

EXCEPÇÕES EM JAVA

EXCEPTION = Exceptional Event

■ Mecanismo que permite o controlo de erros diversos que podem surgir em tempo de execução do código JAVA, alterando o fluxo normal de execução do programa;

■ **Fases fundamentais do mecanismo:**

- 1) Ocorrência da exceção (ou o seu lançamento explícito);
- 2) “Apanhar” a exceção, ou seja, reconhecer a sua ocorrência;
- 3) Fazer o seu tratamento (“handling” por código “handler”);
- 4) Eventualmente executar código adicional.

■ **TIPOS DE EXCEPÇÕES**

- 1) **“Checked Exceptions”**: eventos excepcionais que podem ocorrer e que uma aplicação bem codificada deve poder tratar de forma a prosseguir o seu fluxo sem abortar. Exemplos: divisões por zero, nomes de ficheiros inexistentes, etc.
- 2) **“Excepções de execução”**: eventos internos à aplicação que ocorrem em “runtime”, mas que, em geral, não se pretendem tratar. Deixar que ocorram serve para que a lógica da aplicação seja melhorada, pois estes configuram “bugs” de programação. Exemplos: Má utilização de uma API, excepções de NullPointerException, etc.

- 3) “Erros”:** em geral associados ao hardware e que são irre recuperáveis. Exemplos: “crash” do disco, má leitura de CD.

EXEMPLOS BÁSICOS:

```
private percentagem(double n, double d) {  
    return n/d*100;  
}
```

ou

```
public int main() {  
    Scanner input = new Scanner(System.in);  
    double n = input.nextDouble();  
    out.println("Idade = " + idade);  
    return 0;                                // à la C  
}
```

■ Baseia-se nas seguintes estruturas sintáticas:

```
try {  
    instruções potencialmente "perigosas"  
}  
catch(TipoDeExcepção1 id_var)  
    { código para tratar a exceção1 }  
catch(TipoDeExcepção2 id_var)  
    { código para tratar a exceção2 }  
finally { coisas a fazer depois do try haja ou não exceção }
```

```
throw new Minha_Excepcao();  
throw new Minha_Excepcao("Erro 25");
```

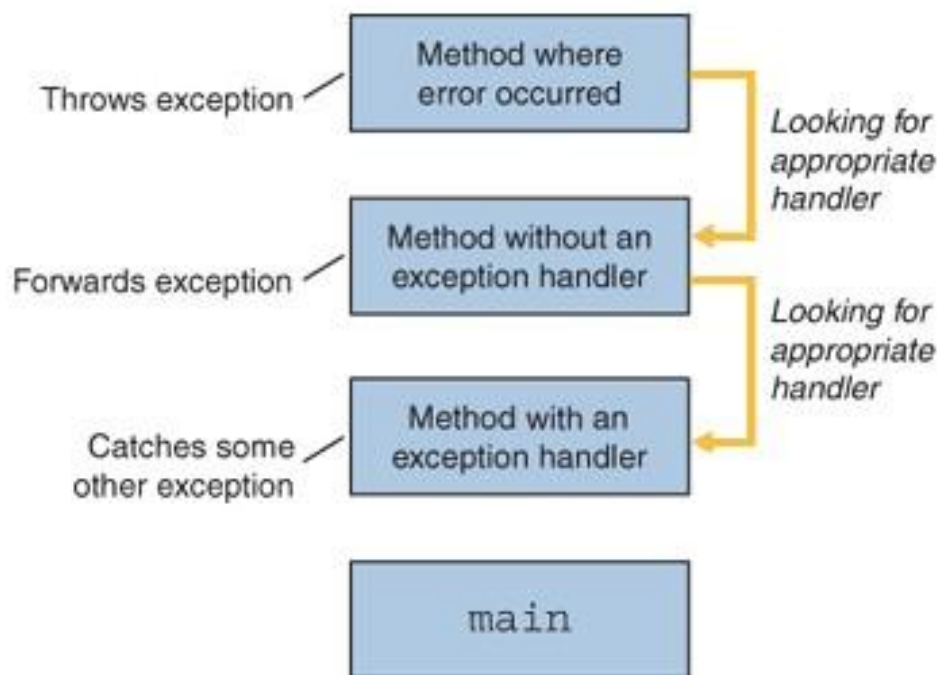
```
public void metodoX() throws MinhaExcepcao1 {  
.....
```

```
if(condição)  
    throw new MinhaExcepcao1("Erro 25");
```

EXEMPLO:

```
→ public void addProd(Produto p) throws StockCheioException { →  
→ → if (stock.size() == MAX_SIZE) {  
→ → → throw new StockCheioException();  
→ → → else {  
→ → → stock.add(p);  
→ → }  
→ }
```

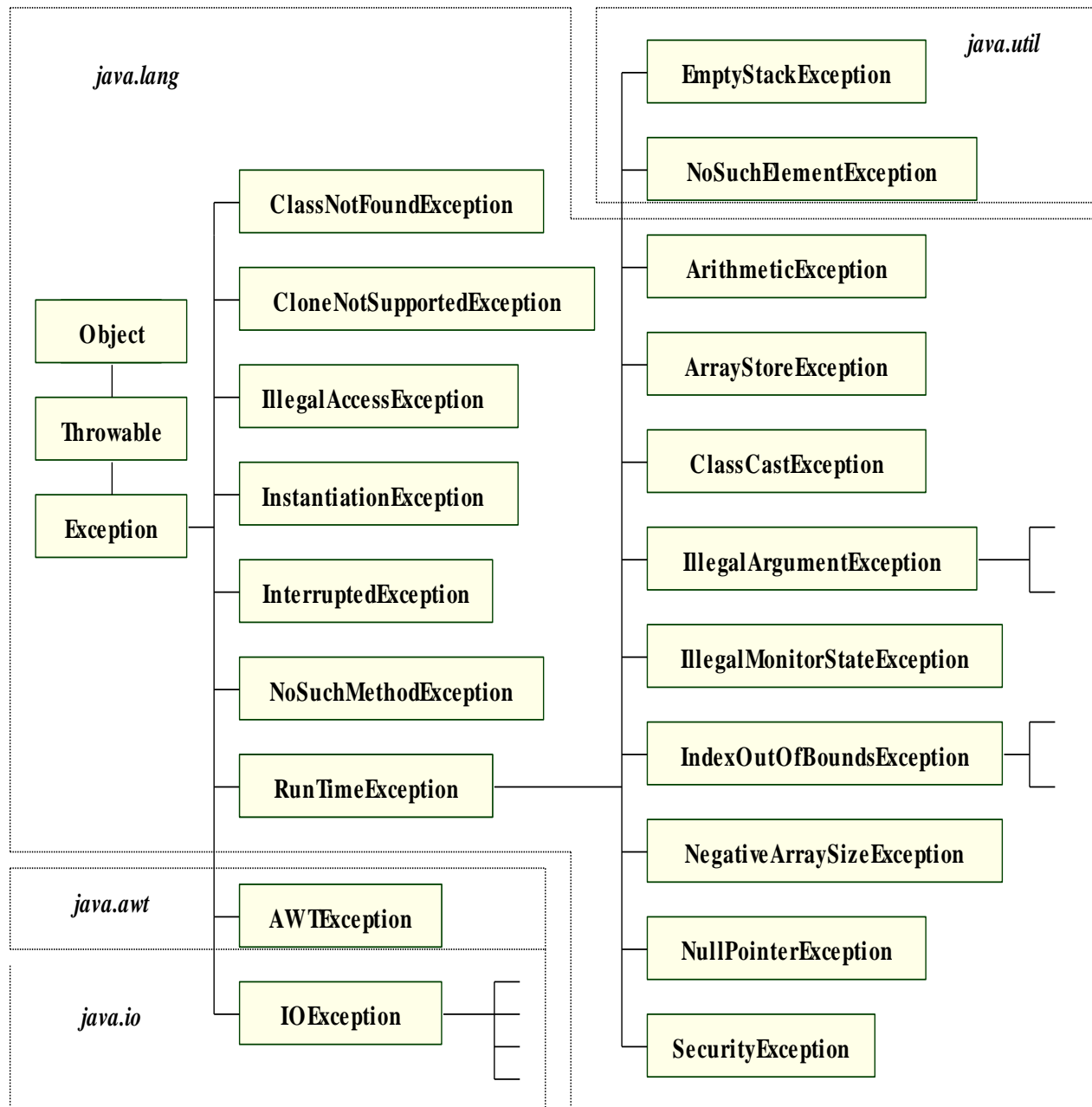
MECANISMO DE EXCEPÇÕES – FUNCIONAMENTO:



NOSSA METODOLOGIA PARA EXCEPÇÕES:

- Em geral os nossos métodos apenas lançarão (**throw**) exceções (declarando-as no seu cabeçalho - declaração **throws**);
- O método **main()** será responsável por realizar o **try/catch** e o tratamento das exceções lançadas pelas camadas mais interiores, ou seja, pelos métodos de classe, de instância e pelos construtores.

QUE EXCEPÇÕES EXISTEM EM JAVA ?



NOTA: As exceções criadas pelo programador serão sempre subclasses de **Exception** !!

EXEMPLOS DE EXCEPÇÕES PRÉ-DEFINIDAS:

public Scanner(File source) throws FileNotFoundException;

public String nextLine() throws NoSuchElementException, IllegalStateException;

public PrintWriter(String fileName) throws FileNotFoundException;

public Integer(String str) throws NumberFormatException;

public static int parseInt(String s) throws NumberFormatException;

public static Integer valueOf(String s) throws NumberFormatException;

EXEMPLO DE try/catch no main:

```
import static java.lang.System.out;
import java.util.Scanner;
public class ProgFactInt1 {
    → // Método auxiliar que calcula o fatorial de n
    → public static int factorial(int n) {
    → → if (n==1) return 1;
    → → else return n*factorial(n-1);
    → }
    → // Programa principal
    → public static void main(String[] args) {
    → → int i = 0;
    → → try {
    → → → i = Integer.parseInt(args[0]);
    → → }
    → → catch (NumberFormatException e) {
    → → → Scanner input = new Scanner(System.in);
    → → → out.println(e.getClass().getSimpleName());
    → → → out.println(e.getMessage());
    → → → out.print("Inteiro: ");
    → → → i = input.nextInt();
    → → }
    → → out.println(i + "!=" + factorial(i));
    → }
}
```

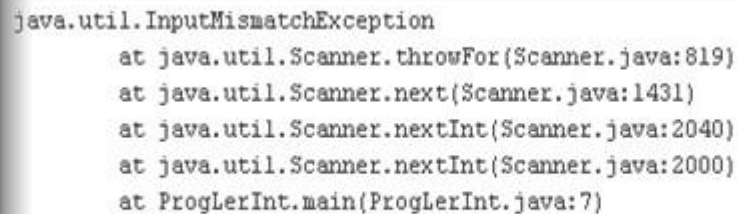
EXEMPLO USANDO Scanner:

Considere-se o programa seguinte contendo “unchecked exceptions” potenciais cf. **nextInt()**

```
public class ProgLerInt {  
→ public static void main(String[] args) {  
→ → Scanner input = new Scanner(System.in);  
→ → out.print("Inteiro: "); int i = input.nextInt();  
→ → out.println("Lido := " + i);  
→ }  
}
```

7

O programa compila correctamente, mas por erro de I/O gera a excepção,



```
java.util.InputMismatchException  
    at java.util.Scanner.throwFor(Scanner.java:819)  
    at java.util.Scanner.next(Scanner.java:1431)  
    at java.util.Scanner.nextInt(Scanner.java:2040)  
    at java.util.Scanner.nextInt(Scanner.java:2000)  
    at ProgLerInt.main(ProgLerInt.java:7)
```

porque em vez de um inteiro introduzimos um número real (exº 123.45).

Vamos reescrever o código de forma a fazer o “check” desta excepção:

```
import static java.lang.System.out;
import java.util.*;
public class ProgLerInt {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        boolean ok = false; int i = 0;
        while(!ok) {
            out.print("Inteiro: ");
            try {
                i = input.nextInt();
                ok = true;
            }
            catch (InputMismatchException e) {
                out.println("Inteiro Inválido!");
            }
            out.println("Lido := " + i);
        }
    }
}
```

I

Este programa, aparentemente correcto, pois de facto faz o “catch” e o tratamento da excepção, entra em ciclo infinito dado que a classe Scanner possui a propriedade de, quando ocorre um erro de leitura, não “limpa o buffer”. Assim, o valor inválido “abc” continua no “buffer”. A solução é tratar de limpar o buffer e reiniciar a leitura.

```

import static java.lang.System.out;
import java.util.*;
public class ProgLerInt {
    //
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        String lixo = "";
        boolean ok = false; int i = 0;
        // leitura e validação
        while (!ok) {
            out.print("Inteiro: ");
            try {
                i = input.nextInt();
                ok = true;
            }
            catch (InputMismatchException e) {
                out.println("Inteiro Invalido");
                lixo = input.nextLine(); // fundamental !
            }
            // resultados
            out.println("Lido = " + i);
            out.println("Lixo = " + lixo);
        }
    }
}

```

EXEMPLOS DE THROWS:

```
→ public void push(Object elem) throws StackCheiaException {  
→ → if (numElem == dim) {  
→ → → throw new StackCheiaException("Stack Cheia!!!");  
→ → → else {  
→ → → { stack.add(elem); numElem++; }  
→ → }  
→ }
```

```
public boolean toma(Cafe cafe) throws CafeQuenteException,  
→ ..... CafeFrioException,  
→ ..... CafeException {  
→ int decisao = 1 + (int) (random() * 100); // 1...100  
→ if (decisao < 10) throw new CafeException("Mal tirado!");  
→ int temp = cafe.getTemp();  
→ if (temp > quente) {  
→ → throw new CafeQuenteException("Quente + " + (temp - quente));  
→ if (temp < frio) {  
→ → throw new CafeQuenteException("Frio - " + (frio - temp));  
→ return true;  
→ }
```

SUMÁRIO (by JAVA TUTORIALS):

A program can catch exceptions by using a combination of the `try`, `catch`, and `finally` blocks.

- The `try` block identifies a block of code in which an exception can occur.
- The `catch` block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The `finally` block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the `try` block.

The `try` statement should contain at least one `catch` block or a `finally` block and may have multiple `catch` blocks.

The class of the exception object indicates the type of exception thrown. The exception object can contain further information about the error, including an error message. With exception chaining, an exception can point to the exception that caused it, which can in turn point to the exception that caused *it*, and so on.



COMO CRIAR EXCEPÇÕES NOSSAS ?

```
public class MinhaExcepcao extends Exception {  
    public MinhaExcepcao() {  
        super();  
    }  
  
    public MinhaExcepcao(String txt) {  
        super(txt);  
    }  
}
```

```

/**
 * Excepção que corresponde a uma tentativa de consultar na
 * biblioteca um Item cuja chave (Titulo) não existe !!
 *
 * @author FMM
 * @version 1.0 (Abril - 2005)
 */

public class TituloNaoExisteException
    extends Exception {
    /** Construtor simples de TituloNaoExisteException */
    public TituloNaoExisteException() { super(); }

    /** Construtor com parâmetro */
    public TituloNaoExisteException(String mensagem) {
        super(mensagem);
    }
}

```



UTILIZAÇÃO

```
/**
 * Classe Equipa
 *
 * @author F. Mário Martins
 * @version 1.0/2006
 */
import java.util.*;
public class Equipa {
    // Variáveis de instância
    private TreeMap<Integer, FichaJogador> equipa;
    private String nomeEquipa;

    // Construtores
    public Equipa(String nomeTeam) {
        nomeEquipa = nomeTeam;
        equipa = new TreeMap<Integer, FichaJogador>();
    }

    public Equipa(String nome, Collection<FichaJogador> team) {
        nomeEquipa = nome;
        equipa = new TreeMap<Integer, FichaJogador>();
        for(FichaJogador ficha : team)
            equipa.put(ficha.getNumero(), ficha.clone());
    }
    // Métodos de Instância

    public String getNomeEquipa() { return nomeEquipa; }
```

```

public void insereJogador(FichaJogador ficha)
    throws JogJaExisteException {
    if(equipa.containsKey(ficha.getNumero()))
        throw new JogJaExisteException("O N° " + ficha.getNumero() + " ja existe !");
    else
        equipa.put(ficha.getNumero(), ficha.clone());
}

```

```

public FichaJogador consultaJogador(int num)
    throws JogNaoExisteException {
    if(!equipa.containsKey(num))
        throw new JogNaoExisteException("O N° " + num + " nao existe !");
    return equipa.get(num).clone();
}

```

```

public int inscritos() { return equipa.size(); }

```

```

public Set<Integer> listaNumeros() {
    Set<Integer> nums = new TreeSet<Integer>();
    for(int num : equipa.keySet()) // filtra null
        nums.add(num);
    return nums;
}

```

```

public String toString() {
    StringBuilder s = new StringBuilder();
    s.append(" EQUIPA " + nomeEquipa + "\n");
    for(FichaJogador ficha : equipa.values())
        s.append(ficha.toString());
    return s.toString();
}

```



```
}
```

----- CLASSES DE EXCEPÇÃO -----

```
/**
 * Write a description of class JogNaoExisteException here.
 *
 * @author F. Mário Martins
 * @version 1/2006
 */
public class JogNaoExisteException extends Exception {
    public JogNaoExisteException() { super(); }
    public JogNaoExisteException(String s) { super(s); }
}
```

```
/**
 * Write a description of class JogJaExisteException here.
 *
 * @author F. Mário Martins
 * @version 1/2006
 */
public class JogJaExisteException extends Exception {
    public JogJaExisteException() { super(); }
    public JogJaExisteException(String s) { super(s); }
}
```