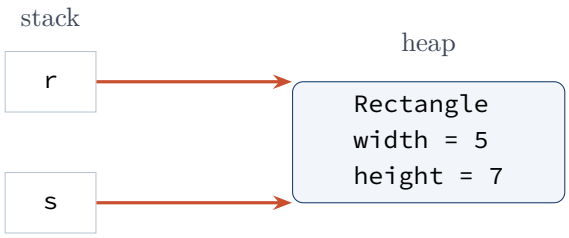
PAJC · Programmazione Avanzata in Java e C · 2026/27

Dispensa 1

Java Fast Forward

Oggetti, riferimenti, ereditarietà, interfacce, enumerazioni, collezioni e risorse



prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

massimiliano.redolfi@unibs.it

Versione 0.1

Indice

Dispensa 1

Java Fast Forward

Dal sorgente all'esecuzione, oggetti e classi, ereditarietà, interfacce, enumerazioni, collezioni, risorse e file.

Questa dispensa raccoglie ciò che di Java diamo per acquisito da un primo corso di programmazione, e lo rilegge con gli occhi di chi deve costruire programmi più grandi: non più esercizi di poche righe, ma applicazioni con molte classi che collaborano. Non è un manuale del linguaggio, e non parte da zero: presuppone di aver già scritto qualche programma Java. Serve a tre cose. Ripassare i concetti su cui tutto il resto si appoggia (oggetti, riferimenti, ereditarietà, interfacce); fissare alcune abitudini che nei programmi piccoli sembrano pignolerie e in quelli grandi fanno la differenza (`equals` e `hashCode`, la chiusura delle risorse, la distinzione fra file di testo e file binari); presentare i costrutti che il linguaggio ha aggiunto negli ultimi anni e che useremo dal primo giorno, come il pattern matching con `instanceof` e le *switch expression*.

Il filo conduttore è un piccolo mondo a due dimensioni, il *bimondo*, popolato da forme geometriche che vanno costruite, confrontate, classificate e contate.

1.1 Dal sorgente all'esecuzione

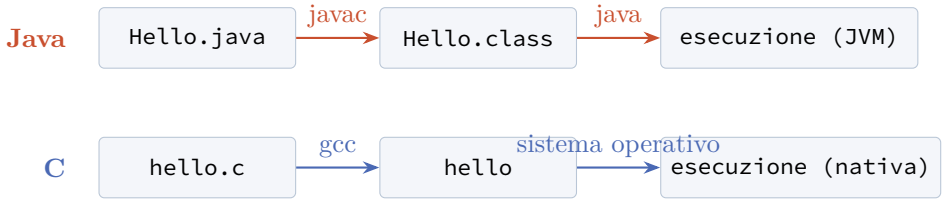
Cominciamo dal programma più piccolo che esista, perché anche in poche righe c'è molto da ripassare.

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

In Java non esistono funzioni libere: ogni pezzo di codice vive dentro una classe. Il punto d'ingresso è il metodo `main`, con una firma fissata: `public` perché deve essere chiamato dall'esterno, `static` perché quando parte il programma nessun oggetto è ancora stato creato, `void` perché non restituisce nulla al sistema, e un parametro `String[] args` che riceve gli argomenti della riga di comando, già separati in un array di stringhe. Il file deve chiamarsi come la classe pubblica che contiene: `Hello.java`.

Il percorso dal sorgente all'esecuzione avviene in due tempi. Il compilatore `javac` traduce il sorgente non in codice macchina, ma in *bytecode*: un linguaggio intermedio, indipendente dal processore, salvato in un file `Hello.class`. Il comando `java` avvia la *Java Virtual Machine* (JVM), che carica il bytecode e lo esegue, traducendolo in codice macchina mentre il programma gira (*compilazione just-in-time*, JIT).

```
$ javac Hello.java        # produce Hello.class
$ java Hello              # avvia la JVM ed esegue main
Ciao
```



Il confronto con il C rende chiaro il senso della scelta. In C il compilatore produce direttamente un eseguibile per una piattaforma precisa, processore e sistema operativo: per un'altra piattaforma si ricompila. In Java si compila una volta sola, e lo stesso `.class` gira ovunque esista una JVM: è la macchina virtuale, non il programma, a essere specifica della piattaforma. Il prezzo è un livello in più tra il programma e il processore; il vantaggio, oltre alla portabilità, è che la JVM può fare cose che un binario nativo non fa, come gestire la memoria automaticamente.

```c
/* hello.c */
#include <stdio.h>

int main(void) {
    printf("Ciao\n");
    return 0;
}
```

In C il `main` è una *funzione*: nessuna classe, nessun oggetto. Torneremo sul C in altre dispense, dove ci servirà proprio per capire che cosa Java fa "sotto".

1.2 Oggetti e classi

Un oggetto tiene insieme due cose: dei *dati* (i campi) e i *comportamenti* che operano su quei dati (i metodi). La classe è lo stampo: descrive quali campi e quali metodi avranno tutti gli oggetti di quel tipo. Ecco un rettangolo del bimondo:

```java
public class Rectangle {
    private int width, height;

    public Rectangle(int w, int h) {
        this.width = w;
        this.height = h;
    }

    public int width()  { return width; }
    public int height() { return height; }

    public int perimeter() {
        return 2 * (width + height);
    }
}
```

Tre cose da notare. I campi sono `private`: dall'esterno si accede ai dati solo tramite i metodi, e la classe resta padrona di come sono rappresentati. Il *costruttore* ha il nome della classe e nessun tipo di ritorno: viene eseguito da `new` e ha il compito di far nascere l'oggetto in uno stato valido. La parola `this` è il riferimento all'oggetto corrente; qui serve a distinguere il campo `width` dal parametro `w`, e in generale a dire "questo oggetto".

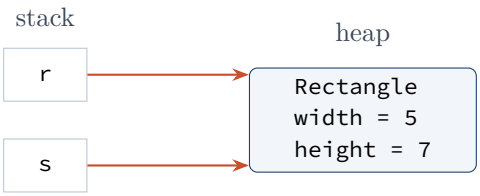
1.2.1 `new`, riferimenti e oggetti

```
Rectangle r = new Rectangle(5, 7);
Rectangle s = r;

System.out.println(r.perimeter());     // 24
```

La prima riga fa due cose: `new` alloca lo spazio per un nuovo oggetto nello *heap*, la zona di memoria gestita dalla JVM, e chiama il costruttore; il risultato è un *riferimento* all'oggetto, che viene copiato nella variabile `r`. La variabile non *contiene* l'oggetto: lo *indica*.

La seconda riga è la conseguenza che più spesso sorprende: `s = r` copia il riferimento, non l'oggetto. Dopo l'assegnamento `r` e `s` indicano lo stesso rettangolo, e una modifica fatta attraverso uno dei due si vede attraverso l'altro. Per avere un secondo rettangolo uguale al primo bisogna crearlo con un altro `new`.



La chiamata `r.perimeter()` va letta come "esegui `perimeter` con `this` uguale a `r`": il metodo è scritto una volta nella classe, e ogni chiamata gli dice su quale oggetto lavorare.

Osservazione 1.1

Le variabili di tipo primitivo (`int`, `double`, `boolean`, ...) contengono direttamente il valore, e l'assegnamento lo copia. Le variabili di tipo classe contengono un riferimento. È la distinzione più importante da tenere a mente quando si passano parametri: un `int` arriva al metodo come copia, un oggetto arriva come riferimento, e il metodo può modificarlo.

1.2.2 Rappresentarsi come testo: `toString`

Stampare un oggetto con `System.out.println(r)` produce una scritta poco utile, il nome della classe seguito da un codice interno:

```
Rectangle@1b6d3586
```

Per ottenere una descrizione leggibile si ridefinisce il metodo `toString`, che ogni classe eredita da `Object` e che `println` (e la concatenazione con `+`) chiamano automaticamente:

```
@Override
public String toString() {
    return "Rectangle " + width + "x" + height;
}
```

L'annotazione `@Override` dichiara che stiamo ridefinendo un metodo ereditato. Non è obbligatoria, ma conviene metterla sempre: se sbagliamo il nome o la firma, il compilatore ce lo dice, invece di lasciarci con un metodo nuovo che nessuno chiama.

1.3 Ereditarietà

Un quadrato è un rettangolo con i lati uguali. In Java questa frase si scrive con `extends`:

```
public class Square extends Rectangle {
    public Square(int side) {
        super(side, side);        // prima si costruisce la parte Rectangle
    }
}
```

```
Square q = new Square(9);
System.out.println(q.perimeter());        // 36: metodo ereditato
```

`Square` *è un* `Rectangle`: ne eredita campi e metodi, e può essere usato ovunque sia atteso un `Rectangle`. Il costruttore di `Square` deve prima far costruire la parte ereditata, con `super(...)`, e questa chiamata deve essere la prima istruzione: un oggetto si costruisce dalla radice della gerarchia verso le foglie. Alla radice di ogni gerarchia c'è `Object`, da cui tutte le classi ereditano, anche quando non lo scrivono: è da lì che arrivano `toString`, `equals` e `hashCode`.

Una sottoclasse può *ridefinire* un metodo ereditato, come abbiamo fatto con `toString`. Quando un metodo ridefinito viene chiamato, la JVM esegue la versione della classe *dell'oggetto*, non quella del tipo con cui l'oggetto è dichiarato: è il *polimorfismo*, e per capirlo bene serve distinguere due tipi che ogni variabile ha.

1.4 Tipo statico e tipo dinamico

```
Object obj = new Square(9);
```

La variabile `obj` ha due tipi. Il *tipo statico* è quello scritto nella dichiarazione, `Object`: è ciò che il compilatore sa, e determina quali metodi si possono chiamare su `obj` (solo quelli di `Object`). Il *tipo dinamico* è quello dell'oggetto realmente indicato, `Square`: è ciò che la JVM sa a runtime, e determina quale versione di un metodo ridefinito viene eseguita.

Per usare `obj` come rettangolo bisogna prima verificare che lo sia, e poi convertire il riferimento. La forma classica separa i due passi:

```
if (obj instanceof Rectangle) {
    Rectangle r = (Rectangle) obj;     // cast: fidati, e' un Rectangle
    System.out.println(r.perimeter());
}
```

Il *cast* `(Rectangle) obj` non trasforma nulla: dice al compilatore di trattare il riferimento come un `Rectangle`. Se l'oggetto non lo fosse, a runtime scatterebbe una `ClassCastException`; per questo il cast va sempre preceduto dal controllo. Test e cast insieme sono così frequenti che il linguaggio li ha fusi in un solo costrutto.

Definizione 1.1: Pattern matching con `instanceof`

JAVA 16 L'espressione `obj instanceof Rectangle r` verifica che `obj` sia un `Rectangle` e, in caso affermativo, dichiara la variabile `r`, già del tipo giusto, che indica lo stesso oggetto. La variabile è utilizzabile solo dove il test è certamente vero.

```
if (obj instanceof Rectangle r) {
    System.out.println(r.perimeter());
}
```

La regola sulla visibilità di `r` è precisa e vale la pena capirla, perché permette scritture compatte:

```
if (obj instanceof Rectangle r && r.width() > 10)   // qui il test e' vero
    System.out.println("largo");

if (!(obj instanceof Rectangle r))
    return;
System.out.println(r.perimeter());     // se non lo fosse, saremmo usciti
```

Osservazione 1.2

Una catena di `instanceof` per distinguere i casi è talvolta necessaria, ma spesso è il segnale che manca un metodo: se ogni forma sa calcolare il proprio perimetro, non serve chiedere a ciascuna che cosa sia. Il polimorfismo fa il lavoro al posto della catena. Il pattern matching resta prezioso quando i tipi arrivano da fuori (un `Object` letto da una collezione eterogenea, un evento, un messaggio).

1.5 `equals` e `hashCode`

Consideriamo una classe `Point` con due campi `x` e `y`, un costruttore e nient'altro, e proviamo a usarla in un insieme:

```
var visited = new HashSet<Point>();
visited.add(new Point(1, 2));
System.out.println(visited.contains(new Point(1, 2)));     // false!
```

Il punto c'è, ma `contains` non lo trova. Il motivo è che `HashSet`, `HashMap` e in generale tutte le collezioni confrontano gli elementi con i metodi `equals` e `hashCode`, e la versione ereditata da `Object` confronta *l'identità*: due oggetti sono uguali solo se sono lo stesso oggetto, come con `==`. I due `new Point(1, 2)` sono oggetti distinti, quindi diversi.

Per confrontare *per contenuto* bisogna ridefinire entrambi i metodi, e ridefinirli in modo coerente:

```java
@Override
public boolean equals(Object o) {
    return o instanceof Point p && x == p.x && y == p.y;
}

@Override
public int hashCode() {
    return Objects.hash(x, y);
}
```

`equals` riceve un `Object`, perché così è dichiarato in `Object`: il pattern matching con `instanceof` risolve in una riga il controllo del tipo e il confronto dei campi (se `o` non è un `Point`, o è `null`, il risultato è `false`). `hashCode` restituisce un intero calcolato dai campi; `Objects.hash` lo fa per noi.

Definizione 1.2: Il contratto di `equals` e `hashCode`

Se due oggetti sono uguali secondo `equals`, devono avere lo stesso `hashCode`. Il viceversa non è richiesto: oggetti diversi possono avere lo stesso hash. Ne segue che ridefinire `equals` senza `hashCode` viola il contratto, e le collezioni basate su hash smettono di funzionare.

Il perché sta in come lavora `HashSet`: usa `hashCode` per decidere *in quale cassetto* cercare, e `equals` per confrontare gli oggetti trovati nel cassetto. Se due punti uguali finiscono in cassetti diversi, `equals` non viene nemmeno chiamato.

Attenzione 1.1

Il `contains` che non trova un elemento presente, la `HashMap` che "perde" le chiavi, i duplicati in un `Set`: sono fra i bug più frequenti nei progetti, e la causa è quasi sempre un `equals` o un `hashCode` mancante. Quando una classe rappresenta un *valore* (un punto, una coppia, una data) i due metodi vanno sempre definiti insieme. Lo stesso vale per `==` applicato agli oggetti: confronta i riferimenti, non il contenuto. Due stringhe con gli stessi caratteri possono essere `==` oppure no; `equals` dà sempre la risposta giusta.

1.6 Interfacce: contratti di comportamento

L'ereditarietà lega classi che sono *la stessa cosa* in versioni più specifiche. Ma spesso vogliamo trattare allo stesso modo oggetti che non hanno nulla in comune, tranne il fatto di saper fare qualcosa. Nel bimondo arrivano le astronavi: non sono forme, non hanno area né perimetro, ma anche loro si disegnano.

```java
public interface Drawable {
    void draw();
}
```

```java
public class Circle    extends Shape2d implements Drawable { ... }
public class SpaceShip implements Drawable { ... }      // nessuna parentela
```

Un'interfaccia è un *contratto*: elenca i metodi che una classe promette di fornire, senza dire come. Una classe può estendere una sola classe, ma implementare quante interfacce vuole; e oggetti di classi senza alcuna parentela possono essere raccolti sotto lo stesso tipo:

```
List<Drawable> scene = List.of(circle, ship);
for (Drawable d : scene)
    d.draw();
```

Il ciclo non sa, e non ha bisogno di sapere, che cosa sia ciascun elemento: sa che sa disegnarsi. L'ereditarietà dice *che cosa è* un oggetto; l'interfaccia dice *che cosa sa fare*. Nella progettazione conviene pensare prima ai contratti e poi alle classi: sono i contratti che permettono di sostituire un'implementazione con un'altra senza toccare chi la usa.

Le interfacce moderne possono contenere anche metodi con un corpo (`default` e `static`) e dichiarare chiuso l'elenco delle proprie implementazioni (`sealed`); e un'interfaccia con un solo metodo si può implementare con una *lambda*. Sono argomenti di altre dispense: qui basta la forma base.

1.7 Enumerazioni e switch expression

Quando un valore può assumere solo alcuni casi fissati, i casi si dichiarano con `enum`:

```
public enum State { IDLE, RUNNING, PAUSED, DONE }
```

Un `enum` è una classe con un numero fisso di istanze, una per costante: si confrontano con `==`, si stampano con il loro nome, hanno un ordine. Il caso d'uso tipico è una *macchina a stati*: lo stato corrente e una tabella di transizione. Per scriverla, lo `switch` nella sua forma moderna è lo strumento giusto.

Definizione 1.3: Switch expression

JAVA 14 Uno `switch` può essere usato come *espressione*: ogni `case` è seguito da `->` e da un valore (o da un blocco che produce un valore con `yield`); lo `switch` nel suo insieme vale il risultato del caso selezionato. Non c'è *fall-through*: un solo caso viene eseguito, senza `break`. Il compilatore richiede che i casi coprano tutti i valori possibili.

```
state = switch (state) {
    case IDLE    -> State.RUNNING;
    case RUNNING -> State.PAUSED;
    case PAUSED  -> State.RUNNING;
    case DONE    -> State.DONE;
};
```

Tre vantaggi rispetto allo `switch` tradizionale. Niente `break`, quindi nessun *fall-through* accidentale, il classico errore in cui un caso "scivola" nel successivo. L'esaustività: se aggiungiamo uno stato all'`enum` e dimentichiamo un caso, il programma non compila, invece di comportarsi in modo strano a runtime. E il fatto di essere un'espressione: il valore va direttamente dove serve, senza una variabile d'appoggio assegnata in ogni ramo.

Quando in un caso servono più istruzioni, si apre un blocco e si produce il valore con `yield`; quando più costanti condividono lo stesso risultato, si elencano separate da virgola:

```
String descrizione = switch (state) {
    case IDLE, DONE -> "fermo";
    case RUNNING    -> "in corso";
    case PAUSED     -> {
        log("pausa");
        yield "sospeso";
    }
};
```

1.7.1 Gli enum sono classi

Le costanti di un `enum` possono avere campi, un costruttore e metodi, e questo li rende molto più di un elenco di nomi. Un operatore aritmetico, per esempio, conosce il proprio simbolo:

```
public enum Operatore {
    SOMMA("+"), SOTTRAZIONE("-"), PRODOTTO("*");

    private final String simbolo;

    Operatore(String simbolo) { this.simbolo = simbolo; }

    public String simbolo() { return simbolo; }
}
```

Il costruttore viene chiamato una volta per costante, con gli argomenti scritti fra parentesi accanto al nome; è implicitamente privato, perché nessuno può creare altre istanze. `Operatore.SOMMA.simbolo()` vale `"+"`, e `Operatore.values()` restituisce l'array di tutte le costanti, nell'ordine di dichiarazione.

1.8 Le collezioni

Gli array hanno dimensione fissa e nessun metodo: per quasi tutto il resto si usano le *collezioni* della libreria standard. Le interfacce da conoscere sono quattro, e per ciascuna basta una o due implementazioni.

Interfaccia	Implementazione	Quando
`List<T>`	`ArrayList`	sequenza ordinata, accesso per indice
`Set<T>`	`HashSet`, `TreeSet`	elementi senza duplicati (ordinati con `TreeSet`)
`Map<K,V>`	`HashMap`, `TreeMap`	ricerca per chiave
`Deque<T>`	`ArrayDeque`	pila (LIFO) e coda (FIFO)

Si dichiara la variabile con il tipo dell'*interfaccia* e si costruisce l'implementazione: così il resto del codice dipende dal contratto, e cambiare implementazione è una riga.

```
List<String> nomi = new ArrayList<>();
nomi.add("Rectangle");
nomi.add("Circle");

Map<String, Integer> conta = new HashMap<>();
conta.put("Rectangle", 3);
System.out.println(conta.get("Circle"));      // null: chiave assente
```

Il tipo fra parentesi angolari, `<String>`, è il *parametro di tipo*: dice al compilatore che cosa la collezione contiene, così `nomi.get(0)` è già una `String` senza cast. Nel `new` il tipo si può omettere, `<>`: il compilatore lo ricava dalla dichiarazione.

```
List<String> fisse = List.of("Rectangle", "Circle");    // immutabile
fisse.add("Diamond");            // UnsupportedOperationException
```

`List.of`, `Set.of` e `Map.of` costruiscono collezioni *immutabili*: comode per le costanti e per i dati di prova, ma ogni tentativo di modifica fallisce a runtime. Se serve una copia modificabile, `new ArrayList<>(fisse)`.

Per scorrere una collezione si usa il `for` esteso, che funziona su qualunque `Iterable`; per una mappa si scorrono le coppie:

```
for (String n : nomi)
    System.out.println(n);

for (Map.Entry<String, Integer> e : conta.entrySet())
    System.out.println(e.getKey() + " -> " + e.getValue());
```

Attenzione 1.2

`Vector`, `Stack` e `Hashtable` esistono ancora nella libreria per compatibilità con il codice degli anni Novanta, e compaiono in molti esempi in rete. Nel codice nuovo si usano `ArrayList`, `ArrayDeque` e `HashMap`. Un altro errore comune: modificare una collezione (`add`, `remove`) mentre la si sta scorrendo con il `for` esteso provoca una `ConcurrentModificationException`.

1.9 Il censimento del bimondo

Mettiamo insieme collezioni e tipi dinamici con un problema concreto: data una lista di forme, contare quante ce ne sono di ciascun tipo.

```
Map<Class<?>, Integer> census = new HashMap<>();

for (Shape2d s : shapes) {
    Integer c = census.get(s.getClass());
    census.put(s.getClass(), c == null ? 1 : c + 1);
}
System.out.println(census);
// {class Rectangle=3, class Circle=5, class Diamond=1}
```

Due ingredienti. `getClass()` restituisce il tipo dinamico dell'oggetto, come un valore che si può usare da chiave: due rettangoli hanno lo stesso `getClass()`. E lo schema *leggi–modifica–scrivi* sulla mappa: si legge il conteggio corrente (`null` se la chiave non c'è ancora), lo si incrementa, lo si riscrive. È corretto ma verboso, e la libreria offre una scorciatoia per il caso "valore di partenza se la chiave manca":

```
census.put(s.getClass(), census.getOrDefault(s.getClass(), 0) + 1);
```

Lo stesso schema conta le parole di un testo, le occorrenze dei caratteri di una stringa, i voti di un'elezione: ogni volta che si contano cose per categoria, c'è una mappa da categoria a intero e un leggi–modifica–scrivi. Con le lambda e gli stream, argomento delle prossime dispense, l'intero censimento diventerà una riga.

1.10 Risorse e chiusura: try-with-resources

Un file aperto, una connessione di rete, un socket: sono *risorse*, e vanno chiuse quando non servono più, anche se nel frattempo qualcosa è andato storto. La gestione manuale è fragile e verbosa:

```
BufferedReader in = null;
try {
    in = new BufferedReader(new FileReader("dati.txt"));
    // ... lettura ...
} finally {
    if (in != null) in.close();
}
```

Il `finally` garantisce la chiusura, ma il controllo su `null`, la variabile dichiarata fuori dal `try` e il fatto che `close` stesso possa lanciare un'eccezione rendono questo codice facile da sbagliare. Il linguaggio ha una forma dedicata.

Definizione 1.4: try-with-resources

JAVA 7 La forma `try (`*dichiarazioni*`) { ... }` apre le risorse dichiarate fra parentesi e garantisce che vengano chiuse, in ordine inverso, all'uscita dal blocco, sia essa normale o per eccezione. Può essere usata con qualunque oggetto che implementi `AutoCloseable`.

```
try (var in = new BufferedReader(new FileReader("dati.txt"))) {
    String line;
    while ((line = in.readLine()) != null)
        System.out.println(line);
}   // chiusura garantita, anche in presenza di eccezioni
```

È la forma da usare sempre: più corta, e corretta in tutti i casi che a mano si dimenticano. Un dettaglio: `FileReader` e `readLine` possono lanciare una `IOException`, che è un'eccezione *controllata*. Il metodo che contiene questo codice deve o gestirla con un `catch`, oppure dichiarare `throws IOException` e lasciarla al chiamante; il compilatore non accetta che venga ignorata.

1.11 File di testo e file binari

Un file è una sequenza di byte. La distinzione fra "file di testo" e "file binario" non sta nel file, ma in come lo leggiamo: se interpretiamo i byte come caratteri secondo una codifica (oggi quasi sempre UTF-8), oppure se li trattiamo per quello che sono. La libreria standard riflette questa distinzione in due famiglie di classi, nel package `java.io`.

	Byte	**Caratteri**
lettura	`InputStream` (`FileInputStream`)	`Reader` (`FileReader`, `BufferedReader`)
scrittura	`OutputStream` (`FileOutputStream`)	`Writer` (`FileWriter`, `PrintWriter`)

Le classi si compongono per *incapsulamento*: un `BufferedReader` non apre un file, ma avvolge un altro `Reader` e gli aggiunge un buffer e il metodo `readLine`. È lo schema `new BufferedReader(new FileReader(path))` già visto: l'oggetto esterno aggiunge capacità a quello interno, e chiudendo l'esterno si chiude anche l'interno.

1.11.1 Testo: righe e caratteri

La lettura riga per riga è quella della §1.10: `readLine` restituisce la riga senza il terminatore, e `null` alla fine del file. Per scrivere, la classe più comoda è `PrintWriter`, che offre gli stessi `print` e `println` di `System.out`:

```
try (var out = new PrintWriter(new FileWriter("report.txt"))) {
    out.println("forme: " + shapes.size());
    for (Shape2d s : shapes)
        out.println(s);
}
```

Il file viene creato se non esiste e *sovrascritto* se esiste; per aggiungere in coda si usa `new FileWriter(path, true)`. Per i file piccoli, quando si vuole tutto il contenuto in memoria, il package `java.nio.file` offre scorciatoie in una riga:

```
List<String> righe = Files.readAllLines(Path.of("dati.txt"));
String tutto = Files.readString(Path.of("dati.txt"));
Files.writeString(Path.of("copia.txt"), tutto);
```

`Path` rappresenta un percorso nel file system, `Files` raccoglie le operazioni su di esso. Le scorciatoie aprono e chiudono il file da sole, quindi non serve il try-with-resources; ma caricano tutto in memoria, e per un file grande la lettura riga per riga resta la scelta giusta.

Attenzione 1.3

Tutte queste classi assumono, se non si dice altro, la codifica predefinita della piattaforma, che dal Java 18 è UTF-8 ovunque. Su versioni precedenti, o quando il file proviene da un altro sistema, conviene dichiararla: `new FileReader(path, StandardCharsets.UTF_8)`. Un file letto con la codifica sbagliata non dà errore: dà caratteri sbagliati, tipicamente sugli accenti.

1.11.2 Binario: byte e tipi primitivi

Un `InputStream` legge byte. Il metodo `read()` restituisce il prossimo byte come intero fra 0 e 255, oppure -1 alla fine del file: è per questo che restituisce un `int` e non un `byte`, che non avrebbe spazio per il valore sentinella. Ecco un programma che stampa un file in esadecimale, sedici byte per riga:

```
try (var in = new BufferedInputStream(new FileInputStream(path))) {
    int b, count = 0;
    while ((b = in.read()) != -1) {
        System.out.printf("%02x ", b);
        if (++count % 16 == 0) System.out.println();
    }
}
```

`BufferedInputStream` ha lo stesso ruolo di `BufferedReader`: leggere un byte alla volta dal disco sarebbe lentissimo, il buffer ne legge migliaia per volta e li serve uno a uno. Applicato a un file `.class`, il programma stampa

```
ca fe ba be 00 00 00 41 ...
```

I primi quattro byte, `CAFEBABE`, sono la "firma" che ogni file di bytecode Java porta in testa; le due coppie successive indicano la versione del formato. È un buon modo di toccare con mano che un `.class` è un file come gli altri, con una struttura che la JVM sa interpretare.

Quando i byte rappresentano numeri, `DataOutputStream` e `DataInputStream` li scrivono e li rileggono nel formato interno di Java:

```
try (var out = new DataOutputStream(new FileOutputStream("punti.bin"))) {
    out.writeInt(2);              // quanti punti seguono
    out.writeDouble(1.5);  out.writeDouble(2.0);
    out.writeDouble(-3.0); out.writeDouble(0.25);
}
```

```
try (var in = new DataInputStream(new FileInputStream("punti.bin"))) {
    int n = in.readInt();
    for (int i = 0; i < n; i++)
        System.out.println(in.readDouble() + ", " + in.readDouble());
}
```

Il file risultante è di $4 + 4 \cdot 8 = 36$ byte, non ha righe, e aperto con un editor di testo mostra solo caratteri senza senso: è binario. In compenso è compatto, e la lettura non richiede analisi del testo. La regola pratica: chi scrive e chi legge devono concordare esattamente la sequenza dei tipi, perché nel file non c'è nulla che la descriva.

Osservazione 1.3

Un `int` occupa 4 byte, un `double` 8, un `char` 2: in Java le dimensioni dei tipi primitivi sono fissate dal linguaggio, non dalla piattaforma, e `DataOutputStream` scrive sempre il byte più significativo per primo (*big-endian*). È una delle differenze con il C, dove le dimensioni e l'ordine dei byte dipendono dalla macchina; ci torneremo quando parleremo di memoria.

1.12 Riepilogo

- Java compila in **bytecode**, eseguito dalla JVM su qualunque piattaforma; il C compila in un binario nativo per una piattaforma.
- Una variabile di tipo classe contiene un **riferimento**; l'assegnamento copia il riferimento, non l'oggetto. `new` alloca nello heap e chiama il costruttore.
- `extends` esprime *è un*; `super(...)` costruisce la parte ereditata; i metodi ridefiniti (con `@Override`) si scelgono in base al **tipo dinamico**.
- `obj instanceof Tipo t` verifica e dichiara in un passo (**pattern matching**).
- Una classe che rappresenta un valore ridefinisce **`equals` e `hashCode` insieme**; altrimenti le collezioni basate su hash non la trovano.
- Un'**interfaccia** è un contratto: dice che cosa un oggetto sa fare, non che cosa è.
- `enum` per gli insiemi chiusi di valori; **switch expression** con `->`, senza fall-through, esaustiva.
- Collezioni: `List`/`ArrayList`, `Set`/`HashSet`, `Map`/`HashMap`, `Deque`/`ArrayDeque`; dichiarare con l'interfaccia; `List.of` è immutabile.
- Le risorse si aprono con **try-with-resources**.
- File: `Reader`/`Writer` per il **testo** (righe con `BufferedReader`, scrittura con `PrintWriter`, scorciatoie in `Files`); `InputStream`/`OutputStream` per i **byte** (`read()` dà -1 alla fine; `Data*Stream` per i tipi primitivi).

1.13 Esercizi

Esercizio 1.1: Area e quadrato

Aggiungere a `Rectangle` un metodo `area()` e verificare che `Square` lo erediti correttamente. Poi aggiungere a `Square` un metodo `side()` e ridefinire `toString` in modo che un quadrato si stampi come `Square 9`.

Esercizio 1.2: Riferimenti

Che cosa stampa il programma seguente, e perché?

```
Rectangle a = new Rectangle(2, 3);
Rectangle b = a;
Rectangle c = new Rectangle(2, 3);
System.out.println(a == b);
System.out.println(a == c);
System.out.println(a.equals(c));
```

Come cambierebbe l'ultima riga se `Rectangle` ridefinisse `equals` e `hashCode`? Scrivere i due metodi.

Esercizio 1.3: Descrivere le forme

Con le classi `Rectangle`, `Circle` (campo `radius`) e `Diamond` (campi `d1`, `d2`) che implementano un'interfaccia `Shape2d`, scrivere un metodo statico `String describe(Shape2d s)` che restituisca una descrizione diversa per ciascun tipo, usando il pattern matching con `instanceof`. Che cosa succede se si aggiunge una classe `Triangle` e ci si dimentica di aggiornare `describe`?

Esercizio 1.4: Semaforo

Dichiarare un `enum Luce { ROSSO, VERDE, GIALLO }` e un metodo `Luce prossima(Luce l)` che realizzi il ciclo rosso → verde → giallo → rosso con una switch expression. Poi dare a ogni costante un campo con la durata in secondi, e un metodo `durata()`.

Esercizio 1.5: Conteggio delle parole

Scrivere un metodo `Map<String, Integer> contaParole(String path)` che legga un file di testo con try-with-resources e restituisca, per ogni parola, il numero di occorrenze. Separare le parole con `line.split(" ")` e ignorare le stringhe vuote.

Esercizio 1.6: Righe numerate

Scrivere un metodo `void print(String path)` che stampi un file di testo facendo precedere ogni riga dal suo numero, allineato a destra su quattro cifre (`String.format("%4d", n)`). Estensione: colorare i numeri con la sequenza ANSI `"\u001B[34m"` (blu) e ripristinare con `"\u001B[0m"`.

Esercizio 1.7: Copia binaria

Scrivere un metodo `void copia(String da, String a)` che copi un file qualunque, byte per byte, e verificarlo su un'immagine: la copia deve aprirsi identica. Perché non si può usare `BufferedReader` e `readLine` per questo compito?

Soluzioni

Soluzione 1.1. In `Rectangle`: `public int area() { return width * height; }`. `Square` lo eredita senza scrivere nulla, perché i suoi campi `width` e `height` sono stati impostati uguali dal costruttore. In `Square`:

```
public int side() { return width(); }

@Override
public String toString() { return "Square " + side(); }
```

Si usa `width()` e non `width` perché il campo è `private` in `Rectangle`, e quindi invisibile anche alle sottoclassi: l'accessor è la via corretta. Se si volesse dare accesso alle sole sottoclassi, il modificatore sarebbe `protected`.

Soluzione 1.2. `true`, `false`, `false`. `a == b` confronta i riferimenti, e i due indicano lo stesso oggetto. `a == c` confronta i riferimenti a due oggetti distinti, anche se con lo stesso contenuto. `a.equals(c)` usa l'`equals` ereditato da `Object`, che confronta l'identità come `==`: ancora `false`. Con

```
@Override
public boolean equals(Object o) {
    return o instanceof Rectangle r && width == r.width && height == r.height;
}

@Override
public int hashCode() { return Objects.hash(width, height); }
```

l'ultima riga stampa `true`. Notare che `r.width` accede a un campo privato di un altro oggetto: è lecito perché siamo dentro la classe `Rectangle`, e `private` limita l'accesso alla classe, non al singolo oggetto.

Soluzione 1.3.

```
static String describe(Shape2d s) {
    if (s instanceof Rectangle r)
        return "rettangolo " + r.width() + "x" + r.height();
    if (s instanceof Circle c)
        return "cerchio di raggio " + c.radius();
    if (s instanceof Diamond d)
        return "rombo " + d.d1() + "x" + d.d2();
    return "forma sconosciuta";
}
```

Con `Triangle` il programma compila e gira, e `describe` risponde "forma sconosciuta": il compilatore non ha modo di sapere che l'elenco dei casi è incompleto, perché la gerarchia è aperta. È il limite di questa tecnica, e il motivo per cui esistono le gerarchie `sealed`, trattate in un'altra dispensa: con un elenco chiuso di sottotipi, uno `switch` sui tipi diventa esaustivo e il caso mancante è un errore di compilazione.

Soluzione 1.4.

```
enum Luce {
    ROSSO(30), VERDE(25), GIALLO(5);

    private final int durata;
    Luce(int durata) { this.durata = durata; }
    public int durata() { return durata; }
}

static Luce prossima(Luce l) {
    return switch (l) {
        case ROSSO  -> Luce.VERDE;
        case VERDE  -> Luce.GIALLO;
        case GIALLO -> Luce.ROSSO;
    };
}
```

Lo `switch` non ha `default`: se un giorno si aggiungesse una costante (`LAMPEGGIANTE`), il compilatore segnalerebbe il caso mancante.

Soluzione 1.5.

```
static Map<String, Integer> contaParole(String path) throws IOException {
    Map<String, Integer> conta = new HashMap<>();
    try (var in = new BufferedReader(new FileReader(path))) {
        String line;
        while ((line = in.readLine()) != null)
            for (String w : line.split(" "))
                if (!w.isEmpty())
                    conta.put(w, conta.getOrDefault(w, 0) + 1);
    }
    return conta;
}
```

Il metodo dichiara `throws IOException` e lascia al chiamante la decisione su che cosa fare se il file non esiste. Il conteggio è il leggi–modifica–scrivi della §1.9, con `getOrDefault` per il caso della prima occorrenza.

Soluzione 1.6.

```
static void print(String path) throws IOException {
    try (var in = new BufferedReader(new FileReader(path))) {
        String line;
        int n = 0;
        while ((line = in.readLine()) != null)
            System.out.println(String.format("%4d", ++n) + "  " + line);
    }
}
```

Per i colori basta racchiudere il numero fra le due sequenze:

```
String num = "\u001B[34m" + String.format("%4d", ++n) + "\u001B[0m";
```

Le sequenze ANSI le interpreta il terminale, non Java: dove non sono supportate compaiono come caratteri spuri.

Soluzione 1.7.

```
static void copia(String da, String a) throws IOException {
    try (var in  = new BufferedInputStream(new FileInputStream(da));
         var out = new BufferedOutputStream(new FileOutputStream(a))) {
        int b;
        while ((b = in.read()) != -1)
            out.write(b);
    }
}
```

Due risorse nello stesso `try`, separate da punto e virgola, chiuse in ordine inverso. `readLine` non va bene: interpreta i byte come caratteri, ne altera alcuni e scarta i terminatori di riga, quindi la copia non sarebbe identica. In una riga: `Files.copy(Path.of(da), Path.of(a))`.