GESTIONE DEGLI ERRORI

- **Spesso vi sono istruzioni "critiche", che in certi casi possono produrre errori.**
- **L'approccio classico consiste nell'*inserire controlli* (if… else..) per *cercare di intercettare a priori* le situazioni critiche**
- **Ma è un modo di procedere spesso *insoddisfacente***
  - **non è facile prevedere tutte le situazioni che potrebbero produrre l'errore**
  - **"gestire" l'errore spesso significa solo stampare a video un messaggio.**

ECCEZIONI

Java introduce il concetto di *eccezione*

- **anziché *tentare di prevedere* le situazioni di errore, si *tenta di eseguire* l'operazione *in un blocco controllato***
- **se si produce un errore, l'operazione *lancia un'eccezione***
- **l'eccezione viene *catturata* dal blocco entro cui l'operazione è eseguita…**
- **… e può essere *gestita* nel modo più appropriato.**

ECCEZIONI

```
try {
   /* operazione critica che può
      lanciare eccezioni */
}
catch (Exception e) {
   /* gestione dell'eccezione */
}
```

- Se l'operazione lancia *diversi tipi* di eccezione in risposta a diversi tipi di errore, *più blocchi* **`catch`** possono seguire lo stesso blocco **`try`**

FLUSSO DI ESECUZIONE

- **Se l'operazione critica *ha successo,* nessuno dei blocchi `catch` viene eseguito.**
- **Se l'operazione critica lancia un'eccezione, l'esecuzione prosegue nel blocco `catch` appropriato.**
- **È possibile specificare un blocco `finally` opzionale *da eseguirsi comunque alla fine*, indipendentemente dal fatto che si sia eseguito il blocco `try` o un blocco `catch`.**

```
try {
   ...
}
catch (Exception e1) {
   ...
}
catch (Exception e2) {
   ...
}
...
finally {
 // operazioni finali
}
```

ESEMPIO

Conversione stringa / numero

- In Java, la conversione stringa / numero intero è svolta dal metodo statico

  `int Integer.parseInt(String s)`

- L'operazione è critica, perché può avvenire *solo se la stringa data contiene la rappresentazione di un intero*
- Se ciò non accade, `parseInt` lancia una `NumberFormatException`

ESEMPIO

```
class EsempioEccezione {
 public static void main(String args[]){
  int a = 0;
  String s = "1123";
  try {
    a = Integer.parseInt(s);
  }
  catch (NumberFormatException e) {
    System.out.println("Stringa mal fatta");
  }
 }
}
```

COS'È UNA ECCEZIONE

- **Una eccezione *è un oggetto*, istanza di `Throwable` o di una sua sottoclasse:** le due sottoclassi tipiche sono **`Exception`** e **`Error`**
- **Un `Error` indica problemi relativi al funzionamento della macchina virtuale Java e va solitamente considerato *irrecuperabile*: perciò *non* è da catturare, né da gestire** (esempi: `LinkageError`, `ThreadDeath`, ...)
- **Una `Exception` indica invece situazioni *recuperabili*, almeno in linea di principio: va quindi *catturata* e *gestita*** (esempi: fine file, indice di un array oltre i limiti, errori di input, etc.).

ECCEZIONI COME OGGETTI

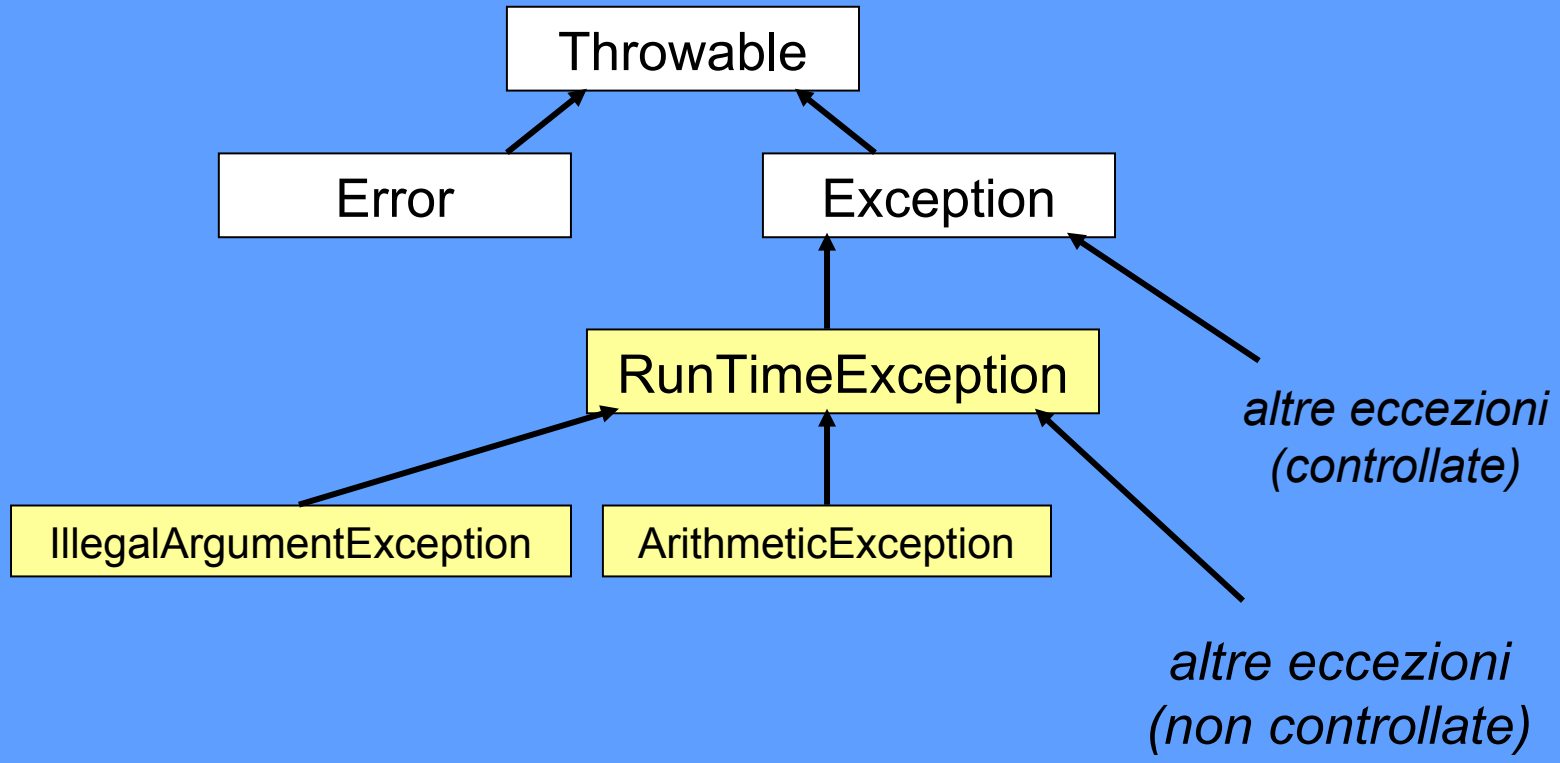
- **Poiché un'eccezione è un oggetto, può *contenere dati* o *definire metodi.***
- **Tutte le eccezioni definiscono un metodo `getMessage()` che restituisce il messaggio d'errore associato**
  - **alcune eccezioni definiscono anche dei campi dati (ad esempio, `bytesTransferred` in `InterruptedIOException`) che danno altre informazioni, utili per gestire la situazione.**

ECCEZIONI CONTROLLATE e NON CONTROLLATE

- **Sebbene sia comunque *utile e raccomandabile* catturare e gestire *tutte* le eccezioni, ciò *non è obbligatorio per tutte le eccezioni***
- **Le eccezioni di classe `RunTimeException` (e derivate) possono *non essere gestite:*** per questo vengono dette *non controllate*.
  È comunque buona norma gestirle.

Una eccezione non controllata può propagarsi di blocco in blocco: *se raggiunge il `main` senza essere stata catturata, il programma abortisce.*

ECCEZIONI CONTROLLATE e NON CONTROLLATE



RILANCIO DI ECCEZIONI

In Java, un metodo che possa generare un'eccezione deve:

- ***o gestire l'eccezione,* con un costrutto `try` / `catch`**
- ***oppure rilanciarla esplicitamente all'esterno* del metodo, delegandone in pratica la gestione ad altri** (unica eccezione: il main)

<u>Se sceglie questa seconda strada</u>, il metodo *deve* indicare quale/i eccezione/i possono "scaturire" da esso, con la clausola `throws`

RILANCIO DI ECCEZIONI

Ad esempio, un metodo che svolga una conversione stringa/numero, anziché gestire la `NumberFormatException` può decidere di *rilanciarla all'esterno:*

```
public int convertStringToInt(String s)
        throws NumberFormatException {
   return Integer.parseInt(s);
}
```

Può lanciare un'eccezione

Non la gestisce, la rilancia all'esterno

DEFINIRE NUOVE ECCEZIONI

Dato che un'eccezione è un normale oggetto Java, è possibile:

- **definire *nuovi tipi di eccezione* definendo *nuove classi***
- ***generare eccezioni* dall'interno di propri metodi.**

Per definire un nuovo tipo di eccezione basta *definire una nuova classe* che estenda la classe base `Exception` (o una delle sue sottoclassi)

ESEMPIO

```
class NumberTooBigException
  extends IllegalArgumentException {
 public NumberTooBigException() {
       super();
 }
 public NumberTooBigException(String s){
       super(s);
 }
}
```

Definisce i due costruttori standard:

- quello di default
- quello con un parametro stringa (il messaggio)

ESEMPIO

Per lanciare un'eccezione:

- **prima si crea l'oggetto Exception da lanciare**
- **poi lo si lancia con l'istruzione `throw`**

```
public int convertStringToInt(String s)
      throws NumberFormatException,
             NumberTooBigException {
  int x = Integer.parseInt(s);
  if (x>100)
    throw new NumberTooBigException();
  return x;
}
```

ESEMPIO

Per sol

- **prim**
- **poi**

Importante non confondere:

- la clausola **`throws`** che dichiara che un metodo rilancia all'esterno un'eccezione
- l'istruzione **`throw`** che lancia un'eccezione

```
public int
      throws NumberFormatException,
             NumberTooBigException {
  int x = Integer.parseInt(s);
  if (x>100)
    throw new NumberTooBigException();
  return x;
}
```