



---

## PROGRAMMAZIONE 2

### 6a. Eccezioni in Java

PR2 2017-2018

1



### Generazione di “errori”

---

- Un metodo può richiedere che gli argomenti attuali soddisfino determinate precondizioni per procedere nell'esecuzione
  - m(List L) con L non vuota
- Componenti esterni potrebbero fallire
  - File non presente
- Implementazioni parziali
  - Modulo con alcune funzionalità non ancora implementate
- Come gestiamo queste situazioni “anomale”?

PR2 2017-2018

2

## Gestione errori

---



- Diverse tecniche
  - Parser per gli errori sintattici
  - Analisi statica (type checker) per gli errori semantici
  - Test covering & Best practice
  - Ignorare gli errori
- Ora noi vedremo il meccanismo delle eccezioni: meccanismi linguistici che permettono di trasferire il controllo del programma dal punto in cui viene rilevato l'errore al codice che permette di gestirlo

## Cosa sono?

---



- Le eccezioni sono dei particolari oggetti usati per rappresentare e catturare condizioni anomale del comportamento di programmi
  - Comportamenti anomali in operazioni I/O, null pointer, ...
- *Sollevare (throwing)* una eccezione significa programmare una sorta di uscita di emergenza nell'esecuzione del programma
- *Catturare (catching)* una eccezione significa programmare le azioni da eseguire per gestire il comportamento anomalo

## Perché sono utili?

---



- Il compilatore non è in grado di determinare tutti gli errori
- *Separation of concern*: separare il codice di gestione degli errori dal codice "normale"
  - Chiarezza del codice (debugging)
  - Raggruppare e differenziare la struttura (tramite tipi) delle situazioni di comportamento anomalo che si possono presentare

## Esempio

---



```
public class ArrayExceptionExample {
    public static void main(String[] args) {
        String[] colori = {"Rossi", "Bianchi", "Verdi"};
        System.out.println(colori[3]);
    }
}
```

*Cosa succede quando compiliamo e poi mandiamo il programma in esecuzione?*

## Esempio

---



```
public class ArrayExceptionExample {
    public static void main(String[] args) {
        String[] colori = {"Rossi", "Bianchi", "Verdi"};
        System.out.println(colori[3]);
    }
}
```

Compilazione OK, ma a run-time...

```
ArrayExceptionExampleException in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 3 at
ArrayExceptionExample.main(ArrayExceptionExample.java:6)
```

PR2 2017-2018

7

## Formato dei messaggi

---



[exception class]:  
[additional description of exception] at  
[class].[method]([file]: [line number])

PR2 2017-2018

8

## Formato

---



- `java.lang.ArrayIndexOutOfBoundsException`: 3 at `ArrayExceptionExample.main(ArrayExceptionExample.java:6)`
- **Exception Class?**
  - `java.lang.ArrayIndexOutOfBoundsException`
- **Quale indice dell' array (additional information)?**
  - 3
- **Quale metodo solleva l'eccezione?**
  - `ArrayExceptionExample.main`
- **Quale file contiene il metodo?**
  - `ArrayExceptionExample.java`
- **Quale linea del file solleva l'eccezione?**
  - 6

## Eccezioni a runtime

---



- Abbiamo visto il caso nel quale le situazioni anomale provocano a run-time la terminazione (anomala) del programma in esecuzione
- Questo tipo di eccezioni a run-time sono denominate *unchecked exception*
- Domanda: è possibile prevedere meccanismi linguistici che permettano di affrontare le situazioni anomale come un "normale" problema di programmazione?

## Codificare le anomalie

---



- Prevedere opportuni meccanismi di codifica per le situazioni anomale
  - ArrayOutOfBounds: l'accesso all'array fuori dalla dimensione restituisce il valore "-1" che codifica l'anomalia
  - L'accesso a un file non presente nello spazio del programma restituisce la stringa "null"
  - È fattibile? È un tecnica scalabile?
- Il modo moderno di affrontare questo aspetto è quello di introdurre specifici meccanismi linguistici
  - OCaml (failwith), Java (throw+try-catch), C++, C# ...

## Java: sollevare eccezioni

---



- Il linguaggio prevede una primitiva specifica per dichiarare e programmare il modo in cui le eccezioni sono sollevate
- Usare il costrutto **throw** all'interno del codice dei metodi
 

```
if (myObj.equals(null))
    throw new NullPointerException( )
```

## throw

---



- Il costrutto throw richiede come argomento un oggetto che abbia come tipo un qualunque sotto-tipo di Throwable
- La classe Throwable contiene tutti i tipi di errore e di eccezioni
- Come si fa a vedere la struttura?
  - Consultate la documentazione on line delle API
  - [docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html)

PR2 2017-2018

13

## Dichiarare eccezioni

---



- Se un metodo contiene del codice che può generare una eccezione allora si deve esplicitare nella dichiarazione del metodo tale possibilità
  - `public void myMethod throws Exception { ... }`
  - `public void myMethod throws IOException { ... }`
- L'eccezione diventa una componente del tipo del metodo!
- Questo tipo di eccezioni è chiamato **checked exceptions**: “They represent exceptions that are frequently considered *non fatal* to program execution” (Java Tutorial)

PR2 2017-2018

14

## Gestione delle eccezioni

---



- Java prevede strumenti linguistici per programmare la gestione delle eccezioni
- Clausola

```
try {  
    // codice che può sollevare l'eccezione  
}  
catch ([tipo eccezione] e) {  
    // codice di gestione della eccezione  
}
```

## Gestioni multiple

---



- È possibile programmare una gestione “multipla” delle eccezioni

```
try {  
    // codice che può sollevare diverse eccezioni  
}  
catch (IOException e) {  
    // gestione IOException  
}  
catch (ClassNotFoundException e) {  
    // gestione ClassNotFoundException  
}
```



## Eccezioni senza speranza



- La clausola `finally` permette di programmare del codice di *clean-up* indipendentemente da quello che è successo nel codice monitorato

```
try {
    // codice che può sollevare diverse eccezioni
}
catch ([tipo eccezione] e) {
    // gestione Exception
}
finally {
    // codice di clean-up che viene sempre eseguito
}
```

PR2 2017-2018

17

## Il nostro esempio



```
public class ArrayExceptionExample {
    public static void main(String[] args) {
        String[] colori = {"Rossi", "Bianchi", "Verdi"};
        System.out.println(colori[3]);
    }
}
```

```
ArrayExceptionExampleException in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 3 at
ArrayExceptionExample.main(ArrayExceptionExample.java:6)
```

Esempio di una eccezione unchecked (run-time)  
 Eccezioni unchecked: il metodo non deve necessariamente prevedere il codice di gestione

PR2 2017-2018

18

## Checked Exception

---



- Le eccezioni checked sono eccezioni che devono essere gestite da opportuni gestori
- Il compilatore controlla che le eccezioni checked siano sollevate (clausola throw) e gestite (clausola catch)

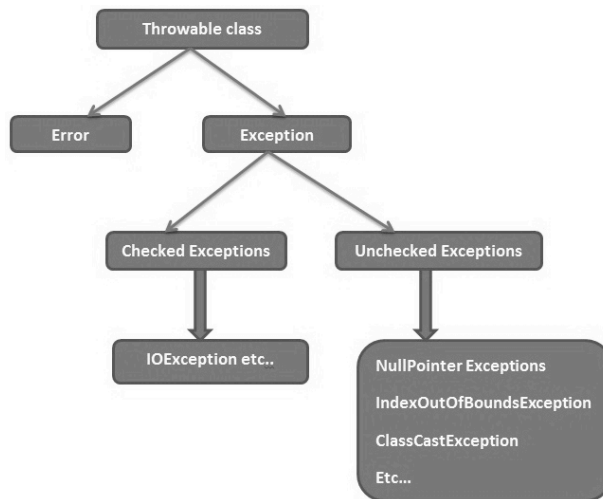
## Ricapitoliamo

---



- I tipi di eccezione sono classi di Java che
  - contengono solo il costruttore
    - ✓ ci possono essere più costruttori overloaded
  - sono definite in “moduli” separati da quelli che contengono i metodi che le possono sollevare
- Le eccezioni sono oggetti
  - creati eseguendo new di un exception type e quindi eseguendo il relativo costruttore
- Esiste una gerarchia “predefinita” di tipi relativi alle eccezioni
  - nuovi tipi di eccezioni sono collocati nella gerarchia con l’usuale extends

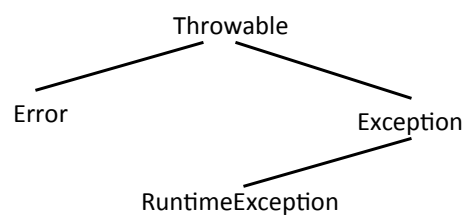
## Java Exception Hierarchy



PR2 2017-2018

21

## La gerarchia di tipi per le eccezioni



- Se un nuovo tipo di eccezione estende la classe Exception, l'eccezione è checked
- Se un nuovo tipo di eccezione estende la classe RuntimeException, l'eccezione è unchecked

PR2 2017-2018

22

## Esempi

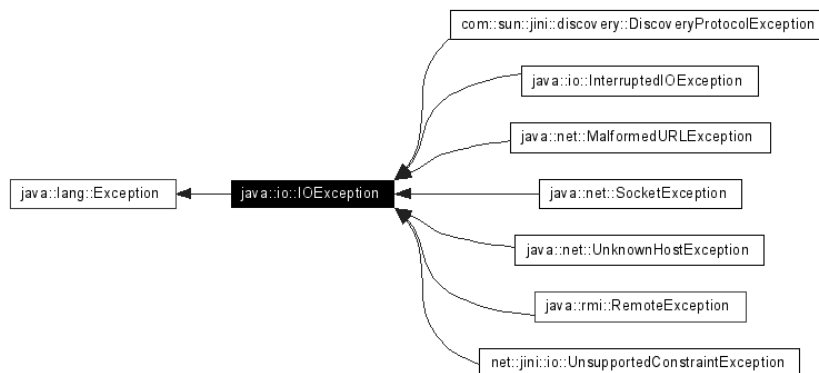


```

java.lang
Class NullPointerException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.NullPointerException
    
```

## Esempi



## Eccezioni checked e unchecked



- Se un metodo può sollevare una eccezione checked
  - deve elencarla nel suo header
    - ✓ che fa parte anche della specifica
  - altrimenti si verifica un errore a tempo di compilazione
- Se un metodo può sollevare una eccezione unchecked
  - può non elencarla nel suo header
    - ✓ il suggerimento è di elencarla sempre, per rendere completa la specifica
- Se un metodo chiamato da obj ritorna sollevando una eccezione
  - se l'eccezione è checked
    - ✓ obj deve gestire l'eccezione (try and catch)
    - ✓ se l'eccezione (o un suo super-tipo) è elencata tra le sollevabili da obj, può essere propagata alla procedura che ha chiamato obj
  - se l'eccezione è unchecked
    - ✓ può essere comunque gestita o propagata

PR2 2017-2018

25

## Definire tipi di eccezione



```
public class NuovoTipoDiEcc extends Exception {
    public NuovoTipoDiEcc(String s) { super(s); }
}
```

- È checked
- Definisce solo un costruttore
  - come sempre invocato quando si crea una istanza con new
  - il costruttore può avere parametri
- Il corpo del costruttore riutilizza semplicemente il costruttore del super-tipo
  - perché deve passargli il parametro?
- Una new di questa classe provoca la creazione di un nuovo oggetto che “contiene” la stringa passata come parametro

PR2 2017-2018

26

## Costruire oggetti eccezione



```
public class NuovoTipoDiEcc extends Exception {
    public NuovoTipoDiEcc(String s) { super(s); }
}
```

- L'invocazione di **new** su questa classe provoca la creazione di un nuovo oggetto che "contiene" la stringa passata come parametro

```
Exception e = new NuovoTipoDiEcc ("Questa è la ragione");
String s = e.toString( );
```

- la variabile `s` punta alla stringa  
"NuovoTipoDiEcc: Questa è la ragione"

PR2 2017-2018

27

## Sollevare eccezioni



- Un metodo può terminare
  - (ritorno normale) con un `return` se deve restituire un valore
  - (ritorno normale) quando le istruzioni che costituiscono il corpo del metodo sono completate
  - (ritorno di una eccezione) con un `throw`

```
public static int fact (int n) throws NonPositiveExc {
    // se n>0, ritorna n!
    // altrimenti solleva NonPositiveExc
    if (n <= 0) throw new NonPositiveExc("Num.fact");
}
```

- La stringa contenuta nell'eccezione è utile soprattutto quando il programma non è in grado di "gestire" l'eccezione
  - permette all'utente di identificare la procedura che l'ha sollevata
  - può comparire nel messaggio di errore che si stampa subito prima di forzare la terminazione dell'esecuzione

PR2 2017-2018

28

## Gestire eccezioni



- Quando un metodo termina con un `throw`
  - l'esecuzione non riprende con il codice che segue la chiamata (call-return tradizionale)
  - il controllo viene trasferito a un pezzo di codice preposto alla gestione dell'eccezione
- Due possibilità per la gestione
  - gestione esplicita, quando l'eccezione è sollevata all'interno di uno statement **try**
    - ✓ in generale, quando si ritiene di poter recuperare uno stato consistente e di portare a termine una esecuzione quasi "normale"
  - gestione di default, mediante propagazione dell'eccezione al codice chiamante
    - ✓ possibile solo per eccezioni unchecked o per eccezioni checked elencate nell'header del metodo che riceve l'eccezione

PR2 2017-2018

29

## Gestione esplicita delle eccezioni



- Gestione esplicita: l'eccezione è sollevata all'interno di uno statement **try**
- Codice per gestire l'eccezione `NonPositiveExc` eventualmente sollevata da una chiamata di `fact`

```
try { x = Num.fact (y); }
catch (NonPositiveExc e) {
    // qui possiamo usare e, cioè l'oggetto eccezione
}
```
- La clausola `catch` non deve necessariamente identificare il tipo preciso dell'eccezione, ma basta un suo super-tipo
 

```
try { x = Arrays.searchSorted (v, y); }
catch (Exception e) { s.Println(e); return; }
// s è una PrintWriter
```
- segnala l'informazione su `NullPointerException` e su `NotFoundExc`

PR2 2017-2018

30

## Try e Catch annidati

---



```
try {
  try { x = Arrays.searchSorted (v, y); }
  catch (NullPointerException e) {
    throw new NotFoundExc( );
  }
}
catch (NotFoundExc b) { ... }
```

- la clausola catch nel try più esterno cattura l'eccezione NotFoundExc se è sollevata da searchSorted o dalla clausola catch più interna

## Catturare eccezioni unchecked

---



- Le eccezioni unchecked sono difficili da catturare: una qualunque chiamata di procedura può sollevarle, ed è dunque difficile sapere da dove vengono

```
try { x = y[n]; i = Arrays.searchSorted (v, x); }
catch (IndexOutOfBoundsException e) {
  // cerchiamo di gestire l'eccezione pensando
  // che sia stata sollevata da x = y[n]
}
// continuiamo supponendo di aver risolto il problema
```

- ma l'eccezione poteva venire dalla chiamata a searchSorted
- L'unico modo per sapere con certezza da dove viene è restringere lo scope del comando try



## Aspetti metodologici

---



- Gestione delle eccezioni
  - riflessione
  - mascheramento
- Quando usare le eccezioni
- Come scegliere tra checked e unchecked
- *Defensive programming*

## Gestione delle eccezioni

---



- Se un metodo chiamato da obj ritorna sollevando un'eccezione, anche obj termina sollevando un'eccezione
  - usando la propagazione automatica
    - ✓ della stessa eccezione (NullPointerException)
  - catturando l'eccezione e sollevandone un'altra
    - ✓ possibilmente diversa (EmptyException)

## Gestione delle eccezioni



```
public static int min (int[ ] a) throws NullPointerException, EmptyException {
    // se a è null solleva NullPointerException
    // se a è vuoto solleva EmptyException
    // altrimenti ritorna il minimo valore in a

    int m;
    try { m = a[0]; }
    catch (IndexOutOfBoundsException e) {
        throws new EmptyException("Arrays.min");
    }
    for (int i = 1; i < a.length; i++)
        if (a[i] < m) m = a[i];
    return m;
}
```

NB: usiamo le eccezioni (catturate) al posto di un test per verificare se a è vuoto

PR2 2017-2018

35

## Gestione eccezioni via mascheramento



- Se un metodo chiamato da obj ritorna sollevando una eccezione, obj gestisce l'eccezione e ritorna in modo normale

```
public static boolean sorted (int[ ] a) throws NullPointerException {
    // se a è null solleva NullPointerException
    // se a è ordinato in senso crescente ritorna true
    // altrimenti ritorna false
    int prec;
    try { prec = a[0]; }
    catch (IndexOutOfBoundsException e) { return true; }
    for (int i = 1; i < a.length ; i++)
        if (prec <= a[i]) prec = a[i]; else return false;
    return true;
}
```

PR2 2017-2018

36

## Quando usare le eccezioni

---



- Le eccezioni non sono necessariamente errori
  - ma metodi per richiamare l'attenzione del chiamante su situazioni particolari (classificate dal progettista come eccezionali)
- Comportamenti che sono errori ad un certo livello, possono non esserlo affatto a livelli di astrazione superiore
  - `IndexOutOfBoundsException` segnala chiaramente un errore all'interno dell'espressione `a[0]`, ma non necessariamente per le procedure `min` e `sort`
- Il compito primario delle eccezioni è di ridurre al minimo i vincoli della strutturazione di un programma in modo da evitare di codificare informazione su terminazioni particolari nel normale risultato

## Checked o unchecked

---



- Le eccezioni checked offrono maggior protezione dagli errori
  - sono più facili da catturare
  - il compilatore controlla che l'utente le gestisca esplicitamente o per lo meno le elenchi nell'header, prevedendone una possibile propagazione automatica
    - ✓ se non è così, viene segnalato un errore
- Le eccezioni checked sono pesanti da gestire in quelle situazioni in cui siamo ragionevolmente sicuri che l'eccezione non verrà sollevata
  - perché esiste un modo conveniente ed efficiente di evitarla o per il contesto di uso limitato
  - solo in questi casi si dovrebbe optare per una eccezione unchecked

## Defensive programming

---



- L'uso delle eccezioni facilita uno stile di progettazione e programmazione che protegge rispetto agli errori
  - anche se non sempre un'eccezione segnala un errore
- Fornisce una metodologia che permette di riportare situazioni di errore in modo ordinato
  - senza disperdere tale compito nel codice che implementa l'algoritmo
- Nella programmazione *defensive* si incoraggia il programmatore a verificare l'assenza di errori ogniqualvolta ciò sia possibile
  - e a riportarli usando il meccanismo delle eccezioni
  - [un caso importante legato alle implementazioni parziali]

## Metodi e eccezioni

---



- Con le eccezioni i metodi tendono a diventare totali
  - anche se non è sempre possibile
- Chi invoca il metodo dovrebbe farsi carico di effettuare tale controllo
  - sollevando una eccezione
    - ✓ questa eccezione può essere catturata, magari a un livello superiore
    - ✓ si suggerisce di usare in questi casi una eccezione generica `unchecked FailureException`



---

## Un esempio: checked vs. unchecked

[thx to [JJ](#)]

PR2 2017-2018

41

```
public void storeDataFromUrl(String url) {
    try { String data = readDataFromUrl(url); }
    catch (BadUrlException e) {
        e.printStackTrace( );
    }
}

public String readDataFromUrl(String url) throws BadUrlException {
    if (isUrlBad(url))
        throw new BadUrlException("Bad URL: " + url);

    String data = null;
    // read lots of data over HTTP and
    // return it as a String instance
    return data;
}
```

PR2 2017-2018

42

## Checked

---



```
public class BadUrlException extends Exception {  
    public BadUrlException(String s) {  
        super(s);  
    }  
}
```

## La propagazione

---



```
public void storeDataFromUrl(String url) throws BadUrlException {  
    String data = readDataFromUrl(url);  
}
```

## Unchecked

---



```
public class BadUrlException extends RuntimeException {  
    public BadUrlException(String s) {  
        super(s);  
    }  
}
```

PR2 2017-2018

45

## Né cattura, né propagazione

---



```
public void storeDataFromUrl(String url) {  
    String data = readDataFromUrl(url);  
}  
  
public String readDataFromUrl(String url) {  
    if (isUrlBad(url))  
        throw new BadUrlException("Bad URL: " + url);  
  
    String data = null;  
    // read lots of data over HTTP and  
    // return it as a String instance  
    return data;  
}
```

PR2 2017-2018

## Checked vs. unchecked

---



- Pro Checked Exceptions
  - Compiler enforced catching or propagation of checked exceptions make it harder to forget handling that exception
- Pro Checked Exceptions
  - Unchecked exceptions makes it easier to forget handling errors since the compiler doesn't force the developer to catch or propagate exceptions (reverse of 1)
- Pro Unchecked Exceptions
  - Checked exceptions that are propagated up the call stack clutter the top level methods, because these methods need to declare throwing all exceptions thrown from methods they call
- Pro Checked Exceptions
  - When methods do not declare what unchecked exceptions they may throw it becomes more difficult to handle them
- Pro Unchecked Exceptions
  - Checked exceptions thrown become part of a methods interface and makes it harder to add or remove exceptions from the method in later versions of the class or interface