Eccezioni

- **Eccezioni per codificare errori**
  - lancio di eccezioni
- **Eccezioni**
  - user defined
  - checked e unchecked
- **Gestione di eccezioni**
  - cattura di eccezioni

Gestione degli errori

- **Approccio tradizionale:**
  - codifica errori mediante codici
  - restituisci il codice dell'errore che si verifica
- **Problemi**
  - il chiamante dimentica di controllare i codici di errore: la notifica può sfuggire
  - il chiamante può non essere in grado di gestire l'errore per mancanza di informazione
  - l'errore deve essere propagato
  - il codice diventa illeggibile …

Continua…

Gestione degli errori

- **Propagazione degli errori:**
  - non è sufficiente gestire i casi di “successo”

```
x.doSomething()
```

  - al contrario: dobbiamo sempre strutturare il codice in funzione delle possibili situazioni di errore

```
if (!x.doSomething()) return false;
```

  - … faticoso e poco leggibile

Gestione di errori con eccezioni

- **Lancio**
  - Andare in errore corrisponde a lanciare (*throw*) una eccezione, che segnala l'occorrere dell'errore
- **Cattura**
  - Quando una eccezione viene lanciata,, un altro pezzo di codice, detto *exception handler*, può catturare (*catch*) l'eccezione
- **Gestione**
  - trattamento dell'eccezione nel tentativo di ripristinare uno stato corretto per riprendere la computazione

Continua…

Lancio di eccezioni

- **Le eccezioni sono oggetti, con tipi associati**

```
IllegalArgumentException exn;
```

- **Devono prima essere costruite**

```
exn = new IllegalArgumentException(“Bad Argument");
```

- **Poi possono essere lanciate**

```
throw exn;
```

Continua…

Lancio di eccezioni

- **Non è necessario separare le tre operazioni**

```
throw new IllegalArgumentException(“Bad argument");
```

- **Lanciare una eccezione in un metodo causa l’interruzione immediata dell’esecuzione del metodo**
  - L’esecuzione prosegue con un gestore di eccezioni (se è stato programmato) o causa un errore

Lancio di eccezioni: sintassi

```
throw exceptionObject;
```

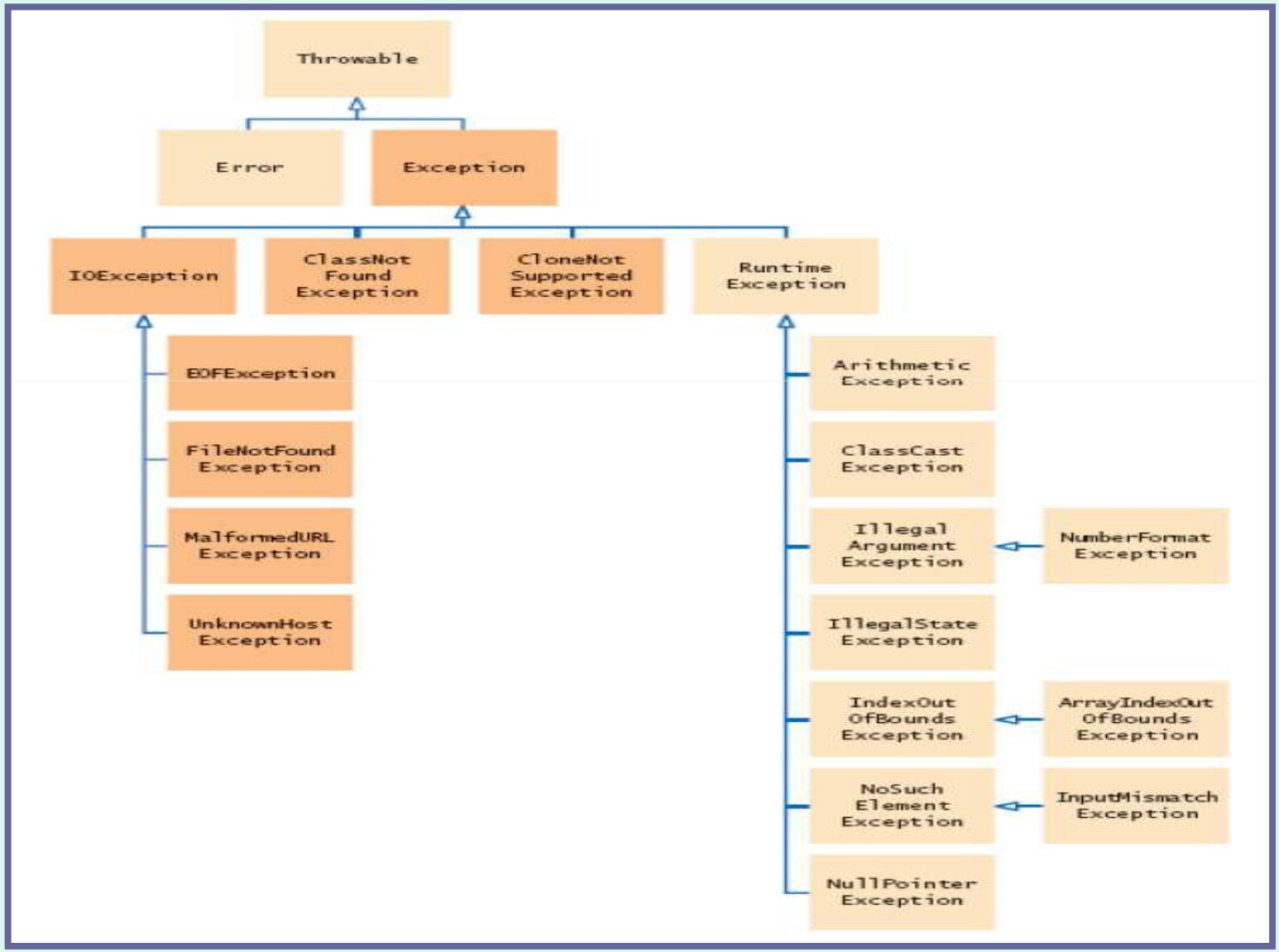
Esempio:

```
throw new IllegalArgumentException();
```

Scopo:

Lanciare una eccezione e trasferire il controllo ad un gestore per il corrispondente tipo di eccezione

Tipi di eccezioni



Eccezioni *checked* e *unchecked*

- **Eccezioni *checked* (controllate)**
  - Derivate dalla classe **`Exception`**
  - Codificano situazioni anomale legate alla logica applicazione, o errori derivanti da condizioni esterne
- **Eccezioni *unchecked* (non controllate)**
  - Derivate dalle classi **`RuntimeException`** o **`Error`**
  - Codificano errori di programmazione
- **Predefinite e/o definite da utente**
- **Diverse solo per il compilatore**
  - i meccanismi computazionali sono identici

Eccezioni unchecked

- **`RuntimeException`: errori “locali”**

```
NumberFormatException
IllegalArgumentException
NullPointerException
```

- **`Error`: errori della JVM**

```
OutOfMemoryError
```

- **Non è necessario gestirle esplicitamente**
- **Possono essere definite da utente**
  - ma è una pratica sconsigliata …

Eccezioni checked

- **Devono essere gestite esplicitamente dal programma**
- **Due possibilità**
  - catturare l'eccezione
  - documentare che l'eccezione è uno dei possibili risultati dell'esecuzione del metodo

```
public void read(String filename)
                        throws FileNotFoundException
{
   File f = new File (filename);
   Scanner in = new Scanner(f);
   . . .
}
```

Continua…

Documentazione di eccezioni

- **Metodi devono dichiarare tutte le eccezioni checked che possono lanciare (e non catturano)**

```
public void read(String filename)
      throws IOException, ClassNotFoundException
```

- **Gerarchia dei tipi eccezione semplifica la documentazione**
  - Se un metodo può lanciare **`IOException`** e **`FileNotFoundException`** è sufficiente dichiarare **`IOException`** (il supertipo)

Metodi e clausole `throws`

```
accessSpecifier returnType
      methodName(parameterType parameterName, . . .)
            throws ExceptionClass, ExceptionClass, . . .
```

Esempio:

```
public void read(BufferedReader in) throws IOException
```

Scopo:

Indicare le eccezioni checked che un metodo può lanciare

Cattura di eccezioni

- **Catturiamo e trattiamo le eccezioni mediante la definizione di un *exception handler***
- **Costrutto `try/catch`**
  - blocco **`try`** include codice che può lanciare una eccezione (direttamente o invocando metodi che a loro volta lanciano eccezioni)
  - clausola **`catch`** contiene il gestore per un dato tipo di eccezione

Continua…

Cattura di eccezioni

```
try
{
   String filename = . . .;
   File f = new File(filename);
   Scanner in = new Scanner(f);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println("Input was not a number");
}
```

Cattura di eccezioni

- **Flusso di esecuzione:**
  - Esegui il codice nel blocco **`try`**
  - Se non si verificano eccezioni, i gestori vengono ignorati le clausole **`catch`** vengono ignorate
  - Se si verifica una eccezione di uno dei tipi dichiarati, l'esecuzione riprende dal primo gestore compatibile con il tipo dell'eccezione sollevata
  - Se nessuno dei gestori è compatibile con il tipo dell'eccezione lanciata, l'eccezione viene rilanciata automanticamente
    - finchè non viene catturata da un altro gestore
    - oppure arriva al **`main`**

Continua…

Cattura di eccezioni

- **`catch (IOException e) block`**
  - **`e`** viene legata all'oggetto di tipo eccezione lanciato
  - può essere utilizzato in **`block`** per programmare la gestione dell'eccezione
  - esempio:

    ```
    catch (IOException e)
      { e.printStackTrace() }
    ```

    restituisce lo stack di chiamate a partire dal metodo in cui l'eccezione è stata lanciata

Continua…

Cattura di eccezioni

- **I blocchi `try-catch` possono definire un qualunque numero di gestori purchè catturino tipi diversi di eccezioni**
- **Gestori esaminati in ordine testuale**
  - ordinare da sottotipo a supertipo

Continua…

Blocchi `try-catch`: sintassi

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
. . .
```

Domanda

- **Quale è l'output di questo blocco?**

```
try
{
 // codice che lascia sfuggire NumberFormatException
}
catch (ArrayIndexOutOfBoundsException e)
{
   System.out.println("Io non centro");
}
catch (NumberFormatException e)
{
   System.out.println("La prendo io");
}
```

Continua…

Risposta

- **Gestori che non catturano l'eccezione lanciata sono ignorati**
  - output: **`"la prendo io"`**

Continua…

Domanda

- **Quale è l'output di questo blocco?**

```
try
{
  // codice che lascia sfuggire NumberFormatException
}
catch (ArrayIndexOutOfBoundsException e)
{
   System.out.println("Non è per me");
}
catch (ClassCastException e)
{
   System.out.println("Neance per me");
}
```

Continua…

Risposta

- **se nessun gestore è compatibile con il tipo dell'eccezione lanciata, questa**
  - output

```
Exception in thread "main"
 java.lang.NumberFormatException
```

Continua…

Domanda

- **Quale è l'output di questo blocco?**

```
try
{
  // codice che lascia sfuggire NumberFormatException
}
catch (ArrayIndexOutOfBoundsException e)
{
   System.out.println("Non per me");
}
catch (Exception e)
{
   System.out.println("Questa è per me");
}
```

Continua…

Risposta

- **compatibilita di tipi nei gestori basata su sottotipo**
  - output: **“`questa è per me`”**

Domanda

- **Quale è l'output di questo blocco?**

```
try
{
 // codice che lascia sfuggire NumberFormatException
}
catch (NumberFormatException e)
{
  throw new NumberFormatException()
}
catch (Exception e)
{
  System.out.println("Ooops!");
}
```

Risposta

- **gestori catturano solo eccezioni lanciate nel corrispondente blocco `try`:**
  - output

```
Exception in thread “main”
 java.lang.NumberFormatException
```

Eccezioni checked e costruttori

- **I costruttori possono dichiarare eccezioni checked nella lista di throws**
  - ogni metodo che invoca il costruttore deve o catturare l'eccezione o dichiararla nella sua lista di throws.
- **Nota : se invochiamo un costruttore utilizzando `this(...)` o `super(...)` non abbiamo modo di catturare eccezioni (perchè le invocazioni esplicite devono essere la prima istruzione)**
- **Inizializzatori di campi possono invocare metodi solo se questi non dichiarano eccezioni**

Eccezioni checked e overriding

- **Ogni eccezione dichiarata dal metodo nella sottoclasse deve essere un sottotipo di almeno una delle eccezioni dichiarate dal metodo nella superclasse**
- **Quindi il metodo nella sottoclasse può dichiarare una lista di throws con**
  - meno eccezioni e/o
  - eccezioni che sono sottotipi delle eccezioni del metodo nella superclasse

Domanda

- **In quali casi le definizioni delle due classi seguenti compilano?**

```
class X
{
  public void m( ) throws E1, E2 { ....}
}

class SX extends X
{
  public void m( ) throws E1, E21, E22 { .... }
}
```

Risposta

- solo se `E21 <: E2` e `E22<:E2`, oppure
  `E21 <: E1` e `E22 <:E1`

La clausola `finally`

- **Permette di definire codice che viene eseguito sempre e comunque all'uscita dal blocco `try`**

```
reader = new File(filename);
Scanner in = new Scanner(reader);
 . . .
reader.close(); // non eseguito in caso di eccezioni
```

Continua…

La clausola `finally`

```
File f = new File (filename);
try
{
   Scanner in = new Scanner(f);
   . . .
}
finally
{
   f.close(); // la clausola finally viene eseguita comunque
              // prima che l'eventuale eccezione venga
              // passata al gestore
}
```

La clausola `finally`

- **Eseguita all'uscita dal blocco `try-catch` in tre possibili situazioni**
  - dopo l'ultimo comando del blocco `try`
  - dopo l'ultimo comando del gestore che ha catturato l'eccezione (se ne esiste uno)
  - quando l'eccezione non è stata catturata
- **`finally` deve essere l'ultimo (o l'unico) gestorre di un blocco `try-catch`**

Domande

```
FileReader reader = null;
try
{
   reader = new FileReader(filename);
   readData(reader);
}
finally
{
   reader.close();
}
```

- Perché dichiariamo la variabile `reader` fuori dal blocco?
- Quale è il flusso di esecuzione se `filename` non esiste?

Risposte

- Semplicemente per una questione di scope
- Il costruttore `FileReader` lancia una eccezione. Il controllo passa al codice nella clausola `finally` … ma `reader` è `null`, quindi `NullPointerException`

Eccezioni user-defined

- **Normalmente sono checked**

```
public class NotEnoughCreditException extends Exception
{
   public NotEnoughCreditException(String msg)
   {
      super(msg);
   }
}
```

Continua…

Eccezioni user-defined

- utili per validare le precondizioni dei metodi pubblici

```
public class BankAccount
{
   public void withdraw(double amount)
               throws NotEnoughCreditException
   {
      if (amount > balance)
         throw new NotEnoughCreditException("Amount
                                    exceeds balance");
      balance = balance - amount;
   }
   . . .
}
```

Gestori di eccezioni

- **Tre possibilità per strutturare la gestione**
  - propagazione automatica
  - riflessione
  - mascheramento

Gestione di eccezioni – propagazione

- **Metodo più semplice, ma meno efficace**
  - Nessun gestore cattura l'eccezione
  - se l'eccezione è checked dobbiamo dichiararla

Gestione di eccezioni – riflessione

- **Eccezione rilanciata dopo una gestione parziale**
  - possiamo rilanciare la stessa eccezione
  - lanciare una eccezione diversa, di diverso grado di astrazione

Continua…

Gestione di eccezioni – riflessione

- **Esempio: un metodo che cerca il minimo in un array**

```
public static int min(int[] a) throws EmptyException {
   int m;
   try { m = a[0]; }
   catch (IndexOutOfBoundException e)
   {
      throw new EmptyException();
   }
   // scorri l'array alla ricerca del minimo . . .
}
```

Gestione di eccezioni – mascheramento

- **Eccezione catturata e gestita localmente**
  - possiamo rilanciare la stessa eccezione
  - lanciare una eccezione diversa, di diverso grado di astrazione

Continua…

Gestione di eccezioni – mascheramento

- **Un metodo che controlla se un array è ordinato**

```
public static boolean sorted(int[] a) {
   int curr;
   try { curr = a[0];
    // scorri l'array e verifica . . .
   }
   catch (IndexOutOfBoundException e) { return true; }
}
```

Esempio

```
class ServerNotFoundException extends Exception
{
   public ServerNotFoundtException(String reason, int p)
   {
      super(reason); this.port = p;
   }
   public String reason()
   {
       return getMessage(); // ereditato da Throwable
   }
   public int port()
   {
       return port;
   }
   private int port;
 }
```

Continua…

Esempio

```
class HttpClient
{

  public void httpConnect(String server)
                      throws ServerNotFoundException
  {
     final int port = 80;
     int connected = open(server, port);
     if (connected == -1)
       throw new
         ServerNotFoundException("Could not connect", port);
  }
```

Continua…

Esempio

```
public void fetchURL(String URL, String proxy) {
   String server = getHost(URL);
   try {
      httpConnect(server);
   }
   catch (ServerNotFoundException e)
   {
     System.out.println(“Server unavailable: trying proxy”);
     try {
        httpConnect(proxy);
     }
     catch (ServerNotFoundtException e)
     {
        System.out.println(“Failed ” + e.port()+ e.reason());
        throw new Error(“I give up”);
     }
    }
    // accedi alla pagina sul server o sul proxy
 }
```

Domanda

- Assumiamo che il seguente blocco compili correttamente.

```
try
{  . . .
   throw new E1();
}
catch (E2 e)
{
   throw new E2();
}
catch (E1 e)
{
   . . .
}
```

- In quali caso lascia sfuggire una eccezione?

Risposta

In nessun caso

- **Il blocco compila, quindi `E2 <: E1`.**
- **Quindi il primo `catch` non cattura l'eccezione, che viene invece catturata dal secondo blocco**
- **Quindi nessuna eccezione sfugge**

Domanda

- Sotto quali ipotesi il seguente blocco compila correttamente?

```
class A {
      public void m() throws E1 { }
  }
  class B extends A {
      public void m() throws E2 {  throw new E1(); }
  }
```

Risposta

In nessun caso

- Poiché `B.m()` compila, deve valere la relazione `E1 <: E2:`
- D'altra parte, affinché `B` sia una sottoclasse legale di `A` , deve essere `E2 <: E1.`
- Quindi dovremmo avere `E1 = E2`
- Impossibile

Domanda

- **Che relazione deve valere tra E1 ed E2 perchè il blocco lasci sfuggire una eccezione?**

```
try
{
    try
    {
        throw new E1( );
    }
    catch (E2 e)
    {
        throw new E2( );
    }
}
catch (E1 e)
{
    . . .
}
```

Risposta

Lascia sfuggire `E2` solo se `E1 <: E2`

- **se `E2 <:E1`, l'eccezione `E1` lanciata nel blocco interno viene catturata dal `catch` esterno**
- **se `E1 <:E2`, l'eccezione `E1` viene catturata dal `catch` interno, che rilancia `E2`. Poichè `E1 <:E2` questa seconda eccezione non viene catturata.**