Università degli Studi di Verona

CORSO DI LAUREA IN INFORMATICA

Programmazione II

Federico Brutti
federico.brutti@studenti.univr.it

If I may introduce a bug, the JVM will manage it for me ensuring both security and portability. Then I'll write my code once, and run it everywhere! with Static and Strong Typing, will let my programs be type safe! GARBAGE COLLECTOR! - Nanowar of Steel

Contents

Chapter 1

Introduzione

1.1 Programmazione Orientata agli Oggetti

Per **Programmazione orientata agli oggetti**, o OOP, si intende un particolare paradigma di programmazione strutturato intorno agli oggetti; delle strutture di dati contenenti attributi e metodi. Il software scritto in un linguaggio orientato agli oggetti baserà il suo funzionamento secondo le interazioni fra questi elementi.

Il vantaggio principale di questo modus operandi è nel rendere il codice estremamente modulare, riutilizzabile e mantenibile, in quanto diviso in sezioni precise. I due elementi base della OOP sono le **classi** e gli **oggetti**. Le prime definiscono una struttura dati, alla quale è possibile associare dati, detti **attributi**, e funzioni, chiamate **metodi**. Dalle classi, che in parole povere fungono da tipo di dato, è possibile ottenere gli oggetti, delle loro istanze, con stessi dati e metodi. A partire da questi elementi, definiamo i principi della programmazione orientata agli oggetti:

- **Incapsulamento**: Restrizione dell'accesso ai dati di un oggetto.
- **Astrazione**: Nascondere dettagli di implementazione e mostrare solamente le caratteristiche essenziali dell'oggetto.
- **Ereditarietà**: Tramandare campi e metodi ad altri elementi a partire da una superclasse.
- **Polimorfismo**: Possibilità di modificare campi e metodi delle singole istanze.

Un buon iter di lavoro generale per lavorare con un linguaggio orientato a oggetti è dato da **identificare** classi e oggetti necessari, per poi **dichiararle** insieme a relativi metodi e attributi utili. Dopodiché bisognerà **definire** le modalità di interazione fra gli oggetti ed infine **minimizzarne** quante più possibile.

Essendo che si andrà a lavorare su progetti di medie o grandi dimensioni, risulta essere buona prassi anche una corretta documentazione del codice tramite **Unified Model**

Languages, i quali consentono di creare una descrizione grafica di classi e oggetti. Verranno approfonditi più avanti nel corso.

1.2 Funzionamento di Java

Il linguaggio utilizzato nel corso sarà **Java**, nato nel '95 e proprietà di Oracle, è diventato lo standard nel mercato del lavoro per il software development. La sua caratteristica principale è la portabilità, ovvero è possibile eseguire programmi su qualunque architettura grazie alla **Java Virtual Machine**, la quale funge da interprete al codice compilato.

Più precisamente, dopo la compilazione, verrà creato in output un file con estensione ".class" contenente il **bytecode**. Questo codice è ciò che viene effettivamente interpretato in runtime da una parte della JVM, il compiler **Just In Time**, utile anche per ulteriori ottimizzazioni. In soldoni, a patto che la macchina abbia installata la JVM, sarà possibile eseguire i files compilati. Altre caratteristiche di Java sono:

- Linguaggio fortemente tipizzato, ovvero è possibile dichiarare variabili di un determinato tipo di dato, le quali non lo possono cambiare una volta aggiunte al codice.
- Non ha manipolazioni esplicite di puntatori grazie alla filosofia dell'incapsulamento, rendendo meno probabili errori riguardanti la memoria.
- Controlla il runtime, rendendo impossibile avere array overflow.
- Il **Garbage collector** domina la memoria dinamica, alloca e dealloca dove necessario. Inoltre gestisce memory leak.
- È possibile usare eccezioni per controllare gli errori.
- Linguaggio fortemente dinamico, poiché fa loading e linking in runtime. Inoltre, usa dimensioni di array dinamiche.

Dove è possibile installare sulla macchina solo la JVM, per scrivere programmi in Java è necessario usare il **Java Development Kit**, compreso di debugger, compiler, disassembler, ed un applicativo per la documentazione.

Per l'esecuzione dei programmi abbiamo poi il **Java Runtime Environment**, avente con sé librerie di classe, il compiler JIT precedentemente menzionato, la JVM e il Java application launcher.

1.3 Struttura, compilazione ed esecuzione dei programmi

Anzitutto, i programmi scritti in Java hanno estensione ".java" e vedono i blocchi di codice completamente all'interno di una classe, il cui nome deve essere uguale a quello del file. Al suo interno è poi necessario dichiarare l'entry point tramite il metodo **main**.

```
1  // Per compilare: javac file1.java file2.java ... filen.java
2  // Per eseguire: java file1 file2 ... filen
3
4  // Dichiarazione di classe, il nome coincide con quello del file.
5  public class HelloWorld {
6    // Entry point del programma, metodo di nome main
7    public static void main (string[] args) {
8      // Blocco di codice
9    }
10 }
```

Post-compilazione, avremo in output un file di estensione ".class". Infatti, ogni file è visto come classe il cui caricamento in compilazione è basato sul **classpath**, la lista di locazioni dove le classi possono essere prese. Se il compiler non trova una classe, lancerà un'eccezione e non verrà creato l'eseguibile.

Questo procedimento è valido se si lavora da terminale. È consigliato l'utilizzo di un IDE per la semplificazione del lavoro; nel corso verrà usato IntelliJ, il quale consente anche di automatizzare il deployment tramite **build tools** come Maven, che vedremo nella prossima sezione.

1.4 Gestione di un progetto su più file

Perché limitarsi ad un singolo file quando è possibile dividere in **unità** ogni funzione? Abbiamo visto nello snippet precedente che possiamo compilare ed eseguire più classi allo stesso tempo, ma la scrittura è estremamente tediosa, inefficiente e soggetta ad errori. La soluzione si ha in due passaggi:

- Una corretta divisione in cartelle del progetto.
- Raggruppamento delle classi in un unico pacchetto.

Il primo passo riguarda uno studio per ingegneria del software, quindi concentriamoci sulla seconda parte. Non è buona prassi, ma generalmente, quando in un file .java non è indicata l'appartenenza ad un pacchetto, il compilatore lo assocerà a quello di default, il quale non ha un nome e non consente ad altre classi di accedervi. Quindi è consigliato specificare l'appartenenza ad un dato pacchetto prima della definizione di classe; ciò consentirà di avere un classpath più compatto e più facile da gestire.

Se volessimo poi gestire facilmente più pacchetti, avremo bisogno di un contenitore più astratto: un file ".jar". Si tratta di un archivio compresso di bytecode e altre metainformazioni; idealmente, si ha un jar per progetto.

```
// Compila: javac -d bin/ src/pkg/MyClass.java
// Impacchetta: jar cvf MyJar.jar -C bin/ pkg/
// Esegui: java -cp MyJar.jar pkg.MyClass

// Segnala che la classe MyClass sta all'interno del pacchetto pkg.
package pkg;

public class MyClass {
  public static void main(String[] args) {
    // Blocco di codice
  }
}
```

Bisogna tuttavia specificare quale sia la classe main; a questo scopo vengono in aiuto le metainformazioni menzionate prima. Tramite un file speciale chiamato "manifest.txt" è possibile fare non solo questo, ma anche specificare quali classi compilare.

```
// Impacchetta: jar cvfm MyJar.jar manifest.txt -C bin/ pkg/
// Esegui: java -jar MyJar.jar

// manifest.txt
Main-Class: pkg.MyClass
```

Chapter 2

Il linguaggio Java

2.1 Tipi primitivi, variabili e costanti, operatori e assegnamenti

Supponendo che tu abbia frequentato o quantomeno ascoltato il corso di programmazione 1, la lettura risulterà nettamente più scorrevole. Infatti Java, per quanto riguarda le informazioni elementari, condivide una sintassi quasi uguale a quella di C, composta da:

- **Parole chiave del linguaggio**: Hanno un significato speciale e non possono essere usate per la dichiarazione di variabili o funzioni.
- **Identificatori**: I nomi scelti per gli elementi di programmazione definiti nel linguaggio.
- **Operatori**: Simboli per effettuare operazioni.
- **Dati**: Valori delle variabili, le informazioni nel codice.

Si mantengono tutte le funzioni legate al linguaggio C per quanto riguarda tipi primitivi, operatori ed assegnamenti. Abbiamo quindi: **int**, **double**, **char**, aggiungendo **boolean** e **String**, quest'ultimo non primitivo, successivamente approfondito.

Di base ogni dato è considerato una variabile se non specificato altrimenti. Infatti, per la dichiarazione di costanti sarà necessario aggiungere la keyword **final**. Ciò vale sia per variabili che per oggetti.

2.2 Dichiarazione di classi e oggetti, this, variabili locali e di istanza

Le classi e gli oggetti sono il fulcro sul quale si basa la programmazione in Java ed il paradigma a oggetti in generale. Come già menzionato, le prime si possono vedere come

un insieme, il quale detiene attributi e metodi, ed i secondi sono le singole istanze delle classi con le suddette specifiche. Data questa struttura, ne consegue che per lavorare dovremo prima definire una classe, per poi istanziare gli oggetti.

```
// Dichiarazione della classe Date nel file Date.java
public class Date {
  // Scope dei campi della classe, questi sono attributi...
  int day, month, year;
  // ...e questo un metodo.
  String printDate() {
    return day + "/" + month + "/" + year;
  }
}
```

```
// Dichiarazione della classe MainDate nel file MainDate.java
public class MainDate {
  public static void main(String[] args) {
    // Istanziazione dell'oggetto Today
    Date Today = new Date();
  }
}
```

Come si può vedere, la classe rappresenta il corpo contenente tutto il codice, mentre gli oggetti vengono istanziati in un altro file, ove richiesto, con la keyword **new**. Questa istruzione fa sì che venga allocato spazio sufficiente per contenere i dati dell'oggetto.

Essendo che i metodi dichiarati sono in grado di ricevere appositi parametri per un'eventuale elaborazione, è necessario chiarire il modo in cui si possono comportare le variabili. Infatti, troviamo due concetti molto importanti:

- **variabile d'istanza**: Corrisponde agli attributi di una classe, i quali sono inizializzati all'istanziazione di un oggetto. Non sono condivisi fra gli oggetti di una stessa classe.
- **variabile locale**: Attributi definiti all'interno di metodi o blocchi di codice, legati allo scope dove nascono. Non sono visibili da altri blocchi di codice e vengono deallocati a fine metodo.

Un'altro concetto molto importante è la parola chiave per far riferimento all'oggetto corrente: la keyword **this**, la quale può essere usata per attributi, variabili, o anche da sola per prendere l'oggetto intero, ed è particolarmente utile per evitare ambiguità fra gli oggetti.

```
public class Cliente {
  private String nome;

  // Il metodo prende il nome dato come parametro
```

```
5     public void getName (String nome) {
6       // Associa il parametro al preciso oggetto corrente
7       this.nome = nome;
8     }
9   }
```

L'utilizzo di this è molto importante per rendere il codice chiaro ed è uno standard da rispettare. In questo snippet, senza this, il programma non avrebbe capito quale stringa nome prendere, causando errori di compilazione.

2.3 Metodo costruttore e overloading

I **metodi costruttore** sono funzioni di una classe particolari, il cui blocco di codice determina ciò che viene eseguito all'istanziazione di un oggetto. In assenza di dichiarazione di tale metodo, il compilatore aggiungerà quello di default, che non esegue niente.

Ogni classe ha quindi un costruttore, il quale avrà il suo stesso nome e potrà prendere parametri, ma sarà sprovvisto della keyword return.

```
1   public class Punto {
2     int x, y;
3
4     // Metodo costruttore. Stampa la stringa ad istanziazione.
5     public Punto () {
6       System.out.println("Punto istanziato");
7     }
8   }
```

È possibile inoltre definire più costruttori in una classe, ma dovranno essere differenziati in base alla segnatura, quindi ai parametri. Per capire quale prendere, introduciamo il concetto di **constructor overloading**.

Supponiamo di definire un costruttore "Punto(int x)"; nell'istanziazione dell'oggetto, se sarà dato un intero come parametro, il compilatore sceglierà quest'ultimo, in alternativa, se non è dato niente, sceglierà il primo. Qualunque altro scenario comporta errori di compilazione.

2.4 Modificatori, visibilità e memoria di un programma

Avrai sicuramente notato le keyword usate nella dichiarazione di classi o utilizzate per le variabili; queste si chiamano **modificatori** e sono capaci di cambiare il significato delle componenti del programma. Approfondiamone il concetto ed il significato.

Modificatore public

Applicabile a classi, oggetti e variabili. Rende la componente accessibile in qualunque altro file. Tendenzialmente, volendo rendere il codice sicuro, non viene utilizzato per variabili.

```
public class PublicClass {
  public PublicClass () { /* Blocco di codice */ }
}
```

Modificatore protected
Applicabile a oggetti, metodi e attributi. Restringe l'accesso alla componente, consentendolo solo alla classe in cui è dichiarata e le sue eventuali sottoclassi, a prescindere dal pacchetto. Attua il principio di ereditarietà, il quale sarà spiegato in dettaglio più avanti.

```
public class PublicClass {
  protected int value = 1;
}

public class TrustedPerson extends PublicClass {
  // La subclass accede a value senza problemi
  int safeVar = value;
}
```

Modificatore private
Applicabile a oggetti e attributi. Rende la componente visibile esclusivamente dalla classe in cui è dichiarata. Usato molto spesso per sfruttare l'incapsulamento, se vi si prova ad accedere da classi differenti, il compiler darà errore.

```
public class PublicClass {
  private int invisible = 1;
}

public class NoobHacker {
  // Azione illegale. Il compiler si lagna.
  int stealVar = invisible;
}
```

Modificatore package private
Assegnato in automatico dal compilatore in assenza di modificatori, restringe l'accesso, consentendolo esclusivamente a classi di uno stesso pacchetto.

Prima di introdurre l'ultimo modificatore, è necessario parlare della memoria utilizzata dai programmi Java. Abbiamo una parte di memoria **static**, usata per elementi condivisi da tutte le istanze di classe, come definizioni di classe e campi statici; un'altra di **heap**, usata per l'allocazione di memoria dinamica e quindi istanziazione di oggetti, ed un'ultima di **stack**, dedita agli elementi creati in runtime. Tradotto in codice, abbiamo:

```
public class memoryTest {
  // Variabile nuova, spazio allocato in heap
```

```
3     int i;
4     // Oggetto String statico, condiviso da ogni classe
5     static String s1;
6     void aMethod () {
7       // s2 non static, elemento creato in runtime, va in stack memory.
8       String s2 = new String("abc");
9       s1 = s2;
10    }
11  }
```

Nota importante da tenere a mente: i metodi statici possono interagire esclusivamente con altri metodi static; altrimenti si avrà un errore di compilazione.

2.5 Costrutti condizionali e ciclici

Ogni linguaggio di programmazione contiene i costrutti per la manipolazione del flow, siano essi condizioni o cicli non importa. Saranno sempre presenti, seppur con forma differente. Supponendo che tu abbia già frequentato programmazione 1, salterò direttamente alle sintassi diverse da quelle del C, perché **if-else**, **while**, **do-while**, **switch** e **for** sono definiti esattamente come nel suddetto linguaggio.

Tuttavia, dalla versione Java 5, è stato introdotto l'**enhanced for loop**, che svolge esattamente allo stesso modo le funzioni del suo gemello, ma ha una sintassi semplificata.

```
1  for (tmpVar : iterObj) {
2    // Blocco di istruzioni
3  }
```

In parole povere, dice "scorri tutto iterObj con indice tmpVar", quindi è consigliabile usarlo quando si ha la certezza di voler scorrere un oggetto iterabile per intero.

Anche il costrutto switch ha ottenuto migliorie. Mettiamo caso di avere più casi che riportano uno stesso numero; in una scrittura tradizionale sarebbe necessario definire casi separati, ma in Java è possibile raggrupparli ed usare un operatore **arrow** per indicare ciò che devono eseguire.

```
1  switch (colore) {
2    case VERDE -> System.out.println("Luce verde");
3    case GIALLO -> System.out.println("Luce gialla");
4    case ROSSO -> System.out.println("Luce rossa");
5  }
```

Non solo, ma possiamo anche ritornare valori dal blocco con la keyword **yield**, per il quale abbiamo due scritture diverse. Una usa arrow, l'altra i due punti e dove entrambe svolgono una stessa funzione, non è possibile mischiarle. Bisogna necessariamente scegliere una delle due notazioni. Questo perché usando la prima, è come se ci fosse un break incorporato nel caso, mentre nella seconda è ancora presente il fall-through.

```
// Notazione con operatore arrow '->'
String season = switch (month) {
  case DECEMBER, JANUARY, FEBRUARY -> "Winter";
  case MARCH, APRIL, MAY -> "Spring";
  case JUNE, JULY, AUGUST -> "Summer";
  case SEPTEMBER, OCTOBER, NOVEMBER -> "Autumn";
}

// Notazione con operatore colon ':'
String season = switch (month) {
  case DECEMBER, JANUARY, FEBRUARY: "Winter";
  case MARCH, APRIL, MAY: "Spring";
  case JUNE, JULY, AUGUST: "Summer";
  case SEPTEMBER, OCTOBER, NOVEMBER: "Autumn";
}
```

2.6 Array, enumerazioni e record

In Java è possibile dichiarare **array** mono- e bidimensionali; sono visti come oggetti speciali allocati in heap, quindi definiti in runtime, e da un punto di vista pratico sono sequenze di puntatori ad oggetto. Alcuni aspetti importanti sono:

- In mancanza di inizializzazione, la JVM li setta a null.
- Post-dichiarazione, sarà impossibile modificarne la lunghezza.
- Essendo sequenze di puntatori, il metodo equals non funzionerà come negli oggetti normali.

```
// Dichiarazione di array
int[] arr = new int[dimensione];
// Scrittura in un indice specifico
arr[numeroIndice] = NomeClasse(eventuali_parametri);
// Accesso ad un elemento in un indice specifico
int var = arr[numeroIndice];
```

Per una manipolazione corretta e agevolata di array è utile la libreria **java.util.Arrays**, contenente metodi per effettuare confronti e stampare il contenuto. Verrà approfondita più avanti. Naturalmente, per scorrerli e lavorarci sarà necessario utilizzare costrutti ciclici.

Le matrici hanno ricevuto delle migliorie; la loro dichiarazione è simile a quella degli array ed è possibile usare su di esse tutti i metodi utili per questi ultimi. Inoltre, in quanto sequenze di puntatori, è possibile scambiare facilmente righe e colonne.

```java
// Dichiarazione ed inizializzazione di una matrice di interi
int[][] matrix = {{1,2,3}, {4,5,6}, {7,8,9}};
// Per scambiare righe o colonne puoi prendere il puntatore.
int[] tmp = matrix[0];
matrix[0] = matrix[matrix.length-1];
matrix[matrix.length-1] = tmp;
// Stampa gli elementi di una matrice
for (int[] row : matrix) System.out.println(Arrays.toString(row));
```

Un altro elemento importante per la programmazione in Java sono i **tipi enumerazioni**. Sono classi che rappresentano un insieme finito di costanti, le quali sono tutte le istanze che può assumere l'eventuale oggetto.

Le enumerazioni si definiscono con la keyword **enum**, ed il loro vantaggio sta nell'efficienza del processo di recupero valori; è infatti consigliato utilizzarle in necessità di richiamare più volte determinati dati.

```java
// Dichiarazione dell'enumerazione
public enum Numbers { ONE, TWO, THREE };
// Istanziazione dell'oggetto
Numbers numOne = Numbers.ONE;
```

Gli elementi delle enumerazioni, in quanto già rappresentanti un'istanza della classe, sono implicitamente del tipo Months, per questo non è necessario utilizzare new.

2.7 Classi innestate e interne

Se ritieni che scrivere un file per ogni singola classe necessaria sia eccessivo e renda il programma eccessivamente complesso, hai la possibilità di dichiararne di più in un singolo file. Fare questo, dipendentemente dalla posizione in cui sono inserite, introduce due concetti:

• **Classe innestata**: Classe statica visibile ed utilizzabile a prescindere da quella in cui è dichiarata. Possono avere accesso ad ogni campo statico della classe primaria, indipendentemente dal modificatore d'accesso.

```java
// Dichiarazione di classe innestata
public class Outer {
  static class Nested { /* Blocco di codice */ }
}

// Istanziazione di un suo oggetto
Outer.Nested nestedObject = new Outer.Nested();
```

• **Classe interna**: Classe non statica con le stesse dinamiche dell'innestata, ma il suo utilizzo dipende dalla primaria.

```java
// Dichiarazione di classe interna
```

```
public class Outer {
  class Inner { /* Blocco di codice */ }
}

// Istanziazione richiede un oggetto della classe primaria.
Outer outerObject = new Outer();
Outer.Inner innerObject = Outer.new Inner();
```

Una dinamica di ottimizzazione molto utilizzata è invece la dichiarazione di classi non statiche all'interno dei metodi. Queste sono dette **locali**, possono accedere esclusivamente a variabili finalizzate e vengono deallocate al termine del metodo. Generalmente ha senso utilizzarle anche per ridurre la complessità generale del codice in runtime, garantire una maggiore leggibilità e organizzazare al meglio le funzionalità del codice, dando anche limitazioni sullo scope.

```
public void outerMethod () {
  int j = 1;

  class Local {
    int addOne () { return j + 1; }
  }

  Local localObject = new Local();
  System.out.println(localObject.addOne()); // Output: 2
}
```

Chapter 3

Paradigmi OOP

3.1 Incapsulamento

L'incapsulamento è un concetto che ha origine nelle modalità di interazione nel mondo reale. Così come per interagire con una persona si usa la voce per ottenere informazioni specifiche, allo stesso modo gli oggetti comunicano con interfacce apposite al fine di ritornare dati. Questa filosofia si ottiene controllando l'accesso ai dati tramite le keyword dei modificatori di accesso. Più precisamente, bisogna:

1. Dichiarare gli attributi di una classe come **privati**, per renderli inaccessibili ad altre classi al di fuori di questa.

2. Creare dei metodi **pubblici** per accedere, modificare e riportare gli attributi privati in maniera controllata, talvolta con dei controlli per assicurarsi che i dati siano presentati con certe caratteristiche. Questi metodi si chiamano **setter**, per assegnare il dato all'oggetto, e **getter**, per accedere al dato dell'oggetto.

```
1  public class Date {
2    private int day, month, year;
3
4    // Metodo setter per garantire che giorno sia un valore corretto
5    public void setDay (int g) {
6      if (j >= 1 && j<= 31) giorno = g;
7      else System.out.println("Invalid day");
8    }
9
10   // Metodo getter per ritornare il valore in sicurezza
11   public int getDay () { return day; }
12
13   public void setMonth (int m) {
14     if (m <= 1 && m <= 12) month = m;
15     else System.out.println("Invalid month");
```

```
16      }
17
18      public int getMonth () { return month; }
19
20      public void setYear (int y) {
21        if (y <= 0) year = y;
22        else System.out.println("Invalid year");
23      }
24
25      public int getYear () { return year; }
26    }
```

3.2 Ereditarietà

Quello di **ereditarietà** è un concetto strettamente legato all'insiemistica. Partiamo dal presupposto che esiste un insieme universo, il quale comprende ogni singola altra cosa nel mondo; in Java, le classi funzionano con la medesima dinamica. Tutte le classi sono un sottoinsieme improprio di **Object**, la quale presenta i seguenti metodi per noi importanti:

- **protected Object clone()**: Istanzia e ritorna una copia dell'oggetto richiesto.
- **boolean equals(Object obj)**: Controlla se un oggetto è uguale a quello chiamante.
- **int hashCode()**: Ritorna il valore dell'hashcode dell'oggetto chiamante, ovvero la parte sinistra alla chiocciola della reference.
- **String toString()**: Ritorna una rappresentazione dell'oggetto sotto forma di stringa.

Tutte le classi, e quindi anche i tipi, sono **estensioni** della classe Object, e di conseguenza potranno chiamare ed eseguire i metodi in essa contenuti. Questo concetto è naturalmente replicabile con ulteriori sottoclassi, quindi ci è concesso creare nuovi metodi e campi riutilizzabili in più file senza che vengano ridefiniti. La potenzialità di questo paradigma sta nella riduzione all'estremo di codice ripetuto; le sottoclassi sono quindi capaci di attingere da ciò che è definito nella superclasse, e, nel caso dei metodi, fare ciò che è chiamato **overriding**

Supponiamo ora di avere una classe "Vehicle". Il concetto di veicolo è specificabile in sottoinsiemi come automobili, biciclette e simili. In Java ciò si traduce come:

```
1   public class Vehicle {
2     protected int num;
3     protected int getNum() { /* Blocco di codice */}
```

```
4     public Vehicle { /* Blocco del costruttore */ }
5   }
6
7   // Estensione di Vehicle, chiama il metodo per ottenere il numero.
8   public class Car extends Vehicle {
9     public Car {
10      this.num = super.getNum();
11    }
12  }
13
14  // Ridefinizione di getNum per mantenere lo stesso nome e
     semplificare il codice
15  public class Bicycle extends Vehicle {
16    @Override
17    public int getNum() { /* Ridefinizione del codice */ }
18  }
```

L'accesso a campi e metodi della superclasse avviene tramite la keyword **super**, se questi sono stati dichiarati come pubblici o protetti. Questa parola è anche strettamente legata al metodo costruttore; questo non è ereditato dalle sottoclassi, tuttavia è necessario che queste abbiano anche le caratteristiche della superclasse e siano costruite in tali termini. In soldoni, i costruttori non sono ereditati ed ogni sottoclasse ha bisogno del prorpio, ma quest'ultimo chiamerà necessariamente il costruttore della sovracclasse con super().

La ridefinizione è invece per prima cosa segnalata da **@Override**, per poi riscrivere il blocco di codice. Quest'ultima feature è particolarmente utile per mantenere il numero di nomi basso e rendere il codice più chiaro a livello di struttura.

Un'ultima caratteristica del linguaggio utile è la possibilità di controllare se una classe è superclasse di un'altra, con la keyword **instanceof**. Per esempio, in un codice che rappresenta Person, Student, Lecturer e Sam avremo:

```
1   Student sam = new Student();
2
3   (sam instanceof Student)  // True, Identita'.
4   (sam instanceof Person)   // True, Person sovracclasse di Student.
5   (sam instanceof Lecturer) // False, Student non legato a Lecturer.
6   (sam instanceof Object)   // Tautologia.
```

Un compito comune da eseguire in campo di ereditarietà ed istanziazione di oggetti è controllare se questi siano uguali o meno; come già specificato, la classe Object ha con sé il metodo equals(), che consente di fare proprio questo. Tuttavia, Java salva gli oggetti con riferimenti diversi per differenziarli, indipendentemente dal loro contenuto. Supponiamo per esempio di istanziare due stringhe sintatticamente uguali; se non ridefinito correttamente, come da suo funzionamento, ritornerà false.

Dunque qui abbiamo due strade per correggere questo comportamento. Possiamo fare overriding del metodo equals() e specificare che cosa deve essere uguale per far sì che ritorni true, oppure più semplicemente ridefinire questo metodo insieme ad hashCode().

Per quest'ultima soluzione è necessario far sì che la stessa reference sia assegnata ad ogni oggetto i cui campi sono uguali ad un altro.

Un'altra dinamica a cui prestare attenzione è legata all'incapsulamento; generalmente, i campi dichiarati nelle superclassi, a meno che non si voglia siano condivisi a tutti i figli, sono dichiarati come privati; sarà quindi necessario utilizzare metodi getter per potervi accedere, in quanto sono gestiti come pubblici.

3.3 Polimorfismo

Per **polimorfismo** intendiamo un'entità che può assumere più forme; nella programmazione a oggetti si può implementare in diverse modalità. Per esempio, restando sempre nel campo dei riferimenti, diciamo che una reference di tipo T può puntare ad un oggetto di tipo S se e solo se S è T oppure una sua sottoclasse. In quest'ultimo caso, gli oggetti di tipo T possono essere rimpiazzati da oggetti di tipo S, usando il **principio di sostituzione di Liskov**; quindi:

```
1  Person p;
2  p = new Person();    // Corretto, T=S.
3  p = new Student();     // Corretto, T sottoinsieme di S
4
5  Person person = new Person("Paul");
6  Student student = new Student ("Sam", 123);
7  person = student;    // Sostituzione corretta, assegnato un sottotipo
8  student = person;    // Errore di compilazione.
```

Nell'ultimo caso, l'errore avviene perché il compilatore effettua una ricerca ricorsiva dei metodi, partendo dalla sottoclasse fino ad arrivare a Object. Se ritorna un problema, è perché non trova il metodo chiamato in nessuna delle classi, quindi bisogna fare attenzione a rispettare i parametri da dare alle funzioni.

Abbiamo già menzionato il concetto di type casting; la stessa dinamica è applicabile agli oggetti, in due direzinoni:

- **Upcast**: Garantito dal principio di sostituzione, type safe. Visto nello snippet di prima, dove ad un oggetto person è assegnato uno student. Si può scrivere implicitamente poiché è eseguito dal compiler in runtime.
- **Downcast**: Necessario esplicitarlo, non ha garanzia di essere type safe. Nel caso di prima, riguarda il caso di errore di compilazione; per far sì che sia corretto bisogna castare il tipo Student prima di person. Per evitare eventuali problemi è consigliato l'utilizzo di instanceof per controllare le superclassi.

Un altro modo per ottenere il polimorfismo, ed universalmente codice meno verboso e utilizzabile in più contesti senza che venga modificato, è dato dai tipi generici, **generics**.

Questa feature si attiva usando la sintassi "<T>" al posto dei parametri. I caratteri convenzionati sono T(ype), E(lement), K(ey), V(alue). Scriveremo quindi, per esempio:

```
// Dichiarazione della classe con parametro di tipo generico.
public class Pair <T> {
  private T first, second;

  // Essendo generico, puo' essere qualunque oggetto.
  public Pair(T first, T second) {
    this.first = first;
    this.second = second;
  }

  public T getFirst () { return first; }
  public T getSecond () { return second; }
}

// Classe utilizzabile con piu' tipi. Cast non necessario.
Pair <String> p = new Pair <> ("one", "two");
Pair <Integer> q = new Pair <> (1, 2);
```

I generics funzionano perché al posto del carattere il compilatore riconosce la classe Object, ed in quanto ogni altra classe è suo sottoinsieme, si effettua un downcast tautologico. L'uso del generic in questo modo lo rende un **parametro libero**, mentre se lo si usa per estendere altri parametri, sarà necessario dare un limite inferiore per evitare che il compiler riscontri problemi nella sua ricerca ricorsiva. T sarà chiamato **parametro legato**.

```
class Clazz <T extends S1 & S2 & ... & Sk>  // Limite superiore
class Clazz <T super S> // Limite inferiore
```

```
public class Pair <T extends Person> {
  private T first, second;

  public Pair (T first, T second) {
    this.first = first;
    this.second = second;
  }

  public T getFirst () { return first.getName(); } // Nessun errore
  public T getSecond () { return second; } // T estende Person, ok.
}
```

3.4 Astrazione

Per **astrazione** intendiamo la possibilità di descrivere la forma del programma senza implementarne le funzionalità. Risulta utile sia per avere una chiara definizione della

struttura delle classi, sia per, come vedremo più avanti, utilizzarle come esempio e ridurre drasticamente la dimensione del codice.

Introduciamo quindi il nuovo modificatore **abstract**, il quale è utilizzabile su metodi e classi; nel primo caso consente di descrivere una funzionalità della classe senza definire il suo funzionamento, ragion per cui il metodo non sarà invocabile e definibile esclusivamente in **classi astratte**. Queste ultime hanno la particolarità di avere almeno un metodo astratto, il quale rende impossibile una loro istanziazione.

Qui entra in gioco l'ereditarietà; per poter utilizzare classi astratte sarà necessario estenderle in sottoclassi normali con tutti i metodi definiti. L'utilità è lampante: descrivere un comportamento generale che avrà caratteristiche specifiche per ogni sottoclasse, ridefinito grazie all'override dei metodi. Questo è un paradigma di programmazione, infatti, e si chiama **Template**.

```
1  // Definizione della classe astratta
2  public abstract class Strumento {
3    public String nome;
4    public String prezzo;
5    // Metodo che rende la classe effettivamente astratta
6    public abstract void SuonaFaDiesis();
7  }
8
9  // Estensione concreta di Strumento per renderla istanziabile
10 public class Chitarra extends Strumento {
11   @Override
12   public void SuonaFaDiesis () { /* Ridefinizione */ }
13 }
14
15 // Estensione astratta di strumento
16 public abstract class StrumentoAFiato extends Strumento { }
17
18 public class Flauto extends StrumentoAFiato {
19   @Override
20   public void SuonaFaDiesis () { /* Ridefinizione */ }
21 }
```

Se portiamo il concetto di classe astratta all'estremo, quindi ogni metodo della classe sarà definito come astratto, arriveremo a quella che è considerata un'**interfaccia** pubblica. Rappresentano la parte dell'oggetto visibile ed interagibile dall'esterno, nascondendo i procedimenti interni; sono tutte intrinsecamente definite come pubbliche e astratte.

Naturalmente, non essendo concrete, non possono essere definite e sono viste come tipo esclusivamente per il riferimento. Inoltre, ogni metodo ivi presente è pubblico, mentre i campi sono statici e finali. Le interfacce non si estendono, ma vengono **implementate** con l'apposita keyword **implements**, la quale svolge un lavoro analogo ad extends, permettendo di ereditare la forma dei metodi e dei campi alla sottoclasse implementativa.

```java
// Dichiarazione di interfaccia
public interface Pesabile {
  String UDM = "Kg";    // Implicitamente anche static final
  double getPeso();
}

// Implementazione di interfaccia
public class Articolo implements Pesabile {
  private double peso;
  private String descrizione;

  public Articolo (String descrizione, double peso) {
    setDescrizione(descrizione);
    setPeso(peso);
  }

  @Override
  public double getPeso () {
    return peso;
  }
}
```

Sebbene non si possano ereditare più classi, per una classe concreta è possibile implementare più interfacce allo stesso tempo, mentre un'interfaccia può estenderne una o multiple senza poterle implementare. Si potrebbe pensare in questi termini, di scrivere un singolo metodo per ogni interfaccia, usando il concetto di **interfaccia funzionale**.

Fondamentalmente risulta utile crearle per evitare dipendenze fra i metodi nelle classi che le implementano. Una classe che fa uso di un'interfaccia deve necessariamente definire ogni suo metodo, ma se questa ne ha uno solo, il problema non si pone.

In precedenza abbiamo nominato come sia possibile scrivere più classi all'interno di un singolo file. Quando si dichiara una classe locale senza nome, questa è detta **anonima**; si tratta di classi istanziate ed utilizzate una singola volta, ovvero quando ne è chiamato il metodo. Questa caratteristica del linguaggio è resa possibile solo grazie all'ereditarietà, perché per essere usate, le classi anonime devono necessariamente estendere una classe oppure implementate un'interfaccia. Sono viste inoltre come **espressioni**, ed in quanto tali devono far parte di un enunciato e non possiamo specificarne un costruttore.

È possibile fondere questo concetto alle interfacce funzionali, permettendoci di ottenere ciò che è conosciuto come **funzione lambda**. La loro ragion d'essere è rendere il codice più compatto, leggibile ed indipendente. Hanno sintassi:

listaDiParametri -> bloccoDiCodice

```java
// Istanziazione di singleInc, come gia' visto.
SingleInterface singleInc = new SingleInterface () {
  @Override
```

```
    public int singleMethod(int param) { return param + 1; }
};

// Utilizzo della lambda. Risparmiate ben tre righe.
SingleInterface singleDec = param -> param-1;

System.out.println(singleInc.singleMethod(3)); // Output: 4
System.out.println(singleDec.singleMethod(3)); // Output: 2
```

Notare che in quanto classi anonime, le lambda erediteranno le loro dinamiche. Inoltre, i parametri che può accettare possono andare da zero a molti, mentre il corpo della funzione è necessariamente una sola espressione oppure un blocco di codice.

Un ultima cosa utile, ma non necessariamente sicura o consigliata è la possibilità di passare ai metodi una variabile argomenti, detta **varargs**; si traduce nel passaggio di un array e può aiutare a ridurre la dimensione del codice. La sintassi è:

nomeMetodo(tipo... nomeParametro)

```
public interface SingleInterface { int singleMethod(int... params); }

// Utilizzo di varargs con un normale metodo, classe anonima.
SingleInterface singleInc = new SingleInterface() {
  @Override
  public int singleMethod (int... params) { return params [0] + 1; }
};

// Utilizzo di varargs con una lambda
SingleInterface singleDec = params -> params [0]-1;

System.out.println(singleInc.singleMethod(3, 4, 5)); // Output: 4
System.out.println(singleDec.singleMethod(3, 2)); // Output: 2
```

Chapter 4

Strutturazione e mantenimento

Nella seconda parte del corso, Ingegneria del Software, si approfondirà meglio il concetto di strutturazione e creazione del codice, tuttavia, è necessario fornire qualche nozione già da ora per iniziare a organizzare correttamente il codice. Un primo modus operandi è dato dai **modelli di progettazione** o design pattern. Si tratta di soluzioni standardizzate a problemi comuni in ambito di sviluppo software e risultano utili per dare un'idea del workflow adattato, evitando di reinventare la ruota. Consentono, di conseguenza, di creare codice più mantenibile ed efficiente in una struttura ripetibile. Si dividono in tre categorie:

- **Creazionali**: Rendono un sistema indipendente dal modo in cui gli oggetti sono istanziati ed elaborati.
- **Strutturali**: Scolpiscono una composizione di classi ed oggetti per formare una struttura più grande. Utili per facilitare la progettazione e minimizzare le dipendenze fra le parti.
- **Comportamentali**: Lavorano sull'assegnazione di responsabilità fra gli oggetti, facendo dipendere un sistema dalla composizione e l'interazione fra gli oggetti.

Il motivo per cui utilizzare queste strategie è autoesplicativo: scrivere codice pulito, mantenibile e performante.

4.1 Singleton

Appartenente ai pattern creazionali, implica che ogni classe vada a rappresentare un concetto il quale richieda l'implementazione di una singola sua istanza. Questo modello è utilizzato in ambiti dove è richiesto limitare l'istanziazione degli oggetti.

```
1  public RingOfPower {
2    private static String owner = "Sauron";
```

```
    private static RingOfPower instance;

    private RingOfPower () {}
    // Blocca eventuali creazioni di istanze e ritorna quella esistente
    public static RingOfPower getInstance () {
      if (instance == null) { instance = new RingOfPower(); }
      return instance;
    }
  }
  public String getOwner () { return owner; }
```

Naturalmente, essendo un metodo static, per ottenere l'istanza sarà necessario chiamarla con **RingOfPower.getInstance()**.

4.2 Documentazione Javadoc

La documentazione del codice è fondamentale per chiunque debba andare a metterci le mani. Java ha un tool apposito chiamato **Javadoc** che consente di generarla automaticamente ed in modo standardizzato. Crea infatti una pagina html che descrive classi, metodi e campi ove indicato nel codice.

Si suppone che si stia utilizzando Maven per costruire in automatico il programma; essendo Javadoc non nativo del linguaggio, bisognerà aggiungere la seguente dipendenza al pom.xml:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>3.6.2</version>
      <configuration>
        <source>1.8</source>
        <show>private</show>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Controllato il corretto inserimento del plugin nel file di costruzione, sarà possibile aggiungere i commenti di tipo Javadoc. Si tratta di commenti multilinea dove è possibile aggiungere una descrizione e, tramite annotazioni, specificare autore, parametri, tipo ritornato e altro.

Attenzione: il commento va posizionato nella riga antecedente l'elemento che vogliamo commentare. Segue esempio completo:

```
/** Il commento sara' letto dalla javadoc
Le righe prive di asterischi non saranno lette.
```

```
 * @author F27
 * @param arr    Segnala un parametro
 * @return     Descrive il valore ritornato dal metodo
 * @throws     Descrive le eccezioni lanciabili
 * @link       Link per portare l'utente alla classe
 * @deprecated   Spiegazione del perche' il metodo sia deprecato
 */
```

4.3 Gestione degli errori

Nella scrittura di programmi è inevitabile incappare in errori o situazioni non ideali; Java porta delle soluzioni per la loro gestione nelle forme di routine di interruzione date dalla classe **Exception**, implementabile con le keyword:

- **try**: Prova ad eseguire del codice
- **throw**: Nel verificarsi di una situazione anomala, lancia l'eccezione
- **catch**: Cattura l'eccezione e gestiscila col conseguente blocco di codice.

Esistono delle classi di eccezioni native di alcune librerie, ma è possibile anche definirle noi stessi, estendendo Exception. Per utilizzare questa funzionalità è necessario dichiarare che una determinata classe può lanciare un'eccezione.

```
// Il metodo pop puo' lanciare le eccezioni indicate
public Object pop () throws EmptyStackException, IOException { ... }

// Prova ad eseguire il codice
try {
  // Chiama il metodo. Se non lancia nulla, continua
  stack.pop();
  // Ad eccezione lanciata, catturala
} catch (EmptyStackException e) {
  // E gestisci l'errore con il seguente blocco di codice
  System.out.println(e);
}
```

Per controllare le eccezioni è possibile definire più blocchi catch, ma verrà eseguito solamente il primo in cui si entra. Inoltre, è presente una gerarchia di eccezioni, definite come:

- **Checked**: Situazioni anomale prevedibili, devono essere dichiarate e quindi controllate dal compiler. Generate con throw. Ne fanno parte **Exception**, **IOException** e le eccezioni user-defined.

- **Unchecked**: Situazioni anomale non prevedibili, non devono essere dichiarate, né catturate. Gestite dalla JVM. Ne fanno parte **Error**, **ThreadDeath**, **RuntimeException**, **NullPointerException**.

Il fatto che le eccezioni controllate debbano essere dichiarate su ogni singolo metodo che può lanciarle aumenta la complessità del codice in modo importante, quindi, sebbene risultino più sicure, è consigliato restringerne l'uso esclusivamente in caso di estrema necessità. In alternativa è possibile sfruttare un loophole, lanciando ogni eccezione controllata sotto forma di non controllata e lasciando il lavoro sporco alla JVM, in questo modo:

```
1  public class MyException extends Exceptions { } // checked
2
3  public void foo () throws MyException {
4    throw new MyException();
5  }
6
7  // Non e' richiesta alcuna dichiarazione...
8  public void bar () {
9    try {
10     // ...perche' la chiamera' il metodo se serve...
11     foo();
12     // ...ed entrando qui, la rilanci come unchecked.
13   } catch (MyException e) {
14     throw new RuntimeException(e);  // Lavoro: sbolognato.
15   }
16 }
```

4.4 Unit testing con Junit

Ora che siamo capaci di gestire gli errori con le eccezioni è ora di fare un passo più avanti nell'ambito del controllo qualità con il concetto di **unit testing**; un workflow mirato alla verifica di un dato comportamento dei metodi in una classe. Quando un dato scenario presenta il risultato previsto, si dice che il test è stato passato.

Risulta particolarmente utile perché in primo luogo costringe il codice ad essere consistente con le nuove versioni, riducendo allo simultaneamente i tempi di debug riutilizzando i test già scritti. Quelli che indichiamo come **test case** rappresentano gli scenari ottenibili dal codice scritto; sono composti da input e output previsto e sono contenuti in classi apposite dette **test suit**.

Per fare unit testing in Java viene utilizzato un framework chiamato **JUnit**, che porta con sé annotazioni apposite per la creazione dei test case. Questi vengono eseguiti tutti e, in presenza problemi, verranno opportunamente segnalati. Non essendo nativo del

linguaggio, è necessario aggiungere la seguente dipendenza al file di configurazione di Maven:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.10.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.5.4</version>
  </dependency>
</dependencies>
```

Nella strutturazione dei progetti è buona prassi separare le classi di codice sorgente da quelle di testing, quindi consiglio di creare una directory un livello superiore alla principale e chiamarla **test** e rispettare la naming convention dei test suits: lo stesso nome della classe che testano con "Test" concatenato alla fine. Detto ciò abbiamo tutti gli strumenti per la scrittura del nostro test suit:

```
// Pacchetto di Junit dal quale e' presa l'annotazione Test
import org.junit.jupiter.api.Test;

// Pacchetto con metodi per la verifica dei comportamenti
import static org.junit.jupiter.api.Assertions.*;

public class AccountTest {
  @Test // Annotazione per indicare che questo e' un test case
  public void checkNullAccount () { assertThrows(NullPointerException
 .class, () -> new Account(null)); }
}
```

Notare che i test case non hanno un return type perché lavorano tramite asserzioni, quindi se il test non è passato JUnit lancia AssertionFailedError, per poi catturare l'errore e gestirlo. Le asserzioni di uso più comune sono:

- static void assertThrows(condition, message): Lancia se si verifica la condizione.
- static void assertTrue(condition, message): Lancia se vede un boolean true.
- static void assertEquals(expected, actual, message): Lancia se expected è uguale ad actual.
- static void assertArrayEquals(expected, actual, message): Idem, ma per gli array.

Nel pacchetto di Junit sono presenti anche altre annotazioni utili per la gestione del flusso dei test:

- @DisplayName: Mostra nome per una classe o metodo test
- @BeforeEach: Il metodo sarà eseguito prima di e per ogni test case
- @AfterEach: Il metodo sarà eseguito dopo di e per ogni test case
- @BeforeAll: Il metodo sarà eseguito una sola volta prima di tutti gli altri test
- @AfterAll: Il metodo sarà eseguito una vola volta dopo tutti gli altri test
- @Disabled: Disabilita una classe o metodo di test

4.5 Gestione files CSV e JSON

Sebbene Java non abbia supporto nativo per la gestione di file testuali standardizzati come i **Comma Separated Values** CSV o i **JavaScript Object Notation** JSON, è possibile aggiungere dipendenze al file di configurazione di Maven per aggiungere metodi utili al parsing.

- File CSV
File testuale rappresentante dati tabulari, come i contenuti di un database. Vede le colonne separate da virgole e le righe separate da newlines. Dipendenza:

```
<dependency>
  <groupId>com.opencsv</groupId>
  <artifactId>opencsv</artifactId>
  <version>5.12.0</version>
</dependency>
```

Per la lettura del file:

```
// Leggi un CSV da un oggetto File con il reader di default
File csvFile = new File ("i.csv");
CSVReader csvReader = new CSVReader(new FileReader(csvFile));

// Leggi un CSV da un fileName con un reader user-defined
CSVReader csvReader = new CSVReader Builder(new FileReader("i.csv"))
  .withSkipLines(1)
  .withSeparator(';')
  .build();

// Leggi il CSV riga per riga
List<String[]> table = new ArrayList<>();

// Qui eventualmente lancia CsvException
for(String[] row; (row = csvReader.readNext()) != null; ) {
  table.add(row);
```

```
}

// Leggi l'intero CSV in una volta; eventualmente lancia CsvException
List<String[]> table = csvReader.readAll();

```

Per la scrittura del file:

```
// Scrivi un CSV da un oggetto File col writer di default
File csvFile = new File("o.csv");
CSVWriter csvWriter = new CSVWriter(new FileWriter(csvFile));

// Scrivi un CSV con un fileName con un writer user-defined
CSVWriter csvWriter = new CSVWriter(new FileWriter("o.csv"),';');

// Scrivi un CSV riga per riga
for (String[] row : table) csvWriter.writeNext(row);
csvWriter.flush();    // Non necessario se si chiude il file

// Scrivi l'intero CSV in una volta
csvWriter.writeAll(table);
csvWriter.flush();    // Non necessario se si chiude il file
```

- File JSON

File testuale che offre una rappresentazione leggibile di dati strutturati. Anche questo è un formato standard, ed è usato universalmente sul web. Ogni Json possiede oggetti, contenuti nelle sotto forma di coppie key-value, e array, liste ordinate di valori delimitati nelle []. I valori in questione possono essere stringhe, numeri, booleani, oggetti o altri array.

Viene generalmente utilizzata la libreria di google, che comprende la classe Gson coi metodi fromJson e toJson, la cui dipendenza per Maven è data da:

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.13.2</version>
</dependency>
```

Costruzione di file JSON da un'istanza di classe:

```
Gson gson = new Gson();
Student sam = new Student("Sam", 123);
String json = gson.toJson(sam);
System.out.println(json); // Stampa: '{" matricola ":123 ," name ":"
 Sam "}'
```

Parsing di un file JSON in un'istanza di classe:

```
Gson gson = new Gson();
String json = "{\"matricola \":123,\"name\":\"Sam\"}";
```

```
Student sam = gson.fromJson(json, Student.class); // Lancia
 JsonSyntaxException
System.out.println(sam.toString()); // Stampa 'Sam 123'
```

Scrivi un file JSON:

```
try (FileWriter writer = new FileWriter("sam.json")) {
  Gson gson = GsonBuilder().setPrettyPrinting().create();
  Student sam = new Student("Sam", 123);
  gson.toJson(sam, writer);   // Lancia JsonIOException
} catch (IOException ioe) {
  throw new RuntimeException(ioe);
}
```

Leggi da un file JSON:

```
try (FileReader reader = new FileReader("sam.json")) {
  Gson gson = new Gson();

  // Lancia JsonSyntaxException, JsonIOException
  Student sam = gson.fromJson(reader, Student.class);
} catch (IOException ioe) {
  throw new RuntimeException(ioe);
}
```

Chapter 5

Librerie utili

5.1 Libreria digitale

Le librerie digitali di Java sono un insieme di funzioni contenute in **pacchetti** appositi e fornite da terze parti. Come altri utenti ne hanno scritte, così potremmo fare anche noi. In ogni caso, tutti i file da eseguire sono visti come classi, ed infatti dovranno essere compresi nel classpath, se non sono standard.

Per utilizzare una libreria è necessario importarla nel file desiderato con la keyword **import**. Per esempio, nella libreria java.util è presente la classe Scanner, che viene usata per ricevere input da tastiera.

```
1  // Aggiunge il pacchetto Scanner dal path java/util/Scanner
2  import java.util.Scanner;
3
4  public class Mult {
5    public static void main(String args[]) {
6
7      // Dichiarazione dell'oggetto keyScan di classe Scanner
8      Scanner keyScan = new Scanner(System.in);
9      int n1, n2;
10
11     System.out.print("Inserisci il primo fattore: ");
12     // Assegna a n1 l'intero letto da tastiera, idem n2.
13     n1 = keyScan.nextInt();
14     System.out.print("Inserisci il secondo fattore: ");
15     n2 = keyScan.nextInt();
16
17     // Chiudi lo scanner con il metodo close().
18     keyScan.close();
19     System.out.println("Risultato: " + n1*n2);
20   }
21 }
```

Quindi abbiamo capito che un pacchetto è un insieme di classi le quali sono logicamente correlate. Lo scopo principale è strutturare il codice in modo più chiaro grazie alla modularità che ne consegue. In termini più semplici, è considerabile come una libreria le cui interfacce sono date dalle classi.

I pacchetti possono essere innestati ed il path è dato dalla segnatura come visto nello snippet; i punti sostituiscono lo slash. In assenza di una dichiarazione del pacchetto, al file verrà associato quello di default, **package private**, il quale lo renderà inaccessibile da altri file. L'uso di quest'ultimo è sconsigliato e considerato cattiva pratica.

La struttura dei pacchetti influenza anche la visibilità delle classi. Per esempio, supponiamo di avere un classpath di pacchetti "it.univr.bugs". Se inseriamo il main all'interno della directory univr e le classi funzionali all'interno di bugs, queste ultime non saranno visibili dalla prima; quindi è necessario importare il pacchetto con le modalità già viste.

```
1 package it.univr; // Specifica del path corrente
2 import it.univr.bugs.*;   // Per includere ogni classe
3 import it.univr.bugs.Hornet;  // Per includere una classe specifica
```

5.2 java.lang.*

Il pacchetto java.lang.* comprende la libreria nativa del linguaggio, contenente tutte le funzioni di base come costrutti, oggetti e altri strumenti.

5.2.1 Classi wrapper

Le classi wrapper rappresentano sotto forma di classe i tipi primitivi. Sono compresi di campi e metodi per lavorare in un ambiente completamente orientato agli oggetti. Sono immutabili e si definiscono con lo stesso nome dei tipi primitivi, ma con la prima lettera maiuscola; abbiamo quindi: **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Char** e **Boolean**.

Originalmente, per ottenere i valori da queste classi, era necessario fare wrapping e unwrapping coi relativi metodi, tuttavia, da Java 5, è stato implementato nel compilatore il concetto di **autoboxing** e **autounboxing**, consentendo di gestire le classi wrapper come si farebbe coi tipi primitivi.

```
1 Integer wi = 2;
2 int j = 0;
3
4 j = wi + 5; // Sostituisce j = wi.intValue()+5;
5 wi = 3;   // Sostituisce wi = new Integer(3);
6 wi = j;   // Sostituisce wi = new Integer(j);
```

In particolare, la classe Character ha alcuni metodi statici di utility dal nome particolarmente esaustivo, come per esempio isLetter(), isDigit(), isSpaceChar(), toUpperCase(), toLowerCase() e altri.

5.2.2 .System

La classe System non può essere istanziata e comprende campi e metodi utili per la gestione di input/output standard e flusso. Dà inoltre accesso a proprietà esternamente definite e variabili di ambiente, permette di caricare file e librerie, ed infine un metodo di utility per copiare velocemente una porzione di array.

```
1  // Campi visti
2  static PrintStream err    // stderr stream
3  static InputStream in   // stdin stream
4  static PrintStream out    // stdout stream
5
6  // Metodi visti
7  static long currentTimeMillis() // Ritorna l'ora attuale
```

5.2.3 .String, .StringBuilder

Le classi String, StringBuilder e StringJoiner sono utilizzate per la manipolazione di stringhe. Lavorano su oggetti immutabili. Le ultime due possono anche prendere delle stringhe già esistenti e convertirle in loro oggetti.

```
1  // Metodi visti - String
2  String("Caratteri")   // Costruttore
3  int length()      // Lunghezza della stringa
4  char charAt(int index)    // Carattere a indice index
5  int indexOf(int ch)   // ch in stringa? Y=indice/n=-1
6  String substring(int beginIndex, int endIndex) // Ottiene substring
7  String replace(char oldChar, char newChar) // Sostituisce old con new
8
9  // Metodi visti - StringBuilder
10 StringBuilder("Caratteri")  // Costruttore
11 StringBuilder append("caratteri") // Concatena alla stringa
12 StringBuilder insert(int index, String str) // inserisce str a
    partire da index
13 StringBuilder delete(int beginIndex, int ednIndex) // Cancella
    caratteri negli indici
```

5.2.4 .Math

La classe Math non è istanziabile e contiene metodi per effettuare operazioni numeriche di base come esponenziali, logaritmi, radici e funzioni trigonometriche.

```
// Campi visti
static double E   // 2.71
static double PI  // 3.14

// Metodi visti
static int abs (int n)  // Valore assoluto di n
static double log (double n)  // logaritmo in base e di n
static double log10 (double n)  // logaritmo in base 10 di n
static int max (int a, int b) // Massimo fra a,b
static int min (int a, int b)   // Minimo fra a,b
static long round (double d)  // Approssimazione per ecc/dif
static double sin (double r)  // Seno di angolo r in radianti
static double cos (double r)  // Coseno di angolo r in radianti
static double tan (double r)  // Tangente di angolo r in radianti
static double sqrt (double n)   // Radice quadrata positiva di n
static double toDegrees (double r)  // Passa da rad a deg
static double toRadians (double d)  // Passa da deg a rad
```

5.2.5 .Comparable, .Comparator

L'interfaccia Comparable porta un metodo utile generalizzato per paragonare due oggetti.

```
// Negativo se this<obj, zero se this=obj e positivo se this>obj
public int compareTo (T obj)
```

Esempio di utilizzo interfaccia Comparable mantenendo la sua logica facendo override del metoto compareTo:

```
public class Student extends Person implements Comparable {
  // ...
  @Override
  public int compareTo(Object obj) {
    Student other = (Student) obj;  // cast al tipo specifico
    return this.matricola - other.matricola;
  }
}
```

5.2.6 .Iterable, .Iterator

Se una classe implementa l'interfaccia Iterable, i suoi oggetti potranno essere iterati con un ciclo for-each grazie al suo uso dell'interfaccia Iterator, parte del JCF, il quale sarà approfondito più avanti.

```
// Metodi visti
boolean hasNext ()  // Vero se l'iterazione ha piu' elementi
E next () // Torna il prossimo elemento
```

Esempio di utilizzo:

```
import java.util.Iterator;
import java.lang.Iterable;

class ClassRoom implements Iterable {
  private Student[] students;

  ClassRoom(Student[] students) {
    this.students = students;
  }

  @Override
  public Iterator iterator () {
    return new CRIterator(students);
  }
}
```

```
import java.util.Iterator;

class CRIterator implements Iterator {
  private Student[] arr;
  private int next = 0;
}

CRIterator (Student[] arr) { this.arr = arr; }

@Override
public boolean hasNext () {
  return next < arr.length;
}

@Override
public Object next () {
  return new Student(arr[next++]);
}
```

```
Student[] arrS = {new Student("Sam", 456), new Student("Paul", 123)};
ClassRoom classRoom = new ClassRoom(arrS);

for (Object obj:classRoom) {
  System.out.println(((Student)obj).toString)
}
```

5.3 java.util.*

Il pacchetto java.util.* contiene varie funzioni di utility per rendere il codice più leggibile, astratto, modulare e sicuro.

5.3.1 .Random

La classe Random è istanziabile, ed è usata per la generazione di variabili pseudocasuali.

```
// Costruttori
Random()
Random(long seed)

// Metodi
boolean nextBoolean() // Casuale true/false
```

```
double nextDouble()   // Decimale fra [0.0, 1.0)
float nextFloat()   // Decimale fra [0.0, 1.0)
int nextInt()     // Casuale intero [-2^31, 2^31)
int nextInt(int max)  // Casuale intero [0, max)
long nextLong()     // Casuale intero [-2^63, 2^63)
```

5.3.2 Java Collection Framework

La Java Collection Framework è un insieme di strutture dati iterabili definite tramite l'uso di parametri generici, rendendole utilizzabili in qualunque contesto. Si tratta di uno dei pacchetti più importanti di tutto il linguaggio.

Tutte le strutture dati del framework estendono l'interfaccia padre **Collection**, la quale porta svariati metodi utili:

```
public interface Collection<E> extends Iterable<E> {
  int size ();
  int hasCode ();
  boolean equals (Object obj); // Notare equals non generic
  boolean isEmpty ();
  boolean contains (Object obj);
  boolean containsAll (Collection<?> c);
  boolean add (E e);
  boolean addAll (Collection<? extends E>);
  boolean remove (Object obj);
  boolean removeAll (Collection <?> c);
  void clear ();
  Object[] toArray ();
  Iterator<E> iterator (); // from Iterable<E>
}
```

le strutture dati sottoclassi di Collection si riassumono in tre macrocategorie, date da:

- **List**: Una lista di elementi, la quale può avere duplicati e dove si mantiene l'ordine di inserimento. Sottoclassi sono **ArrayList** e **LinkedList**.
- **Queue**: Una lista di elementi legati da una relazione di ordinamento. Sottoclassi sono **Dequeue** e **PriorityQueue**.
- **Set**: Insieme di elementi unici. Sottoclasse è **HashSet**.

Il modo corretto per la dichiarazione di queste strutture è istanziare la sottoclasse a partire dalla sovracclasse, funzionale grazie al principio di sostituzione.

```
// Esempio di implementazione
Collection<Person> list = new LinkedList<>(); // Crea lista
list.add(new Person("Joe"));  // Aggiunge Joe in coda alla lista
list.add(new Person("Sam"));
```

Naturalmente ogni struttura avrà metodi per facilitarne il workflow. Abbiamo un insieme di metodi condivisi fra List e Queue, mentre per Set sono sufficienti quelli ereditati da Collection.

```
// Metodi per List e Queue
E get (int i);  // Ottieni l'elemento a posizione i
E set(int i, E e);  // Rimpiazza l'elemento e a posizione i
void set(int i, E e); // Aggiungi elemento e a posizione i
E remove(int i);  // Rimuovi elemento a posizione i
int indexOf(E e); // Ritorna la posizione dell'elemento e
List<E> subList(int l, int r);  // Ritorna lista fra gli indici [l,r)
```

Un'utile implementazione di Set si ha con la sua sottoclasse **HashSet**, la quale consente l'accesso agli elementi della struttura a tempo costante in base a una determinata chiave. Questa struttura dati mantiene ineternalmente una HashMap, la quale garantisce l'unicità degli elementi usando il loro **hashcode** come chiave.

Per una corretta generazione del codice si deve usare il metodo **hash(obj1, obj2)**, dato dalla classe Object, per poi fare override di equals() con lo scopo di basarlo sulla chiave. Un'altra implementazione utile di Set è **TreeSet**, basata sulla logica degli RB-Alberi. Aggiunge i metodi first() e last() per prendere il nodo radice e l'ultimo del cammino rispettivamente.

Un'ultima struttura sempre facente parte del JCF ma non ereditante Collection è la **Map<K, V>**, dove K sta per key e V per value. Si tratta di un insieme di associazioni che collegano una specifica chiave ad un determinato valore. Capirai che Set e Map sono strettamente legati. Presenta i seguenti metodi:

```
V put(K key, V values); // Inserisci associamento
V get(Object key);  // Torna il valore associato alla chiave
V remove(Object key); // Rimuovi l'associamento della chiave
boolean containsKey(Object key);  // Vedi se la mappa ha una chiave
boolean containsValue(Object value);  // Torna se valore in mappa
Set<K> keySet();  // Ritorna le chiavi della mappa
Collection<V> values(); // Ritorna i valori nella mappa
```

La JCF rende il il codice estremamente standardizzato; tuttavia non basta creare gli oggetti, bisogna anche poterci lavorare.

Le mansioni principali sono quelle di scorrere gli elementi e il loro ordinamento; questo lavoro è reso semplice grazie alla sinergia con le interfacce Iterable e Comparable e di conseguenza anche con Iterator e Comparator.

5.3.3 Interfacce funzionali primitive

Le interfacce funzionali sono un argomento già menzionato nei capitoli precedenti, tuttavia è possibile usufruire delle loro versioni base e di uso comune definite in questo

pacchetto, date da:

- Predicate
Prende un parametro di tipo T e ritorna un boolean. Viene spesso usato per filtrare elementi in una collection.

```
@FunctionalInterface
public interface Predicate<T> {
  boolean test(T t);  // Qui T e' generics, ma puo' essere
 specificato
}

// Versione semplice
Predicate<Integer> isEven = new Predicate<>() {
  @Override
  public boolean test(Integer n) { return n%2 == 0; }
}

// Versione lambda
Predicate<Integer> isEven = n -> n%2 == 0;

System.out.println(isEven.test(2)); // true
System.out.println(isEven.test(3)); // false
```

- Function
Prende un parametro di tipo T e ritorna un risultato di tipo R. Spesso usato per mappare un valore ad un altro.

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}

// Function strLen prende String e ritorna Integer
Function<String, Integer> strLen = str -> str.length();

Systm.out.println(strLen.apply("Hello")); // Qua stampa 5
```

- BiPredicate
Prende due parametri di tipi T e U per ritornare un boolean

```
@FunctionalInterface
public interface BiPredicate<T, U> {
  boolean test(T t, U u);
}

// La lambda isGreaterThan prende i due parametri e ritorna se la
 condizione e' vera
BiPredicate<Integer, Integer> isGreaterThan = (a,b) -> a > b

```

```
System.out.println(isGreaterThan.test(2, 5)); // false
System.out.println(isGreaterThan.test(3, 1)); // true
```

- BiFunction

Prende due parametri di tipi T, U e ritorna un risultato di tipo R, spesso usata per implementare operazioni su due valori.

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
  R apply(T t, U u);
}

BiFunction<Integer, Integer, Integer> sumInt = (a,b) -> a+b;

System.out.println(sumInt.apply(3,9));  // Stampa 12
```

- Consumer

Prende un parametro di tipo T e non ritorna niente. Spesso usata per modellare effetti collaterali.

```
@FunctionalInterface
public interface Consumer<T> {
  void accept(T t);
}

// La lambda ha un effetto sull'argomento, ma non ritorna nulla
Consumer<String> printUC = str -> System.out.println(str.toUpperCase
  ());

printUC.accept("string"); // Stampa STRING
```

- Supplier

Funzione che non prende parametri ma ritorna un valore di tipo T; spesso usata per la generazione di valori

```
@FunctionalInterface
public interface Supplier<T> {
  T get();
}

Supplier<Double> randomNum = () -> Math.random();
System.out.println(randomNum.get());    // Stampa il numero casuale
```

Ulteriori interfacce funzionali sono date da:

UnaryOperator<T> extends Function<T, T>, **BinaryOperator<T>** extends BiFunction<T, T, T>

Dove la prima prende un parametro T per ritornarne un altro, mentre la seconda ne prende due di tipo T per ritornarne un terzo. Esiste infine una sintassi compatta per fare method reference:

```
// Sintassi compatta
System.out::println
// Equivale a
str -> System.out.println(str)
```

Generalmente, valgono queste regole:

- Per metodi statici: Class::staticMethod
- Per metodi d'istanza: object::instanceMethod
- Per metodi non statici: Class::nonStaticMethod
- Per il costruttore: Class::new

5.3.4 .Stream

Da Java8, il pacchetto java.util.stream contiene classi apposite per processare i flussi di dati. Un flusso è una sequenza di elementi di una fonte e si definisce con la classe **Stream<T>**. La sequenza supporta operazioni di processione dati, definite tramite interfacce funzionali. I flussi hanno nativamente pipelining, iterazioni interne e parallelizzazioni. Alcuni utilizzi comuni sono:

```
// Stream.of(T... elems) scorre ogni elemento dato come parametro.
Stream<String> stream = Stream.of("a", "b", "c");

// Arrays.stream(T[] arr) scorre ogni elemento di un dato array.
String[] arr = new String[] {"d", "e", "f"};
Stream<String> stream = Arrays.stream(arr);

// Collection.stream() scorre ogni elemento di una collezione del JCF
Set<String> set = Set.of("g", "h", "i");
Stream<String> stream = set.stream();

// Crea una lista a partire dallo stream
List<Integer> list = Stream.of(1, 2, 2, 3).toList();

// Crea un set a partire dallo stream
Set<Integer> set = Stream.of(1, 2, 2, 3).collect(Collectors.toSet());

// Crea un array a partire dallo stream
Integer[] arr = Stream.of(1,2,2,3).toArray(Integer[]::new);
```

È possibile generare degli stream di vario tipo. Parliamo di un'operazione che applica un operatore unario ripetitivamente su un dato seme, la cui segnatura è data da:

Stream<T> iterate(T seed, UnaryOperator<T> generator)

Questo metodo ritornerà un nuovo flusso con gli elementi generati. Abbiamo altre tecniche come:

```
// Generazione infinita (stream lazy)
Stream<Double> stream = Stream.generate(Math::random);
// La computazione sara' infinita
List<Double> list = stream.toList();

// Generazione controllata da 0 con indice i, fintanto che i<10
Stream<Integer> stream = Stream.iterate(0, i -> i+1).limit(10);
// La lista contiene i numeri da 0 a 9
List<Integer> list = stream.toList();
```

Possiamo inoltre effettuare delle operazioni **intermedie** e **terminali**; le prime ritornano un nuovo Stream<T>, non modificano la fonte e possono essere concatenate con la notazione dot, mentre le seconde ritornano un risultato di un tipo specifico e ne è ammessa solo una. Per esempio:

```
String[] arr = {"a", "b", "c", "d"};

// Ritorna lo stream distinct da fonte arr senza duplicati e ordinata
  .
Stream<String> distinct = Arrays.stream(arr).distinct().sorted();

// Ritorna il totale degli elementi di arr distinti
long count = Arrays.stream(arr).distinct().count();
```

Un concetto più raffinato di questo algoritmo si ha con i **filtri**, operazioni intermedie che applicano un predicato ad ogni elemento del flusso; hanno segnatura:

Stream<T> filter(Predicate<T> predicate)

```
// Via lambda
String[] arr = {"abc", "cde", "ecd", "cba"};
// stream conterra' le stringhe con la d
Stream<String> stream = Arrays.stream(arr)
  .filter(s -> s.contains("d"));

// Via method reference
Character[] arr = {'a', '3', '7', 'd'};
// stream conterra' i caratteri 3 e 7
Stream<Character> stream = Arrays.stream(arr)
  .filter(Character::isDigit);
```

Generalmente, vengono usati spesso i seguenti metodi per filtraggio intermedio:

```
// Ritorna lo stream scartando i duplicati
Stream<T> distinct();
// Ritorna i primi n elementi
Stream<T> limit(int n);
```

```
// Scarta i primi n elementi e ritorna i successivi
Stream<T> skip(int n);
// Ordina gli elementi per ordine naturale. T deve essere comparable
Stream<T> sorted();
```

Naturalmente consegue che se è possibile filtrare degli elementi specifici, possiamo anche eseguire delle operazioni o trasformazioni su di essi. Ciò avviene grazie al metodo **map()**, di segnatura:

Stream<R> map (Function<T,R> mapper)

```
// Via lambda
String[] arr = {"abc", "cdef", "ec", "cba"};
// stream contiene 3, 4, 2, 3
Stream<Integer> stream = Arrays.stream(arr).map(s -> s.length());

// Via method reference
String[] arr = {"aBc", "cd7f", "eC", "2Ba"};
// Lo stream contiene ABC, CD7F, EC, 2BA
Stream<String> stream = Arrays.stream(arr).map(String::toUpperCase);
```

Quando parliamo di operazioni terminali, invece, ribadisco, si attuano sull'intero flusso e non lo ritornano. Metodi di uso comune sono:

```
// Controlla se almeno un elemento matcha il predicato
boolean anyMatch(Predicate<T> predicate);
// Controlla che tutti gli elementi matchino il predicato
boolean allMatch(Predicate<T> predicate);
// Controlla che nessun elemento matchi il predicato
boolean noneMatch(Predicate<T> predicate);
// Trova il primo elemento dello stream che matcha un predicato
Optional<T> findFirst();
// Trova un elemento dello stream che matcha un predicato
Optional<T> findAny()
```

In questo ambito esiste il metodo **orElse()** che mostra una scrittura più compatta al posto di usare un costrutto condizionale:

```
// Salva in opt un elemento e minore o uguale a 0
Integer[] arr = {-1, 4, -5, 0, 3};
Opional<Integer> opt = Arrays.stream(arr)
  .filter(e -> e <= 0).findAny();

// Stampa opt se non e' null, altrimenti stampa 0
System.out.println(opt.orElse(0));

/* Scrittura equivalente a quanto segue:
 * if (opt.isPresent()) System.out.println(opt.get());
 * else System.out.println(0);
 */
```

Possiamo inoltre iterare fra gli elementi del flusso. È un'azione eseguita su ogni singolo elemento e lo stream viene **consumato** a fine operazione. Segnatura:

void forEach (Consumer<T> action)

```
// Stampa gli elementi di un flusso
Integer[] arr = {-1, 4, -5, 0, 3};
Arrays.stream(arr).filter(e -> e <= 0).forEach(System.out::println);

// Conta gli elementi di un flusso
System.out.println(Arrays.stream(arr).filter(e -> e<= 0).count());
```

Infine, esiste un'operazione intermedia che estrae un flusso da ogni elemento dello stream dato come fonte, chiamata **flatMap()**, con segnatura:

Stream<R> flatMap(Function<T, Stream<R», mapper)

Quindi si effettua una trasformazione per ogni elemento T, il quale viene poi ritornato come singolo flusso. Questi ultimi saranno concatenati e poi ritornati come un singolo Stream di tipo R. Tipicamente, come è facile dedurre, T rappresenta più valori e il ruolo dello stream finale è tendenzialmente affidato ad una Collection.

```
// Caso di uso con array
Byte[][] arr ={{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

// stream contiene 1, 2, 3, 4, 5, 6, 7, 8, 9
Stream<Byte> stream = Arrays.stream(arr)
  .flatMap(r -> Arrays.stream(r));
```

```
// Caso di uso con Collection
List<List<Integer>> list = new LinkedList<>();
list.add(new LinkedList<>()); list.add(new LinkedList<>());

// Aggiungi cifre a posizione get(i)
list.get(0).add(1); list.get(0).add(3);
list.get(1).add(7); list.get(1).add(9);

// stream contiene 1, 3, 7, 9
Stream<Integer> stream = list.stream().flatMap(Collection::stream);
```

```
// Scrittura ulteriormente concatenata
Integer[][] arr = {{1, -2, 3}, {4, -5, -6}, {-7, -8, 9}};

Arrays.stream(arr)
  .flatMap(Arrays::stream)
  .filter(i -> i > 0)
  .map(i -> String.format("%d is positive", i))
  .forEach(System.out::println);

```

```
/* output risultante:
 * '1 is positive'
 * '3 is positive'
 * '4 is positive'
 * '9 is positive'
 */
```

5.4 java.io.*

Il pacchetto java.io.* fornisce classi ed eccezioni relative alla gestione di un flusso di dati, sia in input, che in output. In particolare, le operazioni dipendono da un'astrazione simil-stream, la quale, sebbene funzioni allo stesso modo dei metodi dell'interfaccia Stream, non ne fa alcun utilizzo.

I flussi di I/O possono essere utilizzati con file statici sul disco, standard input, standard output, standard error, una connessione oppure uno stream di dati da/a dispositivi hardware; come vedremo, esistono due tipi di stream con i quali possiamo lavorare e condivisono i seguenti metodi:

- **read()**: Ritorna uno o più byte/caratteri dallo stream.
- **write()**: Inserisce uno o più byte/caratteri nello stream.
- **close()**: Chiude lo stream.

- Flussi orientati ai Byte
Flusso di Byte gestito dalle classi astratte **InputStream** e **OutputStream**, spesso usato per la gestione di dati in binario come immagini, audio o bytecode. Oltre al costruttore, che apre lo stream, presentano i seguenti metodi, le cui implementazioni concrete sono specifiche del mezzo attraverso il quale si fa I/O e si definiscono con le seguenti sottoclassi:

- **BufferedInputStream/BufferedOutputStream**: Legge/scrive dati con buffer.
- **DataInputStream/DataOutputStream**: Legge/scrive tipi primitivi di Java.
- **FileInputStream/FileOutputStream**: Legge/scrive un file.

```
// Esempio di gestione IO con file binario
public static void main (String args[]) throws IOException {

  FileInputStreamfis = null;
  FileOutputStreamfos = null;

  try {
    fis = new FileInputStream(args[0]);
```

```
      fos = new FileOutputStream(args[1]);
      int b;
      while ((b = fis.read()) != -1) {     // lancia IOException
        fos.write(b);            // lancia IOException
      }
    } catch (FileNotFoundException fnfe) {
      System.out.println("File not found...");
    } finally {
      if (fis != null) fis.close();      // lancia IOException
      if (fos != null) fos.close();      // lancia IOException
    }
  }
```

- Flussi orientati ai caratteri
Flusso di caratteri Unicode a 16b gestito dalle classi astratte **Reader** e **Writer**, tendenzialmente usato per gestire dati testuali, come stringhe. Anche qui le implementazioni concrete dipendono dal mezzo e sono date dalle seguenti sottoclassi:

- **BufferedReader/BufferedWriter**: Leggi/Scrivi con buffer.
- **InputStreamReader/OutputStreamWriter**: Wrapper per leggere/scrivere da/a un mezzo basato su byte.
- **StringReader/StringWriter**: Wrapper per leggere/scrivere da/a una stringa.
- **FileReader/FileWriter**: Legge/scrive da/su un file.

```
// Esempio di gestione IO con file testuale
public static void main (String[] args) throws IOException {

  BufferedReader br = null;
  BufferedWriter bw = null;

  try {
    br = new BufferedReader(new FileReader("i.txt"));
    bw = new BufferedWriter(new FileWriter("o.txt"));
    String line;
    while ((line = br.readLine()) != null) {  // Lancia IOException
      bw.write(line);              // Lancia IOException
    }
  } catch (FileNotFoundException fnfe) {
    System.out.println("File not found");
  } finally {
    if (br != null) br.close();          // Lancia IOException
    if (bw != null) bw.close();          // Lancia IOException
  }
}
```

Attenzione: gli stream da Java 7 si chiudono automaticamente con la IOException, quindi nella gestione si può omettere il metodo close(). In qualunque altro caso, vanno manualmente chiusi.

Naturalmente, è possibile istanziare degli oggetti di classe **File**, la quale ha metodi che permettono di gestire di pathnames assoluti e relativi per la creazione di file e cartelle:

```
boolean createNewFile()   // Crea un file vuoto con relativo pathname
boolean delete()      // Elimina un file con relativo pathname
boolean exists()      // Controlla se un file esiste nella directory
boolean renameTo(File dest) // Rinomina un file col pathname di dest
String getParent()      // Ritorna la stringa di pathname del file
boolean mkdir()       // Crea una cartella con un dato pathname
String getAbsolutePath()  // Ritorna il path assoluto
boolean isFile()      // Controlla se l'oggetto e' un file
boolean isDirectory()   // Controlla se l'oggetto e' una cartella
File[] listFiles()      // Ritorna un array di pathnames astratti
 corrispondenti ai file
```

In perfetta sinergia, il pacchetto java.net porta la classe **URL** per la modellazione degli Uniform Resource Locator, che fungeranno da puntatori a risorse online. Questi si compongono di:

- protocollo: https://
- dominio: www.google.com
- port: :443
- path: /search
- parametri q=str&cr=countryIT

Attenzione, il costruttore degli URL è deprecato, quindi nell'istanziazione di oggetti va utilizzato il metodo **toURL()**, come segue:

```
try {
  URL www = new URI("http://info.cern.ch/index.html").toURL();
  BufferedInputStream in = new BufferedInputStream(www.openStream());
  byte[] dataBuffer = new byte [1024];

  while (in.read(dataBuffer, 0, 1024) != -1) {
    System.out.println(new String(dataBuffer));
  }
} catch (IOException | URISyntaxException e) throw new
 RuntimeException(e);
```