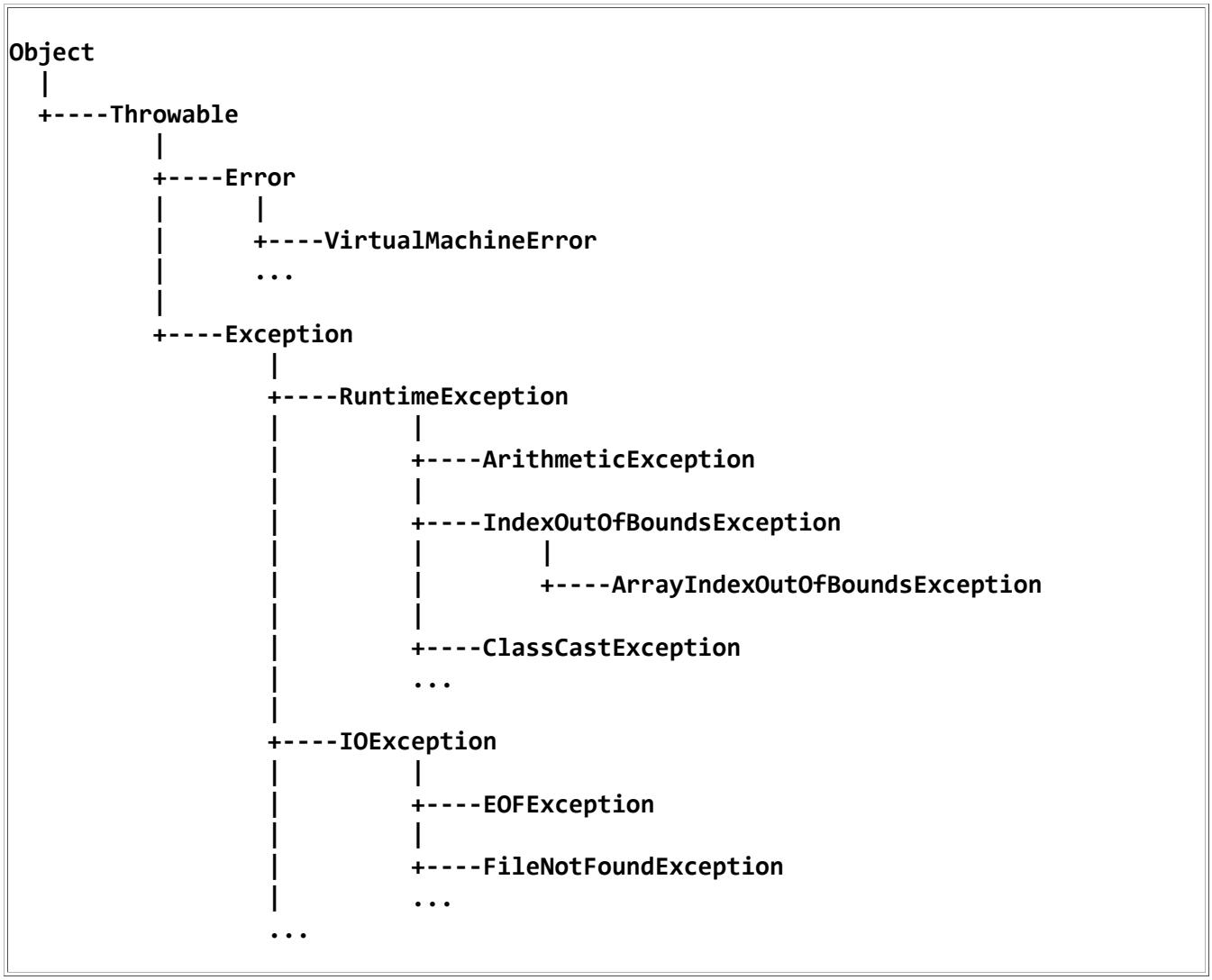
Eccezioni: ricapitolando...

Java fornisce una ricca gerarchia di classi di eccezioni:



Abbiamo visto che:

- Quando si verifica un imprevisto, il metodo attivo **lancia (throw)** un'eccezione che viene passata al metodo chiamante. Il metodo attivo termina l'esecuzione (come con `return`).
- Per default, un metodo che riceve un'eccezione termina l'esecuzione e passa l'eccezione al metodo chiamante.
- Quando l'eccezione raggiunge `main`, l'esecuzione del programma termina stampando un opportuno messaggio di errore.

Ma un metodo chiamante può essere progettato in modo da:

1. **catturare (catch)** un'eccezione lanciata da un metodo invocato;

2. trattarla con opportune istruzioni;

3. proseguire l'elaborazione senza terminare disastrosamente.

Ad esempio, se in un ciclo che legge dati da internet cade la connessione, è naturale gestire questa situazione da programma senza causarne necessariamente la terminazione.

Catturare un'eccezione: `try` e `catch`

Se una chiamata di metodo può generare un'eccezione, possiamo racchiuderla in un blocco `try`, seguito da uno o più blocchi `catch` contenenti le istruzioni da eseguire in corrispondenza dell'eccezione lanciata.

Esempio: Stampa di un array usando eccezioni (`CatchOutOfBounds`):

```
public class CatchOutOfBounds {
    public static void main(String [] args) {
        int [] array = new int [5];
        for (int i = 0; i < array.length; i++) {
            array[i] = (int) (100 * Math.random());
        }

        System.out.println("Contenuto dell'array:");

        try {
            int i = 0;
            while (true)
                System.out.println(array[i++]);
        }
        // catch (Throwable e) {
        // catch (Exception e) {
        // catch (RuntimeException e) {
        // catch (IndexOutOfBoundsException e) {
        catch (ArrayIndexOutOfBoundsException e) {
        // catch (Error e) {
            System.out.println("Stampa terminata...");
        }
    }
}
```

Attenzione: è questo è un PESSIMO stile di programmazione!!!

Sintassi di `try-catch-finally`

```
try {
   <istruzioni-try>; // possono lanciare delle eccezioni
}
catch(<sottoclasse-Throwable1> e1) {
   // catturiamo l'eccezione e1 di tipo <sottoclasse-Throwable1>
   <istruzioni-catch1>; // gestiamo e1
}
catch(<sottoclasse-Throwable2> e2) {
   // catturiamo l'eccezione e2 di tipo <sottoclasse-Throwable2>
   <istruzioni-catch2>; // gestiamo e2
}
...
catch(<sottoclasse-ThrowableN> eN) {
   // catturiamo l'eccezione eN di tipo <sottoclasse-ThrowableN>
   <istruzioni-catchN>; // gestiamo e2
}
finally {
   // istruzioni da eseguire comunque
   <istruzioni-finally>;
}
```

Se è presente almeno un blocco `catch`, allora il blocco `finally` è facoltativo.

Significato di `try-catch-finally`

- Si eseguono le *`<istruzioni-try>`*.
- Se l'esecuzione termina senza fallimenti si eseguono le eventuali *`<istruzioni-finally>`* e poi si prosegue ad eseguire la prima istruzione successiva al blocco *`try-catch`*.
- Altrimenti, se l'esecuzione di *`<istruzioni-try>`* lancia un'eccezione `except`, si cerca il PRIMO BLOCCO `catch` tale che `except` sia istanza di *`<sottoclasse-ThrowableX>`*.
- Se un tale blocco esiste, si eseguono le *`<istruzioni-catchX>`* dopo aver associato `except` all'identificatore `eX` (in questo caso si dice che **l'eccezione è stata catturata con successo**); poi si eseguono le eventuali *`<istruzioni-finally>`* e infine si prosegue ad eseguire la prima istruzione successiva al blocco *`try-catch`*.
- Se invece `except` non è istanza di nessuna *`<sottoclasse-ThrowableX>`*, allora VENGONO COMUNQUE ESEGUITE le eventuali *`<istruzioni-finally>`*, ma poi l'eccezione viene passata al metodo chiamante (ed il metodo attivo viene terminato con fallimento).

Note:

- Le eventuali `<istruzioni-finally>` vengono eseguite ***sempre***, anche in presenza di un `return`

del blocco `try` o `catch`. Il blocco `finally` può contenere delle istruzioni che chiudono dei files oppure rilasciano delle risorse, per garantire la consistenza dello stato.

- I costrutti `try-catch-finally` possono essere annidati a piacere. Esempio: se in un blocco `catch` o `finally` può essere generata un'eccezione, si possono racchiudere le sue istruzioni in un altro blocco `try`.

Esempi di `try-catch`

Il programma seguente chiede una sequenza di interi (terminata da 0) stampandone il quadrato:

```
public class StampaQuadrati {
    public static void main(String[] args) {
        int d;
        do{
            System.out.print("Dammi un numero intero (0 per terminare): ");
            d = Input.readInt();
            if (d != 0)
                System.out.println("Il quadrato di " +
                        d + " e': " + (d*d) + ".");
        }while(d != 0);
        System.out.println("Bye bye...");
    }
}
```

Se durante l'esecuzione forniamo un input sbagliato, viene lanciata un'eccezione:

```
Dammi un numero intero (0 per terminare): 3
Il quadrato di 3 e': 9.
Dammi un numero intero (0 per terminare): tre
Exception in thread "main" java.lang.NumberFormatException: For input string: "tre"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:447)
        at java.lang.Integer.parseInt(Integer.java:497)
        at Input.readInt(Input.java:40)
        at StampaQuadrati.main(StampaQuadrati.java:6)
```

Per evitarlo, possiamo racchiudere il comando di input in un blocco `try-catch`:

```
public class StampaQuadratiCatch {
    public static void main(String[] args) {
        int d = 0;
        try{
            do{
                System.out.print("Dammi un numero intero (0 per terminare): ");
                d = Input.readInt();
                if (d != 0)
                    System.out.println("Il quadrato di " + d + " e': " + (d*d) + ".");
```

```
            }while(d != 0);
            System.out.println("Bye bye...");
        }catch(NumberFormatException e){
            System.out.println("Eccezione: " + e);
        }
    }
}
```

Ora il programma non lancia eccezioni, ma si ferma se l'input non è corretto:

```
Dammi un numero intero (0 per terminare): 3
Il quadrato di 3 e': 9.
Dammi un numero intero (0 per terminare): tre
Eccezione: java.lang.NumberFormatException: For input string: "tre"
```

Per ignorare l'input errato e continuare l'esecuzione, dobbiamo mettere il `try-catch` ***all'interno del ciclo***, nel modo seguente:

```
public class StampaQuadratiRobust {
    public static void main(String[] args) {
        int d = 0;
        do{
            try{
                System.out.print("Dammi un numero intero (0 per terminare): ");
                d = Input.readInt();
                if (d != 0)
                    System.out.println("Il quadrato di " + d + " e': " + (d*d) + ".");
            }catch(NumberFormatException e){
                System.out.println("Input non valido: ritenta...");
            }
        }while(d != 0);
        System.out.println("Bye bye...");
    }
}
```

Ora il ciclo continua anche in presenza di input errato:

```
Dammi un numero intero (0 per terminare): 3
Il quadrato di 3 e': 9.
Dammi un numero intero (0 per terminare): tre
Input non valido: ritenta...
Dammi un numero intero (0 per terminare): 4
Il quadrato di 4 e': 16.
Dammi un numero intero (0 per terminare): 0
Bye bye...
```

Eccezioni controllate e non controllate

Le eccezioni si dividono in due categorie:

- Eccezioni **controllate (checked)**
- Eccezioni **non controllate (unchecked)**

Le eccezioni ***controllate*** DEVONO essere gestite esplicitamente dal programma, altrimenti il compilatore segnalerà un errore.

Ogni volta che scriviamo un'istruzione che potrebbe lanciare un'eccezione ***controllata***, allora

- l'istruzione deve essere racchiusa in un blocco `try-catch` che possa gestire quel tipo di eccezione;

oppure

- il metodo che contiene l'istruzione deve ***delegare*** la gestione dell'eccezione al chiamante, con la clausola `throws` che adesso vediamo.

Sintassi della clausola `throws`

La clausola `throws` viene inserita nella dichiarazione del metodo per informare il compilatore che durante l'esecuzione di quel metodo possono essere generate eccezioni (controllate) dei tipi elencati dopo la parola chiave `throws`, la cui gestione viene delegata al chiamante.

```
<modificatori> <tipo> <nome-metodo>  (<lista-parametri>)
               throws  <Classe-ecc1>, ..., <Classe-eccN> {

   <corpo-metodo>

}
```

Importante: La clausola **`throws`** è sintatticamente simile ma ha un significato diverso dal comando **`throw`** !!!

Cattivo esempio: Per evitare di dover gestire le eccezioni potremmo scrivere tutti i metodi così:

```
public void metodo() throws Exception {
    <istruzioni>;
}
```

O magari fingere di gestire ogni genere di eccezione in modo che il compilatore non segnali errori:

```
public void metodo() {
   try {
      <istruzioni>;
   }
   catch (Exception e) { } // catturata ma non gestita!
}
```

Così si perde molta informazione sul programma. Ignorare un problema non porta mai alla sua soluzione!

Quali eccezioni sono controllate?

- Sono ***non controllate*** tutte le eccezioni che estendono `Error` e `RuntimeException`
- Sono ***controllate*** (dal compilatore) tutte le altre eccezioni.

```
Object
  |
  +----Throwable
          |
          +----Error
          |       |
          |       +----VirtualMachineError
          |       ...
          |
          +----Exception
                  |
                  +----RuntimeException
                  |          |
                  |          +----ArithmeticException
                  |          |
                  |          +----IndexOutOfBoundsException
                  |          |
                  |          +----ClassCastException
                  |          ...
                  |
                  +----IOException
                  |          |
                  |          +----EOFException
                  |          |
                  |          +----FileNotFoundException
                  |          ...
                  |
                  +----ClassNotFoundException
                  ...
```

Perché?

- Una istanza di `Error` non è prevedibile, e comunque non può essere gestita da programma.

- Una eccezione di tipo `RunTimeException` può verificarsi ovunque nel programma: un controllo esplicito appesantisce i programmi senza aumentare l'informazione.
  Corrispondono spesso a errori logici del programma (codice "non robusto" per mancanza di controlli).

- Il controllo esplicito richiesto per gli altri tipi di eccezioni consente di localizzare rapidamente le parti del programma che potrebbero lanciarle.

Riassumendo:

- Il compilatore verifica che le eccezioni controllate vengano gestite.

- Le eccezioni controllate sono dovute a circostanze esterne che anche il più attento dei programmatori non può escludere semplicemente scrivendo codice robusto.

- Le eccezioni non controllate sono dovute a errori fatali che non ha senso prevedere nel codice (es. `OutOfMemoryException`, `StackOverflowError`) oppure ad errori logici (es. `ClassCastException`, `IllegalArgumentException`, `NumberFormatException`, `IndexOutOfBoundsException`, `NullPointerException`) che il programmatore dovrebbe gestire con controlli opportuni, senza dover ricorrere al meccanismo delle eccezioni.

Definire le proprie eccezioni

- Java fornisce una ricca gerarchia di eccezioni. E' buona norma usare le eccezioni predefinite, cercando nella API di `Exception` [locale, Medialab, Sun] se esiste una eccezione adeguata.

- Comunque, un utente può ***definire*** una sua classe di eccezioni e usarle come quelle di Java: può ***lanciarle***, ***catturarle***, ***delegarne la gestione***.

- Le eccezioni di una nuova classe sono ***controllate*** oppure ***non controllate*** a seconda della superclasse.

Un esempio: `IllegalTimeException`

Supponiamo di avere definito la classe `Time` contenente il costruttore di default, il metodo `toString()`, ed il metodo `setTime()` per impostare un orario ammissibile. Come succede spesso nella progettazione di classi di questo tipo, dobbiamo decidere come gestire il passaggio di valori errati al metodo `setTime()`.

Una soluzione semplice è la seguente:

```
public class Time {
    private int ore, minuti, secondi;

    public Time () {   // costruttore
        ore = minuti = secondi = 0;
    }

    public String toString () {
        return (ore + ":" + minuti + ":" + secondi);
    }

    public boolean setTime (int oo, int mm, int ss) {
        if (!checkTime(oo, mm, ss)) // con parametri errati restituisce false
            return false;
        ore = oo;
        minuti = mm;
        secondi = ss;
        return true;
    }

    protected boolean checkTime (int oo, int mm, int ss) {
        return ( 0 <= oo && oo < 24 &&  // oo in [0,23]
                 0 <= mm && mm < 60 &&  // mm in [0,59]
                 0 <= ss && ss < 60 );  // ss in [0,59]
            }
}
```

Una alternantiva ragionevole in questa situazione è quella di lanciare un'opportuna eccezione se il valore dei parametri è errato, ad esempio una `IllegalArgumentException`: dopotutto è dovere di chi invoca il metodo passare valori corretti.

Per indicare in maniera più precisa la natura dell'errore possiamo definire una nuova classe di eccezioni `IllegalTimeException`:

```
public class IllegalTimeException extends RuntimeException {

    public IllegalTimeException() {
        super();
    }

    public IllegalTimeException(String msg) {
        super(msg);
    }
}
```

Adesso possiamo estendere la classe `Time` con la classe `RobustTime`, sovrascrivendo opportunamente il metodo `setTime()`.

```
public class RobustTime extends Time {

    public RobustTime(){
        super();
    }

    public boolean setTime (int oo, int mm, int ss) {
        if (!checkTime(oo, mm, ss))
            throw new IllegalTimeException("Argomenti di setTime errati");
        return super.setTime(oo,mm,ss);
    }
}
```

Infine possiamo testare la classe col seguente programma **TestTime** :

```
public class TestTime {

    public static void main(String[] args) {
        do{
            System.out.print("    Ore [0,23] => ");
            int ore = Input.readInt();
            System.out.print(" Minuti [0,59] => ");
            int min = Input.readInt();
            System.out.print("Secondi [0,59] => ");
            int sec = Input.readInt();

            // Time t = new Time ();
            Time t = new RobustTime();

            try {
                t.setTime(ore,min,sec);
                System.out.println("Sono le ore " + t);
            } catch (IllegalTimeException e) {
                System.out.println ("Errore:" + e);
            }
            System.out.print("Ancora (s/n)? ");
        }while(Input.readChar()=='s');
    }
}
```

Problema: Cosa succederebbe se la classe **IllegalTimeException** estendesse **Exception** invece di **RuntimeException**? Quali modifiche bisognerebbe apportare alle classi **Time**, **RobustTime** e **TestTime** affinché tutto funzionasse come prima?
