

# Aula 05

## Robustez

### Gestão de Falhas

Programação II, 2018-2019

v1.9, 08-03-2018

DETI, Universidade de Aveiro

05.1

#### Objectivos:

- Excepções em Java;
- Gestão de Falhas em Módulos.
- Gestão de Falhas em Programas.

## Conteúdo

<b>1</b>	<b>Mecanismo de Excepções</b>	<b>1</b>
1.1	Fundamentos . . . . .	1
1.2	Apanhar Excepções . . . . .	2
1.3	Especificar Excepções em Métodos . . . . .	3
1.4	Classificação de Excepções . . . . .	3
1.5	Discussão . . . . .	4
<b>2</b>	<b>Gestão de Falhas em Módulos</b>	<b>4</b>
2.1	Técnica da Avestruz . . . . .	5
2.2	Programação Defensiva (caso 1) . . . . .	5
2.3	Programação Defensiva (caso 2) . . . . .	6
2.4	Programação por Contrato . . . . .	7
2.5	Discussão . . . . .	8
<b>3</b>	<b>Gestão de Falhas em Programas</b>	<b>8</b>

05.2

## 1 Mecanismo de Excepções

### 1.1 Fundamentos

- Durante a execução de um programa, por vezes ocorrem *eventos anómalos* ou imprevistos, que interrompem o fluxo normal de execução. É o que chamamos de *excepções*.
- Esses eventos podem ser causados por *erros internos* do programa, que poderiam ter sido previstos e evitados pelo programador, como aceder a um índice inexistente num array, dividir por zero, etc.
- Ou podem ser devidos a *erros externos*, imprevisíveis, como um erro na leitura de um ficheiro, dados mal formatados, falta de memória, etc.
- Quando estes erros acontecem, é importante *interromper* imediatamente o fluxo de execução. Dessa forma evitamos que os erros afetem os resultados do programa com efeitos imprevisíveis.

- Por outro lado, se for possível rectificar a situação, é conveniente ter uma forma de *retomar a execução* normal do programa. Assim, podemos ter código robusto que detecta situações de erro e corrige-as, permitindo que o programa prossiga normalmente.
- O *mecanismo de excepções* serve precisamente estes dois propósitos.

05.3

O mecanismo de excepções em Java funciona assim:

- Quando a excepção ocorre, a instrução que estava a ser executada não termina e a *execução é interrompida*. O programa não avança para a instrução seguinte.
- É criado um tipo especial de objeto que contém informação sobre a excepção, incluindo o seu tipo, o local onde ocorreu e outros dados.
- Este *objeto-excepção* vai sendo propagando para blocos sucessivamente mais exteriores até interromper o bloco do método `main` ou até ser interceptado (ou “apanhado”).
- Existe uma instrução – *throw* – que permite “lançar” excepções.
- Existe uma instrução composta – *try/catch/finally* – que permite interceptar excepções. Ou seja, permite retomar o modo de execução normal.

05.4

```
public class Example {
    public static void main(String[] args) {
        ...; p1(); ⚡ (...); ← não chega a ser executado (4)
    }

    static void p1() {
        ...; p2(); ⚡ (...); ← não chega a ser executado (3)
    }

    static void p2() {
        ...; p3(); ⚡ (...); ← não chega a ser executado (2)
    }

    static void p3() {
        ...; throw ⚡ (...); ← não chega a ser executado (1)
    }
}
```

main	main	main	main	main	main	main ⚡
	p1	p1	p1	p1	p1 ⚡	
		p2	p2	p2 ⚡		
			p3 ⚡			

05.5

## 1.2 Apanhar Excepções

- Delegação do erro (“lançar” excepção):

```
if (t == null)
    throw new NullPointerException();
// throw new NullPointerException("t null");
```

- Tratamento do erro no contexto local (“apanhar” excepções):

```
try {
    /* O que se pretende fazer */
}
catch (Errortype a) {
    /* O que fazer em caso de erro */
}
```

05.6

## Controlo de Excepções

A manipulação de excepções é feita através da instrução `try/catch/finally`:

---

```
try {
    // Código que pode gerar excepções do tipo Type1,
    // Type2 ou Type3
} catch (Type1 id1) {
    // Gerir excepção do tipo Type1
} catch (Type2 id2) {
    // Gerir excepção do tipo Type2
} catch (Type3 id3) {
    // Gerir excepção do tipo Type3
} finally {
    // Bloco executado independentemente de haver
    // ou não uma excepção
}
```

---

05.7

## Classe Throwable

- Podemos usar a excepção `java.lang.Throwable` para capturar qualquer tipo de excepção:

---

```
catch (Throwable e) { // "Apanha" todas as excepções
    exit(1);
}
```

---

- Podemos gerar nova excepção de forma a ser tratada num nível superior:

---

```
catch (Throwable e) {
    ...(faz qualquer coisa)
    throw e; // A excepção vai ser relançada
}
```

---

05.8

## 1.3 Especificar Excepções em Métodos

- A linguagem Java permite que se associe à assinatura dos métodos uma lista de excepções que os mesmos podem lançar:

---

```
void m() throws TooBigException,
              TooSmallException,
              DivByZeroException {
    //...
}
```

---

- Desta forma, o (eventual) lançamento destas excepções passa a fazer parte da informação sintáctica sobre o método. Para certos tipos de excepções, como se verá já a seguir, passa mesmo a ser obrigatório lidar com as excepções (apanhando ou propagando explicitamente).

05.9

## 1.4 Classificação de Excepções

A linguagem Java agrupa as excepções em dois tipos: as *checked* e as *unchecked exceptions*.

- As *excepções checked* obrigam (sintacticamente) o programador a “apanhá-las” ou a “relançá-las”;
- Assim se invocarmos um método que possa lançar uma excepção deste tipo, temos de indicar ao compilador como a vamos tratar:

1. Apanhar (`try ... catch`);
2. Propagar, fazendo com que a excepção passe a fazer parte da assinatura do método (`throws`).

- As *excepções unchecked* diferem das anteriores apenas pelo facto de não obrigarem o programador a lidar com elas (apanhando ou propagando).;
- Apesar do nome, este tipo de excepções não são ignoradas pela linguagem, funcionando da mesma forma no que diz respeito ao mecanismo de excepções;
- As excepções *unchecked* são todas aquelas que derivam das classes: `RuntimeException` ou `Error`.

O significado preciso desta definição obriga a um conhecimento sobre mecanismos específicos da Programação Orientada por Objectos (o que sai fora do âmbito desta disciplina). No entanto, para saber ao certo o tipo de uma qualquer excepção basta ver o cabeçalho da respectiva documentação. Em baixo do nome da excepção aparece uma lista de classes. Se nessa lista for apresentada a classe `Error` ou a classe `RuntimeException`, então estamos na presença de uma excepção *unchecked*.

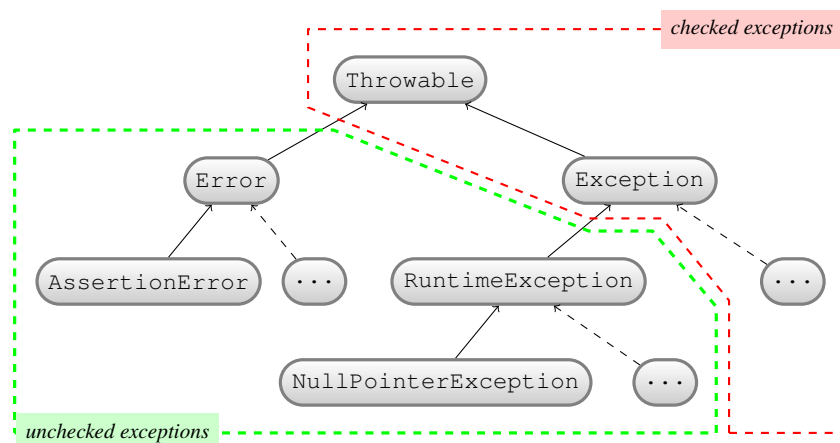
Por exemplo, na documentação da excepção `NullPointerException` aparece o seguinte texto:

---

```
java.lang
Class NullPointerException

java.lang.Object
  extended by java.lang.Throwable
    extended by java.lang.Exception
      (extended by java.lang.RuntimeException) ← unchecked exception
        extended by java.lang.NullPointerException
```

---



## 1.5 Discussão

### Vantagens das Excepções

Algumas vantagens das excepções relativamente à implementação do tratamento de erros no código normal são as seguintes:

- Alguma separação entre o código regular e o código de tratamento de erros;
- Propagação dos erros em chamadas sucessivas;
- Agrupamento de erros por tipos;
- Facilita a implementação de código tolerante a falhas.

## 2 Gestão de Falhas em Módulos

Por forma a definirmos metodologias sistemáticas de lidar com falhas em programas, vamos primeiramente analisar o problema da gestão de falhas em módulos (métodos e classes) dentro de um programa.

Existem basicamente três possibilidades:

- **Técnica da avestruz:**

- Ignorar o problema.
- *Não aconselhável!*



- **Programação Defensiva:**

- Aceitar todas as situações, ter código específico para detectar e lidar com erros.

- **Programação por Contrato:**

- Associar contratos ao módulo;
- O módulo só tem de cumprir a sua parte do contrato;
- Associar asserções aos contratos para detecção de falhas em tempo de execução (as falhas são consideradas erros do programa).

05.13

## 2.1 Técnica da Avestruz

Nesta “estratégia”, de uso inacreditavelmente frequente por muitos programadores, o módulo é construído presumindo a inexistência de erros (quer internos ao módulo, quer externos resultantes de utilizações erradas).

```
public class Data {  
  
    public Data(int dia,int mes,int ano) {  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes,int ano) {  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}  
  
public void main(String[] args) {  
    Data d = new Data(25,4,1974);  
    ...  
    if (Data.diasDoMes(mes,ano) != 31)  
        ...  
}
```



05.14

## 2.2 Programação Defensiva (caso 1)

Uma primeira abordagem a uma real gestão de falhas em programas consiste em fazer-se uso dos mecanismos normais das linguagens de programação. No caso de uma função, podemos, por vezes, utilizar o respectivo resultado para indicar situações de falha, ou, no caso de uma classe, podemos criar um atributo de indicação de erro e incluir na interface uma função que pode ser externamente consultada para

verificar essa situação.

```
public class Data {

    public Data(int dia,int mes,int ano) {
        if (!valida(dia,mes,ano))
            aErro = true;
        else {
            aErro = false;
            aDia = dia; aMes = mes; aAno = ano;
        }
    }

    public static int diasDoMes(int mes,int ano) {
        int result;

        if (!mesValido(mes))
            result = -1;
        else {
            final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};
            result = dias[mes-1];
            if (mes == 2 && anoBissexto(ano))
                result++;
            return result;
        }
    }

    public boolean erro() { return aErro; }
    private boolean aErro = false;
    private int aDia,aMes,aAno;
}

public void main(String[] args) {
    Data d = new Data(25,4,1974);
    if (d.erro())
        doSomethingWithError;
    ...
    int r = Data.diasDoMes(mes,ano);
    if (r == -1)
        doSomethingWithError;
    if (r != 31)
        ...
}
```

erro guardado num atributo

erro no resultado da função

função e atributo de erro

05.15

Muito embora a utilização desta estratégia seja suficiente para o módulo se proteger (internamente robusto), o mesmo já não acontece necessariamente para os seus clientes (externamente não robusto). O problema reside no facto de nada obrigar os clientes do módulo a verificarem sistematicamente a existência de erros (quer nos resultados de funções, quer por uso da função indicadora de erro).

## 2.3 Programação Defensiva (caso 2)

Uma melhor abordagem consiste em combinar a utilização da instrução condicional para testar situações de erro com o lançamento de excepções sempre que estes ocorrem. Desta forma os clientes do módulo já nada têm de fazer para que não haja erros a passar incólumes (externamente mais robusto). É prática frequente em Java forçar os clientes dos módulos a lidar com muitas destas excepções fazendo com

que estas sejam do tipo *checked*.

```
public class Data {  
  
    public Data(int dia,int mes,int ano) throws IllegalArgumentException {  
        if (!valida(dia,mes,ano))  
            throw new IllegalArgumentException();  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes,int ano) throws IllegalArgumentException {  
        if (!mesValido(mes))  
            throw new IllegalArgumentException();  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}
```

erro lançado como excepção

**NOTA MUITO IMPORTANTE**

No código catch deve-se: *terminar o programa*, ou *propagar a excepção*, ou *voltar a tentar o código try* (inserindo todo o bloco try/catch num ciclo). Qualquer outra acção pode gerar problemas de robustez no programa!

```
public void main(String[] args) {  
    Data d;  
    try {  
        d = new Data(25,4,1974);  
        ...  
    }  
    catch(IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
    try {  
        if (Data.diasDoMes(mes,ano) != 31)  
            ...  
    }  
    catch(IllegalArgumentException e) {  
        doSomethingWithError;  
    }  
}
```

05.16

Esta abordagem tem os seguintes problemas:

- Apesar do uso de excepções, não há uma total separação entre o código normal e o código de erro.
- A detecção de erros continua a fazer uso da instrução condicional. Assim, mesmo que haja a forte convicção de que não existem erros, não é possível desactivar (nem total, nem caso a caso) o código de gestão de erros;
- O código normal (que se quer correcto) é contaminado com o código de gestão de erros, situação esta ainda mais agravada quando na presença de excepções do tipo *checked*;
- Por fim, como se chamou a atenção na caixa **NOTA MUITO IMPORTANTE**, é necessário o máximo cuidado na utilização da instrução try/catch já que nesta facilmente se pode ignorar excepções (situação que nos remete para a técnica da Avestruz).

## 2.4 Programação por Contrato

A melhor abordagem para a gestão de falhas internas a um programa é, sem duvida, a programação por contrato. Nesta, a especificação da correcção das várias partes de um programa é feita recorrendo a

asserções. Um erro num programa é simplesmente o resultado de uma asserção falsa.

```
public class Data {  
  
    public Data(int dia,int mes,int ano) {  
        assert valida(dia,mes,ano);  
        aDia = dia; aMes = mes; aAno = ano;  
    }  
  
    public static int diasDoMes(int mes,int ano) {  
        assert mesValido(mes);  
  
        final int[] dias = {31,28,31,30,31,30,31,31,30,31,30,31};  
        int result = dias[mes-1];  
        if (mes == 2 && anoBissexto(ano))  
            result++;  
        return result;  
    }  
  
    private int aDia,aMes,aAno;  
}  
  
public void main(String[] args) {  
    Data d = new Data(25,4,1974);  
    ...  
    if (Data.diasDoMes(mes,ano) != 31)  
        ...  
}
```

05.17

A separação entre o código normal e o código de erro é total, com a garantia de robustez interna e externa do módulo.

## 2.5 Discussão

### Gestão de falhas em módulos

- **Técnica da Avestruz:**
  - Código *simples*, mas *não robusto*.
- **Programação Defensiva:**
  - Código *internamente robusto*, mas sem garantir que os clientes detectam situações de erro (*externamente não robusto*).
  - No caso 2 (excepções *checked*) o programa pode ser externamente robusto desde que se sigam os conselhos dados na caixa: **NOTA MUITO IMPORTANTE**;
  - Código mais *complexo*.
- **Programação por Contrato:**
  - Código *simples*, *interna e externamente robusto*;
  - No caso de se pretender apanhar a excepção `AssertionError`, então os conselhos dados na caixa **NOTA MUITO IMPORTANTE** são também aplicáveis.

05.18

## 3 Gestão de Falhas em Programas

- Na construção de programas nem todas as falhas resultam de erros internos a um programa.
- Por exemplo, quando um programa recebe *informação do exterior* através de argumentos do programa, ou de um processo de interacção com o utilizador, ou ainda quando lida com ficheiros; podem ocorrer falhas que escapam ao controlo interno do programa.
- Nestas situações, a utilização da programação por contrato não é a metodologia mais adequada. Já que, erradamente, considera erros de programa situações fora do seu controlo.
- Para este tipo de falhas (externas), a metodologia que deve ser aplicada é a da programação defensiva. Ou seja, aceitar e tratar este tipo de erros fazendo uso dos mecanismos normais da linguagem para controlo de fluxo do programa (condicionais, excepções, etc.).
- Temos assim dois tipos de erros num programa:



*Internos:* Erros 100% da responsabilidade do programa. A programação por contrato é de longe a melhor metodologia para lidar com estes erros.

*Externos:* Erros que não sejam completamente da responsabilidade do programa (com origem em factores externos ao programa). Para estes casos a programação defensiva é a opção adequada.

05.19

## Falhas Externas

Para facilitar a identificação das *falhas externas* que podem aparecer num programa, apresentam-se alguns dos casos mais frequentes:

- Argumentos do programa (`main(String[] args)`);
  - Quando aplicável, é necessário verificar quantos são, e eventuais problemas de conversão para números, *strings* vazias, etc..
- Entradas do utilizador (`Scanner scin=new Scanner(System.in)`);
  - É necessário verificar eventuais problemas de conversão para números, *strings* vazias, etc..
- Leitura de ficheiros:
  - Lidar com a excepção `FileNotFoundException` (ou se preferir, `IOException`) na criação do `Scanner`;
  - Nas operações de `next...` lidar com as excepções: `InputMismatchException` e `NoSuchElementException`. Pode-se evitar lidar com a excepção `InputMismatchException` se se validar previamente a utilização das operações `next...` com os métodos `hasNext...`
- Escrita de ficheiros:
  - Lidar com a excepção `FileNotFoundException` (`IOException`) na criação do `PrintWriter`;
  - Após uso de `print...`, testar existência de erros de escrita com o método `checkError`.

Note que, para os objectivos desta unidade curricular, em termos de programação defensiva no uso de ficheiros, considera-se suficiente lidar com a excepção `FileNotFoundException`.

05.20

