

# IL LIBRO DI

# CorsoJava.it

## Versione 2.7

*di Michele Sciabarrà*  
[michele@sciabarra.com](mailto:michele@sciabarra.com)

Sponsorizzato da  
**www.ePrometeus.com**  
*Interattiviamo la tua azienda*



<i>Data</i>	<i>Autore</i>	<i>Versione</i>
Maggio - 2003	Michele Sciabarrà	V2.7

## ● Cos'è Java

- Un linguaggio
- Una tecnologia
- Una piattaforma

In questo corso impareremo a programmare in linguaggio Java. Java è considerato da molti semplicemente un linguaggio di programmazione. In realtà si può dire più correttamente che si tratta di una tecnologia, ovvero un insieme di soluzioni software legate all'impostazione fondamentale del linguaggio di programmazione Java. Questo sembra un discorso abbastanza vago, ma per essere capito più a fondo occorre esaminare le caratteristiche delle proprie del linguaggio Java, e in particolare il suo supporto a tempo di esecuzione. Sono proprio le caratteristiche della cosiddetta Java Virtual Machine che fanno di Java una tecnologia.

Ma c'è di più: infatti considerando che grazie a Java e alla sua JVM si riesce a scrivere dei programmi abbastanza complessi senza dipendenze da sistema operativo sottostante, Java è stato da tempo promossa a piattaforma, ovvero una sorta di sistema operativo parallelo e alternativo, ed è proprio lì la sua più importante valenza. C'è da dire comunque che Java non ha sostituito Windows, ma è diventata la importante piattaforma per lo sviluppo di applicazioni Web. In questo corso tuttavia ci occuperemo fondamentalmente del linguaggio di programmazione Java, e basta.

# 1 Introduzione

---

## ● Java come linguaggio di programmazione

### ● Assomiglia al C++

- Derivato da esso per semplificazione

Da un punto di vista strettamente sintattico, Java è un linguaggio di programmazione che assomiglia molto ad un linguaggio di programmazione già esistente. Ovvero il linguaggio C++. Questo somiglianza si vede in molti aspetti. Un programmatore C++ è in grado di programmare in Java, quasi da subito, una volta familiarizzare con le poche differenze, e soprattutto con le numerose mancanze rispetto al C++. Infatti Java è derivato dal C++ operando una serie di semplificazioni ed eliminando alcune caratteristiche che venivano ritenute dai progettisti del linguaggio Java poco usate e scarsamente utili.

La verità è che Java è fondamentalmente diverso dal C++ perché è un linguaggio dinamico, mentre C++ è fondamentalmente un linguaggio statico. Non entro i dettagli di cosa significhi ciò: in pratica mentre l'obiettivo del C++ è quello di compilare un programma nativo che venga eseguito il più rapidamente possibile, l'obiettivo di Java è quello di creare tanti componenti che possono facilmente collegarsi tra di loro anche mentre è in esecuzione un programma.

### ● Riprende da molti altri linguaggi

- In particolare Lisp
- Librerie a-la Visual Basic

In realtà Java ben difficilmente si può dire che figlio solo del C++. Infatti riprende da molti altri linguaggi. Si potrebbe fare un lungo elenco, ma uno dei linguaggi da cui deriva più direttamente è probabilmente il linguaggio Lisp. Infatti per scherzare ogni tanto dico che Java è il Lisp travestito da C++. C'è anche da dire che Java, a differenza del C++, ha una libreria molto completa, che permette di fare programmi con interfaccia grafica, accedere a database, collegarsi in rete, e fare molte altre cose. Questo è tipico di linguaggi come Visual Basic oppure Delphi.

## 1.1 Tecnologia

### ● La tecnologia Java

#### ● Programmi compilati in bytecode

Vediamo ora qualche dettaglio in più riguardo a Java come tecnologia. Un programma in Java per essere eseguito deve venire compilato in formato intermedio. Questo formato intermedio si chiama bytecode, e visto da vicino assomiglia ad una specie di codice macchina di un microprocessore.

#### ● Java Virtual Machine

##### ● incorporata nei browser

Solo che questo microprocessore, non è realizzato in silicio, ma in software. Viene pertanto chiamato Java Virtual Machine, o JVM. In pratica si tratta del programma che permette l'esecuzione dei programmi in Java, ovvero è l'interprete. Per intenderci ricordiamo che alcuni programmi in Visual Basic per funzionare richiedono di un componente chiamato VBRUN.DLL. Si tratta appunto di interprete dei programmi in Visual Basic.

#### ● Lingua franca?

- È di sistema su Solaris, AS/400, MacOS X
- Una versione vecchia pure su Windows
- Versione ufficiale per Linux

La caratteristica più interessante di Java è legata al fatto che questo interprete, che permette l'esecuzione dei programmi in Java, esiste per molti sistemi operativi. Innanzitutto esiste per Windows, ma anche per linux, oltre che Solaris (il suo sistema operativo di origine) e per molti altri sistemi quali il Mac, l'AS/400 ecc. Java ha la potenzialità di diventare una specie di lingua franca, e a tutto oggi è il linguaggio di programmazione più portabile.

#### ● Usata oggi in Web Server, Database Server

La più grande valenza tuttavia si ha ultimamente come linguaggio di estensione di sistemi server. Per esempio è stato utilizzato in origine per realizzare le cosiddette applet. In pratica il navigatore per Internet può scaricare dalla rete un programma in Java, ed eseguirlo all'interno di una pagina. Oggi queste applet non sono più molto in uso perché per via delle difficoltà di compatibilità con le varie versioni di Java che presentano. Tuttavia Java è molto usato per estendere per esempio server Web o server di database.

## 1.2 Piattaforma

### ● Obiettivo: l'autarchia

- Complesse applicazioni
  - Grossi programmi usando solo Java e libreria

A voler considerare Java una piattaforma, occorre dotarla di tutto ciò che caratterizza un sistema operativo. Nelle prime versioni questo non era possibile, in quanto la libreria era abbastanza piccolina, e molte cose non si potevano fare. La prima versione di Java non poteva nemmeno accedere ai database.

### ● Complessa libreria

- La libreria è mastodontica
  - ma è la punta dell'iceberg delle terze parti

L'attuale libreria di Java è molto ampia e completa. Comprende funzioni che permettono di fare complessi programmi senza appoggiarsi in alcun modo al sistema operativo sottostante. Praticamente, avere un interprete Java installato come avere un piccolo sistema operativo, che fornisce tutte le funzioni necessarie per eseguire delle applicazioni abbastanza complesse. Ma il punto cruciale, che rende Java molto interessante, e molto usato per programmare, è legato al fatto che gode di un buon supporto delle terze parti. Per cui esistono numerosissime librerie scritte in Java che completano la già ampia libreria standard. Per questo motivo Java è uno dei linguaggi usati attualmente, specialmente per scrivere applicazioni Web.

## 1.3 JDK

---

### ● Come programmare in Java

Dopo aver descritto le caratteristiche che motivano lo sviluppo in Java, è il momento di immettersi all'opera e di cominciare scrivere programmi. In questa sezione vedremo proprio come fare.

#### ● Installare il JDK

Per programmare occorre procurarsi il Java Development Kit, o JDK. Si tratta di una serie di programmi che permettono di programmare in Java. Il JDK è disponibile ufficialmente per Windows, Linux e Solaris. Almeno queste sono le 3 piattaforme ufficialmente supportate da Sun, che è la ditta che ha inventato il linguaggio e che ne prosegue lo sviluppo.

#### ● Scrivere "Hello"

Una volta che abbiamo il JDK, siamo pronti a scrivere il nostro primo programma. Vedremo che il JDK non fornisce strumenti visuali, ma solamente programmi a riga di comando (che si usano dalla finestra del DOS per gli utenti Windows), e per scrivere programmi dobbiamo utilizzare il blocco note.

#### ● Mandarlo in esecuzione

Una volta scritto un programma, dobbiamo saperlo mandare in esecuzione. Vedremo che questo comporta due passi distinti, il primo è la compilazione, e il secondo è l'effettivo lancio del programma. Fatto questo siamo pronti per imparare a fondo linguaggio Java.

## 1.4 Installazione

### ● Scaricare il JDK

- <http://java.sun.com>

Per avere il JDK, occorre un collegamento di Internet, e andare sul sito java.sun.com. Navigando questo sito è abbastanza facile trovare il JDK. I collegamenti si trovano già nella pagina principale. Bisogna stare attenti che è necessario andare direttamente sul sito della Sun, perché il JDK non è distribuibile da terze parti. Fare attenzione, che anche se kit di sviluppo non è ridistribuibile, il runtime (l'interprete) invece lo è. Per cui potete scrivere programmi in Java e darli ai vostri clienti senza dover pagare nulla.

### ● Installare

- Lanciare il setup o installare l'rpm
  - o dearchiviare in una directory

Sul sito della Sun troverete varie versioni del JDK, e per di più per varie piattaforma. E consigliato scegliere la più recente, ma evitare le versioni beta. Ovviamente dovete prendere quella che si adatta alla vostra piattaforma. Se sviluppatate su Windows potrete scaricare un programma che installa, mentre per linux potete trovare sia l'archivio rpm (tipico dei sistemi compatibili con RedHat Linux), che un archivio da decomprimere.

## 1.5 PATH

### ● Impostare il PATH

In sostanza installazione del JDK è abbastanza semplice: basta decomprimere il kit in una directory, dopo di che non dovrebbe servire altro. In realtà è opportuno impostare il PATH. Questo perché altrimenti si hanno difficoltà nel lanciare comandi che servono per fare nostro lavoro.

#### ● Windows

```
PATH %PATH%;c:\jdk1.3\bin
```

In ambiente Windows occorre semplicemente modificare il file c:\autoexec.bat e inserire la riga mostrata nel listato. Ovviamente se usato una versione diversa nome della directory potrebbe essere diverso. Inoltre fate attenzione che nel PATH deve essere inserita la sottodirectory bin, dove sono tutti comandi.

#### ● Linux

```
export PATH=$PATH:/opt/jdk1.3/bin
```

In ambiente linux, la procedura è identica, cambia il file dove dovete inserire la modifica (che è .profile nella vostra home directory), e la sintassi che è quella mostrata nel listato. Valgono le stesse raccomandazioni di prima per quanto riguarda nome della directory e da sottodirectory da inserire nel PATH.

#### ● Test:

##### ● Eseguire java

Per verificare che si può lavorare, la prima cosa da fare è quella di lanciare il comando java e vedere che cosa succede. Questo comando serve a eseguire programmi in Java. Da solo non fa nulla, o meglio visualizza una pagina di informazione, che è quello che vogliamo. Se compare la pagina di informazione vuol dire che il comando java è stato messo correttamente nel PATH, e quindi possiamo continuare a lavorare. Se invece otteniamo messaggio di errore, vuol dire che dobbiamo rivedere di passi precedenti.

## ● Altre variabili

### ● JAVA\_HOME

In generale non è obbligatorio impostare altre variabili, ma è prassi comune impostare sia nell'AUTOEXEC.BAT che nel .profile, usando una sintassi analoga quella rivista prima, la variabile JAVA\_HOME, che è spesso usata dai programmi per trovare dove è stato installato il kit Java. Fare attenzione che JAVA\_HOME deve puntare alla directory principale di dove è stato installato il kit, e non alla sottodirectory bin.

### ● usato da Tomcat, per esempio

Alcuni programmi necessitano che questa variabile di ambiente sia impostata. Per esempio TomCat, che è un motore per esecuzione di servlet e JSP, richiede per partire che la variabile JAVA\_HOME sia impostata correttamente. E come lui, tanti altri. In realtà solitamente non è Java che usa la variabile JAVA\_HOME, ma gli script (onnipresenti) che lanciano i programmi in Java.

### ● CLASSPATH

#### ● meglio di no...

In teoria è possibile anche impostare la variabile CLASSPATH, ma in pratica è meglio di no. Questo perché la variabile CLASSPATH imposta le librerie da utilizzare per tutti i Java presenti in un sistema. Purtroppo in pratica che si ritrova ad avere più di un Java installato (se si ha Netscape, questo contiene un interprete Java, che viene influenzato dalla variabile CLASSPATH, quando invece voi volete modificare il CLASSPATH solo per il JDK che state e utilizzando. Morale della favola: è meglio impostare volta per volta il CLASSPATH per ogni applicazione, magari scrivendo degli script ad hoc per lanciarla.

## 1.6 Hello

### ● Hello.java

#### ● Digitare il seguente listato

```
public class Hello {  
    public static  
    void main(String[] args) {  
        System.out.println("Hello, world.");  
    }  
}
```

#### ● C'è molto da capire in queste poche righe

Nel listato precedente c'è il nostro primo programma: apriamo il blocco note, oppure un editore testi Linux, descriviamo quello che c'è scritto nel listato. Non è ancora il momento di spiegare che cosa vuol dire, se non che serve a scrivere il messaggio "Hello world". Ma per capirlo approfonditamente, occorre conoscere i numerosi concetti, come la classe, il metodo, che vuol dire `public` e `static`, cose che diventeranno chiare solo nel corso del corso.

## 1.7 Cosa fare

### ● Interprete o compilatore

#### ● Entrambi!

Tradizionalmente i linguaggi di programmazione venivano suddivisi in linguaggi interpretati e compilati. Si considerava linguaggi interpretato un linguaggio, come il basic, che viene eseguito direttamente così com'è. Mentre si considerava un linguaggio compilato, come il pascal, un linguaggio che prima dell'esecuzione dovesse venir tradotto in un programma del linguaggio della macchina direttamente eseguibile. Java sfugge a questa classificazione. In realtà Java è, strettamente parlando, un linguaggio interpretato: infatti l'esecuzione dei programmi avviene grazie a un interprete, la JVM. Tuttavia, a differenza della maggioranza degli altri interpreti, Java prima dell'esecuzione richiede una fase di compilazione.

### ● Prima compilare

#### ● Formato intermedio bytecode

Per cui quando andiamo a eseguire il nostro programma, dobbiamo invocare un comando, javac, che elaborerà il testo del nostro programma in Java, e produrrà un formato intermedio eseguibile dalla JVM.

### ● Poi interpretare

#### ● RunTime per molte piattaforma

A questo punto che entra in gioco la JVM, che prende in pasto il bytecode, e lo esegue.

## 1.8 Compilazione

### ● Andare alla riga di comando

Purtroppo tutti i nostri programmi dovranno essere compilati dalla riga di comando, ovvero la finestra DOS dei sistemi Windows.

#### ● Il JDK non passa un IDE

Questo perché il JDK non comprende un ambiente visuale per lo sviluppo in Java. Questa carenza col tempo si è affievolita. Esistono infatti numerosi programmi che coprono questa lacuna. Uno di questi, gratuito è il NetBeans, mentre esistono numerosi altri ambiente di sviluppo, come JBuilder o VisualAge for Java che sono pagamento (ne esistono comunque delle versioni d'ingresso gratuite). A

```
javac Hello.java
```

#### ● Se tutto è andato OK, avremo

```
Hello.java Hello.class
```

torniamo alla nostra riga di comando, che per fortuna è sufficiente potente per fare molto.

In particolare tutto questo corso non faremo più riferimento a questi strumenti, ma semplicemente spiegheremo il linguaggio Java, lasciando al lettore compito di scriverli, di compilare e di eseguire i programmi. Nel nostro caso, una volta lanciato il comando javac Hello.java, guardando nella directory corrente, troveremo un file in più. È quello con estensione .class.

## 1.9 Esecuzione

### ● Riga di comando:

- Eseguire `java Hello`
- Risultato: `Hello , world`
- Molte cose possono andar storte

Per eseguire il programma, dobbiamo solamente lanciarlo. Questo si fa utilizzando il comando `java` seguito dal nome della classe, senza l'estensione finale `.class`. Il risultato dovremmo vedere è la stampa del messaggio contenuto nel programma. Se non lo vediamo dobbiamo ripercorrere tutti passi precedenti, e assicurarci che sia nel PATH il comando.

## 2 Sintassi

---

### ● Panoramica della sintassi.

Iniziamo questo corso facendo una panoramica generale della sintassi di Java. Vedremo in particolare come costruire dei programmi, facendo un esame della sintassi: dal basso verso l'alto. Esamineremo la sintassi di Java partendo dai gli elementi ultimi che lo costituiscono, per poi via salire verso l'alto, fino ad arrivare a costruire interi programmi.

#### ● Un aspetto semantico: la distinzione tra tipi primitivi e oggetti.

In questa sezione che occuperemo anche di un importante aspetto semantico, che è la principale difficoltà di comprensione (delle caratteristiche specifiche di Java) da chi viene da altri linguaggi di programmazione. Vedremo cioè la distinzione fondamentale tra quelli che sono in tipi primitivi e quelli che sono in tipi derivati, che chiameremo genericamente oggetti, o meglio classi di oggetti.

### ● Java: linguaggio imperativo orientato agli oggetti.

Java è un linguaggio cosiddetto imperativo. Imperativo vuol dire che è orientato al comando. In pratica un programma è composto da una sequenza di comandi che vengono eseguiti uno dopo l'altro. Questo per distinguerlo da altri linguaggi cosiddetti funzionali, che sono orientati al calcolo di valori. La maggior parte dei linguaggi di programmazione in uso oggi, linguaggi come Visual Basic o C, sono appunto di tipo imperativo. Iniziamo adesso il nostro esame della sintassi partendo dai gli elementi più piccoli, che sono le espressioni.

#### ● Espressioni gestite tramite comandi.

Una espressione è un calcolo di valori, che viene eseguita per decidere il risultato di un'operazione, oppure per determinare se una data condizione si è verificata oppure no. Le espressioni fanno parte della sintassi di Java ma non possono essere autonome. Per cui se voglio calcolare  $2 + 2$  posso farlo, ma devo inserire questo calcolo all'interno di un comando.

#### ● I comandi vengono incapsulati in metodi

I comandi permettono invece di eseguire azioni complete una dopo l'altra. In pratica i comandi sono gli elementi che servono per comporre programmi veri e propri, almeno in senso tradizionale. Infatti la programmazione in Java, essendo un linguaggio di ad oggetti, ha un feeling diverso da quello di molti altri linguaggi più tradizionali. Comunque in ultima analisi un programma è composto anche di comandi, che vengono eseguiti uno dopo l'altro. I comandi in Java non possono però stare da soli. Ci sono dei linguaggi che permettono di eseguire dei comandi direttamente. Per esempio il Basic consente di digitare un comando ed eseguirlo immediatamente. Questo in Java non è possibile: un comando deve essere sempre contenuto dentro un metodo. I metodi assomigliano molto alle funzioni o procedure, ma con delle differenze che vedremo nel corso del corso.

### ● I metodi vengono contenuti in classi

A loro volta nemmeno i metodi possono stare da soli. È necessario che vengano scritti all'interno di classi. Le classi sono la prima unità sintattica completa di un programma Java. Un programma in Java infatti è composto da tante classi. Una classe può essere un programma completo, che posso compilare e mandare in esecuzione.

### ● Le classi vengono raccolte in package.

Per un motivo di organizzazione, le varie classi vengono raccolte in pacchetti (package), che sono sostanzialmente una convenzione per dare un nome alle classi. Un pacchetto è insomma un prefisso che viene dato a un gruppo di classi. Lo vedremo in dettaglio.

## 2.1 Commenti

### ● I commenti nei programmi Java:

Iniziamo il corso vedendo come inserire all'interno di un programma Java del testo descrittivo che strettamente non è un programma.

```
/* commento */
```

- I caratteri compresi tra /\* e \*/ sono ignorati.

La forma più semplice di commento è quella che inizia con una barra seguita da un asterisco; poi segue il testo di commento, che infine viene terminato con un asterisco seguito da una barra. Questo tipo di commento si può estendere per varie righe di codice. Posso cominciare un commento in una riga e finirlo dieci riga più avanti.

```
// comment
```

- Tutti i caratteri successivi a // fino a fine linea sono ignorati.

Questo tipo di commento è però scomoda da scrivere per il fatto che si devono sempre inserire due caratteri alla fine. Molto spesso si vuole semplicemente inserire rapidamente un'annotazione ad una riga di codice. Per questo motivo esiste un tipo di commento che inizia semplicemente con due barre e viene terminato alla fine riga.

```
/** comment */
```

- Come /\* \*/, ma viene usato per la documentazione

- Il commento viene usato con il tool javadoc
- Creare automaticamente la documentazione dai sorgenti .

Un altro importante tipo di commento presente in Java, è simile a quello iniziale, ma inizia con due asterischi invece che di uno. In realtà si tratta semplicemente del primo tipo di commento, solo che quando ci sono due asterischi, un programma particolare chiamato javadoc è in grado di interpretare questi commenti, e trasformarli in pagine HTML, che servono a produrre della documentazione direttamente dai sorgenti del codice. La documentazione standard di Java è prodotta utilizzando questo strumento.

## 2.2 Espressioni

- Un programma in Java è innanzitutto composto da espressioni

a+1

Cominciamo adesso ad esaminare in dettaglio le caratteristiche sintattiche nei programmi Java. Abbiamo detto che l'elemento ultimo è l'espressione, come quella che stiamo vedendo.

- Una espressione si può suddividere in:

- *Costanti*: 1 è una costante
- *Operatori*: + è un operatore
- *Variabili*: a è una variabile

Esaminando in dettaglio l'espressione possiamo riconoscere che gli elementi sono ben noti: possiamo vedere un numero, uno, che è una costante. Possiamo vedere un simbolo di operazione, il più, che è un operatore. E possiamo riconoscere infine la lettera a che è una variabile. La sequenza a + 1 costituisce una espressione. Vedremo via via come gli elementi sintattici che stiamo elencando corrispondono a precise caratteristiche dei vari tipi di dati. In una espressione semplicemente operiamo su un tipo di dato. Notare che l'operazione più è un'operazione che va bene quando abbiamo a che fare con dei numeri, ma che non andrebbe bene se avessimo che fare per esempio con delle navi. Si, ho detto proprio delle navi: intendo dire che in Java possiamo avere dei tipi generici con le proprie operazioni. Per cui una nave può essere un tipo, che può navigare come un intero si può sommare.

## 2.3 Comandi

### ● I comandi contengono espressioni

Le espressioni sono sempre contenute nei comandi. Questo lo abbiamo detto dall'inizio. In realtà un comando viene costruito a partire da una espressione.

### ● I comandi possono essere

In particolare abbiamo due tipi di comandi: semplici e composti.

#### ● **Semplici:** una espressione seguita da un ; (punto e virgola)

```
a+1;
```

È sufficiente prendere una espressione, e metterci alla fine un punto e virgola. Questo basta a trasformare quest'espressione in un comando.

#### ● **Composti:** un comando contenente un altro comando

```
if (a>1)  
b=b+1;
```

I comandi composti hanno la caratteristica di essere ottenuti combinando più comandi. Cioè, questo vuol dire che si ha un comando che può essere composto da un sotto comando. Per esempio consideriamo il comando if: questo tipo di comando funziona eseguendo condizionalmente un altro comando. Quindi all'interno dell'if, che il comando vero e proprio, troviamo annidato un altro comando. Naturalmente il comando annidato dentro un comando composto può esser a sua volta un comando composto. Questo gioco di scatole cinesi può continuare per un certo livello di annidamenti, ma alla fine i comandi ultimi di questa gerarchia sono comandi semplici. In pratica i comandi semplici sono le foglie di un albero, i cui nodi interni sono i comandi composti.

## 2.4 Dichiarazioni

- Le *dichiarazioni* danno il tipo ad una variabile.

```
int i;
```

Le espressioni che abbiamo visto sono composte, tra altre cose, anche da variabili. In particolare le variabili sono come delle scatole in cui vengono appoggiati dei valori, e a cui si dà un nome. Java è un linguaggio di programmazione con un controllo dei tipi forte. Questo significa che le variabili possono contenere soltanto un certo tipo di dati. Quindi si fa distinzione tra un intero, una stringa o addirittura una nave. Per ottenere questo risultato occorre specificare il tipo che può essere contenuto in una variabile. In pratica non si può utilizzare una variabile se prima non è stata dichiarata. Dichiarare una variabile significa specificare il suo tipo. La sintassi di Java per le dichiarazioni consiste nello specificare il tipo della variabile prima della variabile stessa (come si vede nella figura).

- Una dichiarazione può essere seguita da una espressione di inizializzazione (opzionale):

```
String s = "hello";
```

Una dichiarazione può anche essere seguita da una inizializzazione. Cioè si può specificare il valore iniziale della variabile che si sta dichiarando. In realtà il valore iniziale viene sempre dato ad una variabile. A seconda dei casi questo valore iniziale può essere deciso implicitamente (zero oppure nullo) oppure il compilatore può imporre che la variabile venga inizializzata opportunamente.

## 2.5 Metodi

- Un *metodo*, quando si dichiara è simile ad una funzione matematica: ha dei parametri e ritorna un valore

```
int sum(int x, int y)
```

- Un metodo che prende due interi come parametri e ritorna un intero.

Dopo aver visto come si compongono gli elementi di programmazione (comandi ed espressioni) vediamo adesso come si creano gli elementi di astrazione: in particolare vedremo classi e metodi. Un metodo assomiglia sintatticamente ad una funzione. In pratica si deve dichiarare il nome del metodo (sum) precedendolo con il tipo ritornato (int) eseguendolo con una lista tra parentesi dei tipi dei parametri. Un metodo può quindi essere considerato anche semanticamente una funzione: ovvero quel costrutto matematico che prende dei valori in ingresso, fa dei calcoli e ritorna un risultato in uscita.

- Un metodo è seguito da un comando che definisce il suo comportamento:

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

Un metodo non è però semplicemente una funzione: è invece una sequenza di comandi che effettuano il calcolo del risultato. Sintatticamente occorre porre la sequenza di comandi che effettuano il calcolo tra parentesi graffe immediatamente dopo la dichiarazione del metodo.

- I metodi sono sempre dentro una classe.

Sintatticamente troveremo sempre i metodi all'interno di una classe. Infatti non parliamo di funzioni ma di metodi, proprio perché questi sono collegati ad una classe. Un metodo è una operazione che viene svolta nel contesto di un oggetto, istanza di una classe.

- Infatti i metodi hanno sempre un *contesto* (o ambiente):

```
int sumk (int x) {  
    return x+k;  
}
```

- *k appartiene al contesto.*

Come possiamo vedere nel listato i metodi vengono sempre valutati all'interno di un contesto. Per capire meglio di cosa si tratta notiamo che nel metodo che stiamo esaminando c'è una variabile, k, che non è né un parametro non è una variabile locale. Come si dice in gergo, è una variabile libera, cioè una variabile non dichiarata come parametro o localmente. In realtà questa variabile non è dichiarata dentro il metodo, ma viene dichiarata fuori, in particolare dentro classe a cui il metodo appartiene. Questo rinforza il fatto che un metodo ha sempre un contesto, da cui vengono presi i valori delle variabili libere.

- **In un metodo possono esserci dichiarazioni di variabili (locali al metodo):**

```
int sumk (int x, int y) {  
    int r=x+y;  
    return r;  
}
```

Facciamo notare come alcune variabili vengano dichiarate localmente al metodo. In particolare possiamo riconoscere le variabili x e y che sono i parametri del metodo, e la variabile r che è invece una variabile locale.

## 2.6 Classi

### ● Le classi contengono campi e metodi:

Un *campo* (**k**) è una cosa **diversa** da una *variabile locale* (**h**) e da un *parametro* di un metodo (**x**) .

```
class Sum {  
    int k=1;           // k campo  
    int add3(int x ) { // x parametro  
        int h = 2;      // h variabile locale  
        return x+h+k  
    }  
}
```

Osserviamo adesso attentamente questo listato, che rappresenta completamente una unità sintattica autonoma di un programma Java. In pratica quello che vediamo è una classe completa che può essere compilata ed eseguita autonomamente. Possiamo riconoscere i tre tipi di variabili che sono presenti in un programma Java.

Le variabili dichiarate all'interno del metodo sono le variabili locali. Invece tra le parentesi all'inizio della dichiarazione del metodo ci sono i parametri. Si tratta a tutti effetti di variabili che però vengono inizializzate quando il metodo viene chiamato, e servono come deposito dei valori passati al metodo.

Le variabili invece che vengono dichiarate al di fuori del metodo (ma sempre dentro una classe) si chiamano genericamente campi. In particolare la terminologia che si usa normalmente fa una distinzione tra campi e variabili, dove per variabili si intendono i parametri e le variabili locali. Quindi un campo è un campo, una variabile è un parametro o una variabile locale. Seguiremo questo terminologia nel resto del corso.

# 3 Tipi di dato

---

● Come abbiamo detto, ci sono due "gruppi" di tipi di dato:

- Primitivi
- Classi

Vediamo adesso una importantissima distinzione sui tipi di dato che possono essere contenuti nelle variabili o nei campi.

Diciamo che in Java che sono due gruppi di tipi di dato: il primo gruppo rappresenta il tipi di dato primitivi, mentre secondo gruppo contiene i tipi di dato derivati, ovvero le classi.

Questi due gruppi hanno molte similitudini ma anche hanno delle fondamentali differenze. In realtà la distinzione tra tipi primitivi e oggetti è la prima difficoltà che si incontra per capire Java quando si viene da un altro linguaggio. In questo, Java si dimostra più radicale di altri, in quanto sacrifica ottimizzazioni che si possono fare in altri linguaggi per ottenere un approccio totalmente orientato agli oggetti. Questo comporta una pesante utilizzazione di strutture dati allocate dinamicamente.

## 3.1 Primitivi

### ● Tipi primitivi

- Sono fissi
- Una "istanza" di tipo primitivo è generata da una costante
- Si agisce su di essi con gli operatori
- Una variabile "contiene" un tipo primitivo
  - il concetto tradizionale di variabile

Esaminiamo prima di tutto le caratteristiche di tipi primitivi. I tipi primitivi sono in numero fisso, che non può in nessun modo essere ampliato. Ovvero i tipi primitivi sono fissati dalla specifica del linguaggio Java. Per ottenere un tipo primitivo utilizziamo una sintassi particolare, che è quella delle costanti. Una sequenze di carattere numerici che servono a specificare, per esempio il n. 17, è una costante.

Sui tipi primitivi possiamo fare delle operazioni ben note, e a queste operazioni corrispondono i familiari operatori. In particolare notiamo il simbolo della somma, il trattino che è il simbolo della sottrazione, oppure l'asterisco che è il simbolo della moltiplicazione.

Quando dobbiamo operare su un tipo primitivo, lo dobbiamo mettere in una scatola. Più precisamente dobbiamo utilizzare una variabile che serve a maneggiarli. La caratteristica dei tipi primitivi è che la scatola li contiene completamente. Questo può sembrare ovvio ma non lo è quando faremo un confronto con i tipi derivati. Diciamo che in Java le variabili sono piuttosto piccole e generalmente non sono in grado di contenere completamente un oggetto, ma solo un "filo" che permette di ritrovarle in memoria.

## 3.2 Classi

### ● Classi:

- Sono in numero illimitato (creati dal programmatore)
- Una "istanza" di oggetto è generata da un costruttore
- Si agisce su di essi con i metodi
- Una variabile "riferisce" un oggetto: è un puntatore!

Vediamo adesso le caratteristiche delle classi. Il numero di classi è illimitato: infatti vengono creati dal programmatore. La programmazione in Java consiste appunto nel creare e realizzare come classi.

Analogamente ai tipi primitivi abbiamo un modo per generare una istanza di una classe, ovvero un oggetto. Per capirsi meglio pensiamo che 17 è un intero: ovvero abbiamo la categoria degli interi e una incarnazione di questa categoria che appunto il n. 17. Così potremo avere la classe delle navi, sia una istanza della classe ovvero la particolare nave che ci porterà nell'isola che abbiamo scelto per la villeggiatura.

Sui tipi primitivi operiamo usando gli operatori. L'analogo degli operatori per le classi sono i metodi. Questi definiscono le operazioni che possono essere effettuate su un determinato oggetto istanza di una classe.

Ma la caratteristica più importante delle classi, è il fatto che le istanze non possono essere contenute completamente in nessuna variabile. Invece le variabili contengono un puntatore all'istanza avere propria, o meglio, come si dice correttamente, un riferimento. In realtà esiste una differenza tra un riferimento e un puntatore ma questa differenza non ci interessa in questo contesto. Possiamo tranquillamente pensare che le variabili il cui tipo è una classe contiene un puntatore all'istanza di una classe. Se avete nozioni di Pascal, potete immaginare una cosa simile a un puntatore del Pascal (anche se realtà non è esattamente la stessa cosa).

## 3.3 Confronto

### ● Tipi Primitivi

#### ● Creazione con le *costanti*

```
int a=1;
```

#### ● Si opera con gli operatori

```
int b = a+1;
```

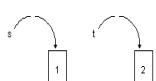
### ● Classi

#### ● Creazione con i *costruttori*

```
Stack s=new Stack();  
Stack t= new Stack();
```

#### ● Si opera con i *metodi*

```
s.push(1);  
s.push(2);
```



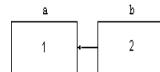
In questa figura entriamo in maggior dettaglio facendo un confronto diretto tra un tipo primitivo e una classe. Nella prima riga a sinistra vediamo la sintassi che serve a dichiarare una variabile il tipo primitivo con un valore iniziale, specificato da una costante. A destra vediamo una variabile una certa classe, Stack, e vediamo come valore il valore iniziale venga assegnato creando una istanza della classe utilizzando quello che si chiama un costruttore.

Nella seconda riga operiamo sul tipo primitivo e sull'oggetto. A sinistra vediamo come si esegua un'operazione utilizzando l'operatore di somma. A destra invece vediamo come debba venire invocato il metodo per operare su un oggetto istanza della classe Stack.

## 3.4 Assegnamento

### ● Primitivi

- $a = b;$
- L'assegnamento ha copiato il valore

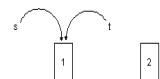


- $a = 3; b == 2$
- $b$  non è stato modificato

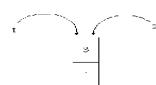


### ● Classi

- $t = s;$
- L'assegnamento ha copiato il riferimento



- $t.push(3); s.pop() == 3$
- $s$  è stato modificato operando su  $t$



Più importante è considerare quello che avviene quando si copiano i valori contenuti nelle variabili. Nella prima riga a sinistra vediamo cosa succede, quando si assegna il valore di una variabile (che contiene un tipo primitivo) ad un'altra variabile. Il valore viene fisicamente copiato dalla prima nella seconda. Invece a destra vediamo cosa avviene quando si assegna il valore di una variabile che contiene un riferimento ad un oggetto. In pratica non viene copiato l'oggetto ma solamente il riferimento con il risultato che avremo 2 variabili che si riferiscono allo stesso oggetto.

L'effetto di questo fatto viene illustrato nella seconda riga. A sinistra notiamo come assegnare un valore ad una variabile non ha alcun impatto su quello che succede all'altra variabile. Con l'assegnamento è stata fatta una copia, e i due tipi primitivi sono ormai cose separate, per di più interamente contenute dentro le loro scatole, che sono le variabili.

Invece con i oggetti si ha un fenomeno completamente diverso: l'oggetto in realtà è uno solo, e facendo assegnamento si è copiato solamente il puntatore all'oggetto. Di conseguenza fare una modifica l'oggetto invocando un metodo su una variabile comporta che questa di modifica sarà visibile anche all'oggetto, anche quando viene acceduto usando l'altra variabile.

## ● Espressioni

È il momento di cominciare a programmare. Lo faremo poco a poco imparando a costruire i mattoni ultimi dei programmi. Iniziamo per così dire dal basso, affrontando i piccoli noiosi cavilli delle espressioni, per poi affrontare il modo con cui si mettono insieme nei comandi. Questa parte è abbastanza ricca di dettagli un po' noiosi, ma che sono fondamentali per scrivere programmi funzionano. Affrontare adeguatamente le espressioni è l'occasione di evitare di rimanere bloccati in futuro, quando non sia avrà ben chiaro come fare delle cose elementari.

### ● Incorporate in comandi

Le espressioni da sole non stanno in piedi, in realtà sono una specie di appendice dei comandi. Nel senso che si trovano completamente solamente quando si sta scrivendo un comando. Però hanno una logica e una vita propria, per cui possono essere spiegate tranquillamente senza far riferimento ai comandi.

### ● Calcoli su tipi primitivi



Una espressione in ultima analisi serve a far un calcolo, che normalmente è sui tipi primitivi. In gran parte tratteremo numerosi operatori e spiegheremo in dettaglio come questi operano sui tipi primitivi. Sarà l'occasione di esaminare dettagli come la differenza tra virgola mobile e intero, e sapere come ci si comporta in caso di infiniti le divisioni per zero.

## ● Costruzione di oggetti e accesso a metodi

In realtà sono espressioni anche i costrutti i che permettono di creare oggetti e accedere a metodi, ma si riducono a soli due operatori, ovvero new e il punto, che tra l'altro abbiamo già visto nell'introduzione.

# 4 Espressioni

## ● Per calcolare le espressioni occorre considerare:

### ● Precedenza, ovvero chi prevale tra due operatori diversi:

Le espressioni sono composte da una sequenza di operatori alternati a operandi. Valgono le regole tradizionali della aritmetica, quelle che abbiamo imparato già dalla scuola elementare. Ci sono però alcuni concetti che vale la pena di ribadire per fissare le idee per fornire delle regole sicure su come vengono effettuati calcoli.

```
a + 2 * 3  
// come va calcolato?  
a + (2 * 3)  
// * ha la precedenza su +
```

se vediamo la prima espressione, le regole della scuola elementare ci dicono che la moltiplicazione va fatta prima della somma. Questa semplice regola empirica viene tradotta nei linguaggi di programmazione con un la regola della precedenza. Ovvero quando un operando, in questo caso il 2, è conteso tra due operatori, in questo caso il + e il \*, prevale chi ha la precedenza. Esiste una tabella, di seguito riportata, che dice l'ordine di precedenza degli operatori. Quindi in una espressione complessa alcuni operatori comandano su altri e inducono un ordine di esecuzione nei calcoli. Quest'ordine può essere ricavato semplicemente applicando le regole di precedenza. A volte è fondamentale applicarle a mano. Comunque nel dubbio si può sempre ricorrere a delle parentesi, che alterano la precedenza, e anche se sono ridondante non creano problemi.

### ● Associatività ovvero chi prevale con un solo operatore:

la regola della precedenza si applica anche quando un operando è conteso tra due occorrenze dello stesso operatore. In questo caso si parla però di precedenza ma di associatività. In generale l'associatività dice che si devono fare calcoli da destra verso sinistra, o da sinistra verso destra.

```
a + b + c
// come va calcolato?
(a + b ) + c
// + associa da destra
a = b = c
// come va calcolato?
a = ( b = c)
// = associa da sinistra
```

in pratica in Java sono rilevanti solamente due casi, il caso in cui si vuole fare un calcolo, e il caso in cui si vuole assegnare un risultato. Nel primo caso, come si vede dalla figura, il calcolo procede da sinistra verso destra come ci si aspetterebbe. Secondo caso invece, siccome si vuole che il valore assegnato risalga verso la variabile, l'associatività è da destra verso sinistra. Il caso dell'assegnamento può sembrare strano, almeno per chi viene da linguaggi diversi dal C. Vediamolo meglio.

### ● Effetti collaterali:

```
a = 1
// espressione, vale 1
// come effetto collaterale di =
// a viene modificata
a = b = 1
// ad a assegno il valore di (b = 1)
a = int b = 1
// ERRORE!
// int b (inizializzazione)
// non è una espressione
```

una delle caratteristiche del Java è il fatto che le espressioni permettono la modifica delle variabili. Questa caratteristica che è dedicata dal linguaggio C. Ricordiamo infatti che Java è una versione semplificata del linguaggio C++, che a sua volta è un'estensione del C. Comunque sia, il concetto fondamentale è che ci sono operatori che possono modificare le variabili. Normalmente si pensa che un operatore fatto solo un calcolo, come nel caso di `a + 1`. In questo caso non c'è niente di speciale: viene effettivamente sommato 1 al valore di `a`.

Ma nel caso  $a = 1$ , le cose sono un pò più strane. Si tratta effettivamente di un operatore, che fa un calcolo, ma questo calcolo è la funzione identità, ovvero  $a = 1$  ritorna semplicemente il valore del secondo operando, ovvero 1. Cioè  $a = 1$  quale 1, come  $a + 1$  vale 2, se  $a$  è 1. Allora ci può chiedere il senso del fare un'operazione come  $a = 1$ . La ragione di quest'operazione sta che fatto che l'uguale ha un effetto collaterale, che è quello di assegnare il valore del secondo operando al primo. Infatti di operando di = dev'essere un lvalue. In gergo significa un valore assegnabile, una variabile in pratica, o un elemento di una schiera. Possiamo adesso quindi esaminare maggior dettaglio  $a = b = 1$ . Succede che applicando la regola della associatività, viene eseguito per primo  $b = 1$ . Quest'espressione vale 1 e modifica  $b$ . A questo punto si calcola  $a = [\text{risultato}]$ , ottenendo come valore  $[\text{risultato}]$ , e assegnando questo  $[\text{risultato}]$  ad  $a$ . In definitiva in questo modo si assegnano ad  $a$  e  $b$  il valore 1.

## 4.1 Primitivi

### ● Dimensione fissa per tutte le Java Virtual Machine:

Visto che la parte fondamentale di questa sezione sono tipi primitivi, cominciamo a esaminarne le caratteristiche più importanti. Java è progettato per essere un linguaggio portatile, ovvero linguaggio che può essere eseguito su sistemi operativi diversi. Per questo motivo si pone la massima attenzione ad dettagli che in altri linguaggi sono considerati con minor rigore. Uno di questi dettagli è la dimensione dei tipi di dato. Nella seguente tabella sono riassunti i tipi di dato primitivo e la loro dimensione come viene manipolata dall'interprete Java.

<i>Tipi Interi</i>	<i>Dimensione</i>
byte	8 bit
short	16 bit
int	32 bit
long	64 bit

I tipi di dato intero sono quattro. Il tipo di dato più usato è l'intero vero e proprio, int, che ha dimensione 32 bit. Se occorrono dei tipi di dato i dimensione inferiore si può ricorrere a byte e short, e sono rispettivamente a 8 e 16 bit. Quando invece occorrono interi di grandi dimensioni, si ricorre al tipo long che ha 64 bit. Notare che quest'ultimo tipo è di uso abbastanza frequente, più che short e byte. Per esempio l'ora attuale viene rappresentato in Java con il numero di secondi trascorsi dal 1 gennaio 1970, numero molto grande, che viene normalmente contenuto in una variabile di tipo long. Ci sarebbe anche da puntualizzare il fatto che i tipi interi contengono dati in rappresentazione cosiddetta in complemento a due. Rimandiamo a testi di informatica di base per dettagli su questa rappresentazione, anche se all'atto pratico non è necessario conoscerla approfonditamente, se non per il fatto che bisogna sapere che sui numeri interi è possibile effettuare solo operazioni di tipo intero.

<i>Tipi Floating</i>	<i>Dimensione</i>
float	32 bit (IEEE 754)
double	64 bit (IEEE 754)

Invece il tipi di dato in virgola mobile sono solamente due, più precisamente il float e il double. Il primo rappresenta un numero in virgola mobile a 32 bit mentre secondo ha 64 bit. In questo caso ciò che differenzia i numeri virgola mobile da quelli interi è appunto la rappresentazione. In Java i numeri virgola mobile vengono rappresentati utilizzando il formato standard IEEE 754. Non ha molta importanza sapere i dettagli di questo formato, quello che è importante sapere è che con questa rappresentazione è possibile specificare numeri decimali approssimati, ma con un campo di valori più grande rispetto a quello dei numeri interi. Un'altra cosa importante è il fatto che quest'aritmetica supporta infiniti e forme indeterminate.

<i>Altri Tipi</i>	<i>Dimensione</i>
<b>boolean</b>	<b>1 bit</b>
<b>char</b>	<b>16 bit (Unicode)</b>

Rimangono da esaminare altri due tipi di dato. Il primo è il boolean, che rappresenta semplicemente un valore vero o falso, che viene utilizzato nei calcoli che servono per verificare condizioni (per esempio negli if o nei cicli while). I soli valori possibili sono appunto true e false.

L'altro tipo di dato da esaminare è il carattere. Per molti versi il carattere è sostanzialmente un numero intero, infatti è molto facile convertirlo a intero e viceversa. L'aspetto importante per carattere in Java, che lo differenzia dai caratteri di altri linguaggi, è il fatto che è a 16 bit, e non a 8 bit come nella maggior parte degli altri casi. Questo perché tradizionalmente i caratteri sono numeri che rappresentano un elemento del set di caratteri ASCII, e 8 bit bastano per individuare il carattere (per la verità ne bastano 7, ma spesso si usano set di caratteri ampliati ad 8 bit). In Java invece i caratteri sono nel set di caratteri internazionale Unicode, e occorrono 16 bit per individuarli. Il fatto che i caratteri in Java siano Unicode si vede particolarmente nelle operazioni di I/O: infatti solitamente i file vengono letti e scritti usando i byte e non i caratteri. Questo perché solitamente vengono scritti assumendo che contengano solo caratteri a 8 bit. Quindi bisogna operare delle conversioni tra Unicode e il set di caratteri del file.

## 4.2 Costanti

---

### ● Ci sono vari tipi di costanti

- Booleane
- Carattere
- Numeriche: intere e float
- Stringa
- Array

stiamo esaminando in dettaglio le espressioni. Ricordiamo il fatto che le espressioni sono composte di costanti, operatori e infine variabili. Vediamo adesso per cominciare quali sono i tipi di variabili disponibili. In particolare osserveremo che ci sono costanti di tipo booleano, costanti di tipo carattere, costanti numeriche sia intere che virgola mobile, e infine le costanti che abbiamo già visto, ovvero schiere e stringe.

## 4.3 Booleane

● Le costanti booleane *non* sono interi

● Sono i due *letterali*:

● **true**

● **false**

Come abbiamo le costanti numeriche come 17, abbiamo anche le costanti booleane, che sono esplicitamente true e false. Non riservano particolari sorprese, unica cosa è che una espressione booleana può essere assegnata solo a una variabile booleana.

● Conversione intero in booleano

● Usare `if(a!=0)` al posto di `if(a)` del C

● Conversione Booleano in Intero

● Usare `a ? 0 : 1`

In altri linguaggi non è disponibile il tipo di dato separato booleano, e allora spesso si utilizza il tipo intero per rappresentare i valori di vero e falso. In Java non è possibile usare un intero per dire vero o falso, ma che vuol poco a trasformare un intero in un valore vero o falso come viene inteso di solito. Siccome spesso un intero i che rappresenta vero è un intero diverso da zero, passa scrivere esplicitamente `i != 0`. Allo stesso modo se si vuole scrivere 1 per vero e zero per falso, basta usare le espressioni in figura. In realtà si tratta di un caso particolare dell'espressione condizionale che vedremo più avanti.

## 4.4 Carattere

- Le costanti carattere sono caratteri tra virgolette singole:

```
'a' '1' '\n'
```

Adesso vediamo le costanti carattere. Normalmente si usare caratteri per scrivere i programmi. Che si vuole che un carattere tenga presso come una costante carattere, bisogna scriverlo tra apici singoli. Fare attenzione al fatto che gli apici sono singoli, perché gli apici doppi hanno spiegato completamente diverso. L'abbiamo già visto: l'apice doppio racchiude di caratteri di una stringa. Ma mentre una costante carattere è un tipo primitivo, una costante stringa è un oggetto, istanza di una classe, con un comportamento completamente diverso.

- Alcuni caratteri sono esprimibili con una sequenza di più caratteri

Per molti caratteri basta scriver carattere stesso tra apici per rappresentarlo, ma questo non è vero per tutti caratteri. Per esempio non è possibile mettere tra apici un ritorno a capo. Per definire costanti carattere che rappresentano questi caratteri speciali, si utilizza delle sequenze di due caratteri o più caratteri. Notare bene che queste sequenze speciali sono composte da due caratteri, ma ne rappresentano solamente uno.

- queste sequenze cominciano con \ (backslash):

Backslash	\ \
Continuazione	\ \
Spazio indietro	\ b
Ritorno del carrello	\ r
Salto Pagina	\ f
Tabulazione orizzonatale	\ t
Nuova linea	\ n
Virgoletta semplice	\ '
Doppia virgoletta	\ "
Carattere ottale	\ddd
Carattere unicode	\udddd

nel listato precedente sono elencate le principali. Il carattere che inizia tutte le sequenze speciali è la barra inversa. Ovviamente se si vuole inserire la stessa barra inversa bisogna usare una sequenza speciale che è la doppia barra inversa. La maggior parte delle sequenze speciali sono composte da soli due caratteri, è il secondo è una lettera. La sequenza di gran lunga più usata è \n, che significa appunto ritorno a capo.

● **Notare la sequenza \ddd che serve ad esprimere un carattere ASCII scrivendone il suo codice *in ottale*:**

- dopo \ ci vogliono da **1 a 3 cifre ottali**

● Esiste anche la sequenza \udddd che serve ad esprimere un carattere UNICODE scrivendone il suo codice *in esadecimale*

- dopo \u ci vogliono **esattamente 4 cifre esadecimali**

Ci sono però delle sequenze che sono composte da più di due caratteri. La prima è \ddd, dove i caratteri che seguono barra inversa sono il codice del numero del set ascii in ottale. Questa sequenza è ereditata dal linguaggio C non è particolarmente utilizzata. Comunque c'è, quando serve è utilizzabile. Notare che dopo la barra ci devono essere uno, due o tre cifre ottale. Un'altra sequenza invece, che composta sempre da 5 caratteri, e le sequenze che permette di inserire costanti carattere nel set di caratteri Unicode. Questa sequenza è formata dalla barra inversa seguita da una U, seguita da esattamente quattro cifre esadecimali, che rappresentano il codice del carattere che si vuole specificare nel set di caratteri unicode.

## 4.5 Numeri

### ● Intere:

- decimali: 17 -1
- esadecimali: 0xff cifre 0-9, A-F, 0x o 0X in testa
- ottali: 0177 cifre 0-8, 0 in testa

possiamo scrivere costanti il tipo intero usando la base decimale, sia la base 8 e 16. Per scrivere un numero intero in base 16 (ovvero esadecimale) si deve iniziare una costante numerica con 0x. Invece per scrivere un numero intero usando come base l'ottale si deve scrivere numero che inizia per 0. Tutto qui. A volte però questa può trarre in inganno, in quanto se si scrive 010 in un contesto cui tutti gli altri numeri sono altre cifre, si vorrebbe scrivere il numero dieci e invece si scrive il numero 8.

### ● Intere lunghe:

- decimali: 17L, -1L

per scrivere una costante di un lungo si utilizza alla fine del numero una lettera elle. Solitamente questa lettera si scrive maiuscola, perché la lettera elle minuscola troppo simile al numero 1 da trarre continuamente in inganno. L'uso della costante lunga esplicita va considerato quando si vuole forzare la precisione di un'espressione a un certo numero di bit. Per rendersi conto di questo concetto occorre però esaminare la regola della promozione vedremo più avanti.

### ● Non ci sono costanti short o byte

- o se ne accorge il compilatore:

`byte b = 17;`

- oppure viene richiesta una conversione:

`b = (byte)257;`

vediamo infatti una caratteristica del calcolo delle espressioni la mancanza delle costanti più piccole. Questo fatto ricondotto al fatto che i calcoli interi vengono fatti sempre a 32 bit, per cui anche a voler prendere in mano un intero piccolo a 8 o 16 bit, questo verrebbe automaticamente ampliato a 32 bit, per cui non ha senso inserire una sintassi per esprimere costanti più piccole dell'intero. L'unico caso in cui può servire è nelle inizializzazioni, nel qual caso ci pensa il compilatore a ridurre opportunamente il valore da assegnare la variabile.

## ● Floating point:

### ● 17d 17D

- double con la d o la D in coda **opzionale**

Una costante, in virgola mobile si caratterizza dal fatto che ha la virgola (ovvero, siccome si utilizza la passione inglese, in realtà il punto decimale). Una costante in virgola mobile può esserlo anche se viene utilizzato invece che il punto decimale, la lettera e dell'esponente. Si può ancora scrivere una costante in virgola mobile anche se il valore reale è intero, semplicemente aggiungendo '.0'.

### ● 19 . 4 19f

- float, in formato decimale con la f o la F in coda **obbligatoria**

A questo punto è importante precisare che nei calcoli in virgola mobile, comanda il tipo double, a 64 bit. Per cui 17.0 è una costante di tipo double. Se esprime il valore in virgola mobile a precisione inferiore, ovvero il float, basta usare come estensione della cifra la lettera f.

## 4.6 Operatori

### ● Gli operatori agiscono sui *tipi primitivi*:

In questa pagina vediamo le categorie degli operatori. Si tratta di un modo arbitrario per categorizzare gli operatori presenti nel linguaggio Java. Seguiremo questo ordine nella descrizione che stiamo per dar nelle pagine successive.

#### ● RELAZIONALI:

- > >= < <= != ==

Gli operatori relazionali sono quelli che servono a fare dei confronti. Uguale, , maggiore uguale, diverso eccetera. In particolare notare la sintassi del diverso che utilizza il punto esclamativo per dire "non uguale".

#### ● ARITMETICI:

- \*\*+ - \* / %\*\*

I operatori aritmetici sono quei soliti, addizione sottrazione riflessione divisione, e in più che c'è l'operatore % che serve indicare il resto della divisione (ovvero il modulo).

#### ● BINARI:

- & | ^ >> << <<<

I operatori binari sono i operatori che lavorano sui bit, ovvero sulla rappresentazione di un numero come una sequenza di zeri e uni.

#### ● LOGICI:

- && || ?:

I operatori logici sono quei operatori che servono a mettere in sequenza delle comparazioni. Assomigliano un pò agli operatori binari, ma hanno della fondamentali differenze per quanto riguarda l'ordine di esecuzione del calcolo.

#### ● INCREMENTO:

- ++ --

Gli operatori di incremento sono caratteristici del linguaggio C, sono stati ereditati da Java passando per il C++. Lo stesso nome C++ significa C incrementato.

## ● ASSEGNAZIONE:

● = += -= \*= /= ...

Anche queste operatori sono caratteristici del C, vengono usati principalmente nei cicli, e ne abbiamo già visto un'occorrenza quando abbiamo fatto notare il += che serve a concatenare una stringa ad una variabile stringa.

## 4.7 Promozione

La regola della promozione è importante da comprendere per rendersi bene conto di come avvengono i calcoli quando si programma in Java. In realtà si capisce meglio il senso di queste regole quando si immagina quello che avviene sotto il pavimento. I calcoli vengono effettuati dalla JVM, ovvero l'interprete Java. Questo interprete assomiglia molto un microprocessore, che ha dei registri per fare dei calcoli. Quando si deve fare una somma, il valore da sommare devono venir caricati nei registri, e solo dopo si può operare. Le operazioni richiedono che entrambi registri abbiano la stessa dimensione. I registri della JVM sono a 32 bit per quanto riguarda l'aritmetica intera, e a 64 bit per quanto riguarda l'aritmetica in virgola mobile.

### ● Nei Calcoli

Quando si effettua un calcolo, questo viene suddiviso in tanti passi, in cui ci prendono due operanti e si opera su di essi. Per effettuare un passo bisogna per prima cosa caricare i valori in due registri di pari dimensione.

#### ● ogni valore intero inferiore ad int diventa int.

per cui se devo operare tra due valori interi, per prima cosa questi devono essere caricati sui registri interi. Siccome i registri sono almeno a 32 bit, ecco che avviene il primo passo della promozione: tutti valori interi di dimensione inferiore ai 32 bit vengono promossi a 32 bit.

#### ● ogni valore float diventa double

analogamente se per caso uno due operanti diventa in virgola mobile, la dimensione dei registri virgola mobile è di 64 bit, per cui calcoli in virgola mobile vengono sempre fatti nella precisione del double.

#### ● un operatore opera solo su valori della stessa dimensione

#### ● tra due operandi, (a + b) il "più grosso" decide

##### ● l'altro viene ingrandito

In tutti gli altri casi, quando uno due operanti è più piccolo dell'altro, il più piccolo viene convertito nel più grosso. In particolare se è uno è long, anche l'altro diventa long. E se uno dei due è double, anche l'altro diventa double.

## per assegnare a variabili più piccole bisogna "convertire" (troncando)

Alla fine dei calcoli si ottiene un numero che ha alla precisione dell'operando più grosso, ma non è detto che il risultato stia nella variabile che si vuole assegnare. Per questo motivo esiste un operatore di conversione, o cast, che serve a convertire esplicitamente nella precisione che serve. Se si vuole ampliare non c'è problema, conversione è addirittura implicita e non occorre specificarlo. Più problematico nel quando si vuole ridurre dimensione o si vuol passare da un numero in virgola mobile un numero intero. In questi casi occorre troncare o arrotondare, con il rischio di perdere informazioni. Per questo motivo l'arrotondamento finale deve essere esplicito. Il programmatore deve sapere, ed deve ordinarlo esplicitamente, che c'è il rischio di perdere precisione.

```
01. byte b=1;
02. short s=2;
03. long l=3;
04. double d=4.0;
05. int t=b+s;           // è int (32 bit)
06. t+l                 // è long (64 bit)
07. l+d                 // è double (64 bit)
08. byte n=t+l;         // ERRORE
09. byte n=(byte)t+l;   // OK
```

esaminando quest'esempio passo passo possiamo renderci conto di quello che succede. Nelle righe da 1 a 4, stiamo semplicemente assegnando del valore a delle variabili. Nelle riga 5 stiamo facendo una somma di uno short con un byte. Per regola che i calcoli interi vengono effettuati almeno in precisione intera, sia b che s diventano interi, e il risultato è intero. Sommando un intero ad un long, otteniamo che l'intero è promosso a long, il risultato è long. Allo stesso modo, sommando un long ad un double, il risultato è doble. Nelle righe 8 vediamo come il tentativo di assegnare un long ad un byte fallisce. Quel codice non compila. Questo perché non si può mettere valore a 32 bit dentro una scatola che può contenere solo 8 senza buttarne via 24. Usando "(byte)" viene effettuata la conversione e l'eventuale troncamento.

## 4.8 Overloading

### ● In realtà gli operatori sono "funzioni overloaded"

uno dei segreti meglio riposti di Java e di molti altri linguaggi di programmazione, e il fatto che gli operatori sono più di uno, ma con lo stesso nome. Per capirci, consideriamo la somma: in Java possibile sommare due interi, oppure sommare due numeri in virgola mobile, oppure sommare due stringhe. È particolarmente evidente nel caso di stringhe, che le operazioni di vengono fatte per sommare due stringhe sono diverse dalle operazioni che vengono fatte per sommare due interi. Ma in realtà sono diverse anche le operazioni che vengono fatte quando si sommano due numeri in virgola mobile.

#### ● la somma prende due int e ritorna un int

una somma di interi utilizza la rappresentazione interna cosiddetta in complemento due. Non importa quello che realmente sia questa rappresentazione, importa che si opera su questo tipo di dati.

#### ● oppure prende due double e ritorna un double

invece la somma in virgola mobile prende due valori che utilizzano la rappresentazione IEEE 754, una rappresentazione completamente diversa da quella in complemento due. Quindi per fare la somma si fanno delle operazioni che sono diverse da quelle fatte per fare la somma intera. Tra l'altro il tipo del risultato è pure differente : del primo caso abbiamo un intero, nel secondo un double.

### ● Analogo ai metodi overload

questo fatto, ovvero che ad un nome corrispondono più operazioni è particolarmente importante quando si programmano metodi, ma è vero pure per gli operatori. Tecnicamente si chiama overloading, cioè sovraccarico. L'operatore somma è come il manager di una ditta, che deve fare sacco di cose, ed è pertanto sovraccarico di lavoro.

```
int sum(int x, int y)
```

#### ● Prende due int e ritona un int

```
double sum(double x, double y)
```

- Ha lo stesso nome, ma prende due double e ritorna un double

il programmatore ha infatti la possibilità di scrivere qualcosa come quello mostrato nel listato. Vedete due metodi di somma, il primo che prende due interi e ritorna un intero, e il secondo che prende due double e ritorna un double. Il punto è che entrambi i metodi hanno stesso nome, ma hanno un corpo (e quindi una implementazione) distinti. Ciò che distingue quale dei due metodi viene chiamato è il tipo dei parametri. Quindi il fatto che sum venga chiamato con due interi causa esecuzione di un algoritmo diverso da quello che viene seguito quando lo chiamo con due double. Questo fatto, che può sembrare un cavillo, è uno dei meccanismi fondamentali di Java, ed è forse uno dei più difficili da gestire. Mentalmente il programmatore deve fare caso al tipo che si ottiene. Per questo motivo è particolarmente importante stare attenti al fatto che sommando un long ad un double si ottiene un double. Dal tipo del risultato dipende quale metodo viene chiamato.

## 4.9 Aritmetici

### ● + - \* / % (binari)

- Somma, sottrazione, prodotto, divisione e modulo.

- Prendono due argomenti, di tipo int o float e ritornano un int o float.

Gli operatori più semplici almeno in apparenza sono quelli aritmetici: somma, sottrazione, moltiplicazione e divisione. Ne esiste un altro, indicato dal simbolo di percentuale, che indica il resto della divisione intera. In realtà dell'uso di questi metodi c'è subito una trappola, che illustra il meccanismo del sovraccarico di operatori. La trappola è: quanto fa  $5 / 2$ ? Se sbagliate, state tranquilli perché in questo errore che si incappa anche quando si ha già molte esperienza, specie in espressioni complicate. La risposta questa domanda è 2. Se mi sorprende, sappiate che invece scrivendo  $5.0 / 2$  si ottiene quello che forse aspettate, ovvero 2.5. In realtà alla barra corrispondono due diverse operazioni: la divisione intera e la divisione in virgola mobile. La divisione intera, ovvero divisione tra due interi, ritorna un intero, e in particolare tronca il risultato. Se vogliamo effettuare un calcolo in virgola mobile, almeno uno dei due operandi dev'essere in virgola mobile. In questo caso scatta il meccanismo, già detto prima, della promozione: ovvero quando uno dei due operandi è più piccolo dell'altro, viene promosso. Per attivare la divisione in virgola mobile occorre che entrambi i operandi sia in virgola mobile (o meglio ne basta uno perché l'altro viene automaticamente promosso).

### ● - + (unari)

- Il meno unario inverte il segno di un int o di un doble

- Il più unario non fa nulla

- ma serve per poter scrivere  $+4$  senza errore

I operatori visti prima prendono due operandi, ma in realtà questi operatori possono essere usati anche con un solo operando. Per esempio,  $-a$  è un operatore che prende  $a$  e ritorna il valore negato di  $a$ . Strettamente parlando esiste anche  $+a$ , che prende  $a$  e restituisce  $a$ . Ovvero non far niente: è la funzione identità, ritorna il suo argomento. Serve solamente per poter scrivere senza problemi  $+4$ , che strettamente non è una costante corretta sintatticamente.

## 4.10 Virgola Mobile

### ● Java conosce l'analisi matematica!

ma se siete rimasti stupiti del fatto che la differenza tra  $5/2$  e  $5.0/2$ , sarete ancora più sorpresi dal fatto che Java in fatto di matematica la sa lunga.

#### ● Segue lo standard IEEE 754

L'aritmetica in virgola mobile di Java segue uno standard definito dall'associazione degli ingegneri americani, che consente di fare calcoli utilizzando alcune regole che si imparano dal liceo quando si studia l'analisi matematica.

##### ● Esistono gli infiniti per

- `Double.POSITIVE_INFINITY`, `Float.NEGATIVE_INFINITY`,
- etc

innanzitutto esistono gli infiniti: se si divide un numero positivo per 0, si ottiene infinito positivo. Regola standard di analisi matematica, ma finora un poco inconsueta nei linguaggi di programmazione. Esistono infatti delle costanti che rappresentano gli infiniti.

#### ● Esiste la *forma indeterminata* `Float.NaN`

- NaN: Not A Number

dall'analisi matematica sappiamo anche che esistono operazioni che non ritornano un risultato. Per esempio  $0/0$ , non ha soluzione, per cui si dice che è una forma indeterminata. Esiste una costante che rappresenta la forma indeterminata, ed è NaN, Not a Number. Nella seguente tabella sono riassunti i vari casi in cui si ottiene infinito o la forma indeterminata.

$x$	$y$	$x/y$	$x\%y$
finito	+/- 0	+/- infinito	NaN
NaN	+/- infinito	NaN	NaN
+/-	finito	+/- infinito	NaN
+/- infinito	+/- infinito	NaN	NaN

## 4.11 Relazionali

### ● >= <= == !=

I operatori relazionali sono quelli che permettono di confrontare due tipi primitivi. Per caso di uguale e diverso il confronto si applica anche agli oggetti. Negli altri casi invece si tratta di confrontare se un numero è minore, maggiore, minore uguale o maggiore uguale di un altro.

#### ● Notare che l'uguaglianza è con *due segni '='*

##### ● mentre l'assegnamento è *un segno '='*)

Sintatticamente bisogna stare attenti che l'uguaglianza utilizza come operatore il doppio =.

Il singolo = è l'assegnamento. La sintassi di Java abbiamo visto tratta gli assegnamento come espressioni, per cui è necessario distinguere in qualche modo l'assegnamento dal confronto per uguaglianza. Questa distinzione viene fatta utilizzando due diverse operatori. Siccome l'assegnamento è molto più frequente del confronto, si è scelto di assegnare a questo operatore = singolo che è più corto scrivere, mentre = doppio è il confronto. (Notare che per esempio pascal è il contrario: := contro =). Ancora bisogna notare che diverso si scrive "!=", invece di "<>" come altri linguaggi. Questo perché l'esiste un operatore, il "!", che significa "non" (è la negazione booleana), per cui per uniformità anche l'operatore di diverso si adegua.

#### ● Operano su interi e double

Gli operatori di maggiore e minore hanno veramente senso solo sui numeri, quindi su interi e virgola mobile.

#### ● Uguale e Diverso operano anche su booleani e oggetti

Invece l'operatore di uguaglianza permette confrontare anche valori booleani, e soprattutto oggetti. Stare attenti a questa cosa: se io scrivo a == b, dove sia a che b sono due oggetti (non due tipi primitivi), voglio dire che l'oggetto a cui si riferiscono è lo stesso. Un errore in cui s'incappa facilmente è quello di confrontare le stringhe con l'uguale, pretendendo un risultato vero se le stringhe contengono gli stessi caratteri. Questo non è in generale vero: posso aver benissimo in oggetti molto simili tra di loro, ma che sono in realtà oggetti distinti. Per confrontare due stringhe per uguaglianza devo utilizzare a.equals(b). In questo caso ottengo vero se le stringhe contengono gli stessi caratteri.

## ● Notare i confronti in floating point di Java

anche i confronti possono riservare qualche sorpresa. In particolare abbiamo visto come il Java abbia una buona cultura matematica, anche i operatori di maggiore e minore dimostrano di aver appreso la lezione. In particolare quando si fanno confronti in virgola mobile, si tiene conto di infiniti e forma indeterminata.

### ● -infinito < finito < + infinito

Qualunque numero finito è più grande di - infinito. Qualunque numero finito è più piccolo di + infinito.

### ● + infinito == +infinito

- `NaN != NaN`, per cui: `isNaN(a) { return ! (a == a); }`

posso confrontare due infiniti, ottenendo vero quando sono entrambi + infinito o meno infinito.

Non posso però confrontare la forma indeterminata. Qualunque tentativo di effettuare un confronto con questo valore speciale, ritorna un risultato falso. In questo modo sembrerebbe impossibile riuscire a sapere se si è ottenuta la forma indeterminata, invece è molto semplice: basta confrontare la forma indeterminata con se stessa. L'unico caso in cui un numero è uguale con se stesso è quello in cui è la forma indeterminata. Esiste comunque il metodo `Math.isNaN` che dice esplicitamente se un numero è una forma indeterminata, ed è il caso di usarlo la maggior parte delle volte perché è più chiaro. Quello che fa comunque è quello che è mostrato nel listato.

## 4.12 Binari

Gli operatori binari sono quelli che considerano i numeri come una maschera di bit, permettono di operare sui singoli bit. Questo è possibile solamente quando un numero è intero.

### ● ~ (not)

- opera solo su interi
- inverte ogni bit

il più semplice è l'operatore detto tilde (o serpentello). Questo operatore inverte i singoli bit della maschera. È importante tenere presente la precisione del numero che si vuole invertire. Se scrivo `~1`, per le regole della promozione il numero uno ha precisione a 32 bit, quindi sto invertendo il numero binario `00000000000000000000000000000001`. Se invece scrivo `~1L`, il numero 1L a una precisione a 64 bit, per cui devo invertire i bit del il numero `0001`.

### ● & (and) | (or) ^ (xor)

- operano su interi
- operano su booleani,

Quanto detto vale anche per i operatori binari di and (&), or (|) e or esclusivo (^).

Queste operatori possono calcolare dei valori di verità e falsità. In generale vengono infatti utilizzati per combinare delle condizioni. Per esempio, per vedere se valore è entro i limiti di un intervallo, occorre combinare due condizioni: "a > MIN & a < MAX". La parte "a > MIN" è una operazione che ritorna un risultato booleano, e lo stesso fa la parte "a < MAX". Abbiamo quindi due valori booleani su cui si deve effettuare un calcolo.

La regola è che and (&) ritorna true solo se entrambi i valori sono true (altrimenti false), mentre or (|) ritorna false solo se entrambi i valori sono a false (altrimenti ritorna true). L'operatore di or esclusivo (^, xor) ritorna true se solo uno dei due valori operati è true, altrimenti ritorna false.

- and, or, ed exclusive-or bit-a-bit sui bit dei valori interi

Questi operatori operano anche su due numeri interi, ottenendo un nuovo numero di cui singoli bit sono il risultato delle operazioni booleane specifiche. Occorre quindi considerare ciascun bit come un valore booleano, dove 0 è false e 1 è true, ed effettuare i calcoli per ciascuno dei bit dei due numeri. Consideriamo un esempio in cui si opera su numeri di 4 bit (tenendo presente comunque che le operazioni sono sempre almeno su numeri di 32 bit). In questo esempio sono mostrati tutti i risultati possibili per ciascuno degli operatori.

```
0011 | 0101 == 0001  
0011 & 0101 == 0111  
0011 ^ 0101 == 0101
```

## ● Notare che f() & g() chiama f() e g() **sempre**

Una cosa che può sembrare ovvia, ma che conosce delle eccezioni, è il fatto che facendo un calcolo, prima di ottenere risultato finale si calcolano i risultati intermedi. Ripeto in generale e così, ma vedremo un caso in cui non succede questo. Con questi operatori si è nel caso generale. Quindi se per esempio faccio f() & g(), succede quanto segue: prima viene calcolato f() (quindi mentalmente chiamando il metodo corrispondente), poi viene calcolato g() (anche in questo caso chiamando il metodo corrispondente), e infine facendo il calcolo. Notare che, scritto in questo modo, vengono sempre comunque chiamati entrambi i metodi, compresi i loro eventuali effetti collaterali. A volte si vuole invece che solo uno dei due dei metodi venne chiamato, per evitare l'effetto collaterale del secondo.

## 4.13 Shift

questa lezione leggermente complessa, ma non richiede di essere capita a fondo anche perché tratta di operazioni che si usano abbastanza di rado.

### ● << >> >>>

#### ● operano su interi

gli operatori visti prima, di tipo logico, si applicano sia a interi che a booleani. Gli operatori che adesso vediamo invece si applicano esclusivamente a interi; questo perché hanno senso solamente sulla maschera di bit di un intero. Si tratta infatti degli operatori cosiddetti di shift, o slittamento. Queste operatori considerano un intero come una maschera di bit, che ruotano una maschera a destra o a sinistra di certo numero di posizioni. Anche in questo caso bisogna tenere conto della precisione: se stiamo operando su un intero, lo slittamento avviene per 32 bit, mentre se stiamo operando su un long, slittamento avviene per 64 bit. Facciamo due esempi ipotizzando che stiamo operando su soli quattro bit (anche se questo non è mai possibile). Tenere presente che non è possibile scrivere direttamente il numero in binario.

```
0101 >> 1 == 0010  
0011 << 2 == 1100
```

La slittamento a sinistra causa sempre inserimento di bit 0 in coda, e non presenta particolari difficoltà. Discorso cambia leggermente quanto si considerano slittamento a destra, in quanto coinvolge l'ultimo bit che è speciale, in quanto rappresenta il segno.

#### ● >> shift con estensione di segno

```
-2>>1 == -1
```

● utile quando si considera l'intero un valore con segno

#### ● >>> shift senza estensione di segno

```
-2>>>1 == 2147483647
```

● utile quando si considera l'intero una maschera di bit

Per capire i problemi occorre sapere alcune cose di come un numero viene trasformato in una maschera di bit. Consideriamo un semplice caso come 12. Che questo essere scritto come somma di potenza di due in questo modo:  $2^3 + 2^2$ , la sua rappresentazione in binario è 1100. In generale lo slittamento a sinistra di una posizione equivale a una moltiplicazione per due, mentre lo slittamento a destra di una posizione equivale ad una divisione per due.

Ma cosa succede quando il numero diventa negativo? Si usa una variazione fa in modo che l'ultimo bit rappresenti il segno. Per evitare di avere due zeri diversi (lo zero positivo e lo zero negativo) i numeri negativi hanno hanno una rappresentazione leggermente diversa. Non importa quale che sia questa rappresentazione, la cosa veramente importante è il fatto che i numeri negativi hanno l'ultimo bit a 1, mentre quelli positivi hanno l'ultimo bit a zero.

Il problema è che bisogna decidere se, quando si slitta, si deve inserire uno zero o qualcos'altro. Si vuole inserire sempre uno zero, si deve usare l'operatore `>>>`. Questo operatore però quando sono coinvolti numeri negativi causa effetti abbastanza inaspettati: siccome fa passare dalla rappresentazione dei numeri negativi a quella dei numeri positivi, il valore ne viene sconvolto. Per fortuna, quando si vuole far corrispondere lo slittamento a destra ad una divisione per due anche per i numeri negativi, si può utilizzare l'operatore `>>`. Questo operatore fa un'operazione che si chiama estensione del segno. Ovvero inseris di ce nel posto che si è liberato un bit uguale a quello che è uscito. Per le regole dell'aritmetica speciale dei numeri negativi valore corrisponde a divisione per due anche per i numeri negativi.

## 4.14 Negazione

- ! (not)
  - opera su booleani
  - inverte il valore true/false

La negazione, ovvero l'inversione del valore vero in falso e viceversa si ha con operatore !.

Questo operatore richiede necessariamente un valore booleano, che non fa confuso con l'operatore ~, che invece opera solamente valori interi.

## 4.15 And

### ● && (and)

#### ● opera su valori booleani

crediamo adesso il vero operatore di and, che si scrive con **&&**, ovvero la **&** viene ripetuta due volte. Questo operatore si distingue dall'operatore con un solo **&** per due motivi: il primo è che opera solamente su valori booleani (non si può utilizzare con gli interi).

#### ● comportamento *short - circuit*

- valore determinato se il primo operando è false
- in tal caso non viene valutato il secondo

Ma il secondo punto differenza, quello fondamentale, è il suo comportamento cosiddetto a corto circuito. Vediamo di capire esattamente che cosa si tratta. Se diciamo che vogliamo dare un osso a Socrate se Socrate è un cane e Socrate scodinzola, nel caso Socrate fosse un uomo, non ci preoccupiamo di controllare che sta scodinzolando (perché generalmente gli uomini non lo fanno). Il concetto è che in una congiunzione (and), perché la congiunzione sia vera è necessario che entrambi i predicati (ovvero le due condizioni) siano vere. Se la prima condizione è falsa, non importa quale sia il risultato della seconda: non è necessario nemmeno prenderla in considerazione: il risultato è comunque falso. Questo fatto viene sfruttato dall'operatore **&&** per arrestare il calcolo nel caso il primo operando è falso. Questo fatto è un dettaglio che può essere fondamentale per scrivere programmi che funzionano.

```
c != -1 && c = in.read()
```

- se c'è EOF (-1), non viene letto un altro carattere

Vediamo un esempio: supponiamo di essere in un ciclo che legge un carattere alla volta dall'input. Questo è un esempio realistico: per leggere da un file o da altre sorgenti si utilizza proprio un ciclo che legge caratteri da un flusso in ingresso. Ovviamente si deve continuare a letto di caratteri dall'input finché che stanno; quando sono finiti si deve interrompere il ciclo. Quindi dell'esempio del listato vediamo la condizione di controllo del ciclo, che si interrompe quando se è alla fine del file. Il trucco è che viene ritornato valore speciale -1 quando si è alla

fine del ciclo. Il cavillo cui bisogna stare attento è che se non si è alla fine del ciclo, si vuole continuare a leggere dall'input; ma se si è alla fine del ciclo, che vuole fermare, SENZA LEGGERE UN ALTRO CARATTERE. Infatti la lettura dell'input quando questo è finito, può causare spiacevoli effetti collaterali come eccezioni o altre situazione anomala. Sfruttando la caratteristica degli operatori booleani a corto circuito, notiamo che quando la prima condizione è falsa (ovvero siamo alla fine del ciclo) la non viene valutata, evitando così di leggere un carattere oltre la fine.

## 4.16 Or

### ● || (or)

#### ● comportamento *short - circuit*

- valore determinato se il primo operando è true
- in tal caso non viene valutato il secondo

la caratteristica a corto circuito è propria anche di un altro operatore booleano, `||`, il quale ha però un comportamento diverso. Infatti se dico che voglio dare un osso a Socrate se Socrate abbaia o Socrate scodinzola, è sufficiente che una sola delle due condizioni si è verificata perché io voglia fare quello che ho detto. Quindi se Socrate sta abbaiano, vado subito a dargli un osso senza controllare che stia pure scodinzolando. L'operatore a corto circuito `||` (notare che anche in questo caso è raddoppiato), se il primo operando è vero non valuta il secondo. Se il primo è falso invece utilizza il secondo operando per calcolare valore finale.

```
n < table.min() || n > table.max()
```

#### ● si risparmia un calcolo (pesante) del max

Il listato mostra un caso in cui questo è utile: supponiamo infatti che si voglia verificare se un dato numero è minore del minimo o maggiore del massimo di una tabella. Per riuscirci dobbiamo effettuare appunto il calcolo del minimo o del massimo. Supponendo che la tabella sia molto grande, che può volere molto tempo per riuscir ottenere il valore corretto. Ora se n è minore del minimo, sicuramente non può essere maggiore del massimo. Quindi, una volta che ho verificato che è vera la prima condizione, non ha bisogno di verificare la seconda. In questo modo si risparmia in alcuni casi un pesante calcolo del massimo (e quindi l'esecuzione ne giova, in quanto ci mette la metà).

## 4.17 Condizionale

### ● ? : operatore condizionale, ternario (l'unico)

un altro operatore interessante, ha anche la caratteristica di essere speciale in termini sintattici: si tratta infatti di un operatore "ternario". Questo operatore prende tre operandi. Per capire come funziona, consideriamo un operatore binario che viene scritto come "a + b". L'operatore ternario in questione viene scritto come "a ? b : c", dove il punto interrogativo e i due punti vanno sempre di pari passo, e vanno scritti entrambi in quest'ordine.

- il primo valore deve essere un valore booleano
  - gli altri due possono essere qualsiasi tipo
  - devono essere dello stesso tipo (o convertibili allo stesso tipo)

Questo operatore è chiamato operatore condizionale in quanto ritorna un valore oppure un altro valore a seconda di una condizione. Il primo operando è la condizione, di conseguenza deve essere una espressione che viene valutata ad un valore booleano. Il secondo e il terzo operando invece sono il risultato finale: se primo operando viene valutato a vero, si ottiene come risultato il secondo operando. Se il primo operando viene valutato falso, si ottiene come risultato terzo operando. È importante rendersi conto che il secondo e il terzo operando devono essere dello stesso tipo, o almeno convertibili allo stesso tipo. In particolare, una cosa che non posso assolutamente fare è ritornare un tipo primitivo come secondo operando, e un oggetto come terzo operando.

### ● comportamento *short - circuit*:

```
(a > b) ? f() : g()
```

- se a > b chiamo f() altrimenti chiamo g()

Anche questo operando ha un comportamento a corto circuito. Viene valutata la condizione: se la prima condizione è vera, allora viene valutata il secondo operando. Se la prima condizione è falsa, viene valutato il terzo operando. Nel listato è mostrato che se a è minore di b, allora chiamo f(), altrimenti chiamo g(). Solamente una delle due di f() o g() viene però effettivamente chiamata.

```
double sqrtmax(double a, double b) {  
    return a > b  
    ? Math.sqrt(a)  
    : Math.sqrt(b);  
}
```

● calcola la radice una volta sola

consideriamo questo esempio, che ritorna la radice quadrata del massimo tra a e b. In questo caso la radice viene calcolata una volta sola. In ogni caso si ha un risparmio di tempo di esecuzione.

## 4.18 Incremento

### ● Operatori di incremento e decremento

vediamo adesso degli operatori che potrebbero anche non esistere. Quello che fanno infatti si può ottenere usando altre operatori che abbiamo già visto. Il loro uso e la loro utilità si ha per la convenienza, in quanto consentono di fare in maniera sintetica delle operazioni molto frequenti nella programmazione, e in particolare nei cicli. Usandoli si può scrivere del codice compatto, e quando ci si prende l'occhio anche più leggibile. Alcuni programmatori considerano il linguaggio C (da cui deriva Java) illeggibile proprio per la presenza di queste operatori, ma in realtà, se usati appropriatamente, la aumentano (ovviamente gli occhi di chi conosce il C e il Java).

Classici del C:

● **++** incremento di 1

    -- decremento di 1

Gli operatori **++** e **--** sono operatori che effettuano un incremento o un decremento. Si tratta essenzialmente di operatori con effetti collaterali. Per esempio scrivere **++a** è (all'incirca) come scrivere **a = a + 1**, mentre scrivere **--a** è come scrivere **a = a - 1**. Si tratta però di operatori: ovvero dopo aver fatto il calcolo ritornano anche un valore. Se vi ricordate quando abbiamo parlato di **a = 1**, abbiamo detto che si tratta di un operatore che ritorna un valore (1) e causa un effetto collaterale (assegna uno ad **a**). Allo stesso modo **++a** ritorna un valore, che è a incrementato di 1, ma ha anche l'effetto collaterale di lasciare questo valore incrementato in **a**. Quindi tipo dire che **++a** incrementa **a** e ritorna il valore incrementato. Stesso discorso (mutatis mutandis) per **--a**.

● **operano su interi e float**

queste operatori possono essere applicati sostanzialmente a numeri, in particolare numeri interi e in virgola mobile.

● **Possono essere prefissi o postfissi:**

un'altra caratteristica di queste operatori è il fatto che in realtà sono quattro e non due. Il comportamento infatti è diverso a secondo se vengono messi prima o dopo il loro operando. Cioè se sono prefissi (++a --a) o postfissi (a++ a--). La differenza tra prefisso e postfisso è che nel primo caso viene prima effettuato l'incremento, e poi restituito il valore; nel secondo caso viene invece prima restituito il valore e poi effettuato l'incremento.

Prefisso	Postfisso
a = 1 ;	a = 1 ;
b = a++;	b = ++a;
Risultato:	Risultato:
b==1 e a==2	b==2 e a==2

Nella tabella è illustrato con un esempio quello che succede. Se un operatore è postfisso, alla fine b vale 1, perché ha preso il valore di a prima dell'incremento; se l'operatore è usato prefisso, alla fine a vale 2, perché prende il valore di a dopo l'incremento. In entrambi i casi alla fine a risulta incrementato e quindi vale 2.

## 4.19 Assegnamento

- $a = b$  è una espressione con effetti collaterali:

- modifica  $a$  e ritorna  $b$

- $a = b = c$  equivale a  $a = (b = c)$

abbiamo già discusso questa caratteristica delle espressioni di Java quando abbiamo discusso di espressioni in generale. Si tratta per fatto che l'operatore  $=$  è un operatore con effetti collaterali, che ritorna un valore e causa la modifica di uno degli operandi.

- $b=c$  modifica  $b$  e ritorna il valore di  $c$  che serve a modificare  $a$ .

- Il tutto vale  $c$ , valore scartato

scrivendo  $a = 1$  si assegna uno ad  $a$ , e si ritorna il valore 1, che può essere assegnato ad un'altra variabile. Alla fine di una serie di assegnamenti si ha comunque un valore di una espressione che però viene scartato.

- $a += b$

- equivale a  $a = a + b$

in Java sono presenti alcuni operatori che non solo effettuano un assegnamento, ma anche si occupano di effettuare una operazione. Il più semplice è  $+=$ . Scrivere  $a+=1$  può essere considerato una abbreviazione di  $a = a + 1$ .

- però  $a[f()] += b$  non equivale a  $a[f()] = a[f()] + b$

C'è una cosa a cui bisogna stare bene attenti: che l'espressione suddetta non è esattamente equivalente a come scritto. In realtà quello che succede è un pò più complesso. Diciamo che l'operatore accede alla zona di memoria dove è contenuto il valore, fa il calcolo e gli rimette dentro il valore incrementato in una sola passata. Se trascrivo letteralmente l'espressione illustrata nell'esempio (che è un pò più complessa), dovrei chiamare due volte la funzione  $f()$ . Questa funzione potrebbe avere effetti collaterali. Usando  $+=$  gli effetti collaterali avvengono una volta sola.

- altri operandi, stesso comportamento (cambia l'operatore di calcolo)

Tutto il discorso fatto per `+=` vale senza cambiamenti anche per gli altri operatori come `*=`, `/=` eccetera. Il comportamento è esattamente lo stesso, quello che cambia è solo l'operatore su cui viene fatto il calcolo. Nella prossima tabella sono riassunti i vari operatori e il loro uso in dettaglio.

Operatore	Significato
<code>--=</code>	sottrazione
<code>*=</code>	moltiplicazione
<code>/=</code>	divisione
<code>%=</code>	modulo
<code>&amp;=</code>	and
<code> =</code>	or
<code>^=</code>	or esclusivo
<code>&gt;&gt;=</code>	shift a destra
<code>&lt;&lt;=</code>	shift a sinistra con segno
<code>&lt;&lt;&lt;=</code>	shift a sinistra senza segno

Si può notare come siano presenti tutti gli operatori che hanno senso. Per esempio abbiamo `&=` che fa l'and binario del contenuto di una variabile con una maschera di bit. Non è invece presente un ipotetico `&&=`, anche perchè, avendo un comportamento a corto circuito, non è possibile tradurlo in maniera sensata con un operatore di assegnamento.

Questa sezione ci permette di esaminare la parte più tradizionale della programmazione e forse quella più nota. Le espressioni sono in un certo senso il "sale", ma il "pane" della programmazione sono forse i comandi. Un programmatore solitamente passa il suo tempo a spiegare al computer come fare a produrre certi risultati.

In verità vedremo (e dovremo renderci conto) che la vera difficoltà nella programmazione Java non è scrivere i comandi (anche se questa è una cosa che bisogna sapere fare), ma progettare gli oggetti.

## ● Categorie di comandi di Java

Per semplicità dividiamo i comandi di Java in alcune categorie che i permetteranno di organizzare il discorso.

### ● comandi semplici e blocchi

Abbiamo visto le espressioni, che comprendono anche modifiche alle variabili (cosa che in altri linguaggi è prerogativa dei comandi). In realtà vedremo che è molto facile trasformare una espressione in un comando "semplice" (non strutturato). Questo tipo di comandi è il mattone ultimo dei programmi in Java. Allo stesso modo, abbiamo modo di raggruppare più comandi in uno solo creando un blocco.

### ● comandi condizionali: **if - else, switch**

Abbiamo comandi che ci permettono di effettuare delle scelte, in base ad un a condizione booleana oppure numerico.

### ● comandi di ciclo: **while, do - while, for**

Abbiamo comandi che ci permettono di effettuare cicli , con dettagli variabili a seconda di inizializzazioni e condizioni di uscita.

### ● comandi di interruzione di ciclo: **break, continue**

Legati ai comandi di ciclo abbiamo comandi che ci permettono di alterare in vario modo la loro esecuzione.

### ● ritorno di valori: **return**

Le classi definiscono i metodi. I metodi sono come funzioni e come tali ritornano valori, utilizzando l'apposito comando.

### ● gestione eccezioni: **try - catch - finally**

La gestione di errori è una categoria a parte, a cui Java deriva appositi costrutti. Le condizioni di errore si chiamano eccezioni, e abbiamo comandi per lanciarle e catturarle. In verità tratteremo questa parte più avanti, solamente dopo aver visto l'ereditarietà.

# 5 Comandi

## ● Comandi semplici e Blocchi

Con questi costrutti segniamo il confine tra le espressioni e i comandi.

### ● Se si ha una espressione $a = 1$

Consideriamo una espressione. In matematica, una espressione non ha "stato", ovvero non causa modifiche ad una "memoria", concetto che non esiste. In matematica una espressione è come un filtro, che prende dei valori in uscita e ritorna una valore di ritorno. In informatica le cose stanno diversamente. Abbiamo che le informazioni vengono sempre memorizzate in una cosa che si chiama "memoria". Le variabili consenteono di accedere alla memoria, e non solo per prendere valori. Alcuni operatori non sono prendono valori dalla memoria per effettuare un coalcolo, ma sono in grado di scrivervi. Abbiamo già visto questi operatori: sono l'= $, il += e simili, il ++ e il --.$

### ● aggiungendo un ' ; ' diventa un comando (semplice)

```
a = 1;
```

Quindi, possiamo trattare queste "espressioni con effetti collaterali" come strumenti per effettuare comandi. Ricordiamo che un comando per definizione modifica la memoria. In un linguaggio imperativo, come java, si lavora con una serie di modifiche della memoria che vengono fatte seguendo cicli e condizioni. Per trasformare una espressione con effetti collaterali in un comando è sufficiente mettere alla fine un punto e virgola. Bisogna stare attenti che ha senso utilizzare una espressione in questo modo solo se alla fine risulta in una qualche modifica della memoria. Il compilatore controlla che i comandi semplici alla fine risultino in un valore che viene memorizzato, e avverte se non è così. Se uno scrive  $a+1;$  si ritrova nell'effettuare un calcolo che non viene memorizzato da nessuna parte e quindi è inutile ai fini della programmazione. In questo caso il compilatore avverte del fatto che si è fatta una operazione inutile.

### ● Se si hanno due comandi



Il tuo sito è sempre uguale ? **EpNuke** la soluzione modulare professionale per il Web Publishing.  
Soluzioni **tutto compreso** con migrazione della grafica, hosting e assistenza.

Per informazioni [www.eprometeus.com](http://www.eprometeus.com) - 68/225

```
a=1;  
b=2;
```

I comandi in Java (semplici e, come vedremo, quelli composti) vengono eseguiti uno dopo l'altro. Non ci sono particolari regole per specificare una sequenza di comandi. Basta scriverli uno dopo l'altro. Quelli che si osservano in figura sono semplicemente due comandi diversi.

- racchiudendoli tra graffe diventano un solo comando (blocco)

```
{  
    a=1;  
    b=2;  
}
```

Ci sono casi in cui due comandi non vanno bene: devono essere uno solo. Infatti i comandi composti sono per definizione "composti" da un altro comando. Uno solo però. Per questo motivo esiste un costrutto che serve a trasformare due comandi in uno solo, che è il blocco. Basta racchiudere due comandi tra graffe per avere quello che, sintatticamente, è un solo comando..

- Notare che il punto e virgola ci vuole sempre

Come osservazione notiamo come sia sempre necessario mettere un punto e virgola alla fine di ogni comando (perchè è quello che trasforma una espressione in un comando) e poi inserire il tutto tra graffe.

- I blocchi delimitano la "portata" delle variabili locali

Il blocco ha anche un'altra funzione: serve a definire la "portata" delle variabili. Vediamo di capire di che cosa si tratta. Innanzitutto abbiamo visto che abbiamo tre tipi di variabili: parametri, variabili locali e campi. Le variabili locali sono abbastanza "persistenti", di questo lo vedremo più in dettaglio nella parte relativa alle classi. I parametri vengono definiti all'inizio di un metodo e finiscono alla fine del metodo stesso. Finiscono significa che spariscano, vengono eliminate.

```
01. int a;  
02. { int b;  
03. char a; // copre l'altra a  
04. }  
05. char a; // errore
```

## ● una variabile può essere ridichiarata in un blocco

Quello che a volte non è totalmente chiaro è il fatto che le variabili locali possono essere dichiarate dentro un blocco e terminano la loro vita alla fine di quel blocco. Una variabile deve essere unica in un blocco ma può essere ridichiarata in un altro blocco più interno. Nella figura abbiamo la variabile a nella riga 01, Aprendo un blocco possiamo ridichiararla, anche con un altro tipo, come si vede in riga 03. Questa variabile è viva fino alla chiusura del blocco che la contiene. Appena si raggiunge la riga 04, spariscono tutte le variabili contenute nel blocco. Notare che la a dentro il blocco copre la a fuori il blocco che non è più utilizzabile dentro il blocco. La dichiarazione in riga 05 è un errore: mentre si può ridichiarare la variabile a nel blocco più interno, non si può nel blocco più esterno, perché è stata già dichiarata ed è ammessa una sola dichiarazione per blocco.

## 5.1 if

### ● Condizionale if

```
if(<condizione>
    <comando>
else
    <comando>
```

Il primo comando strutturato che vediamo è il comando if. La caratteristica di questo comando è quello di essere un comando composto. Questo è solo il primo dei comandi composti che vedremo più avanti nel corso di questo corso. Per comando strutturato si intende un comando che contiene di se un altro sottocomando. Esaminando il listato si vede come funziona. C'è una parte, if() che è la parte principale del comando. Sintatticamente questo comando è incompleto in quanto occorre specificare delle altre parti. Tra parentesi va posta una espressione, mentre dopo il comando va posto un altro comando. Delle espressioni abbiamo già parlato in abbondanza nella prima parte del corso. Il sottocomando invece è un qualsiasi comando, tra quelli già visti e quelli che vedremo. Nel caso più semplice il sottocomando di if può essere un comando semplice. Quindi if(a>1) b=2; è una if completa. Dopo la if può esserci un comando composto, quindi in questo caso possiamo scrivere if(a>1) { b=2; c=3; }. In questo caso si vede chiaramente l'utilità delle graffe. Due comandi vengono raggruppati in uno solo per essere il sottocomando della if. Dopo la if può esserci al più un comando e quindi si usano le graffe per raggruppare.

#### ● la <condizione> deve essere una espressione booleana

Il comportamento del comando è quello di eseguire il sottocomando solo se la condizione è verificata. La condizione deve essere una espressione booleana. Cioè dopo aver effettuato i calcoli il risultato deve essere vero o falso. Tipicamente questo si ottiene con espressioni come a==b oppure a > b && c > d. Si può notare infatti come gli operatori relazionali ritornino appunto vero o falso, e combinando vero o falso si ottiene sempre un risultato booleano.

#### ● l'else lega l'if più vicino

L'if e basta esegue un comando solo se una condizione è verificata. È comunissimo voler fare qualcosa anche nel caso una condizione non è verificata. Per questo motivo l'if ammette una variante più complessa in cui esiste una parte eseguita quando la condizione è falsa. In questo caso è presente la parte else, in cui viene specificato anche un comando. Questo comando viene eseguito solo quando la condizione è false.

● **In caso di ambiguità si devono usare le graffe esplicitamente**

```
// 1 if(<condizione>) // 2 if(<condizione>) <comando> // riferito a 2 else <comando>  
// 3 if(<condizione>) { // 4 if(<condizione>) <comando> } // riferito a 3 else <comando>
```

Il comando di if o else può essere a sua volta un altro if o else. Si può verificare il caso in cui un else può essere riferito a due diversi if, come nell'esempio in figura. La regola è che l'else lega l'if più vicino. Questo è valido anche se il programmatore indenta (ovvero mette degli spazi) per indicare una diversa associazione. Se si vuole forzare una associazione in cui l'else lega con l'if più lontano, bisogna "staccare" l'if mediante l'uso esplicito delle parentesi graffe.

## 5.2 switch

### ● Condizionale switch

#### ● La condizione è un intero

L'altro condizionale di cui dobbiamo parlare è lo switch. Si tratta di un comando che seleziona un comando da eseguire in base a un numero interno e non più, come l'if, in base a un booleano.

Vediamo la sintassi:

```
switch (<condizione>) {  
    case <val1>:  
    case <val2>:  
        <comando>  
        break;  
    case <val>:  
        ...  
        break;  
    default:  
        ...  
        break;  
}
```

C'è da dire molte cose. Innanzitutto che tra le graffe bisogna racchiudere una serie di comandi. Si tratta di comandi come tutti gli altri. L'unica particolarità è che alcuni di questi comandi possono essere etichettati con un 'casè'. Precisiamo meglio che intendo dire per etichettati. Esiste il comando composto "case <valore>: <comando>".

Considerando un comando come " a=1;" allora "case '2': a=1;" è un comando etichettato. Queste etichette possono essere "impilate", nel senso che posso fare "case '1': casè2: a=1;". Notare attentamente che il valore del case deve essere un valore fisso. Ovvero può essere un numero esplicito. Può anche essere una espressione, purchè possa essere calcolata una volta e per tutte. Non può contenere variabili a meno che non siano costanti: cioè può contenere per esempio MAX solo se MAX è un campo finale statico. Il motivo di questa restrizione è legato al fatto che in realtà lo switch, per ragioni di efficienza, viene internamente preparato per l'esecuzione, e non

viene fatta una ricerca dei case ma una solta di "salto calcolato" in base al valore. Per poter preparare il calcolo bisogna avere tutti gli elementi e questo è il motivo principale perchè si impone che il valore dei case venga definito una volta e per tutte, e non dipenda da variabili calcolate durante l'esecuzione.

● valutata la <condizione> si salta al primo case corrispondente

Una volta che abbiamo costruito il nostro switch correttamente è il momento di farlo funzionare. L'espressione che si trova tra parentesi nella testa dello switch deve essere intera. Attenzione, non può essere un double e, curiosamente, non può essere nemmeno un long. Una volta calcolato il valore dell'espressione, si va alla ricerca di un case che ha un valore corrispondente. L'esecuzione salta quindi a questa etichetta e continua da lì, eseguendo i comandi successivi.

● Se nessuno dei case è soddisfatto si salta al default

Nel caso in cui il valore calcolato nell'espressione non corrisponda a nessun caso reale, ovvero non ci siano etichette con un valore corrispondente, si salta, se c'è, all'etichetta speciale default, che non prende valori.

## 5.3 fall-through

### ● Attenzione: eseguito il salto, il flusso prosegue

- Non si esce dallo switch al case successivo.

Abbiamo detto una cosa che va ribadita meglio: l'esecuzione una volta che si raggiunge un comando etichettato, prosegue. Chi magari viene da altri linguaggi può essere porato a pensare che se nel corso dell'esecuzione si incontra un altro case, si vada alla fine dello switch. Così non è. Invece, si esegue pure il comando legato al case.

- Utile per impilare più case.

La ragione di questo strano comportamento è per consentire di "impilare" più case. Quindi posso scrivere "case '1': case '2': case '3': a=b;" per assegnare b ad a ogni volta che calcolo 1, 2 o 3.

### ● Se si incontra un *break* si va alla fine dello switch

Comunque se voglio terminare un case, posso farlo, ma lo devo fare esplicitamente, ponendo un comando break. Questo comando effettivamente mi fa saltare alla fine dello switch. Su richiesta del programmatore.

### ● Esempio di switch

```
n=0;
switch (c) {
    case 1:
    case 2:
        n += 1;
    case 3:
        n += 2;
        break;
    default:
        n += 4;
        break;
}
```

Vediamo un esempio concreto in cui possiamo fare mente locale e concentrarci su quello che effettivamente succede.

- Se all'inizio c vale 1 o 2, n alla fine vale 3

Se  $c$  vale 1 o 2 si salta a  $n+=1$ , ma a questo punto, non essendoci niente, si prosegue eseguendo  $n+=2$ . A questo punto si incontra effettivamente un break e si termina, ottenendo  $0+1+2=3$ .

● **Se all'inizio  $c$  vale 3 ,  $n$  alla fine vale 2**

Nel caso in cui invece abbiamo un valore 3, si salta direttamente a  $n+=2$  e poi alla fine del ciclo, immediatamente. Risultato: 2.

● **Se all'inizio  $c$  vale 4,  $n$  alla fine vale 4**

Un valore diverso dai precedenti fa andare al default, che somma 4 a 0, ottentendo 4 come risultato.

## 5.4 while

Questo comando è uno dei più semplici ma anche uno dei più importanti. Il ciclo, dopo l'if, è il costrutto fondamentale della programmazione strutturata. Java ha anche costrutti per la programmazione orientata agli oggetti che vedremo dopo, ma in questo momento ci limitiamo al sottinsieme di programmazione strutturata del linguaggio.

### ● Comando di ciclo while

```
while (<condizione>
      <comando>
```

Si tratta di un comando composto, come if, per cui nel corpo del comando deve esserci un singolo comando. Al solito, questo singolo comando può essere in realtà molti comandi, a patto che vengano raggruppati in un blocco usando le graffe.

#### ● La condizione deve essere una espressione booleana

Anche il while ha la restrizione sul tipo dell'espressione di controllo: deve essere booleana. Non intera, e nemmeno in virgola mobile, ne tantomeno un oggetto. Deve risultare vera o falsa.

#### ● Il comando viene eseguito finchè è vera la condizione

La logica del while è che si valuta l'espressione booleana, e a quel punto si esegue il comando. A questo punto si rivaluta l'espressione booleana e si riesegue il comando. A meno che non si volgia effettivamente che il ciclo sia infinito (caso che può succedere) presumibilmente il comando eseguito farà qualcosa che, preso o tardi farà sì che l'espressione diventi falsa.

#### ● il comando può anche essere eseguito 0 volte

Notare attentamente che l'espressione del while viene subito calcolata. Se risulta immediatamente falsa, il discorso si chiude subito lì: infatti è possibile che il comando del while non viene mai eseguito.

## 5.5 do-while

### ● Comando di ciclo do-while

```
do  
  <comando>  
  while (<condizione>) ;
```

L'altro comando, simmetrico al while, è il do while. La sintass prevede che la condizione venga messa in fondo. Il comando che contiene va messo dopo il do.

#### ● La condizione deve essere una espressione boolena

Anche in questo caso, l'espressione di controllo deve essere una espressione booleana.

- Il comando viene eseguito finchè è vera la condizione
- il comando verrà sempre eseguito almeno una volta

Il do-while si differenzia dal while per l'ordine in cui esegue le operazioni previste: infatti per primo si esegue il comando, poi si valuta la condizione. La posizione di comando posto alla fine è voluta perchè evidenzia il fatto che il comando viene eseguito sempre almeno una volta, e solo dopo la prima esecuzione si valuta per la prima volta la condizione. Di conseguenza si usa questo comando (più raramente del while) quando si vuole comunque fare qualcosa, e solo dopo vedere se occorre riprovare. Per esempio, aprire un file. Se non è stato aperto magari perchè qualcun altro lo sta usando, aspettare un pò e poi riprovare.

## 5.6 for

### ● Comando di ciclo for

#### ● Simile al while

Il comando di ciclo for è in pratica una variante del ciclo while usata solitamente per cicli di lunghezza determinata. La sintassi è la seguente:

```
for (<inizio>;  
     <condizione>;  
     <incremento>)  
     <comando>
```

Si nota che comprende tre diverse espressioni: una di inizializzazione, una di incremento e una di condizione. Vediamo esattamente cosa fanno e perchè.

#### ● Per prima cosa viene eseguita l'espressione di *<inizio>*

L'espressione iniziale viene valutata una volta sola, all'inizio del ciclo. In realtà si tratta di una espressione speciale perchè può anche essere una dichiarazione. Ovvero posso scrivere `for(int i=0; ... ; ...)`. La cosa curiosa è che la variabile `i` dichiarata "vive" fino alla fine del blocco che segue il `for`.

Un'altra cosa curiosa a proposito dell'inizializzazione del `for` è il fatto che posso inizializzare in serie più di una variabile. È sufficiente cioè che scrivo una serie di dichiarazioni, separandole da virgole. È legale quindi `for(int i=0, int j=1, int k=2; ... ; ...)`. Si tratta comunque di un caso particolare speciale. In C esiste l'operatore `,` che invece non c'è in Java. Solo nell'inizializzazione del ciclo `for` (e in pochi altri casi) si può usare la virgola, mentre in C si può usare in ogni espressione.

#### ● Poi viene valutata la *<condizione>*

La condizione successiva al ciclo `for` è invece una condizione in senso stretto. Deve essere una espressione strettamente booleana e non sono ammesse variazioni.

Questa espressione viene valutata dopo aver effettuato l'inizializzazione ma prima di eseguire il comando che segue il for. È l'espressione di controllo del ciclo ed è del tutto analoga all'espressione di controllo di ciclo del ciclo while. In particolare vale la regola che l'espressione di controllo del ciclo può essere immediatamente falsa e quindi il corpo del ciclo non venire mai eseguito. L'inizializzazione viene però sempre comunque eseguita.

- Viene eseguito il <comando>

Dopo che sono stati fatti i preliminari, si esegue il corpo del ciclo. Una volta che termina il comando non è però finita lì.

- Viene eseguita l'espressione di <incremento>

L'espressione di incremento viene eseguita DOPO che è stato eseguito il corpo del ciclo. E può succedere che non venga eseguita mai. Questo nel caso l'espressione risulti falsa. L'espressione di incremento viene comunque eseguita dopo il corpo, ma viene posta nell'intestazione del ciclo per raggruppare concettualmente tutti i passi che servono per eseguire un ciclo. A questo punto possiamo considerare un caso complessivo:

```
int s=0; for(int i=1; i<=10; i++) s+=i;
```

In questo caso stiamo sommando i numeri da 1 a 10. Il ciclo va da 1 a 10. La prima volta si crea i, si pone a uno, e si controlla se per caso non sia già arrivata a 10. A questo punto si esegue il corpo del ciclo, ovvero `s+=i`, e DOPO l'incremento. Gli operatori `++`, `+=` e simili sono stati creati essenzialmente proprio per coprire questo caso, usandoli nell'espressione di incremento del ciclo, dove normalmente si lavora con incrementi e assegnazioni alla stessa variabile.

Notare infine che, sapendo che la variabile di controllo "muore" alla fine del ciclo, ho dichiarato s prima del ciclo, in modo che sia ancora esistente dopo (e quindi ne posso usare il valore da qualche parte).

- Simile (ma non uguale) a:



```
<inizio>;
while(<condizione>)  {
    <comando>;
    <incremento>;
}
```

Il ciclo for può essere riscritto in termini di un ciclo while, come mostrato in figura. Si comporta in maniera molto simile ma non uguale al 100%. Una prima differenza sta nel fatto che nel ciclo scritto come un while la variabile di controllo del ciclo esiste ancora dopo il ciclo, mentre con il for questo non accade. La seconda differenza è più sottile e ha a che fare con le interruzioni di ciclo che vediamo nel prossimo paragrafo.

## 5.7 interruzioni

### ● Interruzioni di ciclo

In Java non esiste goto. Il goto serve a effettuare salti nel flusso del programma ed era la regola nei linguaggi prima della programmazione strutturata. Il C è un linguaggio strutturato, ma prevede il goto. Viene usato però normalmente solo in casi eccezionali (a meno che il programmatore non sia riuscito a digerire i concetti della programmazione strutturata!). I casi in cui si usa necessariamente il goto in C sono interruzioni di cicli e gestione di errori. Per i secondi abbiamo le eccezioni, che vedremo in seguito. Per i primi la soluzione sono i comandi di interruzione di ciclo, che come vedremo, fanno delle cose in più rispetto al C.

break

### ● Consente di interrompere cicli e il comando switch

Il break è il comando dell'interruzione del ciclo. Dove c'è un break il ciclo termina. Lo abbiamo già visto usato per terminare uno switch, ma lo possiamo applicare anche a for, while e do-while. Nel punto in cui si incontra un break, si ha immediatamente un salto alla fine del ciclo. Implicitamente pertanto il break è appunto un goto alla fine del ciclo che lo contiene.

### ● Reiterazione di ciclo

continue

Supponiamo di stare facendo un ciclo. Supponiamo che a un certo punto mi rendo conto che per questa iterazione non c'è più niente da fare, e voglio andare subito alla prossima iterazione. Posso senz'altro risolvere il problema con una if, ma spesso non rende l'idea e rende il codice meno leggibile a causa dell'annidamento di blocchi dentro blocchi.

### ● Consente di proseguire cicli dal punto in cui si è

Java mette a disposizione il continue per gestire elegantemente questo caso: consente di passare direttamente alla prossima iterazione del ciclo.



Il continue può essere usato solamente all'interno di un ciclo: while, for, do/while. Equivale anche in questo caso ad un salto, un goto implicito, ma questa volta all'inizio del ciclo, per eseguire l'iterazione successiva.

Bisogna stare attenti che è qui che il comportamento del continue differenzia while e for. Consideriamo infatti:

```
for(int i=0; i<10; i++) {  
    ...  
    if(<condizione>)  
        continue;  
    ...  
}  
int i=0;  
while(i<10) {  
    ...  
    if(<condizione>)  
        continue;  
    i++;  
    ...  
}
```

Nel primo caso, ciclo for, dopo il continue viene eseguita la successiva istruzione di incremento del ciclo (i++). Nel secondo caso, essendo posa all'interno del corpo del comando del while, la continue non fa eseguire l'istruzione di incremento del ciclo, i++.

## ● Sono dei "goto"

- Il break equivale ad un salto alla fine del ciclo
- il continue equivale ad un salto all'inizio del ciclo

```
// 1 inizio ciclo  
while(...) {  
    if(...)  
        break;      // salta a 2  
    if(...)  
        continue;   // salta a 1  
} // 2 fine ciclo
```

## 5.8 label

Fin qui abbiamo visto che continue e break servono a interrompere o proseguire il ciclo che le contiene. In questo Java non è diverso dal C. È comunque possibile interrompere più cicli annidati. Vediamo come.

### ● All'inizio dei cicli si possono apporre *label*

Quando abbiamo parlato dei case, abbiamo detto che si tratta di prefissi apposti ad un comando. In generale io posso etichettare qualsiasi comando, non solo dentro switch e non solo usando case.

### ● Una label è un identificatore seguito da due punti

Se ho per esempio "a=1;" posso scrivere "prova: a=1;" Per etichetta posso usare qualsiasi identificatore: una sequenza di lettere e numeri che comincia per una lettera. In C le etichette possono andare ovunque perché servono per marcare i "bersagli" del goto. In Java, non essendoci goto, hanno senso solo quando vengono apposti a cicli: for, while, do/while.

### ● break e continue possono avere come parametro una label

In Java infatti è possibile usare le etichette come destinazione di un break o di un continue. Ovvero dopo break o continue posso specificare una etichetta, una di quelle che ho usato all'interno del programma, per marcare un ciclo.

### ● Uscire o continuare cicli annidati

Che succede quando specifico che un break è seguito da una etichetta? Succede che il ciclo etichettato viene interrotto o continua, a seconda che si sia usato break o continue. Vediamo un esempio concreto.

### ● Esempio di ricerca in più file

```
loop:  
    while(<ci-sono-file>) {  
        while (<ci-sono-righe>) {  
            if(<non-c'è-in-questo-file>)  
                continue loop;  
            if(<trovato>)  
                break loop;  
        }  
    }
```

Supponiamo di stare eseguendo una ricerca di qualcosa in più file di testo. Dovremo pertanto fare un ciclo che elenca tutti i file su cui fare la ricerca. Per ogni file dovremo fare un altro ciclo in cui elenchiamo tutte le righe contenute. Abbiamo quindi due cicli annidati, quello esterno che enumera i file, e quello interno che esamina le righe di un file. Ora possono succedere due casi. Il primo in cui abbiamo trovato la stringa che cercavamo. In tal caso dobbiamo interrompere due cicli: quello che ci permette di scandire le righe e quello che scandisce i file. Un break non basta perchè interromperebbe solamente il ciclo di scansione delle righe del file, e infatti abbiamo scritto nell'esempio 'break loop'.

L'altro caso di interesse è quello in cui abbiamo scoperto che nel file che stiamo esaminando non c'è quello che cerchiamo (per esempio perchè abbiamo letto l'intestazione e abbiamo scoperto che le date non sono quelle necessarie). In tal caso dobbiamo andare al file successivo, non alla riga successiva, e quindi dobbiamo continuare il ciclo più esterno. infatti abbiamo usato 'continue loop'.

## 5.9 return

### ● Ritorna un valore

#### ● **return <espressione>**

In una classe Java abbiamo dei metodi. I metodi sono simili alle funzioni matematiche, in particolare prendono dei valori e ritornano dei risultati. A differenza delle funzioni però, dove il calcolo viene fatto solo con espressioni, i metodi fanno il calcolo eseguendo una serie di operazioni. Ricordiamo il fatto che le operazioni eseguite comportano la modifica della memoria, e che i risultati dei calcoli vengono memorizzati in variabili. Per fare sì che un metodo ritorni un valore occorre dare un comando esplicito. Questo comando è appunto `return`. Se si vuole che il metodo ritorni un determinato valore, calcolato con una espressione o memorizzato in una variabile, occorre usare `return` seguito dall'espressione che calcola il valore da ritornare. `Return` ha anche un'altra importante funzione: termina l'esecuzione del metodo ritornando al chiamante.

#### ● Deve essere compatibile con la dichiarazione

```
int f() {  
    return 1;  
}  
Stack g() {  
    return new Stack();  
}  
void h() {  
    return;  
}
```

L'espressione che segue il `return` non può essere qualsiasi: deve essere una espressione di un certo tipo, compatibile con la dichiarazione del metodo. Nel listato possiamo notare che abbiamo tre diversi metodi, ciascuno con un diverso tipo ritornato. Nel primo caso un tipo primitivo, nel secondo un oggetto e nel terzo niente. Il terzo caso è speciale: usare `void` significa dire che il metodo non ritorna valore(e quindi non può essere invocato in una espressione che richiede un valore come un assegnamento: `x=h()` è un errore).

Si noterà che nel corpo del metodo l'espressione che segue da un risultato di tipo "compatibile" con quello dichiarato dal metodo. Infatti se il metodo ritorna un intero, abbiamo una espressione intera, se il metodo ritorna un oggetto, ne costruiamo uno del tipo richiesto, e se invece il metodo ritorna void, dopo il return non specifichiamo nessuna espressione. Da notare semplicemente che vengono fatte le solite conversioni. Se dichiaro un metodo ritornare double, anche se scrivo una espressione intera non c'è problema perchè questa viene promossa a double. Come pure, se il metodo ritorna uno Stack, sarà possibile ritornare un oggetto di qualsiasi classe derivata. Infatti, come vedremo abbondantemente in seguito, una classe è sostituibile con tutte le classi derivate. Ovvero "un pesce rosso è anche un pesce". Ma riprenderemo il discorso.

## 5.10 eccezioni

### ● Molte operazioni possono generare "eccezioni"

- Le eccezioni si propagano
- Se non gestite causano la terminazione del programma con errore

Nei programmi ogni tanto avvengono delle situazioni inaspettate. Per esempio, si cerca di aprire un file che non c'è, oppure si cerca di creare un nuovo oggetto e la memoria è finita. Sono situazioni in un certo senso impreviste, che spesso i programmatore gestiscono con superficialità. Quando si programma infatti generalmente si lavora in un ambiente ovattato e controllato, in cui il programmatore ha già previsto gran parte delle condizioni necessarie al funzionamento del programma. Solitamente infatti l'ambiente di esecuzione è stato preparato a mano appositamente, e quindi in sviluppo va tutto bene. Ma il programma che va "nel mondo reale", è soggetto a trovare condizioni impreviste che lo fanno andare in errore.

In altri linguaggi di programmazione, la condizione di errore è gestita come tutti gli altri casi.

### ● Trattate in maggior dettaglio con l'ereditarietà

### ● Come ignorarle (per ora!)

- stampando informazioni su dove è avvenuta l'eccezione

```
try {  
    <comando-con-eccezione>  
} catch (Exception ex) {  
    ex.printStackTrace();  
}
```

### ● Come propagare le eccezioni non gestibili

Il programma si interromperà più facilmente in fase di messa a punto ma risulterà più robusto alle condizioni di errore più avanti.

### ● Aggiungendo throws Exception nella dichiarazione del metodo.

Quando si dichiara un metodo che può sollevare eccezioni (generalmente chiamando altri metodi), si può "lasciar passare" le eccezioni

```
class C {  
    void method(int x) throws Exception {  
        <comando-che-genera-eccezione>  
    }  
}
```

# 6 Stringhe/Array

## ● Trattiamo gli elementi a metà tra oggetti e tipi primitivi di Java

Vediamo adesso due importanti tipi di dato che sono a metà tra i tipi primitivi e i oggetti. In realtà strettamente parlando si tratta di classi, ma che per il loro uso particolarmente frequente e intenso sono stati promossi, e hanno ottenuto alcune caratteristiche dei tipi primitivi.

### ● Gli *array*, sequenze (di tipi di dato)

Innanzitutto consideriamo le schiere (o array) ovvero il tipo di dato derivato che rappresenta una sequenza di elementi tutti nello stesso tipo. È importante osservare che una schiera, una volta creata, ha un numero di elementi fisso, e che gli elementi contenuti sono tutti del nello stesso tipo.

Un'altra cosa importante è il fatto che c'è differenza tra una schiera di interi e un intero. Curiosamente l'intero è un tipo primitivo mentre la schiera di interi è un tipo derivato, ovvero una classe.

### ● Le *stringhe*, sequenze di caratteri

L'altra importante classe che ha caratteristiche da tipo primitivo è la stringa. Una stringa è una sequenza di caratteri. Una stringa può contenere solamente caratteri, e una delle sue caratteristiche principali è il fatto che è immutabile.

Dire che le stringhe sono immutabili significa dire che, una volta che è stata creata una istanza, non è possibile modificarla. È possibile creare un'altra stringa che contiene caratteri diversi, ma la classe stringa non dispone di metodi che consentono la modifica dei caratteri in essa contenuti.

## ● Le Stringhe e gli Array sono a tutti gli effetti degli oggetti.

A scanso di equivoci precisiamo che sia le stringhe che le schiere sono a tutti gli effetti degli oggetti, e si comportano come tali. In particolare una variabile stringa o schiera può contenere solamente un riferimento a una stringa o una schiera, ma non contiene completamente l'oggetto. Analogamente vedremo che entrambi tipi di dato hanno sia campi che metodi.

- Hanno una **sintassi speciale** per la costruzione: simili a costanti.
- Hanno una **sintassi speciale** per alcuni metodi: simili a operatori.

Vediamo adesso le differenze che rendono queste classi un pò speciali. In generale le istanze delle classi si creano chiamando costruttori e invocando metodi. Sia stringhe che schiere hanno particolari costrutti sintattici, per cui possono essere create con delle costanti e utilizzate con degli operatori.

- Si tratta di *zucchero sintattico*

La sintassi speciale di stringhe e schiere può essere considerata esclusivamente "zucchero sintattico". Si tratta cioè di abbreviazioni, che permettono di scrivere in maniera più rapide e concisa operazioni che altrimenti richiederebbero una sintassi ben più lunga da scrivere.

## 6.1 Array

### ● Gli array sono in sé degli oggetti.

- Per ogni tipo (primitivo o class) si può dichiarare un array di quel tipo

La schiera in sè non esiste. Esista la schiera di interi, la schiera di Stringhe, la schiera di navi, eccetera. Una schiera è una classe che viene ottenuto come tipo derivato, a partire da un altro tipo primitivo o classe. In entrambi i casi si ottiene una nuova classe. Da un intero si ottiene il tipo schiera di interi, da una stringa si ottiene il tipo schiera di stringhe. La stessa schiera è un tipo "schierabile", per cui da una schiera di interi è possibile ottenere tipo "schiera di schiera di interi".

### ● Varie sintassi per dichiararli

Ci sono varie sintassi che consentono di dichiarare le schiere, come possiamo vedere nel seguente listato:

```
int[] ax;           // array di interi
String ax[];        // array di stringhe
Object[] ay[];      // array di Object
int[][] aay;        // array di array di interi
```

### ● Vengono dichiarati usando [ ] dopo il tipo della classe,

Il modo più semplice per specificare che un tipo è una schiera di un altro tipo, è quello di aggiungere, dopo il nome del tipo, due parentesi quadre.

### ● Oppure mettendo [ ] dopo il nome della variable, o dopo tutti e due.

In realtà Java è un linguaggio che deriva la sua sintassi dal linguaggio C. Per ragioni storiche è ammessa anche una sintassi che prevede le parentesi quadre vengano specificate dopo il nome della variabile (e non dopo il nome del tipo). Per completezza, è anche possibile utilizzare entrambe le sintassi, ovvero le parentesi quadre si possono specificare sia dopo il nome del tipo che dopo il nome della variabile.

## ● Essendo gli array oggetti, ci possono essere array di array.

Come si è detto le schiere sono un tipo derivato da cui si può derivare un altro tipo, ottenendo una schiera di schiere. Quindi per specificare una schiera di schiere di interi, si devono usare due coppie di parentesi quadre. Si consiglia di usare la sintassi più semplice ovvero la prima, che è anche quella che è più in linea con lo stile di Java per la dichiarazione dei tipi. Ovvero si dichiara una variabile scrivendo prima il tipo e poi il nome della variabile.

## 6.2 Caratteristiche

- Gli array hanno un campo `length`, il numero di elementi che contiene.

Le istanze delle classi hanno dei campi, di cui abbiamo visto già qualcosa nell'introduzione. Siccome le schiere sono oggetti, possono avere campi, in particolare ne hanno uno: `length`. Questo campo contiene il numero di elementi della schiera.

Notare che di una schiera, quando viene creata, viene definito il numero di elementi, per cui la lunghezza della schiera è immutabile. In particolare, non è possibile assegnare il campo `length`, nè esistono operazioni che permettono di cambiarlo. Se si vuole ottenere da una data schiera, un'altra schiera con un numero di elementi diverso (per esempio senza il primo o l'ultimo), si deve creare una nuova schiera, e bisogna ricopiare, dalla vecchia schiera, gli elementi che servono nella nuova. C'è una funzione di sistema, `System.arraycopy`, che semplifica e velocizza questa operazione.

- Si accede agli elementi di un array con l'operatore `[ ]`

Una schiera è una raccolta di elementi, che per comodità vengono maneggiati come un tutt'uno. Ma è altrettanto importante poter esaminare, e utilizzare, i singoli elementi della schiera. Infatti l'operazione principale che effettuare è quella di accesso ad un elemento, o indicizzazione. L'operatore di indicizzazione sono le parentesi quadre.

- Es. `ax[i]` accede all' `i`-simo elemento di `ax`

L'indice è un numero intero, che specifica quale elemento della schiera si vuole utilizzare. Se abbiamo una schiera di interi, l'espressione `ax[i]` è di tipo intero. Analogamente se abbiamo una schiera di stringhe `as`, usando `as[i]` otteniamo una stringa.

- Notare che gli indici vanno da 0 a `ax.length-1`

I valori degli indici partono da 0. Curiosamente se una schiera ha 10 elementi, questi elementi sono numerati da 0 a 9. Un errore che si commette facilmente è utilizzare il valore del campo length per accedere all'ultimo elemento. Scrivere a[a.length] scatena una condizione di errore, che in Java viene gestita generando una eccezione. In pratica nel punto in cui viene eseguita quella espressione, l'esecuzione non continua ma salta ad un gestore di errori. Vedremo poi che l'errore che l'errore generato è `ArrayIndexOutOfBoundsException`, e che l'eccezione causa un ritorno al chiamante del metodo che lo contiene.

```
// Stampa gli argomenti sulla riga di comando
class C {
    public static
    void main(String[]args) {
        for(int i=0; i<args.length; i++)
            System.out.println(args[i]);
    }
}
```

Questo listato è un idioma, ovvero un pezzo di codice frequente e ricorrente nei programmi. In generale per accedere in ordine a tutti gli elementi di una schiera si usa un ciclo che va da 0 e finisce prima della lunghezza. L'esempio mostra come si fa a stampare gli argomenti sulla riga di comando. Ogni classe può avere un metodo `main`, da cui inizia l'esecuzione programmi, a cui viene passato una schiera di stringhe. In questa schiera ci sono tutti gli argomenti passati sulla riga di comando. Ci sarebbe anche da notare che `main` è `public` e `static`, ma per affrontare adeguatamente questi concetti dobbiamo sapere molte cose sulle classi che vedremo via via.

## 6.3 Creazione

### ● Una variabile array contiene un *riferimento*

● L'oggetto contenuto potrebbe non essere stato ancora creato.

● Per poterlo utilizzare occorre creare l'array.

● Se l'oggetto array non è stato creato il riferimento è nullo

Vediamo adesso come creare una schiera. La cosa importante da ricordare è il fatto che una schiera è un oggetto tutti gli effetti. Quindi valgono le regole vistre prima, e una variabile di tipo schiera contiene solamente un riferimento ad un oggetto. L'oggetto in questione potrebbe non esistere: infatti per poter usare una schiera devo creare l'istanza della schiera. Se provo a usare una schiera senza averla creata, ho una sgradita sorpresa: ottengo una eccezione. Vediamo un caso pratico:

```
int [] ax;  
ax[2];  
// NullPointerException!
```

In questo esempio ho solo dichiarato la variabile ma non ho creato l'oggetto contenuto, per cui la risposta dell'interprete è quella che ci può aspettare: una eccezione di puntatore nullo. Fare attenzione che queste eccezione scatta ogniqualvolta si cerca di utilizzare il riferimento nullo per accedere a un oggetto che non c'è. Nell'esempio che stiamo, vedendo l'oggetto che manca è proprio schiera.

### ● Sintassi per la costruzione di array:

Una volta constatato che la schiera va sempre creata, vediamo come fare a crearla, in particolare vediamo la sintassi della creazione.

● costruito con *new* e il numero di elementi

Il modo più immediato è quello di utilizzare l'operatore new. La sintassi richiede che venga utilizzato in questo caso il nome della classe, seguito dal numero di elementi che saranno contenuti nella schiera. Il difetto di questo sistema è che lascia tutte le variabili della schiera non inizializzate. Cioè in questo modo esiste la schiera ma possono non esistere gli elementi della

schiera. In realtà questo dipende dal fatto che gli elementi siano tipi primitivi oppure oggetti. Se si tratta di tipi primitivi, creare la schiera crea automaticamente lo spazio per contenere gli elementi. Se invece si tratta di classi, la schiera contiene solo i riferimenti ad oggetti che dovranno essere creati e poi assegnati ai vari elementi.

### ● elencando tra graffe i valori di inizializzazione

```
// un array di tre interi  
int[] ai = new int[3];  
  
// un array di tre interi  
int[] ai = { 0, 1, 2 }  
//a[0]==0, a[1]==1, a[2]==2
```

L'altra sintassi prevede che vengano scritte tra parentesi graffe tutti gli elementi della schiera. Innanzitutto, non occorre specificare il numero di elementi. Questo numero viene calcolato automaticamente dal compilatore, contando gli elementi posti tra parentesi graffe. L'altro importante caratteristica di questa sintassi è che consente di creare anche di inizializzare i vari elementi della schiera, creando eventualmente gli oggetti mancanti in una schiera di una classe.

### ● Differenze tra le due sintassi

a

- Con `new`, un nuovo array di tipi primitivi esiste ed è usabile
- Con `new`, un nuovo array di tipi derivati contiene solo riferimenti nulli
- Con le graffe, un nuovo array anche di oggetti è completo

```
01. Stack[] as = new Stack[3];
02. Stack t = as[2];
03. //ok
04. as[2].push()
05. //NullPointerException
06. Stack[] bs = {
07.     new Stack(), new Stack(), new Stack()
08. };
09. bs[2].push()
10.// ok
```

Vediamo meglio queste caratteristiche esaminando il listato. Nella riga 1 abbiamo creato un nuovo array di Stack. La schiera c'è, e quindi in riga 2 non ottengo errore. Ma la cosa importante è che non c'è uno stack nel secondo elemento. La schiera appena creata contiene solo puntatori nulli. Per cui se tento di invocare un metodo su as[2] il risultato è NullPointerException. Le cose vanno meglio se creo l'array con l'altra sintassi e mi preoccupo anche di creare i vari elementi contenuti nella schiera. Come si vede il risultato è quello che ci si può aspettare, la schiera ha tre elementi e ognuno degli elementi è stato inizializzato correttamente, per cui è nella riga 9 non abbiamo la sorpresa della riga 2.

## 6.4 Stringhe

- Sono degli oggetti con 'sintattassi da tipi primitivi':

- Costanti stringa

```
String s = 'hello';
```

- breve per

```
char[] tmp = { 'h', 'e', 'l', 'l', 'o' };
String s = new String(tmp);
```

Le stringhe hanno delle costanti, ovvero una sequenza di caratteri tra virgolette singole. La sintassi non rappresenta però una costante come con i tipi primitivi. Quello che viene fuori da questa operazione è un oggetto. Per questo motivo possiamo pensare più correttamente che si tratti di una abbreviazione del secondo listato. Infatti è come se sia stato creato temporaneamente una schiera di caratteri, e questa schiera sia stato utilizzata come parametro per un costruttore. Naturalmente non ci deve preoccupare di questi dettagli se non per il fatto che dev'essere ben chiaro che una stringa è in realtà un oggetto.

- operatore di concatenazione +:

```
s = s + 'world';
```

- breve per

```
s=s.concat("world");
```

Così come abbiamo la possibilità di creare stringhe con una sintassi comoda, un'altra funzione molto frequente, ovvero la concatenazione, viene svolta con un operatore: +. Grazie a adesso è abbastanza facile costruire nuove stringhe, e infatti è di frequente. In realtà bisogna stare anche in questo caso attenti: le stringhe in Java sono immutabili, per cui la concatenazione in realtà non modifica una stringa aggiungendo nuovi caratteri alla fine, ma ne crea un'altra. Infatti la concatenazione di stringhe è un'operazione che può essere molto lenta. Implicitamente concatenando una sequenza di stringhe in una singola espressione per ottimizzare viene usata

un'altra classe, StringBuffer. Il compilatore però non è abbastanza furbo da utilizzarla automaticamente quando si concatenano molte stringhe in parte diverse del programma. Per questo motivo in questi casi bisogna usarle esplicitamente. Ma lo vedremo ampiamente quando tratteremo appunto le stringhe in dettaglio, nell'ultima parte del corso.

### ● operatore di assegnamento +=:

```
s += "world";
```

### ● breve per

```
s = s + "world";
```

Un'altro operatore comodo, abbreviazione di quanto scritto nelle riga precedenti, è il +=. In questo modo si può prendere una stringa e accordargli di un pezzo, senza dover ripetere il nome della stringa. Non è di così grande utilità, ed è presente perché è l'analogo dell'operatore += per i numeri, che è molto usato l'inverno dei cicli. Però c'è e funziona.

## ● Programmazione orientata agli oggetti

Java è un linguaggio di programmazione orientato agli oggetti. Questo significa che implementa tre principi: encapsulazione, ereditarietà, polimorfismo. Nel corso delle slide vedremo abbastanza in dettaglio questi concetti, e come vengono realizzati usando Java. È importante sottolineare che questa la parte più difficile del corso. A parte i singoli e noiosi dettagli della programmazione, la cosa fondamentale di apprendere i concetti sottostanti e usarli nei propri programmi.

### ● Incapsulazione

La OOP cerca di applicare alla programmazione alcuni concetti che sono tipici del modo in cui le persone normalmente considerano e trattano gli oggetti fisici. Consideriamo come primo esempio una radio. Si tratta di un oggetto che utilizziamo tutti i giorni e che ha una caratteristica principale: quella di essere encapsulato. Mi spiego meglio: tutti sappiamo che sotto il guscio di una radio ci sono vari componenti elettrici ed elettronici. Solo che non ne abbiamo accesso diretto: per semplificare l'uso della radio e non costringere le persone a una laurea in ingegneria elettronica per utilizzarla, l'oggetto presenta una serie di manopole e pulsanti. Queste manopole

sono "l'interfaccia" dell'oggetto, ovvero di soli modi in cui è previsto che l'oggetto venga usato. Allo stesso modo vedremo che gli oggetti software hanno una protezione sui dettagli interni di come viene usato l'oggetto, e mette a disposizione solamente alcuni "metodi", che sono i soli modi in cui si può usare legalmente la radio.

## ● Ereditarietà

Nel mondo reale a volte si derivano nuovi oggetti estendendo quelli esistenti. Per esempio ad una radio che ha una portata limitata, posso ampliare le sue capacità di ricezione aggiungendo una antenna più potente. In questo caso la nuova radio eredita dalla vecchia radio molte caratteristiche e in più ha una capacità ulteriore che aumenta le stazioni che è in grado di ricevere.

## ● Polimorfismo

Una caratteristica tipica delle cose reali è il fatto che si possono utilizzare allo stesso modo anche se si tratta realmente di oggetti diversi. Per esempio se ho una radio, mi basta sapere questo per sapere che la posso accendere e posso sintonizzarla su una stazione. In realtà io posso avere radio completamente diverse. Quando gli comunico di "accendersi" invocando il metodo di accensione, la radio sa quello che deve fare. Gli oggetti sono "polimorfi", nel senso che possono avere più forme (realmente codice diverso) anche se apparentemente sono uguali. Ma vedremo ampiamente e più in dettaglio che cosa questo significa nella programmazione.

# 7 OOP

## ● Classe

Innanzitutto rendiamoci conto di che cosa è una classe. Ecco un esempio:

```
class Pesce {  
    int velocita;  
    void nuota() { ... }  
}
```

## ● Modello di oggetto

Una classe è un modello di oggetto. È importante capire che quando dichiaro una classe, non ho fatto (ancora) niente: ho solo specificato cosa voglio, ma senza aver creato nulla che fa quello che voglio. Ho creato una forma, uno stampino nel quale mettendo il gesso o la cera creerò effettivamente l'oggetto che potrò poi usare davvero.

### ● campi

Nella classe possiamo riconoscere il campo e il metodo. Il campo nella programmazione ad oggetti rappresenta lo stato interno dell'oggetto, ed è generalmente (ma non sempre) inaccessibile dall'esterno. Per intenderci, consideriamo ancora la radio: quando è sintonizzata su una stazione, significa che certi elementi al suo interno (per esempio, potenziometri) hanno una certa posizione, che è quindi lo stato in cui si trova l'oggetto. Questi elementi sono però inaccessibili direttamente dall'esterno: il guscio della radio li protegge.

### ● metodi

Proprio per consentire a chi sta fuori della radio di manipolarne lo stato, e quindi in ultima analisi di poter cambiare stazione o alzare il volume, il progettista della radio ha messo a disposizione delle manopole, che con artifici meccanici permettono, IN MANIERA CONTROLLATA, di modificare lo stato interno della radio. In un oggetto software, i metodi hanno la stessa funzione: sono l'interfaccia con il mondo esterno, e rappresentano gli elementi con cui deve interagire chi la radio la deve solo usare. Il progettista ha appositamente messo a disposizione solo ciò che è previsto debba venire usato.

## ● Istanza

Abbiamo finora parlato della classe, che rappresenta un modello di oggetti. Infatti definendo una classe, non esistono (con la sola eccezione dei campi statici, che dobbiamo ancora vedere) i campi della classe: questi appartengono ad una istanza che deve essere appositamente creata. Definire una classe è come fare un progetto: progettare le automobili non significa costruirle effettivamente. Solo quando il progetto viene effettivamente realizzato avremo una delle automobili che abbiamo progettato.

### ● Incarnazione del modello

```
Pesce p = new Pesce()
```

Istanziare significa creare un oggetto che occupa memoria e che ha (almeno) lo spazio per memorizzare i campi.

### ● Una classe molte istanze

È importante capire che ad una classe corrispondono in generale molte istanze, e ogni istanza ha le sue variabili ad essa dedicate.

### ● Metodi invocati sull'istanza

```
p.nuota()
```

I metodi però generalmente agiscono solamente su una data istanza. Intuitivamente è facile da capire: il metodo accendi di una radio si applica solo quando ho effettivamente ho la radio sottomano. Allo stesso modo per invocare il metodo ho bisogno dell'oggetto su cui invocarla. Tutto questo discorso per il momento lo manteniamo a livello intuitivo ma lo riprenderemo più tecnicamente per meglio illustrare i modi in cui funziona la programmazione ad oggetti in Java.

## 7.1 Ereditarietà

### ● Una classe estende un'altra

```
class PesceRosso  
    extends Pesce {  
    int tonalita;  
    void mangia(){ ... }  
}
```

Un'altra cosa importante che è possibile fare è quella di ereditare, ovvero creare una classe che ha stessi campi e metodi di un'altra e in più ne aggiunge nuovi.

### ● Aggiunge campi

Vediamo meglio che cosa vuol dire questo. Ricordo innanzitutto che una classe esistente è solamente una specie di modello di ciò che si vuole ottenere. Quando uso l'ereditarietà, creo una nuova forma, un nuovo modello di quello che voglio ottenere. Questo modello ha tutti i campi che prima esistevano, è in più ne aggiunge dei nuovi. Avremo così uno nuovo modello con una maggiore capacità di contenere informazioni. Per riferirsi a quello che c'è nell'esempio, diciamo che il pesce rosso ha in più la tonalità.

### ● Aggiunge metodi

Ma questo non basta: in particolare l'ereditarietà ci consente di aggiungere nuovi comportamenti. Nell'esempio, il pesce rosso ha anche la capacità di mangiare che il pesce base non aveva. Questa caratteristica viene evidenziata dal fatto che la nuova classe ha dei metodi in più.

### ● Eventualmente "ridefinisce"

Un aspetto fondamentale della programmazione oggetti è il polimorfismo. Lo vedremo della prossima slide, per adesso notiamo che possiamo inserire nella classe dei metodi che hanno lo stesso nome (e gli stessi parametri) di metodi già esistenti nella classe base. Il fatto che siano gli stessi di quelli esistenti causa il comportamento di ridefinizione, ovvero di sostituzione del vecchio comportamento con uno nuovo.

## 7.2 Polimorfismo

### ● Ridefinizione di metodi

La capacità di ridefinire metodi, è alla base della polimorfismo. Vediamo nel seguente esempio come stiamo sostituendo al metodo nuota() esistente uno metodo, che ha lo stesso nome di quello della classe base.

```
class PesceRosso
  extends Pesce {
  void nuota() { ... }
```

In tal caso avviene un meccanismo speciale detto polimorfismo. Prima di entrare dettagli, cerchiamo di capire concettualmente che vuol dire che è un oggetto e polimorfo. Non bisogna andare molto lontano con gli esempi: considerando una automobile, questa ha dei comportamenti diversi a seconda del modello. In particolare per accenderla (nel nostro linguaggio invocando il metodo accendi() della classe Auto), il comportamento effettivo del motore per fare questa operazione è diverso a seconda che si tratta di una Ferrari monoposto di formula uno, o una utilitaria per andare alla spesa. In ogni caso, entrambe si accendono, si mettono in moto. In pratica ad una richiesta che è abbastanza "standard" (accendere la macchina), la specifica automobile sa in se come comportarsi. Polimorfismo significa che gli oggetti cambiano il loro comportamento pur mantenendo la stessa forma.

## 7.3 Chiamata

### ● Variabile e Istanza

Vediamo in pratica come questa cosa viene realizzata. Abbiamo una classe derivata, nel nostro caso il pesce rosso. Quando una classe è derivata da una da una classe base, mantiene a tutti gli effetti le caratteristiche della classe base. Di conseguenza, è possibile assegnare a variabili che hanno il tipo della classe base, istanza che sono in realtà classi derivate.

```
Pesce p =  
new PesceRosso()
```

Possiamo vedere un esempio che è in Java una cosa comunissima. Creo un pesce rosso, ma lo assegno ad una variabile del tipo pesce. Questo è del tutto lecito: infatti il pesce rosso è a tutti gli effetti ancora un pesce, e pertanto può essere assegnata a variabili di tipo pesce.

### ● Chiamata

```
p.nuota()
```

Abbiamo però un caso particolare che dobbiamo capire bene: cosa succede quando viene chiamato il metodo nuota() attraverso questa variabile. Notare bene che la variabile è di un tipo, ma punta ad un oggetto che di un altro tipo. Abbiamo il metodo nuota() sia nella classe Pesce, che un metodo nuota() nella classe PesceRosso. Quale dei due viene invocato?

### ● Chiama la nuota di PesceRosso

La risposta sta nel polimorfismo: abbiamo detto che un oggetto risponde di autonomia alle richieste che gli vengono fatte, di modo che a richieste generiche, corrispondono comportamenti specifici. In realtà, questo è un meccanismo per riutilizzare più possibile codice esistente, avendo possibilità, quando il codice esistente non è esattamente come serve, di modificarlo, cambiando qualcuno dei metodi. Nel nostro caso particolare, il risultato dell'invocazione del metodo nuota() usando p causa la chiamata del metodo nuota di PesceRosso. In generale, gli oggetti si comportano in base a ciò che sono (ovvero l'oggetto effettivo che è stato stanziato), non in base ciò che sembrano (la variabile con il quale vengono invocati). Fare attenzione che questo

comportamento non è scontato, e non è nemmeno semplice da realizzare: si ottiene infatti tramite il mantenimento di puntatore a funzioni, che possono venire rimpiazzato nel corso dell'esecuzione. Ma non approfondiremo per il momento più di tanto l'argomento, rimandando alle slide specifiche che incontreremo via via.

## 8 Classi

- Le classi implementano il principio dell'incapsulazione



Esaminiamo adesso come si realizza praticamente ed effettivamente un oggetto. Come abbiamo già detto una classe è il modello di un oggetto, ne definisce il tipo e dice cosa contiene e come si deve comportare l'oggetto in risposta alle richieste dall'esterno (le chiamate dei metodi).

- i campi rappresentano lo stato interno di un oggetto

Nella programmazione ad oggetti i dati non sono esposti al pubblico come nella tradizionale programmazione strutturata. Infatti quando si programma in linguaggi come il pascal classico si hanno "strutture dati" che vengono gestiti da "algoritmi". Ricordiamo infatti che il famoso libro di Wirth che è uno dei capisaldi della storia dell'informatica (e che presenta il linguaggio Pascal) si intitolava: "Algoritmi + Strutture Dati = Programmi". Nella programmazione ad oggetti le strutture dati sono invece un dettaglio, che non deve essere reso noto ma protetto dall'esterno per evitare che qualcuno possa manipolarlo in maniera scorretta. Le informazioni contenute in un oggetto sono racchiuse al suo interno, nelle sue strutture dati che rappresentano appunto l'informazione riservata "incapsulata".

- i metodi permettono di cambiare lo stato in modo controllato

Ovviamente un programma per funzionare deve poter modificare le strutture dati. La programmazione ad oggetti non nega che l'algoritmo deve agire sulle strutture dati. Pretende però che si dica una volta e per tutte quali sono gli algoritmi e su quali strutture dati agiscono. Notare che nemmeno gli algoritmi sono pubblici: anche essi rappresentano un dettaglio implementativo che viene nascosto all'esterno. L'utilizzatore potrà solamente invocare i metodi di un oggetto. Il fatto che i metodi siano i soli a poter accedere alle strutture dati garantisce che

L'oggetto modifichi il suo stato interno in modo controllato e prevedibile. Pensiamo ad uno stack: siccome gli unici modi che ho per modificarlo sono i metodi push e pop, se implemento correttamente gli algoritmi sono messo al sicuro da situazioni in cui per esempio il contatore di elementi non coincida con il numero di elementi effettivamente presenti nello stack.

## ● i costruttori inizializzano l'oggetto in uno stato noto

Quindi, io creo uno stack vuoto. Poi aggiungo un elemento con push. Di conseguenza ho uno stack con un elemento. Da qui posso svuotarlo con pop o aggiungere un altro elemento con push. Come si vede l'oggetto si muove entro dei binari ben predefiniti. L'ultimo elemento che rimane da aggiungere al tassello è appunto la creazione di cui ne abbiamo parlato. Serve per definire e creare nuovi oggetti, ed è un concetto importante quanto quello di metodo o di campo. In programmazione a oggetti, si chiamano costruttori le funzioni che vengono invocate per inizializzare un oggetto. I costruttori in un certo senso sono metodi, ma sono metodi speciali. Infatti non necessitano di un oggetto per essere utilizzati (a differenza dei metodi veri e propri) e come risultato della loro invocazione creano un nuovo oggetto.

## 8.1 Record

- Una classe può essere considerata come un semplice record.

```
class Stack {  
    int top;  
    int[] stack;  
}
```

Per meglio capire come funziona la programmazione ad oggetti faremo una ricostruzione "storica". In effetti il concetto di "classe" è a tutti gli effetti una evoluzione abbastanza naturale del concetto di "record" del Pascal (o di struttura del C): ovvero il tipo di dato derivato risultante dall'aggregazione di altri tipi di dato. Una classe degenera è infatti a tutti gli effetti un record. Consideriamo la struttura dati nel listato precedente. Si tratta di una classe con due campi. In questo caso, si tratta di una dichiarazione in tutto e per tutto analoga alla dichiarazione di un record puro come quelli del Pascal, ovvero una struttura dati aggregata, che mette insieme un intero e un array di interi.

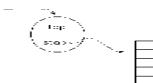
- Quando uso un record devo prima crearlo

```
// creo lo stack  
Stack s=new Stack () ;
```

Ora, una dichiarazione di una struttura dati di tipo record è appunto semplicemente una dichiarazione: dice come deve essere la struttura dati ma non ne crea alcuna. Anche per i record in pascal, quando decido di creare una "istanza" (appunto) del record, la devo creare esplicitamente. Anche in pascal è richiesto l'uso di new. Ma noi usiamo la sintassi di Java anche per questo caso in cui stiamo evidenziando le analogie con il Pascal. E in questo caso dobbiamo scrivere new Stack() invece di new Stack. La differenza (sottile) è che nel primo caso abbiamo usato il nome del record, mentre in Java in realtà stiamo invocando un costruttore. Ignoriamo per ora questa differenza e semplicemente rendiamoci conto che abbiamo solamente creato una struttura dati aggregata che contiene le strutture dati definite nella dichiarazione.

- Poi devo inizializzarlo esplicitamente:

```
s.top=0;  
s.stack = new int (10);
```



Nei vecchi tempi andati (quelli della programmazione strutturata e degli algoritmi + strutture dati), potevamo solamente creare la struttura dati. Che fosse buona all'uso così come era non era detto, per cui spesso (per non dire quasi sempre) una inizializzazione era d'obbligo. E infatti la nostra struttura dati se deve essere usata come Stack è tutt'altro che completa. Infatti al top bisogna dargli il suo valore iniziale, e soprattutto allo stack (che è un array di interi) bisogna crearlo esplicitamente. Non dimentichiamo che un array è un oggetto e dichiarare la variabile significa solamente dichiarare una scatola che contiene un PUNTATORE all'istanza vera e propria. Ma istanziare il record NON mi crea i sotto oggetti: devono venire istanziati esplicitamente. Per cui l'inizializzazione completa deve essere esplicita.

## ● Adesso possiamo utilizzarlo

```
// operazione di push(1)  
s.stack [s.top++]=1;
```

Fatte queste inizializzazioni (esplicite) finalmente lo stack è pronto per l'uso. Notare che quando usiamo la programmazione strutturata classica si deve accedere alla struttura dati direttamente, scrivendo codice come quello nel listato. Manco a dirlo, il risultato sarà sì efficiente (ma serve in questo caso l'efficienza?), ma poco comprensibile. E molto poco modificabile.

## 8.2 Istanze

### ● Le variabili di una classe si chiamano campi

Entriamo adesso un pò in dettaglio su queste istanze. Abbiamo detto che le variabili che vengono messe al di fuori di un metodo ma dentro una classe si chiamano campi.

### ● I campi sono sempre:

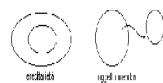
- tipi primitivi
- riferimenti ad altri oggetti

I campi sono come le altre variabili, e come tali hanno un tipo. Le variabili in Java possono contenere tipi primitivi (nel qual caso creata la variabile viene creato anche lo spazio in memoria per contenerlo) e tipi oggetti (nel qual caso viene creata solo la variabile e niente altro).

### ● Non si può contenere un altro oggetto con un campo

- Si deve ricorrere all'ereditarietà.

Notare che se una classe contiene un campo di un'altra classe, questo non significa che adesso la classe ne contiene i campi: invece contiene una "scatola" che a sua volta contiene un'altra scatola. Usando l'ereditarietà riesce invece a creare una nuova classe che ha nuovi campi e mantiene i vecchi. E non ci sono "scatole" (ovvero oggetti) a frapporsi.



### ● Allocazione di una istanza new Stack()

Quando alloco una nuova istanza, senza specificare altro, succede questo:

Quando alloco una nuova istanza, senza specificare altro, succede questo:

- crea i campi tipi primitivi

Vengono innanzitutto creati tutti i campi primitivi. In particolare lo spazio per contenere le variabili c'è ed è inizializzato a zero.

## ● crea i riferimenti nulli ai sotto oggetti

Invece i campi di tipo oggetto vengono si inizializzati, ma al valore nullo. Infatti una variabile di tipo oggetto può eventualmente non contenere nulla. Ed è quello che sicuramente succede se nessuno dà maggiori informazini.

```
Stack s= new Stack () ;
```

Creando uno stack senza ulteriori ottimizzazione, si ottiene quello che è visibile in figura.

## 8.3 Costruttori

XXX

### ● Definire un *costruttore*

Fin qui non abbiamo fatto altro che creare un record, ovvero definirlo (nel senso che abbiamo descritto come è fatto) e "allocarlo" (nel senso che abbiamo riservato memoria per contenere i dati contenuti nel record). Il limite della allocazione tradizionale è che riserva spazio di memoria e basta. Generalmente la memoria allocata non basta per potere utilizzare la struttura dati. Occorre inizializzarla in qualche modo. Per esempio lo stack richiede che venga creato anche un array contenuto dentro lo stack, e che il puntatore al primo elemento libero dello stack (il "top") sia posto a zero. L'allocazione per esempio a volte garantisce che la memoria sia a zero, ma non crea altri oggetti. In programmazione strutturata, dopo aver allocato la memoria il programmatore deve provvedere ad inizializzare la memoria in modo che sia usabile dagli algoritmi.

### ● Legando il codice di inizializzazione all'allocazione

Ora, il punto è che l'inizializzazione è concettualmente legata alla allocazione di memoria. Spesso in programmazione strutturata si scrivono funzioni di inizializzazione che devono essere eseguite subito dopo che la memoria è stata riservata. In programmazione ad oggetti si fa un passo avanti.

### ● Evitando inizializzazioni esplicite

```
class Stack {  
    int top;  
    int [] stack;  
    Stack () {  
        top=0;  
        stack = new int [10];  
    }  
}
```

Il concetto è che quando si fabbrica un oggetto, questo deve essere pronto all'uso. Se andiamo in negozio a comprare una radio, non ci aspettiamo che ci vengano dati i pezzi da assemblare (a meno che non si è un hobbista di elettronica...) Ci aspettiamo una radio completa, assemblata e pronta all'uso. Così in programmazione ad oggetti, quando creiamo un nuovo oggetto, questo viene creato pronto all'uso nell'atto stesso di allorare la memoria. Più esattamente i costruttori sono particolari tipi di metodi che vengono invocati su un'area di memoria appena allocata. Come si vede in figura abbiamo esteso il nostro record per effettuare le inizializzazioni. La prima caratteristica che balza all'occhio è una procedura scritta dentro la classe. Per la precisione la procedura è un metodo, posto all'interno di una classe e obbligatoriamente con lo stesso nome della classe. Dentro abbiamo il codice di inizializzazione dei campi.

## • Adesso basta new Stack() per avere una istanza completa

Con questa definizione, nell'eseguire una new, viene prima allotta la memoria e poi viene chiamato il costruttore che abbiamo definito. Questo spiega definitivamente perché dopo una new c'è una chiamata e non semplicemente il nome del nuovo tipo di dato: la chiamata fa appunto le veci di inizializzare la struttura dati che il programmatore sta richiedendo. In verità ci sono alcune cose ancora da chiarire in particolare come fa il meotodo costruttore a trovare le variabili da inizializzare. Una cosa che deve essere chiara subito e che spesso induce in errore i programmatore avvezzi ad altri linguaggi è il fatto che le variabili della classe (i campi) non sono affatto variabili locali che scompaiono alla fine di un blocco, né tantomeno sono variabili globali universamente disponibili: ma vediamo adesso in dettaglio come funziona.

## 8.4 Campi

### ● Quando uso un campo mi riferisco alla istanza corrente

Una volta che ho definito una classe, posso creare diverse istanze della classe, e sono tutte diverse. Come dire, una volta che ho progettato una automobile posso realizzarne diverse utilizzando lo stesso progetto. O come dire che una volta creata una forma per i calchi in gesso, di calchi se ne possono realizzare innumerevoli. Tuttavia ogni istanza ha i suoi campi e i suoi metodi. E i metodi fanno riferimento ai campi. Ovviamente si vuole che un metodo faccia riferimento ai campi della particolare istanza. Il problema è: come legare i campi ai metodi? Qualcuno potrebbe pensare a qualche misterioso legame con sistemi nascosti che tengono traccia in maniera complicata di queste informazioni. Niente di tutto questo: l'associazione tra campi e metodi viene fatta invece dall'utilizzatore dell'oggetto. Vediamo come.

### ● Consideriamo una semplice classe

```
class Num {  
    int n;  
    Num (int num) {  
        n = num;  
    }  
}
```

La classe più semplice che possiamo immaginare (a parte quella degenera, senza campi, che non fa nulla) è quella con un solo campo, mostrata nel listato precedente. Queste classi sono utili: per esempio la classe Integer di sistema di Java è simile a questa. È quello che si chiama una classe "pacco" (wrapper), non nel senso che rifila un bidone, ma nel senso che impacchetta in un oggetto un tipo primitivo. Nel nostro esempio, la Num ha un campo n, e un costruttore che inizializza il campo ad un valore specificato come parametro del costruttore.

### ● Creiamo due oggetti:

Adesso mettiamo in pratica quello di cui abbiamo appena parlato: istanziamo due oggetti, come nel seguente listato:

```
Num a = new Num(1);  
Num b = new Num(2);
```

- Abbiamo che la n di a (**a.n**) vale 1, mentre la n di b (**b.n**) vale due, e sono separate e distinte.

Il risultato è che creiamo due istanze diverse, ciascuna con un campo n, che ha un valore indipendente dall'altro. Infatti nella prima istanza il campo n vale 1 mentre nella seconda istanza il campo n vale due.

- **I costruttori (e i metodi) vengono invocati in un determinato "ambiente"**

Di conseguenza è facile capire che esiste un meccanismo per distinguere i due oggetti. Dall'esterno è facile da capire: abbiamo la variabile a che punta al primo oggetto e la variabile b che punta al secondo oggetto. Ma bisogna anche rendersi conto che esiste un meccanismo per distinguere i due oggetti dall'interno (ovvero da codice che accede ai campi).

- **questo ambiente è l'oggetto corrente**

In pratica il codice di costruttori e metodi viene invocato in un ambiente: grazie a questo e ambiente si riesce ad accedere all'oggetto corrente.

- **modificare i campi significa modificare il proprio oggetto lasciando inalterati gli altri**

Quando si modificano i campi in realtà si modifica l'ambiente in cui viene invocato oggetto, ed è in questo modo che l'oggetto viene modificato senza che vengano intaccati i campi degli altri oggetti.

## 8.5 this

### ● L'oggetto corrente viene acceduto tramite this

Il meccanismo nascosto che consente di accedere ai campi è la variabile implicita (ma sempre presente) `this`. Questa variabile viene utilizzata ogni volta che nel codice si fa riferimento campo. Infatti possiamo vedere nel seguente listato:

```
Num (int num) {  
    n = num;  
}
```

La `n` non è un parametro del costruttore, e non è nemmeno dichiarato all'interno del costruttore. È una variabile libera, nel senso che nessuno la definisce. In questi casi la cosa è gestita come segue:

### ● equivale a:

```
Num(int num) {  
    this.n = num;  
}
```

Tutte le variabili libere prendono il loro valore dall'ambiente, e vi di accedono tramite la variabile implicita `this`.

- Notare che `n` è campo (persistente) mentre `num` è un parametro (non persistente)

Un'altra cosa che distingue i campi dalle variabili è il fatto che mentre le variabili vanno via alla fine del metodo, invece campi sono più resistenti: sono persistenti e permangono alla fine del metodo.

## 8.6 Allocazione

Proseguiamo il paragone con la programmazione strutturata. Come ricorderete a un certo punto ci siamo preoccupati di creare l'oggetto vero proprio. Abbiamo fatto questo usando l'operatore new seguito dal nome della classe, come nel seguente esempio:

```
Stack s = new Stack();
```

Notare che non stiamo usando esattamente il nome della classe, ma un metodo che si chiama come la classe (mentre quando dichiariamo la variabile usiamo proprio il nome da classe).

### ● In realtà fa due cose:

- new alloca la memoria necessaria pari alla dimensione dell'oggetto
- Stack è un costruttore chiamato da new dopo l'allocazione per inizializzare l'oggetto

Questa è una chiamata di un costruttore, che in realtà fa due cose distinte: alloca la memoria necessaria per contenere l'oggetto e poi chiama il corpo della costruttore per effettuare l'inizializzazione. Tutto questo in unica soluzione: io mi limito chiamare il costruttore e l'oggetto viene ritornato e allocato e inizializzato.

- Se non specifico costruttori, viene definito automaticamente un "costruttore di default"

Possiamo usare questa sintassi anche quando in una classe non abbiamo definito alcun metodo. Infatti un costruttore viene automaticamente aggiunto ogni volta che non ce ne sono. Questo per consentire di inizializzare sempre una classe senza metodi (che come ricorderete è praticamente un record).

```
Stack() {}
```

- il costruttore aggiunto non fa niente

questo costruttore è stato aggiunto non fa niente. Il suo scopo è quello di consentire la chiamata del costruttore, e quindi consentire l'allocazione della memoria. Infatti questo costruttore serve solamente per far allocare la memoria, e basta.

- in realtà, chiama i costruttori della classe base, ma lo vedremo poi

A voler essere precisi questo costruttore fa molte cose oltre ad alloca nella memoria : in particolare non si vede ma chiama il costruttore della classe antenato. Per capire questo concetto dobbiamo però avere le conoscenze del meccanismo della ereditarietà, che è una cosa che vedremo più avanti nel corso.



**Il tuo sito è sempre uguale ? EpNuke** la soluzione modulare professionale per il Web Publishing.  
Soluzioni **tutto compreso** con migrazione della grafica, hosting e assistenza.

Per informazioni [www.eprometeus.com](http://www.eprometeus.com) - 120/225

## 8.7 Più costruttori

### ● Posso avere vari costruttori:

Possiamo creare gli oggetti senza specificare nessuno caratteristica. Per esempio se andiamo dal negoziando accedere una radio senza dire nulla, in negoziante ci darà una radio comune che copre la maggior parte delle esigenze. Come dire che quando viene chiesto una radio mi viene data una radio standard. Questa cosa in programmazione oggetti la esprimo chiamando il costruttore senza argomenti. Ma io posso definire dei costruttori con argomenti. Ecco un esempio:

```
Stack (int n) {  
    top=0;  
    stack = new int [n];  
}
```

Questo è un costruttore che crea uno stack di dimensione fissa. Per tornare al nostro paragone con il negozio, è come se avessimo chiesto al negoziante una radio con certe caratteristiche.

### ● Se ho un costruttore qualsiasi non ho quello di default

Una cosa a cui bisogna stare attenti è che il meccanismo di aggiunta automatica di costruttori viene disabilitato quando l'utente aggiunge esplicitamente un costruttore. Vediamo l'esempio:

```
Stack (int n)  
{ top=0; stack = new int [n]; }  
Stack s = new Stack (); //ERRORE!
```

Se nello stack aggiungiamo il costruttore con un argomento, sparisce il costruttore senza argomenti. Di conseguenza un codice prima perfettamente funzionante che usa il costruttore senza argomenti non funziona più. Per fortuna rimediare è facile, basta aggiungere esplicitamente il costruttore senza argomenti con un corpo vuoto.

## 8.8 this()

Quando vado in negozio a comprare una radio, posso chiedere una radio di un certo tipo, oppure una radio e basta. Quindi abbiamo due casi distinti, che si può tradurre in programmazione oggetti come la costruzione di un oggetto con costruttori diversi.

### ● Posso concatenare i costruttori

Il codice dei costruttori tende ad essere molto simile, e quindi a ripetersi. Nella maggior parte dei casi succede che esiste un costruttore molto generale con numerosi parametri, e vari costruttori più semplici, che sono casi particolari di quello più generale. Esiste un meccanismo per ovviare a questo problema, che consiste nella possibilità di concatenare i costruttori. Vediamo un listato.

```
class Stack {  
    int top;  
    int[] stack;  
    Stack(int n)  
    { top=0; stack = new int[n]; }  
    Stack() { this(10); }  
}
```

### ● this(...) deve essere il primo comando di un costruttore

Bisogna tenere presente che il costruttore di un oggetto fa due cose: prima alloca la memoria, e poi la inizializza. Per questo motivo non posso semplicemente chiamare un altro costruttore, devo utilizzare un meccanismo speciale che evita l'allocazione di memoria. Questo meccanismo è il this(...). Esiste un altro vincolo importante: il this() deve essere il primo comando di un costruttore. In pratica nell'esempio vediamo come viene invocato dal costruttore senza argomenti il costruttore con un argomento. Il concetto è che se chiedo uno stack e basta, me ne viene fornito uno con dieci elementi. Eventualmente posso specificare esplicitamente il numero di elementi contenuti.

● la chiamata **this(...)** è cosa diversa dal puntatore **this.campo**

Notare le parentesi dopo il nome. Questo costrutto non ha niente a che vedere con il this usato come variabile. In questa forma è un meccanismo che consente ad un costruttore di far eseguire l'inizializzazione a un altro costruttore, senza che venga di nuovo allocato la memoria.

## 8.9 Inizializzazioni

Dobbiamo affrontare adesso un altro problema: cosa succede alle espressioni di inizializzazione? Stiamo parlando di quelle espressioni che seguono la dichiarazione campi, come nel seguente esempio.

```
class Stack {  
    // inizializzazione  
    int top=0;  
    int stack;  
    Stack (int size)  
        { stack = new int [size]; }  
    Stack () { this (10); }  
}
```

### ● Le espressioni di inizializzazione:

Abbiamo che `top=0` è una espressione di inizializzazione. Il concetto importante da capire è che queste espressioni vengono eseguite all'atto della costruzione. Ovvero fanno parte del costruttore.

#### ● appartengono a *tutti* i costruttori

Possiamo dire che l'espressione di inizializzazione appartengono a tutti i costruttori. Ovvero se vogliamo che un campo abbia un valore iniziale, lo stesso per tutti i costruttori, definiamo il suo valore in una espressione di inizializzazione.

#### ● vengono eseguite **prima** del costruttore

- ma dopo l'allocazione

Queste espressioni vengono eseguite prima del costruttore. In pratica quando creo un oggetto, per prima cosa viene allocata la memoria, poi vengono eseguite le espressioni di inizializzazione dei campi e infine si raggiunge il corpo del costruttore. In rapace qualcos'altro che avviene prima, lo vedremo più avanti.

#### ● non possono contenere comandi

Ovviamente posso usare questo meccanismo solamente quando posso inizializzare un campo senza fare troppi calcoli. Queste espressioni infatti non possono contenere comandi. Se devo fare delle operazioni prima di ottenere il valore da assegnare, devo spostare l'operazione nel corpo del costruttore.

### ● non possono sollevare eccezioni

Un altro problema è che non posso gestire eccezioni. Anche in questo caso devo spostare le operazioni dentro il corpo di un costruttore.

- notare che `new int[size]`, richiedendo un parametro, va posto in un costruttore

Infine l'ultimo caso, ma non meno importante, è quello che nelle espressioni di inizializzazione non posso usare il parametri passati al costruttore. Anche in questo caso devo ricorrere al corpo di un costruttore. In tutti questi casi, qualora si ripetessero blocchi di codice comune, devo sfruttare il meccanismo del `this()` per ridurlo.

## 8.10 Riferimenti

### ● Notare l'ordine

Osserviamo il seguente listato e chiediamoci se quanto in esso espresso possa considerarsi corretto.

```
class Stack {  
    Stack(){this(10);}  
    Stack(int size)  
    { stack = new int [size]; }  
    int top=0;  
    int stack;  
}
```

### ● È corretto! Riferimenti in avanti sono consentiti

- Prima vengono inizializzati i campi
- Poi viene eseguito il corpo costruttore

A prima vista qualcuno esperto in altri linguaggi potrebbe esprimere qualche dubbio in quanto potrebbe pensare che le dichiarazioni dei campi, essendo posti DOPO il loro uso, potrebbero dare errore. Invece non è così in quanto Java è progettato per massimizzare la facilità d'uso. In effetti i riferimenti in avanti sono consentiti. L'esecuzione rimane comunque quella specificata finora: prima vengono inizializzati i campi e poi viene eseguito il corpo del costruttore. Un metodo può riferirsi a un campo o chiamare un metodo non ancora specificato.

## 8.11 Ordine

### ● L'ordine reciproco delle inizializzazioni è significativo

In una classe l'ordine non è importante se non in casi ben precisi. Il primo caso in cui l'ordine è importante è dentro un metodo. I comandi e le espressioni vengono valutati seguendo il loro ordine, e il flusso varia a secondo di if e while che vengono incontrati. Questo è abbastanza scontato. Ma c'è un altro caso in cui l'ordine conta, ed è l'ordine in cui i campi sono ordinati TRA DI LORO. Per esempio:

```
class Nums {  
    int a=1;  
    int b=a+1; // giusto  
    int c=d; // sbagliato  
    int d=0;  
}
```

### ● Le inizializzazioni vengono eseguite in ordine di apparizione

Tutte le inizializzazioni vengono eseguite nell'ordine in cui compaiono nel testo di una classe. Non importa se in mezzo ci sono metodi o costruttori (quelli vengono eseguiti dopo). In una inizializzazione si possono usare altri campi ma solo se COMPAIONO DOPO. Nell'esempio io posso inizializzare b usando a perchè il valore è già noto, ma non posso assegnare d a c perchè ancora non se ne sa nulla.

# 9 Metodi

- In una classe posso dichiarare dei metodi:

```
class Stack {  
    //...  
    void push (int x){  
        stack[top++]=x;  
    }  
    int pop() {  
        return stack[--top];  
    }  
}
```

Dopo aver visto i campi e i costruttori rimangono i metodi. Nell'esempio vediamo i metodi che implementano le operazioni definite su uno stack: push e pop. Ricordiamo che una classe rappresenta un nuovo tipo di dato, che questo tipo può essere istanziato, che i campi ne mantengono lo stato interno e che i costruttori rappresentano le costanti dei nuovi tipi di dato definite. Per ultimi (ma non meno importanti) i metodi rappresentano le operazioni che manipolano e modificano l'oggetto per fargli compiere i compiti a cui è chiamato. Nell'esempio notiamo che nel testo dei metodi ci sono variabili "libere": stack e top non sono né parametri né variabili locali. In effetti fanno riferimento ai campi.

## 9.1 Invocazione

Esaminiamo il meccanismo complessivo con cui i metodi vengono invocati, e come si fa a eseguire i metodi sui campi di una istanza, tenendo separati i campi di istanze diverse.

### ● I metodi:

- possono essere richiamati solo avendo un oggetto (una *istanza* di una *classe*)
- questa viene costruita invocando un *costruttore*

Il meccanismo di uso di un oggetto prevede che non si possa chiamare un metodo da solo: si deve necessariamente fare riferimento ad esso sfruttando una istanza di un oggetto. Per ottenere una istanza di un oggetto che non si ha si può invocare in qualunque momento un costruttore. Solo dopo aver costruito l'oggetto si possono invocare SU DI ESSO i metodi. Esempio:

```
Stack s = new Stack()  
s.push(4);  
Stack t = new Stack();  
t.push(5);
```

Questo è il completo ciclo di vita di un oggetto: nasce, ovvero viene costruito. Dopo di che viene assegnato ad una variabile grazie alla quale si possono invocare i metodi. Se voglio posso creare più istanze e invocare i metodi separatamente. Se vi ricordate, due istanze hanno i campi distinti perché vengono memorizzati in zone di memoria diverse. Il costruttore usa il this per sapere in quale zona di memoria si deve agire per inizializzare l'oggetto.

- Nei metodi, i riferimenti ai campi hanno sempre un **this**隐式的

```
void push(int x) {  
    stack[top++]=x;  
}
```

- equivale a:

```
void push(int x) {  
    this.stack[this.top++]=x;  
}
```

Un meccanismo pressochè identico esiste per i metodi. Mentre per i costruttori il this viene passato dal new, che alloca prima la memoria, per i metodi l'oggetto corrente viene assegnato a this è quello che viene usato per chiamare il metodo. È come se this fosse un parametro in più, invisibile, passato però sempre alla chiamata del metodo. Ecco perchè i due stack sono distinti e il push del primo non incide sul push del secondo: perchè vengono usati i campi di due istanze distinte, passate diverse in momenti diversi grazie al this. Tutte le "variabili libere" vengono cercate nell'oggetto corrente grazie ad un this prefisso implicito di ogni chiamata. Consigliamo di verificare la comprensione di questo meccanismo accuratamente perchè è assolutamente fondamentale.

## 9.2 Overloading

Passiamo a qualche dettaglio tecnico in più riferito ai metodi. Ricordiamo che abbiamo detto che il '+' è overloaded. Infatti la somma può essere fatta su tipi diversi, numeri interi, numeri in virgola mobile e perfino stringhe. In realtà si tratta di operatori nei fatti completamente diversi che hanno in comune solamente il nome e qualche similitudine CONCETTUALE in quello che fanno (è questa similitudine che giustifica il potenziale errore indotto dal fatto che hanno lo stesso nome). Ma l'uomo è abituato a pensare per similitudini, per cui l'overloading degli operatori, pur non essendo totalmente trasparente a chi è inesperto, si rivelano un utile meccanismo.

- due o più metodi possono avere lo stesso nome

- i metodi devono avere argomenti diversi:

```
void move(int x, int y)
{ ... }
void move(Point p)
{ ... }
```

I metodi sono l'equivalente per le classi di quello che sono gli operatori per i tipi primitivi. Come tali, supportano anche essi una forma di overloading. Infatti i metodi di una stessa classe possono avere lo stesso nome ma un differente tipo e numero di argomenti. Come si può notare nel listato il metodo può venire invocato usando l'unico nome move, ma in realtà si tratta di due metodi diversi. Per selezionare quale metodo viene effettivamente invocato si usa il tipo dei parametri.

- i metodi non si distinguono per il valore ritornato

```
void move(int x, int y)
{ ... }
// errore!
boolean move(int x, int y)
{ ... }
```

Tuttavia se è sufficiente l'uso dei parametri per selezionare il metodo, non è sufficiente il tipo ritornato. Infatti ci sono troppe ambiguità, per cui due metodi non possono avere lo stesso nome, gli stessi parametri e un diverso tipo ritornato. In realtà ci sono casi in cui si presentano ambiguità anche i metodi con parametri di tipo diverso. Si tratta di casi abbastanza patologici e non frequenti, che vengono rilevati dal compilatore come errore.

## 9.3 Attenzione

- i metodi ritornano sempre un valore, anche void
  - i costruttori no, e devono avere lo stesso nome della classe
  - uno degli errori più frequenti:

```
class Stack{  
    void Stack(int x)  
    { } //no!!!  
}
```

- questo è un metodo che si chiama come la classe, non un costruttore

Un errore molto frequente è quello di mettere la parola void prima di un costruttore. Questo è un errore, un costruttore non ritorna nulla (deve solo inizializzare l'istanza la classe che sta costruendo). Se si mette la parola chiave void si dichiara un metodo con lo stesso nome della classe, e non un costruttore.

## 9.4 Finalizzazione

### ● Java rileva automaticamente gli oggetti inutilizzati.

- Ogni oggetto non usato da nessuno viene riciclato automaticamente (garbage collection).

Abbiamo visto come creare un nuovo oggetto. In Java, a differenza di altri linguaggi, non è necessario distruggere gli oggetti. Questo grazie a un meccanismo che si chiama "garbage collection" (raccolta di spazzatura). Per cui l'uso di un oggetto software è simile a quello di un oggetto reale. Si prende l'oggetto, si usa e quando hanno sempre più si poggia sulla scrivania. Ogni tanto passa la donna delle pulizie a raccogliere lattine e cicche di sigaretta.

### ● La garbage collection non ricicla le altre risorse del sistema

- per esempio file, windows, threads...

Bisogna stare attenti che la raccolta di spazzatura lavora solamente su memoria. In un sistema operativo che sono altre risorse oltre la memoria che possono essere impegnate da una istanza. L'esempio più semplice è quello del file: aprire un file comporta impegnare un "descrittore di file", risorsa interna del sistema operativo. Di queste risorse ce ne è un numero limitato: per questo motivo una volta che non si usa più il file, bisogna rilasciarlo esplicitamente: in particolare, occorre chiuderlo.

- è possibile aggiungere un metodo **finalize()** ad ogni classe.

- viene invocato una volta sola prima della garbage collection.

a volte può succedere che non si sa bene quanto bisogna rilasciare le risorse occupate da un oggetto. In generale il momento di liberarle può coincidere con il momento di distruggere l'oggetto, ovvero con la raccolta di spazzatura. La raccolta di spazzatura però non gestisce altre cose oltre la liberazione della memoria. Si può tuttavia specificare delle opere azioni da eseguire quando viene effettuata la raccolta di spazzatura. Ovvero si può inserire un metodo finalize che verrà chiamato dal sistema immediatamente prima di effettuare la raccolta di spazzatura. Tipicamente si aggancia a questo metodo la liberazione di risorse del sistema. Nell'esempio che abbiamo fatto del file, la chiusura dello stesso può avvenire in questo metodo.

# 10 Static e Final

---

## ● Le classi possono avere campi e metodi *static*

Esaminiamo adesso alcune modifiche nel meccanismo di persistenza delle classi. Abbiamo detto che normalmente un oggetto viene creato con new, e viene buttato via quando nessuno lo utilizza più, liberando alla memoria occupata. È possibile cambiare questo comportamento, creando campi che non richiedono la allocazione di memoria per essere utilizzati. Ci sono anche metodi che possono essere invocati senza che occorra una istanza. Stiamo parlando di campi e metodi static.

## ● Le classi possono avere campi e metodi *final*

Normalmente i campi sono modificabili, e i metodi possono essere sostituiti con altri usando i meccanismi che vedremo nella ereditarietà.

### ● I campi possono essere *final*

- cioè non possono essere modificati dopo l'inizializzazione.

È possibile dichiarare un campo final, non modificabile, con la dichiarazione final. In questo modo un campo può essere solo inizializzato, e basta.

### ● I metodi possono essere *final*

- non ridefinibili con l'ereditarietà

Allo stesso modo un metodo final non può essere ridefinito.

## 10.1 Campi statici

- I campi statici equivalgono alle variabili globali

```
class Stack {  
    static int count = 0;  
    int top=0;  
    int [] stack;  
    Stack() {  
        ++count;  
        stack = new int[10];  
    }  
}
```

- c'è un solo count, condiviso tra tutti gli oggetti

Vediamo di capire il meccanismo dei campi statici con questo esempio: in questo caso abbiamo il classico stack. Supponiamo che per qualche motivo ci interessa sapere quanti stack sono stati creati. Occorre una variabile che li conti, ma questa variabile non può essere un campo normale perché ne esiste una copia diversa per ogni istanza. Si potrebbe pensare di metterla da qualche altra parte, e non della classe, ma siccome concettualmente appartiene alla classe, la soluzione migliore è quella di usare un campo statico. Il campo count, essendo static, diventa unico per tutte le istanze della classe. Al campo si accede allo stesso modo con cui si accede agli altri campi, ma la differenza è che esiste un unico campo condiviso tra tutte le istanze. In questo modo funziona il conteggio: ogni volta che viene creato una istanza, questa incrementa l'unica occorrenza condivisa del campo count.

- basta il nome della classe per accedere ad un campo statico

```
System.out.println("hello");
```

- out è campo statico di System

I campi statici hanno anche una ulteriore proprietà: possono essere utilizzati senza istanza. Infatti non vengono memorizzati nel corpo di una istanza, ma separatamente in uno spazio di memoria comune per tutti i campi statici della classe. Per accedere a un campo statico basta il nome della classe. Se ricordate come si fa a scrivere nel terminale, si utilizza System.out. Come si vede si sta utilizzando out, che è un campo statico, e per utilizzarlo basta il nome della classe a cui appartiene (System). È comunque possibile utilizzare una istanza per accedervi, ma viene comunque modificato l'unico campo statico condiviso.

## 10.2 Blocco static

### ● Inizializzazione statica

- un campo statico è una variabile condivisa tra le istanze di una classe
- i campi statici vengono inizializzati al caricamento della classe

Anche un campo statico può essere inizializzato. Bisogna tenere presente che nella inizializzazione non si può accedere a campi non statici. C'è anche una differenza temporale: il campo statico viene inizializzato può quando la classe viene caricata. Spesso la classe viene caricata quando creo una istanza. Ma in realtà la classe viene caricata la prima volta che la utilizzo, e posso utilizzarla, come nel caso di System.out, anche senza creare alcuna istanza.

### ● Esistono i blocchi static

- eseguiti al caricamento della classe

```
class Quadrati {  
    static int[] a = new int[10] ;  
    static {  
        for(int i=0; i<10; ++i)  
            a[i]=i*i;  
    }  
}
```

Poiché le espressioni di inizializzazione hanno i loro limiti, è possibile effettuare delle inizializzazioni complesse dei campi statici grazie ai blocchi statici, come mostrati in figura. Si tratta di codice che viene eseguito al caricamento della classe, e che può fare riferimento solamente a campi statici.

## 10.3 Metodi statici

### • equivalgono alle funzioni

- basta il nome della classe per accedere al metodo

```
int n = Integer.parseInt("123");
```

Abbiamo visto che per usare un metodo che serve sempre una istanza. Questo può essere una limitazione perché alcune procedure sono funzioni: prendono un input e producono un output, senza che debba essere memorizzato uno stato. È fastidioso dover creare ogni volta una istanza. Per fortuna è possibile dichiarare un metodo in modo che non richieda una istanza per essere utilizzata: si tratta dei metodi statici. Come i campi statici, possono essere invocati solamente utilizzando il nome della classe, senza che occorra avere alcuna istanza. Nel listato possiamo notare il metodo parseInt, che serve a trasformare una stringa in un numero intero, e che possiamo invocare semplicemente specificando il nome della classe. Per motivi se vogliamo "organizzativi", i metodi statici devono stare comunque in una classe, ma per utilizzarli basta il loro nome preceduto dal nome della classe. In realtà il metodi statici hanno qualcosa in meno rispetto ai metodi non statici: non possono accedere campi non statici e quindi hanno una minore capacità. Tuttavia sono utili e molto utilizzati.

### • main è un metodo static

```
public static  
void main(String[] args)  
{ ... }
```

Il metodo di avvio di una classe è appunto statico. Infatti l'interprete Java inizia il suo lavoro a partire da un nome di classe: la carica, ma non ne crea alcuna istanza. Per questo motivo l'esecuzione parte da un metodo statico, che deve chiamarsi main, non deve ritornare nulla, e prendere un array di stringhe come argomento.

## 10.4 Visibilità

### ● Visibilità:

- dai metodi, si vedono campi e metodi static e non static
- dai metodi static, si vedono solo i campi e metodi static
  - i metodi static *non hanno* un `this`

```
class C {  
    int x;  
    static int y;  
    int f()  
    { y=1; } // si  
    static int g()  
    { x=1; } // no  
}
```

Dai i metodi statici non sono accessibili i campi e i metodi non statici, ma dai metodi normali sono accessibili sia i campi che i campi e i metodi statici. Demente bisogna tenere presente che un metodo accede ai campi della sua istanza, e ai campi statici condivisi con tutte le altre istanze della classe.

## 10.5 Costanti

### ● Campi final

- non può essere modificato dopo l'inizializzazione

```
final int max=1;
```

### ● Costanti in Java: campi static final

```
class Limit {  
    static final int MAX=999;  
    static final int MIN=0;  
}
```

In Java abbiamole costanti: abbiamo parlato di campi final, che non possono essere modificato dopo l'inizializzazione. Ma i campi final non sono statici, per cui avremmo una copia diversa di un campo non modificabile per ogni istanza. In realtà le vere costanti in Java solo i campi contemporaneamente static e final.

### ● usate come `Limit.MAX`, `Limit.MIN`

- notare che occorre sempre specificare la classe con le costanti

Le costanti si usano come i campi statici, usando come prefisso il nome della classe. Il compilatore effettua delle ottimizzazioni quando si dichiara un campo statico e finale contemporaneamente. Vedremo che i campi static final sono gli unici a poter essere definiti nelle interfacce, che sono come dei gusci di classe senza contenuto.

# 11 Package

## ● Ogni classe ha un "nome lungo"

- nel nome va specificato il package cui appartiene

```
java.lang.String  
java.util.Vector  
java.io.InputStream
```

Finora abbiamo fatto riferimento a classi utilizzando il loro nome abbreviato. In realtà le classi hanno tutte un nome lungo che comprende anche il nome del package a cui appartengono. Nel listato precedente sono elencati i veri nomi di classi quali String, Vector e InputStream. Le librerie infatti sono composta da un ampio numero di classi che vengono suddivise per comodità di consultazione in package.

## ● È possibile abbreviare il nome con import

```
package hello;  
import java.util.Vector;  
import java.lang.*;
```

```
class Hello { void hello () { String s = "hello"; // invece di java.lang.String Vector v = new Vector(); // invece di java.util.Vector } }
```

Ogni classe è opportuno che sia posta in un package. Si tratta di un modo di classificarle. Per collocare una classe in un package occorre mettere come prima riga di un file la dichiarazione del package a cui appartiene. Nell'esempio vediamo che la classe Hello è stata posta nel package "hello". Il suo nome completo è dunque hello.Hello.

Normalmente si usa il nome abbreviato di una classe. Il punto è: come si fa ad abbreviare il nome di una classe? Basta inserire all'inizio di un file la dichiarazione "import".

## 11.1 import

### ● **import java.util.Vector;**

- in realtà: fa considerare Vector abbreviazione di `java.util.Vector`

La dichiarazione import permette di riferirsi alla classe utilizzando il nome breve senza l'indicazione del package a cui appartiene la classe.

### ● **senza import**

```
Vector v = new Vector();  
// Class not found in compilazione  
java.util.Vector=new java.util.Vector();  
// è necessario
```

Se non avessimo fatto questa dichiarazione, per creare l'oggetto Vector e per dichiarare variabili per tipo Vector occorreva utilizzare il nome completo `java.util.Vector`. È importante capire che questa dichiarazione non inserisce niente nel testo del programma compilato: semplicemente permette di abbreviare il nome della classe. Di solito si fanno delle classi che nascono in un package e poi migrano ad altri package. Usare la dichiarazione dei package e in maniera consistente permette di correggere il codice semplicemente cambiando le dichiarazioni package e import all'inizio di ogni file. Usare nel testo di un programma il nome lungo complica questi spostamenti abbastanza comuni.

## 11.2 import \*

### ● import java.util.\*;

- importa tutte le classi del package

- Fa cercare le classi non trovate in java.util

A volte sono numerose le classi che devono essere utilizzate, m in generale tendono a trovarsi in pochi package. È possibile importare un intero package: si utilizza l'asterisco al posto del nome della classe. In ogni caso come l'importazione non di inserisce nulla, anche l'uso della importazione globale non inserisce nulla ma comunica al compilatore di utilizzare il package indicati per ricercare classi che non sono state trovate. Normalmente è possibile utilizzare direttamente classi che si trovano allo stesso package della classe senza impportarle.

### ● Possibili collisioni

```
import java.util.*;
import java.sql.*;
// ambiguo: java.util.Date o java.sql.Date?
Date d = new Date ();
// ERRORE
import java.util.*;
import java.sql.*;
import java.sql.Date;
java.util.Date=new java.util.Date(); // usa
java.util.Date
Date = new Date(); // usa java.sql.Date
```

Esiste la possibilità di collisioni: per esempio se si importano i due package java.util.\* e java.sql.\*, troviamo che la classe Date si trova in entrambi i package. Allora quale dei due deve essere utilizzata: il compilatore segnala un errore. In un questi casi o sui usa esplicitamente il nome lungo, oppure si può disambiguare utilizzando dopo la import con \*, anche una import specifica, come nell'esempio.

### ● Sempre implicito

```
import java.lang.*;
```

- per System, String, Thread, etc non occorrono import

Alcune classi vengono automaticamente importate: in particolare è sempre implicito un import java.lang.\*; in questo modo non è necessario importare classi di uso molto comune come quelle già dette.

## 11.3 Classpath

### ● un package corrisponde ad una directory

- una classe corrisponde ad un file .class nella directory del package

- Esempio: `java.util.Vector` equivale a:

```
File DOS : java\util\Vector.class  
File UNIX: java/util/Vector.class
```

Abbiamo visto il nome del package, che è gerarchico: solitamente sono nomi in minuscolo separati da un punto. In realtà un package è una directory. Il sistema cerca le classi in una directory che ha lo stesso nome del package a partire da le directory nel cosiddetto CLASSPATH. La barra specifica del sistema sostituisce il punto. Per esempio nei sistemi UNIX al posto del punto si utilizza la barra diretta, nei sistemi basati sul DOS si utilizza la barra inversa.

### ● CLASSPATH:

- **Regola: una classe viene cercata nella sottodirectory corrispondente a partire dalle directory nel classpath**

Vediamo adesso le regole che vengono utilizzate per localizzare una classe quando questa viene richiesta. Abbiamo visto che Java carica le classi in memoria quando qualcuno vi fa riferimento. Le classi hanno un nome gerarchico che comprende il loro package. Il package serve a anche per indicare in quale sotto directory bisogna cercare la classe. Le directory da cui parte la ricerca sono elencate nel CLASSPATH. Questo è un parametro che viene fornito all'interprete sulla riga di comando oppure usando lo variabile di ambiente CLASSPATH.

- se `CLASSPATH=c:\java;c:\java\lib`

- allora `java.util.Vector` viene cercata in:

```
1. c:\java\lib\java\util\Vector.class  
2. c:\java\util\Vector.class
```

Quindi se io faccio riferimento alla classe java.util.Vector e ho nel CLASSPATH le directory C:\java e C:\java\lib, si controllerà che non esista un file C:\Java\java\util\Vector.class o un file C:\Java\lib\java\util\Vector.class. Fare attenzione che questo meccanismo a volte sfugge ai principianti.

- il package di default (senza dichiarazione di package) non ha una sottodirectory

- le classi senza package vengono cercate nelle directory del classpath

Il package di default è quello che si ha se si ha omesso la dichiarazione package, corrisponde alla radice di ogni classe nel CLASSPATH.

## 11.4 L'interprete

### ● l'interprete standard del JDK (java):

- se non è specificato il classpath, cerca nella directory corrente

```
c:>javac Hello.java
c:> dir /b
Hello.class
Hello.java
c:> java Hello
Hello!
```

Consideriamo un caso semplice, illustrato nell'esempio: supponiamo che creo un file Hello.java senza specificare il package e lo compilo. Ottengo nella mia directory corrente il file Hello.class. Siccome se non specifico altrimenti la directory corrente si trova nel classpath, scrivendo java Hello la cosa funziona. Se avesse messo il package "hello", avrei dovuto spostare Hello.class nella sottodirectory hello della directory corrente, e chiamare programma con java hello.Hello .

### ● Aggiungere directory al classpath (con JDK > 1.2)

- il modo più semplice è usare lo switch -cp (o -classpath)
  - sparisce la directory corrente
  - Molti altri modi: p.e. con JDK 1.1 occorre usare jre -cp

Il classpath può essere impostato in molti modi, compreso l'uso della variabile di ambiente CLASSPATH che però sconsiglio perché può causare effetti collaterali in altri programmi. Invece è meglio specificare le directory del classpath utilizzando lo switch -cp che è stato introdotto in Java a partire della versione 1.2. Le versioni precedenti avevano modi diversi e talora inconsistenti di impostare il classpath. Per semplicità ometto la descrizione di questi casi.

## 11.5 Zip e Jar

### ● Nel CLASSPATH ci possono essere archivi

- Sono in formato zip
  - Estensioni zip o jar

È importante sapere che nel classpath possono comparire anche dei file. Queste file devono però essere in formato zip, oppure in formato jar. Quest'ultimo è un formato molto simile allo Zip, tanto è vero che usando utility comuni come WinZip è possibile aprire questi file per vederne il contenuto.

### ● Archivi trattati come directory

- Le sottodirectory devono essere specificate correttamente  
nello zip

Questi file si comportano come delle vere e proprio directory agli fini della ricerca di classi. Per cui occorre stare attenti ai path dei file che vengono archiviati. Se per esempio ricercò java.util.Vector, questa deve essere memorizzata nell'archivio come java/util/Vector.class (il formato zip usa internamente le barre diritte per separare i componenti di un file).

### ● Riutilizzare codice esistente

Le basi su cui poggia la programmazione oggetti abbiamo detto essere tre: encapsulazione, ereditarietà e polimorfismo. Finora abbiamo esaminato in dettaglio l'incapsulazione. Vediamo adesso gli altri due piedi su cui poggia la programmazione oggetti. È importante capire che lo scopo di quando vedremo in questa sezione è quello di consentire il massimo riutilizzo di codice esistente.

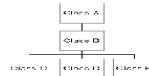
- Ereditarietà: consente di riutilizzare i dati esistenti, aggiungendone di nuovi.

Innanzitutto abbiamo l'ereditarietà: si tratta della possibilità di prendere una classe con tutti gli annessi connessi, e di creare una nuova che possiede nuovi campi e nuovi metodi. Questa nuova classe però eredita tutti i campi e tutti gli metodi della classe su cui si basa.

- Polimorfismo: consente di ridefinire il comportamento

Il meccanismo di ereditarietà però ha il limite di doversi tenere una classe esistente così come è. Succede di frequente però che la classe che vogliamo a volte caratteristiche di una classe esistente, ma qualcosa va cambiato. Fortunatamente il terzo piede della programmazione oggetti pone rimedio questo limite, consentendo al programmatore di sostituire comportamenti esistenti (ovvero metodi) con altri. Il nome polimorfismo viene dal fatto che un oggetto viene considerato come un mutante, il cui comportamento può variare a pari richiesta. Per capirci se noi diciamo a un uccello di volare, questo volerà in un certo modo, che però è diverso dal modo di volare di un aereo. Ma tutte due possono avere in comune il fatto di essere dei volatili . In generale si può pensare di chiedere a un volatile di volare: se è un aereo volerà in un modo se un uccello volerà in un altro.

# 12 Ereditarietà



Nella precedente figura abbiamo rappresentato una gerarchia di classi. Utilizziamo una terminologia che deve essere ben compresa perché viene usato di frequente . Vediamo alcuni casi:

- **A è la classe base di B, e B estende A**

La classe A si dice essere la classe base di B, ovvero la classe da cui è derivata da classe B.

Viceversa si dice che è B è una estensione di A.

- **C D E sono discendenti di B ma anche di A**

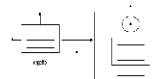
Estendendo quanto detto, per classi C D E sono discendenti della classe A, ma anche della classe B.

- **A e B sono antenate di C, D ed E**

Viceversa possiamo dire che le classi A e B sono antenate di C D E.

## 12.1 L'idea

Cerchiamo di capire qual è l'idea sottostante al meccanismo che stiamo definendo. Vediamo uno schema che rappresenta il meccanismo dell'ereditarietà.



### Nuove classi di oggetti che modificano quelli esistenti

- oggetto = stato(campi) + operatori(metodi)

È come se intorno a un dato oggetto, lo ricoprisse con un nuovo strato, che contiene nuovi campi e nuovi metodi.

#### nuovo oggetto

- stato esteso (nuovi campi) [ereditarietà]
- metodi ridefiniti (metodi con lo stesso nome) [polimorfismo]

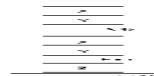
Il nuovo oggetto estende lo stato aggiungendo informazione e cambia il comportamento dei metodi.

Ricordiamo che i campi rappresentano lo stato di un oggetto, per cui il nuovo oggetto ha uno stato esteso, con nuove informazioni mantenute. I metodi possono essere nuovi oppure avere lo stesso nome di metodi già esistenti. Nel caso si tratti di metodi nuovi, interviene l'ereditarietà ampliando i comportamenti preesistenti dell'oggetto. Ma nel caso in cui stiamo definendo dei metodi che hanno lo stesso nome e gli stessi parametri di un metodo già esistente, in questo caso interviene il polimorfismo che sostituisce il comportamento del metodo preesistente.

## 12.2 Esempi

### ● Punto e Punto3D

Possiamo fare immediatamente un semplice esempio mostrando come riutilizzare un oggetto di classe Punto già esistente. Si tratta di un punto bidimensionale, che mantiene le sole informazioni di coordinata x e coordinate y.



```
class Punto {  
    int x;  
    int y;  
}  
class Punto3D extends Punto {  
    int z;  
}
```

L'estensione naturale di questa classe è il punto tridimensionale. Ovviamente vogliamo riciclare il punto preesistente, mantenendo le informazioni che già contiene. Quindi il punto tridimensionale estende il punto, aggiungendogli una nuova coordinata z.

### ● un punto 3D è un punto

● Punto p= new Punto3D();

Il nuovo oggetto è ancora a tutti gli effetti una istanza del vecchio oggetto, perché lo contiene integralmente. Per cui possiamo assegnare a variabili di una classe istanze di classi derivate. Questo non deve stupire: è assolutamente la prassi. Concettualmente a tutti gli effetti il punto tridimensionale è ancora un punto bidimensionale, che contiene tutte sue le informazioni e i metodi. E in più qualcosa è stato aggiunto.

## 12.3 Estensione

### ● Come estendere una classe Pesce

Generalmente si parte da una classe di tipo generico, che ha delle caratteristiche generali per un determinato tipo di oggetti, e la si estende allo scopo di specializzarla.

Consideriamo una classe Pesce, che avrà delle caratteristiche generali, come la capacità di nuotare ad una certa velocità.

```
class Pesce {  
    int velocita = 10;  
    void setVelocita(int x) { velocita = x; }  
    void nuota() { ... }  
}
```

### ● e specializzarla in un PesceRosso

Se adesso decidiamo di specializzarla, per estenderne in un pesce rosso, dovremmo innanzitutto mantenere l'informazione sul colore, e per di più possiamo voler decidere di cambiare la tonalità di rosso (non tutti pesci rossi hanno lo stesso identico colore).

```
class PesceRosso extends Pesce {  
    int colore = Color.red;  
    void setTonalita() { ... }  
}
```

### ● La classe PesceRosso

- ha tutti i campi e i metodi di Pesce
- in più un nuovo campo colore e un nuovo metodo setTonalita()

La nuova classe ha ancora tutte le caratteristiche di quella preesistente, e ha in più la capacità di ricordarsi suo colore, e la possibilità di cambiare la tonalità di rosso.

## 12.4 Ridefinizione

- un metodo con lo stesso nome e stessi parametri ridefinisce un metodo della classe base

un

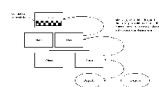
```
class Component {  
    boolean handleEvent (Event e) { ... }  
}  
class Window extends Component {  
    boolean handleEvent (Event e) { ... }  
}
```

- con argomenti di tipo diverso è estensione, non ridefinizione
  - non si può ridefinire con valore ritornato diverso

## 12.5 Invocazione

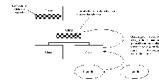
### ● Chiamando un metodo di una classe:

- viene risalita la gerarchia delle classi
- si cerca un metodo con il nome e il tipo di parametri corrispondenti



- la ricerca parte dalla classe finale
- In questo modo un metodo può essere ridefinito.
  - Quando viene ridefinito verrà chiamato soltanto il nuovo.

## 12.6 super.metodo



- Se occorre chiamare il metodo originario, si usa **super**

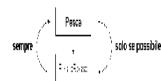
```
boolean handleEvent(Event e) {  
    if (e.id==e.WINDOW_DESTROY)  
        System.exit(0);  
    else return super.handleEvent(e);  
}
```

- Chiamando un metodo con **super** *non* si applica il polimorfismo
  - altrimenti si andrebbe in ciclo

## 12.7 Conversioni

### ● Conversioni:

- una PesceRosso è **sempre** un Pesce
- ma un Pesce non è (in generale) un PesceRosso



```
Pesce p = new PesceRosso();  
PesceRosso s = (PesceRosso) p;
```

- la conversione è possibile solo se effettivamente un dato **Pesce** è in realtà un **PesceRosso**

## 12.8 Object

- Tutti gli oggetti estendono un altro oggetto

- Se non si estende esplicitamente una classe, allora si estende (implicitamente) Object

- Object è l'unico oggetto che non estende nessun altro oggetto

- Tutti gli oggetti ereditano (direttamente o indirettamente) da Object

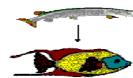
- per cui i suoi metodi sono presenti in tutti gli oggetti Java.

- Ogni oggetto può essere convertito ad Object

- infatti quando occorre riferirsi a *qualsiasi* oggetto si utilizza un riferimento ad Object.

<i>Metodo</i>	<i>Descrizione</i>
equals(Object)	Confronto di oggetti
hashCode()	Codice HASH di un oggetto
toString()	Conversione in stringa
clone()	Duplicazione di un oggetto
getClass()	La classe di un oggetto
finalize()	Finalizzazione

# 13 Poliformismo



## ● Dati un Pesce e un PesceRosso

```
class Pesce {  
    void nuota() { ... }  
}  
class PesceRosso extends Pesce {  
    void nuota() { ... }  
}
```

## ● Posso assegnare un PesceRosso ad un Pesce

```
Pesce p= new PesceRosso();
```

## ● Se eseguo p.nuota(), cosa viene chiamato?

- nuota() di Pesce ?
- o nuota() di PesceRosso ?

## 13.1 Metodi virtuali

- Viene chiamato il metodo:
  - `nuota()` di PesceRosso
- **Polimorfismo:**
  - ogni oggetto risponde in autonomia ai messaggi
    - ogni metodo è virtuale
  - in Java conta ciò che è, non ciò che sembra
    - il tipo dell'istanza, non della variabile
  - Meccanismo base per la ridefinizione a run-time

## 13.2 Esempio

### ● Un Acquario

```
class Acquario
{
    // un array di nuotatori
    Pesce[] elements;
    // aggiungo un pesce
    void add(Pesce n) {...}
    // uso il Pesce anche
    // senza averlo mai visto
    void refresh() {
        //...
        elements[i].nuota();
    }
}
```

### ● Può contenere vari tipi di Pesce

```
Acquario a = new Acquario();
a.add(new PesceRosso());
a.add(new PesceSpazzino());
a.refresh();
```

### ● I pesci aggiunti "nuoteranno" a loro modo

- per l'acquario sono tutti pesci che nuotano
- il polimorfismo li fa "nuotare" ognuno a suo modo

## 13.3 Gestione Eventi

### ● Gestione Eventi (1.0)

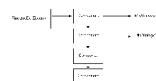
- Il sistema conosce solo Component con handleEvent()

- Le finestre (Window) sono derivate da Component

- Gestione eventi, ereditando e ridefinendo handleEvent

- La libreria AWT *non* può avere idea di come il programmatore gestirà gli eventi

- Grazie al poliformismo, l'evento viene smistato alla handleEvent ultima definita dal programmatore.



## 13.4 Costruttori

- I costruttori non si ereditano: vanno ridichiarati, *uno per uno*
  - Se non ho un costruttore viene implicitamente dichiarato il costruttore di default
    - ovvero il costruttore senza argomenti con un corpo vuoto.
- Ogni costruttore *per prima cosa* costruisce la classe base
  - fa questo chiamando uno dei costruttori della classe base
  - il primo comando di un costruttore è `super(...)`
    - implicitamente o esplicitamente.
  - Infatti, se manca `super()` ne viene aggiunta una
- Eccezioni:
  - classe `Object`
  - viene utilizzato `this(...)` come primo comando
    - lo farà un altro dei costruttori

```
class B{
    B() { ... }
    B(int x) { ... }
}
class D extends B {
    D() {super(0); }
    //chiama B(int)
    D(int x) {this.x=x; }
    //chiama B()
    D(String s) {
        this(Integer.parseInt(s));
    }
    //chiama D(int)
}
```

## 13.5 Ordine

- Gli oggetti vengono costruiti in questo ordine:

- (1) Prima la classe base
- (2) Poi vengono inizializzati i campi
- (3) Infine si esegue il costruttore

- Esempio:

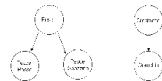
```
class D extends B {  
    int x=1;  
    int y;  
    D() {  
        super(1);  
        y=2;  
    }  
}
```

- Ordine:

- (1) `super(1)`
- (2) `int x=1;`
- (3) `y=2;`

# 14 Interfacce

## ● Il Problema



- Vogliamo inserire un Granchio in un Acquario
  - Un Acquario contiene solo Pesci
- Non possiamo rendere Pesce anche un Granchio
  - Java fornisce solo l'ereditarietà singola
  - e Granchio è già un Crostaceo)

## ● Che facciamo?

## 14.1 Astrazione

### ● Non c'è soluzione

- Dobbiamo modificare il codice
  - la prossima volta ci penseremo prima



### ● Definiamo una interfaccia Nuotatore

- Le interfacce sono classi "vuote"
- Mettiamo nell'acquario dei Nuotatori, anzichè Pesci
  - Java supporta l'ereditarietà multipla di interfacce

## 14.2 Dichiarazione

### ● Dichiarazione di interfaccia:

```
interface Nuotatore {  
    public static final int K=1;  
    public void nuota();  
}
```

- "signature" dei metodi (dichiarazioni)
- solo campi *static final* (costanti)

## 14.3 implements

### ● Una classe supporta una interfaccia

- ovvero si comporta come dichiarato dall'interfaccia

```
class Pesce implements Nuotatore {  
    public void nuota(){...}  
}  
class Granchio implements Nuotatore {  
    public void nuota(){...}  
}
```

- Se si implementa una interfaccia si devono definire tutti i metodi

## 14.4 Uso

### ● Solo riferimenti a interfacce

- Non è possibile costruire oggetti di un tipo interfaccia

```
new Nuotatore(); //no!
```

- È però possibile dichiarare variabili di un tipo interfaccia:

```
Nuotatore p = new PesceRosso();
Nuotatore g = new Granchio();
```

- È quindi possibile chiamare i metodi di una interfaccia:

```
p.nuota();
g.nuota();
```

## 14.5 Genericità

- Una interfaccia è un *modello* del comportamento di un oggetto
- Si può programmare senza nulla conoscere della effettiva implementazione

```
class Acquario
{
    // un array di nuotatori
    Nuotatore[] elements;
    // aggiungo un nuotatore
    void add(Nuotatore n)
    {
        ...
    }
    // astrazione
    void refresh() {
        ...
        elements[i].nuota();
    }
}
```

## 14.6 abstract

### ● Classi astratte

```
abstract class Crostaceo {  
    abstract void cammina();  
}
```

- se un metodo è **abstract** lo deve essere la classe
- non si possono creare istanze
- ereditarietà singola
  - sono classi solo parzialmente implementate

# 15 Visibilità

## ● Modificatori che alterano la visibilità

Attributo	HTML	PHP	SQL	Java
Visible	visible	visible	visible	visible
Visible on mouse over	visible	visible	visible	visible
Visible on mouse out	visible	visible	visible	visible
Visible on click	visible	visible	visible	visible



Il tuo sito è sempre uguale ? **EpNuke** la soluzione modulare professionale per il Web Publishing. Soluzioni **tutto compreso** con migrazione della grafica, hosting e assistenza.

Per informazioni [www.eproneteus.com](http://www.eproneteus.com) - 174/225

## 15.1 Classi

● Una classe può essere `public` o default (nessun modificatore):

- Una classe `public` è visibile da altri package.
  - una classe `public` deve stare in un file con il suo stesso nome.
- Una classe senza modificatori è visibile solo nello stesso package

## 15.2 Membri

- Un membro di una classe (metodo o campo) può avere:

- **public**
  - visibile anche all'esterno del package
- **protected**
  - visibile soltanto da classi derivate
- **<default>**
  - visibile soltanto all'interno del package
  - nessun modificatore
- **private**
  - visibile soltanto all'interno della classe

- Cosa sono:

```
f () { ... g () ... } %%
g () { ... h () ... } %%
h () { ... ECCEZIONE! ... }
```

- Abbiamo f() che chiama g() che chiama h()
- A un certo punto in h() scatta una eccezione
  - se l'eccezione non viene gestita, *viene causato un return*
  - e dal punto di chiamata *viene sollevata una eccezione.*
- Questo meccanismo viene iterato:
  - in h() si ritorna in g(), che causa un ritorno in f() che a sua volta causa un ritorno in f()
- Se nessuno gestisce l'eccezione il programma termina
  - viene stampato un messaggio di errore.

# 16 Eccezioni

- Eccezione: un oggetto che implementa Throwable
  - eccezioni utente derivate da Exception
    - Per convenzione le eccezioni hanno un nome che termina in Exception
- Come sollevare una eccezione:

```
throw new Exception ("disastro!");
```

- Una eccezione è un valore ritornato
  - quindi se viene sollevata una eccezione bisogna dichiararlo:

```
void f()  
throws Exception {  
//...  
throw new Exception();  
//...  
}
```

## 16.1 throws

- Il compilatore si accorge delle eccezioni contenute in un metodo
  - esamina i throw e le dichiarazioni dei metodi
  - obbliga a mettere tutti i throws necessari

```
void g()
throws Exception
{ ... }
// poichè f() contiene g()
// può sollevare le eccezioni di g()
// e questo va dichiarato
void f()
throws Exception {
    //...
    g();
    //...
}
```

## 16.2 uncaught

### ● Eccezioni molto frequenti

#### ● NullPointerException

- a ogni accesso a metodo o campo

#### ● ArrayIndexOutOfBoundsException

- ad ogni indicizzazione di array

#### ● ClassCastException

- ad ogni cast

#### ● Queste sono "uncaught"

- ovvero non bisogna dichiararle con throws

## 16.3 Regola

- Le eccezioni, o si propagano o si catturano

- Per ignorare una eccezione:

```
try {
    f();
} catch (Exception ex) {
    ex.printStackTrace();
}
```

- A volte non si può propagare una eventuale eccezione

- occorre catturarla e ignorarla

- In ogni caso è meglio lasciare *sempre* traccia di una eccezione ignorata

## 16.4 Sintassi

- La sintassi completa della gestione eccezioni è:

```
try {
    // codice che può sollevare eccezioni
} catch (PrimaException ex) {
    // gestione di PrimaException
} catch (SecondaException ex) {
    // gestione di SecondaException
} finally {
    // pulizie finali
}
```

## 16.5 try

- Raffiniamo passo passo un esempio
  - un programma che copia un file.
- Gestione "grossolana"
  - si cattura l'eccezione , si termina e si stampa il trace della stessa.

```
try {  
    String fin = args[0];  
    String fout= args[1];  
    InputStream in  
        = new FileInputStream(fin);  
    OutputStream out  
        = new FileOutputStream(fout);  
    int c;  
    while( (c=in.read()) != -1)  
        out.write(c);  
    in.close();  
    out.close();  
} catch (Exception ex) {  
    ex.printStackTrace();  
}
```

## 16.6 eccezioni

- Ci sono diverse eccezioni che possono essere sollevate:
  - `args[0]`
    - throws `ArrayIndexOutOfBoundsException`
  - `FileInputStream(String)`
    - throws `FileNotFoundException`
  - `in.read()`
    - throws `IOException`

## 16.7 catch

### ● Possiamo raffinare:

```
try {
    String fin = args[0];
    //...
    InputStream in = new FileInputStream(fin);
    //...
} catch(ArrayIndexOutOfBoundsException ex) {
    System.out.println("bad args");
} catch(FileNotFoundException ex) {
    System.out.println("file not found");
}
```

### ● Si deve inserire una catch()

- parametro di tipo compatibile (cioè uguale o derivato)

## 16.8 matching

### ● Ricordiamo:

- Pesce p = new PesceRosso() ma non PesceRosso p = new Pesce()
- si applica la stessa regola alle catch
- Inserendo catch(Exception ex) si intercettano tutte le eccezioni utente
- altre eccezioni non derivate da Exception
  - derivate da RunTimeException e da Error
  - non devono essere catturate

```
try {  
    InputStream in =new FileInputStream (fin);  
    ...  
    c=in.read();  
    ...  
} catch(FileNotFoundException ex) { // 1  
    ...  
} catch(IOException ex) { // 2  
    ...  
}
```

### ● Le eccezioni vengono provate in ordine

- bisogna ordinare le catch()
  - le eccezioni più generali stiano in fondo
- Scambiando 1 e 2, la 1 non viene più raggiunta
  - FileNotFoundException deriva da IOException e quindi cattura e propaga la prima.

## 16.9 propagazione

- Se nessuna catch cattura l'eccezione, questa viene propagata
- Una catch può catturare l'eccezione, esaminarla e risollevarla

```
catch(SpecialException ex) {  
    if(<non-gestibile)  
        throw ex;  
}
```

- Una catch può trasformare l'eccezione

```
catch(SpecialException ex) {  
    if(<non-gestibile)  
        throw new OtherException (ex);  
}
```

- Trasformazione utile con l'ereditarietà

## 16.10 finally

- Il blocco *finally* viene eseguito in qualunque modo il controllo lasci il blocco *try*...

- Sia che la *try* completi normalmente
- Sia che venga sollevata una eccezione e intercettata da una *catch*
- Sia che venga eseguito un *return*

```
try{  
    //...  
} catch (...) {  
    //...  
} catch(...){  
    //...  
} finally {  
    // sempre  
}
```

- Il blocco *finally* viene utilizzato per effettuare pulizie finali.

## 16.11 FileCopy

---

● L'esempio completo:



Il tuo sito è sempre uguale ? **EpNuke** la soluzione modulare professionale per il Web Publishing.  
Soluzioni **tutto compreso** con migrazione della grafica, hosting e assistenza.

Per informazioni [www.eprometeus.com](http://www.eprometeus.com) - 188/225

```
public class Eccezioni {
    public static
    void main(String [] args) {
        InputStream in=null;
        OutputStream out=null;
        try {
            int c;
            in = new FileInputStream(args[0]);
            out = new FileOutputStream(args[1]);
            while( (c=in.read()) != -1)
                out.write (c);
        } catch(
        ArrayIndexOutOfBoundsException ex) {
            System.out.println(
                "not enough args");
        } catch(
        FileNotFoundException ex) {
            System.out.println(
                "file not found");
        } catch (
        IOException ex) {
            System.out.println(
                "I/O error");
        } finally{
            try { in.close(); }
            catch(Exception ex) { }
            try { out.close(); }
            catch(Exception ex) { }
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

## 16.12 ereditarietà

### ● Relazione tra eccezioni e ereditarietà

```
class Pesce {  
    void nuota()  
        throws PesceMortoException  
    {  
        //...  
    }  
}  
class PesceAtomico extends Pesce {  
    void nuota()  
        throws PesceMortoException,  
            EsplosioneException // NO!  
    {  
        //...  
    }  
}
```

## 16.13 trasformazione

- I metodi ridefiniti non possono aggiungere nuove eccezioni
  - Altrimenti codice prima funzionante non funzionerebbe più:

```
class Acquario {  
    ...  
    try {  
        p.nuota() ;  
    }catch (PesceMortoException pme) {  
        ...  
    }  
}
```

- se aggiungo un PesceAtomico, il nuovo pesce potrebbe sollevare nuove eccezioni imprevedibili

- Aggiungere nuove eccezioni *derivando* una eccezione
  - In questo modo il codice esistente rimane valido
- Si possono riconvertire le eccezioni "anomale"

```
class PesceMortoPerEsplosioneException  
    extends PesceMortoException { }  
class PesceAtomico extends Pesce {  
    void nuota()  
        throws PesceMortoPerEsplosioneException  
    {  
        try {  
            <operazione-a-rischio-di-explosione>;  
        } catch (EsplosioneException ex) {  
            throw new PesceMortoEsplosioneException() ;  
        }  
    }  
}
```

## ● Programmazione parallela, analizzando processi e thread.

- Differenze tra il tradizionale multiprocessing e il più moderno multithreading.
- Esamineremo poi come si fa a creare dei thread
- Tratteremo infine i problemi della sincronizzazione

# 17 Thread

---

## ● Multiprocessing

- Il multiprocessing è caratterizzato da:
  - più processi in esecuzione girano in spazi di memoria separati
  - comunicazione attraverso primitive (pipe, socket)
  - inefficienza dell'interazione (praticamente sono processi in rete)
  - maggiore protezione

## ● Multithreading

- Il Multithreading è caratterizzato da:
  - più thread in esecuzione girano nello stesso spazio di memoria
  - comunicazione attraverso memoria condivisa
  - comunicazione molto efficiente e potente
  - problemi di sincronizzazione

## 17.1 Creazione

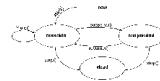
### ● Come si crea un Thread:

- (1) Estendere la classe Thread
- (2) Ridefinire il metodo run()
- (3) Avviare il thread con start()

```
class Contatore1
    extends Thread {
public void run() {
    int n = 0;
    while(true) {
        System.out.println(""+n);
        ++n;
    }
}
public static void main(String[] args) {
    new Contatore1().start();
}
```

## 17.2 Stati

- Stati dei Thread sono:



- Un thread appena creato è nello stato *new*, ma non è attivo
- Per attivare un thread, occorre chiamare **start()**, per renderlo *runnable*
- un thread *runnable* ottiene "ogni tanto" il processore (o uno dei processori)
- un thread *runnable* può cedere il passo agli altri con **yield()** rimanendo *runnable*
- Un thread può sospendersi con **wait()**
  - **suspend()** e **resume()** sono deprecati
- Un thread può terminare ritornando da **return**
  - Non riparte con **start()!!!**
  - **stop()** è deprecato

## 17.3 Runnable

- Thread su un oggetto già derivato da qualcosa' altro.
  - immaginiamo una finestra, derivata da oggetto GUI
- Interfaccia Runnable

```
import java.awt.*;
class Contatore2 extends Frame
    implements Runnable {
    Contatore2() {
        ... // inizializza la GUI
        new Thread(this).start();
    }
    public void run() {
        int n = 0;
        while(true) {
            label.setText(""+n);
            ++n;
        }
    }
    public static void main(String[] args) {
        Contatore2().start();
    }
}
```

- Si rende multithreaded un oggetto non estendibile creando un nuovo oggetto thread
- Il thread si "aggancia" all'oggetto originario eseguendo il metodo **run**



- le interfacce sono un sostituto per i puntatori a funzione.

## 17.4 Problema!

● Programmazione concorrente riserva brutte sorprese.

● Consideriamo uno stack:

```
class Stack {  
    int top; int stack = new int[10];  
    void push(int x) { stack[top]=x; top++; }  
    int pop() { return stack[--top]; }  
}
```

● Sequenza sfortunata di due push concorrenti sullo stesso stack:



## 17.5 Perchè?

- Problema tipico della programmazione multithreaded
- L'operazione di push è atomica:



- I metodi portano un oggetto da uno stato consistente ad un altro stato consistente.  
Il multithreading consente ad un thread di accedere ad un oggetto mentre è in uno stato inconsistente.

## 17.6 synchronized

- Garantire che i metodi critici non siano interrotti

- Il problema in Java ha una soluzione semplice:

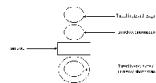
```
class Stack {  
    synchronized void push(int x) { ... }  
    synchronized int pop() { ... }  
}
```

- Garantisce che all'oggetto acceda un thread solo per volta

- Cosa fa in realtà synchronized è una cosa più complessa...

## 17.7 Semafori

- Un oggetto con metodi sincronizzati ha associato un semaforo.



- Quando un thread accede ad un oggetto sincronizzato, sbarra l'accesso a tutti gli altri
- Ogni nuovo thread viene posto in attesa (sospeso)
- Quando un thread finisce l'accesso, viene riattivato il primo in attesa
  - Attenzione: la sincronizzazione ha un impatto *notevole* sull'efficienza.

## 17.8 Osservazioni

- La sincronizzazione di un metodo è legata ad una specifica istanza
  - non ad una classe
- La sincronizzazione di un metodo statico blocca l'accesso a tutti gli altri metodi statici
  - non a quelli dinamici
  - Se accedo ad un oggetto chiamando un metodo (per esempio **push**)
    - sbarro l'accesso all'oggetto utilizzando qualunque altro metodo (per esempio anche **pop**)
- Se si ridefinisce un metodo sincronizzato:
  - il metodo ridefinito non è automaticamente sincronizzato
    - può esserlo, ma va dichiarato esplicitamente
    - può non esserlo; allora chiamando il padre con super si entra in un metodo sincronizzato

## 17.9 wait notify

- Se un thread, mentre accede a un metodo sincronizzato, si accorge di non poter finire il lavoro?

```
class Stack {  
    synchronized void push(int x) {  
        if(top>stack.length)  
            // che faccio ???  
    }  
}
```

- Posso uscire senza fare niente oppure sollevare una eccezione

- La cosa migliore sarebbe aspettare che qualcuno faccia una pop

### La wait() e la notify() fanno parte del linguaggio

- sono metodi di Object.
- wait() ha senso solo in un metodo sincronizzato
  - Sospende il thread corrente e lo pone in attesa

```
class Stack {  
    synchronized void push(int x) {  
        while(top>stack.length)  
            wait();  
        ...  
    }  
    synchronized void pop(int x) {  
        int r = stack[--top];  
        notify();  
        return r;  
    }  
}
```

- il thread viene riattivato non appena qualcuno effettua una notify() sull'oggetto.
- un thread riattivato accede all'oggetto dal punto in cui ha fatto wait()

## 17.10 Attenzione

### ● Il ciclo while non può diventare un if

- non è detto che dopo una `notify()` le condizioni siano soddisfatte

### ● La `notify` riattiva un solo thread

- Per riattivare tutti i thread in attesa si usa `notifyAll()`
- Però è più inefficiente.

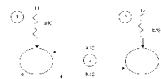
```
class Stack {  
    int[] stack = new int[10]; int top=0;  
    synchronized void push(int x) {  
        while(top>stack.length)  
            wait();  
        stack[top++]=x;  
        notify();  
    }  
}
```

```
synchronized void pop() { while(top==0) wait(); int r = stack[--top]; notify(); return r; }  
}
```

## 17.11 Stallo

### ● Lo stallo: blocco reciproco

- è il principale problema della programmazione concorrente.



- T1 esegue a.f() e blocca a
- T2 esegue b.f() e blocca b
- T1 chiama b.g() ma b è bloccato, e si sospende
- T2 chiama a.g() ma a è bloccato, e si sospende

● STALLO: T1 e T2 sono sospesi ognuno in attesa che l'altro faccia qualcosa.

### ● Stringhe

- StringBuffer, StringTokenizer

### ● Contenitori

- Enumerator, Vector, Hashtable

### ● Input/Output

- File
- Stream

# 18 Stringhe

## ● Costruttori:

```
String()
// crea una stringa vuota
String(String)
// crea una stringa copia di un'altra
```

## ● Metodi:

```
int length()
// ritorna la lunghezza
char charAt(int)
// ritorna l'i-simo carattere
```

## ● Esempi:

```
// codice Cesare (a=d, b=e, etc)
String s = "hello";
for(int i=0; i<s.length(); ++i)
    System.out.print(s.charAt(i)+3);
```

## 18.1 Ricerche

### ● Ricerche

int indexOf(char c [, int start]) int indexOf(String s [, int start]) int lastIndexOf(char c [, int start]) int lastIndexOf(String s [, int start])

- ricerca la prima/ultima posizione del carattere/stringa
  - a partire dalla posizione start; ritorna la posizione o -1

### ● Esempi

```
// Conteggio di asterischi
int count = 0;
for(int curr= s.indexOf('*');
    curr != -1;
    curr = s.indexOf('*', curr))
    ++count;
```



Il tuo sito è sempre uguale ? **EpNuke** la soluzione modulare professionale per il Web Publishing.  
Soluzioni **tutto compreso** con migrazione della grafica, hosting e assistenza.

Per informazioni [www.eproneteus.com](http://www.eproneteus.com) - 206/225

## 18.2 Confronti

- In generale, non si può fare un confronto `s==t`
  - (per fare questo occorre l'internamento)

```
boolean s.equals(t)
s.equalsIgnoreCase(t)
//vale anche per à e À
int s.compareTo(t)
// ritorna -1 se s<t,
// 0 se s=t, 1 se s>t
boolean s.startsWith(String prefix
[, int offset])
boolean s.endsWith(String suffix)
//verifica se combaciano prefix/suffix
//[a partire da offset]
```

- Regioni

```
boolean s.regionMatches(
[boolean ignoreCase,]
int start, int other,
int ostart, int len)
```

## 18.3 intern()

- In generale due stringhe con lo stesso contenuto sono due oggetti diversi

```
String s = "pippo" ;
String t = new String(s);
// ma s != t
// (puntano a oggetti diversi)
```

- È possibile fare in modo che se `s==t` se e solo se `s.equals(t)`
  - Importante per ragioni di efficienza
  - Fondamentale per le Hashtable
- `s.intern()` restituisce un puntatore ad una stringa "internata"
  - tale che se `s.equals(t)` allora `s.intern() == t.intern()`
- Implementazione tramite una tabella

## 18.4 Utility

### ● Utilità delle stringhe

```
String replace(char oldChar, char newChar)
// rimpiazza un carattere con un altro
String toLowerCase()
String toUpperCase()
// to minuscolo/maiuscolo
String trim()
// elimina spazi in testa e coda
String substring(int begin [, int end])
// sottostringa
// inizio incluso, fine ESCLUSA
```

## 18.5 StringBuffer

### ● Due classi per le stringhe

- **String**: stringhe immutabili
  - Non ci sono metodi per modificare una stringa
- **StringBuffer**: stringhe modificabili

```
setCharAt(int index, char ch)  
append(String str)  
insert(int index, String s)
```

String una volta costruita è immutabile: non è possibile cambiare il contenuto né ridimensionarla. String comunque è ragionevolmente efficiente.

## 18.6 StringTokenizer

- `java.util.StringTokenizer` suddivide in token una stringa.
- Utile per il parsing di testo delimitato

```
StringTokenizer st
    = new StringTokenizer("questo e' un test");
while (st.hasMoreTokens()) {
    println(st.nextToken());
}
```

- `StringTokenizer(String str, [String delim])`

- si specificano i caratteri delimitatori
  - default "\r\n\t "

# 19 Contenitori

---

- `java.util.*` contiene classi contenitore
  - `Vector`: array espandibile
  - `Hashtable`: tabella chiave/valore
  - `Enumeration`: enumerazione di elementi



Il tuo sito è sempre uguale ? **EpNuke** la soluzione modulare professionale per il Web Publishing.  
Soluzioni **tutto compreso** con migrazione della grafica, hosting e assistenza.

Per informazioni [www.eproneteus.com](http://www.eproneteus.com) - 212/225

## 19.1 Enumeration

### ● Enumeration: interfaccia standard per enumerare

- ritorna i contenuti dei contenitori

```
boolean hasMoreElements()  
Object nextElement()
```

- Esempio:

```
Enumeration e = table.elements();  
while(e.hasMoreElements())  
    doSomething(e.nextElement());
```

## 19.2 Vector

### ● Vector: un array dinamico

#### ● con dimensione e capacità

```
Object elementAt(int i)
// ritorna l'i-simo elemento.
void setElementAt(int i, Object o)
// modifica l'i-simo elemento.
```

- Vector contiene soltanto Object

- può essere ridimensionato

#### ● Inserimenti

```
void addElement(Object o)
// aggiunge in coda
void insertElementAt(Object o, index i)
// aggiunge dopo l'i-simo
void removeElementAt(index i)
// rimuove l'i-simo elemento
```

#### ● Ricerche

```
int indexOf(Object o [,int index])
// trova l'oggetto.
int lastIndexOf(Object o [,int index])
// trova l'oggetto dalla fine
```

## 19.3 Dictionary

### ● `java.util.Dictionary`: una tabella

#### ● associazione chiave/valore

```
void put(Object k, Object v)
// aggiunge associazione chiave/valore
// sostituisce vm se c'era
Object get(Object k)
// ritorna l'oggetto v
// associato correntemente a k .
void remove(Object k)
// rimuove l'associazione con chiave k
```

#### ● altre operazioni

```
boolean isEmpty()
// dice se è vuoto.
void size()
// numero di associazioni.
Enumeration keys()
// enumerazione delle chiavi.
Enumeration elements()
// enumerazione dei valori
```

## 19.4 Hashtable

### ● `java.util.Hashtable` implementa il Dictionary

- utilizzando la tecnica delle tabelle hash
- utilizza `Object.hashCode()` e `Object.equals(Object)`
- Ha una capacità e un load factor
  - se `size()*loadFactor > capacity()` la capacità viene incrementata.

### ● Ricerche

```
boolean contains(Object)
// contiene il valore
boolean containsKey(Object)
// contiene la chiave
```

# 20 I/O

---

## ● Concetti principali per la gestione di I/O

- package `java.io`

- *stream:*

- un oggetto in cui si scrive (`OutputStream`)
  - o da cui si legge (`InputStream`)

- Due gruppi di classi, una per l'Input ed l'altro per Output

- Casi particolari come `RandomAccessFile`
  - Gestione nomi di file
  - Eccezioni di I/O derivate da `IOException`

## 20.1 Complicazioni

- L'unità di I/O in Java è il *byte* (a 8 bit) non il *char* (a 16 bit)
  - Le Stringhe sono sequenze di caratteri Java a 16 bit
  - l'UTF consente di esprimere una stringa unicode con caratteri ASCII
    - codifica i caratteri non ASCII con sequenze tipo \uABCDE
  - L'I/O di caratteri avviene altrimenti per troncamento/estensione
    - scrittura di caratteri a 16bit: troncati in byte 8bit (si scarta il byte alto)
    - lettura di byte a 8bit: estesi a char 16bit (byte alto nullo)

## 20.2 InputStream

### ● InputStream

● lettura bloccante: si aspetta finchè non è disponibile un carattere

```
int read()
    throws IOException
// ritorna un byte, da 0 a 255
// oppure -1 che significa fine-input
int read(byte[] buf, [int off, int len])
    throws IOException
// legge fino buf.length caratteri
// ritorna il numero di byte letti
long skip (long count)
// salta count caratteri (o fino a EOF)
// ritorna il numero di byte saltati
int available()
// quanti byte posso leggere senza bloccarmi
void close()
// chiude lo stream
```

## 20.3 OutputStream

### ● OutputStream

#### ● Scrittura bloccante

```
void write(int b) throws IOException
// vengono scritti gli 8bit più bassi
// è int per evitare un cast
void write (byte[] buf
[,int offset, int count])
throws IOException
// scrive buf.length caratteri
void flush()
// svuota il buffer
void close()
// chiude lo stream
```



## 20.4 File

### ● File: classe per operazioni legate ai file

- Un oggetto File rappresenta *il nome di un file*
- Gestisce operazioni relative alle directory

```
File (String path)
File (File dir, String name)
// Costruisce un file a partire da un path
// o da un File che rappresenta una directory
// e un nome di file.
```

### ● Funzioni di file

long lenght() // ritorna la lunghezza  
long lastModified() // la data di ultima modifica // in  
termini di millisecondi // dal 1/1/1970  
boolean exist() boolean canRead() boolean canWrite() //  
se un file esiste, è leggibile o scrivibile

### ● Directory

```
boolean isFile()
boolean isDirectory()
boolean isAbsolute()
// È un file , una directory, il path è assoluto?
boolean mkdir()
boolean mkdirs()
// Crea una directory
// o tutte le directory intermedie
boolean renameTo(File newName)
boolean delete()
// Rinomina o cancella un file
String getName()
String getPath()
String getParent()
String getAbsolutePath()
// Estrae il nome, il path, la directory padre
// o il path assoluto
String[] list()
String[] list(FilenameFilter filter)
// Lista i file di una directory
```



Il tuo sito è sempre uguale ? **EpNuke** la soluzione modulare professionale per il Web Publishing.  
Soluzioni **tutto compreso** con migrazione della grafica, hosting e assistenza.

Per informazioni [www.eprometeus.com](http://www.eprometeus.com) - 222/225

## 20.5 File Stream

### ● FileInputStream

- `FileInputStream(String)`
- `FileInputStream(File)`
- `FileOutputStream` analogo

### ● Stream filtro

nel senso che operano su uno stream esistente e vi aggiungono i metodi per potenziare le capacità degli stream base

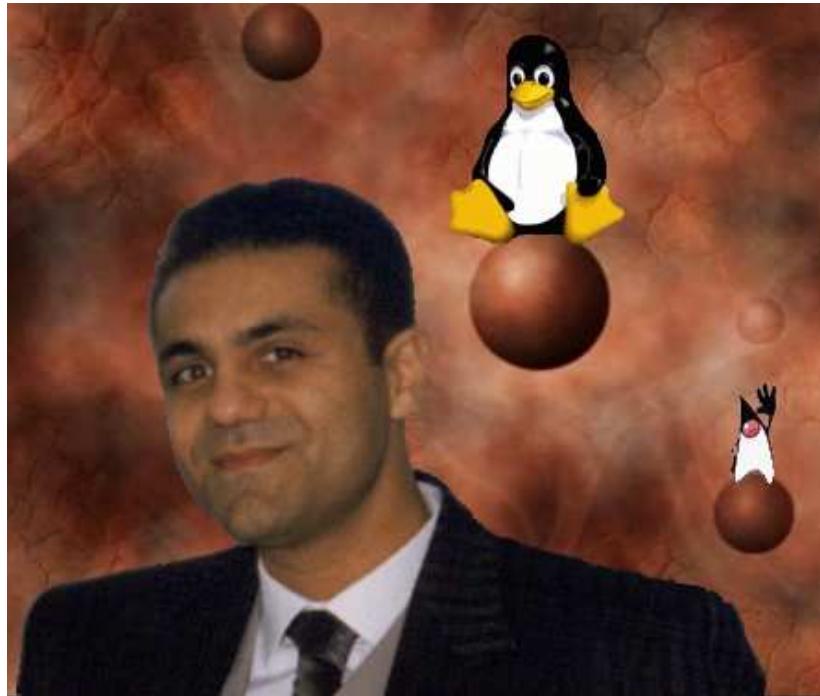
- `DataInputStream` e `DataOutputStream` consentono di leggere e scrivere i vari tipi di Java
  - `char: readChar()`, etc
  - `boolean, char, byte, short, int, long, float, double`

## 20.6 Varie

- **BufferedInputStream , BufferedOutputStream**
  - `readLine()`
- **PrintStream**
  - `print` e `println` per: String, char, int, long...
- **SequenceInputStream**

```
FileInputStream files = {  
    new FileInputStream ("a"),  
    new FileInputStream ("b") };  
InputStream in = SequenceInputStream (files);
```

- **ByteArrayInputStream e  
ByteArrayOutputStream**
  - sono Stream "costanti", in memoria
  - `ByteArrayInputStream(byte[] data)`
  - `ByteArrayOutputStream()`
    - `byte[] toByteArray()`
- **RandomAccessFile**
  - lettura e scrittura in qualunque posizione di un file
- **PushbackInputStream**
  - stream filtro con `unread` per mandare indietro un byte



L'autore: **Michele Sciabarrà**, uno dei massimi esperti italiani di tecnologie Open Source. Già autore del best seller "Linux e Programmazione Web", dedicato alle tecnologie Open Source del Web, è stato fondatore di una delle prime aziende italiane che lavora esclusivamente con software aperto. Collabora come esperto Open Source con numerose riviste del settore; come docente ha tenuto decine di corsi su Linux, e come consulente ha guidato l'introduzione di software Open Source in grandi realtà del settore aeronautico, farmaceutico bancario/assicurativo. Può essere contattato come [michele@sciabarra.com](mailto:michele@sciabarra.com).