Lezione 14

# Gestione delle eccezioni

Per ottenere programmi robusti ed affidabili, non è sufficiente scrivere del codice che funzioni soltanto in condizioni normali: bisogna anche saper prevedere e gestire ogni possibile situazione imprevista. Il percorso di esecuzione di un software è sempre minato dal verificarsi di condizioni eccezionali. Spazio su disco esaurito, ad esempio, rete congestionata, fonti di dati inaccessibili, permessi di sicurezza non concessi, input dell'utente formalmente scorretti, impossibilità di risolvere alcune dipendenze, e chi più ne ha più ne metta. Questa lezione esamina la gestione delle eccezioni con Java, esplorando i meccanismi utili per non farsi cogliere impreparati davanti al verificarsi di un'anomalia.

## 14.1 - Utilità della gestione delle eccezioni

Java si è guadagnato la fama di linguaggio adatto alla stesura di applicazioni robuste, anche grazie ai suoi innati meccanismi per la gestione delle situazioni impreviste. Altri linguaggi non dispongono nativamente di strumenti di questo tipo. Chi li utilizza è costretto alla stesura di lunghe e complesse routine di controllo, incastrate all'interno del codice rischioso. Immaginiamo che un'applicazione si trovi nella necessità di scrivere un file di testo su disco. Senza i meccanismi di gestione delle eccezioni, si è costretti a lavorare seguendo uno schema logico del tipo:

1. Posso scrivere il file? La periferica necessaria è connessa e pronta all'uso? Ho i permessi? C'è abbastanza spazio? In caso affermativo procedi, altrimenti interrompi ed informa l'utente.
2. Apri il canale di comunicazione per la scrittura del file.
3. Il canale è stato realmente aperto? Funziona? Può essere usato? In caso affermativo procedi, altrimenti interrompi ed informa l'utente.
4. Scrivi *n* byte nel canale di comunicazione.
5. Se hai scritto tutti i byte del file esci, altrimenti procedi.
6. Il canale è ancora attivo? Sono intervenuti problemi? Non è che la periferica è stata rimossa? Se tutto è in ordine, torna al punto 4 ed inizia un nuovo ciclo. Altrimenti interrompi ed informa l'utente.

Un codice di questo tipo presenta diversi svantaggi:

- E' facile dimenticarsi di controllare una peculiare situazione inattesa, con la conseguenza di rendere meno stabile il programma risultante.
- Il codice è prolisso.
- La routine che cura il controllo delle situazioni eccezionali e quella che cura l'effettiva scrittura del file sono fortemente accoppiate. Ciò è male. Aggiornare il software è più difficile.

Java risolve elegantemente il problema. Attraverso i meccanismi di gestione delle eccezioni, che tra poco saranno esaminati, tutta la procedura si semplifica al seguente modo:

1. Stabilisci il canale di comunicazione per la scrittura del file.
2. Scrivi il file.
3. Ci sono stati problemi durante le operazioni precedenti? In caso affermativo, esamina le informazioni sulla situazione eccezionale riscontrata, ed agisci di

conseguenza.

Ogni problema è risolto:

- Non è possibile dimenticarsi di alcune situazioni eccezionali, perché Java costringe il programmatore a prenderle tutte in considerazione, ricordandogli quali sono e quando possono accadere.
- Il codice è snello ed agile.
- I primi due punti dell'algoritmo gestiscono la scrittura del file, mentre l'ultimo serve esclusivamente al riscontro delle situazioni eccezionali. I due blocchi, pertanto, non sono accoppiati, e possono essere manipolati l'uno indipendentemente dall'altro.

## 14.2 – Il costrutto try ... catch

Java mette a disposizione due parole chiave, *try* e *catch*, indispensabili per la gestione delle eccezioni. Le situazioni inattese possono essere previste e gestiste alla seguente maniera:

```
try {
  // Codice rischioso
} catch (TipoEccezione nomeEccezione) {
  // Codice che gestisce l'eccezione
}
```

Il codice rischioso, vale a dire la sequenza di istruzioni che può incappare in situazioni eccezionali che ne compromettono il funzionamento, viene racchiuso in uno speciale blocco contrassegnato dalla parola *try*. Ad esso viene fatto seguire un secondo blocco, contraddistinto dalla parola *catch* e dalla specifica del tipo di eccezione che si intende gestire. Al suo interno viene scritto il codice capace di interpretare l'eccezione riscontrata e di agire di conseguenza (ad esempio, informando l'utente del problema, oppure cercando di eliminarne le cause per poi tentare ancora l'operazione).
Il contenuto del blocco *try* viene sempre eseguito per primo, in maniera sequenziale. Non appena una delle istruzioni contenute al suo interno causa un evento eccezionale (in gergo si dice "lancia un'eccezione" o "solleva un'eccezione") di un tipo compatibile con quello dichiarato nel blocco *catch*, l'esecuzione del blocco *try* viene definitivamente interrotta. Il controllo viene passato al contenuto del blocco *catch*, che prenderà le dovute contromisure. Il codice contenuto in un blocco *catch* dispone di una variabile di tipo *TipoEccezione*, chiamata *nomeEccezione*, che riporta tutto quello che c'è da sapere sul problema riscontrato. Terminata l'esecuzione del blocco *catch*, il programma prosegue con quello che c'è dopo l'intera struttura *try ... catch*. Se, durante l'esecuzione del blocco *try*, non viene sollevata alcuna eccezione, il blocco *catch* sarà ignorato, come se non esistesse.
Tornando all'esempio del paragrafo precedente (la scrittura di un file di testo), si elabora il seguente pseudo-codice:

```
try {
  // Apri il canale per la scrittura.
  // Scrivi il file.
} catch (EccezioneNellaCreazioneDiUnFile e) {
  // Gestisci il problema, i cui dettagli sono conservati
  // nella variabile e.
}
```

## 14.3 - Rappresentazione delle eccezioni

In questo paragrafo ci si soffermerà sullo speciale parametro di ogni blocco *catch*, che riporta i particolari della situazione anomala da gestire. L'oggetto ha duplice scopo:

1. Informa il motore di runtime su quale sia il tipo di eccezione che il blocco *catch* può gestire.
2. Fornisce al codice del blocco *catch* delle informazioni aggiuntive, che svelano i particolari del problema riscontrato.

Nel package *java.lang*, il pacchetto più basilare della piattaforma Java, c'è la classe *Throwable*. Questa classe è in cima alla gerarchia delle eccezioni. Da essa discendono due sottoclassi: *Error* ed *Exception*. Questa prima suddivisione separa in due rami distinti le situazioni anomale in cui può incappare un programma. Tutte le anomalie discendenti da *Error* sono ad uso interno, impiegate perlopiù per la gestione dei problemi relativi allo stesso ambiente di Java. Il programmatore, di norma, non prende in considerazione questo ramo. Da *Exception*, al contrario, derivano le rappresentazioni delle anomalie che solitamente un programma deve prevedere ed amministrare opportunamente. In alcuni casi non è obbligatorio gestire tutte le eccezioni riscontrabili, mentre in altri lo è. In particolare, da *Exception* deriva la classe *RuntimeException*. Tutte le eccezioni rappresentate da oggetti di tipo *RuntimeException* o da sue sottoclassi, possono non essere amministrate. Le altre anomalie, quelle che derivano direttamente da *Exception*, al contrario, devono sempre essere gestite. Su questi ultimi particolari si tornerà più avanti nel corso della lezione.
La classe *Exception*, da cui derivano tutte le eccezioni cui il programmatore dovrà necessariamente interessarsi, dispone del metodo *getMessage()*. Questo restituisce una stringa, che contiene una rappresentazione descrittiva dell'errore. Ogni sottoclasse di *Exception*, ovviamente, specializza l'anomalia, fornendo metodi aggiuntivi, sempre meno generici. Ad esempio, se qualcosa va storto durante il dialogo con un database, è possibile ottenere un codice di errore specifico che descriva il dettaglio della causa.

## 14.4 - Un semplice esempio pratico

Si prenda in esame la seguente classe:

```
public class Test {

  public static void main(String[] args) {
    int a = 5;
    int b = 0;
    int c = a / b;
    System.out.println(c);
  }

}
```

All'esecuzione del codice, si riscontra il seguente messaggio di errore:

```
java.lang.ArithmeticException: / by zero
      at Test1.main(Test.java:6)
Exception in thread "main"
```

In effetti, è stata tentata una divisione per zero, operazione chiaramente illecita. Ovviamente, nessuno andrà mai a scrivere del codice come quello mostrato nell'esempio, poiché a priori saprà che la divisione per zero non è possibile. Tuttavia, delle cause di

runtime potrebbero portare ad un tentativo del genere. Si supponga che l'intero *b*, anziché essere letteralmente inizializzato all'interno del codice, debba essere acquisito dall'esterno. Ad esempio, può accadere che la sua immissione sia richiesta all'utente. In casi come questo, è probabile che un cattivo input porti ad un'operazione non eseguibile. Il blocco del programma può essere prevenuto in due distinte maniere:

1. Controllando ogni fattore del codice rischioso.
2. Mettendo a lavoro la gestione delle eccezioni di Java.

Benché sia sempre possibile rifarsi al primo modello, con esso si ricade nella inefficiente situazione illustrata nel primo paragrafo della lezione, allorché si parlava dei linguaggi che non hanno un meccanismo intrinseco per la gestione delle eccezioni. Pertanto, il secondo modello è da preferirsi:

```
public class Test {

  public static void main(String[] args) {
    try {
      int a = 5;
      int b = 0;
      int c = a / b;
      System.out.println(c);
    } catch (ArithmeticException e) {
      System.out.println("Blocco del programma evitato!");
    }
  }

}
```

Gestendo le eccezioni di tipo *ArithmeticException*, tra cui rientra la divisione per zero, il blocco del programma è stato evitato. Ora il codice è più robusto.

### 14.5 - Più blocchi catch

Può capitare che un'operazione possa sollevare più eccezioni. Ad esempio, la lettura di un file può incappare in differenti tipi di anomalie: l'inesistenza del file, la mancanza dei permessi necessari, l'improvvisa chiusura del canale di comunicazione stabilito, e così via. Per gestire più situazioni anomale causate da un solo blocco *try*, esistono due soluzioni:

1. Realizzare un solo blocco *catch* che catturi un'eccezione di tipo generico, ad esempio proprio di tipo *Exception*. Poiché tutte le eccezioni discendono da *Exception*, ciò è sempre possibile. Dentro il blocco *catch*, poi, è possibile servirsi dell'operatore *instanceof*, per individuare l'esatta causa del problema (se c'è questa necessità), ossia l'esatto tipo dell'eccezione catturata.
2. Associare più blocchi *catch* ad un solo blocco *try*. Java prevede questa possibilità.

La seconda soluzione è solitamente da preferirsi, perché isola i differenti problemi e li gestisce l'uno indipendentemente dall'altro. Torna alla ribalta il solito discorso, sulla buona abitudine di mantenere separati i blocchi di codice che si occupano di compiti differenti. Mettere in pratica la tecnica è molto semplice:

```
try {
  // Codice rischioso
} catch (TipoEccezione1 e) {
  // Gestisci TipoEccezione1
```

```
} catch (TipoEccezione2 e) {
  // Gestisci TipoEccezione2
} catch (TipoEccezione3 e) {
  // Gestisci TipoEccezione3
}
```

Quando il codice a rischio propaga un'eccezione, i blocchi *catch* sono passati in rassegna l'uno dietro l'altro, nell'esatto ordine in cui compaiono. Non appena viene individuato un blocco capace di gestire l'eccezione, il controllo è passato al suo contenuto. Un blocco *catch* è idoneo alla gestione se l'eccezione gestita è proprio dello stesso tipo di quella propagata, ma anche quando ne è superclasse. Quindi, i differenti blocchi *catch* devono sempre essere organizzati coerentemente. Bisogna viaggiare dall'eccezione più specifica a quella più generica, mai in ordine inverso. Si prenda in considerazione la seguente classe:

```
public class Test {

  public static void main(String[] args) {
    try {
      int a = 5;
      int b = 0;
      int c = a / b;
      System.out.println(c);
    } catch (Exception e) {
      System.out.println("Exception");
    } catch (ArithmeticException e) {
      System.out.println("ArithmeticException");
    }
  }

}
```

Non è possibile compilare. Si riceve l'errore:

```
exception java.lang.ArithmeticException has already been caught
    } catch (ArithmeticException e) {
      ^
```

Java ha la creanza di informare il programmatore che è sbagliato gestire prima *Exception* e poi *ArithmeticException*. La seconda, infatti, è sottoclasse della prima. Gestendo *Exception*, è stata già gestita anche *ArithmeticException*. Java non ama il codice ridondante, quindi informa del disguido e impedisce di proseguire.

## 14.6 - Blocchi try annidati

Un'altra tecnica utile per gestire più eccezioni, valida soprattutto quando molte istruzioni partecipano al codice rischioso, consiste nell'annidare più strutture *try ... catch*, l'una dentro l'altra. Una cosa come la seguente:

```
try {
  // ...
  try {
    // ...
  } catch (Eccezione1 e) {
    // ...
  }
  // ...
```

```
} catch (Eccezione2 e) {
  // ...
}
```

Si supponga che il contenuto del secondo blocco *try*, quello più interno, propaghi un'eccezione. Se l'anomalia può essere gestita dal *catch* corrispondente, cioè se è istanza di *Eccezione1*, il controllo sarà passato proprio a questo blocco. Quindi, l'esecuzione ripartirà esattamente dopo di esso, rimanendo pertanto all'interno del blocco *try* più esterno. Al contrario, se l'eccezione non può essere gestita dal *catch* interno, ma solamente da quello esterno (*Eccezione2*), allora l'esecuzione salterà direttamente all'interno di quest'ultimo. Terminata la gestione, ci si ritroverà fuori del blocco *try* più esterno.

## 14.7 - Lanciare manualmente un'eccezione

Finora è stato visto come catturare le eccezioni. Il meccanismo fornito da Java è estendibile. In altre parole, è possibile generare nuove eccezioni, da propagare quando lo si ritene opportuno. La parola chiave per il lancio di un'eccezione è *throw*:

```
throw eccezione;
```

Un esempio:

```
public class Test {

  public static void main(String[] args) {
    try {
      // ...
      throw new Exception("sono un'eccezione lanciata manualmente");
      // ...
    } catch (Exception e) {
      System.out.println("Ho catturato un'eccezione.");
      System.out.println("Messaggio: " + e.getMessage());
    }
  }

}
```

All'interno del blocco *try*, è stata creata un'istanza di *Exception*, con il messaggio "sono un'eccezione lanciata manualmente", che poi è stata propagata con il comando *throw*. Il blocco *catch* associato cattura e gestisce l'eccezione. L'output è il seguente:

```
Ho catturato un'eccezione.
Messaggio: sono un'eccezione lanciata manualmente
```

## 14.8 - Metodi che possono lanciare delle eccezioni

La possibilità di lanciare eccezioni torna utile soprattutto quando si scrivono dei metodi che potrebbero fallire nel loro intento. Se si desidera che le eventuali anomalie siano notificate e gestite da chi richiamerà il metodo, ecco che il comando *throw* assume rilevante importanza. Ad ogni modo, affinché un metodo possa trasmettere un'eccezione a chi lo richiama, è necessario introdurre una particolare clausola nella sua dichiarazione. In caso contrario, si verrà costretti a gestire le eccezioni al suo interno. La parola chiave necessaria è *throws* (con la "s" finale, quindi da non confondere con il già visto *throw*), che si utilizza al seguente modo:

```
accesso tipo-restituito nomeMetodo(lista-argomenti) throws lista-eccezioni
```

Si prenda in esame la seguente classe:

```
public class Test {

  public static int dividi(int a, int b) throws Exception {
    int c = a / b;
    return c;
  }

  public static void main(String[] args) {
    try {
      int risultato = dividi(5, 0);
      System.out.println(risultato);
    } catch (Exception e) {
      System.out.println("Impossibile dividere");
    }
  }

}
```

Il codice contiene un metodo statico chiamato *dividi()*. Si sarebbe potuta gestire la divisione per zero direttamente al suo interno. Nel farlo, però, ci si sarebbe ritrovati davanti ad un problema: che risultato restituire quando il codice chiamante desidera una divisione per zero? L'unica soluzione coerente è che sia lo stesso codice chiamante a gestire l'anomalia. Per questo, si è fatto in modo che *dividi()* possa propagare delle eccezioni *Exception*. Il codice chiamante, a questo punto, sarà costretto a gestirle. Si provi a rimuovere il blocco *try ... catch* dal metodo *main()*:

```
public class Test {

  public static int dividi(int a, int b) throws Exception {
    int c = a / b;
    return c;
  }

  public static void main(String[] args) {
    int risultato = dividi(5, 0);
    System.out.println(risultato);
  }

}
```

La classe non può più essere compilata:

```
unreported exception java.lang.Exception; must be caught or declared to be
thrown
    int risultato = dividi(5, 0);
                    ^
```

Il metodo *dividi()* avvisa il codice chiamante della possibilità di incappare in anomalie, costringendolo alla gestione delle stesse.

Un metodo può propagare eccezioni al codice chiamante in due maniere:

1. Trasmettendo implicitamente all'indietro le eccezioni non gestite, causate dal codice

contenuto al suo interno (come nell'esempio appena visto, dove l'eccezione è causata dalla divisione).
2. Lanciandole intenzionalmente, con il comando *throw*, al verificarsi di condizioni avverse.

Un metodo può lanciare più di un tipo di eccezione. In questo caso, la sua dichiarazione va eseguita al seguente modo:

```
... nomeMetodo(lista-argomenti) throws Eccezione1, Eccezione2, ...
```

## 14.9 - La parola chiave finally

Quando le eccezioni vengono lanciate, nel codice si verifica un salto inatteso verso un blocco *catch*. Il blocco *try* che ha causato l'anomalia viene abbandonato, per non essere più ripreso. Talvolta, questo approccio comporta dei problemi. Ad esempio, se si apre un canale verso una risorsa esterna, si desidera sempre che il canale venga chiuso quando il suo utilizzo non è più necessario. Una dimostrazione è fornita dal classico esempio della lettura di un file di testo:

```
try {
  // Punto 1: apri un canale verso il file di testo.
  // Punto 2: leggi il file di testo.
  // Punto 3: chiudi il canale.
} catch (Eccezione e) {
  // Gestisci l'eccezione.
}
```

Se il punto 2 causa un'anomalia, il controllo passa al blocco *catch*. Il punto 3 non sarà mai eseguito, ed il canale di comunicazione resterà aperto più a lungo del dovuto.
Java risolve il problema servendosi dei blocchi *finally*, che si impiegano al seguente modo:

```
try {
  // ...
} catch (Eccezione1 e) {
  // ...
} catch (Eccezione2 e) {
  // ...
} catch (Eccezione3 e) {
  // ...
} finally {
  // Codice finale
}
```

Il contenuto di un blocco *finally* viene sempre eseguito, immediatamente prima che l'intera struttura *try ... catch* venga abbandonata, sia nel caso sia stata gestita un'eccezione sia in caso contrario.
La lettura di un file di testo, quindi, è più efficiente se resa alla seguente maniera:

```
// Dichiara un riferimento per il canale di comunicazione.
try {
  // Apri il canale verso il file di testo.
  // Leggi il file di testo.
} catch (Eccezione e) {
  // Gestisci l'eccezione.
} finally {
  // Chiudi il canale.
```

```
}
```

Un blocco *finally* viene sempre e comunque eseguito, anche nel caso in cui dentro il blocco *try* o dentro i blocchi *catch* vi siano istruzioni di salto che apparentemente dovrebbero impedirne l'esecuzione. Ecco un esempio:

```
public class Test {

  public static void test() {
    try {
      System.out.println("Dentro try");
      throw new Exception();
    } catch (Exception e) {
      System.out.println("Dentro catch");
      return;
    } finally {
      System.out.println("Dentro finally");
    }
  }

  public static void main(String[] args) {
    test();
  }

}
```

Il metodo *test()* contiene una struttura *try ... catch*. Dentro il blocco *try*, intenzionalmente, viene propagata un'eccezione, cosicché il controllo passi al blocco *catch*. Al suo interno viene comandato un salto, che porta al termine dell'esecuzione del metodo *test()*. Nonostante questo, il contenuto del blocco *finally* viene ugualmente valutato, subito prima che il salto comandato abbia effettivamente luogo. Infatti, l'output riscontrabile è:

```
Dentro try
Dentro catch
Dentro finally
```

Lo stesso accade quando il salto viene comandato dall'interno del blocco *try*, come nel seguente caso:

```
public class Test {

  public static void test() {
    try {
      System.out.println("Dentro try");
      return;
    } catch (Exception e) {
      System.out.println("Dentro catch");
    } finally {
      System.out.println("Dentro finally");
    }
  }

  public static void main(String[] args) {
    test();
  }

}
```

Qui non si cade mai nel blocco *catch*. In compenso, è direttamente il blocco *try* che lancia il salto *return*. Il blocco *finally* viene eseguito ugualmente, prima che il salto abbia luogo:

```
Dentro try
Dentro finally
```

Lo stesso accade con i blocchi *try* annidati l'uno dentro l'altro:

```
public class Test {

  public static void test() {
    try {
      System.out.println("Dentro try esterno");
      try {
        System.out.println("Dentro try interno");
        throw new Exception();
      } catch (ArithmeticException e) {
        System.out.println("Dentro catch interno");
      } finally {
        System.out.println("Dentro finally interno");
      }
    } catch (Exception e) {
      System.out.println("Dentro catch esterno");
    } finally {
      System.out.println("Dentro finally esterno");
    }
  }

  public static void main(String[] args) {
    test();
  }

}
```

Il blocco *try* più interno lancia un'eccezione che il blocco *catch* corrispondente non sa gestire. Quindi, il controllo del flusso passa al *catch* esterno. Di conseguenza, l'intera struttura *try ... catch* più interna viene definitivamente abbandonata, così come viene abbandonata l'esecuzione del blocco *try* più esterno. Prima che ciò accada, viene eseguito il blocco *finally* della struttura interna:

```
Dentro try esterno
Dentro try interno
Dentro finally interno
Dentro catch esterno
Dentro finally esterno
```

### 14.10 - Eccezioni personalizzate

I pacchetti di base di Java contengono un buon numero di eccezioni, tutte collegate agli specifici argomenti che fanno parte della libreria del linguaggio. Sotto *java.io*, ad esempio, ci sono delle eccezioni che riguardano la gestione dei flussi di input/output; sotto *java.net* ci sono quelle che riguardano il networking; in *java.sql* trovano posto le eccezioni relative ai database, e così via. Nonostante questo, è facile che un nuovo pacchetto o una nuova applicazione necessiti di uno specifico insieme di eccezioni progettate ad hoc. Per estendere e personalizzare il meccanismo di gestione delle eccezioni di Java, è sufficiente realizzare delle classi derivate, direttamente o indirettamente, da *Exception*:

```
public class MiaEccezione extends Exception {

  public MiaEccezione() {
    super("MiaEccezione");
  }

  public String dettagli() {
    return "Dettagli della mia eccezione";
  }

}
```

*MiaEccezione* estende *Exception*, pertanto è un'eccezione. Rispetto ad *Exception*, definisce il nuovo metodo *dettagli()*. Qualsiasi applicazione, ora, può fare uso della nuova eccezione, proprio come se fosse incorporata direttamente in Java:

```
public class Test {

  public static void test(int a) throws MiaEccezione {
    if (a == 0) throw new MiaEccezione();
  }

  public static void main(String[] args) {
    try {
      // Non lancia eccezioni.
      test(1);
      // Lancia un'eccezione di tipo MiaEccezione.
      test(0);
    } catch (MiaEccezione e) {
      System.out.println(e.dettagli());
    }
  }

}
```

## 14.11 - Le RuntimeException incorporate in Java

Java costringe alla gestione di tutte le eccezioni che possono essere propagate da un'istruzione. Solo un tipo di eccezioni sfugge a questa regola: le *RuntimeException*, come anticipato in precedenza. Le eccezioni di questo tipo non devono obbligatoriamente essere gestite, né devono essere esplicitamente dichiarate nella clausola *throws* di un metodo, affinché vengano propagate al codice chiamante. Estendendo *RuntimeException*, quindi, è possibile generare delle eccezioni a gestione facoltativa. Tuttavia, questa non è una buona pratica. Se si realizzano nuovi tipi di eccezioni, è perché si desidera che qualcuno le gestisca, non perché vengano ignorate e causino il blocco del programma. Semplicemente, *RuntimeException* è una classe ad uso interno, e le sue sottoclassi restano fondamentalmente quelle definite in *java.lang*, che rappresentano problemi di runtime facilmente evitabili. Si tratta di errori spesso banali, come la già citata divisione per zero, o l'utilizzo di indici non validi per un array. Queste eccezioni sono utili nella fase di debug del programma. Se il software si interrompe a seguito della propagazione di una *RuntimeException*, è possibile raggiungere la riga che ha causato il danno, per correggere il problema. A tal fine, è bene conoscere le principali *RuntimeException* di Java.

| ***Eccezione*** | ***Significato*** |
|---|---|
| `ArithmeticException` | Operazione matematica non valida. |

| *Eccezione* | *Significato* |
|---|---|
| `ArrayIndexOutOfBoundsException` | L'indice usato in un array non è valido. |
| `ArrayStoreException` | Incompatibilità di tipo durante l'assegnazione di un elemento di un array. |
| `ClassCastException` | Conversione di tipo non valida. |
| `IllegalArgumentException` | Argomento di un metodo non valido. |
| `IllegalMonitorStateException` | Monitor su un thread non valido. |
| `IllegalStateException` | Oggetto in uno stato che non consente l'operazione richiesta. |
| `IllegalThreadStateException` | Operazione incompatibile con lo stato attuale di un thread. |
| `IndexOutOfBoundsException` | Indice non valido. |
| `NegativeArraySizeException` | Si è tentata la creazione di un Array con dimensione negativa. |
| `NullPointerException` | Utilizzo non corretto di un valore *null*. |
| `NumberFormatException` | Conversione non valida di una stringa in un valore numerico. |
| `SecurityException` | Violazione delle norme di sicurezza. |
| `StringIndexOutOfBoundsException` | Indice non valido per i caratteri di una stringa. |
| `UnsupportedOperationException` | Operazione non supportata. |

**Tabella 14.1**
Le *RuntimeException* comprese nel pacchetto *java.lang*.

## 14.12 - Far pratica con le eccezioni

L'insieme degli strumenti deputati alla gestione delle eccezioni costituisce un altro dei punti vincenti di Java. Che se ne faccia un uso buono e frequente! Solo così le applicazioni prodotte potranno essere incrollabili. Un ottimo punto di partenza, per prendere maggiore confidenza con la gestione delle eccezioni, è sfogliare la documentazione delle API di Java. Qui si trova l'elenco di tutte le eccezioni comprese in ogni package. Non solo: la documentazione riporta anche tutte le eccezioni che un qualsiasi metodo può potenzialmente sollevare. Abituandosi all'uso dei meccanismi presenti all'interno della libreria di Java, si acquisisce automaticamente un modello di sviluppo efficace, per i propri pacchetti e per le proprie applicazioni.