

Programmazione Orientata agli Oggetti

Gestione delle Eccezioni

Sommario

- Introduzione alle eccezioni
- Gestione eccezioni in Java
 - Lato Client
 - Lato Server
- La gerarchia delle eccezioni Java
 - *Checked vs Unchecked* Exception
- Testing ed eccezioni
- Conclusioni

Alcune Frequenti Cause di «Anomalie»

- Talvolta le esecuzioni possono risultare «anomale». Alcune delle cause più frequenti:
 - ✓ Implementazione non corretta
 - l'applicazione non è conforme alle specifiche
 - Cfr. *Analisi e Progettazione del Software*
 - ✓ Errori logici
 - Cfr. *Fondamenti di Informatica*
- A causa di queste anomalie un oggetto può trovarsi in uno stato *inconsistente*, ovvero non più rappresentativo delle istanze nel dominio che intende modellare

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 esterna (ad es. file, connessioni verso DBMS)
 - ...
- Situazioni «eccezionali» che è però possibile gestire per contenerne gli effetti negativi, e migliorare le possibilità di recupero
- ✓ N.B. eccezionali non significa *inverosimili*!

Eccezioni


- Uno strumento supportato da alcuni linguaggi specificatamente per gestire situazioni anomale
- Consentono l'implementazione del codice di gestione di situazioni anomale e/o «eccezionali»
 - difficile ottenere soluzioni eleganti per gestire situazioni *eccezionali*
 - per definizione, sono tutte diverse una dall'altra!
- La gestione del caso anomalo, va inquadrata allo stesso modo del caso ordinario?
- Come evitare di offuscare il codice per la gestione del caso *ordinario* con il codice per la gestione dei casi *eccezionali* ?

Oggetti «Client» e «Server» (1)

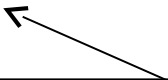
- Programma OO: descrivono l'interazione tra oggetti «client» e oggetti «server»
 - Gli oggetti *server* offrono servizi (metodi)
 - Vengono erogati (invocati) su richiesta degli oggetti *client*
- E' lecito porsi almeno queste domande:
 - ✓ se un oggetto server «fallisce», come comunica l'anomalia all'oggetto client?
 - ✓ a fronte di un fallimento del server:
 - come se ne accorge il client?
 - cosa può fare e come recupera la situazione?

Oggetti «Client» e «Server» (2)

```
class ClasseClient {  
    private ClasseServer server;  
    public void metodoClient() {  
        int i;  
        i = server.metodoServer(0);  
    }  
}
```



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



Metodo in cui si possono
verificare situazioni *anomale*

Eccezioni in Java

- La gestione delle eccezioni nella piattaforma Java prevede dei meccanismi attraverso i quali è possibile:
 - «sollevare» eccezioni:
 - per segnalare ai client la situazione anomala interrompendo il normale flusso di esecuzione
 - «catturare» eccezioni:
 - per implementare eventuali azioni di gestione dell'anomalia
 - «imporre» la gestione:
 - il client è forzato a prendere in carico esplicitamente la gestione dell'eccezione sollevate dal server

Eccezioni vs Oggetti

- Idea generale: le eccezioni sono oggetti
 - estendono `java.lang.Throwable`
 - ✓ vedi gerarchia dei tipi delle eccezioni Java (>>)
- Possono essere «sollevati» / «lanciati» esplicitamente utilizzando l'istruzione **throw**:
throw new IllegalArgumentException()
- è possibile (consultare i javadoc):
 - allegare messaggi diagnostici all'eccezione
 - incapsulare eventuali *altre* eccezioni all'origine dell'eccezione sollevata >>
 - conoscere lo *stack-trace* dell'esecuzione che ha sollevato l'eccezione

java.lang.Throwable

Costruttori:

Throwable() *Constructs a new throwable with null as its detail message.*

Throwable(String message)

Constructs a new throwable with the specified detail message.

Throwable(String message, Throwable cause)

Constructs a new throwable with the specified detail message and cause.

Alcuni dei metodi più significativi:

Throwable getCause()

Returns the cause of this throwable or null if the cause is nonexistent...

String getMessage()

Returns the detail message string of this throwable.

StackTraceElement[] getStackTrace()

*...programmatic access to the stack trace ... by **printStackTrace()***

void printStackTrace()

Prints this throwable and its backtrace to the standard error stream.

Es.: IllegalArgumentException

- Utilizzata ad indicare argomenti che violano il *contratto di un metodo*
- Ad es. usata da: `Collections.nCopies()`:

```
public static <T> List<T> nCopies(int n, T o)
```

returns: an immutable list consisting of n copies of the specified object. **throws**: `IllegalArgumentException` - if $n < 0$
- I costruttori ricalcano quelli di `java.lang.Throwable`:

`IllegalArgumentException(String s)`

Constructs an `IllegalArgumentException` with the specified detail message.

`IllegalArgumentException(String message, Throwable cause)`

Constructs a new exception with the specified detail message and cause.

`IllegalArgumentException(Throwable cause)`

Constructs a new exception with the specified cause ...

Lato Server: Lanciare un'Eccezione con throw

- Per indicare che si sono rilevate anomalie, si lancia un'eccezione; per lanciare un'eccezione:
 - Prima viene costruito l'oggetto **Throwable**:
`e = new IllegalArgumentException("eta>0") ;`
 - Poi l'oggetto viene “lanciato” con l'istruzione:
`throw e;`
 - Spesso, più direttamente:
`throw new IllegalArgumentException("eta>0") ;`
- I tipi di eccezioni che un metodo lancia possono divenire parte integrante della sua stessa segnatura (>>) mediante la clausola **throws**
 - ✓ nei javadoc basta scrivere: `@throws ExType <descrizione>`

Istruzione throw & Clausola throws

```
class ClasseClient {  
    private ClasseServer server;  
    public void metodoClient() {  
        int i;  
        i = server.metodoServer(0);  
    }  
}
```

Equivale ad affermare che
questo metodo può lanciare
una `IllegalArgumentException`

```
/**  
 * @throws IllegalArgumentException  
 */  
class ClasseServer {  
    public int metodoServer(int arg)  
        throws IllegalArgumentException {  
        if (arg<=0) {  
            throw new IllegalArgumentException("arg>0");  
        }  
    }  
    ...}
```



Effetti di una Eccezione

- Il metodo che lancia una eccezione finisce prematuramente
 - ✓ senza eseguire l'istruzione **return**
- Non viene restituito nessun valore
- Il controllo (lato client) NON ritorna al punto di chiamata (da parte del client) del metodo del server
- ✓ A tutti gli effetti, viene interrotta ed abbandonata la *normale* sequenza di esecuzione a favore di una sequenza di esecuzione *eccezionale*

Eccezioni che «Bucano» gli Stack

- Le eccezioni «bucano» lo stack, ovvero causano:
 - la terminazione del metodo che le lanciano (e quindi la rimozione dallo stack del relativo *r.d.a.*, cfr. FdI)
 - la riattivazione del metodo (client) invocante, al quale viene in effetti restituito il controllo, che può esercitare in vari modi (>>)
- Le eccezioni, se non gestite, arrivano sino al metodo da cui l'intera esecuzione è cominciata
 - ovvero, per programmi eseguiti da riga di comando, sino al metodo `main()`
 - se non viene gestita nemmeno al livello iniziale, l'esecuzione del programma da parte della JVM abortisce
 - la JVM termina: stampa *stack-trace* e *messaggio di errore allegato all'eccezione*

Lato Client: Gestione dell'Eccezione

- Se un metodo client chiama un metodo server che lancia una eccezione, allora può/deve «gestirla»
- Esistono due alternative per la gestione dal lato metodo client di una eccezione sollevata dal lato metodo server:
 - I. Presa in carico diretta:* cattura e gestione dell'eccezione
 - II. Rinuncia alla gestione diretta e propagazione verso il metodo chiamante:* si rimanda la gestione al livello immediatamente precedente (nelle nidificazioni delle chiamate di metodo)

Cattura e Gestione Diretta (1)

- Per catturare un'eccezione
 - le chiamate ad un metodo che lancia una eccezione che si vuole catturare devono essere effettuate all'interno di un blocco `try {...}`
 - l'eventuale eccezione viene catturata e gestita in un blocco `catch(...) {...}`

```
try {  
    <blocco codice che può sollevare un'eccezione>  
} catch (ExceptionType e) {  
    <blocco codice di gestione eccezione>  
}
```

Cattura e Gestione Diretta (2)

```
class ClasseClient {  
    ClasseServer server;  
    public void metodoClient() {  
        int i;  
        try {  
            i = server.metodoServer(0);  
        }  
        catch (IllegalArgumentException e) {  
            ... «codice gestione eccezione e»  
        }  
        ...  
    }  
}
```

Chiamata ad un metodo che
dichiara di lanciare
IllegalArgumentException

Codice di gestione di una eccezione
IllegalArgumentException
«catturata» nella variabile locale **e**

```
class ClasseServer {  
    public int metodoServer(int arg)  
        throws IllegalArgumentException {  
        if (arg<=0)  
            throw new IllegalArgumentException("arg>0");  
        ...  
    }  
}
```

Pericoloso Errore Metodologico

- Eliminare le eccezioni ignorandole!
 - Siccome sulla riga `o.metodo()` si osserva una *N.P.E.*, allora si è tentati di risolvere il problema con

```
try {  
    o.metodo();  
} catch (NullPointerException npe) {  
    /* ignora e vai dritto */  
}
```
 - Classica “toppa peggio del buco”:
 - non solo non si è affatto risolto il problema: lo si sta anche rendendo ancora più difficile da individuare!
 - ✓ sarà più arduo capirne la vera origine dell’errore con un’eccezione in più a “nasconderla”!
-

Cattura e Gestione Diretta: FabbricaDiComandi

- In precedenza per lo studio di caso: nella gerarchia con radice in **FabbricaDiComandi** abbiamo cambiato la segnatura del metodo **costruisciComando()**

- ✓ (<<) per non distogliere l'attenzione dall'*introspezione*

- Ora il metodo non dichiara più di lanciare eccezioni

```
public interface FabbricaDiComandi {  
    public Comando costruisciComando(String istruzione)  
        throws Exception;  
}
```

- Gestiamo *direttamente* le eventuali eccezioni sollevate
 - nel corpo del metodo di **FabbricaDiComandiRiflessiva**
 - ✓ risulta migliorata la distribuzione delle responsabilità?
-

FabbricaDiComandiRiflessiva

```
public class FabbricaDiComandiRiflessiva implements FabbricaDiComandi {
    @Override
    public Comando costruisciComando(String istruzione) {
        Scanner scannerDiParole = new Scanner(istruzione);
        String nomeComando = null;
        String parametro = null;
        Comando comando = null;

        if (scannerDiParole.hasNext())
            nomeComando = scannerDiParole.next(); // prima parola: nome del comando
        if (scannerDiParole.hasNext())
            parametro = scannerDiParole.next(); // seconda parola: eventuale parametro
        try {
            String nomeClasse = "it.uniroma3.diadia.comandi.Comando";
            nomeClasse += Character.toUpperCase(nomeComando.charAt(0));
            nomeClasse += nomeComando.substring(1);
            comando = (Comando) Class.forName(nomeClasse).newInstance();
            comando.setParametro(parametro);
        } catch (Exception e) {
            comando = new ComandoNonValido();
            System.out.println("Comando inesistente");
        }
        return comando;
    }
}
```

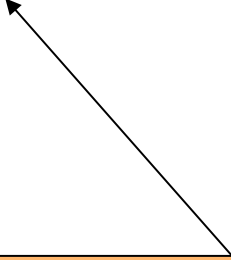
Propagazione al Chiamante (1)

- In alternativa, il metodo client a sua volta può dichiarare che *risolleva* l'eccezione lanciata dal metodo server di cui fa uso
- In effetti si limita a «propagarla» al *suo* metodo chiamante, non gestendolo direttamente
- Quando questo tipo di gestione è opportuna?
 - se esisteva quella clausola era proprio perché si consigliava/richiedeva una gestione esplicita
 - ✓ d'altra parte talvolta il chiamante *diretto* può essere in una posizione meno favorevole per una gestione corretta e per una diagnostica più efficace rispetto ad un chiamante *indiretto*

Propagazione al Chiamante (2)

```
class ClasseClient {  
    ClasseServer server;  
    public void metodoClient()  
        throws IllegalArgumentException {  
        int i;  
        i = server.metodoServer(0);  
    }  
}
```

Le eccezioni sollevate nel corpo
del metodo sono propagate al suo
metodo invocante, a livello più alto



```
class ClasseServer {  
    public int metodoServer(int arg)  
        throws IllegalArgumentException {  
        ...  
    }  
}
```

Propagazione al Chiamante (3)

- Quando questo tipo di gestione è opportuna?
 - se esisteva quella clausola era proprio perché si consigliava/richiedeva una gestione esplicita
 - ✓ d'altra parte talvolta il chiamante *diretto* può essere in una posizione meno favorevole per una gestione corretta e per una diagnostica più efficace rispetto ad un chiamante *indiretto*
- ✓ Si pensi ad un errore dovuto a un input errato che si manifesta in una chiamata di metodo molto “profonda”: può essere conveniente rileggere l'input ad un livello molto più alto

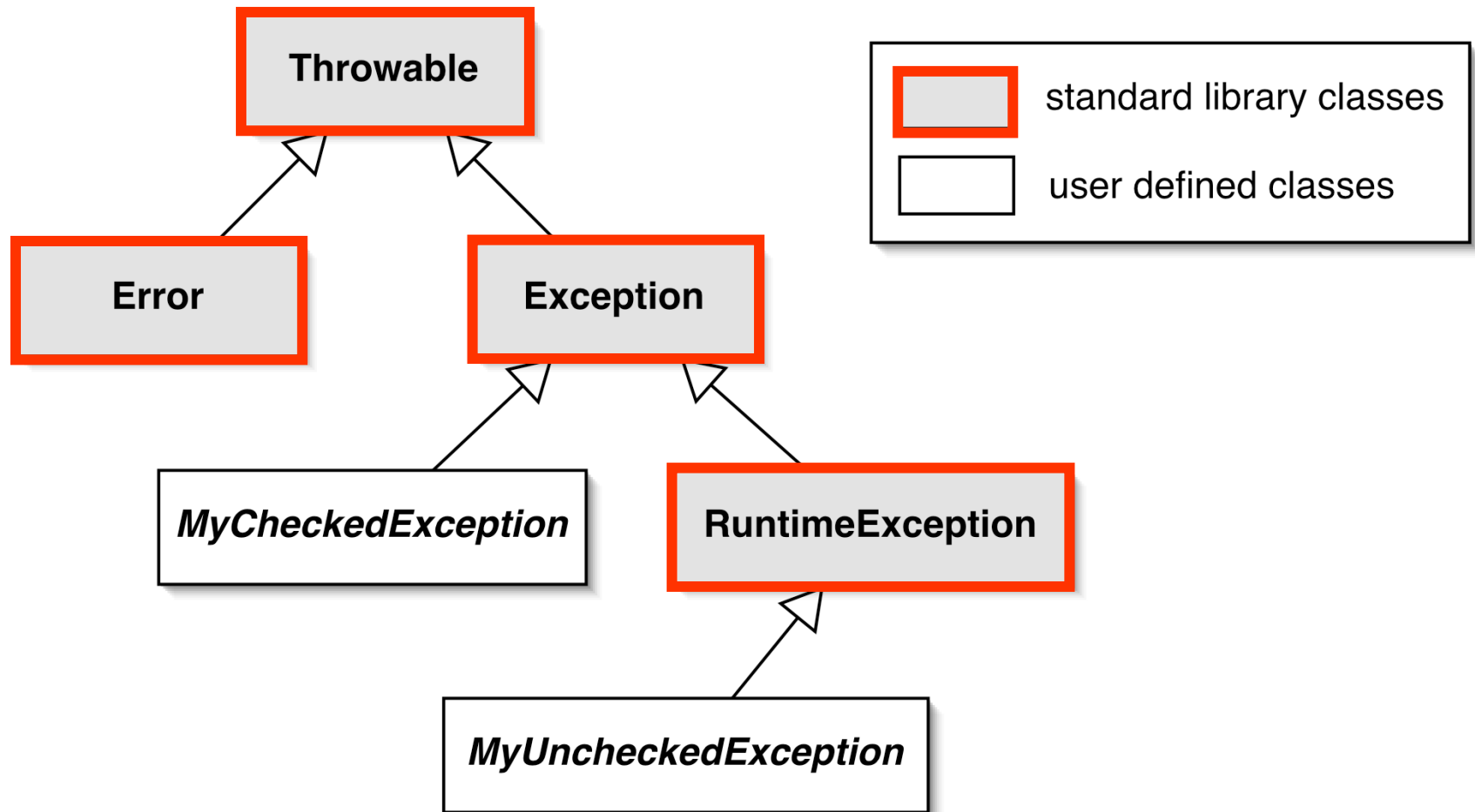
Definizione di Nuovi Tipi di Eccezione

- Parte integrante della normale attività di progettazione
- Fa parte della definizione dei *tipi del dominio*
 - ✓ forniscono al chiamante informazioni diagnostiche tenendo conto del dominio applicativo
 - ✓ chiariscono la libreria da cui si è originato il problema

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

- Classi che definiscono nuove tipologie di eccezione devono estendere **Exception** o **RuntimeException**, e quindi essere sottotipi di **java.lang.Throwable**

Gerarchia delle Eccezioni Java (1)



Gerarchia delle Eccezioni Java (2)

- *Error*
 - veri e propri errori, non recuperabili, dovuti a fattori esterni (ad es. esaurimento memoria)
- *Checked exception*
 - sottoclasse di **Exception**
 - il client **deve** gestire esplicitamente questo tipo di eccezioni: il compilatore altrimenti si rifiuta di compilare
 - si usa quando è possibile e conveniente «forzare» una gestione lato client
 - un esempio: **FileNotFoundException**
- *Unchecked exception*
 - sottoclasse di **RuntimeException**
 - client non obbligato (durante la compilazione) a gestirle
 - un esempio: **IllegalArgumentException**

Checked vs Unchecked Exception (1)

- Nel caso di *checked exception*, il client è obbligato a gestire l'eccezione
 - il programmatore **deve** specificare il blocco **try-catch** che cattura e gestisce l'eccezione, oppure propargarla esplicitamente
 - ✓ si forza la gestione lato client
 - in caso contrario il codice non compila

```
Main.java:7: error: unreported exception
FileNotFoundException; must be caught or declared to
be thrown
```
 - un esempio ben noto: **FileNotFoundException**
- Nel caso di una *unchecked exception*
 - il compilatore non esegue verifiche
 - se non gestite esplicitamente, sono automaticamente propagate al metodo chiamante
 - un ben noto esempio: **NullPointerException**

Checked vs Unchecked Exception (2)

- I metodi che lanciano una *checked* exception
 - **devono** dichiararlo esplicitamente
- I metodi che lanciano una *unchecked* exception
 - **possono** dichiararlo esplicitamente
- *Sintatticamente* si usa in entrambi i casi sempre la stessa clausola **throws** ad integrazione della dichiarazione della segnatura di un metodo
- ✓ *Semanticamente* solo le checked exception sono considerate parte integrante della segnatura
 - per rendersene conto, basta dichiararle nei metodi di una interfaccia
 - Solo le checked possono far fallire un override

Diatriba *Checked* vs *Unchecked* (1)

- Le checked exception hanno generato vere e proprie diatribe sulla loro opportunità di esistere, con analisi dei benefici e dei costi ben diverse, di volta in volta, ma sempre opinabili
- Sta di fatto che capita di scrivere:

```
public static final URL createURL(String url) {  
    try {  
        return new URL(url);  
    } catch (MalformedURLException e) {  
        throw new RuntimeException(e, "Cannot create URL from: " + url);  
    }  
}
```

...per incapsulare la *checked* `MalformedURLException` dentro ad una *unchecked* `RuntimeException` e non essere più obbligati alla gestione esplicita imposta dalla classe `java.net.URL` che fa ampio uso di checked exception

Diatriba *Checked* vs *Unchecked* (2)

- La più recente `java.net.URI` (da Java 4) recepisce la questione: lascia `java.net.URISyntaxException` checked ma allo stesso tempo offre un metodo factory statico per nascondere dietro una unchecked:

```
public URI(String str) throws URISyntaxException
```

```
public static URI create(String str)  
Creates a URI by parsing the given string.
```

*This convenience factory method works as if by invoking the `URI(String)` constructor; **any `URISyntaxException` thrown by the constructor is caught and wrapped in a new `IllegalArgumentException` object, which is then thrown.***

This method is provided for use in situations where it is known that the given string is a legal URI, for example for `URI` constants declared within in a program, and so it would be considered a programming error for the string not to parse as such. The constructors, which throw `URISyntaxException` directly, should be used situations where a `URI` is being constructed from user input or from some other source that may be prone to errors.

Unchecked Exception: *Generate vs Programmatiche*

- `ArrayIndexOutOfBoundsException`
- `ClassCastException`
- `NullPointerException`
- Sono tutti esempi di ben note unchecked exception generate direttamente anche dalla JVM
- Non c'è modo (e/o interesse) a distinguerle da quelle generate programmaticamente, ad es.:
`throw new NullPointerException();`

try-catch Multipli

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

```
try {  
    server.metodoServer();  
}  
catch (ExType1 e) {  
    «Gestione eccezione in e di tipo ExType1»  
}  
...  
catch (ExTypeN e) {  
    «Gestione eccezione in e di tipo ExTypeN»  
}
```

try-catch Multipli: Esempio

- Se viene generata un'eccezione:
 - il primo (e solo il primo) blocco `catch` il cui argomento è associabile al tipo di eccezione sollevata viene attivato
 - le istruzioni del suo blocco `catch` sono eseguite
 - l'eccezione è considerata servita

```
try {  
    comando = (Comando)Class.forName(nomeClasseComando).newInstance();  
} catch (InstantiationException e) {  
    /* possibile causa: lo sviluppatore si è dimenticato di aggiun-  
    gere un costruttore no-args in una sottoclasse di Comando */  
} catch (IllegalAccessException e) {  
    /* possibile causa: lo sviluppatore si è dimenticato di rendere  
    pubblico il costruttore no-args di un sottoclasse di Comando */  
} catch (ClassNotFoundException e) {  
    /* possibile causa: comando ignoto - errore digitazione utente */  
}
```

Attenzione: P.d.S. & Clausole catch

- ✓ Quindi un supertipo finisce per «nascondere» i suoi sottotipi che lo seguono lungo nella catena di blocchi

catch

```
try {  
    server.metodoServer();  
}  
catch (Exception e) { ←  
    // <gestione di una generica exception>  
    ...  
}  
catch (IOException e) {  
    // <gestione generica I/O exception>  
    ...  
}  
catch (FileNotFoundException e) {  
    // <gestione di una file-not-found exception>  
    ...  
}
```


qualsiasi
eccezione
verrebbe
catturata
dal primo
catch!

**FileNotFoundException è sottotipo di
IOException!**

Ordinamento delle Clausole catch

- ✓ Ordinare sempre le clausole `catch` dal tipo più specifico catturato a quello meno specifico

```
try {  
    server.metodoServer();  
}  
catch (FileNotFoundException e) {  
    // <gestione di una file-not-found exception>  
    ...  
}  
catch (IOException e) {  
    // <gestione generica I/O exception>  
    ...  
}  
catch (Exception e) {  
    // <gestione di una generica exception>  
    ...  
}
```



Catch Disgiuntive

- In Java 7 furono introdotte sintassi abbreviate per alleviare l'eccessiva verbosità dei costrutti inerenti la gestione delle eccezioni:

```
try {  
    « codice che solleva diverse eccezioni »  
} catch (IllegalArgumentException |  
        IOException |  
        IllegalStateException e ) {  
    « codice comune per la gestione »  
    « di tutti i tipi di eccezione »  
}
```

- Qual'è il tipo statico di `e`?

try-with-resources Statement (1)

- Da Java 7 è stata introdotta anche una sintassi abbreviata per la gestione di «risorse»
- *Risorsa*: in questo contesto si intende un qualsiasi oggetto con protocollo di utilizzo che preveda il rilascio esplicito:
 - «richiesta oggetto/risorsa»
 - «utilizzo»
 - «rilascio oggetto/risorsa»
- Il rilascio prevede l'invocazione del metodo `close()`
 - Cfr. interface `java.io.Closeable`
- utili per interagire con risorse limitate e gestite dal S.O. che le API Java «nascondono» dietro un oggetto, ad. es. **File**

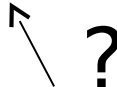
try-with-resources Statement (2)

- E' lecito scrivere (da Java 7+):

```
static String readFistLine(String path) throws IOException {  
    try (BufferedReader br =  
        new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }  
}
```

- Al posto del più verboso e poco leggibile ma corretto (per quanto concerne la gestione delle risorse):

```
static String readFirstLine(String path) throws IOException {  
    BufferedReader br = null;  
    try {  
        br = new BufferedReader(new FileReader(path));  
        return br.readLine();  
    } finally {  
        if (br!=null) br.close();  
    }  
}
```



Clausola `finally`: Utilizzo

```
try {  
    ...  
}  
catch (Exception e) {  
    ...  
}  
finally {  
    « blocco di codice sempre eseguito »  
}
```


Clausola `finally`: Semantica

- Il meccanismo di gestione delle eccezioni è completato dal blocco `finally` {...}
- Il blocco `finally` viene eseguito *sempre*, anche se l'eccezione non è stata rilevata
 - ✓ Attenzione, *sempre* è *sempre*: anche se nel blocco `try` o `catch` c'è una istruzione `return` che viene eseguita
- Tipicamente serve a garantirsi il rilascio di risorse costose (come file, o connessioni ad un DBMS) *anche* in presenza di situazioni anomale
 - ✓ previene la “perdita di risorse” («resource leak»)
 - ✓ le eccezioni sollevate potrebbero impedire il corretto rilascio delle risorse già richieste

Linee Guida Gestione Eccezioni (1)

- ✓ Se il metodo incontra una condizione anomala che non sa gestire, allora dovrebbe lanciare un'eccezione
- ✓ Evitare l'uso di eccezioni solo fornire risultati
- ✓ Se un metodo scopre che il client ha violato il suo contratto (ad es. inviandogli argomenti errati), meglio che sollevi una unchecked exception, ad es. una `IllegalArgumentException`
- ✓ Per approfondimenti:

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

Linee Guida Gestione Eccezioni (2)

- ✓ Se un metodo non riesce a rispettare il contratto, allora sollevi una checked o una unchecked exception
 - *Punto oggetto di accese discussioni: meglio usare solo eccezioni unchecked? (come in C# ???)*
- ✓ Se si lancia una eccezione per una anomalia, che si ritiene il programmatore del client *possa voler gestire*, allora si sollevi una checked exception
 - *(discussioni come sopra)*
- ✓ Si definisca una nuova classe exception (o si riusi una già esistente) per ciascun distinto tipo di condizione anomala che potrebbe spingere un metodo a lanciare eccezioni
 - Un progetto spesso possiede una «capostipite»

Eccezioni e Testing

- JUnit (4+) supporta direttamente la scrittura di test-case per verificare che un metodo sollevi eccezioni
- Utile ogni qualvolta si vuole verificare che il proprio codice sappia gestire correttamente situazioni «anomale»
- Il test ha successo se e solo se l'eccezione specificata nell'annotazione `@Test` viene sollevata:

```
@Test(expected=NoSuchElementException.class)
```

```
public void testMinOfEmptyCollectionNotDefined() {  
    final List<Comparable<Object>> empty =  
        Collections.emptyList();  
    Collections.min(empty);  
}
```

Consultare javadoc:
`Collections.min()` su
collezione vuota *DEVE* sollevare
`NoSuchElementException`

Conclusioni

- Java quando fu introdotto innovò (ad es. rispetto al C++) i meccanismi offerti per la gestione delle situazioni anomale
- In Java le anomalie sono gestite tramite oggetti particolari chiamati *eccezioni*
- Il fatto che dopo molti anni i nuovi linguaggi di programmazione non abbiano introdotto alternative valide, tutto sommato è una garanzia sulla bontà delle scelte fatte
 - le diatribe *checked vs unchecked* possono quasi essere viste anche come una “conferma”

Conclusioni

- La gestione delle anomalie risulta parte integrante, tutt'altro che trascurabile, della normale attività di programmazione
- La modellazione dei diversi tipi di anomalie è parte integrante della normale attività di modellazione del dominio
- I framework a supporto della scrittura di test di unità, come JUnit 4+, permettono di verificare il comportamento di metodi che sollevano eccezioni