

Appunti di Progettazione del Software

Canale 2 · Prof. Massimo Mecella

Capitolo 3: Introduzione a Java

Autore: Tommaso C.

Appunti tratti dalle lezioni in aula e dal materiale del corso

Argomenti: Il linguaggio Java · Struttura di un programma · Compilazione ed esecuzione
Errori · Oggetti e metodi · La classe String · Argomenti da riga di comando
Variabili e tipi · Stack, heap e garbage collection · Riferimenti, `new` e `null`

Indice

1	Il linguaggio Java	1
2	La struttura di un programma	2
3	Compilazione ed esecuzione	4
4	Errori	5
5	Oggetti e invocazione di metodi	6
6	La classe String	7
7	Argomenti da riga di comando	9
8	Variabili e tipi	9
9	Stack, heap e garbage collection	11
10	Riferimenti, new e null	13
11	Esercizi	16
12	Soluzioni	17

1 Il linguaggio Java

Definizione – Java

Java è un linguaggio di programmazione ad alto livello, nato nel 1995, **orientato agli oggetti**. Supporta anche il paradigma **imperativo**, in cui l'unità fondamentale è la procedura, e quello **funzionale**.

L'approccio a oggetti si può seguire in qualsiasi linguaggio, con la dovuta disciplina; i linguaggi orientati agli oggetti, come Java, C++, C# e Objective-C, lo rendono più semplice. Sono simili fra loro, e ciò che si impara su Java vale in gran parte anche per gli altri.

Caratteristica	Che cosa vuol dire
Indipendente dalla piattaforma	lo stesso programma gira su Windows, Unix e macOS: è la portabilità del Capitolo 1
Librerie ricche	la libreria standard offre classi pronte per quasi ogni esigenza; è una delle ragioni principali del successo di Java
Progettato per Internet	la programmazione di rete fa parte della libreria standard; in C si usano direttamente i servizi del sistema operativo, ed è molto più complicato
Basato su una macchina virtuale	il programma è eseguito dalla JVM (Sezione 3), che gli dà la portabilità
Sicuro	la JVM impedisce a un programma scaricato da Internet di compiere operazioni di basso livello sul computer che lo esegue

Sulla sicurezza si fondò il successo di Java nel web degli anni Novanta, fatto di pagine statiche. Le **applet** erano applicazioni scaricate da una pagina web ed eseguite dalla JVM, senza il rischio di far girare codice malevolo sul proprio computer. Oggi quel ruolo lo svolge JavaScript; le applet sono state abbandonate, e da Java 26 la loro libreria non esiste più.

Nota – Versioni di Java

Le funzionalità a oggetti di Java sono cambiate poco nelle versioni successive, che hanno soprattutto integrato altri paradigmi: tutto ciò che c'è in Java 8 c'è anche nelle versioni seguenti. Le versioni con supporto a lungo termine (**LTS**) sono la 8, la 11, la 17, la 21 e la 25.

2 La struttura di un programma

Definizione – Programma Java

Un programma Java è una raccolta di una o più **classi**:

- una classe contiene uno o più **metodi**;
- un metodo contiene le **istruzioni**;
- la classe da cui si avvia il programma contiene il metodo speciale **main**.

Il primo programma stampa una riga sulla console:

```
public class Primo {  
    public static void main(String[] args) {  
        System.out.println("Questo è il mio primo programma Java.");  
    }  
}
```

`public class Primo` dichiara la classe `Primo`, visibile da tutto il programma. Più classi possono avere un `main`: si avvia quella indicata quando si esegue il programma (Sezione 3).

Definizione – Il metodo `main`

L'esecuzione di un programma comincia dal metodo

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

<code>public</code>	il metodo è visibile da tutto il programma, e la JVM lo può invocare
<code>static</code>	il metodo si invoca senza un oggetto
<code>void</code>	il metodo non restituisce alcun valore
<code>String[] args</code>	il parametro, un array di stringhe: gli argomenti scritti nel comando che avvia il programma (Sezione 7)

Osservazione – Il problema del bootstrap

Nella computazione a oggetti un metodo si invoca su un oggetto, e ogni oggetto ne chiama altri. All'avvio però nessun oggetto del programma esiste ancora: è il **problema del bootstrap** (dell'avvio). Java lo risolve con `main`, che è `static`: la JVM lo invoca senza oggetto, e da lì il programma comincia a usare gli oggetti.

Nota – Da Java 25

Da Java 25 la JVM avvia anche un `main` più semplice, senza `public`, `static` e parametri, come `void main()`: in quel caso crea prima un oggetto della classe. La forma `public static void main(String[] args)` resta valida in ogni versione.

Definizione – Segnatura e intestazione di un metodo

La **segnatura** (o firma, *signature*) di un metodo è il suo nome con il tipo, il numero e la posizione dei parametri. L'**intestazione** è la segnatura più il tipo del risultato:

`String concat(String s)`

è l'intestazione del metodo `concat` della classe `String`.

Intestazione e corpo. L'intestazione della classe, `public class MioProgramma`, è seguita dal **corpo della classe**, fra le parentesi graffe. Allo stesso modo l'intestazione di un metodo è seguita dal **corpo del metodo**, che contiene le istruzioni. Il corpo della classe contiene i metodi: sono i servizi che la classe offre (Capitolo 2).

```
// commenti sulla classe
public class MioProgramma {

    // commenti sul metodo
    public static void main(String[] args) {
        // istruzioni
    }
}
```

Commenti. I commenti sono la **documentazione inline** del programma: servono a spiegarne lo scopo e le funzionalità, non ne cambiano il funzionamento e il compilatore li ignora. Si possono mettere ovunque, in due forme: `//` commenta fino alla fine della riga, `/* ... */` delimita un commento anche su più righe.

Spazi bianchi. Spazi, righe vuote e tabulazioni sono gli **spazi bianchi**. Separano le parole e i simboli del programma; per il resto il compilatore li ignora, e il programma si può formattare come si vuole. Una formattazione coerente lo rende leggibile. A differenza di Python, in Java l'indentazione non cambia il significato del programma, che è dato dalle parentesi e dalle intestazioni: come in C.

File e nomi. Il codice sorgente si scrive in un file di testo con estensione `.java`. Una classe `public` deve stare in un file con il suo stesso nome: **Primo** va in **Primo.java**. Java

distingue maiuscole e minuscole (*case sensitive*): **Primo** e **primo** sono nomi diversi, e la classe **Primo** in un file **primo.java** dà l'errore *class Primo is public, should be declared in a file named Primo.java*.

Per convenzione i nomi delle classi cominciano con la lettera maiuscola, come tutte le classi della libreria (**String**, **System**). Il compilatore non lo impone, ma chi legge riconosce subito una classe e il codice è più facile da mantenere e da rivedere.

3 Compilazione ed esecuzione

Definizione – Scrivere, compilare ed eseguire

1. Si scrive il testo del programma nel file **NomeClasse.java**.
2. Lo si compila con il compilatore **javac**: **javac NomeClasse.java** produce il file **NomeClasse.class**, che contiene il **bytecode**.
3. Lo si esegue con la **Java Virtual Machine (JVM)**: **java NomeClasse**.

```
$ javac Primo.java
```

```
$ java Primo
```

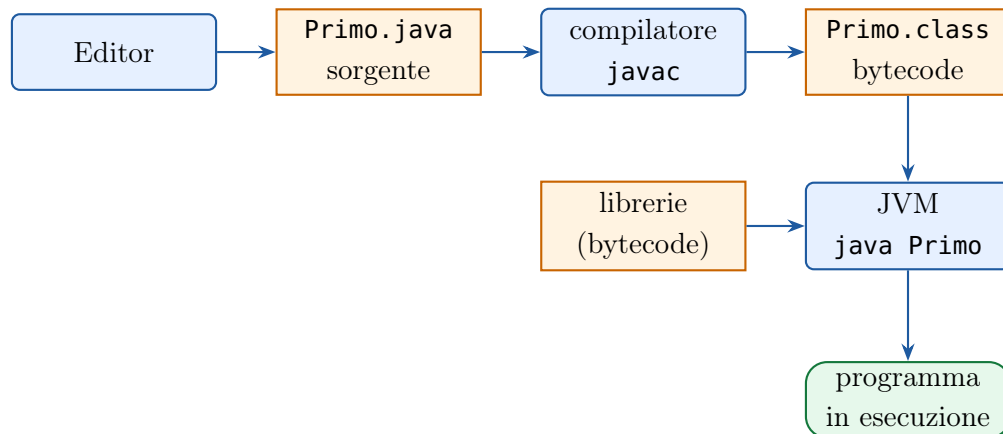
Questo è il mio primo programma Java.

Al comando **java** si dà il nome della classe, senza **.class**: la JVM cerca da sola il file del bytecode, che va prima prodotto con **javac**.

Definizione – Bytecode e JVM

Il **bytecode** è il linguaggio macchina di un processore virtuale, uguale su tutte le piattaforme. La **JVM** è il programma che lo esegue su una macchina reale, traducendolo nelle istruzioni del suo processore.

È il meccanismo della portabilità: il programma si compila una volta sola in bytecode, e lo stesso file **.class** gira su un processore ARM come su uno x86, purché ci sia una JVM per quella piattaforma. Il C, invece, si compila nel codice macchina di un processore preciso.



Dal sorgente al programma in esecuzione: la JVM carica il bytecode del programma e quello delle librerie che usa.

Definizione – CLI, GUI e IDE

- La **CLI** (*Command Line Interface*) è l'interfaccia a riga di comando del sistema operativo: il terminale, o prompt dei comandi.
- La **GUI** (*Graphical User Interface*) è l'interfaccia grafica, a finestre.
- Un **IDE** (*Integrated Development Environment*), come Eclipse o NetBeans, riunisce scrittura, compilazione ed esecuzione: con un pulsante fa gli stessi passi che da riga di comando.

Saper compilare ed eseguire dalla CLI serve anche a chi usa un IDE. Molti dispositivi non hanno un ambiente a finestre, che costa memoria e potenza di calcolo: una telecamera di videosorveglianza, un sensore domotico, un server nel cloud. Vi si accede da remoto con SSH, e si lavora solo dal terminale.

4 Errori

Definizione – Tipi di errore

Errore	Quando si scopre	Esempi
di compilazione: sintassi e semantica statica	lo segnala il compilatore, che non produce il file <code>.class</code>	<code>;</code> mancante, nome di classe o di metodo inesistente
di esecuzione: semantica dinamica	durante l'esecuzione; il programma termina in modo anomalo	divisione intera per zero, indice fuori dai limiti di un array
logico	il programma termina, ma i risultati non sono quelli voluti	un testo stampato sbagliato

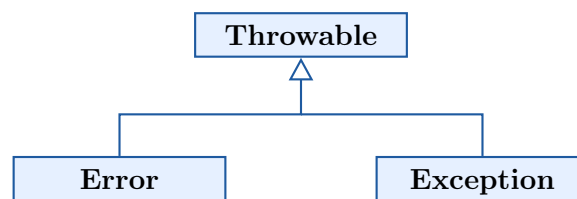
```
public class Divisione {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 0;  
        System.out.println(a / b);  
    }  
}
```

Il programma compila, ma durante l'esecuzione si ferma con *Exception in thread "main" java.lang.ArithmeticException: / by zero*.

In Java un errore di esecuzione è un **oggetto**, di una classe che il sistema definisce già: la JVM lo crea quando si verifica il problema, e il programma lo riceve. Queste classi, dette **eccezioni**, formano una gerarchia:

- **Error** per i problemi gravi, che il programma di solito non può gestire, come la memoria esaurita;
- **Exception** per quelli che il programma può gestire, come la divisione per zero (*ArithmeticException*).

Error ed Exception sono due sottoclassi distinte di *Throwable*.



La radice della gerarchia delle eccezioni: *Error* ed *Exception* sono sottoclassi di *Throwable*.

Classificare e correggere gli errori di un programma è l'Esercizio 2.

5 Oggetti e invocazione di metodi

Definizione – Invocazione di un metodo

Si chiede un servizio a un oggetto con

`oggetto.nomeMetodo(parametri)`

- **oggetto** è l'**oggetto di invocazione**, o meglio un riferimento che lo denota;
- **nomeMetodo** è il metodo invocato;
- **parametri** sono i **parametri attuali**, i valori passati al metodo.

I **parametri formali** sono quelli dichiarati nell'intestazione del metodo, con tipo e nome; al momento dell'invocazione ricevono i parametri attuali. Un metodo con il parametro formale `int n` si invoca, per esempio, con il parametro attuale `5`.

A differenza di una funzione del C, un metodo non si invoca da solo, ma su un oggetto. Per scrivere un'invocazione ci si chiede quindi *chi* deve svolgere l'operazione: quell'oggetto è l'oggetto di invocazione.

```
System.out.println("Ciao!");
```

<code>System.out</code>	l'oggetto di invocazione: un oggetto della classe <code>PrintStream</code> che scrive sull'output della console
<code>println</code>	il metodo invocato: stampa e va a capo
<code>"Ciao!"</code>	il parametro attuale, una stringa

`System.out` esiste già quando il programma parte: la JVM lo crea all'avvio. Gli altri oggetti il programma li deve creare. Sullo stesso oggetto si possono invocare metodi diversi, ma solo quelli definiti dalla sua classe:

```
System.out.print("Ciao "); // stampa senza andare a capo
System.out.println("Ciao!"); // stampa e va a capo: "Ciao Ciao!"
System.out.pippo("Ciao!"); // errore di compilazione
```

L'ultima riga non compila: *cannot find symbol, symbol: method pippo(String), location: variable out of type PrintStream.*

6 La classe `String`

Definizione – La classe `String`

Gli oggetti della classe `String` sono **stringhe**, cioè sequenze di caratteri, spazi compresi. Un'espressione racchiusa fra doppi apici, come `"java"`, denota un oggetto `String`.

Gli apici singoli non delimitano stringhe: `'a'` è un singolo carattere, di tipo `char` (Sezione 8).

Intestazione	Che cosa restituisce
<code>String concat(String str)</code>	una nuova stringa: quella su cui si invoca il metodo seguita da <code>str</code>
<code>int length()</code>	la lunghezza della stringa
<code>String substring(int i)</code>	la sottostringa dall'indice <code>i</code> alla fine
<code>String substring(int i, int j)</code>	la sottostringa dall'indice <code>i</code> all'indice <code>j</code> , escluso

Proposizione – Indici e sottostringhe

In una stringa lunga n i caratteri hanno indici da 0 a $n - 1$. La sottostringa

`s.substring(i, j)`

contiene i caratteri di indice da i a $j - 1$: è lunga $j - i$.

<code>substring(1, 4)</code>							
S	o	f	t	w	a	r	e
0	1	2	3	4	5	6	7
				<code>substring(4)</code>			

La stringa "Software", lunga 8, con gli indici da 0 a 7: `substring(1, 4)` è "oft",
`substring(4)` è "ware".

```
public class ProvaStringhe {
    public static void main(String[] args) {
        String s = "Soft";
        String t = s.concat("ware");
        System.out.println(t);           // Software
        System.out.println(s);           // Soft
        System.out.println(t.length());  // 8
        System.out.println(t.substring(4)); // ware
        System.out.println(t.substring(1, 4)); // oft
        System.out.println("ja" + "va"); // java
    }
}
```

`concat` non modifica la stringa su cui si invoca: `s` resta "Soft", e il risultato è una stringa nuova. Anche l'operatore `+` fra due stringhe le concatena.

Nota – Due errori tipici

Gli errori più comuni di chi comincia con Java sono due: dare al file un nome diverso da quello della classe (Sezione 2) e contare da 1 invece che da 0.

7 Argomenti da riga di comando

Definizione – Argomenti da riga di comando

Le parole scritte dopo il nome della classe nel comando `java` arrivano a `main` come stringhe nell'array `args`, nell'ordine: la prima in `args[0]`, la seconda in `args[1]`, e così via. `java` e il nome della classe non ne fanno parte.

Gli argomenti sono separati dagli spazi; fra virgolette, più parole diventano un solo argomento, che contiene gli spazi.

Comando	<code>args[0]</code>	<code>args[1]</code>	<code>args[2]</code>	<code>args[3]</code>
<code>java Pippo A B fai questo</code>	"A"	"B"	"fai"	"questo"
<code>java Pippo A B "fai questo"</code>	"A"	"B"	"fai questo"	–

```
public class Pippo {
    public static void main(String[] args) {
        System.out.println(args.length); // numero degli argomenti
        System.out.println(args[0]);     // il primo
    }
}
```

Con il primo comando stampa 4 e A, con il secondo 3 e A. Senza argomenti stampa 0 e poi si ferma con un errore di esecuzione, *ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0*: `args` è vuoto e `args[0]` non esiste.

8 Variabili e tipi

Definizione – Variabile

Una **variabile** denota un dato del programma. Ha:

- un **nome**, un identificatore valido come in C e in Python;
- un **tipo**;
- un **indirizzo** di memoria;
- un **valore**.

Definizione – Tipi di dato

Java è un linguaggio **fortemente tipizzato**: il tipo di ogni variabile e di ogni espressione è noto al momento della compilazione. I tipi sono di due categorie:

- i **tipi primitivi**: interi, reali, caratteri, booleani;
- i **riferimenti a oggetti**: il tipo è una classe, della libreria o scritta dal programmatore.

Categoria	Tipo	Bit	Valori
interi	byte, short, int, long	8, 16, 32, 64	numeri senza parte decimale
reali (in virgola mobile)	float, double	32, 64	numeri con parte decimale
caratteri	char	16	un carattere, come 'a'
booleani	boolean	–	true e false

Le dimensioni sono fissate dal linguaggio e sono le stesse su ogni JVM e su ogni processore: un **int** è sempre di 32 bit e arriva a poco più di due miliardi (**Integer.MAX_VALUE** = 2147483647). È un'altra faccia della portabilità; in C, invece, la dimensione dei tipi interi dipende dal processore. I nomi dei tipi primitivi sono minuscoli, a differenza di quelli delle classi: i loro valori non sono oggetti.

Gli altri tipi sono classi, come **String**: una variabile di tipo classe non contiene l'oggetto, ma un **riferimento** all'oggetto, cioè il suo indirizzo. Gli oggetti si creano e si allocano in memoria indipendentemente dalle variabili, e hanno un tempo di vita diverso (Sezione 9). Una variabile di tipo primitivo contiene invece direttamente il valore.

Dichiarazione e assegnazione.

	Sintassi	Esempi
dichiarazione	tipo nomeVariabile;	int a; String s;
assegnazione	nomeVariabile = espressione;	a = b + 5; s = "Ciao";

```
public class Variabili {
    public static void main(String[] args) {
        int a, b, c;
        a = 5;
        b = 2;
        c = a + b;
        System.out.println("Il risultato è " + c);    // Il risultato è 7
    }
}
```

Definizione – L-value e R-value

In un'assegnazione, come `c = a + b;`, una variabile si usa in due modi:

- delle variabili a destra dell'uguale si legge il valore: è il loro **R-value**;
- della variabile a sinistra conta l'indirizzo, cioè dove scrivere il risultato: è il suo **L-value**.

Con `a` che vale 5 e `b` che vale 2 si leggono gli R-value 5 e 2, se ne fa la somma e il 7 si scrive all'indirizzo di `c`. Da lì 7 è l'R-value di `c`.

In Python una variabile nasce con la prima assegnazione, `a = 2`, e il tipo lo prende dal valore. In Java va dichiarata prima dell'uso, e il tipo è fissato dalla dichiarazione: una variabile non dichiarata è un errore di compilazione.

Proposizione – Inizializzazione delle variabili locali

Una variabile locale, dichiarata in un metodo, va inizializzata prima di essere letta, qualunque sia il suo tipo; altrimenti il compilatore rifiuta il programma.

```
String s;  
System.out.println(s);    // errore di compilazione
```

Il compilatore segnala *variable s might not have been initialized*.

9 Stack, heap e garbage collection

Definizione – Stack e heap

La memoria di un programma Java in esecuzione comprende due aree:

- lo **stack** (pila) contiene i **record di attivazione** dei metodi in corso, con le loro variabili locali. Si gestisce a pila: il record di un metodo invocato va in cima e si toglie quando il metodo termina;
- lo **heap** contiene gli oggetti, che restano in vita finché servono, anche dopo la fine del metodo che li ha creati.

Il primo record, alla base della pila, è quello di `main`, che la JVM invoca per primo. Una variabile locale di tipo classe sta nel record del suo metodo e contiene il riferimento; l'oggetto a cui punta sta nello heap.

```

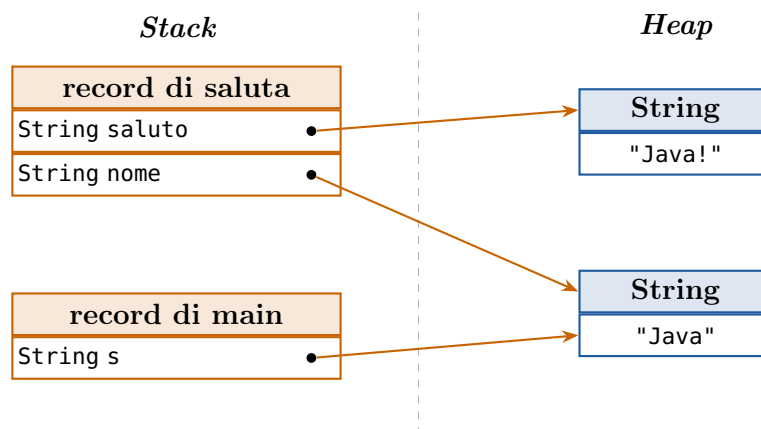
public class Pila {
    static void saluta(String nome) {
        String saluto = nome.concat("!");
        System.out.println(saluto);
    }

    public static void main(String[] args) {
        String s = "Java";
        saluta(s);
        System.out.println(s.length());
    }
}

```

Il programma stampa **Java!** e poi **4**. Nella memoria:

1. parte **main**: il suo record va alla base dello stack, con **s** che punta alla stringa "Java" nello heap;
2. **main** invoca **saluta(s)**: il record di **saluta** va in cima. Il parametro formale **nome** riceve il valore del parametro attuale **s**, cioè il riferimento alla stessa stringa; **saluto** punta alla nuova stringa "Java!";
3. **saluta** termina e il suo record si toglie: **nome** e **saluto** non esistono più.



*La memoria durante l'esecuzione di **saluta**: il suo record sta sopra quello di **main**, e **nome** e **s** puntano alla stessa stringa. Per semplicità non si disegnano **args** e la stringa `"!"`.*

Definizione – Garbage collection

La **garbage collection** è il recupero della memoria degli oggetti che il programma non può più raggiungere: nessuna variabile porta più a loro, né direttamente né attraverso altri oggetti. In Java la svolge automaticamente la JVM; non esiste un'istruzione per liberare un oggetto.

Nel programma **Pila**, quando **saluta** termina, nessun riferimento porta più a **"Java!"**: la JVM può recuperare la memoria. La stringa **"Java"** resta in vita, perché **s** la raggiunge.

ancora. Il lavoro del garbage collector durante l'esecuzione ha un costo in tempo di CPU e in memoria.

Osservazione – Java e C

In C la memoria si chiede con **malloc** e si restituisce con **free**. Un programma che dimentica **free** perde memoria (*memory leak*) e può esaurire lo heap: è un errore tipico dei dispositivi embedded programmati in C. In Java lo spazio per un oggetto lo trova la JVM quando l'oggetto nasce, e lo recupera il garbage collector: né **malloc** né **free**.

10 Riferimenti, new e null

Definizione – Creazione di un oggetto

L'operatore **new** crea un oggetto della classe indicata:

```
new NomeClasse(parametri)
```

alloca nello heap lo spazio per l'oggetto e restituisce il riferimento, che di solito si assegna a una variabile. I parametri servono a inizializzare l'oggetto.

```
String s = new String("ciao"); // una stringa con i caratteri di "ciao"
String v = new String();      // la stringa vuota: v.length() vale 0
```

Definizione – Il valore null

Una variabile di tipo classe può valere **null**: non denota nessun oggetto. **null** non è un indirizzo di memoria, ma un valore speciale che indica l'assenza del riferimento.

	<code>String s = null;</code>	<code>String s = "";</code>
oggetto nello heap	nessuno	una stringa di 0 caratteri
<code>s.length()</code>	errore di esecuzione, <code>NullPointerException</code>	0
<code>s == null</code>	true	false

La stringa vuota è un oggetto, che esiste ma non contiene caratteri; **null** vuol dire che l'oggetto non c'è. Invocare un metodo su **null** è un errore di esecuzione.

Proposizione – Assegnazione fra variabili

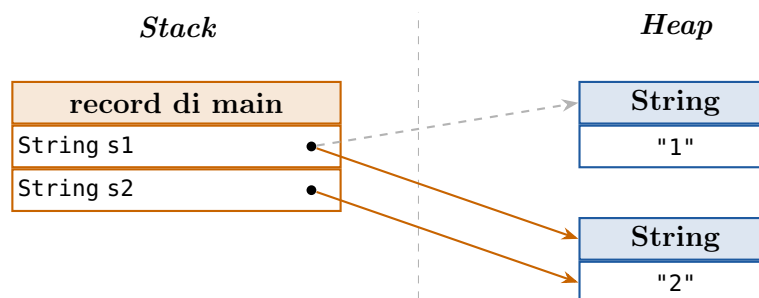
- Fra variabili di tipo classe, $t = s$ copia il riferimento: s e t denotano lo **stesso oggetto**.
- Fra variabili di tipo primitivo, $b = a$ copia il valore: le due variabili restano indipendenti.

Dopo `int a = 5; int b = a; a = 12;` la variabile `b` vale ancora 5. Dopo `String t = s;` invece c'è un solo oggetto, con due variabili che lo raggiungono.

Riassegnazione e garbage collection. Una variabile di tipo classe può essere riassegnata a un altro oggetto; quello che denotava prima può restare irraggiungibile.

```
String s1 = new String("1");
String s2 = new String("2");
s1 = s2;    // ora s1 e s2 denotano la stessa stringa "2"
```

Dopo l'assegnazione nessuna variabile raggiunge più la stringa "1": il garbage collector ne recupera la memoria. In Java non esiste un'istruzione per cancellare un oggetto (Sezione 9).



Dopo `s1 = s2`: `s1` non denota più "1" (freccia tratteggiata), che resta irraggiungibile ed è recuperato dal garbage collector.

Riepilogo

- **Programma:** classi con metodi; parte dal metodo statico `main`, che la JVM invoca senza oggetto; `args` contiene le parole dopo il nome della classe.
- **Esecuzione:** `javac Nome.java` produce il bytecode `Nome.class`, `java Nome` lo esegue sulla JVM.
- **Errori:** di compilazione, di esecuzione, logici; le eccezioni sono oggetti, `Error` ed `Exception` sottoclassi di `Throwable`.
- **Invocazione:** `oggetto.metodo(parametri)`, solo con i metodi della classe dell'oggetto.
- **String:** letterali fra doppi apici; indici da 0; `substring(i, j)` esclude `j`.
- **Tipi:** otto primitivi in quattro categorie, di dimensione fissa; le locali vanno inizializzate.
- **Memoria:** variabili locali nei record dello stack, oggetti creati con `new` nello heap; il garbage collector recupera quelli non raggiunti.
- **Riferimenti:** `t = s` fa denotare a `t` lo stesso oggetto di `s`; `null` non denota nessun oggetto.

11 Esercizi

Esercizio 1 – Nome e matricola

Scrivere un programma Java che stampi due righe di testo: nella prima il proprio nome e cognome, separati da uno spazio, nella seconda la matricola. Usare tre istruzioni di stampa diverse.

Soluzione a pagina 17 →

Esercizio 2 – I primi errori

Il programma seguente dovrebbe stampare le due righe `Questi sono i miei primi errori...` e `...e non saranno gli ultimi!!!`.

```
public class Errori {  
    public static void main(String[] args) {  
        System.out.println("Questi sono i miei primi errori...")  
        Sistem.out.println("...e non saranno gli ultm!!!");  
    }  
}
```

1. Individuare gli errori e classificarli.
2. Compilare il programma così com'è e verificare che cosa segnala il compilatore.
3. Correggere tutti gli errori.

Soluzione a pagina 17 →

12 Soluzioni

Soluzione dell'Esercizio 1 – Nome e matricola

Le istruzioni sono tre e le righe due: una delle stampe non deve andare a capo. È `print`, che scrive il nome e lo spazio; due `println` scrivono il cognome e la matricola, ciascuna chiudendo la sua riga.

```
public class Studente {
    public static void main(String[] args) {
        System.out.print("Mario ");
        System.out.println("Rossi");
        System.out.println("1234567");
    }
}
```

Il file si chiama `Studente.java`; con `javac Studente.java` e `java Studente` stampa Mario Rossi sulla prima riga e 1234567 sulla seconda.

[← Torna all'Esercizio 1, pagina 16](#)

Soluzione dell'Esercizio 2 – I primi errori

Riga	Errore	Tipo
3	manca il ; alla fine dell'istruzione	di compilazione, di sintassi
4	<code>Sistem</code> non esiste: la classe è <code>System</code>	di compilazione, di semantica statica
4	il testo dice <code>ultm</code> invece di <code>ultimi</code>	logico

Il compilatore segnala prima solo l'errore di sintassi, `';' expected` alla riga 3: finché il testo non è sintatticamente corretto, non passa ai controlli di semantica. Aggiunto il `;`, segnala `package Sistem does not exist` alla riga 4: `Sistem.out` non denota nessun oggetto. Corretto anche `Sistem`, il programma compila e gira, ma stampa `ultm`: l'errore logico il compilatore non lo vede.

```
public class Errori {
    public static void main(String[] args) {
        System.out.println("Questi sono i miei primi errori...");
        System.out.println("...e non saranno gli ultimi!!!");
    }
}
```

[← Torna all'Esercizio 2, pagina 16](#)