Linguaggio Java

Andrea Domenici
Scuola Superiore S.Anna

26, 28 e 29 settembre 2000

Indice

Capitolo 1

Il linguaggio: sintassi base

1.1 Il classico Hello World

```
/*
 * Un classico Hello World con commenti
 * multilinea
*/
/** commenti di documentazione */
public class HelloWorld {
    public static void    // e commenti monolinea
    main(String args[])
    {
        System.out.println("Hello world!");
    }
}
```

- Un programma è composto di **classi**.
- Ogni programma ha una "*classe principale*" che ha un **metodo** `main()`.

  Il file sorgente contenente la classe principale ha il nome di tale classe con l'estensione `.java`.

- Per compilare il programma:
  `javac HelloWorld.java`
  Il compilatore produce un file di *bytecode* (linguaggio macchina della *macchina virtuale Java*, JVM) di nome `HelloWorld.class`.
- Per eseguire il programma:
  `java HelloWorld`
- Per compilare ed eseguire i programmi Java, bisogna che il compilatore e l'interprete di bytecode trovino i file necessari. Questo richiede che si fornisca la posizione dei file nella variabile d'ambiente `CLASSPATH` o con un'opzione sulla linea di comando.
- Una classe è composta da **dichiarazioni**.
- La classe `HelloWorld` comprende soltanto la dichiarazione dell'identificatore `main`. Questo identificatore è il nome di un metodo, e la sua dichiarazione definisce le operazioni svolte dal metodo.
- Il metodo `main()` della classe principale viene invocato dall'interprete Java. Se la linea di comando con cui si invoca l'interprete ha degli altri argomenti oltre al nome della classe, questi vengono messi nell'array di stringhe `args`.
- Il metodo `main()` della classe `HelloWorld` si limita a invocare un metodo di un'altra classe.

```
public class Buongiorno {
    public static void main(String[] args)
    {
        System.out.println("Buongiorno, " + args[0]);
    }
}

% java Buongiorno Silvana
Buongiorno, Silvana
```

1.2 Tipi primitivi ed espressioni

```
public class TipiPrimitivi {
    public static void main(String[] args)
    {
        boolean b = false;
        char c = '\u2297';
        int i = 666;
        double d = 3.14;
        System.out.println(b);
        System.out.println(c);
        System.out.println("Numero diabolico: " + i);
        System.out.println(d);
    }
}
```

- Come i linguaggi tradizionali, il Java offre un insieme di **tipi primitivi**.
- Il tipo `char`, a differenza di quanto avviene in C, occupa due byte, in modo da rappresentare l'insieme di caratteri **Unicode**.
- L'operatore `+`, quando il suo primo operando è una stringa, produce una rappresentazione testuale del secondo operando e la concatena al primo.

La seguente tabella riassume le caratteristiche dell'implementazione dei tipi primitivi.

booleani	`boolean`		
interi	`char`	2 byte	
	`byte`	1 byte	-128 – 127
	`short`	2 byte	-32768 – 32767
	`int`	4 byte	-2147483648 – 2147483647
	`long`	4 byte	-9223372036854775808L – 9223372036854775808L
floating	`float`	4 byte	7 cifre sig.
	`double`	8 byte	15 cifre sig.

Le due tabelle successive riassumono gli operatori che si possono applicare ai tipi primitivi.

Operatori sui booleani

`== !=`	uguaglianza
`!`	negazione
`& ^ |`	AND, XOR, OR
`&& ||`	AND e OR condizionali
`? :`	operatore condizionale
`+`	concatenazione

Operatori sugli interi

`== !=`	uguaglianza
`< <= > >=`	confronto
`+ -`	*piú* e *meno* unari
`* / %`	moltiplicazione, divisione e resto
`+ -`	somma e sottrazione
`++ --`	incremento e decremento
`<< >> >>>`	operatori di shift
`~ & ^ |`	complemento, AND, XOR, OR
`? :`	operatore condizionale
`(` *type* `)`	conversione di tipo
`+`	concatenazione

- l'operatore `>>` estende il bit di segno.
- l'operatore `>>>` inserisce zeri.
- I tipi floating hanno gli stessi operatori degli interi, esclusi quelli bit a bit. I tipi floating seguono lo standard IEEE 754.

```
public class Expressions1 {
    public static void main(String[] args)
    {
        int i = 0;
        System.out.println("vecchio i: " + i++);
        System.out.println("nuovo i: " + i);
        int j = 2;
        System.out.println("nuovo j: " + ++j);
        System.out.println("condizionale: " + (j > i ? j : i));
    }
}
```

- Un'**espressione** è un insieme di identificatori, costanti ed operatori la cui valutazione produce un valore o una variabile. Per ora consideriamo solo espressioni che producono valori.
- Un' espressione può produrre **effetti collaterali**.
- Le espressioni di **assegnamento** cambiano il valore della variabile denotata dal primo operando dell'operatore di assegnamento.
- L'espressione `x += 3` equivale a `x = x + 3`. Analogamente per gli operatori `-=`, `*=`, etc.

1.3 Istruzioni

```
public class Statements1 {
    public static void main(String[] args)
    {
        int i = 0;
        if (i != 0)           // espressione booleana!
            System.out.println("i diverso da zero");
        else
            System.out.println("i uguale a zero");

        i = Integer.parseInt(args[0]);
        switch(i) {
        case 1:
            System.out.println("uno");
            break;
        case 2:
            System.out.println("due");
            break;
        case 3:
            System.out.println("tre");
            break;
        default:
            System.out.println("sbagliato");
            break;
        }

        int factorial[] = {     // array
            1,
            1,
            2,
            6,
            24,
```

```
        };

        i = 0;
        while (i < factorial.length) {
            System.out.println("" + i + "\t" + factorial[i]);
            i++;
        }
        i = 0;
        do {
            System.out.println("" + i + "\t" + factorial[i]);
            i++;
        } while (i < factorial.length);

        for (int j = 0; j < factorial.length; j++)
            System.out.println("" + j + "\t" + factorial[j]);
    }
}
```

- L'attributo `length` è definito per tutti gli array.
- La stringa vuota ‘‘ serve a far riconoscere la concatenazione.
- La variabile `j` è visibile solo nell'intestazione e nel corpo del `for`.

```
public class Statements2 {
    public static void main(String[] args)
    {
        int i = 0;
        for ( ; i < 10; i++)
            if (i == 3)
                break;
        System.out.println("" + i);
```

```
        for ( ; i < 4; i++) {
            System.out.println("" + i);
            if (i % 2 != 0)
                continue;
            System.out.println("pari");
        }

        i = 0;
        block1: {
            System.out.println("block1");
            for ( ; i < 4; i++)
                for (int j = 0; j < 4; j++) {
                    if (j == 3)
                        break block1;
                }
        }
        System.out.println("i == " + i);

        i = 0;
        block2:
            for ( ; i < 4; i++) {
                System.out.println("block2");
                for (int j = 0; j < 4; j++) {
                    if (j == 3)
                        continue block2;
                }
            }
        System.out.println("i == " + i);
    }
}
```

- `break` senza etichetta fa uscire dal ciclo piú interno.

- `continue` senza etichetta termina l'iterazione corrente del ciclo piú interno.
- `break` con etichetta fa uscire dal *blocco* etichettato.
- `continue` con etichetta termina l'iterazione corrente del *ciclo* etichettato.

Capitolo 2

Classi e oggetti

2.1 Nozioni elementari

- Un **oggetto** è un modulo che offre dei servizi ad altri moduli (*clienti*) e richiede altri servizi da altri moduli ancora (*fornitori*).
- Una **classe** definisce un insieme di oggetti aventi lo stesso comportamento. Ogni oggetto è un'**istanza** della propria classe.
- L'**interfaccia** di una classe (ovvero degli oggetti da essa definiti) è costituita, propriamente, dai servizi offerti e richiesti, ma di solito questo termine viene usato solo in riferimento ai servizi offerti.
- Ciascun servizio può essere svolto in modi differenti, purché portino allo stesso risultato. Il modo in cui vengono svolti i servizi definiti da un'interfaccia è l'**implementazione** di tale interfaccia. È ben nota la necessità di *nascondere* l'implementazione agli utenti dell'interfaccia.
- Una classe Java è formata dall'elenco delle strutture dati (**campi**) e delle operazioni (**metodi**) che sono **membri** dell'interfaccia o dell'implementazione della classe.

- L'**accessibilità** di un membro determina la sua appartenenza all'interfaccia o all'implementazione: i membri **pubblici** appartengono all'interfaccia, quelli **privati** appartengono all'implementazione. (Esistono altri tipi di accessibilità, che vedremo in seguito).

```
class Complex {
    private double re;
    private double im;

    public Complex() { re = 0.0; im = 0.0; }
    public Complex(double r, double i) { re = r; im = i; }
    public double real() { return re; }
    public double imaginary() { return im; }
    public static Complex sum(Complex x, Complex y)
    {
        Complex z = new Complex();
        z.re = x.re + y.re;
        z.im = x.im + y.im;
        return z;
    }
}

public class ComplexT {
    public static void main(String[] args)
    {
        Complex xx = new Complex(1.2, 2.1);
        Complex yy = new Complex(3.2, 4.1);
        Complex zz;
        zz = Complex.sum(xx, yy);
        System.out.println("" + zz.real() + ", " + zz.imaginary());
    }
}
```

- I metodi di una classe possono accedere ai membri privati della stessa classe.
- Un metodo (non statico, v. oltre) viene **invocato** (o **chiamato**) scrivendo il nome di una variabile seguito da un punto e dal nome del metodo, con i parametri attuali fra parentesi. All'interno del metodo, ogni occorrenza del nome di un membro della classe si riferisce al membro corrispondente dell'oggetto per cui è stato invocato il metodo. L'oggetto può essere riferito esplicitamente per mezzo della variabile `this`.
- Un metodo **statico** non viene applicato ad un oggetto, ma può accedere ai membri privati di oggetti della stessa classe, passati come parametri. Viene invocato scrivendo il nome della classe, un punto, il nome del metodo ed i parametri.
- Un metodo generalmente restituisce un valore, usando l'istruzione `return`. I metodi che non restituiscono un valore (metodi `void`) possono usare la `return` non seguita da un'espressione.
- Un **costruttore** è un metodo che viene invocato automaticamente quando è necessario creare un oggetto, ed ha il compito di inizializzarlo.
- I due costruttori di `Complex` mostrano un esempio di **overloading**. Si ha overloading quando due o piú metodi di una stessa classe hanno lo stesso nome ed argomenti di tipo diverso. La differenza nei tipi degli argomenti permette al compilatore di scegliere quale metodo invocare per ciascuna chiamata.

2.2 Gestione della memoria

- Gli oggetti risiedono in una zona di memoria chiamata **memoria libera** o **heap**.
- L'accesso agli oggetti avviene attraverso variabili i cui valori sono **riferimenti** (indirizzi) a oggetti. Nel metodo `ComplexT.main()`, le variabili `xx`, `yy` e `zz` sono riferimenti ad oggetti `Complex`.

```
class Alias {
    public int i;
    public Alias(int n) { i = n; }
    public boolean equals(Alias x) { return i == x.i; }
}

public class AliasT {
    public static void main(String[] args)
    {
        Alias a = new Alias(1);
        Alias b = new Alias(1);
        Alias c = a;
        if (a != b)
            System.out.println("a != b");
        if (a.equals(b))
            System.out.println("a.equals(b)");
        if (a == c)
            System.out.println("a == c");
    }
}

a != b
a.equals(b)
a == c
```

- Gli operatori di uguaglianza (‘`==`’ e ‘`!=`’), se applicate a due variabili di tipo riferimento, controllano se le due variabili si riferiscono allo stesso oggetto.
- Per verificare l’uguaglianza membro a menbro di due *oggetti* distinti, il programmatore deve definire il metodo `equals()`.
- Le variabili di tipo primitivo e di tipo riferimento, se dichiarate all’interno di un blocco (variabili **automatiche**) vengono create quando il flusso di controllo incontra la loro dichiarazione, e vengono distrutte all’uscita dal blocco.
- Gli oggetti vengono creati quando viene invocato l’operatore `new` e vengono mantenuti in memoria finché esistono riferimenti ad essi. Quando un oggetto non viene piú riferito può essere distrutto. La distruzione automatica degli oggetti non piú accessibili, col conseguente recupero della memoria occupata, si chiama **garbage collection**.
- I campi (non statici) di una classe, sia di tipo primitivo che di tipo riferimento, vengono creati entro ogni oggetto e vengono mantenuti in memoria finché l’oggetto non viene distrutto.
- In Java non esiste un’operazione di deallocazione esplicita.
- È possibile definire, in qualsiasi classe, un metodo chiamato `finalize()` che viene invocato per un oggetto sottoposto a garbage collection. Bisogna tener presente che gli oggetti inaccessibili non vengono necessariamente distrutti, quindi se è necessario compiere delle operazioni di “pulizia” al termine della vita di un oggetto (per esempio, chiudere dei file o distruggere finestre in un’interfaccia grafica), occorre farlo esplicitamente.

Capitolo 3

Eredità

```
class Employee {                        // classe base
    private String name;
    public Employee(String s) { name = new String(s); }
    public void print()
    {
        System.out.println("EMPLOYEE: ");
        System.out.println("    name: " + name);
    }
}

class Manager extends Employee {    // classe derivata
    private String department;
    public Manager(String n, String d)
    {
        super(n);                       // costruttore base;
        department = new String(d);
    }
    public int forecast()
    {
        return (int)(Math.random() * 100);
    }
```

```
    public void print()             // overriding
    {
        System.out.println("MANAGER: ");
        super.print();              // metodo base
        System.out.println("    department: " + department);
    }
}

public class EmployeeT {
    public static void main(String[] args)
    {
        Employee e = new Employee("Dilbert");
        Manager m = new Manager("Pointed Hair",
                                "Infrastructural Synergy");
        e.print();
        m.print();
        System.out.println("profits: "
                                    + m.forecast() + "%");
        e = m;                       // upcasting
        e.print();
//      errore!
//      System.out.println("profits: "
//                                    + e.forecast() + "%");
    }
}

EMPLOYEE:
    name: Dilbert
MANAGER:
EMPLOYEE:
    name: Pointed Hair
    department: Infrastructural Synergy
profits: 21%
```

```
MANAGER:
EMPLOYEE:
    name: Pointed Hair
    department: Infrastructural Synergy
```

- La classe derivata **eredita** campi e metodi (ma non il costruttore) della classe base.
- L'identificatore `super` permette alla classe derivata di riferirsi all'oggetto della classe base contenuto (implicitamente) nella classe derivata. In particolare, l'espressione `super(...)` invoca il costruttore della classe base.
- La classe derivata può aggiungere nuovi membri a quelli della classe base.
- La classe derivata può **ridefinire** (**override**) uno o piú metodi accessibili della classe base.
- La classe derivata può **ridefinire** (o **nascondere**) uno o piú campi accessibili della classe base (non appare nell'esempio).
- Se un riferimento `d` ad un oggetto di classe derivata `D` viene assegnato ad un riferimento `b` alla classe base `B` (*upcasting*), si ha che: i) applicando a `b` un metodo `m()` ridefinito in `D`, viene eseguita la versione ridefinita di `m()` (cioè, `b.m()` equivale a `d.m()`), e ii) non si può applicare a `b` un metodo definito originariamente in `D`.
- I membri privati della classe base sono inaccessibili alla classe derivata.
- Oltre che privato o pubblico, un membro può essere **protetto**: i membri protetti di una classe sono accessibili solo ai metodi delle classi derivate da tale classe.

- Una classe derivata ha una sola classe base.
- Una classe dichiarata `final` non può essere usata come classe base.
- Una variabile dichiarata `final` non può essere modificata.

3.1 Polimorfismo

```
class Shape {
    public void draw() { System.out.println("drawing Shape"); }
}

class Circle extends Shape {
    public void draw() { System.out.println("drawing Circle"); }
}

class Rectangle extends Shape {
    public void draw() { System.out.println("drawing Rectangle"); }
}

class ScreenMgr {
    public Shape screen[] = new Shape[10];
    public void drawall()
    {
        for (int i = 0; i < screen.length; i++)
            if (screen[i] != null)
                screen[i].draw();    // invocazione polimorfica
    }
}

public class ScreenMgrT {
    public static void main(String args[])
```

```
    {
        ScreenMgr sm = new ScreenMgr();  // costr. predef.
        sm.screen[0] = new Circle();
        sm.screen[1] = new Rectangle();
        sm.screen[2] = new Circle();
        sm.drawall();
    }
}
```

- Un riferimento ad un oggetto della classe derivata può sempre essere usato al posto di un riferimento alla classe base. **A tempo di esecuzione** (**binding dinamico**) viene scelto il metodo appropriato.
- Il **polimorfismo** è la possibilità di usare uno stesso riferimento o una stessa invocazione di metodo per oggetti di classi differenti.
- In Java il polimorfismo viene ottenuto per mezzo dell'eredità e del binding dinamico (anche l'overloading è una forma di polimorfismo, ma meno potente).
- Il vantaggio principale del polimorfismo è la possibilità di scrivere metodi generici, come `drawall()` nell'esempio precedente. Il polimorfismo facilita il **riuso** del codice.
- Se deriviamo una nuova classe da `Shape`, p.es. `Triangle`, il codice di `ScreenMgr` rimane immutato e funzionante.
- Nell'esempio precedente, la classe base `Shape` definisce un metodo `draw()` che viene invocato per gli oggetti `Shape` e per oggetti di eventuali classi derivate da `Shape` che non ridefiniscono il metodo.
- Se si prevede che non vengano istanziati oggetti `Shape` e che tutte le classi derivate da `Shape` (che permettano di istanziare

oggetti) ridefiniscano `draw()`, allora possiamo omettere la definizione di `Shape.draw()`, dichiarando `abstract` sia il metodo che la classe. Una classe che contiene almeno un metodo astratto si dice **astratta**.

```
abstract class Shape1 {
    public abstract void draw();    // niente corpo
}

class Circle extends Shape1 { /* ... */ }

class Rectangle extends Shape1 { /* ... */ }

class ScreenMgr1 { /* ... */ }

public class ScreenMgr1T { /* ... */ }
```

- Una classe astratta non può essere istanziata (però si può istanziare un array di riferimenti alla classe astratta).
- Se una classe derivata da una classe astratta non implementa tutti i metodi astratti della classe base, allora anche la classe derivata è astratta e deve essere dichiarata come tale.

3.2 Interfacce

- Un'**interfaccia** è una classe dichiarata con la parola `interface` al posto di `class`, i cui metodi sono tutti astratti e pubblici.
- Da un'interfaccia se ne possono derivare altre, con `extends`. Per le interfacce è possibile l'eredità multipla.

- Da un'interfaccia si possono derivare delle classi, usando la parola `implements` al posto di `extends`. Una classe può implementare piú di un'interfaccia. Inoltre, una classe può estendere una classe base ed implementare una o piú interfacce.

```
interface IntrfA { void methodA(); }

interface IntrfB { void methodB(); }

interface IntrfC extends IntrfA, IntrfB { void methodC(); }

class ClassX implements IntrfA {
    public methodA() { /* implementazione */ }  // pubblico!
}

class ClassY extends ClassX implements IntrfB {
    public methodB() { /* implementazione */ }  // pubblico!
}
```

- Le interfacce (e l'eredità multipla) permettono di trattare separatamente aspetti diversi di un'unica entità, diminuendo le dipendenze reciproche fra diverse parti di un sistema.

```
abstract class Aircraft {
    public double speed;
    public abstract void fly();
}

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

```
class JetLiner extends Aircraft implements Drawable {
    public JetLiner() { speed = 850.0; }
    public void fly() { /* flight simulation */ }
    public void draw() { /* graphic rendering */ }
}

class DisplayMgr {
    private Drawable d;
    public DisplayMgr(Drawable dd) { d = dd; }
    public void display() { d.draw(); }
}

class AirTrafficCtrl {
    private Aircraft c;
    public AirTrafficCtrl(Aircraft ac) { c = ac; }
    public void simulate() { c.fly(); }
}

public class VideoGame {
    public static void main(String[] args)
    {
        JetLiner tu204 = new JetLiner();
        DisplayMgr dm = new DisplayMgr(tu204);
        AirTrafficCtrl atc = new AirTrafficCtrl(tu204);
        dm.display();
        atc.simulate();
    }
}
```

- Nell'esempio precedente, la classe `JetLiner` ha due funzioni distinte: simulare un aereo e fornirne una rappresentazione grafica. La classe astratta `Aircraft` definisce le informazioni e i me-

todi necessari per la simulazione, mentre l'interfaccia `Drawable` definisce i metodi necessari per la rappresentazione grafica.

- La classe `DisplayMgr` è responsabile della parte grafica del videogioco, e può gestire oggetti di qualsiasi tipo, purché implementino l'interfaccia `Drawable`.
- La classe `AirTrafficCtrl` è responsabile della parte del videogioco relativa alla simulazione, e può gestire oggetti di qualsiasi tipo, purché derivati da `Aircraft`.

Capitolo 4

Eccezioni

```
class NewException extends Exception {
    public int excType;         // info supplementare
    public NewException(int t) { excType = t; }
}

public class Exceptional {
    public void raiseExc()
    {
        // eccezioni ignorate
        // ...
        try { // eccezioni catturate
            // ...
            if (/* problema n. 17 */)
                throw new NewException(17);
            // ...
        } catch (NewException ne) {
            // handler per NewException
        } catch (ArithmeticException ae) {
            // handler per ArithmeticException
        } catch (Exception e) {
            // handler per Exception
```

```
        } finally {
            // eseguito in ogni caso
        }
    }
}
```

- L'esecuzione di un'istruzione può causare situazioni erronee o comunque considerate eccezionali, dette **eccezioni**.
- Il supporto run-time del Java rileva una serie di eccezioni o errori predefiniti.
- Il Java permette di
  - individuare le sezioni di codice che possono causare (**sollevare** o **lanciare**) eccezioni;
  - definire le azioni da prendere in seguito ad un'eccezione;
  - definire nuove eccezioni e sollevarle.
- Quando viene lanciata (implicitamente o esplicitamente) un'eccezione:
  - viene costruito l'oggetto corrispondente;
  - se nel contesto locale c'è uno handler per l'eccezione, viene eseguito passandogli l'oggetto lanciato;
  - altrimenti, si cerca uno handler nei contesti esterni; se non si trova uno handler, il programma termina con dei messaggi diagnostici.

  La clausola `finally` (se presente) viene eseguita comunque.
- Entro un contesto, gli handler vengono considerati in ordine testuale, e viene scelto il primo il cui argomento sia della stessa classe o di una classe base dell'eccezione lanciata.

- Uno handler può gestire l'eccezione e riprendere l'esecuzione (ma non al punto dove è avvenuta l'eccezione), oppure rilanciare l'eccezione, oppure lanciare un'eccezione di tipo diverso.

- I metodi che possono causare eccezioni che vengono propagate all'esterno devono dichiararlo nella loro intestazione. Non è necessario dichiarare le eccezioni delle classi `Error` e `RuntimeException` (e derivate).

- Quando un metodo viene ridefinito in una classe derivata, il nuovo metodo può lanciare solo le eccezioni dichiarate nel metodo della classe base.

- Tutte le eccezioni discendono da `Throwable`, da cui derivano direttamente `Error` ed `Exception`. `Error` rappresenta eccezioni che si possono verificare ovunque e sono irrimediabili. Da `Exception` derivano `RuntimeException` e tutte le altre classi definibili dal programmatore.

- `RuntimeException` comprende gli errori di programmazione, fra cui:
  - `ArithmeticException`
  - `ArrayStoreException`
  - `ClassCastException`
  - `IllegalArgumentException`
  - `IndexOutOfBoundsException`
  - `NullPointerException`

- `Throwable` e le classi derivate definiscono dei metodi per stampare informazioni sull'eccezione, a supporto del debugging.

```
public void raiseExc() throws NewException { /* ... */ }
```

- La clausola `throws` specifica quali eccezioni possono essere sollevate durante l'esecuzione di un metodo, e permette al compilatore di verificare che, nei contesti in cui viene invocato tale metodo, l'eccezione venga gestita.

Capitolo 5

Ingresso/uscita

- Per l'I/O, bisogna creare istanze di classi che modellano (1) il tipo di sorgente o destinazione del trasferimento e (2) il modo in cui esso avviene. Le classi del tipo (1) derivano da `InputStream` o da `OutputStream`.

Classi per l'input

sorg./dest.	classe	richiede
memoria	`ByteArrayInputStream`	array di byte
stringa	`StringBufferInputStream`	stringa
file	`FileInputStream`	file
pipe	`PipedInputStream`	`PipedOutputStream`
modo	**classe**	**richiede**
tipi primitivi	`DataInputStream`	`InputStream`
lettura bufferizzata	`BufferedInputStream`	`InputStream`
numerazione righe	`LineNumberInputStream`	`InputStream`
pushback	`PushbackInputStream`	`InputStream`

Classi per l'output

sorg./dest.	classe	richiede
memoria	`ByteArrayOutputStream`	dimens. buffer
file	`FileOutputStream`	file
pipe	`PipedOutputStream`	`PipedInputStream`
modo	classe	richiede
tipi primitivi	`DataOutputStream`	`OutputStream`
output formattato	`PrintStream`	`OutputStream`
scrittura bufferizzata	`BufferedOutputStream`	`OutputStream`

- per usare, p.es., un file, bisogna:
  1. istanziare una classe del tipo (1) (p.es. `FileInputStream`, fornendo al costruttore il nome del file);
  2. passare l'oggetto cosí creato al costruttore di una classe del tipo (2) (p.es. `DataInputStream`);
  3. usare quest'ultimo oggetto con i metodi forniti dalla classe.

```
import java.io.*;
public class IO1 {
    public static void main(String[] args)
    {
        try {
        DataInputStream in =
            new DataInputStream(new FileInputStream("test"));
        String s;
        while ((s = in.readLine()) != null)
            System.out.println(s);
        in.close();
        } catch (EOFException e) {
            System.out.println("EOF!");
        } catch (Exception e) {
            // imbroglio...
        }
```

```
    }
}
```

Alcuni metodi di `DataInputStream`

```
boolean readBoolean()
byte readByte()
short readShort()
char readChar()
int readInt()
double readDouble()
String readLine()
```

- I metodi di `DataInputStream` leggono file binari.
- Esistono metodi simili (`void writeBoolean()`, `void writeInt()`...) per la classe `DataOutputStream`, che scrivono su file binari (non formattati).
- La classe `RandomAccessFile` permette lettura e scrittura con accesso casuale.
- La classe `File` permette di manipolare nomi di file e di compiere ricerche sui directory.

```
import java.io.*;
public class IO2 {
    public static void main(String[] args)
    {
        DataInputStream in =
            new DataInputStream(
                new BufferedInputStream(System.in));
        try {
            String s;
            while ((s = in.readLine()) != null)
```

```
                System.out.println(s);
            in.close();
        } catch (EOFException e) {
            System.out.println("EOF!");
        } catch (Exception e) {
        }
    }
}
```

- Le classi `StreamTokenizer` e `StringTokenizer` forniscono dei metodi utili per l'analisi lessicale (*scanning*) di sequenze di caratteri.

```
import java.io.*;
import java.util.*;
public class IO3 {
    public static void main(String[] args)
    {
        DataInputStream in =
            new DataInputStream(new BufferedInputStream(System.in));
        try {
            // lettura di una riga e creazione di un tokenizer
            String line = in.readLine();
            in.close();
            StringTokenizer t = new StringTokenizer(line);
            // estrazione e conversione di un token
            String token = t.nextToken();
            int i = java.lang.Integer.parseInt(token);
            // estrazione e conversione di un token
            token = t.nextToken();
            double d = java.lang.Double.valueOf(token).doubleValue();
```

```
            System.out.println(i);
            System.out.println(d);
        } catch (EOFException e) {
            System.out.println("EOF!");
        } catch (Exception e) {
        }
    }
}

123 3.14
123
3.14
```

- La versione 1.1 del linguaggio offre nuove classi, che permettono di usare caratteri Unicode, e vengono usate in modo simile a quelle della version 1.0. Alcune delle nuove classi sono `FileReader`, `FileWriter`, `BufferedReader`, `BufferedWriter`, `PrintWriter`.
- Altre nuove classi, come `ZipOutputStream`, `ZipInputStream`, `GZIPOutputStream`, e `GZIPInputStream`, permettono di leggere e scrivere file compressi.

Capitolo 6

Librerie standard

- Le librerie sono organizzate in **package**.
- I directory contenenti le librerie (sia quelle standard che quelle scritte dall'utente) sono elencati nella variabile di ambiente `CLASSPATH`. Per esempio, in Unix si può settare la variabile con l'istruzione:
  `setenv CLASSPATH /usr/local/lib/jdk1.1.3/lib:.`
- Il package `java.lang` contiene:
  - `Object`, da cui derivano implicitamente *tutte* le altre classi;
  - le classi **wrapper** per i tipi primitivi (`Integer`, `Boolean`...);
  - l'interfaccia `Runnable` e le classi `Thread` e `ThreadGroup`;
  - la classe `Math`;
  - le classi `String` e `StringBuffer`;
  - la classe `Throwable` e derivate;
  - altre classi.
- Il package `java.util` contiene:
  - `Random`, `Bitset`, `Date`;
  - le classi **contenitore** come `Hashtable`, `Vector`, `Stack`, `Dictionary`, `Properties`;
  - l'interfaccia `Enumeration`;

  - altre classi.

- Il package `java.io` contiene: ... indovinate?
- I package `javax` contengono le librerie Swing per la realizzazione di interfacce utente grafiche.

6.0.1 Stringhe

- La classe `String` serve ad usare sequenze di caratteri. Una stringa non può essere modificata dopo l'inizializzazione.
- Alcuni metodi:
  - `String()` stringa vuota;
  - `String(String s)` copia di `s`;
  - `String(char[] d)` copia di un array di caratteri;
  - `boolean equals(Object o)` uguaglianza "vera";
  - `int hashCode()`;
  - `int length()`;
  - `char charAt(int index)`;
  - `void getChars(int srcBeg, int srcEnd, char dst[], int dstBeg)`;
  - `void getBytes(int srcBeg, int srcEnd, byte dst[], int dstBeg)`;
  - `int indexOf(int ch)`;
  - `int indexOf(String s)`;
  - `int compareTo(String s)`;
  - `String substring(int begIdx, int endIdx)`;
  - `String concat(String s)`;
  - `String toLowerCase()`;
- La classe `StringBuffer` serve ad usare sequenze modificabili di caratteri. La classe ha dei metodi `append()` e `insert()` per modificare il buffer.

Capitolo 7

Run-Time Type Identification

- Ad ogni classe è associato un oggetto di classe `Class`, che contiene tutte le informazioni sulla classe, che quindi sono accessibili a tempo di esecuzione.
- Il metodo (statico) `Class.forName(String s)` restituisce un riferimento all'oggetto `Class` della classe il cui nome è uguale alla stringa `s`. Tale riferimento si può ottenere anche scrivendo l'identificatore della classe seguito da un punto e dalla parola `class`, p.es. `int.class`. Per le classi wrapper si può usare il campo statico `TYPE`, come in `Int.TYPE`.
- Il metodo `getClass()` della classe `Object` restituisce un riferimento all'oggetto `Class` della classe a cui appartiene l'oggetto per cui è stato invocato.
- L'operatore booleano `instanceof` serve a verificare se un oggetto è istanza di una classe: `tu204 instanceof JetLiner`. Viene valutato a tempo di compilazione.
- Il metodo booleano `isInstance(Object o)` serve a verificare se un oggetto è istanza di una classe rappresentata da un oggetto `Class`:

`JetLiner.class.isInstance(tu204)`. Viene valutato a tempo di esecuzione.

- Il metodo `getInterfaces()` restituisce un array contenente gli oggetti `Class` delle interfacce implementate dalla classe. Il metodo `getSuperclass()` restituisce l'oggetto `Class` della superclasse.

- Il metodo `newInstance()` costruisce una nuova istanza della classe e ne restituisce un riferimento.

7.0.2 Riflessione

- Il meccanismo di RTTI è "run-time" in quanto gli oggetti `Class` vengono usati a tempo di esecuzione, però le informazioni relative alle classi devono essere disponibili a tempo di compilazione.

- Le tecniche piú moderne di programmazione in ambienti distribuiti (per esempio secondo il modello CORBA) richiedono la disponibilità delle informazioni di tipo a tempo di esecuzione. Per esempio, un'applicazione può chiedere a un server remoto se esistono delle classi con determinte caratteristiche, sceglierne una ed istanziarla per collegare l'oggetto risultante al resto dell'applicazione.

- Per questo tipo di problemi è stato introdotto il meccanismo della **riflessione**. Il supporto a questo meccanismo consiste in alcune classi (appartenenti al package `java.lang.reflect`) che descrivono (*riflettono*) le proprietà cercate, e in metodi della classe `Class` che estraggono tali proprietà dalle classi, a tempo di esecuzione.

- Alcune delle nuove classi sono `Constructor`, `Field` e `Method`. Il metodo `invoke(Object obj, Object[] args)` di `Method` permette l'invocazione dinamica del metodo.

- Alcuni dei nuovi metodi di `Class` sono `getConstructor(Class t[])`, che restituisce un `Constructor` che riflette il costruttore avente una certa segnatura, e `getDeclaredMethods()`, che restituisce un array di `Method` che riflette i metodi della classe.

Capitolo 8

Multithreading

Un *thread* è un flusso di controllo (ovvero una sequenza di operazioni programmate) che può essere eseguito concorrentemente ad altri thread.

In Java si crea un thread istanziando un oggetto di una classe derivata da `Thread`. Questa classe deve implementare il metodo `run`.

```
public class SimpleThread extends Thread {
    private int cntDown = 4;
    private int threadId;
    private static int threadCnt = 0;
    public SimpleThread()
    {
        threadId = ++threadCnt;
        System.out.println("Making thread " + threadId);
    }
    public void run()
    {
        for (;;) {
            System.out.println("thread "
                      + threadId + " countdown: " + cntDown);
            if (--cntDown == 0)
```

```
                return;
            }
        }
        public static void main(String[] args)
        {
            for (int i = 0; i < 5; i++)
                new SimpleThread().start();
            System.out.println("All threads started");
        }
}
```
