

Tratamento de erros

Exceções e assert

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

Controlo de Exceção

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

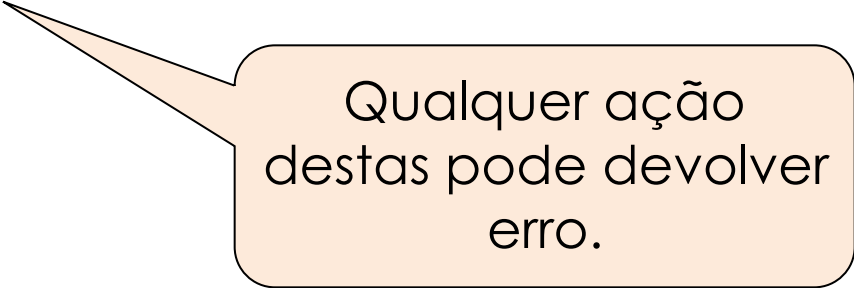
```
try {  
    // Code that might generate exceptions Type1,  
    // Type2 or Type3  
} catch(Type1 id1) {  
    // Handle exceptions of Type1  
} catch(Type2 id2) {  
    // Handle exceptions of Type2  
} catch(Type3 id3) {  
    // Handle exceptions of Type3  
} finally {  
    // Executada independentemente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

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;  
}
```

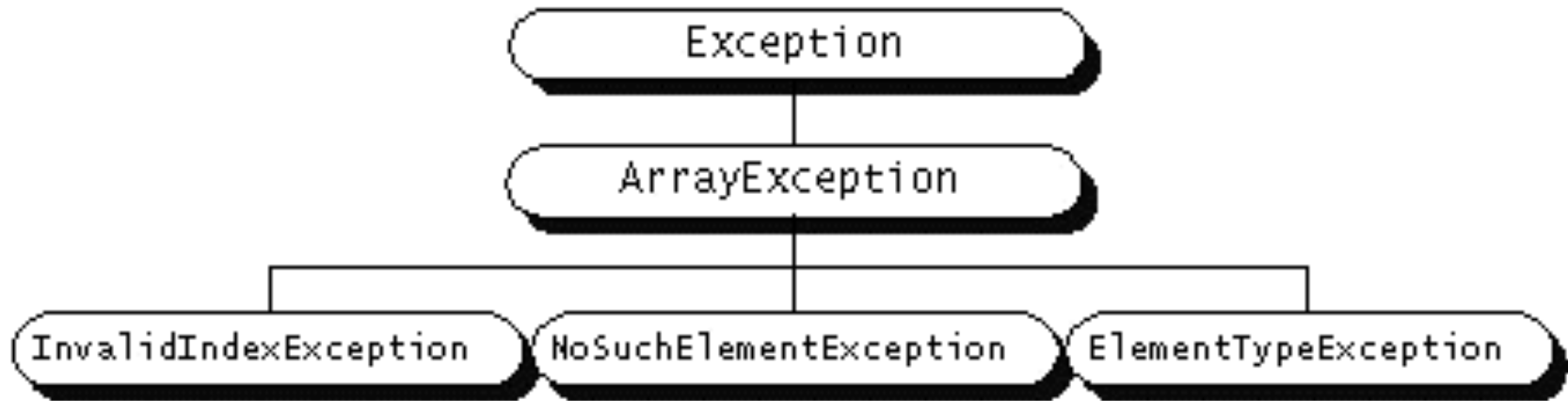
Propagação dos erros (2)

```
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

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

**Importante a
ordem dos handle's**

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

Má Solução

Tipos de Exceções

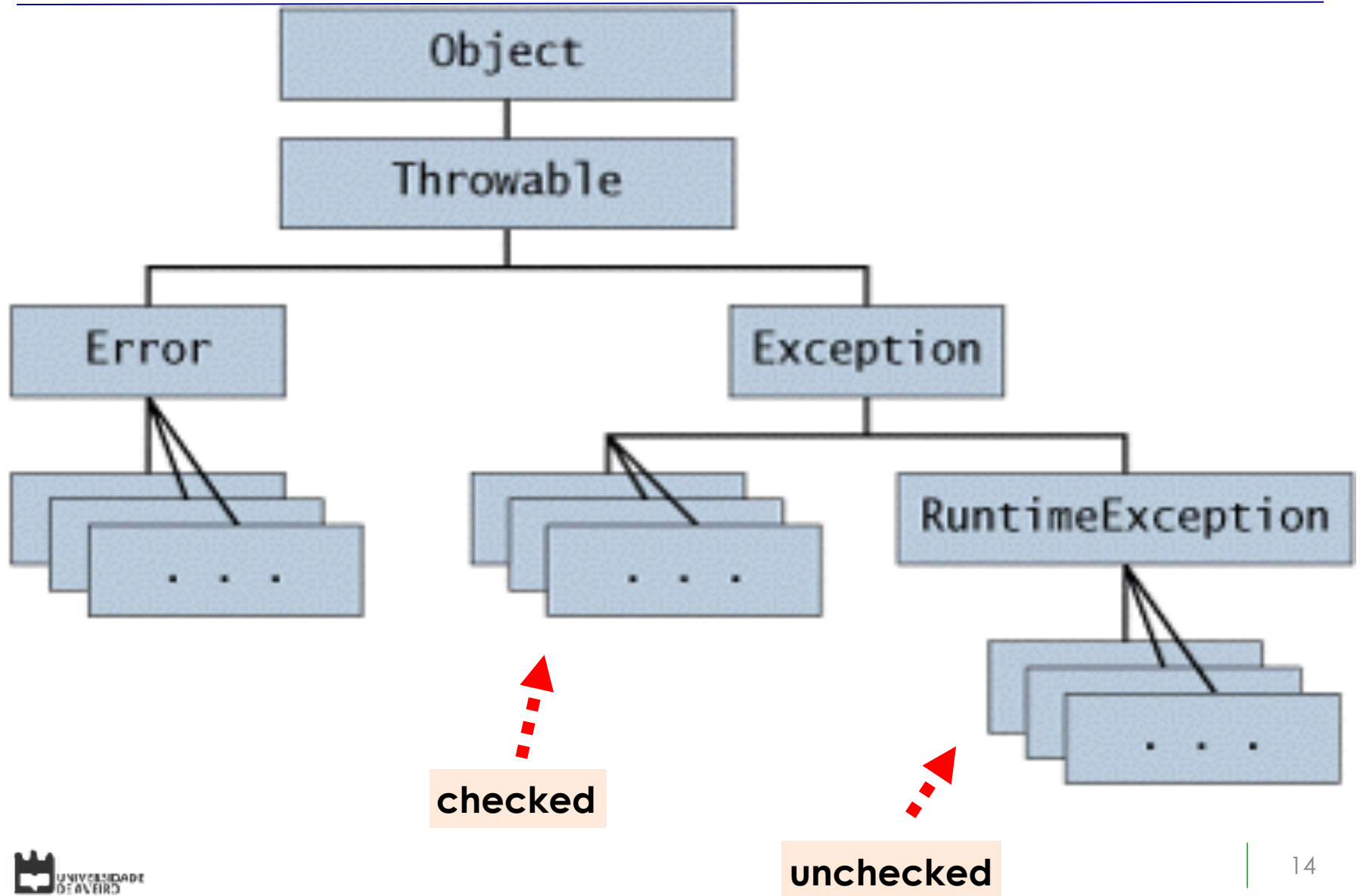
❖ checked

- 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



Classe *java.lang.Exception*

- ❖ A classe *Exception* é derivada da classe *Throwable*
- ❖ Podemos usar a classe base *java.lang.Exception* para capturar qualquer exceção

```
catch (Exception e) {  
    System.out.println("caught an exception");  
}
```

- ❖ Podemos regenerar nova exceção de forma a ser tratada num nível superior

```
catch (Exception e) {  
    System.out.println("Exception was thrown");  
    throw e;  
}
```

Declaração de Exceções

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

```
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) {}
```

Java

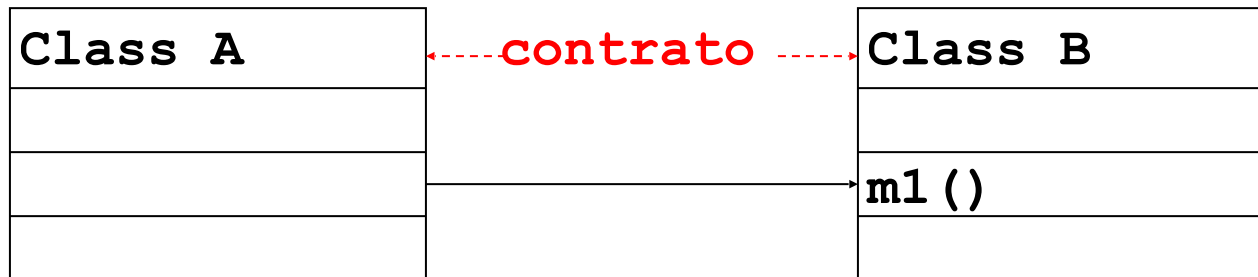
Programação por Contrato

Fiabilidade do Software

- ❖ A qualidade da programação OO mede-se por vários factores:
 - Reutilização
 - Extensibilidade
 - Compatibilidade
- ❖ Independentemente destas características arquiteturais qualquer programa deve ser fiável
 - **Correto** - capacidade para cumprir de acordo com a sua especificação
 - **Robusto** – capacidade para reagir a situações que saiam fora da especificação

Desenho por Contrato

- ❖ Considere-se a relação entre uma classe e os seus clientes



- ❖ Em DpC estas relações significam deveres e direitos (contratos)
 - Sem uma definição precisa e formal (Especificação) não é possível avaliar se a classe “funciona bem”
- ❖ A violação de um contrato leva a erros de run-time

Exemplo

- ❖ Este programa está correto?

```
r = x;  
q = 0;  
while (r > y) do {  
    r = r - y;  
    q = q + 1;  
}
```

- ❖ Não sabemos! Depende do que é suposto ele fazer.
- ❖ Especificação
 - Calcula o quociente q e o resto r como resultados da divisão inteira de x por y

Exemplo

- ❖ Este programa calcula o quociente q e o resto r como resultados da divisão inteira de x por y .

- Está correto?

```
r = x;  
q = 0;  
while (r > y) do {  
    r = r - y;  
    q = q + 1;  
}
```

- ❖ → **SIM!** De acordo com a especificação podemos provar que no final:

$\{ x = y * q + r \}$

Algun tempo mais tarde

O programa não está correto !
Não termina quando $y = 0$!



```
r = x;  
q = 0;  
while ( r > y) do {  
    r = r - y;  
    q = q + 1;  
}
```

Não podemos dividir por zero. Logo a especificação está incompleta.

Devíamos ter “dito” $y > 0$

Exemplo

Assumindo que:

{ $y > 0$ } ←

```
r = x;  
q = 0;  
while ( r > y) do {  
    r = r - y;  
    q = q + 1;  
}
```

Asserções

Podemos provar:

{ $x = y*q + r$ } ←

Algun tempo mais tarde



O programa ainda não está correto !
Quando $x=6$ e $y=3$ o resultado é:
 $q = 1; r = 3$, em vez de
 $q = 2; r = 0$

```
r = x;  
q = 0;  
while ( r > y) do {  
    r = r - y;  
    q = q + 1;  
}
```


Oops! É um erro ... vamos ver...

Exemplo

Assumindo que: $\{ y > 0 \}$  Pré-condições

```
r = x ;  
q = 0 ;  
while (r >= y) do {  
    r = r - y ;  
    q = q + 1 ;  
};
```

Podemos provar: $\{ x = y * q + r \text{ and } r < y \}$


Pós-condições

Formalizando uma Especificação

- ❖ Considere-se qualquer expressão A. A sua formulação pode ser :

$$\{ P \} \ A \ \{ Q \}$$

- ❖ P e Q são asserções:

- P é a pré-condição
- Q é a pós-condição

- ❖ Significado:

- Qualquer execução de A, começando no estado P deverá terminar no estado Q

- ❖ Exemplo

$$\{x \geq 9\} \quad x = x + 5 \quad \{x \geq 14\}$$

Invariantes

- ❖ As **pré-condições** e **pós-condições** descrevem propriedades à entrada e à saída de métodos
- ❖ Os **invariantes** são condições que devem ser sempre respeitadas nos estados estáveis do método, classe ou bloco
- ❖ Exemplo:
 - a classe Stack poderá ter os seguintes invariantes:
`count >= 0`
`count <= capacity`
- ❖ A programação por contrato pode ser parcialmente resolvida em Java à custa da instrução *assert*

Java Asserções

Assertões

- ❖ Uma assertão permite verificar uma invariante, i.e. uma condição que deve ser sempre verdadeira
- ❖ Sintaxe:

`assert booleanExpression [: expression];`

- Se `booleanExpression` for true, a assertão passa.
- Se for false, a assertão falha, sendo gerada um objecto do tipo `AssertionError`
- `expression` é uma expressão opcional que permite passar informação adicional sobre o tipo de problema

Exemplos

```
public class Assert1 {  
    public static void main(String[] args) {  
        assert false;  
    }  
}
```

```
} Exception in thread "main" java.lang.AssertionError  
    at Assert1.main(Assert1.java:8)
```

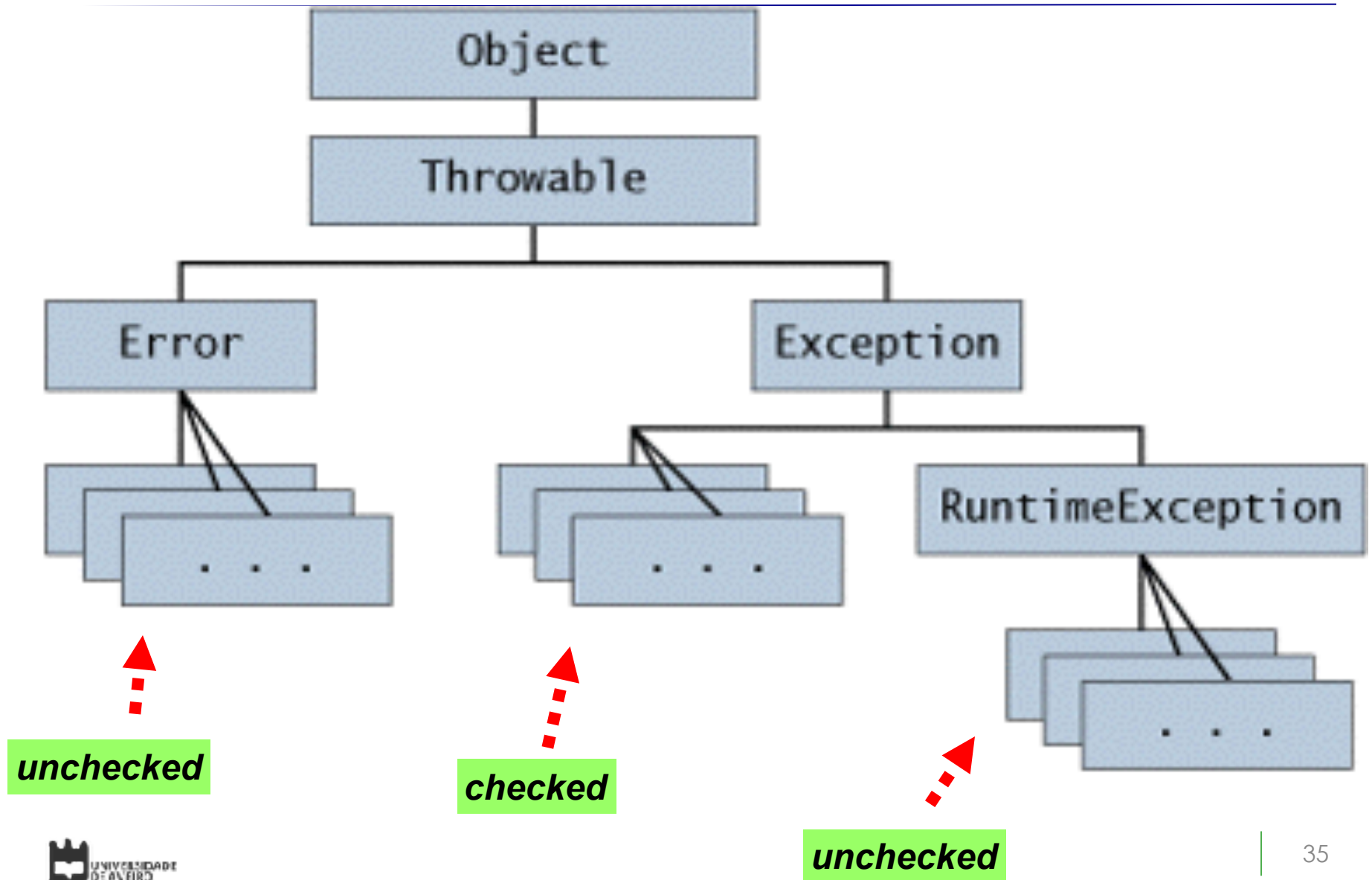
```
public class Assert2 {  
    public static void main(String[] args) {  
        assert false: "Este programa vai parar!";  
    }  
}
```

```
} Exception in thread "main" java.lang.AssertionError: Este  
programa vai parar!  
    at Assert2.main(Assert2.java:6)
```

Asserções

- ❖ Um *AssertionError* é um Erro
 - Não é necessário fazer o `try .. catch`
 - A utilização de um `assert` não pressupõe trabalho adicional por parte do programador
- ❖ As asserções devem ser usadas apenas para testar condições que nunca devem falhar num programa correto
 - Testar se o estado corrente do objecto é correto.
 - Verificar o fluxo do código.

Exceções - Hierarquia de Classes



Exemplos

```
switch (month) {  
    case 1: ... ; break;  
    case 2: ... ; break;  
    ...  
    case 12: ... ; break;  
    default:  
        assert false :  
            "Invalid month: " +  
month  
    }  
}
```

```
...  
double divide( double num,  
               double denom) {  
    assert denom != 0;  
    ...  
}
```

Assertões versus Exceções

- ❖ Uma exceção informa o utilizador do programa que qualquer coisa correu mal
 - Criamos exceções para lidar com problemas que sabemos que podem ocorrer
- ❖ As assertões não devem ser usadas para testar maus funcionamentos do programa
 - Para isso devemos usar exceções (*IOException*, *NullPointerException*, ...).
- ❖ Uma assertão informa sobre a existência de um bug
 - Escrevemos assertões para “confirmar” estados que conhecemos sobre o programa

Quando usar Exceções

- ❖ Testar a validade de parâmetros de entrada de métodos ou construtores públicos
 - Os erros podem surgir por culpa do utilizador.
 - A Exceção é semanticamente mais rica
 - Permite usar `IllegalArgumentException`, `NullPointerException`, `IndexOutOfBoundsException`, ...
- ❖ Gerir entrada e saída de dados
 - O autor de uma classe não pode garantir que estes dados são adequados
- ❖ Resumindo,
 - Pensar apenas como autor da classe
 - Se alguma coisa correr mal, não podendo ser controlada dentro da classe, deve ser gerada uma exceção

Quando usar Asserções

- ❖ Para confirmar estados que devem ser verdadeiros em vários pontos do programa
- ❖ Podem ser usadas para testar a validade de parâmetros de entrada de métodos privados
- ❖ Podem ser usadas para depuração (embora não seja esta a intenção original)
 - Indicando as condições de entrada e de saída
- ❖ São simples de usar.

```
assert age >= 0;  
..  
assert i==0;
```
- ❖ Não devem ser usadas para fazer trabalho necessário
 - Porque podem ser desativadas

As Asserções estão desativadas

- ❖ Por omissão, as asserções não são avaliadas.
 - É possível ligar e desligar as asserções
 - Quando as asserções estão desligadas não são avaliadas, por isso as expressões das asserções não devem produzir efeitos colaterais.

`assert ++i < max;` 😞

`i++; assert i < max;` 😊

❖ Ativar/Desativar Asserções

- Ativar (flags “-enableassertions”)

`java -ea Prog`

- Desativar (flags “-disableassertions”)

`java -da Prog`

Sumário

- ❖ Mecanismos de Controlo de Falhas
- ❖ Exceções
 - try
 - catch
 - throw
- ❖ Programação por Contrato
- ❖ Asserções