Programmazione Avanzata

Marco Sgobino

7 novembre 2022

Indice

Capitolo 1

Caratteristiche generali

Java è stato creato a partire da ricerche effettuate alla Stanford University agli inizi degli anni novanta. Nel 1992 nasce il linguaggio Oak (in italiano "quercia"), prodotto da Sun Microsystems™ e realizzato da un gruppo di esperti sviluppatori capitanati da James Gosling. Il vero e proprio linguaggio Java nasce nel 1995, sotto la guida dello stesso sviluppatore.

Le ragioni per adoperare Java sono molteplici,

- il linguaggio stesso adotta il paradigma dello “scrivi una volta sola, gira ovunque”: Java è un linguaggio *indipendente dall’hardware*. L’indipendenza dall’hardware è raggiunta mediante una piattaforma intermedia che è sita fra hardware e software - la **Java Virtual Machine**;
- il supporto della community è solido e competente;
- strumenti come gli IDE sono molto versatili e funzionali;
- vi sono tante librerie API disponibili;
- è molto anziano e solido; molti pattern di programmazione sono stati creati in Java;
- è free (and open source).

Il linguaggio di per sé favorisce l’*astrazione*: l’**oggetto** diviene dunque l’astrazione principale, diversamente da quanto avviene per il paradigma orientato alle funzioni[1]. Java è dunque in grado di realizzare molti concetti chiave relativi alla programmazione in generale.

L’astrazione è relativa all’attività mentale del *computational thinking* – richiede una conoscenza del *dominio* della realtà da modellare, oltre che alla conoscenza degli *strumenti* da adoperare per la modellazione e la realizzazione. Java si sposa particolarmente bene con l’astrazione, poiché è un linguaggio **tipizzato** (tipizzazione forte), e fortemente **orientato agli oggetti**. Ogni classe in Java modella un concetto, una possibile entità reale, una manifestazione di un oggetto concreto.

Java è sia una piattaforma software che un linguaggio di programmazione, e la distinzione fra le due sarà delineata in seguito.

[1] Il paradigma orientato alle funzioni è un paradigma che precede storicamente il paradigma ad oggetti. Un linguaggio orientato alle funzioni è un linguaggio che ha come elemento principale le *funzioni*, porzioni di codice con una propria interfaccia e dedicate allo svolgimento di operazioni con particolari strutture dati, direttamente fornite tramite input della funzione stessa. Il paradigma ad oggetti è un’estensione di quello a funzioni, comportando anche numerosi benefici dal punto di vista della modellazione.

1.1 La piattaforma Java

La piattaforma Java è composta da:

- un *motore di esecuzione*;
- un *compilatore*;
- un *set di specifiche*;
- un *set di librerie*, con le loro API (Application Program Interfaces).

Le piattaforme, chiamate anche *edizioni*, differiscono fra loro principalmente nelle librerie disponibili. Le opzioni più comuni sono:

- *Java 2 Platform, Standard Edition* (J2SE);
- *Java 2 Platform, Enterprise Edition* (J2EE), API e specifiche per architetture client-server e applicazioni enterprise;

Importanti strumenti sono:

- Il **Java Development Kit** (JDK, ad esempio *OpenJDK*). I JDK Sono caratterizzati da venditore, piattaforma, versione, OS e architettura;
- API documentation (chiamata **javadoc**);
- Integrated Development Environment (IDE);

Il JDK deve tenere conto dell'architettura dell'host: in particolare, il motore di esecuzione deve tenere conto del sistema operativo sottostante.

1.2 Compilazione ed esecuzione

In Java, ci sono due tipi di files:

- il *source code* in uno o più file `.java`;
- l'*eseguibile* in uno o più file `.class`.

La compilazione permette di ottenere un file `.class` da un file `.java`,

```
javac Greeter.java
```

Il codice sorgente ha una sintassi definita per favorire la comprensione da parte dell'essere umano, il file compilato, invece, è eseguibile e non è fatto per essere apprezzato dall'essere umano.

Ciascun file `.java` contiene **esattamente una** classe,

```
public class Greeter {
   \\ ...
}
```

In realtà, per un file java possono essere messe più di una classe, ma verrà visto in seguito. Nel caso di più classi, esse possono essere o *inner classes*, cioè classi definite all'interno di altre classi, oppure solo la prima classe dichiarata nel file deve avere l'identificatore `public`.

```
public class Greeter {
   \\ ...
}

class Helper {
   \\ ...
}
```

```
public class Greeter {
   \\ ...
   class InnerHelper {
      \\ ...
```

```
   }
}
```

Tipicamente definire una singola classe per ogni file è meglio, con una maggiore leggibilità, modularità e semplicità del codice.

L'esecuzione del codice avviene all'interno di un metodo `main`: possono essere eseguiti soltanto i file class che hanno al loro interno definito il metodo main, mentre tutti i file che contengono classi non aventi un metodo main **non** possono essere eseguite. Un metodo main è così definito:

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

e la sintassi per l'esecuzione è la seguente,

```
java Greeter
```

In altre parole, il cosiddetto *entrypoint* per il programma è costutuito dal metodo `main`, il quale, dunque, viene invocato all'esecuzione di un determinato programma. Durante l'esecuzione, viene eseguito il codice (*bytecode*) contenuto in un file `.class`. Il bytecode viene così **interpretato** dal programma `java`, il quale simula una macchina, la **Java Virtual Machine** (JVM). Tale macchina simula una macchina fisica, offre un'interfaccia dell'elaboratore al codice soprastante, a sua volta interloquendo col sistema operativo sottostante. Come una macchina reale, ha un suo insieme di istruzioni e una sua memoria da gestire. la JVM è inoltre agnostica rispetto alla macchina fisica, e conosce solamente il bytecode, **non** il linguaggio Java. Dunque, la JVM è un concetto diviso e separato dal Java, con il compilatore che diventa il punto di contatto fra i due mondi. La JVM è *portabile*: è sufficiente esista (e sia installato) un JDK per una determinata piattaforma hardware per poter eseguire del codice su di essa (write once, run anywhere). Abbiamo dunque la seguente cascata concettuale, dall'"alto livello" verso il "basso livello":

```
| .class file |
        |
        V
| Eseguibile Java | -- Executes bytecode
        |
        V
| Sistema Operativo | -- Executes java executable
        |
        V
| Macchina fisica | -- Executes OS
```

Il programma eseguibile `java` fa parte, specificatamente, del **Java Runtime Environment** (JRE), che contiene anche le librerie necessarie. JRE è parte integrante dello JDK. Lo JDK contiene, in aggiunta, gli eseguibili `javac` e `java`, alcune classi precompilate `.class` di libreria, e altri strumenti utili allo sviluppo. Non contiene, tuttavia, i codici sorgenti delle librerie precompilate, e non contiene documentazione – ambedue devono pertanto essere scaricati separatamente, e il download può avvenire sia tramite IDE o simili strumenti che manualmente.

La documentazione e il codice sorgente servono entrambi (specialmente la documentazione). La documentazione è reperibile online, oppure attraverso IDE o javadoc. Il codice sorgente è reperibile comunque online, oppure attraverso l'IDE.

In Java, una volta terminato lo sviluppo di un programma, ci sono due fasi:

1. la **compilazione**, da `.java` a `.class`;

2. l'**interpretazione**, con esecuzione *virtuale* di una classe in un file `.class`. L'esecuzione non è reale, ma virtuale, nel senso che la JVM deve trasformare il bytecode 'interpretato' in assembly code per il sistema operativo sottostante. La trasformazione sarà differente a seconda del sistema operativo su cui la JVM è in esecuzione, poiché la Java Virtual Machine deve procedere ad.

Java non è né compilato né interpretato in senso assoluto, ma è **entrambe le cose**: Java produce dei file class attraverso la *compilazione*, ed il bytecode viene poi *interpretato* dall'eseguibile (la JVM). Tipicamente, anche se è interpretato, il codice è *veloce abbastanza* (l'unica metrica di cui tener conto). Quel che accade di solito è che il problema è raramente nel fatto che il codice viene interpretato anziché compilato – benché l'interpretazione effettuata dalla JVM comporti una perdita di prestazioni – semmai il problema risiede nella scelta dell'algoritmo o in altre (vacanti) tecniche di ottimizzazione. Perciò possiamo preoccuparci relativamente della perdita di prestazioni dovuta all'interpretazione del codice anziché all'esecuzione di un vero "programma precompilato"; semmai è di gran lunga più importante far sì che l'algoritmo adoperato e le tecniche di ottimizzazione messe in campo siano efficienti.

Una tecnica tipica di ottimizzazione di un programma si svolge in 3 passi:

1. *profilazione* del software in esecuzione, monitorando la JVM (opzionale, sebbene molto utile);
2. individuazione delle *possibili migliorie al programma*, soprattutto modificando l'algoritmo e come esso viene eseguito o come impiega le strutture dati messe in campo;
3. applicazione delle *migliorie* e verifica del miglioramento nelle performance.

Il baratto sulla velocità in Java permette, però, di ridurre i costi in molte altre maniere, inizialmente meno evidenti:

- maggiore *manutenibilità* del codice – il codice è molto più chiaro, leggibile, e viene scritto ovunque alla stessa maniera, senza preoccuparsi della portabilità, del *locale*, o di altri aspetti dipendenti dalla macchina su cui il programma andrà in esecuzione. Come metro di paragone si pensi al C, e a tutte le attenzioni necessarie alla realizzazione corretta dei vari *locale*, al supporto di varie architetture hardware, tutto ciò dovendo preoccuparsi anche di una robusta fase di *testing* del prodotto;
- maggiore *supporto della community*, buona documentazione – Java è molto usato, aiuto e documentazione sono reperibili online;
- alta *qualità degli strumenti di sviluppo*, come *IntelliJ*;
- disponibilità abbondante di *librerie*;
- è molto rodato e dunque c'è molta conoscenza pregressa; spesso è facile trovare esempi su internet di programmi già esistenti svolti proprio in Java.

Java, tuttavia, **non consente ancora lo sfruttamento della capacità di calcolo della GPU**.

Capitolo 2

Concetti base del linguaggio Java

2.1 Creazione degli oggetti e dei riferimenti

Analizziamo il seguente codice in estremo dettaglio, il classico primo programma, che stampa sullo schermo la stringa `Hello World!`:

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

La prima distinzione che viene fatta è fra gli **oggetti** e gli **identificatori**. Le "cose" o entità che sono manipolate nel codice sono gli *oggetti*. Gli oggetti *esistono, ma non hanno un nome*. Il codice manipola gli oggetti mediante i *riferimenti*. In particolare, i riferimenti hanno un nome, chiamato *identificatore*. Dunque, gli oggetti – in inglese *objects* – sono un concetto diverso dai riferimenti – in inglese *references*. Nella pratica, *i riferimenti sono manipolati tramite il loro identificatore*. Entrambi, oggetti ed identificatori, possono essere creati.

Per cominciare, lo snippet

```
String s;
```

crea un riferimento ad un oggetto di tipo `String` con identificatore di nome simbolico `s` (cioè la reference `s`) – si osserva che un riferimento è *sempre dotato di un nome*. In questo caso, *non* si assiste alla creazione di alcun oggetto. Inoltre, i riferimenti hanno un unico nome. Assieme al nome simbolico del riferimento, indicheremo il *tipo* di oggetto al quale il riferimento viene creato. Il rispettivo diagramma oggetti-riferimenti sarà

s
*

Il frammento

```
String s = new String("Content");
```

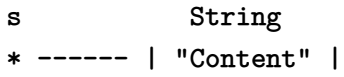
fa invece tre cose:

1. **crea un riferimento** `s` di tipo `String`;
2. **crea un oggetto** di tipo `String` e lo inizializza con `"Content"`;

3. **fa sì che `s` faccia riferimento al nuovo oggetto** – viene dunque creata l'associazione alle due entità appena create.

L'associazione rifierimento-oggetto è essenziale per rendere manipolabile un nome e al contempo ci consente di dare un nome preciso all'oggetto che stiamo manipolando, garantendo la possibilità di chiamarlo, di nominarlo. Senza l'associazione riferimento-oggetto diviene impossibile manipolare l'oggetto che, benché possa essere stato creato ed avere una propria posizione in memoria, non ha un nome manipolabile.

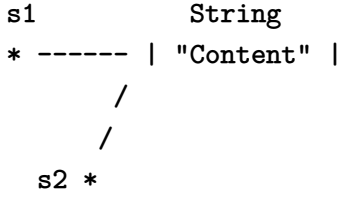
La creazione di un oggetto prevede l'uso di una parola chiave. La parola chiave **`new`** fa parte del linguaggio Java, ed è una parola chiave che permette di creare un nuovo oggetto. In questo caso, avremo dunque un diagramma oggetti-riferimenti del tipo



Più riferimenti per il medesimo oggetto possono essere creati. Ad esempio, nel seguente frammento,

```
String s1 = new String("Content");
String s2 = s1;
```

viene creato un riferimento **`s1`** con associazione all'oggetto appena creato, ed **`s2`** viene associato all'oggetto a cui si riferiva **`s1`**.

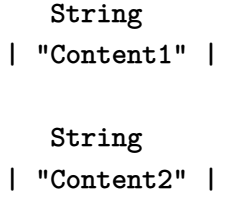


In altre parole, quando si ha un'assegnazione fra due riferimenti, il riferimento a sinistra farà riferimento all'oggetto a cui si riferiva il riferimento a destra.

Possono essere creati degli oggetti senza riferimenti. Infatti, eseguendo

```
new String("Content1");
new String("Content2");
```

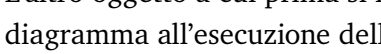
vengono creati due oggetti di tipo **`String`** con il contenuto espresso nelle stringhe. Tuttavia, poiché i due oggetti sono privi di riferimento, essi *non possono essere operati in alcuna maniera dal momento che non hanno nome.*



I riferimenti possono cambiare il loro riferimento. Infatti,

```
String s1 = new String("Content1");
String s2 = new String("Content2");
String s1 = s2;
```

all'esecuzione di questo codice si ottiene che entrambi i riferimenti si riferiscono al medesimo oggetto, di tipo **`String`** e di contenuto **`"Content2"`**. L'altro oggetto a cui prima si riferiva **`s1`** non potrà più essere manipolato. Il diagramma all'esecuzione della riga 2 sarà



```
s2           String
* ------ | "Content2" |
```

mentre all'esecuzione completa del frammento avremo una situazione diversa,

```
s1           String
*        | "Content1" |
   \
     \
s2     \     String
* ------ | "Content2" |
```

2.2 Manipolazione degli oggetti

Brevemente,

- le operazioni che possono essere applicate agli oggetti dipendono dal *tipo* dell'oggetto, poiché ogni oggetto è un'**istanza** di un tipo; un tipo definisce le operazioni che sono *applicabili alle istanze*. Ad esempio, `String` è un tipo di oggetto e `String s = new String("Example");` è un riferimento che riferisce ad un'istanza di un oggetto di tipo `String`. Ogni tipo ha dentro di sé insite le operazioni che possono essere effettuate su di lui;
- per specificare un'operazione nel codice, si adopera la sintassi detta **dot notation**, nella quale il punto viene così adoperato, `refName.opName()`, separando tramite un punto il nome del riferimento all'oggetto con l'operazione che viene invocata. Le parentesi tonde devono essere aggiunte per invocare l'operazione;
- nel tipo sono anche specificati i **parametri di input** e **di output**. In particolare, i parametri di input sono collocati all'interno delle parentesi tonde, mentre l'output può essere a sua volta referenziato, tramite assegnazione. Ad esempio,

```
retRefName = refName.opName(otherRefName1, otherRefName2);
```

`retRefName` riferirà l'oggetto che è il risultato dell'operazione a destra dell'assegnazione.

2.3 Le classi e i tipi primitivi

2.3.1 Classi

In Java una **classe** è un tipo. Ogni oggetto è un'*istanza di una classe*. Tipicamente, in un programma Java si adoperano molte classi, alcune facenti parte delle API. *Ciascuna applicazione include almeno una classe, definita dallo sviluppatore*.

Le classi non sono oggetti. Un riferimento riferisce un oggetto, il quale è un'istanza di una classe.

2.3.2 Tipi primitivi

I tipi primitivi non sono classi. Sono 8, e sono

- `boolean`;
- `char`;
- `byte`;
- `short`;
- `int`;
- `long`;

- `float`;
- `double`.

2.3.3 Utilizzo delle classi

Le classi si utilizzano attraverso i **metodi**. Ogni operazione applicabile ad un oggetto è un metodo della classe. Ciascun metodo ha una propria **signature**, che consiste in:

- nome;
- sequenza di tipi di parametri di input (il nome non è rilevante);
- tipo di parametro di output (detto **return type**). Nel caso di nessun valore di ritorno, la sintassi `void` permette di non avere nessun valore di output.

Alcuni esempi di signature possono essere

```
char charAt(int index)
int indexOf(String str, int fromIndex)
String replace(CharSequence target, CharSequence replacement)
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

dove nell'ultimo esempio la notazione `char[]` indica un *array* di char. Una signature è di estrema importanza: vengono definite delle operazioni, e delle entità di input ed output.

I metodi si invocano nella seguente maniera,

```
String s = new String("Hello!");
int l = s.length();
```

In questo caso, l'invocazione del metodo su `s` fa si che `l` diventi un riferimento ad un tipo primitivo `int` di valore 6 – il risultato dell'operazione su `s` – mentre `s` è un riferimento che fa riferimento all'oggetto appartenente alla classe `String`.

Alla compilazione, il compilatore `javac` verifica anche che le seguenti condizioni siano rispettate:

- il tipo di `s` abbia un metodo di nome `length`;
- il metodo è usato consistentemente con la signature, ovvero che i tipi di input e di output siano rispettati;

Il controllo è fatto in *compile time*, ed è molto più arduo di quanto sembri. Tuttavia, il controllo è estremamente utile per evitare errori e per disegnare correttamente buon software. Java è colloquialmente detto essere **strongly typed** (fortemente tipizzato).

2.3.4 Overloading dei metodi

Più metodi possono avere lo stesso nome. Una classe può avere più metodi, anche aventi lo stesso nome, ma con parametri differenti di input. Due metodi con stessi parametri di input e stesso nome **non** possono esistere (nemmeno se hanno un diverso parametro di ritorno). In particolare,

```
String s = new String("Hello!");
PrintStream ps = new PrintStream(/*...*/);
ps.println(s);
```

I metodi in seguito hanno tutti lo stesso nome, ma differiscono per i loro parametri di input:

```
void println()  // Terminates the current line by writing the line
    separator string.
void println(boolean x) // Prints a boolean and then terminate the
    line.
void println(char x) // Prints a character and then terminate the
    line.
void println(double x) // Prints a double and then terminate the
    line.
void println(float x) // Prints a float and then terminate the
    line.
void println(int x) // Prints an integer and then terminate the
    line.
void println(long x) // Prints a long and then terminate the line.
void println(String x) // Prints a String and then terminate the
    line.
```

In questo caso non è importante l'ordine e non ci devono essere duplicati – dunque, trattasi di un insieme di metodi di una classe. Questa capacità di avere un insieme di metodi è detta **method overloading**. Per discernere i vari metodi è opportuno adoperare la documentazione (javadoc), ed utilizzare in maniera efficiente l'IDE mediante autocompletamento. Solitamente, nel nome del metodo è insito che cosa faccia, tenendo anche conto delle **naming conventions**.

In definitiva, due metodi possono avere il medesimo nome, ma devono differire nella signature per quanto riguarda i parametri di input.

2.4 Le convenzioni sui nomi

Le convenzioni sui nomi sono di cruciale importanza: il codice viene letto da umani, ed è dunque necessario che il codice esprima la modellazione della realtà. Seguire le convenzioni sui nomi è un passo fondamentale per garantire una buona **qualità del codice**. La scelta corretta dei nomi, dunque, è di importanza fondamentale.

2.4.1 Nomi di classi

I nomi delle classi sono *nomi di attori*, ad esempio `Dog`, `Sorter`, `Tree`. Dovrebbero essere *rappresentativi dell'entità che la classe rappresenta*. La naming convention per i nomi di classe è quella della **upper camel case** (dromedary case), ad esempio `HttpServerResponse`. In questo ultimo caso, il nome della classe è composto da più parole (attributi) rispetto all'attore principale (`Response`). **Ogni nome di classe inizia con la maiuscola**, assieme alle iniziali degli attributi.

2.4.2 Nomi di metodi

I nomi dei metodi sono *verbi di un'azione*, ad esempio `bark`, `sort`, `countNodes` con alcune eccezioni accettabili (`size`, `length`, `nodes`). I nomi dei metodi dovrebbero essere rappresentativi delle operazioni che i metodi svolgono. La naming convention per i nomi di metodi è quella della **lower camel case**, ad esempio `removeLastChar`. **Ogni nome di metodo inizia con la minuscola**, diversamente dalle iniziali delle parole successive.

2.4.3 Nomi di riferimenti

I nomi dei riferimenti sono coerenti con il loro tipo. Devono essere anche però rappresentativi dell'uso specifico di quell'oggetto, nel contesto in cui compare. Nel caso dei riferimenti, si adopera la **lower camel case**. È meglio essere specifici, ad esempio `numOfElements` è meglio di `n`.

2.5 Dot operator associativo

Spesso capita di voler evocare un'operazione ad un oggetto che è stato ottenuto come valore di ritorno di un'invocazione di un altro metodo, senza necessariamente voler riferire questo oggetto che è stato ritornato. In questo caso, si usa la *dot notation associativa:*

```
String s = new String(" shout!");
s = s.trim().toUpperCase();
```

E dunque `toUpperCase()` è un metodo evocato sull'oggetto ritornato da `s.trim()`. In parole povere, la dot notation viene valutata "da sinistra", eseguendo prima il metodo `trim()` sull'oggetto di riferimento `s`, per poi eseguire il metodo `toUpperCase()` sul risultato dell'invocazione precedente, senza fare riferimento all'oggetto di tipo `String` generato dall'operazione intermedia. Attenzione, poiché verranno creati tre oggetti, indicati in Figura 2.1, con ciascun oggetto creato all'esecuzione dei metodi. La classe `String` è *immutabile* – ciò significa che ad ogni operazione viene creato un nuovo oggetto, e non viene modificata la stringa di per sé.

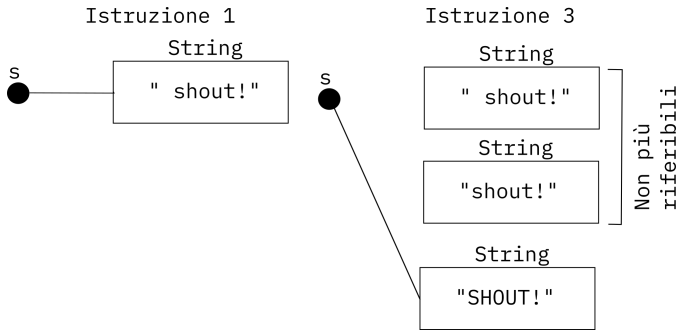


Figura 2.1: Creazione di oggetti multipli nel semplice contesto di una creazione di un oggetto e di un'assegnazione dove sono invocati dei metodi dell'oggetto. Ciò avvviene poiché la classe `String` è *immutabile*.

2.6 Il costrutture

Ogni classe ha *almeno* un **costruttore**. Un costruttore di una classe `T` è un metodo speciale che, quando invocato, ha come risultato:

- la *creazione* di un nuovo oggetto di classe `T`;
- l'*inizializzazione* del nuovo oggetto.

Un costruttore viene invocato con lo scopo di istanziare un oggetto di una classe. I costruttori hanno vincoli *sul nome* e sul *parametro di uscita* – il nome del costruttore è *uguale al nome della classe*, mentre il parametro di uscita deve essere *dello stesso tipo della classe*. La sintassi speciale per l'invocazione del costruttore è tramite la keyword `new`, che viene adoperata come negli esempi precedenti. Un esempio di sintassi potrebbe essere il seguente,

```
Greeter greeter = new Greeter();
```

L'uso della keyword è obbligatorio, con poche eccezioni, come ad esempio

```
String s = "hi!";
```

produrrà lo stesso effetto di

```
String s = new String("hi!");
```

All'inizializzazione, il costruttore eseguirà delle operazioni. In particolare, per ogni tipo e per ogni classe ci possono essere più di un costruttore, spesso con differenze proprio su come viene inizializzato un oggetto.

Supponiamo di avere una classe `Date`, che rappresenta uno specifico istante di tempo con precisione al millisecondo. Il costruttore della classe `Date` è così fatto,

`Date()` Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.

Il codice

```
Date now = new Date();
// some code doing long things
Date laterThanNow = new Date();
```

con molta probabilità creerà due oggetti con stato interno diverso (la data sarà differente).

I costruttori possono essere da 1 singolo a molti per una singola classe. Valgono le medesime regole dei metodi, con la lista dei parametri di ingresso che deve essere differente per ciascuno di loro (il nome è vincolato, dunque essi devono necessariamente differire per la lista dei parametri d'argomento). Esempi si hanno tramite la documentazione della classe **`String`**,

`String()` Initializes a newly created String object so that it represents an empty character sequence.

`String(char[] value)` Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

`String(String original)` Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

e tramite la classe **`Socket`**,

`Socket()` Creates an unconnected Socket.

`Socket(String host, int port)` Creates a stream socket and connects it to the specified port number on the named host.

`Socket(String host, int port, boolean stream)` Deprecated. Use DatagramSocket instead for UDP transport.

`Socket(String host, int port, InetAddress localAddr, int localPort)` Creates a socket and connects it to the specified remote host on the specified remote port.

`Socket(InetAddress address, int port)` Creates a stream socket and connects it to the specified port number at the specified IP address.

`Socket(InetAddress host, int port, boolean stream)` Deprecated. Use DatagramSocket instead for UDP transport.

`Socket(InetAddress address, int port, InetAddress localAddr, int localPort)` Creates a socket and connects it to the specified remote address on the specified remote port.

`Socket(Proxy proxy)` Creates an unconnected socket, specifying the type of proxy, if any, that should be used regardless of any other settings.

`protected Socket(SocketImpl impl)` Creates an unconnected Socket with a user-specified SocketImpl.

2.7 Il concetto di information hiding

L'*information hiding* è il principio mediante il quale le *decisioni sul design* e sulla modellazione che subiscono frequenti variazioni vengono nascoste all'utilizzatore, di modo da garantirne la protezione e la sicurezza d'utilizzo nel caso in cui drastiche variazioni di design abbiano luogo. Ciò che viene esplicitamente fornito è l'*interfaccia* che tale programma mostra nei confronti di altri programmi.

Chi crea la classe è solitamente diverso dall'utilizzatore della classe stessa. L'utente può adoperare la classe, ma potrebbe non conoscere il codice sorgente. Per la classe **`Date`**, ad esempio,

`boolean after(Date when)` Tests if this date is after the specified date.

`boolean before(Date when)` Tests if this date is before the specified date.

`long getTime()` Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.

`String toString()` Converts this Date object to a String of the form:

L'idea è quella di poter effettuare operazioni apparentemente banali senza doverci preoccupare di *come* queste vengano svolte sull'oggetto: in questo caso si parla di **information hiding**, poiché si conosce il risultato e l'effetto di un'operazione senza conoscerne il codice sorgente o come questa operazione venga di fatto svolta dal metodo della classe. L'utilizzatore finale, dunque, è *protetto* da qualsivoglia cambiamento interno del modulo, protetto dal principio dell'information hiding – egli è tenuto ad utilizzare soltanto ciò che viene mostrato esternamente. Lo *stato* di un oggetto e il *codice* della classe potrebbe cambiare, ma per l'utente *non è necessario conoscere tali cambiamenti interni*.

L'altro aspetto dell'information hiding è la **modularità**. Un utente di una classe conosce *quali operazioni esistono, come si usano, che cosa fanno,* tuttavia **non** sanno che cosa ci sia *all'interno* dell'oggetto, o *come* esattamente funziona un'operazione.

Il concetto di modularità, dunque, si rifà alla nozione di information hiding – ciascuno si prende cura di una singola parte del codice, senza che all'utente tutto ciò non comporti alcun effetto su come vengono adoperati i moduli stessi, poiché le interfacce esterne vengono mantenute pressoché le stesse. Una conseguenza assolutamente notevole di ciò è che nella software community è sufficiente che ciascuno sviluppatore si preoccupi soltanto dei moduli di cui egli è responsabile.

2.8 Riassunto del capitolo

Oggetti Gli *oggetti* sono entità che esistono, ma non hanno un nome. Essi sono creati in Java, manipolati attraverso il codice.

Riferimenti Il codice manipola gli oggetti attraverso i *riferimenti*. I riferimenti hanno un nome, detto *identificatore*, grazie al quale è possibile la loro manipolazione. I riferimenti possono fare riferimento ad oggetti esistenti e creati in precedenza.

Creazione degli oggetti e dei riferimenti Sia i riferimenti che gli oggetti possono essere creati nel codice. La creazione di un oggetto prevede l'uso di una parola chiave, `new`, la quale fa parte del linguaggio Java e permette di creare un nuovo oggetto.

Associazione dei riferimenti agli oggetti Nel codice è possibile far sì che i riferimenti facciano riferimento (siano associati) ad un oggetto esistente. In Java lo si può fare in vari modi, in particolare mediante l'operazione di *assegnazione* (`=`) viene fatto si che il riferimento faccia riferimento all'oggetto indicato. Un riferimento può riferire ad altri oggetti durante il corso dell'elaborazione nel codice. È sufficiente riassegnarlo ad un altro oggetto, o ad un altro riferimento: in quel caso, esso farà riferimento all'oggetto il cui riferimento che viene indicato nell'assegnazione fa riferimento. Più oggetti possono avere più riferimenti che ne fanno riferimento.

Oggetti senza riferimento Oggetti senza riferimento non hanno nome, e non possono pertanto più essere manipolati attraverso il codice.

Le classi Le classi sono i tipi in Java. Ogni classe è un tipo. Ogni oggetto è un'*istanza* di una classe. Il riferimento fa riferimento agli oggetti, i quali sono istanze delle classi.

I tipi primitivi I tipi primitivi sono 8: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`.

Utilizzo delle classi In Java, le classi si adoperano tramite i *metodi* e i loro *field*. Ogni classe può svolgere operazioni tramite metodi, e collezionare strutture dati tramite i suoi campi.

Operazioni sugli oggetti Le operazioni sugli oggetti dipendono dal *tipo* dell'oggetto. In particolare, un tipo definisce le operazioni che sono *applicabili alle istanze*.

La *dot notation* La *dot notation* consente di specificare un'operazione su un oggetto, oppure di invocare un field di tale oggetto. Tipicamente la dot notation viene applicata alle *istanze* di un oggetto, tuttavia non mancano i casi in cui essa è da applicarsi alle *classi* in sé (o più precisamente, ai nomi di riferimento di tale classe), come nel caso dei metodi dichiarati `static`.

La signature I parametri del tipo sono indicati nella *signature*. La signature contiene *nome*, sequenza dei *parametri di input*, e *parametro di output*. Il nome della sequenza dei parametri di input non è rilevante. La signature indica sia il tipo di output previsto dall'operazione, che il tipo di input che essa accetta. La signature di un metodo deve essere indicativa di cosa esso rappresenta, delle operazioni che esso svolge, e dovrebbe essere auto-esplicativa. Possono esistere più signature aventi lo stesso valore di return e lo stesso nome, tuttavia esse devono presentare differenti parametri di input, la cui sequenza è detta *argomento*. La sintassi `void` consente di esprimere nessun valore di ritorno. Il compilatore valuta le signature, e il codice viene poi valutato di conseguenza.

Strong typing Java è un linguaggio *fortemente tipizzato*: ciò significa che, a tempo di compilazione, è effettuata la verifica dell'esistenza di un metodo invocato, e successivamente che i tipi di input e di output siano rispettati.

Overloading di un metodo I metodi possono subire *overloading*: più metodi possono avere lo stesso nome, ma devono differire in sequenza di tipi di input (differire dunque nell'argomento). Essi sono dunque un insieme di metodi aventi lo stesso nome.

Nomi di classi I nomi di classi dovrebbero essere *nomi di attori*, rappresentativi dell'entità che la classe rappresenta. La naming convention è la *upper camel case*.

Nomi di metodi I nomi di metodi dovrebbero essere *verbi di un'azione*: essi dovrebbero essere rappresentativi delle operazioni che i metodi svolgono. La naming convention per i metodi è la *lower camel case*.

Nomi di riferimenti I nomi di riferimenti sono coerenti con il loro tipo, e rappresentativi dell'uso specifico di quell'oggetto, istanza di una determinata classe. La naming convention per i nomi di riferimenti è la *lower camel case*.

Dot operator associativo Il dot operator associativo consente di "concatenare" più invocazioni di metodi, con associatività da sinistra. I risultati "intermedi", cioè che non corrispondono all'ultima esecuzione di metodo o riferimento a field, saranno scartati.

Costruttore Si dice costruttore della classe `T` un metodo speciale che, quando invocato, crea un nuovo oggetto di classe `T` e lo inizializza. I nomi di costruttori sono uguali ai nomi della classe che li ospita. Più costruttori possono coesistere, purché essi differiscano nell'argomento. Ogni classe ha un costruttore predefinito, detto *costruttore di default* – il costruttore di default è automaticamente introdotto da Java qualora non fosse stato esplicitamente dichiarato. Il costruttore di default è invece **assente** nel caso in cui vengano dichiarati altri costruttori con argomento diverso. L'invocazione del costruttore avviene mediante la parola chiave `new`, specifica del linguaggio Java, ed obbligatoria nella quasi totalità dei casi, eccezion fatta per oggetti della classe `String`.

Information hiding L'*information hiding* è il principio mediante il quale le *decisioni sul design* e sulla modellazione che subiscono frequenti variazioni vengono nascoste all'utilizzatore, di modo di garantirne la protezione

e la sicurezza d’utilizzo nel caso drastiche variazioni di design abbiano luogo.

Capitolo 3

Codifica delle classi

3.1 Esempio: i numeri complessi

Il seguente frammento, è un esempio di utilizzo della classe `Complex`,

```
Complex c1 = new Complex(7.46, -3.4567);
Complex c2 = new Complex(0.1, 9.81);
Complex c3 = c1.add(c2);
// same for subtract(), multiply(), divide()
double norm = c2.getNorm();
double angle = c2.getAngle();
String s = c2.toString();
double real = c2.getReal();
double imaginary = c2.getImaginary();
```

Si definisce dunque il concetto di *numero complesso* come classe, assieme alle operazioni (metodi) che si intendono svolgere con essi. Un numero complesso può essere costruito a partire dai numeri reali, con un numero per la parte reale ed un altro numero per la parte immaginaria. È dunque necessario conoscere *il dominio* (dei numeri complessi), definire quali sono *le parti che compongono* tale numero complesso, e quali sono *le operazioni che si possono eseguire* con tale entità. La rappresentazione non può mai prescindere dalle entità o dalle operazioni: in Java sono necessarie entrambe, le prime per poter produrre una classe e degli oggetti "reali", le seconde per poterle manipolare. Ecco dunque che il paradigma *ad oggetti*, pietra angolare del linguaggio Java, risulta essere un efficace paradigma *di modellazione*, dove l'entità "oggetto" incarna un proprio stato e delle proprie operazioni, a seconda dell'entità modellata e del tipo di oggetto da modellare.

Il seguente codice modella la classe:

```
public class Complex {
  private double real;
  private double imaginary;
  public Complex(double real, double imaginary) {
    this.real = real;
    this.imaginary = imaginary;
  }
  public double getReal() {
    return real;
  }
  public double getImaginary() { /* ... */ }
  public Complex add(Complex other) {
    return new Complex(
```

```
      real + other.real,
      imaginary + other.imaginary
    );
  }
  /* other methods */
}
```

dove nella prima parte vengono introdotti due **field**, uno per la parte reale ed uno per la parte immaginaria. Viene poi definito un costruttore e due metodi (con possibilmente altri da aggiungere). I field sono *oggetti contenuti in oggetti* (di altri tipi o dello stesso tipo) – i field di un oggetto ne rappresentano lo **stato**. Uno stato è un possibile valore di un'istanza appartenente all'universo dei possibili valori di un tipo. I field sono referenziati, e sono dunque anche dotati di un identificatore. I field hanno un tipo ed un nome del riferimento scelto (identificatore). Essi possono essere manipolati tramite la dot notation, in egual maniera che i metodi. Ad esempio,

```
public Complex add(Complex other) {
  return new Complex(
    real + other.real,
    imaginary + other.imaginary
  );
}
```

dove `other` deve essere un riferimento ad un oggetto per cui possono valere le operazioni che vengono evocate, o esistono i field richiesti. I field sono corredati dai **modificatori d'accesso** – essi dichiarano *quanto visibile* è l'entità appena dichiarata nei confronti di entità esterne, ovverosia se sia lecito accedere a tale entità tramite la dot notation (vale per field, metodi, classi). Esistono due modificatori d'accesso,

- `private`: visibile soltanto all'interno del codice della classe stessa, o da altre istanze della medesima classe;
- `public`: visibile dappertutto.

Essi vanno collocati *prima* della definizione del field o del metodo. Due esempi possono essere

```
// File Complex.java
public class Complex {
  private double real;
  /* here real can be used */
}
/* here real can not be used */

////////////////////////////
// File ComplexCalculator.java
public class ComplexCalculator {
  public Complex add(Complex c1, Complex c2) {
    /* here real can not be used */
  }
}
```

La scelta del modificatore d'accesso opportuno per uno specifico field dipende esclusivamente da come vogliamo modellare il concetto di information hiding nella classe che stiamo scrivendo. In definitiva, dipende dalla *natura* dell'entità che vogliamo modellare e che la classe rappresenta, dunque dalla conoscenza di dominio di cui disponiamo. La regola generale è che

- i field dovrebbero essere `private` – essi rappresentano lo stato interno di un oggetto, e vorremmo evitare che un utilizzatore esterno possa manipolarlo con esiti imprevisti;

- i metodi dovrebbero essere **public** se definiscono operazioni che l'oggetto deve poter svolgere, mentre dovrebbero essere **private** nel caso in cui l'operazione che viene definita è riutilizzata frequentemente da altre operazioni pubbliche della classe stessa e in tal caso devono essere invocati esclusivamente dall'interno della classe (essi sono detti metodi di utilità interna).

Le eccezioni notevoli, tuttavia, non mancano. L'identificatore **this** è un identificatore speciale (una keyword) del riferimento che fa riferimento all'oggetto stesso su cui un metodo è eseguito. Nell'esempio

```
public class Complex {
  public Complex add(Complex other) {
    return new Complex(
      this.real + other.real,
      this.imaginary + other.imaginary
    );
  }
}
```

l'identificatore **this** fa riferimento all'oggetto stesso, è implicitamente definita (quando viene menzionato un field senza la dot notation, è implicito il **this**) e permette di adoperare i field dell'oggetto stesso. Nel caso di sopra, non c'è ambiguità e pertanto non è necessario adoperare l'identificatore **this**. In altri casi è invece necessario, come ad esempio nei costruttori:

```
public class Complex {
  public Complex add(Complex other) {
    return new Complex(
      real + other.real,
      imaginary + other.imaginary
    );
  }
}
```

in questo caso non è necessario. Tuttavia, per evitare l'ambiguità,

```
public class Complex {
  private double real;
  private double imaginary;
  public Complex(double real, double imaginary) {
    this.real = real;
    this.imaginary = imaginary;
  }
}
```

nel caso di sopra è necessario adoperare l'identificatore **this**. Solitamente, questo caso si presenta esclusivamente all'interno di costruttori 'di routine', che spesso sono automaticamente generabili da degli IDE.

3.1.1 Il diagramma oggetti-riferimenti per una classe con field

Supponendo di eseguire il seguente codice,

```
Complex c1 = new Complex(7.46, -3.4567);
Complex c2 = new Complex(0.1, 9.81);
Complex c3 = c1.add(c2);
double norm = c2.getNorm();
```

Nel diagramma oggetti-riferimenti, un numero complesso viene rappresentato nella seguente maniera in Figura 3.1.

Il contenuto degli oggetti di classe `Complex` non contiene i due double, bensì due *riferimenti ad oggetti di tipo double*. Il riferimento `norm` viene riferito ad un nuovo oggetto di tipo double, al di fuori degli oggetti istanziati.

Si ricorda che gli oggetti, di per sé, hanno bisogno sia di un loro *stato interno*, che di loro *operazioni che possono compiere su se stessi* – se si rimuove qualsiasi fra le due caratteristiche, gli oggetti perdono la loro capacità descrittiva, e diventano delle mere sequenze di funzioni (se si tolgono i field) o delle strutture dati (se si rimuovono i metodi).

3.2 Riassunto del capitolo

Paradigma ad oggetti Il *paradigma ad oggetti* è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi – le strutture dati e le procedure che operano su esse sono confinate all'interno di zone circoscritte del codice, dette **classi**. Le classi possono, infine, essere istanziate per creare gli **oggetti**.

Field Un *field* o *campo* è un oggetto contenuto in un oggetto. Ciascun field può essere inteso come una vera e propria struttura dati in senso lato – si può dunque affermare che ogni oggetto può contenere varie strutture dati. L'insieme dei field di una classe è manifestazione dello *stato* di una classe, un possibile valore di un'istanza appartenente all'universo dei possibili valori di un tipo. I field sono referenziati, dunque dotati di un identificatore. I field possono essere manipolati tramite la dot notation, similmente ai metodi.

Modificatore d'accesso Un *modificatore d'accesso* è una parola chiave in Java che consente di controllare la *visibilità* (o *scope*) di un particolare campo o metodo. Esistono tre possibili valori per la visiblità:

- visibilità con modificatore d'accesso `public`: il campo o metodo è visibile *dall'esterno* della classe; è dunque accessibile, modificabile o invocabile senza restrizioni si tratta di un metodo;
- visibilità con modificatore d'accesso `private`: il campo o metodo è visibile *solo all'interno* della classe dove viene dichiarato. Non è dunque possibile, per un utilizzatore esterno, averne accesso, modificarlo o invocare il metodo;
- visibilità senza modificatore d'accesso, o *di default*: il campo o metodo, in assenza di un modificatore d'accesso specificato, sarà visibile *all'interno del pacchetto* delle classi.

I modificatori d'accessono consentono, dunque, di modellare a nostro piacimento il principio dell'information hiding e decidere di mostrare o di celare particolari campi o metodi dall'esterno della classe. Tipicamente, i field dovrebbero essere `private` ed accessibili solo mediante metodi pubblici appositi; viceversa, i metodi dovrebbero essere `public`, di modo da poter essere disponibili dall'esterno, a meno che non si tratti di metodi di utilità interna.

L'identificatore `this` Un identificatore di particolare interesse è `this`, il quale fa esplicito riferimento a *questo* oggetto dove tale identificatore è incluso. L'identificatore `this` non è obbligatorio, e risulta importante soltanto nei casi in cui è necessario eliminare delle ambiguità, come nella definizione dei costruttori dove spesso i nomi dei parametri di input hanno lo stesso nome dei field interni.

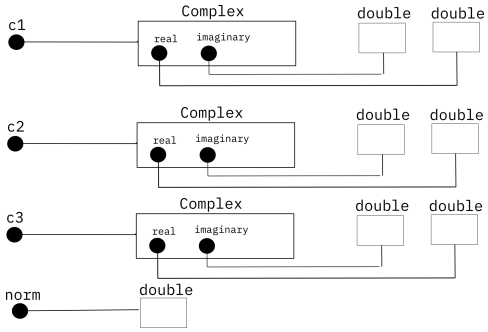


Figura 3.1: Diagramma oggetti-riferimenti successivamente all'esecuzione del codice dove vengono create tre oggetti appartenenti alla classe **`Complex`**, ed un oggetto primitivo **`norm`** di tipo double.

Capitolo 4

I pacchetti (packages)

Un certo identificatore può essere accessibile (utilizzabile), oppure no – nel caso dell'identificatore `private`, il metodo o field non è accessibile al di fuori dall'oggetto. Nel caso di metodi o campi denotati con identificatore `public`, il file `.class` vede esportati dal compilatore *gli identificatori delle classi, dei metodi e dei field di tale classe*. L'esportazione comprende sia i modificatori d'accesso private e public, con private inaccessibili dall'esterno (il compilatore ci avvisa gentilmente di ciò). Di default, il comportamento è quello di un modificatore d'accesso `public`:

- `private`: non visibile, c'è ma non si può usare;
- `public`: (esplicitamente) visibile;
- *default*: (identificatore assente) visibile.

I **pacchetti (packages)** sono un *insieme* di classi con un nome. All'interno di un package, il modificatore di accesso di default rende visibile tale identificatore *soltanto all'interno del pacchetto*. Gli identificatori di default, benché abbiano un comportamento peculiare all'interno del pacchetto, sono molto rari e tendono a sfavorire la modularità, poiché non hanno un comportamento modulare corretto. Infatti, gli identificatori di default si trovano ad essere una sorta di "via di mezzo" fra l'identificatore `private` e l'identificatore `public`: in questa maniera, tendono a non avere un comportamento netto e ben definito. Il nome di un pacchetto, dal punto di vista sintattico, è una sequenza di lettere (token) separata da punti. Non c'è una gerarchia formale, sebbene i nomi di pacchetti possano suggerire diversamente, per esempio osservando la somiglianza fra i nomi di pacchetti e i nomi di dominio. I pacchetti `java.util` e `java.util.logging` **non** hanno una vera e propria gerarchia, non esistono sotto-pacchetti o relazioni fra pacchetti. Tuttavia, i nomi sono scelti adoperando una *convenzione*, in particolare, vengono adoperati raggruppando i pacchetti secondo una logica non casuale. I pacchetti sono utili per evitare il *conflitto fra nomi*: due classi aventi lo stesso nome, infatti, non potrebbero essere "utilizzate" assieme, se non all'interno di differenti pacchetti. I nomi di pacchetti sono, solitamente, in lowercase, con *reverse institution/company name, product/internal product organization*, dove il dominio con maggior grado nella "gerarchia" è collocato più a sinistra. I nomi dei pacchetti hanno anche un forte impatto nell'organizzazione dei file e delle directory di lavoro Java.

4.0.1 Java modules

I **moduli**, introdotti in Java dalla versione 9, sono molto potenti per costruire software estremamente complesso. Tuttavia, in questo corso saranno ignorati.

4.1 Fully Qualified Name

Unendo il nome della classe al nome del pacchetto si ottiene il **Fully Qualified Name**, identificatore *univoco* della classe. Supponendo, dunque, per esempio un nome di pacchetto `java.net` ed una classe in esso contenuta di nome `Socket`, il FQN corrispondente sarà `java.net.Socket`. Mediante il FQN, ogni classe può essere univocamente riferita da qualunque punto del codice. Ad esempio, nel frammento

```
double r = 10;
it.units.shapes.Circle circle = new it.units.shapes.Circle(r);
double area = circle.area();
```

viene invocato il costruttore della classe `Circle` contenuta nel pacchetto `it.units.shapes`.

4.2 La keyword `import`

I FQN rendono il codice piuttosto verboso, in quanto sarebbe molto prolisso scrivere ogni volta da quale pacchetto vogliamo trovare una classe. Per evitare questo fenomeno, si adopera la keyword **import**, solitamente all'inizio del file:

```
import it.units.shapes.Circle;
public class ShapeCompound {
  private Circle circle;
  /* ... */
}
```

La parola chiave `import` permette di scrivere soltanto `Circle` al posto di `it.units.shapes.Circle`, e rappresenta dunque una scorciatoia sintattica, non una qualsivoglia tipologia di importazione[1]. Con `import`, dichiaro l'uso dell'abbreviazione con il solo nome della classe, proveniente dal pacchetto indicato. La sintassi prevede anche l'uso dell'asterisco per riferirsi a classi multiple dello stesso pacchetto (*star import*) – al giorno d'oggi si evita di utilizzarlo, poiché è l'IDE a creare, in automatico, gli import necessari al momento della scrittura del codice. Il Fully Qualified Name è, a volte, necessario nel caso di due *classi omonime*, appartenenti a pacchetti differenti.

Quando il compilatore processa il codice sorgente `.java` e trova una classe `C`, egli deve conoscere i suoi metodi, e field, oltre che il nome, la signature, e i modificatori d'accesso. C'è oltretutto una priorità con la quale il compilatore va a cercare una determinata classe `Classname`, priva di Fully Qualified Name, nel codice:

1. nel codice sorgente del file attuale `.java`;
2. nello stesso pacchetto (nella medesima directory) – questo rende non necessario porre l'`import` nel caso di classi provenienti dallo stesso pacchetto;
3. nei package importati mediante star import.

Esiste un pacchetto speciale, `java.lang`, le cui classi sono **disponibili di default** – ciò equivale ad uno star import `import java.lang.*` implicitamente presente all'inizio del file.

[1] Serve soltanto al programmatore umano, per rendere agevole la lettura del codice – non comporta pre-caricamento del codice, lettura del codice, o altri effetti pratici se non quello di rendere più agevole la scrittura del codice.

4.2.1 Sintassi dei Fully Qualified Name

I pacchetti e i nomi di classe non possono essere identificati soltanto guardando il FQN. Infatti, se prendiamo come esempio `it.units.UglySw.Point`, possono insorgere delle ambiguità:

- `Point` potrebbe essere la classe;
- `UglySw.Point` e `UglySw` sono classi;
- `units.UglySw.Point`, `units.UglySw` e `units` sono tutte classi, mentre `it` è un pacchetto;
- sono tutte classi annidate.

4.3 Il modificatore `static`

Un *field* di una classe C può essere definito con il modificatore di non-accesso `static`, in modo che:

- l'oggetto è **unico** per tutti gli oggetti di quell'istanza C;
- **esiste sempre**, anche se nessuna istanza di C viene istanziata;

Un field statico ha come riferimenti *condivisi* con tutte le istanze della medesima classe (potenzialmente anche nessuna istanza).

```
public class Greeter {
  public static String msg;
  private String name;
  /* ... */
}
Greeter g1 = new Greeter();
Greeter g2 = new Greeter();
```

Anche i metodi possono essere statici. Un metodo statico:

- quando invocato, non è applicato all'istanza della classe sulla quale è invocato. Cambia dunque, l'idea stessa di scopo ed utilizzo di un metodo, che non agisce più sulla medesima istanza della classe. Un metodo statico dunque esiste a prescindere dagli oggetti della classe a cui appartiene, e può essere, quasi paradossalmente, invocato *anche quando non esiste alcuna istanza della classe a cui appartiene*;
- da dentro un metodo `static` si possono adoperare **solo** field o altri metodi `static`: questo è necessario, poiché devono poter essere eseguiti anche in assenza di istanze (per via di questo motivo, campi e metodi non statici non esisterebbero). In altre parole, i metodi e i field static sono unici e devono poter esistere o produrre elaborazioni a prescindere dalle istanze della classe. La correttezza di questo assunto è verificata dal compilatore.

L'identificatore `static` viene adoperato facendo, per convenzione, riferimento mediante dot notation con il *nome della classe*, e **non** con la dot notation sul riferimento che fa riferimento ad un'istanza esistente – questo è fatto per evitare qualsiasi ambiguità sul fatto che il metodo statico non può avere effetto sull'oggetto. Se infatti si adopera la dot notation con un metodo statico su un riferimento ad un'istanza della classe, allora si potrebbe lasciar intendere

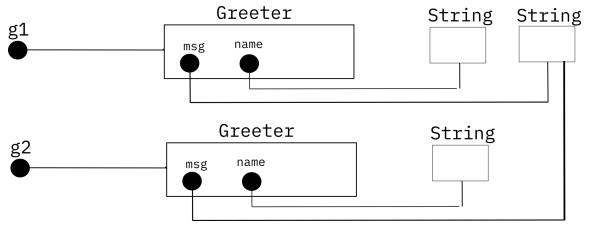


Figura 4.1: Relazioni fra entità con due oggetti di classe `Greeter` contenenti un oggetto con modificatore `static`.

che l'oggetto in sé abbia un ruolo nell'invocazione e nell'elaborazione del metodo statico, cosa non vera. Gli oggetti, infatti, non hanno un vero e proprio ruolo nell'invocazione e fungono soltanto come una sorta di "contenitore" dei metodi e dei field `static`. Un metodo dovrebbe essere progettato come `static` nel caso in cui esso rappresenti un'operazione che non coinvolga nessuna istanza di tale classe – in teoria, non dovrebbe nemmeno essere definita nella classe. Tuttavia, le operazioni sono poste in una classe piuttosto che in un'altra per comodità e per inerenza con la classe in sé. L'identificatore static dovrebbe anche essere adoperato per metodi `private`, qualora le condizioni di cui sopra siano rispettate, anche se non sono disponibili al di fuori della classe (concettualmente, vanno comunque posti come `static` per correttezza nei confronti degli altri sviluppatori). Tipicamente, tali metodi sono utili ad altri metodi statici. Come linea guida, se un metodo *non opera sullo stato di un'istanza di una classe*, esso dovrebbe essere `static`.

```
public class Dog {
  public static Dog getCutest(Dog dog1, Dog dog2) { /* ... */ }
  // Pseudo-costruttore: il nome non e' un'azione
  // serve a costruire un oggetto (createFromString,
      buildFromString)
  public static Dog fromString(String name) { /* ... */ }
}
```

4.4 Comprensione di Hello World!

Dato il programma "Hello World!",

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

La signature `public static void main(String[] args)`: main è il nome del metodo convenzionale, con nessuna variante ammessa sulla signature, dell'*entry point* dell'esecuzione sulla JVM – almeno una classe deve avere un metodo `main` per poter eseguire un programma. Il metodo main deve avere tali caratteristiche:

- essere `public`: invocabile da qualunque parte, anche al di fuori della classe;
- essere `static`: deve essere invocabile senza che alcuna istanza di `Greeter` sia stata istanziata;
- non avere alcun valore di ritorno (`void`): non ritorna nulla, è una convenzione;
- soltanto `args` può essere modificato, ma per convenzione si lascia così com'è. Gli argomenti sono passati dal sistema operativo – nel caso non ce ne siano, l'argomento è vuoto. La separazione fra argomenti dipende dal sistema operativo, in Linux si separano con il carattere di spazio.

L'istruzione `System.out.println("Hello World!");` invoca il metodo `println`, del field statico `out` appartenente alla classe `System`. Un'interpretazione alternativa avrebbe potuto essere che `println` è un metodo statico nella classe `out`, appartenente al pacchetto `System`, ipotesi da scartare per via della convenzione che non sarebbe rispettata nel caso di `System`, con un nome di pacchetto ad S maiuscola e con `out` classe a lettere tutte minuscole. La classe `System` è già presente, poiché è appartenente al pacchetto `java.lang`, di default già importato dal compilatore. Dunque, `java.lang.System` è il Fully Qualified Name per la classe `System`. Guardando la documentazione per la classe `System`,

`static PrintStream err` The "standard" error output stream. `static InputStream in` The "standard" input stream. `static PrintStream out` The "standard" output stream.

`out` risulta essere un field di tipo `PrintStream`, lo "standard" output stream[2].

Dentro `println` vi è una stringa, che viene immediatamente istanziata e passata (senza riferimento) – `println` è un metodo che stampa una stringa.

`void println(int x)` Prints an integer and then terminate the line. `void println(long x)` Prints a long and then terminate the line. `void println(Object x)` Prints an Object and then terminate the line. `void println(String x)` Prints a String and then terminate the line.

4.4.1 Classi coinvolte

Le classi coinvolte sono le classi definite e utilizzate. In particolare, sono coinvolte:

- la classe `Greeter` definita nel codice;
- le classi definite altrove, ma usate nel codice sono `System`, `String`, `PrintStream`;
- `System` e `String` sono incluse nel pacchetto `java.lang`, mentre `PrintStream` è contenuta nel pacchetto `java.io`. Tuttavia, l'import è una scorciatoia: non è necessario importare `PrintStream` perché non è direttamente coinvolto nel codice – sarà invece importato o comunque presente nella classe `System`.

4.5 Riassunto del capitolo

Pacchetti Java Un *pacchetto* o *package* in Java è un insieme di classi dotato di un nome. Il nome di un pacchetto è una sequenza di token separata da punti. In Java non esiste una gerarchia formale fra i token, e possono essere liberamente scelti – gli sviluppatori Java, tuttavia, cercano di porre le classi all'interno dei pacchetti con un certo ordine. I pacchetti evitano il conflitto fra nomi, e raccolgono a sé varie tipologie di classi aventi funzionalità correlate. I nomi di pacchetti sono in lowercase, con la notazione *reverse*: nome di compagnia/istituto, prodotto/organizzazione interna, nome specifico.

Fully Qualified Name Un *fully qualified name* è un identificatore univoco della classe all'interno dell'ambiente Java. Esso comprende sia il nome del pacchetto che il nome della classe, consentendo di eliminare ogni ambiguità nel caso di omonimie fra classi provenienti da pacchetti differenti. Il Fully Qualified Name è composto dai token del nome di pacchetto, con il nome della classe separato da un punto. Il FQN è necessario nel caso di ambiguità e omonimie.

La parola chiave `import` La parola chiave `import` rappresenta una scorciatoia sintattica, ed indica al compilatore Java che sarà sufficiente scrivere il solo nome della classe al posto del FQN, senza dunque scrivere anche il pacchetto dal quale la classe proviene. Esso rappresenta soltamente un'abbreviazione, e non sta a significare un qualsivoglia "import" della classe da parte del compilatore, o all'inizio del file sorgente.

Lo *star import* Uno *star import* è un tipo di import dove vengono, in un colpo solo, importate (in senso Java) tutte le classi appartenenti ad un pacchetto. Lo star import è effettuato tramite il carattere dell'asterisco, a mo' della wildcard Linux *, che sta ad indicare "tutte le possibili occorrenze di un carattere o di un'espressione".

Il caso particolare della classe `java.lang` La classe `java.lang` è importata di default. Le sue classi sono dunque disponibili per l'uso senza indicare il FQN. È l'equivalente di uno star import per tale pacchetto Java.

[2] Typically this stream corresponds to display output or another output destination specified by the host environment or user.

Il modificatore `static` Il modificatore `static` può essere applicato ad un metodo o un field di una classe, e si applica durante la definizione di esso. Introducendo un modificatore `static`, si dichiara che l'oggetto è **unico** per tutti gli elementi di quell'istanza di classe `C`, e che tale oggetto **esiste sempre**, prima ancora che un'istanza di tale classe sia stata creata. Di fatto, il modificatore `static` consente di produrre metodi e field che *non hanno bisogno* di oggetti esistenti - tipicamente ciò è utile per creare classi di utilità o metodi che manipolano più oggetti. Un metodo `static` può operare **solo** su field dichiarati come `static`: ciò è desiderabile, in quanto un metodo statico non dovrebbe poter operare su oggetti esistenti, né dovrebbe aver bisogno di istanze della classe a cui appartiene. Per tale ragione e per convenzione, si è esortati ad invocare tali metodi *direttamente sulla classe* anziché su oggetti istanziati (ciò è possibile in quanto ogni metodo static condivide i riferimenti con tutte le istanze della medesima classe), per evitare ogni sorta di confusione.

Capitolo 5

Costruzione di una classe

5.1 Le basi della programmazione

5.1.1 Controllo del flusso d'esecuzione

Le tipiche istruzioni di controllo del flusso sono presenti in java:

- if then else;
- while;
- for;
- switch;
- break;
- continue;
- return;

5.1.2 Basic Input/Output

Tipicamente, lo standard input proviene dalla tastiera, sotto forma di cartteri inseriti dall'utente. Lo standard output è, solitamente, anche'esso sotto forma di caratteri, via terminale. Di solito essi sono input o output di oggetti della classe `String` o altri tipi primitivi, anche se tipi differenti verranno visti in futuro nel corso. Java è, solitamente, eseguito all'interno di un *terminale*. Il modo più basilare per produrre un output è quello di utilizzare un'istanza della classe `PrintStream`: il field statico `out`[1] della classe `System`. La classe `PrintStream` modella un **flusso** per la stampa: ha una serie di metodi, chiamati `println`, espressi nell'elenco seguente:

`void println()` Terminates the current line by writing the line separator string.
`void println(boolean x)` Prints a boolean and then terminate the line.

`void println(char x)` Prints a character and then terminate the line.

`void println(char[] x)` Prints an array of characters and then terminate the line.

`void println(double x)` Prints a double and then terminate the line.

`void println(float x)` Prints a float and then terminate the line.

`void println(int x)` Prints an integer and then terminate the line.

`void println(long x)` Prints a long and then terminate the line.

`void println(Object x)` Prints an Object and then terminate the line.

`void println(String x)` Prints a String and then terminate the line.

[1] Ci si aspetta che un'applicazione veda già il sistema alla partenza, senza che debba essere istanziata un'entità della classe `System`.

Ugualmente, per il metodo `print`, che però non termina la linea con una sequenza di caratteri newline. Un metodo più evoluto, `printf`, può stampare informazioni mediante un *template* predefinito (mediante placeholders, riempiti al runtime con informazione passata come ulteriori argomenti).

Per leggere l'input, adoperiamo un oggetto della classe `BufferedReader`:

```
BufferedReader reader = new BufferedReader(
  new InputStreamReader(System.in) // lettore di input stream
);
/* ... */
String line = reader.readLine();
```

il metodo `readLine`, quando evocato, legge una riga e crea un oggetto della classe `String`. La chiamata in sé richiede aggiungere `throws Exception` dopo la signature del `main`.

L'alternativa è l'uso di un'istanza della classe `Scanner`. Lo Scanner modella una componente software che scansione uno stream di input alla ricerca di sequenze di byte interessanti:

```
Scanner scanner = new Scanner(System.in);
/* ... */
String s = scanner.next();
int n = scanner.nextInt();
double d = scanner.nextDouble();
```

Tali metodi fanno tre cose:

- leggono una linea dall'`InputStream` passatogli;
- dividono la linea in *token*, col separatore spazio (`" "`)[2];
- converte e restituisce il primo token del tipo indicato.

Successive chiamate di next forniscono i prossimi token.

Se si adopera il BufferedReader, può essere utile adoperare i vari metodi concatenati, come nel frammento seguente

```
String line = reader.readLine();
int n = Integer.parseInt(line); // pseudo-constructor
float f = Float.parseFloat(String);
double d = Double.parseDouble(String);
```

5.1.3 Array

Gli array in Java sono molto simili a quelli in C. Un **array** è una sequenza di oggetti dello stesso tipo, con lunghezza *fissata* che non può mai cambiare al runtime. Sono cosiddetti "0-based indexing"[3]. Ciascun oggetto è accessibile mediante l'operatore `[]` applicato al riferimento. Gli array **contengono i riferimenti**.

```
String[] firstNames;
String[] lastNames = new String[3]; // crea array con dimensione 3
    di String
lastNames[1] = new String("Medvet");
```

Per i nomi di riferimenti ad array è, tipicamente, la *forma plurale* del nome che si darebbe al riferimento di un'istanza del tipo presente nell'array (e.g., *dog* diventa *dogs*, *cat* diviene *cats*, e così via). Definendo gli array, si possono adoperare due sintassi,

- `Person persons[]`: fortemente sconsigliato;

[2] In alternativa, è possibile fornire un diverso *delimiter* come argomento del metodo. Per ulteriori informazioni, si cerchi nella documentazione ufficiale.

[3] Questo assunto, che per uno studentello di C parrebbe ovvio, non è affatto scontato: sia il linguaggio Matlab che `R` fanno uso di array "1-based", cioè che contano a partire da 1.

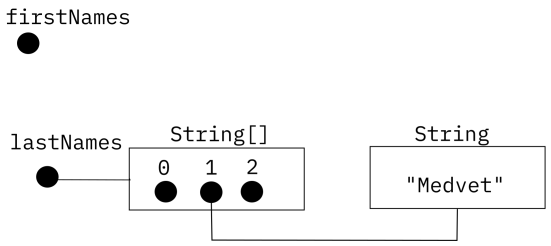
- `Person[] persons`: fortemente consigliato;

Per creare gli array, due metodi equivalenti:

```
String[] dogNames = {"Simba", "Gass"};
//same of new String[]{"Simba", "Gass"}
////////////////////////////////////
String[] dogNames = new String[2];
dogNames[0] = "Simba"; //same of = new String("Simba");
dogNames[1] = "Gass";
```

Per conoscere la lunghezza di un array, esiste un field chiamato `length`, che tuttavia può soltanto essere letto, e non modificato:

```
String[] dogNames = {"Simba", "Gass"};
System.out.println(dogNames.length); //prints 2
dogNames = new String[3];
System.out.println(dogNames.length); //prints 3
```



5.1.4 Iterazione all'interno di array

L'iterazione su array può avvenire in due forme, una "classica" e una dove avviene una semplice iterazione senza contatore, *itera su tutti gli elementi dell'array*.

```
String[] dogNames = {"Simba", "Gass"};
for (int i = 0; i<dogNames.length; i++) {
  System.out.println("Dog " + i + " name is " + dogNames[i]);
}
////////////////////////////////////////
String[] dogNames = {"Simba", "Gass"};
for (String dogName : dogNames) {
  System.out.println("A dog name is " + dogName);
}
```

L'indice `i`, tuttavia, non è disponibile all'interno della seconda iterazione.

5.1.5 I varargs

Nella signature di un metodo, l'*ultimo parametro di input*, se di tipo array, può essere specificato con tre puntini `...` anziché `[]`. Le due istruzioni di seguito sono assolutamente equivalenti,

```
public static double max(double... values) { /* 1 ... */}
public static double max(double[] values) { /* 2 ... */}
```

tuttavia, la seconda permette di creare un vettore opportunamente in maniera più comoda:

```
double max = max(4, 3.14, -1.1); //values cong double[3]; OK for 1
max = max(); //values cong double[0]; OK for 1
```

```
max = max(new double[2]{1, 2}); //Ok!; OK for 1 and 2
```

Tuttavia, c'è una limitazione importante, che sia l'ultimo parametro di input – questo evita ambiguità al prezzo di espressività. Un esempio di utilizzo:

```
public static double max(double... values) {
  double max = values[0];
  for (int i = 1; i<values.length; i++) {
    max = (max > values[i]) ? max : values[i];
  }
  return max;
}
```

Un caso in cui vi sono ambiguità nella notazione è il seguente:

```
// does NOT compile!
public static int intersectionSize(String... as, String... bs) {
  /* ... */
}
intersectionSize("hello", "world", "cruel", "world");
```

5.1.6 Command line arguments

Disponibile soltanto come contenuto degli argomenti `main`:

```
public class ArgLister {
  public static void main(String[] args) {
    for (String arg : args) {
      System.out.println(arg);
    }
  }
}
```

darà come output, se adoperato,

```
eric@cpu:~$ java ArgLister Hello World
Hello
World
```

Un diagramma di tale esecuzione è il seguente in Figura 5.1:

dove all'interno del `for` loop vi è un ulteriore riferimento, con nome di identificatore `arg`, che riferisce, rispettivamente, al primo e al secondo oggetto dell'array degli argomenti `args`, a seconda del ciclo.

5.1.7 Operazioni fra stringhe

Operare con stringhe si fa tramite la classe opportuna.

- empty: `String s = new String();`
- specified: `String s = new String("hi!");`
- the same of `String s = "hi!";`
- same of another: `String s = new String(otherString)`.

Altre sono, ad esempio (slide 130),

> `int compareTo(String anotherString)` Compares two strings lexicographically.

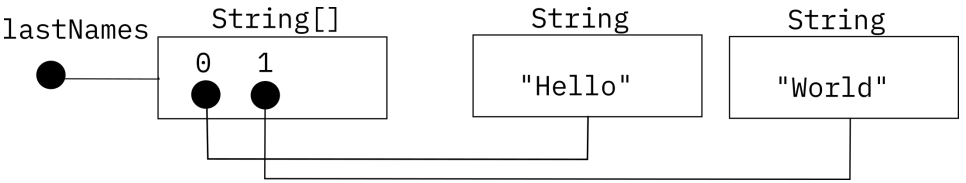


Figura 5.1: Esempio di cattura di argomenti da linea di comando.

`int compareToIgnoreCase(String str)` Compares two strings lexicographically, ignoring case differences.

`boolean endsWith(String suffix)` Tests if this string ends with the specified suffix.

`int indexOf(int ch)` Returns the index within this string of the first occurrence of the specified character.

`int indexOf(int ch, int fromIndex)` Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

`int indexOf(String str)` Returns the index within this string of the first occurrence of the specified substring.

`int indexOf(String str, int fromIndex)` Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

`int length()` Returns the length of this string. boolean matches(String regex) Tells whether or not this string matches the given regular expression.

`String replaceAll(String regex, String replacement)` Replaces each substring of this string that matches the given regular expression with the given replacement.

Il metodo statico **`format`** consente di ritornare una stringa formattata utilizzando uno specifico formato indicato nell'argomento. In pratica,

```
String reference = String.format(
  "FPR=%4.2f\tFNR=%4.2f%n",
  fp / n,
  fn / p
);
//results in FPR=0.11 FNR=0.10
```

Le stringhe sono **immutabili**, e indicate nella documentazione come costanti. Tuttavia, a volte la documentazione è contradditoria:

`String concat(String str)` Concatenates the specified string to the end of this string.

`String toUpperCase()` Converts all of the characters in this String to upper case using the rules of the default locale.

Anche se, leggendo con cura, si scopre che

`public String concat(String str)` Concatenates the specified string to the end of this string.

If the length of the argument string is 0, then this String object is returned. Otherwise, a String object is returned that represents a character sequence that is the concatenation of the character sequence represented by this String object and the character sequence represented by the argument string.

Esempio con risoluzione in diagramma riferimenti-oggetti (Figura 5.1.7).

```
// black color in diagram
String s1, s2;
s1 = "hamburger";
s2 = s1.substring(3, 7);
System.out.print("s1 = ");
System.out.println(s1);
s1 = s1.replace('a', 'b');
String[] names = { "John", "Fitzgerald", "Kennedy"};
// blue color in diagram
String firstInit, middleInit, lastInit;
firstInit = names[0].substring(0, 1);
middleInit = names[1].substring(0, 1);
lastInit = names[2].substring(0, 1);
firstInit.concat(middleInit).concat(lastInit);
```

5.1.8 Sintassi alternativa per il metodo `concat`

Una possibile (ed alternativa) scorciatoia sintattica per la concatenazione fra stringhe avviene mediante l'operatore **+** . L'operatore **+** viene valutato a partire dalla sinstra (si dirà che è *associativo a sinistra*). Alcuni esempi sono pertanto i seguenti,

```
String s1 = "today is ";
String s2 = s1.concat(" Tuesday").concat("!");
```

che compie le stesse operazioni del frammento seguente,

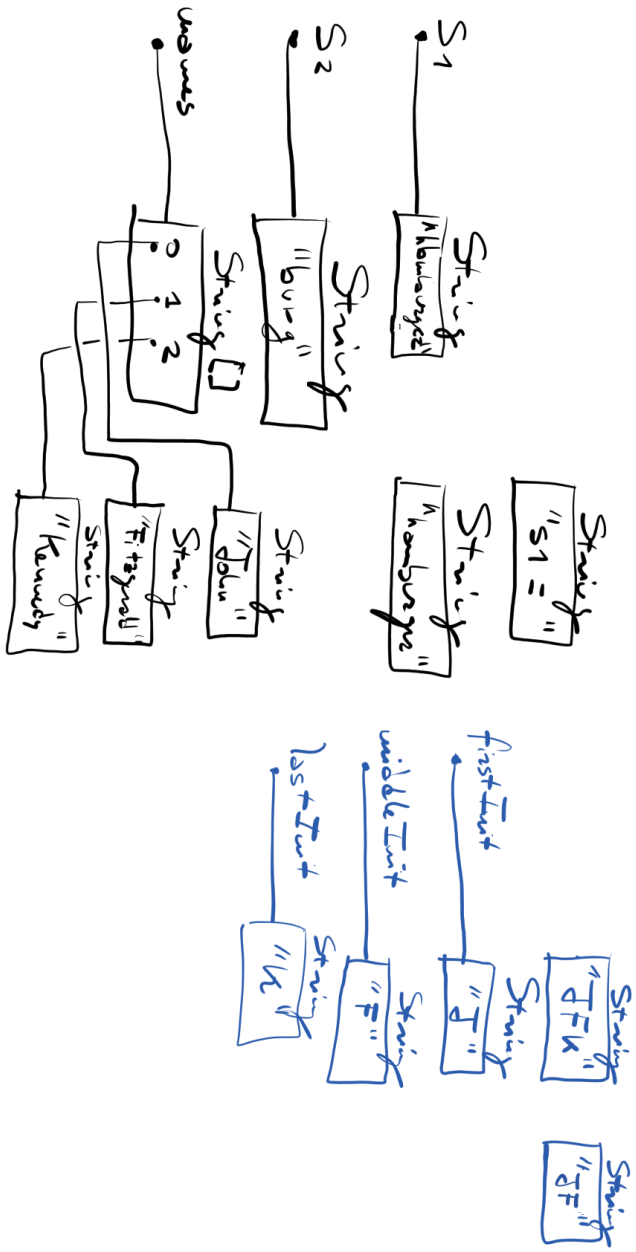
```
String s2 = s1 + " Tuesday" + "!";
```

come la creazione degli oggetti che si avrebbero con la prima sintassi per la concatenazione. A livello del compilatore, le operazioni eseguite sono, infatti, le stesse: la sintassi offre soltanto all'utilizzatore un modo più agevole (e forse più naturale) per esprimere la concatenazione fra stringhe, benché al compilatore l'operazione risulti esattamente la stessa.

Possono essere concatenati anche operandi di tipo differente dalle stringhe. In particolare, possiamo eseguire

```
int age = 42;
String statement = "I'm " + age + " years old";
```

il quale dapprima convertirà implicitamente l'intero **age** nella corrispettiva stringa, e successivamente procederà con le concatenazioni, da sinistra verso destra. Nel caso l'operando non stringa sia un *tipo non primitivo*, il meccanismo sarà chiaro in seguito.



5.1.9 Utilizzo dei tipi primitivi

I tipi primitivi non vengono adoperati come gli oggetti appartenenti alle classi. In particolare, per essi non è previsto l'uso della keyword **new**, non hanno metodi e field e pertando non possono vedere applicata su loro stessi la dot notation. Possono essere operandi di *operazioni binarie*, ad esempio

```
int n = 512;
double d = 0d;
char a = 'a';
boolean flag = false;
flag = flag || true;
double k = 3 * d - Math.sqrt(4.11);
```

dove il metodo `Math.sqrt(4.11)` avrà una signature

```
public static double sqrt(double)
```

ed apparterrà alla classe `Math`.

I tipi primitivi godono di un'inizializzazione di default. Essi, se creati ma non inizializzati, sono inizializzati in automatico al loro valore di default:

- 0 o l'equivalente dello zero per interi, double, float e char;
- `false` per i booleani.

Questo però vale soltamente per i field appartenenti agli oggetti istanziati: **non** vale per le variabili locali, le quali non vengono inizializzate e dunque producono un errore in compilazione se utilizzate.

I tipi primitivi godono di un'inizializzazione di default anche all'interno degli array. In particolare, durante la creazione di un array con tipi primitivi, essi vengono posti al loro valore di default qualora non fosse specificato altrimenti. Gli array richiedono la keyword **new**: all'invocazione del costruttore, gli elementi interni vengono inizializzati al loro valore di default (proprio come se fossero dei field).

I field possono essere inizializzati all'interno dell'oggetto, direttamente: si dice che sono inizializzati *inline*:

```
public class Greeter {
  private String greet = "Hello";
  public void greet() {
    System.out.println(greet + " World!");
  }
}
```

In questa maniera, è equivalente al costruttore che produce la stessa assegnazione, come primissima cosa[4]. A seguito di un'assegnazione inline, *tutti gli oggetti istanziati partono con il medesimo riferimento indicato*. Ad esempio, il seguente codice,

```
public class Greeter {
  private String greet = "Hello";
```

[4] L'inizializzazione dei campi è eseguita *prima* del primo statement di qualsiasi costruttore appartenente a quella classe

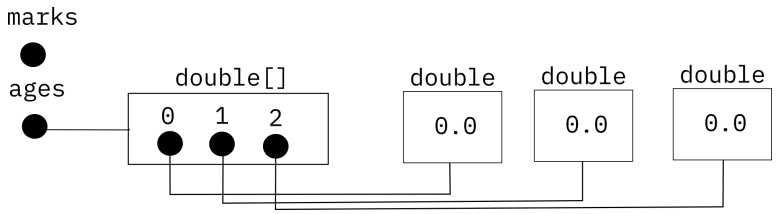
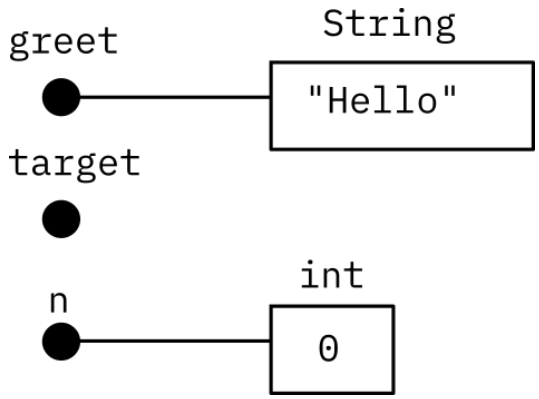


Figura 5.2: Vettore di **double** inizializzato al valore di default. Ogni campo dell'array è inizializzato, sempre.

```
  private String target;
  private int n;
  public void greet() {
    System.out.println(greet + " " + target);
    System.out.println(n);
  }
}
```

produrrà il diagramma riferimenti-oggetti illustrato in Figura 5.1.9.



L'inizializzazione dei tipi primitivi avviene anche all'interno degli array. Ad esempio,

```
int[] marks;
double[] ages = new double[3];
```

produrrà il diagramma in Figura 5.2.

Per i tipi primitivi, l'assegnazione copia il contenuto invece che referenziare il medesimo oggetto. Ad esempio,

```
String s1 = "hello!";
String s2 = s1;
double d1 = 3.14;
double d2 = d1;
```

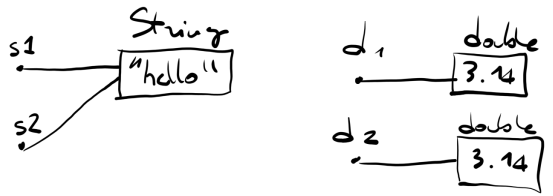
produrrà il diagramma in Figura 5.1.9.

5.1.10 Il tipo `null`

Un tipo speciale è il tipo `null`. Si tratta di un valore speciale che può essere referenziato da un riferimento di qualsiasi tipo non primitivo. I field dei tipi non primitivi sono implicitamente riferiti a `null` – ciò avviene diversamente dai tipi primitivi, i quali sono invece impostati al loro valore predefinito. Per le variabili locali avviene la stessa cosa: se non inizializzate, portano ad un errore di compilazione. Qualsiasi riferimento ad oggetto di tipo non-primitivo può *esplicitamente* essere fatto riferire a `null`, di modo che il riferimento in sé non riferisca nulla. In termini pratici,

```
String s1 = "hi";
String s2 = null;
String s3 = null;
```

produrrà dei riferimenti s2 ed s3, i quali riferiranno ad un'area di memoria (`null`), la quale però verrà omessa nei diagrammi per semplicità (si dirà che non riferiscono "a niente"). Altro esempio si ha nello snippet seguente:



```java
public class Greeter {
  private String s1 = "hi";
  private String s2 = null;
  private String s3;
  private int n;
}
```

La dereferenziazione può anche essere valutata con l'uguaglianza `==`, in particolare si avrà

```java
String s1 = "hi";
String s2 = s1;
s1 = null;
boolean isNull = s1 == null;
```

dove il valore di `isNull` sarà `true`.

5.2 Riassunto del capitolo

Input L'input in Java proviene solitamente dalla tastiera, sotto forma di caratteri inseriti dall'utente. Un input che proviene da tastiera (se si preferisce, da terminale) è detto **standard input**. Vi sono due modi per "catturare" lo standard input, il primo è mediante un'istanza della classe `BufferedReader`, il secondo mediante un'istanza di `Scanner`.

Output L'output in Java è solitamente anch'esso sotto forma di caratteri testuali su terminale, e viene pertanto detto **standard output**. Lo standard output può essere prodotto mediante un'istanza della classe `PrintStream`, ad esempio con il metodo `print()` o `println()`; in alternativa e per compiti più complessi, si utilizza il metodo `printf()`, sempre appartenente alla medesima classe.

Array Gli array in Java hanno *lunghezza fissata*, sono "0-based" (contano gli elementi a partire da 0) e contengono i riferimenti ad oggetti di un determinato tipo. Tipicamente, per i nomi di array si adotta il **plurale** della parola che denota gli oggetti che l'array contiene. Di indiscussa utilita vi è il field `length`, il quale ci fornisce la lunghezza del vettore in questione.

Iterazione su array Oltre alla "classica" forma di iterazione su array con ciclo `while` o `for` (chi conosce il C la dovrebbe già padroneggiare), esiste una seconda possibilità, con il cosiddetto *iteratore* - un iteratore è un oggetto del medesimo tipo contenuto nell'array che, in un ciclo `for`, va a pescare uno ad uno tutti gli elementi dell'array, con indice in ordine crescente. Attenzione, perché così facendo l'indice non sarà disponibile all'interno del ciclo `for`, cioè non sappiamo esattamente a che iterazione siamo (se abbiamo bisogno di saperlo, faremo un'iterazione alla vecchia maniera).

I `varargs` I `varargs`, abbreviazione di *var*iable *arg*ument*s*, permettono di inserire un numero arbitrario di parametri di input, purché siano rispettate determinate condizioni:

- l'**ultimo** parametro di input è di tipo *array*;
- i parametri in argomento sono del medesimo tipo;
- vi è un **solo ed unico** `vararg` nella signature di un metodo.

Dunque, un `vararg` si definisce con la sintassi `double... values` anziché `double[] values`, e ha la particolare utilità di poter definire un nuovo oggetto *anche* solo ponendo i valori dell'array direttamente nell'argomento, ad esempio così: `double max = max(4, 3.14, -1.1);`.

Parametri a linea di comando Java supporta il passaggio di parametri di riga di comando. Essi sono contenuti nel `main`, più precisamente nel

vettore `args[]`, a partire dall'indice 0. Si osservi che essi sono oggetti di tipo `String`, *separati da spazi*.

Operazioni fra stringhe Le operazioni fra stringhe che siano degne di nota si trovano tutte nella documentazione. Tuttavia, è utile rimarcare la scorciatoia sintattica per il metodo `concat()`, rappresentata dall'operatore "`+`" fra stringhe - l'operatore prende due stringhe, e le *concatena*. L'operatore è anche in grado di concatenare stringhe e oggetti di altro tipo: lo fa, infatti, invocando implicitamente il metodo `toString()` dell'oggetto in questione.

Inizializzazione dei tipi primitivi I tipi primitivi hanno come inizializzazione un valore di default, naturalmente supponendo di non fornirne uno noi. Per i numeri il valore di default equivale allo 'zero' (`0` per gli interi, `0.0` per i numeri con la virgola mobile, e così via), mentre per i `boolean` il valore di default è `false`. Un po' di attenzione va prestata al caso dei `char`: essi sono inizializzati al corrispondente carattere codificato con lo 'zero', in questo caso `'\u0000'`, che corrisponde al carattere `null` che di fatto non è stampabile (non si può stampare il 'niente'). I tipi primitivi godo **sempre** di un'inizializzazione di default, siano essi collocati all'interno di array o come field di oggetti. In questo ultimo caso, si possono inizializzare dentro l'oggetto (anche quindi non all'interno del costruttore), pertanto si dirà che essi sono inizializzati **inline**. L'inizializzazione inline corrisponde a porre i valori indicati ai field come *prima* azione all'interno di un costruttore qualsiasi.

Inizializzazione dei tipi non primitivi I tipi non primitivi non godono di un'inizializzazione di default. I riferimenti ad oggetti di tipo non primitivo che non sono stati inizializzati faranno riferimento a

Assegnazione di un tipo primitivo L'assegnazione per i tipi primitivi copia il valore, non il riferimento (diversamente da quanto accade per gli oggetti di tipo non primitivo). `null`.

Il tipo `null` Il tipo `null` rappresenta *il nulla*. Esso è un valore speciale che può essere referenziato da un riferimento di qualsiasi tipo, non primitivo. Invocare qualcosa che fa riferimento a `null` comporta un errore di compilazione. Esiste un metodo apposito, `isNull()`, che ci fornisce un booleano e che ci consente di capire se un oggetto in particolare è `null` oppure no.

Capitolo 6

Ereditarietà delle classi

Sia dato il seguente frammento di codice, dove viene definita la classe **Greeter**, che compie un "saluto"

```
import java.util.Date;
public class Greeter {
  public static void main(String[] args) {
    System.out.print("Hello World! Today is: ");
    System.out.println(new Date());
  }
}
```

Esso produrrà un output della forma:

```
eric@cpu:~$ java Greeter
Hello World! Today is: Mon Mar 16 10:36:13 UTC 2020
```

In parole povere, il codice stampa una *data*. Il metodo **println**, stampa varie cose, fra cui, secondo la documentazione,

> **void println()** Terminates the current line by writing the line separator string.
>
> **void println(boolean x)** Prints a boolean and then terminate the line.
>
> **void println(char x)** Prints a character and then terminate the line.
>
> **void println(char[] x)** Prints an array of characters and then terminate the line.
>
> **void println(double x)** Prints a double and then terminate the line.
>
> **void println(float x)** Prints a float and then terminate the line.
>
> **void println(int x)** Prints an integer and then terminate the line.
>
> **void println(long x)** Prints a long and then terminate the line.
>
> **void println(Object x)** Prints an Object and then terminate the line.
>
> **void println(String x)** Prints a String and then terminate the line.

Si osservi il fatto che, nello specifico, *non esiste* un **println** in grado di ricevere un oggetto di tipo **Date**, cioè nella documentazione non esiste una voce che illustra un metodo **println** avente nella signature un argomento di classe **Date**.

Similmente, il seguente frammento

```
import java.util.Date;
public class Greeter {
  public static void main(String[] args) {
    System.out.println("Hello World! Today is: " + new Date());
  }
```

```
}
```

produce un output della forma

```
eric@cpu:~$ java Greeter
Hello World! Today is: Mon Mar 16 10:36:13 UTC 2020
```

dove la data è, almeno in apparenza, trasformata in qualche sorta di oggetto `String`.

L'**ereditarietà** è una proprietà *statica* del linguaggio – essa ha effetto al *tempo della compilazione*. Essa rende possibile definire un **nuovo tipo a partire da un tipo preesistente**, specificando *solo le parti nuove o modificate*. Soltanto dunque le nuove parti, metodi e campi, saranno definiti. La sintassi è la seguente: si aggiunge la parola chiave `extends`, seguita dall'identificatore del tipo che si intende estendere. Ad esempio,

```
public class Derived extends Base {
  /* ... */
}
```

la classe `Derived`, appena definita, estende `Base` – i metodi e i campi preesistenti saranno *ereditati* dalla classe `Derived`. Una conseguenza di tutto ciò è che non si possa "de-definire" metodi esistenti nella classe `Base`. Nella definizione di tale classe che estende, sarà possibile fare due cose:

- definire **nuovi** field e metodi – metodi e field non possono essere nascosti o rimossi;
- **ridefinire** field e metodi.

Si dice che la classe `Derived` **estende**, **eredita** o **deriva da** `Base`.

6.0.1 La classe `Object`

Esiste una classe particolare in Java, definita nella JVM: la classe `Object`. Ogni classe in Java eredita da `Object` – se è assente, in automatico eredita da Object. I due frammenti seguenti sono perfettamente equivalenti:

```
public class Greeter {
  /* ... */
}
```

```
public class Greeter extends Object {
  /* ... */
}
```

Dunque, la classe `Object` è alla radice della gerarchia delle ereditarietà. Se definiamo una classe con extends,

```
public class EnhancedGreeter extends Greeter { /* ... */ }
```

possiamo dire che esiste una relazione ad 'albero di ereditarietà' fra EnhancedGreeter e Greeter, con la radice di tale albero che eredita dalla classe Object. Altri esempi possono essere la classe `BufferedReader`, la quale eredita dalla classe `Reader`, la quale a sua volta avrà tutti i metodi della classe radice Object. Ciò però, dal punto di vista dello sviluppatore di una ipotetica classe `Base`, non comporta alcuna differenza: la classe `Base` non dovrà preoccuparsi se qualcuno, in futuro, estenderà mai la classe. Dal punto di vista dello sviluppatore di `Derived`, invece, è necessario conoscere la classe `Base`. Non è tuttavia necessario conoscere il codice sorgente, i field ed i metodi `private`, o come sono implementate le funzionalità – in generale, sarà soltanto necessario conoscere i metodi ed i campi pubblici. Trattasi di

un esempio di modularità, detto *separation of concerns*. In Java, infine, **non esiste l'ereditarietà multipla**: si può estendere una sola classe.

Un problema non banale è invece l'invocazione del costruttore: chi inizializza i campi della classe da derivare? Ad esempio,

```
public class Derived extends Base {
  public Derived() {
    /* init things */
  }
}
```

con quale sistema viene inizializzato lo stato dell'istanza Derived, dal momento in cui esso contiene anche le funzionalità di Base? Le funzionalità di Base adoperate da Derived, infatti, assumono che la classe Base sia stata inizializzata. Come è possibile dunque inizializzare la porzione ereditata? Il meccanismo è quello di una *invocazione implicita del costruttore di default*, aggiunto implicitamente qualora non presente nel codice. Il compilatore, in assenza di un metodo di costruzione esplicito, assume che la prima azione da intraprendere (il primo statement del costruttore) sia evocare il costruttore nullo per la classe Base.

Cosa accade però se nella classe Base non è presente il costruttore senza argomenti (esso potrebbe non esistere)? In tal caso, il costrutture di default non può essere invocato, e *il codice non compila*: lo sviluppatore dovrà provvedere a fornire un costruttore adeguato per Base all'interno del costruttore di Derived.

La sintassi è tramite la parola chiave **super**:

```
public class Greeter {
public Greeter(String name) {
  /* init things */
  }
}

  public class EnhancedGreeter extends Greeter {
    public EnhancedGreeter(String firstName, String lastName) {
      super(firstName + lastName);
      /* init things */
    }
  }
```

Il costruttore di default in questo caso non esiste (poiché è stato definito un costruttore con argomento String): la parola chiave **super** consente di evocare tale costruttore con argomento definito dentro **super**, della classe che estende. In altre parole, **super** invoca il costruttore della *superclasse*.

Lo snippet

```
  public class Base {
    public Base(int n) { /* ... */ }
    public Base(double v) { /* ... */ }
  }

  public class Derived extends Base {
  private int m;
  public Derived() {
    m = 0;
    super(5);
    /* ... */
  }
}
```

non compila perché il costruttore **`super` deve** essere la prima cosa ad essere eseguita[1]. Lo snippet seguente, allo stesso modo,

```
  public class Base {
  public Base(int n) { /* ... */ }
  public Base(double v) { /* ... */ }
}

public class Derived extends Base {
  private int m;
  public Derived() {
    m = 0;
    /* ... */
  }
}
```

non compila poiché non non esiste un costruttore senza argomenti. Diversamente,

```
  public class Base {
    public Base(int n) { /* ... */ }
    public Base(double v) { /* ... */ }
    public Base() { /* ... */ }
  }

  public class Derived extends Base {
  private int m;
  public Derived() {
    m = 0;
    /* ... */
  }
}
```

[1]Base deve essere inizializzato prima di qualsiasi altra operazione, altrimenti non si potrebbe sapere lo stato dell'oggetto parent.

compila poiché il costruttore di default è invece definito.

Se vi sono le inline initializations,

```
  public static class Base {
    public int n = 1;
  }
/////
public static class Derived extends Base {
  public int m = n + 1;
}
/////
Derived derived = new Derived();
System.out.printf("m=%d%n", derived.m); // -> m=2
```

questo frammento funziona, infatti l'inizializzazione di qualsiasi campo avviene **dopo** l'invocazione del costruttore super. In questo caso, la dot notation **`derived.n`** è valida, e verrebbe valutata ad 1.

6.1 Riassunto del capitolo

Ereditarietà Proprietà statica del linguaggio in cui è possibile definire un nuovo tipo a partire da un tipo esistente in precedenza. Si dirà che il nuovo tipo **estende** il vecchio tipo.

Sintassi dell'ereditarietà La sintassi è aggiungere il termine **`extends`** nella definizione della classe.

Effetto dell'ereditarietà L'effetto dell'ereditarietà è che il nuovo tipo **eredita** tutti i field e metodi della classe che estende. In parole povere, il nuovo tipo sarà dotato degli stessi metodi e field. Può, opzionalmente,

ridefinirli **a patto che la signature sia la medesima**. I field o metodi pubblici non possono essere nascosti.

Design dell'ereditarietà Dal punto di vista del software design, l'ereditarietà è un modo per estendere di funzionalità (o modificarne alcune) un oggetto già esistente. In particolare, sebbene lo sviluppatore dell'oggetto non si interessi a tutte le possibili estensioni, un secondo sviluppatore può preoccuparsi di una particolare estensione: vi è dunque una separazione dello sforzo di progettazione.

Costruttore di una classe che estende Il costruttore di una classe `Derived` che estende `Base` chiamerà implicitamente il costruttore della classe `Base`, anche qualora non effettuato esplicitamente. La sintassi per una chiamata esplicita è tramite la parola chiave `super`, la quale ci consente di chiamare il costruttore con determinati argomenti della superclasse (nella gerarchia dell'ereditarietà). Qualora non indicato, il compilatore assume che sarà chiamato il costruttore nullo – se non presente, non compila. La chiamata a `super` deve essere **la prima azione** intrapresa all'interno del costruttore della subclasse. Adoperando le inline initializations, esse avvengono comunque dopo l'invocazione del costruttore della superclasse, dunque il codice compila.

La classe `Object` La classe `Object` è la particolare classe dalla quale **ereditano tutte le altre classi**, ed è dunque collocata alla radice dell'albero dell'ereditarietà. Anche se non presente nella signature della classe, la classe automaticamente erediterà da `Object`.

Capitolo 7

Il Polimorfismo

Supponiamo di avere definito

```
public class Derived extends Base { /* ... */ }
```

in questo caso, qualsiasi codice scritto per Base può funzionare anche con Derived: in particolare, un riferimento al tipo Base può riferire ad un oggetto di tipo Derived. In altre parole,

```
public void use(Base base) { /* ... */ }

Base base = new Derived(); // OK!!!

use(new Base()); // OK, sure!
use(new Derived()); // OK!!!
```

Cambiando solo la signature, tutte le dot notations effettuate sul **Base** valgono anche per la classe **Derived**, possiamo scambiare **Base** con **Derived** e il compilatore non si lamenterà. Il compilatore lo ritiene legittimo, la seconda istruzione del precedente frammento di codice compilerebbe ed eseguirebbe senza problemi. Il metodo **use()** compila poiché è stato scritto per funzionare con una **Base** – ciascun metodo o field valido nel **Base** è anche valido per il **Derived**, anche se vi sono diversi comportamenti dei metodi.

Filosoficamente, l'ereditarietà ci permette di instaurare relazioni fra classi, "filosoficamente", con una specie di *sillogismo*

```
public class LivingBeing { /* ... */ } // is an Object

public class Animal extends LivingBeing { /* ... */ }

public class Mammal extends Animal { /* ... */ }

public class Dog extends Mammal { /* ... */ }

public class Chihuahua extends Dog { /* ... */ }
```

Grazie all'ereditarietà possiamo definire *gerarchie concettuali*. È oggetto di disputa[1] il fatto che l'ereditarietà sia la maniera corretta per modellare tale relazione fra oggetti, poiché vi sono altri modi per modellare questa relazione fra concetti: uno dei motivi è perché non esiste l'ereditarietà multipla.

[1] https://qr.ae/pNnOEU

In generale, bisogna sempre tendere ad adoperare il livello più astratto di qualche entità, piuttosto che quello più specifico:

```
// better
public void putLeash(Dog dog) { /* ... */ }

// not goot
public void putLeash(Chihuahua dog) { /* ... */ }
```

Similmente,

```
  // best
  public Milk milk(Mammal mammal) { /* ... */ }

  // not so good
  public Milk milk(Cow cow) { /* ... */ }

  // wrong!
  public Milk milk(Animal animal) { /* ... */ }
```

Una classe che estende non può rimuovere metodi della classe che estende. In particolare, ciò renderebbe falso l'affermazione chiave dell'ereditarietà "Ogni Derived è un Base". Lo stesso deve valere anche per la visibilità dei metodi di Base, che *non può essere ridotta*. La riduzione della visibilità, dunque, **non è ammissibile in Java**.

Metodi omonimi, tuttavia, possono essere ridefiniti per buone ragioni. In particolare,

```
  public void greet(Person person) {
    System.out.println("Hi, " + person.getFullName());
  }
```

```
public class Person {
  /* ... */
  public String getFullName() {
    return firstName + " " + lastName;
  };
}

public class Doc extends Person {
  /* ... */
  public String getFullName() {
    return "Dr. " + firstName + " " + lastName;
  };
}
```

In questo caso siamo di fronte alla proprietà del **polimorfismo**: la conseguenza *dinamica* (al runtime) dell'ereditarietà. Il medesimo codice può avere *comportamenti diversi* al runtime a seconda del tipo effettivo dell'oggetto che viene manipolato al momento dell'esecuzione. Naturalmente, lo sviluppatore della classe `Derived` **deve decidere** se tale classe sia effettivamente derivata da `Base`, oppure no: in altre parole, è compito dello sviluppatore assicurarsi che i metodi chiamati rispettino le necessità, di modo che possano funzionare anche sul metodo non generale con una certa, perlomeno, somiglianza con il metodo definito in `Base`.

Il polimorfismo è conseguenza del fatto che una classe `Derived` ha le medesime proprietà (metodi, field) degli oggetti di tipo `Base` – in questa maniera, il compilatore consente allo sviluppatore di adoperare metodi costruiti specificatamente per la classe `Base` su oggetti di tipo `Derived`, mentre non vale il contrario, poiché la classe `Derived` possiede metodi e field peculiari, potenzialmente in aggiunta rispetto a quelli proposti dagli oggetti della classe `Base`.

Naturalmente, non vale il contrario:

```
public Base createBase() { /* ... */ }
Derived derived;
derived = createBase();
```

In questo caso, `derived` contiene le proprietà degli oggetti della classe `Derived`. Applicare il metodo `createBase` su `derived`, *non compila* perché al runtime **non è garantito che un oggetto `Base` contenga un metodo invocato con dot notation, della classe `Derived`**. Una situazione potrebbe essere la seguente,

```
public Base createBase() {
  if (System.currentTimeMillis() % 2 == 0) {
    return new Base();
  }
  return new Derived();
}
/////
Derived derived;
derived = createBase();
```

è un metodo maledettamente diabolico.

Esiste una sintassi per forzare la compilazione, detta **downcasting** (rispetto all'albero dell'ereditarietà):

```
Derived derived;
derived = (Derived) createBase();
```

dove si inserisce fra parentesi[2] l'identificatore del tipo di cui si intende effettuare il cast. Il compilatore lo lascia inalterato, però al runtime potrebbe comunque non essere un Derived (in tal caso, potremmo incappare in errori). Tale sintassi,

- "cambia" il tipo di riferimento downcast;
- funziona soltanto se Derived deriva direttamente o indirettamente da Base – questo poiché `createBase()` **deve** tornare un `Base`, dunque non è possibile accettare qualcosa che non può accadere;
- funziona solo se i tipi sono in relazione, cioè se ogni tipo in return è alto in gerarchia quanto il tipo dichiarato, altrimenti non compila.

Oltre alle verifiche in atto di compilazione, al runtime vi sono verifiche ad ogni invocazione.

Le ragioni per l'uso del downcasting potrebbero essere quelle in cui un tipo particolare è resituito, ma dipende *a seconda di un parametro*.

7.0.1 La sintassi `instanceof`

La sintassi `instanceof` serve per controllare il tipo di un oggetto al runtime. In altre parole, a sinistra abbiamo un tipo `boolean`, e alla destra vi è un predicato (riferimento - oggetto). Nell'esempio,

```
boolean isString = ref instanceof String;
```

`isString` è al valore `true` se `ref` è un riferimento ad un oggetto di tipo `String`, mentre sarà di valore `false` altrimenti.

[2] La sintassi stessa, dove il tipo dell'oggetto viene specificato fra parentesi, indica la precarietà intrinseca dell'operazione.

Formalmente, `a instanceof B` è `true` se e solo se l'oggetto riferenziato da `a` può essere legittimamente referenziato da un riferimento `b` ad oggetti di tipo `B`. In altre parole, l'esempio è esplicativo:

```
public class Derived extends Base { /* ... */ }
Derived derived = new Derived();
boolean b1 = derived instanceof Object; // -> true
boolean b2 = derived instanceof Base; // -> true
boolean b3 = derived instanceof Derived; // -> true
```

Il compilatore, in certi casi di falsità tautologica, si rifiuta di compilare. Ad esempio,

```
Base b = new Base();
... = b instanceof String; // sempre falso, non compila!
```

L'uso tipico di `instanceof` potrebbe essere

```
Object obj = someOddMethod();
if (obj instanceof String) {
  String string = (String) obj;
  System.out.println(string.toUpperCase());
  // obj.toUpperCase() would not compile!
}
/////
// Pattern matching
Object obj = someOddMethod();
if (obj instanceof String string) {
  System.out.println(string.toUpperCase());
}
```

con la possibilità di discernere fra vari casi e tipologie di oggetto. La scorciatoia è elegante, e prende il nome di **Pattern matching**.

7.0.2 Il caso dell'esempio della classe `Date` e il metodo `toString`

Tornando al frammento di inizio capitolo,

```
import java.util.Date;
public class Greeter {
  public static void main(String[] args) {
    System.out.print("Hello World! Today is: ");
    System.out.println(new Date());
  }
}
```

Esiste infatti un metodo println tale che

> `void println(Object x)` Prints an Object and then terminate the line.

stampa qualsiasi oggetto. La classe `Object` ha un metodo di nome `toString`. Tale metodo *restituisce una rappresentazione in Stringa dell'oggetto*[3]. In questo caso, la documentazione è fornita a due tipi di sviluppatori: l'utilizzatore, e il designer che andrà a ridefinire il metodo. È opportuno che tutte le classi che estendono `Object` *ridefiniscano il metodo*. È anche specificato il *contratto* che deve essere rispettato nel ridefinire tale metodo. Inoltre, vengono specificate anche le istruzioni per gli utilizzatori. In particolare, vengono restituiti nome dell'oggetto e un suo *hash*. In via definitiva, *ogni oggetto ha la capacità di essere rappresentato come una stringa*.

Tipicamente, si desidera fare l'*overriding* del metodo `toString`. Le linee guida sono le seguenti:

- restituisce una stringa che *rappresenta testualmente* l'oggetto;

[3] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Object.html#toString()

- conciso ma informativo;
- sia di facile lettura per una persona umana.

Ad esempio, il metodo `toString` per la classe Date,

> `public String toString()`
>
> Converts this Date object to a String of the form:
>
> dow mon dd hh:mm:ss zzz yyyy
>
> where:
>
> dow is the day of the week (Sun, Mon, Tue, Wed, Thu, Fri, Sat).
>
> mon is the month (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec).
>
> dd is the day of the month (01 through 31), as two decimal digits.
>
> hh is the hour of the day (00 through 23), as two decimal digits.
>
> mm is the minute within the hour (00 through 59), as two decimal digits.
>
> ss is the second within the minute (00 through 61), as two decimal digits.
>
> zzz is the time zone (and may reflect daylight saving time). Standard time zone abbreviations include those recognized by the method parse. If time zone information is not available, then zzz is empty - that is, it consists of no characters at all.
>
> yyyy is the year, as four decimal digits.

In definitiva, il metodo `println` per `Object` fa una cosa simile:

```
public void println(Object x) {
  if ( x == null ) {
    println("null");
  } else {
    println(x.toString()); // esiste sempre!
  }
}
```

dove, in realtà, vi è un information hiding dove un println verifica che il riferimento non riferisca a `null`, e poi *delega* al secondo println la stampa, e in via definitiva, a tutti gli altri sviluppatori di definire la rappresentazione in stringa di un oggetto.

Anche la concatenazione funziona. Infatti, il compilatore adopera implicitamente il metodo `toString` qualora servisse una stringa dell'oggetto. Ad esempio, in

```
System.out.println("Now is " + new Date());
```

avviene una sorta di conversione implicita, del tipo

```
(ref == null) ? "null" : ref.toString()
```

In parole povere, quando necessario e quando richiesto, ciascun oggetto viene convertito nella sua rappresentazione a stringa, mediante il metodo `toString`.

7.1 Il funzionamento del polimorfismo

La Java Virtual Machine è in grado di effettuare il polimorfismo. Al runtime, ogni oggetto `o` non primitivo ha un *riferimento ad un field che è la descrizione del suo tipo*. Esiste un tipo (una classe) di nome `Class` – ogni oggetto ha un field di tipo `Class`[4], ereditato dalla classe `Object`. Inoltre, la descrizione del tipo è la medesima per tutte le istanze della medesima classe di `o`. In altre parole, il field `Class` è `static` con modificatore d'accesso `private`, e può essere ottenuto mediante un metodo della classe `Object`, `getClass`. Nella classe `Object`, vi sarà un field del tipo

[4] Trattasi dell'unico caso dove un field ha il proprio nome di riferimento indicato con la iniziale maiuscola.

```java
public class Object {
  private static Class<?> clazz; // set by JVM, probably wrote in
      bytecode directly
  public Class<?> getClass() {
    return clazz;
  }

  // metodo toString...

  // ...
}
```

Il metodo **`getClass`** restituisce il **tipo al runtime** della classe **`Object`**.

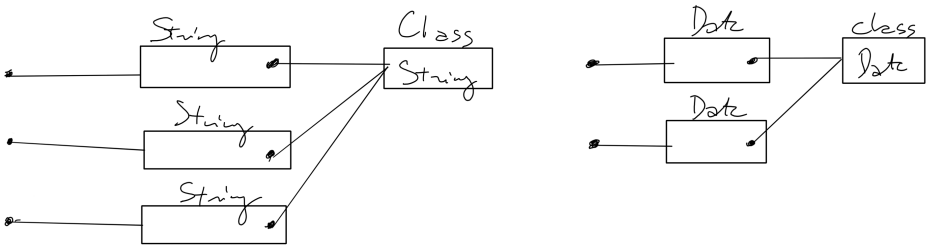
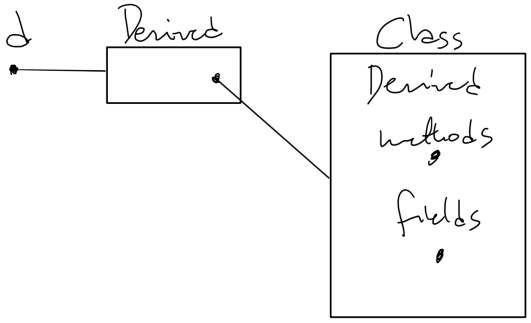


Figura 7.1: Presenza del field di tipo **`Class`** all'interno degli oggetti di tipo **`String`** e di tipo **`Date`**.



Concettualmente, all'interno di un'istanza di `X`, c'è un field di tipo `Class`[5] dove risiede l'informazione relativa a tutti i metodi ed i field **dichiarati**, privati o pubblici che siano. All'interno dell'oggetto `Class` vi sono concettualmente[6] dei riferimenti ad array,

- il riferimento `method` ad un array che contiene tutti e soli i metodi dichiarati in `X`;
- il riferimento `field` ad un array che contiene tutti e soli i field dichiarati in `X`.

La classe `Derived` conterrà tutti i metodi della classe `Base` – il viceversa non varrà.

Un ulteriore field contenuto in oggetti di tipo `Class` è il field `superclass`.

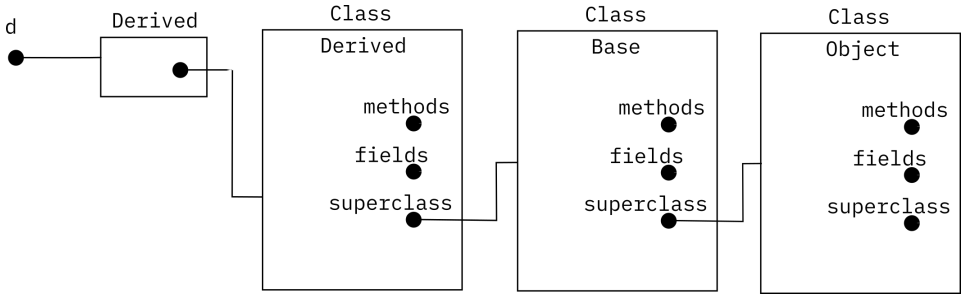


Figura 7.2: Riferimento alle superclassi nel caso del metodo `getClass()`.

Supponendo di avere un oggetto di tipo `Derived`. In questo caso,

- `methods` di `d.getClass()` contiene `doDerived`;
- `methods` di `d.getClass().superclass` contiene `doBase`;
- `methods` di `d.getClass().superclass.superclass` contiene `toString`, `getClass`;

Questo tipo di ispezione degli oggetti è detto **reflection**.

7.1.1 Esempio di polimorfismo: il metodo `println`

Ritornando al frammento che stampa la rappresentazione testuale di un oggetto,

```
  System.out.println(x); // -> derived.toString()

public void println(Object x) {
  if ( x == null ) {
    println("null");
  } else {
    println(x.toString());
  }
}
```

vengono effettuati numerosi passaggi concettuali. In particolare,

1. Java ottiene la `Class c` dell'oggetto `x`;
2. successivamente, ottiene il field `methods` di `c`;
3. se `methods` contiene `toString()`, utilizza quel metodo – altrimenti, imposta `c` to `c.superclass`, e torna al punto 2. La medesima verifica viene fatta per le superclassi.

Nel caso peggiore, l'unico metodo ad ottenere una rappresentazione testuale è la radice della gerarchia dell'albero dell'ereditarietà, la classe `Object`.

[5] Attenzione a non far confusione: `Class` e `Object` sono classi. Non si può adoperare `class` per identificatori di nessun tipo, poiché è una parola riservata (una buona alternativa è `clazz`). È possibile, d'altronde, adoperare l'identificatore `object` perché non è riservato dal linguaggio Java.

[6] Osservando nel dettaglio la classe `Class`, potrebbe esserci un maggiore dettaglio. Pertanto, l'esistenza dei due field è soltanto un'*approssimazione* del funzionamento interno di tale classe.

7.1.2 I metodi della classe `Object`

Abbiamo almeno 3 metodi d'interesse,

- `Class<?> getClass()`: returns the runtime class of this Object;
- `String toString()`: returns a string representation of the object;
- `equals(Object obj)`: indicates whether some other object is **"equal to"** this one.

Il metodo `equals` è differente dall'adoperare l'operatore `==` – infatti, l'operatore binario `a == b` ha la seguente descrizione del funzionamento:

- `a` e `b` sono tipi primitivi, ma sono di tipi diversi, oppure primitivi e non primitivi, oppure ancora di tipi non primitivi non compatibili[7]; allora *il codice non compila*;
- `a` e `b` sono dello stesso tipo primitivo; allora viene valutato in `true` se e solo se `a` e `b` hanno lo stesso valore, altrimenti viene valutato a `false`. Esistono casi specifici dove l'operatore binario viene valutato correttamente anche per tipi non primitivi con tipi primitivi;
- altrimenti, viene valutato `true` se `a` e `b` **fanno riferimento allo stesso oggetto** o entrambi non riferiscono ad alcun oggetto – dunque se sono `null` – altrimenti viene valutato a `false`.

```
String s1 = "hi!";
String s2 = "hi!";
String s3 = s2;
boolean b;
b = s1 == s2; // -> false
b = s2 == s3; // -> true
b = s1.length() == s2.length(); // -> true
```

[7] Non compatibili significa che non c'è modo che i due riferimenti possano far riferimento al tipo dell'altro – in altre parole, sono in diversi rami della gerarchia.

Il metodo `equals()` consente di valutare qualora due oggetti sono "uguali". Il suo funzionamento dipende se sia stato definito in un altro metodo, e dunque *dipende dal runtime*. Alcuni esempi,

- String: "true if and only if [...] represents the same sequence of characters [...]"
- Date: "true if and only if [...] represents the same point in time, to the millisecond [...]"
- PrintStream: does not override equals(): the developers of PrintStream decided that the "equal to" notion of Object hold for PrintStreams

Con la classe `Object` viene poi fornita la documentazione, fornita con il doppio intento per l'utilizzatore e per lo sviluppatore. Dalla documentazione,

> Indicates whether some other object is "equal to" this one.
>
> The `equals` method implements an equivalence relation on non-null object references:
>
> It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
>
> It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
>
> It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
>
> It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in equals comparisons on the objects is modified.
>
> For any non-null reference value `x`, `x.equals(null)` should return `false`.
>
> The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Dunque, il metodo *equals* permette di modellare una relazione di equivalenza fra oggetti. Esso andrebbe riscritto dallo sviluppatore, per ogni classe che ne deve fare uso. L'implementazione deve soddisfare determinate proprietà, enunciate sopra nella citazione alla documentazione – nella classe `Object`, tali linee guida sono implementate facendo sì che due oggetti risultino uguali se lo sono nel senso dell'operatore binario `==`, cioé se due riferimenti fanno riferimento allo stesso oggetto. Solitamente, il metodo `equals` viene riscritto soltanto per classi aventi field; in tal caso, infatti, i field permettono di stabilire le relazioni di equivalenza a seconda dei casi.

In altre parole, nella classe `Object` si avrà qualcosa del tipo

```
public class Object {
  /* ... */
  public boolean equals(Object other) {
    if (other == null) {
      return false;
    }
    return this == other;
  }
}
```

dove `this` non può mai essere `null`. I requisiti per l'override del metodo sono i seguenti:

- mantenere la medesima signature,

  ```
  public boolean equals(Object other)
  ```

  caso gestito dal compilatore;

- implementare il concetto di **relazione d'equivalenza** per il caso specifico – non gestito dal compilatore;

- proprietà di *riflessività*, *simmetria*, *transitività*, *consistenza*, gestione del caso `null` – non gestito dal compilatore.

L'implementazione del concetto di relazione d'equivalenza dipende largamente dal contesto. Ad esempio, il concetto stesso di *persona* potrebbe variare a seconda del luogo: in un Ospedale, a livello scientifico, a livello di numero di matricola, e così via, con i casi più disparati che dipendono da *come il mondo reale è modellato*. Una possibile modellazione è la seguente,

```
public class Person {
  private String firstName;
  private String lastName;
  private Date birthDate;
}
```

dove ogni persona è univocamente modellata da 3 campi. Allora il metodo `equals` potrebbe semplicemente comparare i campi all'interno della classe. Tuttavia, questo potrebbe non andare bene a seconda dei contesti, pensiamo al caso ad esempio di una persona che cambi nome, oppure di due persone aventi stesso nome, cognome, e medesima data di nascita per pura coincidenza. La realtà viene modellata dallo sviluppatore: è dunque sua sola responsabilità la scelta di come viene poi implementata la relazione di equivalenza, a seconda del contesto e delle necessità.

Una modellazione differente è la seguente, dove viene aggiunto un campo contenente il codice fiscale,

```
public class Person {
  private String firstName;
  private String lastName;
  private Date birthDate;
  private String fiscalCode;
}
```

In tal caso, il metodo `equals` potrebbe semplicemente richiedere che due persone abbiano lo stesso codice fiscale per essere considerate la stessa persona. Tutto ciò viene introdotto dallo sviluppatore, tramite la **conoscenza di dominio**.

Supponiamo ora di implementare la prima versione,

```
public class Person {
  private String firstName;
  private String lastName;
  private Date birthDate;
  public boolean equals(Object other) {
    if (other == null) {
      return false;
    }
    if (!(other instanceof Person)) {
      return false;
    }
    if (!firstName.equals(((Person) other).firstName)) {
      return false;
    }
    if (!lastName.equals(((Person) other).lastName)) {
      return false;
    }
    if (!birthDate.equals(((Person) other).birthDate)) {
      return false;
    }
    return true;
  }
}
```

Si osservi che sono stati effettuati dei *downcast* per l'oggetto `other`, di modo da farlo funzionare anche per oggetti di tipo più basso in gerarchia.

Implementando la seconda possibilità, si ottiene

```
public class Person {
  private String firstName;
  private String lastName;
  private Date birthDate;
  private String fiscalCode;
  public boolean equals(Object other) {
    if (other == null) {
      return false;
    }
    if (!(other instanceof Person)) {
      return false;
    }
    if (!fiscalCode.equals(((Person) other).fiscalCode)) {
      return false;
    }
    return true;
  }
}
```

7.1.3 La classe `Objects`

`Objects` è una classe **di utilità**[8], static per scelta. Essa fornisce il metodo `equals`. Dalla documentazione,

> This class consists of static utility methods for operating on objects, or checking certain conditions before operation.

[8] Trattasi di una classe che modella un componente capace di fare operazioni, ma non dotato di uno stato. Si differiscono dal nome (con una **s**) e dalla presenza di metodi statici. Tipicamente, il costruttore non è public.

`static int checkIndex(int index, int length)` Checks if the index is within the bounds of the range from 0 (inclusive) to length (exclusive).

`static boolean deepEquals(Object a, Object b)` Returns true if the arguments are deeply equal to each other and false otherwise.

`static boolean equals(Object a, Object b)` Returns true if the arguments are equal to each other and false otherwise.

`static boolean isNull(Object obj)` Returns true if the provided reference is null otherwise returns false.

Il metodo `deepEquals` è simile ad `equals`, ma funziona anche con gli array, facendo il confronto elemento per elemento nel vettore. Un altro esempio di classi di utilità forniti all'interno del JDK è la classe `Arrays`, che fornisce `sort()`, `fill()`, e così via.

7.1.4 Differenze fra `instanceof` e `getClass()`

In apparenza, si somigliano. Tuttavia,

- `a instanceof B` vero se e solo se `a` potrebbe essere legittimamente referenziato da un riferimento `b` ad oggetti di tipo `b`;
- `a.getClass() == b.getClass()` vero se e solo se al runtime la classe di oggetti referenziati da `a` è esattamente la stessa (`==`) della classe di oggetti referenziata da `b`;
- `a.getClass() == B.class` identica alla seconda – `class` è approssimativamente un field statico definito implicitamente in ogni classe, che fa riferimento all'oggetto che rappresenta la classe (in questo caso, la classe `B`). Si usa qualora volessimo confrontare una classe con un oggetto, o viceversa.

Supponiamo di avere lo snippet seguente, con diagramma oggetti-riferimenti mostrato in Figura 7.3.

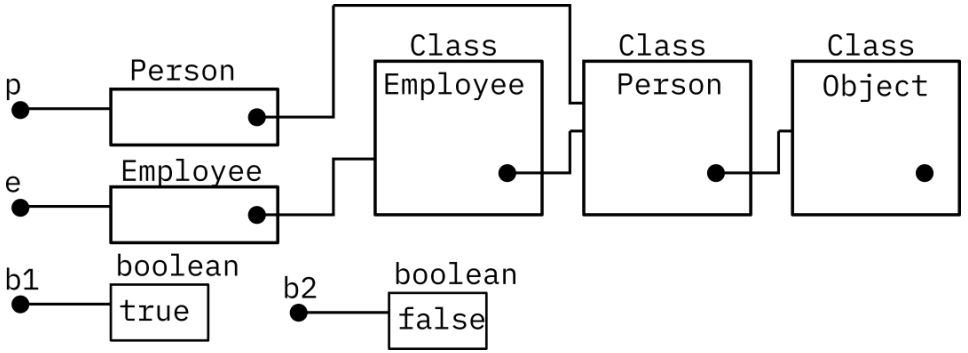


Figura 7.3: Situazione in memoria con l'utilizzo di `instanceof` e `getClass()`.

```
public class Person { /* ... */ }

public class Employee extends Person { /* ... */ }

Person p = new Person( /* ... */ );
Employee e = new Employee( /* ... */ );
boolean b1 = e instanceof Person; // -> true
boolean b2 = p.getClass() == e.getClass(); // -> false
```

La scelta dei due dipende esclusivamente dalla modellazione adoperata. Supponiamo di voler paragonare l'impiegato alla persona – in questo caso, si potrebbe decidere che per una tale eguaglianza che andiamo a modellare l'uguaglianza sia verificata oppure no.

Si vuole modellare i *multiset* e le *sequenze*. Matematicamente,

- un *multiset* o *bag* è un insieme di elementi in cui ogni elemento può apparire da 0 a più volte. Precisamente,

$$\mathscr{X} = \mathscr{M}(\mathbb{Z}) = \{f : \mathbb{Z} \to \mathbb{N}\}.$$

L'equivalenza di x_1 ed x_2 considera sia la lunghezza (stesso numero di elementi) che gli elementi (ogni elemento compare lo stesso numero di volte). Essa potrebbe essere modellata da

```
public class BagOfInts {
  private int[] values;
}
```

- un *insieme delle sequenze* di interi è un

$$\mathscr{X} = \bigcup_{i \in \mathbb{N}} \mathbb{Z}^i.$$

La relazione di equivalenza stavolta considera, per due elementi x_1, x_2, anche l'*ordine* degli elementi presenti. In questo caso, potrebbe essere modellata con

```
public class SequenceOfInts {
  private int[] values;
}
```

Osserviamo che la modellazione, almeno in questo caso, è identica dal punto di vista del compilatore, poiché il codice è diverso – tuttavia, il resto dei dettagli sarà diverso (a partire dal nome). La differenza fra i due modelli dovrebbe essere catturata dal **nome** del tipo.

7.2 Ereditarietà multipla in Java?

Si prenda in considerazione l'idea di voler modellare l'universo dei veicoli con il linguaggio Java. Si potrebbe voler effettuare una modellazione del tipo seguente, basandoci sostanzialmente sulle caratteristiche dei veicoli (hanno un motore, hanno ruote, sono veicoli nautici, e così via):

```
public class Vehicle { /* ... */ }

public class MotorVehicle extends Vehicle {
  public void startEngine() { /* ... */ }
}

public class Motorboat extends MotorVehicle {
  public void startEngine() { System.out.println("bl bl bl"); }
  public void turnPropeller() { /* ... */ }
}

public class Car extends MotorVehicle {
  public void startEngine() { System.out.println("brum brum"); }
  public void turnWheels() { /* ... */ }
}
```

Capita però che ci siano casi in cui la relazione fra le entità non è ad albero, e parrebbe avere un senso realizzare una qualche sorta di *ereditarietà multipla,* del tipo *AmphibiousVehicle* che estenderebbe sia le *Motorboat* che le *Car*:

```
public class AmphibiousVehicle extends Motorboat, Car { // NOOOO!
  public void doAmphibiousFancyThings() { /* ... */ }
}
```

ciononostante, questo codice **non compila**, poiché l'ereditarietà multipla in Java non è implementata. Come effettuiamo dunque la modellazione di tale entità?

Supponiamo infatti di avere un veicolo anfibio istanziato come nel seguente frammento,

```
public class AmphibiousVehicle extends Motorboat, Car { // NOOOO!
```

```
  public void doAmphibiousFancyThings() { /* ... */ }
}

AmphibiousVehicle fiat6640 = new AmphibiousVehicle();
fiat6640.startEngine();
```

all'esecuzione di `startEngine()` che cosa succede? Quale dei due metodi viene eseguito, quello ridefinito nella classe `Motorboat` o quello ridefinito nella classe `Car`? La sintassi Java non consente di specificare allo sviluppatore quale metodo adoperare – una scelta precisa degli sviluppatori Java, allo scopo di non caricare lo sviluppatore di troppe responsabilità e, di fatto, sacrificando l'espressività.

Supponiamo ancora che gli sviluppatori di Java abbiano consentito l'ereditarietà **solo** nel caso in cui le classi padri sono **disgiunte**, come nel frammento seguente

```
public class A { public void doA() { /*...*/ } }

public class B { public void doB() { /*...*/ } }

public class C extends A, B { public void doC() { /*...*/ } }
```

Anche in questo caso vi sono problemi – cosa accadrebbe infatti se gli sviluppatori della classe `A` e della classe `B` decidessero di aggiungere entrambi la ridefinizione del metodo `toString()`? In tal caso la condizione non sarebbe più verificata.

7.2.1 L'espressione `switch`

L'espressione `switch` viene introdotta tramite il seguente esempio,

```
public static RegularPolygon createPolygon(
    int nOfSides,
    double sideLength
) {
  return switch (nOfSides) {
    case 3 -> new Triangle(sideLength);
    case 4 -> new Square(sideLength);
    case 5 -> new Pentagon(sideLength);
    default -> null;
  };
}
```

L'espressione `switch` viene considerata valida dal compilatore solamente nel caso in cui **l'intero spazio delle possibilità è gestito dal costrutto**. Questo equivale a specificare ogni singola possibilità del valore nell'argomento dello switch - poiché è impossibile coprire ogni caso personalmente, si adopera la keyword `default`, che consente di "catturare" tutti i casi possibili. Il costrutto `switch` si sposa bene nel caso vi siano pochi valori d'interesse, con range piccoli di valori da specificare.

Un costrutto `switch` non ammette casi `boolean`, poiché eccessivamente banali.

7.3 Riassunto del capitolo

Legame fra classe estesa e classe che estende Esiste un legame fra la classe estesa e la classe derivata – in particolare, tutto il codice che funziona per la classe estesa funzionerà anche per la classe derivata: infatti, la classe derivata avrà **almeno** i metodi della classe che estende. Perciò, si possono assegnare tipi differenti purché il tipo assegnato abbia i metodi della classe base (cioè, sia dello stesso tipo o derivi da essa).

Dunque, mediante l'ereditarietà siamo in grado di riutilizzare codice ma aggiungendo nuove funzionalità, definendo gerarchie concettuali.

Riduzione della visibilità Non ammessa in Java.

Polimorfismo Il **polimorfismo** è una proprietà dinamica del linguaggio per cui viene realizzata l'ereditarietà, con il codice che esibisce comportamenti differenti a seconda del tipo effettivo dell'oggetto che viene manipolato al momento dell'esecuzione.

Conseguenze del polimorfismo al runtime, uno stesso metodo può essere eseguito con la classe base o con la classe derivata, a seconda del tipo – è una conseguenza dinamica del polimorfismo. Pertanto è possibile che un oggetto base sia inizializzato con un costruttore di classe derivata – in tale circostanza, verranno eseguiti metodi della classe derivata. Non è invece possibile il contrario, poiché potrebbero mancare metodi alla classe base.

Downcasting Poiché è impossibile far sì che una classe derivata sia inizializzata a partire da un costruttore della classe base, la sintassi fra parentesi tonde detta **downcasting** è l'unico modo per forzare la compilazione. La sintassi cambia il tipo di riferimento downcast, funziona soltanto se la classe derivata deriva direttamente o indirettamente da quella base. Tuttavia, dinamicamente, la JVM provvederà a verificare l'esistenza dei metodi: qualora non fossero garantiti, non funzionerebbe e si bloccherebbe tutto.

`instanceof` Operatore binario per controllare il tipo di un oggetto al runtime. Valuta come un `boolean` – `a instanceof b` restituisce `true` se e solo se l'oggetto a cui fa riferimento `a` può essere legittimamente riferito dal riferimento `b` ad oggetti di classe `B`. `instanceof` realizza il cosiddetto **Pattern matching**.

`toString()` Metodo che restituisce la descrizione testuale di un oggetto. Genericamente esso appartiene ad `Object`; dunque, ciascun oggetto di qualsivoglia classe ne sarà dotato.

Il field `Class` Field ereditato dalla classe `Object`, presente in qualunque classe, indicato con la lettera maiuscola, con modificatori `static` e `private`. Per accedervi si utilizza il metodo `getClass()`, anch'esso della classe `Object`. Tale metodo restituisce, al runtime (polimorfismo), il **tipo** della classe dell'oggetto. Tutti gli oggetti di una determinata classe avranno un riferimento al singolo oggetto della classe `Class`.

L'oggetto `Class` L'oggetto `Class` contiene le informazioni relative a tutti i metodi e field **dichiarati**, privati o pubblici che siano – sotto forma di array `method` ed array `field`. Un oggetto di classe `Class` "Derived" conterrà solo ed esclusivamente i metodi dichiarati in Derived, non i metodi dichiarati in Base – l'oggetto relativo di classe `Class` "Base" conterrà a sua volta tutti e soli i metodi dichiarati in Base (lo stesso vale, naturalmente, per i field). Esiste anche un riferimento `superclass`, che contiene il riferimento all'oggetto di classe `Class` che è superclasse della classe che si sta considerando. Concettualmente e un po' più nella pratica, è così che si verificano i controlli dell'ereditarietà e del polimorfismo.

Passaggio alla superclasse Si verifica qualora il compilatore o la JVM non dovesse trovare nell'array "concettuale" `methods`; in tal caso, si imposta `c = c.superclass` e si ripete la verifica fino alla superclasse `Object`.

Il metodo `equals()` Metodo che indica se un oggetto è "equal to" ad un altro. La differenza con l'operatore binario di uguaglianza è che quest'ultimo,

- non compila se sono tipi primitivi diversi, non compatibili, o tipi primitivi non compatibili (non sullo stesso ramo della gerarchia), o tipo primitivo e non primitivo;

- per tipi primitivi, è `true` soltanto se i due oggetti di tipo primitivo hanno lo stesso valore; altrimenti, è `false`. Esistono sporadici casi in cui l'operatore binario viene valutato anche fra un tipo primitivo e un tipo non primitivo.
- per tipi non primitivi, è `true` se i due riferimenti fanno riferimento allo stesso oggetto, è `false` diversamente. Viene valutato `true` anche se ambedue riferiscono a `null`.

mentre invece il metodo `equals()` consente di definire, più precisamente, la maniera in cui due oggetti di tipo non primitivo debbano essere "uguali". Esso andrebbe ridefinito, dunque il suo funzionamento dipende dal runtime.

Proprietà del metodo `equals()` Il metodo `equals()` deve godere di determinate proprietà – **riflessività** (`x.equals(x)` restituisce `true` se `x != null`), **simmetria** (`x.equals(y) == y.equals(x)`), **transitività** (`x.equals(y) == y.equals(z) == true => x.equals(z) == true` e viceversa), **consistenza** (a meno di `null`, il risultato non deve dipendere dal tempo), **gestione del null** (`x.equals(null) == false` se `x != null`). Il metodo `equals()` della classe `Object` implementa `a == b` per tipi non primitivi.

La classe di utilità `Object` La classe di utilità (classe che modella un componente capace di fare operazioni, ma non dotato di uno stato – si differisce dal nome per via del plurale, tipicamente adoperato, e dalla presenza di metodi `static` e costruttore privato) `Objects` fornisce i metodi `isNull()` (restituisce `true` se è il riferimento all'oggetto fa riferimento a `null`) e due metodi per comparare oggetti `equals()` e `deepEquals()` – il secondo è simile al primo, tuttavia esso è in grado di verificare l'uguaglianza fra vettori di oggetti.

Differenza fra `instanceof` e `getClass()` La differenza, in breve, è che facendo `a.getClass() == b.getClass()` si ottiene `true` esclusivamente nel caso in cui le due classi siano **esattamente** le stesse.

Ereditarietà multipla L'ereditarietà multipla non è ammessa in Java.

Capitolo 8

Parametri, visibilità degli identificatori e memoria

8.1 Passaggio dei parametri

Nel seguente frammento di codice, non risulta subito evidente se i parametri sono passati **per valore** o **per riferimento**, in particolare non è ben chiaro se il parametro passato alla funzione subisca modifiche visibili dall'esterno oppure non ne subisca. Infatti, supponendo di modellare una persona che può cambiare nome,

```
// The class models person that can change its name
public class Person {
  private String name;
  public Person(String name) { this.name = name; }
  public String getName(){ return name; } //getter
  public void setName(String name){ this.name = name; } //setter
}

public void modify(Person p) {
  p.setName(p.getName().toUpperCase());
  //no return!
  //does p get modified?
}
```

```
Person eric = new Person("Eric");
modify(eric);
System.out.println(eric.getName());
```

La funzione `public void modify(Person p)` **non presenta valore di return**. Dunque, lo stato interno (il nome) viene modificato?

Il passaggio **by value** prevede che la funzione chiamata riceva **una copia** dell'oggetto — viceversa, il passaggio **by reference** prevede che la funzione chiamata riceva, invece, **l'oggetto stesso**, o meglio una **copia del riferimento** dell'oggetto. I tipi primitivi sono passati sempre *by value*, mentre i tipi non primitivi sono sempre passati *by reference*. Il tipo del passaggo, pertanto, è una fra le differenze cruciali che i tipi non primitivi presentano rispetto ai tipi primitivi.

Poiché `Person p` è un tipo non primitivo, viene passa una copia per riferimento. La modifica perciò avrà un effetto visibile anche successivamente. Per esempio,

```
public void modify(Person p) { // B
  p.setName(p.getName().toUpperCase());
}
```

```
Person eric = new Person("Eric"); // A
modify(eric); // C
```

La situazione in memoria dei tre casi è la seguente:

- nel punto del codice A esiste soltanto un riferimento all'oggetto di tipo `Person` di nome `eric`;
- nel punto B avremo che esiste un ulteriore riferimento, `p`, che riferisce anch'esso all'oggetto di tipo `Person`: una copia del riferimento `eric`. Nel contesto del metodo `modify`, non esiste il riferimento `eric`;
- nel punto C infine, il field interno del nome farà riferimento alla nuova stringa, di contenuto `"ERIC"`. La situazione finale è che il cambiamento è visibile, e l'oggetto di tipo `Person` vede il suo field fare riferimento ad una nuova stringa, non dunque più alla stringa con contenuto `"Eric"`.

Nel caso del passaggio di tipi primitivi, la situazione è diversa. Prendiamo ad esempio il frammento seguente,

```
public void uselesslyModify(int n) { // B
  n = 18;
}

int age = 42;       // A
uselesslyModify(age); // C
```

non si produrranno effetti visibili, poiché useless modify riceve soltanto *una copia* del tipo primitivo, non il riferimento all'oggetto `int`. Dunque, il tipo primitivo esterno viene in qualche maniera "protetto", poiché non viene passato direttamente, ma viene passata esclusivamente una sua copia; in questo caso, il metodo `uselesslyModify()` non produce un cambiamento.

8.2 La parola chiave `final`

Il modificatore `final` è una keyword applicabile a classi, metodi, field e variabili, di tipo **statico**, cioè che i suoi effetti sono prodotti *esclusivamente in tempo di compilazione* - al runtime è come se non esistesse; l'effetto al runtime è dunque esclusivamente implicito (se non compila, non esegue...). La parola chiave `final` ha un effetto diverso a seconda del tipo di definizione, distinguendo fra metodi e classi, field e variabili.

L'effetto su una **classe** è che essa **non può essere estesa**. Una classe dichiarata con modificatore `final` e successivamente estesa vedrà un errore in fase di compilazione.

L'effetto su un **metodo** è che esso **non può essere ridefinito**. Estendendo una classe non `final` con un metodo `final` e provando a ridefinire tale metodo, viene prodotto un errore in fase di compilazione.

Con il modificatore `final` su classi e metodi, nella sostanza, viene limitata l'ereditarietà.

L'effetto su un **field** e **variabili** è che esse **non possono essere assegnate più di una volta**; in altre parole, non può cambiare il loro valore, sono *costanti*. Forzando il cambio di valore viene prodotto un errore in fase di compilazione.

Entrambi i seguenti frammenti di codice non compilano per le ragioni di cui sopra,

```
public class Greeter {
  private final String msg = "Hi!";
  public void update(String msg) {
    this.msg = msg;
  }
}
```

```
/////
public class Greeter {
  public void update(final int n) {
    n = 18;
  }
  public void doThings() {
    final String s = "hi!";
    s = "hello";
  }
}
```

Le ragioni principali per l'adozione del modificatore `final` sono varie; principalmente, si adotta per la modellazione. Il modificatore `final` **modella entità che non possono cambiare**. Domandiamo perciò al compilatore di ricordarci che la nostra modellazione è quella di una costante - non può e non deve essere cambiata. Questa funzionalità serve sia a noi stessi, che agli altri sviluppatori.

Nella pratica, vi sono 3 casi d'uso:

- definire delle **costanti**, specialmente nel caso dei field;
- limitare la **riusabilità**, caso molto raro - in linea di principio, potrebbe esistere dei casi in cui è desiderabile che dei metodi non `private` mantengano un accesso regolato allo stato interno dell'oggetto (ad esempio, per i **getter** ed i **setter**). Tali metodi andrebbero impostati come `final`, di modo che lo stato sia modificabile solo ed esclusivamente in maniera regolata;
- rimarcare l'**immutabilità**, caso ancor più raro. La sintassi del linguaggio può consentire la mutazione di alcune costanti, non ovvie, anche nel caso in cui non sia desiderabile.

Si possono definire costanti sia per oggetti, che per le classi - in questo ultimo caso, il valore è uguale per tutte le istanze di tale classe.

Una costante per l'oggetto (object-wise) è un field **immutabile** che può avere valori differenti per i diversi oggetti, ma che una volta fissato non può cambiare. Di solito, per tali field viene deciso un modificatore `final private`.

Di contro, una costante per la classe (constant-wise) è un field **globale** e **immutabile**, cioè un placeholder per un valore codificato e costante per tutte le istanze. Esse sono dichiarate mediante i modificatori `final static`, con il modificatore `static` che la rende di fatto globale.

L'utilizzo di uno o dell'altro tipo di costanti è una prerogativa esclusiva dello sviluppatore. Un esempio di ciò è la classe `Person` definita nel seguente frammento,

```
public class Person {
  private String firstName;
  private String lastName;
  private final Date birthDate;
  private final String fiscalCode;
  public Person(Date birthDate, String fiscalCode) {
    this.birthDate = birthDate;
    this.fiscalCode = fiscalCode;
  }
}
```

dove i field `birthDate` e `fiscalCode` sono considerati **immutabili** per una singola istanza dell'oggetto (object-wise). Ogni costruttore **deve** necessariamente inizializzare i field immutabili - pena l'invalidità in fase di compilazione. Secondo questa modellazione, per definire una persona bisogna almeno definire codice fiscale e data di nascita. Una conseguenza di ciò è che ambedue i field immutabili non possono essere successivamente modificati.

Un esempio di class-wise constant è invece esplicitato nel frammento seguente,

```
public class Doc extends Person {
  private final static String TITLE_PREFIX = "Dr.";
  public String getFullName() {
    return TITLE_PREFIX + " " + firstName + " " + lastName;
  };
}
```

dove viene specificato che **ogni** istanza della classe `Doc` contiene il prefisso `"Dr."`. Poiché si tratta di una costante **`globale ed immutabile`**, viene adoperata una nuova naming convention: **tutto maiuscolo e parole separate mediante underscore**.

Un esempio di ciò è la costante globale `Math.PI`, con modificatore **`public final static`**.

Per rimarcare l'immutabilità, si adopera l'identificatore **`final`** negli argomenti di un metodo:

```
public void update(final int n) { // remarks immutability, no
    effect outside
  /* ... */
}
```

L'identificatore **`final`** diventa dunque un monito per lo sviluppatore a ricordare che il valore di `n` non deve cambiare, e non ha effetto al di fuori. In realtà, essendo un tipo primitivo sappiamo già che il suo valore non potrebbe ugualmente cambiare; il modificatore assume dunque la funzione di monito, ed andrebbe messo per ogni tipo primitivo.

Per quanto riguarda i tipi non primitivi passati per argomento, è utile definire ugualmente con **`final`**: sarebbe altrimenti possibile **riassegnare il riferimento**, impedendo di cambiare oggetto al quale l'argomento si riferisce[1]. In altre parole, se `p` fosse riassegnato, le successive modifiche avrebbero effetto su un altro oggetto, non sull'oggetto che dovrebbe essere modificato.

```
// Do not change what p refers to!
public void capitalizeName(final Person p) {
  p.setName(
    p.getName().substring(0, 1).toUpperCase() +
    p.getName().substring(1).toLowerCase()
  );
}
```

Nel seguente frammento di codice, viene inizializzato un riferimento ad un vettore di persone, e viene fatta un'iterazione. Con il **`for each`** viene adoperata una variabile d'iterazione **`final`**:

```
Person[] teachers = new Person[] {alberto, eric};
for (final Person teacher : teachers) {
  capitalizeName(teacher);
}
```

rimarcando che all'interno del ciclo **`for`** non vi è alcuna ragione per riassegnare il riferimento **`teacher`**. Supponiamo tuttavia il seguente frammento,

```
Person spareTeacher = new Person(/* ... */);
Person[] teachers = new Person[] {alberto, eric};
for (Person teacher : teachers) {
  if (teacher.isIll()) { // A
```

[1] Difatti, il modificatore **`final`** ha effetto **sul riferimento** dichiarato, non sull'oggetto a cui riferisce tale riferimento.

```
    teacher = spareTeacher;
  } // B
}
```

compila, ma non fa ciò che si immagina. Durante la prima iterazione (punto A), `teacher` viene riferito a riferire ciò che riferisce il primo elemento dell'array `teachers`: `alberto` (fra l'altro, non malato). Tuttavia, durante la seconda esecuzione, `eric` è malato e viene cambiato ciò a cui `teacher` riferisce, cioè lo stesso oggetto a cui riferisce `spareTeacher`. Tuttavia, non cambia il contenuto di `teachers[1]`! Ciò è accaduto perché abbiamo soltanto cambiato ciò che il riferimento riferiva, non un field di un array: sarebbe stato giusto riassegnare il riferimento contenuto in `teachers[1]`. Un modo per accorgersi di ciò è definire il ciclo in questa maniera,

```
Person spareTeacher = new Person(/* ... */);
Person[] teachers = new Person[] {alberto, eric};
for (final Person teacher : teachers) {
  if (teacher.isIll()) { // A
    teacher = spareTeacher; // irrelevant to change teacher
        reference
  } // B
}
```

Come regola quasi generale, i `for each` vogliono vedere un modificatore `final` nella variabile d'iterazione.

8.3 Lo scope e il lifetime

Lo **scope** è una proprietà statica degli identificatori che esplicita in quale porzione di codice l'identificatore può essere usato - in altre parole, menzionare un identificatore fuori dalla porzione di codice dal quale risulta "visibile" produrrà un errore in fase di compilazione. Si tratta di una proprietà **statica** degli identificatori.

Il **tempo di vita (lifetime)** è l'intervallo di tempo entro il quale un oggetto o un riferimento *esiste*. È una proprietà **dinamica** degli identificatori e degli oggetti.

In generale, i due concetti sono propri di altri linguaggi di programmazione, tuttavia il loro utilizzo pratico e il loro significato variano di molto in base al linguaggio. Inoltre, in Java anche il tipo di oggetto influenza il lifetime di tale entità.

8.3.1 Scope per le variabili

In Java, lo scope dei riferimenti dichiarati nella signature di un metodo è **tutto e solo il metodo**. In altre parole, ciò che è dichiarato all'interno della signature è visibile lungo l'intero metodo dove è dichiarato.

I riferimenti dichiarati nel codice, cioè per le *variabili locali*, sono visibili **dal momento della dichiarazione sino alla fine del blocco ove sono state dichiarate**. Dunque sono visibili nella sequenza di statement racchiusi fra parentesi graffe, cioè all'interno dei blocchi.

Un esempio di ciò è il frammento successivo,

```
public String capitalize(final String string) {
  String capitalized = "";
  for (final String token : string.split(" ")) {
    String head = token.substring(0, 1);
    String remaining = token.substring(1);
    capitalized = capitalized
      + head.toUpperCase() + remaining.toLowerCase() + " ";
  }
  return capitalized;
```

```
}
```

e si tratta di un metodo che *capitalizza* una stringa. Il valore di ritorno è una stringa nuova, poiché esse sono immutabili. Il metodo ha la funzionalità di rendere la prima lettera di ogni parola come maiuscola. Vi sono cinque differenti identificatori:

- **`string`** - visibile all'interno dell'intero metodo, poiché è stato dichiarato direttamente nella signature;
- **`capitalized`** - dalla riga 2 alla riga 10;
- **`token`** - dalla riga 3 alla riga 8;
- **`head`** - dalla riga 4 alla riga 8;
- **`remaining`** - dalla riga 5 alla riga 8.

Vi sono tuttavia 5 possibili migliorie al frammento:

1. il metodo potrebbe essere **`static`**;
2. il metodo aggiunge un *trailing space* alla fine della stringa;
3. **`head`** e **`remaining`** potrebbero vedere aggiunto il modificatore **`final`**;
4. esiste una condizione che causa un esito non accettabile, ovvero se vi sono due spazi di fila il metodo **`split`** creerà una stringa vuota. In tal caso, **`substring`** lancia un'eccezione;
5. se la stringa d'ingresso è **`null`**, il metodo non funziona.

8.3.2 Scope per field, metodi e classi - il modificatore `protected`

Abbiamo discusso dello scopde delle variabili, ora invece affronteremo lo scope relativo a field, metodi e classi.

Per quanto riguarda i field, metodi e classi lo scope è **determinato dal modificatore d'accesso**.

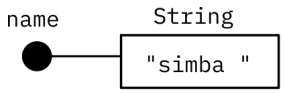
I field, noti anche come **`instance variables`**, i metodi e le classi possono godere di un modificatore d'accesso particolare oltre che a quello **`public`**, **`private`** e di default, chiamato **protected** - il modificatore **`protected`** rende visibile **`solo nella classe che estende`** direttamente o indirettamente la classe dove **`protected`** è utilizzato.

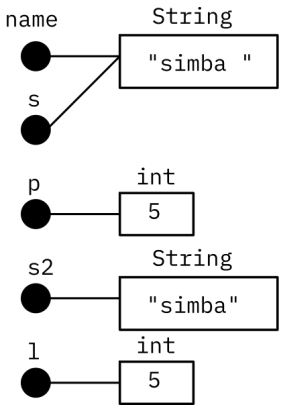
8.3.3 Lifetime

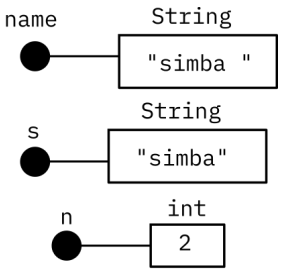
I riferimenti di tipo primitivo esistono **solo durante l'esecuzione del blocco**. Il tempo di vita è esattamente identificato dal blocco, sono noti.

Per quanto riguarda i riferimenti e oggetti di tipo non primitivo il tempo di vita è che **esistono soltanto durante l'esecuzione degli statement di un blocco**, a partire dal **`new`** che li crea ed **esistono almeno finché sono referenziati**. Il blocco è staticamente definito; tuttavia, i tipi non primitivi **esistono solo se referenziati** - in altre parole, se non sono referenziati possono esistere o non esistere (anche se, nella pratica, è la stessa cosa). Per i tipi non primitivi, dunque, è noto solo il tempo d'inizio della loro esistenza, ma non sappiamo esattamente quando l'oggetto cessa di esistere per davvero.

```
public int doThings(String s) {
  int p = 2;
  String s2 = s.trim();
  int l = s2.length(); // B
  return l / p;
}
/////
public void run() {
  String name = "simba "; // A
  int n = doThings(name); // C
```


name
String
"simba "


name
String
"simba "
s
p
int
5
s2
String
"simba"
l
int
5


name
String
"simba "
s
String
"simba"
n
int
2

```
}
```

Capitolo 9

La memoria della Java Virtual Machine

Contrariamente a quanto detto più volte, la memoria non è un blocco destrutturato come il diagramma oggetti-riferimenti suggerirebbe in prima analisi. La memoria della JVM è infatti organizzata in due zone distinte, lo **stack** e lo **heap**:

- lo **heap** è la zona della memoria dove risiedono i dati *a lungo termine*. Lo heap è fatto per immagazzinare informazioni a lungo termine. Lo heap è **globale** - c'è un solo heap per ogni JVM. Nello heap sono collocati **tutti gli oggetti creati con `new`**, dunque **tutti gli oggetti non primitivi**, con i loro field. Gli *array* stanno nello heap, anche quelli di tipi primitivi.

- lo **stack** è pensato per immagazzinare dati che vivono per meno tempo - **non è unico**, è organizzato in blocchi. Fisicamente, lo stack è costruito per disporre di un accesso più rapido. Nello stack ci vanno **tutti i riferimenti** e **tutti gli oggetti di tipo primitivo che non siano un field**, ad esempio tutte le variabili locali dichiarate all'interno dei metodi o nella loro signature. La dimensione di ogni blocco dello stack è determinabile al momento dell'invocazione del metodo - nel caso di vari percorsi possibili di computazione, viene scelta la massima quantità possibile. L'accesso allo stack è *più veloce*.

Tipicamente, lo heap è molto più grande. Esso viene *periodicamente svuotato*. Lo stack è invece organizzato in blocchi - ogni qualvolta che inizia l'esecuzione di un metodo viene **creato un nuovo blocco dello stack** associato a tale metodo. Nel nuovo blocco sono introdotte tutte le variabili locali dichiarate nel metodo. Il blocco viene prontamente rimosso **al termine dell'esecuzione del metodo per cui è stato generato**

Nella memoria, **gli oggetti di tipo primitivo sono il loro riferimento** - in altre parole, c'è un legame indissolubile fra il tipo primitivo ed il loro riferimento, cioè non possono essere concettualmente separati. Per i tipi primitivi, in definitiva, l'identificatore *è* l'oggetto. Questo cambio di mentalità spiega finalmente tutte le differenze e simitudini fra tipi primitivi e non primitivi; in particolare:

- i tipi primitivi **hanno il medesimo lifetime dei riferimenti** - di fatto, sono la stessa cosa;

- i tipi primitivi non hanno un vero e proprio riferimento, dunque al passaggio vengono passati **by value**, diversamente dai tipi non primitivi che sono invece passati **by reference**;

- l'assegnazione crea una **copia del valore** per i tipi primitivi, una copia del riferimento per gli oggetti di tipo non primitivo;

- l'operatore di uguaglianza `==` confronta il valore per i tipi primitivi, il riferimento per i tipi non primitivi;
- i tipi primitivi non possono fare riferimento a `null` - essi non hanno veri e propri riferimenti!

Dunque, ciò che prima erano riferimenti a tipi primitivi, ora sono i tipi primitivi stessi - nella stessa maniera, ciò che prima erano riferimenti a tipi non primitivi, ora sono **indirizzi di memoria dello heap** dove gli oggetti a cui fanno riferimento sono ospitati. Ad esempio `String* 0xAAF3` fa riferimento al tipo non primitivo `String` ospitato all'indirizzo `0xAAF3`. Per i tipi primitivi si cambierà notazione, per quelli non primitivi si manterrà la medesima.

La quantità di memoria occupata da un tipo primitivo è nota a priori:

- `byte` richiede 1 byte;
- `short` richiede 2 byte;
- `int` richiede 4 byte;
- `long` richiede 8 byte;
- `float` occupa 4 byte;
- `double` occupa 8 byte;
- `char` occupa 2 byte;
- ed infine `boolean`, che richiede probabilmente 1 solo bit - con la conseguenza che più variabili booleane sono solitamente inserite all'interno di un singolo byte.

I riferimenti, tipicamente, occupano 4 byte se la dimensione dello heap è inferiore a 32GB, mentre occupano 8 byte per dimensioni dello heap maggiore.

Per esempio,

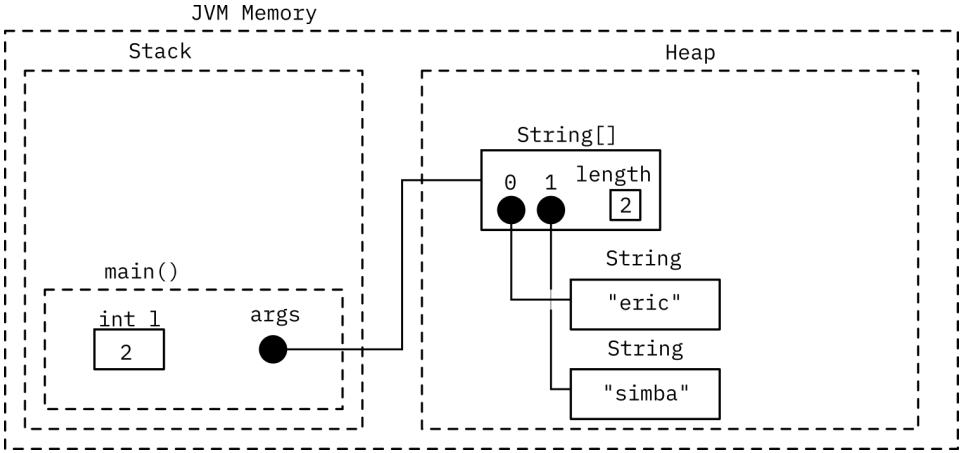


Figura 9.1: Diagramma oggetti-riferimenti aggiornato alle nuove nozioni di stack e heap.

```
public static void main(String[] args) {
  int l = args.length;
}

eric@cpu:~$ java Greeter eric simba
```

con il diagramma a riferimenti "aggiornato" alla nuova notazione espresso in Figura 9.1. Si osservi come, all'inizio della memoria, gli argomenti passati per linea di comando siano ospitati nello heap.

Sia il seguente frammento di codice

```
public static void main(String[] args) {
  (new Main()).run();
}

public void run() {
```

```
  String name = "simba "; // A
  int n = doThings(name); // C
}

public int doThings(String s) {
  int p = 2;
  String s2 = s.trim();
  int l = s2.length(); // B
  return l / p;
}

/////
eric@cpu:~$ java Greeter
```

Nello stato A, lo stack è popolato esclusivamente dagli stack di `main()`, dove nessun tipo primitivo è definito (vi è solo `args`, di grandezza 4 byte) e di `run()`, anch'essa di dimensione 8 byte, poiché ha al suo interno un riferimento di un tipo String ed un e un `int` che ancora dovrà formarsi ma che si sa già che dovrà esservi collocato. Nello heap vengono collocati `String[]` di `args` e la stringa `"simba "`, approssimativamente occupando 4 e 12 byte. Nello stato B, viene creato un nuovo blocco dello stack per `doThings()`. Vengono adoperati 16 byte, 8 per i due `int` ed altri 8 per i due riferimenti. Il riferimento `s` farà riferimento alla stringa nello heap. Al punto C vi è il ritorno del metodo `doThings()`: il suo blocco nello stack viene rimosso, e viene finalmente allocato l'`int` nello stack di `run()`.

9.1 La garbage collection

Nell'esempio di cui sopra, la stringa `"simba"`, dapprima con riferimento `s2` e poi rimasta nello heap e senza alcun riferimento associato è inutile; quando un oggetto non è più riferito, non è materialmente utilizzabile, nessuna istruzione può coinvolgere l'oggetto - è perciò **garbage**. La Java Virtual Machine dovrà dunque liberare la memoria in qualche maniera, di modo da non rischiare di riempire l'intero spazio dello heap. Si osservi che sono *garbage* anche tutti gli oggetti che sono riferiti soltanto da riferimenti anch'essi *garbage*.

Rimuovere la garbage non è semplice. Non è infatti computazionalmente facile eliminare tutta la garbage, poiché ciò equivarrebbe ad una ricerca nell'intero grafo per nodi non aventi archi entranti - un compito che la JVM non può svolgere in continuazione.

L'operazione di rimuovere tutti i rettangoli senza un adeguato riferimento dallo heap è detta **garbage collection**. La JVM decide ad intervalli regolari **quando** e **cosa** considerare garbage, e la rimuove. Di fondamentale importanza è il fatto che la garbage collection viene effettuata solo in determinate circostanze - in quel caso, trova almeno un po' di garbage, e la rimuove. Vi è un compromesso fra eliminare lo spazio occupato dalla garbage ed eliminare un eccessivo overhead. Non esiste un unico **garbage collector**: ci sono varie tecniche per realizzare un GC, con diversi algoritmi che possono avere bassissimo overhead ma minore garbage collection, o intensi burst di overhead con grandi raccolte di oggetti inutilizzabili.

Tipicamente, l'impatto che il garbage collector ha sul sistema in esecuzione dipende dal tipo di applicazione e dal momento in cui esso entra in azione.

Lo sviluppatore può **suggerire** alla JVM di eseguire la garbage collection mediante il metodo `System.gc()`. La documentazione dice in proposito

> `static void gc()` Runs the garbage collector in the Java Virtual Machine.
>
> Runs the garbage collector in the Java Virtual Machine.
>
> Calling the `gc` method suggests that the Java Virtual Machine expend effort toward recycling unused objects in order to make the memory they currently occupy available for reuse by the Java Virtual Machine. When control returns from the method call, the Java Virtual Machine has made a best effort to reclaim space from all unused objects. There is no guarantee that this effort will recycle any

particular number of unused objects, reclaim any particular amount of space, or complete at any particular time, if at all, before the method returns or ever.

La garbage collection ha un fondamentale aspetto positivo, quello di *togliere la responsabilità* allo sviluppatore del problema dello svuotamento della memoria. Prima della garbage collection, lo sviluppatore doveva esplicitamente liberare la memoria allocata con `malloc()` mediante il termine `free()`. I problemi di questo approccio sono i soliti - la possibilità di dimenticare i `free()` e conseguentemente la memoria potrebbe terminare lo spazio libero, oppure l'esecuzione di `free()` su indirizzi errati, la possibilità di scrivere su una memoria in precedenza allocata. Con la garbage collection, al prezzo di piccoli momenti in cui la garbage collection interviene e blocca l'applicazione in esecuzione, questi problemi legati allo sviluppo non esistono più.

La GC richiede molto tempo, non è predicibile *quanto* e non è predicibile nemmeno *quando*; potrebbe addirittura essere indesiderato in talune applicazioni o momenti particolari. Il grafo realizzato dagli oggetti e dai loro riferimenti potrebbe essere *aciclico* e molto grande. È possibile migliorare le performance del GC, modificandone le impostazioni oppure chiamandolo direttamente all'interno del codice.

9.2 Impostare la dimensione della memoria della JVM

La dimensione della memoria della JVM può essere impostata tramite flag da riga di comando, in particolare con il flag `-X`:

- `-Xms` indica la dimensione di partenza dello heap - la dimensione dello heap può cambiare poiché si tratta di una macchina virtuale;
- `-Xmx` indica la dimensione massima dello heap;
- `-Xss` determina la dimensione dello stack - solo uno poiché ha una *dimensione fissa*.

Ad esempio, l'invocazione seguente

```
eric@cpu:~$ java MyBigApplication -Xmx8G
```

imposta una massima dimensione dello heap pari a 8 Gigabyte.

Tipici errori durante l'esecuzione relativi alla mancanza di memoria sono i due seguenti,

- `java.lang.OutOfMemoryError: Java heap space` quando lo heap è eccessivamente riempito, o non vi sono "buchi" dove inserire i nuovi oggetti o array;
- `java.lang.StackOverFlowError` quando avviene uno stack overflow, nel caso in cui lo stack sia stato impostato a dimensione troppo piccola e vengano allocate troppe variabili in esso. Difficilmente lo stack è riempibile: in esso vi sono contenute variabili molto piccole. Tipicamente, anche la ricorsione senza condizione d'arresto può provocare lo stack overflow.

9.3 Le wrapper classes

Esistono delle classi, dette **wrapper classes**, che hanno la funzione di contenere i valori dei tipi primitivi pur non essendo tipi primitivi - tali classi permettono di modellare i tipi primitivi come non primitivi, avendo anche la potenzialità di svolgere metodi su di essi.

Le classi sono

- `Integer` per gli `int`;
- `Double` per i `double`;
- `Character` per i `char`;
- `Boolean` per i `boolean`;

- e così via.

Le wrapper classes sono oggetti **immutabili**, come le stringhe, e si collocheranno nella memoria heap.

La classe integer, secondo la documentazione,

> The `Integer` class wraps a value of the primitive type `int` in an object. An object of type `Integer` contains a single field whose type is int.
>
> In addition, this class provides several methods for converting an `int` to a String and a String to an `int`, as well as other constants and methods useful when dealing with an int.

La classe `Integer` contiene anche delle costanti:

- `static int BYTES` che contiene il numero di bytes usato per rappresentare un `int` in complemento a due;
- `MAX_VALUE`;
- `MIN_VALUE`;

e contengono gli pseudo-costruttori

> `static int parseInt(String s)` Parses the string argument as a signed decimal integer.
>
> `static int parseInt(String s, int radix)` Parses the string argument as a signed integer in the radix specified by the second argument.
>
> `static Integer valueOf(int i)` Returns an Integer instance representing the specified int value

per esempio

```
int n = Integer.parseInt("1100110", 2); // -> 102
```

ed altri esempi, relativi al return

> `int intValue()` Returns the value of this Integer as an int.
>
> `long longValue()` Returns the value of this Integer as a long after a widening primitive conversion.

9.3.1 Autoboxing ed autounboxing

In Java esistono anche gli **autoboxing** e gli **autounboxing** - speciali scorciatoie sintattiche adottate per semplificare l'uso delle wrapper classes. Ad esempio,

```
public void doIntThings(int n) { /* ... */ }
public void doIntegerThings(Integer n) { /* ... */ }

Integer i = 3; // autoboxing
doIntThings(i); // autounboxing

int n = 3; // autoboxing
doIntegerThings(n); //autounboxing

Integer i = 2; // autoboxing
i++; // autounboxing, esecuzione del '++', autoboxing
```

che viene tradotto in

```
Integer i = Integer.valueOf(3);
doIntThings(i.intValue());

int n = 3;
doIntegerThings(Integer.valueOf(n));

Integer i = Integer.valueOf(2);
i = Integer.valueOf(i.intValue()+1);
```

In generale è bene adoperare la classe `int` qualora non fosse assolutamente richiesta una wrapper class.

A livello di memoria, gli `Integer` sono disposti nello heap e hanno un riferimento nello stack, mentre gli `int` sono direttamente disposti nello stack. Gli `Integer` tipicamente producono maggiore lavoro per il garbage collector, in quanto sono immutabili e tendono ad accumularsi ad ogni modifica.

Attenzione però: per le classi wrapper **non c'è unboxing per l'operazione di uguaglianza ==**! L'autounboxing avviene **solo** se uno dei due operandi è un tipo primitivo; invece **non avviene** nel caso in cui **entrambi gli operandi sono wrapper classes**, o comunque quando entrambi sono oggetti.

Ad esempio,

```
Integer n = 300;
Integer m = 300;
int k = 300;
System.out.printf("n ?= m -> %b%n", n==m); // -> false!!!
System.out.printf("n ?= k -> %b%n", n==k); // -> true!!!
```

Nel caso in cui `(n+1)==(m+1)` produce `true` poiché avviene l'autounboxing dei numeri di sopra, e vengono paragonati i valori `301 == 301`, che forniscono il risultato `true`.

È meglio adoperare il metodo `equals()` quando vi è da fare un paragone (l'IDE lo evidenzia).

Capitolo 10

Input ed output avanzati

Le due maniere fondamentali per ottenere l'input sono mostrate di seguito, ed erano già state illustrate tempo fa,

```
BufferedReader reader = new BufferedReader(
  new InputStreamReader(System.in)
);
/* ... */
String line = reader.readLine();

Scanner scanner = new Scanner(System.in);
/* ... */
String s = scanner.next();
int n = scanner.nextInt();
double d = scanner.nextDouble();
```

Un'importante astrazione da introdurre è quella dello **stream**. Uno stream è un flusso di dati, un "tubo" con sequenzialità *fra due endpoint*, con una *direzione*, ed un *tipo di dati*.

Due modelli fondamentali sono i seguenti,

- l'**output stream** ad un dispositivo dove i dati possono essere scritti come `byte[]`;
- l'**input stream** da un dispositivo da cui i dati possono essere letti come `byte[]`;

TODO disegni 277

La rappresentazione adoperata sarà quella in Figura **??**: essa rappresenta il *generico dispositivo*, e si tratta di un'astrazione del dispositivo. L'interazione con uno stream avviene dapprima creando lo stream, e poi reiterando la lettura o la scrittura di `byte[]` array sullo stream. Un **device** potrebbe essere qualunque cosa - un file, la memoria, la rete.

Un output stream può avere delle *capacità di processamento* dei dati. Si potrebbe voler comprimere o decomprimere il flusso di `byte[]`, o la trasformazione di `byte[]` da (o in) altri tipi. Impilare gli stream di fila è una maniera per realizzare un'operazione dopo l'altra.

Le classi che realizzano gli stream sono `InputStream` ed `OutputStream`. Uno stream si può comporre sovrapponendolo ad un altro stream: il risultato è comunque uno stream nella medesima direzione. Ciascuno stream è un **device** per il dispositivo sopra. Ciascuno stream, dunque, offre la medesima

interfaccia; questo particolare design si chiama **filter pattern** - l'idea è di fornire un'"unica" interfaccia alle applicazioni che hanno bisogno di fare input—output, al contempo realizzando alcune funzionalità. Ciascun livello della pila di stream fornisce un "filtraggio" delle funzionalità.

Non c'è bisogno di modificare il codice: grazie al **polimorfismo**, il tipo del device farà sì che lo stream si comporti in maniera ogni volta differente. Lo stesso vale per i differenti processing. È possibile anche comporre tipi di dati maggiormente complessi, ciò che si fa è porre in cima alla pila un `InputStream` che è in grado di gestire un flusso `int[]`.

Vi sono molti tipi di `InputStream`, ne vedremo soltanto alcuni.

10.1 I/0 di byte

La classe `OutputStream`,

> This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

ed ha i metodi,

> **`void close()`** Closes this output stream and releases any system resources associated with this stream.
>
> **`void flush()`** Flushes this output stream and forces any buffered output bytes to be written out.
>
> **`static OutputStream nullOutputStream()`** Returns a new OutputStream which discards all bytes.
>
> **`void write(byte[] b)`** Writes b.length bytes from the specified byte array to this output stream.
>
> **`void write(byte[] b, int off, int len)`** Writes len bytes from the specified byte array starting at offset off to this output stream.
>
> **`abstract void write(int b)`** Writes the specified byte to this output stream.

e scrive soltanto `byte[]`.

La classe è **abstract**, cioè non può essere utilizzata o istanziata.

L'errore `OutputStream.nullOutputStream()` fornisce un device che rifiuta sempre tutti i byte.

Più in dettaglio,

> **`void write(byte[] b, int off, int len)`** Writes len bytes from the specified byte array starting at offset off to this output stream.
>
> Writes len bytes from the specified byte array starting at offset off to this output stream. The general contract for write(b, off, len) is that some of the bytes in the array b are written to the output stream in order; element b[off] is the first byte written and b[off+len-1] is the last byte written by this operation.

Un uso pratico può essere il seguente,

```
byte[] data = /* ... */
OutputStream os = /* ... */
os.write(data, 0, 3); //A
os.write(data, 3, 2); //B
os.write(new byte[2]); //C
```

Lo stream:

- dopo l'esecuzione di A, contiene 3 byte - il cursore punta 3;
- dopo l'esecuzione di B, contiene 5 byte - il cursore punta 5;
- dopo l'esecuzione di C, contiene infine 7 byte - il cursore punterà sull'ottavo byte, indice 7.

Vediamo ora con maggiore dettaglio i metodi `write()`,

```
write(byte[] b, int off, int len)
```

> The general contract for `write(b, off, len)` is that some of the bytes in the array `b` are written to the output stream in order; element `b[off]` is the first byte written and `b[off+len-1]` is the last byte written by this operation.

```
write(byte[] b)
```

> The general contract for `write(b)` is that it should have exactly the same effect as the call `write(b, 0, b.length)`.

In altre parole, la documentazione ci fornisce il contratto che dovremmo seguire nel caso facessimo l'override del metodo `write()` - le sottoclassi di `OutputStream` **devono** almeno fornire un metodo che è in grado di scrivere un singolo byte. Il seguente metodo andrebbe riscritto ugualmente,

```
write(byte[] b, int off, int len)
```

poiché chiama, uno dopo l'altro, `write(b[i])`, cioè fa una cosa simile

```
public void write(byte[] b, int off, int len) {
  for (int i = off; i<off+len; i++) {
    write(b[i]); // calls write
  }
}
```

ed è poco efficiente. Un esempio di scarsa efficienza è l'uso di esso per un protocollo di rete; a livello di TCP è meglio inviare un numero maggiore di byte per volta. C'è infatti un costo fisso (overhead) nella scrittura su dispositivo, a prescindere dalle dimensioni dei dati, nella scrittura di un maggior numero di byte per volta. Anche i vecchi Hard Disk possono godere di un beneficio nell'ottimizzazione del metodo `write(byte[] b, int off, int len)`.

10.1.1 Associazione con i device

In pieno stile UNIX, Java tratta tutti i device con la medesima interfaccia. Dunque, si desideri accedere ad un *file*, ad una *memoria* o ad un *dispositivo di rete*, il codice da scrivere è del tutto simile nei tre casi:

- nel caso del *file*:

```
OutputStream os = new FileOutputStream(/* ... */);
byte[] data = /* ... */
os.write(data);
```

- nel caso di un *dispositivo di rete*:

```
Socket socket = /* ... */
OutputStream os = socket.getOutputStream();
byte[] data = /* ... */
os.write(data);
```

- nel caso di una *memoria*:

```
OutputStream os = new ByteArrayOutputStream();
byte[] data = /* ... */
os.write(data);
```

A seconda del dispositivo però varierà il modo in cui la JVM dovrà interfacciarsi con esso. Per esempio, nel caso di `socket`, il sistema operativo sottostante dovrà creare un corrispondente socket con relativa connessione TCP - prima di poterla adoperare, dunque, essa dovrà essere creata.

La classe `File` modella l'ubicazione di un file, con le corrispondenti *system calls* dentro e sotto la JVM.

Dalla documentazione della classe `FileOutputStream`,

> A file output stream is an output stream for writing data to a File or to a FileDescriptor. **Whether or not a file is available or may be created depends upon the underlying platform**. Some platforms, in particular, allow a file to be opened for writing by only one FileOutputStream (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.
>
> `FileOutputStream(File file)` Creates a file output stream to write to the file represented by the specified File object.
>
> `FileOutputStream(File file, boolean append)` Creates a file output stream to write to the file represented by the specified File object.
>
> `FileOutputStream(String name)` Creates a file output stream to write to the file with the specified name.
>
> `FileOutputStream(String name, boolean append)` Creates a file output stream to write to the file with the specified name.

Si noti innanzitutto che la classe `FileOutputStream` gestisce l'output stream per la scrittura verso oggetti di classe `File` e `FileDescriptor`; l'effettiva riuscita della scrittura **dipende dal sistema operativo sottostante**. Per alcune di esse, ad esempio, potrebbe essere possibile utilizzare esclusivamente un singolo `FileOutputStream` per volta.

Nella seconda parte, sono riportati alcuni costruttori degni di nota - si noti il booleano `append`, di default `false`, il quale ci fornisce la possibilità di "appendere" i dati al file (anziché ricominciare con la scrittura dall'inizio del file).

Per quanto riguarda la classe `socket` osserviamo un metodo in particolare,

> `OutputStream getOutputStream()` Returns an output stream for this socket.

dove l'utente non è tenuto a sapere necessariamente la classe dell'output stream fornito da `getOutputStream()`.

Esiste poi la classe `ByteArrayOutputStream`: essa modella il concetto

> This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be retrieved using `toByteArray()` and `toString()`.
>
> `int size()` Returns the current size of the buffer. `byte[] toByteArray()` Creates a newly allocated byte array.

in pratica, modella un **buffer** (buffer è il nome gergale per un byte array).

L'utilizzo del `ByteArrayOutputStream` è nella maniera seguente,

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
byte[] data = /* ... */
baos.write(data);
byte[] written = baos.toByteArray();
System.out.println(Arrays.equals(data, written)); // -> true
```

con una prima invocazione del metodo `write()` contentente i dati da scrivere, e successivamente da estrarre con metodo `toByteArray()`.

Tutte le classi "output stream" viste fino ad ora sono sottoclassi (ereditano) della classe `OutputStream`.

10.1.2 L'End Of Stream

L'**end of stream** (EOS) è un marcatore logico di cui non interessa la dimensione, il cui scopo è delimitare lo stream - oltre a tale marcatore, non c'è più nulla. Il marcatore EOS è inserito dal comando `os.close()`, il che chiude definitivamente lo stream, e non è più possibile scrivere nulla con esso:

```
byte[] data = /* ... */
OutputStream os = /* ... */
os.write(data, 0, 3); //A
os.close(); //B
```

Al punto A è ancora possibile scrivere dati, la "freccia" punterà al prossimo byte da scrivere. Al punto B invece viene collocato l'End of Stream alla fine della sequenza, e nessun altro dato sarà scrivibile - lo stream è definitivamente chiuso.

Rispettare la chiusura degli output stream è di fondamentale importanza a livello del sistema operativo, dove un output stream potrebbe inificiare le prestazioni dell'intero sistema, o nei casi migliori, consumare semplicemente inutili risorse.

Dalla documentazione della classe `OutputStream.close()`:

> Closes this output stream and releases any system resources associated with this stream. The general contract of close is that it closes the output stream. A closed stream cannot perform output operations and cannot be reopened.
>
> The `close` method of OutputStream does nothing.

In altre parole, la classe `OutputStream` non modella alcun device in particolare, **esso modella una generica risorsa di sistema**.

Un altro caso particolare è quello della classe `ByteArrayOutputStream`, per cui la chiusura non ha alcun effetto - i metodi comuni possono essere chiamati anche successivamente alla chiusura del byte array output stream.

10.2 L'input stream

La classe `InputStream` è la *superclasse di tutte le classi che rappresentano un input stream di byte*.

> `abstract int read()` Reads the next byte of data from the input stream. `int read(byte[] b)` Reads some number of bytes from the input stream and stores them into the buffer array b. `int read(byte[] b, int off, int len)` Reads up to len bytes of data from the input stream into an array of bytes.

L'input stream legge solo array `byte[]`, e il metodo `InputStream.nullInputStream()` fornisce un dispositivo di input dal quale non vi sono byte da leggere (EOS collocato alla posizione 0, l'equivalente di `/dev/null`).

In lettura, si adopera il metodo `read()`. In particolare,

> `int read(byte[] b, int off, int len)` Reads up to len bytes of data from the input stream into an array of bytes.
>
> Reads up to len bytes of data from the input stream into an array of bytes. An `attempt` is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer.

Il numero di byte letti viene restituito sotto forma di intero.

Inoltre, un po' come nel caso della scrittura `read(b)` è *per contratto* come eseguire `read(b, 0, b.length)`.

```
byte[] buffer = new byte[100];
InputStream is = /* ... */ //A - cursor at 0
is.read(buffer, 0, 4); //B - cursor at 4
is.read(buffer, 4, 1); //C - cursor at 5
```

In pari modo, l'associazione con il device è simile a quella con il metodo `write`:

```
//File:
InputStream is = new FileInputStream(/* ... */);
byte[] buffer = new byte[100];
is.read(buffer, 0, 10);

//Network (TCP):
Socket socket = /* ... */
InputStream is = socket.getInputStream();
byte[] buffer = new byte[100];
is.read(buffer, 0, 10);
```

```
//Memory:
InputStream is = new ByteArrayInputStream(/* ... */);
byte[] buffer = new byte[100];
is.read(buffer, 0, 10);
```

Le similitudini non fiscono qua: si possono infatti istanziare le sottoclassi figlie di (che ereditano da) `InputStream`, infatti secondo la documentazione della classe `FileInputStream`

> A FileInputStream obtains input bytes from a file in a file system. What files are available depends on the host environment.
>
> `FileInputStream(File file)` Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system. `FileInputStream(String name)` Creates a FileInputStream by opening a connection to an actual file, the file named by the path name name in the file system.

Un oggetto della classe `ByteArrayInputStream` contains an internal buffer (il dispositivo) that contains bytes that may be read from the stream. An internal counter keeps track of the next byte to be supplied by the read method.

Essa ha come costruttori

> `ByteArrayInputStream(byte[] buf)` Creates a ByteArrayInputStream so that it uses buf as its buffer array.

10.2.1 Differenze nell'astrazione fra input ed output stream

La principale differenza fra input ed output stream risiede nella gestione dell'End of Stream. Se infatti per un output stream **non vi sono in principio limiti nella scrittura** e bisognerà piazzare un EOS qualora servisse chiudere il dispositivo, per un input stream **i dati leggibili sono limitati**, poiché potrebbe esistere già un EOS nel device che `InputStream` astrae.

Sia infatti nel caso in cui si faccia uso di un `FileInputStream` che di un `ByteArrayInputStream`, i device avranno una dimensione limitata, perciò esisterà un EOS ad essi relativo (o comunque il device avrà una dimensione limitata di dati da leggere, almeno temporaneamente).

Infatti, esiste una distinzione netta nel caso di lettura con EOS e senza:

> `int read(byte[] b, int off, int len)` Reads up to len bytes of data from the input stream into an array of bytes.
>
> Reads up to len bytes of data from the input stream into an array of bytes. **An attempt is made to read as many as len bytes, but a smaller number may be read**. The number of bytes actually read is returned as an integer.
>
> This method **blocks until input data is available, end of file is detected, or an exception is thrown**.
>
> If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.

In altre parole, al termine della disponibilità dei dati proveniente da un input buffer, o viene trovato un EOS, oppure il metodo viene bloccato finché non sono disponibili altri dati.

Nel caso il prossimo byte sia un EOS, viene restituito immediatamente -1, nel caso di nessun dato disponibile metti in attesa. Diversamente, se vi sono dati disponibili, li leggerà tutti fino ad un massimo di `len` bytes letti restituendo il numero di byte letti.

Per esempio, si supponga un device contenente 5 byte all'istante zero, e ricevente al secondo 10 altri 3 byte, seguiti da un EOS. Eseguendo le istruzioni di seguito si otterrebbe l'effetto indicato nei commenti,

```
is.read(buf, 0, 3);
// t=0 -> t~0, ret 3
```

```
is.read(buf, 0, 3);
// t~0 -> t~0, ret 2
is.read(buf, 0, 3); // BLOCK!
// t~0 -> t~10, ret 3
is.read(buf, 0, 3);
// t~10 -> t~10, ret -1
```

dunque incontrando un EOS durante la lettura si otterrebbe il valore di ritorno -1.

Il blocco e l'attesa di input del metodo `read()` sono necessari per l'attesa di input: si pensi all'`InputStream System.in`, in questo caso finché non viene premuto il tasto di "invio" l'input stream non può ricevere alcunché. Fra l'altro, il `close()` su `System.in` non ha alcun effetto. Parimenti, per un dispositivo di rete `InputStream getInputStream()` legge dati dal socket mentre essi arrivano, e mette il tutto in attesa finché non arrivano altri dati.

10.2.2 Lettura e scrittura di un file

La classe `File` eredita direttamente da `Object` e, contrariamente a quanto ci si possa aspettare, **non rappresenta un file**: An abstract representation of file and directory pathnames. User interfaces and operating systems use system-dependent pathname strings to name files and directories. This class presents an abstract, system-independent view of hierarchical pathnames.

La classe `File` non appartiene alla gerarchia degli stream - si limita infatti a modellare i **nomi di percorso di file e directory**. I file sono dunque modellati nel senso di nomi di percorso, anch'essi aventi una struttura ad albero. I `File` descrivono in termini formali come descrivere un pathname, con l'ultimo elemento della gerarchia file o directory. I file sono dunque i nodi terminali - ogni directory ha 0 o più nodi figlio.

Tramite la classe `File` è possibile rinominare i file ed eseguire altre operazioni analoghe,

> `boolean canExecute()` Tests whether the application can execute the file denoted by this abstract pathname.
>
> `boolean canRead()` Tests whether the application can read the file denoted by this abstract pathname.
>
> `boolean canWrite()` Tests whether the application can modify the file denoted by this abstract pathname.
>
> `boolean delete()` Deletes the file or directory denoted by this abstract pathname.
>
> `boolean exists()` Tests whether the file or directory denoted by this abstract pathname exists.
>
> `boolean isDirectory()` Tests whether the file denoted by this abstract pathname is a directory.
>
> `String[] list()` Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
>
> `File[] listFiles()` Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
>
> `boolean mkdir()` Creates the directory named by this abstract pathname.
>
> `boolean renameTo(File dest)` Renames the file denoted by this abstract pathname.
>
> `boolean setExecutable(boolean executable)` A convenience method to set the owner's execute permission for this abstract pathname.

non è tuttavia possibile leggere o scrivere dai file (non sono stream!).

L'apertura avviene tramite lo stream `FileOutputStream(File file)` o `FileOutputSt name)`. Quando si esegue un tentativo di scrittura di un file tramite uno stream, questi accederà al file "reale" ospitato tramite il sistema operativo mediante l'oggetto di classe `File`. Qualora la stringa dello stream non rappresenti un file realmente esistente in memoria, **questi viene creato**. Attenzione,

poiché **il contenuto del file viene cancellato se non viene specificato il flag booleano append**.

L'effettiva scrittura e lettura di un file dipendono dal sistema operativo sottostante.

10.2.3 Copia ed incolla di un file

Il seguente codice realizza un copia-incolla di un file, da una posizione ad un'altra specificata negli argomenti,

```
public class FileCopier {
  public static void main(String[] args) throws
      FileNotFoundException, IOException {
    InputStream is = new FileInputStream(args[0]);
    OutputStream os = new FileOutputStream(args[1]);
    byte[] buffer = new byte[1024];
    while (true) {
      int nOfBytes = is.read(buffer);
      if (nOfBytes == -1) {
        break;
      }
      os.write(buffer, 0, nOfBytes);
    }
    is.close();
    os.close();
  }
}
```

tuttavia, esso può essere reso molto più generale facendo uso sia dell'*ereditarietà* che del *polimorfismo*, ad esempio tramite la funzione

```
public class Util {
  public static void copyAndClose(InputStream is, OutputStream os)
      throws FileNotFoundException, IOException {
    byte[] buffer = new byte[1024];
    while (true) {
      int nOfBytes = is.read(buffer);
      if (nOfBytes == -1) {
        break;
      }
      os.write(buffer, 0, nOfBytes);
    }
    is.close();
    os.close();
  }
}
```

la quale fa un più generico uso del concetto di input stream ed output stream, generalizzando ad ogni tipo di stream.

10.3 Input e output di tipi primitivi

Finora abbiamo visto le funzionalità I/O offerte dagli stream. Ora vediamo però la classe `DataOutputStream`, la quale offre **capacità di filtraggio** e va montata su un `OutputStream` opportuno,

> A data output stream lets an application write primitive Java data types to an output stream in a **portable** way. An application can then use a data input stream to read the data back in.
>
> `DataOutputStream(OutputStream out)` Creates a new data output stream to write data to the specified underlying output stream.
>
> `void writeDouble(double v)` Converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.

`void writeFloat(float v)` Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.

`void writeInt(int v)` Writes an int to the underlying output stream as four bytes, high byte first.

`void writeLong(long v)` Writes a long to the underlying output stream as eight bytes, high byte first.

La documentazione afferma varie volte che l'azione indicata viene svolta **high byte first**, garantendo dunque **la portabilità**. La classe `DataOutputStream` è figlia della classe `FilterOutputStream`.

Parimenti, la classe `DataInputStream` sarà figlia della classe `FilterInputStream`, e modellerà uno stream con capacità di filtraggio, stavolta di input,

A data input stream lets an application read primitive Java data types from an underlying input stream in a **machine-independent** way. An application uses a data output stream to write data that can later be read by a data input stream.

`DataInputStream(InputStream in)` Creates a DataInputStream that uses the specified underlying InputStream.

`double readDouble()` See the general contract of the readDouble method of DataInput. `float readFloat()` See the general contract of the readFloat method of DataInput. `int readInt()` See the general contract of the readInt method of DataInput. `long readLong()` See the general contract of the readLong method of DataInput.

La classe `FilterOutputStream` è la superclasse di tutte le classi che filtrano output stream. Questi stream risiedono sopra output stream *preesistenti* (underlying output streams) i quali vengono adoperati come destinazione base di dati, ma implementando capacità di filtraggio, trasformazioni di dati lungo il tragitto o fornendo funzionalità aggiuntive.

Similmente, la classe `FilterInputStream` è la superclasse di tutte le classi che filtrano input stream. Questi stream contengono input stream, i quali vengono adoperati come sorgente base di dati, possibilmente trasformando i dati lungo il percorso o fornendo funzionalità aggiuntive alla stregua degli output stream.

Quando si adopera le classi `DataInputStream` e `DataOutputStream`, scritte assieme dagli sviluppatori Java, non dobbiamo preoccuparci della conversione, poiché esse convertono qualunque tipo di dato in `byte[]`. Grazie l'astrazione ad interfaccia degli stream, ogni macchina legge i dati alla stessa maniera (si realizza la portabilità). Tuttavia, questa portabilità è garantita per il trattamento del singolo dato - essa non viene assolutamente garantita per **il significato** dei dati in viaggio attraverso gli stream! Bisogna infatti rispettare un **protocollo valido** per entrambe le parti.

10.3.1 Input ed output stream con compressione Zip

Esistono input stream ed output stream in grado di **comprimere i dati**. In particolare, le classi `GZIPInputStream` e `GZIPOutputStream` modellano, rispettivamente, input stream ed output stream in grado di produrre una compressione dei dati `gzip` (si tratta di un algoritmo di compressione e decompressione). Entrambe le classi offrono la medesima interfaccia degli stream già incontrati, tuttavia non offrono alcuna garanzia della avvenuta scrittura nel device.

La classe `GZIPInputStream` eredita da `java.util.zip.DeflaterOutputStream`, che a sua volta eredita da `java.io.FilterOutputStream` - similmente ciò avviene per la classe `GZIPOutputStream`.

Dalla documentazione, per `GZIPOutputStream`

This class implements a stream filter for writing compressed data in the GZIP file format.

`GZIPOutputStream(OutputStream out)` Creates a new output stream with a default buffer size.

`void finish()` Finishes writing compressed data to the output stream without closing the underlying stream.

`void write(byte[] buf, int off, int len)` Writes array of bytes to the compressed output stream.

mentre invece per la classe `GZIPInputStream`,

> This class implements a stream filter for reading compressed data in the GZIP file format.
>
> **`GZIPInputStream(InputStream in)`** Creates a new input stream with a default buffer size.
>
> **`void close()`** Closes this input stream and releases any system resources associated with the stream.
>
> **`int read(byte[] buf, int off, int len)`** Reads uncompressed data into an array of bytes.

Il protocollo **`gzip`** è un protocollo di compressione, il quale prende una sequenza di byte e la modifica, generando una nuova sequenza di byte[1]. In Java, tuttavia, esiste anche una classe `Zip` che modella il concetto di **stream che operano con i file**: essi possono contenere file, directory, e così via. Un file `Zip` codifica un insieme di file compressi dall'algoritmo **`gzip`**, con annessa struttura di directory.

Esistono anche `ZipInputStream` e `ZipOutputStream`, classi per modellare gli stream di file `Zip`.

[1] In pratica, sia $|B| = 2^8$ il totale numero di byte, gzip è una funzione

$$f : B^* \to B^*,$$

con $B^* = \cup_{i=0}^{\infty} B^i$.

Capitolo 11

Buffered input and output

La maggior parte dei dispositivi sono scritti e letti dal sistema operativo. In quasi tutti i casi (tranne che nel `ByteArrayOutputStream`) il device è un *dispositivo fisico*, per il quale accesso è necessario fare domanda al sistema operativo da parte della JVM. Vi sono vari livelli di astrazione,

- l'output stream;
- il sistema operativo;
- il device fisico.

I dispositivi fisici sono dotati, per loro caratteristica, di una **dimensione ideale** delle richieste di lettura e scrittura - ad esempio, nel caso degli hard disk tipicamente si richiedono dimensioni di 8KB per questioni di ottimalità. La medesima cosa avviene per i protocolli di rete.

Invocare il sistema operativo è **costoso**. Da una parte sappiamo per conoscenza del dominio informatico che ogni dispositivo ha una dimensione ideale di lettura e scrittura, mentre dall'altra parte sappiamo che gli input ed output stream possono richiedere letture e scritture di dimensione arbitraria. L'idealità in questo caso non sarebbe rispettata.

Tipicamente ciò che avviene è che la lettura e scrittura per gli output stream invia alla JVM una determinata richiesta per *n* byte - successivamente, la JVM chiede al sistema operativo di scrivere o leggere altrettanti *n* byte. Ci si aspetta dunque che il sistema operativo renda ottimale la richiesta, adattandola al caso ideale del dispositivo.

La soluzione sono i **buffered stream**. I buffered stream sono stream dotati di un buffer. Il buffer mantiene i dati in esso, finché non è il momento opportuno per inviare i dati al sistema operativo. Esistono due classi, `BufferedOutputStream`, e `BufferedInputStream`. Sono sempre montati su un `OutputStream`, dunque offrono la medesima interfaccia, tuttavia sono dotati di **buffering**.

Un `BufferedOutputStream` riceve richieste come al solito, però mette i dati in un buffer, e **solo quando pieno** fa una richiesta al successivo strato. Simmetricamente, il `BufferedInputStream` legge come al solito, mette i dati in un buffer, e li consegna esclusivamente quando il buffer è pieno. L'applicazione può ignorare il fatto che l'output stream sia dotato di buffering o no. Tipicamente, un buffered stream risiede **sopra** ad un `OutputStream`, dunque nella catena degli stream esso è collocato sopra un `OutputStream`.

Dunque, la consegna dei dati al livello sottostante (per gli output stream) o soprastante (per gli input stream) avviene esclusivamente quando il buffer è pieno.

La documentazione per `BufferedInputStream` è la seguente,

> A `BufferedInputStream` adds functionality to another input stream - namely, the ability to buffer the input and to support the `mark` and `reset` methods. When the `BufferedInputStream` is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time.
>
> `BufferedInputStream(InputStream in)` Creates a `BufferedInputStream` and saves its argument, the input stream `in`, for later use.
>
> `BufferedInputStream(InputStream in, int size)` Creates a `BufferedInputStream` with the specified buffer size, and saves its argument, the input stream `in`, for later use.

mentre per la classe `BufferedOutputStream`,

> The class implements a buffered output stream. By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.
>
> `BufferedOutputStream(OutputStream out)` Creates a new buffered output stream to write data to the specified underlying output stream.
>
> `BufferedOutputStream(OutputStream out, int size)` Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size.

Le due classi, rispettivamente, estendono `FilterInputStream` e `FilterOutputStream` - in altre parole, le due classi fanno ampio uso della modularità, aggiungendo una nuova funzionalità nascondendo la complicazione agli strati superiori. Per il `BufferedOutputStream`, tuttavia, la modularità non è completa: viene ridefinito il metodo `flush()`. Il metodo `flush()` permette di effettuare delle **esplicite richieste di scrittura**, in altre parole si può scrivere con tale metodo senza aspettare che il buffer sia completamente pieno. All'esecuzione di `flush` viene svuotato completamente il buffer:

> `public void flush()` Flushes this buffered output stream. This forces any buffered output bytes to be written out to the underlying output stream.

Invocare `flush` si propaga a tutti gli stream sottostanti - tuttavia, non si potrà mai propagare oltre la JVM (non raggiungerà il sistema operativo, naturalmente). In altre parole, **non è comunque garantita la scrittura sul dispositivo fisico**, poiché il sistema operativo potrebbe non essere pronto per scrivere, o potrebbero esserci altre ragioni per non poter scrivere immediatamente.

L'utilizzo dei buffered stream è buona prassi nella progettazione di un'applicazione, poiché garantisce un maggiore controllo sulla scrittura. Ad esempio,

```
OutputStream os = new BufferedOutputStream(
  new FileOutputStream(file)
);
```

L'utilizzo di `flush()` è automatico all'invocazione del metodo `close()`, ragion per cui se l'applicazione è robusta e fatta con decoro, non sarà necessario svuotare manualmente il buffer. Ciononostante, `flush()` è necessario per gli output stream, specialmente nel caso del **dialogo fra applicazioni**. Per gli input stream invece non è necessario: `BufferedInputStream` legge da sotto almeno la quantità di dati richiesta, possibilmente leggendo di più.

Capitolo 12

Input ed output di testo

L'input e l'output di testo è una sequenza di caratteri, perciò viene gestito con degli array `char[]`. Un `char` è codificato con uno o più `byte` - la maniera precisa con cui ciascuno di essi è codificato è specificato nello **charset**. Per ogni carattere nella pratica serviranno più di un byte, quali e quanti saranno descritti nel charset. Java si prende carico dei charset attraverso la classe `Charset`, tipicamente però non si fa uso dei charset.

L'input—output avviene tramite classi analoghe a quelle già viste per i dati binari, con la stessa interfaccia ed il nome semplicemente diverso: avremo le classi `Writer` e `Reader` che sostituiranno le classi `OutputStream` ed `InputStream`. Essi possono entrambi essere manipolati tramite il filter pattern, e possono essere associati con i dispositivi. Sono, nella pratica, del tutto simili agli stream binari.

Esempi di protocolli che manipolano dati testuali sono il protocollo HTTP, i file CSV, e così via.

La documentazione per `Writer`,

> Abstract class for writing to character streams. The only methods that a subclass must implement are `write(char[], int, int)`, `flush()`, and `close()`. Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

> `Writer append(char c)` Appends the specified character to this writer.
>
> `Writer append(CharSequence csq)` Appends the specified character sequence to this writer.
>
> `Writer append(CharSequence csq, int start, int end)` Appends a subsequence of the specified character sequence to this writer.
>
> `abstract void close()` Closes the stream, flushing it first.
>
> `abstract void flush()` Flushes the stream.
>
> `static Writer nullWriter()` Returns a new Writer which discards all characters.
>
> `void write(char[] cbuf)` Writes an array of characters.
>
> `abstract void write(char[] cbuf, int off, int len)` Writes a portion of an array of characters.
>
> `void write(int c)` Writes a single character.
>
> `void write(String str)` Writes a string.
>
> `void write(String str, int off, int len)` Writes a portion of a string.

con metodi che sono in grado di scrivere una stringa. Alcuni esempi sono

```
public void write(String str) throws IOException {
```

```
  write(str.toCharArray());
}
//-----------------------------------------------------------
public Writer append(CharSequence csq) throws IOException {
  write(csq.toString());
}
```

I metodi **append** hanno come valore di ritorno `Writer`: ciò fa sì che si possa fare **append** con con **chain invocation**, cioè

```
Writer writer = /* ... */;
writer
  .append("Lorem ipsum dolor sit amet, ")
  .append("consectetur adipiscing elit, ")
  .append("sed do eiusmod tempor incididunt ")
  .append("ut labore et dolore magna aliqua.");
```

il cui vantaggio è che non causa la creazione di un rettangolo per ogni concatenazione.

Esiste anche un writer per i file, la classe `FileWriter`:

> Writes text to character files using a default buffer size. Encoding from characters to bytes uses either a specified charset or the platform's default charset.
>
> Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one FileWriter (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.
>
> The FileWriter is meant for writing streams of characters. For writing streams of raw bytes, consider using a FileOutputStream.
>
> **`FileWriter(String fileName, Charset charset, boolean append)`** Constructs a FileWriter given a file name, charset and a boolean indicating whether to append the data written.

`FileWriter` non eredita da `Writer`, bensì da `OutputStreamWriter`.

```
Writer writer = new FileWriter("file.txt");
writer.write("Hello world!");
writer.close();
```

La classe `OutputStreamWriter`,

> An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.
>
> Each invocation of a write() method causes the encoding converter to be invoked on the given character(s). The resulting bytes are accumulated in a buffer before being written to the underlying output stream. Note that the characters passed to the write() methods are not buffered.
>
> **`OutputStreamWriter(OutputStream out)`** Creates an OutputStreamWriter that uses the default character encoding. **`OutputStreamWriter(OutputStream out, String charsetName)`** Creates an OutputStreamWriter that uses the named charset. **`OutputStreamWriter(OutputStream out, Charset cs)`** Creates an OutputStreamWriter that uses the given charset. **`OutputStreamWriter(OutputStream out, CharsetEncoder enc)`** Creates an OutputStreamWriter that uses the given charset encoder.

In parole povere, un `FileWriter` fa uso degli `OutputStreamWriter`, i quali fanno "da ponte" con i `FileOutputStream` convenzionali. La classe `OutputStreamWrit` fa due cose aggiuntive:

1. converte gli stream di caratteri in stream di byte;
2. fa uso di buffering.

```
OutputStream os = /* ... */;
Writer writer = new OutputStreamWriter(os); // si puo' utilizzare
    per scrivere chars
```

```
                                    // ad esempio dentro un
                                       socket
```

Esistono anche altre classi utili,

- `CharArrayWriter`, la quale implementa un buffer di caratteri che può essere adoperato esattamente come un `Writer`. I dati possono essere ottenuti invocando `toCharArray()`, oppure `toString()`;
- `StringWriter`, il quale raccoglie gli output in uno string buffer. Ha dunque una funzionalità simile a quella del `CharArrayWriter`.
- `BufferedWriter`, il quale estende la classe `Writer` e vi aggiunge il buffering:

  > The buffer size may be specified, or the default size may be accepted. The default is large enough for most purposes.
  >
  > A `newLine()` method is provided, which uses the platform's own notion of line separator as defined by the system property line.separator. Not all platforms use the newline character ('\n') to terminate lines. Calling this method to terminate each output line is therefore preferred to writing a newline character directly.
  >
  > In general, a `Writer` sends its output immediately to the underlying character or byte stream. Unless prompt output is required, it is advisable to wrap a `BufferedWriter` around any `Writer` whose `write()` operations may be costly, such as `FileWriters` and `OutputStreamWriters`.
  >
  > `BufferedWriter(Writer out)` Creates a buffered character-output stream that uses a default-sized output buffer.
  >
  > `BufferedWriter(Writer out, int sz)` Creates a new buffered character-output stream that uses an output buffer of the given size.

  e che può fare uso del metodo `flush()`.

12.1 Input con `Reader`

La documentazione della classe `Reader`,

> Abstract class for reading character streams. The only methods that a subclass must implement are `read(char[], int, int)` and `close()`. Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.
>
> `abstract void close()` Closes the stream and releases any system resources associated with it. `static Reader nullReader()` Returns a new Reader that reads no characters. `int read()` Reads a single character. `int read(char[] cbuf)` Reads characters into an array. `abstract int read(char[] cbuf, int off, int len)` Reads characters into a portion of an array.

e non si possono fare chain invocation, poiché ciascun `read` ha un valore di ritorno. Similmente, esistono `FileReader(File file, Charset charset)` per i file, `InputStreamReader(InputStream in, Charset cs)` per qualsiasi stream e `CharArrayReader(char[] buf)`, `StringReader(String s)`.

Di grande utilità è la classe `BufferedReader`, la quale legge sequenze di caratteri, array, e **linee**. In altre parole,

> `public String readLine() throws IOException`
>
> Reads a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), a carriage return followed immediately by a line feed, or by reaching the end-of-file (EOF).
>
> Returns: A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached without reading any characters

esiste questo metodo che legge una singola riga di testo, e restituisce `null` nel caso in cui l'end-of-stream sia stato raggiunto. Possiamo usare infatti il valore `null` per codificare la non esistenza di una stringa da leggere, un tipo non primitivo. Un utilizzo pratico è mostrato di seguito,

```
BufferedReader br = new BufferedReader(
  new InputStreamReader(new FileInputStream(fileName))
);
while (true) {
  String line = br.readLine();
  if (line == null) {
    break;
  }
  /* do things with the line */
}
br.close();
```

Capitolo 13

La stampa

Esiste la classe `PrintStream` che estende `OutputStream` e `FilterOutputStream`, la quale:

- aggiunge metodi per stampare qualsiasi tipo di oggetto mediante la sua **rappresentazione testuale** (mediante `toString()`); per i tipi primitivi, viene effettuata una conversione ragionevole in una stringa di testo;
- **non lancia mai eccezioni** - ciò è utile per rendere meno prolisso il codice. Tuttavia, l'utilizzo di questa classe potrebbe comportare problemi, nel senso che si potrebbe scrivere con un `PrintStream` su un altro output stream "delicato";
- con buffering, opzionalmente con *flush automatico*.

Un esempio di `PrintStream` è `System.out` - manca dunque una corrispondente classe per gli input. Le motivazioni dietro a ciò sono l'impossibilità di controllare l'input a differenza dell'output che è sotto il controllo del metodo. Per gli input, infatti, è importante gestire eccezioni, specialmente poiché si intende adoperare il valore di ritorno dei metodi di lettura (mentre ci può interessare meno della stampa dello schermo). Infatti, `System.in` è un puro `InputStream`, con nessun metodo aggiunto e con eccezioni.

Dalla documentazione,

> A PrintStream adds functionality to another output stream, namely the ability to print representations of various data values conveniently. Two other features are provided as well. Unlike other output streams, a PrintStream never throws an IOException; instead, exceptional situations merely set an internal flag that can be tested via the checkError method. Optionally, a PrintStream can be created so as to flush automatically; this means that the flush method is automatically invoked after a byte array is written, one of the println methods is invoked, or a newline character or byte ('\n') is written.
>
> All characters printed by a PrintStream are converted into bytes using the given encoding or charset, or platform's default character encoding if not specified. The PrintWriter class should be used in situations that require writing characters rather than bytes.
>
> **`PrintStream(OutputStream out, boolean autoFlush, Charset charset)`** Creates a new print stream, with the specified OutputStream, automatic line flushing and charset.

Alcuni fra i metodi disponibili,

> **`PrintStream append(char c)`** Appends the specified character to this output stream.
>
> **`PrintStream append(CharSequence csq)`** Appends the specified character sequence to this output stream.

`PrintStream append(CharSequence csq, int start, int end)` Appends a subsequence of the specified character sequence to this output stream.

`boolean checkError()` Flushes the stream and checks its error state.

`protected void clearError()` Clears the internal error state of this stream.

`PrintStream format(String format, Object... args)` Writes a formatted string to this output stream using the specified format string and arguments.

`PrintStream format(Locale l, String format, Object... args)` Writes a formatted string to this output stream using the specified format string and arguments.

`void print(long l)` Prints a long integer.

`void print(Object obj)` Prints an object.

`void print(String s)` Prints a string.

`PrintStream printf(String format, Object... args)` A convenience method to write a formatted string to this output stream using the specified format string and arguments.

Esiste anche un `PrintWriter`, che è un `PrintStream` costruito su di un `Writer`,

> Prints formatted representations of objects to a text-output stream. This class implements all of the print methods found in PrintStream. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.
>
> Unlike the PrintStream class, if automatic flushing is enabled it will be done only when one of the println, printf, or format methods is invoked, rather than whenever a newline character happens to be output. These methods use the platform's own notion of line separator rather than the newline character.
>
> Methods in this class never throw I/O exceptions, although some of its constructors may. The client may inquire as to whether any errors have occurred by invoking checkError().

il quale è tipicamente sfavorito in favore di `BufferedWriter`.

Capitolo 14

Socket input ed output

Un po' di background riguardo il **networking**,

- ciascun **host** ha un **indirizzo IP** univoco;
- ciascun **processo** su di un host ha un **numero di porta** univoco. Dunque, numero di porta e indirizzo IP servono ad identificare *univocamente* un endpoint; essi sono **indirizzi**: essi esistono a prescindere dall'esistenza fisica di una macchina con tali indirizzi IP e di porta;
- le **connessioni TCP** sono una coppia di "tubi" che trasferiscono byte – il primo dall'host A all'host B, il secondo viceversa. Esse danno 3 **garanzie**:
  1. ciascun byte inviato ad un endpoint viene correttamente consegnato all'altro endpoint;
  2. ciascun byte arriva al medesimo ordine nella sequenza con cui è stato inviato;
  3. non vi sono duplicazioni;
- se una delle tre garanzie non può essere rispettata, uno dei due endpoint può venirne a conoscenza ragionevolmente presto;
- un **Server** è un processo che attende una richiesta di connessione su una porta;
- un **Client** è un processo che richiede la connessione ad un server.

Tipicamente, una richiesta di connessione viene generata dal client e il server la accetta – a quel punto, entrambi intraprendono la comunicazione.

In Java, esiste la classe `InetAddress` che *modella un indirizzo IP*, e che eredita direttamente da `Object`,

> **`static InetAddress[] getAllByName(String host)`** Given the name of a host, returns an array of its IP addresses, based on the configured name service on the system.
>
> **`static InetAddress getByAddress(byte[] addr)`** Returns an InetAddress object given the raw IP address.
>
> **`static InetAddress getByAddress(String host, byte[] addr)`** Creates an InetAddress based on the provided host name and IP address.
>
> **`static InetAddress getByName(String host)`** Determines the IP address of a host, given the host's name.
>
> **`static InetAddress getLocalHost()`** Returns the address of the local host.

Per istanziare un oggetto di questa classe, si adoperano degli pseudo–costruttori come specificato sopra. Lo pseudo–costruttore

```
static InetAddress  getByAddress(byte[] addr)
```

riceve un vettore di byte come argomento (l'indirizzo IPv4 o IPv6). Lo pseudo–costruttore `getByName(String host)` è in grado di modellare un indirizzo IP a partire dal nome del dominio che si intende modellare.

La seconda astrazione è la classe `Socket`, anch'essa che eredita da `Object`. Un socket è un

> This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.
>
> `Socket()` Creates an unconnected Socket.
>
> `Socket(String host, int port)` Creates a stream socket and connects it to the specified port number on the named host.
>
> `Socket(InetAddress address, int port)` Creates a stream socket and connects it to the specified port number at the specified IP address.
>
> `void close()` Closes this socket.
>
> `OutputStream getOutputStream()` Returns an output stream for this socket.
>
> `InputStream getInputStream()` Returns an input stream for this socket.

dove il metodo `close()` chiude il socket. Un socket chiuso non può più essere riutilizzato per networking, riconnesso o ricollegato (bisogna crearne uno nuovo). Per adoperare il socket è necessario dapprima creare degli input stream e degli output stream a partire da esso, con gli ultimi due metodi elencati di sopra.

Un possible utilizzo dal lato client è il seguente,

```
Socket socket = new Socket("theserver.org", 10000);
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
/* do I/O things */
```

Dal lato server, bisogna usare la classe `ServerSocket` (da `Object`),

> This class implements server sockets. A server socket waits for requests to come in over the network.
>
> `ServerSocket(int port)` Creates a server socket, bound to the specified port.
>
> `Socket accept()` Listens for a connection to be made to this socket and accepts it.
>
> `void close()` Closes this socket.

con utilizzo

```
ServerSocket serverSocket = new ServerSocket(10000);
Socket socket = serverSocket.accept();
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
/* do I/O things */
```

Si osservi come nel socket del server non viene reso necessario fornire un hostname nel costruttore: il socket viene creato sulla macchina stessa!

Il metodo `accept()` blocca l'intera esecuzione del programma – fintanto che non viene fatta una richiesta di connessione, il programma è in attesa. Quando l'esecuzione viene ripresa, il metodo ritorna un **socket**, il socket del client richiedente la connessione.

Inoltre, se un server socket viene collegato ad un numero di porta, il sistema operativo bloccherà qualsiasi altro accesso a tale porta da parte di altri processi.

14.1 Un primo esempio: un server che trasforma le lettere in maiuscole

Solitamente, ci occupiamo di due aspetti legati al **protocollo**:

- qual'è la funzionalità offerta dal server?
- quale è il modo corretto per comunicare col server, per disporre della funzionalità da esso offerta?

Un esempio di protocollo potrebbe essere il seguente,

1. subito dopo la connessione, il client invia una linea di testo `l`;
2. il server risponde con `L`, una versione a lettere maiuscole di `l`;
3. se `L = BYE`, il server chiude la connessione; altrimenti, aspetta per la successiva linea di testo. Il programma, in questo caso, non termina: il server torna ad ascoltare per nuove connessioni.

Il server ascolta sulla porta 10000, gestisce un client alla volta al massimo, e non termina mai.

La realizzazione potrebbe essere la seguente,

```
public class SimpleUppercaserServer {
  private static final int PORT = 10000;
  private static final String QUIT_COMMAND = "BYE";
  public static void main(String[] args) throws IOException {
    ServerSocket serverSocket = new ServerSocket(PORT);
    while (true) {
      Socket socket = serverSocket.accept();
      BufferedReader br = new BufferedReader(
        new InputStreamReader(socket.getInputStream())
      );
      BufferedWriter bw = new BufferedWriter(
        new OutputStreamWriter(socket.getOutputStream())
      );
      while (true) {
        String line = br.readLine();
        bw.write(line.toUpperCase() + System.lineSeparator());
        bw.flush();
        if (line.toUpperCase().equals(QUIT_COMMAND)) {
          break;
        }
      }
      socket.close(); // libera le risorse del sistema operativo
    }
  }
}
```

I **BufferedReader** e **BufferedWriter** sono consigliati per i dispositivi fisici, in particolare per il protocollo TCP. In questo caso, useremo i reader ed i writer poiché si tratta di un protocollo che funziona mediante caratteri di testo.

Inoltre, chiudere il socket equivale anche a chiudere i suoi **InputStream** ed **OutputStream**. Si rende necessario qualora si debba liberare risorse per il sistema operativo (mentre invece i **BufferedReader** e simili non occupano direttamente risorse *proprie del sistema operativo*).

Per il client,

```
public class LineClient {
  public static void main(String[] args) throws IOException {
    InetAddress serverInetAddress;
    if (args.length > 0) {
      serverInetAddress = InetAddress.getByName(args[0]);
    } else {
      serverInetAddress = InetAddress.getLocalHost();
    }
    Socket socket = new Socket(serverInetAddress, 10000);
    BufferedReader br = new BufferedReader(
      new InputStreamReader(socket.getInputStream())
    );
    BufferedWriter bw = new BufferedWriter(
```

```
      new OutputStreamWriter(socket.getOutputStream())
    );
    for (int i = 0; i < 10; i++) {
      String sent = String.format("Hello world n. %d!", i);
      bw.write(sent + System.lineSeparator());
      bw.flush();
      String received = br.readLine();
      System.out.printf("Sent: %s%nReceived: %s%n",
          sent, received
      );
    }
    bw.write("bye" + System.lineSeparator());
    bw.flush();
    socket.close();
  }
}
```

14.2 A more general example

Il protocollo:

- client sends line `l`;
- if `l = lquit`, server closes connection; otherwise it processes line `l = p(l)`.

The server:

- listens on port `np`;
- handles 1 client at a time;
- never terminates;
- it is **designed** to be extended;

- p : String $\rightarrow$ String, l_{quit}, port number is provided in parameters.

Il codice,

```
public class SimpleLineProcessingServer {
  private final int port;
  private final String quitCommand;
  public SimpleLineProcessingServer(int port, String quitCommand) {
    this.port = port;
    this.quitCommand = quitCommand;
  }
  public void run() throws IOException {
    ServerSocket serverSocket = new ServerSocket(port);
    while (true) {
      Socket socket = serverSocket.accept();
      BufferedReader br = new BufferedReader(new
          InputStreamReader(socket.getInputStream()));
      BufferedWriter bw = new BufferedWriter(new
          OutputStreamWriter(socket.getOutputStream()));
      while (true) {
        String line = br.readLine();
        if (line.equals(quitCommand)) {
          break;
        }
        bw.write(process(line) + System.lineSeparator());
        bw.flush();
      }
      socket.close();
    }
  }
  protected String process(String input) {
    return input;
  }
}
```

Il metodo deve essere **inviato come parametro** (in maniera simile ai *valori–funzione*). Per fare ciò, si valuta p facendo l'override di `process()` mentre si estende `SimpleLineProcessingServer`:

```
protected String process(String input) {
  return input.toUpperCase();
}
```

14.2.1 Server Logging

Un **registro degli avvenimenti** o **log** è molto utile, specialmente nel server. Esso è composto dai *log entries*, ad esempio

```
eric@cpu:~$ java SimpleLineProcessingServer 10000 bye
[2020-04-30 18:17:54] Connection from /127.0.0.1.
[2020-04-30 18:18:06] Disconnection of /127.0.0.1 after 2 requests.
```

14.2.2 Costruttori e campi dell'esempio

Come campi, adoperiamo un numero di porta, un comando d'uscita (una stringa) ed un `PrintStream`. Quest'ultimo è adoperato per tre ragioni,

- facciamo logging, sappiamo che scriveremo solo stringhe di testo;
- fa del buffering;
- non ci preoccupiamo degli errori nei log.

```
public class SimpleLineProcessingServer {
  private final int port;
  private final String quitCommand;
  private final PrintStream ps;
  public SimpleLineProcessingServer(int port, String quitCommand,
      OutputStream os) {
    this.port = port;
    this.quitCommand = quitCommand;
    ps = new PrintStream(os);
  }
  /* ... */
  protected String process(String input) {
    return input;
  }
}
```

Si è soliti, inoltre, dividere il `run()` in più parti, ovvero `run()` e `handleClient()`:

```
public void run() throws IOException {
  ServerSocket serverSocket = new ServerSocket(port);
  while (true) {
    Socket socket = serverSocket.accept();
    handleClient(socket);
  }
}
protected void handleClient(Socket socket) throws IOException {
  ps.printf("[%1$tY-%1$tm-%1$td %1$tT] Connection from %2$s.%n",
      System.currentTimeMillis(), socket.getInetAddress());
  BufferedReader br = new BufferedReader(new
      InputStreamReader(socket.getInputStream()));
  BufferedWriter bw = new BufferedWriter(new
      OutputStreamWriter(socket.getOutputStream()));
  int requestsCounter = 0;
  while (true) {
    String line = br.readLine();
    if (line.equals(quitCommand)) {
      break;
    }
```

```
    bw.write(process(line) + System.lineSeparator());
    bw.flush();
    requestsCounter = requestsCounter + 1;
  }
  socket.close();
  ps.printf("[%1$tY-%1$tm-%1$td %1$tT] Disconnection of %2$s after
      %3$d requests.%n", System.currentTimeMillis(),
      socket.getInetAddress(), requestsCounter);
}
```

Un'altro metodo utile potrebbe essere un `private void log(String)`, per stampare in maniera agevole i log.

Alla riga del printf,

```
ps.printf("[%1$tY-%1$tm-%1$td %1$tT] Connection from %2$s.%n",
    System.currentTimeMillis(), socket.getInetAddress());
```

si nota:

- `%1$` – prendiamo il primo argomento. Per il secondo, si usa `%2$`;
- `tY` – consideriamo il l'oggetto come una data, e ne stampiamo l'**anno**; e così via.

14.3 Un vector processing server

Supponiamo di voler costruire un *vector processing server* che avrà un protocollo del genere,

- il client invia un vettore di numeri reali;
- se $|\mathbf{v}| = \mathbf{0}$ il server chiude la connessione, altrimenti invia come risposta la linea processata $\mathbf{v}' = \mathbf{p}(\mathbf{v})$.

Dettagli del protocollo sono che per inviare un vettore di numeri reali, il client dapprima invia un intero $n = |\mathbf{v}|$ e invia successivamente n doubles da 8 byte l'uno. Il server ascolta sulla porta 10000, gestisce un client alla volta, non termina mai. Possiamo ad esempio estendere `SimpleLineProcessingServer`,

```
protected void handleClient(Socket socket) throws IOException {
  ps.printf("[%1$tY-%1$tm-%1$td %1$tT] Connection from %2$s.%n",
      System.currentTimeMillis(), socket.getInetAddress());
  int requestsCounter = 0;
  DataInputStream dis = new DataInputStream(new
      BufferedInputStream(socket.getInputStream()));
  DataOutputStream dos = new DataOutputStream(new
      BufferedOutputStream(socket.getOutputStream()));
  while (true) {
    double[] input = readVector(dis);
    if (input.length == 0) {
      break;
    }
    writeVector(dos, process(input));
    requestsCounter = requestsCounter + 1;
  }
  socket.close();
  ps.printf("[%1$tY-%1$tm-%1$td %1$tT] Disconnection of %2$s after
      %3$d requests.%n", System.currentTimeMillis(),
      socket.getInetAddress(), requestsCounter);
}

private double[] readVector(DataInputStream dis) throws IOException
    {
  int length = dis.readInt();
  double[] values = new double[length];
  for (int i = 0; i<values.length; i++) {
    values[i] = dis.readDouble();
  }
```

```
  return values;
}

private void writeVector(DataOutputStream dos, double[] values)
    throws IOException {
  dos.writeInt(values.length);
  for (double value : values) {
    dos.writeDouble(value);
  }
  dos.flush();
}
protected double[] process(double[] input) {
  return input;
}
```

Un'alterantiva potrebbe essere quella di definire **EnhancedDataInputStream** ed **EnhancedDataOutputStrean**, con metodi del tipo

```
double[] readDoubleArray()
void writeDoubleArray(double[])
```

14.4 TCP: un protocollo non orientato ai messaggi

Diversamente dai protocolli precedenti che erano **orientati ai messaggi** (la comunicazione avviene con lo scambio di messaggi, la *richiesta* e la *risposta*; tipicamente non c'è bisogno di suddividere in ulteriori ontologie), TCP non è un protocollo orientato ai messaggi. Il protocollo TCP realizza infatti un *reliable byte stream* (con tutto ciò che ne consegue), ma non garantisce che un **write()** di m byte in un endpoint corrisponda esattamente ad un **read()** di m bytes in altri endpoint.

In pratica, $n \geq 1$ **read** possono essere necessarie per leggere gli m byte, in altre parole, le **read** possono essere in maggior numero. L'ultima **read** potrebbe, ad esempio, includere altri byte oltre agli m. di "messaggio".

Assumendo che **oBuffer** sia di lunghezza l inferiore a 1024,

```
byte[] oBuffer = /* ... */
os.write(oBuffer);
```

il server S connette a C,

```
byte[] iBuffer = new byte[1024];
int n = is.read(iBuffer);
```

con n che è un valore compreso fra 1 ed l. **iBuffer** potrebbe contenere solo una porzione (quella all'inizio) di quello che conteneva **oBuffer**. In altre parole, TCP non può essere pensato come un protocollo a messaggi.

Supponendo che l_1 ed l_2 siano le lunghezze di **oBuffer1** ed **oBuffer2**,

```
os.write(oBuffer1);
os.write(oBuffer2);

int n1 = is.read(iBuffer1);
int n2 = is.read(iBuffer2);
```

i possibili risultati:

- n1 $= l_1$, n2 $= l_2$ (fortuna sfacciata!);
- n1 $\leq l_1$, n2 $\leq l_2$;
- n1 $> l_1$, n2 $l_2 - (\text{n1} - l_1)$.

TCP modella un *flusso di byte*: per questa ragione, abbiamo bisogno di un protocollo che prenda un `byte[]`, risultato della concatenazione di 1 + `read()` e lo divida in messaggi. Il protocollo può essere realizzato in varie maniere:

- per linee di testo: `BufferedReader` legge tanti byte quanti sono necessari fino ad arrivare al carattere *new line*;
- per un double array: i primi 4 byte (`DataInputStream`) specificano quanti byte devono essere letti, come l'esempio di sopra;
- l'HTTP termina con due linee vuote.

Capitolo 15

Programmazione multithreading

Sia il seguente frammento,

```
public void run() throws IOException {
  ServerSocket serverSocket = new ServerSocket(port);
  while (true) {
    Socket socket = serverSocket.accept();
    handleClient(socket);
  }
}
```

Le limitazioni sono che `accept()` blocca l'intera applicazione finché un client non invia una richiesta, e `handleClient()` fa ritorno solo al completamento della gestione del client: il server può gestire al massimo 1 client alla volta. In altre parole, in questo modo il server **serializza** le richieste di rete, non le **parallelizza**. Un eventuale client che si connette al secondo posto mentre un primo client è servito viene accolto dal sistema operativo, e tutte le richieste vengono eseguite nel momento in cui il primo client è stato servito completamente e la connessione è stata chiusa. Quello che desideriamo è un server che possa gestire *molti* client alla volta, qualcosa come $1 + n$ **processi concorrenti**:

- il primo, sempre in attesa di richieste di connessione;
- i successivi n, uno per ogni client che ha fatto la richiesta (ciascuno esegue `handleClient()`;

Risolvere questa limitazione è possibile tramite l'utilizzo dei **thread**, dei **flussi di esecuzione paralleli**. La JVM:

- la JVM esegue più di un thread alla volta;
- un thread può essere *bloccato* (ad esempio, quando aspetta per input) mentre gli altri continuano ad eseguire – questo accade nel caso dell'`accept()`, ad esempio.

Dal punto di vista di Java, esistono solo thread (non *processi* di Java Virtual Machine), anche se ci possono essere più JVM in esecuzione (un processo per ciascuna).

I processi sono i flussi di esecuzione paralleli gestiti dal sistema operativo, mentre i thread sono i flussi concorrenti gestiti dalla JVM. Ambedue la JVM e il sistema operativo sono stati realizzati in modo da sfruttare la parallelizzazione a livello di hardware; in alternativa, viene fatto il *time–sharing* delle risorse della CPU.

15.0.1 La classe `Thread`

La classe `Thread`, che eredita da Object, è siffatta:

> A *thread* is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.
>
> There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started.
>
> `void start()` Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
>
> `void run()` If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.

Vi sono due modi per creare un flusso d'esecuzione: il primo, è quello di creare una classe `T` che *estende* `Thread` e fa l'override di `run()`, ed invocare `T.start()` che a sua volta invocherà `run()`, il secondo non verrà visto. Il punto dove il flusso d'esecuzione del thread viene fatto partire è alla chiamata del metodo `start()`.

Ad esempio, una classe `SlowCounter`

```
public class SlowCounter extends Thread {
  public void run() {
    for (int i = 0; i < 10000; i++) {
      if (i % 1000 == 0) {
        System.out.printf("%s: %2d%n", toString(), i / 1000);
      }
    }
  }
}
```

si invoca così:

```
SlowCounter c1 = new SlowCounter();
SlowCounter c2 = new SlowCounter();
c1.start();
c2.start();
```

Questa scrive sullo schermo:

```
Thread[Thread-0,5,main]: 0
Thread[Thread-1,5,main]: 0
Thread[Thread-0,5,main]: 1
Thread[Thread-1,5,main]: 1
Thread[Thread-0,5,main]: 2
Thread[Thread-1,5,main]: 2
Thread[Thread-0,5,main]: 3
Thread[Thread-1,5,main]: 3
Thread[Thread-0,5,main]: 4
Thread[Thread-1,5,main]: 4
Thread[Thread-0,5,main]: 5
...
Thread[Thread-0,5,main]: 9
Thread[Thread-1,5,main]: 9
```

Si osservi che viene adoperato il `toString()` di `Thread`. Invocando `start()` si ottiene subito il suo ritorno (**ritorna immediatamente**), ma l'effetto è quello di creare un nuovo thread di esecuzione, eseguendo `run()`. La JVM, prima di spegnersi al termine del programma principale, aspetterà il termine dell'esecuzione di tutti i thread di esecuzione lasciati aperti.

Una differenza importante, è che il codice seguente

```
SlowCounter c1 = new SlowCounter();
SlowCounter c2 = new SlowCounter();
```

```
c1.run();
c2.run();
```

non causa la partenza di nuovi thread! È necessario l'uso del metodo `start()` per la creazione di nuovi thread di esecuzione.

15.0.2 I thread e la memoria

Ciascun thread possiede **il suo stack**, ma **condivide lo heap con tutti gli altri thread**. Possiamo vedere del codice con il suo diagramma (slide 403 (attenzione, dentro lo stack T0 c'è anche `args`))

```
class SlowCounter extends Thread {
  public void run() {
    PrintStream ps = System.out;
    for (int i = 0; i < 10; i++) {
      ps.println(i);
    }
  }
}

public static void main(String[] args) {
  SlowCounter c1 = new SlowCounter();
  SlowCounter c2 = new SlowCounter();
  c1.start();
  c2.start();
}
```

In altre parole, siccome ciascuno dei thread condivide lo heap, è possibile *condividere informazioni* fra thread, poiché i riferimenti possono far riferimento allo stesso oggetto contenuto nello heap.

Quando dei thread hanno la necessità di scambiare le informazioni con altri thread, siano essi thread principali o altri thread, ciò deve avvenire tramite **riferimenti condivisi**. La buona pratica è quella di **impostarli come field**, poiché non possono essere passati come argomento. Il costruttore della classe che estende `Thread` dovrà occuparsi di passare tutti e soli i field che devono essere condivisi. Ad esempio:

```
public class LineProcessingServer {
  private final int port;
  private final String quitCommand;
  public LineProcessingServer(int port, String quitCommand) {
    this.port = port;
    this.quitCommand = quitCommand;
  }
  public void run() throws IOException {
    ServerSocket serverSocket = new ServerSocket(port);
    while (true) {
      Socket socket = serverSocket.accept();
      ClientHandler clientHandler = new ClientHandler(socket,
          quitCommand);
      clientHandler.start();
    }
  }
}
```

Ora estendiamo il `Thread` tramite `ClientHandler`,

```
public class ClientHandler extends Thread {
    private final Socket socket;
    private final String quitCommand;
    public ClientHandler(Socket socket, String quitCommand) {
        this.socket = socket;
        this.quitCommand = quitCommand;
```

```
    }
    protected String process(String input) {
        return input;
    }
    // cannot add throws IOException
    // because otherwise inheritance would be impossible
    // all methods using run() wouldn't work!
    public void run() {
    try {
        BufferedReader br = new BufferedReader(new
            InputStreamReader(socket.getInputStream()));
        BufferedWriter bw = new BufferedWriter(new
            OutputStreamWriter(socket.getOutputStream()));
        while (true) {
        String line = br.readLine();
        if (line.equals(quitCommand)) {
            socket.close();
            break;
        }
        bw.write(process(line) + System.lineSeparator());
        bw.flush();
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        try {
        socket.close();
        } catch (IOException e) {
        e.printStackTrace();
        }
    }
    }
}
```

L'utilizzo della coppia di classi soprastante è il seguente:

- si estende `ClientHandler` con la classe `H` e si fa l'override di `process()`;
- si estende con `S` o la si crea simile a `LineProcessingServer` e istanzia `H` tramite `S`.

Questa maniera non è particolarmente comoda: bisogna definire due classi sempre, con piccole modifiche.

15.0.3 L'esecuzione concorrente

Lo **stesso metodo** dello **stesso oggetto** può essere eseguito allo stesso tempo da thread different. L'idea è la seguente:

- una classe server `S` specifica i dettagli applicativi, come processare una richiesta e quale è il comando di uscita (quit command);
- il `ClientHandler` sa solanto come gestire un client – ad esempio, gestendo l'interazione fra richiesta e risposta, delegando ad `S` il comportamento tipico dei server (gestione di più client). Il `ClientHander` avrà un riferimento alla classe `S`, **così che si possa invocare `S.process()`, definita dentro `S`**.

```
public class LineProcessingServer {
  private final int port;
  private final String quitCommand;
  public LineProcessingServer(int port, String quitCommand) {
    this.port = port;
    this.quitCommand = quitCommand;
  }
  public void run() throws IOException {
    ServerSocket serverSocket = new ServerSocket(port);
    while (true) {
      Socket socket = serverSocket.accept();
```

```
      ClientHandler clientHandler = new ClientHandler(socket, this);
      clientHandler.start();
    }
  }
  public String process(String input) {
    return input;
  }
  public String getQuitCommand() {
    return quitCommand;
  }
}

public class ClientHandler extends Thread {
  private final Socket socket;
  private final LineProcessingServer server;
  public ClientHandler(Socket socket, LineProcessingServer server) {
    this.socket = socket;
    this.server = server;
  }
  public void run() {
    try {
      BufferedReader br = new BufferedReader(new
          InputStreamReader(socket.getInputStream()));
      BufferedWriter bw = new BufferedWriter(new
          OutputStreamWriter(socket.getOutputStream()));
      while (true) {
        String line = br.readLine();
        if (line.equals(server.getQuitCommand())) {
          socket.close();
          break;
        }
        bw.write(server.process(line) + System.lineSeparator());
        bw.flush();
      }
    } catch (IOException e) { /* ... */
    } finally { /* ... */ }
  }
}
```

In altre parole, anche se i due thread eseguono le istruzioni `server.process(line)`, lo fanno con uno stack diverso (non con uno heap diverso però) e non presentano fra loro interferenza per questi motivi.

Il design ora è più elegante: è possibile definire soltanto `S` che estende `LineProcessingServer` e fa l'override di `process()`, con `ClientHandler` che gestisce un client, ma che non presenta customizzazioni ulteriori (`process()` non è ridefinito lì).

Al runtime, più thread potranno eseguire `process()` della classe server. Questo, a seconda dei casi, può rappresentare un problema se non gestito opportunamente.

15.0.4 Thread scheduling

La JVM conosce l'insieme T dei thread esistenti, e a quale **istruzione bytecode** essi si trovano. Conosce inoltre il sottoinsieme $T' \subseteq T$ dei thread non in stato di blocco e conosce anche l'insieme C dei core disponibili, hardware o virtuali, forniti dal sistema operativo.

A ciascun intervallo di tempo, per ciascun core C, la JVM seleziona un thread T' ed esegue la successiva istruzione bytecode (insomma, procede con l'esecuzione di uno fra i thread disponibili).

Tuttavia, i thread sono in esecuzione che **solo in apparenza è concorrente**. Infatti, supponiamo di essere dotati di un singolo core ($|C| = 1$), e avere due thread con istruzioni successive,

- t_1 esegue a_1, b_1, c_1;
- t_2 esegue a_2, b_2, c_2;

alcune esecuzioni possibili possono essere in qualsiasi ordine fra i processi, ad esempio

- a1, b1, c1, a2, b2, c2;
- a1, a2, b1, b2, c1, c2;
- a2, b2, a1, c2, b1, c1;

e così via. (Non sono possibili esecuzioni come a2, b1, a1, c2, b2, c1, ad eccezione di casi rarissimi in cui la CPU direttamente esegue prima operazioni successive poiché lo ritiene più conveniente per qualche ragione).

Una conseguenza in Java è che, il seguente statement

```
System.out.printf("a+b=%d%n", a + b);
```

corrisponde a tantissime istruzioni in bytecode. In un'esecuzione multithread non è detto che tutte queste esecuzioni **accadano di fila**. L'esatta sequenza di esecuzione **non è predicibile**, e la causa di tanti bug **non è sempre riproducibile**. Il codice diventa **non deterministico**!

Supponiamo di avere due classi siffatte,

```
public class Counter {
  private int c = 0;
  public int incAndGet() {
    c = c + 1;
    return c;
  }
  public int decAndGet() {
    c = c - 1;
    return c;
  }
}
```

```
Counter c = new Counter();
(new IncThread(c)).start();
(new DecThread(c)).start();

/////
public class IncThread extends Thread {
  private final Counter c;
  public IncThread(Counter c) {
    this.c = c;
  }
  public void run() {
    System.out.print(c.incAndGet() + " ");
  }
}

public class DecThread extends Thread {
  /* ... */
}
```

I possibili risultati sono:

- `1 0` se siamo fortunati;
- `0 0`;
- `-1 0`.

15.0.5 Il modificatore `synchronized`

La soluzione al problema dell'interferenza di thread è la parola chiave **synchronized**.

Tutti i metodi definiti con il modificatore `synchronized` non possono essere eseguiti *nello stesso oggetto con più di un thread alla volta*. Oltre a ciò, quando

un oggetto esegue un metodo `synchronized` nessuna istanza dello stesso oggetto può invocare *alcun* altro metodo `synchronized`, oltre che al metodo stesso.

I modificatori `synchronized` e `static` non possono essere messi assieme (o comunque, non ha effetto), anche poiché non c'è alcuna istanza esistente dell'oggetto nel caso di metodi `static`.

Dunque, l'esecuzione di tale metodo è detta essere **atomica**. Attenzione però: l'atomicità è garantita **solo** nel metodo `synchronized`, non nell'ipotetica sequenza di metodi (risolubile con un unico metodo `synchronized` contenente la sequenza).

Vediamo ora del codice,

```
public class AtomicCounter extends Coutner {
  private int c = 0;
  public synchronized int incAndGet() {
    c = c + 1;
    return c;
  }
  public synchronized int decAndGet() {
    c = c - 1;
    return c;
  }
}

Counter c = new AtomicCounter();
(new IncThread(c)).start();
(new DecThread(c)).start();

/////
public class IncThread extends Thread {
  private final Counter c;
  public IncThread(Counter c) {
    this.c = c;
  }
  public void run() {
    System.out.print(c.inc() + " ");
  }
}

public class DecThread extends Thread {
  /* ... */
}
```

i cui possibili output sono `1 0` e `-1 0` – non è più possibile l'output `0 0`.

15.0.6 Le classi *Thread–safe*

L'accezione **non thread–safe** indica tutti i metodi o classi per cui il risultato nel caso di utilizzo in thread multipli può non essere specificato o potrebbe causare eccezioni. Dal momento che l'atomicità è un requisito comune, il JDK fornisce classi e metodi `thread-safe`, controparti di classi e metodi importanti (ad esempio `AtomicInteger`, la quale è *mutabile* e `thread-safe`), e maniere per rendere `thread-safe` alcune classi non thread–safe, ad esempio `Collections.synchronizedCollection()`.

Il modificatore `synchronized` può essere collocato anche in **blocchi di codice**. La sintassi è la seguente,

```
public class CounterThread extends Thread {
  public void run() {
    Counter c = /* Comes from somewhere... */
    for (int i = 0; i < 10; i++) {
      synchronized (c) { // reference to non-prymitive type
        System.out.print(c.get() + " -> ");
```

```
        c.inc();
        System.out.println(c.get());
      }
    }
  }
}
```

dove indichiamo che vogliamo mettere un *lock* sull'oggetto `c`. In questo caso, il contatore può essere utilizzato *da al massimo un thread alla volta*.

Capitolo 16

Le eccezioni in Java

La vecchia maniera di gestire gli errori era che ogni software gestiva una *propria convenzione* per gli errori – non esisteva una nozione di "errore" nel linguaggio, e ogni software era destinato a **gestire le eccezioni autonomamente**. In alcuni casi, l'input per cui l'output dovrebbe essere prodotto non è valido o accettabile per varie ragioni (si pensi alla divisione per zero).

Ad esempio, una funzione doveva dapprima *rilevare un errore o un'anomalia*, e poi *propagare l'informazione* attraverso un **valore di ritorno speciale**.

Ad esempio, in linguaggio C:

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE
    *stream);
```

> Description:
>
> This function writes `size * nmemb` bytes from ptr to stream.
>
> Return value:
>
> The number of items of size `nmemb` written **or -1 on error**.

Il ritorno "or -1 on error" è una convenzione! Non è rappresentato in alcuna maniera "nel linguaggio".

Sempre in C,

```
Syntax:

#include <stdlib.h>
int atoi(const char *str);
```

> Description:
>
> Converts as much of the string as possible to an equivalent integer value.
>
> Return value:
>
> The equivalent value, **or zero if the string does not represent a number**.

Nell'esempio `atoi` una stringa `"0"` fornisce un output equivalente al codice d'errore. Il chiamante, in ogni caso, deve controllare gli errori. Questo potrebbe non avvenire:

1. il chiamante deve *rilevare* l'errore;
2. il chiamante deve poi *gestire* l'errore.

Il codice chiamato, in C, tende soltanto a *rilevare* l'errore – lo sviluppatore attraverso il chiamante **potrebbe non occuparsi delle condizioni d'errore del chiamato**.

Il linguaggio Java **obbliga sempre** lo sviluppatore a gestire l'errore, tranne in casi triviali oppure di notevole ricorrenza. Ogni metodo in Java deve, almeno, *riscontrare* l'errore (in inglese, *catch*); opzionalmente può gestirlo. Qualora nessun metodo gestisca l'errore, il programma **si blocca**.

Il linguaggio Java, dunque, si comporta così:

- considera l'errore come un concetto chiave;
- richiede di definire due tipi di flussi d'esecuzione nel codice sorgente:
    1. flusso **normale** quando non avviene alcun errore;
    2. flusso **anomalo** quando avviene un errore – quando si è in condizione d'anomalia, c'è sempre stato un errore.
- lo sviluppatore deve adoperare il flusso anonimo per **gestire l'errore correttamente**;
- il compilatore notifica lo sviluppatore se egli—ella si dimenticano di definire il flusso anomalo quando necessario.

Supponiamo un metodo `m` chiamato da `mc`, un metodo chiamante. Se `m` ha un flusso anomalo, l'anomalia viene risolta, ritorna, ed il chiamante `mc` prosegue in flusso normale. Se invece `m` ha un flusso anomalo e non sa come gestirlo, fa immediatamente ritorno e `mc` deve gestire il flusso anomalo.

Se nessun metodo sa come gestire l'anomalia, **l'esecuzione si ferma immediatamente**. Questo approccio fa sì che non sia mai possibile che un programma continui ad eseguire in flusso anomalo.

16.1 Gli errori in Java: `try-catch`, `throws`, `throw`

Esiste la classe `Exception`,

> Class Exception
>
> java.lang.Object java.lang.Throwable java.lang.Exception
>
> The class `Exception` and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch.

In Java, il concetto chiave d'errore è modellato dalla classe `Exception`, fa parte del pacchetto `java.lang` (non serve importarla) ed eredita da `Throwable`. Gli errori sono implicitamente definiti come le *condizioni che un'applicazione ragionevole vorrebbe catturare, recuperare e gestire*. In altre parole, sono **eventi anomali** che hanno come risultato una condizione che dovrebbe essere presa in considerazione con cura – dunque, **eccezioni** alla "normalità". La classe può (e deve) essere estesa, per rappresentare i tipi specifici di eccezione.

Alcune sottoclassi di `Exception`:

> Package java.net
>
> Class UnknownHostException
>
> java.lang.Object java.lang.Throwable java.lang.Exception java.io.IOException java.net.UnknownHostException
>
> Thrown to indicate that the IP address of a host could not be determined.
>
> Package javax.sound.sampled
>
> Class UnsupportedAudioFileException
>
> java.lang.Object java.lang.Throwable java.lang.Exception javax.sound.sampled.UnsupportedAudioF
>
> An UnsupportedAudioFileException is an exception indicating that an operation failed because a file did not contain valid data of a recognized file type and format.

Vediamo i costruttori della classe `Exception`,

> `Exception()` Constructs a new exception with null as its detail message.

`Exception(String message)` Constructs a new exception with the specified detail message.

`Exception(String message, Throwable cause)` Constructs a new exception with the specified detail message and cause.

In Java, la divisione dei flussi normale—anomalo si fa con il costrutto **try—catch**. Ad esempio, nel seguente metodo,

```
doThings() {
  try {
    /* N */
  } catch (UnknownHostException e) {
    /* A1 */
  } catch (InvalidTypeException e) {
    /* A2 */
  }
}
```

dove `e` è un identificatore di oggetto che estende `Throwable`, la parte `N` è il flusso normale, `A1` è il flusso anomalo se `UnknownHostException` viene lanciato, e `A2` è il flusso anomalo se invece viene lanciata `InvalidTypeException`. In altre parole, se l'eccezione `e` di tipo `E` viene a cadere durante l'esecuzione del flusso normale, la JVM controlla *in ordine* se il blocco esiste per `E`:

- se trovato, l'esecuzione prosegue nel primo statement del blocco `catch` corrispondente ed `e` farà riferimento all'oggetto *e* di tipo `E`;
- altrimenti, l'esecuzione torna a `doThings()`, che dovrà gestire l'eccezione `e`.

Dato uno statement `s`, il compilatore saprà per filo e per segno (lo verifica in compilazione) se esso può generare o meno una o più eccezioni ed il loro tipo. Inoltre, compila un blocco di codice `C = (s1, s2,...)` soltanto se:

- **nessuna eccezione** può essere generata per nessuno degli statement in `C`;
- `C` è in un **blocco `try`**, per cui esiste un `catch` relativo ad ogni possibile eccezione generata da `C`, mediante ereditarietà;
- lo sviluppatore ha **esplicitamente definito il metodo** che contiene `C` come un metodo che può generare le eccezioni possibili generate da `C` (l'**eccezione è stata dichiarata**).

Per la terza maniera, si usa la parola chiave `throws`:

```
public String getLastName(String fullName) throws
    MalformedNameException {
  /* ... */
}

// con piu eccezioni
public void doHardJob() throws TooHardException, TooLazyException {
  /* ... */
}
```

Si dice che lo sviluppatore può fare il `throw` di un'eccezione di tipo `MalformedNameExce` quest'ultima detta **`throws` clause** (clausola throws). Questa maniera risolve due problemi: in primis, dichiariamo la necessità di gestire un'eccezione, e al contempo permette al compilatore – soltanto guardando la signature del metodo – di comprendere le eccezioni che può lanciare tale metodo.

La clausola throws fa parte della signature del metodo. Se una classe `C'` estende `C` con un metodo `m`, `C'` non può *aumentare* la clausola di `m`:

```
public class Worker {
  void work() {
    /* ... */
```

```
  }
}

//Does not compile
public class LazyWorker extends Worker {
  void work() throws TooHardException {
    /* ... */
  }
}
```

Il compilatore non ce lo lascia fare, poiché del codice esistente potrebbe funzionare con `Worker.work()` senza eccezioni che improvvisamente si potrebbe ritrovare con nuove eccezioni dettate da `LazyWorker.work()`. Le eccezioni in clausola possono, invece, essere **decrementate**, ad esempio:

```
public class Worker {
  void work() throws TooHardException {
    /* ... */
  }
}

public class TirelessWorker extends Worker {
  void work() {
    /* ... */
  }
}
```

Il viceversa funziona: l'importante è non introdurre **nuove** eccezioni nel flusso d'esecuzione di un metodo quando si eredita.

Si possono *cambiare* le clausole di throw, a patto di scegliere una classe di eccezioni che **estende** la precedente:

```
public class Worker {
  void work() throws Exception {
    /* ... */
  }
}

public class PreciseWorker extends Worker {
  void work() throws ForbiddenByUnionsException {
    /* ... */
  }
}
```

Lanciare un'eccezione si fa con lo statement `throw`. In particolare, lo si fa in maniera simile:

```
public String getLastName(String fullName) throws
    MalformedNameException {
  String[] pieces = fullName.split(" ");
  if (pieces.length == 0) {
    throw new MalformedNameException("Empty name!");
  }
  return pieces[pieces.length-1];
}
```

e deve avere un oggetto di tipo `Throwable` (anche invocato ad hoc con `new`). Nello specifico esempio, non è presente il costrutto `try-catch`, e viene immediatamente terminato il metodo, restituendo l'eccezione al metodo chiamante e portandolo in flusso anomalo. Il `throw` corrisponde dunque ad una sorta di "accensione" della **modalità anomala**.

Le differenze fra `throw` e `throws`:

- `throw` è un imperativo: lancia *ora* un'eccezione, sta in uno statement;

- `throws` è una definizione: *il metodo può farlo*.

Lo statement `throw` manda dunque in flusso anomalo l'esecuzione, e specifica **in quale anomalia** tale flusso si troverà.

Il compilatore effettua anche altre verifiche statiche in fase di compilazione. Il compilatore controlla se l'eccezione è assente, catturata o dichiarata. Inoltre, considera:

- l'**effettiva utilità** della cattura o della dichiarazione – non si può evitare di gestire il problema in qualche maniera;
- l'**ereditarietà** delle classi che estendono `Exception`s.

Ad esempio, il seguente codice non compila,

```
public String getLastName(String fullName) throws
    MalformedNameException {
  /* ... */
}

public void showName(String fullName) {
  String lastName = getLastName(fullName);
  System.out.println(lastName);
}
```

Il problema qua è che `getLastName` può lanciare `MalformedNameException`, ma nel codice che lo chiama non c'è alcun costrutto `try-catch` o nessun `throws`.

Altre varianti: dichiarato ma non lanciato,

```
public void showName(String fullName) throws MalformedNameException
    {
  System.out.println(fullName);
}
```

Compila, però l'IDE tipicamente ci avverte che il codice, di fatto, non lancia mai `MalformedNameException`. Altro,

```
public void showName(String fullName) throws Exception,
    MalformedNameException {
  String[] pieces = fullName.split(" ");
  if (pieces.length == 0) {
    throw new MalformedNameException("Empty name!");
  }
  return pieces[pieces.length-1];
}
```

il compilatore compila, però l'IDE ci avvisa che `MalformedNameException` è già rappresentata da `Exception`, dunque non è necessaria.

Altro ancora,

```
public void showName(String fullName) {
  try {
    System.out.println(fullName);
  } catch (MalformedNameException e) {
    /* ... */
  }
}
```

non compila perché il codice non lancia mai `MalformedNameException` – in questo caso, la parte `try` non va mai in eccezione, e dunque la parte nel `catch` non viene mai raggiunta. Parimenti,

```
public void showName(String fullName) {
```

```
  try {
    System.out.println(getLastName(fullName));
  } catch (Exception e) {
    /* ... */
  } catch (MalformedNameException e) {
    /* ... */
  }
}
```

non compila poiché `MalformedNameException` è una sottoclasse di `Exception`, e non verrà mai raggiunto il relativo blocco `catch` – il viceversa, però, si può fare.

16.1.1 Il costrutto `try-catch-finally`

Esiste anche una parte del costrutto precedente `finally`, il "flusso finale":

```
try {
  /* normal flow */
} catch (Exception e) {
  /* anomalous flow */
} finally {
  /* finally flow */
}
```

che consiste in operazioni eseguite **a prescindere** dall'esito delle precedenti esecuzioni (uscendo sia da `try` che da `catch`). Per esempio, i `close()`. Il costrutto con `finally` **non modifica lo stato** di flusso anomalo o quello normale: non cambia lo stato di anomalo. La parte in `finally` viene eseguita con la medesima condizione dei costrutti sopra – **e viene eseguita sempre, anche se il `try` torna dal metodo chiamante.** Il finally viene eseguita comunque anche con un `return` collocato nel `try` (da non fare!).

Ad esempio,

```
public void doThings(int n) {
  try {
    System.out.print("n1");
    if (n == 0) {
      throw new Exception();
    }
    System.out.print("n2");
  } catch (Exception e) {
    System.out.print("a");
  } finally {
    System.out.print("f");
  }
  System.out.println("n3");
}
```

Invocato con `n==1`, non lancia eccezioni e stampa `n1 n2 f n3`; se invece `n==0` stampa `n1 a f n3`.

Altro esempio più complesso,

```
public void doThings(int n)
  throws Exception {
  try {
    System.out.print("n1");
    if (n == 0) {
      throw new Exception();
    }
    System.out.print("n2");
  } finally {
    System.out.print("f");
  }
  System.out.println("n3");
```

```
}

public void externalDoThings(int n) {
  try {
    System.out.println("en1");
    doThings(n);
    System.out.println("en2");
  } catch (Exception e) {
    System.out.println("ea");
  }
  System.out.println("en3");
}
```

Se si invoca `n==1`, stampa `en1 n1 n2 f n3 en2 en3`; altrimenti, `n==0` stampa `en1 n1 f ea en3`.

Lo stato di essere anomalo è **relativo** rispetto al flusso che ha chiamato il flusso atuale – si possono, perciò, **annidare** costrutti `try-catch`:

```
try {
  throw new Exception("problem"); // A
} catch (Exception e) {
  try {
    throw new Exception("further problem"); // B, anomalous wrt A
  } catch (Exception furtherE) {
    /* ... C, anomalous wrt B */
  }
}
```

dunque, l'informazione sull'anomalia è uno *stack*.

16.1.2 Eccezioni non controllate, throwables

Package java.lang

Class Exception

java.lang.Object java.lang.Throwable java.lang.Exception

La classe `Error` è una diramazione di `Throwable`:

Package java.lang

Class Error

java.lang.Object java.lang.Throwable java.lang.Error

Un'altra differenza è `RuntimeException`:

Package java.lang

Class RuntimeException

java.lang.Object java.lang.Throwable java.lang.Exception java.lang.RuntimeException

Il linguaggio specifica che solo i `Throwable` e le sottoclassi possono essere argomenti dello statement `throw`, `throws` e `catch`. Ciascuna `Exception` e sottoclasse deve essere raccolta o dichiarata, **a meno che non sia una `RuntimeException`** – le `RuntimeException` sono **eccezioni non controllate**, per le quali non è necessario gestirle e catturarle (non serve nemmeno dichiararle). Dunque, non siamo obbligati nel caso di `RuntimeException` a mettere blocchi `try-catch`. Anche gli errori `Error` sono eccezioni non controllate.

Dunque,

Package java.lang

Class RuntimeException

java.lang.Object java.lang.Throwable java.lang.Exception java.lang.RuntimeException

RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine.

RuntimeException and its subclasses are unchecked exceptions. Unchecked exceptions do not need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

Sostanzialmente, esse sono le cose che capitano *così di frequente* rispetto agli statement che sarebbe pesante obbligarne la gestione. Ad esempio, la `NullPointerException`,

> Package java.lang
>
> Class NullPointerException
>
> java.lang.Object java.lang.Throwable java.lang.Exception java.lang.RuntimeException java.lang.NullPointerException
>
> Thrown when an application attempts to use null in a case where an object is required. These include:
>
> Calling the instance method of a null object.
>
> Accessing or modifying the field of a null object.
>
> Taking the length of null as if it were an array.
>
> Accessing or modifying the slots of null as if it were an array.
>
> Throwing null as if it were a Throwable value.
>
> Applications should throw instances of this class to indicate other illegal uses of the null object.

Alcuni esempi,

```
String s = //.., rets null
s = s.toUpperCase();

int[] a = //.., rets null
int l = a.length;

int[] a = //.., rets null
a[0] = 0;
```

e può capitare *in ogni dot notation*. La JVM si occupa di generare l'eccezione `NullPointerException` al posto nostro.

Un'altra causa di `RuntimeException` è la seguente,

> Package java.lang
>
> Class ArrayIndexOutOfBoundsException
>
> java.lang.Object java.lang.Throwable java.lang.Exception java.lang.RuntimeException java.lang.IndexOutOfBoundsException java.lang.ArrayIndexOutOfBoundsException
>
> Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

Ora veniamo agli **errori**.

> Package java.lang
>
> Class Error
>
> java.lang.Object java.lang.Throwable java.lang.Error
>
> An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The ThreadDeath error, though a "normal" condition, is also a subclass of Error because most applications should not try to catch it.
>
> A method is not required to declare in its throws clause any subclasses of Error that might be thrown during the execution of the method but not caught, since these errors are abnormal conditions that should never occur. That is, Error and its subclasses are regarded as unchecked exceptions for the purposes of compile-time checking of exceptions.

Dunque, il problema è anormale e *così serio* che un'*applicazione ragionevole* non dovrebbe provare a gestire. In altre parole, **quando si verifica un'errore è troppo tardi**! Gli errori estendono direttamente la classe `Throwable`.

Esempi di errori sono l'`OutOfMemoryError` (manca spazio nell'heap: sarebbe inutile gestire l'errore, la memoria è completamente piena),

> Package java.lang
>
> Class OutOfMemoryError

java.lang.Object java.lang.Throwable java.lang.Error java.lang.VirtualMachineError java.lang.OutOfMemoryError

Thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory could be made available by the garbage collector.

e lo `StackOverflowError` (manca spazio nello stack: parimenti, è inutile gestire l'errore quando lo stack è completamente riempito):

Package java.lang

Class StackOverflowError

java.lang.Object java.lang.Throwable java.lang.Error java.lang.VirtualMachineError java.lang.StackOverflowError

Thrown when a stack overflow occurs because an application recurses too deeply.

Vediamo ora il concetto più generale possibile. La classe `Throwable` possiede i seguenti costruttori,

`Throwable()` Constructs a new throwable with null as its detail message.

`Throwable(String message)` Constructs a new throwable with the specified detail message.

`Throwable(String message, Throwable cause)` Constructs a new throwable with the specified detail message and cause.

e ha i seguenti metodi,

`String getMessage()` Returns the detail message string of this throwable.

`void printStackTrace()` Prints this throwable and its backtrace to the standard error stream.

`void printStackTrace(PrintStream s)` Prints this throwable and its backtrace to the specified print stream.

`void printStackTrace(PrintWriter s)` Prints this throwable and its backtrace to the specified print writer.

In particolare, esiste un metodo di solito così usato, `printStackTrace(System.err)`, estremamente utile per il debugging, ma che **non** dovrebbe essere adoperato su software in produzione, poiché in essi si desidera intercettare le eccezioni in un appropriato **sistema di logging**. In particolare, esistono tre varianti di esso, e tipicamente viene adoperato sullo *standard error* stream del sistema operativo. Lo standard error è uno stream "di output" a caratteri di sistema modellato dalla Java Virtual Machine, in cui è lecito inviare i messaggi relativi alle condizioni d'errore.

In alternativa, è possibile adoperare un qualunque `PrintWriter`.

Ad esempio, un suo utilizzo è il seguente,

```
public static void dec(int n)
    throws Exception {
  if (n == 0) {
    throw new Exception("surprise!");
  }
  dec(n - 1);
}
public static void main(String[] args) {
  try {
    dec(5);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

che stamperà sullo standard error i seguenti messaggi:

```
java.lang.Exception: surprise!
    at it.units.erallab.hmsrobots.objects.Robot.dec(Test.java:185)
    at it.units.erallab.hmsrobots.objects.Robot.dec(Test.java:185)
    at it.units.erallab.hmsrobots.objects.Robot.dec(Test.java:185)
    at it.units.erallab.hmsrobots.objects.Robot.dec(Test.java:185)
    at it.units.erallab.hmsrobots.objects.Robot.dec(Test.java:185)
```

```
at it.units.erallab.hmsrobots.objects.Robot.dec(Test.java:185)
at it.units.erallab.hmsrobots.objects.Robot.main(Test.java:191)
```

In altre parole, anziché farne il log si stampa semplicemente messaggi d'errore sullo standard error. Il metodo stamperà il metodo, il file sorgente e il codice sorgente della linea avente lo statement in cui il flusso è divenuto anomalo. Risulta dunque molto utile in caso di debugging del software.

16.1.3 Design della gestione dell'errore e dell'eccezione

Gli errori in Java sono dunque classi, in particolare avremo `Throwable`, `Error`, `Exception` e `RuntimeException`; la sintassi per definire flussi normali ed anomali è il `try-catch-finally` e per passare in flusso anomalo si adopera `throw` (per tornare al flusso normale è necessario uscire da un `catch`).

Resta però da domandarsi qualora sia opportuno gestire un errore e i casi in cui, invece, è meglio *propagarlo* ai metodi chiamanti: quando, dunque, **va creato un errore**?

Purtroppo non esiste una risposta univoca. Le linee guida al riguardo saranno elecante qui di seguito.

L'idea di fondo è la seguente: supponiamo di avere un frammento di codice o un metodo `m` in cui un'eccezione `E` esiste; se lo sviluppatore di `m` sa gestire la condizione anomala risultante da `E`, **allora lo sviluppatore lo deve gestire all'interno del metodo**. Altrimenti, se lo sviluppatore non sa gestire la situazione anomala derivante da `E`, **deve propagare l'errore**.

Ad esempio, `String getWebPageTitleIfAny(String url)`, con `MalformedURLException` all'interno; fa una promessa "ben qualificata" (IfAny – potrebbe non esistere, la sa gestire, restituisce `""` in quel caso).

Dunque, lo sviluppatore sa gestire `MalformedURLException` **internamente** al metodo.

Un altro caso opposto, `Configuration loadFromFile(String fileName)` con all'interno `FileNotFoundException` – in questo caso è meglio propagare, poiché il metodo promette di restituire una configurazione. Nel caso di errore, infatti, il metodo non avrebbe alcuna possibilità di restituire una configurazione (non può nemmeno restituire una non–configurazione, come nel caso di una non–stringa, la stringa vuota).

La metodologia, in generale, ricade sotto il nome di **design by contract**, in cui si stabilisce una sorta di **contratto** fornito dal metodo. Per ciascun metodo, si definiscono,

- le **precondizioni** che l'input deve rispettare, l'insieme dei predicati per cui l'input deve essere vero – esse sono responsabilità che ricadono **sul chiamante**;
- le **postcondizioni** che l'output dovrà rispettare, i predicati che devono essere sempre garantiti (veri) dall'output – esse sono responsabilità invece che saranno **del chiamato**;

a questo punto, il metodo dovrebbe ritornare in stato anomalo **se e solo se**:

- le precondizioni non sono rispettate;
- le postcondizioni non possono essere rispettate, nonstante le precondizioni lo siano – per problemi irrisolvibili all'interno del metodo, o comunque non risolvibili opportunamente.

Ogni qualvolta si debba sviluppare una funzionalità software (un oggetto, un metodo, un blocco input—output) si dovrebbero individuare le precondizioni e le postcondizioni. Le precondizioni sono una qualità che deve essere rispettata dal chiamante; viceversa, le postcondizioni sono una qualità che deve garantire, invece, il metodo chiamato.

In realtà, la gestione delle precondizioni è in parte demandata al linguaggio (`String` significa che si accettano solo stringhe) e, almeno in parte, deman-

data al metodo chiamato (verificare se un intero è positivo). Non esiste dunque una metodologia generale che si può applicare in tutti i casi possibili indistintamente.

Dunque, nel gestire l'errore si cerca di rispettare le postcondizioni. Nel crearlo, invece, si guarda le precondizioni (se non sono soddisfatte) e le postcondizioni (non si possono soddisfare nello stato attuale) – ad esempio,

```
public String getLastName(String fullName) throws
    MalformedNameException {
  String[] pieces = fullName.split(" ");
  if (pieces.length == 0) {
    throw new MalformedNameException("Empty name!");
  }
  return pieces[pieces.length-1];
}
```

Nel frammento di sopra, le precondizioni sono che `fullName` non sia `null` e che non sia la stringa vuota (in tal caso si incapperebbe in un unchecked `ArrayIndexOutOfBoundsException`).

La pratica migliore è, in generale, di **nascondere l'implementazione interna**. Se una componente, libreria o software `S1` trova un'eccezione `E2` relativa a specifiche componenti, librerie o software `S2` usate in `S1`, non dovrebbe propagarlo, bensì creare una nuova eccezione `E1` legata ad `S1`. In altre parole, il codice che utilizza `S1` non dovrebbe avere la necessità di conoscere `S2`! È molto meglio, infatti, fornire un'eccezione *di libreria*, non nota all'esterno.

Un errore comune è quello dello **fake handling**. Il fake handling è la pratica di fare un `catch` senza gestire l'errore – una pratica da evitare assolutamente per ovvie ragioni (si ritornerebbe a produrre software alla vecchia, complicata maniera). Se non si sa come gestire l'errore in fase di scrittura di codice, è **almeno** opportuno mettere un `e.printStackTrace()`.

Il compilatore non compila comunque con `catch` vuoto se `doRiskyThing()` ha un valore di ritorno, ad esempio una stringa – in quel caso esisterebbe almeno un ramo di computazione in cui tale stringa verrebbe adoperata.

La pratica è generalmente abusata quando si adoperano thread multipli, anche se è particolarmente errata anche in questo caso.

Una cosa estremamente importante è che `run()` non può vedere aumentata la clausola `throws` – ciascuna eccezione deve essere gestita; la tentazione di non farlo è dunque elevatissima.

Se nessun metodo gestisce l'eccezione, ad esempio `NullPointerException`, il *thread si interrompe*:

- se avviene nel thread principale, l'applicazione si ferma e il danno è evidente;
- altrimenti, muore soltanto un thread, **silenziosamente**.

Un'altra pratica sbagliata è quella di fare **`catch` eccessivamente generali** – quando cioè uno sviluppatore pigro cerca di gestire più eccezione all'interno di un solo ramo. Ogni eccezione dovrebbe essere gestita indipendentemente.

Sbagliato (il codice prende anche altre eccezioni non volute, ad esempio `NullPointerException`):

```
try {
  doRiskyThing();
} catch (Exception e) {
  /* handling code for both */
}
public void doRiskyThing()
    throws BadFooException, BadBarException {
  /* ... */
}
```

mentre la pratica corretta è

```
try {
  doRiskyThing();
} catch (BadFooException e) {
  /* handling code for foo */
} catch (BadBarException e) {
  /* handling code for bar */
}
```

Qualora sia importante fare diversamente, da Java 7 è stata introdotta una sintassi che semplifica la gestione di errori multipli nello stesso `catch`:

```
try {
  doRiskyThing();
} catch (BadFooException|BadBarException e) {
  /* handling code for both */
}
```

A tempo di compilazione, `e` è il supertipo più specifico di quelli elencati nella sintassi (è solo una scorciatoia sintattica), e invocando la dot notation su `e` verrà preso `Exception` poiché è il supertipo più specifico in comune ad entrambi (ammesso che `BadFooException` e `BadBarException` estendano `Exception`) – questo va bene solamente nel caso in cui si desideri fare la stessa cosa per ambedue le eccezioni.

Ricapitolando, in Java spostiamo tutto il controllo a tempo di compilazione – **esplicitiamo la responsabilità** della gestione delle eccezioni, a livello del linguaggio; "non compila se la responsabilità non è correttamente assegnata".

Dunque:

- se le precondizioni sono rispettate e le postcondizioni possono essere rispettate, è **responsabilità del chiamato**;
- se le precondizioni non sono rispettate, è **responsabilità del chiamante**;
- se non possono essere rispettate le postcondizioni per problemi di natura grave e seria, **propaga comunque** anche se la responsabilità non è del chiamante.

Nella pratica, *bisogna sempre fare log* se forzati a gestire l'eccezione in qualunque caso, ed *è importante nascondere gli aspetti interni del modulo*.

Gestione delle eccezioni unchecked

La gestione delle eccezioni unchecked deve essere fatta implicitamente o esplicitamente:

- implicitamente quando siamo sicuri che l'eccezione non può capitare – l'abbiamo gestita con il codice stesso;
- esplicitamente, se sappiamo che può capitare, la mettiamo nel codice – dunque, l'eccezione va gestita all'interno del metodo che morirebbe con un `try-catch` con un `Throwable` per gestire ad esempio `ArrayIndexOutOfBoundsException`.

In ogni caso, in qualche maniera esse vanno gestite anche se il compilatore ci lascia non farlo.

Capitolo 17

Adoperare le eccezioni per gestire le risorse di sistema

Il tipico workflow adoperato per utilizzare risorse di sistema è il seguente:

- **open** – prepara la risorsa di sistema;
- **use** – utilizza la risorsa, compi un'azione sulla risorsa in questione;
- **close** – finisco di adoperare la risorsa.

Un possibile flusso anomalo derivante dall'utilizzo di tali risorse è quello in cui la risorsa stessa può terminare violentemente; i componenti della JDK riflettono questo aspetto facendo degli opportuni `throw` di eccezioni (ad esempio, i `throw` che si riscontrano utilizzando gli stream su risorse di sistema).

Chiudere una risorsa di sistema è importantissimo! La maggior parte dei sistemi operativi rilasciano le risorse una volta terminato il programma, però **non lo fanno** quando **il processo è ancora in esecuzione**: il programma ha la **responsabilità** di chiudere le risorse ad esso associate. In assenza di tale comportamento, il sistema operativo potrebbe uccidere il processo oppure eliminare le risorse di sistema – ma tale comportamento rende il tutto estremamente impredicibile.

Dunque, la regola fondamentale è **assicurarsi di chiudere sempre le risorse di sistema, anche sotto flusso anomalo**. La parola *sempre* acquisisce un nuovo significato – in particolare, chiudiamo anche in flusso anomalo, sempre!

La prima soluzione sub–ottimale è illustrata di seguito (con una finta classe `Resource`),

```
try {
  Resource r = open();
  r.use();
  r.close();
} catch (AException e) {
  log(e);
  r.close();
} catch (BException e) {
  log(e);
  r.close();
}
```

dove il frammento sembra chiudere sempre. Tuttavia, non è assolutamente vero che succeda – infatti, in questo caso la risorsa non viene chiusa per i seguenti problemi:

1. quando un'eccezione diversa da quelle elencate è lanciata;
2. `r` non è definita nel blocco `catch` (infatti, il blocco soprastante non compila nemmeno);
3. `close` è ripetuto 3 volte, è dunque ridondante e pesante (problema di *eleganza*).

Una maniera più corretta dunque è **chiudere nei `finally`**:

```
Resource r;
try {
  r = open();
  r.use();
} catch (AException e) {
  log(e);
} catch (BException e) {
  log(e);
} finally {
  r.close();
}
```

Il blocco `finally` è infatti **sempre** eseguito, non importa il tipo di flusso: l'ideale per **chiudere sempre** le risorse.

Tuttavia, cosa accade se l'eccezione è lanciata in `open()`? Purtroppo, `r` non è inizializzato e dunque `r.close()` lancerebbe una `NullPointerException`. La terza problematica è che comunque il codice è ancora parecchio verboso.

La maniera corretta – ma vetusta – di risolvere il problema è nel frammento successivo,

```
Resource r;
try {
  r = open();
  r.use();
} catch (AException e) {
  log(e);
} catch (BException e) {
  log(e);
} finally {
  if (r != null) {
    r.close();
  }
}
```

ma ha un problema per cui `close()` potrebbe ulteriormente generare un'eccezione. Il problema è risolto in Java 7: infatti, la nuova sintassi è la seguente:

```
try (Resource r = open()) {
  r.use();
} catch (AException e) {
  log(e);
} catch (BException e) {
  log(e);
}
```

dove `try` precede una parentesi tonda dove sono inclusi tutti gli statement che soddisfano determinati requisiti (**try–with–resources**) – tutti gli identificatori dichiarati all'interno, sono automaticamente chiusi in un `finally` dove tutte le risorse di sistema sono chiuse. L'ipotetico `finally` si preoccupa di verificare che il riferimento non sia `null`, di applicare un `try-catch` attorno al `close` e di chiudere dunque la risorsa dichiarata. Quando si esce dal blocco `try`, anche andando nel flusso anomalo, si chiude la risorsa di sistema, a prescindere dunque (una sorta di `finally` anticipato), senza cambiare lo stato normale o anomalo in qualsivoglia maniera.

Quando si intendono adoperare risorse multiple, si fa così:

```
byte[] data = new byte[100];
try (
    FileInputStream fis = new FileInputStream(inputFile);
    FileOutputStream fos = new FileOutputStream(outputFile)
) {
  while (true) {
    int nOfReadBytes = fis.read(data);
    if (nOfReadBytes == -1) {
      break;
    }
    fos.write(data, 0, nOfReadBytes);
  }
} catch (IOException e) {
  System.err.printf("Cannot copy due to %s", e);
}
```

e le risorse sono chiuse **in ordine inverso** rispetto a come erano state aperte. Le risorse vanno elencate **con punto e virgola**.

```
public void log(String msg, String filePath) throws IOException {
  try (BufferedWriter bw = new BufferedWriter(new
      FileWriter(filePath))) {
    bw.append(msg + System.lineSeparator());
  }
}
```

Naturalmente, le risorse devono per forza avere un metodo `close()` (tipi **autocloseable**) – non è però adoperata l'ereditarietà per il controllo (ma un altro meccanismo).

Senza blocco `catch` è ugualmente possibile aprire risorse di sistema (senza però alcuna gestione delle eccezioni),

```
public static void log(String msg, String filePath) throws
    IOException {
  try (BufferedWriter bw = new BufferedWriter(new
      FileWriter(filePath))) {
    bw.append(msg + System.lineSeparator());
  }
}
```

in questo caso il `BufferedWriter` è comunque chiuso sempre, però non gestisce eccezioni e il compilatore richiede di gestire o aggiungere clausole `throws` che prenderanno il posto del costrutto `catch`. Concettualmente, il `finally` viene eseguito prima di ridare il controllo al chiamante, chiudendo tutte le risorse aperte.

Una **risorsa** è dunque tutto ciò che è **chiudibile** – cioè che presenta un'**interfaccia** `AutoCloseable`; tutte le classi che offrono un metodo `close()`. Lo sono tutte le classi che gestiscono l'input e l'output.

Attenzione – si possono definire delle **inner classes**, come ad esempio `Pipe.SinkChann` una classe **public** definita all'interno della classe `Pipe`. Essa non ha alcun rapporto di ereditarietà con la classe in cui è definita – il legame fra le due è che quest'ultima ha una **funzionalità** essenziale nella classe che definisce.

17.1 Esempi di server robusti

Sia il seguente protocollo:

- il client invia una linea di test `l`;

- se `l == lQuit`, il server chiude la connessione; altrimenti risponde con la linea processata `lp = p(l)`.

Il server ascolta sulla porta `nPort`, gestisce tanti client alla volta, **e non termina mai in maniera violenta** – le eccezioni possono accadere (un client chiude la connessione) e rilascia le risorse di sistema correttamente. Il design vogliamo sia estendibile. Infine, semplicemente `p: String -> String`, `lQuit`, con numero di porta parametro.

Ad esempio,

```
public class RobustLineProcessingServer extends
    LineProcessingServer {
  public RobustLineProcessingServer(int port, String quitCommand) {
    super(port, quitCommand);
  }
  public void run() throws IOException { /* ... */ }
}
```

che estende `LineProcessingServer`, già implementato – da esso eredita i field (resi come `protected`) e i metodi `process()` e `getQuitCommand()`.

La funzione `run()`,

```
public void run() throws IOException {
  try (ServerSocket serverSocket = new ServerSocket(port)) {
    while (true) {
      Socket socket;
      try {
        socket = serverSocket.accept();
        ClientHandler clientHandler = new
            RobustClientHandler(socket, this);
        clientHandler.start();
      } catch (IOException e) {
        System.err.printf("Cannot accept connection due to %s", e);
      }
    }
  }
}
```

il costruttore `ServerSocket` può lanciare un'eccezione `IOException`, ma non sappiamo come gestirla (non possiamo **garantire le postcondizioni**: dobbiamo propagare l'errore, non facendo il `catch` per la risorsa invocata nel `try`). Il metodo `accept()` può lanciare l'eccezione `IOException`: in quel caso, il server deve comunque rimanere in vita, devo dunque **gestire l'errore**. In questo caso, la gestione dell'errore avviene semplicemente facendo il log. Non avviene un `try`-with-resources: questo perché vogliamo che la gestione di `close()` avvenga da parte di `ClientHandler` (in questo caso, "irrobustito" in `RobustClientHandler`). Infatti, non può chiudersi qua, altrimenti il client non potrebbe utilizzare la risorsa (immediatamente chiusa dal try-with-resources).

Il `RobustClientHandler`:

```
public class RobustClientHandler extends ClientHandler {
  public RobustClientHandler(Socket socket, LineProcessingServer
      lineProcessingServer) {
    super(socket, lineProcessingServer);
  }
  public void run() { /* ... */ }
}
```

estende `ClientHandler`, e non richiede la versione robusta del server (ereditarietà, servono solo `process()` e `getQuitCommand()`. Il suo metodo `run()`, invece,

```
public void run() {
  try (socket) {
    BufferedReader br = new BufferedReader(new
        InputStreamReader(socket.getInputStream()));
    BufferedWriter bw = new BufferedWriter(new
        OutputStreamWriter(socket.getOutputStream()));
    while (true) {
      String line = br.readLine();
      if (line == null) {
        System.err.println("Client abruptly closed connection");
        break;
      }
      if (line.equals(lineProcessingServer.getQuitCommand())) {
        break;
      }
      bw.write(lineProcessingServer.process(line) +
          System.lineSeparator());
      bw.flush();
    }
  } catch (IOException e) {
    System.err.printf("IO error: %s", e);
  }
}
```

e quando `line == null` il server si accorge che il client ha chiuso violentemente la connessione. Il metodo `readLine()` infatti ritorna `null` qualora si sia raggiunto l'end-of-file.

Nel caso della `NullPointerException` non gestita (supponiamo niente controllo sul `null` e niente `catch`) il thread morirebbe, poiché il chiamante `start()` non gestisce l'eccezione – il thread muore.

La sintassi `try (Socket)` è possibile da Java 9.

Supponiamo ora avvenga una `IOException` alla n+1-esima `readLine()` – non sapremo **mai** se il messaggio n-esimo è arrivato, o se lo sono addirittura i precedenti n! Questo perché non abbiamo garanzie fornite dal TCP **in caso di fallimento**, le abbiamo infatti soltanto in caso di successo. Riguardo all'arrivo dei messaggi lato client, dall'1 all'n-esimo, non possiamo dire nulla: l'eccezione viene generata dal sistema operativo e arriva al programma tramite la JVM.

Ad esempio, un client `C` scrive la richiesta, legge la risposta – il server fa il viceversa: il client dapprima chiama il sistema operativo su cui gira, a quel punto il sistema operativo invoca il device per la trasmissione; la trasmissione avviene, il device del server riceve il segnale, lo interpreta, e lo manda al sistema operativo, il quale dialogherà col server fornendogli la richiesta (il messaggio) tramite una richiesta di lettura (che viene "congelata" finché non arriva il messaggio). Lato server, avviene il viceversa.

Dunque, le write request e write response passano per tantissime entità (sistema operativo, device client, device server) mentre le read coinvolgono soltanto il sistema operativo ed il client o il server.

Siccome `C` non manda mai una richiesta finché è arrivata la risposta: per questa ragione, avendo ricevuto con successo l'ennesima risposta, **è possibile solo in quel caso** capire che l'n-1 esimo messaggio è arrivato; tuttavia, non è chiaro capire **dove esattamente** o **quando** ciò sia avvenuto.

Questo può essere un problema per i protocolli **pipelined message–oriented**, i protocolli dove possono essere inviate più richieste di fila e avere più risposte di fila. In questo caso, sappiamo solo se gli ultimi n - k messaggi sono arrivati – quelli per cui abbiamo ricevuto una risposta se all'ultimo passaggio abbiamo inviato k messaggi.

Questo è un problema **fondamentale** nel caso in cui si voglia **scrivere** o mandare messaggi **importanti** – cioè quelli che producono **cambiamenti**

di stato; meno grave per richieste che forniscono solo letture, che cioè non cambiano lo stato e per cui possono sussistere duplicati di richieste.

Capitolo 18

Caricamento delle classi e documentazione Java

18.1 Gestione delle classi da parte della JVM

La domanda che ci poniamo ora è, effettivamente, *come* ***`java`*** *esegue i* ***`.class`***. In particolare, una volta compilato il main, come fa `java` ad eseguirlo?

Nei linguaggi compilati, ci vuole una *traduzione* (la compilazione), con una fase di controlli che avviene in tale momento.

I file `.class` contengono:

- il codice dei metodi;
- il codice dei costruttori;
- i nomi dei field e il loro tipo;
- e così via.

Quando la JVM inizia ad eseguire `Main.class`, essa ha il codice di `Main.main()`, ma non ha inizialmente il codice per oggetti provenienti da altre classi, come ad esempio un `Greeter()` e `Greeter.greet()`. Dovrà dunque andarlo a prelevare da un'altra locazione della memoria. La JVM deve adoperare il corretto `.class` qualora un costruttore, un metodo o un campo statico sono eseguiti o vi è accesso. In memoria viene utilizzato (caricato in memoria, oppure usato se già presente) il corrispondente `.class` quindi quando il costruttore o un metodo di tale file viene eseguito, o quando un field statico viene utilizzato.

In altre parole, non tutti i `.class` vengono caricati in memoria in anticipo, ma vengono caricati dinamicamente **al primo dei tre utilizzi sopracitati**.

```
public class Main {
  public static void main(String[] args) {
    Greeter greeter = new Greeter("hi");
    greeter.greet("Eric");
  }
}
public class Greeter {
  private final String formula;
  public Greeter(String formula) {
    this.formula = formula;
  }
  public void greet(String name) {
    System.out.println(
      formula + " " + name
    );
  }
}
```

```
eric@cpu:~$ java Main
hi Eric
```

La sequenza delle classi caricate sarà:

1. `String.class`;
2. `CharSequence.class` (di conseguenza);
3. `Object.class` (di conseguenza);
4. `Main.class`;
5. `Greeter.class`;
6. `System.class`;
7. ...
8. `PrintStream.class`;
9. ...

I controlli in fase di compilazione sono chiamati **controlli di consistenza o coerenza statica**. Essi sono controllati da `javac` alla compilazione, per verificare che ogni utilizzo di field, metodo o costruttore sia legittimo.

Esistono anche controlli in fase di esecuzione, detti **controlli di consistenza o coerenza dinamica**: controllati da `java` al runtime, effettuano ugualmente il controllo di cui sopra, al tempo d'esecuzione. Tali controlli sono fatti *al momento del caricamento del* `.class`; supponiamo infatti possano avvenire modifiche illecite al contenuto di un `.class` – per questa ragione è importante anche il controllo al runtime.

Entrambi i tipi di controllo statico e dinamico avvengono mediante l'ispezione dei `.class`:

- quando si compila, si va a controllare i `.class` per l'esistenza dei metodi;
- quando si esegue, si fa la stessa cosa!

Un controllo di consistenza avviene anche guardando la signature – un dettaglio omesso è anche che **il `.class` contiene anche le signature** di metodi e costruttori. La compilazione statica garantisce che le signature vengano rispettate, pena la non riuscita della compilazione. Se la verifica dinamica (controllo di consistenza dinamica) fallisce, invece, **viene lanciata un'eccezione**; `ClassNotFoundException` (una `RuntimeException`) se non viene trovato banalmente il `.class`, `NoSuchMetodException` se il metodo o il costruttore non viene trovato nel corrispondente `.class` (anche qua una `RuntimeException`); similmente, se non si trova un field si avrà una `NoSuchFieldException`.

18.1.1 L'autocompilazione

L'**autocompilazione** è il meccanismo mediante il quale avviene la compilazione di qualsiasi `.java` per cui il codice ha bisogno del corrispondente `.class`. Ad esempio, supponiamo il main abbia necessità di un metodo, costruttore o field della classe `Greeter` – se non compilato, il compilatore `javac` lo compilerà automaticamente.

Le posizioni in memoria dove il compilatore va a cercare i file sorgente sono organizzati in una particolare maniera. Supponiamo che la cartella base sia `/java-projects/APFinalProject/src/main/java`,

- `it.units.approject.Main` deve essere un `Main.java` in `~/java-projects/APF`
  `main/java/it/units/approject`;
- `it.units.approject.util.Parser` deve essere un `Parser.java` in `~/java-pro`
  `src/main/java/it/units/approject/util`.
- ...

In realtà, nella pratica l'applicazione viene opportunamente impacchettata secondo pratiche standard di Java. Vi sono dunque tante opzioni per impacchettare (uno dei tanti passi del processo di **build**):

- nelle cartelle JDK per le classi appartenenti del JDK;
- nella stessa cartella del file principale `.java` (la maniera vetusta);
- nel directory tree che sia uguale nella parte terminale a quello del `.java`, ma in un'altra radice, ad esempio dentro una cartella `target` (la maniera facile e comunemente adoperata per il software stesso sviluppato da noi);
- in un file `.jar` (Java ARchive) opportunamente collocato (compattando i `.class` della cartella `target`), sotto un formato analogo allo `zip` (adoperato per software usato dal software che stiamo costruendo).

Dunque, il processo di **build** viene gestito o dall'IDE o da tool detti **build–automation tools**, ad esempio *Maven* e *Gradle* – essi sono in grado di **gestire le dipendenze** automaticamente.

18.2 Documentare il proprio software

La **documentazione** può e (dovrebbe) essere scritta opportunamente su qualsiasi software. La documentazione API (`javadoc`) è il principale metodo.

Tuttavia, nel corso non verrà dara una risposta vera e propria alla questione su come documentare il software.

La domanda utile è però quale sia il **target** a cui la documentazione è rivolta:

- sono gli **utenti finali**? (non interessati a come il software è fatto, servono solo le istruzioni);
- sono gli **sviluppatori di altri software**? (useranno il nostro software; ha senso solo se il nostro software verrà usato anche da altre utenze oltre che dall'utente finale);
- sono gli **sviluppatori del software stesso**? (serve per coordinare gli sforzi, evitare errori, ricordarsi di aspetti nel futuro);

Dunque, la documentazione serve a **trasferire conoscenza** ad utenti finali, ad altri sviluppatori, o a noi stessi. Idealmente, **la conoscenza dovrebbe risiedere nel software stesso**, cioè nel cosiddetto **codice autoesplicativo**, ad esempio il pseudo–codice può essere un valido esempio di tale principio di trasmissione della conoscenza (ad esempio, illustrare un algoritmo).

Agli sviluppatori che usano il nostro software è importante fornire una buona struttura software e dei **buoni nomi** per metodi e campi – inoltre, per noi stessi è importante la qualità del codice.

Non esiste soltanto la documentazione per lo scopo – è infatti possibile trasmettere conoscenza adeguatamente anche con *esempi*, *wiki*, e così via; spesso questi documenti sono *supplementari* a quanto fornito con la documentazione javadoc.

La documentazione **ha un costo** in termini di tempo, e andrebbe opportunamente aggiornata con le modifiche del codice.

La javadoc è scritta con una sintassi opportuna **direttamente all'interno dei `.java`**. L'API documentation viene automaticamente generata a partire dai sorgenti, mediante `javadoc` – verranno creati un insieme di documenti web, spesso tramite l'aiuto dell'IDE.

Un esempio del formato javadoc è il seguente:

```
package it.units;
/**
 * A class for producing greeting strings with a provided formula.
```

```
 *
 */
public class Greeter {
  private final String formula;
  /**
   * Build a new greeter with the provided {@code String} as
       greeting formula.
   *
   * @param formula A greeting formula.
   */
  public Greeter(String formula) {
    this.formula = formula;
  }
  /**
   * Produces a greeting string with the given {@code name}.
   *
   * @param name The name to be included in the greeting string.
   * @return A {@code String} with a greeting of {@code name}
   * with the formula of this {@code Greeter}.
   */
  public String greet(String name) {
    return formula + " " + name;
  }
}
```

con il commento relativo al javadoc incomincia con `/**`, e va collocato nella precisa posizione (prima del metodo o della classe). Esistono i cosiddetti **tag** che cambiano la formattazione o che diventano un link (`@code`, `@param` diventa un link che spiega il ruolo del parametro, `@return`). Possono anche essere utilizzate alcune sintassi dell'HTML.

La documentazione è opportuna nei casi in cui le cose sono molto complesse, o se i costi in tempo giustificano l'importanza della documentazione del progetto.

Javadoc è un termine usato per denotare 3 cose:

- il file binario `javadoc` adoperato per generare la documentazione;
- il nome colloquiale della API documentation;
- il nome che diamo alla sintassi;

Capitolo 19

Le Interfacce

L'**interfaccia** è una parola chiave (`interface`) che serve per definire **solo la signature** di **zero o più metodi**. L'interfaccia, diversamente dalla classe, non dice quale è lo stato, non dice come devono essere fatte le operazioni, dice **solo che esistono delle operazioni**. L'interfaccia è il massimo dell'astrazione: non ha nulla, è solo un "confine" fra due entità.

Le interfacce sono simili alle classi, ma **dopo ogni metodo mancano le graffe con la definizione**: il metodo non è definito. Non è necessario applicare il modificatore `public` (è di default), e infatti non compila né con `protected`, né con `private`. Il compilatore non ha bisogno del codice del metodo per poter compilare.

```
public interface Listener {
  void listen(Event event);
}
```

Questa interfaccia viene ugualmente compilata in un `.class`.

Il `Listener` è un esempio di **observer pattern**. Spesso capita che una componente software debba essere informata da un'altra componente, con scarsa dipendenza (non importa che l'una abbia per forza l'altra per funzionare). La componente che ascolta adotta l'observer pattern (o listener pattern) per gestire l'ascolto delle informazioni dall'altra componente software. Chi ascolta non produce nulla. In questo caso supponiamo che `Event` sia una classe o un'interfaccia (può essere anche una sottointerfaccia) definita altrove.

L'interfaccia è una superficie che sta al confine fra due spazi, fasi, materiali. Nessuno dei due corpi ha bisogno di avere accesso all'interno dell'altro – ciascuno vede l'altro attraverso l'interfaccia. Dal punto di vista delle componentei software, se le due parti sono il chiamante ed il chiamato l'interfaccia sta in mezzo: ciascuno sa i metodi dell'altro, ma entrambi sono ignari dei funzionamenti interni dell'altro.

Nessuno dei due ha bisogno di sapere chi è l'altro: questo è il motivo per cui si adoperano le interfacce – per produrre, appunto, interfacce fra entità.

Le interfacce soffrono di limitazioni:

- non possono contenere codice (anche se da Java 8 possono farlo), ma solo la signature;
- zero o più metodi (anche zero – si pensi a `Serializable`, `Cloneable`, usati come **marcatore** di classe, come un timbro);

- solo metodi — non costruttori. Un costruttore, di fatto, realizzerebbe un'operazione che esiste perché viene definita come è fatta, deve osservare lo stato, cambiarlo.

L'interfaccia si implementa così:

```
public class StandardOutputListener implements Listener {
  public void listen(Event event) {
    System.out.println(event);
  }
}
public class FileListener implements Listener, AutoCloseable {
  public void listen(Event event) { /* ... */ }
  public void close() { /* ... */ }
}
```

con la parola chiave `implements` – si dichiara che `StandardOutputListener` fornisce le funzionalità dichiarate dall'interfaccia `Listener`. Nel caso sottostante, vengono implementate ben due interfacce (dunque, **implementazioni multiple** in Java esistono).

I metodi implementati **non** possono:

- ridurre la visibilità – devono essere `public`;
- aumentare la clausola di `throws` – deve essere la stessa o una inferiore;

Si può usare così,

```
Listener listener = new FileListener(); // references can now also
    be interfaces
listener.listen(event);
```

utilizzandole come l'ereditarietà – tramite il polimorfismo, avranno comportamenti differenti a seconda di ciò che avviene nel runtime; oppure, nelle signature di metodi,

```
public static void broadcast(Event event, Listener[] listeners) {
  for (Listener listener : listeners) {
    listener.listen(event);
  }
}
```

dove per fare l'operazione espressa è necessario un `Listener`, e dunque non è bene esprimere un tipo particolare di `Listener`.

Attenzione però: **non si possono istanziare le interfacce**,

```
// missing bytecode!
Listener listener = new Listener(); // does not compile!
```

L'operatore `instanceof` funziona anche in questo caso, se l'operatore di destra è un identificatore di riferimento di interfaccia:

```
public class NullListener implements Listener {
  public void listen(Event event) {
    /* do nothing */
  }
}
NullListener listener = new NullListener();
if (listener instanceof Listener) {
  System.out.println("It is!")
}
```

Attenzione però, non vi è **relazione di ereditarietà** (parentela) fra due `Listener`:

```
NullListener listener = new StandardOutputListener(); //does not
    compile!
StandardOutputListener listener = new NullListener(); //does not
    compile!
```

Esiste però una parentela di ereditarietà **fra interfacce**: è possibile che un'interfaccia *estenda* un'altra:

```
public interface MultipleListener extends Listener {
  void listen(Event[] events);
}
```

La classe che implementa `MultipleListener` dovrà implementare **sia** `listen(Event)` **che** `listen(Event[])`.

Una differenza fra l'ereditarietà fra classi e quella fra interfacce è che **un'interfaccia può ereditare fra più interfacce** – questo ora è possibile poiché, mancando la definizione dei metodi, più interfacce possono avere uno stesso metodo `m` con stessa signature senza "pestarsi i piedi"; ciò non avveniva, invece, per le classi, dove era impossibile scegliere quale fra i due metodi con medesima signature eseguire.

In questo modo è possibile spingere ancora più all'estremo l'**astrazione** – l'interfaccia consente di definire anche solamente l'*esistenza* delle capacità, riducendo di molto la complessità ed avvicinandoci più all'alto livello del design e della progettazione. Ad esempio,

```
public interface Listener {
  void listen(Event event);
}
```

ha come definizione che "il `Listener` ascolta", un po' come quando si manipolano i concetti matematici: si risolvono teoremi e si costruisce la teoria senza creare i "casi particolari"!

L'utilizzo dell'interfacce crea **software più modulare**. Definire le interfacce e usare soltanto i loro metodi significa che è sufficiente che le classi che implementano l'interfaccia abbiano le funzionalità da quest'ultima dichiarata.

Attenzione: con la parola chiave `new` si possono passare solo nomi di classi.

Il motivo principe per cui adoperare un'interfaccia riguarda la **separation of concerns**. Lo sviluppatore si preoccupa del suo software, **non dell'interfaccia** – similmente, l'implementatore dell'interfaccia si preoccupa **solo della sua implementazione**: il punto d'incontro è, appunto, l'interfaccia, creando **disaccoppiamento** nella progettazione del software.

Nell'interfaccia vengono definiti i nomi, le funzionalità offerte, come sono definite, le loro signature – nelle classi che *implementano* un'interfaccia vengono espresse *come* tali nomi e funzionalità vengano di fatto implementate.

Le interfacce possono godere di "ereditarietà multipla" – dunque, possono essere estese due interfacce per volta!

Le interfacce aiutano a fare due cose:

- aumentano l'**astrazione** del codice;
- riducono la **complessità** per via della separation of concerns – di fatto, rendendo possibile un maggiore *disaccoppiamento*.

Supponiamo ora di vedere una classe `Listener`,

```
public class Listener {
```

```
  public void listen(Event event) {
  }
}
```

definita con il comportamento *vuoto*.

Le interfacce possono essere un'alternativa all'ereditarietà delle classi. Ci potrebbe essere una qualche sovrapposizione fra le possibilità offerte dalle interfacce e quelle offerte dall'ereditarietà; vi sono tuttavia molte differenze chiave:

- una differenza *concettuale*: le interfacce definiscono *cosa*, non *come* – cosa accadrebbe, ad esempio, se il tipo di ritorno non fosse `void`, come si realizzarebbe con una classe?
- una differenza *concettuale* e *pratica*: non c'è nessuna implementazione di default o vuota per le operazioni; praticamente, non si possono utilizzare le interfacce concretamente senza almeno un'implementazione;
- una differenza *pratica*: supponiamo di avere un tipo `A` con le sue operazioni, idem il tipo `B`, cosa faccio se voglio un tipo `AB` che sa fare ambedue le operazioni? Con l'interfaccia lo posso fare, con l'ereditarietà no.

Il *line processing server* rimane molto simile se ridefinito con l'interfaccia. Tuttavia, cambia che il requisito è che **il server non deve essere ricompilato per un nuovo p**, deve essere passato al momento della costruzione come per il numero di porta e il comando d'uscita. Ovvero, p diventa un **parametro**.

Come facciamo?

Così,

```
public interface CommandProcessor {
  String process(String input);
}
```

```
public class Uppercaser implements CommandProcessor {
  public String process(String input) {
    return input.toUpperCase();
  }
}
```

con un'interfaccia `CommandProcessor` che è in grado di fare `process()`.

Dunque,

```
public class LineProcessingServer {
  private final int port;
  private final String quitCommand;
  private final CommandProcessor processor;
  public LineProcessingServer(int port, String quitCommand,
      CommandProcessor processor) {
    /* ... */
  }
  public void run() throws IOException {
    /* ... */
  }
  /* quitCommand and processor getters */
}
```

il `CommandProcessor` viene passato **come argomento**! Ad esempio,

```
LineProcessingServer uppercaserServer = new LineProcessingServer(
    10000,
    "BYE",
    new Uppercaser()
);
uppercaserServer.run();
LineProcessingServer tokenCounterServer = new LineProcessingServer(
```

```
    10001,
    "BYE",
    new TokenCounter(" ")
);
tokenCounterServer.run();
```

Abbiamo 1 thread principale, 2 che fanno l'accept, e 2 thread in più per ogni client (ad esempio, 2 client sono 7 thread totali).

19.0.1 Il modificatore `abstract`

La parola chiave **abstract** viene adoperata per realizzare un modificatore per le definizioni di classi e per le definizioni di metodi. Il modificatore *abstract*:

- per i metodi, **non posso definire il corpo**, definisco solo la signature – è quasi identico all'interfaccia, ma sta in una classe e volendo può avere una visibilità minore, ad esempio `protected`;
- per la classe, **non posso istanziare la classe**. Inoltre, se esiste almeno un metodo `abstract` la classe **deve** essere `abstract`; può comunque essere `abstract` anche senza metodi `abstract`;
- ha la convenzione `Abstract`[Nome di classe].

Un esempio di utilizzo,

```
public abstract class AbstractFunction {
  public abstract double compute(double x); // should be implemented
  public double max(double[] xs) { // could be overridden
    double max = Double.NEGATIVE_INFINITY;
    for (double x : xs) {
      max = Math.max(compute(x), max);
    }
    return max;
  }
}
```

Parrebbe esserci una sovrapposizione fra utilizzo di interfacce e classi astratte. Esistono linee guida che aiutano nella scelta di quale utilizzare.

Abbiamo un diverso livello di astrazione:

- l'interfaccia è **totalmente astratta** (anche se non è del tutto vero);
- la classe astratta è **parzialmente astratta**.

Si possono addirittura **implementare** interfacce con **classi astratte**:

```
public abstract class CountingListener implements Listener {
  private int counter = 0;
  public void listen(Event event) {
    counter = counter + 1;
    innerListen(event);
  }
  public int count() { // out of convention, but generally accepted
    return counter;
  }
  protected abstract void innerListen(Event event);
}
```

In questo caso, voglio definire una cosa astratta, ma non al 100%; questa classe, infatti, ha bisogno di un contatore interno, con alcuni metodi definiti. Resta però da definire `innerListen`.

Posso definire uno `StandardOutputListener` che *estende* `CountingListener`. In questo caso, posso fare

```
Listener listener = new StandardOutputListener();
```

19.0.2 Le classi anonime

Esiste la possibilità di ridefinire al volo il metodo mancante è data da Java, mediante le **classi anonime** – esse permettono di creare classi "on-th-fly",

```
public class EventGenerator {
  private String prefix;
  public void doThings() {
    final PrintStream ps = /* ... */
    Listener listener = new Listener() {
       public void listen(Event event) {
         ps.println(prefix + event.toString());
       }
    };
  }
}
```

In questo caso, `listener` fa riferimento ad un oggetto appartenente ad una classe **anonima** che implementa `Listener`. Il metodo `ps.println()` non è eseguito in quella parte di codice.

Tale classe anonima ha un tipo, è presente nello heap con un "rettangolo d'istanza", e non è possibile chiamare un altro rettangolo con il medesimo nome.

Alla compilazione, il nome è come se venisse creato ed immediatamente dimenticato – non creiamo un identificatore per il tipo che stiamo definendo. Dentro i metodi della classe nuova sono vibili i metodi e i campi della classe in cui è racchiuso. Non si possono accedere a delle variabili locali, identificatore inline, a meno che non siano `final` o "effettivamente" `final` (si potrebbe mettere `final` anche se non è stato applicato).

Al runtime, non vi sono differenze: è una classe qualsiasi, **con un nome**, tipicamente della forma `Main$1`, con '1' un numero che rappresenta la sequenzalità delle definizioni, ad esempio nel caso sopra sarebbe `EventGenerator$1`.

Con questa sintassi, diventa ancora più semplice l'utilizzo del `LineProcessingServer`:

```
// before
public class LineProcessingServer {
  /* ... */
  public LineProcessingServer(int port, String quitCommand,
      CommandProcessor processor) {
    /* ... */
  }
  /* ... */
}


// Using with anonymous class definition:

LineProcessingServer server = new LineProcessingServer(10000,
    "BYE", new CommandProcessor() {
  public String process(String input) {
    return input.toUpperCase();
  }
});
```

Le interfacce **possono non essere completamente astratte**. I metodi con modificatori `default` e `static` hanno le seguenti caratteristiche, da Java 8:

- `default`, una nuova **parola chiave** – i metodi forniscono un'**implementazione di default**;
- `static` – possono fornire metodi statici, come per le classi regolari.

Un esempio ragionevole,

```
public interface Listener {
  void listen(Event event);
  default void listen(Event[] events) {
    for (Event event : events) {
      listen(event);
    }
  }
  default Listener andThen(final Listener other) { // then is a
      keyword
    final Listener thisListener = this;
    return new Listener() {
      public void listen(Event event) {
        thisListener.listen(event);
        other.listen(event);
      };
    }
  }
  static Listener nullListener() { // null is a keyword
    return new Listener() {
      public void listen(Event event) {/* do nothing */ };
    }
  }
}
```

con il `nullListener()` che funge da pseudo–costruttore. L'utilizzo può essere il seguente,

```
Listener listener = Listener.nullListener(); // if I don't care of
    logging
Listener listener = new StandardOutputListener()
                        .andThen(new FileListener(file)); // combined
                           logging
```

Le interfacce **non possono avere uno stato**, non possono dunque avere field – le classi astratte invece sì. L'altra differenza, a vantaggio delle interfacce, è che **non è necessario ricompilare le classi che implementano un'interfaccia, qualora essa veda aggiunte nuove funzionalità**.

Un altro esempio,

```
public interface Listener {
  void listen(Event event);
  /* ... */
  static Listener nonBlocking(final Listener inner) {
    return new Listener() {
      public void listen(final Event event) {
        Thread t = new Thread() {
          public void run() {
            inner.listen(event);
          }
        }
        t.start();
      }
    }
  }
}
```

Ogni volta che devo esaudire una `listen`, lancio un nuovo thread e risolvo lì. L'opzione numero uno è un metodo `static` che dato un `Listener` ne restituisce un altro. Essa crea un `Listener` come classe anonima, crea un nuovo thread dove viene definito il `run()`, e subito lanciato. Questo può essere utile per godere del beneficio di un **listener asincrono**. Un suo utilizzo,

```
Listener listener = Listener.nonBlocking(
    new StandardOutputListener().andThen(new FileListener(pathName))
);
EventGenerator generator = new EventGenerator(listener);
generator.start();
```

Un'altra possibilità è farlo con un **metodo di default**, dove si decide che ciascun `Listener` può creare una sua versione `nonBlocking()` tramite uno pseudo-costruttore.

I metodi definiti nelle interfacce, sia statici che default, **non possono usare qualsiasi field**, poiché non sono definiti. Nelle classi `abstract` posso aggiungere funzionalità, ma costringo le classi che la estendono a ricompilare – diversamente, nelle interfacce posso aggiungere funzionalità che la estendono senza ricompilare (**retrofitting**).

19.0.3 Il tipo `enum`

Il tipo **enum** è un tipo che rappresenta un insieme di valori **predefinito dallo sviluppatore**. Un'istanza di `enum` corrisponde ad uno di tali valori, *categorici*,

```
public enum Gender {
  MALE, FEMALE, OTHER;
}
public class Person {
  private String firstName;
  private String lastName;
  private final Date birthDate;
  private Gender gender;
}
```

La naming convention è la medesima che per le classi (*upper camel case*), per i valori si adoperano *tutte maiuscole con l'underscore* `_`, come per le costanti. Per i riferimenti, come in generale, si adopera la *lower camel case*. Esempio di utilizzo:

```
public class Person {
  public String toString() {
    String prefix = "";
    if (gender.equals(Gender.FEMALE)) {
      prefix = "Ms. ";
    } else if (gender.equals(Gender.MALE)) {
      prefix = "Mr. ";
    }
    return prefix + firstName + " " + lastName;
  }
}
```

Per gli `enum`, l'operatore binario `==` funziona come `equals()`. Non essendo classi, non posso fare `new Gender()` – non hanno costruttori, poiché hanno poche possibilità. Con lo `switch`,

```
public class Person {
  public String toString() {
    String prefix;
    switch (gender) {
      case FEMALE:
        prefix = "Ms. ";
        break;
      case MALE:
        prefix = "Mr. ";
        break;
      default:
        prefix = "";
```

```
    }
    return prefix + firstName + " " + lastName;
  }
}
```

dove non è necessario specificare `Gender.MALE` ed è sufficiente specificare `MALE`.

I tipi `enum` possono avere metodi, field, costruttori – tuttavia, i costruttori devono poter essere chiamati con i valori che li definiscono,

```
public enum Gender {
  MALE("Mr."), FEMALE("Ms."), OTHER("");
  private final String prefix;
  Gender(String prefix) {
    this.prefix = prefix;
  }
  public String getPrefix() {
    return prefix;
  }
  public String prefix(String string) {
    return prefix.isEmpty() ? string : prefix + " " + string;
  }
}
public String toString() {
  return gender.prefix(firstName + " " + lastName);
}
```

e nel codice di sopra si comprende come il tutto dipenda comunque dalla scelta del `Gender`, ad esempio il codice cambierà fra `MALE`, `FEMALE`, o `OTHER`. Essendo oggetti, derivano tutti da `Object`, pertanto godono del metodo `toString()` ed `equals()`, come visto in precedenza.

Gli `enum` non si possono estendere, tuttavia estendono (forse) interfacce.

19.0.4 Le Annotazioni

Le **annotazioni** sono *annotazioni*. Hanno una loro sintassi che serve per *annotare* definizioni di classi, metodi e field, oltre che ad altre cose. Possono avere *argomenti*, possono essere definite e possono essere oservabili, mantenute e percepite

- a tempo di compilazione;
- al runtime;
- ad entrambi;
- da nessuna parte.

Ad esempio, l'annotazione `@Override`;

```
public class StandardOutputListener implements Listener {
  @Override
  public void listen(Event event) {
    System.out.println(event);
  }
}
```

che dice al compilatore che questo metodo fa l'override di un'altra definizione. Si usa mettere, appunto, durante l'override di un metodo da parte di una classe, nei confronti dell'*estensione* di un'altra classe o classe astratta. Il compilatore **non richiede** l'applicazione dell'annotazione.

Le *annotation type* si definiscono come *interfacce*:

> Annotation Type Override
>
> @Target(METHOD) @Retention(SOURCE)
>
> public @interface Override

> Indicates that a method declaration is intended to override a method declaration in a supertype. If a method is annotated with this annotation type compilers are required to generate an error message unless at least one of the following conditions hold: The method does override or implement a method declared in a supertype. The method has a signature that is override-equivalent to that of any public method declared in Object.

19.1 I Lambda

I **lambda** sono una scorciatoia sintattica in cui viene modellata un'interfaccia che offre *un'unica funzionalità tramite un singolo metodo* – ad esempio, un `Listener` o un `Processor`. Esistono anche interfacce tipo `Runnable()`; esiste un costruttore di `Thread` che prende un `Runnable` e di cui all'invocazione di `start()` ne esegue il `run()`.

I lambda sono detti essere **functional interfaces** – esse modellano un singolo metodo, una singola capacità, senza uno stato proprio. Implementare questa funzionalità sarebbe molto verboso, e da come abbiamo visto si farebbe così,

```
LineProcessingServer server = new LineProcessingServer(
    10000, "BYE", new CommandProcessor() {
      public String process(String input) {
        return input.toUpperCase();
      }
    });
```

Gli sviluppatori di Java (da Java 8) hanno proposto un'aggiunta di una scorciatoia sintattica per definire una *functional interface* ove sia **implementato un unico metodo di un'interfaccia**. Si fa così, con l'annotazione `@FunctionalInterface`,

```
@FunctionalInterface
public interface CommandProcessor {
  String process(String input);
}
CommandProcessor p = (String input) -> {
  return input.toUpperCase()
};
LineProcessingServer server = new LineProcessingServer(10000,
    "BYE", p);
```

La sintassi principale è:

- lista degli argomenti fra parentesi tonde `()`, può essere vuota;
- freccina `->`;
- corpo del metodo.

Non c'è bisogno di mettere la signature del metodo – è l'unico per l'interfaccia. Nemmeno il nome dell'interfaccia è richiesto (il compilatore lo comprende dal contesto). L'annotazione può essere omessa – tuttavia, il suggerimento è di metterla nel caso sia davvero una *functional interface*. Dalla documentazione,

> An informative annotation type used to indicate that an interface type declaration is intended to be a functional interface as defined by the Java Language Specification. Conceptually, a functional interface has exactly one abstract method. Since default methods have an implementation, they are not abstract. If an interface declares an abstract method overriding one of the public methods of java.lang.Object, that also does not count toward the interface's abstract method count since any implementation of the interface will have an implementation from java.lang.Object or elsewhere.
>
> Note that instances of functional interfaces can be created with lambda expressions, method references, or constructor references.
>
> If a type is annotated with this annotation type, compilers are required to generate an error message unless:
>
> The type is an interface type and not an annotation type, enum, or class. The annotated type satisfies the requirements of a functional interface. However, the

compiler will treat any interface meeting the definition of a functional interface as a functional interface regardless of whether or not a FunctionalInterface annotation is present on the interface declaration.

La sintassi si può compattare ulteriormente. Ad esempio, **il tipo degli argomenti si può omettere**, poiché il compilatore riesce a capirlo dal contesto:

```
CommandProcessor p = (input) -> {
  return input.toUpperCase()
};
```

se c'è un solo argomento, posso omettere le parentesi tonde:

```
CommandProcessor p = input -> {return input.toUpperCase()};
If the body is composed by exactly one statement (possibly return),
    brackets {} can be omitted (with return, if present):
```

infine, se c'è un singolo statement, posso rimuovere il return e le parentesi graffe,

```
CommandProcessor p = input -> input.toUpperCase();
```

È possibile addirittura utilizzare un altro metodo da un'altra classe:

```
CommandProcessor p = StringUtils::reverse;
```

in pratica, se lo statement è l'invocazione di un metodo senza alcun argomento, oppure di un metodo con un argomento (ma con corrispondenza di tipi) si può ridurre ancora il tutto con la **method reference operator** notation come sopra. Dunque, supponendo esista `StringUtils`,

```
public class StringUtils {
  public static String reverse(String string) {
    /* ... */
    return reversed;
  }
}
```

Nella pratica,

```
//Convert to uppercase:
server = new LineProcessingServer(10000, "BYE", s ->
    s.toUpperCase());

//Prefix:
final String prefix = /* ... */;
server = new LineProcessingServer(10000, "BYE", s -> prefix + s);

//Count tokens:
server = new LineProcessingServer(10000, "BYE", s -> s.split("
    ").length);
```

Un esempio più complicato, l'idea di *funzione univariata*:

```
@FunctionalInterface
public interface RealFunction {
  double apply(double x);
  default RealFunction composeWith(RealFunction other) {
    return x -> other.apply(apply(x));
  }
  default RealFunction integrate(double x0, double step) {
    return x -> {
      double sum = 0;
```

```
      for (double xv = x0; xv <= x; xv = xv + step) {
        sum = sum + step * apply(xv);
      }
      return sum;
    };
  }
}
```

Ad esempio, f `.composeWith(`g`)` restituisce $f \circ g$ con $(f \circ g)(x) = g(f(x))$. Invece, l'integrale f `.integrate(`$x_0, \Delta x$`)` fornisce

$$\sum_{t \in x_0, x_0 + \Delta x, \ldots, x} f(t)\Delta x,$$

(metodo dei rettangoli).

Si osservi che entrambi i metodi ritornano una funzione lambda!

Un possibile utilizzo,

```
public class RealFunctionUtils {
  public static double max(RealFunction f) { /* ... */ };
  public static double[] zeros(RealFunction f) { /* ... */ };
}
double[] zeros = RealFunctionUtils
    .zeros(x -> (x * x + 1) / (1 + x)); // computes zeros of this
        function
RealFunction f = (x -> x + 1);
// calculates max of g(f(x)) where f is
// an integration of function t + 1
// and g is x^2 + 1
double max = RealFunctionUtils.max(f
    .integrate(-1, 0.1)
    .composeWith(x -> x * x + 1));
```

Le funzioni lambda sono *oggetti di primaria importanza*, diversamente dai metodi, che non hanno dignità primaria, ed esistono solo in qualità di azioni che un determinato oggetto può compiere. Gli oggetti modellano entità, mentre le funzioni lambda modellano *step di processamento*. Il modo di programmare che ruota attorno alle funzione è detto chiamarsi **functional programming**. Per rendere attrattivo Java come opzioni di programmazione, è stato deciso di aggiungere l'opzione delle lambda functions, le quali hanno anche il pregio di ridurre la verbosità del codice. Nel pacchetto`java.util.stream` è presente un grande utilizzo delle funzioni lambda.

Capitolo 20

I Generics in Java

I **Generics** sono l'ultima innovazione della sintassi del linguaggio, che consente di gestire le *collezioni*. Molto spesso abbiamo a che fare con *tipi* (classi) che sono *costruiti su altri tipi*. In particolare, supponiamo di voler creare una classe che modella un *insieme di stringhe*, che consenta oltretutto di aggiungere una stringa all'insieme, verificarne la presenza nell'insieme e contare il numero delle stringhe nell'insieme; vogliamo che le operazioni siano utilizzabili come di seguito,

```
SetOfStrings set = new SetOfStrings();
set.add("hi");
set.add("world");
set.add("hello");
set.add("world");
System.out.println(set.size()); // -> 3
System.out.println(set.contains("dog")); // -> false
```

e potrebbe essere fatta così:

```
public class SetOfStrings {
  private String[] strings = new String[0];
  public void add(String toAddString) {
    if (!contains(toAddString)) {
      String[] newStrings = new String[strings.length + 1];
      System.arraycopy(strings, 0, newStrings, 0, strings.length);
      newStrings[newStrings.length - 1] = toAddString;
      strings = newStrings;
    }
  }
  public boolean contains(String otherString) {
    for (String string : strings) {
      if (string.equals(otherString)) {
        return true;
      }
    }
    return false;
  }
  public int size() {
    return strings.length;
  }
}
```

dove lo stato è codificato con un array di stringhe – ogni volta che si intende aggiungere una stringa, prima si verifica che sia già contenuta, poi si "ricrea" l'array con la nuova stringa.

La classe per l'insieme di persone sarà praticamente identica a `SetOfStrings`, eccezion fatta per i tipi (`Person` array). Tipicamente, si generano tantissime classi che *non sono generiche*, in quanto vanno ridefinite ogni volta. L'opzione migliore è quella di fare il design di una classe **generica** `Set`, che prende "oggetti" generici. Le funzionalità base di `SetOfStrings` non dipendono da alcuna funzionalità particolare di `String` o altri oggetti – la funzionalità chiave è `equals()`, che è presente in qualsiasi `Object`.

Dunque,

```
public class Set {
  private Object[] items = new Object[0];
  public void add(Object toAddItem) {
    if (!contains(toAddItem)) {
      Object[] newItems = new Object[items.length + 1];
      System.arraycopy(items, 0, newItems, 0, items.length);
      newItems[newItems.length - 1] = toAddItem;
      items = newItems;
    }
  }
  public boolean contains(Object otherItem) {
    for (Object item : items) {
      if (item.equals(otherItem)) {
        return true;
      }
    }
    return false;
  }
  public int size() {
    return items.length;
  }
}
```

Ora però, il problema è solo parzialmente risolto – anche se ora posso istanziare sia `Person` che `String`, in quanto non ho dipendenze particolari da metodi o field specifici di classi. Ad esempio, nel codice precedente,

```
SetOfStrings strings = new SetOfStrings();
strings.add("hi");
strings.add(new Person()); // does not compile!
```

l'ultimo statement non compila. Ora invece,

```
Set persons = new Set();
persons.add(new Person());
persons.add("surprise!"); // does compile!!!
```

compila. Questa era (fino a Java 1.5) una vera e propria limitazione del linguaggio – noi vorremmo che se abbiamo fatto `add()` con `T`, allora anche `contains()` deve funzionare anche con `T`: questo non era garantito imponendo solo oggetti di tipo `Object`. Dunque, qualsiasi tipo va bene, purché sia sempre quello durante la vita dell'oggetto.

La soluzione sono i **Generic types** – un tipo definito con uno o più tipi per parametro, ad esempio "un insieme di stringhe" – l'insieme è il tipo, il parametro è "di stringhe".

In informatica, l'idea dei tipi generici è molto datata – circa dagli anni 70. Le ragioni per il ritardo della sua implementazione in Java sono la complessità di scrittura del compilatore; tuttavia, gli sviluppatori hanno preferito introdurlo per via del *collection framework*.

Un esempio appropriato di come si implementano i Generics,

```
public class Set<T> {
```

```
  public void add(T toAddItem) { /* ... */ }
  public boolean contains(T otherItem) { /* ... */ }
  public int size() { /* ... */ };
}
```

che implementa un set di elementi di tipo omogeneo (`T` è il tipo). La classe è definita in modo parametrico (con le *parentesi acute* – **diamond operator**) e `T` è il parametro che indica il tipo con cui è definita la classe.

Tipicamente, la naming convention è una singola lettera uppercase, la prima lettera di un nome adatto per il parametro.

Il tipo `T` deve rappresentare un tipo *non primitivo* – può essere usato ogni volta che la definizione di un tipo può essere adoperato: definizione di field, di argomenti, di riferimenti, nella definizione del tipo di ritorno. Tuttavia, `T` non può essere utilizzato come costruttore – non è detto che `T()` sia un costruttore valido – e parimenti `new T[]` non compila. Non può essere un tipo primitivo. `T` può essere anche un enum.

L'utilizzatore di `Set` può voler utilizzare la classe generica in questo modo,

```
Set<String> strings = new Set<String>();
strings.add("hi");
Set<Double> doubles = new Set<Double>();
doubles.add(3.14d);
// compiles

Set<String> strings = new Set<String>();
strings.add(3d);
strings.add(new Person("Eric", "Medvet"));
// does not compile
```

In questo caso, `String` e `Double` sono valori del parametro `T` – devono essere tipi veri e propri, oppure, se un tipo di parametro è definito nel contesto, un tipo di parametro.

Inoltre,

```
//Does not compile!
Set<double> numbers = new Set<double>();

// Use wrapper classes:
Set<Double> numbers = new Set<Double>(); //compiles!

// Arrays are types!
Set<double[]> numbers = new Set<double[]>(); //compiles!
Set<String[]> sentences = new Set<String[]>(); //compiles!
```

dove gli array di tipi primitivi vanno bene, poiché non sono tipi primitivi.

Esiste una *scorciatoia* che permette di non specificare il tipo nelle parentesi acute quando esso è *evidente* per il compilatore,

```
Set<String> strings = new Set<>();
strings.add("hi"); // compiles!
```

Perciò, usando i Generics,

```
public class Set<T> {
  private Object[] items = new Object[0];
  public void add(T toAddItem) {
    if (!contains(toAddItem)) {
      Object[] newItems = new Object[items.length + 1];
      System.arraycopy(items, 0, newItems, 0, items.length);
      newItems[newItems.length - 1] = toAddItem;
```

```
      items = newItems;
    }
  }
  public boolean contains(T otherItem) {
    for (Object item : items) {
      if (item.equals(otherItem)) {
        return true;
      }
    }
    return false;
  }
  public int size() {
    return items.length;
  }
}
```

non compare `T[]` e c'è `Object[]` poiché non possiamo definire (o istanziare) `T[]`. Per via della modularità, al compile time sono generici oggetti, al runtime sono il tipo che deve essere. Tuttavia, ciò non è visibile da fuori (modularità del software, information hiding).

Voglimo modellare il generico concetto di funzione da A a B. Una **functional interface** può essere modellata in questo modo,

```
@FunctionalInterface
public interface Function<A, B> {
  B apply(A a);
}
```

Questa interfaccia restituisce B dato A tramite un metodo da definire. Esempi di `Function` potrebbero essere

```
Function<Double, Double> squarer = new Function<>() {
  @Override
  public Double apply(Double x) {
    return x * x;
  }
};
double y = squarer.apply(2); // -> 4

Function<String, String> shortener = new Function<>() {
  @Override
  public String apply(String s) {
    String acronym = "";
    for (String token : s.split(" ")) {
      acronym = acronym + token.substring(0, 1).toUpperCase();
    }
    return acronym;
  }
};
String shortened = shortener.apply("Eric Medvet"); // -> EM

Function<String, Integer> tokenCounter = s -> s.split(" ").length;
int nOfTokens = tokenCounter.apply("I love generics!"); // -> 3

Function<String, Integer> length = String::length;
length.apply("wow"); // -> 3
```

È possibile adoperare la notazione dei Generics anche per *parametrizzare un singolo metodo* – a volte si vuole definire metodi che funzionano per qualsiasi tipo, ma che non sia legato al tipo generico della classe. Ad esempio,

```
@FunctionalInterface
public interface Function<A, B> {
  B apply(A a);
  default <C> Function<A, C> composeWith(Function<B, C> other) {
    return a -> other.apply(apply(a));
```

```
  }
}
```

e `C` è visibile esclusivamente all'interno della signature e del body di `composeWith`, mentre `A` e `B` sono visibili dovunque all'interno dell'interfaccia d'esempio. Un possibile utilizzo,

```
int n = tokenCounter.composeWith(n -> n*n).apply("Eric Medvet");
```

e crea una composizione di funzioni da stringhe a interi e poi da interi ad interi (facendo il quadrato del numero di token, in questo caso 2, quindi il risultato è 4).

Un Generic type è possibile adoperarlo **anche senza specificare il tipo `T`** – in questo caso si sottintende che l'oggetto in questione è `Object`. Dunque,

```
Set set = new Set();
set.add("hi");
set.add(3.14); //autoboxing
set.add(new Object());
```

anche se può portare a problemi – il compilatore (o l'IDE) ci avvisa (la tipizzazione forte responsabilizza lo sviluppatore in fase di compilazione).

```
Set<Object> set = new Set<>();
set.add("hi");
set.add(3.14); //autoboxing
set.add(new Object());
```

in questo caso, responsabilmente, il `Set` è adoperato al massimo della capacità di generalizzazione.

I Generics possono sposarsi con l'ereditarietà, sia dal punto di vista per cui un oggetto parametrizzato può essere esteso, sia dall'altro punto di vista per cui è possibile *vincolare* i tipi d'ereditarietà. Inoltre, in compilazione l'ereditarietà può essere adoperata per un tipo parametrizzato. Dunque,

```
public class Person { /* ... */ }
public class Worker extends Person { /* ... */ }
Set<Person> persons = new Set<>();
persons.add(new Person("Eric", "Medvet"));
persons.add(new Worker("Bruce", "Wayne", "businessman")); // -> OK
```

e inoltre, nella definizione,

```
public interface Container<T> {
  void add(T t);
  T getOne();
}
public class ArrayContainer<T> implements Container<T> {
  /* ... */
}
public interface LimitedContainer<T> extends Container<T> {
  boolean hasSpace();
}
```

In queste due definizioni, `<T>` appare due volte – nella prima è una *definizione* di tipo parametrico, nel secondo è un *primo utilizzo* di `T`. Dunque, devono in pratica essere la stessa lettera.

È anche possibile definire nuovi tipi, interfacce o classi che siano, *valorizzando uno o più parametri generici*. Ad esempio,

```
public class StringContainer implements Container<String> {
```

```
  /* ... */
}
```

ed il placeholder `<T>` è stato **valorizzato** in `String`.

La relazione ereditaria fra tipi non generici non è sempre semplice:

```
// Assume:
class Worker extends Person {}
interface LimitedContainer<T> extends Container<T> {}
class ArrayContainer<T> implements Container<T> {}
class LimitedArrayContainer<T> implements LimitedContainer<T> {}

// Ok:
Container<Person> persons = new ArrayContainer<>();
Container<Person> persons = new LimitedArrayContainer<>();

// Not ok (does not compile):
Container<Person> persons = new ArrayContainer<Worker>();
Container<Person> persons = new LimitedArrayContainer<Worker>();
```

Gli ultimi due statement non compilano, questo perché come scelta di design è stato deciso così.

Come raccomandazione generale, è meglio usare **sempre il tipo più generale** che sia abbastanza specifico, ad esempio

```
Container<Person> persons = new ArrayContainer<>();
```

serve ad indicare che tutto il codice che segue deve adoperare solo operazioni di `Container` su `persons`. Qua invece,

```
ArrayContainer<Person> persons = new ArrayContainer<>();
```

non si fa, e si danno vincoli più stretti a cosa deve essere `persons`.

A volte è necessario **vincolare i possibili valori di un tipo generico**. Dunque, si fa così:

```
public interface Union<W extends Worker> {
  void register(W worker);
  W[] listAlphabetically();
}
```

dove `W` deve essere almeno un `Worker`. Spesso e volentieri, si adopera `extends` con le `interface`. Ad esempio,

```
public class ComplexStuff<T extends Doer> { /* ... */ }
```

dove anche per le interfacce si usa `extends` (**attenzione**, poiché `T` deve essere almeno un `Doer`, cioè estendere un'interfaccia).

Di solito, in alcuni casi qualsiasi tipo è okay. Nella signature (**non nella definizione**), però, si adopera il cosiddetto **wildcard** `<?>`. Dunque,

```
public static int count(Set<?> set) { /* ... */ }

// no need to know the type for counting
// it suffices to be a type that extends Sized
public static void removeShorterThan(
  Set<? extends Sized> set,
  double size
) { /* ... */
```

Le linee guida per quando usare i generici sono le seguenti: i Generics sono adoperati qualora un qualcosa (`X`) deve modellare in base ad un tipo `Y`. Dunque, sia un'interfaccia `Y` che dichiara alcune funzionalità – `X` definisce alcune funzionalità basate sulle funzionalità di `Y`. Tramite i Generics offriamo un modello di secondo ordine (superiore) basato su qualsiasi cosa avente le capacità di `Y`, tramite la *separation of concerns*.

Il primo caso è quello della `interface Function<T,R>`,

> Module java.base Package java.util.function Interface Function<T,R>
>
> Type Parameters: T - the type of the input to the function R - the type of the result of the function
>
> All Known Subinterfaces: UnaryOperator<T>
>
> Functional Interface: This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.
>
> @FunctionalInterface public interface Function<T,R> Represents a function that accepts one argument and produces a result. This is a functional interface whose functional method is apply(Object).
>
> Since: 1.8

Si guardino i metodi nella documentazione ufficiale. Ad esempio,

> default <V> Function<T,V> andThen(Function<? super R,? extends V> after) Returns a composed function that first applies this function to its input, and then applies the after function to the result.

La sottointerfaccia `UnaryOperator<T> extends Function<T,T>` e modella un'interfaccia che restituisce un insieme che

> Represents an operation on a single operand that produces a result of the same type as its operand. This is a specialization of Function for the case where the operand and result are of the same type.

L'altro caso è l'interfaccia `Comparable<T>`. Dalla documentazione (importante),

> Module java.base Package java.lang Interface Comparable<T>
>
> public interface Comparable<T> This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method. Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or as elements in a sorted set, without the need to specify a comparator.
>
> The natural ordering for a class C is said to be consistent with equals if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class C. Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.
>
> It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals method.
>
> For example, if one adds two keys a and b such that (!a.equals(b) && a.compareTo(b) == 0) to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because a and b are equivalent from the sorted set's perspective.
>
> Virtually all Java core classes that implement Comparable have natural orderings that are consistent with equals. One exception is java.math.BigDecimal, whose natural ordering equates BigDecimal objects with equal values and different precisions (such as 4.0 and 4.00).
>
> For the mathematically inclined, the relation that defines the natural ordering on a given class C is:
>
> (x, y) such that x.compareTo(y) <= 0.
>
> The quotient for this total order is:
>
> (x, y) such that x.compareTo(y) == 0.

It follows immediately from the contract for compareTo that the quotient is an equivalence relation on C, and that the natural ordering is a total order on C. When we say that a class's natural ordering is consistent with equals, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's equals(Object) method: (x, y) such that x.equals(y). This interface is a member of the Java Collections Framework.

Since: 1.2 See Also: Comparator

`Comparable<T>` non è una functional interface, poiché non modella una componente in grado di fare un processamento ma che non ha uno stato, ed esprime infatti una dipendenza dallo stato di `T`.

L'**ordinamento totale**, ha tre proprietà – antisimmetria, transitività, connettività:

- $a \leq b \land b \leq a \Rightarrow a = b$ (antisimmetria);
- $a \leq b \land b \leq c \Rightarrow a \leq b$ (transitività);
- $a \leq b \lor b \leq a \Rightarrow a = b$ (connettività);

dunque, l'output del metodo di `interface Comparable<T>`, è TODO 593

Esiste un'interfaccia `Comparator<T>` che è simile a `Comparable<T>`, ma che utilizza una comparazione "naturale" dell'oggetto, cioè già definita. In particolare, `Comparator<T>` è una functional interface, poiché esprime un'entità che agisce a prescindere dallo stato. Esso ha un metodo `int compare(T t1, T t2)` che è utile per comparare oggetti che non implementano `Comparable`. Esso è anche dotato di molti metodi `default` utili. Si legga la documentazione nella slide 595 o su internet.

Definiamo l'ordinamento naturale per delle persone, in base al cognome. Dunque, facciamo implementare `Comparable<Person>` a `Person`:

```
public class Person implements Comparable<Person> {
  private final String firstName;
  private final String lastName;
  private final Date birthDate;
  /* getters */

  @Override
  public int compareTo(Person other) {
    return lastName.compareTo(other.lastName); // not null safe
  }
}
```

ora vogliamo comparare le persone per cognome, in caso di parità per data di nascita, in caso di parità per cognome, alla fine per nome. Dunque,

```
Comparator<Person> c = Comparator
    .<Person>naturalOrder() // parameter is passed before its
        identifier
    .thenComparing(Person::getBirthDate)
    .thenComparing(Person::getFirstName)
    .reversed();
```

Per riassumere, `Comparable<T>` dichiara che per un tipo esiste un ordine naturale; `Comparator<T>` è una componente in grado di effettuare un ordinamento totale.

Capitolo 21

Le Java Collections

Le **collections** modellano il concetto di *collezione*, un gruppo di elementi omogeneo – e concetti derivati. Dunque, un insieme di elementi, una lista, una coda sono tutti appartenenti alle collezioni. Nella JDK, le **Java collection framework** sono un insieme di *interfacce* che modellano i concetti chiave con operazioni, e con proprietà implicite. Inoltre, forniscono un insieme di *classi* che implementano queste interfacce.

Per le interfacce, il concetto modellato è catturato dal nome stesso. Le classi, invece, differiscono per *eventuali proprietà aggiuntive*, *comportamento nel caso di multithreading* (thread-safe vs thread-unsafe), e *prestazioni* (qualche classe potrebbe essere più veloce con l'implementazione di alcuni metodi).

Le interfacce principali sono le seguenti, `interface Collection<E>` per le **collezioni** e `interface Map<K,V>` per le **mappe**. Le `Collection<E>` sono una collezione che rappresenta un gruppo di oggetti, nota per i suoi elementi. Dunque, `E` sta per **elementi**. Una mappa invece è un oggetto che mappa chiavi a valori. Una mappa non può contenere chiavi duplicate – ciascuna chiave può mappare al più un singolo valore. Dunque, `K` sta per **chiave**, `V` sta per **valore**. Questa mappa è generalmente chiamata **dizionario**.

Le collezioni e le mappe sono disgiunte rispetto all'ereditarietà, e `Map` non è precisamente una collezione (ciononostante, è parte della Java collection framework).

L'interfaccia `Collection<E>` ha alcuni metodi importanti,

> **`boolean add(E e)`** Ensures that this collection contains the specified element (optional operation).
>
> **`boolean contains(Object o)`** Returns true if this collection contains the specified element.
>
> **`boolean remove(Object o)`** Removes a single instance of the specified element from this collection, if it is present (optional operation).
>
> **`int size()`** Returns the number of elements in this collection.

che definiscono le proprietà elementari delle collezioni.

I metodi `add()` e `remove()` sono detti **metodi mutatori**, poiché potenzialmente modificano lo stato *del gruppo*. Il booleano come valore di ritorno specifica se la collezione è mutata per davvero.

I metodi `contains()` e `remove()` prendono `Object` per ragioni storiche, anche se può avere senso domandarsi se altri tipi sono presenti all'interno della collezione (anche se è ovvio che non potranno essere presenti).

Oltre alle operazioni fondamentali, esistono le **bulk operations** – per eseguire operazioni su più elementi contemporaneamente:

`boolean addAll(Collection<? extends E> c)` Adds all of the elements in the specified collection to this collection (optional operation).

`boolean containsAll(Collection<?> c)` Returns true if this collection contains all of the elements in the specified collection.

`boolean removeAll(Collection<?> c)` Removes all of this collection's elements that are also contained in the specified collection (optional operation).

`boolean retainAll(Collection<?> c)` Retains only the elements in this collection that are contained in the specified collection (optional operation).

Alcuni metodi "derivati" (ma ridefiniti) sono

`void clear()` Removes all of the elements from this collection (optional operation). `boolean isEmpty()` Returns true if this collection contains no elements.

Una collection può essere convertita in array, dunque

`Object[] toArray()` Returns an array containing all of the elements in this collection.

`default <T> T[] toArray(IntFunction<T[]> generator)` Returns an array containing all of the elements in this collection, using the provided generator function to allocate the returned array.

`<T> T[] toArray(T[] a)` Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

`toArray()` restituisce un array di oggetti `Object[]`, non `T[]`, perché non si può fare! Al giorno d'oggi si usa il secondo metodo, che fa uso di una funzione che genera un array di `T[]` (tramite construction reference di array). TODO 607-608

Ci sono 4 modi per iterare su una collezione:

1. `toArray()` e poi un ciclo `for` per iterare – antico;
2. `iterator()` – molto vecchio, restituisce un iteratore sulla lista;
3. tramite un `for-each` (se non serve l'indice) – la maniera corretta e attuale per farlo;
4. tramite gli *stream*.

Supponiamo di avere una `Collection<Person>`,

```
Collection<Person> persons = /* ... */
Person[] personArray = persons.toArray(new Person[persons.size()]);
// alternativa
// rappresenta una IntFunction<Person[]>
// Person[] apply(int n) {
//     return new Person[n];
// }
// dunque,
// Person[] personArray = persons.toArray(Person[]::new);
for (int i = 0; i < personArray.length; i++) {
  Person person = personArray[i];
  // do things
}
```

In questa maniera (antica) stiamo iterando su una *copia* della collezione, piuttosto che nella collezione stessa – la collezione vera e propria potrebbe subire cambiamenti durante l'iterazione, nonostante l'array punti agli elementi "veri" nel diagramma oggetti-riferimenti (si creano soltanto nuovi riferimenti, non su nuovi oggetti – sto iterando su una **snapshot** della collection, sotto forma di array).

Riguardo la naming convention – la stessa degli array (persons, nicePersons, population, axis → axes).

Esiste un metodo di `interface Collection<E>`,

`Iterator<E> iterator()` Returns an iterator over the elements in this collection.

che ha i metodi

> `boolean hasNext()` Returns true if the iteration has more elements.
>
> `E next()` Returns the next element in the iteration.

dunque, il secondo tipo di iterazione si fa così:

```
Collection<Person> persons = /* ... */
Iterator<Person> iterator = persons.iterator();
while (iterator.hasNext())
  Person person = iterator.next();
  // do things
}
```

Un'interfaccia che rappresenta cose "iterabili" è `interface Iterable<T>`, dunque `interface Collection<E> extends Iterable<E>`. Una collezione estende un iterable, poiché vi aggiunge il concetto di *lunghezza*, è possibile *aggiungervi oggetti*, e offre nuove funzionalità che vanno oltre al concetto di essere "iterabile".

La modalità preferibile è la **for-each**:

```
Collection<Person> persons = /* ... */
for (Person person : persons) {
  // do things
}
```

e il compilatore lo traduce nella maniera con l'`iterator()`.

Importante: non bisogna **mai** assumere che, iterando su una collezione, il risultato abbia uno specifico ordine – nemmeno che l'ordine uscito sia ripetibile! Le collezioni implicano che **l'ordine non è importante**. Tutto ciò, a meno che non sia una proprietà dell'attuale *tipo* della collezione,

```
Collection<String> strings = /* ... */
strings.add("hi");
strings.add("world");
for (String string : strings) {
  System.out.println(string);
}
```

potrebbe risultare in `hi`, `world`, oppure in `world`, `hi`.

Finora abbiamo `Collection<E>` che eredita da `Iterable<E>`. Ora vediamo un'altra estensione di `Collection<E>`,

Un'importante interfaccia che è un'estensione delle collezioni è `Set<E>`,

> `interface Set<E> extends Collection<E>`
>
> A collection that contains no duplicate elements. More formally, sets contain no pair of elements e1 and e2 such that e1.equals(e2), and at most one null element. As implied by its name, this interface models the mathematical set abstraction.

dunque,

```
Set<String> names = /* ... */
names.add("Eric");
names.add("Pippi");
names.add("Eric");
System.out.println(names.size()); // -> 2
```

in altre parole è proprio l'astrazione matematica di insieme – nessun duplicato, nessun ordinamento. Non ha metodi aggiuntivi rispetto a `Collection<E>`, con la differenza che i duplicati vengono individuati mediante un metodo `equals()`.

La documentazione afferma che non deve contenere duplicati. L'effettiva realizzazione sta nell'implementazione delle classi – nel metodo `add()`. Il

compilatore non può di certo verificare questa assunzione nell'interfaccia, sta dunque allo sviluppatore l'onere di implementare nella maniera corretta l'interfaccia.

Dunque, per esempio,

```
public class Person {
  private final String firstName;
  private final String lastName;
  public Person(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}

//In a Set:
Set<Person> persons = /* ... */
persons.add(new Person("Eric", "Medvet"));
persons.add(new Person("Eric", "Medvet"));
System.out.println(persons); // -> 2
```

Pertanto risulta sempre utilissimo ridefinire il metodo `equals()`, soprattutto nei casi in cui un oggetto debba essere adoperato all'interno di collezioni o insiemi.

Ogni qualvolta un'implementazione di `Set<E>` deve verificare se due oggetti sono uguali dovrebbe adoperare il metodo `equals()` – tuttavia, gli elementi sono suscettibili a **cambiamenti**: grande attenzione deve essere esercitata se elementi **mutabili** sono adoperati come elementi dell'insieme. Dunque, se il cambiamento produce variazioni al comportamento di `equals()` può essere pericoloso introdurre modifiche agli oggetti.

Un insieme che garantisce anche l'ordinamento è l'interfaccia

> **`interface SortedSet<E> extends Set<E>`**
>
> A Set that further provides a total ordering on its elements. The elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time. The set's iterator will traverse the set in ascending element order. Several additional operations are provided to take advantage of the ordering.

dunque, non vi sono duplicati, ma **c'è ordinamento naturale** per `E`, fornito da `Comparator<E>`, tipicamente fornito al momento della creazione. Alcuni metodi in più rispetto a `Set<E>`,

> **`E first()`** Returns the first (lowest) element currently in this set.
>
> **`SortedSet<E> headSet(E toElement)`** Returns a view of the portion of this set whose elements are strictly less than toElement.
>
> **`E last()`** Returns the last (highest) element currently in this set.
>
> **`SortedSet<E> subSet(E fromElement, E toElement)`** Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
>
> **`SortedSet<E> tailSet(E fromElement)`** Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

Dunque, come esempio,

```
SortedSet<String> names = /* ... */
names.add("Eric");
names.add("Pippi");
names.add("Eric");
names.add("Alice");
for (String name : names) {
  System.out.println(name);
}

//Gives:
//Alice
```

```
//Eric
//Pippi
```

Ora, estendiamo `Collection<E>` in una maniera particolarmente utile:

> **`interface List<E> extends Collection<E>`**
>
> An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.
>
> Unlike sets, lists typically allow duplicate elements.

in altre parole, modella il concetto matematico di *sequenza*, con possibili duplicati e un ordinamento basato su **come sono stati inseriti gli oggetti nella sequenza**, dunque non basato sull'ordinamento totale.

In altre parole, possiamo adoperare una `List<T>` al posto di un array `T[]`, con la possibilità di **ridimensionarla dinamicamente**.

Essa fa parte del pacchetto `java.util.List`, mentre esiste una seconda classe `List` in `java.awt.List` (advanced window toolkit) che "emerge spesso" nell'autocompletamento.

I metodi chiave per `List<E>` sono

> **`void add(int index, E element)`** Inserts the specified element at the specified position in this list (optional operation).
>
> **`boolean addAll(int index, Collection<? extends E> c)`** Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
>
> **`E get(int index)`** Returns the element at the specified position in this list.
>
> **`int indexOf(Object o)`** Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
>
> **`E remove(int index)`** Removes the element at the specified position in this list (optional operation).
>
> **`E set(int index, E element)`** Replaces the element at the specified position in this list with the specified element (optional operation).
>
> **`default void sort(Comparator<? super E> c)`** Sorts this list according to the order induced by the specified Comparator.
>
> **`List<E> subList(int fromIndex, int toIndex)`** Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

Esiste un metodo `sort()` che accetta un `Comparator<? super E>` (è sufficiente poter ordinare una superclasse di `E`) che modifica la lista applicando l'ordinamento. Una lista, infatti, non "gode intrinsecamente" di un ordinamento.

Un esempio di utilizzo,

```
List<String> knownJavaThings = /* ... */
knownJavaThings.add("interfaces");
knownJavaThings.add("class");
knownJavaThings.add("generics");
knownJavaThings.add("GUI");
knownJavaThings.add("serialization");
knownJavaThings.set(1, "classes"); // 0-based
knownJavaThings.remove(3);
System.out.println(knownJavaThings.get(3));
```

Vediamo ora `Map<K,V>`,

> **`interface Map<K,V>`**
>
> An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value.
>
> The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings. The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the TreeMap class, make specific guarantees as to their order; others, like the HashMap class, do not.

Una `Map<K,V>` astrae il concetto di **mappa**. Una chiave può avere al più un valore, o nessuno di loro. Non possono esistere duplicati di chiavi – è dunque un *set di chiavi*, con una *collection di valori*. Perciò, fornisce una **vista** di chiavi con i loro rispettivi **valori** – non estende `Iterable`, non può essere dunque iterata con un for-each. Le `Map<K,V>` modellano una funzione che va da `K` a `V` con in più il valore nullo (somiglia alla `Function<K,V>` – solo che una `Map<K,V>` modella un insieme di chiavi, è presente il concetto di "essere dentro", mentre le `Function` modellano un processo).

Dunque, alcuni metodi chiave,

> `V get(Object key)` Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
>
> `V put(K key, V value)` Associates the specified value with the specified key in this map (optional operation).
>
> `V remove(Object key)` Removes the mapping for a key from this map if it is present (optional operation).
>
> `int size()` Returns the number of key-value mappings in this map.
>
> `void clear()` Removes all of the mappings from this map (optional operation).
>
> `void putAll(Map<? extends K,? extends V> m)` Copies all of the mappings from the specified map to this map (optional operation).
>
> `boolean containsKey(Object key)` Returns true if this map contains a mapping for the specified key.
>
> `boolean containsValue(Object value)` Returns true if this map maps one or more keys to the specified value.

dove *optional operation* significa che per implementazioni "sola lettura" non è necessario implementare il metodo (bisogna fare `throw new UnsupportedOperationException()`).

Esistono delle **viste** su una mappa:

> `Set<K> keySet()` Returns a Set view of the keys contained in this map.
>
> `Set<Map.Entry<K, V>» entrySet()` Returns a Set view of the mappings contained in this map.
>
> `Collection<V> values()` Returns a Collection view of the values contained in this map.

e può restituire collezioni o insiemi, come visti prima. Inoltre,

> `interface Map.Entry<K,V>`
>
> `K getKey()` Returns the key corresponding to this entry.
>
> `V getValue()` Returns the value corresponding to this entry.

Per iterare su una mappa:

```
//Depending on the specific case:
Map<String, Integer> ages = /* ... */
ages.put("Eric", 42); //autoboxing
ages.put("Simba", 14);

//Keys matter:
for (String key : ages.keySet()) { /* ... */ }

//Values matter:
for (int value : ages.values()) { /* ... */ } //autounboxing

//Both matter:
for (Map.Entry<String, Integer> entry : ages.entrySet()) {
  String name = entry.getKey();
  int age = entry.getValue();
  /* ... */
}
```

Similmente a `SortedSet` esiste `SortedMap<K,V>`,

> interface SortedMap<K,V>

A Map that further provides a total ordering on its keys. The map is ordered according to the natural ordering of its keys, or by a Comparator typically provided at sorted map creation time. This order is reflected when iterating over the sorted map's collection views (returned by the entrySet, keySet and values methods). Several additional operations are provided to take advantage of the ordering. (This interface is the map analogue of SortedSet.)

`K firstKey()` Returns the first (lowest) key currently in this map.

`K lastKey()` Returns the last (highest) key currently in this map.

`SortedMap<K,V> headMap(K toKey)` Returns a view of the portion of this map whose keys are strictly less than toKey.

`SortedMap<K,V> subMap(K fromKey, K toKey)` Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.

`SortedMap<K,V> tailMap(K fromKey)` Returns a view of the portion of this map whose keys are greater than or equal to fromKey.

Attenzione però – `keySet()` restituisce ancora un `Set<K>`, non un `SortedSet<K>` – però, questo potrebbe essere cambiato nell'ultima versione. Al runtime, è un `SortedSet` e può essere fatto downcasting.

Altre collezioni interessanti possono essere:

`interface Queue<E> extends Collection<E>`

A collection designed for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, extraction, and inspection operations. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation).

`interface Deque<E> extends Queue<E>`

A linear collection that supports element insertion and removal at both ends. The name deque is short for "double ended queue" and is usually pronounced "deck". Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.

`interface BlockingQueue<E> extends Queue<E>`

A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

Esistono alcune implementazioni interessanti di tutto ciò, ad esempio

- Set → HashSet, LinkedHashSet, EnumSet – nessuna delle 3 è thread-safe; `HashSet` è quella "di base", internamente è implementata con gli hash; `LinkedHashSet` è ottimizzata per le modifiche, e fornisce la garanzia di **mantenere l'ordine di inserimento**, dunque permettendo la **riproducibilità**; `EnumSet` fa sì che gli elementi siano tipi *enum*, cioè ottimizzando le performance poiché i possibili valori sono noti a priori;
- SortedSet → TreeSet;
- List → ArrayList, LinkedList – esse non forniscono garanzie aggiuntive, né sono thread-safe; `ArrayList` implementa mediante un array, e benché sia tutto ottimizzato, iterazione e copia sono velocissimi, modifica invece non è rapida; `LinkedList` è diversa, la modifica è molto veloce, l'iterazione è più lenta;
- Map → HashMap, LinkedHashMap, EnumMap – similmente per i `Set`, abbiamo equivalenze sull'implementazione, dunque valgono le cose dette sopra;
- SortedMap → TreeMap;

Le implementazioni differiscono per 3 aspetti:

1. proprietà aggiuntive;
2. comportamento rispetto al multithreading;
3. prestazioni.

Alcuni metodi chiave,

```java
List<String> names = new ArrayList<>();
names.add("Eric");
names.add("Simba");
System.out.println("Names = " + names);

//Gives:
//Names = [Eric, Simba]
```

Vediamo ora `hashCode()`,

> `public int hashCode()` (in `Object`)
>
> Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by HashMap.
>
> The general contract of hashCode is:
>
> Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application. If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Il valore di `hashCode()` non dovrebbe cambiare a meno che non cambi lo **stato** (cioè, che `equals()` fornisca esiti diversi). Il metodo `hashCode()` deve fornire lo stesso hash – **non** è richiesto tuttavia che il *distillato* sia diverso dall'altro per due oggetti qualora `equals()` restituirebbe `false` per essi. Una buona implementazione di `hashCode()` è generalmente fornita dall'IDE (parimenti per `equals()`.

Tutte le implementazioni hanno un costruttore di default, per collezioni o mappe vuote. Dunque,

```java
List<String> names = new ArrayList<>();
SortedMap<Person, Double> marks = new TreeMap<>();
```

Esistono anche costruttori a partire da altre collezioni o mappe, cioè

```java
List<String> names = new ArrayList<>();
/* ... */
Set<String> uniqueNames = new HashSet<>(names);
```

e **non è una vista** e nemmeno una **copia** dell'altra collezione, cioè **ogni elemento è lo stesso**, non cambiano solo i riferimenti!

In linea di massima, si adopera il tipo più generale:

```java
//Ok!
List<String> names = new ArrayList<>();

//Not ok! Rarely necessary to need an ArrayList
//Moreover, names is a List of names, not an ArrayList!
ArrayList<String> names = new ArrayList<>();
```

per i motivi già citati. Esistono dei metodi *constructor-like* che sono `static` e producono **collezioni immutabili**:

> Set: `static <E> Set<E> of()` Returns an unmodifiable set containing zero elements.
>
> `static <E> Set<E> of(E... elements)` Returns an unmodifiable set containing an arbitrary number of elements.

List: `static <E> List<E> of()` Returns an unmodifiable list containing zero elements.

`static <E> List<E> of(E... elements)` Returns an unmodifiable list containing an arbitrary number of elements.

Dunque, l'utilizzo

```
List<String> names = List.of("Eric", "Simba");
names.add("Pippi"); // -> java.lang.UnsupportedOperationException
```

La maggior parte delle implementazioni non sono thread-safe. In particolare, al runtime quando ci sono più thread the eseguono operazioni sullo stesso oggetto lanciano una `ConcurrentModificationException` (non si possono iterare e fare modifiche sullo stesso oggetto, nemmeno con uno stesso thread).

Alcune sono `thread-safe`,

- `class ConcurrentHashMap<K,V> implements Map<K,V>` (che implementa in realtà `implements ConcurrentMap<K,V>`);
- `class ConcurrentSkipListSet<E> implements SortedSet<E>`.

Esistono poi anche **viste thread-safe**, come `Collections` che forniscono metodi statici di utilità per ottenere viste thread-safe di collezioni esistenti, e lo fanno come wrapper in blocchi `syncrhonized`. Dunque,

- `static * synchronized*()` è il modello generico;
- `static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);`
- `static <T> List<T> synchronizedList(List<T> list);`

Attenzione però: la thread-safety è garantita **solo sulla view**, non su ciascun elemento!

TODO 638 documentazione `Collections`

TODO 639 schema complessivo

21.1 Esempio d'esercizio

Data una frase, restituire il conteggio delle parole. Dunque:

```
public static Map<String, Integer> wordOccurrences(String sentence)
    {
  String[] words = sentence.split("\\W+");
  Map<String, Integer> occurrences = new HashMap<>();
  for (String word : words) {
    occurrences.put(
        word,
        occurrences.getOrDefault(word, 0) + 1
    );
  }
  return occurrences;
}

// default V getOrDefault(Object, V) returns
// the second argument if the key (1st argument)
// is not present

System.out.println(wordOccurrences("today the cat is on the
    table."));
// {the=2, today=1, cat=1, is=1, table=1, on=1}

// same, but ordered alphabetically
SortedMap<String, Integer> sortedOccurrences = new TreeMap<>(
    wordOccurrences("today the cat is on the table.")
);
```

```
System.out.println(sortedOccurrences);
// {cat=1, is=1, on=1, table=1, the=2, today=1}

// same, but ordered by increasing occurrences
List<Map.Entry<String, Integer>> entries = new ArrayList<>(
  wordOccurrences("today the cat is on the table.").entrySet()
);
entries.sort(Comparator.comparing(Map.Entry::getValue));

System.out.println(entries);
//[today=1, cat=1, is=1, table=1, on=1, the=2]

// sort() line is equivalent to
// entries.sort((e1, e2) -> e1.getValue()
// .compareTo(e2.getValue()))
```

Ci sono alcuni tipi di dati (modellazioni relative a come raggruppare certi elementi) non modellate nel collection framework – ad esempio, i multiset o le bag. I primi sono modellati da `Multiset<E>` di *Google Guava*, le seconde da `Bag<E>` di *Apache Commons*. Guava e Commons sono insiemi di librerie curate con particolare attenzione da Google e dalla Apache Foundation.

Capitolo 22

Executors – gli Esecutori

Gli **executors** modellano la capacità di una componente di eseguire qualcosa – ad esempio task, job, e così via. Molto spesso si intende semplificare la gestione al runtime e lo sviluppo di task paralleli. Il parallelismo può essere eseguito grazie a `tread()` e similari – si ha spesso però la necessità di gestire il tutto *più ad alto livello*, senza entrare nei dettagli come con `start()`. Si individuano due concetti chiave:

- il **task** – unità di processamento dotata di input ed output;
- l'**esecutore** – una componente in grado di eseguire i task.

Per modellare questi concetti esistono varie classi ed interfacce – la prima è la functional interface `Callable<V>`, qualcosa che si può chiamare in senso informatico,

> A task that returns a result and may throw an exception. Implementors define a single method with no arguments called call.
>
> `V call()` Computes a result, or throws an exception if unable to do so.

Il metodo `call()` viene implementato, con `V` tipo di ritorno (l'output del task), con l'input implicito nella classe che istanzia `Callable` – inoltre, `call() throws Exception`, così da poter gestire le eccezioni.

Ad esempio, calcoliamo $\sum_{i=0}^{10^5} i!$,

```
final long x = 100000;
// cannot parametrize with primitive types
Callable<Long> sumOfFactorialsTask = () -> {
  long sum = 0;
  for (int i = 0; i < x; i++) {
    sum = sum + factorial(i);
  }
  return sum;
};
try {
  long result = sumOfFactorialsTask.call();
  System.out.println("sum=" + result);
} catch (Exception e) {
  System.err.println(String.format("Cannot compute due to %s", e));
}

public static long factorial(long n) {
  return (n <= 1) ? n : (n * factorial(n - 1));
}
```

L'esecuzione di tale frammento di codice è *sincrona*.

Un'altra interfaccia è `Runnable`, una `Callable` semplificata che non ammette risultato:

> The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.
>
> `void run()` When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.

e non ha `Exception` e fa uso di `run()`. Trattasi di una functional interface che esisteva da prima di `Callable`. Inoltre, `Thread implements Runnable`. Il suo utilizzo,

```
final long x = 100000;
new Thread(() -> {
  long sum = 0;
  for (int i = 0; i < x; i++) {
    sum = sum + factorial(i);
  }
}).start();
```

con `Thread(Runnable target)` uno dei costruttori esistenti di `Thread`. L'esecuzione è **asincrona**. Il risultato va prelevato in qualche modo: mediante coordinamento, raccolta del risultato, sincronizzazione, active polling, variabili condivise.

La differenza fra i due è scritta nella documentazione di `Callable`:

> The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.
>
> The `Executors` class contains utility methods to convert from other common forms to Callable classes.

dunque, tramite la classe `Executors` consente la conversione fra le due classi (interfacce). Ora, abbiamo l'interfaccia `ExecutorService`:

> An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.
>
> An ExecutorService can be shut down, which will cause it to reject new tasks. Two different methods are provided for shutting down an ExecutorService. The shutdown() method will allow previously submitted tasks to execute before terminating, while the shutdownNow() method prevents waiting tasks from starting and attempts to stop currently executing tasks. Upon termination, an executor has no tasks actively executing, no tasks awaiting execution, and no new tasks can be submitted. An unused ExecutorService should be shut down to allow reclamation of its resources.
>
> Method submit extends base method Executor.execute(Runnable) by creating and returning a Future that can be used to cancel execution and/or wait for completion.
>
> `Future<?> submit(Runnable task)` Submits a Runnable task for execution and returns a Future representing that task.
>
> `<T> Future<T> submit(Callable<T> task)` Submits a value-returning task for execution and returns a Future representing the pending results of the task.

Dunque, `ExecutorService` modella l'esecutore che esegue i task, permettendo il ritorno del loro risultato tramite `Future`. Inoltre, il risultato è del tipo `T`. Un `ExecutorService` può anche essere spento. Internamente, un'implementazione di `ExecutorService` ha dei thread the crea e che usa poi per eseguire i task, il tutto mediante interfaccia di alto livello.

Un `Future`:

> A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready.

`V get()` Waits if necessary for the computation to complete, and then retrieves its result.

Dunque, `get()` consente di ottenere il `V` di risultato – l'invocazione di `get()` è **bloccante** qualora il risultato non sia pronto. Ad esempio,

```
ExecutorService executorService = /* ... */
long n = 20;
Future<Long> future = executorService.submit(() -> factorial(n));
System.out.println("Computing");
try {
  long result = future.get();
  System.out.printf("%d!=%d%n", n, result);
} catch (InterruptedException | ExecutionException e) {
  System.err.println(String.format("Cannot compute due to %s", e));
}
executorService.shutdown();
```

con `factorial(n)` eseguito in parallelo, in maniera asincrona. Un caso più interessante,

```
ExecutorService executorService = /* ... */
List<Callable<Long>> callables = new ArrayList<>();
for (int i = 0; i < 20; i++) {
  final long n = i;
  callables.add(() -> factorial(n));
}
try {
  List<Future<Long>> futures = executorService.invokeAll(callables);
  for (Future<Long> future : futures) {
    // asking for results in order
    System.out.printf("Got %d%n", future.get());
  }
} catch (InterruptedException | ExecutionException e) {
  System.err.println(String.format("Cannot compute due to %s", e));
}
```

Le principali abilità di un `ExecutorService`, nelle varie implementazioni di `ExecutorService`, possono essere viste creandole con metodi `static` della classe `Executors`:

`static ExecutorService newCachedThreadPool()` Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

`static ExecutorService newFixedThreadPool(int nThreads)` Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

`static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)` Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

Nella fattispecie:

- `newFixedThreadPool` prende solo fino a n thread, quindi al più n task sono eseguiti assieme **da questo esecutore**;
- `ScheduledExecutorService` invece aggiunge metodi per l'esecuzione schedulata, ad esempio `schedule(Callable<V> callable, long delay, TimeUnit unit)` e `scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)`.

Un esempio di protocollo, dove si intendono servire `n` client al più alla volta (slide 657),

```
public class ExecutorLineProcessingServer {
  private final int port;
  private final String quitCommand;
  private final Function<String, String> commandProcessingFunction;
  private final ExecutorService executorService;
```

```
  public ExecutorLineProcessingServer(int port, String quitCommand,
      Function<String, String> commandProcessingFunction, int
      concurrentClients) {
    this.port = port;
    this.quitCommand = quitCommand;
    this.commandProcessingFunction = commandProcessingFunction;
    executorService =
        Executors.newFixedThreadPool(concurrentClients);
  }
  public void start() throws IOException {
    /* ... */
  }
}
```

e, tutto in un singolo metodo,

```
public void start() throws IOException {
  try (ServerSocket serverSocket = new ServerSocket(port)) {
    while (true) {
      try {
        final Socket socket = serverSocket.accept();
        executorService.submit(() -> {
          try (socket) {
            BufferedReader br = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
            BufferedWriter bw = new BufferedWriter(new
                OutputStreamWriter(socket.getOutputStream()));
            while (true) {
              String command = br.readLine();
              if (command == null) {
                System.err.println("Client abruptly closed
                    connection");
                break;
              }
              if (command.equals(quitCommand)) {
                break;
              }
              bw.write(commandProcessingFunction.apply(command) +
                  System.lineSeparator());
              bw.flush();
            }
          } catch (IOException e) {
            System.err.printf("IO error: %s", e);
          }
        });
      } catch (IOException e) {
        System.err.printf("Cannot accept connection due to %s", e);
      }
    }
  } finally {
    executorService.shutdown();
  }
}
```

Esiste un metodo della classe `Runtime`, `availableProcessors()` che restituisce il numero di processori disponibli nella JVM (si guardino anche gli altri metodi interessanti di `Runtime`).

Capitolo 23

Gli Stream

Non sono gli IOStream, ma sono **Stream**: essi supportano operazioni in stile funzionale su flussi di elementi, ottenendo codice più conciso e più chiaro. Lo stream è *"un continuo flusso o successione di qualcosa di omogeneo"*. Non si tratta, tuttavia, del gruppo di elementi (bensì del suo flusso) e non di una collection (quello è il contenitore degli oggetti). Uno stream è **potenzialmente infinito**, mentre le collezioni non lo sono. Gli stream sono usati per fare processamenti su flussi, sempre la medesima sequenza di operazioni – ad esempio, trasforma in uppercase, calcola la frequenza delle lettere, **map-reducing**.

Dunque, l'interfaccia `Stream<T>`,

> A sequence of elements supporting sequential and parallel aggregate operations.

Dal pacchetto `java.util.stream`,

> The key abstraction introduced in this package is stream. The classes Stream, IntStream, LongStream, and DoubleStream are streams over objects and the primitive int, long and double types. Streams differ from collections in several ways:
>
> - No storage. A stream is not a data structure that stores elements; instead, it conveys elements from a source such as a data structure, an array, a generator function, or an I/O channel, through a pipeline of computational operations.
> - Functional in nature. An operation on a stream produces a result, but does not modify its source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
> - Laziness-seeking. Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization. For example, "find the first String with three consecutive vowels" need not examine all the input strings. Stream operations are divided into intermediate (Stream-producing) operations and terminal (value- or side-effect-producing) operations. Intermediate operations are always lazy.
> - Possibly unbounded. While collections have a finite size, streams need not. Short-circuiting operations such as limit(n) or findFirst() can allow computations on infinite streams to complete in finite time.
> - Consumable. The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

Gli stream sono utilizzati mediante 3 fasi:

1. lo stream si prende da una **sorgente** (una collezione, un array);
2. si definisce una **pipeline di operazioni intermedie** sullo stream;

3. si applica una qualche **operazione terminale**, cioè operazioni tipo "conta quanti ce ne sono", "estrai un elemento", "si riversa il contenuto dello stream in un'altra collection".

Ad esempio, si può ottenere uno stream,

- da `Collection<T>`, invocando `Stream<T> stream()` o `Stream<T> parallelStream()`, con l'ultimo che ove possibile effettua le operazioni in parallelo;
- da un array, con `Stream<T> Arrays.stream(T[])`;
- da altri tipi:
  - `Stream <String> lines()` in `BufferedReader`;
  - `Stream<String> splitAsStream(CharSequence input)` in `Pattern`.

Le operazioni che si possono effettuare:

> `Stream<T> distinct()` Returns a stream consisting of the distinct elements (according to Object.equals(Object)) of this stream.
>
> `Stream<T> filter(Predicate<? super T> predicate)` Returns a stream consisting of the elements of this stream that match the given predicate.
>
> `<R> Stream<R> map(Function<? super T,? extends R> mapper)` Returns a stream consisting of the results of applying the given function to the elements of this stream.
>
> `DoubleStream mapToDouble(ToDoubleFunction<? super T> mapper)` Returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
>
> `IntStream mapToInt(ToIntFunction<? super T> mapper)` Returns an IntStream consisting of the results of applying the given function to the elements of this stream.
>
> `LongStream mapToLong(ToLongFunction<? super T> mapper)` Returns a LongStream consisting of the results of applying the given function to the elements of this stream.
>
> `Stream<T> sorted(Comparator<? super T> comparator)` Returns a stream consisting of the elements of this stream, sorted according to the provided Comparator.

`interface Predicate<T>` è una `@FunctionalInterface` con un metodo `boolean test(T)`, e `interface ToDoubleFunction<T>` è una functional interface con un metodo `double applyAsDouble(T)`, lo stesso c'è per `int` e `long`.

Per quanto riguarda le operazioni terminali,

> `<R,A> R collect(Collector<? super T,A,R> collector)` Performs a mutable reduction operation on the elements of this stream using a Collector.
>
> `T reduce(T identity, BinaryOperator<T> accumulator)` Performs a reduction on the elements of this stream, using the provided identity value and an associative accumulation function, and returns the reduced value.
>
> `Object[] toArray()` Returns an array containing the elements of this stream.
>
> `<A> A[] toArray(IntFunction<A[]> generator)` Returns an array containing the elements of this stream, using the provided generator function to allocate the returned array, as well as any additional arrays that might be required for a partitioned execution or for resizing.
>
> `long count()` Returns the count of elements in this stream.

e altre operazioni, tipo `average()`, `min()`, `max()` negli stream di numeri.

"interface BinaryOperator<T> extends BiFunction<T,T,T> is a @FunctionalInterface with no further methods interface BiFunction<T,U,R> is a @FunctionalInterface with a method R apply(T, U)"

Alcuni esempi:

"Given a collection of person names (first+last), get the initials corresponding to names with more than 2 words in the names."

```
Collection<String> names = List.of(
```

```
    "Andrea De Lorenzo",
    "Eric Medvet",
    "Alberto Bartoli",
    "Felice Andrea Pellegrino"
);
names = names.stream()
    .map(s -> s.split(" "))
    .filter(ts -> ts.length>2)
    .map(ts -> Arrays.stream(ts)
        .map(s -> s.substring(0, 1))
        .collect(Collectors.joining()))
    .collect(Collectors.toList());
System.out.println(names);
```

fornisce `[ADL, FAP]`. Altro esempio,

“Given a collection of strings, compute the average number of consonants.”

```
System.out.println(strings.stream()
    .map(s -> s.toLowerCase().replaceAll("[aeiou]", ""))
    .mapToInt(String::length)
    .average() // it is a so-called optional
    .orElse(0d) // returned if average does not exist
);
```