

Java SE: Eccezioni, I/O ...

- Eccezioni
- Date e Time
 - Da java.util a java.time
- Input / Output
 - PrintWriter
 - Scanner
- Try with resources
- Progetto di riferimento
 - <https://github.com/egalli64/mpjp> (*modulo 3*)

Eccezioni

- Obbligano il chiamante a gestire gli errori
 - Unhandled exception → terminazione del programma
- Evidenziano il flusso normale di esecuzione
- Semplificano il debug esplicitando lo stack trace
- Possono chiarire il motivo scatenante dell'errore
 - NullPointerException, ArrayIndexOutOfBoundsException, ...
- Checked vs unchecked



try – catch – finally

- **try**: esecuzione protetta
- **catch**: gestisce uno o più possibili eccezioni
- **finally**: sempre eseguito, alla fine del try o dell'eventuale catch
- Ad un blocco try deve seguire almeno un blocco catch o finally
- **“throws”** nella signature
 “throw” per “tirare” una eccezione.

```
public void f() {  
    try {  
        g();  
    } catch (Exception ex) {  
        // ...  
    } finally {  
        cleanup();  
    }  
}  
  
// ...  
  
public void g() throws Exception {  
    // ...  
    if (somethingUnexpected()) {  
        throw new Exception();  
    }  
}
```

Gerarchia delle eccezioni



Test eccezioni in JUnit 3

Math.abs() di
Integer.MIN_VALUE
è
Integer.MIN_VALUE!

```
public int negate(int value) {  
    if(value == Integer.MIN_VALUE) {  
        throw new IllegalArgumentException("Can't negate MIN_VALUE");  
    }  
    return -value;  
}
```

```
@Test  
void negateException() {  
    Simple simple = new Simple();  
  
    try {  
        simple.negate(Integer.MIN_VALUE);  
    } catch (IllegalArgumentException iae) {  
        String message = iae.getMessage();  
        assertEquals("Can't negate MIN_VALUE", message);  
        return;  
    }  
    fail("An IllegalArgumentException was expected");  
}
```

JUnit 4.7 ExpectedException

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void negateMinInt() {
    thrown.expect(IllegalArgumentException.class);
    thrown.expectMessage("Can't negate MIN_VALUE");

    Simple simple = new Simple();
    sample.negate(Integer.MIN_VALUE);
}
```

Nel @Test
si dichiara
quale eccezione
e messaggio
ci si aspetta

Si definisce una
variabile di istanza
ExpectedException
taggata come @Rule

JUnit 5 assertThrows()

Il metodo fallisce se quanto testato non tira l'eccezione specificata

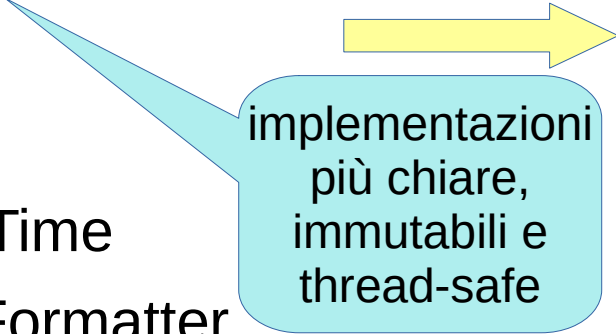
L'eccezione attesa viene tornata per permettere ulteriori test

```
@Test
public void negateMinInt() {
    IllegalArgumentException exc = assertThrows(IllegalArgumentException.class,
        () -> simple.negate(Integer.MIN_VALUE));
    assertThat(exc.getMessage(), is("Can't negate MIN_VALUE"));
}
```

L'assertion è eseguita su di un Executable, interfaccia funzionale definita in Jupiter

Date e Time

- java.util
 - Date
 - DateFormat
 - Calendar
 - GregorianCalendar
 - TimeZone
 - SimpleTimeZone
- java.time (JDK 8)
 - LocalDate
 - LocalTime
 - LocalDateTime
 - DateTimeFormatter, FormatStyle
 - Instant, Duration, Period
- java.sql.Date



implementazioni
più chiare,
immutabili e
thread-safe

LocalDate e LocalTime

- Non hanno costruttori pubblici
- Static factory methods: now(), of()
- Formattazione via DateTimeFormatter con FormatStyle
- LocalDateTime aggrega LocalDate e LocalTime

```
LocalDate date = LocalDate.now();
System.out.println(date);
System.out.println(LocalDate.of(2019, Month.JUNE, 2));
System.out.println(LocalDate.of(2019, 6, 2));
System.out.println(date.format(DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));

LocalTime time = LocalTime.now();
System.out.println(time);

LocalDateTime ldt = LocalDateTime.of(date, time);
System.out.println(ldt);
```

java.sql Date, Time, Timestamp

- Supporto JDBC a date/time SQL
 - Date, Time, Timestamp
- Conversioni
 - *.valueOf(Local*)
 - Date.toLocalDate()
 - Time.toLocalTime()
 - Timestamp.toLocalDateTime()
 - Timestamp.toInstant()

La libreria java.io

- Supporto a operazioni di input e output
- In un programma solitamente i dati sono
 - Letti da sorgenti di input
 - Scritti su destinazioni di output
- Basata sul concetto di stream
 - Flusso sequenziale di dati
 - binari (byte)
 - testuali (char)
 - Aperto in lettura o scrittura prima dell'uso, va esplicitamente chiuso al termine
 - Astrazione di sorgenti/destinazioni (connessioni di rete, buffer in memoria, file su disco ...)



File

- Accesso a file e directory
- I suoi quattro costruttori
 - `File dir = new File("/tmp");`
 - `File f1 = new File("/tmp/hello.txt");`
 - `File f2 = new File("/tmp", "hello.txt");`
 - `File f3 = new File(dir, "hello.txt");`
 - `File f4 = new File(new URI("file:///C://tmp/hello.txt"));`
- Creazione di una directory e di un file su memoria di massa
 - `dir.mkdir()`
 - `f1.createNewFile()`

Forward slash anche per Windows

Check File

- exists()
- isFile()
- isDirectory()
- isHidden()
- canRead()
- canWrite()
- canExecute()
- isAbsolute()

Alcuni altri metodi di File

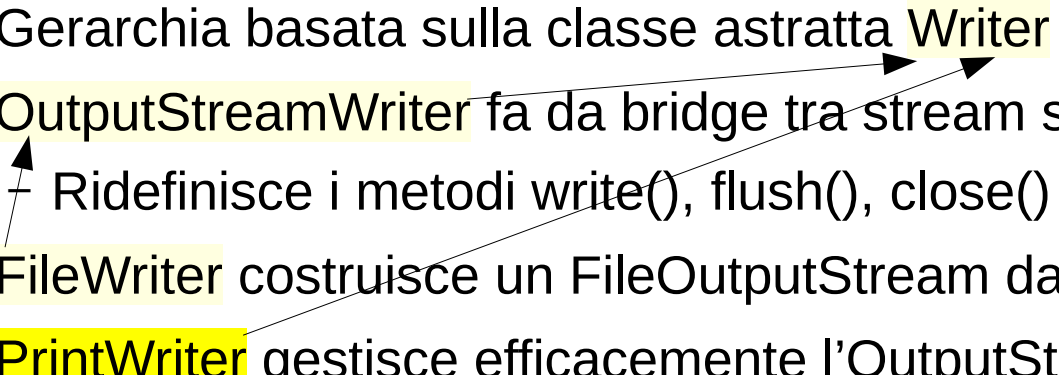
- `getName()` // "hello.txt"
- `getPath()` // "\\tmp\\hello.txt"
- `getAbsolutePath()` // "D:\\tmp\\hello.txt"
- `getParent()` // "\\tmp"
- `lastModified()` // 1559331488083L
- `length()` // 4L
- `list()` // ["hello.txt"]

usa separatore (File.separator)
e formato del SO corrente

UNIX time in milliseconds

se invocato su una directory:
array dei nomi dei file contenuti

Scrittura in un file di testo

- Gerarchia basata sulla classe astratta **Writer**
 - **OutputStreamWriter** fa da bridge tra stream su caratteri e byte
 - Ridefinisce i metodi `write()`, `flush()`, `close()`
 - **FileWriter** costruisce un `FileOutputStream` da un `File` (o dal suo nome)
 - **PrintWriter** gestisce efficacemente l'`OutputStream` passato con i metodi `print()`, `println()`, `printf()`, `append()`
- 

```
File f = new File("/tmp/hello.txt");
PrintWriter pw = new PrintWriter(new FileWriter(f));
pw.println("hello");
pw.flush();
pw.close();
```

Lettura da un file di testo

- Gerarchia basata sulla classe astratta **Reader**
- **InputStreamReader** fa da bridge tra stream su caratteri e byte
 - Ridefinisce i metodi `read()` e `close()`
- **FileReader** costruisce un `FileInputStream` da un `File` (o dal suo nome)
- **BufferedReader** gestisce efficacemente l'`InputStream` passato con un buffer e fornendo metodi come `readLine()`

```
File f = new File("/tmp/hello.txt");  
BufferedReader br = new BufferedReader(new FileReader(f));  
String line = br.readLine();  
br.close();
```


Input con Scanner

- Legge input formattato con funzionalità per convertirlo anche in formato binario
- Può leggere da input Stream, File, String, o altre classi che implementano Readable o ReadableByteChannel
- Uso generale di Scanner:
 - Il ctor associa l'oggetto scanner allo stream in lettura
 - Loop su `hasNext...()` per determinare se c'è un token in lettura del tipo atteso
 - Con `next...()` si legge il token
 - Terminato l'uso, ricordarsi di invocare `close()` sullo scanner

Un esempio per Scanner

```
import java.util.Scanner;

public class Adder {
    public static void main(String[] args) {
        System.out.println("Please, enter a few numbers");
        double result = 0;

        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNext()) {
            if (scanner.hasNextDouble()) {
                result += scanner.nextDouble();
            } else {
                System.out.println("Bad input, discarded: " + scanner.next());
            }
        }
        scanner.close(); // see try-with-resources
        System.out.println("Total is " + result);
    }
}
```

try-with-resources

Per classi che implementano **AutoCloseable**

```
double result = 0;

// try-with-resources
try(Scanner scanner = new Scanner(System.in)) {
    while (scanner.hasNext()) {
        if (scanner.hasNextDouble()) {
            result += scanner.nextDouble();
        } else {
            System.out.println("Bad input, discarded: " + scanner.next());
        }
    }
}

System.out.println("Total is " + result);
```