

Programmazione Orientata agli Oggetti

Gestione delle Eccezioni,
Gestione File (cenni)

Concetti

- Gestione di eccezioni
- Gestione di file

Alcune cause di errore

- Implementazione non corretta
 - L'applicazione non è conforme alle specifiche
 - cfr "Analisi e Progettazione del Software"
- Errori logici
 - cfr "Fondamenti di Informatica"
- Un oggetto può trovarsi in uno stato inconsistente o inappropriato
 - ...

Non sempre errori del programmatore

- Alcune situazioni anomale possono essere causate dall'ambiente "esterno" al programma
 - URL o nome file errato
 - Hard Disk pieno
 - Interruzione di rete
 - Mancanza di permessi appropriati per una risorsa (es. file, dbms)
 - ...
- Situazioni "eccezionali", che però potremmo prevedere e di gestire

Defensive programming

- Un programma OO può essere visto come una interazione tra oggetti "client" e oggetti "server"
- Gli oggetti server offrono servizi (metodi) invocati dagli oggetti client
- In questa prospettiva
 - Se un oggetto server "fallisce" come comunica tale situazione anomala all'oggetto client? A fronte di un fallimento del server che cosa fa il client?

Problemi

- Se un oggetto server riscontra una anomalia che gli impedisce di portare a termine il suo compito, come deve riportare gli errori al client?
- Se un oggetto server fallisce il suo compito, l'oggetto client come può/deve gestire l'errore?

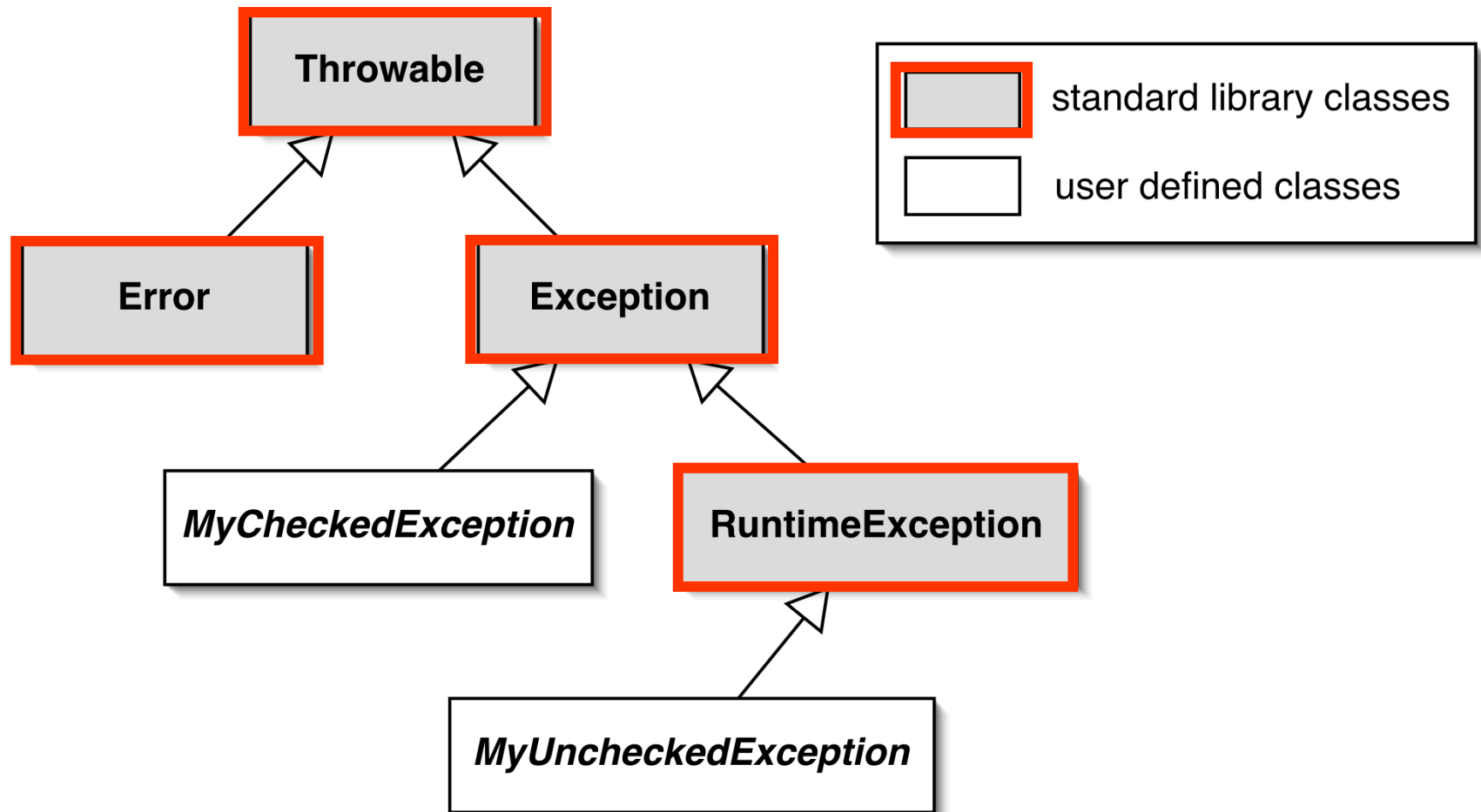
Gestione delle eccezioni

- Una caratteristica del linguaggio
- Meccanismo che consente di gestire situazioni anomale "eccezionali", ma che possono occorrere (ed essere previste)
- In particolare
 - Le eccezioni sono uno strumento attraverso il quale un oggetto server, in caso di anomalie interrompe il flusso di esecuzione e comunica attraverso un oggetto (*lancia un'eccezione*) tale situazione al client
 - Il linguaggio supporta le eccezioni con costrutti per imporre al programmatore del client di scrivere codice per la gestione dell'eccezione (cioè per costringere il programmatore del client a non ignorare l'eccezione lanciata dal server) e per implementare eventuali azioni di ripristino

Gestione delle eccezioni

- Idea generale
 - Se si verifica una condizione anomala, un metodo può interrompere il flusso dell'esecuzione e "lanciare" una eccezione al client
 - L'eccezione è un oggetto e come tale può contenere informazioni sulla natura dell'errore
 - Quando il client riceve un'eccezione, a seconda del tipo di eccezione, può **ignorarla** oppure può essere **costretto a gestirla**

La gerarchia delle eccezioni Java



Categorie di eccezione

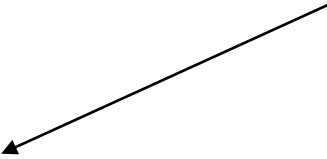
- Error
 - veri e propri errori, non gestibili come casi eccezionali, dovuti a fattori esterni (es. fine memoria)
- Checked exception
 - Sottoclasse di **Exception**
 - Il client è costretto a gestire questo tipo di eccezione
 - In genere si usa quando è possibile un ripristino
- Unchecked exception
 - Sottoclasse di **RuntimeException**
 - Il client non è obbligato a gestire l'eccezione
 - Se non viene gestita il programma abortisce

Gestione eccezioni

```
class ClasseClient {  
    ClasseServer server;  
    public void metodoClient() {  
        int i;  
        i = server.metodoServer();  
    }  
}
```

```
class ClasseServer {  
    public int metodoServer() {  
        ...  
    }  
}
```

Qualcosa può
andare storto in
questo metodo



Lanciare un'eccezione

- Se qualcosa va storto, lanciamo un'eccezione
- Per lanciare un'eccezione:
 - Viene costruito l'oggetto "eccezione":
`new ExceptionType("... ");`
 - L'oggetto eccezione viene *lanciato* con l'istruzione:
`throw eccezione`
- Nota: documentazione Javadoc:
`@throws ExceptionType motivo`


La clausola **throws**

- I metodi che lanciano una checked exception devono dichiararlo
- Sulla base di questa dichiarazione il compilatore verifica che i client che usano questo metodo hanno previsto codice per la gestione dell'eccezione
- Sintatticamente, per dichiarare che un metodo lancia una eccezione, si usa una clausola **throws**

Gestione eccezioni

```
class ClasseClient {  
    ClasseServer server;  
    public void metodoClient() {  
        int i;  
        i = server.metodoServer();  
    }  
}
```

Se qualcosa va
storto questo metodo
lancia una eccezione
MiaEccezione



```
class ClasseServer {  
    public int metodoServer() throws MiaEccezione{  
        ...  
        if (qualcosa è andato storto)  
            throw new MiaEccezione();  
    }  
}
```

L'effetto di una eccezione

- Il metodo che lancia una eccezione finisce prematuramente
- Non viene ritornato nessun valore
- Il controllo non ritorna al punto di chiamata del client, ma il client può catturare e gestire l'eccezione

Gestione della eccezione

- Nel caso di **checked exception** il client è **obbligato** a gestire l'eccezione
 - Ovvero il programmatore deve scrivere codice che specifichi come comportarsi in caso di eccezione
 - Il compilatore verifica che il programmatore abbia scritto il codice di gestione della eccezione
- Nel caso di una **unchecked exception**
 - Il compilatore non fa nessuna verifica
 - Se non vengono gestite causano la terminazione del programma. Un esempio che già conosciamo:
NullPointerException

Checked exception

- Le checked exception **devono** essere gestite dall'oggetto client
- La verifica della gestione dell'eccezione da parte del client viene svolta dal compilatore
- Se il client non gestisce l'eccezione il programma non viene compilato

Gestione dell'eccezione

- Il client che chiama un metodo che dichiara di lanciare una eccezione deve "gestire" l'eccezione
- Per la gestione di una eccezione il client ha due possibilità
 - non se ne preoccupa direttamente, ma delega a sua volta la gestione dell'eccezione al metodo chiamante
 - cattura e gestisce direttamente l'eccezione

Delegare la gestione dell'eccezione

- In questo caso il metodo entro il quale viene fatta la chiamata ad un metodo che può sollevare l'eccezione deve a sua volta dichiarare che potrebbe sollevare un'eccezione (di quel tipo)
- Non sempre questa è una buona pratica

Gestione eccezioni

```
class ClasseClient {  
    ClasseServer server;  
    public void metodoClient() throws MiaEccezione {  
        int i;  
        i = server.metodoServer();  
    }  
}
```

Qualcosa può andare storto
In questo caso lancio
(in realtà propago)
una eccezione
MiaEccezione

```
class ClasseServer {  
    public int metodoServer() throws MiaEccezione{  
        ...  
        if (qualcosa è andato storto)  
            throw new MiaEccezione();  
    }  
}
```

Catturare e gestire l'eccezione

- Per gestire un'eccezione
 - le chiamate ad un metodo che lancia una checked exception devono essere effettuate all'interno di un blocco **try**
 - l'eventuale eccezione viene catturata e "gestita" in blocco **catch**

Gestione eccezioni

```
class ClasseClient {  
    ClasseServer server;  
    public void metodoClient() {  
        int i;  
        try {  
            i = server.metodoServer();  
        }  
        catch (MiaEccezione e) {  
            ... codice gestione situazione anomala  
        }  
        ...  
    }  
}
```

Provo a chiamare
un metodo che dichiara
di lanciare una eccezione
se qualcosa va storto

Se il metodo ha sollevato
una eccezione
MiaEccezione
gestisco la situazione
come segue

```
class ClasseServer {  
    public int metodoServer() throws MiaEccezione{  
        ...  
        if (qualcosa è andato storto)  
            throw new MiaEccezione();  
    }  
}
```

Catturare eccezioni multiple

- Un metodo di un oggetto server potrebbe lanciare diversi tipi di eccezione (corrispondenti a diversi tipi di anomalie)
- Il client può gestire diversamente queste situazioni

Catturare eccezioni multiple

```
try {  
    ...  
    ref.process();  
    ...  
}  
catch(ExType1 e) {  
    // Take action on a pb1 exception.  
    ...  
}  
catch(ExType2 e) {  
    // Take action on a pb2 exception.  
    ...  
}  
catch(ExType3 e) {  
    // Take action on a pb3 exception.  
    ...  
}
```


Catturare eccezioni multiple

- Se viene generata un'eccezione
 - il gestore delle eccezioni cerca il primo blocco **catch** con argomento che corrisponde al tipo di eccezione sollevata
 - esegue le istruzioni del blocco **catch** selezionato
 - l'eccezione è considerata gestita
- Attenzione all'ereditarietà
 - Un supertipo nasconde i sottotipi

Attenzione all'ereditarietà

```
try {  
    ...  
    ref.process();  
    ...  
}  
catch(Exception e) {  
    // Take action on an end-of-file exception.  
    ...  
}  
catch(ExType2 e) {  
    // Take action on a file-not-found exception.  
    ...  
}  
catch(ExType2 e) {  
    // Take action on a file-not-found exception.  
    ...  
}
```

*qualsiasi eccezione
verrebbe catturata
dal primo catch !*

La clausola **finally**

- Il meccanismo di gestione delle eccezioni è completato dal blocco **finally**
- Il codice nel blocco **finally** viene eseguito sempre, anche se l'eccezione non è stata rilevata
 - anche se nel blocco **try** o **catch** c'è una istruzione **return** !
- Tipicamente serve a chiudere file o connessioni (ad esempio ad un DBMS)

```
try {  
    ...  
}  
catch(Exception e) {  
    ...  
}  
finally {  
    azioni che verranno eseguite in ogni caso  
}
```

Definire nuove eccezioni

- Ogni eccezione estende **Exception** o **RuntimeException**
- Definire nuovi tipi di eccezioni permette di fornire al chiamante informazioni diagnostiche
 - Include informazioni utili per decidere come gestire l'eccezione

Definire nuove eccezioni

```
public class MiaExeption extends Exception {  
  
    public MiaException(String message) {  
        super(message);  
    }  
  
}
```

Ripristino degli errori

- In caso di checked exception il client non può ignorare l'eccezione
- Può tentare di ripristinare il programma in uno stato valido
 - usando eventuali informazioni passate "dentro" l'eccezione

Linee guida per la gestione delle eccezioni

Per approfondimenti: <http://www.javaworld.com/javaworld/jw-07-1998/jw-07-techniques.html>

- Se il metodo incontra una condizione fuori dal normale che non può gestire, allora dovrebbe lanciare una eccezione
- Evitare di usare eccezioni per indicare condizioni che possono ragionevolmente essere attese come parte del normale funzionamento del metodo
- Se il tuo metodo scopre che il client ha violato gli obblighi stabiliti dal contratto (ad esempio, passando parametri errati), lancia una unchecked exception
- Se il tuo metodo non riesce a rispettare il contratto, allora lancia una checked o unchecked exception (ATTENZIONE QUESTO PUNTO ULTIMAMENTE È STATO MESSO IN DISCUSSIONE: SECONDO MOLTI, LE ECCEZIONI DOVREBBERO ESSERE TUTTE UNCHECKED, COME IN C#)
- Se stai lanciando una eccezione per una condizione anormale, che pensi il programmatore del client possa decidere di gestire, allora lancia una checked exception (COME SOPRA)
- Definisci una exception class (o scegli una già esistente) per ciascun tipo di condizione anormale che potrebbe spingere il tuo metodo a lanciare una eccezione

Gestione dell'IO in Java

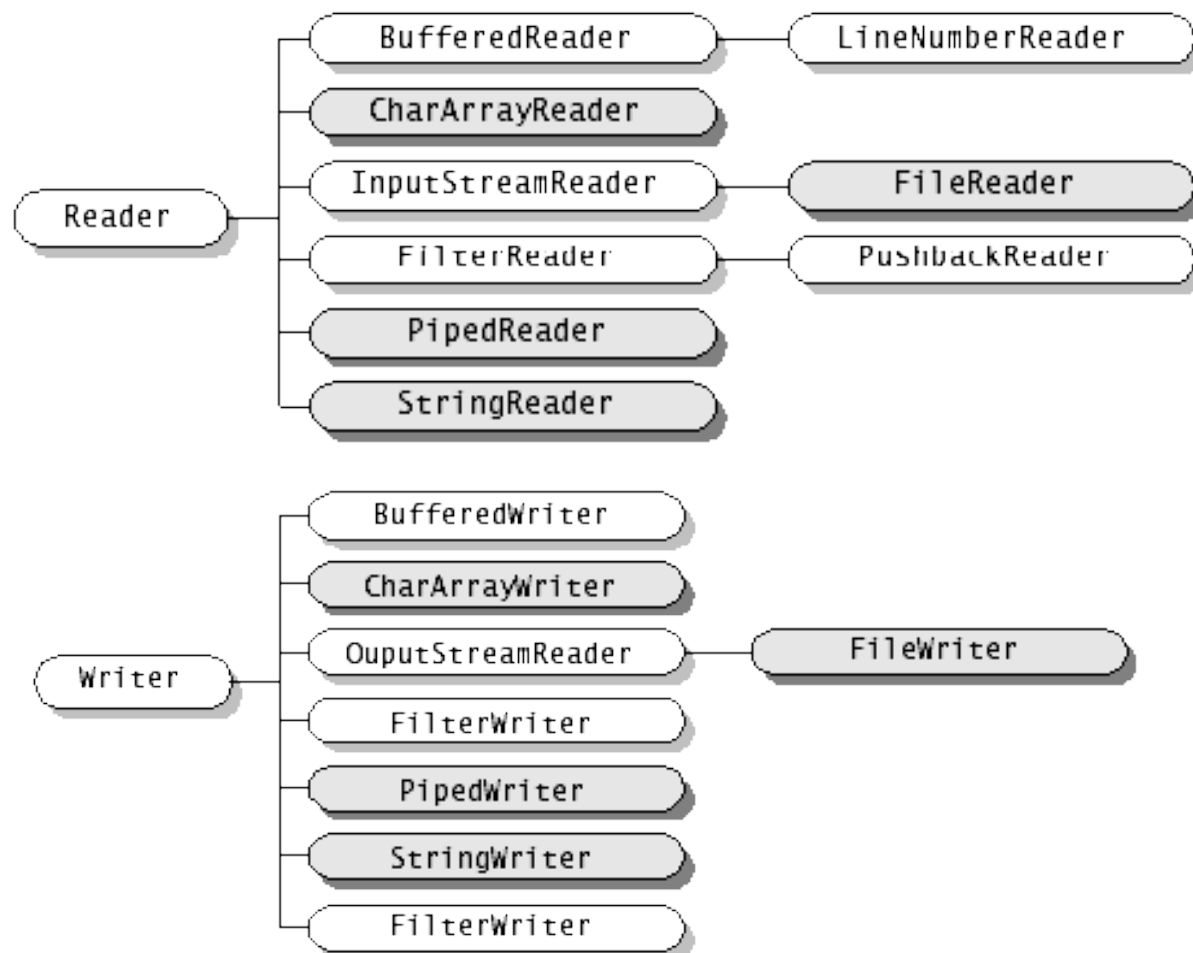
- Il package IO di Java
- L'IO è definito in termini di *flussi* (stream)
 - Stream: sequenza ordinata di dati che hanno una sorgente (flussi di ingresso) o una destinazione (flussi di uscita)
 - Stream di caratteri (testi)
 - Stream di byte (es. immagini, binari, etc)
- Per gestire gli stream abbiamo classi che agiscono da "lettori" e "scrittori"
 - Stream di caratteri: Reader e Writer
 - Stream di byte: InputStream, OutputStream

Gestione dell'IO in Java

- Premessa.
 - Per semplicità facciamo riferimento solo agli stream di caratteri (ma tutti i concetti hanno un duale molto simile per gli stream di byte)
- Possiamo avere molti tipi di sorgenti (e destinazioni) per uno stream
 - Esempi: Stringhe, File
- La libreria offre classi (di lettori e scrittori) per gestire le varie tipologie
 - Esempi: `StringReader`, `StringWriter`
`FileReader`, `FileWriter`

Gestione dell'IO

- Una gerarchia molto ricca



Gestione IO

- I principali metodi di **Reader**

```
int read()
```

```
int read(char cbuf[])
```

```
int read(char cbuf[], int offset, int length)
```

- E i principali metodi di **Writer**

```
int write(int c)
```

```
int write(char cbuf[])
```

```
int write(char cbuf[], int offset, int length)
```

- Attenzione: tutti questi metodi lanciano una **IOException**

VEDERE javadoc

Gestione IO

- In generale il modo di operare sugli stream è il seguente:

Reading

```
open a stream
while more information
    read information
close the stream
```

Writing

```
open a stream
while more information
    write information
close the stream
```

Gestione File

- Se la sorgente del nostro stream di dati è un file, possiamo usare `FileReader` e `FileWriter`
- Vediamo un programmino che legge da un file e copia in un altro

```
import java.io.*;
public class Copy {
    public static void main(String[] args)
        throws IOException {
        FileReader in = new FileReader("fileIn.txt");
        FileWriter out = new FileWriter("fileOut.txt");
        int c;
        while ((c = in.read()) != -1)
            out.write(c);
        in.close();
        out.close();
    }
}
```

Gestione eccezioni e gestione file nello studio di caso

- Potremmo scrivere i dati relativi al labirinto in un file, che viene caricato all'inizio del gioco
- Il file potrebbe essere un file di testo, con una semplice sintassi:

Stanze:

biblioteca

N10

N11

Estremi:

N10

N11

Attrezzi:

martello 10 biblioteca

pinza 2 N10

Uscite:

biblioteca nord N10

biblioteca sud N11

Gestione eccezioni e gestione file nello studio di caso

- Nella classe **Labirinto** mettiamo un nuovo costruttore, che inizializza il labirinto caricandolo da file.
 - "Ovviamente" l'operazione di caricare i dati la deleghiamo ad un opportuno oggetto

```
public Labirinto(String nomeFile) {  
    CaricatoreLabirinto c =  
        new CaricatoreLabirinto(nomeFile);  
    c.carica();  
    this.stanzaIniziale = c.getStanzaIniziale();  
    this.stanzaVincente = c.getStanzaVincente();  
}
```


Il caricatore di labirinto

- Vedi classe CaricatoreLabirinto.java