

03

Programmazione strutturata in Java

Mirko Viroli

`mirko.viroli@unibo.it`

C.D.L. Ingegneria e Scienze Informatiche
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2018/2019



Goal della lezione

- Mostrare la parte imperativa/strutturata del linguaggio Java
- Illustrare le differenze rispetto al linguaggio C
- Approfondire alcuni aspetti nuovi (array, foreach)
- Discutere alcuni aspetti preliminari di qualità del software

Argomenti

- Tipi e operatori di Java
- Il caso degli array
- Istruzioni di Java
- Costrutti di programmazione strutturata
- Costruzione e uso di utility class
- Elementi iniziali di buona programmazione

- 1 Tipi e operatori di Java
- 2 Java arrays
- 3 Statements
- 4 Sulla qualità del software

Tipi

Cos'è un tipo

- È un meccanismo per classificare valori (e oggetti)
- È costituito da un nome, un set di valori, e un set di operatori/meccanismi per manipolarli

Tipi di Java

- Primitivi: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`
- Array: `boolean[]`, `byte[]`, `String[]`, `String[][]`,...
- Classi: `Object`, `String`, `Integer`, `ArrayList`, `JFrame`,...
- Altri che vedremo in seguito: interfacce, classi innestate,...

Java e i tipi..

- Java ha “typing statico”: ogni espr. ha un tipo noto al compilatore
 - Java ha “typing forte”: non si accettano espressioni con errori di tipo
- ⇒ .. permette l'intercettazione a priori di molti errori

Nome: `boolean`

- Valori: `true`, `false`
- Operatori unari: `!` (not)
- Operatori binari: `&` (and), `|` (or), `^` (xor), `&&` (and-c), `||` (or-c)
 - ▶ `&&` e `||` valutano il secondo argomento solo se necessario
 - ▶ `false && X` dà comunque `false`
 - ▶ `true || X` dà comunque `true`
- Operatori di confronto numerici: `>`, `<`, `>=`, `<=`
- Operatori di uguaglianza (su tutti i tipi): `==`, `!=`
 - ▶ `10 == 20`
 - ▶ `new Object() == new Object()` (confronta i riferimenti)
- Operatore ternario (booleano, tipo, tipo): `?:`
 - ▶ `b?v1:v2` restituisce `v1` se `b` è vero, `v2` altrimenti



Tipi numerici

Primitive type	Size	Minimum	Maximum	Wrapper type
boolean	—	—	—	Boolean
char	16 bits	Unicode 0	Unicode $2^{16}-1$	Character
byte	8 bits	-128	+127	Byte
short	16 bits	-2^{15}	$+2^{15}-1$	Short
int	32 bits	-2^{31}	$+2^{31}-1$	Integer
long	64 bits	-2^{63}	$+2^{63}-1$	Long
float	32 bits	IEEE754	IEEE754	Float
double	64 bits	IEEE754	IEEE754	Double
void	—	—	—	Void



Interi: `byte`, `short`, `int`, `long`

Operatori

- Base: `+`, `-`, `*`, `/` (con resto), `%` (resto), `+` e `-` anche unari
- Bit-a-bit: `&` (and), `|` (or), `^` (xor), `~` (not)
- Shift: `>>` (dx senza segno), `<<` (sx), `>>>` (dx con segno)
- Operatori unari/binari applicati ad un tipo, restituiscono il tipo stesso

Codifica, rappresentazione

- Interi codificati in complemento a 2 (ciò impatta il suo range)
- Rappresentazione decimale (200), ottale (0310), esadecimale (0xC8)
- Di default sono `int`, per avere un `long` va aggiunta una L (15L)

Prassi

- Raro l'uso di `byte` e `short`, non molto più efficienti di `int`
- `int` più efficiente di `long`

Numeri in virgola mobile: `float`, `double`

Operatori

- Base: `+`, `-`, `*`, `/` (con resto), `%` (resto), `+` e `-` anche unari

Codifica, rappresentazione

- Codificati secondo lo standard IEEE 754
- Rappresentazione standard (`-128.345`), o scientifica (`-1.2835E+2`)
- Di default sono `double`, per avere un `float` va aggiunta una `F`

Prassi

- Raro l'uso di `float`, anche se più efficiente di `double`
- Attenzione agli errori di precisione!!



Provate voi stessi..

```
1 class Try {
2     public static void main(String[] s) {
3         System.out.println(10 + 20); // 30
4         System.out.println(010 + 020); // 24
5         System.out.println(0xFFFFFFFF); // -1
6         System.out.println(0x7FFFFFFFF); // 2147483647
7         System.out.println(0x80000000); // -2147483648
8         System.out.println(0x80000000 - 1); // 2147483647
9         System.out.println(2147483647 + 1); // -2147483648
10        System.out.println(2147483647L + 1); // 2147483648
11        System.out.println((0x0F0F | 0xF0F0) == 0xFFFF); // true
12        System.out.println((0x0F0F & 0xF0F0) == 0); // true
13        System.out.println((0x0F0F << 4) == 0xF0F0); // true
14        System.out.println((0x0F0F >> 4) == 0x00F0); // true
15        System.out.println(~0x0F0F == 0xFFFFF0F0); // true
16        System.out.println(10 / 3); // 3
17        System.out.println(10 % 3); // 1
18        System.out.println(3.5 / 51 * 51); // 3.5000000000000004
19        System.out.println(3.5f / 51 * 51); // 3.5000002
20    }
21 }
```

Conversioni

Conversioni di tipo, dette anche **cast**: (tipo)valore

- Fra tipi numerici sono sempre consentite
- Possono causare perdita di informazione
- Es.: (int)3.33, ((double)10)/3, (short)100

Conversioni automatiche, dette anche **coercizioni**

- Le inserisce automaticamente il compilatore in certi casi
- Quando ci si aspetta un tipo, e si usa un valore diverso
- Solo da un tipo più specifico a uno più generale
 - ▶ Spec→Gen: byte, short, int, long, float, double
- Due casi:
 - ▶ In assegnamenti: long l=100; diventa long l=(long)100;
 - ▶ Operazioni fra tipi diversi: 10.1+20 diventa 10.1+(double)10
 - ▶ Passando valori a funzioni

I caratteri

Rappresentazione

- Singolo carattere: `'a', 'z', 'A', '='`
- Codice ASCII: 65 (`'A'`), 66 (`'B'`)
- Caratteri escape: `'\n', '\\', '\0'`
- Caratteri UTF16: `'\u6C34'`

Codifica

- 16 bit UTF16
- automaticamente convertibile ad un numerico fra 0 e 65535

```
1 class TryChars {  
2     public static void main(String[] s) {  
3         System.out.println("lettera 'a' e a capo" + 'a');  
4         System.out.println("acqua (cinese) e a capo " + '\u6C34');  
5         System.out.println("backslash e a capo" + '\\');  
6         System.out.print("a capo " + '\n');  
7         System.out.print("a capo " + (char) 10);  
8     }  
9 }
```



Outline

- 1 Tipi e operatori di Java
- 2 Java arrays**
- 3 Statements
- 4 Sulla qualità del software

Caratteristiche generali

- Internamente sono degli oggetti
- Quindi sono gestiti con riferimenti sullo heap
- Notazione ad-hoc (e C-like) per creare, leggere e scrivere elementi

Principale differenza rispetto al C

- Un array ha una lunghezza esplicita e accessibile (non modificabile)
- È impossibile violare i limiti di una array, pena un errore (`ArrayIndexOutOfBoundsException`)
- L'accesso all'array è di conseguenza leggermente rallentato



Sintassi Array

Creazione array

- Due notazioni, per elenco e per dimensione
 - ▶ `int[] ar1 = new int[]{10,20,30,40,50,7,8,9};`
 - ▶ `int[] ar2 = new int[200]; // new int[]{0,0,...,0}`
- quando creati per dimensione, gli elementi sono inizializzati come se fossero campi di una classe
- la creazione di array di array è analoga:
 - ▶ `int[][] m=new int[][]{new int[]{...},...};`
 - ▶ `int[][] m2=new int[200][200];`

Accesso array

- `ar1.length` // lunghezza
- `ar2[23]` // espressione per leggere 24-esimo elemento
- `ar2[23]=10;` // assegnamento del 24-esimo elemento
- `m[1][2]=10;` // assegnamento riga 2 colonna 3

Qualche esempio d'uso di array

```
1 class UseArrays {
2     public static void main(String[] s) {
3         int[] a = new int[] { 10, 20, 30 };
4         System.out.println("a: " + a[0] + " " + a[1] + " " + a[2]);
5         double[] b = new double[] { 10.1, 10.2 };
6         b[1] = 10.21;
7         String[] c = new String[] { "10", "20", "3" + "0" };
8         System.out.println("c.length: " + c.length); // 3
9         boolean[] d = new boolean[10000];
10        System.out.println("d[5000]: " + d[5000]); // false
11        int[][] e = new int[5][5]; // matrice 5x5 di zeri
12        int[][] f = new int[][] { new int[] { 11, 12, 13, 14 },
13            new int[] { 21, 22, 23, 24 }, new int[] { 31, 32, 33, 34 }
14        };
15        System.out.println("f.length: " + f.length); // 3
16        System.out.println("f[0].length: " + f[0].length); // 4
17        System.out.println("f[1][2]: " + f[1][2]); // 23
18        System.out.println("Error on f[1][4]: " + f[1][4]); //Exception
19        // Versioni semplificate:
20        // int[] aa = {10,20,30};
21        // int[][] aaa = {{10},{20},{30}};
22    }
```

Array di oggetti

Creazione array – stessa notazione

- Per elenco:
 - ▶ `Object[] ar = new Object[]{new Object(),new Object()};`
- Per dimensione
 - ▶ `Object[] ar2 = new Object[200];`
 - ▶ in ogni posizione c'è `null`
 - ▶ frequente errore dello studente è pensare che sia un array di nuovi oggetti automaticamente creati
 - ▶ ricorda: è una sola `new` quindi si crea un solo oggetto, l'array stesso
- l'accesso segue una simile sintassi



Outline

- 1 Tipi e operatori di Java
- 2 Java arrays
- 3 Statements**
- 4 Sulla qualità del software

I linguaggi OO sono anche imperativi/strutturati

Java “estende” il C

- Come C++ e C#, Java è alla base anche imperativo/strutturato – altri linguaggi come Scala invece no
- Il codice di un metodo è un insieme di comandi C-like
- Ecco perché li si chiama object-oriented e non object-based

Panoramica istruzioni

- Variabili e assegnamenti: `int x`; `int x=5`; `x=5`;
- Ritorno: `return 5`;
- Chiamate: `meth(3,4)`; `obj.meth(3)`; `cls.meth(4)`;
- Costrutti: `for`, `while`, `do`, `switch`, `if`, `break`, `continue`
- Qualche altra tipologia, che vedremo nel prosieguo



Principali differenze

- La condizione dell'`if`, `for`, `while` e `do` è un `boolean`
- Nel `for` è possibile dichiarare la variabile di ciclo (come nel C99), sarà visibile solo internamente
 - ▶ `for(int i=0;i<10;i++){..}`

Differenza filosofica

- Java è molto più restrittivo del C
- Molto di ciò che è solo warning in C, è errore in Java
 - ▶ unreachable statement: istruzioni non raggiungibili (per errore)
 - ▶ variable may not have been initialised (uso variabile prima del suo init)
 - ▶ missing return statement (un return finale è obbligatorio)
- Può sembrare una filosofia che rende la programmazione “rigida”, e invece è cruciale per supportare lo sviluppo di software di qualità
- Le prassi che discuteremo ci porteranno ulteriori rigidità

Java come linguaggio puramente strutturato

Un uso limitato (ma a volte utile) di Java

- Una classe ha solo metodi o campi dichiarati **static**
- In questo caso tale classe definisce un **modulo**, dove i metodi assomigliano a pure funzioni e i campi a pure variabili globali a quella classe, ossia una struttura analoga a quella di una libreria C
- Un metodo (o campo) statico viene richiamato nel seguente modo:
 - ▶ da fuori la classe (se dichiarato **public**):
`<nome-classe>.<nome-metodo>(...)`
 - ▶ da dentro la classe: `<nome-metodo>(...)`
- E' una tecnica usata per realizzare "utility class", come ad esempio la classe delle funzioni matematiche `java.lang.Math`



Qualche prova di java.lang.Math

```
1 import java.util.Arrays;
2
3 public class UseMath { // E' una utility class
4
5     // Metodo statico, crea e torna un array di double
6     // lungo size, contenente numeri random 0<=x<=1
7     static double[] randomValues(int size){
8         double[] d = new double[size];
9         for (int i = 0; i<size; i++){
10             d[i] = Math.random();
11         }
12         return d;
13     }
14
15     public static void main(String[] args) {
16         System.out.println(Math.random());
17         System.out.println(Math.random());
18         System.out.println(Math.random());
19
20         // Ricorda: toString è un metodo statico della classe Arrays
21         System.out.println(Arrays.toString(randomValues(10)));
22
23         for (double x = 0.0; x <= Math.PI*2; x = x + Math.PI/10){
24             System.out.println("x = " + x + "\t sin(x) = "+Math.sin(x));
25         }
26
27     }
28 }
```

Un esercizio sugli array

Costruire una funzione che dato un array ne produce in uscita uno della stessa lunghezza, ma invertendo il primo elemento con l'ultimo, il secondo col penultimo, etc..

```
1 import java.util.Arrays; // Classe con funzioni di utilità
2
3 class Reverse {
4     static int[] reverse(int[] array) { // funzione reverse
5         int[] out = new int[array.length]; // creo array per lunghezza
6         for (int i = 0; i < out.length; i++) { // for con var. interna
7             out[i] = array[out.length - 1 - i];
8         }
9         return out;
10    }
11    // l'uso di public qui sotto è necessario per il main
12    public static void main(String[] s) { // prova funz. di reverse
13        int[] a = new int[] { 10, 20, 30, 40 };
14        int[] b = reverse(a);
15        System.out.println(Arrays.toString(a)); // [10, 20, 30, 40]
16        System.out.println(Arrays.toString(b)); // [40, 30, 20, 10]
17    }
18 }
```

For-each

Java introduce una variante del ciclo `for`

- supporta l'astrazione di “per ogni elemento della collezione fai..”
- utile con gli array quando non importa il valore corrente dell'indice
- utilizzabile anche con le Collection di Java (liste, insiemi,..)

Sintassi – caso di array di interi

- `for(int var: array){ /* uso var */ }`
- `array` è una espressione che restituisce un `int[]`
- nel corpo del `for`, `var` vale via via ogni elemento dell'array
- leggi “per ogni `var` in `array` esegui il corpo”



For-each: esempio

```
1 class Sum {
2     public static int sum(int[] array) { // soluzione standard
3         int sum = 0;
4         for (int i = 0; i < array.length; i++) {
5             sum = sum + array[i];
6         }
7         return sum;
8     }
9
10    public static int sum2(int[] array) { // soluzione con for-each
11        int sum = 0;
12        for (int elem : array) { // per ogni elem in array..
13            sum = sum + elem;
14        }
15        return sum;
16    }
17
18    public static void main(String[] s) {
19        int res = sum2(new int[] { 10, 20, 30, 40 });
20        System.out.println("Somma : " + res);
21    }
22 }
```


Fornire input da linea di comando: argomenti del main

```
1 class SumMain {
2     public static int sum(int[] array) { // soluzione con for-each
3         int sum = 0;
4         for (int elem : array) { // per ogni elem in array..
5             sum = sum + elem;
6         }
7         return sum;
8     }
9     // da invocare con: java SumMain 10 20 30 40
10    public static void main(String[] args) {
11        int[] input = new int[args.length];
12        for (int i = 0; i < input.length; i++){
13            input[i] = Integer.parseInt(args[i]);
14        }
15        int res = sum(input);
16        System.out.println("Somma : " + res);
17    }
18 }
```



Teoria o applicazioni?

- La parte applicativa è maggiormente sviluppata in laboratorio e poi sperimentata nella realtà nella prova d'esame di progetto
- In aula si illustrano i concetti, i meccanismi, e gli elementi metodologici
- Spesso comunque si mostreranno applicazioni di esempio, semplici ma “paradigmatiche”, dove discutere alcuni aspetti tecnici e metodologici



Applicazione: GuessMyNumberApp

Problema

Realizzare una applicazione che, scelto un numero a caso compreso fra 1 a 100, chieda all'utente di indovinarlo, dandogli 10 tentativi e indicando ogni volta se il numero in input è maggiore o minore di quello scelto all'inizio

Alcune scelte progettuali

- L'applicazione è realizzabile in prima battuta come codice strutturato dentro al `main`
- Le (max) 10 iterazioni sono realizzabili da un ciclo (p.e., `for`)

Elementi implementativi

- `java.io.Console.readLine` usabile per leggere input da tastiera
- `java.lang.Integer.parseInt` usabile per convertire una stringa in un numero
- `java.util.Random.nextInt` usabile per ottenere un numero random

Implementazione GuessMyNumberApp

```
1 import java.io.Console;
2 import java.util.Random;
3
4 public class GuessMyNumberApp {
5
6     public static void main(String[] args) {
7         int number = new Random().nextInt(99) + 1;
8         for (int i = 1; i <= 10; i++){
9             System.out.println("Attempt no. "+i);
10            System.out.println("Insert your guess.. ");
11            int guess = Integer.parseInt(System.console().readLine());
12            if (guess == number){
13                System.out.println("You won!!");
14                return;
15            } else if (guess > number){
16                System.out.println("Your guess is greater, try again..");
17            } else {
18                System.out.println("Your guess is lower, try again..");
19            }
20        }
21    }
22 }
23
24 }
```

Outline

- 1 Tipi e operatori di Java
- 2 Java arrays
- 3 Statements
- 4 Sulla qualità del software**

Come è fatto del “buon software”?

Qualità esterna – aspetti funzionali

- Realizza correttamente il suo compito
- In termini di quali funzionalità fornisce

Qualità esterna – aspetti non-funzionali

- Performance adeguate (alle specifiche)
- Uso adeguato delle risorse del sistema (memoria, CPU)
- Caratteristiche di sicurezza, usabilità, etc..

Qualità interna – software ben costruito

- Facilmente manutenibile (leggibile, flessibile, riusabile)
- E quindi: meno “costoso”, a breve-/medio-/lungo-termine

⇒ ci concentriamo per ora su questa tipologia

Caratteristiche di qualità interna

Elementi necessari per il funzionamento

- Sintassi: soddisfa la grammatica del linguaggio
- Semantica: passa tutti i check del compilatore

Elementi aggiuntivi di qualità

- Convenzioni: soddisfa le convenzioni d'uso del linguaggio
- Commenti: usa i commenti mirati necessari a comprenderlo
- Efficace: usa tecniche che portano a evitare problemi futuri



Sintassi

- Gli errori di sintassi sono i primi ad essere eliminati
- Molti IDE li segnalano durante la digitazione
- La Java Language Specification fornisce la grammatica del linguaggio (V7, p.591)

Esempi comuni d'errore (in questo corso, in genere all'inizio)

- Parentesi non bilanciate (specialmente le graffe)
- Refusi nelle keyword: `Class`, `clas`, `boolean`
- Punteggiatura errata o mancante: `for(int i=0,i<10,i++){..}`



Correttezza semantica

Correttezza semantica – check del compilatore

- Oltre ad errori sintattici (forma), il compilatore segnala anche gli errori semantici (significato)
- Ed è molto più rigido del C
- Possono essere di varia natura, e a volte più difficili da comprendere

Esempi comuni d'errore

- Uso inappropriato dei tipi: `int a=true`; `int a=5+false`; `if (5)..`
- Refusi nel nome di campi, metodi, classi: `string`, `system`, `Sistem`
- Accesso a variabili, campi, metodi, classi che non esistono o non sono visibili
- Errori di flusso: `missing return statement`



Convenzioni

Convenzioni sul codice

- Riguardano indentazione, commenti, dichiarazioni, convenzioni sui nomi
- Migliorano leggibilità, e quindi comprensione
- È cruciale che vengano seguite
- <http://www.oracle.com/technetwork/java/codeconv-138413.html>

Li vedremo via via.. intanto focalizziamoci sulla formattazione

- Una istruzione per linea
- Formattazione parentesi graffe (E' CRUCIALE!)
- Inserire commenti nel codice—ne vedremo i dettagli



Programmazione efficace

- Vi sono un insieme di tecniche di programmazione che l'esperienza ha mostrato migliorare l'efficacia della programmazione
- Molte sono connesse a buone pratiche di programmazione OO e all'uso di pattern di progettazione noti
- Si vedano i libri “Effective Java” e “Design Patterns”
- Ne vedremo un po' mano a mano, e in dettaglio a fine del corso



Una nota sulle performance

Performance e JVM

- I linguaggi OO sono stati spesso criticati perchè più lenti rispetto ai linguaggi imperativi/strutturati
- Java e C# in più hanno una VM d'esecuzione che introduce ulteriori rallentamenti
- Di recente, tecniche avanzate nelle VM hanno ridotto, se non in alcuni casi annullato, le differenze in performance

In questo corso

- Non ci occuperemo in dettaglio di questo aspetto
- Prediligeremo sempre la soluzione più semplice/compatta
- Ci si aspetta non si usino soluzioni sia più complesse che più lente
- Ci si aspetta che si sappiano allocare correttamente le risorse (memoria, tempo) qualora richiesto