

# Controlo de erros

## Exceções

UA.DETI.POO

# Problema

---

- ❖ Nem todos os erros são detectados na compilação.
- ❖ Estes são, geralmente, os que são mais menosprezados pelos programadores:
  - Compila sem erros → Funciona!!!
- ❖ Tratamento Clássico:
  - Se sabemos à partida que pode surgir uma situação de erro em determinada passagem podemos tratá-la nesse contexto (if...)

```
if ((i = doTheJob()) != -1) {  
    /* tratamento de erro */  
}
```

# Exceção

---

❖ Uma exceção é gerada por algo imprevisto que não é possível controlar.

❖ Utilização de Exceções:

– Tratamento do erro no contexto local

```
try {  
    /* 0 que se pretende fazer */  
}  
catch (Errortype a) {  
}
```

– Delegação do erro - gerar um objecto exceção (throw) no qual se delega esse tratamento.

```
if (t == null)  
    throw new NullPointerException();  
// ou  
// throw new NullPointerException("reference t can't be null");
```

# Controlo de Exceções

---

- ❖ A manipulação de exceções é feita através de um bloco especial: **try .. catch**.

```
try {  
    // Code that might generate exceptions Type1,  
    // Type2 or Type3  
} catch(Type1 id1) {  
    // Handle exceptions of Type1  
} catch(Type2 id2) {  
    // Handle exceptions of Type2  
} catch(Type3 id3) {  
    // Handle exceptions of Type3  
} finally {  
    // Executada independentemente de haver ou não  
    // uma exceção  
}
```

# Controlo de Exceções

---

- ❖ A manipulação de exceções pode ainda ser feita através de um bloco **try-with-resources**.
  - Assegura que os recursos são fechados
  - Neste caso não existe bloco finally

```
try ( Code that might generate exceptions Type1, Type2 or Type3 )  
{  
    // other code that does not cause exceptions  
} catch(Type1 id1) {  
    // Handle exceptions of Type1  
} catch(Type2 id2) {  
    // Handle exceptions of Type2  
} catch(Type3 id3) {  
    // Handle exceptions of Type3  
}
```

# Vantagens das Exceções

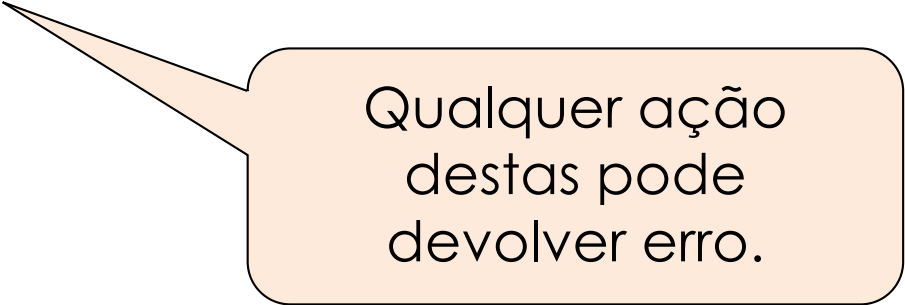
---

- ❖ Separação clara entre o código regular e o código de tratamento de erros
- ❖ Propagação dos erros em chamadas sucessivas
- ❖ Agrupamento de erros por tipos

# Separação de código – exemplo (1)

---

```
readFile {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
}
```



Qualquer ação  
destas pode  
devolver erro.

# Separação de código – exemplo (2)

Sem Exceções

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) { errorCode = -1; }
            } else { errorCode = -2; }
        } else { errorCode = -3; }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else { errorCode = errorCode and -4; }
    } else { errorCode = -5; }
    return errorCode;
}
```



# Separação de código – exemplo (3)

---

```
readFile {  
  try {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
  } catch (fileOpenFailed) {  
    doSomething;  
  } catch (sizeDeterminationFailed) {  
    doSomething;  
  } catch (memoryAllocationFailed) {  
    doSomething;  
  } catch (readFailed) {  
    doSomething;  
  } catch (fileCloseFailed) {  
    doSomething;  
  }  
}
```

**Com Exceções**

# Propagação dos erros (1)

```
method1 {  
    call method2;  
}  
method2 {  
    call method3;  
}  
method3 {  
    call readFile;  
}
```

Solução **sem**  
Exceções

```
method1 {  
    errorCodeType error;  
    error = call method2;  
    if (error)  
        doErrorProcessing;  
    else  
        proceed;  
}  
  
errorCodeType method2 {  
    errorCodeType error;  
    error = call method3;  
    if (error)  
        return error;  
    else  
        proceed;  
}  
  
errorCodeType method3 {  
    errorCodeType error;  
    error = call readFile;  
    if (error)  
        return error;  
    else  
        proceed;  
}
```

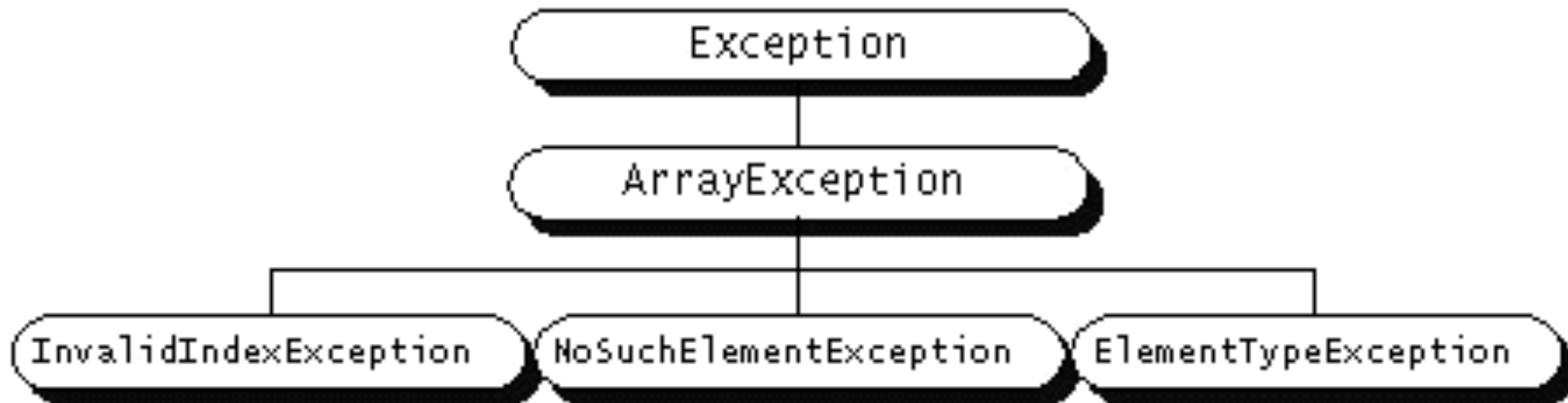
# Propagação dos erros (2)

```
method1 {  
    call method2;  
}  
method2 {  
    call method3;  
}  
method3 {  
    call readFile;  
}
```

Solução **com**  
Exceções

```
method1 {  
    try {  
        call method2;  
    } catch (exception) {  
        doErrorProcessing;  
    }  
}  
  
method2 throws exception {  
    call method3;  
}  
  
method3 throws exception {  
    call readFile;  
}
```

# Agrupamento de erros por tipos



```
catch (InvalidIndexException e) {  
    . . .  
}
```

**Diferenciação**

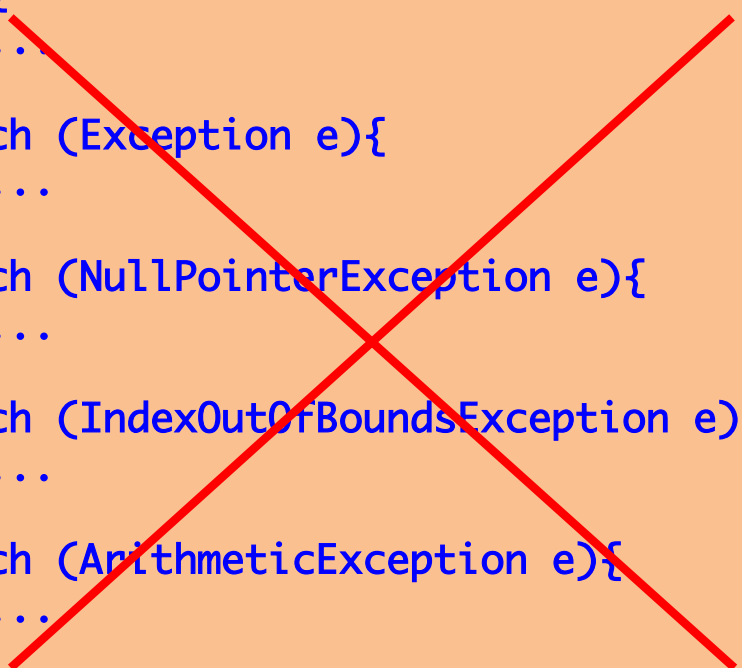
```
catch (ArrayException e) {  
    . . .  
}
```

**Agrupamento**

# Exceções - Hierarquia de Classes

A ordem dos *catch* é importante

```
try {  
    ...  
}  
catch (Exception e){  
    ...  
}  
catch (NullPointerException e){  
    ...  
}  
catch (IndexOutOfBoundsException e){  
    ...  
}  
catch (ArithmeticException e){  
    ...  
}
```



```
try {  
    ...  
}  
catch (NullPointerException e){  
    ...  
}  
catch (IndexOutOfBoundsException e){  
    ...  
}  
catch (ArithmeticException e){  
    ...  
}  
catch (Exception e){  
    ...  
}
```

# Tipos de Exceções

---

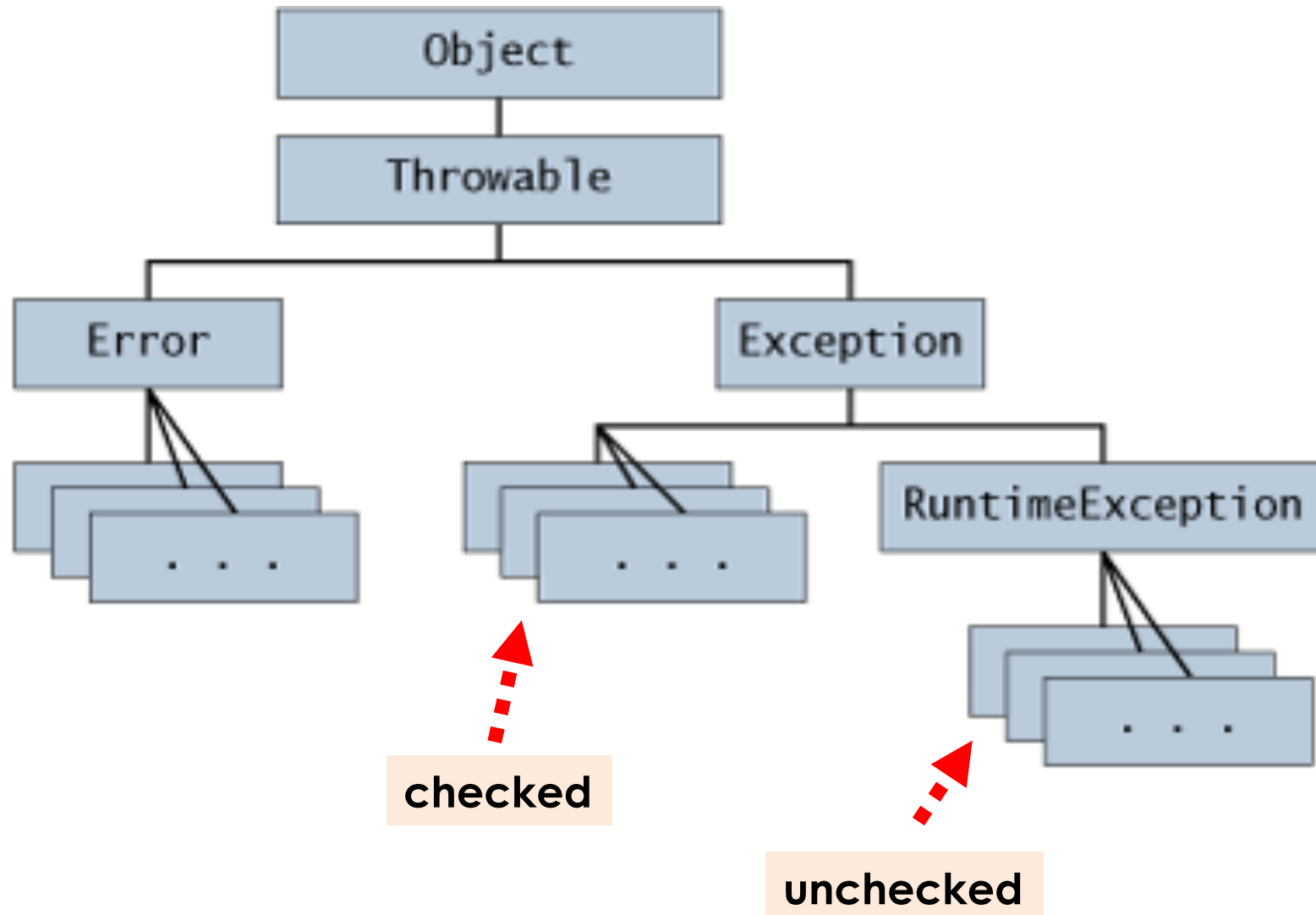
## ❖ checked

- Se invocarmos um método que gere uma checked exception, temos de indicar ao compilador como vamos resolvê-la:
  - 1) Resolver try .. catch, ou
  - 2) Propagar throw

## ❖ unchecked

- São erros de programação ou do sistema (podemos usar Asserções nestes casos)
- São subclasses de *java.lang.RuntimeException* ou *java.lang.Error*

# Exceções - Hierarquia de Classes



# Declaração de Exceções

---

- ❖ Quando desenhamos métodos que possam gerar exceções devemos assinalá-las explicitamente

- ❖ Declaração **throws**

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



# Criar Novas Exceções

---

- ❖ Podemos usar o mecanismo de herança para personalizar algumas exceções

```
class MyException extends Exception {  
    // interface base  
    public MyException() {}  
    public MyException(String msg) {  
        super(msg);  
    }  
    // podemos acrescentar construtores e dados  
}
```

# Boas Práticas

---

## ❖ Usar exceções apenas para condições excepcionais

- Uma API bem desenhada não deve forçar o cliente a usar exceções para controlo de fluxo
- Uma exceção não deve ser usada para um simples teste

**X**

```
try {  
    s.pop();  
} catch(EmptyStackException es) {...}
```

**V**

```
if (!s.empty()) s.pop();    // melhor!
```

# Boas Práticas

---

- ❖ Usar preferencialmente exceções standards
  - *IllegalArgumentException*  
valor de parâmetros inapropriado
  - *IllegalStateException*  
Estado de objecto incorreto
  - *NullPointerException*
  - *IndexOutOfBoundsException*
- ❖ Tratar sempre as exceções (ou delegá-las)

**X**

```
try {  
    // .. código que pode causar exceções  
} catch (Exception e) {}
```

# Sumário

---

## ❖ Controlo de Erros

- Bloco try .. catch
- Instrução throw

## ❖ Exceções

- checked
- unchecked

## ❖ Declaração throws