Coppia.creaIdentica("pippo")
Coppia.creaIdentica(10)
Coppia.creaIdentica(new Data())
La prima chiamata restituisce un riferimento di tipo Coppia<String, String> , la seconda di tipo Coppia<Integer, Integer> e la terza di tipo Coppia<Data, Data>.

I metodi generici sono utilizzati solitamente quando vi è un legame tra il tipo degli argomenti del metodo e il tipo del risultato , o quando sia necessario utilizzare, per implementare il metodo, il nome del tipo paramentro.

Come per i tipi argomento delle classi è possibile introdurre dei vincoli sui tipi parametro dei metodi, utilizzando le parole riservate "extends" e "super". Il seguente metodo, ad'esempio, riceve una sequenza di oggetti di un tipo T e il riferimento t ad un oggetto di tipo T. il metodo restituisce il numero di elementi della sequenza che risultano minori dell'oggetto di riferimento t. In questo caso il tipo T non può essere un tipo qualunque, ma deve possedere un ordine totale.

Ciò è garantito se T è un sottotipo di Comparable<? super T>>

```
public static <T extends Comparable<?  super T>> int contaMinoriDi(Sequenza<T>  s,  T  t) {
   int n = 0;
   for (T o : s)
      if (o.compareTo(t) < 0)
         n++;
   return n;
}
```

DEFINIZIONE DI INTERFACCE

Oltre ad utilizzare quelle già disponibili in Java, se ne possono creare di nuove.

L'intestazione è simile a quella delle classi, mentre nel corpo bisogna ricordarsi è possibile specificare solo i prototipi dei metodi e non le loro implementazioni. Mostriamo l'esempio di Comparable

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

Tutti i metodi di un'interfaccia sono public, quindi implementando un'interfaccia il metodo definito nella classe deve avere lo stesso prototipo di quello indicato nell'interfaccia.

Si può definire un'interfaccia estendendone di già definite tramite la parola xhiave "extends". A differenza di quanto accade per le classi, è possibile che un'interfaccia estenda più interfacce.

Le interfacce non hanno costruttori (non si può costruirne istanze) e non possono avere campi; possono però contenere costanti statiche (campi static e final).

USO DEL SUPERTIPO DEFINITO DALL'INTERFACCIA

Vediamo come scrivere metodi di uso generale ricorrendo alle interfacce. Definiamo in particolare alcuni metodi statici di una classe di utilità per gli array chiamata GestioneArray

ORDINAMENTO DI UN ARRAY DI int

L'obiettivo è quello di scrivere un metodo che permetta di ordinare un array di numeri interi. Per ordinare una sequenza di numeri possiamo usare il metodo del bubblesort:

```
public static void ordina(int[] a) {
   int temp;
   boolean scambiato;
   do{
      scambiato = false;
      for (int i = 1; i < a.length; i++)
```

```
        if (a[i - 1] > a[i]) {
            temp = a[i - 1];
            a[i - 1] = a[i];
            a[i] = temp;
            scambiato = true;
        }
    } while (scambiato)
}
```

ORDINAMENTO DI UN ARRAY DI String E DI UN ARRAY DI Integer

Possiamo generalizzare il metodo precedente per definire metodi di ordinamento per altri tipi su cui sia definito un ordine totale. Possiamo ad'esempio usare il medesimo algoritmo per ordinare un array di stringhe. Nel caso delle stringhe, l'ordinamento è definito dal metodo public int compareTo(String s) della classe String. La classe String implementa il tipo parametrizzato Comparable<String>. Utilizzando questo metodo compareTo possiamo riscrivere l'algoritmo di ordinamento per array di stringhe:

```
public static void ordina(String[] a) {
    String temp;
    boolean scambiato;
    do{
        scambiato = false;
        for (int i = q; i < a.length; i++)
            if (a[i -1].compareTo(a[i]) > 0) {
                temp = a[i -1];
                a[i - 1] = a[i];
                a[i] = temp;
                scambiato = true;
            }
    } while (scambiato);
}
```

ORDINAMENTO DI UN ARRAY DI Comparable

Osserviamo che il codice dei due metodi precedenti è identico: ordinano un array basandosi sul fatto che gli oggetti da ordinare mettono a disposizione il metodo compareTo previsto dall'interfaccia Comparable. se volessimo scrivere un metodo di ordinamento per array di oggetti di un tipo T che implementa Comparable, le uniche modifiche nel codice precedente sarebbero nella dichiarazione della variabile temp, di tipo T, e nella dichiarazione del parametro a del metodo, di tipo T[].

Possiamo quindi definire un metodo generico, con un tipo parametro T, in grado di ordinare qualunque array di oggetti di tipo T. per garantire che il tipo argomento fornito sia "ordinabile", richiediamo che T implementi Comparable<T>, o più in generale che T sia un sottotipo di Comparable<S> dove S è supertipo di T:

```
public static <T extends Comparable<? super T>> void ordina(T[] a) {
   T temp;
   boolean scambiato;
   do {
      scambiato = false;
      for (int i = 1; i < a.length; i++)
         if (a[i - 1].compareTo(a[i] > 0) {
            temp = a[i - 1];
            a[i - 1] = a[i];
            a[i] = temp;
            scambiato = true;
         }
   } while  (scambiato);
}
```

Supponiamo di avere dichiarato una variabile di tipo array di Frazione: `Frazione[] frazioni;`

, di averle assegnato un array di Frazioni e di averlo "riempito" di frazioni assegnando un riferimento a una frazione a ogni posizione dell'array. Dato che la classe Frazione implementa l'interfaccia Comparable<Frazione>, per ordinare l'array possiamo utilizzare la chiamata:

```
GestioneArray.ordina(frazioni)
```

RICERCA DICOTOMICA

Dopo aver illustrato una tecnica per ordinare un array, ora vediamo come individuare un elemento all'interno di un array.

Se l'array non è ordinato, occorre esaminare i suoi elementi uno per uno fino ad individuare l'elemento cercato o fino a raggiungere la fine dell'array.

Se l'array è ordinato, è possibile utilizzare una tecnica più rapida, detta "ricerca dicotomica". Funziona come per trovare una parola in un dizionario: si inizia aprendo il dizionario più o meno a metà e si osservano le parole presenti; salvo il caso fortunato di trovare subito la parola cercata, si possono verificare due situazioni:

- la parola cercata precede in ordina alfabetico quelle della pagina selezionata → ripeto la ricerca nella prima metà del dizionario usando la stessa tecnica
- la parola cercata segue in ordine alfabetico quelle della pagina selezionata → ripeto la ricerca nella seconda metà del dizonario usando la stessa tecnica

Ripetendo la ricerca si vanno ad esaminare porzioni di dizionario sempre più strette fino ad arrivare alla pagina contenente la parola desiderata.

Supponiamo ora di disporre, anziché di un dizionario, di un array di stringhe. Una pagina del dizionario equivale ad un elemento dell'array.per delimitare la parte di array di volta in volta considerata, utilizziamo due indici: sx e dx. inizialmente i due indici indicano la prima e l'utima posizone dell'array

```
int sx = 0, dx = a.length – 1;
```

la condizione del ciclo while è sx < dx. Possiamo confrontare alfabeticamente le stringhe tramite il metodo compareTo:

```
public static <T extends Comparable<? super T>> int cerca(T[] a, T chiave) {
   int sx = 0, dx = a.length – 1, m;
   while (sx < dx( {
      m = (sx + dx) / 2;
      if (a[m].compareTo(chiave) < 0)
         sx = m + 1;
      else if(a[m].compareTo(chiave) > 0)
         dx = m – 1;
      else
         sx = dx = m;
   }
   If (a[sx].compareTo(chiave) == 0)
      Return sx;
   else
      return -1;
}
```

CAPITOLO 13 – STRUTTURE DATI DINAMICHE

Durante l'esecuzione dei programmi è necessario memorizzare e organizzare i dati utilizzati in apposite struttre, dette "strutture dati". La scelta delle strutture dati da usare dipende dalle operazioni che dovranno essere effettuate su di esse; ad'esempio in alcune strutture le operazioni di inserimento di un elemento sono molto efficienti, in altre no.

Introdurremo alcune strutture dati fondamentali, concentrandoci sulle strutture dinamiche, cioè quelle caratterizzate da un evoluzione dinamica durante l'esecuzione in base ai dati che vengono utilizzati.

IMPLEMENTAZIONE DI STRUTTURE A PILA

Come primo esempio implementiamo una parte della classe generica Stack<E> . in particolare forniremo i seguenti metodi fondamentali:

- **public void push(E o)** : aggiunge in cima alla pila che esegue il metodo l'oggetto fornito tramite il parametro (in realtà restituisce un riferimento all'oggetto inserito nello stack)

- **public E pop()** : restituisce un riferimento all'oggetto che si trova in cima alla pila che esegue il metodo eliminandolo dalla pila stessa; se la pila è vuota, il metodo solleva una EmptyStackException
- **public boolean empty()** : restituisce true se e solo se la pila che esegue il metodo è vuota

la classe disporrà poi di un costruttore privo di argomenti che crea un oggetto che rappresenta una pila vuota.

Con la classe Stack definiamo anche la classe per l'eccezione (non controllata) che può essere sollevata dal metodo pop. La classe è estremamente semplice:

```
public class EmptyStackException extends RuntimeException {
}
```

IMPLEMENTAZIONE DI PILE MEDIANTE ARRAY

Un'implementazione naturale della classe Stack può essere basata sugli array. Una pila di oggetti viene rappresentata da un array nel quale ogni posizione è utilizzata per memorizzare un elemento della pila.

In particolare la posizione 0 dell'array viene usata per memorizzare l'oggetto che si trova più in basso nella pila.

Prima di fornire l'implementazione della classe generica Stack presentiamo l'implementazione della classe (non generica) Pila, per rappresentare pila di oggetti (cioè di elementi di tipo Object). La classe viene realizzata con due variabili di istanza: l'array di dati (i cui elementi sono di tipo Object) e l'indice "top" che rappresenta, volta per volta, la prima posizione libera. C'è inoltre un campo statico SIZE utilizzato per definire la lunghezza dell'array. Il costruttore della classe ha il compito di creare una pila vuota, pertanto costruisce l'array e assegna al campo "top" il valore 0 per indicare che questa è la prima posizone libera.

Inseriamo tre metodi:

- **push(Object o)** : assegna il riferimento ricevuto tramite parametro alla posizione indicata da top e incrementa top
- **pop()** : svolge le operazioni simmetriche a push, in caso di pila vuota solleva un'eccezione
- **empty()** : controlla il valore di top

```
public class Pila {
  //CAMPI
  private static final int SIZE = 10;
  private Object[] dati;
  private int top;
  //COSTRUTTORI
  public Pila() {
    dati = new Object[SIZE];
    top = 0;
  }
  //METODI
```

```
    public void push(Object o) {
        if (top == 0)
            throw new EmptyStackException();
        else
            return dati[--top];
    }
    public boolean empty() {
        return top == 0;
    }
}
```

Questa implementazione delle strutture a pila è molto semplice, lo svantaggio è che la dimensione massima della pila va fissata a priori: se si tenta di eseguire push con la pila piena verrà sollevata un eccezione.

IMPLEMENTAZIONE DI PILE MEDIANTE STRUTTURE DINAMICHE

Presentiamo ora una seconda implementazione, basata su una struttura dati dinamica: anziché creare subito lo spazio per la pila (come nel caso precedente), lo creiamo dinamicamente man mano che risulta necessario. In particolare creiamo lo spazio per il nuovo elemento da aggiungere alla pila ogni volta che effettuiamo l'operazione push. Simmetricamente, quando si esegue un'operazione pop, lo spazio occupato dall'elemento prelevato dalla pila può essere rilasciato.

La struttura usata per rappresentare la pila è una "lista concatenata", cioè un insieme di NODI collegati tra loro tramite riferimenti.

Un nodo è un oggetto costituito da due campi:

- "**dato**" : destinato a contenere il riferimento a ciò che si vuole memorizzare nel nodo
- "**pros**" : che permette di accedere all'elemento successivo della lista, cioè all'elemento sottostante nella pila

Si osservi che il campo "pros" è un riferimento a un oggetto dello stesso tipo del nodo. In altre parole, se chiamiamo NodoStack il tipo del nodo, allora pros è di tipo NodoStack.

Per inserire un nuovo nodo ALL'INIZIO della lista bisogna:

- creare un nuovo nodo
- copiare il riferimento "o" nel campo dato del nuovo nodo
- inserire il nuovo nodo all'inizio della lista.

Per eliminare l'oggetto che si trova ALL'INIZIO della lista bisogna:

- memorizzare in una variabile "risultato" di tipo E il riferimento contenuto nel primo elemento della lista (cioè nel campo "dato")
- eliminare dalla lista il primo nodo (cioè l'elemento in cima alla pila)
- restituire il riferimento contenuto in "risultato"

```
public class Stack<E> {
   private NodoStack cima;
   private class NodoStack {
      E dato;
      NodoStack pros;
   }
   public Stack() {
      cima = null;
   }
   public void push(E o) {
      NodoStack t = new NodoStack():
      t.dato = o;
      t.pros = cima;
      cima = t,
   }
   public E pop() {
      if (cima == null)
         throw new EmptyStackException();
      else {
         E risultato = cima.dato;
         cima = cima.pros;
         return risultato,
      }
   }
   Public boolean empty() {
      Return cima == null;
   }
}
```

LE CODE

Nelle pile gli elementi vengono prelevati e inseriti sempre dallo stesso lato. In questo modo il primo elemento a essere prelevato è sempre quello che è stato inserito per ultimo.

Nelle strutture coda invece, gli elementi vengono prelevati da un lato e inseriti dal lato opposto. Il primo elemento che può essere prelevato è quindi quello che è stato inserito per primo. Pertanto gli elementi vengono prelevati nello stesso ordine con il quale sono stai inseriti.

Anche le code si possono realizzare sia con array che con liste. Presenteremo solo la realizzazione di una coda di oggetti tramite liste.

Per inserire un nuovo nodo ALLA FINE della lista occorre:

- creare un nuovo nodo

- percorrere l'intera lista fino a raggiungere l'ultimo nodo
- collegare il nuovo nodo alla lista facendo puntare a esso il riferimento contenuto nell'ultimo nodo

in termini di tempo questa tecnica è molto dispendiosa. Per evitare questo processo si può rappresentare la struttura con due riferimenti, uno al primo, e l'altro all'ultimo elemento.

```
public class Coda<E> {
   private NodoCoda primo, ultimo;
   private class NodoCoda {
      E dato;
      NodoCoda pros;
   }
   public Coda() {
      primo = ultimo = null;
   }
   public void aggiungi(E x) {
      //creazione del nuovo nodo
      NodoCoda t = new Nodocoda();
      t.dato = x;
      t.pros = null;
      //inserimento del nodo
      If (primo == null)              //caso di coda inizialmente vuota
         Primo = ultimo = t;
      else {
         ultimo.pros = t;             //caso di coda non vuota
         ultimo = t;                  //aggiorna il riferimento ultimo
      }
   }
   public E preleva() {
      if (primo == null)
         throw new CodaVuotaException();
      else {
         E risultato = primo.dato;
         primo = primo.pros;
         if (primo == null)           //caso in cui la coda sia rimasta vuota
            ultimo = null;
            return risultato;
      }
   }
   public E primo() {
      if (primo == null)
         throw new CodaVuotaException();
      else
```

```
        return primo.dato;
    }
    public String toString() {
        return this.toString("   ");
    }
    public String toString(String separatore) {
        if (primo == null)
            return " ";
        else {
            String s = primo.dato.toString();
            for (NodoCoda nodo = primo.pros;  nodo != null;  nodo = nodo.pros)
                s = s + separatore + nodo.dato.toString();
            return s;
        }
    }
    public boolean vuota() {
        return primo = = null;
    }
}
```

LE LISTE ORDINATE

Nelle strutture a pila e a coda la posizione degli elementi viene determinata in base al loro ordine di arrivo. In molte applicazioni però, la posizione degli elementi dev'essere determinata da dei criteri (ad esempio ordine alfabetico). Per fare ciò gli array sono poco efficienti, quindi utilizzeremo strutture a lista.

Studieremo quindi le "liste ordinate", nelle quali le operazioni di inserimento vengono effettuate in modo che la struttura rimanga ordinata.

Per poter mantenere ordinata la struttura deve esistere tra gli elementi della struttura una relazione d'ordine. Pertanto il tipo E dei dati contenuti non potrà essere un tipo qualunque. Costuiremo una classe ListaOrdinata nella quale i dati memorizzati saranno istanze di classi di un tipo parametro E che implementi l'interfaccia Comparable. Più precisamente richiederemo che il tipo parametro E estenda Comparable<? super E>.

```
public class ListaOrdinata<E extends Comparable<?  super E>> {
    private NodoLista inizio;

    private class NodoLista {
        E dato;
        NodoLista pros;
    }
    //costruisce una lista vuota
    public ListaOrdinata() {
        inizio = null;
    }
```

```
//restituisce la posizione di un oggetto nella lista (o 0 nel caso l'oggetto non sia presente)
Public int trova(E x) {
   NodoLista p = inizio;
   int posizione = 1;
   while (p != null && p.dato.compareTo(x)   < 0) {
      p = p.pros;
      posizione++;
   }
   if (p == null || p.dato.compareTo(x)  < 0)   //se non c'è
      return 0;
   else
      return posizione;
}

//inserisce un nuovo elemento nella lista
public void inserisci(E x) {
   //ricerca della posizione per l'inserimento
   NodoLista p = inizio, q = null;
   while (p != null && p.dato.compareTo(x) < 0) {
      q = p;
      p = p.pros;
   }
   //creazione del nuovo nodo
   NodoLista r = new NodoLista();
   r.dato = x;
   //inserimento del nodo nella lista tra i nodi riferiti da q e p
   r.pros = p;
   if (q == null)    //inserimento all'inizio
      inizio = r;
   else              //inserimento dopo il nodo riferito a q
      q.pros = r;
}

//cancella un elemento dalla lista
Public void cancella(E x) {
   //ricerca della posizione per l'inserimento
   NodoLista p = inizio, q = null;
   while (p != null && p.dato.compareTo(x) < 0) {
      q = p;
      p = p.pros;
   }
   //eliminazione del nodo
   if (p != null && p.dato.equals(x))
      if (q == null)   //cancellazione all'inizio
         inizio = inizio.pros;
      else             //cancellazione dopo il nodo riferito da q
         q.pros = p.pros;
}

//restituisce una stringa corrispondente al contenuto della lista
public String toString() {
```

```
        return this.toString("   ");
    }

    /*restituisce una stringa corrispondente al contenuto della lista utilizzando l'argomento per separare i
vari elementi */
    public String toString(String separatore) {
        if (inizio == null)
            return " ";
        else {
            String s = inizio.dato.toString();
            for (NodoLista nodo = inizio.pros; nodo != null; nodo = nodo.pros)
                s = s + separatore + nodo.dato.toString();
            return s;
        }
    }

    //restituisce true se la lista è vuota
    public boolean vuota() {
        return inizio == null;
    }
}
```