

Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

Capitolo 4: Librerie, controllo del flusso e uguaglianza

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Librerie e pacchetti · Input da tastiera · Visibilità delle variabili
Classi wrapper · Selezione e cicli · Uguaglianza fra oggetti · String e StringBuilder

Indice

1	Librerie e pacchetti	1
2	Input da tastiera	2
3	Visibilità delle variabili	4
4	Classi wrapper	5
5	Istruzioni di selezione	5
6	Cicli	6
7	Uguaglianza fra oggetti	7
8	String e StringBuilder	8
9	Esercizi	11
10	Soluzioni	12

1 Librerie e pacchetti

Definizione – Libreria

Una **libreria** è una collezione di classi già scritte, che si usano nei propri programmi invece di riscriverle. La **libreria standard** è quella distribuita con Java: ne fanno parte, per esempio, **String** e **System**.

Ogni libreria è accompagnata dalla sua **documentazione**, che elenca classi e metodi con le loro intestazioni: è lì che si scopre come usare una classe. Quella della libreria standard è la documentazione ufficiale di Java, e le librerie di terze parti la distribuiscono di solito nello stesso formato.

Definizione – Pacchetto

Un **pacchetto** (*package*) è un gruppo di classi correlate, con un nome formato da identificatori separati da punti, come **java.util**. Le classi di una libreria sono distribuite organizzate in pacchetti.

Pacchetto	Contenuto
<code>java.lang</code>	le classi fondamentali, come String , System e Math ; si importa automaticamente
<code>java.util</code>	classi di utilità, come Scanner e Random
<code>java.io</code>	input e output, come BufferedReader
<code>java.net</code>	comunicazione in rete
<code>java.awt</code> , <code>javax.swing</code>	interfacce grafiche: Swing offre i componenti già pronti (<i>widget</i>), come le finestre di dialogo di JOptionPane

Definizione – L'istruzione **import**

Per usare una classe di un pacchetto con il suo nome semplice, la si importa all'inizio del file, prima della classe:

- `import java.util.Random;` importa la sola classe **Random**;
- `import java.util.*;` importa tutte le classi del pacchetto.

Le classi di `java.lang` non vanno importate.

Senza **import** il compilatore non riconosce il nome: una classe che usa **Scanner** senza importarlo dà l'errore *cannot find symbol, symbol: class Scanner*. In alternativa si può scrivere ogni volta il nome completo, `java.util.Scanner`.

Osservazione – Che cosa fa `import`

`import` abbrevia soltanto i nomi: non copia nessuna classe nel programma. Con `import java.util.*;` e con `import java.util.Random;` il compilatore produce lo stesso bytecode. Importare le singole classi rende evidente quali classi usa il file ed evita i conflitti fra classi con lo stesso nome in due pacchetti importati per intero, come `Date` in `java.util` e in `java.sql`.

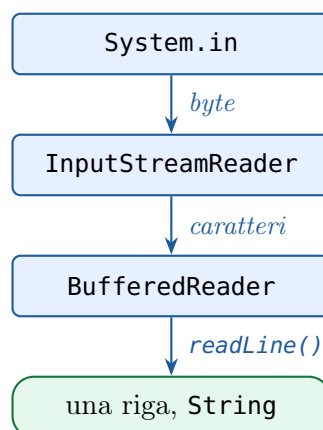
2 Input da tastiera

Definizione – Stream

Uno **stream** (flusso) è una sequenza di dati che arrivano uno dopo l'altro. L'input da tastiera è uno stream: l'oggetto `System.in` rappresenta l'input standard, e ne legge un byte per volta.

Per leggere una riga intera si costruisce una catena di tre oggetti, ciascuno dei quali riceve nel costruttore quello precedente:

1. `System.in` legge i byte digitati;
2. un `InputStreamReader` li trasforma in caratteri;
3. un `BufferedReader` raccoglie i caratteri e li consegna una riga alla volta: il suo metodo `readLine()` restituisce come `String` la riga digitata, fino all'Invio escluso.



La lettura bufferizzata: ogni oggetto legge da quello sopra di sé.

```
import java.io.*;

public class Tastiera {
    public static void main(String[] args) throws IOException {
        BufferedReader r =
            new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Come ti chiami? ");
        String nome = r.readLine();
        System.out.println("Ciao " + nome);
    }
}
```

Se si digita **Mario** e Invio, il programma stampa **Ciao Mario**. `readLine` può segnalare un errore di input, un'eccezione `IOException` che il compilatore obbliga a dichiarare: per ora basta aggiungere `throws IOException` all'intestazione di `main`.

Definizione – Allocazione implicita

Un oggetto creato con `new` e passato direttamente come parametro, senza assegnarlo a una variabile, è allocato in modo **implicito**. Si fa così con gli oggetti che servono una volta sola: nella catena, `InputStreamReader` serve solo a costruire il `BufferedReader`, che invece si conserva in `r` perché si usa per ogni lettura.

L'`InputStreamReader` anonimo resta in vita finché serve: non lo raggiunge nessuna variabile, ma lo raggiunge il `BufferedReader`, e il garbage collector non lo recupera.

La classe `Scanner`. Per la tastiera c'è anche una classe di livello più alto, `Scanner` del pacchetto `java.util`. Si costruisce su `System.in`, e il suo metodo `nextLine()` legge una riga. La stessa libreria offre così più strumenti per lo stesso problema, e si sceglie quello adatto al contesto.

```
import java.util.Scanner;

public class NomeCognome {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Inserisci il nome: ");
        String nome = in.nextLine();
        System.out.print("Inserisci il cognome: ");
        String cognome = in.nextLine();
        System.out.println(nome + " " + cognome);
    }
}
```

3 Visibilità delle variabili

Definizione – Visibilità di una variabile locale

La **visibilità** (*scope*) di una variabile è la parte del programma in cui la si può usare. Una variabile locale è visibile dalla sua dichiarazione fino alla parentesi graffa che chiude il **blocco** in cui è dichiarata; un parametro, in tutto il metodo.

Un blocco è un gruppo di istruzioni fra parentesi graffe, come il corpo di un metodo o il ramo di un `if`. In Java, come in C, i blocchi li delimitano le graffe; in Python l'indentazione.

Nomi semplici e nomi qualificati. Nel programma `NomeCognome` il nome `in` compare due volte, senza ambiguità:

- `System.in` è un nome **qualificato**: il campo `in` della classe `System`;
- `in`, da solo, è la variabile locale di tipo `Scanner`, visibile in tutto il corpo di `main`.

Un nome semplice si cerca a partire dal blocco più interno verso quelli esterni; un nome qualificato dice esplicitamente dove cercare.

Proposizione – Due errori di visibilità

- Una variabile locale non si può usare fuori dal suo blocco.
- In un blocco interno non si può dichiarare una variabile locale con lo stesso nome di un'altra ancora visibile: a differenza del C, Java non la maschera, ma dà un errore di compilazione.

```
public class Visibilita {
    public static void main(String[] args) {
        int x = 1;
        if (args.length > 0) {
            String primo = args[0];
            int x = 2;           // errore: x è già definita
        }
        System.out.println(primo); // errore: primo non è visibile qui
    }
}
```

Il compilatore segnala *variable x is already defined in method main(String[])* e *cannot find symbol, symbol: variable primo*.

4 Classi wrapper

Definizione – Classi wrapper

A ogni tipo primitivo corrisponde una **classe wrapper** (involucro), come **Integer** per **int** e **Double** per **double**. Offre i metodi per passare dal tipo primitivo agli oggetti e viceversa, fra cui la conversione da stringa:

```
int n = Integer.parseInt(s);
```

restituisce l'intero scritto nella stringa *s*.

Ciò che si legge da tastiera è sempre una stringa: se l'utente digita **123**, il programma riceve i caratteri **"123"**. Per usarli come numero si convertono:

```
String s = "123";  
int n = Integer.parseInt(s);  
System.out.println(n + 1);    // 124: somma fra interi  
System.out.println(s + 1);    // 1231: concatenazione di stringhe
```

`parseInt` è un metodo **static**: si invoca sulla classe **Integer**, senza oggetto, come **main**. Se la stringa non contiene un intero, come **"12a"**, si ha un errore di esecuzione, **NumberFormatException**.

5 Istruzioni di selezione

Definizione – if

if (condizione) **blocco1** **else** **blocco2** esegue **blocco1** se la condizione è vera, **blocco2** se è falsa. Il ramo **else** si può omettere.

```
if (a > b) {  
    System.out.println(a);  
} else {  
    System.out.println(b);  
}
```

Il frammento stampa il maggiore fra **a** e **b**.

Definizione – switch

switch sceglie fra più casi secondo il valore di un'espressione:

- l'esecuzione comincia dal **case** con l'etichetta uguale al valore;
- prosegue fino a un **break**, anche nelle istruzioni dei **case** successivi;
- più etichette di seguito condividono le stesse istruzioni.

```
public class GiorniMese {  
    public static void main(String[] args) {  
        int mese = Integer.parseInt(args[0]);  
        int giorni = 0;  
        switch (mese) {  
            case 4: case 6: case 9: case 11:  
                giorni = 30;  
                break;  
            case 1: case 3: case 5: case 7: case 8: case 10: case 12:  
                giorni = 31;  
                break;  
            case 2:  
                giorni = 28;  
                break;  
        }  
        System.out.println(giorni);  
    }  
}
```

Con `java GiorniMese 4` stampa **30**, con `java GiorniMese 2` stampa **28** (anni non bise-stili). Una catena di **if** avrebbe richiesto un confronto per ciascuno dei dodici mesi. Senza il primo **break**, per aprile l'esecuzione proseguirebbe nel gruppo dei mesi di 31 giorni fino al suo **break**, e il programma stamperebbe **31**.

6 Cicli

Definizione – while e for

- **while** (condizione) blocco ripete il blocco finché la condizione è vera; la condizione si controlla prima di ogni ripetizione.
- **for** (inizializzazione; condizione; aggiornamento) blocco esegue l'inizializzazione, poi ripete il blocco finché la condizione è vera, eseguendo l'aggiornamento dopo ogni ripetizione.

while	for
<pre>int i = 0; while (i < 10) { System.out.print("*"); i++; }</pre>	<pre>for (int i = 0; i < 10; i++) { System.out.print("*"); }</pre>

Entrambi stampano dieci asterischi sulla stessa riga. Il **for** riunisce in una riga inizializzazione, condizione e aggiornamento del contatore, che nel **while** stanno in tre punti diversi: si usa quando il numero di ripetizioni è fissato da un contatore. Il **while** si usa quando dipende da ciò che succede nel ciclo, come in una lettura che continua finché l'utente non scrive una certa parola.

7 Uguaglianza fra oggetti

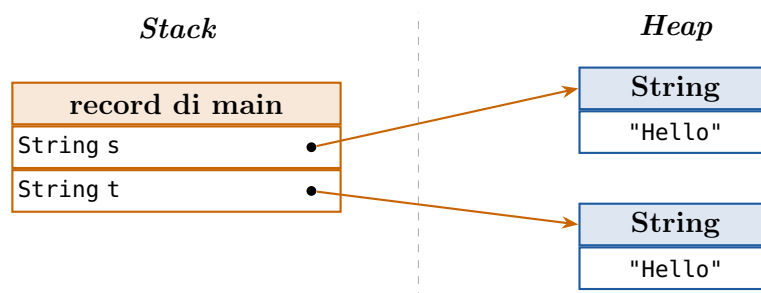
Una stringa si crea in due modi: con un letterale, `s = "Hello"`, oppure con `new`, `t = new String("Hello")`. La forma con il letterale vale solo per le stringhe; gli oggetti delle altre classi si creano con `new`.

Definizione – Uguaglianza superficiale e profonda

- `s == t`, fra due riferimenti, è vero se `s` e `t` denotano lo **stesso oggetto**: è l'**uguaglianza superficiale**.
- `s.equals(t)` è un metodo che confronta il **contenuto** dei due oggetti, se la classe lo definisce così: è l'**uguaglianza profonda**.

Per le stringhe, `equals` confronta i caratteri.

```
String s = "Hello";
String t = new String("Hello");
System.out.println(s == t);           // false: due oggetti diversi
System.out.println(s.equals(t));      // true: stessi caratteri
```



s e *t* denotano due oggetti diversi con gli stessi caratteri: `s == t` è falso, `s.equals(t)` è vero.

Nota – Due letterali uguali

La JVM tiene una sola copia di ogni letterale stringa: con `String u = "Hello";` anche `s == u` è vero. Per questo `==` sembra funzionare con le stringhe scritte nel programma e sbaglia con quelle calcolate o lette: il contenuto si confronta sempre con `equals`.

Si usa `==` quando si vuole sapere proprio se due variabili denotano lo stesso oggetto, e per controllare se un riferimento vale `null` (`s == null`); per il contenuto si usa `equals`. Non tutte le classi però definiscono `equals` come uguaglianza profonda: va verificato nella documentazione della classe (Esercizio 1).

8 String e StringBuilder

Definizione – Oggetti immutabili

Un oggetto è **immutabile** se nessun metodo della sua classe ne modifica il contenuto. Le stringhe sono immutabili: i metodi di `String`, come `concat`, restituiscono una stringa nuova e lasciano com'è quella su cui sono invocati.

```
public class HamburgerString {  
    public static void main(String[] args) {  
        String s, t;  
        s = new String("ham");  
        t = s.concat("burger");  
        System.out.println(s);    // ham  
        System.out.println(t);    // hamburger  
    }  
}
```

Nello heap ci sono "ham", denotata da `s`, "burger", che non ha una variabile perché è passata direttamente a `concat`, e la nuova stringa "hamburger", denotata da `t`.

Definizione – StringBuilder

`StringBuilder` è una classe di sequenze di caratteri **modificabili**. Il metodo

`s.append(x)`

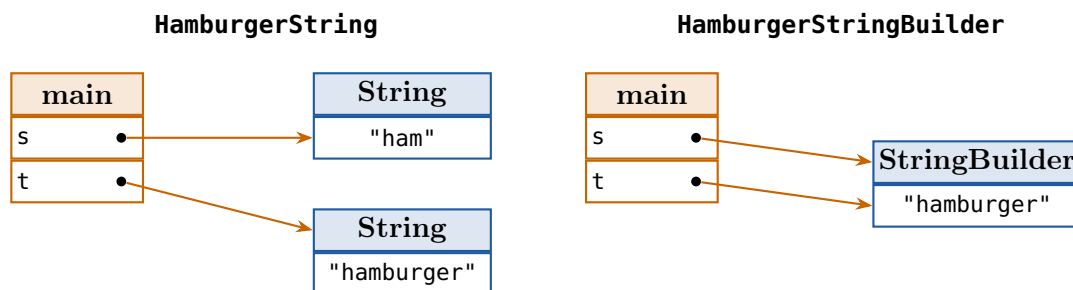
aggiunge `x` in fondo all'oggetto `s` stesso e restituisce il riferimento a `s`. Con `s.toString()` si ottiene la `String` costruita.

```

public class HamburgerStringBuilder {
    public static void main(String[] args) {
        StringBuilder s, t;
        s = new StringBuilder("ham");
        t = s.append("burger");
        System.out.println(s);    // hamburger
        System.out.println(t);    // hamburger
    }
}

```

`append` ha modificato l'oggetto denotato da `s` e ne ha restituito il riferimento, che è stato assegnato a `t`: le due variabili denotano lo stesso oggetto (`s == t` è vero), e la stringa "ham" non c'è più.



La memoria alla fine dei due programmi (senza il letterale "burger"): con `String` due oggetti, con `StringBuilder` un solo oggetto modificato, denotato da entrambe le variabili.

Definizione – Side effect

Quando due variabili denotano lo stesso oggetto, una modifica fatta attraverso una si vede anche dall'altra: è un **side effect** (effetto collaterale), detto anche **interferenza**. Con i tipi primitivi non succede, perché ogni variabile ha la sua copia del valore (Capitolo 3, Sezione 10).

Con gli oggetti immutabili, come le stringhe, il side effect non può capitare; con quelli modificabili sì. È una delle cause più frequenti di errore nella programmazione a oggetti: si cambia un oggetto attraverso una variabile e cambiano anche valori a cui non si pensava. Se nel programma sopra servisse ancora "ham", sarebbe perso.

Riepilogo

- **Pacchetti:** `import java.util.Scanner;` o `import java.util.*;`; `java.lang` non si importa; `import` non aggiunge codice.
- **Tastiera:** `BufferedReader` su `InputStreamReader` su `System.in` e `readLine()`, oppure `Scanner` e `nextLine()`.
- **Visibilità:** dalla dichiarazione alla fine del blocco; niente variabili locali con lo stesso nome in blocchi annidati.
- **Wrapper:** `Integer.parseInt(s)` converte una stringa in `int`.
- **Selezione:** `if/else`; `switch` esegue dal `case` scelto fino al `break`.
- **Cicli:** `while` finché la condizione è vera; `for` con inizializzazione, condizione e aggiornamento.
- **Uguaglianza:** `==` stesso oggetto, `equals` stesso contenuto (se la classe lo definisce così).
- **Side effect:** `String` è immutabile; `StringBuilder.append` modifica l'oggetto, visibile da tutte le variabili che lo denotano.

9 Esercizi

Esercizio 1 – Due StringBuilder uguali?

Che cosa stampa il frammento seguente? Come si controlla se `s` e `t` contengono gli stessi caratteri?

```
StringBuilder s = new StringBuilder("Hello");  
StringBuilder t = new StringBuilder("Hello");  
System.out.println(s == t);  
System.out.println(s.equals(t));
```

Soluzione a pagina 12 →

10 Soluzioni

Soluzione dell'Esercizio 1 – Due StringBuilder uguali?

Stampa **false** due volte. **s** e **t** sono due oggetti distinti, creati da due **new**, quindi **s == t** è falso. **StringBuilder** non definisce **equals** come uguaglianza profonda: il suo **equals** confronta i riferimenti, come **==**, ed è falso anche se i caratteri sono gli stessi.

Per confrontare il contenuto si trasformano i due oggetti in stringhe e si usa l'**equals** di **String**, che confronta i caratteri:

```
System.out.println(s.toString().equals(t.toString()));    // true
```

[← Torna all'Esercizio 1, pagina 11](#)