Laboratorio di Programmazione

Lezione 13 – Strings, Array, ArrayList e altro in Java

Marco Anisetti (teoria)
Dipartimento di Informatica
marco.anisetti@unimi.it

Matteo Luperto (lab. turno A)
Dipartimento di Informatica
matteo.luperto@unimi.it

Nicola Bena (lab. turno B)
Dipartimento di Informatica
nicola.bena@unimi.it

Incremento prefisso e suffisso

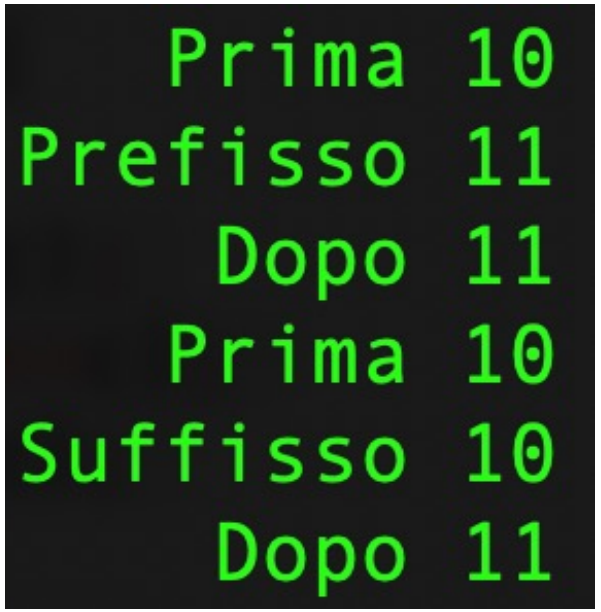
Java supporta sia incremento prefisso e suffisso

- Prefisso: prima incremento, poi valuto espressione.
- Suffisso: prima valuto espressione, poi incremento.

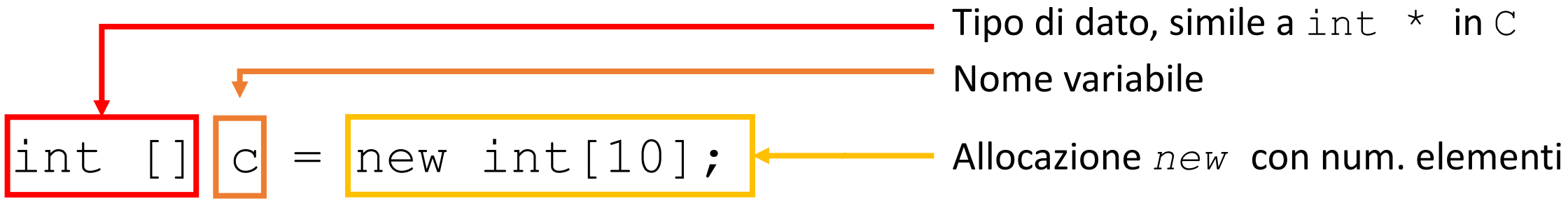
```
int i = 10;
++i; // incremento prefisso
i++; // incremento suffisso
--i; //  decremento prefisso
i--; //  decremento suffisso
```

Incremento prefisso e suffisso

```
int c = 10;
System.out.printf(" Prima %d%n",c);
System.out.printf("Prefisso %d%n",++c);
System.out.printf(" Dopo %d%n",c);
c=10;
System.out.printf(" Prima %d%n",c);
System.out.printf("Suffisso %d%n",c++);
System.out.printf(" Dopo %d%n",c);
```



Array in Java



Creazione e allocazione dell'array possono essere disaccoppiati.

La grandezza dell'array è **fissa**, ma può essere riallocato

```
int [] c;
…
c = new int[10];
…
c = new int[20];
```

Il vecchio array viene deallocato dal garbage collector e sostituito con uno nuovo

Array in Java

- Allocazione e dimensione avviene a runtime con `new`
- Deallocazione viene gestita dal garbage collector
- Attributo `length` → conosco a runtime quanti elementi contiene

```
int [] c = new int[10];
c.length // vale 10
for(int i=0; i < c.length; i++)
   // do something
```

- Controllo accesso agli indici (`index-out-of-bounds`)

Array in Java

- Allocazione e dimensione avviene a runtime con `new`
- Deallocazione viene gestita dal garbage collector
- Attributo `length` → conosco a runtime quanti elementi contiene

```
int [] c = new int[10];
c.length // vale 10
for(int i=0; i < c.length; i++)
   // do something
```

- Controllo accesso agli indici (`index-out-of-bounds`)

In `C` potevo sapere solo la grandezza degli array statici o la grandezza degli elementi con `sizeof`

In `C` posso accedere anche fuori dall'array, è un undefined behaviour

Array in Java

- Inizializzazione con valori noti

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

- Lunghezza nota a priori, usare un valore `final` (simile alla `define`)

```
final int N = 10;
int [] c = new int[N];
```

Il for e gli array

```
int [] arr = {1,2,3,4,5};
for (int i : arr)
     System.out.println(i);
```

Java vi permette di iterare tra gli elementi di un array direttamente all'interno del `for`, elemento per elemento, in alternativa al classico accesso posizionale

```
for (int i=0; i < arr.length; i++)
     System.out.println(arr[i]);
```

Funzioni e passaggio per valore o indirizzo

- Java funziona come `C`
- I tipi primitivi vengono passati per valore / per copia
  - Una modifica di un `int` all'interno della funzione non si ripercuote nel valore della variabile del chiamante
- Gli array vengono passati per indirizzo
  - Come in `C`, il passaggio per indirizzo è "simulato", viene passato per copia l'indirizzo in memoria dell'array
  - Eventuali modifiche all'array effettuate dalla funzione si ripercuotono sui valori dell'array nel chiamante

Array multidimensionali / matrici

- Array di puntatori, di cui ogni valore indirizza una *riga*
- Comportamento simile a `C` con array multidimensionali dinamici (allocati con `malloc`)
- Posso avere righe di lunghezza diversa, ogni riga è un array (con attributo `length`, ecc)

```
int [][] b = {{1,2},{3},{4,5}};
int [][] b = new int[N][M];
// oppure
int [][] b = new int[2][];
b[0] = new int[6];
b[1] = new int[7];
```

Nel terzo caso, devo allocare riga per riga con una nuova `new`

Attenzione!
Le singole righe della matrice possono essere non consecutive tra loro in memoria.

Le collezioni: ArrayList

- Un `Array` ha dimensione fissa:
  viene usato quando so (a *runtime*) quanti oggetti devo processare
  - + accesso diretto
  - + veloce e leggero
  - + accesso posizionale
  - + tipi primitive e oggetti
  - - limitato
- Un `ArrayList` ha dimensione dinamica:
  aggiungo e tolgo elementi mano a mano che arrivano/vengono processati
  - + astrazione
  - + funzioni di libreria per manipolare la collezione, (aggiungere/togliere dati)
  - - solo oggetti, per i tipi primitivi si usano le **wrapper class *(vedi teoria)***
  - - più verboso / complesso / lento

Le collezioni: ArrayList

```
import java.util.ArrayList;
…
ArrayList <Tipo> items = new ArrayList <Tipo>();
```

Collezione di dati omogenei di tipo *Tipo*

```
ArrayList <String> strs = new ArrayList <String>();
```

Le collezioni: ArrayList

```
import java.util.ArrayList;
. . .
// dichiaro la collezione
ArrayList <Integer> items = new ArrayList <Integer>();
// aggiungo il numero 2 in prima posizione
items.add(2);
System.out.println(items);
// aggiungo il numero 1 in prima posizione (posizione 0)
items.add(0,1);
System.out.println(items);
// aggiungo il valore 2 in seconda posizione (posizione 1)
items.add(1,Integer.valueOf(2));
System.out.println(items);
```

ArrayList: metodi

La classe ArrayList fornisce diversi metodi implementati

- `add` aggiunge un elemento alla fine di un array o alla posizione data
- `clear` elimina tutti gli elementi dell'array
- `contains` restituisce `true` se l'elemento è contenuto nell'`arrayList`

ArrayList: metodi

La classe ArrayList fornisce diversi metodi implementati

- `get` restituisce l'elemento all'indice specificato
- `indexOf` restituisce il primo elemento dell' `arrayList` che ha indice come specificato
- `remove` rimuove la prima occorrenza del valore specificato
- `size` restituisce numero di elementi memorizzati

… e molti altri, inoltre supporta il `for` potenziato

```
for (Integer i : items)
     System.out.println(i);
```

ArrayList: implementazione

Astrazione! Di norma non vi serve sapere come è implementata una struttura dati …

… ma male non fa saperlo per capire meglio come (non) usarla,

Struttura dati interna: array sequenziale allocato dinamicamente, con

- `size` = quanto è grande ora il mio `ArrayList`
- `capacity` = quanti elementi può contenere al massimo

Ogni elemento di un array è un puntatore all'area di memoria dove il dato è effettivamente salvato.

ArrayList.add: implementazione

```
ArrayList <String> items = new ArrayList <String>();
//...
items.add("Ciao!");
```

1. Controllo se ho spazio (`size < capacity`)
   Se non c'è spazio:
   1. Alloco nuova memoria (`capacity = 1.5*capacity);`
   2. Copio elemento nel nuovo `ArrayList`
2. Aggiungo il valore, incremento `size`
   1. Se non specificato, in coda
   2. Se specificato, alla posizione indicata *shiftando* a destra tutti gli elementi successivi (aggiorno i puntatori)

ArrayList.add: implementazione

```
ArrayList <String> items = new ArrayList <String>();
//...
items.add("Ciao!");
```

1. Controllo se ho spazio (`size < capacity`)
   Se non c'è spazio:
   1. Alloco nuova memoria (`capacity = 1.5*capacity);`
   2. Copio elemento nel nuovo `ArrayList`
2. Aggiungo il valore, incremento `size`
   1. Se non specificato, in coda
   2. Se specificato, alla posizione indicata *shiftando* a destra tutti gli elementi successivi (aggiorno i puntatori)

Aggiungere un valore quindi può essere una operazione costosa! Devo fare diverse operazioni che modificano l'intera struttura dell'array

ArrayList.get: implementazione

```
ArrayList <String> items = new ArrayList <String>();
//...
items.get(3);
```

Accesso diretto con indice dell’i-esimo elemento

ArrayList.get: implementazione

```
ArrayList <String> items = new ArrayList <String>();
//...
items.get(3);
```

Accesso diretto con indice dell'i-esimo elemento

Accedere al valore invece è immediato!

ArrayList.remove: implementazione

```
ArrayList <String> items = new ArrayList <String>();
//...
items.remove(3);
```

1. Accesso diretto all'elemento
2. Sposto indietro (copio) tutti gli elementi successivi a quello eliminato(aggiorno i puntatori)
3. Decremento `size` (e volendo ottimizzo l'array riducendo `capacity`)

ArrayList.remove: implementazione

```
ArrayList <String> items = new ArrayList <String>();
//...
items.remove(3);
```

Rimuovere un valore quindi può essere una operazione costosa! Devo fare diverse operazioni che modificano l'intera struttura dell'array

1. Accesso diretto all'elemento
2. Sposto indietro (copio) tutti gli elementi successivi a quello eliminato(aggiorno i puntatori)
3. Decremento `size` (e volendo ottimizzo l'array riducendo `capacity`)

Stringhe in Java

- Immutabili: ogni operazione di modifica prevede la sostituzione della vecchia stringa con una nuova

```
String s = "abcd";
s = "aacd"; // s[1] = 'a'; errore
```

- Sono *String Literal* memorizzate in una area dedicata dello *heap*, la String Pool
- Java controlla gli indici, `Index Out of Bounds Exception`

Stringhe in Java

- Immutabili: ogni operazione di modifica prevede la sostituzione della vecchia stringa con una nuova

```
String s = "abcd";
s = "aacd"; // s[1] = 'a'; errore
```

- Sono ***String Literal*** memorizzate in una area dedicata dello *heap*, la String Pool
- Java controlla gli indici, `Index Out of Bounds Exception`

Anche C supporta gli String Literal, ma non sono la norma.

```
char *strLit = "String Literal";
```

Modificare uno String Literal in C è un *undefined behaviour.*

C non ve lo segnala come un errore.

Stringhe in Java

Astrazione: possiedono metodi ed attributi:

- lunghezza

```
String s1 = "abcd";
s1.length(); //lunghezza
```

- uguaglianza

```
String s2 = "mondo";
s1.equals(s2) //false
```

Stringhe in Java

```
String s = "Ciao";
s = s + " Mondo!"; // crea un NUOVO oggetto, concatenazione

char[] charArray = {'C','i','a','o','_','1','2','3','.'};
String s1 = new String(); //vuota
String s2 = new String(s); // copia
String s3 = new String(charArray)
String s4 = new String(charArray,4,2); // subset
```

Confronti tra Stringhe

`equals` e operatore `==`

- `equals` verifica che le due stringhe siano identiche come caratteri
- `==` verifica che le due stringhe siano lo stesso oggetto

```
String s1 = "Ciao";
String s2 = "Ciao";
s.equals("Ciao"); //true
(s1 == s2); // false
```

Confronti tra Stringhe

- `equalsIgnoreCase` ignora le differenze tra lettere e maiuscole
- `compareTo` restituisce valori maggiori, minori o uguali a zero se la prima stringa è lessicograficamente maggiore, minore, o uguale alla seconda
- `regionMatches` compara sotto-sezioni di due stringhe, con indici
- `startsWith` e `endsWith` verificano che la stringa inizi o finisca con una sottostringa (volendo con un offset)

```
String s = "start";
s.startsWith("art",2); // true
```

Caratteri e sotto-Stringhe

- `indexOf` restituisce l'indice della prima occorrenza di un carattere o di una stringa

```
String s = "start";
s.indexOf('t'); // = 1
s.indexOf("art"); // = 2
s.indexOf('q'); // = -1
```

- `lastIndexOf` restituisce l'indice della ultima occorrenza di un carattere o di una stringa
- `substring(i,[j])` restituisce la sottostringa di caratteri dall'i-esimo al j-esimo (o all'ultimo se il secondo non è definito)

Caratteri e sotto-Stringhe

- `replace(i,j)` sostituisce le occorrenze di un carattere `i` con un altro carattere `j`
- `toUpperCase` converte i caratteri in maiuscolo
- `toLowerCase` converte i caratteri in minuscolo
- `charAt(i)` restituisce il carattere all'i-esima posizione

https://docs.oracle.com/javase/8/docs/api/java/lang/String.html

Librerie e documentazione

Java fornisce delle classi già pronte, pensate per risolvere problemi comuni:

- Stringhe
- Array
- Collezioni

Fanno parte della Java Standard Library.

Per imparare a programmare bene in Java è fondamentale abituarsi a **leggere la documentazione ufficiale**, perché ci dice quali classi esistono, quali metodi offrono e come usarli.

https://docs.oracle.com/javase/8/docs/api/