



UNIVERSITÀ DEGLI STUDI DELL'AQUILA

# Laboratorio di Programmazione ad Oggetti

Ph.D. Juri Di Rocco  
[juri.dirocco@univaq.it](mailto:juri.dirocco@univaq.it)  
<http://jdirocco.github.io/>





# Sommario

---

- › String, StringBuffer e StringBuider
- › Control-flow statements



# Stringhe (1)

---

- › Corrispondono a sequenze di caratteri Unicode
- › Java non prevede un tipo string predefinito ma è contenuto nella libreria (package `java.lang`)
- › Java le stringhe sono **immutabili** ovvero una volta creato un oggetto di tipo `String` non è più possibile modificarlo
- › Da notare che in `String` i metodi che **manipolano** la stringa restituiscono **sempre** una nuova stringa
- › Essendo una classe contiene metodi che permettono di manipolare la stringa



## Stringhe (2)

---

### › Costruttori

- `public String()`
  - › Crea un stringa vuota
  - › Tale costruttore **non** è necessario poiché le stringhe sono immutabili
- `public String(String original)`
  - › Inizializza una nuova stringa che rappresenta la stessa sequenza di caratteri dell'argomento
  - › E' una copia dell'originale
  - › Tale costruttore **non** è necessario poiché le stringhe sono immutabili
  - › Esempio
    - `String news = new String("news");`
- Ne esistono altri di costruttori



## Stringhe (3)

---

### › Metodo `length()`

- Determina il numero di caratteri contenuti nella Stringa
- Esempio

```
› String greeting = "hello";  
› int n = greeting.length(); //Restituisce 5
```

### › Metodo `charAt(n)`

- Determina il carattere n-esimo della stringa
- Da notare che si inizia a contare da 0
- Esempio

```
› String greeting = "hello";  
› char c = greeting.charAt(1); //Restituisce e
```



## Stringhe (4)

---

- › Metodo `String substring(int beginIndex)`
  - Restituisce una nuova stringa che è una sottostringa dell'originale
  - La sottostringa inizia con il carattere all'indice `beginIndex` specificato fino al termine della stringa
  
- › Esempi
  - › `"unhappy".substring(2)` restituisce `"happy"`
  - › `"Harbison".substring(3)` restituisce `"bison"`



## Stringhe (5)

---

- › Metodo `String substring(int begin, int end)`
  - Restituisce una nuova stringa che è una sottostringa dell'originale
  - La sottostringa inizia con il carattere all'indice `begin` specificato fino al carattere con indice `end-1`
  
- › Esempi
  - › `"hamburger".substring(4, 8)` restituisce `"urge"`
  - › `"smiles".substring(1, 5)` restituisce `"mile"`



## Stringhe (6)

---

- › Metodo `String[] split(String regex)`
  - Suddivide la stringa in un array di stringhe utilizzando come separatore l'argomento `regex`
- › Esempio
  - `"boo:and:foo".split(":")`
  - ```
{  
  "boo",  
  "and",  
  "foo"  
}
```





## Stringhe (7)

---

- › Metodo `static String valueOf(boolean b)`
  - Ritorna la rappresentazione in stringa dell'argomento booleano
  - Se l'argomento è `true` viene ritornato `true` altrimenti `false`
- › Metodo `static String valueOf(int i)`
  - Ritorna la rappresentazione in stringa dell'argomento `int`
  - Esempio `String.valueOf(100)` ritorna `"100"`
- › Sono presenti anche metodi per `char`, `long`, `float` e `double`



## Stringhe (8)

---

- › I letterali di tipo string sono posti in una speciale area di memoria della JVM detta **pool di stringhe**
- › Permette di riusare le stringhe già istanziate
- › Possibile in quanto le stringhe sono **immutabili** pertanto è possibile riusarle



# Stringhe (9)

---

## › Esempio

```
public class EsempiPoolStringhe {  
    private static String LINGUAGGIO_JAVA = "Java";  
    public static void main(String[] args) {  
        String a = "Java";  
        String b = "Java";  
        String c = new String("Java");  
        System.out.println(a==b);  
        System.out.println(b==c);  
        System.out.println(a==LINGUAGGIO_JAVA);  
        System.out.println(c==LINGUAGGIO_JAVA);  
        System.out.println(Test.LINGUAGGIO_JAVA==LINGUAGGIO_JAVA);  
        System.out.println(a==Test.LINGUAGGIO_JAVA);  
    }  
}  
  
class Test { public static String LINGUAGGIO_JAVA = "Java"; }
```



# Stringhe (10)

---

## Concatenamento

- › Per concatenare due stringhe si può utilizzare l'operatore +
- › E' possibile concatenare ad una stringa anche un valore diverso dalla stringa
- › Da notare che viene creata sempre una nuova stringa
- › USO CONVERSIONE/PROMOZIONE DI STRINGHE?
- › Esempi

```
String expletive = "Expletive";
```

```
String PG13 = "deleted";
```

```
String message = expletive + PG13;
```

```
int age = 13;
```

```
String rating = "PG" + age;
```

```
System.out.println("The answer is " + answer)
```



# Stringhe (11)

---

## Uguaglianza

- › Viene utilizzato il metodo `equals (String s)`
- › Metodo restituisce `true` se sono le stringhe sono uguali (ovvero se i caratteri sono gli stessi) `false` altrimenti
- › Per verificare l'uguaglianza a prescindere dalle letterale maiuscole/minuscole utilizzare `equalsIgnoreCase()`
- › Da notare che non si deve utilizzare l'operatore `==` in quanto confronta i riferimenti alle stringhe e non il contenuto



# Stringhe (12)

---

## › Esempio

```
public void esegui(String comando) {  
    if (comando.equals("AVANTI")) {  
  
        } else if {  
        }  
  
}
```

```
public void esegui(String comando) {  
    if ("AVANTI".equals(comando)) {  
  
        } else if {  
        }  
  
}
```



# StringBuilder e StringBuffer(1)

- › Sequenza **mutabile** di caratteri
- › **thread-safe** -> la stringa può essere condivisa in modo sicuro tra più thread
- › Metodi più importanti `append` e `insert`
- › Java 5.0 ha introdotto **StringBuilder** come sostituta di **StringBuffer**

## StringBuffer vs StringBuilder

1. Thread-Safe
2. Synchronized
3. Since Java 1.0
4. Slower

1. Not Thread-Safe
2. Not Synchronized
3. Since Java 1.5
4. Faster

String vs StringBuffer vs StringBuilder



## StringBuilder (2)

---

### › Esempio

```
public class StringsDemo {  
    public static void main(String[] args) {  
        String palindrome = "Dot saw I was Tod";  
        int len = palindrome.length();  
        StringBuilder dest = new StringBuilder(len);  
  
        for (int i = (len - 1); i >= 0; i--) {  
            dest.append(palindrome.charAt(i));  
        }  
        System.out.println(dest.toString());  
    }  
}
```





# Flusso di controllo (1)

---

- › Permettono di variare il normale flusso di esecuzione di un programma
- › Tipi
  - Condizionali
    - › if, if-else, switch-case
  - Cicli
    - › while, do-while, for
  - Branching
    - › break, continue, label:, return



# Flusso di controllo: return

## › Sintassi

- `return <valore ritorno>;`

## › Due scopi

- Specificare il valore restituito da un metodo (a meno che non abbia il valore di ritorno di tipo `void`)
- Fare in modo che l'esecuzione del metodo termini e quel valore venga immediatamente restituito

## › Esempio

```
public void esegui(String comando) {  
    if ("AVANTI".equals(comando) {  
        return;  
    } else if {  
    }  
}
```

```
public int somma(int x, int y) {  
    return x + y;  
}
```



# Flusso di controllo: if (1)

---

## › Sintassi

```
if (espressione booleana) {  
    statement(s)  
}
```

```
if (espressione booleana) {  
    statement(s)  
} else {  
    statement(s)  
}
```



## Flusso di controllo: if (2)

---

```
if (espressione booleana) {  
    statement(s)  
}  
else if (espressione booleana) {  
    statement(s)  
}  
else if (espressione booleana) {  
    statement(s)  
}  
else {  
    statement(s)  
}
```

Nota: Parentesi possono essere sostituite con una singola istruzione



# Flusso di controllo: if (3)

---

## › Esempi

```
if (yourSales >= target) {  
    performance = "Satisfactory";  
    bonus = 100;  
}
```

```
if (yourSales >= target) {  
    performance = "Satisfactory";  
    bonus = 100;  
} else {  
    performance = "Unsatisfactory";  
    bonus = 0;  
}
```



# Operatore Ternario if-else

---

› Generalmente viene utilizzato quando è necessario **assegnare** il valore di una variabile a seconda del verificarsi o meno di una condizione

› Sintassi

Condizione ? Espressione1 : Espressione2

› Esempio

```
int i = (x > 10) ? 100 : -1;
```



# Flusso di controllo: while (1)

---

› Esegue un'istruzione o un blocco di istruzioni fino a quando una determinata condizione restituisce `true`

› Sintassi

```
while (espressione) {  
    statement(s)  
}
```

› Espressione deve restituire un valore `boolean`

› Se l'espressione restituisce sempre `false` il blocco non viene mai eseguito



# Flusso di controllo: while (2)

```
public class Retirement {  
    public static void main(String[] args) {  
        double goal = Double.parseDouble( args[ 0 ] );  
        System.out.println("Obiettivo: " + goal );  
        double payment = Double.parseDouble( args[ 1 ] );  
        System.out.println("Rata annuale: " + payment);  
        double interestRate = Double.parseDouble( args[ 2 ] );  
        System.out.println( "Tasso interesse percentuale: " + interestRate );  
        double balance = 0;  
        int years = 0;  
        double interest = 0;  
        while (balance < goal) {  
            balance += payment;  
            interest = balance * interestRate / 100;  
            balance += interest;  
            years++;  
        }  
        System.out.println("Puoi ritirare in " + years + " anni.");  
    }  
}
```





## Flusso di controllo: do-while

---

- › A differenza del `while` viene eseguito prima il blocco e poi viene valutata l'espressione
- › Blocco viene ripetuto fintanto che l'espressione è `true`

### Sintassi

```
do {  
    statement(s)  
} while (expression);
```



# Flusso di controllo: for (1)

---

- › Costrutto generico che supporta le iterazioni controllate da un contatore o da una variabile analoga aggiornata dopo ciascuna iterazione
- › Sintassi

```
for ([inizializzazione]; [condizione terminazione];  
    [incremento]) {  
    statement  
}
```



## Flusso di controllo: for (2)

---

Caratterizzato da 3 fasi

- › Inizializzazione
  - Eseguita prima di iniziare la prima iterazione
- › Test sulla condizione per proseguire l'esecuzione del corpo del ciclo
- › Alla fine di ogni iterazione qualche forma di avanzamento



# Flusso di controllo: for (3)

---

## › Esempi

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

```
for (int i = 10; i >= 0; i--) {  
    System.out.println(i);  
}
```



## Flusso di controllo: for (4)

---

```
for (int i = 1, j = i + 10; i < 5; i++, j = i * 2) {  
    System.out.println("i = " + i + " j = " + j);  
}
```

### Ciclo infinito

```
for (; ; ) {  
    System.out.println(i);  
}
```



# Flusso di controllo: foreach (1)

---

- › Java 5 ha introdotto una sintassi nuova e sintetica da utilizzare con array e contenitori eliminando l'utilizzo dell'indice per iterare sugli elementi
- › foreach si incarica di produrre ogni voce automaticamente
- › Sintassi

```
for (Type t : elements) {  
}
```



## Flusso di controllo: foreach: Array (2)

---

```
int sum(int[] a) {  
    int result = 0;  
    for (int j=0; j< a.length; j++) {  
        result += a[j];  
    }  
    return result;  
}
```

```
int sum(int[] a) {  
    int result = 0;  
    for (int i : a) {  
        result += i;  
    }  
    return result;  
}
```



## Flusso di controllo: foreach: Collection (3)

---

```
void cancelAll(Collection<TimerTask> c) {  
    for (Iterator<TimerTask> i = c.iterator();  
         i.hasNext(); )  
        i.next().cancel();  
}
```

```
void cancelAll(Collection<TimerTask> c) {  
    for (TimerTask t : c) {  
        //Non si puo' rimuovere t da c  
    }  
}
```





## Flusso di controllo: foreach (4)

---

- › Non è possibile utilizzare il foreach ovunque
- › In particolare,
  - Non è possibile **eliminare l'elemento i-esimo** durante l'iterazione con `Iterator`
  - Non è possibile **sostituire l'elemento i-esimo** in una lista o in un array



# Flusso di controllo: switch (1)

---

- › Costrutto if/else può risultare scomodo quando si ha a che fare con selezioni multiple che prevedono diverse alternative
- › Espressione
  - Tipi primitivi: `byte`, `short`, `char`, `int`
  - Tipi **Enumerativi**: `String`
  - Tipi classi speciali: `Character`, `Byte`, `Short`, `Integer`
- › Sintassi

```
switch (espressione) {  
    case espressione: statement(s)  
    break;  
    ...  
    default: statement(s)  
    break;  
}
```



# Flusso di controllo: switch (2)

---

## › Esempio 1

```
int choice = ...;
switch (choice) {
    case 1:
        ...

        break;
    case 2:
        ...

        break;
    case 3:
    case 4:
        ...

        break;
    default:
        ...

        break;
}
```



# Flusso di controllo: switch (3)

---

## › Esempio 2

```
switch (month.toLowerCase()) {  
    case "january":  
        monthNumber = 1;  
        break;  
    case "february":  
        monthNumber = 2;  
        break;  
    case "march":  
        monthNumber = 3;  
        break;  
    .....  
    default:  
        monthNumber = 0;  
        break;  
}
```



# Flusso di controllo: break (1)

---

- › Utilizzato all'interno di `switch`, `for`, `while`, `do-while`
- › Termina il flusso di controllo attuale e l'esecuzione passa all'istruzione successiva
- › Ha due forme
  - Non etichettata
  - Etichettata
    - › Utile quando sono presenti molti flussi di controllo annidati e si vuole uscire dal ciclo più annidato
    - › Simula in qualche modo il `goto`



## Flusso di controllo: break (2)

---

```
int[] arrayOfInts = { 32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127 };
int searchfor = 12;
int i = 0;
boolean foundIt = false;
for ( ; i < arrayOfInts.length; i++) {
    if (arrayOfInts[i] == searchfor) {
        foundIt = true;
        break;
    }
}
if (foundIt) {
    System.out.println("Found " + searchfor + " at index " + i);
} else {
    System.out.println(searchfor + "not in the array");
}
```



# Flusso di controllo: break (3)

```
public class BreakWithLabelDemo {  
    public static void main(String[] args) {  
        int[][] arrayOfInts = { { 32, 87, 3, 589 }, { 12, 1076, 2000, 8 }, { 622, 127, 77, 955 } };  
        int searchfor = 12;  
        int i = 0;  
        int j = 0;  
        boolean foundIt = false;  
        search:  
        for ( ; i < arrayOfInts.length; i++) {  
            for (j = 0; j < arrayOfInts[i].length; j++) {  
                if (arrayOfInts[i][j] == searchfor) {  
                    foundIt = true;  
                    break search;  
                }  
            }  
        }  
        if (foundIt) {  
            System.out.println("Found " + searchfor + " at " + i + ", " + j);  
        } else {  
            System.out.println(searchfor + "not in the array");  
        }  
    }  
}
```



## Flusso di controllo: continue (1)

---

- › Utilizzato all'interno di `for`, `while`, `do-while`
- › Interrompe il flusso regolare e trasferisce il controllo all'intestazione del ciclo più interno
- › Come il `break` ha due forme
  - Non etichettata
  - Etichettata





## Flusso di controllo: continue (2)

---

```
public class ContinueDemo {  
    public static void main(String[] args) {  
        StringBuffer searchMe = new StringBuffer("peter piper picked a peck of pickled peppers");  
        int max = searchMe.length();  
        int numPs = 0;  
        for (int i = 0; i < max; i++) {  
            //interested only in p's  
            if (searchMe.charAt(i) != 'p')  
                continue;  
            //process p's  
            numPs++;  
            searchMe.setCharAt(i, 'P');  
        }  
        System.out.println("Found " + numPs + " p's in the string.");  
        System.out.println(searchMe);  
    }  
}
```



# Flusso di controllo: continue (3)

```
public class ContinueWithLabelDemo {  
    public static void main(String[] args) {  
        String searchMe = "Look for a substring in me";  
        String substring = "sub";  
        boolean foundIt = false;  
        int max = searchMe.length() - substring.length();  
  
        test:  
        for (int i = 0; i <= max; i++) {  
            int n = substring.length();  
            int j = i, k = 0;  
            while (n-- != 0)  
                if (searchMe.charAt(j++) != substring.charAt(k++))  
                    continue test;  
            foundIt = true;  
            break test;  
        }  
        System.out.println(foundIt ? "Found it" : "Didn't find it");  
    }  
}
```



## Flusso di controllo: continue (4)

---

...

```
System.out.println(foundIt ? "Found it" : "Didn't find it");
```

```
}
```

```
}
```



# Break vs continue

| Parameter                                           | Break statement     | Continue statement     |
|-----------------------------------------------------|---------------------|------------------------|
| Allows you to exit from an overall loop construct   | Yes                 | No                     |
| Can be used by switch statement                     | Yes                 | No                     |
| The control exits immediately from a loop construct | Yes                 | No                     |
| Causes a loop to termination                        | Yes                 | No                     |
| Can be used with “label”                            | Yes                 | Yes                    |
| Syntax                                              | <code>break;</code> | <code>continue;</code> |



# Come leggere input da linea di comando

---

- › In Java, puoi leggere input dalla linea di comando in una delle possibili alternative:
  - Usare l'oggetto globalmente accessibile `Java Console` object ().
  - Creare una istanza della classe `Java Scanner` class.
  - Usare `Java's System.in` dato un `InputStream`.



# Usando l'oggetto Console

---

- › È la via più semplice per leggere un input da console. Un esempio di codice:

```
String input = System.console.readLine();  
System.out.println("You typed in: " + input);
```

- › La classe `Console` è molto facile e semplice da usare. Accessibile attraverso l'oggetto universalmente disponibile `System`, quindi non richiede nessun import o la creazione di nuovi oggetti.
- › Il maggior lato negativo è che alcuni Java runtime environments non permettono l'uso di `Console`. Ad esempio Eclipse e the Spring Tool Suite, disabilitano l'uso della `Console`, quindi questa alternativa non funziona in questi IDEs.



# User input con la classe the Java Scanner

- › Il modo più comune di prendere gli input da un utente usa la classe `Scanner`. Per usare tale classe:
  1. Importare il package `java.util`;
  2. Creare un istanza della classe `Scanner`
  3. Passere in all costruttore dello scanner.

```
System.out.println("What is your name?");  
Scanner scanner = new Scanner(System.in);  
String name = scanner.nextLine(); System.out.println(name + "  
is a nice name!");
```

- › La classe `Scanner` funziona in tutti gli ambienti.
- › La classe `scanner` include funzionalità per processare `int`, `float`, `boolean` e `string`-based input.
- › Il maggior lato negativo è che gli sviluppatori sono intimoriti dalla sua semantica non proprio intuitiva.



# User input con Java's System.in

---

- › Nelle prime versioni di JAVA gli sviluppatori potevano ottenere gli input dagli utenti solo concatenando Java I/O classes.

```
InputStreamReader reader = new InputStreamReader(System.in);  
BufferedReader br = new BufferedReader(reader);  
System.out.println("What is your name?");  
String input = br.readLine();  
System.out.println("Your input was: " + input);
```

- › Questa alternativa non è consigliata agli sviluppatori, soprattutto per chi si approccia per la prima volta al linguaggio. Il codice richiede diversi import ed è verboso e non semplice da capire.
- › Evitare di usare questo approccio.