12
Errori di esecuzione ed Exceptions

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2025/2026

Outline

Goal della lezione

- Illustrare i vari meccanismi di gestione delle eccezioni in Java
- Dare linee guida per la progrettazione di sistemi che usano eccezioni

Argomenti

- Errori a run-time e necessità di una loro gestione
- Tipi di eccezioni/errori in Java
- Istruzione `throw`
- Costrutto `try-catch-finally`
- Dichiarazioni `throws`

Outline

Errori nei programmi

Errori a tempo di compilazione (compile-time)

- sono quelli più grossolani, sono intercettati dal compilatore
- quindi rientrano nella fase dell'implementazione, sono innocui
- un linguaggio con strong typing consente di identificarne molti a compile-time
- editor moderni spesso li rendono "edit-time"
- non sono considerabili "bug", ma situazioni temporanee nello sviluppo

Errori a tempo di esecuzione (run-time) ($\Leftarrow$ oggetto della lezione)

- sono condizioni anomale dovute alla dinamica del sistema
  - parametri anomali a funzioni, errori nell'uso delle risorse di sistema,..
- in genere è possibile (i) identificare/descrivere dove potrebbero accadere, (ii) intercettarli e (iii) gestirli prevedendo procedure di compensazione (rimedio al problema che le ha causate)
- alcuni linguaggi (come Java/C#, non il C) forniscono costrutti di alto livello per agevolarne la gestione
- nella programmazione moderna li si cerca di associare a bug

Errori per causa interna: lanciati dalla JVM

Errore numerico

```
1 int divide(int x, int y){ return x/y; }
2 ...
3 int z = divide(5,0);
4 // ERRORE: divisione per 0
```

Overflow memoria

```
1 int f(int i){ return i==0 ? 0 : f(i+1); }
2 ...
3 int n=f(1);
4 // ERRORE: out of (stack) memory
```

Riferimento null

```
1 int mysize(List<?> l){ return l.size(); }
2 ...
3 int n=mysize(null);
4 // ERRORE: invocazione metodo size() su null
```

Violazioni del contratto d'uso di un oggetto: librerie Java

Operazione non supportata

```
1 Collections.<Integer>emptySet().add(1);
2 // UnsupportedOperationException
3 /* ERRORE: emptySet() torna un Set immutabile
4    deve essere impedita l'invocazione di add() */
```

Elemento non disponibile

```
1 Iterator<Integer> i = Arrays.asList(1,2).iterator();
2 i.next();
3 i.next();
4 i.next(); // NoSuchElementException
5 /* ERRORE: il contratto d'uso degli Iterator prevede di non
6    invocare next() se hasNext() dà false */
```

Formato illegale

```
1 Integer.parseInt("10.4");
2 // NumberFormatException
3 /* ERRORE: parseInt() si aspetta una stringa che contenga,
4    carattere per carattere, un intero valido */
```

Violazioni del contratto d'uso di un oggetto: nostro codice

Argomento errato

```
1 public class LampsRow{
2     private SimpleLamp[] row;
3     public LampsRow(int size){
4       if (size<0) {throw ???;} // lancio eccezione
5       this.row = new SimpleLamp[size];
6     }
7     ..
```

Elemento non disponibile

```
1 public class RangeIterator implements Iterator<Integer>{
2     private int current;
3     private int stop;
4
5     public Integer next(){
6       if (current > stop) { throw ???} // lancio eccezione
7       return this.current++;
8     }
9     ..
```

L'importanza della "error-aware programming"

Contratti

- Molti oggetti richiedono determinate condizioni di lavoro (sequenze di chiamata, argomenti passati, aspettative d'uso di risorse computazionali)
- Al di fuori queste condizioni è necessario interrompere il lavoro ed effettuare azioni correttive

Il progettista della classe deve:

1. identificare le condizioni di lavoro definite "normali"
2. intercettare quando si esce da tali condizioni
3. eventualmente segnalare l'avvenuto errore

Il cliente (a sua volta progettista di un altro oggetto) deve:

1. essere informato di come l'oggetto va usato
2. intercettare gli errori e porvi rimedio con un *handler*

Eccezioni in Java

Riassunto Java Exceptions

- Gli errori a run-time in Java sono rappresentati da oggetti della classe `java.lang.Throwable`. Vengono "lanciati":
  - da esplicite istruzioni del tipo: `throw <exception-object>;`
  - o, direttamente dalla JVM per cause legate al "sistema operativo"
- Tali oggetti portano informazioni utili a capire la causa dell'errore
- Si può dichiarare se un metodo potrà lanciare una eccezione:
  `<meth-signature> throws <excep-class>{..}`
- Si può intercettare una eccezione e porvi rimedio:
  `try{ <instructions> } catch(<excep-class> <var>){...}`

Tutti meccanismi che impareremo a progettare e implementare in questa lezione!

Tipologie di errori in Java

Errori: `java.lang.Error` e sottoclassi

- Dovute a un problema "serio" (e non risolvibile) interno alla JVM
- Di norma una applicazione non si deve preoccupare di intercettarli (non ci sarebbe molto di più da fare che interrompere l'esecuzione)

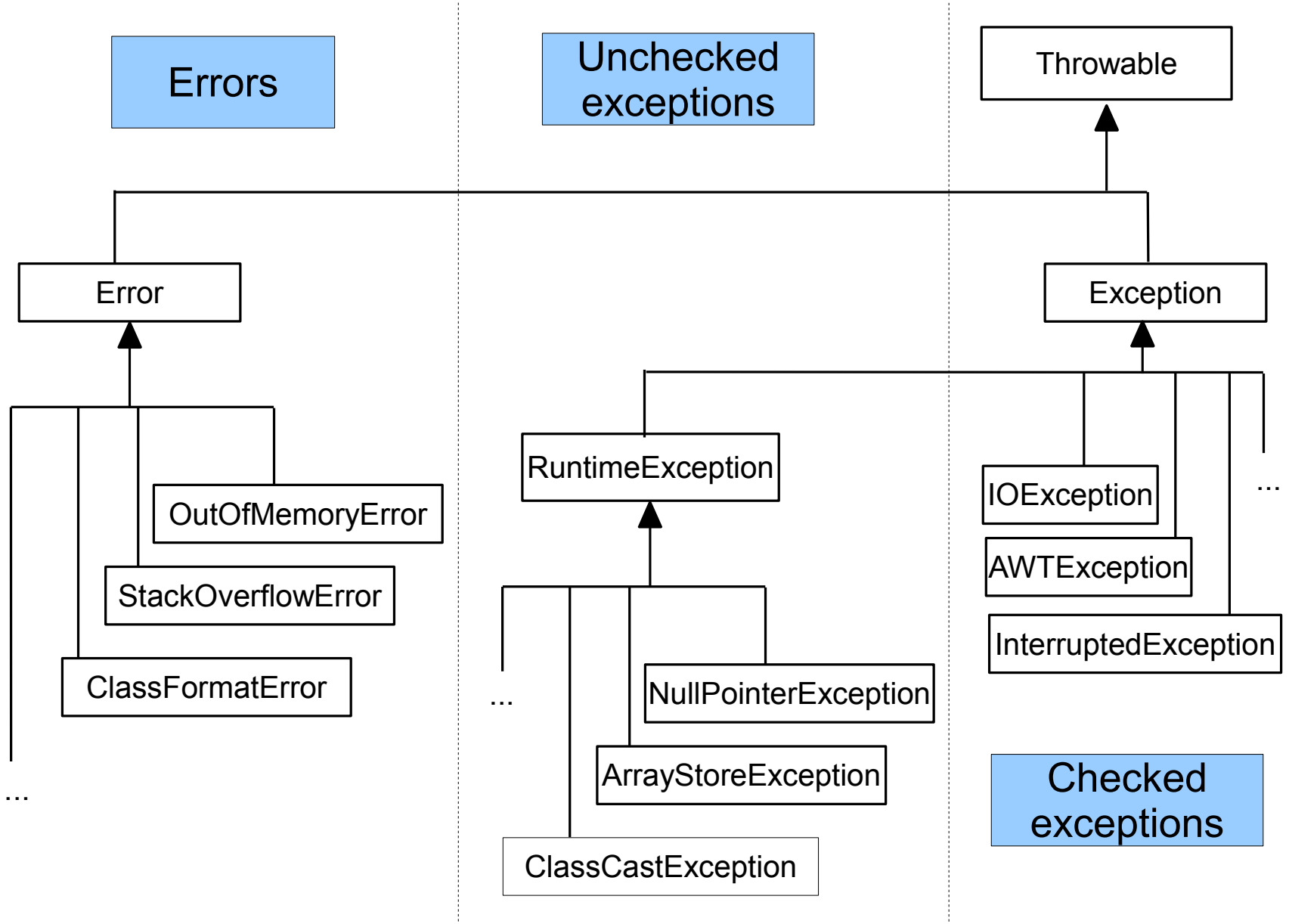
Eccezioni unchecked: `java.lang.RuntimeException` e sottoclassi

- Causate da un bug nella programmazione
- Di norma una applicazione non si deve preoccupare di intercettarli (dovrebbero essere risolti tutti in fase di debugging del sistema)

Eccezioni checked: i `java.lang.Throwable` tranne le precedenti

- Causate da un problema prevedibile ma non rimediabile a priori
- Le applicazione devono dichiarli esplicitamente, e vanno intercettati e gestiti esplicitamente

Tipologie di errori in Java: UML



Usuale gestione

Errori

- Nessuna gestione necessaria ("se capitano, capitano...")

Eccezioni unchecked

- Si potrebbero dichiarare con un commento al codice
- Di norma si riusano le classi `java.lang.RuntimeException` del JDK, ossia non se ne definiscono di nuove tipologie
- Si lanciano con l'istruzione `throw`

Eccezioni checked

- Vanno dichiarate nel metodo con la clausola `throws`
- La documentazione deve spiegare in quali casi vengono lanciate
- Vanno intercettate con l'istruzione `try-catch`
- Di norma si costruiscono sotto-classi ad-hoc di `Exception`

Errori ed eccezioni unchecked: cosa accade

Quando accadono, ossia quando vengono lanciate..

- Causano l'interruzione dell'applicazione
- Comportano la scrittura su console di errore (`System.err`) di un messaggio che include lo StackTrace – `Thread.dumpStack();`
  - nota, solitamente `System.err` coincide con `System.out`
- Dal quale possiamo desumere la sequenza di chiamate e il punto del codice in cui si ha avuto il problema

Errori/eccezioni unchecked comuni e già viste

- `StackOverFlowError`: stack esaurito (ricorsione infinita?)
- `NullPointerException`, `ArrayStoreException`, `ClassCastException`, `ArrayIndexOutOfBoundsException`, `NumericException`, `OperationNotSupportedException`
- Altri andranno verificati sulla documentazione quando incontrati

Esempio di stampa

```java
public class UncheckedStackTrace{
    public static void main(String[] args){
      final int[] a = new int[]{10,20,30};
      final int b = accessArray(a,1); // OK
      final int c = accessArray(a,3); // Eccezione
      final int d = accessArray(a,5); // Eccezione
    }

    public static int accessArray(final int[] array, final int pos){
      return array[pos];
    }
}

/* Stampa dell'errore:

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
        at UncheckedStackTrace.accessArray(UncheckedStackTrace.java:9)
        at UncheckedStackTrace.main(UncheckedStackTrace.java:5)
*/
```

L'istruzione throw

```
1  public class UncheckedThrow{
2      public static void main(String[] args){
3        final int[] a = new int[]{10,20,30};
4        final int b = accessArray(a,1); // OK
5        final int c = accessArray(a,3); // Eccezione
6      }
7
8      public static int accessArray(final int[] array, final int pos){
9        if (pos < 0 || pos >= array.length){
10         final String msg = "Accesso fuori dai limiti, in posizione "+pos;
11         throw new java.lang.IllegalArgumentException(msg);
12       }
13       return array[pos];
14     }
15 }
16
17 /* Stampa dell'errore:
18
19 Exception in thread "main" java.lang.IllegalArgumentException: Accesso fuori
       dai limiti, in posizione 3
20   at it.unibo.apice.oop.p13exceptions.classes.UncheckedThrow.accessArray(
       UncheckedThrow.java:13)
21   at it.unibo.apice.oop.p13exceptions.classes.UncheckedThrow.main(
       UncheckedThrow.java:7)
22 */
```

L'istruzione `throw`: una variante equivalente

```
1  public class UncheckedThrow2{
2      public static void main(String[] args){
3        final int[] a = new int[]{10,20,30};
4        final int b = accessArray(a,1); // OK
5        final int c = accessArray(a,3); // Eccezione
6      }
7
8      public static int accessArray(final int[] array, final int pos){
9        if (pos < 0 || pos >= array.length){
10         final String msg = "Accesso fuori dai limiti, in posizione "+pos;
11         RuntimeException e = new java.lang.IllegalArgumentException(msg);
12         throw e;
13       }
14       return array[pos];
15     }
16 }
```

L'istruzione `throw`

Sintassi generale

```
throw <expression-that-evaluates-to-a-throwable>;
```

Casi tipici

```
throw new <exception-class>(<message-string>);
throw new <exception-class>(<ad-hoc-args>);
throw new <exception-class>();
```

Effetto

- Si interrompe immediatamente l'esecuzione del metodo in cui ci si trova (se non dentro una `try-catch`, come vedremo dopo..)
- L'oggetto eccezione creato viene "riportato" al chiamante
- Ricorsivamente, si giunge al `main`, con la stampa su `System.err` (exception chaining)

Outline

Riconsideriamo l'implementazione di `RangeIterator`

Elementi da considerare

- Controllare l'interfaccia `java.util.Iterator`
- Verificare la documentazione presente nel sorgente (ed in particolare, come si specificano le eccezioni lanciate)
- Il comando: `javadoc Iterator.java`
- La documentazione HTML prodotta
- Realizzazione e prova di `RangeIterator`

Documentazione di Iterator: header

```
1  /*
2   * Copyright (c) 1997, 2010, Oracle and/or its affiliates. All rights
        reserved.
3   * .. informazioni generali della Oracle
4   */
5
6  package java.util;
7
8  /**
9   * An iterator over a collection.
10  * .. descrizione generale della classe..
11  *
12  * <p>This interface is a member of the
13  * <a href="{@docRoot}/../technotes/guides/collections/index.html">
14  * Java Collections Framework</a>.
15  *
16  * @param <E> the type of elements returned by this iterator
17  *
18  * @author  Josh Bloch
19  * @see Collection
20  * @see ListIterator
21  * @see Iterable
22  * @since 1.2
23  */
24 public interface Iterator<E> { ...
```

Documentazione di Iterator: next() e hasNext()

```
1  public interface Iterator<E> {
2      /**
3       * Returns {@code true} if the iteration has more elements.
4       * (In other words, returns {@code true} if {@link #next} would
5       * return an element rather than throwing an exception.)
6       *
7       * @return {@code true} if the iteration has more elements
8       */
9      boolean hasNext();
10
11     /**
12      * Returns the next element in the iteration.
13      *
14      * @return the next element in the iteration
15      * @throws NoSuchElementException if the iteration has no more elements
16      */
17     E next();
18     ...
```

Documentazione di Iterator: `remove()`

```
1     ..
2     /**
3      * Removes from the underlying collection the last element returned
4      * by this iterator (optional operation).  This method can be called
5      * only once per call to {@link #next}.  The behavior of an iterator
6      * is unspecified if the underlying collection is modified while the
7      * iteration is in progress in any way other than by calling this
8      * method.
9      *
10     * @throws UnsupportedOperationException if the {@code remove}
11     *         operation is not supported by this iterator
12     *
13     * @throws IllegalStateException if the {@code next} method has not
14     *         yet been called, or the {@code remove} method has already
15     *         been called after the last call to the {@code next}
16     *         method
17     */
18    void remove();
19 }
```

Documentazione generata: pt1

Package Class Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Interface Iterator<E>

Type Parameters:

E - the type of elements returned by this iterator

---

public interface **Iterator<E>**

An iterator over a collection. `Iterator` takes the place of `Enumeration` in the Java Collections Framework. Iterators differ from enumerations in two ways:

- Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics.
- Method names have been improved.

This interface is a member of the Java Collections Framework.

Since:

1.2

See Also:

`Collection`, `ListIterator`, `Iterable`

Method Summary

Methods

Modifier and Type	Method and Description
boolean	**hasNext**() Returns true if the iteration has more elements.
E	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by this iterator (optional operation).

Documentazione generata: pt2

Method Detail

hasNext

```
boolean hasNext()
```

Returns `true` if the iteration has more elements. (In other words, returns `true` if `next()` would return an element rather than throwing an exception.)

Returns:

`true` if the iteration has more elements

next

```
E next()
```

Returns the next element in the iteration.

Returns:

the next element in the iteration

Throws:

`NoSuchElementException` - if the iteration has no more elements

remove

```
void remove()
```

Removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to `next()`. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Throws:

`java.lang.UnsupportedOperationException` - if the `remove` operation is not supported by this iterator

`java.lang.IllegalStateException` - if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method

Realizzazione di RangeIterator

```
1  public class RangeIterator implements java.util.Iterator<Integer> {
2
3    private static final String MSG = "start can't be bigger than stop";
4    private int current;
5    private final int stop;
6
7    public RangeIterator(final int start, final int stop) {
8      if (start > stop) { // parametri errati
9        throw new java.lang.IllegalArgumentException(MSG);
10     }
11     this.current = start;
12     this.stop = stop;
13   }
14
15   public Integer next() {
16     if (!this.hasNext()) {
17       throw new java.util.NoSuchElementException();
18     }
19     return this.current++;
20   }
21
22   public boolean hasNext() {
23     return this.current <= this.stop;
24   }
25
26   public void remove() { // non supportiamo remove
27     throw new UnsupportedOperationException();
28   }
29 }
```

Outline

Il costrutto try-catch

Sintassi (da estendere successivamente)

```
try { <body-maybe-throwing-an-exception>}
   catch (<throwable-class> <var>) { <handler-body>}
```

Esempio

```
try { RangeIterator r = new RangeIterator(a,b);}
   catch (RuntimeException e) { System.out.println(e);}
```

Significato

- Se il body nella `try` lancia una eccezione del tipo specificato nella `catch`
- Allora si esegue il corrispondente handler, e non si ha la terminazione della applicazione
- Se non c'è eccezione si salta l'handler e si prosegue

Uso della RangeIterator senza try-catch

```
1  public class UseRange{
2      public static void main(String[] args){
3        // args tiene inizio e fine, passate da line di comando
4        final int a = Integer.parseInt(args[0]); // "5"
5        final int b = Integer.parseInt(args[1]); // "10"
6        final RangeIterator r = new RangeIterator(a,b);
7        // remove(); // cosa succede con remove?
8        try {
9          System.out.print(r.next()+" ");
10         System.out.print(r.next()+" ");
11         System.out.println(r.next());
12       } catch (final java.util.NoSuchElementException e){
13         System.out.println("exception... but we go on "+e.toString());
14       }
15       System.out.println("end");
16     }
17 }
18 /* Esecuzione: java UseRange 5 10
19    args vale: new String[]{"5","10"}
20    risultato:  5 6 7 */
21
22 /* Esecuzione: java UseRange 5 10.1
23    risultato:  NumberFormatException */
24
25 /* Esecuzione: java UseRange 5 3
26    risultato:  IllegalArgumentException */
27
28 /* Esecuzione: java UseRange 3 4
```

Uso della RangeIterator con try-catch

```
1  public class UseRange2{
2      public static void main(String[] s){
3        RangeIterator r = null; // va creata fuori dal try..
4        try{
5          final int a = Integer.parseInt(s[0]);
6          final int b = Integer.parseInt(s[1]);
7            r = new RangeIterator(a,b);
8        } catch (Exception e){ // catturo una qualsiasi Exception
9            System.out.println("Wrong arguments!");
10           System.out.println(e);
11           System.exit(1); // abnormal termination
12       }
13       try{
14           System.out.print(r.next()+" ");
15           System.out.print(r.next()+" ");
16           System.out.println(r.next());
17       } catch (java.util.NoSuchElementException e){
18           System.out.println("Iteration not correct...");
19           System.out.println(e);
20           System.exit(1); // abnormal termination
21       }
22       System.exit(0); // ok termination
23     }
24 }
```

Il costrutto try-catch-finally

Sintassi generale

```
try { <body-maybe-throwing-an-exception>}
   catch (<throwable-class> <var>) { <handler-body>}
   catch (<throwable-class> <var>) { <handler-body>}
   ...
   catch (<throwable-class> <var>) { <handler-body>}
   finally { <completion-body>} // clausola finale opzionale
```

Significato

- Se il body nella `try` lancia una eccezione
- La prima `catch` pertinente esegue l'handler (non ci possono essere sovrapposizioni!)
- Poi si eseguirà anche il `completion-body`
- Il body nella `finally` sarà comunque eseguito!

catch multipli e finally

```
1  public class UseRange3{
2      public static void main(String[] s){
3        RangeIterator r = null; // creabile anche dentro al try..
4        try{ // attenzione alla formattazione di questo esempio!
5          final int a = Integer.parseInt(s[0]);
6            final int b = Integer.parseInt(s[1]);
7            r = new RangeIterator(a,b);
8            System.out.print(r.next()+" ");
9            System.out.print(r.next()+" ");
10           System.out.println(r.next());
11       } catch (ArrayIndexOutOfBoundsException e){
12           System.out.println("Need two arguments!");
13       } catch (NumberFormatException e){
14           System.out.println("Need integer arguments!");
15       } catch (IllegalArgumentException e){
16           System.out.println(e);
17       } catch (Exception e){ //ogni altra eccezione
18           throw e; // rilancio l'eccezione
19       } finally{
20           // questo codice comunque eseguito
21           System.out.println("bye bye..");
22       }
23     }
24 }
```

Spiegazione

Come funziona la `finally`?

- garantisce che il codice nel suo handler sarà sicuramente eseguito
- ..sia se ho avuto eccezione
- ..sia se non ho avuto eccezione
- ..sia se uno degli handler delle catch ha generato eccezione

A cosa serve?

- in genere contiene del codice di cleanup che deve comunque essere eseguito
- rilascio risorse, chiusura file, stampa messaggi, etc..

Vedremo la prossima settimana il costrutto chiamato `try`-with-resources

- consente di non esprimere il `finally`

catch multiplo con "alternative types"

```
1  public class UseRange4{
2      public static void main(String[] s){
3        RangeIterator r = null; // creabile anche dentro al try..
4        try{ // attenzione alla formattazione di questo esempio!
5          final int a = Integer.parseInt(s[0]);
6            final int b = Integer.parseInt(s[1]);
7            r = new RangeIterator(a,b);
8            System.out.print(r.next()+" ");
9            System.out.print(r.next()+" ");
10           System.out.println(r.next());
11       } catch (ArrayIndexOutOfBoundsException |
     NumberFormatException e){
12           System.out.println("Incorrect type or number of
     parameters");
13       } catch (Exception e){ //ogni altra eccezione
14           throw e; // rilancio l'eccezione
15       } finally{
16           // questo codice comunque eseguito
17           System.out.println("bye bye..");
18       }
19     }
20 }
```

Eccezioni del JDK da riusare

Tutte nel package `java.lang`

- `NullPointerException`: quando viene passato un riferimento `null` inatteso
- `IllegalArgumentException`: quando un input non ha le caratteristiche richieste
- `IndexOutOfBoundsExecption`: quando un indice/posizione è fuori dai limiti imposti
- `IllegalStateException`: quando la chiamata di metodo è fatta al momento sbagliato
- `OperationNotSupportedException`: quando la classe non supporta questa funzionalità

Outline

Creazione di una nuova classe di eccezioni

Nuove eccezioni

- Un sistema potrebbe richiedere nuovi tipi di eccezioni, che rappresentano eventi specifici collegati al dominio applicativo
  - Persona già presente (in un archivio cittadini)
  - Lampadina esaurita (in una applicazione domotica)
- Semplicemente si fa una estensione di `Exception` o `RuntimeException`
  - a seconda che la si voglia checked o unchecked
  - per il momento stiamo considerando solo le unchecked
- Non vi sono particolari metodi da ridefinire di solito
- Solo ricordarsi di chiamare correttamente il costruttore del padre
- Se si vuole incorporare una descrizione articolata della causa dell'eccezione, la si può inserire nei campi dell'oggetto tramite il costruttore o metodi setter..

Esempio: MyException

```
1  public class MyException extends RuntimeException{
2
3      // tengo traccia degli argomenti che hanno causato il problema
4      private final String[] args;
5
6      public MyException(final String s, final String[] args){
7        super(s);
8        this.args = args;
9      }
10
11     // modifico la toString per evidenziare this.args
12     public String toString(){
13       String str = "Stato argomenti: ";
14       str = str + java.util.Arrays.toString(args);
15       str = str + "\n" + super.toString();
16       return str;
17     }
18 }
```

Esempio: UseMyException

```
1  public class UseMyException{
2      public static void main(String[] s){
3        try{ // attenzione alla formattazione di questo esempio!
4          final int a = Integer.parseInt(s[0]);
5          final int b = Integer.parseInt(s[1]);
6          final RangeIterator r = new RangeIterator(a,b);
7          System.out.print(r.next()+" ");
8          System.out.print(r.next()+" ");
9          System.out.println(r.next());
10       } catch (Exception e){
11         final String str = "Rilancio di: "+e;
12         RuntimeException e2 = new MyException(str,s);
13         throw e2;
14       }
15     }
16 }
17 /* Esempio: java UseMyException 10 13.1
18 Exception in thread "main" Stato argomenti: [10, 13.1]
19 it.unibo.apice.oop.p13exceptions.classes.MyException: Rilancio di: java.lang
       .NumberFormatException: For input string: "13.1"
20   at it.unibo.apice.oop.p13exceptions.classes.UseMyException.main(
       UseMyException.java:14)
21 */
```

Outline

Checked vs Unchecked

Unchecked: `RuntimeException` o sottoclassi

- Quelle viste finora, dovute ad un bug di programmazione
- Quindi sono da catturare opzionalmente, perché rimediabili

$\Rightarrow$ ..le linee guida più moderne le sconsigliano

Checked: `Exception` o sottoclassi ma non di `RuntimeException`

- Rappresentano errori non riconducibili ad una scorretta programmazione, ma ad eventi abbastanza comuni anche nel sistema una volta installato e funzionante
  - Funzionamento non normale, ma non tale da interrompere l'applicazione (p.e., l'utente fornisce un input errato inavvertitamente)
  - Un problema con l'interazione col S.O. (p.e., file inesistente)
- I metodi che le lanciano lo devono dichiarare esplicitamente (`throws`)
- Chi chiama tali metodi deve obbligatoriamente gestirle
  - o catturandole con un `try-catch`
  - o rilanciandole al chiamante con la `throws`

Una eccezione checked: IOException e input da tastiera

```
1  import java.io.*;
2
3  public class IOFromKeyboard {
4
5    // La dichiarazione throws qui è obbligatoria!
6    public static int getIntFromKbd() throws IOException {
7      InputStreamReader ISR = new InputStreamReader(System.in);
8      BufferedReader keyboardInput = new BufferedReader(ISR);
9      String line = null;
10     line = keyboardInput.readLine(); // IOException
11     return Integer.parseInt(line);
12   }
13
14   public static void main(String[] args) throws Exception {
15       System.out.print("Inserisci un numero: ");
16       int a = getIntFromKbd();
17       System.out.println("Hai inserito il num.: " + a);
18
19   }
20 }
```

Qualche variante: campi statici

```
1  import java.io.*;
2
3  public class IOFromKeyboard2 {
4
5    private static final BufferedReader KBD =
6      new BufferedReader(new InputStreamReader(System.in));
7
8    private static int getIntFromKbd() throws IOException {
9      return Integer.parseInt(KBD.readLine());
10   }
11
12   public static void main(String[] args) {
13     try {
14       System.out.print("Inserisci un numero: ");
15       final int a = getIntFromKbd();
16       System.out.println("Hai inserito il num.: " + a);
17     } catch (IOException e) {
18       System.out.println("Errore di I/O: " + e);
19     } catch (NumberFormatException e) {
20       System.out.println(e);
21     }
22   }
23 }
```

Qualche variante: input iterato e rilancio

```
1  import java.io.*;
2
3  public class IOFromKeyboard3 {
4
5    private static final BufferedReader KBD = new BufferedReader(
6        new InputStreamReader(System.in));
7
8    private static int getIntFromKbd() throws IOException {
9      return Integer.parseInt(KBD.readLine());
10   }
11
12   public static void main(String[] args) throws NumberFormatException {
13     while (true) {
14       try {
15         System.out.print("Inserisci un numero: ");
16         final int a = getIntFromKbd();
17         System.out.println("Hai inserito il num.: " + a);
18       } catch (IOException e) {
19         System.out.println("Errore di I/O: " + e);
20       }
21     }
22   }
23 }
```

Input da tastiera senza eccezioni)

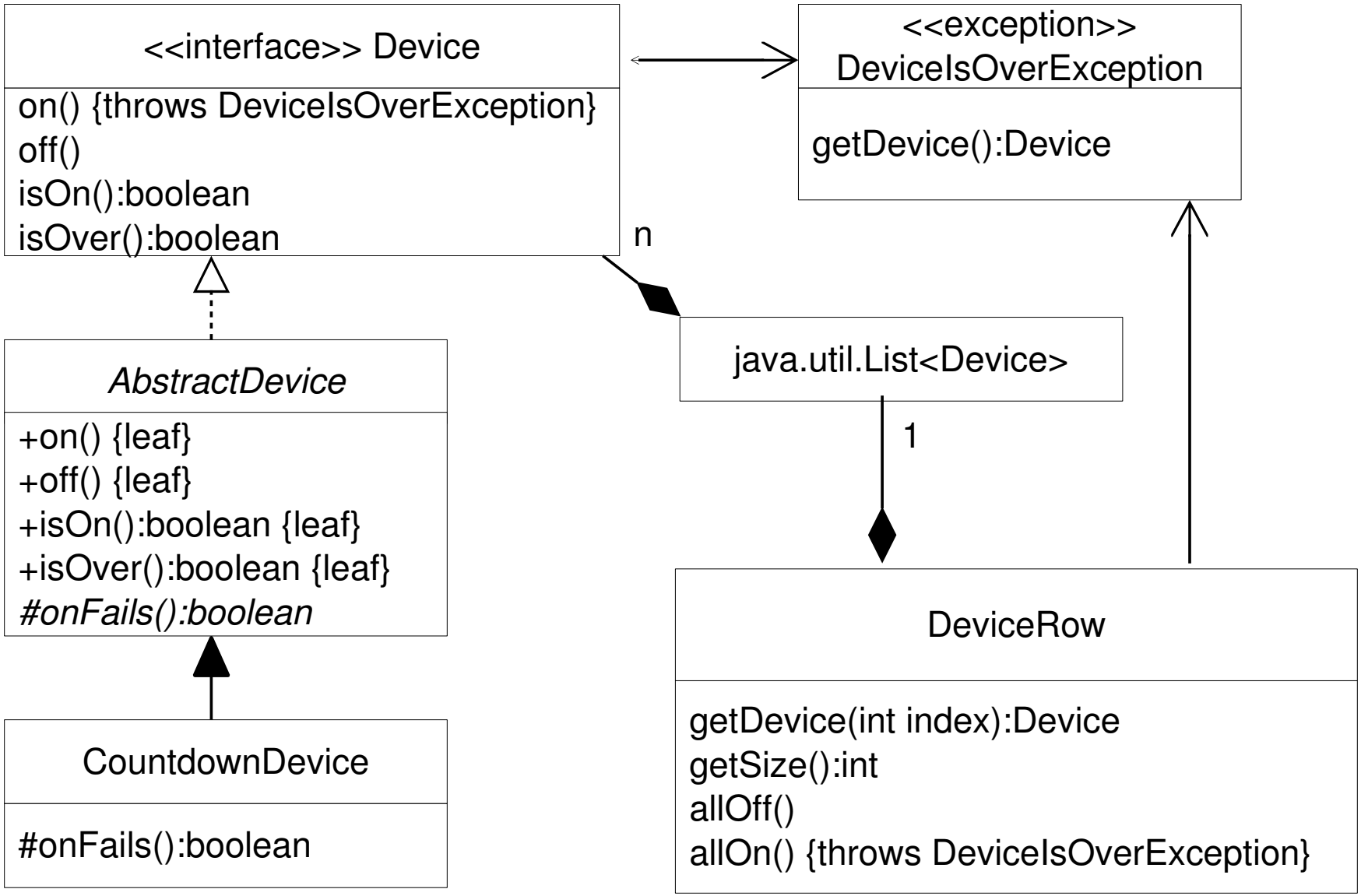
```java
public class IOFromKeyboard4 {

  // L'uso di System.console().readLine() non lancia eccezioni
  public static void main(String[] args) {
    while (true) {
      System.out.print("Insertisci un numero: ");
      final int a = Integer.parseInt(System.console().readLine());
      System.out.println("Hai inserito il num.: " + a);
    }
  }
}

/*
 * Si controlli la classe java.lang.Console, fornisce varie
    funzioni utili per
 * l'I/O, come le stampe formattate tipo printf
 */
```

Outline

Requirements

- Una fila di n `Device` con tempo di vita limitato
- Il sistema dovrà supportare in futuro diverse politiche di fine-vita dei device
- Il fine-vita viene rilevato al tentativo di accensione, ed è segnalato da una eccezione checked
- Esistono comandi per accendere e spegnere tutti i device
- Il sistema dovrà essere a prova di qualunque eccezione

UML: Modellazione Device



Interfaccia Device

```java
public interface Device {

  /**
   * Switches Off the Device. It does nothing if it is already switched off.
   */
  void off();

  /**
   * Switches On the Device. It does nothing if it is already switched on.
   * If already over or becoming over it goes over and off.
   *
   * @throws DeviceIsOverException
   *             if it ends up off
   */
  void on() throws DeviceIsOverException;

  /**
   * @return whether it is on
   */
  boolean isOn();

  /**
   * @return whether it is over
   */
  boolean isOver();
}
```

Eccezione DeviceIsOverException

```java
public class DeviceIsOverException extends Exception {

  private final Device device;

  /**
   * Reported for clarity, not really needed
   */
  public DeviceIsOverException(final Device device) {
    this.device = device;
  }

  public Device getDevice() {
    return this.device;
  }
}
```

AbstractDevice, pt1

```java
public abstract class AbstractDevice implements Device {

  private boolean on;
  private boolean over; // over implies not on

  /**
   * Setting the lamp as working and off
   */
  public AbstractDevice() {
    this.on = false;
    this.over = false;
  }

  final public boolean isOn() {
    return this.on; // getter
  }

  final public boolean isOver() { // getter
    return this.over;
  }

  final public void off() {
    this.on = false; // setter
  }
```

AbstractDevice, pt2

```
1   /*
2    * It makes sure we call onFails() properly, and over and off are
3    * consistently changed.
4    *
5    * @see safedevices.Device#on()
6    */
7   final public void on() throws DeviceIsOverException {
8     if (!this.on) { // is this a real switch-on?
9       this.over = this.onFails(); // is it over?
10      this.on = !this.over; // correspondingly switch
11    }
12    if (!this.on) { // could I switch?
13      throw new DeviceIsOverException(this); // raise exception
14    }
15  }
16
17  /**
18   * Implement the strategy to recognise whether it is over
19   *
20   * @return whether should become over
21   */
22  protected abstract boolean onFails();
23
24  public String toString() {
25    return this.over ? "over" : this.on ? "on " : "off";
26  }
27 }
```

CountdownDevice

```java
public class CountdownDevice extends AbstractDevice {

  private int countdown;

  public CountdownDevice(final int countdown) {
    super();
    if (countdown < 1) {
      throw new IllegalArgumentException();
    }
    this.countdown = countdown;
  }

  protected boolean onFails() {
    if (this.countdown == 0) {
      return true;
    }
    this.countdown--;
    return false;
  }

  public String toString() {
    return super.toString() + "." + this.countdown;
  }
}
```

DeviceRow: Campi e costruttore

```java
public class DeviceRow {

  /**
   * Default countdown for devices
   */
  private static final int COUNTDOWN = 3;

  /**
   * The row of devices as a java.util.List, deferring actual implementation
   */
  private final List<Device> list;

  /**
   * This constructor creates and initializes a list of CountdownDevice
   *
   * @param size is the number of devices to use
   * @throws an IllegalArgumentException if size < 0
   *
   */
  public DeviceRow(int size) {
    if (size < 0) {
      throw new IllegalArgumentException();
    }
    this.list = new ArrayList<>();
    for (int i=0; i<size; i++) {
      this.list.add(new CountdownDevice(COUNTDOWN));
    }
  }
```

DeviceRow: Selettori e allOff()

```java
/**
 * @param index is the position of the device to get
 * @return the device
 */
public Device getDevice(final int index) {
  return this.list.get(index);
}

/**
 * @return the number of devices
 */
public int getSize() {
  return this.list.size();
}

/**
 * Switches all devices off
 */
public void allOff() {
  for (final Device d : this.list) {
    d.off();
  }
}
```

DeviceRow: allOn() e toString()

```java
  /**
   * Switches all devices on, no matter whether one fails.
   *
   * @throws the last DeviceIsOverException raised, if any
   */
  public void allOn() throws DeviceIsOverException {
    DeviceIsOverException e = null;
    for (final Device d : this.list) {
      try {
        d.on();
      } catch (DeviceIsOverException de) {
        e = de;
      }
    }
    if (e != null) {
      throw e;
    }
  }

  public String toString() {
    return "DeviceRow " + list;
  }

}
```

UseDevice

```java
public class UseDevice {

  public static void main(String[] args) {
    final DeviceRow dr = new DeviceRow(3);
    System.out.println(dr);
    // DeviceRow [off.3, off.3, off.3, off.3]
    try {
      dr.allOn();
      dr.allOff();
      dr.allOn();
      dr.allOff();
      dr.getDevice(0).on();
      dr.getDevice(0).off();
      dr.getDevice(1).on();
      dr.getDevice(1).off();
      System.out.println(dr);
      // DeviceRow [off.0, off.0, off.1, off.1]
      dr.allOn(); // Eccezione
    } catch (DeviceIsOverException e) {
      System.out.println("Eccezione...");
    }
    System.out.println(dr);
    // DeviceRow [over.0, over.0, on .0, on .0]
  }
}
```

Best practice con le eccezioni

Che ricapitoleremo e discuteremo a fine corso

- i metodi pubblici controllino subito la validità degli input, lanciando eccezioni unchecked per segnalare il "bug"
- si riusino in via prioritaria le eccezioni delle librerie Java
- se un problema non è dovuto a bug, sarebbe meglio non lanciare eccezioni, ma fornire un output che rappresenta l'anomalia