

Hlídané a vlastní výjimky, blok **finally**

Tomáš Pitner, Radek Ošlejšek, Marek Šabo

Blok `finally`

- Blok `finally` obvykle následuje po blocích `catch`.
- Zavolá se vždy, tj.
 - když je výjimka zachycena blokem `catch`
 - když je výjimka **propuštěna** do volající metody
 - když výjimka nenastane



Používá se typicky pro uvolnění (systémových) zdrojů, např. uzavření souborů.

Příklad na `finally`

- Zavírání vstupu (IO bude probíráno později):

```
InputStream is = null;
try {
    // trying to read from the file
    is = new FileInputStream("data.bin");
} catch (IOException e) {
    e.printStackTrace();
} finally {
    if (is != null) is.close();
}
```

Blok `finally` bez `catch`

- Dokonce existuje možnost `try-finally`
 - pro případy typu "potřebuji zavřít soubor jestli se výjimka vyhodí anebo ne"

```
InputStream is = null;
try {
    // trying to read from the file
    is = new FileInputStream("data.bin");
} finally {
    if (is != null) is.close();
}
```

Na zamyšlení

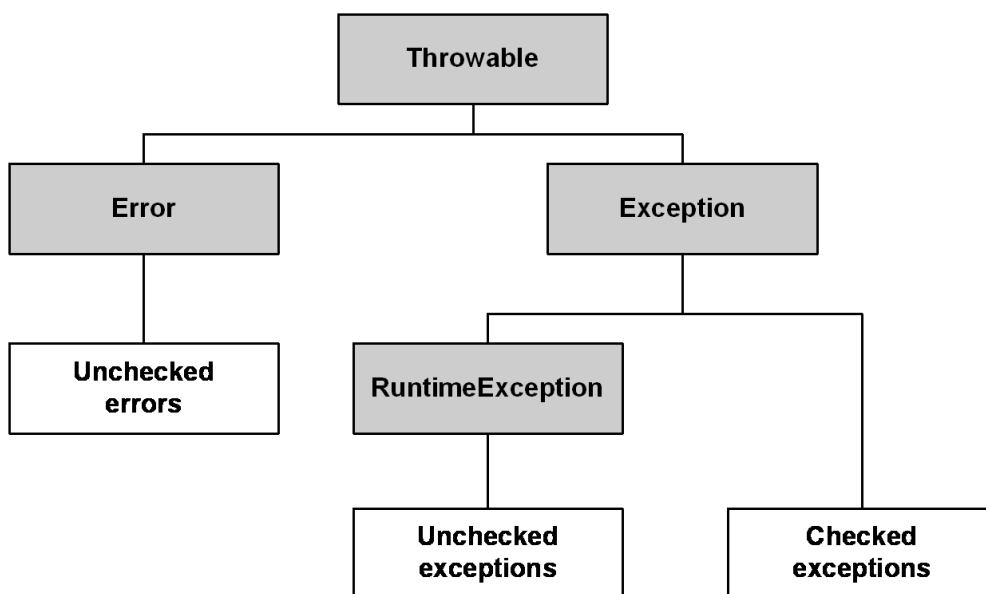
- Příkaz `return` udělá okamžitý návrat z metody.
- Blok `finally` se vždy vykoná.

- Co vrátí následující kód?

```
try {  
    return 5;  
} finally {  
    return 4;  
}
```

Řešení: Obsah `finally` se vykoná skutečně vždy.

Hierarchie výjimek I



Hierarchie výjimek II

Throwable — obecná třída pro vyhazovatelné objekty, dělí se na:

- **Error** — chyby způsobené prostředím
 - `OutOfMemoryError`, `StackOverflowError`
 - je téměř nemožné se z nich zotavit
 - překladač o nich neví
- **Exception** — chyby způsobené aplikací
 - `ClassCastException`, `NullPointerException`, `IOException`
 - je možné se z nich zotavit
 - překladač neví **jenom** o podtřídě `RuntimeException`

Hlídané & nehlídané výjimky

Existují 2 typy výjimek (**Exception**):

- běhové (**nehlídané**), *unchecked*:
 - dědí od třídy `RuntimeException`
 - `NullPointerException`, `IllegalArgumentException`
 - netřeba je definovat v hlavičkách metody
 - reprezentují **chyby v programu**
- hlídané, *checked*:
 - dědí přímo od třídy `Exception`
 - `IOException`, `NoSuchMethodException`
 - je nutno definovat v hlavičkách metody použitím `throws`
 - reprezentují **chybový stav** (zlý vstup, chybějící soubor)

Metody propouštějící výjimku I

- Výjimky, které nejsou zachyceny pomocí `catch` mohou z metody propadnout výše.
- Tyhle výjimky jsou indikovány v hlavičce metody pomocí `throws`:

```
public void someMethod() throws IOException {
    ...
}
```

- Pokud hlídaná výjimka nikde v těle nemůže vzniknout, překladač ohlásí chybu.

```
// Will not compile
public void someMethod1() throws IOException { }
```

Metody propouštějící výjimku II

Pokud `throws` u **hlídaných výjimek** chybí, program se nezkompiluje:

```
// Ok
public void someMethod1() {
    throw new NullPointerException("Unchecked exception");
}
// Will not compile
public void someMethod1() {
    throw new IOException("Checked exception");
}
```



Pozor na rozdíl mezi `throw` a `throws`

Příklad s propouštěnou výjimkou

```
public static void main(String[] args) {
    try {
        openFile(args[0]);
        System.out.println("File opened successfully");
    } catch (IOException ioe) {
        System.err.println("Cannot open file");
    }
}

private static void openFile(String filename) throws IOException {
    System.out.println("Trying to open file " + filename);
    FileReader r = new FileReader(filename);
    // success, now do further things
}
```

Stručné shrnutí

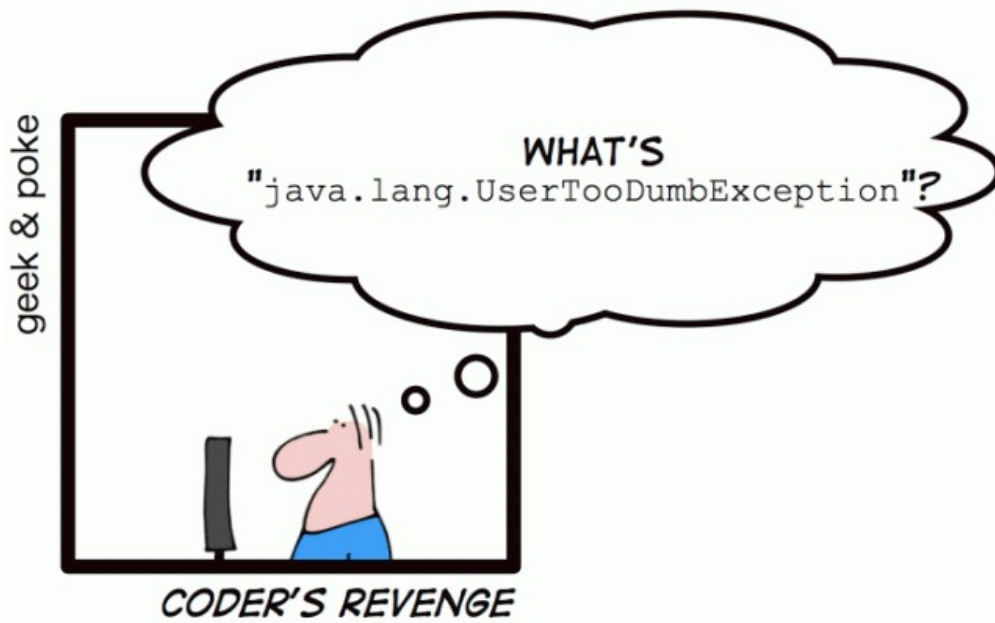
- u hlídaných výjimek **musí** být `throws`
- u nehlídaných výjimek **může** být `throws`

```
// throws is not necessary
public void someMethod1() throws NullPointerException {
    throw new NullPointerException("Unchecked exception");
}

// throws is necessary
public void someMethod2() throws IOException {
    throw new IOException("Checked exception");
}
```

Vytváření vlastních výjimek

- Třídy výjimek si můžeme definovat sami.
- Bývá zvykem končit názvy tříd výjimek na `Exception`:



Konstruktory výjimek

Ideální je definovat 4 konstruktory:

Constructor	Description
<code>IllegalArgumentException()</code>	Constructs an <code>IllegalArgumentException</code> with no detail message.
<code>IllegalArgumentException(String s)</code>	Constructs an <code>IllegalArgumentException</code> with the specified detail message.
<code>IllegalArgumentException(String message, Throwable cause)</code>	Constructs a new exception with the specified detail message and cause.
<code>IllegalArgumentException(Throwable cause)</code>	Constructs a new exception with the specified cause and ...

Příklad vlastní výjimky

- Vytvoření **hlídané** výjimky (nehlídaná dědí od `RuntimeException`):

```
public class MyException extends Exception {  
    public MyException(String s) {  
        super(s);  
    }  
    public MyException(Throwable cause) {  
        super(cause);  
    }  
    ...  
}
```



U výjimek stejně jako u jiných tříd můžeme mít atributy, konstruktory, atd.

Použití vlastní výjimky I

- Použití konstruktoru se `String message`:

```
public void someMethod(Thing badThing) throws MyException {
    if (badThing.happened()) {
        throw new MyException("Bad thing happened.");
    }
}
```

Použití vlastní výjimky II

- Konstruktor s `Throwable` se používá na obalování výjimek (i errorů).
- Výhodou je, že při volání `someMethod()` nemusíme řešit všechny možné typy výjimek:

```
public void someMethod() throws MyException {
    try {
        doStuff();
    } catch (IOException | IllegalArgumentException e) {
        throw new MyException(e);
    }
}
public void doStuff() throws IOException, IllegalArgumentException {
    ...
}
```

- Výjimky se dají zachytávat a řetězit:

```
try {
    int[] array = new int[1];
    a[4] = 0;
    System.out.println("Never comes to here");
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("ArrayIndexOutOfBoundsException has been thrown, continue in code");
    // Puts chain of previous exception
    throw new MyException("Exception occurred", e);
} finally {
    System.out.println("This always happens");
}
```