10. Gestione delle Eccezioni e Input/Output in Java

Questo capitolo esplora i meccanismi di gestione delle eccezioni in Java, fondamentali per la robustezza del software, e le API per la gestione dell'Input/Output (I/O), essenziali per l'interazione con risorse esterne come file e console.

4.1. Gestione delle Eccezioni

La gestione delle eccezioni è un meccanismo cruciale in Java per affrontare condizioni anomale (errori a run-time) che possono verificarsi durante l'esecuzione di un programma.

4.1.1. Errori nei Programmi: Compile-time vs. Run-time

È importante distinguere tra i tipi di errori:

- **Errori a tempo di compilazione (Compile-time)**: Sono errori **grossolani** intercettati dal compilatore (es. errori di **sintassi**, errori di **tipo**). Rientrano nella fase di implementazione e sono **innocui**, poiché impediscono al programma di essere compilato ed eseguito. Un linguaggio con *strong typing* (tipizzazione forte) come Java consente di identificarne molti a compile-time.
- **Errori a tempo di esecuzione (Run-time)**: Sono **condizioni anomale** che si verificano durante l'esecuzione del programma, dovute alla dinamica del sistema. Esempi includono parametri anomali a funzioni, errori nell'uso delle risorse di sistema (es. **file non trovato**, connessione di rete persa), o tentativi di accedere a un indice fuori dai limiti di un array. Questi errori sono l'oggetto della gestione delle eccezioni.

4.1.2. Eccezioni: Concetto e Tipi in Java

Un'**eccezione** è un evento che interrompe il normale flusso di esecuzione di un programma. In Java, le eccezioni sono oggetti che rappresentano queste condizioni anomale.

La gerarchia delle eccezioni in Java è radicata nella classe `java.lang.Throwable`:

```
java.lang.Object
  └─ java.lang.Throwable
      ├─ java.lang.Error
      └─ java.lang.Exception
          └─ java.lang.RuntimeException
```

Questa gerarchia si divide in due rami principali:

1. `Error`: Rappresenta problemi gravi da cui un'applicazione tipicamente non dovrebbe cercare di riprendersi (es. `OutOfMemoryError`, `StackOverflowError`). Sono errori del sistema JVM e non sono destinati a essere catturati dal codice dell'applicazione.
2. `Exception`: Rappresenta condizioni che un'applicazione può voler catturare e gestire. Si suddividono ulteriormente in:
   - **Checked Exceptions (Eccezioni Controllate)**: Sottoclassi dirette di `Exception` (escluse `RuntimeException`). Il compilatore *forza* il programmatore a gestirle (catturandole con `try-catch` o dichiarandole con `throws`). Se non gestite, il codice non compila. Esempi comuni includono `IOException`, `SQLException`, `ClassNotFoundException`.
   - **Unchecked Exceptions (Eccezioni Non Controllate)**: Sottoclassi di `RuntimeException` (es. `NullPointerException`, `ArrayIndexOutOfBoundsException`, `NumberFormatException`, `ClassCastException`). Il compilatore *non forza* la loro gestione. Sono generalmente indicative di errori di programmazione (bug) e **dovrebbero essere prevenute piuttosto che catturate**

4.1.3. Il Costrutto `try-catch-finally`

Il blocco `try-catch-finally` è il meccanismo principale per la gestione delle eccezioni in Java.

- `try`: Contiene il codice che potrebbe generare un'eccezione.
- `catch`: Segue il blocco `try` e specifica il tipo di eccezione che può essere catturato e il codice da eseguire per gestirla. Possono esserci più blocchi `catch` per gestire diversi tipi di eccezioni. L'ordine dei blocchi `catch` è importante: le eccezioni più specifiche devono essere catturate prima di quelle più generiche.

- `finally`: (Opzionale) Contiene il codice che viene sempre eseguito, indipendentemente dal fatto che un'eccezione sia stata lanciata o meno nel blocco `try`, o che sia stata catturata. È utile per rilasciare risorse (es. chiudere file, connessioni di database).

Sintassi:

```
try {
    // Codice che potrebbe lanciare un'eccezione
} catch (TipoEccezione1 e1) {
    // Gestione dell'eccezione di TipoEccezione1
} catch (TipoEccezione2 e2) {
    // Gestione dell'eccezione di TipoEccezione2
} finally {
    // Codice che viene sempre eseguito
}
```

Esempio:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class IOFromKeyboard2 {
    private static final BufferedReader KBD = new BufferedReader(new InputStreamReader(System.in));

    // Metodo che lancia IOException (checked exception) e NumberFormatException (unchecked exception)
    private static int getIntFromKbd() throws IOException {
        return Integer.parseInt(KBD.readLine());
    }

    public static void main(String[] args) {
        try {
            System.out.print("Inserisci un numero: ");
            final int a = getIntFromKbd();
            System.out.println("Hai inserito il num.: " + a);
        } catch (IOException e) { // Cattura IOException
            System.out.println("Errore di I/O: " + e);
        } catch (NumberFormatException e) { // Cattura NumberFormatException
            System.out.println("Input non valido: " + e.getMessage());
        } finally {
            System.out.println("Operazione completata.");
            // Qui si potrebbe chiudere KBD se fosse una risorsa da gestire
        }
    }
}
```

4.1.4. Dichiarazione `throws`

Se un metodo può lanciare una checked exception ma non la gestisce internamente, deve dichiararla nella sua firma usando la keyword `throws`. Questo avvisa i chiamanti del metodo che devono gestire quella specifica eccezione.

Sintassi: `public void myMethod() throws IOException, SQLException { ... }`

4.1.5. Lancio Manuale di Eccezioni (`throw`)

È possibile lanciare manualmente un'eccezione utilizzando la keyword `throw` seguita da un'istanza di `Throwable`.

Sintassi: `throw new MyCustomException("Messaggio di errore");`

4.1.6. `try-with-resources` (Java 7+)

Introdotto in Java 7, il costrutto `try-with-resources` è un modo più pulito ed efficiente per gestire risorse che devono essere chiuse (es. `InputStream`, `OutputStream`, `Connection`). Qualsiasi risorsa dichiarata nel blocco `try` deve implementare l'interfaccia `java.lang.AutoCloseable`.

Il vantaggio è che la risorsa viene **automaticamente chiusa** alla fine del blocco `try`, anche in caso di eccezioni, eliminando la necessità di un blocco `finally` esplicito per la chiusura.

Esempio:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TryWithResourcesExample {
    public static void main(String[] args) {
        String line;
        try (BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            System.err.println("Errore durante la lettura del file: " + e.getMessage());
        }
    }
}
```

In questo esempio, `br` **viene automaticamente chiuso** alla fine del blocco `try`.

4.2. Input/Output (I/O) in Java

La libreria `java.io` fornisce un ricco set di classi per gestire le operazioni di input e output, permettendo ai programmi di interagire con diverse fonti e destinazioni di dati.

4.2.1. Il Problema dell'Input/Output

L'I/O è un problema fondamentale per i sistemi operativi, che gestiscono le comunicazioni tra la CPU e i dispositivi affacciati sul BUS (console, tastiera, mouse, dischi, rete, sensori, schermo). Esistono varie modalità di interazione (sequenziale, random-access, buffered, per carattere/linea/byte/oggetto). La libreria `java.io` fornisce i concetti di `File` e `Stream` di dati, consentendo una gestione flessibile.

4.2.2. Classi per Gestire File (`java.io.File`)

La classe `java.io.File` non gestisce direttamente il contenuto di un file, ma **rappresenta il file o una directory nel file system**. Permette di eseguire operazioni sui metadati del file.

Metodi principali:

- `boolean exists()`: Verifica se il file o la directory esiste.
- `boolean isDirectory()`: Verifica se è una directory.
- `boolean isFile()`: Verifica se è un file.
- `String getName()`: Restituisce il nome del file/directory.
- `String getPath()`: Restituisce il percorso (relativo o assoluto).
- `String getAbsolutePath()`: Restituisce il percorso assoluto.
- `boolean mkdirs()`: Crea la directory (e le directory padre se non esistono).
- `boolean delete()`: Elimina il file/directory.
- `File[] listFiles()`: Restituisce un array di oggetti `File` che rappresentano i contenuti di una directory.
- `long length()`: Restituisce la dimensione del file in byte.

Esempio:

```
import java.io.File;

public class UseFile {
    public static void main(String[] args) {
        // Creazione di un oggetto File che rappresenta un file esistente
        File file = new File("file.txt");

        if (file.exists()) {
            System.out.println("Nome: " + file.getName());
            System.out.println("Percorso assoluto: " + file.getAbsolutePath());
            System.out.println("È un file? " + file.isFile());
            System.out.println("Dimensione: " + file.length() + " bytes");
        } else {
            System.out.println("Il file non esiste.");
            // Creazione di una directory
            File dir = new File("nuova_cartella");
            if (dir.mkdir()) {
                System.out.println("Cartella 'nuova_cartella' creata.");
            }
        }
    }
}
```

4.2.3. Stream di Input/Output

Gli stream sono il concetto fondamentale per leggere e scrivere dati. Rappresentano un flusso di dati da una sorgente a una destinazione.

- `InputStream` / `OutputStream`: Classi astratte per leggere/scrivere byte. Sono la base per l'I/O binario.
- `Reader` / `Writer`: Classi astratte per leggere/scrivere caratteri. Sono la base per l'I/O testuale e gestiscono la codifica dei caratteri.

4.2.3.1. Stream di Byte

- `FileInputStream` / `FileOutputStream`: Per leggere/scrivere byte da/verso file.
- `ByteArrayInputStream` / `ByteArrayOutputStream`: Per leggere/scrivere byte da/verso array in memoria.
- `BufferedInputStream` / `BufferedOutputStream`: Aggiungono funzionalità di buffering per migliorare le prestazioni, riducendo le interazioni dirette con il dispositivo.

4.2.3.2. Stream di Caratteri

- `FileReader` / `FileWriter`: Per leggere/scrivere caratteri da/verso file.
- `BufferedReader` / `BufferedWriter`: Aggiungono funzionalità di buffering per migliorare le prestazioni e metodi per leggere/scrivere linee intere (`readLine()`, `newLine()`).
- `InputStreamReader` / `OutputStreamWriter`: Ponti tra stream di byte e stream di caratteri, specificando la codifica dei caratteri.

4.2.4. Il Pattern Decorator nell'I/O Java

Le classi di I/O in Java fanno ampio uso del **Pattern Decorator** (strutturale, su oggetti). Questo pattern permette di **aggiungere dinamicamente nuove responsabilità a un oggetto**, "decorandolo" con funzionalità aggiuntive.

- **Idea**: Un oggetto "decoratore" incapsula un altro oggetto (il "componente") e aggiunge funzionalità, mantenendo la stessa interfaccia del componente. Questo permette di combinare funzionalità in modo flessibile.
- **Applicazione nell'I/O**: Le classi base (`FileInputStream`, `FileReader`) forniscono funzionalità di I/O grezze. I "decoratori" (es. `BufferedInputStream`, `BufferedReader`, `DataInputStream`) aggiungono funzionalità come il buffering, la lettura di tipi primitivi, ecc.

Esempio:
Per leggere linee di testo da un file in modo efficiente:

`new BufferedReader(new FileReader("file.txt"))`
Qui, `BufferedReader` decora `FileReader`, aggiungendo il buffering e la capacità di leggere linee.

4.2.5. Esempi Pratici di I/O

4.2.5.1. Lettura da Tastiera (Console)

Storicamente, la lettura da tastiera in Java era complessa a causa della gestione delle eccezioni.

Esempio con `BufferedReader` (vecchio stile):

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class IOFromKeyboard {
    public static void main(String[] args) {
        BufferedReader KBD = new BufferedReader(new InputStreamReader(System.in));
        try {
            System.out.print("Inserisci il tuo nome: ");
            String name = KBD.readLine();
            System.out.println("Ciao, " + name + "!");
        } catch (IOException e) {
            System.err.println("Errore di I/O: " + e.getMessage());
        } finally {
            try {
                if (KBD != null) KBD.close(); // Chiudere la risorsa
            } catch (IOException ex) {
                System.err.println("Errore nella chiusura: " + ex.getMessage());
            }
        }
    }
}
```

Alternativa moderna con `Scanner`:
La classe `java.util.Scanner` (non parte di `java.io` ma utile per l'I/O) semplifica la lettura di input formattato.

```
import java.util.Scanner;

public class IOWithScanner {
    public static void main(String[] args) {
        System.out.print("Inserisci un numero intero: ");
        Scanner scanner = new Scanner(System.in);
        if (scanner.hasNextInt()) {
            int num = scanner.nextInt();
            System.out.println("Hai inserito: " + num);
        } else {
            System.out.println("Input non valido. Non è un numero intero.");
        }
        scanner.close(); // È buona pratica chiudere lo Scanner
    }
}
```

4.2.5.2. Gestione di File di Testo

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
```

```
import java.io.IOException;
import java.util.stream.Collectors; // Per Java 8+

public class TextFileOperations {

    // Scrive testo su un file
    public static void writeToFile(String filename, String content) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter(filename))) {
            writer.write(content);
            System.out.println("Contenuto scritto su " + filename);
        } catch (IOException e) {
            System.err.println("Errore durante la scrittura su file: " + e.getMessage());
        }
    }

    // Legge testo da un file
    public static String readFromFile(String filename) {
        StringBuilder sb = new StringBuilder();
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = reader.readLine()) != null) {
                sb.append(line).append(System.lineSeparator());
            }
        } catch (IOException e) {
            System.err.println("Errore durante la lettura da file: " + e.getMessage());
        }
        return sb.toString();
    }

    // Legge testo da un file usando Stream API (Java 8+)
    public static String readFromFileWithStreams(String filename) {
        try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
            return reader.lines().collect(Collectors.joining(System.lineSeparator()));
        } catch (IOException e) {
            System.err.println("Errore durante la lettura da file con Streams: " + e.getMessage());
            return "";
        }
    }

    public static void main(String[] args) {
        String testFile = "esempio.txt";
        String content = "Questa è la prima riga.\nQuesta è la seconda riga.";

        writeToFile(testFile, content);
        System.out.println("\nContenuto letto:\n" + readFromFile(testFile));
        System.out.println("\nContenuto letto con Streams:\n" + readFromFileWithStreams(testFile));
    }
}
```

4.2.6. Serializzazione di Oggetti

La serializzazione è il processo di **conversione dello stato di un oggetto in un flusso di byte**, che può essere poi salvato su file o trasmesso attraverso una rete. La deserializzazione è il processo inverso.

- **Interfaccia** `Serializable`: Una classe deve implementare questa interfaccia (un'interfaccia "marker", senza metodi) per poter essere serializzata.
- `ObjectOutputStream` / `ObjectInputStream`: Utilizzati per scrivere/leggere oggetti serializzati.

Esempio:

```
import java.io.*;

// La classe Person deve essere Serializable
class Person implements Serializable {
    private static final long serialVersionUID = 1L; // Per compatibilità di versione
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + "]";
    }
}

public class ObjectSerialization {
    public static void main(String[] args) {
        Person p1 = new Person("Alice", 30);
        String filename = "person.ser";

        // Serializzazione
        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(filename))) {
            oos.writeObject(p1);
            System.out.println("Oggetto Person serializzato con successo.");
        } catch (IOException e) {
            System.err.println("Errore di serializzazione: " + e.getMessage());
        }

        // Deserializzazione
        try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(filename))) {
            Person p2 = (Person) ois.readObject(); // Necessario cast
            System.out.println("Oggetto Person deserializzato: " + p2);
        } catch (IOException | ClassNotFoundException e) {
            System.err.println("Errore di deserializzazione: " + e.getMessage());
        }
    }
}
```

4.2.7. Formati Standard per la Serializzazione (JSON, YAML)

Oltre alla serializzazione nativa di Java, esistono formati standard basati su testo per la serializzazione di oggetti, che offrono maggiore leggibilità e interoperabilità:

- **JSON (JavaScript Object Notation)**: Nato in seno a JavaScript, è molto usato in ambito web. Librerie Java come Jackson e Gson sono popolari per la gestione di JSON.
- **YAML (YAML Ain't Markup Language)**: Un superset di JSON dalla versione 1.2, con supporto per funzioni avanzate (es. anchoring). Molto usato per file di configurazione complessi.

I formati testuali sono generalmente preferiti quando le prestazioni e lo spazio non sono stringenti, grazie alla loro leggibilità.

4.2.8. I/O con la Nuova API `java.nio.file` (NIO.2 - Java 7+)

Java 7 ha introdotto la "New I/O 2" (NIO.2) nel package `java.nio.file`, che offre un'API più moderna, robusta ed efficiente per la gestione dei file e delle directory.

- `Path`: Rappresenta un percorso nel file system. Sostituisce parzialmente `java.io.File`.
- `Files`: Classe di utilità con metodi statici per operazioni comuni su file e directory (copia, spostamento, eliminazione, lettura/scrittura di tutte le linee, ecc.).
- `DirectoryStream`: Per iterare sui contenuti di una directory in modo efficiente.

Vantaggi di NIO.2:

- Migliore gestione degli errori.
- Supporto per link simbolici e attributi di file.
- API più orientata agli oggetti e più facile da usare per operazioni complesse.

Esempio:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.util.List;

public class NIO2Example {
    public static void main(String[] args) {
        Path filePath = Paths.get("nuovo_file_nio.txt");

        // Scrivere su un file
        try {
            Files.writeString(filePath, "Ciao da NIO.2!\nQuesta è una nuova riga.", StandardOpenOption.CREATE, StandardO
penOption.TRUNCATE_EXISTING);
            System.out.println("Scritto su " + filePath.toAbsolutePath());
        } catch (IOException e) {
            System.err.println("Errore di scrittura con NIO.2: " + e.getMessage());
        }

        // Leggere da un file
        try {
            List<String> lines = Files.readAllLines(filePath);
            System.out.println("\nContenuto letto con NIO.2:");
            lines.forEach(System.out::println);
        } catch (IOException e) {
            System.err.println("Errore di lettura con NIO.2: " + e.getMessage());
        }

        // Copiare un file
        Path copiedFilePath = Paths.get("copia_file_nio.txt");
        try {
            Files.copy(filePath, copiedFilePath, java.nio.file.StandardCopyOption.REPLACE_EXISTING);
            System.out.println("\nFile copiato in " + copiedFilePath.toAbsolutePath());
        } catch (IOException e) {
            System.err.println("Errore di copia con NIO.2: " + e.getMessage());
        }

        // Eliminare un file
        try {
            Files.deleteIfExists(copiedFilePath);
            System.out.println("\nFile " + copiedFilePath.getFileName() + " eliminato.");
```

```
        } catch (IOException e) {
            System.err.println("Errore di eliminazione con NIO.2: " + e.getMessage());
        }
    }
}
```