Gestione delle eccezioni

Individuazione e ripristino

- Nei programmi possono verificarsi errori e situazioni impreviste
- Esempi:
  - Si cerca di accedere tramite il metodo **`get`** a un elemento di un oggetto **`ArrayList`** che non esiste (fuori dai limiti)
  - Si cerca di interpretare una stringa come un numero intero tramite il metodo **`parseInt`**, ma la stringa non contiene la rappresentazione di un numero intero

Individuazione e ripristino

- Un buon programma dovrebbe gestire il più possibile queste situazioni
- Questo è un compito difficile perché in genere il punto del programma in cui si verifica l'errore è diverso dal punto del programma in cui si hanno tutte le informazioni per gestire l'errore e ripristinare uno stato corretto

Individuazione e ripristino

- Riprendendo gli esempi: i metodi **`get`** di **`ArrayList`** e **`parseInt`**, in cui si può verificare l'errore, non hanno sufficiente informazione per sapere come gestirlo
- Si dovrebbe chiedere all'utente di fare un'altra operazione?
- Si dovrebbe terminare il programma bruscamente dopo aver salvato il lavoro dell'utente?
- Le decisioni di questo tipo sono indipendenti da quello che deve fare il metodo in cui gli errori hanno origine
- È da qualche altra parte, scendendo nella pila delle attivazioni, che si pùo arrivare ad un punto del programma in cui si può gestire la situazione

Eccezioni

- Il meccanismo delle eccezioni fornito da Java è un modo flessibile per realizzare una corretta gestione delle situazioni di errore
- Esso permette di passare il controllo dal punto in cui si verifica l'errore direttamente a un altro punto dove l'errore può essere gestito e uno stato corretto dell'applicazione ripristinato

Segnalazione di errore

- Un approccio semplice al problema può essere quello di segnalare, in un metodo, che è successo qualcosa di sbagliato tramite un valore speciale per il tipo di ritorno
- Ad esempio il metodo **`showInputDialog`** della classe **`JOptionPane`** restituisce **`null`** se l'utente preme Cancel
- Per gestire l'eventuale errore il codice che chiama questo metodo deve preoccuparsi di controllare il valore di ritorno e, nel caso sia **`null`**, fare qualcosa per gestire la situazione

Segnalazione di errore

- Questo semplice approccio presenta i seguenti inconvenienti:
  - Il programmatore è costretto a programmare non solo il comportamento normale, ma tutta una serie di possibili casi di insuccesso
  - Il codice risulta molto complicato e poco leggibile
  - Il programmatore potrebbe dimenticarsi di controllare il valore del tipo di ritorno e l'errore rimarrebbe non gestito
  - La segnalazione di errore da parte di un metodo al suo chiamante potrebbe non bastare: il chiamante potrebbe non avere tutte le informazioni per gestire l'errore e sarebbe costretto a risegnalare esplicitamente l'errore al suo chiamante e così via

Eccezioni

- Un approccio come quello che abbiamo visto può andare bene se usato ogni tanto, ma in generale è bene utilizzare le eccezioni
- Le eccezioni sono state progettate per risolvere i seguenti due problemi:
  1. Le eccezioni **non** devono poter essere trascurate
  2. Le eccezioni devono poter essere gestite da un gestore **competente**, non semplicemente dal chiamante del metodo che fallisce

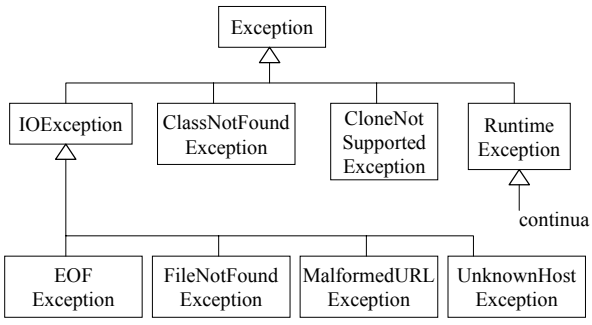
Segnalazione di una eccezione

- Nel caso in cui in un certo punto del codice si verifica una situazione inaspettata tutto quello che bisogna fare è lanciare un'eccezione
- Un'eccezione è un oggetto di una classe
- Per lanciare un'eccezione si usa il comando **`throw`** seguito da un riferimento a un oggetto di una classe appropriata
- Esempio: viene chiesto un prelievo maggiore del saldo in un conto bancario

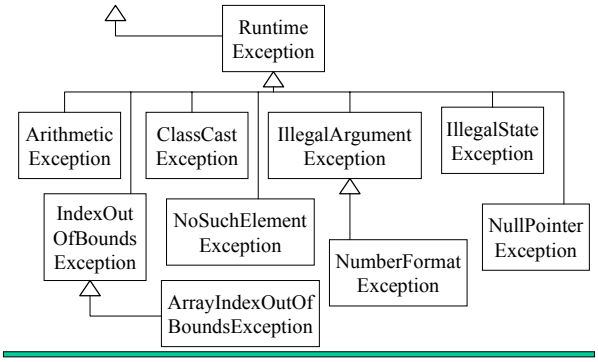
Segnalazione di una eccezione

- La libreria standard di Java mette a disposizione una gerarchia di classi che rappresentano eccezioni di diversa natura
- Basta scegliere quella che fa di più al caso nostro, creare l'oggetto eccezione e lanciarlo
- L'eccezione più adatta a descrivere il problema della richiesta di prelievo maggiore del saldo disponibile è **`IllegalArgumentException`** in quanto è l'argomento, **`amount`**, del metodo a far sorgere il problema

Gerarchia di eccezioni (incompleta)



Gerarchia di eccezioni (incompleta)



Esempio

```
public class BankAccount {
  public void withdraw(double amount) {
    if (balance < amount) {
    // Parametro amount troppo grande
    IllegalArgumentException exception =
           new IllegalArgumentException(
             "Amount exceedes balance");
    throw exception;
    } else ...
```


Oppure

```
public class BankAccount {
  public void withdraw(double amount) {
    if (balance < amount) {
    // Parametro amount troppo grande
    throw new IllegalArgumentException(
             "Amount exceedes balance");
    } else ...
```


Semantica del throw

- Quando un'eccezione viene lanciata con il comando **`throw`** il metodo termina immediatamente (come se si eseguisse un **`return`**)
- Il controllo non torna di default al metodo chiamante, ma viene individuato un gestore dell'eccezione nella pila di attivazioni
- Il controllo viene passato quindi al gestore
- Per ora lasciamo in sospeso il modo in cui questo gestore viene individuato

Quando lanciare le eccezioni?

- Consideriamo il metodo **`readLine`** della classe **`BufferedReader`**
- Sappiamo che quando viene incontrata la fine del file il metodo restituisce **`null`**
- Perché non lancia una **`EOFException`**?
- Il motivo è che il fatto che un file termini **non** è un evento eccezionale!
- È un evento che va previsto poiché si verifica sempre

Quando lanciare eccezioni?

- L'eccezione **`EOFEXception`** va lanciata solo quando la fine del file giunge **inaspettatamente**
- Ad esempio mentre si sta leggendo una sequenza di dati che dovrebbe essere completata da altri dati
- Qui l'errore è dovuto a un caso eccezionale, per esempio il fatto che il file che si sta leggendo sia corrotto

Quando lanciare eccezioni?

- Questa considerazione vale in generale
- Si deve valutare di volta in volta se l'evento che si vuole segnalare è un caso eccezionale oppure una condizione che si verifica sempre e che va comunque gestita
- Solo nel primo caso è opportuno lanciare un'eccezione
- Nel secondo caso la situazione va gestita con il meccanismo semplice della restituzione di valori speciali al metodo chiamante

Eccezioni controllate e non controllate

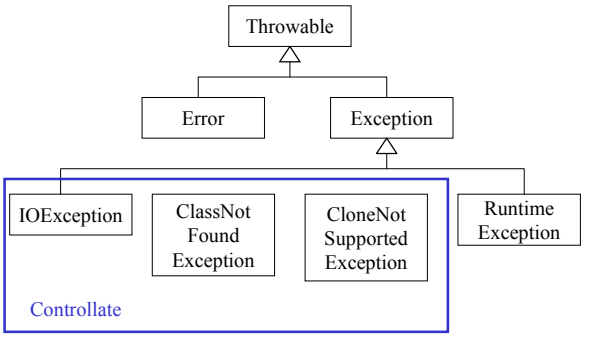
- In Java le eccezioni si dividono in due categorie:

1. Controllate: il compilatore pretende che ogni metodo specifichi cosa fare se una eccezione di questo tipo viene lanciata da un comando al suo interno. Es: **`IOException`** e tutte le sue sottoclassi
2. Non controllate: il compilatore non richiede che si specifichi cosa fare se l'eccezione viene lanciata. Es: **`RuntimeException`** e tutte le sue sottoclassi

Eccezioni controllate e non controllate

- In generale tutte le eccezioni che appartengono a sottoclassi di **`RuntimeException`** non sono controllate
- Tutte le altre eccezioni sottoclassi di **`Exception`** sono invece controllate
- In più esiste un'altra categoria di errori interni rappresentati da oggetti di sottoclassi della classe **`Error`**. Es: **`OutOfMemoryError`**
- Eccezioni ed errori estendono entrambe la classe **`Throwable`** (vedi API)

Gerarchia di Throwable



Perché due tipi di eccezioni?

- Le eccezioni controllate sono pensate per modellare quel tipo di situazioni di errore nelle quali il programmatore non ha responsabilità
- Ad esempio la lettura di un file può interrompersi per cause esterne (un guasto a un hard disk, un collegamento di rete che si interrompe, etc.) sulle quali il programmatore della classe non ha nessun controllo
- Le eccezioni non controllate invece rappresentano un errore del programmatore!
- Una **`NullPointerException`** non dipende da cause esterne: un buon codice dovrebbe controllare se un riferimento è nullo prima di utilizzarlo

Perché due tipi di eccezioni?

- Il compilatore quindi forza a gestire in qualche modo le situazioni che non si possono prevenire
- In realtà, comunque, la distinzione non è nitida: ad esempio l'eccezione **`NumberFormatException`**, tipicamente lanciata dal metodo **`parseInt`** o **`parseDouble`**, è **non** controllata anche se non è certo responsabilità del programmatore se un utente ha inserito una stringa che non rappresenta un numero

Metodi che rilanciano eccezioni

- Abbiamo già visto che il metodo **`readLine`** della classe **`BufferedReader`** può lanciare un'eccezione della classe **`IOException`** o di una delle sue sottoclassi
- Questo tipo di eccezioni sono controllate
- Se in un metodo chiamiamo **`readLine`** su un oggetto della classe **`BufferedReader`** potrà accadere che **`readLine`** sollevi l'eccezione
- Cosa deve fare il nostro metodo in questa eventualità?

Metodi che rilanciano eccezioni

- Il compilatore richiede che il nostro metodo dichiari cosa fare (deve essere specificato nel codice)
- In generale abbiamo due scelte:
  1. Installare un gestore di eccezioni e gestire l'eccezione (vedremo fra poco come si fa)

Metodi che rilanciano eccezioni

2. Decidere che il metodo non ha abbastanza informazioni per gestire competentemente l'eccezione: in questo caso segnaliamo al compilatore che se l'eccezione viene lanciata il metodo dovrà essere immediatamente terminato e l'eccezione "rilanciata" (propagata) al metodo chiamante nella speranza che questo sia in grado di gestirla (naturalmente il metodo chiamante potrà anche lui aver deciso di rilanciare quel tipo di eccezione e così via)

Metodi che rilanciano eccezioni

- Per segnalare al compilatore che il nostro metodo decide di non gestire certi tipi di eccezioni controllate bisogna utilizzare la clausola **`throws`** dopo l'intestazione del metodo:

Rilancio: esempio

```
public class Coin {
  public void read(BufferedReader
           in) throws IOException {
    value = Double.parseDouble(
                    in.readLine());
    name = in.readLine();
  }
...
}
```


Rilancio: altro esempio

```
public class Purse {
  public void read(BufferedReader in)
                  throws IOException {
    while (...) {
      Coin c = new Coin();
      // read di Coin rilancia!
      c.read(in);
      add(c);
    }
...
}
```


Metodi che rilanciano eccezioni

- Il metodo **`read`** della classe **`Coin`** chiama il metodo **`readLine`** della classe **`BufferedReader`** che può sollevare una **`IOException`**
- Decide di non gestire l'eccezione e usa la clausola **`throws`**
- Il metodo **`read`** della classe **`Purse`** chiama il metodo **`read`** della classe **`Coin`** (che può lanciare una **`IOException`**)
- Quindi anche lui è costretto a dichiarare cosa fare dell'eccezione (anche qui non viene gestita)

Metodi che rilanciano eccezioni

- All'interno di un metodo possono verificarsi più eccezioni controllate
- Per segnalare la decisione di non gestire diversi tipi di eccezioni la clausola **`throws`** può essere usata con una lista di nomi di classi di eccezioni separate da virgola:

```
public void m() throws IOException,
                       ClassNotFoundException {
...
```


Metodi che rilanciano eccezioni

- Il fatto che un metodo decida di non gestire una o più eccezioni può sembrare un comportamento irresponsabile
- **Tuttavia** è giusto che sia così se il metodo non ha la possibilità di rimediare correttamente all'errore!
- Pensiamo a un metodo **`read`** di basso livello come quelli visti come esempi
- Come dovrebbe gestire l'**`IOException`**?

Metodi che rilanciano eccezioni

- Dovrebbe stampare un messaggio sullo standard output?
- Ma se la nostra classe si trovasse ad operare in un sistema in cui l'utente non vede affatto lo standard output (es. un distributore automatico)?
- E se l'utente non comprendesse la lingua in cui scriviamo il messaggio?
- Potremmo pensare di riparare mettendo a **`null`** o a zero qualche variabile: ma questo in genere porta a stati errati e alla terminazione (misteriosa!) del programma dopo alcune istruzioni e in una situazione difficile da interpretare!

Metodi che rilanciano eccezioni

- La nostra applicazione potrebbe avere alcuni metodi che sanno esattamente come comunicare un problema all'utente
- Quindi la cosa migliore è permettere all'eccezione di arrivare a uno di questi (tutti gli altri devono rilanciare)
- Il metodo competente potrà catturare l'eccezione e gestirla opportunamente

Progetto di eccezioni

- Quando decidiamo di lanciare un'eccezione possiamo sceglierne una fra le classi predefinite
- Tuttavia la nostra situazione di errore potrebbe non essere adeguatamente descritta da nessuna delle classi predefinite
- Non c'è problema: possiamo definire una nostra classe di eccezione che descrive esattamente la situazione

Progetto di eccezioni

- Ad esempio per il conto bancario:

```
if (amount > balance)
  throw new
     InsufficientFundsException(
       "withdrawal of " + amount +
       " exceeds balance of " +
       balance);
else ...
```

- La classe **`InsufficientFundsException`** va definita

Dichiarazione di un nuovo tipo di eccezione

- Per prima cosa bisogna decidere se la nuova eccezione dovrà essere controllata oppure no
- Nel nostro caso il programmatore può benissimo controllare se l'importo richiesto supera il saldo, prima di chiamare il metodo **`withdraw`**
- Quindi questo tipo di errore rientra nella categoria delle eccezioni non controllate
- Per far questo basta definire la classe come estensione di **`RuntimeException`**:

Dichiarazione di un nuovo tipo di eccezione

```
public class InsufficientFundsException
             extends RuntimeException {
  public InsufficientFundsException() {
  }
  public
     InsufficientFundsException(
                            String reason) {
    super(reason);
  }
}
```


Dichiarazione di un nuovo tipo di eccezione

- In genere per una classe eccezione vengono forniti due costruttori:
  - Uno senza parametri che non fa nulla
  - Uno con una stringa come parametro. La stringa dovrebbe descrivere la situazione di errore
- Per creare una nuova eccezione controllata basta estendere una delle classi di eccezioni controllate

Gestori di eccezioni

- Tutte le eccezioni che si possono verificare in una certa applicazione dovrebbero essere gestite da qualche parte
- Se un'eccezione non ha nessun gestore allora il programma termina bruscamente stampando la pila di attivazioni che l'eccezione ha attraversato prima di far terminare il programma

Gestori di eccezioni

- Un gestore di eccezioni si installa con un blocco **`try-catch`**
- Se un comando che si trova all'interno di un blocco **`try`** lancia un'eccezione allora il tipo dell'eccezione viene confrontato con i tipi elencati nelle clausole **`catch`** associate al blocco **`try`**
- Se uno dei tipi indicati nelle clausole **`catch`** è compatibile (uguale o superclasse) con il tipo dell'eccezione lanciata allora la propagazione dell'eccezione viene fermata, viene eseguito il blocco di codice associato alla **`catch`** e l'esecuzione continua con ciò che segue il blocco **`try-catch`**

Gestori di eccezioni

```
try {
  BufferedReader in = new
      BufferedReader(new
          InputStreamReader(System.in));
  System.out.println(
                "How old are you?");
  String inputLine = in.readLine();
  int age = Integer.parseInt(inputLine);
  age++;
  System.out.println(
        "Next year you'll be " + age);
} continua
```

Gestori di eccezioni

```
catch (IOException exception) {
  System.out.println(
             "Input/Output exception " +
             exception);
}
catch (NumberFormatException exception)
  {
  System.out.println(
              "Input was not a number");
}
```


Gestori di eccezioni

- In questo semplice esempio catturiamo eccezioni di tipo **`IOException`** (può essere lanciata da **`readLine`**) e **`NumberFormatException`** (può essere lanciata da **`parseInt`**) e di loro eventuali sottotipi
- La gestione, semplice in questo caso, consiste solo nell'informare l'utente dell'accaduto
- Si potrebbe ad esempio migliorare la gestione di **`NumberFormatException`** permettendo all'utente di riprovare a inserire il numero

Gestori di eccezioni

- La clausola **`catch(`***Type*** `e)`** *Blocco* viene attivata solo quando all'interno del blocco **`try`** uno dei comandi lancia un'eccezione di tipo *Type* (o di un sottotipo di *Type*)
- L'attivazione consiste nell'assegnare alla variabile **`e`** il riferimento all'oggetto eccezione che è stato lanciato e nell'eseguire il blocco di comandi
- Dopo di ciò l'esecuzione prosegue normalmente dopo il blocco **`try`**

Gestori di eccezioni

- Su oggetti di tipo eccezione possono essere chiamati alcuni metodi standard come **`printStackTrace()`** che stampa tutte le attivazioni che sono state attraversate dall'eccezione prima di arrivare al presente blocco **`try`** in cui è stata catturata
- Consultare le API di **`Throwable`**, **`Exception`**, **`Error`** per altri metodi

Consigli

- Evitare di definire gestori di eccezioni troppo generici (ad esempio con clausola **`catch(Exception e)`** o addirittura **`catch (Throwable t)`**)
- Resistere alla tentazione di zittire il compilatore, che lamenta la mancata gestione di un'eccezione controllata, inserendo un gestore di eccezioni che non fa nulla
- Cercare sempre di gestire nella maniera migliore e nel punto più appropriato le diverse eccezioni che possono essere lanciate nella propria applicazione

La clausola finally

- A volte è utile poter specificare che alcune operazioni devono essere compiute sia nel caso di esecuzione normale che nel caso di lancio di eccezione
- Questo è quello che fa la clausola **`finally`**
- Essa può essere abbinata ad un blocco **`try-catch`** e, in qualunque modo il controllo lasci il blocco (return, lancio di un'eccezione non gestita, cattura di un' eccezione, terminazione normale), i comandi all'interno del blocco associato a **`finally`** vengono comunque **sempre** eseguiti (immediatamente prima dell'abbandono del blocco)

La clausola finally

- È molto utile per effettuare operazioni necessarie come ad esempio la chiusura di file:

```
BufferedReader in;
try {
  in = new BufferedReader(
          new FileReader(fileName));
  purse.read(in);
}
finally {
if (in != null) in.close();
}
```


La clausola finally

- Sia che la lettura vada a buon fine sia che invece ci sia qualche intoppo (es. **`NumberFormatException`**, etc.) il file verrà sempre e comunque chiuso

Esempio completo

- Scriviamo un programma che chiede all'utente di inserire il nome di un file di testo contenente la descrizione di monete, aggiunge tutte le monete a un borsellino e poi stampa il valore totale
- Cosa può andare storto?
  - Il file potrebbe non esistere
  - Il file potrebbe contenere dati in formato errato

Esempio completo

- Chi può porre rimedio a questi errori?
- Il metodo **`main`** della classe test interagisce con l'utente e quindi è in esso che si ha la possibilità di gestire in maniera adeguata questi errori informando l'utente e permettendogli eventualmente di reinserire dei dati
- In esso dovremmo quindi mettere il gestore delle eccezioni

Metodo read della classe Coin

```
public boolean read(BufferedReader in)
                    throws IOException {
  String input = in.readLine();
  if (input == null)
  // Non ci sono più monete da leggere
  // Segnalo con false in uscita
  return false;
  value = Double.parseDouble(input);
  name = in.readLine()
  if (name == null)
    // Situazione anomala non prevista!
     throw new EOFException(
                  "Coin name expected");
  return true;
}
```


Metodo read della classe Coin

- Il metodo rilancia tutte le **`IOException`** che possono essere lanciate da **`readLine()`**
- In più lancia una **`EOFException`** in caso di fine file inattesa
- Il metodo distingue tra una fine di file attesa (tutti i file prima o poi terminano) - dopo la descrizione dell'ultima moneta - e una fine di file inattesa - nel mezzo della descrizione di una moneta
- Nel primo caso segnala la cosa semplicemente restituendo **`false`**, mentre nel secondo caso lancia giustamente l'eccezione **`EOFException`**

Metodo read della classe Purse

```
public boolean read(
            BufferedReader in)
                throws IOException {
  boolean done = false;
  while (!done) {
    Coin c = new Coin();
    if (c.read())
      add(c);
    else done = true;
}
```


Metodo read della classe Purse

- Il metodo rilancia tutte le eccezioni **`IOException`** che possono essere lanciate dal metodo **`read`** di **`Coin`**
- Il caso di fine attesa del file viene gestito invece in maniera tradizionale
- Il metodo legge tutte le monete dal file e le aggiunge al borsellino

Metodo readFile della classe Purse

```
public void readFile(String fileName)
                throws IOException {
  BufferedReader in = null;
  try {
  in = new BufferedReader(
             new FileReader(fileName));
  read(in)
  }
  finally { if (in != null) in.close()
  }
}
```


Metodo readFile della classe Purse

- Tenta di aprire il file e di chiamare il metodo read
- In ogni caso il file viene chiuso se era stato precedentemente aperto

Main

```
boolean done = false;
String fileName = JOptionPane.showInputDialog(
                      "Enter File name");
while(!done) {
  try {
    Purse myPurse = new Purse();
    myPurse.readFile(fileName);
    System.out.println("Total: " +
                      myPurse.getTotal());
    done = true;
  }
```

continua

Main

```
  catch (IOException e) {
    System.out.println(
                "Input/Output error: " + e);
  }
  catch (NumberFormatException e) {
    e.printStackTrace();
  }
  if (!done) {
    fileName = JOptionPane.showInputDialog(
                            "Try another file");
    if (fileName == null) done = true;
  }
}
```

Main

- Chiede all'utente il nome di un file, crea un nuovo borsellino e lo riempie con le monete la cui descrizione si trova nel file. Infine stampa il valore totale di tutte le monete
- Se qualcosa va storto stampa il tipo di errore che si è verificato e dà la possibilità all'utente di inserire un altro nome di file (l'utente può anche premere Cancel per terminare comunque).