

Programmazione I e Programmazione II

Prof. Alberto Leporati

Prof.ssa Micucci Daniela

“Programmazione di base e avanzata con Java”

Walter Savitch - Pearson, 2014

SOMMARIO

CONCETTI PRELIMINARI	10
1 INTRODUZIONE AI COMPUTER E A JAVA	11
1.1 CONCETTI DI BASE SUI COMPUTER	11
1.1.1 HARDWARE E MEMORIA.....	11
1.1.2 PROGRAMMI.....	11
1.1.3 LINGUAGGI DI PROGRAMMAZIONE, COMPILATORI E INTERPRETI	12
1.1.4 BYTECODE JAVA.....	12
1.1.5 CLASS LOADER.....	13
1.2 UN ASSAGGIO DI JAVA	13
1.2.1 APPLICAZIONI E APPLET.....	13
1.2.2 IL MIO PRIMO PROGRAMMA JAVA.....	14
1.2.3 SCRIVERE, COMPILARE ED ESEGUIRE PROGRAMMI JAVA	15
1.3 CONCETTI BASE DI PROGRAMMAZIONE	15
1.3.1 PROGRAMMAZIONE A OGGETTI	15
1.3.2 ALGORITMI	16
1.3.3 COLLAUDO E DEBUGGING	16
1.3.4 RIUTILIZZO DEL SOFTWARE.....	17
2 NOZIONI DI BASE.....	18
2.1 VARIABILI ED ESPRESSIONI.....	18
2.1.1 VARIABILI	18
2.1.2 TIPI.....	18
2.1.3 IDENTIFICATORI JAVA.....	19
2.1.4 ISTRUZIONI DI ASSEGNAZIONE	19
2.1.5 SEMPLICI OPERAZIONI DI INPUT E OUTPUT	20
2.1.6 COSTANTI	20
2.1.7 COMPATIBILITÀ DI ASSEGNAZIONE.....	21
2.1.8 OPERATORI ARITMETICI.....	21
2.2 LA CLASSE STRING.....	22
2.2.1 STRINGHE COSTANTI E VARIABILI	22
2.2.2 METODI DI STRING.....	23
2.2.3 CARATTERI DI ESCAPE.....	24
2.3 OPERAZIONI DI I/O: LA TASTIERA E LO SCHERMO	24
2.3.1 OUTPUT SU SCHERMO.....	24
2.3.2 INPUT DA TASTIERA	25
2.3.3 OUTPUT FORMATTATO CON PRINTF	27
2.4 DOCUMENTAZIONE E STILE	27
2.4.1 NOMI SIGNIFICATIVI PER LE VARIABILI	27
2.4.2 COMMENTI	27
2.4.3 IDENTAZIONE.....	28
2.4.4 UTILIZZARE LE COSTANTI CON NOME.....	28

3 FLUSSO DI CONTROLLO: LA SELEZIONE	29
3.1 ISTRUZIONI IF-ELSE	29
3.1.1 ESPRESSIONE BOOLEANA	30
3.1.2 CONFRONTO TRA STRINGHE	31
3.1.3 OPERATORE CONDIZIONALE	33
3.1.4 IL METODO EXIT	33
3.2 TIPO BOOLEAN	34
3.2.1 VARIABILI BOOLEANE	34
3.2.2 REGOLE DI PRECEDENZA	34
3.2.3 INPUT E OUTPUT DI VALORI BOOLEANI	35
3.3 ISTRUZIONI SWITCH	35
3.3.1 ENUMERAZIONI	36
4 FLUSSO DI CONTROLLO: I CICLI	37
4.1 CICLI IN JAVA	37
4.1.1 ISTRUZIONE WHILE	37
4.1.2 ISTRUZIONE DO-WHILE	37
4.1.3 CICLI INFINITI	38
4.1.4 ISTRUZIONE FOR	38
4.1.5 DICHIARARE VARIABILI ALL'INTERNO DI UN'ISTRUZIONE FOR.....	38
4.1.6 USARE UNA VIRGOLA IN UN'ISTRUZIONE FOR.....	39
4.1.7 ISTRUZIONI FOR-EACH.....	39
4.2 PROGRAMMARE CON I CICLI.....	39
4.2.1 IL CORPO DEL CICLO	39
4.2.2 CONTROLLARE IL NUMERO DI ITERAZIONI IN UN CICLO	39
4.2.3 ISTRUZIONI BREAK E CONTINUE NEI CICLI	40
4.2.4 CICLI DIFETTOSI.....	40
4.2.5 TRACCIARE LE VARIABILI	40
4.2.6 CONTROLLO DELLE ASSERZIONI	40
5 I METODI: CONCETTI BASE	41
5.1 DEFINIZIONE E INVOCAZIONE DI METODI	41
5.1.1 DEFINIRE E INVOCARE METODI VOID	41
5.1.2 DEFINIRE METODI CHE RESTITUISCONO UN VALORE	42
5.1.3 VARIABILI LOCALI	42
5.1.4 BLOCCHI.....	43
5.1.5 PARAMETRI DI TIPO PRIMITIVO	43
5.1.6 ANCORA SULL'ISTRUZIONE RETURN	44
5.2 LA CLASSE MATH	44
5.3 COSA ACCADE REALMENTE QUANDO SI INVOCÀ UN METODO?	45
5.4 COME SCRIVERE I METODI	46
5.4.1 DECOMPOSIZIONE.....	46
5.4.2 AFFRONTARE I PROBLEMI DI COMPILAZIONE.....	47
5.4.3 COLLAUDARE I METODI	47

6 ARRAY.....	48
6.1 CONCETTI DI BASE SUGLI ARRAY.....	48
6.1.1 CREAZIONE E ACCESSO A UN ARRAY	48
6.1.2 DETTAGLI SUGLI ARRAY	48
6.1.3 LA PROPRIETÀ LENGTH	49
6.1.4 ULTERIORI DETTAGLI SUGLI INDICI DI UN ARRAY	49
6.1.5 INIZIALIZZARE GLI ARRAY.....	50
6.1.6 UTILIZZARE IL CICLO FOR-EACH CON GLI ARRAY.....	50
6.2 UTILIZZARE GLI ARRAY NEI METODI	50
6.2.1 VARIABILI INDICIZZATE COME ARGOMENTI DI UN METODO	51
6.2.2 ARRAY COME ARGOMENTI DI UN METODO.....	51
6.2.3 ARGOMENTI DEL METODO MAIN.....	52
6.2.4 ASSEGNAIMENTO E UGUAGLIANZA DI ARRAY.....	53
6.2.5 METODI CHE RESTITUISCONO ARRAY	53
6.3 ARRAY MULTIDIMENSIONALI.....	54
6.3.1 FONDAMENTI SUGLI ARRAY MULTIDIMENSIONALI	55
6.3.2 ARRAY IRREGOLARI.....	55
7 RICORSIONE	56
7.1 LE BASI DELLA RICORSIONE	56
7.1.1 COME FUNZIONA LA RICORSIONE	56
7.1.2 LO STACK E LA RICORSIONE	57
7.1.3 CONFRONTO TRA METODI RICORSIVI E ITERATIVI.....	57
7.1.4 METODI RICORSIVI CHE RESTITUISCONO UN VALORE.....	57
8 DEFINIRE CLASSI E CREARE OGGETTI	58
8.1 DEFINIZIONE DI CLASSI	58
8.1.1 FILE DELLE CLASSI E COMPILAZIONE	59
8.1.2 VARIABILI DI ISTANZA.....	59
8.1.3 METODI DI ISTANZA.....	60
8.1.4 LA PAROLA CHIAVE THIS.....	60
8.2 INFORMATION HIDING E INCAPSULAMENTO	61
8.2.1 INFORMATION HIDING	61
8.2.2 COMMENTI CON PRECONDIZIONI E POSTCONDIZIONI.....	61
8.2.3 I MODIFICATORI D'ACCESSO PUBLIC E PRIVATE	61
8.2.4 METODI GET E SET.....	62
8.2.5 LA PAROLA CHIAVE THIS APPLICATA ALLE VARIABILI D'ISTANZA	62
8.2.6 METODI CHE INVOCANO ALTRI METODI.....	63
8.2.7 INCAPSULAMENTO	63
8.2.8 DOCUMENTAZIONE AUTOMATICA CON JAVADOC	63
8.3 OGGETTI E RIFERIMENTI.....	64
8.3.1 VARIABILI DI TIPO CLASSE.....	64
8.3.2 DEFINIRE UN METODO EQUALS PER UNA CLASSE	64
8.3.3 TEST DI UNITÀ	64

9 APPROFONDIMENTI SU CLASSI, OGGETTI E METODI	66
9.1 COSTRUTTORI	66
9.1.1 DEFINIRE I COSTRUTTORI	66
9.1.2 INVOCARE METODI DA COSTRUTTORI	67
9.1.3 INVOCARE UN COSTRUTTORE DA UN ALTRO COSTRUTTORE	68
9.1.4 LA COSTANTE NULL	68
9.2 VARIABILI STATICHE E METODI STATICI.....	69
9.2.1 VARIABILI STATICHE	69
9.2.2 METODI STATICI.....	69
9.2.3 SUDDIVIDERE LE ATTIVITÀ DEL METODO MAIN IN SOTTO-ATTIVITÀ	71
9.2.4 AGGIUNGERE UN METODO MAIN A UNA CLASSE	71
9.2.5 CLASSI WRAPPER.....	72
9.3 OVERLOADING	73
9.3.1 CONCETTI DI BASE DELL'OVERLOADING	73
9.3.2 OVERLOADING E CONVERSIONE AUTOMATICA DI TIPO	73
9.3.3 OVERLOADING E TIPO DI RITORNO.....	73
9.4 INFORMATION HIDING RIVISITATO.....	74
9.4.1 PRIVACY LEAK	74
9.5 RAPPRESENTARE IN UML LE RELAZIONI ASSOCIATIVE FRA CLASSI (APPUNTI IN CLASSE).....	75
9.5.1 CLASSIFICAZIONE E ISTANZIAZIONE	75
9.5.2 ASTRAZIONI	75
9.5.3 FORMALISMI DI RAPPRESENTAZIONE.....	76
9.5.4 ESEMPI DI DIAGRAMMA	79
9.5.5 LE CARATTERISTICHE DI UN OGGETTO.....	80
9.5.6 DEFINIZIONE DI CLASSE IN JAVA	81
9.6 ARRAY NELLE DEFINIZIONI DI CLASSE	83
9.6.1 ARRAY DI RIFERIMENTI	83
9.7 ENUMERAZIONI COME CLASSI.....	83
9.8 PACKAGE	83
9.8.1 PACKAGE E ISTRUZIONE IMPORT	83
9.8.2 NOMI DI PACKAGE E CARTELLE.....	83
9.8.3 CONFLITTI TRA NOMI	84
10 EREDITARIETÀ	85
10.1 CONCETTI DI BASE SULL'EREDITARIETÀ	85
10.1.1 CLASSI DERIVATE	85
10.1.2 METODI RIDEFINITI: OVERRIDING	86
10.1.3 CAMBIARE IL TIPO DI RITORNO DI UN METODO RIDEFINITO	86
10.1.4 CAMBIARE I MODIFICATORI D'ACCESSO DI UN METODO RIDEFINITO.....	86
10.1.5 OVERRIDING VS. OVERLOADING	87
10.1.6 EREDITARIETÀ NEI DIAGRAMMI UML	87
10.2 INCAPSULAMENTO ED EREDITARIETÀ	87
10.2.1 USO DELLE VARIABILI DI ISTANZA PRIVATE DELLA CLASSE BASE	87
10.2.2 I METODI PRIVATI NON SONO ACCESSIBILI	87
10.2.3 MODALITÀ D'ACCESSO PROTECTED.....	87
10.3 PROGRAMMARE CON L'EREDITARIETÀ	88

10.3.1	COSTRUTTORI NELLE CLASSI DERIVATE.....	88
10.3.2	ANCORA SUL METODO THIS.....	88
10.3.3	INVOCARE UN METODO RIDEFINITO	88
10.3.4	COMPATIBILITÀ DI TIPO	89
10.3.5	LA CLASSE OBJECT	89
10.3.6	UN METODO EQUALS MIGLIORATO.....	90
10.4	VISIBILITÀ	90

11 POLIMORFISMO, CLASSI ASTRATTE E INTERFACCE91

11.1	POLIMORFISMO.....	91
11.1.1	BINDING DINAMICO.....	91
11.1.2	BINDING DINAMICO CON TOSTRING	95
11.1.3	IL MODIFICATORE FINAL	95
11.1.4	METODI PER CUI IL BINDING DINAMICO NON VIENE APPLICATO	96
11.1.5	DOWNCASE E UPCASE	96
11.2	CLASSI ASTRATTE	97
11.2.1	CONCETTI DI BASE	97
11.2.2	LA CLASSE ASTRATTA È UN TIPO	98
11.2.3	ULTERIORI DETTAGLI.....	98
11.3	INTERFACCE	98
11.3.1	INTERFACCIE JAVA.....	98
11.3.2	IMPLEMENTARE UN'INTERFACCIA.....	99
11.3.3	UN'INTERFACCIA COME UN TIPO	99
11.3.4	ESTENDERE UN'INTERFACCIA.....	99

12 ARRAYLIST E GENERICI.....100

12.1	STRUTTURE DI DATI BASATE SU ARRAY.....	100
12.1.1	LA CLASSE ARRAYLIST	100
12.1.2	CREARE UN'ISTANZA DI ARRAYLIST	100
12.1.3	UTILIZZARE I METODI DI ARRAYLIST	100
12.1.4	FOREACH	101
12.1.5	TRIMTOSIZE	102
12.1.6	CLONE.....	102
12.1.7	CLASSI PARAMETRICHE E TIPI DI DATO GENERICO	102
12.2	GENERICI	102
12.2.1	FONDAMENTI	102
12.2.2	VINCOLI SU TIPI PARAMETRICI.....	103
12.2.3	METODI GENERICI	104
12.2.4	EREDITARIETÀ CON CLASSI GENERICHE	104

13 ECCEZIONI **105**

13.1	CONCETTI DI BASE SULLA GESTIONE DELLE ECCEZIONI.....	105
13.1.1	ECCEZIONI IN JAVA	105
13.1.2	CLASSI DI ECCEZIONI PREDEFINITE	106

13.2 DEFINIRE NUOVE CLASSI DI ECCEZIONI	107
13.3 APPROFONDIMENTI SULLE CLASSI DI ECCEZIONI.....	108
13.3.1 DICHiarare le eccezioni.....	108
13.3.2 TIPI DI ECCEZIONI	109
13.3.3 ERRORI.....	110
13.3.4 THROW E CATCH MULTIPLI	110
13.3.5 BLOCCO FINALLY	110
13.3.6 RILANCIARE UN'ECCEZIONE	111

14 STREAM E I/O DA FILE..... 112

14.1 INTRODUZIONE AI FLUSSI DI DATI E ALL'I/O SU FILE.....	112
14.1.1 IL CONCETTO DI STREAM	112
14.1.2 PERCHÉ UTILIZZARE L'I/O SU FILE?.....	112
14.1.3 FILE DI TESTO E FILE BINARI	112
14.2 I/O CON FILE DI TESTO..... 113	
14.2.1 CREARE UN FILE DI TESTO	113
14.2.2 AGGIUNGERE DATI A UN FILE DI TESTO	114
14.2.3 LEGGERE DA UN FILE DI TESTO	114
14.2.4 LEGGERE UN FILE DI TESTO CON LA CLASSE SCANNER.....	114
14.2.5 LEGGERE UN FILE DI TESTO CON LA CLASSE BUFFEREDREADER.....	115
14.3 TECNICHE PER LA GESTIONE DEI FILE	116
14.3.1 LA CLASSE FILE	116
14.3.2 PERCORSI.....	117
14.3.3 METODI DELLA CLASSE FILE.....	117
14.4 BASI DELL'I/O CON FILE BINARI	118
14.4.1 CREARE UN FILE BINARIO	118
14.4.2 SCRIVERE VALORI DI TIPO PRIMITIVO IN UN FILE BINARIO	118
14.4.3 SCRIVERE STRINGHE IN UN FILE BINARIO	119
14.4.4 ALCUNI DETTAGLI SUL METODO WRITEUTF	119
14.4.5 LEGGERE DA UN FILE BINARIO	119
14.4.6 LA CLASSE EOFEXCEPTION (END OF FILE)	121
14.5 I/O SU FILE BINARI DI OGGETTI E ARRAY	121
14.5.1 I/O BINARIO CON OGGETTI DI TIPO CLASSE	121
14.5.2 ALCUNI DETTAGLI SULLA SERIALIZZAZIONE	122
14.5.3 ARRAY NEI FILE BINARI	123
14.6 RIEPILOGO	123
14.7 SLIDE LEZIONE	124
14.7.1 SCANNER.....	125
14.7.2 LETTURA DI UN FILE CSV	125
14.7.3 RANDOMACCESSFILE.....	125

15 STRUTTURE DATI DINAMICHE 126

15.1 LISTE CONCATENATE	126
15.1.1 GENERALITÀ SULLE LISTE CONCATENATE.....	126
15.1.2 IMPLEMENTARE LE OPERAZIONI DI UNA LISTA CONCATENATA.....	127

15.1.3	PRIVACY LEAK	127
15.1.4	INNER CLASS.....	128
15.1.5	CLASSI NODO COME INNER CLASS	128
15.1.6	ITERATORI.....	129
15.1.7	ITERATORI INTERNI ED ESTERNI	129
15.1.8	L'INTERFACCIA JAVA ITERATOR	129
15.1.9	GESTIONE DELLE ECCEZIONI CON LE LISTE CONCATENATE.....	130
15.2	VARIANTI DELLE LISTE CONCATENATE	130
15.2.1	LISTE CONCATENATE DOPPIE	130
15.2.2	PILE	130
15.2.3	CODE	130
15.3	TABELLE DI HASH	130
15.3.1	UNA FUNZIONE DI HASH PER LE STRINGHE	131
15.3.2	EFFICIENZA DELLE TABELLE DI HASH	131
15.4	INSIEMI	132
15.4.1	OPERAZIONI DI BASE SUGLI INSIEMI	132
15.5	ALBERI	132
15.5.1	PROPRIETÀ DEGLI ALBERI	133
16	<u>SLIDE: JAVA COLLECTION FRAMEWORK (PARTE DEL CAPITOLO 16 DEL LIBRO)</u>	135
16.1	L'INTERFACCIA COLLECTION	135
16.2	L'INTERFACCIA SET	135
16.3	L'INTERFACCIA LIST.....	136
16.4	L'INTERFACCIA MAP	136
16.5	ORDINAMENTO	137
16.5.1	COERENZA COMPARETO E EQUALS	138
16.6	L'INTERFACCIA SORTEDSET.....	138
16.7	L'INTERFACCIA SORTEDMAP	138
16.8	IMPLEMENTAZIONI DISPONIBILI	139
16.8.1	ESEMPIO ARRAYLIST E LINKEDLIST	139
16.8.2	ESEMPIO TREESET.....	140
16.8.3	ESEMPIO TREEMAP	141
16.9	HASH	142
16.9.1	HASHSET	142
16.9.2	HASHCODE()	142
16.9.3	ESEMPIO HASHSET.....	143
16.9.4	HASHMAP	143
16.9.5	OVERRIDE DI HASHCODE().....	144
16.10	ESEMPIO: PROGETTO ACCOUNT	144
16.11	ITERATORI	146
16.12	FOREACH.....	147
<u>APPENDICE: TEST E DEBUG CON JUNIT.....</u>	148	
TEST DI REGRESSIONE E AUTOMAZIONE DEL TEST	150	
DEBUGGING.....	150	

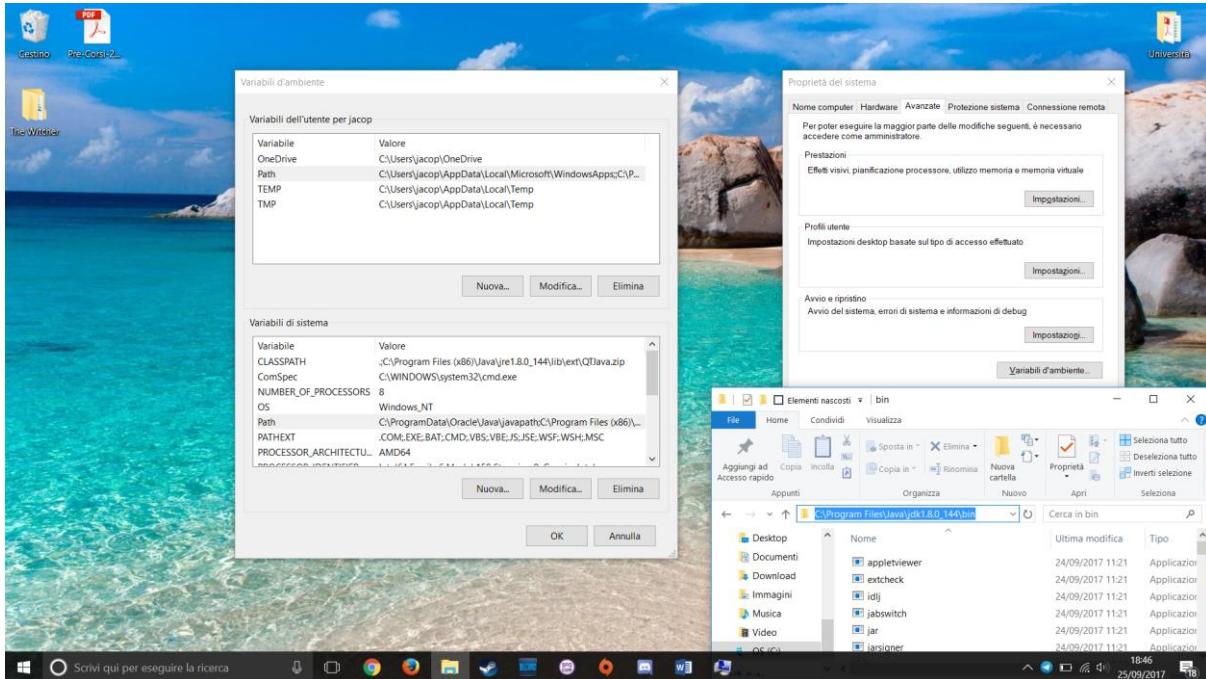
LIBRERIE DI ASSEGNAZIONI IN JUNIT.....	152
ESEMPIO DI INTEGRAZIONE DI JUNIT	155

CONCETTI PRELIMINARI

Programmi usati:

- Java SDK
- Notepad++
- Eclipse Java

Dopo aver installato Java SDK (d'ora in poi JDK) per poterlo chiamare dalla prompt dei comandi bisogna andare in pannello di controllo/ricerca “ambiente”/modificare le variabili d’ambiente/path e aggiungere la cartella d’installazione di Java/bin (di default C:\Program Files\Java\jdk1.8.0_144\bin)



A questo punto con cmd si possono richiamare sempre javac (java compiler) e java (java interpreter)

Promemoria:

- Il prompt dei comandi non è case sensitive (users == user)
- Il tasto tab scorre tra i file della cartella attuale, se si scrive qualcosa e si preme tab verranno cercate solo le cartelle con l'inizio già scritto

Comandi prompt:

- **cd [cartella]**: “Change Directory”, serve per selezionare la cartella (es. c:\ cd Users)
 - **cd.** : Seleziona la cartella attuale, utile per la programmazione
 - **cd..** : seleziona la cartella genitore, utile per la programmazione
- **cd** : serve per tornare a C:\
- **more [file]**: serve per aprire all'interno della prompt i file di testo e leggerli direttamente dalla console
- **javac [file.java]**: compila il file in java
- **java [file]**: esegue il file in java
- **del [file]**: cancella il file
- **del [*.*estensione* (es. docx)]**: cancella tutti i file con l'estensione precisata
- **[comando] [*.*estensione*]**: esegue il [comando] su tutti i file con l'estensione segnata

1 INTRODUZIONE AI COMPUTER E A JAVA

1.1 CONCETTI DI BASE SUI COMPUTER

Un computer è un insieme di:

- **Hardware:** è la macchina fisica
- **Software:** indica genericamente tutti i possibili tipi di programmi (insieme di istruzioni da eseguire) utilizzabili

1.1.1 Hardware e memoria

- **CPU** (Central Processing Unit) o processore: è il dispositivo interno che segue le istruzioni di un programma. Può eseguire solo funzioni semplici.
- **Memoria:** serve per conservare i dati che il computer deve elaborare. Ci sono due tipi di memoria:
 - **Memoria principale - RAM:** conserva il programma che attualmente è in esecuzione e gran parte dei dati che il programma sta utilizzando. Questa memoria è volatile, ovvero si cancella quando la macchina è spenta, ma è più veloce rispetto ad una memoria persistente. È costituita da un lungo elenco di byte numerati con un proprio indirizzo.
 - **Memoria secondaria:** persiste a computer spento. Il supporto può essere:
 - *Hard disk (HDD)*: meccanico, magnetico, più lento
 - *Solid State Drive (SSD)*: a cellule di memoria, più veloce
 - *Optane*: via di mezzo tra SSD e RAM, memoria persistente, più veloce di un SSD ma meno della RAM, bassa capacità di memorizzazione, usata più come supporto
 - *CD, DVD, Flash drive ecc.*

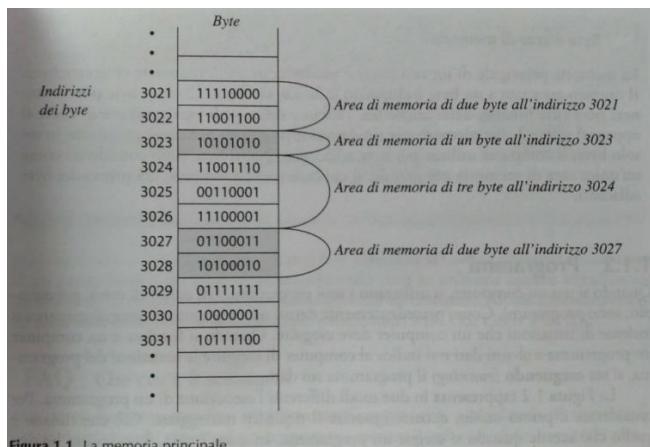


Figura 1.1 La memoria principale.

Un byte (8 bit) è la più piccola unità di memoria indirizzabile. Per convenzione, un byte contiene otto cifre (digit), ciascuna delle quali può contenere 1 o 0. Ciascuna di queste cifre è detta bit (cifra binaria).

Se un dato necessita di più di un byte per essere memorizzato allora verranno usati più byte consecutivi detti “area di memoria” dove l’indirizzo del primo byte viene utilizzato come indirizzo dell’intera area.

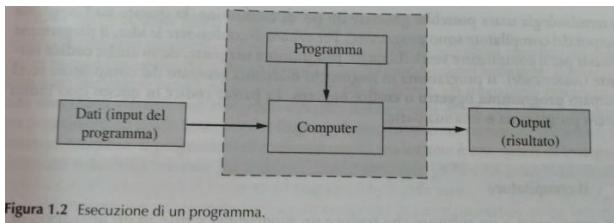
Nella memoria ausiliaria vengono immagazzinati i

file, gruppi più grandi di byte. Ciascun file ha un nome e gruppi di file sono raggruppati in cartelle (directory).

1.1.2 Programmi

Quando si usa un computer si utilizzano i suoi programmi, ovvero semplici istruzioni che il computer deve eseguire. **Quando si danno dei dati su cui lavorare tramite un programma, si dice che il computer sta “eseguendo (running) il programma sui dati”.** Spesso vengono confuse funzioni hardware e software. Ad esempio l'accensione del computer è l'esecuzione del programma "sistema operativo", un programma supervisore che controlla il funzionamento del computer. Se si vuole eseguire un programma si indica al sistema operativo cosa si intende fare e in questo modo il sistema operativo recupera il programma e lo avvia.

Nell'immagine sottostante vengono raffigurati **due metodi di esecuzione**:



- *Senza tratteggio*: il computer ha due input, i dati del programma da eseguire e quello del programma stesso che contiene le istruzioni da eseguire.
- *Con tratteggio*: i dati sono considerati l'unico input del programma. Secondo quest'ottica il computer e il programma sono la stessa cosa.

1.1.3 Linguaggi di programmazione, compilatori e interpreti

La maggior parte dei linguaggi di programmazione moderni è stata progettata per essere facile da utilizzare e comprendere e vengono definiti “**linguaggi di alto livello**”.

L'hardware del computer, però, non è in grado di comprendere immediatamente questi linguaggi e per questo deve prima essere tradotto in “**linguaggio macchina**”, il codice di base dell'hardware.

Il “**linguaggio assembly**” è invece una rappresentazione simbolica del linguaggio macchina più comprensibile anche da parte di una persona. Questi linguaggi vengono detti “**linguaggi di basso livello**”.

Per tradurre da linguaggio di alto livello a linguaggio di basso livello si utilizza un programma chiamato “compilatore” il quale elabora il programma scritto in linguaggio di alto livello in modo che possa essere eseguito dal computer. **Questa operazione è detta compilazione e basta eseguirla una volta per poi rendere il programma risultante sempre eseguibile.** Il programma di input viene detto “programma\codice sorgente” mentre il risultato della compilazione è il “programma\codice oggetto”



Alcuni linguaggi di alto livello non usano compilatori ma “**interpreti**”, ovvero programmi che traducono le istruzioni da un linguaggio di alto livello ad uno di basso ma, a differenza del compilatore, l'interprete esegue ogni porzione di codice subito dopo averla tradotta e non prima traducendo l'intero codice ed eseguendolo dopo. Inoltre la traduzione si esegue ogni volta e non solo la prima volta. Un programma compilato è più veloce di uno interpretato.



Lo svantaggio è che per ogni codice serve un interprete o compilatore apposito in base al sistema operativo (da qui la non uscita garantita di programmi sia per Windows che per Mac e Linux), non ce n'è uno universale.

1.1.4 Bytecode Java

Java utilizza un approccio diverso. Il compilatore non traduce il programma nel linguaggio macchina specifico del computer ma lo traduce in linguaggio **bytecode**, un linguaggio a sé stante e non proprio di un sistema operativo. È un linguaggio macchina di una “**macchina virtuale**”. Tradurre poi questo **bytecode**

nel linguaggio macchina desiderato è abbastanza semplice grazie ad una specie di interprete chiamato “macchina virtuale Java” (JVM). La sequenza di azioni è la seguente:

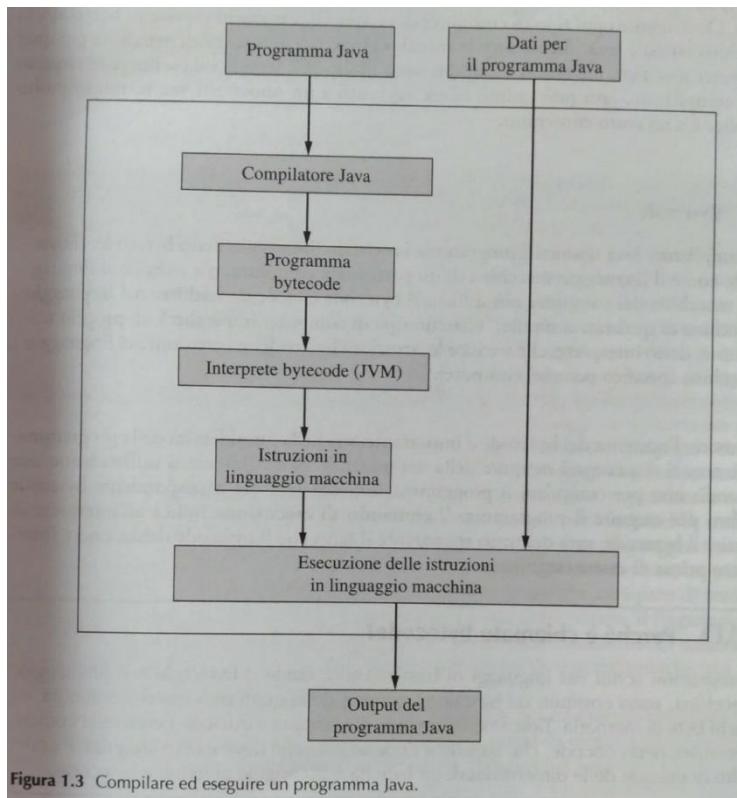


Figura 1.3 Compilare ed eseguire un programma Java.

- Si utilizza il compilatore per tradurre il programma sorgente Java nel bytecode corrispondente
- Si utilizza la JVM per tradurre il bytecode nel linguaggio macchina e per eseguire le istruzioni in linguaggio macchina.

Questo passo aggiuntivo, assente in altri linguaggi di programmazione, **scambia la velocità di esecuzione con una caratteristica molto importante: la portabilità**. Dopo aver tradotto il codice sorgente in bytecode è infatti eseguibile su qualsiasi JVM. Per questo Java viene utilizzato molto per le applicazioni internet.

1.1.5 Class Loader

Solo raramente un programma Java è contenuto in unico file sorgente, tipicamente è diviso in diverse porzioni dette “classi”. I bytecode delle diverse classi devono essere collegati fra loro. L’operazione di collegamento viene svolta da un programma detto “class loader”, un programma chiamato generalmente “linker”.

1.2 UN ASSAGGIO DI JAVA

1.2.1 Applicazioni e Applet

Ci sono due tipi di programmi Java:

- **Applicazioni**: sono concepite per essere **eseguite localmente** su un computer
- **Applet**: sono concepite per essere inviate via internet ed **eseguite** su un computer **in remoto**

1.2.2 Il mio primo programma Java

```

1 import java.util.Scanner; /*Carica la classe Scanner dal package (libreria)
2   java.util*/
3
4 public class PrimoProgramma { /* nome della classe a scelta*/
5
6     public static void main(String[] args){
7
8         System.out.println("Ciao!"); /*Invia l'output allo schermo*/
9         System.out.println("Esegua la somma di due numeri.");
10        System.out.println("Digita entrambi i numeri sulla stessa riga");
11
12        int n1, n2; /*Indica che n1 e n2 sono variabili che contengono numeri interi*/
13
14        Scanner tastiera = new Scanner(System.in); /*predisponde il programma affichè
15          possa leggere l'input dalla tastiera*/
16
17        n1 = tastiera.nextInt(); /*Legge un numero intero(Int) dalla tastiera*/
18        n2 = tastiera.nextInt();
19
20        System.out.println("Ecco la somma dei due numeri:");
21        System.out.println(n1+n2);
22    }
23 }
```

```

Ciao!
Esegua la somma di due numeri.
Digita entrambi i numeri sulla stessa riga
4 5
Ecco la somma dei due numeri:
9
```

Prima annotazione: il programmatore non deve aspettarsi che l'utente comprenda immediatamente come interfacciarsi con il programma. Le istruzioni devono essere comprensibili.

Per ora guarderemo alcuni concetti generali:

- **import java.util.Scanner**: indica al compilatore che questo programma usa la classe Scanner. Questa classe è definita nel package `java.util`. Un package è una libreria di classi che sono definite in precedenza.

Tra le parentesi graffe ci sono una o più porzioni di codice, denominate **metodi**. Ogni classe Java ha un metodo chiamato `main`

- **public static void**: sono necessarie ma vedremo poi perché

Ogni istruzione (statement) all'interno di un metodo definisce un compito, l'insieme delle istruzioni è detto corpo del metodo.

- **System.out.println("")**: stampa a video il contenuto delle parentesi.

Per eseguire le azioni, i programmi Java utilizzano oggetti software detti semplicemente "Oggetti". Le azioni sono definite da metodi. `System.out` è un oggetto utilizzato per inviare un output allo schermo. `println` è il metodo che esegue l'azione per l'oggetto `System.out`. Il contenuto della parentesi è l'argomento e forniscono al metodo l'informazione di cui necessita.

Oggetto.metodo(argomento)

Un oggetto esegue un'azione quando viene invocato (chiamato) uno dei suoi metodi. In un programma Java si ottiene un'invocazione di metodo (o chiamata di metodo) scrivendo il nome dell'oggetto seguito da un punto (dot) seguito dal metodo e infine da una coppia di parentesi tonde.

- **Int n1, n2:** dice che n1 e n2 sono variabili e sono numeri interi.

Una variabile può memorizzare dati. Int è un esempio di tipo di dato (data type), ovvero la specificazione di che insieme di valori possibili e le operazioni definite per questi valori.

- **Scanner tastiera = new Scanner (System.in):** abilita il programma a leggere (accettare) i dati che l'utente inserisce tramite la tastiera
- **n1 = tastiera.nextInt():** legge il numero digitato alla tastiera e lo memorizza alla variabile n1
- **System.out.println(n1+n2):** n1+n2 è l'espressione e non una stringa di caratteri rinchiusa tra apici.

Il ; ha il ruolo di terminazione di un'istruzione.

1.2.3 Scrivere, compilare ed eseguire programmi Java

È possibile scrivere una classe Java utilizzando un semplice editor di testi.

Di norma, ciascuna definizione di classe viene scritta su un file distinto e il nome del file deve coincidere con il nome della classe.

Per compilare una classe Java bisogna usare il prompt dei comandi, andare nella cartella contenente il file java e scrivere javac [nomefile.java]. A questo punto il programma viene tradotto nel suo bytecode generando così un file .class. Sebbene un file Java possa includere un numero indefinito di classi, la sola classe da eseguire è quella che rappresenta l'intero programma. Questa classe conterrà un metodo main che inizia con una formulazione identica o simile a: public static void main (Strings [] args). Questi termini si trovano spesso all'inizio del file. I termini chiave obbligatori sono public static void main.

L'esecuzione del file viene eseguita scrivendo nel prompt "java nomefile" senza .class.

Questo procedimento è reso più semplice dall'utilizzo di un IDE, un ambiente integrato di sviluppo che comprende editor di testo e compilatore per Java (ad esempio BlueJ, Eclipse e NetBeans)

1.3 CONCETTI BASE DI PROGRAMMAZIONE

1.3.1 Programmazione a oggetti

Java è un linguaggio di programmazione a oggetti (Object-Oriented Programming – OOP). La OOP è una metodologia di programmazione che considera **il programma come costituito da oggetti (o istanze) che possono agire da soli e anche interagire fra loro.** Ad esempio si immagini un programma che simula un incrocio stradale con lo scopo di analizzare il flusso del traffico. Questo programma utilizzerà tanti oggetti, ognuno dei quali rappresenta una singola automobile che entra nell'incrocio e, probabilmente, altri oggetti che rappresentano ciascuna corsia, i semafori e così via. Le interazioni tra questi oggetti permettono di giungere ad alcune conclusioni.

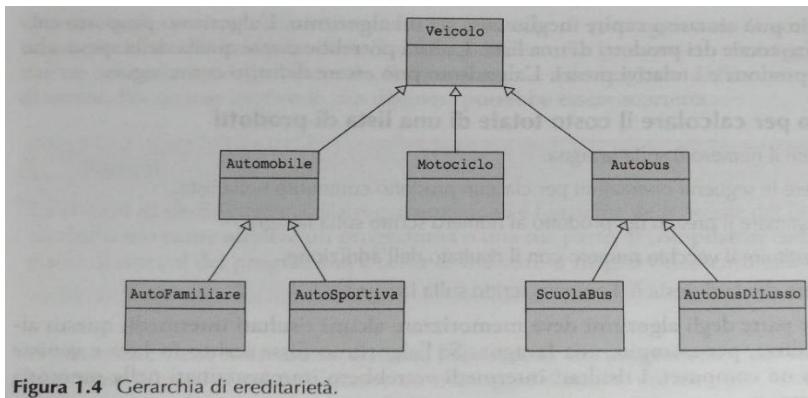
La programmazione a oggetti ha una propria **terminologia**:

- **Attributi:** caratteristiche degli oggetti (es. la velocità di una macchina, la quantità di carburante.)
- **Stato:** valori degli attributi
- **Comportamenti:** le azioni che un oggetto può effettuare. Ogni comportamento è definito in un metodo Java

Gli oggetti di uno stesso tipo condividono lo stesso tipo di dato. Una classe definisce il tipo di dato di un oggetto. Tutti gli oggetti di una classe hanno gli stessi attributi e lo stesso comportamento. Ciò non vuol dire che tutti gli oggetti si comporteranno in maniera identica poiché, a parità iniziale, si troveranno in diversi stati e quindi seguiranno diversi comportamenti.

I tre principi principali della OOP sono:

- **Incapsulamento (capitolo 9):** detto anche information hiding, serve per **rendere visibile solo una parte del codice** in modo che altri programmatore possano utilizzarlo senza sapere tutti i dettagli sul suo funzionamento.
- **Polimorfismo (capitolo 11):** la possibilità per una **stessa istruzione** di un programma di avere **significati diversi in contesti differenti**. Ad esempio premere il tasto ON su un PC, un iPod e uno spazzolino elettrico porta all'accensione di questi ma in maniera differente.



- **Ereditarietà (capitolo 10):** è un modo di organizzare le classi. Grazie ad essa è possibile definire una sola volta gli attributi e i comportamenti comuni e applicarli poi a un intero insieme di classi. Si può definire una classe generica e poi, a posteriori, definire classi specializzate che vanno a ridefinire solo alcuni dati della classe generica come si vede nell'immagine

1.3.2 Algoritmi

Un algoritmo è un insieme di direttive atte a risolvere un problema. Queste istruzioni devono essere espresse in modo completo e preciso in modo che chiunque possa seguirle anche senza ulteriori informazioni. Ad esempio l'algoritmo per calcolare il costo totale di una lista di prodotti è:

- Scrivere il numero 0 sulla lavagna
- Ripetere le seguenti operazioni per ciascun prodotto contenuto nella lista:
 - Sommare il prezzo del prodotto al numero scritto sulla lavagna
 - Sostituire il vecchio numero con il risultato dell'addizione
- Indicare che la risposta è il numero scritto sulla lavagna

Questo algoritmo è scritto in **pseudocodice**, una via di mezzo tra un codice di programmazione e un linguaggio naturale, per il quale le istruzioni devono essere codificate in blocchi come in un codice macchina ma vengono espresse nel linguaggio naturale per spiegare cosa deve accadere.

1.3.3 Collaudo e debugging

Un errore in un programma viene comunemente chiamato bug o difetto (fault). Per questo motivo il processo di rimozione degli errori del programma è detto debugging. Ci sono tre tipi di errori possibili:

- **Errore di sintassi:** è un **errore grammaticale nel programma**, ad esempio l'omissione di un ;
- **Errore a runtime:** è un **errore riscontrato durante l'esecuzione del programma**, ad esempio quando si divide per 0
- **Errore logico:** è quando **un programma** è scritto correttamente ma **non esegue l'azione desiderata**, ad esempio se al posto di + si usa -

Ci sono poi gli insidiosi **errori nascosti**. Quando un programma è scritto correttamente e non presenta errori logici non significa che sia necessariamente corretto. Per questi errori è meglio fare più prove con risultati deducibili e provare ad eseguirli con carta e penna.

1.3.4 *Riutilizzo del software*

La maggior parte dei programmi contiene componenti già esistenti. Questi componenti, essendo stati riutilizzati più volte, riducono costi e tempo di lavoro, sono ben collaudati e quindi più affidabili. Questo è anche il principio dell'incapsulamento però non è l'unico. Le classi vanno sviluppate in modo che siano abbastanza generiche. Un esempio di riutilizzo sono le classi fornite da Java, ovvero le librerie, chiamate anche Java Application Programming Interface o API. Queste classi sono organizzate in package.

2 NOZIONI DI BASE

2.1 VARIABILI ED ESPRESSIONI

2.1.1 Variabili

In un programma le variabili sono utilizzate per memorizzare dati. Qualsiasi dato contenuto nella variabile è detto “valore” della variabile. Questi valori possono essere modificati a runtime.

```

1  public class CestiniUova{
2
3      public static void main (String [] args){
4
5          int numeroDiCestini, uovaPerCestino, totaleUova;
6          numeroDiCestini=10;
7          uovaPerCestino=6;
8
9          totaleUova=numeroDiCestini*uovaPerCestino;
10         System.out.println("Se hai");
11         System.out.println(uovaPerCestino + "uova per cestino e" );
12         System.out.println(numeroDiCestini + " cestini allora in totale hai");
13         System.out.println(totaleUova);
14     }
15 }
```

Se hai
6uova per cestino e
10 cestini allora in totale hai
60

È buona norma scegliere un nome significativo per le variabili. Il nome dovrebbe, infatti, suggerirne lo scopo o indicare il tipo di dato che conterrà.

Prima di poter utilizzare una variabile è necessario fornire alcune informazioni descrittive. Il compilatore ha bisogno di conoscere il nome della variabile, quanto spazio di memoria deve allocare per essa e il modo in cui codificare i contenuti nella variabile. Queste informazioni vengono fornite nella **dichiarazione della variabile**.

Per esempio, nel listato vediamo “int numeroDiCestini, uovaPerCestino, totaleUova;”. Qua vediamo la dichiarazione del tipo “int” e l’elencazione delle variabili che avranno questo tipo.

2.1.2 Tipi

Nome del tipo primitivo	Tipo di valore	Memoria usata	Intervallo di valori
Byte	Intero	1 byte	Da -128 a 127
Short	Intero	2 byte	Da -32.768 a 32.767
Int	Intero	4 byte	Da -2.147.483.648 a 2.147.483.647
Long	Intero	8 byte	Da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
Float	Numero in virgola mobile	4 byte	Da $\pm 3,40282347 \times 10^{38}$ A $\pm 1.40239846 \times 10^{-45}$
Double	Numero in virgola mobile	8 byte	Da $\pm 1,79769313486231570 \times 10^{308}$ A $\pm 4.94065645841246544 \times 10^{-324}$
Char	Carattere singolo (Unicode)	2 byte	Tutti i valori Unicode da 0 a 65.535
Boolean		1 bit	True o False

Un tipo di dato specifica un insieme di valori e operazioni che possono essere eseguite su di esso. Java possiede **due tipi di dato principali**:

- **Tipi classe:** è un tipo di oggetto di una classe. Essa specifica il modo in cui i valori del suo tipo sono memorizzati e definisce le possibili operazioni che possono essere compiute su di essi
- **Tipi primitivi:** sono più semplici degli oggetti (i valori delle classi) poiché includono sia metodi che valori. Un valore di tipo primitivo è un valore non decomponibile, come un singolo numero o una singola lettera

Un numero che presenta una parte decimale è detto “numero in virgola mobile” (floating point number).

Particolarità di “char” è che i caratteri devono essere racchiusi tra singoli apici (es carattere='a').

In Java i tipi hanno l'iniziale maiuscola

2.1.3 Identificatori Java

Il termine tecnico utilizzato nei linguaggi di programmazione per indicare un nome (per esempio di una variabile) è **“identificatore”**.

In Java **un identificatore può essere composto solo da lettere, numeri e _**, per di più **il primo carattere NON può essere una cifra**. Non ci sono limiti di lunghezza. **Java è case sensitive**.

Alcuni termini come i tipi primitivi e la parola if, sono detti “parole chiave” (keywords) o parole riservate. Questi nomi non possono essere utilizzati come nomi di variabili, metodi o per qualsiasi altro scopo che non sia quello predefinito. Tutte le parole chiave sono scritte in minuscolo.

Parole chiave				
abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extend	false	final	finally
float	for	goto	if	implements
import	instanceof	long	native	new
null	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	true	try	void	volatile
while				

2.1.4 Istruzioni di assegnamento

Il modo più veloce per assegnare un valore ad una variabile è quello dell'istruzione di assegnamento (es. `risposta=42`). Il simbolo `=`, quando viene utilizzato in un'istruzione di assegnamento viene detto “operatore di assegnamento”. In generale “variabile = espressione” dove l'espressione può essere un'altra variabile, un numero oppure un'espressione più complicata costituita utilizzando operatori aritmetici.

È importante ricordare che le variabili vanno inizializzate, ovvero bisogna attribuire un valore a queste prima del loro utilizzo per evitare malfunzionamenti del codice.

2.1.5 Semplici operazioni di input e output

```

1 import java.util.Scanner; /*Carica la classe Scanner dal package java.util*/
2
3 public class CestiniUova2{
4     public static void main(String[] args){
5
6         int numeroDiCestini, uovaPerCestino, totaleUova;
7
8         Scanner tastiera = new Scanner(System.in); /*Predisponde il programma
9 affinchè possa leggere da tastiera*/
10        System.out.println("Inserisci il numero di uova per ogni cestino");
11        uovaPerCestino=tastiera.nextInt(); /*Legge un numero da tastiera*/
12        System.out.println("Inserisci il numero di cestini");
13        numeroDiCestini=tastiera.nextInt();
14
15        totaleUova=numeroDiCestini*uovaPerCestino;
16
17        System.out.println("Se hai");
18        System.out.println(uovaPerCestino+" uova per cestino e ");
19        System.out.println(numeroDiCestini + " allora il numero totale di uova è");
20        System.out.println(totaleUova);
21        System.out.println("Rimuoviamo ora due uova da ciascun cestino");
22
23        uovaPerCestino=uovaPerCestino-2;
24        totaleUova=numeroDiCestini*uovaPerCestino;
25
26        System.out.println("Ora hai");
27        System.out.println(uovaPerCestino + " e ");
28        System.out.println(numeroDiCestini + " cestini, quindi in totale hai ");
29        System.out.println(totaleUova + " uova");
30    }
}

```

```

Inserisci il numero di uova per ogni cestino
5
Inserisci il numero di cestini
7
Se hai
5 uova per cestino e
7allora il numero totale di uova Ä
35
Rimuoviamo ora due uova da ciascun cestino
Ora hai
3 e
7cestini, quindi in totale hai
21uova

```

Notare che in questo caso, nella linea
`System.out.println(uovaPerCestino + " e ");` il + non è un'operazione aritmetica ma un simbolo di concatenazione.

2.1.6 Costanti

Vengono chiamate costanti quei valori che non cambiano mai, ad esempio 2 è sempre 2, non dipende da altri dati. Mentre per il tipo char è facile creare una costante, basta scrivere “variabile='A'”, per i numeri ci si deve ricordare che NON bisogna

usare le virgolette ma i punti. Ad esempio “2,37” è sbagliato in Java, mentre “2.37” è giusto. Un altro modo per scrivere i numeri decimali è usare la notazione scientifica. Una notazione simile è detta “e notation”. Poiché le tastiere non permettono di scrivere gli esponenti allora il 10 viene omesso assieme al carattere di moltiplicazione. Ad esempio $8.65 \cdot 10^8$ viene scritto 8.65e8.

Ci si deve ricordare che spesso i numeri decimali non periodici e non finiti vengono approssimati.

Java permette anche di definire delle costanti immutabili. La sintassi è la seguente:

```
public static final tipo variabile = costante;
```

Per esempio, per definire π si può scrivere `public static final float PI = 3.1415;`

Il termine `final` fa sì che il valore non possa più cambiare. Una convenzione per la scrittura delle costanti è l'utilizzo delle maiuscole e che le parole siano separate da _.

```
Es. public static final GIORNI_DELLA_SETTIMANA = 7;
```

2.1.7 Compatibilità di assegnamento

Le conversioni tra i valori, in Java, sono effettuate automaticamente. Bisogna tenere però a mente che **non tutti i tipi sono compatibili**. Ad esempio una variabile double può contenere una variabile int ma non viceversa. **Per sopperire a questo problema si può usare la funzione di “type cast”, ovvero una conversione esplicita di tipo.** Per utilizzare il type cast bisogna premettere al valore assegnato la scritta “(tipo di destinazione)” per segnalare a Java che il valore che viene letto successivamente deve essere convertito nel tipo espresso.

<pre>int x=5; double y=7.0; y = X è un assegnamento lecito X = y è un assegnamento illecito <u>type cast--> X = (int)y è un assegnamento lecito</u></pre>	<p>Per i tipi numerici l'ordine è:</p> <p>byte → short → int → long → float → double</p> <p>Un utilizzo peculiare è quello di convertire un simbolo char in un valore intero per ottenere il suo codice ASCII. Ad esempio l'istruzione:</p> <div style="border: 1px solid black; padding: 5px;"> <pre>char simbolo = '7'; System.out.print((int)simbolo) restituirà il valore 55.</pre> </div>
--	--

2.1.8 Operatori aritmetici

Operatori unari	+[x], -[x], ++, --	Precedenza più alta
Operatori aritmetici binari	*, /, %	
Operatori aritmetici binari	+, -	Precedenza più bassa

Se gli operatori sono dello stesso tipo non ci sono problemi, se un operatore è intero e l'altro in virgola mobile il risultato sarà in virgola mobile.

Bisogna prestare attenzione alla divisione poiché se gli operandi sono in virgola mobile il risultato sarà in virgola mobile ma se gli operandi sono di tipo intero il risultato sarà ugualmente di tipo intero. Ad esempio $5/2=2$. Per ottenere invece il resto dell'operazione si utilizza il simbolo %. Risultato e resto saranno di tipo intero, quindi, ad esempio: $5/2=2$ e $5\%2=1$.

L'operatore resto, nel caso di divisione tra valori in virgola mobile, restituisce il valore $n \% d = n - (d * q)$ dove q è la parte intera di n/d .

```
//System.out.printf("%f\n", (Soldi - (Soldi%500)) / 500); //Soldi%500 = 300, è il resto intero della divisione
if ((Soldi - (Soldi%500))/500!= 0){
    System.out.println("Banconote da 500 euro: " + (Soldi - (Soldi%500))/500);
    Soldi = Soldi%500;
}
```

$(soldi - (soldi \% 500))$ è un multiplo di 500, $(soldi - (soldi \% 500)) / 500$ è il numero di banconote necessarie.

Gli operatori unari si comportano in maniera particolare. $+[x]$ e $-[x]$ sono semplicemente il segno del valore, ad esempio $a=-b$. Gli operatori $++$ e $--$ indicano, invece, l'operazione $+1$ e -1 . È importante notare anche dove vengono posti rispetto al valore. Infatti porre questi operatori PRIMA del valore implica che

l'addizione/sottrazione venga eseguita prima che venga svolta l'istruzione, porli DOPO il valore implica che l'addizione/sottrazione venga eseguita dopo lo svolgimento dell'istruzione.

```

1  public class Prova{
2      public static void main(String[] args){
3
4          int x=3;
5          System.out.println("x = " + x);
6          // Qua viene stampato a schermo il valore originale
7          System.out.println("x = " + ++x);
8          // Qua viene stampato a schermo il valore originale +1
9          // perchè l'operatore è anteposto
10         System.out.println("x = " + x++);
11         // Qua viene stampato a schermo il valore visto prima
12         // perchè l'operatore è postposto
13         System.out.println("x = " + x--);
14         // Qua viene stampato a schermo il valore precedente +1
15         // poichè l'operatore è postposto
16         System.out.println("x = " + --x);
17         // Qua viene stampato a schermo il valore originale
18         // poichè l'operatore è anteposto
19
20     }
21 }
```

```

x = 3
x = 4
x = 4
x = 5
x = 3

```

Ci si deve ricordare che gli spazi non hanno rilevanza nelle opzioni aritmetiche, quindi “5+3” e “5 + 3” sono uguali, la seconda notazione è preferita solo per favorire la leggibilità del codice.

Per l'ordine delle operazioni vengono rispettate come in aritmetica le parentesi e gli operatori.

Ci sono anche degli operatori di assegnamento ausiliari che velocizzano il codice ma non sono indispensabili.

$$X = X + 5 \rightarrow X += 5 \quad X = X - 5 \rightarrow X -= 5 \quad X = X * 5 \rightarrow X *= 5$$

2.2 LA CLASSE STRING

2.2.1 *Stringhe costanti e variabili*

Una stringa come “Inserisci un numero compreso tra 1 e 99” è considerata costante di tipo `string`.

Una stringa di tipo `string` è una stringa racchiusa tra doppi apici.

Una variabile di tipo `string` viene scritta come:

`String [identificatore variabile]`

Successivamente la variabile viene inizializzata come se fosse un tipo classico (perché bisogna ricordarsi che `string` non è un tipo) come `[identificatore variabile] = "[stringa]"`.

Una stringa può contenere anche zero caratteri, in questo caso è utile come divisorio o per aggirare alcuni sistemi di Java, ad esempio per stampare due caratteri concatenati bisogna scrivere:

```

1  public class Prova{
2      public static void main(String[] args){
3
4          char x='a', y='b';
5
6          System.out.print("x + y --> ");
7          System.out.println(x + y);
8          System.out.print("x + \"\\\" + y --> ");
9          System.out.println(x + " " + y);
10
11    }

```

char x='a', y='b';
System.out.println(x + " " + y);
poiché senza l'utilizzo del divisorio i caratteri verrebbero sommati come valore intero corrispondente ai loro codici ASCII.

```
x + y --> 195
x + "" + y --> ab
```

Come si può vedere, per concatenare più stringhe si utilizza il simbolo di concatenazione +.

2.2.2 Metodi di *String*

La maggior parte dei metodi di `String` restituisce un valore. Per esempio il metodo `length` restituisce il numero di caratteri presenti nell'oggetto, quindi ad esempio `"ciao".length()` restituisce il valore 4. Lo si può anche memorizzare come `int n="ciao".length()`. Per alcuni metodi non è necessario specificare alcun argomento, da qui le parentesi vuote. Nel calcolo di una lunghezza della stringa vengono considerati tutti i suoi caratteri, compresi gli spazi, i simboli e i caratteri ripetuti.

Molti dei metodi di `String` dipendono dalla posizione dei caratteri della stringa. Ci si deve ricordare che le stringhe iniziano la numerazione dall'indice 0. L'indicazione di un indice fuori dalla stringa causa un errore a runtime.

Il termine **sottostringa** (`substring`) indica una porzione di una stringa.

<code>Nomestringa.charAt(indice)</code>	Restituisce il carattere che si trova alla posizione indice della stringa corrente
<code>Nomestringa.compareTo(altra stringa)</code>	Confronta le due stringhe per individuare quale viene prima in ordine lessicografico. <ul style="list-style-type: none"> • Valore negativo: stringa corrente per prima • 0: uguali • Valore positivo: altra stringa per prima
<code>Nomestringa.concat(altra stringa)</code>	Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente concatenati a quelli dell'altra stringa. È equivalente al simbolo di concatenamento +
<code>Nomestringa.equals(altra stringa)</code>	Restituisce <code>true</code> se le stringhe sono uguali, altrimenti restituisce <code>false</code>
<code>Nomestringa.equalsIgnoreCase(altra stringa)</code>	Si comporta come <code>equals</code> ma non considera maiuscole e minuscole
<code>Nomestringa.indexOf(altra stringa)</code>	Restituisce l'indice della prima occorrenza della sottostringa "altra stringa" nella stringa corrente. Restituisce -1 se la stringa non è contenuta nella prima
<code>Nomestringa.lastIndexOf(altra stringa)</code>	Restituisce l'indice dell'ultima occorrenza della sottostringa "altra stringa" nella stringa corrente. Restituisce -1 se la stringa non è contenuta nella prima
<code>Nomestringa.length()</code>	Restituisce la lunghezza della stringa
<code>Nomestringa.toLowerCase()</code>	Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente ma in cui tutte le lettere sono minuscole
<code>Nomestringa.toUpperCase()</code>	Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente ma in cui tutte le lettere sono maiuscole
<code>Nomestringa.replace(vecchio carattere, nuovo carattere)</code>	Restituisce una stringa che presenta gli stessi caratteri della stringa corrente ma in cui tutte le occorrenze del vecchio carattere sono sostituite dal nuovo carattere

Nomestringa.substring(inizio)	Restituisce una nuova stringa che presenta gli stessi caratteri della sottostringa che inizia all'indice inizio della stringa corrente
Nomestringa.substring(inizio, fine)	Restituisce una nuova stringa che presenta gli stessi caratteri della sottostringa che inizia all'indice inizio della stringa corrente e termina all'indice fine
Nomestringa.trim()	Restituisce una nuova stringa che presenta gli stessi caratteri della stringa corrente ma in cui sono stati rimossi tutti i caratteri di spaziatura in testa e in coda alla stringa

2.2.3 Caratteri di escape

I caratteri di escape indicano al compilatore di sospendere la normale funzione del carattere successivo.

Ad esempio per stampare a schermo “Il temine “Java” indica il nome di un linguaggio!” non è possibile inserire come argomento di `print` direttamente questa frase in quanto causerebbe un errore di compilazione. Infatti le virgolette di “Java” andrebbero ad interrompere la stringa. Per questo si usa il carattere di escape backslash (\). Quindi, per stampare la frase di esempio, si dovrà scrivere:

```
System.out.print("Il temine \"Java\" indica il nome di un linguaggio!");
```

```

1  public class StringDemo {
2      public static void main(String[] args) {
3
4          String frase = "Elaborazione di testi? Difficile!";
5          int posizione = frase.indexOf("Difficile");
6
7          System.out.println(frase);
8          System.out.println("01234567890123456789012345678901234567");
9          System.out.println("La parola \"Difficile\" inizia all'indice " + posizione);
10
11         frase = frase.substring(0, posizione) + "Facile!";
12         frase = frase.toUpperCase();
13
14         System.out.println("La stringa modificata e:");
15         System.out.println(frase);
16     }
17 }
```

\"	Doppio apice
\'	Singolo apice
\\\	Backslash
\n	Manda a capo la stringa
\r	“carriage return”, sposta l’output della riga all’inizio della riga corrente
\t	Tab

```
Elaborazione di testi? Difficile!
01234567890123456789012345678901234567
La parola "Difficile" inizia all'indice 23
La stringa modificata e:
ELABORAZIONE DI TESTI? FACILE!
```

2.3 OPERAZIONI DI I/O: LA TASTIERA E LO SCHERMO

2.3.1 Output su schermo

Ci sono due modi per stampare a schermo delle stringhe: `System.out.println("stringa")` e `System.out.print ("stringa")`. Il primo indica che alla fine della stringa corrente l’istruzione successiva inizierà su di un’altra stringa mentre la seconda permette di stampare sulla stessa.

```

1 public class Prova {
2     public static void main(String[] args) {
3
4         System.out.print("Uno!");
5         System.out.print("Due!");
6         System.out.println("Tre!");
7         System.out.print("Quattro!");
8     }
9 }
```

Uno!Due!Tre!
Quattro!

2.3.2 Input da tastiera

Il primo passo è includere come prima cosa la libreria Scanner includendo fuori dalla classe:

```
import java.util.Scanner;
```

per leggere l'input inserito dalla tastiera occorre creare un oggetto di questo tipo con l'istruzione:

```
Scanner nome_oggetto_scanner = new Scanner(System.in);
```

dopo aver definito il nuovo oggetto scanner a questo punto si possono usare i metodi della classe Scanner.

Ad esempio “int x=tastiera.nextInt()” legge il successivo input come numero intero e lo assegna alla variabile int. Il metodo next() legge invece una parola.

Le varie parole vengono separate dagli spazi. In questo contesto i caratteri di separazione sono chiamati delimitatori. Per il metodo next una “parola” corrisponde ad una qualsiasi stringa di caratteri che non contiene caratteri di spaziatura. Per leggere un'intera riga bisogna usare nextLine(). La fine di una riga è indicata dal carattere di escape \n.

<code>Nome_oggetto_scanner.next()</code>	Restituisce un valore string che corrisponde al prossimo input da tastiera fino al primo carattere di delimitazione escluso. Il carattere di spaziatura è una delimitazione di default
<code>Nome_oggetto_scanner.nextLine()</code>	Restituisce un valore string che corrisponde al prossimo input da tastiera. Il carattere di riga \n viene ignorato
<code>Nome_oggetto_scanner.nextInt()</code>	Legge il prossimo input come un valore int
<code>Nome_oggetto_scanner.nextDouble()</code>	Legge il prossimo input come un valore double
<code>Nome_oggetto_scanner.nextFloat()</code>	Legge il prossimo input come un valore float
<code>Nome_oggetto_scanner.nextLong()</code>	Legge il prossimo input come un valore long
<code>Nome_oggetto_scanner.nextByte()</code>	Legge il prossimo input come un valore byte
<code>Nome_oggetto_scanner.nextShort()</code>	Legge il prossimo input come un valore short
<code>Nome_oggetto_scanner.nextBoolean()</code>	Legge il prossimo input come un valore boolean. I valori true e false sono scritti esattamente true e false. Non è case sensitive
<code>Nome_oggetto_scanner.useDelimiter(parola di delimitazione)</code>	Fa sì che la stringa “parola di delimitazione” sia l'unico delimitatore utilizzato per separare l'input. Solo le stringhe corrispondenti a questa parola sono considerate delimitatori. In particolare i caratteri di delimitazione di default non saranno più considerati tali a meno che non facciano parte della parola di delimitazione

Attenzione: i metodi next e nextLine della classe Scanner leggono il testo **a partire dall'ultima posizione raggiunta dall'ultimo comando di lettura**. Per esempio, con il codice qua sotto, scrivere come

```
n = tastiera.nextInt();    42 è la risposta
s1 = tastiera.nextLine();  e non lo
s2 = tastiera.nextLine();  dimenticare
```

nella seconda immagine accanto alla prima porta all'assegnazione s1 = “è la risposta” e s2 = “e non lo”. Questo

```

1 import java.util.Scanner;
2
3 public class DelimitatoriDemo {
4
5     public static void main(String[] args) {
6
7         Scanner tastiera1 = new Scanner(System.in);
8         Scanner tastiera2 = new Scanner(System.in);
9         tastiera2.useDelimiter("##");
10        //I delimitatori di tastiera1 sono i caratteri di spaziatura.
11        //L'unico delimitatore di tastiera2 È la stringa #.
12
13        String s1, s2;
14
15        System.out.println("Scrivi una riga di testo con due parole:");
16        s1 = tastiera1.next();
17        s2 = tastiera1.next();
18        System.out.println("Le due parole sono \"" + s1 + "\" e \"" + s2 + "\"");
19
20        System.out.println("Scrivi una riga di testo con due parole");
21        System.out.println("delimitate da #:");
22        s1 = tastiera2.next();
23        s2 = tastiera2.next();
24        System.out.println("Le due parole sono \"" + s1 + "\" e \"" + s2 + "\"");
25    }
26}

```

```

1 import java.util.Scanner;
2
3 public class ScannerDemo {
4
5     public static void main(String[] args) {
6         Scanner tastiera = new Scanner(System.in);
7
8         System.out.println("Digita due numeri interi");
9         System.out.println("separati da uno o più spazi:");
10
11        int n1, n2;
12        n1 = tastiera.nextInt();
13        n2 = tastiera.nextInt();
14        System.out.println("Hai digitato " + n1 + " e " + n2);
15
16        System.out.println("Ora digita altri due numeri.");
17        System.out.println("È ammesso anche il separatore decimale.");
18
19        double d1, d2;
20        d1 = tastiera.nextDouble();
21        d2 = tastiera.nextDouble();
22        System.out.println("Hai digitato " + d1 + " e " + d2);
23
24        System.out.println("Ora digita due parole:");
25
26        String s1, s2;
27        s1 = tastiera.next();
28        s2 = tastiera.next();
29        System.out.println("Hai digitato \"" + s1 + "\" e \"" + s2 + "\"");
30
31        s1 = tastiera.nextLine(); //Necessario per gestire il '\n'
32
33        System.out.println("Digita ora una riga di testo:");
34        s1 = tastiera.nextLine();
35        System.out.println("Hai digitato: \"" + s1 + "\"");
36    }
37}

```

```

Digita due numeri interi
separati da uno o più spazi:
5 6
Hai digitato 5 e 6
Ora digita altri due numeri.
È ammesso anche il separatore decimale.
2.3
Hai digitato 2.0 e 3.0
Ora digita due parole:
forchette coltelli
Hai digitato "forchette" e "coltelli"
Digita ora una riga di testo:
C'è un errore nel codice ma non so quale
Hai digitato: "C'? un errore nel codice ma non so quale"

```

perché il metodo `nextInt` non legge il `\n` a fine linea, quindi `nextLine` legge l'input dopo questo fino al primo `\n`.

i delimitatori personalizzati sono utili nel caso si vogliano includere all'interno degli input dei caratteri di delimitazione utilizzati come default. Nell'esempio

```

Scrivi una riga di testo con due parole:
Pronostico Risultato
Le due parole sono "Pronostico" e "Risultato"
Scrivi una riga di testo con due parole
delimitate da #:
Pronostico Risul##tato

fine##
Le due parole sono "Pronostico Risul" e "tato
fine"

```

vediamo come la seconda tastiera utilizzi `##` come carattere di delimitazione, permettendo così l'utilizzo di caratteri come `\n` e lo spazio.

2.3.3 Output formattato con printf

È analogo alla funzione printf di C. infatti, nel caso non si voglia concatenare una serie di output tramite il simbolo +, è possibile riferirsi ordinatamente ad essi tramite questa funzione. I valori vengono invocati tramite la specifica di formato "%specifico" e, nel caso si voglia segnare più precisamente la lunghezza

```

1  public class Esempio{
2      public static void main(String[] args){
3
4          int a=5, b=-9;
5          double x=3.1415, y=23.4;
6
7          System.out.println("a = " + a + " b = " + b + " x = " + x + " y = " + y);
8          /*Variabili concatenate alle stringhe*/
9          System.out.printf("a = %d b = %d x = %.3f y = %1.2f", a, b, x, y);
10         /*Variabili segnate dopo la fine della stringa*/
11     }
12 }
```

```

a = 5 b = -9 x = 3.1415 y = 23.4
a = 5 b = -9 x = 3,142 y = 23,40

```

prima e dopo il ., si può utilizzare "%[x].[y]specifico" come si vede nell'esempio. Le variabili di riferimento vanno poi elencate

dopo la stringa in maniera ordinata rispetto alle specifiche di riferimento.

%c	Un singolo carattere	Per queste è possibile annotare la lunghezza del campo tramite la forma %[x]specifico
%s	Stringa	
%d	Valore intero	
%f	Valore in virgola mobile	Per queste è possibile annotare la lunghezza del campo tramite la forma %[x].[y]specifico
%e	Valore in virgola mobile con notazione scientifica	

2.4 DOCUMENTAZIONE E STILE

Un programma che produce un output corretto non è necessariamente un buon programma. Molti programmi vengono riutilizzati più volte, modificandoli per correggere eventuali difetti. Se il programma non è di facile lettura o impossibile da modificare senza uno sforzo considerevole allora non è un buon programma.

2.4.1 Nomi significativi per le variabili

I nomi x e y non sono mai una buona scelta per una variabile. Il nome assegnato dovrebbe essere **significativo** e dovrebbe suggerire lo scopo per la quale viene utilizzata.

- I nomi possono contenere lettere o cifre
- Non possono iniziare con un numero
- Per convenzione una variabile inizia con una minuscola e nel caso sia costituita da più parole, solo le successive iniziano con la maiuscola per distinguere (es. nomeVariabileCasuale)

2.4.2 Commenti

I programmi ben realizzati sono autoesplicativi. Questo vuol dire che grazie ad uno stile di programmazione pulito e con nomi degli identificatori rappresentativi, qualsiasi programmatore dovrebbe poter comprendere il codice senza troppe difficoltà. Per aiutare questa comprensione si possono utilizzare commenti all'interno del codice per spiegare alcune particolarità o dare alcune motivazioni. // rende un commento tutto ciò che c'è sulla stessa riga dopo il simbolo, /* */ rende un commento ciò che è inserito tra i simboli.

Un commento speciale è quello che inizia con `/**`. Utilizzando il programma javadoc è possibile estrarre in maniera automatica la documentazione del codice.

Troppi commenti possono essere dannosi quanto troppo pochi. Buona norma è inserire un commento iniziale che spieghi il fine del codice sottostante e usare i commenti per descrivere ciò che non è ovvio.

2.4.3 Identazione

```

1 import java.util.Scanner;
2
3 public class DelimitatoriDemo {
4
5     public static void main(String[] args) {
6
7         Scanner tastieral = new Scanner(System.in);
8         Scanner tastiera2 = new Scanner(System.in);
9         tastiera2.useDelimiter("##");
10        //I delimitatori di tastieral sono i caratteri di spaziatura.
11        //L'unico delimitatore di tastiera2 È la stringa ##.
12
13        String s1, s2;
14
15        System.out.println("Scrivi una riga di testo con due parole:");
16        s1 = tastieral.next();
17        s2 = tastieral.next();
18        System.out.println("Le due parole sono \"" + s1 + "\" e \"" + s2 + "\"");
19
20        System.out.println("Scrivi una riga di testo con due parole");
21        System.out.println("delimitate da #:");
22        s1 = tastiera2.next();
23        s2 = tastiera2.next();
24        System.out.println("Le due parole sono \"" + s1 + "\" e \"" + s2 + "\"");
25    }
26 }
```

Un programma è composto da più parti. I programmi utilizzano una struttura annidata, ovvero ogni parte tra parentesi graffe è un livello del codice. Ad esempio nel codice qua a sinistra si possono notare i tre livelli, dal più esterno (verde), al più interno (rosso).

2.4.4 Utilizzare le costanti con nome

Un ottimo modo per segnalare quali valori siano costanti è l'utilizzo di una scrittura fatta solo di lettere maiuscole divise da `_`. Ad esempio “`public static final double TASSO_INTERESSE_PASSIVO = 6.5`” permette di capire immediatamente cosa sia il valore 6.5 e il fatto che abbia l'attributo `final` fa sì che non possa essere modificato se non all'interno del codice stesso.

3 FLUSSO DI CONTROLLO: LA SELEZIONE

3.1 ISTRUZIONI IF-ELSE

Le istruzioni `if-else` servono per direzionare l'istruzione in base allo stato attuale delle variabili. Per esempio, se un conto bancario fosse in attivo la banca verserebbe un interesse, se fosse in passivo addebiterebbe una mora. Questa istruzione viene codificata come un `if-else`.

```

1 import java.util.Scanner;
2
3 public class SaldoBanca{
4
5     public static final double PENALITA=8.00;
6     public static final double TASSO_INTERESSE=0.02; //2% annuo, tasso semplice
7
8     public static void main(String[] args){
9
10        double saldo;
11
12        System.out.print("Inserisci il saldo del tuo conto: ");
13
14        Scanner tastiera = new Scanner(System.in);
15        saldo=tastiera.nextDouble();
16        System.out.println("Saldo originale: " + saldo);
17
18        if(saldo>=0)
19            saldo = saldo + (TASSO_INTERESSE*saldo)/12;
20        else
21            saldo = saldo - PENALITA;
22
23        System.out.print("Dopo gli adeguamenti al mese corrente, ");
24        System.out.println("legati a interessi e penalita'");
25        System.out.printf("il saldo corrente è: %.2f", saldo);
26
27    }
28 }
```

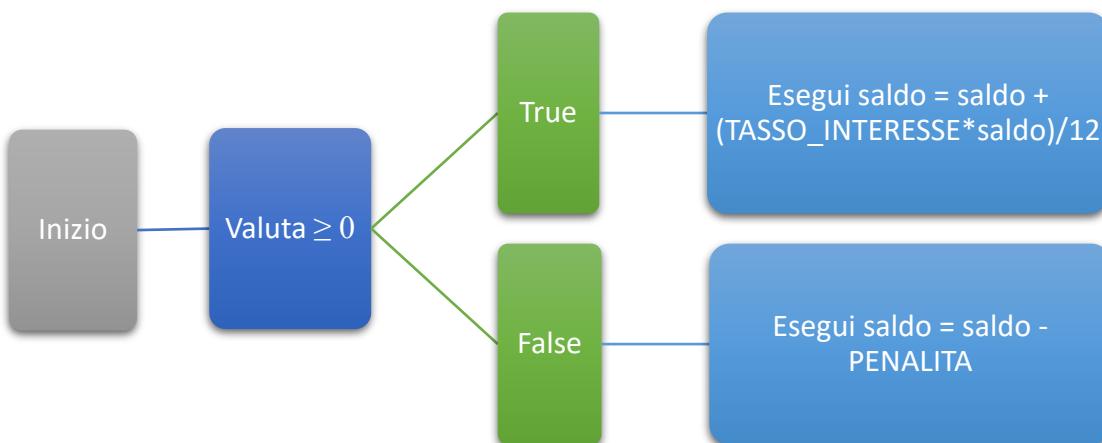
Quando il programma si trova davanti ad una espressione `if-else` per prima cosa controlla il risultato all'interno della parentesi. Se vero allora esegue l'istruzione successiva (nel caso di sequenza di istruzioni viene delimitata tra graffe), altrimenti segue l'istruzione successiva all'`else`. In altri termini l'istruzione `if-else` permette di scegliere tra due rami (branch).

```

Inserisci il saldo del tuo conto: 500,22 ➤ 0
Saldo originale: 500.22
Dopo gli adeguamenti al mese corrente, legati a interessi e penalita'
il saldo corrente →: 501,05

Inserisci il saldo del tuo conto: -27,35 ⬅ 0
Saldo originale: -27.35
Dopo gli adeguamenti al mese corrente, legati a interessi e penalita'
il saldo corrente →: -35,35

```



L'espressione `saldo ≥ 0` è una espressione booleana, semplicemente tratta l'espressione come un vero o falso. Può anche essere omesso l'`else`, semplicemente verrà eseguita l'istruzione di `if` solo se l'espressione booleana è vera, altrimenti prosegue direttamente.

Le istruzioni `if-else` possono essere anche annidate, ovvero presentare più livelli di `if-else` all'interno della stessa istruzione.

Un modo per annidare le istruzioni `if-else` è quello del costrutto `if-else if-else...`

```

1  import java.util.Scanner;
2
3  public class AssegnazioneVoti{
4      public static void main(String[] args){
5
6          int punteggio;
7          char voto;
8
9          System.out.println("Inserisci il tuo punteggio: ");
10         Scanner tastiera= new Scanner(System.in);
11         punteggio = tastiera.nextInt();
12
13         if (punteggio>=90)
14             voto = 'A';
15         else if (punteggio>=80)
16             voto = 'B';
17         else if (punteggio>=70)
18             voto = 'C';
19         else if (punteggio>=60)
20             voto = 'D';
21         else
22             voto = 'E';
23
24         System.out.println("Punteggio = " + punteggio);
25         System.out.println("Voto = " + voto);
26     }
27 }
```

3.1.1 Espressione booleana

Notazione matematica	Nome	Notazione Java
=	Uguale	==
≠	Diverso	!=
>	Maggiore	>
≥	Maggiore o uguale	≥
<	Minore	<
≤	Minore o uguale	≤
U	Unione/e	&&
∩	Intersezione/o	
Non	Negazione	![x]

A	B	A&&B	A B	!A	!B	!A B
V	V	V	V	F	F	F
V	F	F	V	F	V	V
F	V	F	V	V	F	V
F	F	F	F	V	V	V

Leggi di De Morgan
$!(A \cup B) = !A \cap !B$
$!(A \cap B) = !A \cup !B$

Per i confronti di un'espressione booleana possono essere utilizzati tutti i simboli visti nelle tabelle precedenti ma anche l'invocazione di metodi.

3.1.2 Confronto tra stringhe

Quando si valuta l'uguaglianza tra due tipi primitivi si utilizza il simbolo == ma nel caso di confronto tra oggetti bisogna ragionare in maniera differente. Ad esempio chiedersi "stringa 1 == stringa 2" può essere comprensibile per un essere umano ma una macchina non può interpretare al nostro stesso modo l'istruzione. Per questo bisogna utilizzare la formula **s1.equals(s2)** o **s1.equalsIgnoreCase(s2)**, viste tra i metodi degli oggetti stringa. Utilizzando == viene controllato se le due frasi siano nella stessa area di memoria, cosa che vedremo successivamente.

```

1 import java.util.Scanner;
2
3 public class UguaglianzaFrasi{
4     public static void main(String[] args){
5
6         String s1, s2;
7
8         System.out.println("Scrivi due frasi di testo");
9
10        Scanner tastiera = new Scanner(System.in);
11
12        s1=tastiera.nextLine();
13        s2=tastiera.nextLine();
14
15        if (s1.equals(s2))
16            System.out.println("Le due frasi sono uguali, maiuscole e minuscole
17            comprese");
18        else
19            System.out.println("Le due frasi non sono uguali");
20
21        System.out.println("Ora verrà utilizzato il metodo ignoreCase");
22
23        if (s1.equalsIgnoreCase(s2)) {
24            if (s1.equals(s2))
25                System.out.println("Le due frasi sono uguali");
26            else
27                System.out.println("Le due frasi sono uguali se si escludono le
28                maiuscole e le minuscole");
29        }
30    }

```

```
C:\Users\jacop\Desktop>java UguaglianzaFrasi
Scrivi due frasi di testo
Prima frase
prima frase
Le due frasi non sono uguali
Ora verrÀ utilizzato il metodo ignoreCase
Le due frasi sono uguali se si escludono le maiuscole e le minuscole

C:\Users\jacop\Desktop>java UguaglianzaFrasi
Scrivi due frasi di testo
Prima frase
Seconda frase
Le due frasi non sono uguali
Ora verrÀ utilizzato il metodo ignoreCase
Le due frasi non sono uguali

C:\Users\jacop\Desktop>java UguaglianzaFrasi
Scrivi due frasi di testo
Prima frase
Prima frase
Le due frasi sono uguali, maiuscole e minuscole comprese
Ora verrÀ utilizzato il metodo ignoreCase
Le due frasi sono uguali
```

Il metodo **s1.compareTo(s2)** invece verifica l'ordine lessicografico delle stringhe:

- <0: s1 viene prima
- 0: sono uguali
- >0: s2 viene prima

Bisogna tenere conto che **ordine alfabetico** e **ordine lessicografico** sono due cose differenti. Infatti nel secondo le maiuscole vengono prima delle minuscole mentre nel primo vengono considerate indipendentemente da questa caratteristica. In questo caso, nel caso si voglia un confronto alfabetico, è conveniente convertire le frasi in caratteri maiuscoli usando il metodo `toUpperCase`.

```
1 import java.util.Scanner;
2
3 public class OrdineFrasi{
4     public static void main(String[] args){
5
6         String s1, s2;
7
8         System.out.println("Scrivere due frasi");
9
10        Scanner tastiera = new Scanner(System.in);
11
12        s1=tastiera.nextLine();
13        s2=tastiera.nextLine();
14
15        System.out.printf("\nOrdine lessicografico\n");
16        if (s1.compareTo(s2)<0)
17            System.out.printf("La frase \"%s\"" viene prima della frase \"%s\"\n",
18                           s1, s2);
19        else if (s1.compareTo(s2)==0)
20            System.out.println("Le due frasi sono uguali");
21        else System.out.printf("La frase \"%s\"" viene prima della frase \"%s\"\n",
22                           s2, s1);
23
24        String Upper1 = s1.toUpperCase(), Upper2 = s2.toUpperCase();
25
26        System.out.printf("\nOrdine alfabetico\n");
27        if (Upper1.compareTo(Upper2)<0)
28            System.out.printf("La frase \"%s\"" viene prima della frase \"%s\"\n",
29                           s1, s2);
30        else if (Upper1.compareTo(Upper2)==0)
31            System.out.println("Le due frasi sono uguali");
32        else System.out.printf("La frase \"%s\"" viene prima della frase \"%s\"\n",
33                           s2, s1);
34    }
35 }
```

Scrivere due frasi

Frase a

Frase B

Ordine lessicografico

La frase "Frase B" viene prima della frase "Frase a"

Ordine alfabetico

La frase "Frase a" viene prima della frase "Frase B"

3.1.3 Operatore condizionale

Al fine di essere compatibile con i vecchi stili di programmazione, , Java presenta un operatore che costituisce una variante sintattica di if-else.

if (n1>n2) max=n1; else max=n2;	Equivale a	Max = (n1 > n2) ? n1 : n2;
--	------------	----------------------------

3.1.4 Il metodo exit

Alle volte i programmi si trovano in situazioni che rendono insensato il proseguimento dell'esecuzione. In questi casi l'esecuzione del programma può essere terminata tramite l'invocazione del metodo exit:

System.exit(0);

```

1 import java.util.Scanner;
2
3 public class VincitaSuperenalotto{
4     public static void main(String[] args){
5
6         int vincitori;
7         double montepremi=1500000, saldo, vincita;
8
9         Scanner tastiera = new Scanner(System.in);
10
11        System.out.println("Inserire il numero di vincitori al superenalotto");
12        vincitori = tastiera.nextInt();
13
14        if(vincitori==0){ // In questo modo non c'è il rischio di dividere per 0
15            System.out.println("Non ci sono vincitori questa volta");
16            System.exit(0);}
17        else {
18            saldo = (int)montepremi%vincitori;
19            vincita = montepremi/vincitori;
20            montepremi=saldo;
21
22            System.out.printf("\nCi sono %d vincitori, ognuno vince %.2f euro\n",
23                            vincitori, vincita);
24        }
25    }

```

```
Inserire il numero di vincitori al superenalotto
0
Non ci sono vincitori questa volta

C:\Users\jacop\Desktop>java VincitaSuperenalotto
Inserire il numero di vincitori al superenalotto
51

Ci sono 51 vincitori, ognuno vince 29411,76 euro
Il nuovo montepremi parte da 39,00 euro
```

```
1 public class Esempio{
2     public static void main(String[] args){
3
4         int numero = (-5);
5         boolean positivo = numero>0;
6
7         if (positivo) /*Posso indicare solo il nome della variabile booleana e
8             l'if eseguirà il primo ramo se la variabile è vera*/
9             System.out.println("Il valore è positivo");
10        else
11            System.out.println("Il valore è negativo");
12
13        numero = 5;
14        positivo = numero>0; /*La variabile non continua a controllare la
15                           veridicità
16                           dell'affermazione da sola, bisogna richiederla*/
17        if (positivo)
18            System.out.println("Il valore è positivo");
19        else
20            System.out.println("Il valore è negativo");
21    }
}
```

3.2 TIPO BOOLEAN

3.2.1 Variabili booleane

Le variabili booleane possono essere utilizzate come qualsiasi altra variabile, semplicemente i valori accettati sono true e false.

3.2.2 Regole di precedenza

Operatori unari	+[x], -[x], ++, --	Precedenza più alta
Operatori aritmetici binari	*, /, %	
Operatori aritmetici binari	+, -	
Operatori booleani	>, <, ≤, ≥	
Operatori booleani	==, !=	
Operatore booleano	&	
Operatore booleano		
Operatore booleano	&&	
Operatore booleano		Precedenza più bassa

La differenza tra &, | e && || risiede nel metodo di valutazione dell'espressione logica:

- **&&, || → valutazione pigra o a corto circuito:** viene valutata la prima espressione, se questa soddisfa la richiesta viene valutata anche la seconda e così via. Nel caso la prima non risulti valida le successive non vengono valutate
- **&, | → valutazione completa:** vengono valutate tutte le espressioni logiche, indipendentemente dal loro valore, e alla fine vengono eseguite le tabelle di verità

3.2.3 Input e output di valori booleani

```

1 import java.util.Scanner;
2
3 public class Esempio{
4     public static void main(String[] args) {
5
6         boolean booleanVar=false;
7         System.out.println(booleanVar);
8
9         Scanner tastiera = new Scanner(System.in);
10        System.out.print("Inserisci un valore boolean: ");
11        booleanVar = tastiera.nextBoolean();
12
13        System.out.println(booleanVar);
14    }
15 }
```

```

false
Inserisci un valore boolean: true
true
```

Come si può vedere nell'esempio la classe Scanner possiede un metodo chiamato nextBoolean() che legge un singolo valore di tipo boolean. Affichè lo legga, l'utente deve scrivere true o false (indipendentemente dalle maiuscole o minuscole). All'interno del programma, invece, le variabili devono essere scritte espressamente con lettere minuscole.

3.3 ISTRUZIONI SWITCH

Nel caso un'istruzione if-else inizi a presentare troppi rami e che questa dipenda da un valore intero, da un carattere o da una stringa, l'istruzione switch può migliorare la comprensibilità del codice.

Un'istruzione switch è rappresentata da switch (espressione di controllo) {...} dove tra le graffe ci sono i vari casi rappresentati da case (etichetta case): elenco istruzioni). Le istruzioni devono essere terminate con break per evitare che vengano eseguiti anche i casi successivi.

Si può anche precisare un caso default nel caso nessuno dei casi precedenti possa applicarsi.

```

1 import java.util.Scanner;
2
3 public class Esempio{
4     public static void main(String[] args) {
5
6         int numeroNeonati;
7         System.out.print("Inserisci il numero di neonati: ");
8         Scanner tastiera = new Scanner(System.in);
9         numeroNeonati = tastiera.nextInt();
10
11        switch (numeroNeonati){
12            case 1:
13                System.out.println("Congratulazioni");
14                break;
15            case 2:
16                System.out.println("Wow, gemelli!");
17                break;
18            case 3:
19                System.out.println("Wow, tre!");
20                break;
21            case 4:
22                //qua manca l'istruzione e quindi salta al case 5
23            case 5:
24                System.out.printf("Incredibile: %d bambini!\n", numeroNeonati);
25                break;
26            default:
27                System.out.println("Non ci credo!!!!");
28                break;
29        }
30    }
31 }
```

```

C:\Users\jacop\Desktop>java Esempio
Inserisci il numero di neonati: 1
Congratulazioni

C:\Users\jacop\Desktop>java Esempio
Inserisci il numero di neonati: 2
Wow, gemelli!

C:\Users\jacop\Desktop>java Esempio
Inserisci il numero di neonati: 3
Wow, tre!

C:\Users\jacop\Desktop>java Esempio
Inserisci il numero di neonati: 4
Incredibile: 4 bambini! }

C:\Users\jacop\Desktop>java Esempio
Inserisci il numero di neonati: 5
Incredibile: 5 bambini! }

C:\Users\jacop\Desktop>java Esempio
Inserisci il numero di neonati: 6
Non ci credo!!!
}
```

3.3.1 *Enumerazioni*

Immaginiamo un recensore che valuta la qualità dei film catalogandoli come eccellenti, buoni o pessimi. Se si scrivesse un codice per organizzare queste recensioni si potrebbero rappresentare questi voti come 1, 2, 3 o E, B, P. Tuttavia, qualora si definisse una variabile di tipo `int` o `char` per contenere questa valutazione, potrebbe accadere che questa finisca per contenere un valore diverso da quelli validi. Per restringere il contenuto di una variabile a un certo insieme di valori si può definire la variabile come di tipo enumerazione (enum) per elencare solo i valori legittimi. Ad esempio “`enum PunteggioFilm{E, P, B}`”. Notare come non ci sia un ; dopo l’istruzione anche se il suo utilizzo non verrebbe segnato come errore. Si noti anche che i valori non sono racchiusi tra apici perché non sono di tipo `char`, questo perché un’enumerazione si comporta come un tipo classe. Gli elementi elencati tra le graffe rappresentano i valori che è possibile assegnare alla variabile, quindi si deve segnare “`PunteggioFilm.punteggio`” come variabile e per assegnarli si deve scrivere “`punteggio = PunteggioFilm.B`”. Verranno presentate meglio successivamente.

L’enumerazione deve essere posta all’esterno del main perché non può essere una variabile locale.

4 FLUSSO DI CONTROLLO: I CICLI

4.1 CICLI IN JAVA

Spesso i programmi hanno la necessità di ripetere una o più azioni. La parte del programma che ripete un'istruzione o un gruppo di istruzioni è chiamata **ciclo (loop)**. L'istruzione o il gruppo di istruzioni che vengono ripetuti nel ciclo sono chiamati corpo (body) del ciclo. Ogni ripetizione del corpo del ciclo è chiamata iterazione del ciclo.

Quando si definisce un ciclo, occorre determinare l'azione svolta nel corpo del ciclo. È necessario, inoltre, definire un meccanismo che permetta di determinare quando la ripetizione del corpo del ciclo deve terminare.

4.1.1 Istruzione while

Un'istruzione `while` ripete più e più volte l'azione definita nel corpo del ciclo finché un'espressione booleana di controllo rimane vera. Spesso il corpo del ciclo viene ripetuto fintantoché l'espressione booleana di controllo rimane vera. Spesso il corpo del ciclo è costituito da un'istruzione composta,

```

1 import java.util.Scanner;
2
3 public class WhileDemo {
4     public static void main(String[] args) {
5
6         int conteggio, numero;
7
8         System.out.println("Inserisci un numero");
9         Scanner tastiera = new Scanner(System.in);
10        numero = tastiera.nextInt();
11
12        conteggio = 1;
13        while (conteggio <= numero) {
14            System.out.print(conteggio + ", ");
15            conteggio++;
16        }
17
18        System.out.println();
19        System.out.println("Sorpresa!");
20    }
21 }
22

```

racchiusa tra parentesi graffe.

```
Inserisci un numero
6
1, 2, 3, 4, 5, 6,
Sorpresa!
```

Un ciclo `while` può anche essere eseguito zero volte nel caso l'espressione booleana non sia vera.

4.1.2 Istruzione do-while

È molto simile al ciclo `while` ma ha una differenza fondamentale: il ciclo viene eseguito ALMENO una volta.

```

1 import java.util.Scanner;
2
3 public class DoWhileDemo {
4     public static void main(String[] args) {
5
6         int conteggio, numero;
7
8         System.out.println("Inserisci un numero");
9         Scanner tastiera = new Scanner(System.in);
10        numero = tastiera.nextInt();
11
12        conteggio = 1;
13
14        ESCdo {
15            System.out.print(conteggio + ", ");
16            conteggio++;
17        } while (conteggio <= numero);
18
19        System.out.println();
20        System.out.println("Sorpresa!");
21    }
22 }

```

```
Inserisci un numero
0
1,
Sorpresa!
```

Notare nell'esempio che anche inserendo a tastiera numero = 0 (e quindi conteggio = 1 < numero) il ciclo venga eseguito una volta.

```

1 import java.util.Scanner;
2
3 /**
4  * Calcola la media di un elenco di voti non negativi relativi a un esame.
5  * Ripete il calcolo per piu' esami fino a quando l'utente non indica di
6  * fermarsi.
7 */
8
9 public class MediaEsami {
10
11     public static void main(String[] args) {
12
13         System.out.println("Questo programma calcola la media");
14         System.out.println("di un elenco di voti non negativi.");
15         double somma;
16         int numeroStudenti;
17         double successivo;
18         String risposta;
19         Scanner tastiera = new Scanner(System.in);
20
21         do { //Inizio istruzione do
22             System.out.println();
23             System.out.println("Inserisci tutti i voti di cui");
24             System.out.println("vuoi calcolare la media.");
25             System.out.println("Poi inserisci un numero negativo");
26             System.out.println("dopo che hai inserito tutti i voti.");
27             somma = 0;
28             numeroStudenti = 0;
29             successivo = tastiera.nextDouble();
30             while (successivo >= 0) { //Istruzione while all'interno del do
31                 somma = somma + successivo;
32                 numeroStudenti++;
33                 successivo = tastiera.nextDouble();
34             }
35
36             if (numeroStudenti > 0) { //Istruzione if all'interno del do
37                 System.out.println("La media e' " + (somma / numeroStudenti));
38             } else {
39                 System.out.println("La media non e' calcolabile.");
30             }
31
32             System.out.println("Vuoi calcolare la media di un altro esame?");
33             System.out.println("Scrivi si o no.");
34             risposta = tastiera.next();
35             } while (risposta.equalsIgnoreCase("si")); //Fine istruzione do
36         }
37     }
38 }
```

L'istruzione for permette di scrivere facilmente un ciclo controllato da un contatore. L'istruzione è composta da **tre istruzioni**:

- Istruzione da eseguire prima dell'inizio delle iterazioni
- Condizione
- Istruzione di fine ciclo

```

1 public class ForDemo {
2
3     public static void main(String[] args) {
4
5         int contoAllaRovescia;
6
7         for (contoAllaRovescia = 3; contoAllaRovescia >= 0; contoAllaRovescia--) {
8             System.out.println(contoAllaRovescia);
9             System.out.println("attendere...");
10        }
11
12        System.out.println("Partito!");
13    }
14 }
```

Attenzione! Scrivere “`for(...);`” porta ad un'istruzione vuota poiché non bisogna mettere il `;`. Allo stesso modo scrivere “`while(...);`” porta al medesimo risultato.

```

3
attendere...
2
attendere...
1
attendere...
0
attendere...
Partito!

```

4.1.5 Dichiare variabili all'interno di un'istruzione for

Nella parte dell'inizializzazione di un'istruzione all'interno della parentesi di un `for` si possono dichiarare variabili. Ad esempio scrivere “`for (int n=1; n<=10; n++)`” porta alla creazione di una variabile accessibile solo all'interno del ciclo `for` nella quale è stata dichiarata. Si dice che `n` è “locale” al ciclo `for`.

4.1.3 Cicli infiniti

Bisogna prestare attenzione ai cicli infiniti. Se ad esempio venisse dato come estremo superiore un numero impossibile da raggiungere, ad esempio

```

While (numero < numero + 1) {
    System.out.printf("%d\r", numero);
    numero++;
}
```

Il ciclo non terminerebbe mai. Nella maggior parte dei sistemi operativi, durante l'esecuzione, si può premere **CTRL + C** per interrompere l'esecuzione.

4.1.4 Istruzione for

La porzione di programma in cui è disponibile una variabile è detto visibilità (scope) della variabile. Quando una variabile viene usata solo all'interno del ciclo `for` è buona prassi definirla così e non prima.

4.1.6 Usare una virgola in un'istruzione `for`

Un'istruzione `for` può anche eseguire più istruzioni di inizializzazione. Per usare più istruzioni basta separare le istruzioni con una virgola come nell'esempio seguente:

```
for(int numero = 1, int prodotto = 1; numero <=10; numero++)
    prodotto = prodotto * numero;
```

Si noti che in questo caso per separare due azioni è stata utilizzata una virgola e non un punto e virgola. La virgola utilizzata in questo contesto viene detta operatore virgola.

In maniera analoga si possono avere più azioni di aggiornamento:

```
for(int numero = 1, int prodotto = 1; numero <=10; prodotto = prodotto * numero,
    numero++)
```

4.1.7 Istruzioni `for-each`

Quando è necessario ripetere una certa azione per ciascun elemento di un'enumerazione, si può usare una qualsiasi delle istruzioni presentate nei paragrafi precedenti. Tuttavia Java fornisce un'altra forma di `for` da utilizzare in questo caso: `for-each`.

```
1 public class SemeCarte {
2
3     enum Seme{CUORI, QUADRI, PICCHE, FIORI};
4
5     public static void main(String[] args) {
6
7         for(Seme semeSuccessivo : Seme.values())
8             System.out.println(semeSuccessivo);
9     }
10 }
```

L'espressione `Seme.values()`
rappresenta tutti i valori dell'insieme.

4.2 PROGRAMMARE CON I CICLI

Un ciclo è costituito tipicamente da tre elementi:

- Le istruzioni di inizializzazione che devono precedere l'iterazione
- Il corpo del ciclo
- Il meccanismo per la terminazione del ciclo

4.2.1 Il corpo del ciclo

Un primo modo per definire il corpo di un ciclo è quello di scrivere la sequenza di azioni che devono essere eseguite dal programma quando viene avviato. Dopo aver scritto la sequenza di azioni, occorre individuare un insieme di azioni da ripetere più volte.

4.2.2 Controllare il numero di iterazioni in un ciclo

Esistono cicli nei quali si conosce esattamente il numero di iterazioni prima di eseguire il ciclo e vengono chiamati "cicli count-controlled". I cicli count-controlled non devono essere necessariamente implementati con istruzioni `for`, anche se questo è il modo più semplice per farlo.

Un modo semplice per capire quando terminare un’iterazione consiste nel chiedere all’utente quando sia il momento giusto per farlo. Questa tecnica è detta “ask before iterating” e consiste nel richiedere un input specifico all’utente per il quale si interrompe l’esecuzione del ciclo. Questo sistema, però, non è adatto a lunghe esecuzioni in quanto diventerebbe scomodo per l’utente.

Per questi casi è possibile usare un valore sentinella per indicare la fine dell’input, ad esempio chiedere di inserire un numero negativo per terminare la sequenza.

4.2.3 Istruzioni break e continue nei cicli

I cicli possono terminare quando l’espressione booleana di controllo risulta falsa o se, forzosamente, viene eseguita un’istruzione break. L’istruzione break esce solo dal ciclo più interno nel quale è contenuto.

L’istruzione continue all’interno del corpo di un ciclo termina, invece, solo l’iterazione corrente e passa a quella successiva.

Queste due istruzioni sarebbero da evitare in quanto rendono difficoltosa la lettura del codice.

4.2.4 Cicli difettosi

I programmi che contengono cicli sono soggetti a difetti con maggiore probabilità dei programmi più semplici. Fortunatamente gli errori nei cicli sono riconducibili a poche tipologie, elencate di seguito. I due errori più comuni sono:

- Cicli infiniti indesiderati
- Errori di un’unità

Nel primo caso bisogna stare attenti perché un ciclo di istruzioni potrebbe comportarsi normalmente per certi valori e, solo per alcuni, trasformarsi in un ciclo infinito.

L’errore di una unità è quando il ciclo viene eseguita una volta di troppo o una volta in meno di quanto desiderato. Spesso questi errori derivano da un’istruzione booleana sbagliata concettualmente.

4.2.5 Tracciare le variabili

Se un programma non si comporta come si vorrebbe ma non si riesce a capire che cosa ci sia di sbagliato al suo interno, il tracciamento di alcune variabili chiave potrebbe aiutare a risolvere il problema.

Tracciare le variabili vuol dire osservare il valore che assumono le variabili mentre il programma è in esecuzione. Un metodo classico di tracciamento è la richiesta di stampa continua delle variabili tracciate in modo da poter vedere sempre quando vengono eseguite le modifiche e quando no.

Alle volte durante il debugging di un programma si vuole evitare temporaneamente di eseguire le istruzioni di tracciamento del valore delle variabili. Questo può essere fatto definendo una costante booleana che può chiamarsi DEBUG e definirla come vera. Nel caso si vogliano evitare queste istruzioni in futuro basta assegnare il valore false.

```
Public static final boolean DEBUG = true;
...
If (DEBUG) {....}
```

4.2.6 Controllo delle asserzioni

Un’asserzione è un’istruzione che specifica un’ipotesi sullo stato del programma. Un’asserzione può essere vera o falsa ma deve risultare vera se non ci sono errori nel programma. In Java si può verificare in modo

automatico se un'asserzione è vera o falsa e terminare il programma con un messaggio d'errore qualora questa non sia vera. In Java un controllo di asserzione ha il seguente aspetto:

```
assert espressione_booleana;
```

se si esegue il programma in un certo modo e l'espressione è falsa, il programma termina dopo aver mostrato un messaggio di errore che indica il fallimento dell'asserzione. Il controllo delle asserzioni può essere abilitato o disabilitato a piacere quando è utile. Il comando comunemente utilizzato per eseguire i programmi lascia disattivato il controllo delle asserzioni. Per eseguire il programma attivando il controllo delle asserzioni bisogna usare il seguente comando: `java -enableassertions Programma`.

5 I METODI: CONCETTI BASE

5.1 DEFINIZIONE E INVOCAZIONE DI METODI

Alcune sequenze di istruzioni possono dover essere ripetute più volte all'interno di un programma. Risulta quindi comodo poter scrivere tali sequenze una volta sola e poter far riferimento ad essere all'interno del programma tutte le volte che la loro esecuzione risulta necessaria. I metodi costituiscono lo strumento di programmazione che realizza quanto sopra.

Un metodo raggruppa una sequenza di istruzioni che realizzano una funzionalità del programma e assegna loro un nome. Ogni qualvolta è necessario eseguire quella funzionalità, è sufficiente richiamarla attraverso il nome. Quando si usa un metodo si dice che si "invoca" o "chiama" il metodo.

Java ha due tipi di metodi:

- Metodi che restituiscono un valore
- Metodi che eseguono alcune istruzioni, ma non restituiscono alcun valore (`void`)

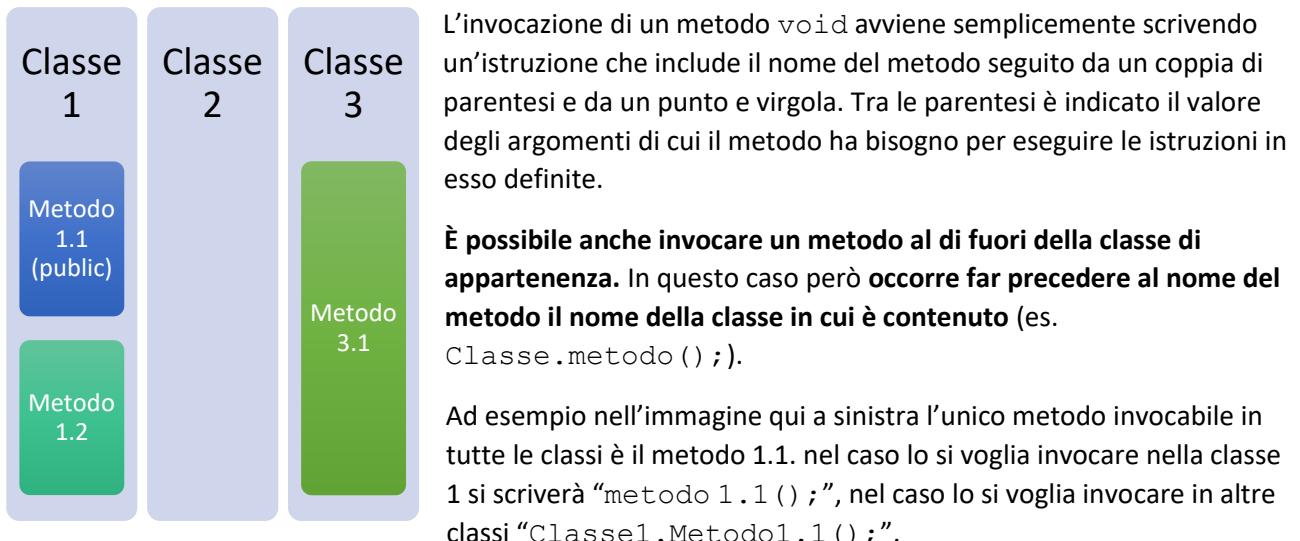
5.1.1 Definire e invocare metodi `void`

Un metodo è definito all'interno di una classe. Pertanto si dice che un metodo appartiene alla classe in cui è stato definito. Esistono delle parole chiamate "modificatori d'accesso" che modificano l'accessibilità dei metodi all'interno dei programmi:

- **Public:** un metodo `public` può essere invocato in ogni parte del programma, anche in classi diverse da quelle in cui è definito
- **Static:** un metodo `static` sono detti "metodi statico" e non possono modificare le variabili d'istanza ma solo quelle statiche
- **Void:** per definire un metodo che non restituisce valori si specifica la chiave `void`. Dopo il nome del metodo si mette una coppia di parentesi tonde. Tra le parentesi è indicata la specifica degli argomenti di cui il metodo ha bisogno per poter eseguire le istruzioni in esso definite (es. `public static void Esempio ()`)

Questa prima parte della definizione di un metodo è detta "intestazione".

Dopo l'intestazione viene riportata la parte rimanente, il "corpo", ovvero le istruzioni contenute nel metodo. Le variabili utilizzate per la definizione di un metodo devono essere dichiarate all'interno della definizione del metodo stesso e vengono dette "variabili locali".



5.1.2 Definire metodi che restituiscono un valore

I metodi che restituiscono un valore vengono definiti in maniera simile ai metodi `void` con l'aggiunta della specifica del tipo di valore che restituiscono.

Attenzione!

Java non ha le funzioni così come vengono definite in altri linguaggi. Infatti negli altri linguaggi vengono chiamate funzioni i metodi che restituiscono valori in quanto svolgono il concetto matematico di funzione. In Java, indipendentemente dal fatto che restituisca o meno un valore, si usa il nome “metodo”.

L'intestazione di un metodo che restituisce un valore è simile a quella di un metodo `void`. L'unica differenza è che un metodo che restituisce un valore indica, al posto della parola chiave `void`, il nome del tipo di ritorno. L'intestazione inizia con la chiave `public`, seguita da `static` e quindi dal valore restituito dal metodo, seguito a sua volta dal nome del metodo e da una coppia di parentesi che racchiudono i valori passati.

Il corpo è uguale a quello di un metodo `void` tranne per l'espressione di ritorno al suo interno `return` che indica il valore da restituire all'metodo chiamate. Un metodo che restituisce un valore, lo si può invocare in qualsiasi parte del codice in cui si potrebbe usare un elemento dello stesso tipo di ritorno.

5.1.3 Variabili locali

Una variabile dichiarata all'interno di un metodo è locale di tale metodo. Si dice che la variabile è locale perché può essere usata esclusivamente all'interno del metodo che l'ha definita.

Si supponga di avere due metodi definiti all'interno di una stessa classe. Se ciascun metodo dichiara una variabile con lo stesso nome si avranno due variabili che hanno lo stesso nome all'interno della stessa classe. Questa situazione è perfettamente legale. Notare qua nell'esempio che entrambi i metodi definiscono la variabile double risultato.

```

1 import java.util.Scanner;
2
3 public class CerchioTerzaVersioneDemo {
4
5     public static double areaCerchio(double raggio){ //metodo
6         return 3.14159 * raggio * raggio;
7     }
8
9
10    public static void main(String[] args) { //main
11
12        System.out.println("Area del cerchio di raggio 2: " + areaCerchio(2));
13        //chiamata funzione, passato 2
14        System.out.print("Si inserisca il raggio del cerchio: ");
15        Scanner tastiera = new Scanner(System.in);
16
17        double valore = tastiera.nextDouble();
18        double area = areaCerchio(valore); //chiamata funzione, passato parametro
19        //valore"
20
21    }

```

Java non ha variabili globali. Tuttavia ha le variabili statiche, che vedremo più avanti, le quali in un certo senso possono essere considerate come globali.

5.1.4 Blocchi

Il termine blocco indica un insieme di istruzioni racchiuse tra parentesi graffe. Una variabile definita all'interno di un blocco è visibile solo in esso.

5.1.5 Parametri di tipo primitivo

Quando all'interno di una funzione si definiscono dei parametri che devono essere passati dal metodo chiamante al metodo chiamato si dice che si utilizza un parametro formale. Come si vede nell'esempio il valore "raggio" viene prima definito come 2 e successivamente viene passato il parametro "valore", precedentemente immesso a piacere dall'utente. 2 e valore vengono chiamati "argomenti". Dato che viene usato il valore dell'argomento, questo meccanismo di assegnamento degli argomenti ai parametri formali è chiamato "passaggio per valore" o "chiamata per valore".

Si tenga da conto che non sempre i valori passati devono essere dello stesso tipo del valore ricevente. Infatti Java attua automaticamente una conversione di tipo qualora nell'invocazione di metodo venga usato un argomento il cui tipo non corrisponde a quello del parametro formale. Per esempio se il tipo dell'argomento in un'invocazione di metodo è int e il parametro formale è double, java convertirà int nel corrispondente valore double.

È possibile avere più di un parametro formale nella definizione di un metodo. In questo caso ciascun parametro formale è elencato nell'intestazione del metodo e ciascun parametro è preceduto da un tipo. Ad esempio "public static void faiQualcosa (int n1, int n2, double costo, char codice)". Anche se più parametri hanno lo stesso tipo bisogna sempre far precedere la denominazione del tipo al parametro. Quando si dovrà chiamare il metodo si scriverà, per esempio "faiQualcosa (42, 100, 9.99, 'Z')";.

5.1.6 Ancora sull'istruzione `return`

```

1  public static int maggiore(int primo, int secondo){ 1  public static int maggiore(int primo, int secondo){
2      int risultato; 2      int risultato;
3      if (primo >= secondo){ 4
4          risultato=primo; 5      if (primo >= secondo){
5              risultato=primo; 6          risultato=primo;
6          } 7              return risultato; //primo return
7      else{ 8
8          risultato=secondo; 9      }
9      } 10     else{
10         risultato=secondo; 11         risultato=secondo;
11     } 12         return risultato; //secondo return
12 } 13     }
14 }
```

I programmi qua sopra sono identici nel risultato ma diversi dal punto di vista della programmazione. Infatti il primo presenta una sola istruzione `return` mentre il secondo due. In questo caso, essendo l'istruzione semplice, non si presentano grandi problemi di lettura ma nel caso il metodo fosse stato più complicato, il secondo metodo di scrittura sarebbe stato di comprensione più difficoltosa. È buona norma, quindi, usare una sola istruzione di `return`.

È possibile utilizzare le istruzioni `return` anche all'interno dei metodi `void`. Basta scrivere `return` senza farlo seguire da alcuna istruzione. In questo modo l'azione che ne segue è la sospensione del metodo e il ritorno al metodo chiamante.

5.2 LA CLASSE MATH

La classe predefinita Math fornisce una serie di metodi matematici standard. Questa classe viene fornita automaticamente nel linguaggio Java, pertanto non è necessaria alcuna dichiarazione d'importazione. I metodi della classe Math sono statici, di conseguenza possono essere invocati utilizzando il nome della classe Math.

La classe Math definisce anche due costanti: PI (pi greco) ed E (numero di Nepero). È buona norma utilizzare queste costanti e non definirle da sé. Vengono chiamate come `Math.PI` e `Math.E`.

Nome	Descrizione	Tipo di argomento	Tipo restituito	Esempio	Valore restituito
Pow	Potenza	Double	Double	Math.pow(2.0, 3.0)	8.0
Abs	Valore assoluto	Int, long, float, double	Int, long, float, double	Math.abs(-7)	7
Max	Massimo	Int, long, float, double	Int, long, float, double	Math.max(5.5, 5.3)	5.5
min	Minimo	Int, long, float, double	Int, long, float, double	Math.min(5.5, 5.3)	5.3
Round	Arrotondamento	Float, double	Int, long rispettivamente	Math.round(6.2)	6.0
Ceil	Intero maggiore	Double	Double	Math.ceil(6.3)	7.0
Floor	Intero minore	Double	Double	Math.floor(6.3)	6.0
sqrt	Radice quadrata	Double	Double	Math.sqrt(9)	3.0

5.3 COSA ACCADE REALMENTE QUANDO SI INVOCA UN METODO?

Quando si invoca un metodo l'esecuzione passa al corpo di attivazione, che contiene tutte le informazioni necessarie per gestire correttamente l'esecuzione del metodo invocato. Un record di attivazione contiene dati relativi al metodo invocato e informazioni per la gestione dei metodi da esso eventualmente invocati.

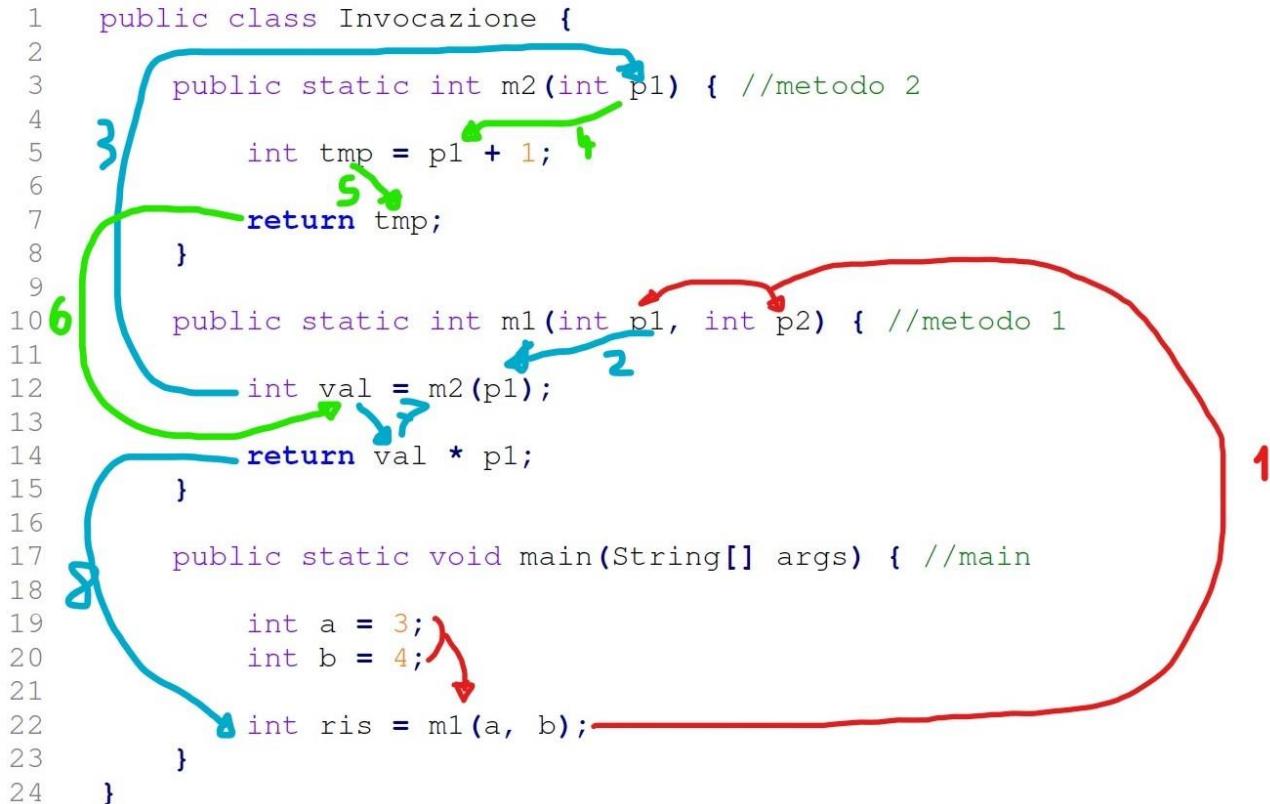
Dati del metodo	Parametro 1: valore Parametro 2: valore ... Parametro n: valore	Valori passati per indirizzo	
	Variabile locale 1: valore Variabile locale 2: valore ... Variabile locale n: valore		
Informazioni per la gestione dei metodi invocati	Rientro: indirizzo	Risultato: valore	Valori interni al metodo

Le informazioni relative al metodo invocato includono parametri formali del metodo con gli argomenti attuali e le variabili locali al metodo con i valori che man mano assunti durante l'esecuzione del metodo.

Le informazioni per la gestione dei metodi da esso invocati includono:

- L'indirizzo di rientro: specifica quale istruzione del metodo deve essere eseguita nel momento in cui il metodo termina la sua esecuzione
- Il risultato

Il record d'attivazione viene quindi creato dinamicamente nel momento in cui il metodo viene chiamato e viene posto in cima a un'area di memoria denominata "stack" (pila). Rimane nello stack per tutto il tempo in cui il metodo è in esecuzione e viene rimosso al termine dell'esecuzione. Metodi che invocano altri metodi danno origine ad una sequenza di record d'attivazione gestiti secondo la modalità LIFO (Last In First Out).



- Main: passa le variabili a e b alla funzione m1 (1)
 - M1: p1 = a, p2 = b
 - Passa p1 al metodo m2
 - M2: p1 = p1
 - Tmp = p1 + 1
 - Restituisce tmp a m1
 - M1 registra tmp come val
 - Restituisce val * p1 a main
- Ris = val * p1

I valori finali sono a = 3, b = 4, tmp = val = 4, ris = val * p1 = 12

5.4 COME SCRIVERE I METODI

5.4.1 Decomposizione

Se una sotto attività è di grandi dimensioni, è buona norma scomporla in sotto attività più piccole da risolvere separatamente. Queste sotto attività possono essere a loro volta scomposte in sotto attività più piccole finché non diventano abbastanza piccole da poter essere progettate e implementate facilmente.

5.4.2 *Affrontare i problemi di compilazione*

Il compilatore controlla che siano state svolte tutte le operazioni necessarie, come l'inizializzazione delle variabili o l'inclusione di un'istruzione `return` nella definizione di un metodo che deve restituire un valore. A volte il compilatore chiede di effettuare una di queste operazioni anche se il programmatore è convinto di averla effettuata o di non aver bisogno di farla. In questi casi, di solito ha ragione il compilatore.

5.4.3 *Collaudare i metodi*

Per verificare la correttezza di un metodo si può usare un programma driver. I programmi driver vengono usati dal programmatore per collaudare il sistema e possono essere molto semplici. Per esempio non hanno bisogno di output elaborati o di interfacce grafiche. Tutto quello che questi programmi devono fare è fornire al metodo da collaudare una serie di argomenti e invocare il metodo stesso. Ci sono più metodi di collaudo:

- **Testing bottom-up:** se un metodo A invoca il metodo B, l'approccio bottom-up prevede che il metodo B venga collaudato a fondo prima di collaudare il metodo A. L'approccio bottom-up è efficace, tuttavia potrebbe diventare tedioso.
- **Utilizzo di uno stub (prototipo):** è una versione semplificata di un metodo che non può essere utilizzata all'interno del programma finale ma è sufficiente per permettere il collaudo del sistema ed è abbastanza semplice per far sì che il programmatore sia certo che il risultato restituito da questo metodo sia corretto.

6 ARRAY

6.1 CONCETTI DI BASE SUGLI ARRAY

Un array è una collezione di elementi dello stesso tipo. È paragonabile a un elenco di variabili, ma con una gestione dei nomi più immediata e compatta.

6.1.1 Creazione e accesso a un array

In java un array è un particolare tipo di oggetto, ma è più semplice considerarlo come una collezione di variabili dello stesso tipo. Ad esempio un elenco di sette variabili double viene indicato come:

```
double [] temperatura = new double[7]
```

Le variabili come temperatura [0] e temperatura [1] che hanno un'espressione intera fra parentesi quadre sono dette “variabili indicizzate”, “elementi dell'array” o semplicemente “elementi”. L'espressione tra parentesi quadre è detta “indice”. **Gli indici partono da 0.**

Le variabili sono molto più che semplici variabili di tipo double. Il **numero fra parentesi quadre** è parte integrante del nome di ognuna di queste variabili e **non deve essere necessariamente una costante intera**. Infatti è possibile utilizzare una qualsiasi espressione intera che assuma valori tra 0 e 6, anche una variabile separata, ad esempio:

```
1 double[] temperatura = new double[7] //dichiarazione
2
3 Scanner tastiera = new Scanner(System.in);
4 System.out.println("Inserire il numero del giorno (0-6)");
5
6 int indice = tastiera.nextInt(); //creazione indice
7
8 System.out.println("Inserire la temperatura per il giorno " + giorno);
9 temperatura[indice] = tastiera.nextDouble(); //assegnazione valore
```

Poiché un indice può essere un'espressione, è possibile scrivere un ciclo per l'inserimento dei valori di temperatura nell'array e anche per la stampa a video. Ad esempio:

```
for (indice=0; indice<7; indice++){
    System.out.println("Inserire la temperatura per il giorno " + indice);
    temperatura[indice] = tastiera.nextDouble(); //assegnazione valore
    System.out.println("La temperatura del giorno " + indice + " è " +
        temperatura[indice]);
}
```

6.1.2 Dettagli sugli array

È possibile creare un array tramite l'operazione new.

Il tipo degli elementi dell'array è detto “tipo base”, il numero di elementi “lunghezza”, “dimensione” o “capacità”.

Nel caso si sappia il numero di elementi dell'array a priori è buona norma creare una costante come NUMERO_ELEMENTI_ARRAY = numero conosciuto e poi usare questa costante all'interno delle parentesi.

È inoltre possibile usare un'espressione che restituisce un intero al posto dell'indice dell'array, ad esempio [indice + 3].

6.1.3 La proprietà `length`

È possibile far riferimento alle proprietà accessibili degli oggetti utilizzando una notazione che prevede il nome dell'oggetto seguito da un punto e quindi dal nome della proprietà. **Un array ha una sola proprietà accessibile: `length`.** Tale variabile è la lunghezza dell'array. Ad esempio per `temperatura[20]`, `temperatura.length` darà 20.

Utilizzare nel codice “`temperatura.length`” invece che un valore semplice per l'output è più comprensibile al lettore e al programmatore.

```

1  /* Legge I valori di 7 temperature inserite dall'utente e mostra quai di
2 Queste sono al di sopra e al di sotto della media delle temperature stesse. */
3 import java.util.Scanner;
4
5 public class ArrayDiTemperature2 {
6
7     public static void main(String[] args) {
8
9         Scanner tastiera = new Scanner(System.in);
10
11         System.out.println("Quante temperature si devono inserire?");
12
13         int dimensione = tastiera.nextInt();
14         double[] temperatura = new double[dimensione];
15
16         // Lettura delle temperature e calcolo della loro media:
17         System.out.println("Inserire " + temperatura.length + " temperature:");
18         double somma = 0;
19
20         for (int indice = 0; indice < temperatura.length; indice++) {
21             temperatura[indice] = tastiera.nextDouble();
22             somma = somma + temperatura[indice];
23         }
24
25         double media = somma / temperatura.length;
26         System.out.println("La temperatura media e' " + media);
27
28         // Mostra ogni temperatura e la relazione rispetto alla temperatura media:
29         System.out.println("Le " + temperatura.length + " temperature sono");
30
31         for (int indice = 0; indice < temperatura.length; indice++) {
32             if (temperatura[indice] < media)
33                 System.out.println(temperatura[indice] + " sotto la media");
34             else if (temperatura[indice] > media)
35                 System.out.println(temperatura[indice] + " sopra la media");
36             else //temperatura[indice] == media
37                 System.out.println(temperatura[indice] + " pari alla media");
38         }
39         System.out.println("Buona settimana.");
40     }
41 }
```

6.1.4 Ulteriori dettagli sugli indici di un array

Poiché gli indici di un array iniziano da 0, vuol dire che le celle accessibili sono `array.length-1`.

Un errore comune di programmazione è cercare di accedere a indici non esistenti. Se l'indice restituito da un'espressione non è compreso fra 0 e `array.length-1` si dice che l'array è “fuori dai limiti” o “non valido”. **Anche se il codice contiene un indice non valido il compilatore non lo noterà ma verrà restituito un errore a runtime.**

Spesso gli indici di un array escono dai limiti in seguito a troppe iterazioni. Ad esempio se venisse chiesto di riempire un array e venisse usato un valore sentinella negativo per interrompere l'iterazione, si potrebbe andare oltre al limite. Per questo l'ideale sarebbe usare una clausola limitatrice in `while`, `if` o altri costrutti.

Nel caso si voglia accedere ad un array di lunghezza 100 ma si voglia usare come numerazione 1-100 e non 0-99 per gli indici, è buona norma usare un array di lunghezza 101 e far partire il ciclo di iterazione da indice = 1. Non ci si deve preoccupare di sprecare spazio se questo porta ad un codice più pulito e comprensibile.

6.1.5 Inizializzare gli array

Un array può essere inizializzato in fase di dichiarazione. Per fare ciò, basta racchiudere i valori delle variabili indicizzate fra parentesi graffe dopo l'operatore di assegnamento. Ad esempio:

```
double[] valore = {3.3, 15.8, 9.7}
```

La dimensione dell'array diventa in tal caso la dimensione minima per contenere i dati inseriti.

Nel caso l'array non venga inizializzato i valori assumeranno il valore base per il loro tipo.

6.1.6 Utilizzare il ciclo for-each con gli array

Si può utilizzare il ciclo for-each per interagire con l'array.

```
1 double[] a = new double [10];      Si può leggere la linea che inizia con il for come "per ogni
2 //codice che riempie l'array a      elemento in a, esegui le operazioni che seguono". Si noti
3 for (double elemento:a)          che la variabile "elemento" è dello stesso tipo dell'array.
4     System.out.println(elemento);
```

L'utilizzo del ciclo `for-each` può rendere più pulito e meno soggetto ad errori il codice. Se non serve utilizzare l'indice dell'array in un ciclo `for` per altro scopo che non sia scorrere gli elementi dell'array è preferibile usare il ciclo `for-each`.

6.2 UTILIZZARE GLI ARRAY NEI METODI

I metodi possono ricevere come argomento una variabile indicizzata o un intero array e possono restituire un array.

6.2.1 Variabili indicizzate come argomenti di un metodo

```

1 import java.util.Scanner;
2
3 /*Utilizzo di variabili indicizzate.*/
4 public class ArgomentiDemo {
5
6     public static void main(String[] args) { //main
7         Scanner tastiera = new Scanner(System.in);
8         System.out.println("Inserire il voto dell'esame 1:");
9         int punteggioIniziale = tastiera.nextInt();
10        int[] punteggioSeguente = new int[3];
11
12        for (int i = 0; i < punteggioSeguente.length; i++)
13            punteggioSeguente[i] = punteggioIniziale + 5 * i;
14
15        for (int i = 0; i < punteggioSeguente.length; i++) {
16            double possibileMedia = getMedia(punteggioIniziale, punteggioSeguente[i]);
17            System.out.println("Se il voto all'esame 2 sara' " +
18                punteggioSeguente[i]);
19            System.out.println("la media sara' uguale a " + possibileMedia);
20        }
21
22        public static double getMedia(int n1, int n2) { //metodo
23            return (n1 + n2) / 2.0;
24        }
25    }

```

Nell'esempio la variabile indicizzata è utilizzata come argomento di un metodo. Il metodo `getMedia` ha due argomenti di tipo `int`. L'array `punteggioSeguente` ha `int` come tipo base.

```

Inserire il voto dell'esame 1:
5
Se il voto all'esame 2 sara' 5
la media sara' uguale a 5.0
Se il voto all'esame 2 sara' 10
la media sara' uguale a 7.5
Se il voto all'esame 2 sara' 15
la media sara' uguale a 10.0

```

6.2.2 Array come argomenti di un metodo

Il modo con cui si specifica che l'argomento di un metodo è un array simile al modo con cui si dichiara un array. Per esempio il seguente metodo `incrementaArrayDi2` accetta come argomento un qualsiasi array di `double`:

```

1 public class classeEsempio{
2
3     public static void incrementaArrayDi2(double[] unarray) {
4         for(int i=0; i<unArray.length; i++)
5             unArray[i] = unArray[i] + 2;
6     }
7 }
8
9     /*inserire il resto del codice per la definizione della classe
10 */

```

Quando si utilizza come parametro un array è necessario indicare il tipo base dell'array, ma non si deve impostare la lunghezza dell'array stesso. Tale sintassi è simile a quella alternativa utilizzata in fase di dichiarazione dell'array.

Per illustrare l'utilizzo della classe esempio, si consideri il seguente codice:

```

1  public class classeEsempio{
2
3      public static void incrementaArrayDi2(double[] unarray){
4          for(int i=0; i<unArray.length; i++)
5              unArray[i] = unArray[i] + 2;
6      }
7
8
9      public static void main (String[] args){
10         double[] a = new double[10];
11         double[] b = new double[30];
12
13         incrementaArrayDi2(a);
14         incrementaArrayDi2(b);
15     }
16 }
```

Quindi, quando si passa un intero array come argomento di un metodo, non devono essere usate le parentesi quadre. **Si può passare un array di qualsiasi lunghezza come argomento a un metodo che accetta come parametro un array.** Un metodo può modificare il valore degli elementi di un array passato come argomento.

6.2.3 Argomenti del metodo main

La dichiarazione del metodo main di un programma è la seguente

```
public static void main (String[] args)
```

la dichiarazione del parametro `String[] args` indica che `args` è un array il cui tipo di base è `String`. Di conseguenza, il metodo main accetta come parametro un array di valori di tipo `String`. Ma finora non si è mai passato alcun argomento al metodo main, come funziona quindi?

Quando si esegue un programma il metodo main viene invocato automaticamente e come argomento gli viene fornito un array di stringhe di default. È però possibile fornire delle stringhe come argomento del programma e queste stringhe diventano automaticamente argomenti dell'array `args` che rappresenta l'argomento del main.

Di solito si passano argomenti a un programma quando lo si esegue dalla riga di comando, ad esempio:

```
java ProgrammaDiTest Mario Rossi
```

questo comando assegna “Mario” ad `args[0]` e “Rossi” ad `args[1]`. Queste due variabili possono essere utilizzate all’interno del metodo main.

Ad esempio:

```

1  public class ProgrammaDiTest{
2      public static void main (String[] args){
3
4          System.out.println("Ciao " + args[0] + " " + args[1]);
5      }
6  }
```

```
C:\Users\jacop\Desktop>java ProgrammaDiTest Mario Rossi
Ciao Mario Rossi
```

È necessario segnalare che `args` è un array di stringhe, se si volessero usare numeri, si devono convertire le relative stringhe in uno dei tipi numerici. Poiché `args` è un parametro, è possibile utilizzare qualsiasi altro nome al posto di questo, l’utilizzo di `args` è una pratica comune.

6.2.4 Assegnamento e uguaglianza di array

È necessario comprendere come gli array gestiscano gli operatori = e ==. Essendo oggetti vengono trattati come qualsiasi altro oggetto: assegnamento e uguaglianza.

Bisogna comprendere un'altra peculiarità. **Essendo un array una sequenza di celle di memoria, quando si esegue un assegnamento tra array del tipo b = a non si sta dicendo “l'array b assume tutti i valori dell'array a”, bensì “l'array b PUNTA AGLI STESSI INDIRIZZI dell'array a”**, in questo caso, anche se un valore di a venisse modificato successivamente alla dichiarazione di assegnamento, il valore verrebbe cambiato ugualmente in b.

```

1 import java.util.Scanner;
2
3 public class ProgrammaDiTest{
4     public static void main (String[] args){
5
6         Scanner tastiera = new Scanner(System.in);
7
8         int i;
9         int [] a = new int[3];
10        int [] b = new int[3];
11
12        for (i=0; i<a.length; i++)
13            a[i] = tastiera.nextInt();
14        for (i=0; i<b.length; i++)
15            b[i] = tastiera.nextInt();
16
17        for (i=0; i<a.length; i++)
18            System.out.println("a[" + i + "] = " + a[i]);
19
20        System.out.println("");
21        for (i=0; i<b.length; i++)
22            System.out.println("b[" + i + "] = " + b[i]);
23        System.out.println("");
24        b = a; //assegnamento degli stessi indirizzi di memoria
25        System.out.println("E' stato eseguito l'assegnamento b=a");
26
27        for (i=0; i<a.length; i++)
28            System.out.println("a[" + i + "] = " + a[i]);
29        System.out.println("");
30        for (i=0; i<b.length; i++)
31            System.out.println("b[" + i + "] = " + b[i]);
32        System.out.println("");
33        System.out.println("Ora verrà' modificato solo a");
34
35        for (i=0; i<a.length; i++) //modifica a
36            a[i] += 15;
37
38        for (i=0; i<a.length; i++)
39            System.out.println("a[" + i + "] = " + a[i]);
40        System.out.println("");
41        for (i=0; i<b.length; i++)
42            System.out.println("b[" + i + "] = " + b[i]);
43
44    }
45 }
```

The screenshot shows the execution of the provided Java code. The console output is annotated with handwritten notes:

- Initial State:** The arrays a and b are initialized with different values: a[0]=12, a[1]=36, a[2]=222; b[0]=153, b[1]=23, b[2]=1.
- Assignment:** The line `b = a;` is annotated with a blue bracket under `b = a;` and a red bracket under `a[0]=12`, `a[1]=36`, `a[2]=222`. The output shows `E' stato eseguito l'assegnamento b=a`.
- First Modification:** The loop `for (i=0; i<a.length; i++) a[i] += 15;` is annotated with a green bracket under `a[0]=12`, `a[1]=36`, `a[2]=222`. The output shows the modified values: a[0]=27, a[1]=51, a[2]=237.
- Final State:** The final output shows both arrays having the same modified values: a[0]=27, a[1]=51, a[2]=237; b[0]=27, b[1]=51, b[2]=237. A green bracket is placed under the final values of array a.

6.2.5 Metodi che restituiscono array

Un metodo java può restituire un array. Per fare ciò bisogna specificare il tipo restituito dal metodo allo stesso modo con cui si specifica un parametro di tipo array. Ad esempio due modi per scrivere lo stesso programma li vediamo nei listati sottostanti. Nel caso RitornoDiArrayDemo i nuovi metodi restituiscono punteggi di un array. Per fare ciò, crea un nuovo array e lo restituisce con i seguenti passi

```

double[] temp = new double[punteggioSeguente.length];

for (int i = 0; i < temp.length; i++)
    temp[i] = getMedia(punteggioIniziale, punteggioSeguente[i]);

return temp;

```

```

1 import java.util.Scanner;
2
3 public class RitornoDiArrayDemo {
4
5     public static void main(String[] args) {
6
7         Scanner tastiera = new Scanner(System.in);
8
9         System.out.println("Inserire il voto dell'esame 1:");
10        int punteggioIniziale = tastiera.nextInt();
11
12        int[] punteggioSeguente = new int[3];
13
14        for (int i = 0; i < punteggioSeguente.length; i++)
15            punteggioSeguente[i] = punteggioIniziale + 5 * i;
16
17        double[] punteggioMedio = ottieniArrayDiMedie(punteggioIniziale,
18                                                       punteggioSeguente); //chiamata metodo 1
19
20        for (int i = 0; i < punteggioSeguente.length; i++) {
21            System.out.println("Se il voto all'esame 2 sara' " +
22                               punteggioSeguente[i]);
23            System.out.println("la media sara' uguale a " + punteggioMedio[i]);
24        }
25
26        public static double[] ottieniArrayDiMedie(int punteggioIniziale, int[] punteggioSeguente) { //metodo 1
27            double[] temp = new double[punteggioSeguente.length];
28
29            for (int i = 0; i < temp.length; i++)
30                temp[i] = getMedia(punteggioIniziale, punteggioSeguente[i]); //chiamata
31                                            metodo 2
32
33            return temp;
34        }
35
36        public static double getMedia(int n1, int n2) { //metodo 2
37            return (n1 + n2) / 2.0;
38        }
39    }
40
41    import java.util.Scanner;
42
43    /*Utilizzo di variabili indicizzate.*/
44    public class ArgomentiDemo {
45
46        public static void main(String[] args) { //main
47            Scanner tastiera = new Scanner(System.in);
48            System.out.println("Inserire il voto dell'esame 1:");
49            int punteggioIniziale = tastiera.nextInt();
50            int[] punteggioSeguente = new int[3];
51
52            for (int i = 0; i < punteggioSeguente.length; i++)
53                punteggioSeguente[i] = punteggioIniziale + 5 * i;
54
55            for (int i = 0; i < punteggioSeguente.length; i++) {
56                double possibileMedia = getMedia(punteggioIniziale, punteggioSeguente[i]);
57                System.out.println("Se il voto all'esame 2 sara' " +
58                               punteggioSeguente[i]);
59                System.out.println("la media sara' uguale a " + possibileMedia);
60            }
61
62            public static double getMedia(int n1, int n2) { //metodo
63                return (n1 + n2) / 2.0;
64            }
65        }
66    }

```

```

Inserire il voto dell'esame 1:
18
Se il voto all'esame 2 sara' 18
la media sara' uguale a 18.0
Se il voto all'esame 2 sara' 23
la media sara' uguale a 20.5
Se il voto all'esame 2 sara' 28
la media sara' uguale a 23.0

```

6.3 ARRAY MULTIDIMENSIONALI

Array che hanno esattamente due indici possono essere visualizzati su di un foglio come tabelle bidimensionali e vengono chiamati array bidimensionali. Per convenzione si attribuisce al primo indice la numerazione delle righe e al secondo delle colonne (x, y).

Gli array con più indici vengono generalmente chiamati array multidimensionali. In particolare un array con n indici viene detto array n-dimensionale.

6.3.1 Fondamenti sugli array multidimensionali

La dichiarazione rispetta quella degli array monodimensionali, ad esempio:

```
int[][] tabella = new int[10][6];
```

le variabili indicizzate sono identiche a quelle degli array monodimensionali, semplicemente devono essere indirizzate tramite due o più indici.

```
// tupla con x fisso
int[][] tabella = new int[3][];
tabella[0] = new int[2];
tabella[1] = new int[3];
tabella[2] = new int[5];
tabella[3] = new int [13];
```

6.3.2 Array irregolari

Nei precedenti esempi di array bidimensionali, tutte le righe avevano lo stesso numero di elementi. Se si volesse creare un array irregolare bisognerebbe definire prima l'array, poi definire ogni tupla alla volta.

7 RICORSIONE

7.1 LE BASI DELLA RICORSIONE

Accade spesso che il modo più naturale per progettare un algoritmo implichi l'applicazione dell'algoritmo stesso in uno o più casi particolari.

Quando una parte di un algoritmo costituisce una versione ridotta dell'algoritmo completo, quest'ultimo è definito "ricorsivo". Un algoritmo ricorsivo può essere implementato tramite un "metodo ricorsivo", cioè un metodo che contenga una chiamata a se stesso, detta "chiamata ricorsiva".

Prendiamo ad esempio questo codice per un conto alla rovescia, scritto tramite ricorsione:

```

1  public class ContoAllaRovesciaRicorsivo {
2      public static void main(String[] args) {
3
4          contoAllaRovescia();
5      }
6
7      public static void contoAllaRovescia(int num) {
8
9          if (num <= 0) {
10              System.out.println();
11          } else {
12              System.out.print(num);
13              contoAllaRovescia (num - 1);
14          }
15      }
16  }
17
18 }
```

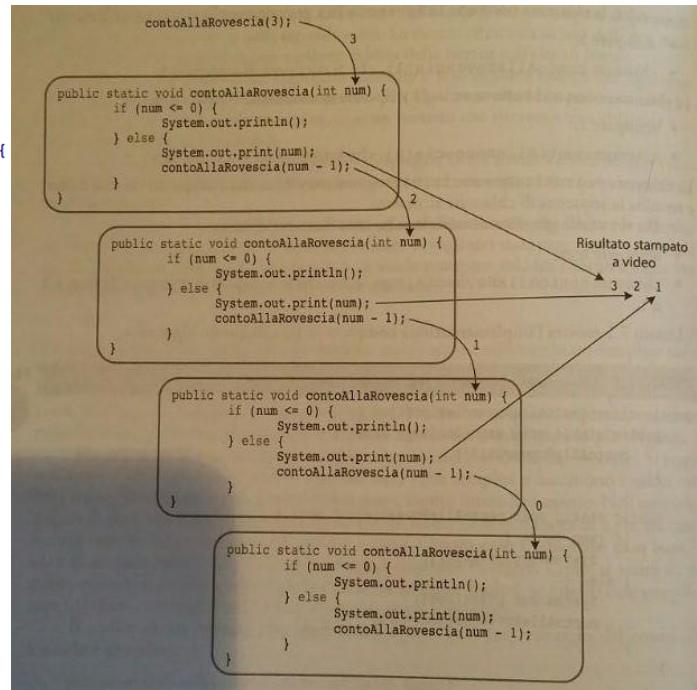


Figura 7.1 Chiamate ricorsive per il metodo `contoAllaRovescia`.

7.1.1 Come funziona la ricorsione

Quando viene eseguita una chiamata di un metodo ricorsivo non accade nulla di speciale: gli argomenti vengono copiati nei parametri e viene eseguito il codice contenuto nella definizione del metodo, così come accadrebbe per la chiamata di qualunque metodo.

Quando si incontra una chiamata ricorsiva, l'elaborazione viene temporaneamente sospesa, dato che per procedere è necessario conoscere il risultato della chiamata ricorsiva. Vengono salvate tutte le informazioni necessarie per poter riprendere l'esecuzione in seguito e viene eseguita la chiamata ricorsiva. Una volta completata quest'ultima viene ripresa l'elaborazione più esterna (vedi cap. 5.3).

Il linguaggio Java non pone vincoli all'utilizzo di chiamate ricorsive nella definizione di un metodo. Tuttavia, affinché tale definizione sia utile, **deve essere progettata in modo che qualunque chiamata al metodo si concluda necessariamente con una qualche porzione di codice che non dipenda dalla ricorsione**.

La struttura generale per una corretta definizione di un metodo ricorsivo è la seguente:

- Uno o più casi nei quali il metodo esegue il compito previsto tramite una o più chiamate ricorsive al fine di risolvere una o più versioni ridotte del problema originale

- Uno o più casi nei quali il metodo esegue il compito previsto senza ricorrere ad alcuna chiamata ricorsiva. Questi casi, privi di chiamate ricorsive, sono chiamati “casi base” o “di arresto”

Spesso si usa un blocco `if-else` per determinare quale caso debba essere eseguito.

Ogni chiamata al metodo deve portare alla fine a un caso di arresto, altrimenti non terminerà mai, a causa di una sequenza infinita di chiamate ricorsive.

7.1.2 *Lo stack e la ricorsione*

Le chiamate ricorsive sono chiamate a metodi. Quindi, come specificato nel Capitolo 5, l’invocazione di un metodo comporta la creazione di un nuovo record di attivazione e il suo posizionamento in cima allo stack.

Grazie allo stack, il computer può tenere traccia facilmente delle chiamate ricorsive.

C’è sempre un limite alle dimensioni dello stack. Se si verifica una lunga catena di chiamate ricorsive di un metodo, ogni chiamata ricorsiva produrrà il salvataggio sullo stack di un’altra elaborazione sospesa. **Se questa sequenza è troppo lunga, lo stack cercherà di estendersi oltre i limiti. Questa condizione di errore è detta stack overflow.**

7.1.3 *Confronto tra metodi ricorsivi e iterativi*

Qualunque definizione di un metodo che includa una chiamata ricorsiva può essere riscritta in modo da svolgere lo stesso compito senza l’uso della ricorsione. Un processo ripetitivo e non ricorsivo è definito “iterazione” (ad esempio il costrutto `for`), e un metodo che implementi un processo di questo tipo è detto metodo iterativo.

Un metodo ricorsivo utilizza più memoria rispetto a una versione iterativa, a causa del carico aggiuntivo sul sistema che deriva dalla necessità di tenere traccia delle chiamate ricorsive e delle elaborazioni rimaste in sospeso. A causa di questo carico aggiuntivo, l’esecuzione di un metodo ricorsivo può essere più lenta di quella del corrispondente metodo iterativo. **In alcuni casi, l’incremento del tempo di esecuzione per un particolare metodo ricorsivo è tale da spingere a evitare la ricorsione.**

7.1.4 *Metodi ricorsivi che restituiscono un valore*

Un metodo ricorsivo può essere un metodo `void`, come visto in precedenza, o può restituire un valore. Le modalità di progettazione di un metodo che restituisce un valore sono essenzialmente valide per i metodi `void` e pertanto rimangono valide le stesse linee guida.

Almeno una delle alternative dovrebbe contenere una chiamata ricorsiva del metodo che produce il valore da restituire. Queste chiamate ricorsive devono utilizzare argomenti in qualche modo “più piccoli” o risolvere versioni “ridotte” del compito realizzato dal metodo.

8 DEFINIRE CLASSI E CREARE OGGETTI

8.1 DEFINIZIONE DI CLASSI

Gli oggetti di un programma possono rappresentare oggetti del mondo reale o astrazioni. Una classe è la definizione di un tipo di oggetto, è come uno stampo per la costruzione di oggetti di un certo tipo.

Ciascun oggetto definito dalla classe è detto istanza. Si possono creare, o meglio istanziare, più oggetti della stessa classe. Ad esempio:

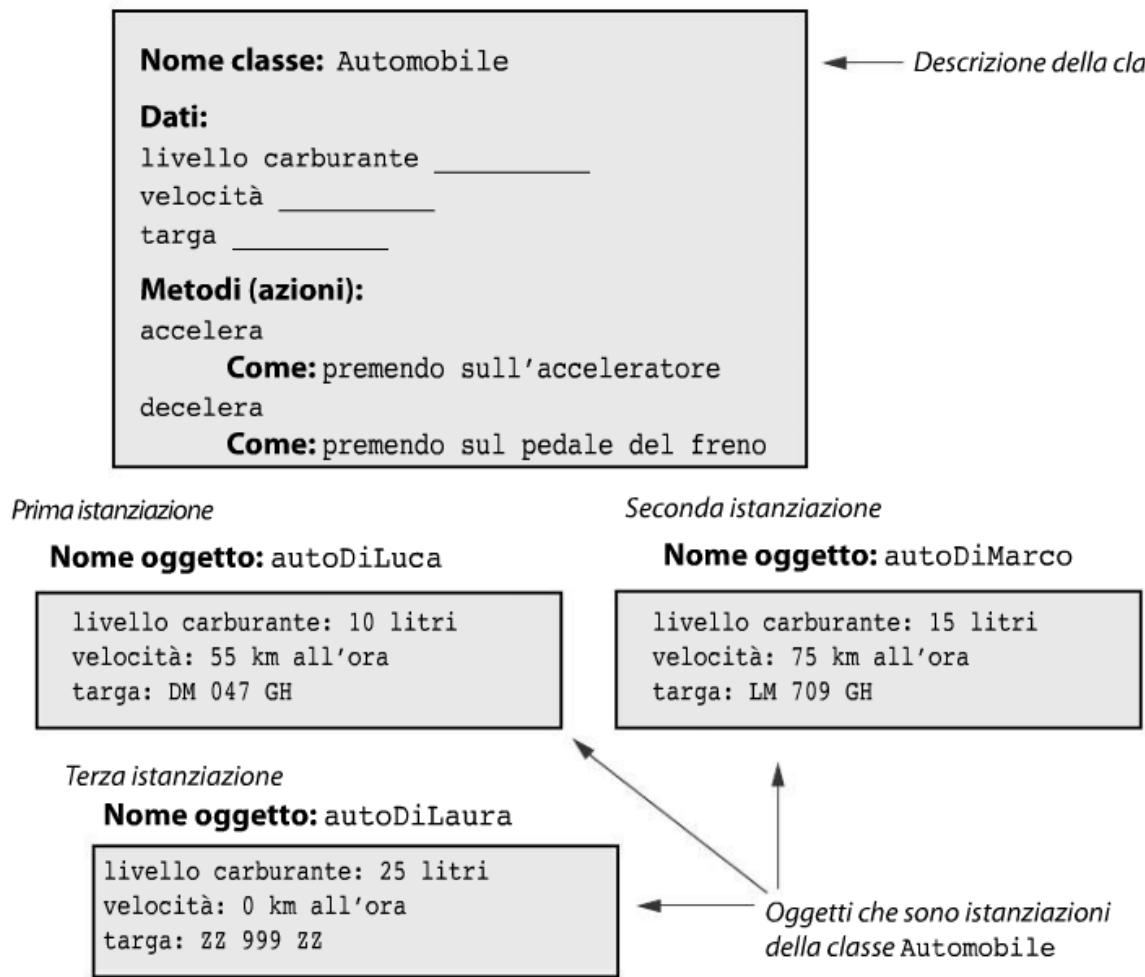


Figura 8.1 Una classe come *blueprint*.

La classe della figura definisce i dati base e le azioni degli oggetti appartenenti alla classe “Automobile”. Sotto abbiamo tre istanze (oggetti) della classe.

Quindi una classe specifica gli attributi degli oggetti della classe. La definizione di una classe non specifica il valore degli attributi poiché questi sono specifici dei singoli oggetti, la classe specifica solo il tipo di dato di questi attributi. Una classe specifica anche le azioni che possono essere svolte dagli oggetti.

La notazione vista prima, però, non è molto comoda in quanto esteticamente troppo ricca di dettagli, per questo gli sviluppatori usano una notazione più sintetica, detta “diagramma delle classi UML (Universal Modeling Language)”.

La classe vista sopra può essere riscritta così:

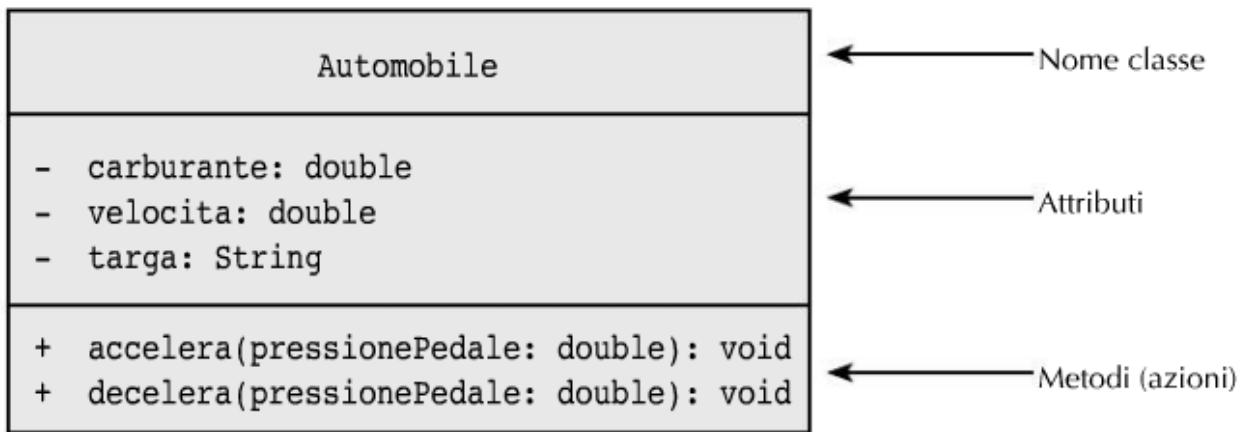


Figura 8.2 La classe rappresentata sotto forma di diagramma delle classi UML.

8.1.1 File delle classi e compilazione

È necessario salvare ciascuna definizione di classe in un file distinto se non per alcune rare eccezioni.

È possibile compilare una classe Java prima di avere un programma che la utilizzi.

8.1.2 Variabili di istanza

```

1  public class Cane {
2
3      public String nome;
4      public String razza;
5      public int anni;
6
7      public void scriviOutput() {
8          System.out.println("Nome: " + nome);
9          System.out.println("Razza: " + razza);
10         System.out.println("Eta': " + anni);
11     }
12
13     public int getEtaInAnniUmani() {
14         int etaUmana = 0;
15         if (anni <= 2) {
16
17             } else {
18                 etaUmana = 22 + ((anni-2) * 5);
19             }
20         return etaUmana;
21     }
22 }
  
```

public, però, non è consigliabile.

Questa classe è stata progettata per mantenere alcune informazioni relative ai cani. Ciascun oggetto di questa classe contiene tre dati: nome, razza, età. Gli oggetti presentano due comportamenti definiti come `scriviOutput` e `getEtaInAnniUmani`. Sia i dati che i metodi vengono chiamati membri dell'oggetto. **Questo testo chiamerà i dati con il termine “variabili d’istanza” mentre i metodi come “metodi d’istanza”.**

Nella definizione delle variabili la parola chiave `public` indica che non ci sono restrizioni su come e dove queste variabili d’istanza possano essere usate. L’utilizzo di

Si può pensare a un oggetto di una classe come a un elemento complesso contenente al proprio interno una serie di variabili di istanza, ogni istanza possiede una propria copia di queste variabili.

```

1  public class CaneDemo {
2      public static void main(String[] args) {
3
4          Cane balto = new Cane();
5          balto.nome = "Balto";
6          balto.anni = 8;
7          balto.raffa = "Husky Siberiano";
8
9          balto.scriviOutput(); //metodo d'istanza
10
11         Cane scooby = new Cane();
12         scooby.nome = "Scooby";
13         scooby.anni = 9;
14         scooby.raffa = "Alano";
15
16         System.out.println(scooby.nome + " è un " + scooby.raffa + ".");
17         System.out.print("Ha " + scooby.anni + " anni, oppure ");
18
19         int anniUmani = scooby.getEtaInAnniUmani();
20
21         System.out.println(anniumani + " in anni umani.");
22     }
  
```

La riga `Cane balto = new Cane();` crea un oggetto di tipo cane e associa a questo oggetto la variabile `balto`.

Si può fare riferimento a una di esse scrivendo il nome dell’oggetto seguito da un punto e quindi il nome della variabile d’istanza, ad esempio `balto.nome` denota la variabile d’istanza `nome` appartenente all’oggetto

balto. Essendo una variabile `String` può essere usata come una variabile `string` qualsiasi.

8.1.3 Metodi di istanza

Un metodo è un insieme di istruzioni con un nome la cui invocazione comporta l'esecuzione delle istruzioni in esso definite. Un metodo d'istanza è un metodo che viene invocato su un oggetto e che può manipolare lo stato dell'oggetto stesso.

Un metodo d'istanza definito in una classe viene invocato usando un oggetto di quella classe che prende il nome di "oggetto chiamante" o "oggetto ricevente". L'invocazione viene effettuata scrivendo il nome dell'oggetto seguito da un punto e dal nome del metodo, ad esempio `balto.scriviOutput()` ;

```
public void scriviOutput() {
    System.out.println("Nome: " + nome);
    System.out.println("Razza: " + razza);
    System.out.println("Eta': " + anni);
}
```

si noti che nell'intestazione del metodo manca il modificatore `static`, infatti se non scritto vuol dire che il metodo è un metodo d'istanza.

La differenza tra i metodi classici e quelli d'istanza risiede nel corpo del metodo. Infatti i secondi fanno riferimento a variabili contenute nell'oggetto di riferimento. Infatti quando viene invocato un metodo con la linea, ad esempio, `balto.scriviOutput()` il metodo farà riferimento alle variabili contenute all'interno dell'oggetto `balto`. Questi metodi possono essere utilizzati esclusivamente oggetti della classe dentro alla quale vengono definiti.

8.1.4 La parola chiave `this`

```
1 import java.util.Scanner;
2
3 public class SpeciePrimaProva {
4
5     public String nome;
6     public int popolazione;
7     public double tassoCrescita;
8
9     public void leggiInput() { //metodo 1
10        Scanner tastiera = new Scanner(System.in);
11
12        System.out.println("Qual'e' il nome della specie?");
13        nome = tastiera.nextLine();
14        System.out.println("A quanto ammonta la popolazione?");
15        popolazione = tastiera.nextInt();
16        System.out.println("Inserisci il tasso di crescita " +
17                           "(% crescita per anno):");
18        tassoCrescita = tastiera.nextDouble();
19    }
20
21    public void scriviOutput() { //metodo 2
22        System.out.println("Nome = " + nome);
23        System.out.println("Popolazione = " + popolazione);
24        System.out.println("Tasso crescita = " + tassoCrescita + "%");
25    }
26
27    public int prediciPopolazione(int anni) { //metodo 3
28        int risultato = 0;
29        double totalePopolazione = popolazione;
30        int contatore = anni;
31
32        while ((contatore > 0) && (totalePopolazione > 0)) {
33            totalePopolazione = (totalePopolazione +
34                                  (tassoCrescita / 100) * totalePopolazione);
35            contatore--;
36        }
37
38        if (totalePopolazione > 0)
39            risultato = (int) totalePopolazione;
40        return risultato;
41    }
42 }
```

che risiede nella stessa classe della variabile d'istanza, si utilizza semplicemente il nome della variabile.

Ogni variabile d'istanza appartiene ad un oggetto. Nei casi come questo si sottintende l'oggetto omettendone il nome. Il nome sottinteso di questo oggetto è `this`. Sebbene spesso sia omesso, si può inserire, per esempio `this.nome = tastiera.nextLine();`.

La parola chiave `this` è come uno spazio vuoto che aspetta di essere riempito con il nome dell'oggetto che riceve l'invocazione del metodo. In alcuni casi è necessario utilizzarlo.

```
1 public class SpeciePrimaProvaDemo {
2
3     public static void main(String[] args) {
4
5         SpeciePrimaProva specieDelMese = new SpeciePrimaProva();
6
7         System.out.println("Inserisci i dati della specie del mese:");
8         specieDelMese.leggiInput();
9         specieDelMese.scriviOutput();
10
11         int popolazioneFutura = specieDelMese.prediciPopolazione(10);
12
13         System.out.println("Tra dieci anni la popolazione sara' di " +
14                           popolazioneFutura + " individui.");
15
16         //Cambia la specie per verificare come
17         //si modificano i valori delle variabili di istanza:
18         specieDelMese.nome = "Panthera tigris tigris";
19         specieDelMese.popolazione = 3750;
20         specieDelMese.tassoCrescita = 30;
21
22         System.out.println("La nuova specie del mese:");
23         specieDelMese.scriviOutput();
24         System.out.println("Tra dieci anni la popolazione sara' di " +
25                           specieDelMese.prediciPopolazione(10) + " individui.");
26     }
27 }
```

Si noti che le variabili di istanza sono scritte in maniera diversa, a seconda che siano all'interno della definizione della classe o all'esterno. Al di fuori della classe, il nome delle variabili d'istanza è composto dal nome dell'oggetto della classe, seguito da un punto e dal nome della variabile d'istanza. Al contrario all'interno della definizione di un metodo

8.2 INFORMATION HIDING E INCAPSULAMENTO

In un programma, la possibilità di nascondere alcune informazioni viene vista come una qualità del linguaggio che semplifica il lavoro dei programmatore e rende più comprensibile il codice sviluppato.

8.2.1 *Information hiding*

Un programmatore che usa un metodo definito da un altro programmatore non ha bisogno di conoscerne i dettagli, ad esempio l'utilizzo del metodo `nextInt` della classe `Scanner` non presuppone la conoscenza di tutti i procedimenti necessari.

L'information hiding è una pratica che consente di progettare un metodo in modo che possa essere usato senza alcun bisogno di comprendere i dettagli del codice che lo implementa. Al posto del termine information hiding si può utilizzare il termine "astrazione".

8.2.2 *Commenti con precondizioni e postcondizioni*

Un modo efficiente e standard per descrivere le finalità di un metodo consiste nell'usare alcuni commenti specifici, noti come precondizioni e postcondizioni.

- **Precondizione:** descrive le condizioni che devono sussistere prima che il metodo sia invocato
- **Postcondizione:** descrive tutti gli effetti prodotti dall'invocazione del metodo

Se l'unica postcondizione è una descrizione del valore restituito, il programmatore omette il termine postcondizione.

Commento completo	Commento con omissione
<pre>/* Precondizione: anni è un numero non negativo Postcondizione: restituisce la predizione della popolazione dell'oggetto invocante dopo il numero di anni specificato */</pre>	<pre>/* Precondizione: anni è un numero non negaticvo Restituisce la predizione della popolazione dell'oggetto invocante dopo il numero di anni specificato */</pre>

Si noti come i commenti di precondizione e postcondizione siano esempi di asserzioni.

8.2.3 *I modificatori d'accesso public e private*

Come affermato in precedenza, il modificatore `public`, quando applicato a una classe, a un metodo o a una variabile d'istanza, indica che qualsiasi altra classe può usare direttamente quella classe, metodo o variabile di istanza attraverso il suo nome.

Sebbene sia normale avere metodi e classi pubblici, non è invece considerata una buona abitudine rendere pubbliche le variabili d'istanza di una classe. **Tipicamente, tutte le variabili d'istanza dovrebbero essere private.**

Le parole chiave `public` e `private` sono dette "modificatori d'accesso" o "modificatori di visibilità".

Quando una variabile d'istanza è privata, il suo nome non è accessibile al di fuori della definizione della sua classe. La variabile può però essere usata all'interno di uno qualsiasi dei metodi definiti nella sua stessa classe.

Nel caso non venga impostato un modificatore d'accesso, questo viene impostato al valore di default.

8.2.4 Metodi get e set

```

1 import java.util.Scanner;
2
3
4 public class SpecieTerzaProvaDemo {
5
6     public static void main(String[] args) {
7
8         SpecieTerzaProva specieDelMese = new SpecieTerzaProva();
9
10        System.out.println("Inserisci i dati sulla specie del mese:");
11        specieDelMese.leggiInput();
12        specieDelMese.scriviOutput();
13
14        System.out.println("Inserisci il numero di anni da predire:");
15        Scanner tastiera = new Scanner(System.in);
16
17        int numeroAnni = tastiera.nextInt();
18
19        int popolazioneFutura = specieDelMese.prediciPopolazione(numeroAnni);
20
21        System.out.println("Tra " + numeroAnni + " anni la popolazione sara' di " +
22                           popolazioneFutura + " individui.");
23
24        //Cambia la specie per verificare come
25        //si modificano i valori delle variabili di istanza:
26        specieDelMese.setSpecie("Panthera tigris tigris", 100, 10);
27        System.out.println("La nuova specie del mese:");
28        specieDelMese.scriviOutput();
29        popolazioneFutura = specieDelMese.prediciPopolazione(numeroAnni);
30        System.out.println("Tra " + numeroAnni + " anni la popolazione sara' di " +
31                           popolazioneFutura + " individui.");
32    }
33
34    import java.util.Scanner;
35
36    public class SpecieTerzaProva {
37
38        private String nome;
39        private int popolazione;
40        private double tassoCrescita;
41
42        public void leggiInput() {
43            Scanner keyboard = new Scanner(System.in);
44            System.out.println("Qual'e' il nome della specie?");
45            nome = keyboard.nextLine();
46            System.out.println("A quanto ammonta la popolazione?");
47            popolazione = keyboard.nextInt();
48            System.out.println("Inserisci il tasso di crescita (% crescita per anno):");
49            tassoCrescita = keyboard.nextDouble();
50        }
51
52        public void scriviOutput() {
53            System.out.println("Nome = " + nome);
54            System.out.println("Popolazione = " + popolazione);
55            System.out.println("Tasso crescita = " + tassoCrescita + "%");
56        }
57
58        /**
59         * Restituisce una prolezione della popolazione dopo un numero specificato
60         * di anni
61         */
62        public int prediciPopolazione(int anni) {
63
64            int risultato = 0;
65            double totalePopolazione = popolazione;
66            int conteggio = anni;
67
68            while ((conteggio > 0) && (totalePopolazione > 0)) {
69                totalePopolazione = (totalePopolazione + (tassoCrescita / 100) *
70                                     totalePopolazione);
71                conteggio--;
72            }
73            if (totalePopolazione > 0)
74                risultato = (int) totalePopolazione;
75            return risultato;
76        }
77
78        public void setSpecie(String nuovoNome, int nuovaPopolazione, double
79                               nuovoTassoCrescita) { //metodo set
80
81            nome = nuovoNome;
82            if (nuovaPopolazione >= 0)
83                popolazione = nuovaPopolazione;
84            else {
85                System.out.println("ERRORE: si sta usando un numero negativo per la
86                                   popolazione.");
87                System.exit(0);
88            }
89            tassoCrescita = nuovoTassoCrescita;
90        }
91
92        public String getNome() { //metodo get
93            return nome;
94        }
95
96        public int getPopolazione() { //metodo get
97            return popolazione;
98        }
99
100       public double getTassoCrescita() { //metodo get
101           return tassoCrescita;
102       }
103   }
104 }
```

Rendere private tutte le variabili di una classe permette di avere un controllo completo su di esse però vengono anche rese inaccessibili dall'esterno. In questo caso devono essere creati dei metodi di accesso (**metodi get**). Questi metodi permettono di osservare i dati contenuti in una variabile d'istanza. Per convenzione il nome di questi metodi inizia con “get” (ad esempio `getNome`).

I metodi d'accesso permettono di osservare i dati contenuti, per modificarli bisogna utilizzare i metodi di modifica (**metodi set**).

Definire metodi `set` e `get` sembra annullare lo scopo del modificatore d'accesso `private` ma non è così. Un metodo `set` può verificare se un cambiamento è appropriato e notificare l'utente in caso di problemi. Ad esempio `setSpecie` controlla se il programma sta assegnando un valore negativo a `popolazione`.

8.2.5 La parola chiave `this` applicata alle variabili d'istanza

Si è affermato che all'interno di un blocco non possono coesistere due variabili che hanno lo stesso nome.

C'è un'eccezione: se una delle due variabili è una variabile d'istanza, è possibile avere due nomi uguali. Se vogliamo infatti nominare la variabile d'istanza e la variabile locale del metodo con lo stesso nome, possiamo farlo aggiungendo la parola `this` prima della variabile d'istanza. In questo modo si sta segnalando che quella variabile è quella riferita all'oggetto che chiama il metodo.

8.2.6 Metodi che invocano altri metodi

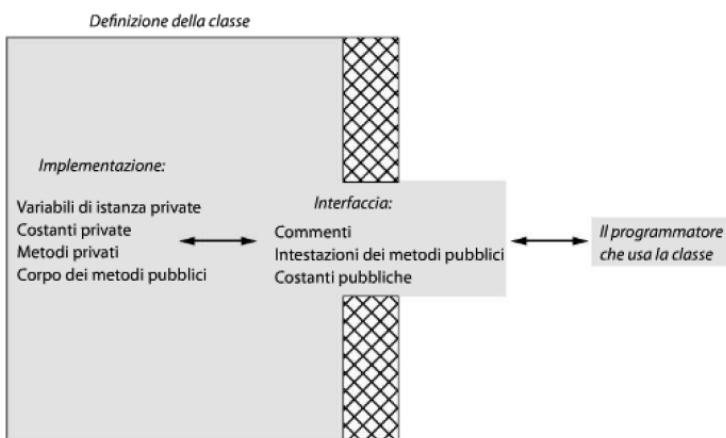
Se il metodo invocato si trova nella stessa classe, l'invocazione viene effettuata senza scrivere il nome dell'oggetto.

In più i metodi all'interno di una classe possono invocare un metodo privato ma questo non è accessibile direttamente, quindi i metodi invocati in seconda battuta vengono nominati “metodi ausiliari”, accessibili solo dal metodo pubblico contenuto nella stessa classe.

8.2.7 Incapsulamento

L'incapsulamento consiste nel nascondere tutti i dettagli della definizione di una classe che non sono necessari per usare le istanze create da quella classe.

Affinché l'incapsulamento sia utile, la definizione di una classe deve essere tale per cui un programmatore possa usarla senza conoscerne i dettagli.



L'interfaccia di una classe indica ai programmatore ciò di cui hanno bisogno per usare la classe nei loro programmi. **L'interfaccia della classe consiste nell'intestazione dei suoi metodi pubblici e delle sue costanti pubbliche.**

L'implementazione di una classe consiste di tutti gli elementi privati della classe, principalmente le variabili d'istanza private e le definizioni dei metodi pubblici e privati.

Quando si usa l'incapsulamento per definire una classe in questo modo, si dice che la classe è “ben encapsulata”. Per definire una classe ben encapsulata:

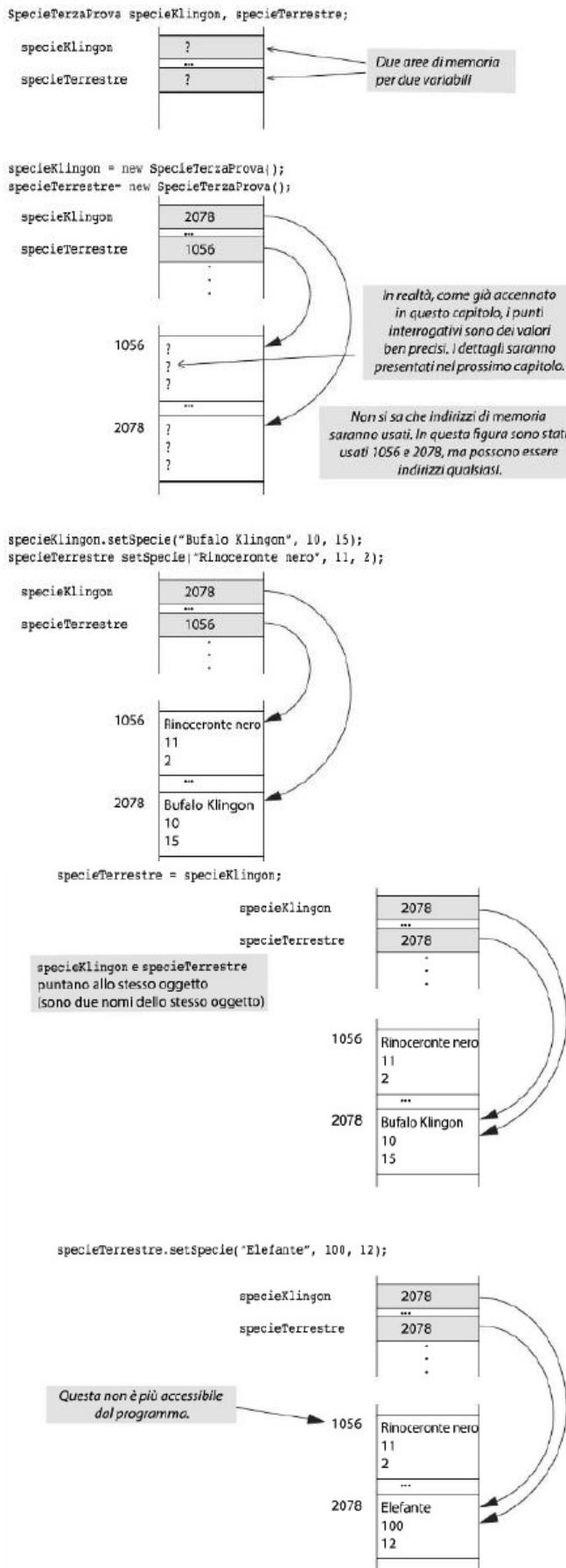
- Si predisponga un commento prima della definizione della classe che descriva al programmatore cosa rappresenta la classe senza descrivere come lo fa
- Si dichiarino tutte le variabili di istanza della classe come private
- Si forniscano metodi get pubblici per recuperare i dati di un oggetto. Si forniscano metodi pubblici per qualsiasi altra necessità, tra i quali anche i metodi set
- Si predisponga un commento prima di ogni intestazione di metodo pubblico che specifichi chiaramente come usare il metodo
- Si rendano privati i metodi ausiliari
- Si scrivano commenti all'interno della classe per descrivere i dettagli implementativi

Buona norma è scrivere i commenti usando `/** ... */` per i commenti delle interfacce della classe e `//...` per i commenti sull'implementazione.

8.2.8 Documentazione automatica con javadoc

Se si commenta correttamente una classe, javadoc prenderà in input i commenti e produrrà un documento ben formattato che potrà essere letto comodamente per comprendere l'interfaccia delle classi.

8.3 OGGETTI E RIFERIMENTI



8.3.1 Variabili di tipo classe

Le variabili di tipo classe forniscono un nome agli oggetti: vengono loro assegnati degli oggetti, ma il processo è diverso rispetto all'assegnamento di valori di tipo primitivo.

Un oggetto di tipo primitivo è contenuto all'interno di una cella di memoria dello stack.

Un oggetto di tipo classe contiene, invece, l'indirizzo che punta all'area di memoria dell'heap dove sono conservati i vari dati delle variabili. Questa differenza deriva dal fatto che una variabile primitiva ha un numero fisso di bit richiesti per la sua rappresentazione mentre una variabile di tipo classe no (per esempio una stringa non ha una lunghezza predefinita, infatti è un oggetto di tipo classe).

Quindi l'assegnamento (prendendo dall'esempio a sinistra) `specieTerrestre = specieKlingon` in realtà assegna l'**indirizzo della seconda alla prima e non i suoi valori** essendo queste variabili di classe. In questo caso assumono la stessa identità.

8.3.2 Definire un metodo `equals` per una classe

Usare `==` tra oggetti di tipo classe controlla se puntano allo stesso indirizzo di memoria, non se hanno gli stessi valori.

Per verificare se due oggetti sono uguali secondo una propria concezione di uguaglianza, **si deve definire un metodo `equals`.**

Non esiste una definizione unica di `equals` che possa essere usata per tutti gli oggetti.

8.3.3 Test di unità

Un approccio alla soluzione di questo problema consiste nella scrittura di test di unità. Secondo questa metodologia il programmatore verifica la correttezza delle singole unità di codice. Una collezione di test di unità è detta suite di test. Questi test sono automatizzati, in questo modo possono essere ripetuti frequentemente. Il procedimento consistente nell'esecuzione ripetuta dei test è detto test di regressione.

Riassunto di Jacopo De Angelis

Un tipo di test che si potrebbe considerare è il cosiddetto test negativo. Si tratta di un tipo di test volto a verificare che il programma non termini in modo inatteso se gli vengono forniti dati di input con caratteristiche diverse da quelle attese.

Non bisogna commettere l'errore di pensare che il passare tutti i test voglia dire che il programma è corretto. Potrebbe esserci un problema per una combinazione specifica di valori.

9 APPROFONDIMENTI SU CLASSI, OGGETTI E METODI

LISTATO 9.1 La classe Animale: un esempio sui costruttori e sui metodi set.

```


/*
 * Classe che descrive un animale
 */
public class Animale {

    private String nome;
    private int eta; //in anni
    private double peso; //in Kg

    public Animale() { ← Costruttore di default.
        nome = "Nessun nome";
        eta = 0;
        peso = 0;
    }

    public Animale(String nomeIniziale, int etaIniziale, double pesoIniziale) {
        nome = nomeIniziale;
        if ((etaIniziale < 0) || (pesoIniziale < 0)) {
            System.out.println("Errore: eta' o peso negativi.");
            System.exit(0);
        } else {
            eta = etaIniziale;
            peso = pesoIniziale;
        }
    }

    public void setAnimale(String nuovoNome, int nuovaEta, double nuovoPeso) {
        nome = nuovoNome;
        if ((nuovaEta < 0) || (nuovoPeso < 0)) {
            System.out.println("Errore: eta' o peso negativi.");
            System.exit(0);
        } else {
            eta = nuovaEta;
            peso = nuovoPeso;
        }
    }

    public Animale(String nomeIniziale) {
        nome = nomeIniziale;
        eta = 0;
        peso = 0;
    }

    public void setNome(String nuovoNome) {
        nome = nuovoNome; //eta' e peso rimangono invariate
    }
}


```

```


public Animale(int etaIniziale) {
    nome = "Nessun nome";
    peso = 0;
    if (etaIniziale < 0) {
        System.out.println("Errore: eta' negativa.");
        System.exit(0);
    } else {
        eta = etaIniziale;
    }
}

public void setEta(int nuovaEta) {
    if (nuovaEta < 0) {
        System.out.println("Errore: eta' negativa.");
        System.exit(0);
    } else {
        eta = nuovaEta;
        //nome e peso rimangono invariate
    }
}

public Animale(double pesoIniziale) {
    nome = "Nessun nome";
    eta = 0;
    if (pesoIniziale < 0) {
        System.out.println("Errore: peso negativo.");
        System.exit(0);
    } else {
        peso = pesoIniziale;
    }
}

public void setPeso(double nuovoPeso) {
    if (nuovoPeso < 0) {
        System.out.println("Errore: peso negativo.");
        System.exit(0);
    } else {
        peso = nuovoPeso;
        //nome e eta' rimangono invariate
    }
}

public String getNome() {
    return nome;
}

public int getEta() {
    return eta;
}

public double getPeso() {
    return peso;
}

public void scriviOutput(){
    System.out.println("Nome: " + nome);
    System.out.println("Eta: " + eta + " anni");
    System.out.println("Peso: " + peso + " Kg");
}


```

9.1 COSTRUTTORI

Quando si crea un oggetto di una classe utilizzando l'operatore new, si invoca un particolare metodo chiamato “costruttore”.

9.1.1 Definire i costruttori

Un costruttore è un particolare metodo che viene invocato quando si utilizza l'operatore new per creare un nuovo oggetto.

Nelle classi viste finora, i costruttori creano gli oggetti e forniscono alle loro variabili di istanza un valore iniziale di default. Questi valori potrebbero non essere quelli desiderati.

I costruttori hanno essenzialmente lo stesso compito dei metodi set ma, a differenza di questi, i costruttori creano un oggetto oltre a inizializzarlo.

Ogni costruttore ha lo stesso nome della sua classe (ad esempio se la classe si chiama Animale, il costruttore si chiamerà Animale).

I metodi costruttori non sono metodi void. Nel caso non vengano assegnati valori a tutte le variabili, quelle non inizializzate avranno valori di default.

I costruttori hanno spesso più definizioni, ognuna delle quali presenta un differente numero di parametri o differenti tipi di parametri e, a volte, assomigliano ai metodi set della classe. I costruttori, però, non contengono la parola `void` o un tipo di ritorno. A differenza di alcuni metodi set, i costruttori forniscono un valore a tutte le variabili d'istanza, anche se non hanno un parametro per ognuna di esse.

Il listato sopra include un costruttore chiamato animale senza parametri. Questo costruttore è chiamato “costruttore di default”, ovvero quello integrato automaticamente da Java nel caso non vengano creati altri costruttori. Per invocare un costruttore è necessario usare `new` e poi il nome del costruttore con i parametri richiesti (ad esempio `Animale pesce = new Animale("Bavosetta", 2, 0.11)`). È buona norma inserire anche il costruttore di default manualmente.

Generalmente i costruttori vengono scritti prima di ogni altro metodo.

9.1.2 *Invocare metodi da costruttori*

I costruttori possono invocare altri metodi appartenenti alla stessa classe ma bisogna stare attenti. Il problema ha a che fare con l'ereditarietà, ovvero un'altra classe può alterare il comportamento dei metodi pubblici e, di conseguenza, può alterare di conseguenza il comportamento di un costruttore.

9.1.3 Invocare un costruttore da un altro costruttore

```

1  public class Animale3 {
2      private String nome;
3      private int eta; //in anni
4      private double peso; //in Kg
5
6      public Animale3(String nomeIniziale, int etainiziale, double pesoIniziale) {
7          set(nomeIniziale, etainiziale, pesoIniziale);
8      }
9
10     public Animale3(String nomeIniziale) {
11         this(nomeIniziale, 0, 0);
12     }
13
14     public Animale3(int etainiziale) {
15         this("Nessun nome", etainiziale, 0);
16     }
17
18     public Animale3(double pesoIniziale) {
19         this("Nessun nome", 0, pesoIniziale);
20     }
21
22     public Animale3() {
23         this("Nessun nome", 0, 0);
24     }
25
26     public void setAnimale(String nomeIniziale, int etainiziale, double
27     pesoIniziale) {
28         set(nomeIniziale, etainiziale, pesoIniziale);
29     }
30
31     public void setNome(String nuovoNome) {
32         nome = nuovoNome;
33         //eta' e peso rimangono invariate
34     }
35
36     public void setEta(int nuovaEta) {
37         set(nome, nuovaEta, peso);
38         //nome e peso rimangono invariate
39     }
40
41     public void setPeso(double nuovoPeso) {
42         set(nome, eta, nuovoPeso);
43         //nome e eta' rimangono invariate
44     }
45
46     private void set(String nuovoNome, int nuovaEta, double nuovoPeso) {
47         nome = nuovoNome;
48         if ((nuovaEta < 0) || (nuovoPeso < 0)) {
49             System.out.println("Errore: eta o peso negativi.");
50             System.exit(0);
51         } else {
52             eta = nuovaEta;
53             peso = nuovoPeso;
54         }
55     }
56
57     public String getNome() {
58         return nome;
59     }
60
61     public int getEta() {
62         return eta;
63     }
64
65     public double getPeso() {
66         return peso;
67     }
68
69     public void scriviOutput(){
70         System.out.println("Nome: " + nome);
71         System.out.println("Eta: " + eta + " anni.");
72         System.out.println("Peso: " + peso + " Kg.");
73     }
74 }
```

I costruttori possono invocarsi tra di loro.
L'invocazione deve essere la prima azione eseguita all'interno del costruttore. Si noti che i metodi set non hanno l'istruzione `this`. **L'utilizzo della parola chiave `this` per invocare un costruttore è consentito, infatti, solo all'interno di un altro costruttore della stessa classe** (avendo lo stesso nome).

9.1.4 La costante null

La costante `null` è una speciale costante che più può essere assegnata a una variabile di qualunque tipo classe. È usata per indicare che la variabile non ha alcun “valore reale”. Se il compilatore richiede che una variabile di tipo classe venga inizializzata e non sono disponibili oggetti per inizializzarla, si può utilizzare il valore `null` (ad esempio `MiaClasse mioOggetto = null`).

9.2 VARIABILI STATICHE E METODI STATICI

Le variabili e i metodi statici appartengono all'intera classe e non a un singolo oggetto.

9.2.1 Variabili statiche

Le variabili statiche (o variabili di classe) presentano come modificatore static.

Il termine variabile statica è utilizzato per non creare confusione tra variabile di classe e variabile di tipo classe.

9.2.2 Metodi statici

Quando si definisce un metodo statico, il metodo è ancora membro di una classe, poiché è definito all'interno di una classe, ma può essere invocato senza usare alcun oggetto. Normalmente si invoca un metodo statico utilizzando il nome della classe anziché quello di un oggetto.

Ad esempio se si volesse calcolare il massimo tra due interi, convertire una lettera maiuscola in minuscola, non si avrebbe un oggetto specifico a cui riferirsi.

```

4  public class ConvertitoreDimensioni {
5      public static final int POLLICI_PER_PIEDE = 12;
6
7      public static double convertiPiediInPollici(double piedi) {
8          return piedi * POLLICI_PER_PIEDE;
9      }
10
11     public static double convertiPolliciInPiedi(double pollici) {
12         return pollici / POLLICI_PER_PIEDE;
13     }
14 }
15 import java.util.Scanner;
16 /**
17  * Dimostrazione di utilizzo della classe ConvertitoreDimensioni
18 */
19 public class ConvertitoreDimensioniDemo {
20
21     public static void main(String[] args) {
22         Scanner tastiera = new Scanner(System.in);
23         System.out.print("Inserisci una misura in pollici: ");
24         double pollici = tastiera.nextDouble();
25
26         double piedi = ConvertitoreDimensioni.convertiPolliciInPiedi(pollici);
27         System.out.println(pollici + " pollici = " + piedi + " piedi.");
28
29         System.out.print("Inserisci una misura in piedi: ");
30         piedi = tastiera.nextDouble();
31         pollici = ConvertitoreDimensioni.convertiPiediInPollici(piedi);
32         System.out.println(piedi + " piedi = " + pollici + " pollici.");
33     }
34 }
```

Un esempio più complesso è invece quello del seguente listato riguardante i conti di una banca.

Tutti i conti hanno il medesimo tasso di interesse, perciò la classe ha una variabile statica chiamata `tassolInteresse`. Inoltre tiene traccia del numero di conti aperti, utilizzando la variabile statica `numeroDiConti`.

```

1  /**
2   * Una classe con variabili di istanza e variabili statiche.
3   */
4  public class ContoBanca {
5
6      private double saldo;                      //variabile di istanza
7      public static double tassoInteresse = 0;    //variabile STATICA
8      public static int numeroDiConti = 0;         //variabile STATICA
9
10     public ContoBanca() {
11         saldo = 0;
12         numeroDiConti++; //Un metodo di istanza può accedere a una variabile statica
13     }
14
15     public static void setTassoInteresse(double nuovoTasso) {
16         tassoInteresse = nuovoTasso; //Un metodo statico può accedere a
17                                         //una variabile statica, ma non a
18                                         //una variabile di istanza
19     }
20
21     public static double getTassoInteresse() {
22         return tassoInteresse;
23     }
24
25     public static int getNumeroDiConti() {
26         return numeroDiConti;
27     }
28
29     public void deposita(double somma) {
30         saldo = saldo + somma;
31     }
32
33     public double preleva(double ammontare) {
34         if (saldo >= ammontare)
35             saldo = saldo - ammontare;
36         else
37             ammontare = 0;
38         return ammontare;
39     }
40
41     public void aggiungiInteresse() {
42         double interesse = saldo * tassoInteresse;
43         //si può sostituire tassoInteresse con getTassoInteresse()
44         saldo = saldo + interesse;
45     }
46
47     public double getSaldo() {
48         return saldo;
49     }
50
51     public static void mostraSaldo(ContoBanca conto) {
52         System.out.print(conto.getSaldo()); //Un metodo statico non può invocare
53                                         //un metodo di istanza, a meno che
54                                         //non lo faccia tramite un oggetto
55     }
56 }

```

La classe ha metodi statici get e set per il tasso di interesse, un metodo statico get per il numero di conti e metodi di istanza per depositare, prelevare, aggiungere interessi e ottenere il saldo del conto. Infine ha un metodo statico per mostrare il saldo di ogni conto.

Nella definizione di un metodo statico, non è possibile far riferimento a una variabile di istanza. Infatti, dato che un metodo statico può essere invocato senza utilizzare alcun oggetto, non esiste alcuna variabile di istanza alla quale ci si può riferire. Per esempio, il metodo statico setTassolInteresse può fare riferimento alla variabile statica tassolInteresse, ma non alla variabile di istanza saldo, che non è statica.

Un metodo statico non può invocare un metodo di istanza senza avere un oggetto da usare nell'invocazione. Ad esempio il metodo statico mostraSaldo accetta come parametro un oggetto di tipo

ContoBanca. Il metodo usa l'oggetto per invocare il metodo di istanza getSaldo. Infatti, mostraSaldo può contenere informazioni sul saldo di un conto solo attraverso un oggetto che rappresenti il conto.

```

1  public class ContoBancaDemo {
2
3      public static void main(String[] args) {
4          ContoBanca.setTassoInteresse(0.01);
5          ContoBanca mioConto = new ContoBanca();
6          ContoBanca tuoConto = new ContoBanca();
7
8          System.out.println("Ho depositato 10.75 Euro.");
9          mioConto.deposita(10.75);
10         System.out.println("Hai depositato 75 Euro.");
11         tuoConto.deposita(75.00);
12         System.out.println("Hai depositato 55 Euro.");
13         tuoConto.deposita(55.00);
14
15         double contante = tuoConto.preleva(15.75);
16         System.out.println("Hai prelevato " + contante + " Euro.");
17         if (tuoConto.getSaldo() > 100.00) {
18             System.out.println("Hai ricevuto un interesse.");
19             tuoConto.aggiungiInteresse();
20         }
21         System.out.println("Il tuo conto è di " + tuoConto.getSaldo() + " Euro.");
22
23         System.out.print("Il mio conto è di ");
24         ContoBanca.mostraSaldo(mioConto);
25         System.out.println(" Euro.");
26         int conti = ContoBanca.getNumeroDiConti();
27         System.out.println("Abbiamo aperto " + conti + " conti in banca oggi.");
28     }
29 }
```

Nella definizione di un metodo statico, come il metodo main, non si può utilizzare un metodo che ha come oggetto chiamante un oggetto this implicito o esplicito.

9.2.3 Suddividere le attività del metodo main in sotto-attività

Quando un programma definito nel metodo main ha una logica complicata o il suo codice è ripetitivo, è possibile definire più metodi statici nei quali eseguire le varie sotto-attività e richiamare questi metodi dal metodo main.

Quando si sviluppano programmi, anche semplici, è bene semplificare la logica del metodo main di un'applicazione attraverso invocazioni di metodi ausiliari. Questi metodi dovrebbero essere statici, in quanto main è statico. Poiché questi sono metodi ausiliari, dovrebbero essere anche privati.

9.2.4 Aggiungere un metodo main a una classe

A volte ha senso avere un metodo main all'interno di una definizione di classe. La classe può quindi avere due scopi:

- può essere utilizzata per creare oggetti in altre classi
- può essere eseguita come programma

In pratica se si esegue la classe come programma verrà invocato il suo metodo main, se viene invocata invece da un'altra classe il main verrà ignorato.

```

import java.util.Scanner;

public class Specie {
    private String nome;
    private int popolazione;
    private double tassoCrescita;

    <I metodi leggiInput, scriviOutput, prediciPopolazione, setSpecie, getNome,
    getPopolazione, getTassoDiCrescita e equals vanno in questa posizione.
    Sono gli stessi definiti nel Listato 8.16>

    public static void main(String args[]){
        Specie specieAdOggi = new Specie();

        System.out.println("Inserisci i dati sulla specie ad oggi:");
        specieAdOggi.leggiInput();
        specieAdOggi.scriviOutput();
        System.out.println("Inserisci il numero di anni" +
            " su cui calcolare la proiezione:");

        Scanner tastiera = new Scanner(System.in);
        int numeroAnni = tastiera.nextInt();
        int popolazioneFutura = specieAdOggi.prediciPopolazione(numeroAnni);
        System.out.println("Tra " + numeroAnni +
            " la popolazione sara' di " + popolazioneFutura);
        specieAdOggi.setSpecie("Felini", 10, 15);
        System.out.println("La nuova specie e':");
        specieAdOggi.scriviOutput();
    }
}

```

9.2.5 Classi wrapper

Java distingue fra tipi primitivi e classi definite dal programmatore.

Se un metodo ha bisogno di un argomento di un tipo classe ma si ha a disposizione un valore di tipo primitivo, è necessario convertire il valore primitivo in un equivalente “valore” di un certo tipo classe che corrisponde al tipo int primitivo. Per effettuare questa conversione Java fornisce una classe wrapper per ciascuno dei tipi primitivi. Tali classi definiscono i metodi che possono agire sui valori di un tipo primitivo.

Tipo primitivo	Classe wrapper	
int	Integer	Ad esempio, la classe wrapper per il tipo primitivo int è la classe predefinita Integer. Se si vuole convertire un valore int in un oggetto di tipo integer occorre utilizzare la seguente formula:
long	Long	
float	Float	Integer n = new Integer ([valore desiderato]);
double	Double	la sua istruzione contraria è:
char	Character	int i = n.intValue();

la conversione da tipo primitivo a classe wrapper viene definito **boxing** mentre il suo contrario viene chiamato **unboxing**.

Le operazioni di boxing e unboxing vengono svolte in automatico da Java. Quindi, se n referenzia un oggetto della classe “Integer, int i = n.intValue()” può essere scritto semplicemente come “int i = n”.

L’importanza principale delle classi wrapper è che esse contengono una serie di costanti e metodi statici molto utili. Per esempio, è possibile utilizzare la classe wrapper associata per sapere il più grande e più piccolo valore associabile ai corrispettivi tipi primitivi.

Poiché un metodo main è statico, non può contenere una chiamata di un metodo di istanza della stessa classe a meno che al suo interno non sia definito un oggetto della classe stessa e che si usi questo oggetto per l’invocazione del metodo di istanza. Ad esempio, nell’esempio accanto viene creato un oggetto Specie all’interno del main e questo viene utilizzato per invocare i metodi di classe.

I metodi di parsing invece convertono una stringa numerica nel rispettivo numero (ad esempio `Integer.parseInt(args[i])`).

Le classi wrapper non hanno un costruttore di default.

9.3 OVERLOADING

Java può avere più metodi con lo stesso nome.

9.3.1 Concetti di base dell'overloading

```

1  /**
2   * Questa classe mostra l'overloading.
3   */
4  public class Overload {
5
6      private static double calcolaMedia(double primo, double secondo) { //due double
7          return (primo + secondo) / 2.0;
8      }
9
10     private static double calcolaMedia(double primo, double secondo, double terzo) { //tre double
11         return (primo + secondo + terzo) / 3.0;
12     }
13
14     private static char calcolaMedia(char primo, char secondo) { //due char
15         return (char)((int)primo + (int)secondo) / 2;
16     }
17
18     public static void main(String args[] ) {
19         double medial = Overload.calcolaMedia(40.0, 50.0);
20         double media2 = Overload.calcolaMedia(1.0, 2.0, 3.0);
21         char media3 = Overload.calcolaMedia('a', 'c');
22
23         System.out.println("medial = " + medial);
24         System.out.println("media2 = " + media2);
25         System.out.println("media3 = " + media3);
26     }
27 }
```

Quando si assegna lo stesso nome a due o più metodi all'interno di una stessa classe si dice che si sta effettuando l'overloading del nome del metodo.
 L'overloading consiste nell'assegnare lo stesso nome a più definizioni di metodi di una stessa classe ma **differenziate in base ai parametri che ricevono**.

Il nome di un metodo, il numero e il tipo di parametri che utilizza sono detti firma (signature) del metodo.

L'overloading può essere applicato ad ogni tipo di metodo.

9.3.2 Overloading e conversione automatica di tipo

Java cerca di usare l'overloading prima di usare la conversione automatica di tipo. Se java individua una definizione di metodo che corrisponde al tipo di argomenti fornito, utilizza tale definizione. Java non effettua una conversione automatica di tipo degli argomenti passati a un metodo finché non si accerta che non esista una definizione di metodo che corrisponde al tipo degli argomenti passati al metodo.

L'overloading deve essere utilizzato quando si ha una buona ragione per farlo, ma non quando nomi di metodo differenti sarebbero più descrittivi. Questa indicazione chiaramente non può essere seguita quando si definiscono costruttori dal momento che devono tutti avere lo stesso nome della classe.

9.3.3 Overloading e tipo di ritorno

Non si può effettuare l'overloading di un nome di metodo fornendo due definizioni la cui intestazione differisce solo per il tipo di valore restituito.

9.4 INFORMATION HIDING RIVISITATO

Questo paragrafo discute un problema subdolo che può presentarsi nel momento in cui si definiscono certi tipi di classi. Questo problema non riguarda classi le cui variabili di istanza sono di tipo primitivo.

9.4.1 Privacy leak

```

1  public class CoppiaAnimali {
2      private Animale primo, secondo;
3
4      public CoppiaAnimali(Animale primoAnimale, Animale secondoAnimale) {
5          primo = primoAnimale;
6          secondo = secondoAnimale;
7      }
8
9      public Animale getPrimo() {
10         return primo;
11     }
12
13     public Animale getSecondo() {
14         return secondo;
15     }
16
17     public void scriviOutput() {
18         System.out.println("Primo animale nella coppia:");
19         primo.scriviOutput();
20         System.out.println("\nSecondo animale nella coppia:");
21         secondo.scriviOutput();
22     }
23 }
24 /**
25 * Programma giocattolo che dimostra come un programmatore
26 * possa accedere e modificare gli attributi privati
27 * della classe CoppiaAnimali
28 */
29 public class Hacker {
30
31     public static void main(String[] args) {
32
33         Animale buono = new Animale("Cane da Guardia", 5, 75.0);
34         Animale altro = new Animale("Fido", 4, 60.5);
35         CoppiaAnimali coppia = new CoppiaAnimali(buono, altro);
36         System.out.println("La coppia:");
37         coppia.scriviOutput();
38
39         Animale cattivo = coppia.getPrimo();
40         cattivo.setAnimale("Bulla", 1200, 500);
41
42         System.out.println("\nLa coppia ora:");
43         coppia.scriviOutput();
44
45         System.out.println("L'animale non era molto privato!");
46         System.out.println("Sembra una falla di sicurezza.");
47     }
48 }
```

Una classe può avere variabili di istanza di qualsiasi tipo, anche di tipo classe. Tuttavia, usare variabili di istanza di tipo classe può introdurre un problema. Questo si presenta poiché le variabili di tipo classe contengono l'indirizzo di memoria in cui si trova fisicamente l'oggetto. Quindi un assegnamento errato potrebbe portare due oggetti differenti a referenziare lo stesso indirizzo. Nonostante le variabili siano di tipo privato, queste possono essere viste tramite i metodi get e set. Quindi, assegnando lo stesso indirizzo tramite metodo get, si ottiene così l'accesso alle modifiche anche all'altro oggetto referenziato per sbaglio.

Un altro metodo è l'utilizzo della duplicazione degli oggetti. Questi duplicati sono chiamati cloni a cui un estraneo può avere liberamente accesso per ottenerne i dati.

Nell'esempio qui mostrato viene creato un oggetto di tipo CoppiaAnimali e quindi viene modificato lo stato dell'oggetto referenziato dalla sua variabile privata. Invocando il metodo setAnimale sulla nuova variabile il programmatore modifica lo stato dell'oggetto referenziato. Questo fenomeno è detto privacy leak.

Un modo semplice per evitare ciò consiste nell'usare solo variabili di istanza di tipo primitivo, limitazione molto forte per lo sviluppatore. Un'altra non creare metodi set. Un'altra ancora definire metodi get che restituiscano i singoli attributi delle variabili di istanza.

9.5 RAPPRESENTARE IN UML LE RELAZIONI ASSOCIATIVE FRA CLASSI (APPUNTI IN CLASSE)

Cane	Luna	Kira
<ul style="list-style-type: none"> • Attributi <ul style="list-style-type: none"> • Razza, nome, peso, colore • Comportamenti <ul style="list-style-type: none"> • Abbaia • Mangia • Morde 	<ul style="list-style-type: none"> • Carlino, 7Kg, albicocca 	<ul style="list-style-type: none"> • Labrador retriver, 16 Kg, Beige

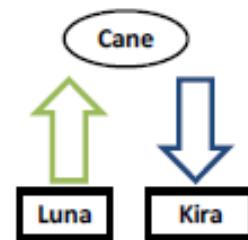
9.5.1 Classificazione e istanziazione

Una classe modella le proprietà comuni di un insieme di oggetti.

Che legame esiste tra classi e istanze? Intuitivamente “Cane” è un concetto più astratto di Luna e Kira.

Il legame è in due sensi:

- **Classificazione:**
 - Parto dall’oggetto e cerco di classificarlo
 - Capire che “Luna è un cane”
- **Istanziazione:**
 - Parto dalla definizione di classe e creo istanze
 - Creare Kira come esemplare di Cane



9.5.2 Astrazioni

Il legame concettuale che esiste tra una classe e le sue istanze è un particolare esempio di astrazione.

Un’astrazione è un procedimento concettuale per mezzo del quale:

- Si definisce un concetto più generale (e in tal senso astratto) a partire da concetti più elementari (e in tal senso più concreti)
- Rimuovendo dalla descrizione del concetto gli aspetti di dettaglio e particolari
- Mettendo in evidenza gli aspetti comuni e generali

Come si applica al legame tra classe e le sue istanze? Definisco la classe osservando le istanze, raccogliendo a fattor comune le caratteristiche condivise e operando una selezione tra queste in base a cosa mi serva.

Astrazioni strutturali: modellano oggetti e classi e relazioni fra essi.

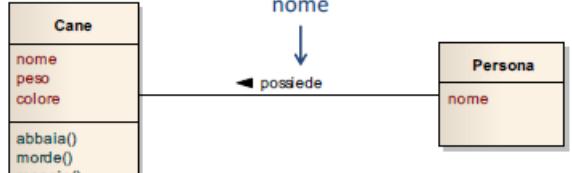
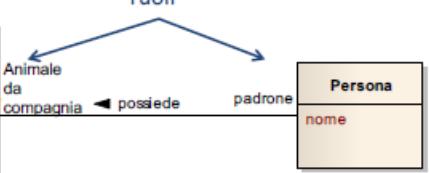
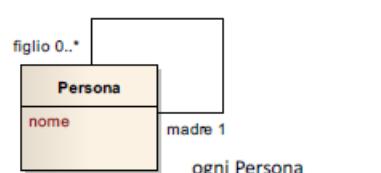
- **Classificazione** (“instance_of”)
 - Lega istanze e classi
 - La classe definisce le caratteristiche comuni degli oggetti di un insieme
 - Ogni oggetto della classe possiede le proprietà definite dalla classe
 - Esempio: “Luna è un (esemplare di) Cane”
- **Generalizzazione** (“is_a”)
 - Lega una classe genitore (superclasse) a una o più classi figlie (sottoclassi) che ne sono sottinsieme
 - Una classe figlia è una classe genitore

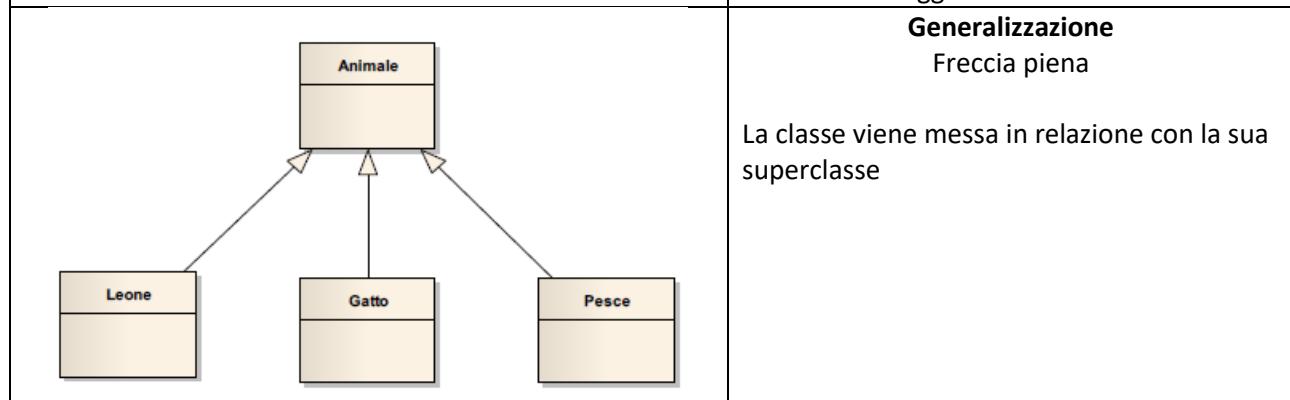
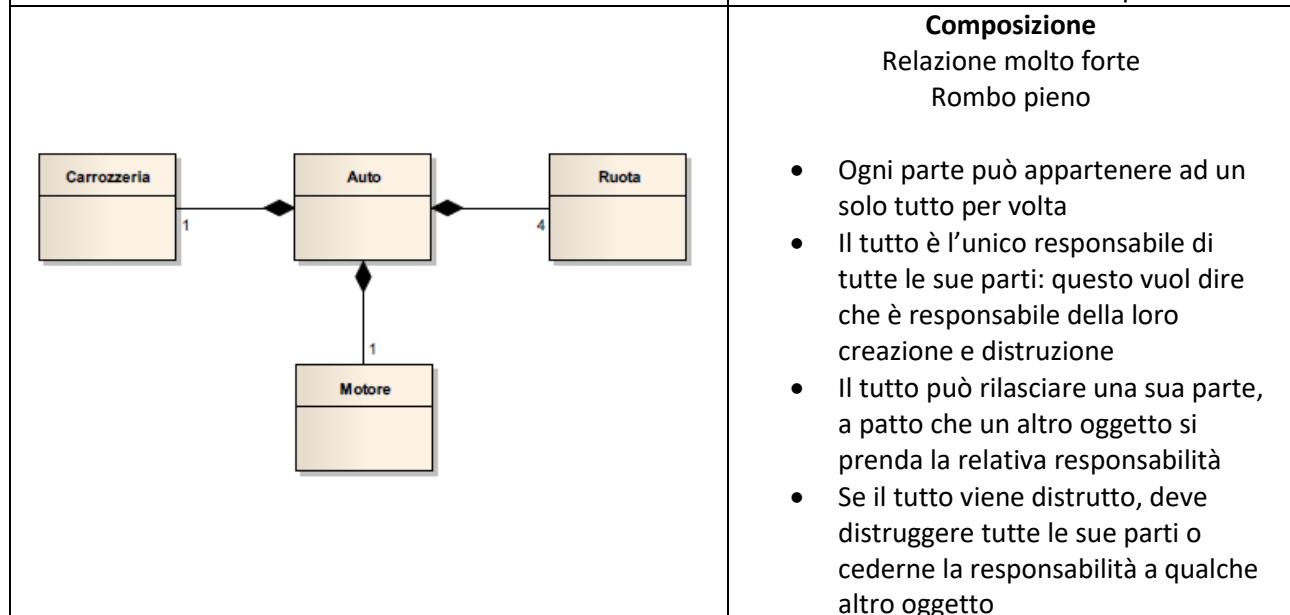
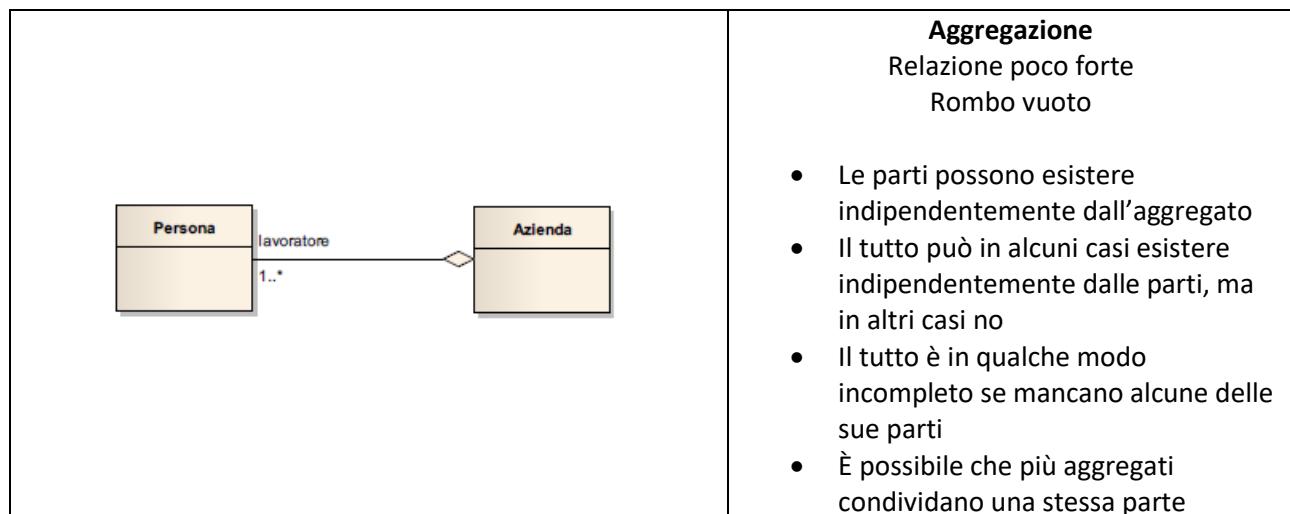
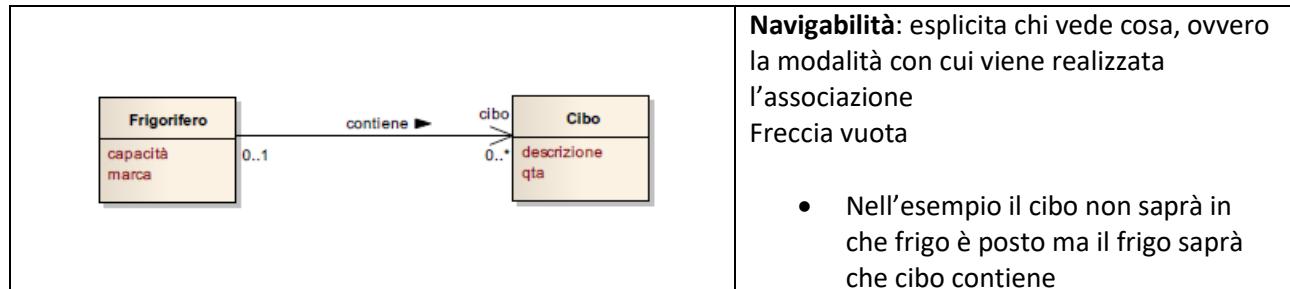
- Ogni esemplare della classe figlia possiede tutte le proprietà definite nella classe padre
 - Esempio: la classe Cane è una sottoclasse della classe Mammifero → i cani sono mammiferi. Ogni esemplare di Cane possiede tutte le proprietà definite dalla classe Cane e anche tutte le proprietà definite dalla classe Mammifero
- Ogni oggetto della sottoclasse appartiene anche alla superclasse (astrazione classificazione – “instance_of”)
 - Esempio: Kira è un Cane, quindi è anche un Mammifero
- **Aggregazione (“part_of”)**
 - Lega una classe “aggregato” con un insieme di classi “parti”
 - Ogni oggetto di Aggregato è costituito da oggetti delle classi “parti”
 - Caso particolare di associazione
 - Esempi: un’automobile comprende motore, carrozzeria, ruote...
 - ATTENZIONE: non confondere aggregazione e generalizzazione
 - Un cane è un mammifero (generalizzazione)
 - Un cane fa parte di un branco (aggregazione)
 - Un cane NON È un branco
- **Associazione (“has_a”)**
 - Definisce una connessione logica fra oggetti di una classe e oggetti di un’altra classe
 - Esempi:
 - Classi: Cane, persona → associazione: una persona è padrona di un cane

9.5.3 Formalismi di rappresentazione

ER (Entity-Relationships)	UML (Unified Modeling Language)
<ul style="list-style-type: none"> • Entità • Relazioni • Nato nell’area “basi dati” 	<ul style="list-style-type: none"> • Classi • proprietà • relazioni • comportamenti • ... • nato nell’area “programmazione”

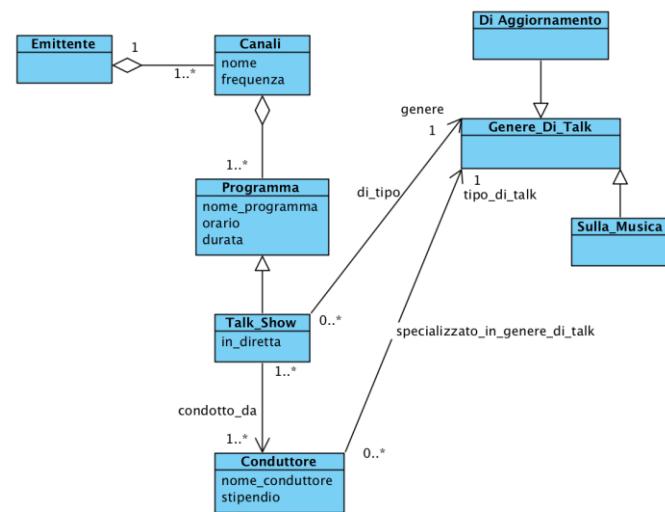
<p>Nome della classe → </p> <p>Attributi → </p> <p>Operazioni → </p>	<p>Identificazione delle classi e delle relazioni fra le stesse: attività fondamentale in fase di analisi e di progetto</p>
	<p>Identificazione degli oggetti del sistema e dei loro stati durante la loro vita</p>

 <p style="text-align: center;">associazione</p>	Associazione: è una linea che collega le classi coinvolte
 <p style="text-align: center;">nome</p> <p style="text-align: center;">← possiede</p>	Nome: esprime il significato dell'associazione <ul style="list-style-type: none"> • Spesso è un verbo • È opzionale (ma fortemente consigliato) • Indica il verso di lettura, ad esempio qua è "Persona possiede Cane"
 <p style="text-align: center;">ruoli</p> <p style="text-align: center;">Animale da compagnia ← possiede padrone</p>	Ruolo: esprime il ruolo giocato dal partner <ul style="list-style-type: none"> • Spesso è un sostantivo o un aggettivo • È opzionale (ma fortemente consigliato)
 <p style="text-align: center;">Animale da compagnia ← possiede padrone</p> <p style="text-align: center;">0..* ← 0..1</p> <p style="text-align: center;">cardinalità</p>	Cardinalità: esprime quante istanze della classe possono essere associate all'altra classe <ul style="list-style-type: none"> • Molto importante
 <p style="text-align: center;">voloInPartenza parte ➤ partenza</p> <p style="text-align: center;">voloInArrivo atterrta ➤ arrivo</p> <p style="text-align: center;">cardinalità</p>	Associazione multipla: esplicita più associazioni tra una coppia di classi per arricchirne la descrizione
<p>ogni Persona ha un numero qualsiasi di figli (che sono Persone)</p> <p style="text-align: center;">figlio 0..*</p>  <p style="text-align: center;">madre 1</p> <p style="text-align: center;">ogni Persona ha una madre (che è una Persona)</p>	Associazioni cappio: una classe può essere messa in relazione con sé stessa <ul style="list-style-type: none"> • ATTENZIONE: in questo caso vi è un problema a definire la prima madre originale ("Eva") in quanto madre ha cardinalità 1, ovvero ogni persona ha una madre, ma la prima madre non ne ha una, quindi si potrebbe correggere con cardinalità 0..1

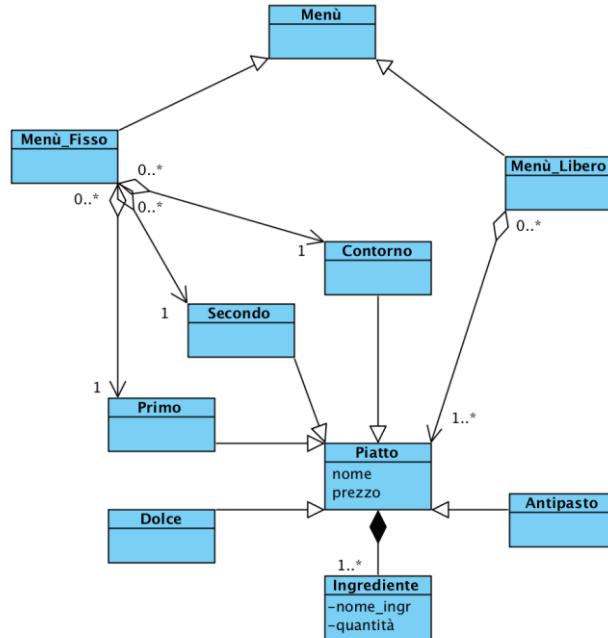


9.5.4 Esempi di diagramma

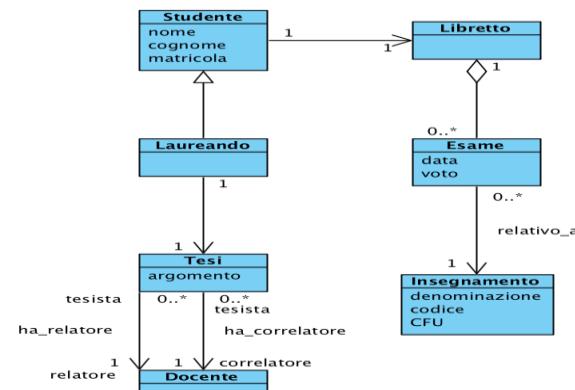
Una emittente televisiva ha un insieme di canali, ciascuno caratterizzato da un nome e una frequenza. Ciascun canale ha una programmazione costituita da un insieme di programmi televisivi caratterizzati da un nome, una durata e un orario d'inizio. Se i programmi televisivi sono talk show, allora hanno specificato se sono in diretta e l'insieme di conduttori coinvolti. Un conduttore ha un nome, uno stipendio percepito e un genere di talk show (basato sulla musica, di aggiornamento, ecc...)



Un ristorante propone due tipi di menù: il primo (denominato fisso) ha esattamente un primo, un secondo e un contorno. Il ristorante di volta in volta deciderà quale sarà il primo, il secondo e i contorni. Il secondo (denominato libero) è costituita da un insieme di piatti che possono essere dei primi, secondi, contorni, dolci, antipasti. Il piatto è caratterizzato da un prezzo, da un nome e un insieme di ingredienti. Ciascun ingrediente è caratterizzato da un nome.

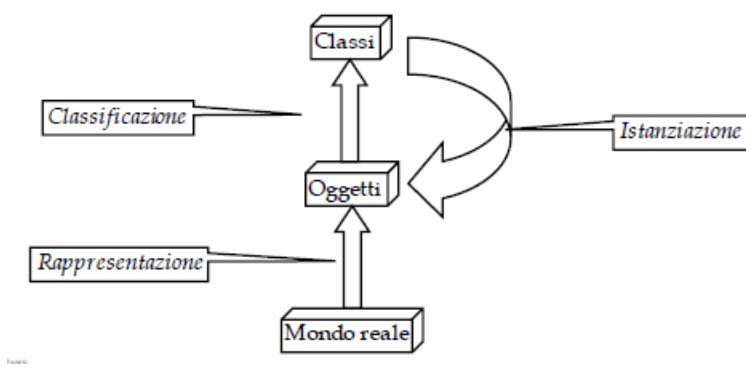


Uno studente è caratterizzato dall'avere una matricola, un nome e un cognome. Possiede inoltre un libretto che contiene gli esami da lui sostenuti con data e voto, i quali sono relativi a insegnamenti. Un insegnamento è caratterizzato da una denominazione, un codice identificativo e da un numero di CFU. Uno studente laureando è uno studente che sta svolgendo una tesi di laurea su un certo argomento e il cui relatore e correlatore sono docenti universitari



9.5.5 Le caratteristiche di un oggetto

- **Stato:** è una delle possibili condizioni in cui può trovarsi
 - È dato dal valore dei suoi attributi
 - Le variabili sono chiamate attributi o variabili d'istanza
 - Gli attributi di un oggetto possono essere tipi semplici o a loro volta oggetti
- **Comportamento:** determina come un oggetto risponde alle richieste di altri oggetti
 - È definito dalle sue operazioni
 - È la modalità con cui un oggetto risponde alle richieste da parte di altri oggetti
 - Gli oggetti comunicano tra loro scambiandosi messaggi (chiamati anche comandi) attraverso le rispettive interfacce
 - I messaggi attivano i metodi che determinano il comportamento degli oggetti
- **Identità:** due oggetti anche se si trovano nello stesso stato, sono comunque due entità ben distinte
 - Ogni oggetto ha un proprio OID (Object Identifier) che è:
 - Univoco nel sistema
 - Invariante nel tempo
 - Un oggetto esiste indipendentemente dal valore dei suoi attributi



Per poter creare oggetti serve un modello che incorpora le caratteristiche di stato e di comportamento di un oggetto da cui partire, è una sorta di modulo che definisce oggetti omogenei una volta per tutte. Una volta definito è possibile creare gli oggetti. Tale modello si chiama classe.

La classe viene creata tramite un processo di astrazione.

Cos'è una classe?

Tutti gli oggetti che condividono le stesse proprietà (attributi) e comportamenti (metodi) possono essere classificati insieme.

Una classe rappresenta una e una sola astrazione a partire dalla quale è possibile creare oggetti.

La classe è quella “struttura” che definisce le caratteristiche delle sue istanze. L’istanza o oggetto è un esemplare creato a partire dalla sua classe.

La classe:

- Specifica l’interfaccia che ogni suo oggetto offre verso gli altri con cui interagisce
- Implementa attributi e metodi interni e operazioni dei metodi

9.5.6 Definizione di classe in java

Come descrivere al programmatore le classi? C'è un metodo "esteso" ma scomodo e il metodo di rappresentazione tramite UML, molto più semplice. Ecco ad esempio la definizione di classe "Punto":

Versione estesa	UML
<p>«Realizzare una classe che rappresenta un punto in uno spazio bidimensionale a cui attribuire nome Punto. Qualsiasi oggetto creato a partire dalla classe Punto è caratterizzato dall'avere un attributo con identificativo x e di tipo int (che rappresenta l'ascissa) e un altro attributo con identificativo y e di tipo int (che rappresenta l'ordinata). Entrambi gli attributi hanno visibilità pubblica.»</p>	<pre> classDiagram class Punto { +x: int +y: int } </pre>

Modificatore d'accesso	Java	Simbolo UML	A chi è visibile
Pubblico	Public	+	A tutti
Default	Non viene segnato	~	A tutte le classi del package
Protetto	Protected	#	Attribuibile solo ai metodi e alle variabili interni alla classe. Visibile all'interno del package e classi derivate ovunque
Privato	Private	-	Più stretto di tutti, riferimenti ammessi solo alle sottoclassi appartenenti alla medesima classe

```

public class Punto{
    public int x;
    public int y;
}

public class TestPunto{
    public static void main(String args[]){
        punto p1, p2; //creazione nello stack
        p1 = new Punto(); //creazione nell'heap
        p2 = new Punto();

        p1.x = 3; //assegnazioni variabili nell'heap
        p1.y = 4;

        p2.x = 50;
        p2.y = 90;

        if(p1.x >= p2.x){
            System.out.println("Il punto p1 ha x maggiore");
        } else {
            System.out.println("Il punto p2 ha x maggiore");
        }
    }
}

public class TestPunto{
    public static void main(String args[]){
        Punto p1, p2; //creazione nello stack
        p1 = new Punto(); //creazione nell'heap
        p2 = new Punto();

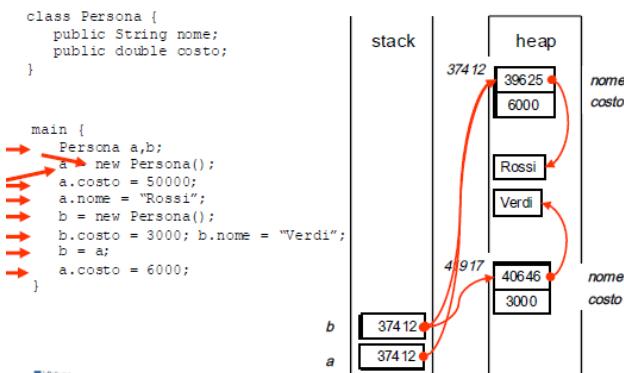
        int n, m;
        n = 10;
        m = n; //in questo caso sto modificando VALORI, quindi
               // n = 10 e m = 10
        n = 9;

        p1.x = 1; //assegnazioni variabili nell'heap
        p1.y = 2;

        p2 = p1; // ATTENZIONE: qua si sta cambiando l'INDIRIZZO
                  // nello stack e non i valori interni, quindi, di fatto
                  // dopo questa modifica p2 e p1 hanno la stessa identità
                  // (e non sono solo nello stesso stato)

        p2.x = 3;
        p2.y = 4;

        System.out.println("p1 x: " + p1.x + " y: " + p1.y);
        System.out.println("p2 x: " + p2.x + " y: " + p2.y);
    }
}
  
```



area di memoria adibita al contenimento dei valori delle variabili.

Nel secondo codice vediamo che `p2 = p1` non effettua l'operazione preventiva ma cambia il riferimento nello stack. Questo perché stack e heap funzionano in maniera particolare. **Nel caso di definizione di variabili primitive, nello stack viene creata un'area di memoria contenente i valori delle variabili. Nel caso di creazione di variabili complesse, nello stack vengono inseriti gli indirizzi di memoria a cui fare riferimento nell'heap, dove verrà creata (tramite l'istruzione new) l'apposita**

La costante null viene utilizzata per inizializzare un reference ad un valore di default

Puntatore (“pointer”): una variabile che contiene un indirizzo.

Ref(“reference”) in Java: un puntatore:

- *Protetto*: non si possono eseguire manipolazioni (aritmetiche, logiche, ecc.)
- “*tipato*”: può riferirsi solo a oggetti di un certo tipo (controllo a compilazione e a run-time)

Garbage collector (presente in Java): gli oggetti non più referenziati sono deallocated automaticamente.

Attenzione! Scrivere tra reference (`a == b`) controlla che puntino allo stesso indirizzo, non controlla i loro contenuti.

“`Instanceof`” permette di verificare se un oggetto è stato istanziato a partire dalla classe specificata.

Sintassi: reference `instanceof` NomeClasse (esempio: `p1 instanceof Punto`).

9.6 ARRAY NELLE DEFINIZIONI DI CLASSE

Gli array possono essere utilizzati all'interno delle definizioni di classe.

9.6.1 Array di riferimenti

Scrivere, ad esempio, “`Specie[] esemplare = new Specie[3]`” crea automaticamente un array di tre elementi che contiene oggetti della classe Specie.

9.7 ENUMERAZIONI COME CLASSI

Quando viene creata un'enumerazione, Java crea automaticamente una classe. Ad esempio “enum Semi {FIORI, QUADRI, PICCHE, CUORI}” crea la classe Semi e crea quattro variabili di classe. Questa classe ha diversi metodi fra cui equals, compareTo, ordinal, toString, valueOf.

È lecito l'assegnamento “`Semi s = Semi.Quadri`”.

9.8 PACKAGE

Un package è una collezione di classi correlate a cui viene assegnato un nome. Un package svolge il ruolo di libreria di classi, le quali possono essere utilizzate all'interno di un qualsiasi programma.

9.8.1 Package e istruzione import

Un package sono classi in una cartella. Il nome della cartella deve corrispondere al nome del package. Ciascun file del package deve contenere la seguente riga all'inizio del file stesso:

```
package nome_package;
```

niente può precedere questa riga a parte commenti e righe bianche.

Qualsiasi programma o classe può usare tutte le classi contenute in un package inserendo un'opportuna istruzione import all'inizio del file contenente il programma o la definizione della classe.

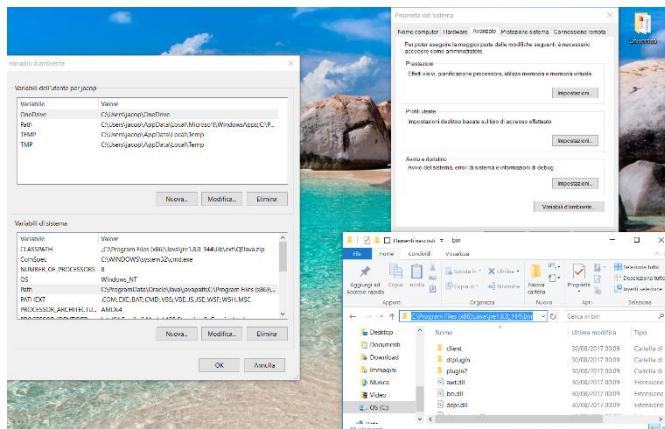
Per importare una classe del package si scrive:

```
import nome_package.nome_classe;
```

Per importare direttamente tutte le classi di un package si scrive:

```
import nome_package.*;
```

9.8.2 Nomi di package e cartelle

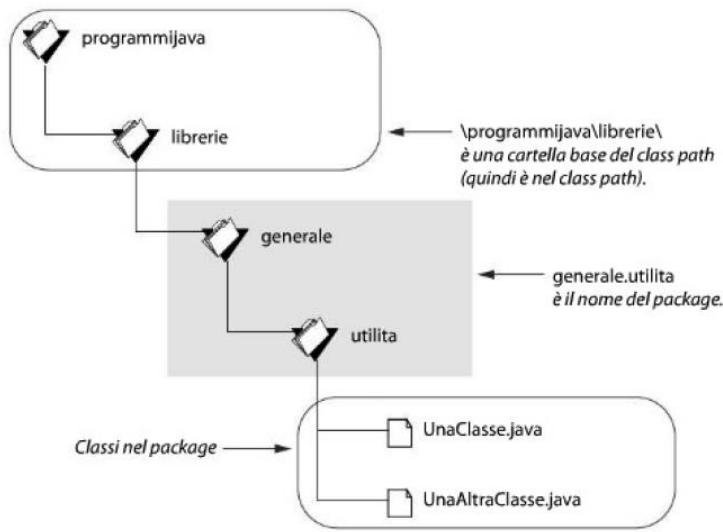


Per individuare la cartella del package, il compilatore ha bisogno di due informazioni:

- Il nome del package
- Le cartelle elencate all'interno della variabile percorso (class path)

Il valore della variabile class path indica a java dove iniziare la propria ricerca per individuare un certo package. La variabile class path è propria del sistema operativo. Queste cartelle sono dette cartelle base del class path.

Le cartelle base del class path vengono specificate attraverso la variabile d'ambiente CLASSPATH.



un modo per dire nel CLASSPATH di cercare anche nella cartella corrente è scrivere semplicemente punto (".") nel CLASSPATH.

9.8.3 Conflitti tra nomi

Esiste un modo alternativo per utilizzare i package. Nel caso vi sia un conflitto tra nomi per le classi ma queste siano in cartelle differenti, i package risolvono il problema. Notare che questa soluzione è alla base del sistema operativo poiché due file con lo stesso nome e nella stessa classe non possono coesistere.

10 EREDITARIETÀ

L'ereditarietà permette di definire una classe in una forma molto generale e, in un secondo momento, di utilizzarla come base di partenza per definire nuove classi, che sono specializzazioni della classe generale da cui, appunto, ereditano metodi e variabili di istanza da questa.

10.1 CONCETTI DI BASE SULL'EREDITARIETÀ

```

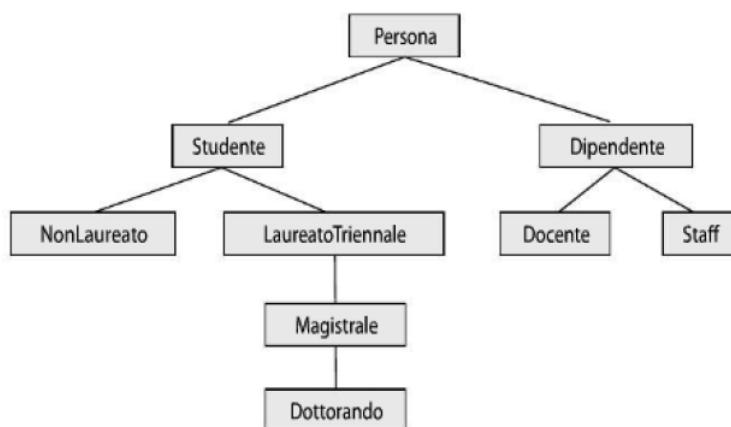
1  public class Persona {
2      private String nome;
3
4      public Persona() {
5          nome = "Ancora nessun nome";
6      }
7
8      public Persona(String nomeIniziale) {
9          nome = nomeIniziale;
10     }
11
12     public void setNome(String nuovoNome) {
13         nome = nuovoNome;
14     }
15
16     public String getNome() {
17         return nome;
18     }
19
20     public void scriviOutput() {
21         System.out.println("Nome: " + nome);
22     }
23
24     public boolean haLoStessoNome(Persona altraPersona) {
25         return this.nome.equalsIgnoreCase(altraPersona.nome);
26     }
27 }
```

L'ereditarietà permette di definire una classe più generale e di definire in seguito classi specializzate che aggiungono nuovi dettagli alla classe generale.

Lo scenario che analizzeremo sarà quello di una semplice classe chiamata “Person”, il cui unico attributo è il nome e i metodi interagiscono solo con questo.

10.1.1 Classi derivate

Si supponga di dover progettare un programma per l'archiviazione di informazioni di un'università. Il sistema gestisce le schede informative degli studenti, dei docenti e di altro personale.



Si supponga di dover definire una classe per rappresentare dei veicoli. Un veicolo è caratterizzato dall'avere definite le variabili di istanza per memorizzare il numero di ruote e il numero massimo di occupanti. Nella classe dovrebbero essere definiti anche i metodi get e set.

Si immagini ora di dover definire una nuova classe per rappresentare delle automobili. Invece di ripetere nella automobile le definizioni delle variabili e dei metodi della classe veicolo, si può sfruttare il meccanismo dell'ereditarietà.

Esiste una gerarchia naturale in quanto tutti loro sono persone, quindi tutte le schede gestite dal sistema sono schede di persone.

Una sottoclasse intermedia è quella dei dipendenti dell'università, al cui interno ci sono sia i docenti che il resto del personale. Queste sottoclassi potrebbero essere suddivise in maniera ancora più specifica.

Il listato seguente contiene la definizione

di una classe che rappresenta gli studenti. Uno studente è una persona e pertanto si può definire una classe derivata (o sottoclasse) di Persona. La classe già esistente è definita “classe base” o “superclasse”. **Nella definizione della sottoclasse viene specificato nel nome della classe “extends [nome superclasse]”.**

```

1  public class Studente extends Persona {
2      private int matricola;
3
4      public Studente() {
5          super(); //non è ancora stato visto
6          matricola = 0; //Ancora nessuna matricola
7      }
8
9      public Studente(String nomeIniziale, int matricolaIniziale) {
10         super(nomeIniziale);
11         matricola = matricolaIniziale;
12     }
13
14     public void reimposta(String nuovoNome, int nuovaMatricola) {
15         setNome(nuovoNome);
16         matricola = nuovaMatricola;
17     }
18
19     public int getMatricola() {
20         return matricola;
21     }
22
23     public void setMatricola(int nuovaMatricola) {
24         matricola = nuovaMatricola;
25     }
26
27     public void scriviOutput() {
28         System.out.println("Nome: " + getNome());
29         System.out.println("Matricola: " + matricola);
30     }
31
32     public boolean equals(Studente altroStudente) {
33         return this.haLoStessoNome(altroStudente) &&
34         (this.matricola ==
35         altroStudente.matricola);
36     }
37 }

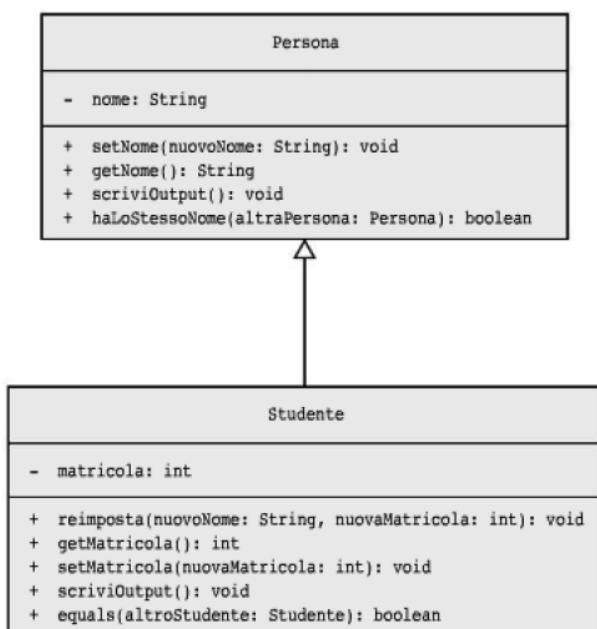
```

scriviOutput, quindi la classe studente avrebbe due metodi con lo stesso nome e gli stessi parametri.

In questo caso il metodo della classe derivata ridefinisce il metodo presente nella classe base (overriding). Quando si ridefinisce un metodo, si può cambiare a piacere il cormo della sua definizione ma non la sua intestazione. Esiste un solo caso in cui è possibile modificare il tipo di ritorno ma lo vedremo poi.

10.1.3 Cambiare il tipo di ritorno di un metodo ridefinito

In una classe derivata, quando si ridefinisce un metodo ereditato, in generale non è possibile modificare il metodo di ritorno. Per esempio, non è possibile cambiare un metodo `void` in un metodo che restituisce un valore (di un qualsiasi tipo) o cambiare il tipo restituito. **L'unica eccezione è: se il tipo di ritorno è una classe, il metodo ridefinito può restituire una qualsiasi delle sue classi derivate.** Per esempio, se un metodo restituisce un oggetto `Persona`, questa può essere riscritta restituendo una classe `Studente`.



La classe `Studente`, come ogni altra classe derivata, eredita le variabili di istanza e i metodi pubblici della classe base che estende. Quando si definisce una classe derivata si definiscono esclusivamente le variabili di istanza e i metodi aggiuntivi.

La relazione tra `Studente` e `Persona` è nota come “relazione *is-a*”. Per l'ereditarietà vengono spesso usati termini famigliari: classe figlia, classe genitore e, nel caso di classe genitore di classe genitore, classe antenato e, per converso, la classe figlia di una classe figlia è una classe discendente.

10.1.2 Metodi ridefiniti: overriding

La classe `Studente` e la classe `Persona` definiscono entrambi un metodo

Il tipo di ritorno così modificato prende il nome di “tipo di ritorno covariante”.

10.1.4 Cambiare i modificatori d'accesso di un metodo ridefinito

Un metodo dichiarato come privato nella classe base può essere ridefinito come pubblico in una classe derivata. **In generale è possibile estendere l'accesso ma non restringerlo.** Questa regola deve essere rispettata perché il codice scritto per i metodi nella classe base deve funzionare anche per i metodi nelle classi derivate.

10.1.5 Overriding vs. overloading

Non si deve confondere l'overriding con l'overloading. L'overriding ha lo stesso nome, lo stesso tipo di ritorno, gli stessi parametri in termini di tipo, ordine e numero.

10.1.6 Ereditarietà nei diagrammi UML

Per indicare l'ereditarietà si utilizza una freccia vuota che punta verso la classe genitore. Il percorso creato dalle frecce è anche quello che deve eseguire la macchina nella ricerca dell'albero di classi quando viene invocato un metodo o un attributo di un antenato.

10.2 INCAPSULAMENTO ED EREDITARIETÀ

10.2.1 Uso delle variabili di istanza private della classe base

Uno dei membri che un oggetto di tipo `Studente` eredita dalla classe `Persona` è la variabile di istanza `nome`.

Bisogna sempre gestire con prudenza le variabili di istanza come `nome`, infatti questa è una variabile privata ereditata. Questo significa che si può accedere alla variabile di istanza `nome` solamente attraverso i metodi di `Persona`.

```

1  public void scriviOutput() {
2      System.out.println("Nome: " + getNome());
3      System.out.println("Matricola: " + matricola);
4  }
5
6  public void scriviOutput(){ //versione errata
7      System.out.println("Nome: " + nome);
8      System.out.println("Matricola: " + matricola);
9  }

```

evitare la possibile introduzione di errori.

Per accedere quindi a questi attributi, anche se uno pensa faccia base della classe figlia, bisogna per forza passare tramite un metodo pubblico.

Questo sistema viene utilizzato per

10.2.2 I metodi privati non sono accessibili

Relativamente alle modalità d'accesso, i metodi privati di una classe base si comportano come le variabili di istanza private. Nel caso dei metodi, però, la restrizione è ancora più forte. **Si può accedere a una variabile privata attraverso i metodi `get` e `set`, mentre i metodi privati sono del tutto indisponibili, è come se non venissero ereditati.**

In realtà, i metodi privati di una classe base possono essere indirettamente disponibili nella classe privata. Se un metodo privato viene utilizzato nella definizione di un metodo pubblico della classe base, il metodo pubblico può essere invocato nella classe derivata che quindi accede indirettamente al metodo privato.

I metodi privati, infatti, dovrebbero essere utilizzati come metodi di supporto ad altri metodi, quindi il loro utilizzo dovrebbe essere limitato alla classe nella quale sono definiti.

10.2.3 Modalità d'accesso `protected`

Esistono due tipi di modificatori per le variabili di istanza e per i metodi che permettono l'accesso per nome delle classi derivate (a parte `public`). I due modificatori sono `protected`, che permette sempre l'accesso alle classi derivate, e `package`, che permette l'accesso se la classe derivata appartiene allo stesso package della classe base.

Un metodo o una variabile d'istanza dichiarato con modificatore `protected` è accessibile per nome dalla classe alla quale appartiene, dalle classi derivate dalla classe a cui appartiene e da qualsiasi classe contenuta nello stesso package.

Quindi, ad esempio, se un metodo è dichiarato come `protected` in `Persona` è accessibile senza problemi anche da `Studente`.

10.3 PROGRAMMARE CON L'EREDITARIETÀ

10.3.1 Costruttori nelle classi derivate

Nella definizione di un costruttore per la classe derivata, la prima tipica azione è di invocare un costruttore della classe base.

Ad esempio, nella classe `Studente` ci saranno costruttori che alla richiesta di inizializzare il nome dovranno utilizzare il costruttore adibito nella classe `Persona`. Occorre quindi delegare a uno dei costruttori di `Persona` l'inizializzazione del solo nome.

In questo caso si utilizza la parola riservata `super` come un nome di un metodo per invocare un costruttore della classe base. Sebbene la classe base definisca due costruttori, l'invocazione con `super` rimanda al costruttore della classe `Persona` che ha un parametro dello stesso tipo. Attenzione ad utilizzare la parola `super` e non il nome della classe base.

```
public Studente () {
    super();
    matricola = 0;
}
```

equivalente a

```
public Studente () {
    matricola = 0;
}
```

L'utilizzo di `super` implica alcuni dettagli: **super deve essere sempre la prima azione specificata da un costruttore.** Se in ogni costruttore della classe derivata non si include un'invocazione esplicita al costruttore della classe base, Java includerà automaticamente un'invocazione al costruttore di default della classe base.

Questo costruttore di default potrebbe non essere quello che doveva essere invocato, di conseguenza spesso è opportuno esplicitare la chiamata al costruttore della classe base in modo tale da scegliere il costruttore più idoneo.

Se la classe base non ha un costruttore di default, l'omissione di `super()` genera un errore di compilazione.

10.3.2 Ancora sul metodo `this`

Si può utilizzare `this` per invocare un altro costruttore della classe di base. **this deve essere la prima linea del costruttore.**

10.3.3 Invocare un metodo ridefinito

Un metodo di una classe derivata che ridefinisce un metodo nella classe base può utilizzare `super` per invocare il metodo ridefinito, ma in modo leggermente differente.

Il modo per invocare il metodo della classe base, nonostante questo sia stato ridefinito nella classe derivata, è l'utilizzo della parola chiave `super`, ovvero "`super. [nomeMetodo]`".

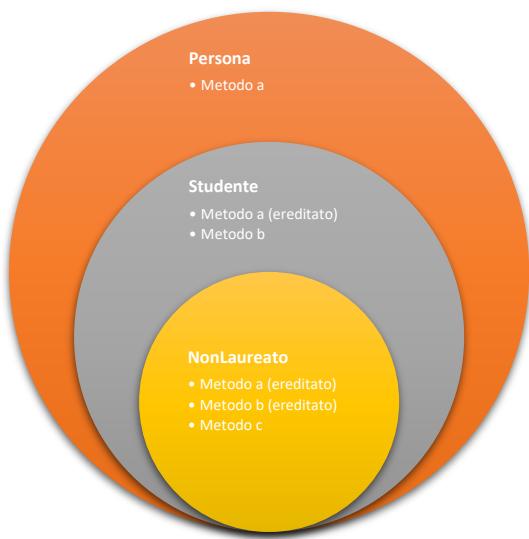
Alcuni linguaggi di programmazione come C++, consentono di derivare una classe da due classi base differenti. Cioè si può derivare la classe C dalle classi A e B- questa caratteristica, nota come **ereditarietà**

multipla, non è permessa in Java. Si può tuttavia derivare la classe B dalla classe A e poi derivare C dalla classe B.

Attenzione: non si può usare super ripetutamente per invocare un metodo di una classe antenata.

10.3.4 Compatibilità di tipo

Un oggetto può comportarsi come se fosse di più tipi in virtù dell'ereditarietà. La classe NonLaureato deriva dalla classe Studente, che a sua volta è derivata dalla classe Persona. Questo significa che ogni oggetto della classe NonLaureato è anche un oggetto di tipo Studente, ma anche un oggetto di tipo Persona. In questo modo, tutto ciò che funziona per gli oggetti della classe Persona funziona anche per gli oggetti della classe NonLaureato.



Quindi, quando si invocano metodi che richiedono come parametro una classe base, si possono passare anche oggetti di una classe derivata ma non il contrario.

Ad esempio la dichiarazione “`Persona p1 = new Studente ()`” è valida ma “`Studente s1 = new Persona ()`” no perché Persona è più restrittiva rispetto a Studente.

Quindi, prendendo lo schema qua accanto, si può capire come venga immaginata l'ereditarietà dei metodi.

Un NonLaureato potrà usare i metodi a, b e c ed essere definito come Studente o Persona ma Persona non potrà essere definito come Non Laureato in quanto la classe derivata contiene attributi in più.

10.3.5 La classe Object

Java ha una classe antenata di ogni classe. In Java, ogni classe deriva dalla classe Object. Ogni oggetto di ogni classe è anche di tipo Object.

La classe Object consente di scrivere metodi Java che hanno parametri di tipo Object.

La classe Object ha alcuni metodi che sono ereditati da ogni classe Java. Per esempio, ogni classe eredita dalle classi antenate o direttamente dalla classe Object i metodi `equals` e `toString`. Tuttavia questi metodi non possono funzionare correttamente per tutte le classi perché sono troppo generici.

`toString` è necessario ridefinirlo ogni volta in quanto non è esaustivo.

Un altro metodo ereditato dalla classe Object è il metodo clone. Questo metodo non ha argomenti e restituisce una copia dell'oggetto chiamante. Un clone è un oggetto che ha dati identici a quelli dell'oggetto che ha invocato il metodo. Anche questo metodo deve essere ridefinito perché possa funzionare correttamente nella classe derivata.

```

1  public class Studente extends Persona {
2      private int matricola;
3
4      public Studente() {
5          super(); //non è ancora stato visto
6          matricola = 0; //Ancora nessuna matricola
7      }
8
9      public Studente(String nomeIniziale, int matricolaIniziale) {
10         super(nomeIniziale);
11         matricola = matricolaIniziale;
12     }
13
14     public void reimposta(String nuovoNome, int nuovaMatricola) {
15         setNome(nuovoNome);
16         matricola = nuovaMatricola;
17     }
18
19     public int getMatricola() {
20         return matricola;
21     }
22
23     public void setMatricola(int nuovaMatricola) {
24         matricola = nuovaMatricola;
25     }
26
27     public void scriviOutput() {
28         System.out.println("Nome: " + getNome());
29         System.out.println("Matricola: " + matricola);
30     }
31
32     public boolean equals(Studente altroStudente) {
33         return this.haLoStessoNome(altroStudente) &&
34         (this.matricola ==
35         altroStudente.matricola);
36     }
37 }

```

Studente.

Questo primo tentativo porta il metodo `equals` a funzionare con qualsiasi oggetto, anche non `Studente`. Si dovrebbe modificare la definizione in modo che accetti un qualsiasi oggetto, ma se l'oggetto non è uno `Studente`, si restituirà semplicemente `false`. Ma come si può affermare che il parametro non è di tipo `Studente`? **Si può usare l'operatore `instanceof` per verificare se un oggetto è di tipo `Studente`. La sintassi è “oggetto `instanceof nome_della_classe`”.**

```

1  public boolean equals(Object otherObject) {
2
3      boolean uguale = false;
4
5      if ((otherObject != null) && (otherObject instanceof Studente)) {
6          Studente2 altroStudente = (Studente2)otherObject;
7          uguale = this.haLoStessoNome(altroStudente) && (this.matricola ==
8              altroStudente.matricola);
9      }
10
11     return uguale;
12 }

```

il metodo `equals` qua sopra è stato ridefinito in modo da tenere conto di queste differenze.

10.4 VISIBILITÀ

Infine ricordiamo la visibilità tra package e classi.

Modificatore d'accesso	Java	Simbolo UML	A chi è visibile
Pubblico	Public	+	A tutti
Default	Non viene segnato	~	A tutte le classi del package
Protetto	Protected	#	Attribuibile solo ai metodi e alle variabili interni alla classe. Visibile all'interno del package e classi derivate ovunque
Privato	Private	-	Più stretto di tutti, riferimenti ammessi solo alle sottoclassi appartenenti alla medesima classe

10.3.6 Un metodo `equals` migliorato

Nel listato qua accanto si nota come il metodo `equals` effettua solo un overloading rispetto a quello della classe `Object` in quanto hanno dei parametri in entrata di tipo differente.

In alcuni casi è fondamentale aver definito una reale ridefinizione (overriding) del metodo `equals`.

Infatti se utilizzassimo il metodo `equals` con un parametro `Object` e un parametro `Studente` in entrata, verrebbe utilizzato il metodo della classe `Object`.

Per risolvere questo problema è necessario cambiare da `Studente` a `Object` il tipo del parametro del metodo `equals` nella classe

11 POLIMORFISMO, CLASSI ASTRATTE E INTERFACCE

Con il termine polimorfismo si intende la possibilità di associare più significati a un nome di metodo per mezzo di un meccanismo conosciuto come binding dinamico.

11.1 POLIMORFISMO

11.1.1 Binding dinamico

Si immagini di dover progettare un insieme di classi che rappresentano diverse tipologie di figure geometriche: rettangoli, cerchi....

```

1  public class Figura{
2
3      public void centra(){
4          //istruzioni per spostare la figura
5          visualizza();
6      }
7
8      public void visualizza(){
9          //implementazione vuota: in Figura non
10         //si sa come implementarlo
11         //poichè dipende dalla figura specifica
12     }
13 }
14
15 public class Rettangolo extends Figura{
16     private double altezza;
17     private double larghezza;
18     private double puntoCentraleX;
19     private double puntoCentraleY;
20
21     public void visualizza(){
22         //istruzioni per visualizzare il rettangolo
23     }
24 }
```

contenere un metodo `centra()` che sposta una figura al centro dello schermo, cancellandola dalla posizione corrente e ridisegnandola al centro. Questo metodo può utilizzare a sua volta il metodo `visualizza()`.

Quando si pensa a come usare il metodo `centra()`, ereditato da `Figura`, per le classi `Rettangolo` e `Cerchio` emergono delle complicazioni.

Il metodo `centra()` utilizza il metodo `visualizza()` che è implementato in tutte le classi e noi vogliamo che utilizzi quello specifico di ogni figura, in modo che quando si invoca, ad esempio `r.centra()` il `visualizza` utilizzato sia quello di `Rettangolo` e non di `Figura`. Questa cosa accade già automaticamente in Java. Questo meccanismo si chiama “binding dinamico”.

Il binding indica il processo con cui l'invocazione di un metodo viene associata a una definizione specifica del metodo. Si definisce binding statico l'associazione tra invocazione e definizione di metodo che viene prodotta al momento della compilazione del codice. Se l'associazione tra invocazione e definizione di metodo viene prodotta quando il metodo è invocato (a run-time) si parla di binding dinamico. Java utilizza il binding dinamico per tutti i metodi fatta eccezione per pochi casi che vedremo poi.

Quindi, il binding dinamico permette il polimorfismo, ovvero la capacità di assegnare più significati a uno stesso nome di metodo, sfruttando il binding dinamico.

Ogni figura può essere un oggetto classe differente con attributi diversi. Idealmente tutte queste classi sono derivate dalla classe `Figura`. Si supponga di dover definire un metodo `visualizza()`. Per visualizzare un cerchio serve sicuramente una procedura differente rispetto a quella richiesta per un rettangolo. Scrivendo un metodo `visualizza` in tutte le classi avremo un comportamento diverso per `r.visualizza()` e `c.visualizza()` (dove `r` ed `c` sono rispettivamente `rettangolo` e `cerchio`).

La classe base `Figura` può avere dei metodi utilizzabili da tutte le figure. Per esempio potrebbe

Classe Vendita

```


/**
La classe rappresenta la vendita di un singolo elemento.
La classe ignora tasse, sconti e qualsiasi altro aggiustamento del prezzo.
Il prezzo assume valori non negativi; il nome è una stringa non vuota
*/
public class Vendita {
    private String nome; //Una stringa non vuota
    private double prezzo; //non negativo

    public Vendita() {
        nome = "Nessun nome";
        prezzo = 0;
    }
    /**
Precondizione: ilNome è una stringa non vuota; ilPrezzo è non negativo.
*/
    public Vendita(String ilNome, double ilPrezzo) {
        setNome(ilNome);
        setPrezzo(ilPrezzo);
    }

    public Vendita(Vendita oggettoOriginale) {
        if(oggettoOriginale==null) {
            System.out.println("Errore: oggetto Vendita null.");
            System.exit(0);
        } //else
        nome = oggettoOriginale.nome;
        prezzo = oggettoOriginale.prezzo;
    }

    public static void annuncio() {
        System.out.println("Questa e' la classe Vendita.");
    }

    public double getPrezzo() {
        return prezzo;
    }
    /**
Precondizione: nuovoPrezzo è non negativo.
*/
    public void setPrezzo(double nuovoPrezzo) {
        if(nuovoPrezzo >= 0)
            prezzo = nuovoPrezzo;
        else {
            System.out.println("Errore: Prezzo negativo.");
            System.exit(0);
        }
    }

    public String getNome() {
        return nome;
    }
    /**
Precondizione: nuovoNome è una stringa non vuota.
*/
    public void setNome(String nuovoNome) {
        if(nuovoNome != null && nuovoNome != "")
            nome = nuovoNome;
        else {
            System.out.println("Errore: nome errato.");
            System.exit(0);
        }
    }

    public String toString() {
        return ("Componente = " + nome +
               ", Prezzo e costo totale = " + prezzo);
    }

    public double totale() {
        return prezzo;
    }
    /**
Restituisce true se i nomi ed i totali delle vendite sono gli stessi;
altrimenti restituisce false.


```

```

Il metodo restituisce false anche se altraVendita è null.
*/
public boolean uguaglianzaVendite(Vendita altraVendita) {
    if(altraVendita == null)
        return false;
    else
        return (nome.equals(altraVendita.nome) &&
                totale() == altraVendita.totale());
}
/** Restituisce true se il totale dell'oggetto è minore del totale
dell'oggetto altraVendita; altrimenti restituisce false.
*/
public boolean minoreDi(Vendita altraVendita) {
    if(altraVendita == null) {
        System.out.println("Errore: oggetto Vendita è null.");
        System.exit(0);
    }
    //else
    return (totale() < altraVendita.totale());
}

public boolean equals(Object altroOggetto) {
    if(altroOggetto == null)
        return false;
    else if(!(altroOggetto instanceof Vendita))
        return false;
    else {
        Vendita altraVendita = (Vendita)altroOggetto;
        return (nome.equals(altraVendita.nome) &&
                (prezzo == altraVendita.prezzo));
    }
}

public void mostraAnnuncio(){
    annuncio();
    System.out.println(toString());
}
}

```

Classe VenditaScontata extends Vendita

```

Classe per la vendita di un componente con lo sconto espresso
come una percentuale del prezzo, ma senza altri aggiustamenti.
Il prezzo e lo sconto assumono valori non negativi;
il nome è una stringa non vuota.
*/
public class VenditaScontata extends Vendita {

    private double sconto; //Una percentuale del prezzo.
                           //Non può essere negativa.

    public VenditaScontata() {
        super();
        sconto = 0;
    }

    /**
     * Precondizione: ilNome è una stringa non vuota; ilPrezzo è non negativo;
     * loSconto è espresso come una percentuale del prezzo ed è non negativo.
     */
    public VenditaScontata(String ilNome, double ilPrezzo, double loSconto) {
        super(ilNome, ilPrezzo);
        setSconto(loSconto);
    }

    public VenditaScontata(VenditaScontata oggettoOriginale) {
        super(oggettoOriginale);
        sconto = oggettoOriginale.sconto;
    }

    public static void annuncio() {
        System.out.println("Questa e' la classe VenditaScontata.");
    }

    public double totale() {
        double frazione = sconto / 100;
        return (1 - frazione) * getPrezzo();
    }

    public double getSconto() {
        return sconto;
    }

    /**
     * Precondizione: nuovoSconto è non negativo.
     */
    public void setSconto(double nuovoSconto) {
        if (nuovoSconto >= 0)
            sconto = nuovoSconto;
        else {
            System.out.println("Errore: sconto negativo.");
            System.exit(0);
        }
    }

    public String toString() {
        return ("Componente = " + getNome() +
               ", Prezzo = " + getPrezzo() +
               " Sconto = " + sconto + "%\n" +
               "Costo totale = " + totale());
    }

    public boolean equals(Object altroOggetto) {
        if (altroOggetto == null)
            return false;
        else if (!(altroOggetto instanceof VenditaScontata))
            return false;
        else {
            VenditaScontata altraVenditaScontata =
            (VenditaScontata) altroOggetto;
            return (super.equals(altraVenditaScontata) && sconto ==
                   altraVenditaScontata.sconto);
        }
    }
}

```

BindingDinamico

```

public class BindingDinamicoDemo {

    public static void main(String[] args) {
        Vendita semplice = new Vendita("tappetino auto", 10.00); //un prodotto a E10.00.
        VenditaScontata scontato = new VenditaScontata("tappetino auto", 11.00, 10);
        //un prodotto a E11.00 con il 10% di sconto.

        System.out.println(semplice.toString());
        System.out.println(scontato.toString());

        if (scontato.minoreDi(semplice))
            System.out.println("Il prodotto scontato costa meno.");
        else
            System.out.println("Il prodotto scontato non costa meno.");

        Vendita prezzoNormale = new Vendita("portabici", 9.90); //un prodotto a E9.90.
        VenditaScontata prezzoSpeciale = new VenditaScontata("portabici", 11.00, 10);
        //un prodotto a E11.00 con il 10% di sconto.

        System.out.println(prezzoNormale.toString());
        System.out.println(prezzoSpeciale.toString());

        if (prezzoSpeciale.uguaglianzaVendite(prezzoNormale))
            System.out.println("Il costo totale e' lo stesso.");
        else
            System.out.println("Il costo totale e' diverso.");
    }
}

Componente = tappetino auto, Prezzo e costo totale = E10.0
Componente = tappetino auto, Prezzo = E11.0 Sconto = 10.0%
Costo totale = E9.9
Il prodotto scontato costa meno.
Componente = portabici, Prezzo e costo totale = E9.9
Componente = portabici, Prezzo = E11.0 Sconto = 10.0%
Costo totale = E9.9
Il costo totale e' lo stesso.

```

11.1.2 Binding dinamico con `toString`

Poniamo caso che nella classe Vendita con oggetto unaVendita venga ridefinito un metodo `toString` (presente anche nella classe Object). L'invocazione

`System.out.println(unaVendita.toString())` equivale a

`System.out.println(unaVendita)`. Questo perché? L'oggetto `System.out` ha numerose definizioni di `println` tra le quali una che accetta in ingresso un parametro di tipo `Object`. Quindi, idealmente, l'invocazione `System.out.println(unaVendita)` dovrebbe invocare il metodo `toString()` della classe `Object` ma questo non accade in quanto il metodo `println` è in overloading. Riconosce l'oggetto `unaVendita` sia come di classe `Object` che come di classe `Vendita` e quindi, tramite il binding dinamico, l'esecuzione prediligerà l'utilizzo del metodo ridefinito.

11.1.3 Il modificatore `final`

È possibile specificare che un metodo non può essere ridefinito (overriding) nelle classi derivate. Per fare questo basta aggiungere `final` alla dichiarazione del metodo. Un'intera classe può essere definita come `final`, in questo modo non possono essere usate come classi base.

Se un metodo è dichiarato come `final` il compilatore può utilizzare il binding statico per migliorare l'efficienza dell'applicazione ma il miglioramento è così irrisono da poter essere tralasciato per quanto riguarda le classi e il consiglio è di utilizzarlo per i metodi nel caso si voglia migliorare la sicurezza dell'applicazione.

11.1.4 Metodi per cui il binding dinamico non viene applicato

Java non utilizza il binding dinamico per i metodi privati, i metodi final e i metodi statici. Nei primi due casi l'assenza del binding dinamico non è un limite. Per i metodi statici, invece, può essere un problema in quanto viene invocato spesso utilizzando un oggetto chiamante.

In questi casi viene utilizzato il binding statico, la scelta del metodo da applicare dipende dunque dall'oggetto chiamante.

```

BindingStatico
public class BindingStaticoDemo {

    public static void main(String[] args) {
        Vendita.annuncio();
        VenditaScontata.annuncio();
        System.out.println("Questo esempio mostra che e' possibile fare
l'overriding di un metodo statico.");

        Vendita vendita = new Vendita();
        VenditaScontata scontata = new VenditaScontata();
        vendita.annuncio();
        scontata.annuncio();
        System.out.println("Nessun sorpresa finora, ma...");

        Vendita scontata2 = scontata; // (scontata e scontata2 fanno riferimento allo stesso
oggetto, ma una variabile e' di tipo Vendita e l'altra e' di tipo VenditaScontata)
        System.out.println("scontata2 e' di tipo VenditaScontata ma e' \nreferenziata da una
variabile di tipo Vendita.");
        System.out.println("Quale definizione di annuncio() sara' usata?");
        scontata2.annuncio();
        System.out.println("usa la versione di annuncio() definita in Vendita!");
    }
}

Questa e' la classe Vendita.
Questa e' la classe VenditaScontata.
Questo esempio mostra che e' possibile fare l'overriding di un metodo statico.
Questa e' la classe Vendita.
Questa e' la classe VenditaScontata.
Nessun sorpresa finora, ma...
scontata2 e' di tipo VenditaScontata ma e'
referenziata da una variabile di tipo Vendita.
Quale definizione di annuncio() sara' usata?
Questa e' la classe Vendita.
//Se Java avesse usato il binding dinamico con i metodi statici, questa riga sarebbe stata
//prodotta dal metodo annuncio della classe VenditaScontata anziché dal metodo annuncio
//della classe Vendita
usa la versione di annuncio() definita in Vendita!

```

11.1.5 Downcast e upcast

```
Vendita variabileVendita;
VenditaScontata variabileScontata =
    new VenditaScontata("verniciatura", 15, 10);
variabileVendita = variabileScontata;
System.out.println(variabileVendita.toString());
```

della classe VenditaScontata.

Questa dichiarazione è valida in quanto un oggetto VenditaScontata è anche un oggetto Vendita.

Tramite il binding dinamico sappiamo che variabileVendita.toString() utilizzerà il metodo

Assegnare un oggetto di una classe derivata a una variabile del tipo della classe base è spesso chiamato “upcast” perché è come fare la conversione dal tipo derivato al tipo della classe base.

La conversione da una classe base a una classe derivata viene chiamata “downcast”.

L'upcast non crea problemi mentre il downcast è più complesso perché, innanzitutto, non ha sempre senso.

Ad esempio `variabileScontata = (VenditaScontata) variabileVendita` darà un errore a run-time mentre `variabileScontata = variabileVendita` darà direttamente un errore durante la compilazione.

Ogni tanto però il downcast è necessario. Per esempio è inevitabile quando viene definito il metodo `equals` di una classe.

```
Vendita altraVendita = (Vendita) altroOggetto;
    return
        (nome.equals(altraVendita.nome) &&
        (prezzo == altraVendita.prezzo));
```

Ad esempio in questo estratto di codice estratto dal metodo `equals` di `Vendita`, si vede un downcast da `Object` a `Vendita` poiché, senza questo downcast, le variabili di istanza `nome` e `prezzo`

sarebbero utilizzate illegalmente nell'istruzione di `return`.

11.2 CLASSI ASTRATTE

Una classe astratta è una classe che ha alcuni metodi non completamente definiti. Non è possibile creare nuovi oggetti utilizzando il costruttore di una classe astratta ma è possibile utilizzare una classe astratta come classe base per la definizione di classi derivate.

11.2.1 Concetti di base

Classe FormaGenerica

```
/*
Una classe per disegnare semplici forme sullo schermo utilizzando solo caratteri.
Questa classe disegnerà un asterisco sullo schermo come prova.
Non si intende creare una forma "reale", questa classe è stata pensata
per essere usata come classe base per altre classi di forme.
*/
public class FormaGenerica {

    private int scostamento;

    public FormaGenerica() {
        scostamento = 0;
    }

    public FormaGenerica(int scostamentoIniziale) {
        scostamento = scostamentoIniziale;
    }

    public void setScostamento(int nuovoScostamento) {
        scostamento = nuovoScostamento;
    }

    public int getScostamento() {
        return scostamento;
    }

    public void disegnaDa(int numeroLinee) {
        for(int conteggio=0; conteggio<numeroLinee; conteggio++)
            System.out.println();
        disegnaQui();
    }

    public void disegnaQui() {
        for(int conteggio=0; conteggio<scostamento; conteggio++)
            System.out.print(' ');
        System.out.println('*');
    }

    // Scrive il numero indicato di spazi.
    protected static void saltaSpazi(int numero) {
        for(int conteggio=0; conteggio<numero; conteggio++)
            System.out.print(' ');
    }
}
```

La classe

FormaGenerica non è stata creata con l'intenzione di creare oggetti ma solo di fornire un supporto come classe base ad altre classi.

Possiamo vedere che è stato creato il metodo `disegnaQui()` ma la giusta domanda da porsi è “com’è la forma di una forma generica?”. Dipende dalla figura.

Il metodo `disegnaQui()` è stato creato solo per sfruttare il polimorfismo, infatti in questo modo le varie classi hanno dovuto ridefinire solo il metodo `disegnaQui()`.

Tuttavia, invece di fornire una definizione inventata di un metodo che si pensa di

ridenominare in una classe derivata, si può dichiarare il metodo astratto facendo precedere all'intestazione il termine `abstract` e terminando la riga con un `;`, quindi ad esempio “`public abstract void`

disegnaQui () ;” . Definire un metodo astratto significa posticipare la sua definizione al momento in cui si saprà effettivamente come definirla.

Java richiede **che se una classe ha almeno un metodo astratto, la classe deve essere dichiarata astratta** includendo abstract nella definizione della classe.

Classe FormaGenerica (abstract)

```
public class abstract FormaGenerica {
    [...]
    public void disegnaDa (int numeroLinee) {
        for(int conteggio=0; conteggio<numeroLinee; conteggio++)
            System.out.println();
        disegnaQui();
    }

    public void abstract disegnaQui();
    [...]
}
```

Una classe definita in questo modo è detta “classe astratta” e quindi, per converso, una classe non astratta è detta “classe concreta”.

La definizione di metodi astratti è utile per non dover definire più volte lo stesso metodo nelle varie classi.

11.2.2 La classe astratta è un tipo

Non è possibile creare un oggetto da una classe astratta, ma ha perfettamente senso avere una variabile del tipo di una classe astratta alla quale può essere assegnato un oggetto di una qualsiasi delle classi discendenti.

11.2.3 Ulteriori dettagli

Una classe astratta può avere un numero qualsiasi di metodi astratti oltre a eventuali metodi non astratti. Se una classe derivata non è in grado di definire uno o più metodi della classe base astratta, anche lei sarà una classe astratta e dovrà includere nella propria definizione abstract.

11.3 INTERFACCE

Ricordiamo dal capitolo 8 che l’interfaccia è costituita dalle intestazioni dei metodi pubblici e dalle costanti pubbliche della classe, insieme con alcuni commenti esplicativi. Conoscendo solo l’interfaccia di una classe, un programmatore può scrivere un programma che utilizza la classe senza sapere nulla dell’implementazione della classe stessa.

L’interfaccia e l’implementazione possono essere separate, realizzando due file sorgenti distinti.

11.3.1 Interfacce Java

Un’interfaccia Java è una componente di un programma che contiene le intestazioni di un certo numero di metodi e/o delle costanti pubbliche, assieme ad un’eventuale descrizione dei metodi per permettere di comprendere come implementarli.

```
1  /**
2  * Un'interfaccia che contiene metodi che calcolano
3  * e restituiscono il perimetro e l'area di un oggetto
4  */
5  public interface Misurabile {
6
7      /**
8      * Restituisce il perimetro.
9      */
10     public double getPerimetro();
11
12     /**
13     * Restituisce l'area.
14     */
15     public double getArea();
16 }
```

Un’interfaccia Java inizia come una definizione di classe, tranne per il fatto che class è sostituito da interface.

L’interfaccia può contenere un numero qualsiasi di intestazioni di metodi pubblici, ognuna seguita da un punto e virgola.

Per convenzione il nome di un’interfaccia inizia con una lettera maiuscola.

Un’interfaccia non dichiara alcun costruttore. I metodi di un’interfaccia devono essere pubblici.

Un’interfaccia può anche dichiarare un numero qualsiasi di costanti pubbliche. Non contiene variabili di istanza o definizioni complete di metodo.

11.3.2 Implementare un’interfaccia

Quando si scrive una classe che definisce i metodi dichiarati in un’interfaccia, si dice che la classe “implementa” l’interfaccia. La classe deve definire tutti i metodi dichiarati nell’interfaccia, altrimenti deve essere dichiarata come astratta. La classe potrebbe dichiarare anche metodi aggiuntivi.

```

1  /**
2  * Una classe che rappresenta quadrati
3  */
4  public class Quadrato implements Misurabile {
5      private double lato;
6
7      public Quadrato(double ilLato) {
8          super();
9          this.lato = ilLato;
10     }
11
12     public double getArea() {
13         return lato*lato;
14     }
15
16     public double getPerimetro() {
17         return lato*4;
18     }
19 }
```

interfaccia.

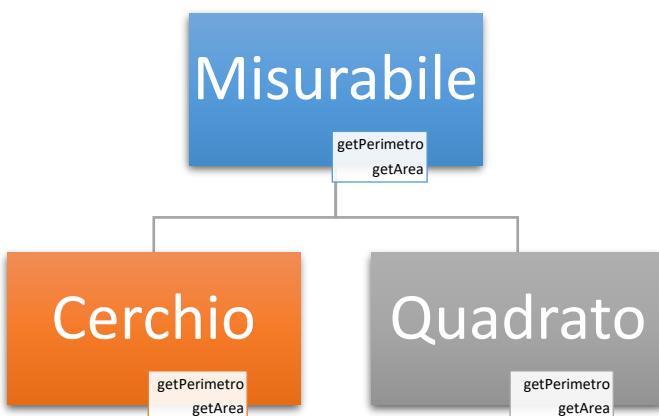
Per implementare un’interfaccia si devono fare due cose:

- **Includere l’espressione `implements nome_interfaccia1, nome_interfaccia2, ...` all’inizio della definizione di classe.** Si possono implementare anche più interfacce
- **Definire ogni metodo dichiarato nell’interfaccia (o nelle interfacce) per creare una classe concreta,** altrimenti sarà una classe astratta

Classi differenti possono implementare la stessa

11.3.3 Un’interfaccia come un tipo

Un’interfaccia è un tipo riferimento. Così si può scrivere un metodo che ha un parametro di un tipo interfaccia. Il programma può invocare un metodo passandogli un oggetto di una qualsiasi classe che implementi l’interfaccia Misurabile.



```

Misurabile scatola = new Quadrato(5.0);
Misurabile disco = new Cerchio(5.0);

visualizza(scatola);
//Perimetro = 20.0, area = 25.0

visualizza(disco);
//Perimetro = 31.4, area = 78.5
  
```

Anche se il tipo di entrambe le variabili è Misurabile, gli oggetti si riferiscono a scatola e disco hanno i metodi getPerimetro e getArea definiti diversamente. Quindi la chiamata del metodo visualizza () utilizza i metodi propri degli

oggetti e non di Misurabile.

Il binding dinamico e il polimorfismo si applicano anche alle interfacce.

11.3.4 Estendere un’interfaccia

È possibile definire una nuova interfaccia che estende (extends) un’interfaccia già esistente. In questo modo si può creare un’interfaccia formata da un insieme di metodi: quelli da lei definiti e quelli ereditati.

12 ARRAYLIST E GENERICI

12.1 STRUTTURE DI DATI BASATE SU ARRAY

In java la lunghezza di un array può essere letta anche durante l'esecuzione del programma, ma una volta che il programma crea un array di una certa lunghezza, questa non può essere cambiata. Per ovviare a questo limite si potrebbero spendere risorse creando un secondo array e poi copiando il contenuto del primo nel secondo.

12.1.1 *La classe ArrayList*

Un secondo modo per ovviare a questo limite è l'utilizzo di un'istanza della classe **ArrayList**, che si trova nel package **java.util**. tale istanza può offrire gli stessi servizi offerti da un array, tranne per il fatto che è in grado di cambiare la propria lunghezza durante l'esecuzione del programma.

Quindi perché studiare gli array se si possono usare le ArrayList?

- Un'istanza di ArrayList è meno efficiente di un array
- **Un'istanza di ArrayList può memorizzare solo oggetti**, non può contenere valori primitivi

In realtà l'utilizzo di una ArrayList significa nascondere lo stesso procedimento descritto prima, quindi richiederà un tempo di computazione maggiore, che in alcuni casi potrebbe anche essere significativo, di conseguenza sarebbe meglio valutare quando sia veramente necessario.

Un escamotage per evitare problemi con i dati primitivi è quello di utilizzare le classi wrapper, tuttavia l'utilizzo di queste aggiunge un ulteriore carico per la macchina.

12.1.2 *Create un'istanza di ArrayList*

Utilizzare un'ArrayList implica alcune differenze:

- La classe non è fornita automaticamente, **bisogna includerla tramite l'istruzione "import java.util.ArrayList"**
- Si crea e nomina un'istanza di ArrayList nello stesso modo con cui si crea e si nomina un oggetto di una qualsiasi classe, ad eccezione del fatto che occorre specificare il tipo base, ad esempio:
 - `ArrayList<String> lista = new ArrayList<String>(20)`
- Il tipo memorizzato in un'ArrayList deve essere necessariamente un oggetto e non un tipo primitivo

L'ArrayList dell'esempio ha allocato come valore iniziale uno spazio di memoria per 20 elementi ma nel caso ne serva di più l'allicherà dinamicamente dopo. Nel caso la memoria allocata inizialmente sia sufficiente, l'utilizzo dell'ArrayList sarà rapido ma nel caso sia in eccesso si sprecherà memoria. **Nel caso non venga specificata la grandezza verrà impostata di default a 10.**

12.1.3 *Utilizzare i metodi di ArrayList*

Un oggetto della classe ArrayList può essere utilizzato come un array, ma occorre utilizzare i suoi metodi al posto della notazione a parentesi quadre, tipica degli array.

Il metodo set può sostituire un qualsiasi elemento esistente, ma non può essere utilizzato per inserire un elemento in una posizione che non contiene ancora un elemento, come invece è possibile fare con un array. **Il metodo set sostituisce un valore, non lo crea.**

Per aggiungere bisogna utilizzare il metodo **add**. questo metodo ha due definizioni, in una si specifica l'indice, nell'altra solo l'elemento da aggiungere. Si possono aggiungere elementi tra due elementi già esistenti (l'ArrayList sposterà a destra gli elementi già esistenti), nella prima casella vuota ma non in una casella che non rientri tra queste due categorie, questo perché una Lista è ordinata.

Istruzione	Array	ArrayList
Creazione	String [] unArray = new String[20]	ArrayList<String> lista = new ArrayList<String>(20)
Set	unArray[indice] = "stringa"	lista.set(indice, "stringa")
Get	String temp = unArray[indice]	String temp = lista.get(indice)
Creazione valore	unArray[indice] = "stringa"	lista.add(indice, "stringa") lista.add("stringa")
Grandezza	unArray.length	lista.size()
Rimozione	unArray[indice] = null	lista.remove(indice) lista.remove("stringa") (rimuove solo la prima occorrenza dell'oggetto, non eventuali doppiioni e decrementa la dimensione della lista di 1. Nel caso indice<0 o indice ≥size() lancia un IndexOutOfBoundsException)
Cancellazione	-----	lista.clear()
Equals	for(i=0; i<unArray.length; i++) If(unArray[i].equals(elemento)) Contiene = true	lista.contains(elemento)
Indice	for(i=0; i<unArray.length; i++) If(unArray[i].equals(elemento)) indice = i	lista.indexOf(elemento) (restituisce solo la prima occorrenza dell'oggetto, non eventuali doppiioni, -1 se assente)
Se è vuota	for(i=0; i<unArray.length; i++) If(unArray[i] != null) Vuota = false	lista.isEmpty()

12.1.4 Foreach

Per mostrare tutti gli elementi di un'istanza di ArrayList si possono usare due metodi:

```

for
for(i=0; i<unArray.length; i++)
    If(unArray[i].equals(elemento))
        System.out.println(lista.get(i))
foreach
for(String elemento: lista)
    System.out.println(elemento)

```

12.1.5 *trimToSize*

Gli ArrayList raddoppiano automaticamente la propria capacità quando il programma richiede loro di crescere, tuttavia questa capacità può essere eccessiva. Si può risparmiare spazio utilizzando il metodo **trimToSize()** per ridurre la grandezza dell'ArrayList a quella minima necessaria. È consigliabile utilizzare questo metodo solo quando si è certi che non sarà necessario incrementare nuovamente la grandezza della lista a breve.

12.1.6 *Clone*

Non è possibile copiare un'ArrayList tramite il solo operatore di assegnamento poiché ciò porterebbe solo al cambio di riferimento nello stack cos' come abbiamo visto con qualsiasi altro oggetto.

Per copiare un'ArrayList bisogna utilizzare l'istruzione `clone()`:

```
ArrayList<[tipo Oggetto]> [nome seconda lista] =  
(ArrayList<[tipo Oggetto]>) [nome vecchia lista].clone()
```

Per gli oggetti diversi dalle stringhe il metodo `clone` risulta essere più complicato.

12.1.7 *Classi parametriche e tipi di dato generico*

La classe ArrayList è una classe parametrica, ciò vuol dire che ha un parametro che può essere sostituito con un qualunque tipo classe per ottenere una classe che memorizza oggetti di quel tipo. Il tipo classe è anche detto tipo di dato generico.

12.2 GENERICI

12.2.1 *Fondamenti*

Le classi e i metodi possono utilizzare un tipo parametrico al posto di uno specifico tipo di dato. Serve molta prudenza nell'utilizzo dei generici.

```
1  public class Esempio<T> {  
2      private T dati;  
3  
4      public void setDati(T nuovoValore) {  
5          dati = nuovoValore;  
6      }  
7  
8      public T getDati() {  
9          return dati;  
10     }  
11 }
```

- Non si possono creare array
- Non si possono usare in tutti i contesti nei quali è prescritta la parola `new`

Il listato qua accanto mostra una definizione di classe molto semplice che utilizza un tipo parametrico `T`. Per convenzione i tipi parametrici usano una lettera maiuscola come inizio.

Fondamentalmente un tipo parametrico si adatta al tipo di ingresso. Ci sono dei vincoli:

Si può creare un oggetto di una classe parametrica nello stesso modo in cui si crea un oggetto di qualsiasi altra classe, tranne per il fatto che si deve specificare un tipo classe attuale (al posto del tipo parametrico) all'interno delle parentesi angolari.

Non si può sostituire un tipo parametrico con un tipo primitivo.

Si possono anche usare tipi parametrici multipli.

```

1  public class CoppiaADueTipi<T1, T2> {
2      private T1 primo;
3      private T2 secondo;
4
5      public CoppiaADueTipi() {
6          primo = null;
7          secondo = null;
8      }
9
10     public CoppiaADueTipi(T1 primoElemento, T2 secondoElemento) {
11         primo = primoElemento;
12         secondo = secondoElemento;
13     }
14
15     public void setPrimo(T1 nuovoPrimo) {
16         primo = nuovoPrimo;
17     }
18
19     public void setSecondo(T2 nuovoSecondo) {
20         secondo = nuovoSecondo;
21     }
22
23     public T1 getPrimo() {
24         return primo;
25     }
26
27     public T2 getSecondo() {
28         return secondo;
29     }
30
31     public String toString() {
32         return ("primo: " + primo.toString() + "\n"
33                 + "secondo: " + secondo.toString());
34     }
35
36     public boolean equals(Object altroOggetto) {
37         if (altroOggetto == null)
38             return false;
39         else if (getClass() != altroOggetto.getClass())
40             return false;
41         else {
42             CoppiaADueTipi<T1, T2> altraCoppia =
43                 (CoppiaADueTipi<T1, T2>) altroOggetto;
44
45             return (primo.equals(altraCoppia.primo)
46                     && secondo.equals(altraCoppia.secondo));
47         }
48     }
49 }
1  import java.util.Scanner;
2
3  public class CoppiaADueTipiDemo {
4      public static void main(String[] args) {
5          CoppiaADueTipi<String, Integer> giudizio =
6              new CoppiaADueTipi<String, Integer>("The Car Guys", 8);
7          Scanner tastiera = new Scanner(System.in);
8          System.out.println(
9                  "Il nostro voto attuale per " + giudizio.getPrimo());
10         System.out.println(" e' " + giudizio.getSecondo());
11         System.out.println("Tu che voto daresti?");
12         int punteggio = tastiera.nextInt();
13         giudizio.setSecondo(punteggio);
14         System.out.println(
15                 "Il nostro nuovo voto per " + giudizio.getPrimo());
16         System.out.println(" e' " + giudizio.getSecondo());
17     }
18 }
```

12.2.2 Vincoli su tipi parametrici

A volte non ha senso sostituire un tipo di riferimento qualunque a un tipo parametrico. Ad esempio il metodo `compareTo` è richiesto in ogni classe che implementi l'interfaccia `Comparable`. È un'interfaccia della libreria standard e richiede la presenza del seguente metodo: `public int compareTo(Object altro)`. È utile ricordare che il metodo deve restituire:

- < 0 se l'oggetto chiamato viene prima

- 0 se sono uguali
- > 0 se l'oggetto chiamato viene dopo

C'è un problema: tutto questo funziona solo se il tipo sostituito al parametro T rispetta l'interfaccia Comparable. In java è possibile esprimere restrizioni sui possibili tipi sostituibili a un tipo parametrico. Per garantire che le classi sostituibili siano comparabili basta scrivere, ad esempio: `public class Coppia<T extends Comparable>`. Nel caso si provi a sostituire una classe non comparabile si otterrà un errore in compilazione. Notare che si deve usare `extends` e non `implements`. Il vincolo è possibile con qualsiasi altra classe, anche una creata dall'utente. Si possono estendere più classi e interfacce tramite la congiunzione `&`, ad esempio `public class Coppia<T extends Comparable & AltraClasse>`. Si possono estendere infinite interfacce ma al massimo una classe.

12.2.3 Metodi generici

Quando si definisce una classe generica, si possono utilizzare i tipi parametrici nelle definizioni dei suoi metodi. Si possono anche definire metodi generici che abbiano i propri tipi parametrici, diversi da quelli della classe. Ad esempio scrivendo `public static<T> T getPrimo(T[] a)` in una classe non parametrica si crea un metodo che accetta un tipo parametrico. Sarà accortezza poi scrivere il tipo nella chiamata, ad esempio `MetodiUtili.<String>getPrimo(c)`.

12.2.4 Ereditarietà con classi generiche

Una classe generica può estendere una classe ordinaria o un'altra classe generica.

13 ECCEZIONI

Un modo per scrivere programmi consiste nel presupporre che durante l'esecuzione di un programma non si presentino situazioni anomale o eccezionali. Una volta che il programma funziona per il caso normale, si aggiunge il codice che gestisce i casi eccezionali.

Java fornisce gli strumenti necessari per supportare questa pratica operativa.

13.1 CONCETTI DI BASE SULLA GESTIONE DELLE ECCEZIONI

Java fornisce gli strumenti necessari per gestire alcuni tipi di anomalie. Questi strumenti permettono di dividere un programma o un metodo in sezioni distinte: quella che gestisce il normale funzionamento e quella che gestisce il caso eccezionale.

Un'eccezione è un oggetto che segnala l'accadere di un evento anomalo durante l'esecuzione di un

```

1 import java.util.Scanner;
2
3 public class PrendiLatte {
4
5     public static void main(String[] args) {
6
7         Scanner tastiera = new Scanner(System.in);
8
9         System.out.println("Inserire il numero di ciambelle:");
10        int conteggioCiambelle = tastiera.nextInt();
11
12        System.out.println("Inserire il numero di bicchieri di latte:");
13        int conteggioLatte = tastiera.nextInt();
14
15        //Gestione degli eventi eccezionali senza utilizzare le strutture
16        //di gestione delle eccezioni di Java
17        if (conteggioLatte < 1) {
18            System.out.println("Niente latte!");
19            System.out.println("Vai a comprare del latte.");
20        } else {
21            double ciambellePerBicchiere = conteggioCiambelle / (double) conteggioLatte;
22            System.out.println(conteggioCiambelle + " ciambelle.");
23            System.out.println(conteggioLatte + " bicchieri di latte.");
24            System.out.println("Hai " + ciambellePerBicchiere + " ciambelle per ogni bicchiere di latte.");
25        }
26        System.out.println("Fine programma.");
27    }
28 }
```

programma. Il processo di creazione di questo oggetto è chiamato “lancio di un'eccezione”. Il codice che rileva e che si occupa dell'eccezione si dice che “gestisce l'eccezione”.

L'utilizzo delle eccezioni è comodo nel caso questa debba essere gestita in maniera differente a seconda del programma che usa quel metodo.

13.1.1 Eccezioni in Java

Qua un esempio di programma nel quale le eccezioni vengono gestite tramite un semplice if-else. Infatti se conteggioLatte fosse uguale a 0, la divisione non sarebbe valida.

Quindi:

- Caso normale: una persona non può finire il latte, quindi conteggioLatte>1
- Caso anomalo: conteggioLatte uguale a 0

Il secondo codice è una revisione del primo con l'introduzione delle parole try e catch che, per quanto più complesso, è più pulito.

L'istruzione if-else è stata sostituita da

```

if (conteggioLatte < 1)
    throw new
    Exception("Eccezione: Niente latte!");
```

```

1 import java.util.Scanner;
2
3 public class PrendiLatteConEccezioni {
4
5     public static void main(String[] args) {
6
7         Scanner tastiera = new Scanner(System.in);
8
9         try {
10            System.out.println("Inserire il numero di ciambelle:");
11            int conteggioCiambelle = tastiera.nextInt();
12
13            System.out.println("Inserire il numero di bicchieri di latte:");
14            int conteggioLatte = tastiera.nextInt();
15
16            if (conteggioLatte < 1)
17                throw new Exception("Eccezione: Niente latte!");
18
19            double ciambellePerBicchiere = conteggioCiambelle / (double) conteggioLatte;
20            System.out.println(conteggioCiambelle + " ciambelle.");
21            System.out.println(conteggioLatte + " bicchieri di latte.");
22            System.out.println("Hai " + ciambellePerBicchiere + " ciambelle per ogni bicchiere di latte.");
23        } catch(Exception e) {
24            System.out.println(e.getMessage());
25            System.out.println("Vai a comprare del latte.");
26        }
27
28        System.out.println("Fine programma.");
29    }
30 }
```

L'istruzione richiede che nel caso non vi sia latte, il programma compia qualcosa di eccezionale, specificato dalla parola `catch`. L'idea è che le situazioni normali siano gestite dal codice che segue la parola `try`, mentre il codice che segue la parola `catch` venga utilizzato solitamente in circostanze eccezionali.

La gestione di base delle eccezioni in Java avviene mediante l'utilizzo del gruppo di tre istruzioni: `try-throw-catch`.

Il blocco `try` viene chiamato così perché non vi è la sicurezza assoluta che ogni cosa vada come ci si aspetta.

Nel caso qualcosa vada storto viene lanciata un'eccezione per indicare che sono sorti dei problemi. Si aggiunge quindi un'istruzione `throw`. Se viene eseguita, l'istruzione `throw` crea un nuovo oggetto della classe predefinita `Exception`.

Blocco try

```
try{
    codice_da_provare
    eventuale_generazione_di_eccezione
        throw new Exception(Texto_eccezione);
    altro_codice
} catch (Exception parametro_del_blocco_catch) {
    istruzioni_da_eseguire
        System.out.println(e.getMessage());
}
```

Quando viene lanciata un'eccezione, l'esecuzione del blocco `try` viene arrestata e viene eseguita un'altra porzione di codice, chiamata blocco `catch`. Eseguire questo blocco è detto "catturare l'eccezione".

Il blocco `catch` assomiglia ad un metodo, è una porzione di codice distinta che viene eseguita se il programma esegue un'istruzione `throw` all'interno del blocco `try` precedente.

Ogni oggetto eccezione ha un metodo chiamato `getMessage` e, a meno che non lo si ridefinisca, questo restituisce la stringa fornita al costruttore come argomento in fase di creazione dell'oggetto eccezione.

Sono possibili altri tipi di eccezioni e un blocco `try` può essere seguito da più blocchi `catch`, sarà comunque invocato quello che corrisponde al tipo di eccezione generata. Anche se possibile, questo comportamento è sconsigliato, è meglio prevedere più blocchi `catch` specifici rispetto ad uno generale.

Un blocco `catch` è legato al solo blocco `try` precedente. Quando viene lanciata un'eccezione, il controllo passa all'istruzione `catch` appropriata, il resto del corpo del `try` viene sorpassato, quindi il controllo non torna mai al codice `try`.

Quando il blocco `try` non genera eccezioni, l'esecuzione prosegue con il codice che si trova dopo l'ultimo blocco `catch`.

13.1.2 Classi di eccezioni predefinite

Quando si inizia a utilizzare i metodi delle classi predefinite, ci si accorge che a volte questi possono lanciare certi tipi di eccezioni. Tali eccezioni appartengono alle librerie di base di Java. Se si utilizzano questi metodi, è possibile inserire le varie invocazioni in un blocco `try` e utilizzare il blocco `catch` per catturare la sua eventuale eccezione.

Il nome delle eccezioni predefinite è stato scelto in modo da chiarirne il significato, ad esempio:

- `BadStringOperationException`: operazione non consentita sulla stringa
- `ClassNotFoundException`: classe non trovata
- `IOException`: errori di input o in output
- `NoSuchMethodException`: metodo inesistente

Quando si cattura un'eccezione di una di queste classi predefinite, la stringa restituita dal metodo `getMessage` fornisce informazioni sufficienti per identificare la causa dell'eccezione.

La maggior parte delle eccezioni presentate in questo testo non hanno bisogno di essere importate. Alcune però sono contenute in package differenti, ad esempio la classe `IOException` si trova nel package `java.io`.

13.2 DEFINIRE NUOVE CLASSI DI ECCEZIONI

Si possono definire nuove classi di eccezioni, tuttavia devono essere classi derivate da classi di eccezioni preesistenti. Infatti gli esempi di questo paragrafo sono tutti classi derivate dalla classe `Exception`.

```

1 public class DivisionePerZeroException extends Exception {
2
3     public DivisionePerZeroException() {
4         super("Divisione per zero!");
5     }
6
7     public DivisionePerZeroException(String messaggio) {
8         super(messaggio);
9     }
10 }
11
12 import java.util.Scanner;
13
14 public class DividiPerZeroDemo {
15     private int numeratore;
16     private int denominatore;
17     private double quoziente;
18
19     public void fai() {
20         try {
21             System.out.println("Inserisci numeratore:");
22             Scanner tastiera = new Scanner(System.in);
23             numeratore = tastiera.nextInt();
24             System.out.println("Inserisci denominatore:");
25             denominatore = tastiera.nextInt();
26
27             if (denominatore == 0)
28                 throw new DivisionePerZeroException();
29
30             quoziente = numeratore / (double)denominatore;
31             System.out.println(numeratore + "/" + denominatore + " = " +
32                                 quoziente);
33
34         } catch(DivisionePerZeroException e) {
35             System.out.println(e.getMessage());
36             daiSecondaPossibilita();
37         }
38         System.out.println("Fine programma.");
39     }
40
41     public void daiSecondaPossibilita() {
42         System.out.println("Tenta di nuovo.");
43         System.out.println("Inserisci numeratore:");
44         Scanner tastiera = new Scanner(System.in);
45         numeratore = tastiera.nextInt();
46         System.out.println("Inserisci denominatore:");
47         System.out.println("Accertati che il denominatore non sia zero.");
48         denominatore = tastiera.nextInt();
49
50         if (denominatore == 0) {
51             System.out.println("Non posso dividere per zero.");
52             System.out.println("Poiche' non posso fare cio' che chiedi.");
53             System.out.println("il programma terminera' ora.");
54             System.exit(0);
55         }
56
57         quoziente = ((double)numeratore) / denominatore;
58         System.out.println(numeratore + "/" + denominatore +
59                             " = " + quoziente);
60     }
61
62     public static void main(String[] args) {
63         DividiPerZeroDemo unaVolta = new DividiPerZeroDemo();
64         unaVolta.fai();
65     }
66 }
```

classe di eccezione predefinita nella libreria Java Class, questo perché non sono abbastanza specifiche.

quando si definisce una classe di eccezione, i costruttori spesso costituiscono i metodi più importanti, se non addirittura gli unici, a parte quelli ereditati dalla classe base. Ad esempio nella classe accanto vengono ereditati due costruttori, uno che accetta come parametro una stringa e uno che agisce in maniera classica, ovvero scrivendo direttamente la stringa. Quindi, se non viene specificato nulla durante il `throw`, verrà restituito con `getMessage ()` "Divisione per zero!".

Nella seconda parte della demo si nota che viene eseguita per una seconda volta la divisione (viene invocato il metodo `daiSecondaPossibilità` dal `catch`) e, nel caso continui a ripetersi l'errore, è meglio gestire un caso eccezionale senza lanciare un'eccezione.

Le due caratteristiche più importanti di un oggetto eccezione sono:

- **Il tipo dell'oggetto**, ovvero il nome della classe di eccezione
- **Il messaggio che l'oggetto si porta dietro** memorizzato in una variabile di istanza di tipo `String`.

Se nel codice si andrà ad inserire un'istruzione `throw`, è buona norma definire una propria classe di eccezione in modo da poter differenziare tra le eccezioni personalizzate e le eccezioni generate dalle eventuali invocazioni di metodi definiti nelle classi predefinite.

La cosa da evitare è quella di utilizzare una

Quando si definisce una classe di eccezione si può tenere conto di alcune linee guida:

- Si utilizzi la classe `Exception` come base se non vi sono particolari esigenze
- Si definiscano due costruttori che includono uno di default e uno con un parametro di tipo `String`
- Si dovrebbe iniziare ogni definizione di costruttore con una chiamata al costruttore della classe base, utilizzando `super`
- `getMessage` non dovrebbe essere ridefinito
- normalmente non è necessario definire nessun altro metodo, anche se è lecito farlo

13.3 APPROFONDIMENTI SULLE CLASSI DI ECCEZIONI

13.3.1 Dichiарare le eccezioni

A volte è sensato ritardare la gestione di un'eccezione. Per esempio si potrebbe avere un metodo in cui il codice lancia un'eccezione, che però non cattura, ad esempio terminando in quel momento l'esecuzione. Un altro programma potrebbe invece essere in grado di gestire l'eccezione e di continuare l'esecuzione.

La gestione dell'eccezione dipende quindi dal programma che utilizza il metodo, di conseguenza, il metodo stesso non sarebbe in grado di gestire l'eccezione. In queste situazioni ha senso lanciare il metodo in un blocco `try` e non mettere il blocco `try` all'interno del metodo stesso.

```

1 import java.util.Scanner;
2
3 public class FaiDivisione {
4
5     private int numeratore;
6     private int denominatore;
7     private double quoziente;
8
9     public static void main(String[] args) {
10         FaiDivisione fai = new FaiDivisione();
11
12         try {
13             fai.casoNormale();
14         } catch(DivisionePerZeroException e) {
15             System.out.println(e.getMessage());
16             fai.daiSecondaPossibilita();
17         }
18         System.out.println("Fine Programma.");
19     }
20
21     public void casoNormale() throws DivisionePerZeroException {
22         System.out.println("Inserisci numeratore:");
23         Scanner tastiera = new Scanner(System.in);
24         numeratore = tastiera.nextInt();
25         System.out.println("Inserisci denominatore:");
26         denominatore = tastiera.nextInt();
27
28         if (denominatore == 0)
29             throw new DivisionePerZeroException();
30
31         quoziente = numeratore / (double)denominatore;
32         System.out.println(numeratore + "/" + denominatore + " = " +
33                             quoziente);
34     }
35
36     public void daiSecondaPossibilita() {
37         System.out.println("Tenta di nuovo.");
38         System.out.println("Inserisci numeratore:");
39         Scanner tastiera = new Scanner(System.in);
40         numeratore = tastiera.nextInt();
41         System.out.println("Inserisci denominatore:");
42         System.out.println("Accertati che il denominatore non sia zero.");
43         denominatore = tastiera.nextInt();
44
45         if (denominatore == 0) {
46             System.out.println("Non posso dividere per zero.");
47             System.out.println("Poiche' non posso fare cio' che chiedi.");
48             System.out.println("Il programma terminera' ora.");
49             System.exit(0);
50         }
51
52         quoziente = ((double)numeratore) / denominatore;
53         System.out.println(numeratore + "/" + denominatore + " = " +
54                             quoziente);
55     }
56 }
```

Se un metodo non cattura l'eccezione, deve almeno informare il programmatore che ogni sua invocazione potrebbe causare un'eccezione. Questa avvertenza è chiamata "clausola throws". Ad esempio si può lanciare `DivisionePerZeroException` senza che questa catturi l'eccezione, quindi ci sarà un'intestazione del tipo "public void metodoDiEsempio() throws DivisionePerZeroException". Questa riga di codice dichiara che un'invocazione al metodo può lanciare un'eccezione. La maggior parte delle eccezioni che possono essere lanciate tramite l'invocazione di un metodo devono essere gestite in uno di questi due modi:

- Si cattura la possibile eccezione in un blocco `catch` definito all'interno del metodo
- Si dichiara la possibile eccezione utilizzando la clausola `throws` nell'intestazione del metodo e si delega la gestione dell'eccezione a chi utilizza il metodo

Si possono innestare i lanci di eccezioni tramite l'invocazione di metodi, ad esempio A invoca B che invoca C e in entrambe le invocazioni viene specificato "throws Exception", in questo caso la gestione dell'eventuale eccezione sarà compito di A.

Inoltre una clausola throws può contenere più tipi di eccezioni, in questi casi vengono separati da una virgola come ad esempio "public void metodoDiEsempio () throws DivisionePerZeroException, IOException".

Attenzione:

- **throw**: utilizzato per lanciare un'eccezione nel **corpo di un metodo** o del main
- **throws**: utilizzato **nell'intestazione del metodo**

13.3.2 Tipi di eccezioni

Java ha alcune eccezioni che possono non essere trattate come tale, sebbene possano sempre essere catturate tramite catch.

Java suddivide le eccezioni in:

- **Eccezioni controllate (checked)**: deve essere catturata in un blocco catch oppure dichiarata in una clausola throws. Queste eccezioni spesso indicano la presenza di seri problemi che potrebbero portare alla terminazione del programma
- **Eccezioni non controllate (unchecked) o eccezione run-time**: può non essere catturata in un blocco catch o dichiarata in una clausola throws. **Queste eccezioni indicano che nel codice c'è qualcosa di sbagliato e che dovrebbe essere corretto.** Solitamente sono generate durante l'esecuzione di un'espressione o lanciate da un metodo presente in una delle classi predefinite. Per esempio se un programma tenta di utilizzare un indice che non rientra nei limiti di un array viene generata l'eccezione `ArrayIndexOutOfBoundsException`.

Un'eccezione a run-time non catturata, causa la terminazione del programma.

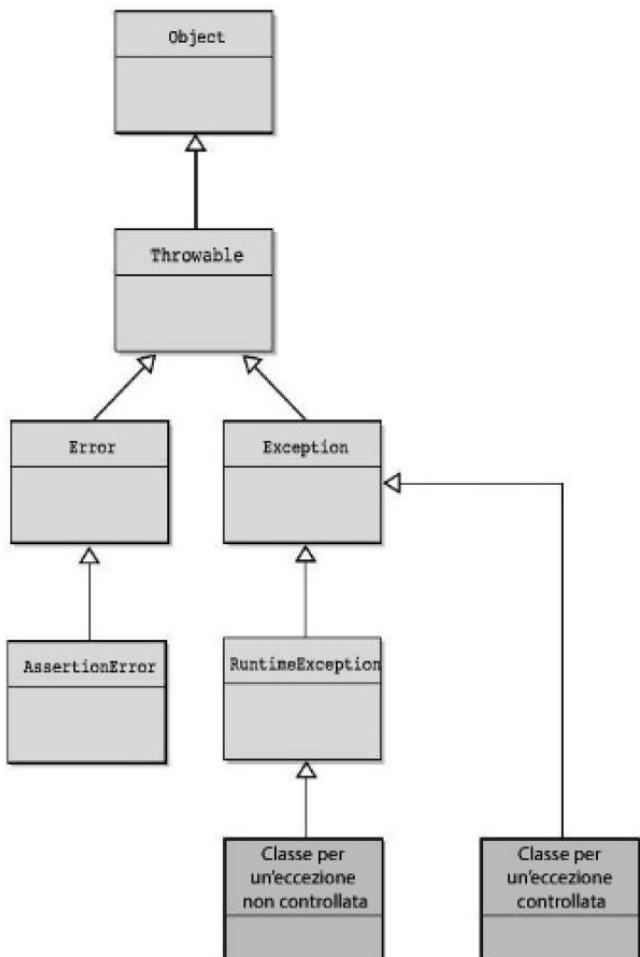
Come si può sapere se un'eccezione è controllata o no? Si può consultare la documentazione della classe Java Class Library per conoscere la classe base dell'eccezione e da quella dedurre se è controllata o meno.

Le eccezioni controllate derivano dalla classe Exception o da una sua classe derivata, le classi di eccezioni non controllate derivano dalla classe RuntimeException.

Se ci si dimentica di qualche eccezione che richiede una gestione, il compilatore ce ne informerà.

A questo punto si può decidere di catturare l'eccezione o aggiungerla in una clausola throws.

Riassunto di Jacopo De Angelis



13.3.3 Errori

Un errore è un oggetto della classe **Error**, derivata da **Throwable**. Da essa derivano le classi che non sono considerate classi di eccezioni. Sono comunque simili alle eccezioni, più nello specifico alle eccezioni non controllate perché non vi è necessità di `catch` o `throw`.

Gli errori sono il più delle volte fuori dal controllo del programmatore, ad esempio un `OutOfMemoryError` quando il programma ha esaurito la memoria disponibile, in questo caso o si modifica il programma o si cambiano le impostazioni in modo che Java possa accedere a più memoria.

Ad esempio il messaggio di errore derivante dall'utilizzo di un `assert` è in realtà un `AssertionError`.

13.3.4 throw e catch multipli

```

1 import java.util.Scanner;
2
3 public class DueCatchDemo {
4
5     public static double divisioneEccezionale(double numeratore, double
6         denominatore) throws DivisionePerZeroException {
7         if (denominatore == 0)
8             throw new DivisionePerZeroException();
9         return numeratore / denominatore;
10    }
11
12    public static void main(String[] args) {
13        try {
14            System.out.println("Inserire il numero di oggetti prodotti:");
15            Scanner tastiera = new Scanner(System.in);
16            int oggetti = tastiera.nextInt();
17
18            if (oggetti < 0)
19                throw new NumeroNegativoException("oggetti");
20
21            System.out.println("Quanti di questi erano difettosi?");
22            int difettosi = tastiera.nextInt();
23
24            if (difettosi < 0)
25                throw new NumeroNegativoException("oggetti difettosi");
26
27            double rapporto = divisioneEccezionale(oggetti, difettosi);
28            System.out.println("Un oggetto ogni " + rapporto + " e' "
29                + "difettoso");
30        } catch(DivisionePerZeroException e) {
31            System.out.println("Congratulazioni! Un record perfetto!");
32        } catch(NumeroNegativoException e) {
33            System.out.println("Impossibile avere un numero negativo di " +
34                e.getMessage());
35        }
36        System.out.println("Fine programma.");
37    }
38 }
```

un'eccezione.

Quando viene eseguito il blocco `try-catch-finally` ci si può trovare in tre situazioni:

- Il blocco `try` non genera eccezioni, in questo caso il blocco `finally` viene eseguito alla fine del blocco `try`
- Viene lanciata un'eccezione all'interno del blocco `try`, viene catturata da uno dei blocchi `catch` e successivamente viene eseguito il blocco `finally`
- Viene lanciata un'eccezione all'interno del blocco `try` ma non c'è un `catch` appropriato, in questo caso viene eseguito il blocco `finally` e l'eccezione viene mandata al metodo chiamante (se ce n'è uno)

Un blocco `try` può potenzialmente lanciare un numero qualsiasi di eccezioni, che possono essere di differenti tipi. Ogni blocco `catch` può catturare eccezioni di un solo tipo ma è possibile catturarne più tipi inserendo più blocchi `catch` dopo un blocco `try`.

È buona norma catturare prima l'eccezione più specifica.

È possibile annidare blocchi `try` all'interno di blocchi `catch` ma questa pratica è raramente utile.

13.3.5 Blocco finally

È possibile inserire un blocco `finally` dopo una sequenza di blocchi `catch`. Il codice nel blocco `finally` viene eseguito comunque, indipendentemente dal fatto che l'eccezione venga lanciata. Questo blocco offre la possibilità di risolvere problemi di coerenza che potrebbero crearsi in seguito a

13.3.6 Rilanciare un'eccezione

È lecito lanciare un'eccezione all'interno di un blocco catch. In alcuni rari casi può presentarsi la necessità di catturare un'eccezione e di dover decidere, in base alla stringa restituita, se rilanciare la stessa eccezione o una differente, in modo che venga gestita da altri blocchi catch più esterni.

La documentazione dovrebbe descrivere le possibili eccezioni.

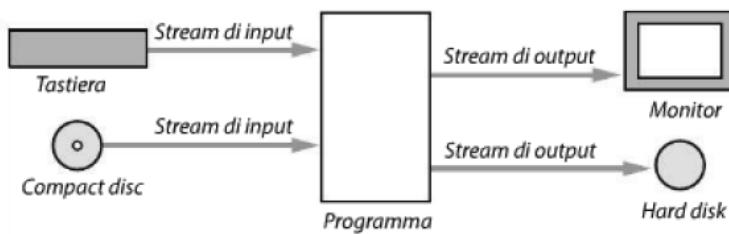
14 STREAM E I/O DA FILE

14.1 INTRODUZIONE AI FLUSSI DI DATI E ALL'I/O SU FILE

14.1.1 Il concetto di stream

In Java, l'I/O da e su file, è gestito in termini di flusso di dati. Un “flusso di dati” (stream) può essere costituito da caratteri, numeri o generici byte. Se i dati fluiscano nel programma, lo stream è detto stream

di input, se fluiscano dal programma è detto stream di output.



In Java, gli stream sono realizzati come istanze di alcune classi speciali di tipo stream. Gli oggetti di tipo Scanner sono di input, l'oggetto System.out è un esempio di output.

14.1.2 Perché utilizzare l'I/O su file?

Fino ad ora abbiamo sempre utilizzato dati temporanei. I file forniscono un mezzo per immagazzinarli in modo permanente.

14.1.3 File di testo e file binari

In qualunque tipo di file i dati sono immagazzinati per mezzo di bit, tuttavia queste stringhe hanno un'interpretazione, in certi casi come caratteri di testo. Questi file vengono detti “file di testo”, gli altri sono “file binari”.

I programmi Java sono salvati in file di testo mentre immagini e musica, ad esempio, sono immagazzinati in file binari. I programmi java possono leggere e scrivere sia file di testo che binari, la differenza consiste in quali classi debbano essere utilizzate per l'input e l'output.

File di testo	File binari
Possono essere visualizzati su qualsiasi computer tramite un editor di testi.	Soltamente le operazioni di scrittura e lettura sono legate ad un programma apposito, in certi casi addirittura ad un preciso SO o linguaggio di programmazione.
Immagazzinano i valori come singoli caratteri.	Immagazzinano tutti valori dello stesso tipo primitivo nello stesso formato.
<i>Un file di testo</i> 	<i>Un file binario</i>

Riprendendo le nozioni viste all'inizio, la caratteristica dei file binari di Java è che sono scritti in bytecode, universale tra tutte le macchine e quindi potranno essere spostati tra vari dispositivi.

Nei file di testo ogni carattere è rappresentato da uno (ASCII) o due (Unicode) byte.

14.2 I/O CON FILE DI TESTO

14.2.1 Creare un file di testo

La classe `PrintWriter` nella Java Class Library definisce i metodi per creare file di testo e scrivere in essi. È la classe da utilizzare, preferibilmente, per l'output su file di testo. La classe fa parte del package `java.io`, da importare esplicitamente.

```
String nomeFile = "out.txt"; //Lo si potrebbe chiedere all'utente
PrintWriter outputStream = null;
try {
    outputStream = new PrintWriter(nomeFile);
} catch (FileNotFoundException e) {
    System.out.println("Errore nell'apertura del file " + nomeFile);
    System.exit(0);
}
```

Prima di poter scrivere su di un file di testo, è necessario collegarlo a uno stream di output, cioè aprire il file. Per fare ciò bisogna usare il nome che può cercare il sistema operativo, ad esempio `out.txt`, ed è necessario dichiarare una variabile di

stream da utilizzare come riferimento allo stream. **Il tipo di variabile è `PrintWriter`. Per aprire un file per l'output si invoca il costruttore della classe `PrintWriter`.** Poiché questa azione può generare eccezioni bisogna includerla in un blocco `try`.

Anche la classe `FileNotFoundException` deve essere importata dal package `java.io`. Il nome del file deve essere importato come valore di tipo `String`. Quando si collega un file a uno stream di output in questo modo, il programma parte sempre da un file vuoto, tant'è che se il file esiste già verrà sovrascritto mentre se il file non esiste viene creato.

Poiché, come detto, il file viene creato se non esistente e se esistente viene sovrascritto, l'eccezione che può essere generata e gestita dal blocco `catch` è quella dell'impossibilità di creare il file, ad esempio perché il nome specificato è già utilizzato come nome di una directory.

I metodi `print` e `println` della classe `PrintWriter` si comportano esattamente come quelli della classe `out`, semplicemente l'output è contenuto nel file di testo.

Una volta aperto il file ci si riferisce ad esso con il nome della variabile associata allo stream (cioè l'oggetto `PrintWriter`, nell'esempio `outputStream`) **e non con il nome del file.** Attenzione, la variabile è dichiarata all'esterno del blocco `try`, quindi è comunque disponibile.

La classe `PrintWriter` non invia immediatamente l'output al file ma aspetta di aver accumulato una quantità abbastanza grande di dati. Questi, nel frattempo, vengono salvati nel buffer. Quando pieno il suo contenuto viene scritto. Questa tecnica è detta buffering e consente un'elaborazione più veloce del file.

Una volta che l'interno file di testo è stato scritto lo si disconnette dallo stream tramite l'istruzione `"nome_file.close()"`. In questo modo il sistema libera qualunque risorsa utilizzata per connettere il file allo stream e svolge varie operazioni di pulizia. Se non viene chiuso esplicitamente, Java lo farà automaticamente alla fine del programma, tuttavia è meglio farlo manualmente poiché una chiusura inaspettata del file rischia di portare al non salvataggio del file nella maniera corretta.

Nel caso si voglia scrivere e poi leggere un file, è obbligatorio prima chiuderlo.

È buona norma comunicare, dopo la chiusura, quando sono stati salvati i file, questo per evitare un "programma silenzioso" che non comunica all'utente se il file è stato salvato.

14.2.2 Aggiungere dati a un file di testo

In certi casi si desidera aggiungere testo ad un file già esistente. Per poter aggiungere l'output ad un file di testo il cui nome è riportato nella variabile di tipo `String nomeFile`, la connessione tra il file e lo stream `OutputStream` dovrà essere effettuata in questo modo:

```
OutputStream = new PrintWriter(new FileOutputStream(nomeFile, true));
```

poiché la classe `PrintWriter` non offre direttamente un costruttore che consenta l'operazione di aggiunta, si ricorre alla classe `FileOutputStream` (sempre contenuta nel package `java.io`). Il secondo argomento (`true`) indica che si vogliono aggiungere dati al file se questo esiste già. Quindi se il file non esiste verrà creato, altrimenti l'output verrà aggiunto.

Utilizzando questo costruttore si devono sempre usare i blocchi `try` e `catch` come nell'esempio precedente.

14.2.3 Leggere da un file di testo

Le due classi di tipo stream più utilizzate per leggere file di testo sono `Scanner` e `BufferedReader`. La classe `Scanner` è più ricca ed è la soluzione preferibile.

14.2.4 Leggere un file di testo con la classe `Scanner`

```
1 import java.io.File;
2 import java.io.FileNotFoundException;
3 import java.util.Scanner;
4
5
6 public class FileDiTestoInputConScannerDemo {
7     public static void main(String[] args) {
8         String nomeFile = "out.txt";
9         Scanner inputStream = null;
10        System.out.println("Il file " + nomeFile + "\ncontiene le righe
11        seguenti:\n");
12        try {
13            inputStream = new Scanner(new File(nomeFile));
14        } catch (FileNotFoundException e) {
15            System.out.println("Errore nell'apertura del file " + nomeFile);
16            System.exit(0);
17        }
18        while (inputStream.hasNextLine()) {
19            String riga = inputStream.nextLine();
20            System.out.println(riga);
21        }
22        inputStream.close();
23    }
24 }
```

argomento un nome di file.

La seconda parte del codice legge il file e lo stampa a schermo fino a quando il file ha una riga successiva all'attuale.

Ecco alcuni metodi aggiuntivi della classe `Scanner`, utili quando si lavora con un file.

`nome_oggetto_scanner.hasNext()`

Restituisce true se sono disponibili altri dati da leggere tramite il metodo `next`

`nome_oggetto_scanner.hasNextDouble()`

Restituisce true se sono disponibili altri dati da leggere tramite il metodo `nextDouble`

`nome_oggetto_scanner.hasNextInt()`

Restituisce true se sono disponibili altri dati da leggere tramite il metodo `nextInt`

Il listato qui accanto presenta un programma che legge dati da un file di testo utilizzando la classe `Scanner` e li mostra sullo schermo. Il file `out.txt` è un file di testo che potrebbe essere stato creato da qualcuno usando un editor di testi o un programma Java utilizzando la classe `PrintWriter`.

Non si può passare direttamente il nome del file al costruttore di `Scanner`.

La classe `Scanner` ha un costruttore che accetta come argomento un'istanza della classe standard `File`, la quale ha un costruttore che accetta come

<code>nome_oggetto_scanner.hasNextLine()</code>	Restituisce true se sono disponibili altri dati da leggere tramite il metodo nextLine
---	---

14.2.5 Leggere un file di testo con la classe `BufferedReader`

La classe `BufferedReader` non ha un costruttore che accetti un nome di file come argomento, quindi è necessario utilizzare un'altra classe (in questo caso `FileReader`) per convertire il nome del file in un oggetto che possa essere passato.

```

1 import java.io.BufferedReader;
2 import java.io.FileReader;
3 import java.io.FileNotFoundException;
4 import java.io.IOException;
5
6 public class FileDiTestoInputConBufferedReaderDemo {
7     public static void main(String[] args) {
8
9         String nomeFile = "originale.txt";
10        BufferedReader inputStream = null;
11        System.out.println("Il file " + nomeFile + "\ncontiene le righe
12        seguenti:\n");
13
14        try {
15            inputStream = new BufferedReader(new FileReader(nomeFile));
16
17        } catch (FileNotFoundException e) {
18            System.out.println("Problema nell'apertura del file.");
19        }
20
21        try {
22            String riga = inputStream.readLine();
23            while (riga != null) {
24                System.out.println(riga);
25                riga = inputStream.readLine();
26            }
27            inputStream.close();
28        } catch (IOException e) {
29            System.out.println("Errore nella lettura da " + nomeFile);
30        }
31    }
32 }
```

specializzazione di `IOException`, quindi la prima potrebbe essere gestita anche con la seconda, tuttavia così si avrebbero meno informazioni.

La classe `BufferedReader` non ha modi per leggere valori numerici. Per leggere un singolo valore numerico che occupa un'intera riga si utilizza `readLine` e poi il metodo `Integer.parseInt` o simili per convertire quella parte della stringa in un numero.

```

 StringTokenizer riga = new StringTokenizer("4 7 9");
while (riga.hasMoreTokens()) {
    System.out.println(riga.nextToken());
}
```

La classe `StringTokenizer` è presente nel package `java.util`. Uno degli utilizzi più comuni consiste nel decomporre una riga di testo nelle sue parole, come nell'esempio qua accanto. Queste singole parole si chiamano token. Il metodo `nextToken` restituisce il primo token quando è invocato la prima volta, il secondo la seconda volta e così via. Il metodo `hasMoreTokens` restituisce un valore booleano.

la classe `BufferedReader` offre solo due metodi per la lettura di dati da un file di testo, `readLine` e `read`. Il metodo `readLine` è uguale a quello della classe `Scanner`, il metodo `read` legge un singolo carattere restituendo però un valore di tipo `int` che corrisponde al carattere letto e non il carattere stesso.

Così come `readLine`, anche `read` restituisce un valore speciale nel caso si raggiunga la fine del file. Tale valore è -1.

Notare come l'apertura del file possa generare un `FileNotFoundException` mentre il `readLine` può generare un `IOException`.

`FileNotFoundException` è una

La classe `StringTokenizer` è presente nel package `java.util`. Uno degli utilizzi più comuni consiste nel decomporre una riga di testo nelle sue parole, come nell'esempio qua accanto. Queste singole parole si chiamano token. Il metodo `nextToken` restituisce il primo token quando è invocato la prima volta, il secondo la seconda volta e così via. Il metodo `hasMoreTokens` restituisce un valore booleano.

Le classi BufferedReader e FileReader appartengono al package java.io

```
public BufferedReader(Reader oggettoReader)
```

Questo è l'unico costruttore che si utilizza comunemente. Poiché non esiste un costruttore che accetti come argomento il nome di un file, se si vuole specificare un file mediante il suo nome bisogna utilizzare...

```
new BufferedReader(new FileReader(nome_file))
```

Il costruttore di FileReader può generare una FileNotFoundException.

Se si vuole creare uno stream a partire da un oggetto della classe File si usa l'istruzione...

```
new BufferedReader(new FileReader(oggetto_file))
```

Il costruttore di FileReader può generare una FileNotFoundException

```
public String readLine() throws IOException
```

Legge una riga dallo stream di input e la restituisce. Se la lettura supera la fine del file viene restituito null

```
public int read() throws IOException
```

Legge un singolo carattere dallo stream di input e lo restituisce sotto forma di valore di tipo int. Se la lettura oltrepassa la fine del file, viene restituito -1.

```
public long skip(long n) throws IOException
```

Salta n caratteri

```
public void close() throws IOException
```

Chiude lo stream

14.3 TECNICHE PER LA GESTIONE DEI FILE

Le tecniche che verranno presentate possono essere utilizzate sia con i file di testo che con i file binari.

14.3.1 La classe File

```

1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4
5 public class FileDiTestoInputConScannerDemo2 {
6     public static void main(String[] args) {
7         System.out.print("Inserire il nome di un file: ");
8         Scanner tastiera = new Scanner(System.in);
9         String nomeFile = tastiera.nextLine();
10        Scanner inputStream = null;
11        System.out.println("Il file " + nomeFile + "\n" +
12                           "contiene le righe seguenti:\n");
13
14        try {
15            inputStream = new Scanner(new File(nomeFile));
16        } catch (FileNotFoundException e) {
17            System.out.println("Errore nell'apertura del file " + nomeFile);
18            System.exit(0);
19        }
20        while (inputStream.hasNextLine()) {
21            String riga = inputStream.nextLine();
22            System.out.println(riga);
23        }
24        inputStream.close();
25    }
26 }
```

Tuttavia il nome del file potrebbe essere sconosciuto al momento della creazione del codice (nell'esempio è l'utente a dare il nome, quindi non si può sapere a priori). **Quindi, come si vede nell'esempio, il nome viene utilizzato per creare un oggetto di tipo File che ha come input per il costruttore la variabile String.**

La classe File consente una gestione omogenea dei file a partire dal loro nome.

Una stringa "tesoro.txt", infatti, potrebbe rappresentare il nome di un file, ma Java lo interpreterà come una stringa. Se invece si passa questa stringa al costruttore di File, creerà un oggetto che sa di essere interpretato come l'identificativo di un file.

Fino ad ora i nomi dei file da utilizzare sono stati riportati direttamente come stringhe nel codice (prima parte sottolineata).

14.3.2 Percorsi

Quando si utilizza il nome di un file per aprirlo in uno dei modi visti in precedenza, si presuppone sia nella stessa directory nella quale viene eseguito il programma. **Tuttavia, nel caso la cartella sia differente, si può specificare il suo datapath. Ci sono due tipi di datapath:**

- **Absoluto o completo (full path name):** fornisce indicazioni partendo dalla cartella radice
- **Relativo:** fornisce il percorso per raggiungere il file relativamente alla directory di esecuzione del programma

Il modo nel quale vengono specificati dipende dal sistema operativo. Un esempio di percorso per Windows è C:\lavoro\dati.txt, in questo caso, per creare uno stream di input si scriverebbe: Scanner

inputStream = new Scanner(new File("C:\\lavoro\\dati.txt")). Si utilizza \\ perché altrimenti i backslash verrebbero interpretati come caratteri di escape, ad esempio \d. Nel caso, invece,

venga letto da tastiera, non ci sono problemi. Infatti scrivere C:\lavoro\dati.txt permette di interpretare correttamente la stringa.

Un consiglio è utilizzare il formato UNIX, ovvero / al posto di \, infatti Java accetterà sempre l'input in formato UNIX.

```

1 import java.util.Scanner;
2 import java.io.File;
3 import java.io.FileNotFoundException;
4
5 public class FileDiTestoInputConScannerDemo3 {
6     public static void main(String[] args) {
7         System.out.print("Inserire il nome di un file: ");
8         Scanner tastiera = new Scanner(System.in);
9         String nomeFile = tastiera.next();
10        Scanner inputStream = null;
11        System.out.println("Il file " + nomeFile + "\n" +
12                           "contiene le righe seguenti:\n");
13
14        try {
15
16            File oggettoFile = new File(nomeFile);
17            boolean fileOK = false;
18            while (!fileOK) {
19                if (!oggettoFile.exists())
20                    System.out.println("Il file non esiste");
21                else if (!oggettoFile.canRead())
22                    System.out.println("Il file non può essere letto.");
23                else
24                    fileOK = true;
25
26                if (!fileOK) {
27                    System.out.println("Reinserire il nome del file:");
28                    nomeFile = tastiera.next();
29                    oggettoFile = new File(nomeFile);
30                }
31            }
32
33            inputStream = new Scanner(oggettoFile);
34
35        } catch (FileNotFoundException e) {
36            System.out.println("Errore nell'apertura del file " + nomeFile);
37            System.exit(0);
38        }
39
40        while (inputStream.hasNextLine()) {
41            String riga = inputStream.nextLine();
42            System.out.println(riga);
43        }
44        inputStream.close();
45    }
46 }
```

public boolean canRead()	Verifica se il programma può leggere il file
public boolean canWrite()	Verifica se il programma può scrivere nel file
public boolean delete()	Prova ad eliminare il file. Restituisce true se è stato possibile
public boolean exists()	Verifica se esiste un file con il nome utilizzato come argomento al costruttore quando è stato creato l'oggetto File
Public String getName()	Restituisce il nome del file (non il percorso)
public String getPath()	Restituisce il percorso del file
Public long length()	Restituisce la lunghezza del file in byte

14.4 BASI DELL'I/O CON FILE BINARI

In questo paragrafo verranno utilizzate le classi `ObjectInputStream` e `ObjectOutputStream` per leggere e scrivere file binari. Queste classi forniscono metodi per scrivere file un byte alla volta.

14.4.1 Creare un file binario

```

1 import java.io.FileOutputStream;
2 import java.io.ObjectOutputStream;
3 import java.io.FileNotFoundException;
4 import java.io.IOException;
5 import java.util.Scanner;
6
7 public class FileBinarioOutputDemo {
8     public static void main(String[] args) {
9         String nomeFile = "numeri.dat";
10        try {
11            ObjectOutputStream outputStream = new ObjectOutputStream(new
12                           FileOutputStream(nomeFile));
13            Scanner tastiera = new Scanner(System.in);
14            System.out.println("Inserire degli interi non negativi.");
15            System.out.println("Inserire un numero negativo per terminare.");
16
17            int unIntero;
18            do {
19                unIntero = tastiera.nextInt();
20                outputStream.writeInt(unIntero);
21            } while (unIntero >= 0);
22
23            System.out.println("I numeri e il valore di terminazione");
24            System.out.println("sono stati scritti nel file " + nomeFile);
25            outputStream.close();
26        } catch (FileNotFoundException e) {
27            System.out.println("Errore nell'apertura del file " + nomeFile);
28        } catch (IOException e) {
29            System.out.println("Errore nella scrittura nel file " +
30                           nomeFile);
31        }
32    }
33 }
```

Per creare un file binario si può utilizzare la classe di tipo stream `ObjectOutputStream`.

La parte principale del programma è inclusa in un blocco `try`. Tutte le istruzioni basate su file binari possono generare una `IOException`.

La prima operazione evidenziata è detta apertura del file. Se il file non esiste verrà creato, altrimenti verrà sovrascritto. Notare che il costruttore della classe `ObjectOutputStream` non può ricevere come argomento una stringa, cosa che può invece fare il costruttore di `FileOutputStream`. Il costruttore di quest'ultimo può generare un `FileNotFoundException`.

14.4.2 Scrivere valori di tipo primitivo in un file binario

La classe `ObjectOutputStream` non offre un metodo `println`. Tuttavia offre un metodo `writeInt` che scrive in un file binario un singolo valore di tipo `int`. Quindi, una volta che è stato ottenuto uno stream `outputStream` di tipo `ObjectOutputStream` connesso al file, è possibile scrivere valori interi scrivendo `outputStream.writeInt(unIntero)`. Il metodo `writeInt` può generare una `IOException`. Alla fine dello stream è sempre bene chiuderlo tramite l'istruzione `outputStream.close()`.

Ecco alcuni metodi della classe `ObjectOutputStream`.

```
public ObjectOutputStream(OutputStream oggettoStream)
```

Crea uno stream di output collegato al file binario specificato. Non esiste un costruttore che accetti come argomento il nome del file. Per creare uno stream a partire dal nome del file, bisogna utilizzare...

```
new ObjectOutputStream(new FileOutputStream(nome_file))
```

Oppure utilizzando la classe `File`...

```
new ObjectOutputStream(new FileOutputStream(new File(nome_file)))
```

Entrambe le istruzioni creano un file vuoto, se esisteva un file di nome `nome_file`, il contenuto del file viene perso.

Il costruttore di `FileOutputStream` può generare una `FileNotFoundException`. Se ciò non accade, il costruttore di `ObjectOutputStream` può generare una `IOException`

```
public void writeInt(int n) throws IOException
```

Scrive il valore `n` di tipo `int` nello stream di output

```
public void writeLong(long n) throws IOException
```

Scrive il valore `n` di tipo `long` nello stream di output

```
public void writeDouble(double x) throws IOException
```

Scrive il valore x di tipo double nello stream di output

```
public void writeFloat(float x) throws IOException
```

Scrive il valore x di tipo float nello stream di output

```
public void writeChar(int c) throws IOException
```

Scrive un valore char nello stream di output. Si noti che il parametro c è di tipo int. Tuttavia, Java convertirà automaticamente un valore char in un int. Quindi, la seguente istruzione rappresenta un utilizzo corretto del metodo

```
public void writeChar('A')
```

```
public void writeBoolean(boolean b) throws IOException
```

Scrive il valore b di tipo boolean nello stream di output

```
public void writeUTF(String unaStringa) throws IOException
```

Scrive la stringa unaStringa nello stream di output. La sigla UTF si riferisce a una particolare codifica per le stringhe. Per leggere la stringa del file si utilizzerà il metodo `readUTF` della classe

`ObjectInputStream`

```
public void writeObject(Object unOggetto) throws IOException,  
NotSerializableException, InvalidClassException
```

Scrive l'oggetto unOggetto nello stream di output. L'argomento deve essere un oggetto di una classe serializzabile, argomento discusso più avanti.

Il metodo genera un'eccezione `NotSerializableException` se l'oggetto è di una classe non serializzabile.

Genera una `InvalidClassException` se c'è stato un problema nella serializzazione.

Il metodo `writeObject` sarà discusso più avanti

```
public void close() throws IOException
```

Chiude lo stream

14.4.3 Scrivere stringhe in un file binario

Per scrivere stringhe in un file binario si utilizza il metodo `writeUTF`. Per esempio, se `OutputStream` è uno stream di tipo `ObjectOutputStream`, la seguente istruzione scriverà la stringa "Stringa di prova": `outputStream.writeUTF("Stringa di prova")`. Si possono scrivere anche dati di tipo diverso all'interno di un file, tuttavia mescolare dati di tipo diverso richiede attenzione affinché possano essere letti correttamente, in particolare occorre tenere traccia dell'ordine nel quale sono stati scritti poiché ogni tipo di dato richiede un metodo differente di lettura.

14.4.4 Alcuni dettagli sul metodo `writeUTF`

Mentre tutti i tipi di metodi di scrittura per tipi primitivi usano un numero di bit standard in base al tipo, `writeUTF` utilizza un numero variabile di byte. Ciò può rappresentare un problema per Java dato che in un file binario non esistono separatori. Per ovviare a questo problema, Java inserisce delle informazioni aggiuntive all'inizio di ogni stringa, contenenti da quanti byte è composta la striga, così che il metodo `readUTF` sa quanti byte leggere e decodificare.

In realtà il funzionamento è più complicato perché per caratteri differenti si utilizzano diversi numeri di byte.

14.4.5 Leggere da un file binario

Se un file binario è stato creato usando un `ObjectOutputStream`, lo si può leggere sfruttando la classe `ObjectInputStream`. Ogni metodo di scrittura ha un corrispondente metodo di lettura.

L'apertura di un file binario in lettura avviene in modo simile a quella in scrittura. Un `ObjectInputStream` permette di leggere dati di tipo diverso dallo stesso file. Tuttavia, se si prova a leggere un dato di tipo diverso da quello atteso, il risultato non sarà quello desiderato.

```
public ObjectInputStream(InputStream oggettoStream)
```

Crea uno stream di input collegato al file binario specificato. Non esiste un costruttore che accetti come argomenti il nome del file. Per creare uno stream a partire dal nome del file, bisogna utilizzare...

```
new ObjectInputStream(new FileInputStream(nome_file))
```

Ottobre, utilizzando la classe File...

```
new ObjectInputStream(new FileInputStream(new File(nome_file)))
```

Entrambe le istruzioni creano un file vuoto o ne sovrascrivono uno preesistente.

Il costruttore di `FileInputStream` può generare una `FileNotFoundException`. Se ciò non accade, il costruttore di `ObjectInputStream` può generare una `IOException`

```
public int readInt() throws EOFException, IOException
```

Legge un valore di tipo `int` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeInt` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`

```
public long readLong() throws EOFException, IOException
```

Legge un valore di tipo `long` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeLong` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`.

Si noti che non è possibile scrivere un valore intero usando `writeLong` e in seguito leggerlo usando `readInt`, o viceversa, scriverlo con `writeInt` e leggerlo con `readLong`. Se si prova a fare ciò, si otterranno risultati imprevedibili.

```
public double readDouble() throws EOFException, IOException
```

Legge un valore di tipo `double` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeDouble` della classe

`ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`

```
public float readFloat() throws EOFException, IOException
```

Legge un valore di tipo `float` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeFloat` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`.

Si noti che non è possibile scrivere un valore intero usando `writeDouble` e in seguito leggerlo usando `readFloat`, o viceversa, scriverlo con `writeFloat` e leggerlo con `readDouble`. Se si prova a fare ciò, si otterranno risultati imprevedibili.

```
public char readChar() throws EOFException, IOException
```

Legge un valore di tipo `char` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeChar` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`.

```
public boolean readBoolean() throws EOFException, IOException
```

Legge un valore di tipo `boolean` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeBoolean` della classe

`ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Se il tentativo di lettura oltrepassa la fine del file, viene generata una `EOFException`.

```
public String readUTF() throws UTFDataFormatException, IOException
```

Legge un valore di tipo `String` dallo stream di input e lo restituisce. Se il metodo cerca di leggere un valore che non è stato scritto utilizzando il metodo `writeUTF` della classe `ObjectOutputStream` (o in qualche modo equivalente) si avranno problemi. Può generare una `UTFDataFormatException` o una `IOException`

```
public Object readObject () throws ClassNotFoundException,  
InvalidClassException, OptionlDataException, IOException
```

Legge un oggetto dallo stream di input e lo restituisce. Genera una `ClassNotFoundException` se non è stato possibile trovare un oggetto di una classe serializzabile.

Genera una `InvalidClassException` se c'è stato un problema con una classe serializzabile.

Genera una `OptionalDataException` se nello stream è stato trovato un valore di tipo primitivo al posto di un oggetto.

Genera una `IOException` se c'è stato qualche altro tipo di problema di I/O.

Il metodo `readObject` sarà analizzato con maggiore dettaglio più avanti

```
public void close() throws IOException
```

Chiude lo stream

```

1 import java.io.FileInputStream;
2 import java.io.ObjectInputStream;
3 import java.io.EOFException;
4 import java.io.FileNotFoundException;
5 import java.io.IOException;
6
7 public class FileBinarioInputDemo {
8     public static void main(String[] args) {
9         String nomeFile = "numeri.dat";
10        try {
11            ObjectInputStream inputStream = new ObjectInputStream(new
12                           FileInputStream(nomeFile));
13            System.out.println("Lettura dei numeri non negativi");
14            System.out.println("nel file " + nomeFile);
15
16            int unIntero = inputStream.readInt();
17            while (unIntero >= 0) {
18                System.out.println(unIntero);
19                unIntero = inputStream.readInt();
20            }
21
22            System.out.println("Fine della lettura dal file.");
23            inputStream.close();
24        } catch (FileNotFoundException e) {
25            System.out.println("Errore nell'apertura del file " + nomeFile);
26        } catch (EOFException e) {
27            System.out.println("Errore nella lettura del file " + nomeFile);
28            System.out.println("Raggiunta la fine del file.");
29        } catch (IOException e) {
30            System.out.println("Errore nella lettura del file " + nomeFile);
31        }
32    }
33 }
```

eccezioni lavorando con i file binari rispetto ai file di testo.

```

1 import java.io.FileInputStream;
2 import java.io.ObjectInputStream;
3 import java.io.EOFException;
4 import java.io.FileNotFoundException;
5 import java.io.IOException;
6
7 public class EOFExceptionDemo {
8     public static void main(String[] args) {
9         String nomeFile = "numeri.dat";
10        try {
11            ObjectInputStream inputStream = new ObjectInputStream(new
12                           FileInputStream(nomeFile));
13            System.out.println("Lettura di TUTTI gli interi");
14            System.out.println("nel file " + nomeFile);
15
16            try {
17                while (true) {
18                    int unIntero = inputStream.readInt();
19                    System.out.println(unIntero);
20                }
21            } catch (EOFException e) {
22                System.out.println("Fine della lettura dal file.");
23            }
24            inputStream.close();
25        } catch (FileNotFoundException e) {
26            System.out.println("Errore nell'apertura del file " + nomeFile)
27        } catch (IOException e) {
28            System.out.println("Errore nella lettura del file " + nomeFile)
29        }
30    }
31 }
```

14.4.6 La classe `EOFException` (End Of File)

Molti dei metodi per la lettura da file binari generano una `EOFException` quando cercano di leggere oltre la fine del file. È importante distinguere tra i due programmi qui a sinistra. Il primo controlla se è stata raggiunta la fine del file cercando un valore negativo, ciò non permette di utilizzare valori sotto allo 0 nel file. Il secondo invece cerca un'eccezione `EOFException` e avvisa che il file è finito, permettendo così l'utilizzo anche dei valori negativi.

È opportuno cercare sempre la fine del file per non incorrere in vari tipi di problemi.

Certe operazioni è più facile che generino più eccezioni lavorando con i file binari rispetto ai file di testo. È quindi opportuno dedicare più tempo alla gestione delle eccezioni con i primi.

14.5 I/O SU FILE BINARI DI OGGETTI E ARRAY

In questo paragrafo si discute l'I/O per file binari in relazione a oggetti e array. Per fare questo, si utilizzeranno le classi `ObjectInputStream` e `ObjectOutputStream`.

14.5.1 I/O binario con oggetti di tipo classe

Come si possono leggere e scrivere oggetti di tipo diverso? Si potrebbero salvare in un file i valori delle variabili di istanza dell'oggetto e

```

1 import java.io.Serializable;
2 import java.util.Scanner;
3
4 /**
5 Classe serializzabile per descrivere le specie a rischio.
6 */
7 public class Specie implements Serializable {
8     private String nome;
9     private int popolazione;
10    private double tassoCrescita;
11
12    public Specie(String nomeIniziale, int popolazioneIniziale,
13                  double tassoCrescitaIniziale) {
14        nome = nomeIniziale;
15        if (popolazioneIniziale >= 0)
16            popolazione = popolazioneIniziale;
17        else {
18            System.out.println("ERRORE: Popolazione negativa.");
19            System.exit(0);
20        }
21        tassoCrescita = tassoCrescitaIniziale;
22    }
23
24    public Specie() {
25        this(null, 0, 0);
26    }
27
28    public String toString() {
29        return ("Nome = " + nome + "\n" + "Popolazione = " + popolazione +
30                "\n" + "Tasso di crescita = " + tassoCrescita + "%");
31    }
32
33 // <Gli altri metodi sono gli stessi del Listato 8.14 del Capitolo 8,
34 // ma non serviranno per la discussione in questo capitolo.>
35 }

1 import java.io.FileInputStream;
2 import java.io.FileOutputStream;
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5 import java.io.ObjectOutputStream;
6
7 public class IOoggettiClasseDemo {
8     public static void main(String[] args) {
9         String nomeFile = "specie.registrazioni";
10
11     ObjectOutputStream outputStream = null;
12     try {
13         outputStream = new ObjectOutputStream(new FileOutputStream(
14             nomeFile));
15     } catch (IOException e) {
16         System.out.println("Errore nell'apertura del file di output " +
17                           nomeFile + ".");
18         System.exit(0);
19     }
20
21     Specie condorCalifornia = new Specie("Condor della California", 27, 0.02);
22     Specie rinoceronteNero = new Specie("Rinoceronte Nero", 100, 1.0);
23
24     try {
25         outputStream.writeObject(condorCalifornia);
26         outputStream.writeObject(rinoceronteNero);
27         outputStream.close();
28     } catch (IOException e) {
29         System.out.println("Errore nella scrittura del file " + nomeFile +
30                           ".");
31         System.exit(0);
32     }
33     System.out.println("Le registrazioni sono state scritte nel file " +
34                         nomefile + ".");
35     System.out.println("Ora il file verrà riaperto e verranno mostrate " +
36                         "le registrazioni.");
37     ObjectInputStream inputStream = null;
38     try {
39         inputStream = new ObjectInputStream(new FileInputStream(
40             nomeFile));
41     } catch (IOException e) {
42         System.out.println("Errore nell'apertura del file di input " +
43                           nomeFile + ".");
44         System.exit(0);
45     }
46
47     Specie lettaUno = null, lettaDue = null;
48     try {
49         lettaUno = (Specie)inputStream.readObject();
50         lettaDue = (Specie)inputStream.readObject();
51         inputStream.close();
52     } catch (Exception e) {
53         System.out.println("Error nella lettura del file " + nomeFile + ".");
54         System.exit(0);
55     }
56
57     System.out.println("Sono stati letti dal file " + nomeFile +
58                         "\ni seguenti dati.");
59     System.out.println(lettaUno);
60     System.out.println();
61     System.out.println(lettaDue);
62     System.out.println("Fine del programma.");
63 }

```

ricostruire in qualche modo un nuovo oggetto a partire dai dati salvati nel file. Tuttavia le variabili di istanza potrebbero essere oggetti a loro volta.

Java offre un modo chiamato “serializzazione degli oggetti”, per rappresentare un soggetto sotto forma di una sequenza di byte che possono essere scritti in un file binario. Questa procedura avviene automaticamente per gli oggetti delle classi serializzabili.

Rendere serializzabile una classe è molto semplice, basta aggiungere alla sua dichiarazione “`implements Serializable`”. L’interfaccia `Serializable` fa parte della libreria standard Java che appartiene al package `java.io`.

l’interfaccia è vuota ma la sua importazione indica a Java di rendere la classe serializzabile, quindi bisogna scrivere anche “`import java.io.Serializable`”.

La scrittura di oggetti di classi serializzabili in un file binario avviene per mezzo del metodo `writeObject` della classe `ObjectOutputStream`, mentre la lettura può essere effettuata tramite il metodo `readObject` della classe `ObjectInputStream`.

ObjectInputStream. Nel secondo listato si mostra un esempio. Per scrivere un oggetto di tipo `Specie` in un file binario, si passa l’oggetto come argomento al metodo `writeObject`, come in “`outputStream.writeObject (unaSpecie)`”.

Un oggetto scritto mediante `writeObject` viene letto tramite `readObject` ma attenzione, poiché bisogna leggere un oggetto non di classe `Object`, bisogna fare il typecast durante la lettura, come “`letta = (Specie)inputStream.readObject()`”.

Attenzione, la classe `Specie` ha un metodo `toString`, necessario per produrre un output leggibile quando si scrive su schermo o in un file di testo utilizzando il metodo `println`, tuttavia il metodo `toString` non ha nulla a che fare con l’I/O di oggetti su file binari.

14.5.2 Alcuni dettagli sulla serializzazione

Quando una classe serializzabile ha delle variabili di istanza di tipo classe, anche le classi delle

variabili di istanza devono essere serializzabili, e così via per tutti i livelli.

Ogni classe che deriva da una classe serializzabile è serializzabile.

Rendere una classe serializzabile non cambia niente dal punto di vista della classe ma rende più efficienti le operazioni di I/O su file, assegnando ad ogni oggetto della classe un numero di serie.

Se tutte le classi non vengono rese serializzabili è per motivi di sicurezza. Il sistema basato sui numeri di serie rende infatti semplice per i programmatore accedere agli oggetti salvati su memorie secondarie.

14.5.3 Array nei file binari

Dato che Java tratta gli array come oggetti, è possibile utilizzare il metodo `writeObject` per scrivere un intero array in un file binario e successivamente rileggerlo tramite il metodo `readObject`.

Quando si fa questo, se il tipo base dell'array è una classe, questa deve essere serializzabile. Quindi, se tutti i dati il cui tipo è una classe serializzabile sono organizzati in un array, è possibile scriverli tutti in un file binario tramite una singola chiamata a `writeObject`.

```

1 import java.io.FileInputStream;
2 import java.io.FileOutputStream;
3 import java.io.IOException;
4 import java.io.ObjectInputStream;
5 import java.io.ObjectOutputStream;
6
7 public class IOArrayDemo {
8     public static void main(String[] args) {
9         Specie[] unArray = new Specie[2];
10        unArray[0] = new Specie("Condor della California", 27, 0.02);
11        unArray[1] = new Specie("Rinoceronte Nero", 100, 1.0);
12        String nomeFile = "array.dat";
13
14        try {
15            ObjectOutputStream outputStream = new ObjectOutputStream(new
16                           FileOutputStream(nomeFile));
17            outputStream.writeObject(unArray);
18            outputStream.close();
19        } catch (IOException e) {
20            System.out.println("Errore nella scrittura del file " +
21                               nomeFile + ".");
22            System.exit(0);
23        }
24
25        System.out.println("L'array è stato scritto nel file " + nomeFile +
26                           " e il file è stato chiuso.");
27        System.out.println("Ora il file verrà riaperto e verrà stampato
28                           l'array.");
29
30        Specie[] unAltroArray = null;
31        try {
32            ObjectInputStream inputStream = new ObjectInputStream(new
33                           FileInputStream(nomeFile));
34            unAltroArray = (Specie[])inputStream.readObject();
35            inputStream.close();
36        } catch (Exception e) {
37            System.out.println("Errore nella lettura del file " +
38                               nomeFile + ".");
39            System.exit(0);
40        }
41
42        System.out.println("I seguenti dati sono stati letti dal file " +
43                           nomeFile + ":" );
44        for (int i = 0; i < unAltroArray.length; i++) {
45            System.out.println(unAltroArray[i]);
46            System.out.println();
47        }
48        System.out.println("Fine del programma.");
49    }

```

si noti che la classe che costituisce il tipo base dell'array, `Specie`, è serializzabile. Si noti inoltre la necessità di effettuare una conversione esplicita.

14.6 RIEPILOGO

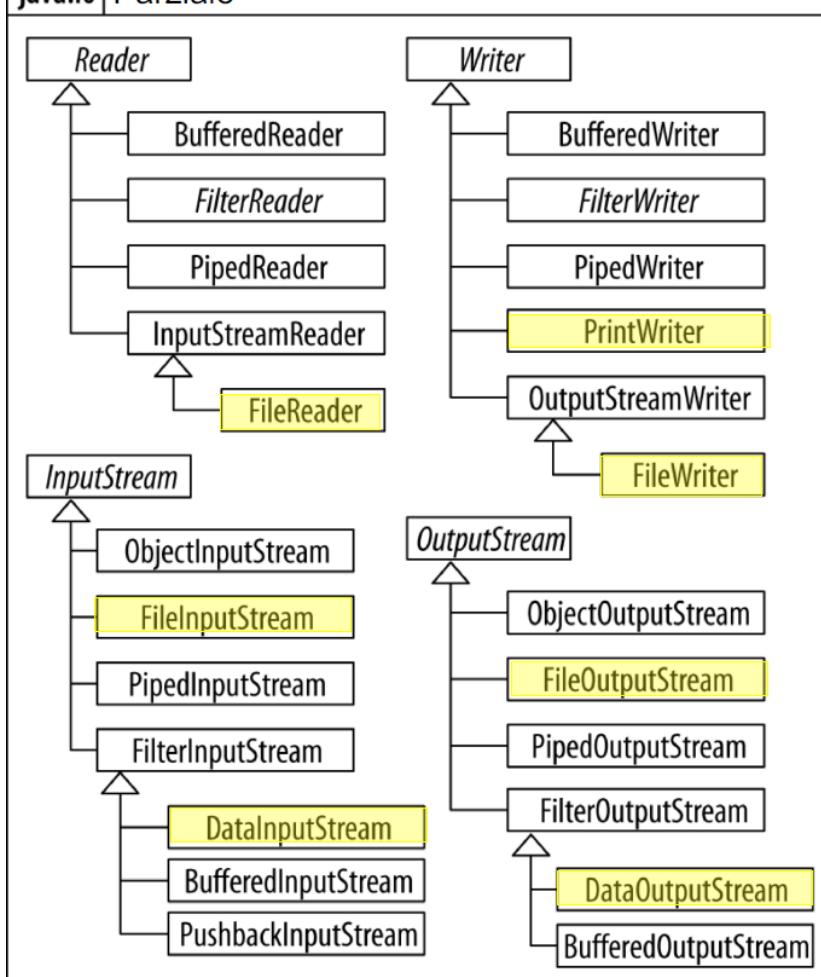
- I file che vengono interpretati come sequenze di caratteri dai programmi Java e degli editor di testo sono detti file di testo. Tutti gli altri sono file binari
- Si può utilizzare la classe `PrintWriter` per scrivere in un file di testo e la classe `Scanner` o la classe `BufferedReader` per leggere da un file di testo
- Quando si legge un file, bisogna sempre verificare se è stata raggiunta la fine del file e in tal caso eseguire le operazioni appropriate. Il modo in cui si può verificare se è stata raggiunta la fine del file dipende dal tipo di file
- Il nome di un file può essere letto da tastiera in una variabile di tipo `String`
- La classe `File` può essere utilizzata per controllare se esiste

un file con un dato nome. Può inoltre essere sfruttata per verificare se il programma ha i permessi per leggere e scrivere

- Per la scrittura nei file binari si può usare la classe `ObjectOutputStream`, mentre per la lettura dei file binari è disponibile la classe `ObjectInputStream`
- È possibile utilizzare il metodo `wrtieObject` della classe `ObjectOutputStream` per scrivere oggetti di tipo classe o array in un file binario. Oggetti e array possono essere letti da un file binario tramite il metodo `readObject` della classe `ObjectInputStream`
- Affinchè si possano utilizzare i metodi `writeObject` della classe `ObjectOutputStream` e `readObject` della classe `ObjectInputStream`, ogni classe le cui istanze vengono scritte nel file deve implementare l'interfaccia `Serializable`

14.7 SLIDE LEZIONE

`java.io` Parziale



`Reader` e `Writer` sono classi astratte per leggere e scrivere stream di caratteri.

`InputStream` e `OutputStream` sono classi astratte per leggere e scrivere stream di valori binari.

- “Filter” aggiunge operazioni a uno stream esistente. Le sottoclassi aggiungono effettivamente le operazioni.
- “Data” definisce metodi per la lettura di valori primitivi.
- “Buffered” usa un buffer per ottimizzare le operazioni di scrittura/lettura.
- “Pushback” permette di annullare operazioni di lettura.
- “Piped” comunicazione diretta tra un sender e un receiver. La pipe in lettura viene collegata alla pipe in scrittura.
- “Printwriter” per la stama di caratteri e stringhe secondo un formato.
- “InputStreamReader”

e “`OutputStreamWriter`” per la lettura e scrittura semplice da stream.

- “File” per lettura e scrittura da file.
- “Object” per lettura e scrittura di oggetti.

I casi evidenziati in giallo sono quelli che considereremo attualmente.

- `System.in`: standard input, in genere dalla tastiera.
- `System.out`: standard output, in genere per il monitor
- `System.err`: standard output per errori, in genere per il monitor

14.7.1 Scanner

Semplice classe per leggere dati di tipo primitivo e stringhe da uno stream

Divide l'input in tokens, delimitati da un separatore.

- Separatore di default: spazi bianchi
- Token convertito in un valore indipendentemente dallo specifico metodo `next` invocato

`FileWriter`: scrittura di caratteri senza formattazione

`PrintWriter`: scrittura di caratteri con formattazione (anche su file binari)

14.7.2 Lettura di un file CSV

Un file Comma-Separated Values (CSV) è un formato per file di testo che memorizzano liste di record.

14.7.3 RandomAccessFile

Abilita l'accesso in lettura/scrittura in posizioni arbitrarie di un file. Stessi metodi di `DataInputStream` e `DataOutputStream`.

Costruttori:

- `RandomAccessFile(File fileObj, String access)`
- `RandomAccessFile(String filename, String access)`

Metodi:

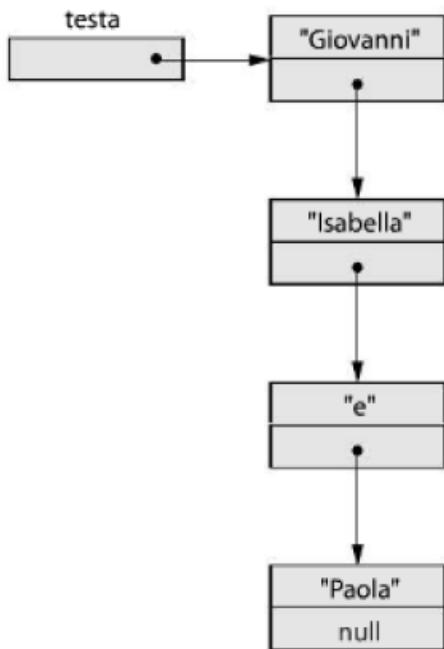
- `void seek(long newPos)`
- `long length()`

15 STRUTTURE DATI DINAMICHE

Una struttura dati concatenata consiste di blocchi di dati, denominati nodi, connessi tra loro tramite dei collegamenti.

15.1 LISTE CONCATENATE

15.1.1 Generalità sulle liste concatenate



Il tipo più semplice di struttura dati concatenata consiste in una singola catena di nodi. Una struttura di questo tipo è chiamata lista concatenata (linked list).

Se si definisce una lista concatenata in Java, i nodi sono realizzati come oggetti di una classe nodo, mentre i collegamenti sono realizzati tramite riferimenti a variabili di istanza della stessa classe dei nodi.

Java fornisce una classe `LinkedList`, che fa parte del package `java.util`. In molti casi ha senso utilizzare questa classe perché è stata progettata e verificata accuratamente.

l'esempio qua accanto mostra una lista semplice.

Il collegamento testa non è nella lista dei nodi, infatti non è un nodo, ma un collegamento che punta al primo nodo. Testa conterrà un riferimento a un nodo, così testa è una variabile del tipo di nodo.

```

1  public class NodoLista {
2      private String dati;
3      private NodoLista collegamento;
4
5      public NodoLista() {
6          collegamento = null;
7          dati = null;
8      }
9
10     public NodoLista(String valoreDati, NodoLista valoreCollegamento) {
11         dati = valoreDati;
12         collegamento = valoreCollegamento;
13     }
14
15     public void setDati(String nuoviDati) {
16         dati = nuoviDati;
17     }
18
19     public String getDati() {
20         return dati;
21     }
22
23     public void setCollegamento(NodoLista nuovoCollegamento) {
24         collegamento = nuovoCollegamento;
25     }
26
27     public NodoLista getCollegamento() {
28         return collegamento;
29     }
30 }
  
```

Ogni nodo è un oggetto di una classe che ha due variabili di istanza: una per i dati e una per il collegamento.

Nell'esempio qua accanto, i dati dei nodi sono costituiti da un valore di tipo `String` (teoricamente non "contiene" la stringa ma solo un riferimento ad essa).

Si noti che la variabile di istanza `collegamento` della classe `NodoLista` è di tipo `NodoLista`.

Quando si utilizza una lista

concatenata, il codice deve essere in grado di collocarsi sul primo nodo e poi di scoprire quando raggiunge l'ultimo nodo. Per raggiungere il primo nodo, si usa una variabile di tipo `NodoLista` che contiene un riferimento al primo nodo. È consuetudine utilizzare il nome `testa` per il nodo di testa.

In java si indica la fine di una lista concatenata impostando a null la variabile di istanza collegamento all'ultimo oggetto.

Per controllare l'uguaglianza di due nodi si utilizza il metodo equals.

Di norma le liste concatenate nascono vuote. Poiché si suppone che la variabile testa contenga solo un riferimento al primo nodo, per indicare una lista vuota si assegna a testa il valore null.

15.1.2 Implementare le operazioni di una lista concatenata

```

1  public class ListaConcatenataDiStringhe {
2      private NodoLista testa;
3
4      public ListaConcatenataDiStringhe() {
5          testa = null;
6      }
7
8      //Mostra i dati della lista.
9      public void mostraLista() {
10         NodoLista posizione = testa;
11         while (posizione != null) {
12             System.out.println(posizione.getDati());
13             posizione = posizione.getCollegamento();
14         }
15     }
16
17     //Restituisce il numero che compongono la lista.
18     public int lunghezza() {
19         int conteggio = 0;
20         NodoLista posizione = testa;
21         while (posizione != null) {
22             conteggio++;
23             posizione = posizione.getCollegamento();
24         }
25         return conteggio;
26     }
27
28     //Aggiunge all'inizio della lista un nodo contenente datiDaAggiungere
29     public void aggiungiNodoInTesta(String datiDaAggiungere) {
30         testa = new NodoLista(datiDaAggiungere, testa);
31     }
32
33     //Elimina il primo nodo sulla lista.
34     public void eliminaNodoDiTesta() {
35         if (testa != null)
36             testa = testa.getCollegamento();
37         else {
38             System.out.println("Si sta eliminando da una lista vuota");
39             System.exit(0);
40         }
41     }
42
43     //Va a vedere se elemento e' nella lista.
44     public boolean nellaLista(String elemento) {
45         return trova(elemento) != null;
46     }
47
48     // Restituisce un riferimento al primo nodo che contiene elemento.
49     // Se elemento non è nella lista, restituisce null.
50     private NodoLista trova(String elemento) {
51         boolean trovato = false;
52         NodoLista posizione = testa;
53         while ((posizione != null) && !trovato) {
54             String datiAllaPosizione = posizione.getDati();
55             if (datiAllaPosizione.equals(elemento))
56                 trovato = true;
57             else
58                 posizione = posizione.getCollegamento();
59         }
60         return posizione;
61     }
62 }
```

Bisogna prestare attenzione alle NullPointerException.

15.1.3 Privacy leak

Ricontrollare prima il capitolo 9.4.1.

La restrizione private sulla variabile di istanza potrebbe essere facilmente aggirata se si invoca sulla variabile di istanza un metodo get. Il problema nasce quando ci sono metodi get pubblici e anche set,

Notare che il listato qua accanto ha solo una variabile di tipo NodoLista nominata testa, impostata a null per dichiarare che la lista è vuota.

Per mostrare la lista si può vedere che nel primo metodo si continuano a stampare i dati e si cambia posizione a catena fino a quando il collegamento è null.

Un processo che si muove da nodo a nodo in una lista concatenata si dice "attraversa la lista".

Il metodo aggiungiNodoInStesta aggiunge un nodo all'inizio della lista concatenata, così che il nuovo nodo diventi il primo della lista. In altre parole si genera una nuova testa che prenderà il posto della precedente mentre la vecchia testa verrà riempita con una variabile.

Il metodo eliminaNodoDiTesta rimuove il primo nodo della lista e fa sì che la variabile testa faccia riferimento a quello che prima era il secondo nodo della lista.

```

1  public class ListaConcatenataDiStringheAutoContenuta {
2      private NodoLista testa;
3
4      public ListaConcatenataDiStringheAutoContenuta() {
5          testa = null;
6      }
7
8      //Mostra i dati della lista
9      public void mostraLista() {
10         NodoLista posizione = testa;
11         while (posizione != null) {
12             System.out.println(posizione.dati);
13             posizione = posizione.collegamento;
14         }
15     }
16
17     //Restituisce il numero di nodi sulla lista.
18     public int lunghezza() {
19         int conteggio = 0;
20         NodoLista posizione = testa;
21         while (posizione != null) {
22             conteggio++;
23             posizione = posizione.collegamento;
24         }
25         return conteggio;
26     }
27
28     //Aggiunge un nodo contenente i dati datiDaAggiungere
29     //all'inizio della lista.
30     public void aggiungiNodoInTesta(String datiDaAggiungere) {
31         testa = new NodoLista(datiDaAggiungere, testa);
32     }
33
34     //Elimina il primo nodo sulla lista.
35     public void eliminaNodoDiTesta() {
36         if (testa != null)
37             testa = testa.collegamento;
38         else {
39             System.out.println("Si sta eliminando da una lista vuota.");
40             System.exit(0);
41         }
42     }
43
44     //Va a vedere se elemento è nella lista.
45     public boolean nellaLista(String elemento) {
46         return trova(elemento) != null;
47     }
48
49     // Restituisce un riferimento al primo nodo che contiene
50     // i dati elemento. Se Se l'elemento non è nella lista, restituisce
51     // null.
52     private NodoLista trova(String elemento) {
53         boolean trovato = false;
54         NodoLista posizione = testa;
55         while ((posizione != null) && !trovato) {
56             String datiAllaPosizione = posizione.dati;
57             if (datiAllaPosizione.equals(elemento))
58                 trovato = true;
59             else
60                 posizione = posizione.collegamento;
61         }
62         return posizione;
63     }
64
65     //Restituisce un array di elementi presenti nella lista.
66     public String[] convertiInArray() {
67         String[] unArray = new String[lunghezza()];
68         NodoLista posizione = testa;
69         int i = 0;
70         while (posizione != null) {
71             unArray[i] = posizione.dati;
72             i++;
73             posizione = posizione.collegamento;
74         }
75         return unArray;
76     }
77
78     private class NodoLista {
79         private String dati;
80         private NodoLista collegamento;
81
82         public NodoLista() {
83             collegamento = null;
84             dati = null;
85         }
86
87         public NodoLista(String valoreDati,
88                           NodoLista valoreCollegamento) {
89             dati = valoreDati;
90             collegamento = valoreCollegamento;
91         }
92     }
93 }

```

Ad esempio nel listato qua accanto si nota come `NodoLista` sia una inner class.

Se non si intende usare una inner class altrove, dovrebbe essere resa privata.

rischiando così la modifica dei dati da parte di terzi. In questo caso non ci sono rischi perché `String` non ha metodi `set` ma per altri casi c'è il rischio. Bisogna stare attenti a quali metodi vengono resi pubblici e quali privati.

Il metodo più semplice per evitare rischi è rendere la classe `NodoLista` una inner class privata della classe `ListaConcatenataDiStringhe`, ovvero collocando entrambe nello stesso package, cambiando la restrizione delle variabili da `private` a `package` e omettere il metodo `getCollegamento`.

15.1.4 Inner class

Le inner class sono classi definite all'interno di altre classi.

Definire una inner class è molto facile: basta includere la sua definizione all'interno di un'altra classe, la classe esterna (`outer class`).

È buona prassi posizionare la definizione della inner class all'inizio oppure alla fine. La inner class non deve essere necessariamente privata.

La inner class è locale alla classe esterna, quindi il nome può essere riutilizzato in altri contesti.

Il vantaggio delle inner class è avere accesso ai metodi delle outer class e viceversa, anche a quelli privati.

15.1.5 Classi nodo come inner class

Si possono definire liste auto-contenute realizzando classi nodo come inner class.

Rendere `NodoLista` una inner class privata è anche più sicuro perché nasconde il metodo `getCollegamento` al mondo esterno ad eccezione della outer class. Se si rendere `NodoLista` una inner class privata si possono eliminare i metodi `get` e `set` e usare un accesso diretto ai suoi dati.

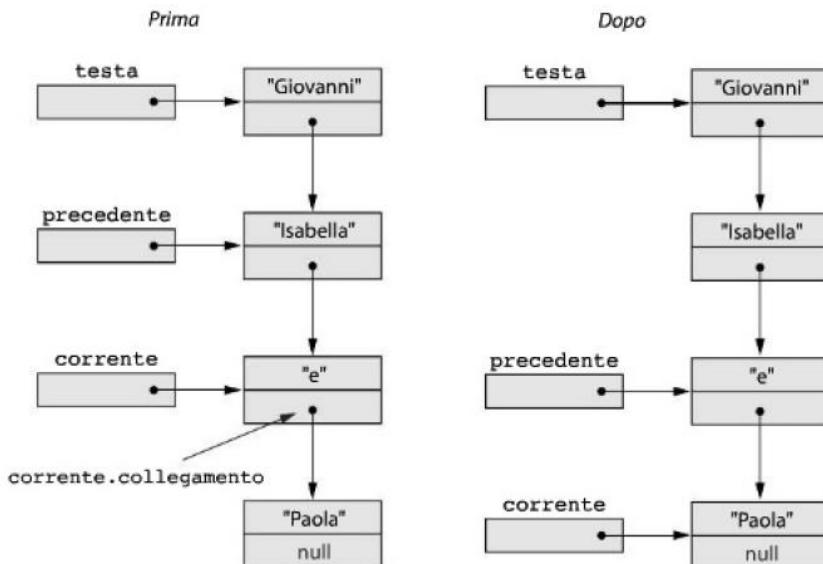
15.1.6 Iteratori

Quando si ha a che fare con una collezione di oggetti, come i nodi di una lista concatenata, spesso occorre attraversare tutti gli oggetti della collezione ed eseguire qualche azione su ogni oggetto.

Un iteratore è una variabile che consente di visitare la lista.

```

1 //Restituisce un array degli elementi presenti nella lista.
2 public String[] convertiInArray() {
3     String[] unArray = new String[lunghezza()];
4     NodoLista posizione = testa;
5     int i = 0;
6     while (posizione != null) {
7         unArray[i] = posizione.dati;
8         i++;
9         posizione = posizione.collegamento;
10    }
11 }
12 }
```



L'iteratore e avrà bisogno dei suoi metodi per modificarla.

L'iterazione avverrà nel modo mostrato qua accanto.

15.1.7 Iteratori interni ed esterni

Un iteratore definito all'interno della classe della lista concatenata è un iteratore interno, altrimenti è esterno.

15.1.8 L'interfaccia Java `Iterator`

Java considera formalmente un iteratore come un oggetto, non semplicemente una variabile.

L'interfaccia chiamata `Iterator` nel package `java.util` stabilisce come si dovrebbe comportare un iteratore. L'interfaccia specifica i tre metodi seguenti:

- `hasNext`: restituisce vero se l'iterazione ha un altro elemento da restituire
- `next`: restituisce il successivo elemento
- `remove`: rimuove dalla collezione l'elemento restituito dall'ultima invocazione `next`

Un array è una collezione di elementi dello stesso tipo di dato. Un iteratore per un array è una variabile di tipo `int`, per chiarezza è l'indice usato per scorrere gli elementi dell'array.

Si può iterare, cioè muoversi da un elemento all'altro, con l'azione `indice++`.

un metodo semplice di iterazione è convertire la lista in un array e iterare su questo. Un array che contiene i dati di una lista concatenata non è però sufficiente se si vuole che un iteratore effettui altre operazioni oltre all'attraversamento.

Ad esempio inserendo due variabili di istanza private `NodoLista "corrente"` e `"precedente"`. Corrente sarà

15.1.9 Gestione delle eccezioni con le liste concatenate

```

1 public class ListaConcatenataException extends Exception {
2     public ListaConcatenataException() {
3         super("Eccezione di lista concatenata");
4     }
5
6     public ListaConcatenataException(String messaggio) {
7         super(messaggio);
8     }
9 }
```

Per consentire al programmatore di fornire un'azione alternativa specifica per queste situazioni anomale, ha più senso sollevare un'eccezione, in modo da lasciar decidere al programmatore come gestire la situazione.

15.2 VARIANTI DELLE LISTE CONCATENATE

15.2.1 Liste concatenate doppie

Una lista concatenata ordinaria può essere percorsa in un'unica direzione, **una lista concatenata doppia ha un collegamento al nodo seguente ma anche precedente, permettendo di scorrere in entrambi i versi.**

Rispetto ad una lista concatenata ordinaria, costruttori e metodi andranno rivisti per la gestione del collegamento aggiuntivo, le modifiche più rilevanti saranno per l'aggiunta e l'eliminazione dei nodi.

15.2.2 Pile

Una pila (stack) non è necessariamente una struttura dati concatenata, ma può essere implementata per mezzo di una lista concatenata. Una pila è una struttura dati che **segue il metodo last-in/first-out (LIFO).**

Si possono aggiungere elementi in cima alla lista (azione detta `push`) o rimuoverli (azione detta `pop`).

15.2.3 Code

Mentre una pila usa il metodo LIFO per la gestione dell'ordine, **una coda usa il sistema FIFO, ovvero first-in/first-out.** La caratteristica di una coda è che deve tenere traccia sia della testa che della fine della lista poiché le azioni o rimuoveranno la testa creandone una nuova o aggiungeranno un elemento alla cosa.

15.3 TABELLE DI HASH

Una tabella di hash, oppure mappa di hash, è una struttura che immagazzina e recupera dati in memoria in maniera efficiente.

Ci sono vari modi per costruire una tabella di hash, in questo caso useremo la combinazione di un array e di una lista concatenata ordinaria.

Mentre la ricerca in una lista ordinaria solitamente richiede un tempo lineare in base alla grandezza della lista, la ricerca in una tabella di hash può potenzialmente eseguire un numero fisso di passi.

Per immagazzinare un elemento in una tabella di hash gli si assegna una chiave. Nota la chiave è possibile recuperare l'elemento.

Se un elemento non fornisce intrinsecamente una chiave univoca, si può utilizzare una funzione di hash per generarne una. Nella maggior parte dei casi, una funzione di hash genera un numero.

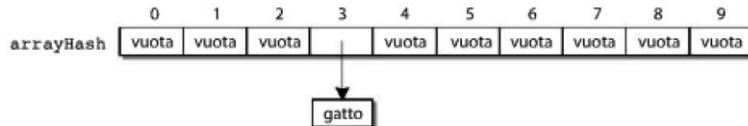
Immaginiamo una tabella di hash per un dizionario. La chiave assocerà ogni parola all'indice corrispondente nell'array nel caso la parola non sia presente, probabilmente è stata scritta in maniera scorretta.

L'implementazione vede un array di liste concatenate.

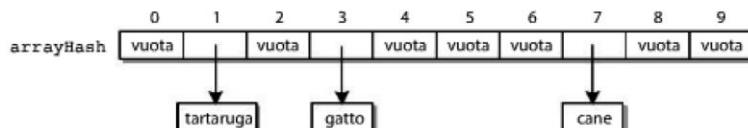
1. Tabella di hash esistente inizializzata con 10 liste concatenate vuote

```
arrayHash = new ListaConcatenataDiStringhe[DIMENSIONE]; // DIMENSIONE = 10
arrayHash[0] = vuota; arrayHash[1] = vuota; arrayHash[2] = vuota; arrayHash[3] = vuota; arrayHash[4] = vuota; arrayHash[5] = vuota; arrayHash[6] = vuota; arrayHash[7] = vuota; arrayHash[8] = vuota; arrayHash[9] = vuota;
```

2. Dopo l'aggiunta di "gatto" con hash pari a 3



3. Dopo l'aggiunta di "cane" con hash 7 e "tartaruga" con hash 1



4. Dopo l'aggiunta di "uccello" con hash 3 – collisione e collegamento alla lista concatenata con "gatto"

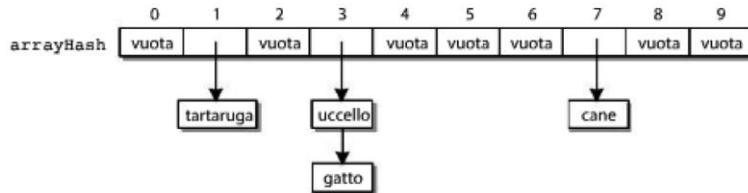


Figura 15.11 Costruzione di una tabella di hash.

```
1 public class TabellaHash {
2
3     //ListaConcatenataDiStringhe già data
4     private ListaConcatenataDiStringhe[] arrayHash;
5     private static final int DIMENSIONE = 10;
6
7     public TabellaHash() {
8         arrayHash = new ListaConcatenataDiStringhe[DIMENSIONE];
9         for (int i = 0; i < DIMENSIONE; i++)
10             arrayHash[i] = new ListaConcatenataDiStringhe();
11     }
12
13     private int calcolaHash(String s) {
14         int hash = 0;
15         for (int i = 0; i < s.length(); i++) {
16             hash += s.charAt(i);
17         }
18         return hash % DIMENSIONE;
19     }
20
21     //Restituisce true se obiettivo è nella tabella,
22     //false altrimenti.
23     public boolean contieneStringa(String obiettivo) {
24         int hash = calcolaHash(obiettivo);
25         ListaConcatenataDiStringhe lista = arrayHash[hash];
26         if (lista.nellaLista(obiettivo))
27             return true;
28         return false;
29     }
30
31     //Salva la stringa s nella tabella di hash
32     public void aggiungi(String s) {
33         int hash = calcolaHash(s); // Calcola il valore di hash
34         ListaConcatenataDiStringhe lista = arrayHash[hash];
35         if (!lista.nellaLista(s)) {
36             // Aggiunge la stringa solo se non è già
37             // nella lista
38             lista.aggiungiNodoInTesta(s);
39         }
40     }
41 }
```

peggiori si avranno se tutti gli elementi inseriti vengono associati alla stessa chiave. Tuttavia se gli elementi

Nel caso due elementi abbiano

lo stesso indice (chiave) si creerà una collisione come si vede all'indice 3. Si effettua quindi una concatenazione, cioè si aggiunge semplicemente un nuovo nodo alla lista concatenata già presente.

Per cercare un elemento da una tabella di hash, per prima cosa si calcola il suo valore di hash, successivamente si cerca, per mezzo di una ricerca sequenziale, l'elemento nella lista concatenata presente nella posizione dell'array di indice pari al valore di hash ottenuto.

15.3.1 Una funzione di hash per le stringhe

Un modo semplice per calcolare il valore di hash di una stringa è quello di sommare i codici ASCII di ogni carattere della stringa e calcolare il resto della divisione della somma per la lunghezza dell'array.

Nella pratica, la lunghezza dell'array viene solitamente scelta pari a un numero primo maggiore del numero di elementi che saranno immagazzinati nella tabella di hash. L'utilizzo di un numero primo è per evitare che escano fattori comuni nel resto della divisione che potrebbero produrre collisioni (ad esempio due stringhe differenti ma con valori 20 e 30 su di un array di 10 elementi produrrebbero entrambe 0).

15.3.2 Efficienza delle tabelle di hash

Per prima cosa, è utile analizzare due casi estremi. Le prestazioni

```

1 // Utilizza una lista concatenata come struttura dati
2 // interna per immagazzinare gli elementi di un insieme
3 public class Insieme<T> {
4     private class Nodo<T> {
5         private T dati;
6         private Nodo<T> collegamento;
7
8         public Nodo() {
9             dati = null;
10            collegamento = null;
11        }
12        public Nodo(T nuoviDati, Nodo<T> valoreCollegamento) {
13            dati = nuoviDati;
14            collegamento = valoreCollegamento;
15        }
16    } // Fine della inner class Nodo<T>
17
18    private Nodo<T> testa;
19
20    public Insieme() {
21        testa = null;
22    }
23
24    //Aggiunge un nuovo elemento all'insieme.
25    //Se l'elemento è già presente, restituisce false,
26    //altrimenti restituisce true.
27    public boolean aggiungi(T nuovoElemento) {
28        if (!nellaInsieme(nuovoElemento)) {
29            testa = new Nodo<T>(nuovoElemento, testa);
30            return true;
31        }
32        return false;
33    }
34
35    public boolean nellaInsieme(T elemento) {
36        Nodo<T> posizione = testa;
37        T elementoAllaPosizione;
38
39        while (posizione != null) {
40            elementoAllaPosizione = posizione.dati;
41            if (elementoAllaPosizione.equals(elemento))
42                return true;
43            posizione = posizione.collegamento;
44        }
45        return false; // l'elemento non è stato trovato
46    }
47
48    public void mostraInsieme() {
49        Nodo<T> posizione = testa;
50        while (posizione != null) {
51            System.out.print(posizione.dati.toString() + " ");
52            posizione = posizione.collegamento;
53        }
54        System.out.println();
55    }
56
57    //Restituisce un nuovo insieme corrispondente all'unione
58    //dell'insieme con l'insieme specificato.
59    public Insieme<T> unione(Insieme<T> altroInsieme) {
60        Insieme<T> insiemeUnione = new Insieme<T>();
61        // Copia l'insieme corrente nell'unione
62        Nodo<T> posizione = testa;
63        while (posizione != null) {
64            insiemeUnione.aggiungi(posizione.dati);
65            posizione = posizione.collegamento;
66        }
67
68        // Copia l'altro insieme nell'unione.
69        // Il metodo aggiungi elimina automaticamente
70        // i duplicati.
71        posizione = altroInsieme.testa;
72        while (posizione != null) {
73            insiemeUnione.aggiungi(posizione.dati);
74            posizione = posizione.collegamento;
75        }
76        return insiemeUnione;
77    }
78
79    //Restituisce un nuovo insieme corrispondente all'intersezione
80    //tra l'insieme e l'insieme specificato.
81    public Insieme<T> intersezione(Insieme<T> altroInsieme) {
82        Insieme<T> insiemeIntersezione = new Insieme<T>();
83        // Copia solo gli elementi presenti in entrambi
84        // gli insiempi.
85        Nodo<T> posizione = testa;
86        while (posizione != null) {
87            if (altroInsieme.nellaInsieme(posizione.dati))
88                insiemeIntersezione.aggiungi(posizione.dati);
89            posizione = posizione.collegamento;
90        }
91        return insiemeIntersezione;
92    }
93
94    public int numeroElementi() {
95        int conteggio = 0;
96        Nodo<T> posizione = testa;
97        while (posizione != null) {
98            conteggio++;
99            posizione = posizione.collegamento;
100       }
101       return conteggio;
102   }

```

da distribuire hanno una distribuzione casuale sarà difficile come caso. Il miglioramento della funzione di hash riduce anche i casi di concatenazione. Ad esempio per il caso di prima “attore” e “teatro” generano la stessa chiave, si potrebbe risolvere quindi moltiplicando ogni valore ascii per un peso che cresce in base alla posizione della lettera come nell'esempio qua accanto.

Un altro modo per ridurre la probabilità di collisioni è aumentare la dimensione della tabella di hash. Uno svantaggio di questo approccio è però lo spreco di memoria.

15.4 INSIEMI

Un insieme è una collezione di elementi dei quali si ignorano ordine e molteplicità. Un modo semplice di implementare un insieme è una variante di una lista concatenata. Gli elementi dell'insieme vengono immagazzinati in una lista concatenata ordinaria. Poiché una lista concatenata contiene riferimenti agli elementi dell'insieme, è possibile includere lo stesso elemento in più insiemi referenziandolo da più liste concatenate.

15.4.1 Operazioni di base sugli insiemi

Le operazioni di base che una classe che implementi un insieme dovrebbe consentire sono le seguenti:

- Aggiunta di un elemento
- Verifica dell'appartenenza di un elemento dell'insieme
- Unione
- Intersezione

Sarebbe inoltre opportuno fornire un iteratore per estrarre gli elementi dell'insieme.

15.5 ALBERI

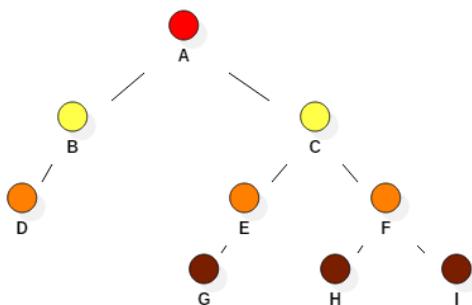
Un albero è un esempio di struttura dati concatenata di tipo più complesso.

15.5.1 Proprietà degli alberi

```

1  public class AlberoInteri {
2      private class NodoAlberoInt {
3          private int dati;
4          private NodoAlberoInt collegamentoSinistro;
5          private NodoAlberoInt collegamentoDestro;
6
7          public NodoAlberoInt(int nuoviDati, NodoAlberoInt
8              nuovoCollegamentoSinistro, NodoAlberoInt nuovoCollegamentoDestro) {
9              dati = nuoviDati;
10             collegamentoSinistro = nuovoCollegamentoSinistro;
11             collegamentoDestro = nuovoCollegamentoDestro;
12         }
13     } // Fine della inner class NodoAlberoInt
14
15     private NodoAlberoInt radice;
16
17     public AlberoInteri() {
18         radice = null;
19     }
20
21     public void aggiungi(int elemento) {
22         inserisciInSottoAlbero(elemento, radice);
23     }
24
25     public boolean nellAlbero(int elemento) {
26         return nelSottoAlbero(elemento, radice);
27     }
28
29     public void mostraAlbero() {
30         mostraSottoAlbero(radice);
31     }
32
33     //Restituisce il nodo radice dell'albero avente come radice radiceSottoAlbero
34     //e nel quale è stato inserito un nodo con il nuovo elemento specificato.
35     private NodoAlberoInt inserisciInSottoAlbero(int elemento,
36                                                 NodoAlberoInt radiceSottoAlbero) {
37         if (radiceSottoAlbero == null)
38             return new NodoAlberoInt(elemento, null, null);
39         else if (elemento < radiceSottoAlbero.dati) {
40             radiceSottoAlbero.collegamentoSinistro =
41                 inserisciInSottoAlbero(elemento,
42                                     radiceSottoAlbero.collegamentoSinistro);
43             return radiceSottoAlbero;
44         } else { // elemento >= radiceSottoAlbero.dati
45             radiceSottoAlbero.collegamentoDestro =
46                 inserisciInSottoAlbero(elemento,
47                                     radiceSottoAlbero.collegamentoDestro);
48             return radiceSottoAlbero;
49         }
50     }
51
52     private static boolean nelSottoAlbero(int elemento,
53                                         NodoAlberoInt radiceSottoAlbero) {
54         if (radiceSottoAlbero == null)
55             return false;
56         else if (radiceSottoAlbero.dati == elemento)
57             return true;
58         else if (elemento < radiceSottoAlbero.dati)
59             return nelSottoAlbero(elemento,
60                                 radiceSottoAlbero.collegamentoSinistro);
61         else // elemento >= radiceSottoAlbero.dati
62             return nelSottoAlbero(elemento,
63                                 radiceSottoAlbero.collegamentoDestro);
64     }
65
66     private static void mostraSottoAlbero(NodoAlberoInt radiceSottoAlbero) {
67         if (radiceSottoAlbero != null) {
68             mostraSottoAlbero(radiceSottoAlbero.collegamentoSinistro);
69             System.out.print(radiceSottoAlbero.dati + " ");
70             mostraSottoAlbero(radiceSottoAlbero.collegamentoDestro);
71         } // altrimenti non fare niente:
72         // un albero vuoto non ha elementi da mostrare
73     }
    
```

- Sottoalbero destro
- Nodo radice



Preorder: ABDCEGFHI

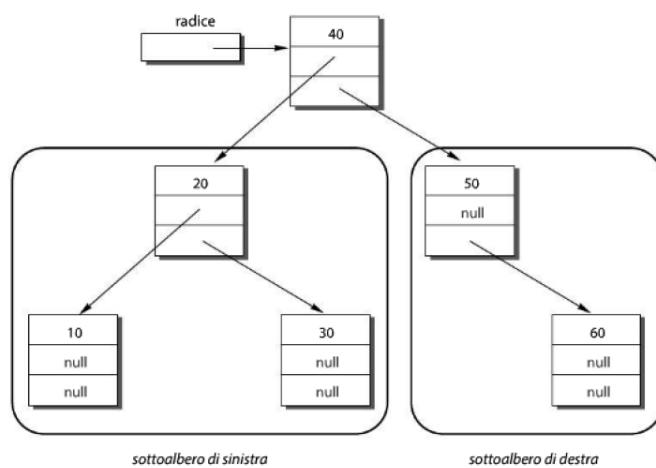
Inorder: BDAGECHFI

Postorder: DBGEHIFCA

Un albero che rispetta i seguenti criteri della regola di immagazzinamento per la ricerca in alberi binari è detto albero di ricerca binaria:

- Tutti i valori del sottoalbero sinistro sono minori del valore nel nodo radice

Riassunto di Jacopo De Angelis



- Tutti i valori del sottoalbero destro sono maggiori o uguali del valore nel nodo radice
- La regola si applica ricorsivamente

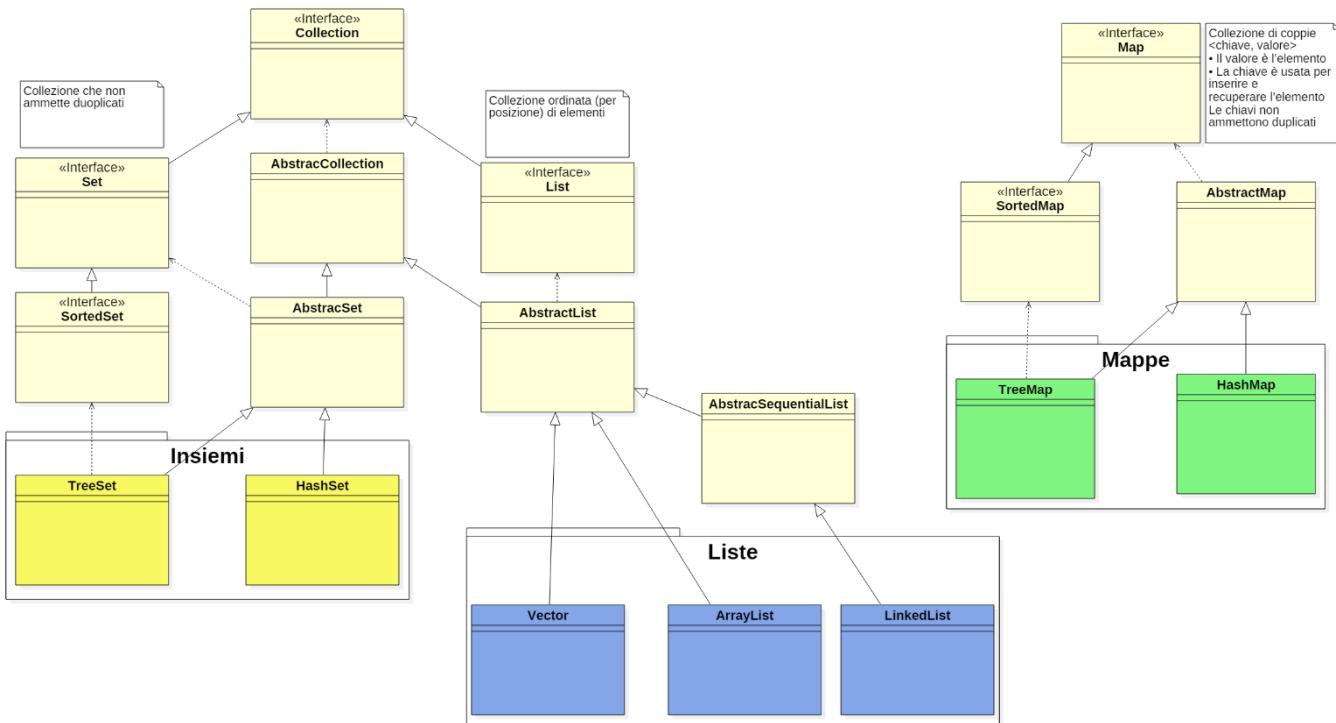
Ad esempio quest' albero è un albero di ricerca binaria.

16 SLIDE: JAVA COLLECTION FRAMEWORK (PARTE DEL CAPITOLO 16 DEL LIBRO)

Una collezione è un elemento “contenitore” di oggetti denominati elementi. La necessità di raggruppare elementi è molto comune (mail folder, rubrica, carrello, ecc.).

Java collection framework: insieme di interfacce e di classi per l’implementazione di collezioni.

- Interfacce e classi astratte per rappresentare tipi di collezioni
- Implementazioni concrete pronte per essere utilizzate



16.1 L’INTERFACCIA COLLECTION

La dichiarazione di tipo usa “`Collection<E>`”, ad esempio:

- `Collection<Object> c1;`
- `Collection<Animale> c2;`

alcuni metodi dell’interfaccia:

- `Add ([tipo Oggetto scelto] e): aggiunge e alla collezione`
- `Clear (): svuota la collezione`
- `contains (Object o): ritorna true se esiste un elemento e tale che e.equals(o)`
- `remove (Object o): rimuove l’elemento o (identificato tramite equals) dalla collezione`
- `size (): ritorna il numero degli elementi della collezione`
- `toArray (): ritorna un array con gli elementi della collezione`

16.2 L’INTERFACCIA SET

`Set` extends `Collection`: è una `Collection` che non ammette elementi duplicati, la duplicazione viene individuata eseguendo il metodo `equals`.

Dichiarazione di tipo usa “Set<E>”, ad esempio:

- Set<Object> s1;
- Set<Animale> s2;

non aggiunge metodi a Collection, ma si aspetta una diversa implementazione (diversa semantica) per alcuni di essi:

- Add ([tipo Oggetto scelto] e): aggiunge e alla collezione se non già presente

16.3 L'INTERFACCIA LIST

List extends Collection: è una collezione con elementi ordinati la cui posizione è individuata da un indice intero.

Dichiarazione di tipo usa “List<E>”, ad esempio:

- List<Object> l1;
- List<Animale> l2;

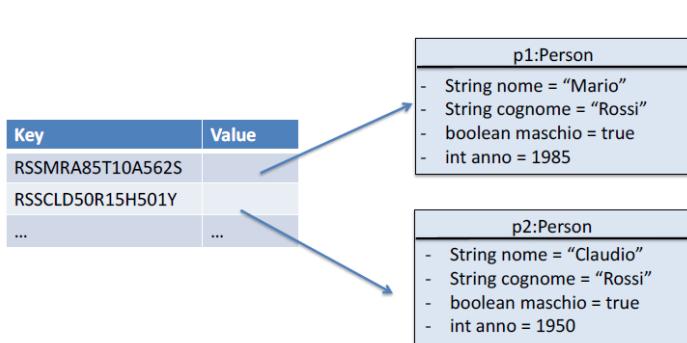
Cambia la semantica di alcuni metodi:

- add ([tipo Oggetto scelto] e): aggiunge e in fondo alla collezione se non già presente
- remove (Object o): rimuove il primo elemento o (identificato tramite equals) dalla collezione

aggiunge nuovi metodi per lavorare con un insieme ordinato:

- add(int index, ([tipo Oggetto scelto] e): aggiunge e alla posizione index
- [tipo Oggetto scelto] get(int index) ritorna l'elemento alla posizione index
- int indexOf(Object o): ritorna la posizione della prima occorrenza di un elemento e tale che e.equals(o), oppure -1
- int lastIndexOf(Object o): come indexOf ma ritorna l'ultima occorrenza
- remove(int index): rimuove l'elemento alla posizione index
- set(int index, [tipo Oggetto scelto] element): sostituisce l'elemento della collezione alla posizione index con element
- List<[tipo Oggetto scelto]> sublist(int fromIndex, int toIndex): ritorna la porzione della lista che va da fromIndex (incluso) a toIndex (escluso)

16.4 L'INTERFACCIA MAP



La dichiarazione di tipo usa “Map<K, V>”, ad esempio:

- Mappa codiceFiscale → Persona
 - Map<String, Persona> rubrica;
- Mappa idProdotto → prodotto
 - Map<String, Prodotto> catalogo;

cercare, aggiungere, modificare ed eliminare gli elementi della collezione per chiave.

Una chiave può occorrere al più una volta, ad esempio `k1` è una chiave duplicata se `map.containsKey(k1) == true`, implica esiste `k2` tra le chiavi già presenti tale che `k2.equals(k1) == true`.

Alcuni metodi dell'interfaccia:

- `clear()`: svuota la mappa
- `containsKey(Object key)`: ritorna `true` se esiste una chiave `k` tale che `k.equals(key)`
- `containsValue(Object value)`: ritorna `true` se esiste almeno un valore `v` tale che `v.equals(value)`
- `V get(Object key)`: ritorna l'elemento associato alla chiave `key`, oppure `null`
- `Set<K> keyset()`: ritorna un set con tutte le chiavi
- `put(K key, V value)`: aggiunge `value` con chiave `key`, nel caso `key` esista già l'elemento viene sostituito da `value`
- `V putIfAbsent(K key, V value)`: se la chiave non esiste già funziona come `put` e ritorna `null`, altrimenti ritorna il valore già associato a `key` nella mappa senza modificarla
- `remove(Object key)`: rimuove l'elemento associato alla chiave `key`
- `remove(Object key, Object value)`: rimuove l'elemento associato alla chiave `key` solamente se già presente con valore `value`
- `size()`: ritorna il numero di coppia (chiave, valore) nella mappa
- `Collection<V> values()`: ritorna una collection con i valori della mappa

Perché l'insieme delle `key` è estratto come un `Set` e invece l'insieme dei valori come una `collection`?

Perché le chiavi sono uniche, non possono ripetersi (caratteristica delle liste) e non presuppone un ordine, nelle collezioni i valori possono ripetersi e hanno un ordine dato dall'indice.

Il caso specifico è se viene implementata l'interfaccia `SortedSet` allora gli elementi saranno memorizzati in modo crescente.

Se una classe implementa l'interfaccia `SortedMap` allora le coppie <chiave, valore> saranno memorizzate in modo crescente di chiave.

16.5 ORDINAMENTO

Gli elementi nelle collezioni e le chiavi possono essere oggetti qualsiasi. Come facciamo ad ordinare in modo crescente delle persone, delle auto, degli animali, ecc.?

Gli oggetti di una classe possono essere ordinati se la classe implementa l'interfaccia `Comparable`.

L'interfaccia prescrive l'implementazione di un unico metodo `compareTo`. Dati `o1` e `o2` istanziati da una classe che implementa l'interfaccia `Comparable`, `o1.compareTo(o2)` vale:

- `O` se sono "uguali"
- `< 0` se `o1` è più piccolo di `o2`
- `> 0` se `o1` è più grande di `o2`

Numerose altre classi Java implementano questa interfaccia, ad esempio `Date`, `Integer`, `String`, ecc.

Quando si implementa questa interfaccia bisogna scegliere il tipo con cui effettuare il confronto, tipicamente lo stesso tipo della classe:

```
public class [nome_classe] implements Comparable<[nome_classe_confronto]>.
```

16.5.1 Coerenza compareTo e equals

`e1.compareTo(e2) == 0` \leftrightarrow `e1.equals(e2) == true`

se non si mantiene la coerenza, le collezioni ordinate possono avere un comportamento prevedibile.

16.6 L'INTERFACCIA SORTEDSET

SortedSet extends Set: è un Set totalmente ordinato.

La dichiarazione di tipo usa “SortedSet<E>”, ad esempio:

- `SortedSet<Object> s1;`
- `SortedSet<Animale> s2;`

aggiunge alcuni metodi a Set:

- `[nome_classe_oggetto] first():` ritorna il primo elemento dell'insieme
- `SortedSet<[nome_classe_oggetto]> headSet([nome_classe_oggetto] toElement):` ritorna una vista dell'insieme contenente tutti gli elementi strettamente minori di toElement
- `[nome_classe_oggetto] last():` ritorna l'ultimo elemento dell'insieme
- `SortedSet<[nome_SortedSet]> subSet([nome_classe_oggetto] fromElement, [nome_classe_oggetto] toElement):` ritorna una vista dell'insieme contenente tutti gli elementi da fromElement a toElement(escluso)
- `SortedSet<[nome_classe_oggetto]> tailSet([nome_classe_oggetto] fromElement):` ritorna una vista dell'insieme contenente tutti gli elementi di valore maggiore o uguale a fromElement

16.7 L'INTERFACCIA SORTEDMAP

SortedMap extends Map: è una mappa con le chiavi totalmente ordinate.

La dichiarazione di tipo usa “SortedMap <K, V>”, ad esempio:

- `SortedMap<String, Persona> rubrica;`
- `SortedMap<String, Prodotto> catalogo;`

aggiunge alcuni metodi a Map:

- `K firstKey():` ritorna la prima chiave
- `SortedMap<K, V> headMap(K toKey):` ritorna una vista della mappa contenente tutti gli elementi con chiave strettamente minore di toKey
- `K lastKey():` ritorna l'ultimo valore della chiave
- `SortedMap<K, V> subMap(K fromKey, K toKey):` ritorna una vista della mappa contenente tutti gli elementi da fromKey a toKey (escluso)
- `SortedMap<K, V> tailMap(K fromKey):` ritorna una vista della mappa contenente tutti gli elementi di chiave maggiore o uguale a fromKey

16.8 IMPLEMENTAZIONI DISPONIBILI

Intefacce	Implementazioni disponibili			
	Hash table	Array ridimensionabile	Albero ordinato	Liste concatenate
Set	HashSet		TreeSet	
List		ArrayList (e vettori)		LinkedList
Map	HashMap		TreeMap	

16.8.1 Esempio ArrayList e LinkedList

```

1 import java.util.ArrayList;
2 import java.util.LinkedList;
3 import java.util.List;
4
5
6 public class Zoo {
7     private List<Animale> animali;
8 //    private Animale animali[];
9
10    public Zoo(int maxNumAnimali) {
11 //        animali = new Animale[maxNumAnimali];
12 //        animali = new ArrayList<Animale>();
13 //    per usare LinkedList
14 //        animali = new LinkedList<Animale>();
15    }
16
17    public boolean aggiungiAnimale(Animale animale) {
18 /*        for(int i=0; i<animali.length; i++) {
19             if (animali[i]==null) {
20                 animali[i]=animale;
21                 return true;
22             }
23         }
24         return false;
25 */
26         animali.add(animale);
27         return true;
28     }
29
30    public void parla() {
31 /*        for(int i=0; i<animali.length; i++) {
32             if (animali[i] != null) System.out.println(animali[i].verso());
33         }
34 */
35        for(int i=0; i<animali.size(); i++) {
36            System.out.println(animali.get(i).verso());
37        }
38    }
39
40    public void sbrana() {
41 /*        for(int i=0; i<animali.length; i++) {
42             if (animali[i] != null && animali[i] instanceof Felino)
43                 System.out.println(((Felino)animali[i]).sbrana());
44         }
45 */
46        for(int i=0; i<animali.size(); i++) {
47            if (animali.get(i) instanceof Felino) System.out.println(((Felino)
48                animali.get(i)).sbrana());
49        }
50    }
51

```

Si può notare come gli array (nei commenti) siano stati sostituiti da Liste.

16.8.2 Esempio TreeSet

```
1 import java.util.HashSet;
2 import java.util.Set;
3 import java.util.TreeSet;
4
5 public class MyBasicTreeSet {
6
7     public static void main(String a[]){
8
9         Set<String> ts = new TreeSet<String>();
10        //Set<String> ts = new HashSet<String>();
11
12        ts.add("one");
13        ts.add("two");
14        ts.add("three");
15        System.out.println("Elements: "+ts);
16        //check is set empty?
17        System.out.println("Is set empty: "+ts.isEmpty());
18        //delete all elements from set
19        ts.clear();
20        System.out.println("Is set empty: "+ts.isEmpty());
21        ts.add("one");
22        ts.add("two");
23        ts.add("three");
24        System.out.println("Size of the set: "+ts.size());
25        //remove one string
26        ts.remove("two");
27        System.out.println("Elements: "+ts);
28        ts.add("one");
29        ts.add("three");
30        System.out.println("Size of the set: "+ts.size());
31
32    }
33 }
```

16.8.3 Esempio TreeMap

```

1 import java.util.Collection;
2 import java.util.Map;
3 import java.util.TreeMap;
4
5 public class Carrello {
6
7     //private Prodotto prodotti[];
8     private Map<String, Prodotto> prodotti;
9
10    public Carrello(int numProdottiMax) {
11        /*      if (numProdottiMax<=0) {
12            prodotti = new Prodotto[5];
13        }
14        prodotti = new Prodotto[numProdottiMax];
15    */
16        // Anche il parametro del costruttore è inutile
17        prodotti = new TreeMap<String, Prodotto>();
18    }
19
20    public boolean aggiungiProdotto(Prodotto p) {
21        /*      for (int i=0; i<prodotti.length; i++) {
22            if (prodotti[i]==null) {
23                prodotti[i]=p;
24                return true;
25            }
26        }
27        return false;*/
28        prodotti.put(p.getCodice(), p);
29        return true;
30    }
31
32    public Prodotto getProdotto(String codice) {
33        /*      for(int i=0; i< prodotti.length; i++) {
34            if ((prodotti[i]!=null) && (prodotti[i].getCodice().equals
35                (codice))) {
36                return prodotti[i];
37            }
38        return null;*/
39        return prodotti.get(codice);
40    }
41
42    public boolean esisteProdotto(Prodotto p) {
43        /*      for (int i=0; i<prodotti.length; i++) {
44            if (prodotti[i]!=null && prodotti[i].equals(p)) {
45                return true;
46            }
47        }
48        return false;*/
49
50        //Se i codici sono univoci uso il codice invece del prodotto
51        return prodotti.containsKey(p.getCodice());
52    }
53
54    public String toString() {
55        /*      String s = "Carrello con: \n";
56        for(int i=0; i<prodotti.length && prodotti[i] != null; i++) {
57            s+=prodotti[i].toString() + "\n";
58        }
59        return s; */
60        return prodotti.toString();
61    }
62 }

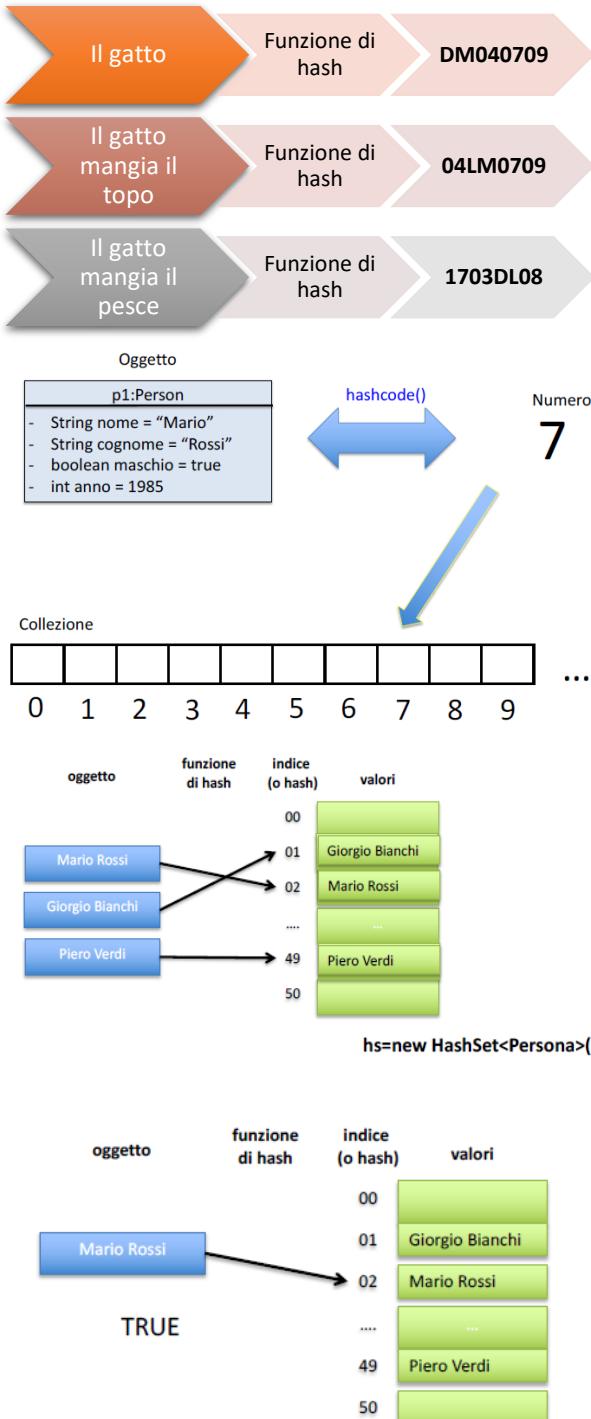
```

Prodotti nel carrello memorizzati in una mappa dove il codice prodotto è la chiave e il prodotto è il valore.

Il costo di cercare/aggiungere/eliminare un elemento non dipende dal numero degli elementi nella collezione ma solo dal costo della trasformazione matematica.

L'idea funziona bene se ogni elemento va in una posizione differente, quindi ogni casella è vuota oppure è piena con l'unico elemento che può essere aggiunto in quel punto. Le caselle vuote non necessariamente devono essere create in memoria.

16.9 HASH



Funzione di hash: una funzione che trasforma dei dati di lunghezza arbitraria in dati di lunghezza fissa.
Individuare e recuperare dati di lunghezza fissa è più semplice che lavorare con dati di lunghezza variabile.
Spesso il dato a lunghezza fissa è un numero a n bit.

La classe Object implementa il metodo int hashCode() che ritorna il codice hash di un oggetto. L'implementazione si basa sull'indirizzo dell'oggetto.

16.9.1 HashSet

`hashcode()` viene utilizzato per memorizzare e recuperare in modo efficiente gli oggetti del Set

- `hs.add(new Persona("Mario", "Rossi"));`
- `hs.add(new Persona("Giorgio", "Bianchi"));`
- `hs.add(new Persona("Piero", "Verdi"));`

Per la ricerca in tempo costante:

```
contains(new Persona("Mario", "Rossi"));
```

16.9.2 hashCode()

Scrivere funzioni di hash perfette è difficile.
Possono esistere due valori diversi associati allo stesso codice hash. Una buona funzione minimizza il numero di volte in cui questo accade.

Proprietà che `hashCode()` deve soddisfare nella pratica:

- `o.hashCode()` ritorna sempre lo stesso risultato quando invocato all'interno di un programma
- se `o1.equals(o2)` allora `o1.hashCode() == o2.hashCode()`

- se o1.hashCode() == o2.hashCode(), non necessariamente o1.equals(o2)

due oggetti uguali hanno sempre lo stesso hashCode() ma non è vero il contrario, quindi due oggetti con lo stesso hashCode() possono essere differenti.

Cosa accade se due oggetti hanno lo stesso hashCode? Si genera un conflitto, per risolverlo bisogna fare una scansione lineare degli elementi, quindi è bene definire delle funzioni di hashCode che minimizzino il numero di conflitti.

16.9.3 Esempio HashSet

```

1 import java.util.HashSet;
2 import java.util.Set;
3 import java.util.TreeSet;
4
5 public class MyBasicTreeSet {
6
7     public static void main(String a[]){
8
9         //Set<String> ts = new TreeSet<String>();
10        Set<String> ts = new HashSet<String>();
11
12        ts.add("one");
13        ts.add("two");
14        ts.add("three");
15        System.out.println("Elements: "+ts);
16        //check is set empty?
17        System.out.println("Is set empty: "+ts.isEmpty());
18        //delete all elements from set
19        ts.clear();
20        System.out.println("Is set empty: "+ts.isEmpty());
21        ts.add("one");
22        ts.add("two");
23        ts.add("three");
24        System.out.println("Size of the set: "+ts.size());
25        //remove one string
26        ts.remove("two");
27        System.out.println("Elements: "+ts);
28        ts.add("one");
29        ts.add("three");
30        System.out.println("Size of the set: "+ts.size());
31
32    }
33 }
```

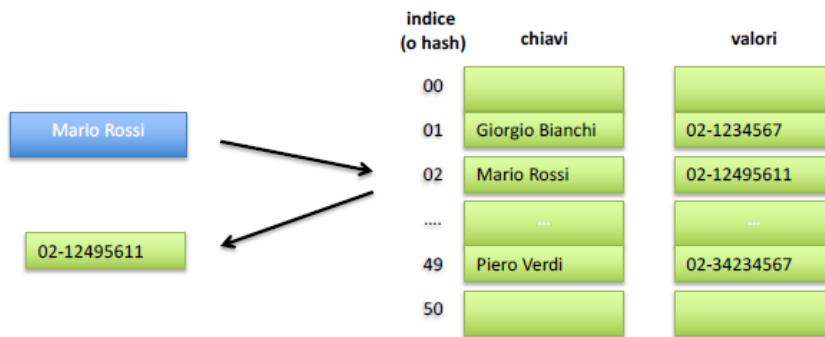
16.9.4 HashMap

hashCode() viene utilizzato per memorizzare e recuperare in modo efficiente gli oggetti della mappa.

- Put(new Persona(“Mario”, “Rossi”), “01-1239611”);
- Put(new Persona(“Giorgio”, “Bianchi”), “02-1234567”);
- Put(new Persona(“Piero”, “Verdi”), “02-34234567”);

data la chiave, il valore può essere recuperato in tempo costante.

```
String telefono = get(new Persona("Mario", "Rossi"))
```



16.9.5 Override di hashCode()

Possiamo reimplementare il metodo se abbiamo un buon algoritmo.

Se implementiamo equals() dobbiamo implementare hashCode(), altrimenti le collezioni che usano la funzione di hash non funzionerebbero correttamente. Quindi fin tanto che gli oggetti non sono inseriti in una collezione che usa una HashTable possiamo evitare di implementare hashCode() anche se abbiamo implementato equals().

Ricordarsi che due oggetti uguali secondo equals() devono produrre lo stesso hashCode().

16.10 ESEMPIO: PROGETTO ACCOUNT

Intefacce	Implementazioni disponibili			
	Hash table	Array ridimensionabile	Albero ordinato	Liste concatenate
Set	HashSet		TreeSet	
List		ArrayList (e vettori)		LinkedList
Map	HashMap		TreeMap	

Set				
	Add	Remove	Get	Contains
HashSet	$\Theta(1)$ caso medio $O(n)$ caso peggiore			
TreeSet	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

List					
	Add(E)	Add(E, i)	Remove(i)	Get(i)	Contains
ArrayList	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$
LinkedList	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$

Map				
	Add	Remove	Get	Contains
HashMap	$\Theta(1)$ caso medio $O(n)$ caso peggiore			
TreeMap	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

```

1 public class Account
2 {
3     private int accountNumber;
4     private String holderName;
5
6     public Account(int accountNumber, String holderName) {
7         this.accountNumber = accountNumber;
8         this.holderName = holderName;
9     }
10
11    public String getHolderName() {
12        return holderName;
13    }
14
15    public int getAccountNumber() {
16        return accountNumber;
17    }
18
19    @Override
20    public String toString() {
21        return "Name " + holderName + " account:" + accountNumber;
22    }
23
24    @Override
25    public int hashCode() {
26        final int prime = 31;
27        int result = 1;
28
29        //uso di accountNumber
30        result = prime * result + accountNumber;
31        //uso di holderName
32        result = prime * result
33            + ((holderName == null) ? 0 : holderName.hashCode());
34
35        return result;
36    }
37
38    //Confronta solamente account number
39    @Override
40    public boolean equals(Object obj) {
41        if (this == obj)
42            return true;
43        if (obj == null)
44            return false;
45        if (getClass() != obj.getClass())
46            return false;
47        Account other = (Account) obj;
48
49        //uso di account number
50        if (accountNumber != other.accountNumber)
51            return false;
52        //uso di holdername
53        if (holderName == null) {
54            if (other.holderName != null)
55                return false;
56        } else if (!holderName.equals(other.holderName))
57            return false;
58
59        return true;
60    }
61 }

```

Come scegliamo l'implementazione? Ricordiamo:

- **Set**: elementi non ripetuti, eventualmente ordinati
- **List**: elementi ripetuti, ordinati in modo posizionale
- **Map**: mappa di elementi, eventualmente ordinati per chiave

Nell'hashSet la differenza tra caso medio e caso peggiore dipende dal numero dei conflitti, quindi dipende dalla distribuzione dei valori e dalla qualità dell'hashCode().

16.11 ITERATORI

L'iteratore è un oggetto che permette di effettuare la scansione di una Collection.

Tutti gli iteratori implementano una interfaccia con i metodi:

- `Next()`: ritorna il prossimo elemento della collezione, spostandosi in avanti di una posizione
- `hasNext()`: ritorna true se c'è almeno un altro elemento nella collezione

gli iteratori possono essere ottenuti da una collezione nel seguente modo:

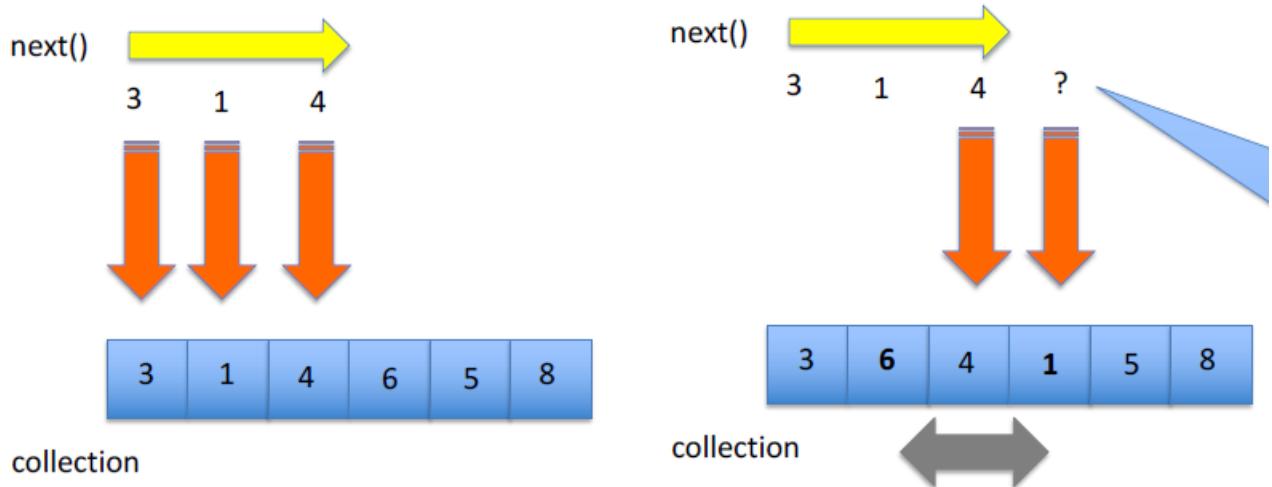
- `iterator()`: ritorna un iteratore, implementato in ogni Collection
- le mappe implementano `keySet()` e `values()` che ritornano un Set e una Collection dove è poi possibile invocare `iterator()`

```

1 import java.util.ArrayList;
2 import java.util.Iterator;
3 import java.util.LinkedList;
4 import java.util.List;
5
6 public class Zoo {
7     private List<Animale> animali;
8
9     public Zoo(int maxNumAnimali) {
10         animali = new ArrayList<Animale>();
11         // per usare LinkedList
12         // animali = new LinkedList<Animale>();
13     }
14
15     public void aggiungiAnimale(Animale animale) {
16         animali.add(animale);
17     }
18
19     public void parla() {
20         for (int i = 0; i < animali.size(); i++) {
21             System.out.println(animali.get(i).verso());
22         }
23     }
24
25     public void sbrana() {
26         Iterator<Animale> it = animali.iterator();
27
28         while(it.hasNext()) {
29             Animale animale = it.next();
30             if (animale instanceof Felino)
31                 System.out.println(((Felino) animale).sbrana());
32         }
33
34     /*      for(Animale animale:animali) {
35         if (animale instanceof Felino)
36             System.out.println(((Felino) animale).sbrana());
37     }
38 */
39 }
```

L'iteratore viene utilizzato per iterare sui valori di un arrayList.

Attenzione: la chiamata a next() può ritornare una ConcurrentModificationException nel caso si provi a modificare una collezione mentre viene scandita da un iteratore.



16.12 FOREACH

se il vostro codice non deve far altro che iterare su una colezione possiamo utilizzare il costrutto foreach per renderlo più compatto e nascondere la presenza di un iteratore.

La sintassi è:

- `for([tipo_variabile] [nome_variabile]: [collezione/array])`

per esempio vedere l'ultimo listato mostrato, nei commenti a sbrana().

17 INTERFACCE UTENTE GRAFICHE

APPENDICE: TEST E DEBUG CON JUNIT

Distinguiamo tra:

- **Malfunzionamento (failure):** il programma non funziona, restituisce un risultato scorretto
- **Difetto (fault):** il codice è sbagliato, è stato scritto male introducendo errori

```
public int raddoppia(int num) {
    int risultato;
    risultato = num * num;
    return risultato;
}
```

Malfunzionamento: raddoppia(5) restituisce 25.
Difetto: alla riga 3 viene utilizzato * al posto di +.

Tipi di difetti:

- Difetti sintattici o dipendenti dal linguaggio:
 - La forma del programma non segue le regole richieste dal linguaggio di programmazione (ad esempio manca una parentesi, parametro di tipo diverso rispetto a quello richiesto).
- Difetti semantici:
 - Violazioni della semantica (significato) del programma, ovvero il comportamento del programma non corrisponde alle attese (ad esempio un ciclo `for` con un numero errato di iterazioni).
 - Non vengono rilevati dal compilatore ma provocano malfunzionamenti a runtime

Difetti e malfunzionamenti sono sempre relativi a una specifica, cioè una descrizione del programma e/o del linguaggio.

Si effettua:

- Testing per rilevare malfunzionamenti causati da possibili difetti semantici
- Debugging per individuare i difetti semantici (fault) che causano i malfunzionamenti rilevati

Non si può eseguire il test per tutti i possibili input per verificare la correttezza assoluta di un programma.

Per un progetto di test bisogna selezionare un insieme di input significativi che diano una ragionevole confidenza sulla correttezza del programma.

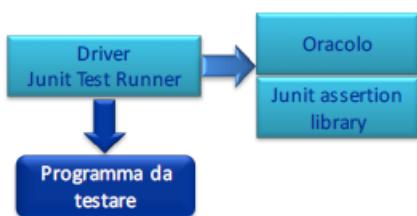
- Casi normali: almeno un test per ogni comportamento normale identificato dalla specifica
- Casi errati/anomali: almeno un test per le condizioni anomale o errate che devono essere gestite dal programma
- Casi di frontiera: almeno un test per i confini

Questi test hanno dei problemi se vengono integrati nel main:

- Poco flessibili: bisognerebbe continuare a modificare il main
- Poco efficaci

Per questo è stato creato un ambiente specializzato per l'esecuzione dei test.

Junit fornisce un ambiente di supporto all'esecuzione automatica di test per Java. JUnit è già integrato come plug-in in Eclipse.



- Driver: sostituisce il main, ma permette un'organizzazione/gestione più flessibile dei casi di test
- Oracolo: specifica dei valori attesi per ogni test

I casi di test sono metodi annotati con @Test.

Cose da ricordare:

- La libreria di JUnit deve far parte del progetto (build path). Viene automaticamente integrata da Eclipse
- I metodi di test vengono riconosciuti se è presente l'annotazione @Test
- Uso di metodi assert per definire il risultato atteso dei test
- Nel menù contestuale bisogna eseguire il test come "JUnit test" (Run as → JUnit Test)
- In casi di fallimenti, click su "failure trace" per vedere quale delle assert non è stata rispettata

TEST DI REGRESSIONE E AUTOMAZIONE DEL TEST

Quando modifico il programma devo verificare che il nuovo codice si comporti bene almeno quanto la versione precedente. Per questo, ad ogni cambiamento di un programma è necessario eseguire nuovamente il test per verificare che le modifiche non abbiano introdotto dei difetti in funzionalità precedentemente corrette.

Tecnicamente si testa che la funzionalità del programma non sia regredita (e da qua test di regressione) a fronte dei cambiamenti.

In caso di estensione del set di funzionalità è anche necessario progettare nuovi test per le nuove funzionalità.

DEBUGGING

Il test rileva i malfunzionamenti, il debugging è l'attività che permette di localizzare i difetti che causano i malfunzionamenti. Per eseguire il debugging è necessario:

- considerare un caso di test che provoca un malfunzionamento
- eseguire il programma fermando l'esecuzione in punti intermedi
- visionare i valori delle variabili in punti intermedi
- cercare di capire in quale punto il programma produce uno stato scorretto

Le funzionalità essenziali per il debugging sono:

- selezione di breakpoint: punti del programma in cui l'esecuzione deve essere temporaneamente sospesa
- esecuzione passo passo: permette di avanzare l'esecuzione di un comando per volta
- esame dello stato intermedio: permette di visualizzare il valore delle variabili

Come operare:

- Si parte da un malfunzionamento riscontrato in fase di testing
- Si identifica uno stato corretto (iniziale) e uno scorretto (finale)
- Si inseriscono breakpoint nei punti ritenuti significativi
- Si lancia l'esecuzione e ci si ferma ad ogni breakpoint esaminando lo stato
- Ci si ferma quando si trova uno stato scorretto

- Quando ho trovato il primo stato scorretto ho isolato la porzione di codice tra il breakpoint corrispondente allo stato scorretto e il breakpoint precedente
- Ripeto aggiungendo altri breakpoint tra i due breakpoint identificati
- Oppure eseguo passo passo (se le istruzioni tra i due breakpoint sono poche)

LIBRERIE DI ASSERZIONI IN JUNIT

Costruttori		
protected	Assert()	Rende <code>protected</code> il costruttore essendo solo una classe statica
Metodi		
static void	<code>assertArrayEquals</code> (byte[] expecteds, byte[] actuals)	“Asserisce” che due array di byte sono uguali
static void	<code>assertArrayEquals</code> (char[] expecteds, char[] actuals)	“Asserisce” che due array di char sono uguali
static void	<code>assertArrayEquals</code> (int[] expecteds, int[] actuals)	“Asserisce” che due array di int sono uguali
static void	<code>assertArrayEquals</code> (long[] expecteds, long[] actuals)	“Asserisce” che due array di long sono uguali
static void	<code>assertArrayEquals</code> (java.lang.Object[] expecteds, java.lang.Object[] actuals)	“Asserisce” che due array di oggetti sono uguali
static void	<code>assertArrayEquals</code> (short[] expecteds, short[] actuals)	“Asserisce” che due array di short sono uguali
static void	<code>assertArrayEquals</code> (java.lang.String message, byte[] expecteds, byte[] actuals)	“Asserisce” che due array di byte sono uguali. Nel caso non lo siano viene mostrato message.
static void	<code>assertArrayEquals</code> (java.lang.String message, char[] expecteds, char[] actuals)	“Asserisce” che due array di char sono uguali. Nel caso non lo siano viene mostrato message.
static void	<code>assertArrayEquals</code> (java.lang.String message, int[] expecteds, int[] actuals)	“Asserisce” che due array di int sono uguali. Nel caso non lo siano viene mostrato message.
static void	<code>assertArrayEquals</code> (java.lang.String message, long[] expecteds, long[] actuals)	“Asserisce” che due array di long sono uguali. Nel caso non lo siano viene mostrato message.
static void	<code>assertArrayEquals</code> (java.lang.String message, java.lang.Object[] expecteds, java.lang.Object[] actuals)	“Asserisce” che due array di oggetti sono uguali. Nel caso non lo siano viene mostrato message.
static void	<code>assertArrayEquals</code> (java.lang.String message, short[] expecteds, short[] actuals)	“Asserisce” che due array di short sono uguali. Nel caso non lo siano viene mostrato message.
static void	<code>assertEquals</code> (double expected, double actual)	Sconsigliato. Utilizzare invece Il metodo successivo.
static void	<code>assertEquals</code> (double expected, double actual, double delta)	“Asserisce” che due double o float sono uguali entro un certo delta.
static void	<code>assertEquals</code> (long expected, long actual)	“Asserisce” che due long sono uguali.
static void	<code>assertEquals</code> (java.lang.Object[] expecteds, java.lang.Object[] actuals)	Sconsigliato. Utilizzare invece

		Il metodo assertArrayEquals.
static void	assertEquals (java.lang.Object expected, java.lang.Object actual)	“Asserisce” che due oggetti sono uguali
static void	assertEquals (java.lang.String message, double expected, double actual)	Sconsigliato. Utilizzare invece Il metodo successivo. Nel caso non lo siano viene mostrato message.
static void	assertEquals (java.lang.String message, double expected, double actual, double delta)	“Asserisce” che due double o float sono uguali entro un certo delta. Nel caso non lo siano viene mostrato message.
static void	assertEquals (java.lang.String message, long expected, long actual)	“Asserisce” che due long sono uguali. Nel caso non lo siano viene mostrato message.
static void	assertEquals (java.lang.String message, java.lang.Object[] expecteds, java.lang.Object[] actuals)	Sconsigliato. Utilizzare invece Il metodo assertArrayEquals.
static void	assertEquals (java.lang.String message, java.lang.Object expected, java.lang.Object actual)	“Asserisce” che due oggetti sono uguali. Nel caso non lo siano viene mostrato message.
static void	assertFalse (boolean condition)	“Asserisce” che una condizione è falsa.
static void	assertFalse (java.lang.String message, boolean condition)	“Asserisce” che una condizione è falsa. Nel caso non lo sia viene mostrato message.
static void	assertNotNull (java.lang.Object object)	“Asserisce” che un oggetto non è null.
static void	assertNotNull (java.lang.String message, java.lang.Object object)	“Asserisce” che un oggetto non è null. Nel caso non lo sia viene mostrato message.
static void	assertNotSame (java.lang.Object unexpected, java.lang.Object actual)	“Asserisce” che due oggetti non sono riferiti allo stesso indirizzo.
static void	assertNotSame (java.lang.String message, java.lang.Object unexpected, java.lang.Object actual)	“Asserisce” che due oggetti non sono riferiti allo stesso indirizzo. Nel caso lo siano viene mostrato message.
static void	 assertNull (java.lang.Object object)	“Asserisce” che un oggetto è null.
static void	 assertNull (java.lang.String message, java.lang.Object object)	“Asserisce” che un oggetto è null. Nel caso lo sia viene mostrato message.
static void	assertSame (java.lang.Object expected, java.lang.Object actual)	“Asserisce” che due oggetti sono riferiti allo stesso indirizzo.
static void	assertSame (java.lang.String message, java.lang.Object expected, java.lang.Object actual)	“Asserisce” che due oggetti sono riferiti allo stesso indirizzo. Nel caso non lo

		siano viene mostrato message.
static <T> void	assertThat (java.lang.String reason, T actual, org.hamcrest.Matcher<T> matcher)	“Asserisce” che actual soddisfa la condizione espressa in matcher. Nel caso non lo sia viene mostrato reason. T è il tipo statico accettato da matcher.
static <T> void	assertThat (T actual, org.hamcrest.Matcher<T> matcher)	“Asserisce” che actual soddisfa la condizione espressa in matcher. T è il tipo statico accettato da matcher.
static void	assertTrue (boolean condition)	“Asserisce” che una condizione è vera.
static void	assertTrue (java.lang.String message, boolean condition)	“Asserisce” che una condizione è vera. Nel caso non lo sia viene mostrato message.
static void	fail()	Fallisce un test senza messaggio.
static void	fail (java.lang.String message)	Fallisce un test con messaggio.
Metodi ereditati da java.lang.Object		
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait		

ESEMPIO DI INTEGRAZIONE DI JUNIT

Classe CapoDaLavare

```
public class CapoDaLavare {  
    //immutabili  
    private String descrizione;  
    private int maxTemp;  
  
    public CapoDaLavare(String descrizione, int maxTemp) {  
        this.descrizione = descrizione;  
        this.maxTemp = maxTemp;  
    }  
  
    public CapoDaLavare(String descrizione) {  
        this(descrizione, 30);  
    }  
  
    public int getMaxTemp() {  
        return maxTemp;  
    }  
  
    public String getDescrizione() {  
        return descrizione;  
    }  
  
    public boolean equals(CapoDaLavare altro) {  
        if(this == altro)  
            return true;  
        if(altro == null)  
            return false;  
        if(descrizione.equalsIgnoreCase(altro.descrizione))  
            return true;  
        return false;  
    }  
  
    @Override  
    public String toString() {  
        return "Capo da lavare [descrizione=" + descrizione + ", maxTemp=" +  
        maxTemp  
        + "]";  
    }  
}
```

Classe Lavatrice

```

public class Lavatrice {
    //non accessibile in lettura
    private CapoDaLavare capiDaLavare[];
    //mutabile
    private int temperaturaLavaggio;

    public Lavatrice(int capienza) {
        this.capiDaLavare = new CapoDaLavare[capienza];
        setTemperaturaLavaggio(30);
    }

    public int getTemperaturaLavaggio() {
        return temperaturaLavaggio;
    }

    public void setTemperaturaLavaggio(int temperaturaLavaggio) {
        if(temperaturaLavaggio > 0) {
            boolean cambiabile=true;
            for (int i = 0; i < capiDaLavare.length; i++) {
                if(capiDaLavare[i] != null &&
                    capiDaLavare[i].getMaxTemp() <
                    temperaturaLavaggio && cambiabile) {
                    cambiabile=false;
                }
            }
            if(cambiabile)
                this.temperaturaLavaggio = temperaturaLavaggio;
        }
    }

    public boolean aggiungiCapo(CapoDaLavare capo) {
        if(capo != null && capo.getMaxTemp() >= temperaturaLavaggio) {
            for (int i = 0; i < capiDaLavare.length; i++) {
                if(capiDaLavare[i] == null) {
                    capiDaLavare[i] = capo;
                    return true;
                }
            }
        }
        return false;
    }

    public boolean aggiungiCapo(String descrizione, int temp) {
        return aggiungiCapo(new CapoDaLavare(descrizione, temp));
    }
}

```

```
public CapoDaLavare[] rimuoviCapo(CapoDaLavare capo) {
    int numeroCapi = 0;
    for (int i = 0; i < capiDaLavare.length; i++) {
        if(capiDaLavare[i] != null && capiDaLavare[i].equals(capo))
            numeroCapi++;
    }

    if (numeroCapi == 0) {
        return null;
    }

    CapoDaLavare[] ret = new CapoDaLavare[numeroCapi];

    int pos = 0;
    for (int i = 0; i < capiDaLavare.length; i++) {
        if(capiDaLavare[i] != null && capiDaLavare[i].equals(capo)) {
            ret[pos++] = capiDaLavare[i];
            capiDaLavare[i] = null;
        }
    }
    return ret;
}

public int massimaCapienza() {
    return capiDaLavare.length;
}
```

Classe JUnit TestEsame

```

import static org.junit.Assert.*;
import org.junit.Test;

public class TestEsame {

    @Test
    public void testCreazioneCapo() {
        CapoDaLavare c1 = new CapoDaLavare ("Camicia cotone");
        assertEquals ("Camicia cotone", c1.getDescrizione ());
        assertEquals (30, c1.getMaxTemp ());

        CapoDaLavare c2 = new CapoDaLavare ("Lenzuolo", 80);
        assertEquals ("Lenzuolo", c2.getDescrizione ());
        assertEquals (80, c2.getMaxTemp ());
    }

    @Test
    public void testEqualsCapi () {
        //verifico equals su titolo
        CapoDaLavare c1 = new CapoDaLavare ("Camicia cotone", 30);
        CapoDaLavare c2 = new CapoDaLavare ("Camicia cotone", 40);
        CapoDaLavare c3 = new CapoDaLavare ("Camicia COTONE", 40);
        CapoDaLavare c4 = new CapoDaLavare ("Lenzuolo", 80);

        assertTrue (c1.equals (c2));
        assertTrue (c1.equals (c3));
        assertFalse (c1.equals (c4));
        assertFalse (c1.equals (null));
    }

    @Test
    public void testCreazioneLavatrice () {
        Lavatrice l = new Lavatrice (10);
        assertEquals (30, l.getTemperaturaLavaggio ());
        assertEquals (10, l.massimaCapienza ());
    }

    @Test
    public void testInserisciCapo () {
        Lavatrice l = new Lavatrice (2);
        l.setTemperaturaLavaggio (40);

        CapoDaLavare c1 = new CapoDaLavare ("Camicia cotone", 30);
        CapoDaLavare c2 = new CapoDaLavare ("Lenzuolo", 80);
        CapoDaLavare c3 = new CapoDaLavare ("Giacca", 40);

        assertFalse (l.aggiungiCapo (c1));
        assertTrue (l.aggiungiCapo (c2));
        assertTrue (l.aggiungiCapo (c3));
    }
}

```

```

@Test
public void testMassimaCapienza() {
    Lavatrice l = new Lavatrice(2);
    l.setTemperaturaLavaggio(30);

    CapoDaLavare c1 = new CapoDaLavare("Camicia cotone", 30);
    CapoDaLavare c2 = new CapoDaLavare("Lenzuolo", 80);
    CapoDaLavare c3 = new CapoDaLavare("Giacca", 40);

    assertTrue(l.aggiungiCapo(c1));
    assertTrue(l.aggiungiCapo(c2));
    assertFalse(l.aggiungiCapo(c3));
}

@Test
public void testRimuoviCapo() {
    Lavatrice l = new Lavatrice(4);
    l.aggiungiCapo(new CapoDaLavare("Camicia", 30));
    l.aggiungiCapo(new CapoDaLavare("Camicia", 40));
    l.aggiungiCapo(new CapoDaLavare("Camicia", 50));
    l.aggiungiCapo(new CapoDaLavare("Lenzuolo", 80));

    CapoDaLavare[] rimossi = l.rimuoviCapo(new CapoDaLavare("Camicia"));
    assertNotNull(rimossi);
    assertEquals(3, rimossi.length);

    rimossi = l.rimuoviCapo(new CapoDaLavare("Lenzuolo"));
    assertNotNull(rimossi);
    assertEquals(1, rimossi.length);
    assertEquals("Lenzuolo", rimossi[0].getDescrizione());
}

@Test
public void testRimuoviCapoNonPresente() {
    Lavatrice l = new Lavatrice(4);
    l.aggiungiCapo(new CapoDaLavare("Camicia", 30));

    CapoDaLavare[] rimossi = l.rimuoviCapo(new
        CapoDaLavare("Pantalone"));
    assertNull(rimossi);

    rimossi = l.rimuoviCapo(null);
    assertNull(rimossi);
}

@Test
public void testModificaTemperaturaLavaggio() {
    Lavatrice l = new Lavatrice(4);
    l.aggiungiCapo(new CapoDaLavare("Camicia", 50));
    l.setTemperaturaLavaggio(40);
    assertEquals(40, l.getTemperaturaLavaggio());

    l.aggiungiCapo(new CapoDaLavare("Lenzuolo", 80));
    l.setTemperaturaLavaggio(90);
    assertEquals(40, l.getTemperaturaLavaggio());

    l.setTemperaturaLavaggio(-90);
    assertEquals(40, l.getTemperaturaLavaggio());
}
}

```