

Introduzione a Java

Caratteristiche generali del linguaggio.

Possiamo riassumere tali considerazioni nel seguente elenco:

- Semplicità
- Object-oriented
- Solidità
- Multi-threading
- Indipendenza dall'architettura
- Interpretazione con elevate prestazioni
- Possibilità di lavorare in modo distribuito
- Dinamicità

Indipendenza dall'architettura...

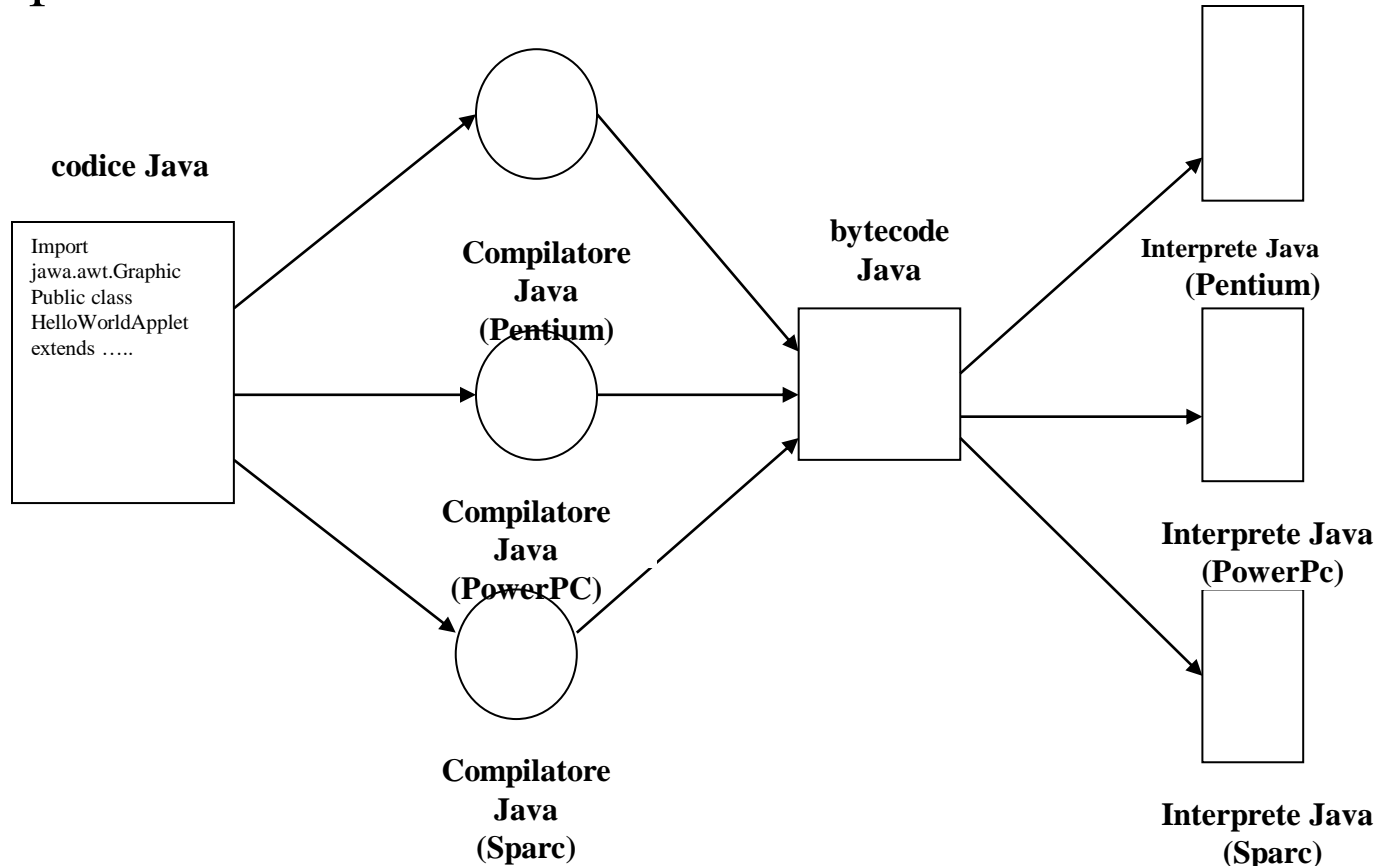
Un programma Java può essere eseguito su architetture diverse.

Esiste una indipendenza a livello di :

- *codice sorgente*: tipi di dati primitivi completamente specificati, ordine di valutazione delle espressioni rigorosamente definito.
- *bytecode* : un programma Java viene compilato in un codice intermedio detto **bytecode**. I file di codice intermedio sono interpretati dalla *Java Virtual Machine (JVM)*. È sufficiente implementare una JVM specifica per ogni piattaforma, senza per questo dover modificare il bytecode.

...Indipendenza dall'architettura.

L'implementazione della JVM non è a carico dell'utente.



La JVM è pertanto una emulazione software di un'architettura hardware ideale in grado di eseguire bytecode in Java.

Interpretazione con elevate prestazioni...

Il bytecode può essere interpretato su qualsiasi sistema che offre una Java Virtual Machine.

Altri sistemi interpretati come Basic e Perl presentano deficit di prestazioni insormontabili.

Pur avendo delle prestazioni inferiori ai linguaggi direttamente compilati, il bytecode di Java è stato progettato in modo tale da poter essere tradotto senza difficoltà in codice macchina nativo garantendo così delle prestazioni abbastanza elevate.

...Interpretazione con elevate prestazioni.

I sistemi run-time Java che eseguono questa ottimizzazione “*just in time*” o “*on-fly*” non perdono nessuno dei vantaggi offerti dal codice indipendente da piattaforma e allo stesso tempo hanno delle buone prestazioni.

Nota: la compilazione just-in-time è oggetto di ricerca. I compilatori convenzionali sono pensati per produrre codice altamente ottimizzato **senza prestare attenzione al tempo di compilazione.**

Ambienti di sviluppo: JDK

Esistono diversi ambienti di sviluppo per Java, il primo, denominato **Java Development Kit** (Jdk) e messo a disposizione dalla Sun, è il più diffuso.

Ogni altro ambiente di sviluppo per Java aggiunge funzionalità, nuovi *tool*, nuove classi, pur avendo come piattaforma base un Java Development Kit.

Ambienti di sviluppo: JDK

- tecnologia **Java Archive** (Jar) , un formato file indipendente dalla piattaforma che aggrega più file in un singolo package.

Ambienti di sviluppo: JDK

Sun struttura la piattaforma Java in due edizioni:

- Java Runtime Environment (JRE)
- Java Software Development Kit (Java SDK)

La prima è rivolta a utenti di applicazioni Java, la seconda è rivolta agli sviluppatori di applicazioni Java.

La JRE permette di eseguire applicazioni Java per cui non include il compilatore e altre componenti di sviluppo.

La Java SDK mette a disposizione degli sviluppatori il compilatore, le librerie, i sorgenti, le specifiche API e, in generale, tutto quanto è necessario per lo sviluppo di applicazioni Java.

Ambienti di sviluppo: JDK

L' edizione SDK si distingue in:

- Java 2 Standard Edition J2SE
- Java 2 Enterprise Edition J2EE
- Java 2 Micro Edition J2ME

La prima (J2SE) rappresenta la piattaforma Java standard e include tutti gli applicativi e le librerie per lo sviluppo di applicazioni desktop, applet e client-server.

JDK: tool di base...

Gli strumenti di base presenti nel JDK sono:

- *javac* compilatore Java
- *java* interprete Java

Riferimenti Bibliografici

Bruce Eckel

Thinking in Java

Prentice-Hall, 2000

Oggetti in java

Creazione di oggetti in Java...

Ogni linguaggio di programmazione ha i propri mezzi per manipolare i dati.

Il programmatore deve essere costantemente consapevole del tipo di manipolazione che si sta effettuando. In particolare, trattando gli oggetti siamo interessati a sapere se:

Si sta manipolando direttamente un oggetto, o si sta trattando qualche tipo di rappresentazione indiretta (un puntatore in C o C++) che dev'essere assoggettata a una qualche sintassi specifica.

In C++ sono possibili entrambi i casi.

...Creazione di oggetti in Java...

Tutto ciò è semplificato in Java. In Java **tutto viene trattato come oggetto** in modo tale da avere una sintassi consistente da usare in ogni situazione. Sebbene ciò sia sempre vero, l'identificatore che si va a manipolare è in realtà il ***riferimento*** ad un oggetto.

Cosa comporta ciò?

Quando si ha un riferimento, non è detto che ci sia necessariamente un oggetto ad esso legato.

...Creazione di oggetti in Java.

Esempio

Se abbiamo bisogno di una stringa allora si può creare un riferimento del tipo

```
String box;
```

Se a questo punto si vuole mandare un messaggio a `box` si otterrà un errore al run-time perché non abbiamo inizializzato la stringa

```
String box="abcd";
```

Una volta creato un riferimento, è possibile connetterlo ad un nuovo oggetto mediante l'operatore ***new***.

```
String box=new String("abcd");
```

che oltre a creare un nuovo oggetto di tipo `String`, indica anche “come” creare l'oggetto in questione.

Tipi primitivi: un caso particolare

In Java abbiamo una grande varietà di **tipi predefiniti** ai quali si aggiungono quelli che è possibile creare da sé. In Java i tipi **primitivi** godono di un trattamento particolare.

Ciò è dovuto al fatto che la creazione mediante l'operatore *new* non è molto efficiente, visto che viene allocata nella **heap**.

Per i tipi primitivi, Java si affida all'approccio C/C++ in cui viene creata una variabile (contenente il valore assegnato) memorizzata nello **stack**, anziché un riferimento ad essa come avviene con l'operatore *new*.

Tipi primitivi: dove vivono gli oggetti (Stack e Heap)...

Lo **stack** si serve di uno stack pointer per allocare e rilasciare memoria. Ciò ne fa un modo estremamente **veloce ed efficiente** per allocare memoria secondo solo ai registri. **Il compilatore deve sapere**, mentre alloca il programma, **l'esatta dimensione e tempo di vita di tutti i dati contenuti nello stack** in modo da generare il codice per gestire lo stack pointer.

Questo vincolo limita la **flessibilità** dei programmi, così accade che mentre i riferimenti agli oggetti sono nello stack, gli oggetti non possono esservi memorizzati.

...Tipi primitivi: dove vivono gli oggetti (Stack e Heap).

La **heap** è una memoria generale dove vivono gli oggetti java. Quando si crea un oggetto con l'operatore *new*, viene allocata memoria nella heap (a *run time*). Non si ha il vincolo posto sullo stack, ma il prezzo da pagare per la maggiore flessibilità è la quantità di tempo necessaria per l'allocazione.

Tipi primitivi in java...

Il progetto del linguaggio Java definisce la dimensione di ogni tipo primitivo e tale **dimensione** non varia da un tipo di piattaforma ad un'altra (come avviene negli altri linguaggi) garantendo in tal modo la portabilità.

... Tipi primitivi in java...

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

...Tipi primitivi in java

- Tutti i tipi numerici sono con segno (*signed*). I tipi unsigned del C/C++ non sono supportati.
- La dimensione del tipo *boolean* non è esplicitamente definita. È specificato che deve solo contenere valori letterali *true* o *false*.
- I tipi di dati primitivi possono avere una classe ‘*wrapper*’ associata per poter avere oggetti corrispondenti allocati nella heap.

Numeri ad alta precisione

Java include due classi per l'aritmetica ad alta precisione:

- *BigInteger*: supporta gli interi a '*arbitrary-precision*' ovvero è possibile rappresentare accuratamente valori **integrali** di qualsiasi dimensione, senza perdita di informazione durante le operazioni;
- *BigDecimal*: supporta numeri **decimali** in notazione *fixed-point* a '*arbitrary-precision*'. Possono essere utilizzati per accurati calcoli monetari.

Entrambe possiedono dei metodi analoghi agli operatori disponibili per i tipi primitivi **int** e **float**.

Variabili in Java

In Java esistono tre tipi di variabili:

- *variabili di istanza*: definiscono gli attributi e lo stato di un oggetto;
- *variabili di classe*: definiscono gli attributi e lo stato di una classe (si dichiarano con *static*)
- *variabili locali*: utilizzate nella definizione dei metodi (contatori dei cicli etc.)

Dichiarazione di variabili

DICHIARAZIONE = TIPO + NOME

NOME:

- può iniziare con una lettera, trattino, segno di sottolineatura, \$;
- non può cominciare con una cifra;

TIPO:

- uno tra gli otto tipi primitivi;
- nome di una classe o interfaccia;
- un array;



All'atto di una dichiarazione è anche possibile provvedere alla inizializzazione della variabile.²⁵

Creazione di nuovi tipi di dato: la classe...

Se in Java tutto può essere considerato un oggetto, come rappresentare il fatto che un insieme di oggetti sono accomunati dalle medesime caratteristiche e comportamenti?

La soluzione è quella di utilizzare il concetto di *classe* della programmazione object-oriented.

In Java per definire una classe è usata la parola chiave `class` seguita dal nome della classe.

...Creazione di nuovi tipi di dato: la classe ...

Esempio

```
class Box {  
    double width;  
    double height;  
    double depth;  
}
```

N.B.: una dichiarazione `class` non crea l'oggetto ma introduce un nuovo tipo, un modello. Per avere l'oggetto dobbiamo usare l'operatore *new*

```
Box a = new Box();
```

Dove '*a*' è un riferimento ad un oggetto della classe *Box*
ovvero '*a*' è una istanza della classe *Box*

Metodi e attributi di una classe

Quando si definisce una classe è possibile specificare

- *Componente dati* (data member) o *attributi*, per definire lo **stato** di un oggetto
- *Componente funzioni* (function member) o *metodi* per definire il **comportamento** di un oggetto

Componente dati di una classe...

Le *componenti dato* (**data member**) possono essere dei tipi primitivi oppure dei riferimenti ad altri oggetti.

Se la componente dato è un tipo *primitivo* allora può essere *inizializzata* direttamente nel punto di definizione all'interno della classe.

Nel caso in cui la componente dato *non* è un tipo *primitivo* deve essere necessariamente inizializzata attraverso un metodo apposito chiamato **costruttore**.

....Componente dati di una classe...

Esempio

```
class DataOnly {  
    int i;  
    float f;  
    boolean b;  
}  
...  
/* dot notation: uso dell'oggetto */  
DataOnly mydate = new DataOnly ()  
mydate.i = 47;  
mydate.f = 1.1f;  
mydate.b = false;  
...
```

La classe **DataOnly** non ha metodi e viene solo usata per mantenere i valori dei suoi attributi.

... Componente dati di una classe...

Quando un tipo di dato primitivo è membro di una classe, Java ci assicura la sua inizializzazione ad un valore di default nel caso non venga inizializzato dall'utente.

... Componente dati di una classe...

Primitive type	Default
boolean	false
char	'\u0000' (null)
byte	(byte) 0
short	(short) 0
int	0
long	0L
float	0.0f
double	0.0d

... Componente dati di una classe...



Se la variabile è **locale**, Java non la inizializza ad un valore di default pertanto la mancata inizializzazione verrà segnalata come errore al momento della compilazione.

Il comportamento dell'oggetto: Metodi di una classe...

I metodi in Java sono dei messaggi che permettono agli oggetti di comunicare fra loro.

Un metodo in Java deve essere necessariamente definito in una classe (**non esistono metodi *stand-alone***).

Lo schema per la definizione di un metodo Java è il seguente:

```
tipo_risultato NomeMetodo(/*lista di argomenti */) {  
/* corpo del metodo */  
return espressione  
}
```

...Il comportamento dell'oggetto: Metodi di una classe...

dove:

tipo_risultato: indica il tipo primitivo o la classe dell'oggetto restituito dal metodo;

lista di argomenti: lista dei tipi e dei nomi passati al metodo. I parametri di tipo oggetto sono passati per riferimento mentre quelli di tipo primitivo sono passati per valore

return: è una parola chiave utilizzata per la restituzione del valore elaborato. Se il tipo da restituire è void allora return ha solo la funzione di uscire dal metodo.

...Il comportamento dell'oggetto: Metodi di una classe

Esempio

```
class Prova
{
boolean flag() { return true; }
float logBaseNaturale() { return 2.718f; }
void non_fa_nulla() { return; }
void non_fa_nulla2() {}
}

...

boolean flag=false;
Prova objProva= new Prova();
/*flag=false*/
flag=objProva.flag();
/*flag=true*/
```

...Il comportamento dell'oggetto: Metodi di una classe

Un metodo è identificato univocamente dal suo nome e dalla lista di argomenti.

Un metodo può essere chiamato solo per un oggetto che deve essere a sua volta capace di eseguirne la chiamata.

Quando il tipo di ritorno è void, la parola chiave `return` è usata solo per uscire dal metodo, tuttavia **non è necessaria nel caso in cui si esca solo dalla fine del metodo.**

Costruttori...

Prima di poter essere usati, gli oggetti devono essere creati ovvero *inizializzati*.

Ciò è fatto attraverso un particolare metodo della classe chiamato *costruttore*.

Java infatti richiama automaticamente (con l'operatore *new*) il rispettivo costruttore non appena l'oggetto è stato creato e prima che tale oggetto sia a disposizione dell'utente.

Analogamente al C++, anche in Java *il costruttore assume il nome della classe a cui appartiene*.

Se abbiamo una classe priva di costruttore, il compilatore Java ne crea uno in maniera automatica.

...Costruttori...

Esempio (costruttore)

```
class Rock {  
    Rock() { // This is the constructor  
        System.out.println("Creating Rock");  
    }  
}  
  
public class SimpleConstructor {  
    public static void main(String[] args) {  
        for(int i = 0; i < 10; i++)  
            new Rock();  
    }  
}
```

Quando si crea l'oggetto Rock, viene allocata memoria e richiamato il rispettivo costruttore. L'effetto è quello di stampare su console la stringa “creating Rock”.

...Costruttori

Un costruttore può avere anche degli argomenti

Esempio (costruttore con argomenti)

```
class Rock2 {  
    Rock2(int i) {  
        System.out.println("Creating Rock number " + i);  
    }  
}  
  
public class SimpleConstructor2 {  
    public static void main(String[] args) {  
        for(int i = 0; i < 10; i++)  
            new Rock2(i);  
    }  
}
```

! Il costruttore **non** restituisce alcun valore.

Distruttori...

A differenza del C++, in java non si riscontra il concetto di *distruttore* di oggetti, per la presenza di un metodo ordinario per ripulire la memoria (*garbage collector*).

Overloading di metodi...

L'overloading dei metodi è essenziale per permettere a metodi che hanno lo stesso nome di essere usati con differenti tipi di argomento in modo tale da supportare il polimorfismo.

È indispensabile usare l'overloading nella definizione dei costruttori in quanto il nome è protetto.

Sebbene tale funzionalità sia obbligatoria per i costruttori, generalmente conviene utilizzarla anche nella definizione dei metodi comuni.

Esempio.

...Overloading di metodi...

// Demonstration of both constructor and ordinary method overloading.

```
import java.util.*;
```

```
class Tree {
```

```
int height;
```

```
Tree() {
```

```
prt("Planting a seedling");
```

```
height = 0;
```

```
}
```

```
Tree(int i) {
```

```
prt("Creating new Tree that is "+ i + "feet tall");
```

```
height = i;
```

```
}
```

...Overloading di metodi...

```
void info() {  
    prt("Tree is " + height+ " feet tall");  
}  
  
void info(String s) {  
    prt(s + ": Tree is "+ height + " feet tall");  
}  
  
static void prt(String s) {  
    System.out.println(s);  
}  
}
```

...Overloading di metodi...

```
public class Overloading {  
    public static void main(String[] args) {  
        for(int i = 0; i < 5; i++) {  
            Tree t = new Tree(i);  
            t.info();  
            t.info("overloaded method");  
        }  
        // Overloaded constructor:  
        new Tree();  
    }  
}
```

Il significato di *this*...

La parola chiave *this* è usata all'interno di un metodo per mantenere il riferimento all'oggetto in uso corrente.

Esempio

```
{ return this; }
```

...Il significato di *this*...

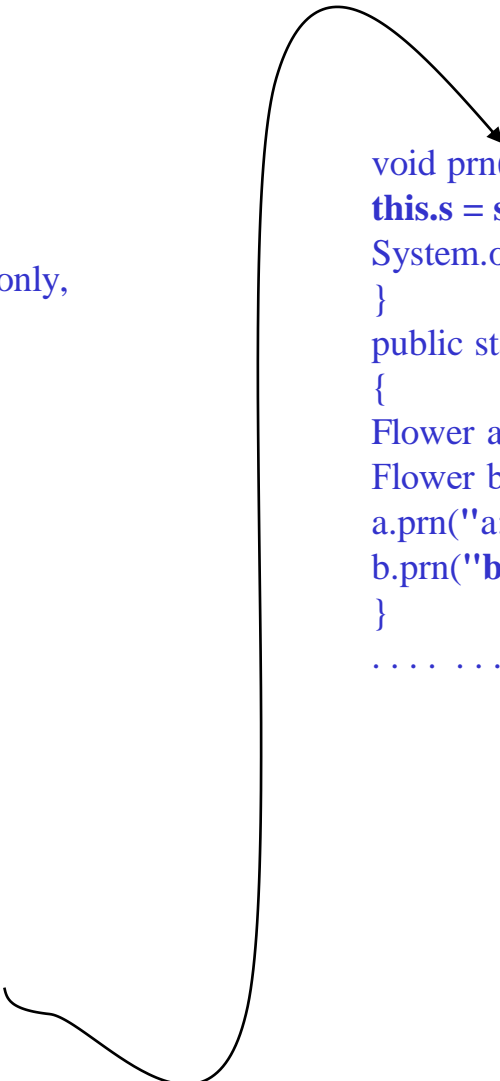
La parola chiave *this* è anche utilizzata per risolvere i conflitti di nome che potrebbero verificarsi tra variabili di istanza e variabili locali.

Se *this* si trova in un costruttore ed ha una lista di argomenti in input, effettua una chiamata esplicita al costruttore che corrisponde a quella data lista di parametri.

Esempi

...Il significato di *this*.

```
public class Flower {  
    int petalCount = 0;  
    String s = new String("null");  
    Flower(int petals) {  
        petalCount = petals;  
        System.out.println("Constructor w/ int arg only,  
            petalCount= "  
        + petalCount);  
    }  
    ....  
    Flower(String s, int petals) {  
        this(petals);  
        //! this(s); // Can't call two!  
        this.s = s; // Another use of "this"  
        System.out.println("String & int args");  
    }  
    Flower() {  
        this("hi", 47);  
        System.out.println(  
            "default constructor (no args)");  
    }  
}
```



```
void prn(String s, int petals) {  
    this.s = s;  
    System.out.println(s+petals);  
}  
public static void main(String args[])  
{  
    Flower a = new Flower(2);  
    Flower b = new Flower("seconda",2);  
    a.prn("a:",2);  
    b.prn("b:",3);  
}  
.....
```