Lezione 16

Le classi di base nel pacchetto java.lang

Finora si sono incontrate e conosciute diverse classi e diverse interfacce appartenenti al package *java.lang*, come *Thread*, *Runnable*, *Exception* e così via. Alcune di esse hanno già trovato il giusto grado di approfondimento, altre sono state trattate marginalmente, mentre altre ancora non sono state ancora menzionate. Questa lezione completa la panoramica sulle interfacce di programmazione (API) presenti all'interno del pacchetto *java.lang*, il package più basilare di Java.

16.1 - Object

Della classe *Object* si è già parlato in altre occasioni, nel corso delle precedenti lezioni. Tuttavia, finora ancora non si è considerata una panoramica d'insieme di tutte le caratteristiche offerte dalla madre di tutte le classi. I membri esposti da *Object* sono riassunti nella seguente tabella.

Costruttori pubblici
`Object()`
Metodi protetti
`Object clone()`
`void finalize()`
Metodi pubblici
`boolean equals(Object obj)`
`final Class getClass()`
`int hashCode()`
`final void notify()`
`final void notifyAll()`
`String toString()`
`final void wait()`
`final void wait(int millis)`
`final void wait(int millis, int nanos)`

Tabella 16.1
I membri della classe *Object*.

Costruttori

Object ha un solo costruttore, privo di argomenti. Questo costruttore è utile solo durante l'impiego dell'ereditarietà. Istanziare *Object*, di norma, non offre utilità.

Metodi protetti

Il metodo ***finalize()*** contiene il codice di finalizzazione dell'oggetto. Ogni classe, che implicitamente deriva da *Object*, può ridefinire questo metodo rendendolo pubblico. In tal

caso, *finalize()* sarà richiamato dal garbage collector prima di rimuovere l'oggetto dalla memoria. Se l'oggetto deve rilasciare delle risorse al momento della sua distruzione, il metodo *finalize()* si rivela molto utile. Tuttavia, non bisogna confondere i finalizzatori di Java con i distruttori di altri linguaggi, come C++. Java, di conseguenza al suo modello di gestione della memoria, non prevede distruttori. I finalizzatori svolgono un ruolo simile a quello dei distruttori, con l'unica differenza che non è dato sapere il momento esatto in cui un finalizzatore sarà richiamato, tranne nel caso in cui la garbage collection e la finalizzazione vengano comandate esplicitamente.

Il metodo ***clone()*** è la variante Java dei costruttori di copia di C++, con alcune differenze. Sostanzialmente, si richiama il metodo *clone()* di un oggetto quando si intende ottenere una sua copia. Affinché un oggetto possa essere clonato, è necessario che la sua classe di appartenenza implementi l'interfaccia *java.lang.Cloneable*. In caso contrario, si otterrà una *java.lang.CloneNotSupportedException*. L'interfaccia *Cloneable* non richiede alcun metodo. Semplicemente, contrassegna la classe, segnalando che le sue istanze supportano la clonazione. Inoltre, è necessario rendere pubblico l'accesso a *clone()*, ridefinendolo. Si sperimenti il seguente esempio:

```
// Rettangolo.java
public class Rettangolo implements Cloneable {

  private int larghezza;
  private int altezza;

  public Rettangolo(int larghezza, int altezza) {
    setLarghezza(larghezza);
    setAltezza(altezza);
  }

  public void setLarghezza(int larghezza) {
    this.larghezza = larghezza;
  }

  public void setAltezza(int altezza) {
    this.altezza = altezza;
  }

  public int getLarghezza() {
    return larghezza;
  }

  public int getAltezza() {
    return altezza;
  }

  public Object clone() {
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      return null;
    }
  }
```

```
}

// Test1.java
public class Test1 {

  public static void main(String[] args) {
    Rettangolo r1 = new Rettangolo(10, 6);
    Rettangolo r2 = (Rettangolo)r1.clone();
    System.out.println("r1 e r2 sono oggetti distinti?");
    if (r1 == r2) {
      System.out.println("No");
    } else {
      System.out.println("Sì");
    }
    System.out.println();
    System.out.println("--- Valori di r1 ---");
    System.out.println("Larghezza: " + r1.getLarghezza());
    System.out.println("Altezza: " + r1.getAltezza());
    System.out.println("--- Valori di r2 ---");
    System.out.println("Larghezza: " + r2.getLarghezza());
    System.out.println("Altezza: " + r2.getAltezza());
    System.out.println();
    System.out.println("Cambio i valori di r1...");
    System.out.println();
    r1.setLarghezza(15);
    r1.setAltezza(9);
    System.out.println("--- Valori di r1 ---");
    System.out.println("Larghezza: " + r1.getLarghezza());
    System.out.println("Altezza: " + r1.getAltezza());
    System.out.println("--- Valori di r2 ---");
    System.out.println("Larghezza: " + r2.getLarghezza());
    System.out.println("Altezza: " + r2.getAltezza());
  }

}
```

La classe *Rettangolo* implementa l'interfaccia *Cloneable* e ridefinisce il metodo *clone()*, al seguente modo:

```
public Object clone() {
  try {
    return super.clone();
  } catch (CloneNotSupportedException e) {
    return null;
  }
}
```

Con la chiamata a *super.clone()*, si ottiene una copia bit a bit dell'oggetto di invocazione. La nuova istanza viene trasmessa al codice chiamante, come un riferimento di tipo *Object*. Per invocare su di esso i metodi di *Rettangolo*, quindi, ci vuole un casting come quello eseguito dalla classe *Test1*:

```
Rettangolo r1 = new Rettangolo(10, 6);
Rettangolo r2 = (Rettangolo) r1.clone();
```

L'output mostrato dalla classe *Test1* è il seguente:

```
r1 e r2 sono oggetti distinti?
Sì

--- Valori di r1 ---
Larghezza: 10
Altezza: 6
--- Valori di r2 ---
Larghezza: 10
Altezza: 6

Cambio i valori di r1...

--- Valori di r1 ---
Larghezza: 15
Altezza: 9
--- Valori di r2 ---
Larghezza: 10
Altezza: 6
```

Come l'applicazione dimostra, *r2* viene generato come copia di *r1*, automaticamente. Pertanto, *r1* ed *r2* non sono alias del medesimo oggetto, ma sono riferimenti diversi ad oggetti distinti. Modificando l'uno, l'altro rimane invariato.
Si faccia attenzione a non cadere in un tranello molto comune. Quando si esegue una copia bit a bit di un oggetto, si duplicano tutte le variabili contenute al suo interno. Finché sono variabili di tipo predefinito (*byte*, *short*, *int*, *long*, *char*, *double*, *float* e *boolean*), non c'è problema. Se si tratta di riferimenti ad oggetti, bisogna stare più attenti. La copia di un riferimento non equivale alla copia dell'oggetto riferito. Se un oggetto *A* mantiene un riferimento ad un oggetto *B*, la copia offerta dal *clone()* di *Object* copia il riferimento verso *B*, non *B* stesso. L'oggetto *A* ed il suo clone faranno riferimento alla stessa versione di *B*. Lo dimostra il seguente esempio:

```
// Persone.java
public class Persone implements Cloneable {

  private int size;
  private String[] nomi;
  private String[] cognomi;

  public Persone(int size) {
    this.size = size;
    nomi = new String[size];
    cognomi = new String[size];
  }

  public void setNome(String nome, int posizione) {
    nomi[posizione] = nome;
  }

  public void setCognome(String cognome, int posizione) {
    cognomi[posizione] = cognome;
  }
```

```
  public int getSize() {
    return size;
  }

  public String getNome(int posizione) {
    return nomi[posizione];
  }

  public String getCognome(int posizione) {
    return cognomi[posizione];
  }

  public Object clone() {
    try {
      return super.clone();
    } catch (CloneNotSupportedException e) {
      return null;
    }
  }

}

// Test2.java
public class Test2 {

  public static void main(String[] args) {
    Persone p1 = new Persone(2);
    p1.setNome("Mario", 0);
    p1.setCognome("Rossi", 0);
    p1.setNome("Luigi", 1);
    p1.setCognome("Bianchi", 1);
    Persone p2 = (Persone)p1.clone();
    System.out.println("--- Contenuto di p1 ---");
    for (int i = 0; i < p1.getSize(); i++) {
      System.out.println(
        p1.getNome(i) + " " + p1.getCognome(i)
      );
    }
    System.out.println("--- Contenuto di p2 ---");
    for (int i = 0; i < p2.getSize(); i++) {
      System.out.println(
        p2.getNome(i) + " " + p2.getCognome(i)
      );
    }
    System.out.println();
    System.out.println("Cambio p2...");
    System.out.println();
    p2.setNome("Enzo", 0);
    p2.setCognome("Grigi", 0);
    p2.setNome("Antonio", 1);
    p2.setCognome("Verdi", 1);
    System.out.println("--- Contenuto di p1 ---");
    for (int i = 0; i < p1.getSize(); i++) {
      System.out.println(
        p1.getNome(i) + " " + p1.getCognome(i)
      );
    }
```

```
    System.out.println("--- Contenuto di p2 ---");
    for (int i = 0; i < p2.getSize(); i++) {
      System.out.println(
        p2.getNome(i) + " " + p2.getCognome(i)
      );
    }
  }

}
```

La classe *Persone* rappresenta un insieme di coppie *nome-cognome*. I valori vengono memorizzati all'interno di due distinti array. Gli array sono oggetti. Pertanto, una copia bit a bit di un'istanza di persone comporta la copia dei riferimenti agli array, non del loro effettivo contenuto. Quando si lancia la classe *Test2*, si nota come cambiando il contenuto degli array dell'oggetto clonato si modifica anche il contenuto dell'oggetto originale:

```
--- Contenuto di p1 ---
Mario Rossi
Luigi Bianchi
--- Contenuto di p2 ---
Mario Rossi
Luigi Bianchi

Cambio p2...

--- Contenuto di p1 ---
Enzo Grigi
Antonio Verdi
--- Contenuto di p2 ---
Enzo Grigi
Antonio Verdi
```

Il problema si può risolvere effettuando una *copia profonda* dell'oggetto. Il metodo *clone()*, in questo caso, va arricchito con le istruzioni capaci di duplicare gli oggetti usati dalla classe:

```
// Persone.java
public class Persone implements Cloneable {

  private int size;
  private String[] nomi;
  private String[] cognomi;

  public Persone(int size) {
    this.size = size;
    nomi = new String[size];
    cognomi = new String[size];
  }

  public void setNome(String nome, int posizione) {
    nomi[posizione] = nome;
  }

  public void setCognome(String cognome, int posizione) {
    cognomi[posizione] = cognome;
  }
```

```
  public int getSize() {
    return size;
  }

  public String getNome(int posizione) {
    return nomi[posizione];
  }

  public String getCognome(int posizione) {
    return cognomi[posizione];
  }

  public Object clone() {
    try {
      // Prima eseguo la copia bit a bit.
      Persone clonato = (Persone)super.clone();
      // Adesso eseguo la copia profonda.
      clonato.nomi = new String[size];
      clonato.cognomi = new String[size];
      for (int i = 0; i < size; i++) {
        clonato.nomi[i] = nomi[i];
        clonato.cognomi[i] = cognomi[i];
      }
      // Restituisco l'elaborato.
      return clonato;
    } catch (CloneNotSupportedException e) {
      return null;
    }
  }

}

// Test3.java
public class Test3 {

  public static void main(String[] args) {
    Persone p1 = new Persone(2);
    p1.setNome("Mario", 0);
    p1.setCognome("Rossi", 0);
    p1.setNome("Luigi", 1);
    p1.setCognome("Bianchi", 1);
    Persone p2 = (Persone)p1.clone();
    System.out.println("--- Contenuto di p1 ---");
    for (int i = 0; i < p1.getSize(); i++) {
      System.out.println(
        p1.getNome(i) + " " + p1.getCognome(i)
      );
    }
    System.out.println("--- Contenuto di p2 ---");
    for (int i = 0; i < p2.getSize(); i++) {
      System.out.println(
        p2.getNome(i) + " " + p2.getCognome(i)
      );
    }
    System.out.println();
    System.out.println("Cambio p2...");
```

```
    System.out.println();
    p2.setNome("Enzo", 0);
    p2.setCognome("Grigi", 0);
    p2.setNome("Antonio", 1);
    p2.setCognome("Verdi", 1);
    System.out.println("--- Contenuto di p1 ---");
    for (int i = 0; i < p1.getSize(); i++) {
      System.out.println(
        p1.getNome(i) + " " + p1.getCognome(i)
      );
    }
    System.out.println("--- Contenuto di p2 ---");
    for (int i = 0; i < p2.getSize(); i++) {
      System.out.println(
        p2.getNome(i) + " " + p2.getCognome(i)
      );
    }
  }

}
```

Il problema è risolto. Cambiando l'elenco contenuto in *p2*, i valori di *p1* non variano. Gli oggetti sono completamente indipendenti:

```
--- Contenuto di p1 ---
Mario Rossi
Luigi Bianchi
--- Contenuto di p2 ---
Mario Rossi
Luigi Bianchi

Cambio p2...

--- Contenuto di p1 ---
Mario Rossi
Luigi Bianchi
--- Contenuto di p2 ---
Enzo Grigi
Antonio Verdi
```

Metodi pubblici

Il metodo ***equals()*** confronta due oggetti, restituendo *true* quando i loro contenuti sono equivalenti. Ridefinendo questo metodo, fornirete ai vostri oggetti la possibilità di essere messi a confronto. Un ottimo esempio è fornito dall'implementazione di *equals()* impiegata dagli oggetti *String*. Le stringhe sono oggetti. Un confronto come

```
stringa1 == stringa2
```

non verifica il contenuto delle due stringhe. Semplicemente, valuta i riferimenti. L'espressione ritorna *true* solo se i due riferimenti puntano allo stesso oggetto. Se si tratta di due oggetti differenti, verrà restituito *false*, anche nel caso in cui le stringhe abbiano contenuto identico. Il confronto tra stringhe, quindi, si effettua mediante *equals()*:

```
stringa1.equals(stringa2)
```

Lo stesso vale per tutti gli altri oggetti di Java. Per dotare le proprie classi di questa interessante caratteristica è necessario ridefinire il metodo *equals()*, includendo al suo interno tutte le verifiche necessarie per il confronto del contenuto degli oggetti. Ecco un esempio:

```
// Rettangolo.java
public class Rettangolo implements Cloneable {

  private int larghezza;
  private int altezza;

  public Rettangolo(int larghezza, int altezza) {
    setLarghezza(larghezza);
    setAltezza(altezza);
  }

  public void setLarghezza(int larghezza) {
    this.larghezza = larghezza;
  }

  public void setAltezza(int altezza) {
    this.altezza = altezza;
  }

  public int getLarghezza() {
    return larghezza;
  }

  public int getAltezza() {
    return altezza;
  }

  public boolean equals(Object obj) {
    if (obj instanceof Rettangolo) {
      Rettangolo r = (Rettangolo)obj;
      return (
        (r.larghezza == larghezza) &&
        (r.altezza   ==   altezza)
      );
    } else {
      return false;
    }
  }

}

// Test4.java
public class Test4 {

  public static void main(String[] args) {
    Rettangolo r1 = new Rettangolo(10, 6);
    Rettangolo r2 = new Rettangolo(15, 9);
    Rettangolo r3 = new Rettangolo(10, 6);
    if (r1.equals(r2)) {
      System.out.println("r1 e r2 sono rettangoli uguali.");
```

```
    } else {
      System.out.println("r1 e r2 sono rettangoli diversi.");
    }
    if (r1.equals(r3)) {
      System.out.println("r1 e r3 sono rettangoli uguali.");
    } else {
      System.out.println("r1 e r3 sono rettangoli diversi.");
    }
  }

}
```

L'output prodotto è:

```
r1 e r2 sono rettangoli diversi.
r1 e r3 sono rettangoli uguali.
```

Il metodo ***getClass()*** restituisce la classe dell'oggetto, sotto forma di istanza di *Class* (l'argomento è trattato nel paragrafo successivo). Giacché il metodo è *final*, non è possibile ridefinirlo.

Il metodo ***hashCode()*** restituisce un *codice hash* (sotto forma di valore *int*), atto ad identificare univocamente i contenuti dell'oggetto. I codici hash sono utili quando si lavora con speciali strutture dati. Per il momento, l'argomento viene tralasciato. Sarà ripreso nella lezione successiva.

Il compito di ***toString()*** è fornire una rappresentazione in stringa dell'oggetto corrente. Ridefinire questo metodo è una pratica molto comune, per fare in modo che ogni nuovo tipo di dato possa vantare una significativa e personale rappresentazione in stringa.

Di ***wait()*** (e delle sue varianti), di ***notify()*** e di ***notifyAll()*** se ne è già parlato nel corso della lezione precedente.

16.2 - Class

La classe *Class* consente di acquisire informazioni di runtime sulle classi e sulle interfacce disponibili all'interno della macchina virtuale. Inoltre, è la chiave per l'accesso alle caratteristiche riflessive di Java. La riflessione, ad ogni modo, non sarà trattata in questo corso. Per ora è interessante conoscere alcune caratteristiche degli oggetti *Class*. Per prima cosa, non è possibile istanziare in maniera diretta degli oggetti di tipo *Class*. Ci sono due modi per ottenere un riferimento ad un oggetto *Class*:

- Desumendolo da un oggetto esistente, attraverso il metodo *getClass()* di *Object*.
- Caricando dinamicamente una classe, attraverso il metodo statico *Class.forName()*.

Un esempio circa il primo caso:

```
public class Test1 {

  public static void main(String[] args) {
    String test = "Stringa di prova";
```

```
    Class classDiTest = test.getClass();
    System.out.println(
      "L'oggetto test è di tipo: " + classDiTest.getName()
    );
  }

}
```

Attraverso il metodo *getClass()* è stato recuperato l'oggetto *Class* associato ad una stringa. Quindi, a puro scopo dimostrativo, si è verificato il nome della classe. L'output mostrato è:

```
L'oggetto test è di tipo: java.lang.String
```

Il metodo statico ***forName()*** fornisce l'oggetto *Class* corrispondente al percorso specificato come argomento. Si usa al seguente modo:

```
Class c = Class.forName("nome completo della classe");
```

Ad esempio:

```
Class c = Class.forName("java.lang.String");
```

Nel caso il percorso specificato non corrisponda ad alcuna classe esistente, sarà propagata una *java.lang.ClassNotFoundException*. Questo tipo di eccezione va sempre gestita. Il seguente esempio è molto interessante. Oltre a dimostrare il funzionamento del metodo *forName()*, introduce il discorso della riflessione:

```
public class EsaminaClasse {

  public static void main(String[] args) {
    if (args.length == 1) {
      try {
        Class c = Class.forName(args[0]);
        System.out.println("La classe " + args[0] + " esiste.");
        Class s = c.getSuperclass();
        if (s != null) {
          System.out.println("E' sottoclasse di " + s.getName() + ".");
        }
        System.out.println(
          "Ha " + c.getConstructors().length + " costruttori."
        );
        System.out.println(
          "Ha " + c.getFields().length + " proprieta'."
        );
        System.out.println(
          "Ha " + c.getMethods().length + " metodi."
        );
      } catch (ClassNotFoundException e) {
        System.out.println("La classe " + args[0] + " non esiste.");
      }
    }
  }
```

```
}
```

Il programma si usa da riga di comando. Richiede un argomento, che serve per specificare il percorso della classe da esaminare. Dopo aver compilato la classe, è possibile richiamarla al seguente modo:

```
java EsaminaClasse java.lang.Thread
```

Si dovrebbe ricevere un output come il seguente (mostrato anche in Figura 16.1):

```
La classe java.lang.Thread esiste.
E' sottoclasse di java.lang.Object.
Ha 8 costruttori.
Ha 3 proprieta'.
Ha 42 metodi.
```

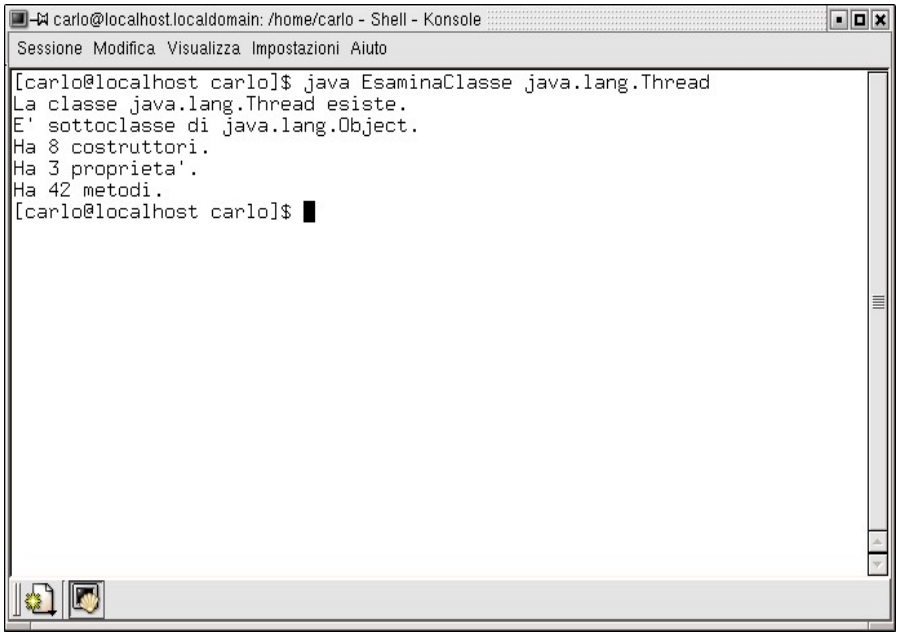

Figura 16.1
L'applicazione *EsaminaClasse* fornisce una panoramica d'insieme sulle classi di Java.

Chi è interessato alla riflessione può consultare la documentazione ufficiale di Java. Le pagine di interesse sono sia quella di *java.lang.Class* sia quelle dedicate al contenuto del package *java.lang.reflect*.

16.3 - System

La classe *System* fornisce una nutrita schiera di utilità per ogni tipo di applicazione. Tutti i membri di *System* si presentano sotto forma statica. La classe agisce da semplice contenitore. Non è possibile creare oggetti di tipo *System*.

Costanti pubbliche statiche
`java.io.PrintStream err`
`java.io.InputStream in`
`java.io.PrintStream out`

Metodi pubblici statici
`void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)`
`long currentTimeMillis()`
`void exit(int status)`
`void gc()`
`java.util.Properties getProperties()`
`String getProperty(String key)`
`String getProperty(String key, String default)`
`SecurityManager getSecurityManager()`
`void runFinalization()`
`void setErr(java.io.PrintStream err)`
`void setIn(java.io.InputStream in)`
`void setOut(java.io.PrintStream out)`
`void setProperties(java.util.Properties properties)`
`String setProperty(String key, String value)`
`void setSecurityManager(SecurityManager s)`

Tabella 16.2
I principali membri della classe *System*.

I campi ***err***, ***in*** e ***out*** rappresentano dei canali di comunicazione. I particolari delle procedure di input ed output con Java saranno esplorate fra un paio di lezioni. Per il momento è sufficiente sapere che questi canali vengono assegnati automaticamente all'avvio della macchina virtuale. Tramite essi, è possibile stabilire una basilare comunicazione con l'utente. Diverse volte è stato usato il canale *out*:

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

Il canale *out* invia messaggi in output; il canale in permette l'acquisizione dei dati dall'esterno; il canale *err* è simile ad *out*, solo che è specifico per i messaggi di errore. Nelle più comuni implementazioni di Java, tutti e tre i canali comunicano con la riga di comando del sistema. Ad ogni modo, si è liberi di dirigerli dove si vuole, attraverso i metodi ***setErr()***, ***setIn()*** e ***setOut()***.

Il metodo ***arraycopy()*** permette la copia rapida degli array. Bisogna specificare cinque argomenti:

1. *Object src*, l'array di origine da copiare.
2. *int srcPos*, la posizione di partenza nell'array di origine.
3. *Object dest*, l'array di destinazione sul quale devono essere copiati gli elementi.
4. *int destPos*, la posizione di partenza nell'array di destinazione.
5. *int length*, il numero di elementi consecutivi che devono essere copiati.

Il metodo ***currentTimeMillis()*** restituisce la data e l'orario correnti, sotto forma di numero *long*. Viene impiegata una rappresentazione che ricorre frequentemente in ambito informatico. Il valore restituito indica il numero di millisecondi trascorsi dall'inizio del 1970 al momento in cui il metodo è stato richiamato. Questa rappresentazione può successivamente essere processata ed adattata ai propri scopi. Il più delle volte, ad ogni modo, le date e gli orari vengono gestiti servendosi della classe *java.util.Date*, che sarà descritta nella prossima lezione. *System.currentTimeMillis()*, invece, trova facile impiego in fase di debug, per misurare le prestazioni del codice. Ecco un esempio:

```
public class TestPrestazioni {

  public static void main(String[] args) {
    // Registro il tempo di partenza.
    long t1 = System.currentTimeMillis();
    // Eseguo l'operazione da cronometrare.
    int a = 5;
    int b = 10;
    for (int i = 0; i < 1000000; i++) {
      int aux = a;
      a = b;
      b = aux;
    }
    // Registro il tempo di arrivo.
    long t2 = System.currentTimeMillis();
    // Calcolo il tempo impiegato.
    long dt = t2 - t1;
    // Mostro il risultato.
    System.out.println(
      "Ho scambiato per un milione di volte il "   +
      "contenuto di due variabili. Ho completato " +
      "l'operazione in " + dt + " millisecondi."
    );
  }

}
```

Questo programma scambia per un milione di volte il contenuto di due variabili. Il metodo *currentTimeMillis()* viene consultato due volte, prima e dopo la procedura. In questo modo è possibile determinare il tempo impiegato per portare a compimento l'operazione, eseguendo una semplice sottrazione. Con il mio computer, anche grazie alla compilazione JIT, sembra che le prestazioni di Java non siano malvagie:

```
Ho scambiato per un milione di volte il contenuto di due variabili. Ho completato
l'operazione in 10 millisecondi.
```

Visto che si è entrati in argomento, consiglio un esperimento interessante. Se nel sistema si dispone sia di un compilatore JIT sia di una macchina virtuale tradizionale, si può provare a mettere a confronto le prestazioni dei due ambienti. Con il kit di sviluppo di Sun per Windows si trovano ambo le macchine virtuali. La compilazione JIT è attiva per default. In ogni caso la si può richiamare esplicitamente al seguente modo:

```
java -client TestPrestazioni
```

La macchina virtuale tradizionale, invece, si richiama al seguente modo:

```
java -server TestPrestazioni
```

Sfruttando quest'ultima modalità, ricevo il risultato:

```
Ho scambiato per un milione di volte il contenuto di due variabili. Ho completato
l'operazione in 230 millisecondi.
```

Il metodo ***exit()*** interrompe bruscamente l'esecuzione del programma. Bisogna fornire un argomento al metodo, di tipo *int*. L'argomento è impiegato per informare il sistema sulle cause dell'uscita dal programma. Per convenzione, zero indica che il programma è terminato normalmente, mentre un altro valore qualsiasi indica un'uscita forzata a causa di un errore.

Il metodo ***gc()*** avvia la garbage collection, il cui compito è individuare tutti gli oggetti che possono essere scaricati dalla memoria. Il metodo ***runFinalization()*** avvia la finalizzazione di tutti gli oggetti raccolti dal garbage collector. Di norma, non c'è bisogno di appellarsi a questi due metodi. La macchina virtuale di Java li richiama di tanto in tanto, quando ha tempo e spazio per farlo. In situazioni particolari, tuttavia, potrebbe essere utile forzare la garbage collection.

I metodi ***getSecurityManager()*** e ***setSecurityManager()*** recuperano ed impostano il *SecurityManager* abbinato alla macchina virtuale. I *SecurityManager* sono impiegati per gestire la sicurezza delle applicazioni Java. L'argomento non è trattato in questo corso.

L'ambiente di esecuzione di Java viene condizionato dalla presenza di alcune proprietà globali associate ad ogni applicazione. Ogni proprietà è costituita da una coppia di stringhe, dette *chiave* e *valore*. In sostanza, la chiave costituisce il nome della singola proprietà, mentre il valore è il suo contenuto. Conoscendo la chiave di una delle proprietà dell'ambiente, è possibile recuperarne il valore con il metodo ***getProperty()***. Si usa al seguente modo:

```
String valore = System.getProperty("chiave");
```

Se la chiave espressa non corrisponde ad alcuna proprietà, il valore restituito sarà un riferimento a *null*. In alternativa, è possibile sfruttare la variante:

```
String valore = System.getProperty("chiave", "default");
```

Ora, nel caso la proprietà ricercata non esista, sarà restituito il valore di default, anziché *null*.

Le proprietà dell'ambiente, oltre che essere lette, possono anche venire impostate, attraverso il metodo ***setProperty()***. L'impiego è semplice:

```
System.setProperty("chiave", "valore");
```

Se già esiste una proprietà associata alla chiave specificata, il suo valore verrà sovrascritto. In caso contrario, una nuova proprietà sarà aggiunta all'elenco dell'ambiente.

Il metodo ***getProperties()*** restituisce tutte le proprietà in un colpo solo, rappresentandole con un'istanza della classe *java.util.Properties*, esaminata nella lezione successiva. Allo stesso modo, tutte le proprietà possono essere scritte con una sola istruzione, usando il metodo ***setProperties()***. Il metodo richiede un argomento di tipo *java.util.Properties*.

Alcune proprietà sono sempre disponibili, all'avvio di ogni applicazione.

Chiave	***Valore***
`java.version`	La versione di Java in uso.
`java.vendor`	Il produttore della versione di Java in uso.
`java.vendor.url`	L'URL del produttore della versione di Java in uso.
`java.home`	La directory di installazione di Java.
`java.vm.specification.version`	La versione delle specifiche della macchina virtuale in uso.
`java.vm.specification.vendor`	Il produttore delle specifiche della macchina virtuale in uso.
`java.vm.specification.name`	Il nome delle specifiche della macchina virtuale in uso.
`java.vm.version`	La versione della macchina virtuale in uso.
`java.vm.vendor`	Il produttore della macchina virtuale in uso.
`java.vm.name`	Il nome della macchina virtuale in uso.
`java.specification.version`	La versione delle specifiche di Java in uso.
`java.specification.vendor`	Il produttore delle specifiche di Java in uso.
`java.specification.name`	Il nome delle specifiche di Java in uso.
`java.class.version`	La versione delle classi di Java.
`java.class.path`	Il percorso delle classi di Java.
`java.library.path`	Il percorso delle librerie di Java.
`java.io.tmpdir`	Il percorso della direcory dei file temporanei.
`java.ext.dirs`	I percorsi delle directory che contengono le estensioni di Java.
`os.name`	Il nome del sistema operativo in uso.
`os.arch`	L'architettura del sistema operativo in uso.
`os.version`	La versione del sistema operativo in uso.
`file.separator`	La sequenza per la separazione degli elementi dei percorsi nel sistema in uso.
`path.separator`	La sequenza per la separazione dei percorsi nel sistema in uso.

Chiave	*Valore*
`line.separator`	La sequenza impiegata dal sistema in uso per esprire il ritorno a capo.
`user.name`	Il nome dell'utente che sta usando l'applicazione.
`user.home`	La home directory dell'utente che sta usando l'applicazione.
`user.dir`	L'attuale cartella di lavoro dell'utente che sta usando l'applicazione.

Tabella 16.3
Le proprietà automaticamente disponibili nell'ambiente Java.

Queste proprietà possono essere verificate con il seguente esempio:

```
public class PropertiesTest {

  public static void main(String[] args) {
    String[] keys = {
      "java.version", "java.vendor", "java.vendor.url",
      "java.home", "java.vm.specification.version",
      "java.vm.specification.vendor", "java.vm.specification.name",
      "java.vm.version", "java.vm.vendor", "java.vm.name",
      "java.specification.version", "java.specification.vendor",
      "java.specification.name", "java.class.version",
      "java.class.path", "java.library.path", "java.io.tmpdir",
      "java.ext.dirs", "os.name", "os.arch", "os.version",
      "file.separator", "path.separator", "line.separator",
      "user.name", "user.home", "user.dir"
    };
    for (int i = 0; i < keys.length; i++) {
      System.out.println(
        "[" + keys[i] + "] " +
        System.getProperty(keys[i])
      );
    }
  }

}
```

Ecco l'output riscontrato nel mio caso:

```
[java.version] 1.4.1_02
[java.vendor] Sun Microsystems Inc.
[java.vendor.url] http://java.sun.com/
[java.home] /usr/lib/j2sdk1.4.1_02/jre
[java.vm.specification.version] 1.0
[java.vm.specification.vendor] Sun Microsystems Inc.
[java.vm.specification.name] Java Virtual Machine Specification
[java.vm.version] 1.4.1_02-b06
[java.vm.vendor] Sun Microsystems Inc.
```

```
[java.vm.name] Java HotSpot(TM) Client VM
[java.specification.version] 1.4
[java.specification.vendor] Sun Microsystems Inc.
[java.specification.name] Java Platform API Specification
[java.class.version] 48.0
[java.class.path] .
[java.library.path]
/usr/lib/j2sdk1.4.1_02/jre/lib/i386/client:/usr/lib/j2sdk1.4.1_02/jre/lib/i386:/usr
/lib/j2sdk1.4.1_02/jre/../lib/i386
[java.io.tmpdir] /tmp
[java.ext.dirs] /usr/lib/j2sdk1.4.1_02/jre/lib/ext
[os.name] Linux
[os.arch] i386
[os.version] 2.4.19-16mdk
[file.separator] /
[path.separator] :
[line.separator]

[user.name] carlo
[user.home] /home/carlo
[user.dir] /home/carlo/testjava
```

16.4 - Runtime e Process

La classe *Runtime* fornisce delle informazioni sul corrente ambiente di runtime. Non esiste un costruttore per questa classe. In compenso, è possibile recuperare l'istanza di *Runtime* associata all'esecuzione corrente, richiamando il metodo statico *Runtime.getRuntime()*:

```
Runtime r = Runtime.getRuntime();
```

Esamineremo alcuni dei metodi più significativi degli oggetti Runtime.

Il metodo ***totalMemory()*** restituisce il quantitativo di memoria totale attualmente assegnato alla Java Virtual Machine. Il valore di ritorno è espresso in byte e rappresentato mediante un valore *long*. Allo stesso modo, *freeMemory()* restituisce il quantitativo delle memoria non utilizzata:

```
public class ControllaMemoria {

  public static void main(String[] args) {
    Runtime r = Runtime.getRuntime();
    long totale = r.totalMemory();
    long libera = r.freeMemory();
    long occupata = totale - libera;
    System.out.println("Memoria totale: " + totale);
    System.out.println("Memoria libera: " + libera);
    System.out.println("Memoria occupata: " + occupata);
  }

}
```

Le diverse varianti del metodo ***exec()***, che invito a verificare nella documentazione ufficiale, permettono un basilare aggancio al sistema operativo in uso. Attraverso *exec()* è possibile eseguire un comando valido sul sistema operativo in uso. Il comando va espresso sotto forma

di stringa. Il metodo *exec()* può lanciare una *java.io.IOException*, nel caso non sia possibile stabilire un canale di comunicazione con il sistema in uso. Su di un sistema Windows la seguente applicazione avvia il solitario (sempre ammettendo che sia stato installato insieme al sistema):

```
public class TestExec {

  public static void main(String[] args) {
    Runtime r = Runtime.getRuntime();
    try {
      r.exec("sol");
    } catch (Exception e) {
      System.out.println("Impossibile eseguire il comando");
    }
  }

}
```

Chio utilizza altri sistemi può modificare la stringa lanciata da *exec()* con un qualsiasi comando valido per la specifica macchina.
Il metodo *exec()* restituisce un'istanza della classe ***Process***. Grazie ad essa è possibile mantenere il controllo del processo lanciato. Ogni istanza di *Process* dispone dei metodi sotto elencati:

- ***void destroy()***. Termina il processo lanciato.
- ***int exitValue()***. Restituisce il valore con ha comunicato il processo alla sua uscita. Ovviamente è possibile richiamare questo metodo solo dopo che il processo è terminato. In caso contrario, si riceve una *RuntimeException* di tipo *IllegalThreadStateException*.
- ***java.io.InputStream getErrorStream()***. Fornisce un appiglio utile per leggere ciò che il processo scrive nel suo canale di errore.
- ***java.io.InputStream getInputStream()***. Fornisce un appiglio utile per leggere ciò che il processo scrive nel suo canale di output.
- ***java.io.OutputStream getOutputStream()***. Fornisce un appiglio utile per scrivere nel canale di input del processo, fornendo dei dati all'applicazione eseguita.
- ***void waitFor()***. Blocca il thread corrente fino alla morte del processo. Può lanciare una *InterruptedException*.

Grazie a questi metodi, è possibile stabilire un controllo ed una comunicazione nei confronti del processo lanciato con *exec()*.

16.5 - StringBuffer

Le stringhe di Java sono immodificabili. Una volta che si è istanziata una stringa, non è possibile cambiarne il contenuto. Al limite, è possibile cambiare il riferimento:

```
String a = "Buongiorno";
a = "Buonasera";
```

Con questo codice, ad esempio, è stata associata la stringa "Buongiorno" alla variabile *a*.

Quando, nella seconda istruzione, si cambia il contenuto di a, si modifica semplicemente il riferimento. Non è stato cambiato il contenuto della stringa "Buongiorno", che vive ancora in qualche parte della memoria. In pratica, non è possibile modificare anche un solo carattere di una stringa di tipo *String*. Questo approccio presenta caratteristiche di efficienza solo nel caso in cui le stringhe gestite non vengano sottoposte a numerose operazioni. In caso contrario, diventa svantaggioso. Si consideri un altro semplice esempio:

```
String a = "Ciao ";
a += "a tutti";
```

Questa sequenza ha portato all'impiego di tre differenti oggetti *String*. Il primo è "Ciao ", il secondo è "a tutti", ed il terzo è "Ciao a tutti". Alla fine, però, si finirà per usere solamente uno di essi, poiché l'unico riferimento che si ha in mano, al termine dell'esecuzione, è quello verso la stringa "Ciao a tutti". Insomma, quando si deve lavorare pesantemente con le stringhe, l'impiego della classe *String* non è tanto conveniente. Per questo motivo, Java fornisce un secondo tipo di stringhe, rappresentato dalla classe *StringBuffer*. Un oggetto *StringBuffer* costituisce una stringa che può essere controllata e modificata, carattere per carattere. Uno *StringBuffer* vuoto si crea alla seguente maniera:

```
StringBuffer s = new StringBuffer();
```

In alternativa, è possibile fornire un valore iniziale alla stringa:

```
StringBuffer s = new StringBuffer("valore iniziale");
```

La seguente tabella riporta e commenta i più importanti metodi degli oggetti *StringBuffer*.

Tipo restituito	***Metodo e parametri***	***Descrizione***
`StringBuffer`	`append(boolean b)`	Aggiunge il valore di *b* in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`append(char c)`	Aggiunge il carattere *c* in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`append(char[] c)`	Aggiunge i caratteri contenuti nell'array in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`append(double d)`	Aggiunge il valore di *d* in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.

Tipo restituito	*Metodo e parametri*	*Descrizione*
`StringBuffer`	`append(float f)`	Aggiunge il valore di *f* in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`append(int i)`	Aggiunge il valore di *i* in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`append(long l)`	Aggiunge il valore di *l* in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`append(Object obj)`	Aggiunge il valore di *obj.toString()* in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`append(String s)`	Aggiunge *s* in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`append(StringBuffer s)`	Aggiunge *s* in coda alla stringa. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`char`	`charAt(int i)`	Restituisce il carattere alla posizione *i*.
`StringBuffer`	`delete(int start, int end)`	Rimuove tutti i caratteri dall'indice *start* (incluso) all'indice *end* (escluso). Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`deleteCharAt(int i)`	Rimuove il carattere alla posizione *i*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`int`	`indexOf(String s)`	Restituisce la prima posizione della sottostringa *s*, oppure -1 nel caso tale sottostringa non compaia nell'oggetto di invocazione.
`int`	`indexOf(String s, int i)`	Come il precedente, con la differenza che la ricerca della sottostringa *s* prende piede dalla posizione *i*.
`StringBuffer`	`insert(int offset, boolean b)`	Aggiunge il valore di *b* alla stringa, inserendolo alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.

Tipo restituito	*Metodo e parametri*	*Descrizione*
`StringBuffer`	`insert(int offset, char c)`	Aggiunge il carattere *c* alla stringa, inserendolo alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`insert(int offset, char[] c)`	Aggiunge alla stringa i caratteri contenuti nell'array, inserendoli alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`insert(int offset, double d)`	Aggiunge il valore di *d* alla stringa, inserendolo alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`insert(int offset, float f)`	Aggiunge il valore di *f* alla stringa, inserendolo alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`insert(int offset, int i)`	Aggiunge il valore di *i* alla stringa, inserendolo alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`insert(int offset, long l)`	Aggiunge il valore di *l* alla stringa, inserendolo alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`insert(int offset, Object obj)`	Aggiunge il valore di *obj.toString()* alla stringa, inserendolo alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`insert(int offset, String s)`	Aggiunge *s* alla stringa, inserendola alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`StringBuffer`	`insert(int offset, StringBuffer s)`	Aggiunge *s* alla stringa, inserendola alla posizione *offset*. Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`int`	`length()`	Restituisce le dimensioni della stringa.

Tipo restituito	*Metodo e parametri*	*Descrizione*
`StringBuffer`	`replace(int start, int end, String str)`	Cambia in *str* la sottostringa dalla posizione *start* (compresa) alla posizione *end* (esclusa). Modifica l'oggetto di invocazione, ed in più restituisce un riferimento allo stesso *StringBuffer*.
`void`	`setCharAt(int i, char c)`	Cambia in *c* il carattere alla posizione *i*.
`String`	`toString()`	Restituisce un oggetto *String* con il medesimo contenuto dello *StringBuffer* di invocazione.

Tabella 16.4
I più importanti metodi di cui sono dotati gli oggetti *StringBuffer*.

Altri metodi, un po' più peculiari, possono essere appresi dalla documentazione ufficiale.

16.6 - Math

La classe *Math* comprende costanti e metodi di carattere matematico. Tutti i suoi membri sono statici, pertanto non bisogna mai istanziare oggetti di tipo *Math*. La classe lavora semplicemente da contenitore.

Costanti pubbliche statiche
`double E`
`double PI`
Metodi pubblici statici
`double abs(double a)`
`float abs(float a)`
`int abs(int a)`
`long abs(long a)`
`double acos(double a)`
`double asin(double a)`
`double atan(double a)`
`double atan2(double y, double x)`
`double ceil(double a)`
`double cos(double a)`
`double exp(double a)`
`double floor(double a)`
`double log(double a)`
`double max(double a, double  b)`
`float max(float a, float  b)`

`int max(int a, int b)`
`long max(long a, long b)`
`double min(double a, double  b)`
`float min(float a, float  b)`
`int min(int a, int b)`
`long min(long a, long b)`
`double pow(double a, double b)`
`double random()`
`double rint(double a)`
`long round(double a)`
`int round(float a)`
`double sin(double a)`
`double sqrt(double a)`
`double tan(double a)`
`double toDegrees(double angrad)`
`double toRadians(double angdeg)`

Tabella 16.5
I membri della classe *Math*.

Le costanti ***E*** e ***PI*** riportano, rispettivamente, la base degli algoritmi naturali (~2.71) ed il valore pi greco (~3.14).

Le quattro varianti di ***abs()*** calcolano il valore assoluto del loro argomento in ingresso.

I metodi ***sin()***, ***cos()*** e ***tan()*** calcolano, rispettivamente, il seno, il coseno e la tangente di un numero. Mentre ***asin()***, ***acos()*** e ***atan()*** si occupano dell'arcoseno, dell'arcocoseno e dell'arcotangente. Il metodo *atan2(double y, double x)* restituisce l'angolo la cui tangente è *y / x*.

Il metodo ***exp(double a)*** restituisce *E* elevato ad *a*; ***log()*** calcola il logaritmo naturale di un numero; ***pow(double a, double b)*** calcola *a* elevato a *b*; ***sqrt()*** calcola la radice quadrata di un numero.

Le differenti varianti del metodo ***max()*** restituiscono il maggiore tra i due numeri accettati in ingresso, mentre ***min()*** restituisce il minore.

Il metodo ***ceil(double a)*** restituisce il più piccolo intero maggiore o uguale ad *a*; ***floor(double a)*** restituisce il più grande intero minore o uguale ad *a*; ***rint(double a)*** restituisce l'intero più vicono ad *a*. Nonostante questi metodi calcolino degli interi, il risultato viene consegnato sotto forma di valore *double*. Al contrario, ***round()*** approssima il suo argomento all'intero più vicino, restituendolo però come un valore *long* se l'argomento è un *double* e come un valore *int* se l'argomento è un *float*.

Il metodo ***toDegrees()*** converte in gradi un angolo espresso in radianti; ***toRadians()*** esegue la conversione inversa.

Il metodo ***random()*** restituisce un valore *double* compreso tra 0 (incluso) ed 1 (escluso), calcolato pseudo-casualmente. La seguente routine è molto utile per ottenere dei numeri interi casuali compresi tra 0 (incluso) ed *n* (escluso):

```
int casuale = (int)Math.floor(Math.random() * n);
```

16.7 - Classi wrapper

I tipi predefiniti di Java sono gestiti per valore, mentre ogni altro tipo di dato viene gestito per riferimento. Talvolta si potrebbe desiderare di gestire per riferimento un valore di tipo predefinito. Questo, senza particolari espedienti, non è possibile in Java. Per fortuna, nel package *java.lang* sono conservate delle classi che possono incapsulare (*wrap*) ciascuno dei tipi semplici. Queste classi sono dette *classi wrapper*.

- *Boolean* è la classe wrapper per il tipo *boolean*.
- *Byte* è la classe wrapper per il tipo *byte*.
- *Character* è la classe wrapper per il tipo *char*.
- *Double* è la classe wrapper per il tipo *double*.
- *Float* è la classe wrapper per il tipo *float*.
- *Integer* è la classe wrapper per il tipo *int*.
- *Long* è la classe wrapper per il tipo *long*.
- *Short* è la classe wrapper per il tipo *short*.

Grazie alle classi wrapper, è possibile gestire i tipi semplici per riferimento. La caratteristica torna molto utile quando si lavora con le collezioni di oggetti.

L'impiego delle classi wrapper è relativamente semplice. Tutte hanno un costruttore che accetta un argomento del tipo incapsulato. Quindi, *Boolean* accetterà un *boolean*, *Byte* un *byte*, *Character* un *char*, e così via. Tutte hanno un metodo nella forma *tipoValue()*, che restituisce il valore incapsulato. Per intenderci, *Boolean* ha *booleanValue()*, che restituisce il *boolean* incapsulato, *Byte* ha *byteValue()*, che restituisce il *byte* incapsulato, *Character* ha *charValue()*, che restituisce il *char* incapsulato, e così via. Al di là di tutto questo, ciascuna delle classi wrapper fornisce dei metodi che hanno utilità nel contesto del particolare tipo trattato. I più impiegati sono certamente i metodi statici della famiglia *parse*:

- ***Byte.parseByte(String s)***
- ***Double.parseDouble(String s)***
- ***Float.parseFloat(String s)***
- ***Integer.parseInt(String s)***
- ***Long.parseLong(String s)***
- ***Short.parseShort(String s)***

Tutti questi metodi hanno l'abilità di interpretare una stringa e di convertirla in un valore

numerico del tipo associato alla specifica classe. In pratica, è possibile convertire "2" in 2. Non è detto, ad ogni modo, che una stringa abbia sempre significato numerico. La stringa "ciao", ad esempio, non può essere scambiata per un numero, in nessun modo. In casi come questo, i metodi sopra elencati propagano una *java.lang.NumberFormatException*.

Si prenda in esame il seguente esempio:

```
public class Raddoppia {

  public static void main(String[] args) {
    if (args.length != 1) return;
    try {
      double d = Double.parseDouble(args[0]);
      System.out.println(d * 2);
    } catch (NumberFormatException e) {
      System.out.println("Numero non valido");
    }
  }

}
```

Questa piccola applicazione deve necessariamente ricevere un argomento da riga di comando. Il suo compito è determinare se l'argomento fornito abbia significato numerico e, in caso affermativo, raddoppiarlo. Si provi ad avviare il programma scrivendo:

```
java Raddoppia 5.234
```

Si otterrà il risultato:

```
10.468
```

Adesso si provi con:

```
java Raddoppia ciao
```

L'applicazione non potrà far altro che rispondere:

```
Numero non valido
```