Contents

[1. Java interfaces 2](#_Toc4969751)

[2. Principiile SOLID 3](#_Toc4969752)

[3. Excepții și erori 3](#_Toc4969753)

[3.1. Blocul try and catch 3](#_Toc4969754)

[3.2. Try with resources 5](#_Toc4969755)

[3.3. Excepții 6](#_Toc4969756)

[4. Enums 6](#_Toc4969757)

[5. Nested class 7](#_Toc4969758)

[6. Fișiere 7](#_Toc4969759)

[7. Referințe 8](#_Toc4969760)

# Java interfaces

1. O interfață se realizează prin utilizarea “interface”
2. Nu se poate instanția
3. Oferă abstractizarea maximă, nu conține corp la metode ( cel puțin până la 1.7, inclusiv)
   1. Metode (public) default 1.8
   2. Metode (public) statice 1.8
   3. Metode (private) 1.9
   4. Metode (private) statice 1.9
4. Nu au constructor
5. Orice atribut al unei interfețe este implicit public static final
6. Orice interfață este implicit public și abstract
7. Interfața nu poate extinde o clasă, însă poate extinde o alta interfață
8. O clasă pentru a implementa o interfață utilizează cuvântul cheie implements
9. O clasă ce implementează o interfață trebuie neapărat să implementeze toate metodele sau să fie abstractă

*Avantaje*

* Oferă un contract pentru implementarea claselor, ce vor fi obligate să îl respecte
* ierarhie de clase se creează ușor plecând de la interfețe
* Pentru că java poate implementa mai multe interfețe, însă o singură clasă, este mai bine să se folosească interfețele ca punct de plecare

*Dezavantaje*

* Metodele ce sunt folosite în interfețe trebuie alese cu grijă altfel orice modificare ulterioară a lor, va duce la modificări în toată ierarhia
* În cazul în care clasele ce implementează o interfață e necesar să folosească și alte metode, diferite de cele din interfață, este necesar cast la clasa concretă.

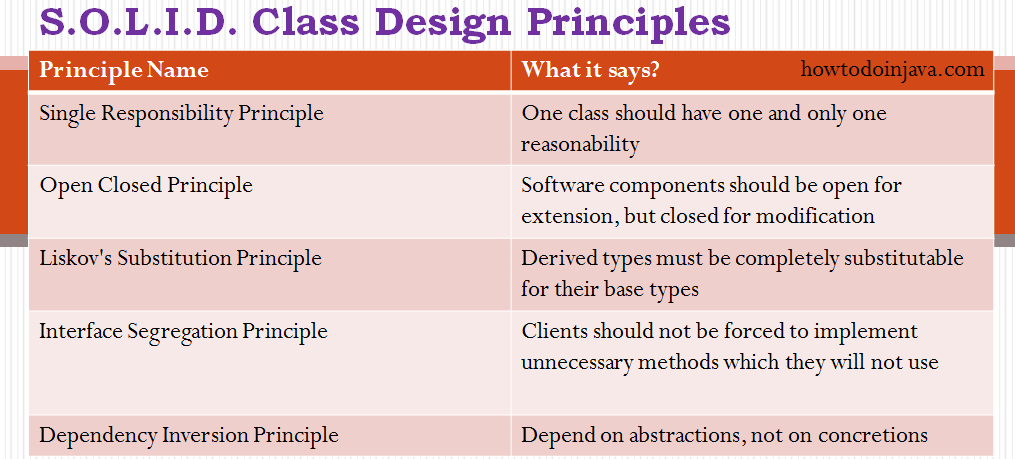
Default methods

* Se moștenesc
* Pot sa fie suprascrise cu respectarea regulilor de la suprascrierea metodelor

Static methods

* Nu se moștenesc spre deosebire de metodele statice din clase

# Principiile SOLID



1. User, Account, Report
2. Orice obiect poate fi extins, orice interfață poate fi implementată dar nu le modificăm
3. Orice obiect copil poate fi folosit in locul părintelui acolo unde acesta este necesar
4. IReport, IInvoice, ICustomer etc
5. Spring beans configuration, ORM db stuff.

# Excepții și erori

## Blocul try and catch

Pentru tratarea excepțiilor se folosesc blocuri pentru

try {

} catch () {

} finally {

}

oricare din catch sau finally poate să lipsească.

În cazul în care există blocul finally, el se va executa întotdeauna.

În cazul în care avem o excepție, blocul finally se va executa imediat după execuția clauzei catch corespunzătoare excepției, dacă există.

De asemenea putem avea mai multe secțiuni catch. Ori una singură cu mai multe tipuri de excepții.

**catch** (Exceptie1 | Exceptie2 exc)

Ordinea în care se tratează are importanță, se merge de la specific către general.

Exemplu de bloc finally , ce se va executa chiar și după return

**public static void** main(String[] args) {  
  
 **int** resultOfMethod = *performFinally*();  
 System.***out***.println(**"resultOfMethod = "** + resultOfMethod);  
 }  
  
 **public static int** performFinally() {  
 **int** result = 1;  
  
 **try** {  
 result = 2;  
 *myExceptionMethod*();  
 } **catch** (Exception exc) {  
 exc.printStackTrace();  
 **return** 0;  
 } **finally** {  
 result = 4;  
 System.***out***.println(**"finally block"**);  
 **return** result;  
 }  
  
*// return result;* }  
  
  
 **public static void** myExceptionMethod() {  
 **throw new** NullPointerException(**"my null pointer exception"**);  
 }

java.lang.NullPointerException: my null pointer exception

at com.fmi.laborator.TryCatchFinally.myExceptionMethod(TryCatchFinally.java:33)

at com.fmi.laborator.TryCatchFinally.performFinally(TryCatchFinally.java:18)

at com.fmi.laborator.TryCatchFinally.main(TryCatchFinally.java:7)

finally block

resultOfMethod = 4

Dacă renunțăm la partea catch, finally se va executa.

**try** {  
 result = 2;  
 **return** 0;  
  
} **catch** (Exception exc) {  
 exc.printStackTrace();  
} **finally** {  
 result = 4;  
 System.***out***.println(**"finally block"**);  
 **return** result;  
}

obținem același rezultat.

## Try with resources

Există o utilizare a blocului try / catch folosind doar try, se numește try with resources și este folosit în utilizarea resurselor precum fișiere și conexiuni. El se ocupă și de eliberarea resurselor.

**public** String readFirstLineFromFile(String path) **throws** IOException {  
 **try** (BufferedReader br = **new** BufferedReader(**new** FileReader(path))) {  
 **return** br.readLine();  
 }  
}

## Excepții

Error, checked și unchecked



O schemă simplistă a împărțirii pe categorii

În funcție de contextul aplicației, se pot implementa excepții particularizate pentru cazul nostru.

Un exemplu de excepție ar putea fi:

**public class** MyException **extends** Exception {  
  
 **public** MyException(String msg) {  
 **super**(msg);  
 }  
}

# Enums

* An enumeration is like a fixed set of constants.
* In Java, an enum is a class that represents an enumeration.
* Enum is a type

public enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY,

THURSDAY, FRIDAY, SATURDAY

}

Pot avea câmpuri și constructor

**public enum** DAY {  
 ***SUNDAY***(0), ***MONDAY***(1), ***TUESDAY***(2), ***WEDNESDAY***(3),  
 ***THURSDAY***(4), ***FRIDAY***(4), ***SATURDAY***(5);  
  
 **private int position**;  
  
 DAY(**int** position) {  
 **this**.**position** = position;  
 }  
  
 **public int** getPosition() {  
 **return position**;  
 }  
}

# Nested class

A nested class is a class that is defined within another class. A nested class that is not static is called an inner class. There are four of types of nested classes:

* A member inner class is a class defined at the same level as instance variables. It is not static. Often, this is just referred to as an inner class without explicitly saying the type.
* A local inner class is defined within a method.
* An anonymous inner class is a special case of a local inner class that does not have a name.
* A static nested class is a static class that is defined at the same level as static variables.

# Fișiere

Există două abordări IO și NIO

|  |  |
| --- | --- |
| IO | NIO |
| Orientat stream | Orientat buffer |
| Blocking IO | Non blocking IO |

Avantajul principal al NIO este citirea unui buffer (ferestre) din întreg cât și posibilitatea de a prelucra datele asincron.

Odată cu apariția versiuni de nio au apărut câteva clase utilitare precum: Files și Paths (‚s’ face diferența dintre o clasă utilitară și o clasă utilizată, în acest caz)

Files permite utilizarea de metode de lucru asupra fișierelor și Paths asupra căilor de acces la fișiere.

<https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html>

<https://docs.oracle.com/javase/8/docs/api/java/nio/file/Paths.html>

File este utilizată

* pentru a citi meta informația referitoare la un fișier sau folder
* creare și ștergere de fișiere / foldere

O instanță are o cale pathname

Nu poate fi utilizată să citească informația dintr-un fișier (aici intervin stream-urile)

# Referințe

1. SOLID, <https://howtodoinjava.com/best-practices/5-class-design-principles-solid-in-java/>
2. SOLID, <https://www.baeldung.com/solid-principles>
3. Java interfaces discussions, <https://www.journaldev.com/12850/java-9-private-methods-interfaces>
4. Enums, <https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>
5. Files, <https://docs.oracle.com/javase/8/docs/api/java/nio/file/Files.html>
6. Paths, <https://docs.oracle.com/javase/8/docs/api/java/nio/file/Paths.html>