8. EXCEÇÕES E STREAMS

PROGRAMAÇÃO ORIENTADA AOS OBJETOS

Desenvolvido por:

Carlos Urbano
Catarina Reis
José Magno
Marco Ferreira
Ricardo Antunes

ÍNDICE

8.1. CONCEITO DE EXCEÇÃO

Uma exceção é um evento que ocorre durante a execução de um programa e que interrompe o fluxo normal das instruções desse programa

Mais concretamente, é um objeto que representa uma condição anormal que ocorre durante a execução de um programa

Cada vez que ocorre uma exceção (hardware ou software), o interpretador cria um objeto que descreve essa exceção

Uma exceção contém informação sobre:

- Tipo
- O estado do programa quando ocorreu

8.1. CONCEITO DE EXCEÇÃO

Após a ocorrência de uma exceção, o interpretador procura o código que a possa tratar

Um bloco típico para tratar exceções tem a seguinte forma:

```
try {
    // bloco de código que pode originar erros
} catch (ExcecaoTipo1 id) {
    // código para tratar a exceção ExcecaoTipo1
} catch (ExcecaoTipo2 | ExcecaoTipo3 id) {
    // código para tratar a exceção ExcecaoTipo2 ou ExcecaoTipo3
    throw(id); // torna a lançar a exceção se for necessário
} finally {
    // código de finalização (sempre executado)
}
```

8.1. CONCEITO DE EXCEÇÃO

Vantagens

1 - separa o código de tratamento de erros do restante

2 - permite a propagação de erros através dos métodos invocados

3 - permite agrupar tipos de erros relacionados

8.1. CONCEITO DE EXCEÇÃO

Separação do código de tratamento de erros do restante
Exemplo:

```
lerFicheiro() {
    abre o ficheiro;
    determina o seu tamanho;
    reserva memória para guardar o ficheiro;
    lê o ficheiro para a memória;
    fecha o ficheiro;
}
```

O que acontece se:

- **O ficheiro não pode ser aberto?**
- **O tamanho do ficheiro não pode ser determinado?**
- **A memória não pode ser reservada?**
- **A leitura falha?**
- **O ficheiro não pode ser fechado?**

8.1. CONCEITO DE EXCEÇÃO

ABORDAGEM TRADICIONAL

```
int lerFicheiro() {
    int tipoErro = 0;
    abre o ficheiro;
    if (ficheiroAberto) {
        determina o seu tamanho;
        if (obteveTamanho) {
            reserva memória para guardar o ficheiro;
            if (obteveMemoria) {
                lê o ficheiro para a memória;
                if (leituraFalha) tipoErro = -1;
            } else tipoErro = -2;
        } else tipoErro = -4;
        fecha o ficheiro;
        if (ficheiroNaoFechou && tipoErro == 0)
            tipoErro = -8;
        else
            tipoErro = tipoErro - 8;
    } else
        tipoErro = -16;
    return tipoErro;
}
```

8.1. CONCEITO DE EXCEÇÃO

ABORDAGEM UTILIZANDO EXCEÇÕES

```
void lerFicheiro() {
    try {
        abre o ficheiro;
        determina o seu tamanho;
        reserva memória para guardar o ficheiro;
        lê o ficheiro para a memória;
        fecha o ficheiro;
    } catch (ficheiroNaoAbre) {
        Trata o erro;
    } catch (naoObteveTamanho) {
        Trata o erro;
    } catch (naoObteveMemoria) {
        Trata o erro;
    } catch (leituraFalha) {
        Trata o erro;
    } catch (ficheiroNaoFecha) {
        Trata o erro;
    }
}
```

8.1. CONCEITO DE EXCEÇÃO

PROPAGAÇÃO DE ERROS ATRAVÉS DOS MÉTODOS INVOCADOS

VEJAMOS O SEGUINTE CENÁRIO:

```
metodo1() {
   metodo2();
}

metodo2() {
    metodo3();
}

metodo3() {
    lerFicheiro();
}
```

O ÚNICO MÉTODO INTERESSADO NO TRATAMENTO DE ERROS É O **metodo1()**

8.1. CONCEITO DE EXCEÇÃO

ABORDAGEM TRADICIONAL

```
void metodo1() {
    tipoErro = metodo2();
    if (tipoErro != 0) {
        trata o erro;
    } else {
        continua;
    }
}

TipoErro metodo2() {
    tipoErro = metodo3();
    if (tipoErro != 0) {
        return erro;
    }
    continua;
}

TipoErro metodo3() {
    tipoErro = lerFicheiro();
    if (tipoErro != 0) {
        return erro;
    }
    continua;
}
```

metodo2() E metodo3()
SÃO OBRIGADOS A PROPAGAR O ERRO

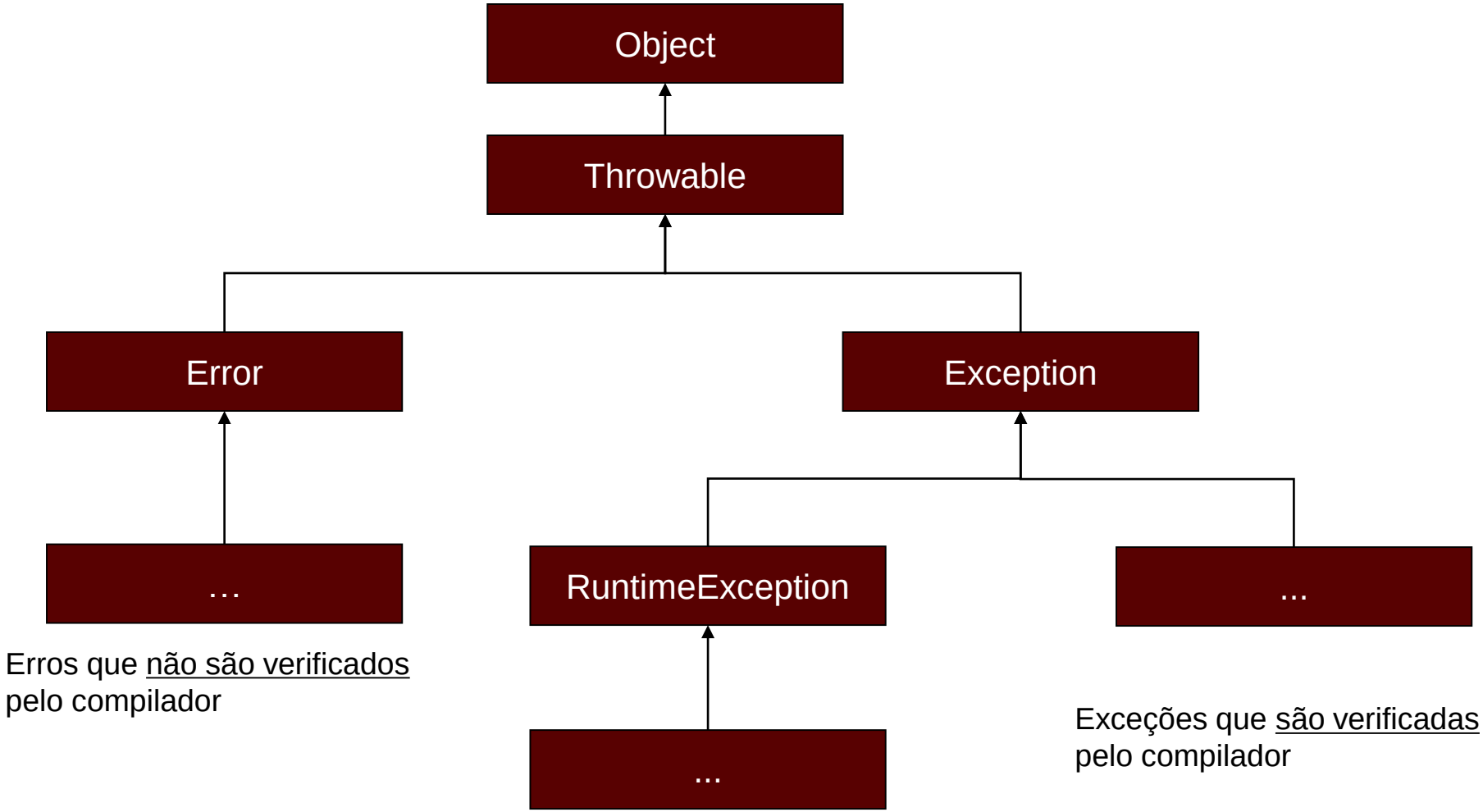
8.1. CONCEITO DE EXCEÇÃO

ABORDAGEM UTILIZANDO EXCEÇÕES

```
void metodo1() {
    try {
        metodo2();
    } catch (ficheiroNaoAbre) {
        trata o erro;
    } catch (naoObteveTamanho) {
    }
    …
}

void metodo2() throws ficheiroNaoAbre, naoObteveTamanho,
                      naoObteveMemoria, leituraFalha,
                      ficheiroNaoFecha {
    metodo3();
}

void metodo3() throws ficheiroNaoAbre, naoObteveTamanho,
                      naoObteveMemoria, leituraFalha,
                      ficheiroNaoFecha {
    lerFicheiro();
}
```

MÉTODOS APENAS INDICAM AS EXCEÇÕES QUE PROPAGAM (`throws`)

8.2. HIERARQUIA DE EXCEÇÕES



8.2. HIERARQUIA DE EXCEÇÕES

- **`Error`**

Classe base que descreve os erros não esperados ou erros que não devem ser tratados em circunstâncias normais (Ex: `VirtualMachineError`, `OutOfMemoryError`)

- **`Exception`**

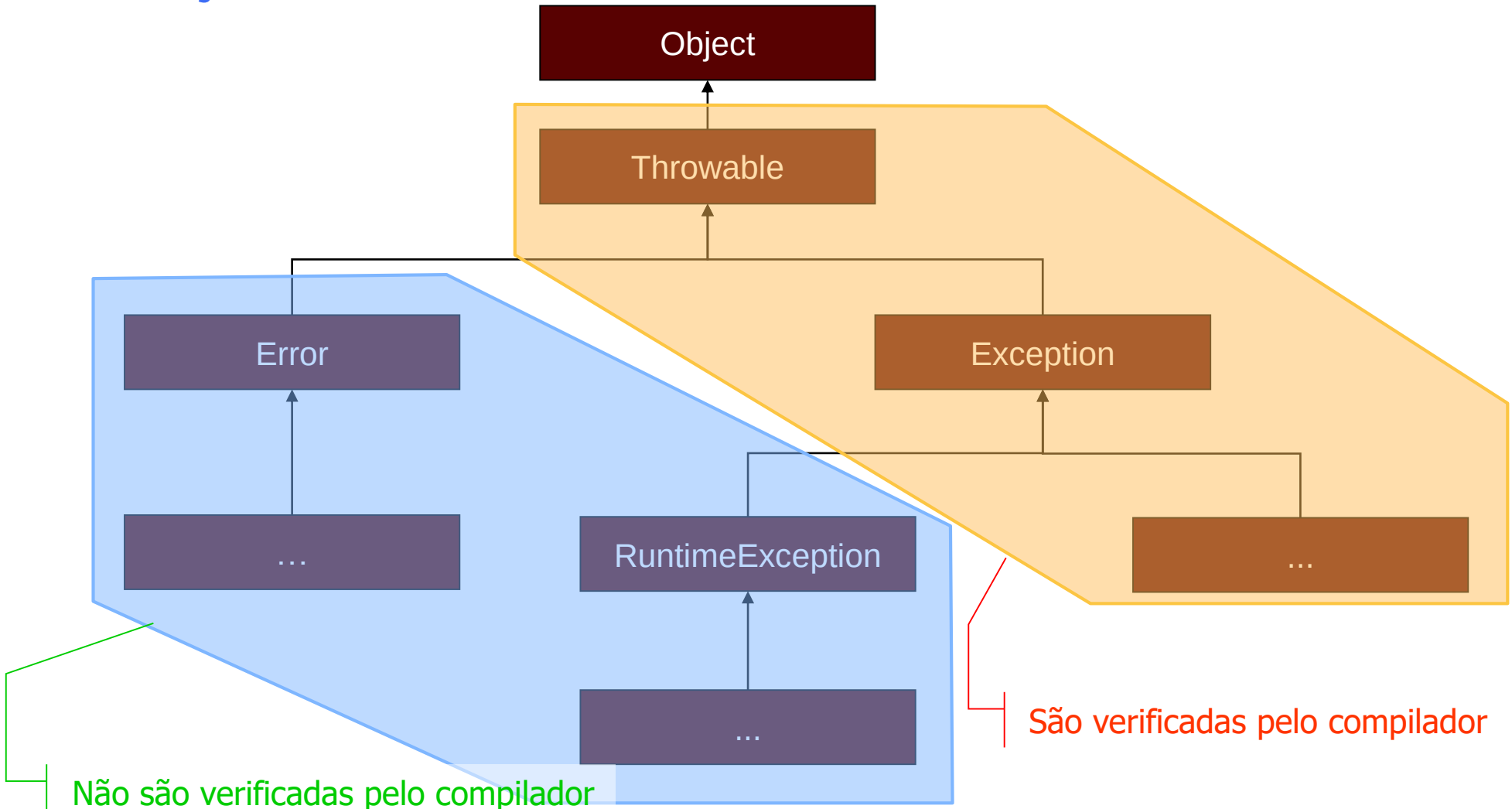
Classe base que descreve uma condição anormal que deve ser tratada pelo programa

- **`RuntimeException`**

Classe base que descreve as exceções que não são verificadas em tempo de compilação e mas apenas em tempo de execução (Ex: `ArithmeticException`)

8.2. HIERARQUIA DE EXCEÇÕES

DURANTE A COMPILAÇÃO HÁ EXCEÇÕES VERIFICADAS E EXCEÇÕES NÃO VERIFICADAS PELO COMPILADOR



8.3. TIPOS DE EXCEÇÕES

EXCEÇÕES QUE NÃO SÃO VERIFICADAS PELO COMPILADOR

```
public class Exemplo1 {
    public static void main(String[] args) {
        int a = 5;
        int b = 0;
        int res = a / b;

        System.out.println(a + "/" + b + "=" + res);
    }
}
```

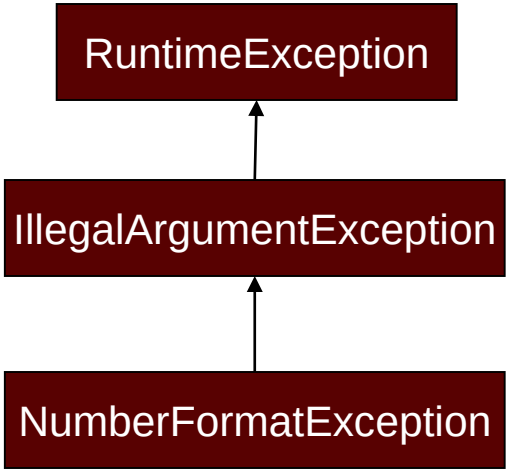
```
public class ArithmeticException
extends RuntimeException
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at cap8_excecoes_streams.Exemplo1.main(Exemplo1.java:7)
```

8.3. TIPOS DE EXCEÇÕES

EXCEÇÕES QUE NÃO SÃO VERIFICADAS PELO COMPILADOR

```
public class Exemplo2 {
    public static void main(String[] args) {
        String numStr = "5";
        // converte uma string contendo
        // um número binário (radix=2) num int
        int valor = Integer.parseInt(numStr, 2);

        System.out.println(numStr +" = " + valor);
    }
}
```




```
Exception in thread "main" java.lang.NumberFormatException: For input string: "5"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at cap8_excecoes_streams.Exemplo2.main(Exemplo2.java:8)
```

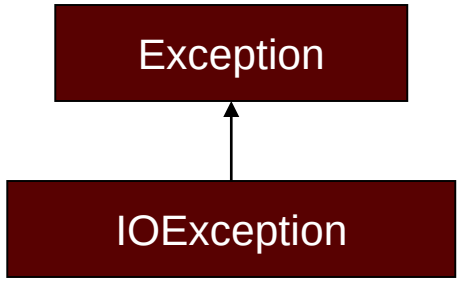
```
public static int parseInt(String s, int radix)
                    throws NumberFormatException
```

A PALAVRA **throws** INDICA QUE O MÉTODO PODE GERAR UMA EXCEÇÃO DO TIPO **NumberFormatException**

8.3. TIPOS DE EXCEÇÕES

EXCEÇÕES QUE SÃO VERIFICADAS PELO COMPILADOR

```
public class Exemplo3 {
    public static void main(String[] args) {
        System.out.print("Introduza um caracter: ");

        char ch = (char) System.in.read(); Unhandled exception: java.io.IOException

        System.out.println("Caracter = " + ch);
    }
}
```




```
public abstract int read()
                throws IOException
```

8.4. TRATAMENTO DE EXCEÇÕES

Todas as exceções verificadas, pelo compilador, têm de ser tratadas, caso contrário o programa não compila

Existem duas formas possíveis de abordar exceções:

- utilizando um bloco `try-catch`
- declarando que vai propagar a exceção (`throws`)

Um método pode apanhar exceções, com origem:

- no próprio método, através do `throw`
- na chamada de outros métodos

8.4. TRATAMENTO DE EXCEÇÕES

BLOCO try-catch

```
try {
    // bloco de código que pode originar erros
} catch (ExcecaoTipo1 id) {
    // código para tratar a exceção ExcecaoTipo1
} catch (ExcecaoTipo2 | ExcecaoTipo3 id) {
    // código para tratar a exceção ExcecaoTipo2 ou ExcecaoTipo3
    throw(id); // torna a lançar a exceção se for necessário
} finally {
    // código de finalização (sempre executado)
}
```

O bloco de código compreendido entre o **try** e o primeiro **catch** contém o código que pode originar erros

8.4. TRATAMENTO DE EXCEÇÕES

Cada bloco `catch` trata um ou mais tipos de exceção

O bloco `finally` é utilizado, usualmente, para libertar recursos detidos por um ou mais objetos independentemente da ocorrência de erros

Dentro de um bloco `catch` é possível relançar uma exceção para ser tratada noutro método

8.4. TRATAMENTO DE EXCEÇÕES

BLOCO try-catch

```
public class Exemplo3Tratado {
    public static void main(String[] args) {
        System.out.print("Introduza um caracter: ");
        try {
            char ch = (char) System.in.read();
            System.out.println("Caracter = " + ch);
        } catch (IOException e) {
            System.err.println("Ocorreu um erro de leitura");
        }

    }
}
```


```
Introduza um caracter: a
Caracter = a
```

8.4. TRATAMENTO DE EXCEÇÕES

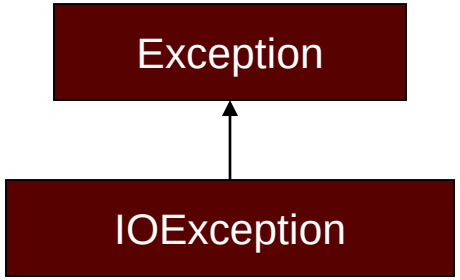
CONSIDERE O SEGUINTE EXEMPLO :

```
public class Exemplo4 {
    // Código para ler facilmente valores do teclado
    private static BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));

    public static void main(String[] args) {
        long n1 = leLong();
        int n2 = leBinario();
        System.out.println(n1 + "+" + n2 + "=" + (n1 + n2));
    }

    private static long leLong() {
        System.out.print("Introduza um long (base 10): ");
        String num = teclado.readLine();
        long valor = Long.parseLong(num);
        return valor;
    }

    private static int leBinario() {
        System.out.print("Introduza um inteiro (base 2): ");
        String num = teclado.readLine();
        int valor = Integer.parseInt(num, 2);
        return valor;
    }
}
```

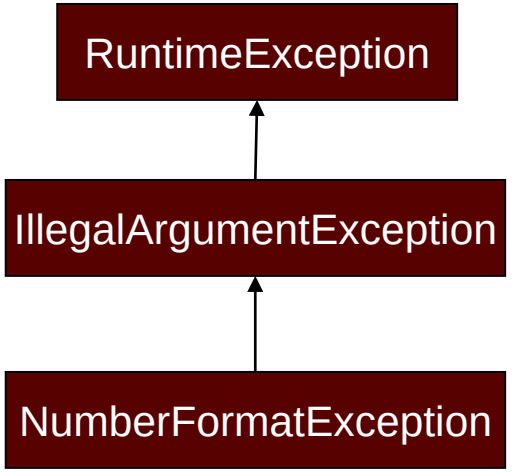
8.4. TRATAMENTO DE EXCEÇÕES

SABENDO QUE:

- **O MÉTODO `readLine()` PODE GERAR UMA EXCEÇÃO DO TIPO `IOException`**



- **OS MÉTODOS `parseInt(…)` E `parseLong(…)` PODEM GERAR UMA EXCEÇÃO DO TIPO `NumberFormatException`**



8.4. TRATAMENTO DE EXCEÇÕES

ADICIONE À CLASSE `Exemplo4` O CÓDIGO NECESSÁRIO PARA EFETUAR O TRATAMENTO DE EXCEÇÕES UTILIZANDO BLOCOS `try-catch`

- UTILIZANDO APENAS BLOCOS `try-catch` CONSEGUE TRATAR AS EXCEÇÕES DE FORMA SATISFATÓRIA?

8.4. TRATAMENTO DE EXCEÇÕES


```
public class Exemplo4Tratado1 {
    // Código para ler facilamente valores do teclado
    private static BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));

    public static void main(String[] args) {
        long n1 = leLong();
        int n2 = leBinario();
        System.out.println(n1 + "+" + n2 + "=" + (n1 + n2));
    }

    private static long leLong() {
        System.out.print("Introduza um long (base 10): ");
        long valor = -1;
        try {
            String num = teclado.readLine();
            valor = Long.parseLong(num);
        } catch (IOException e) {
            System.err.println("Ocorreu um erro de leitura");
        } catch (NumberFormatException e) {
            System.err.println("Formato inválido");
        }
        return valor;
    }

    …
```

8.4. TRATAMENTO DE EXCEÇÕES


```
    …

    private static int leBinario() {
        System.out.print("Introduza um inteiro (base 2): ");
        int valor = -1;
        try {
            String num = teclado.readLine();
            valor = Integer.parseInt(num, 2);
        } catch (IOException e) {
            System.err.println("Ocorreu um erro de leitura");
        } catch (NumberFormatException e) {
            System.err.println("Formato inválido");
        }
        return valor;
    }
}
```

8.4. TRATAMENTO DE EXCEÇÕES

PROPAGAÇÃO DE EXCEÇÕES

- Um método utiliza a palavra **throws** para declarar as exceções que pode lançar
- A sintaxe completa para definir um método é a seguinte:

```
TipoRetorno nomeMetodo(listaParametros) throws ExcecaoTipo1, …, ExcecaoTipoN {
    // Código do método
}
```

EXEMPLO:

```
import java.io.IOException;
public class ThrowsDemo {
    public static void main(String[] args) {
        try {
            metodo1();
        } catch (IOException e) {
            System.out.println("Apanhei a exceção");
        }
    }

    private static void metodo1() throws IOException {
        throw new IOException();
    }
}
```

8.4. TRATAMENTO DE EXCEÇÕES

UTILIZANDO BLOCOS `try-catch` E A PROPAGAÇÃO DE EXCEÇÕES (`throws`) ALTERE A CLASSE `Exemplo4` A FIM DE EFETUAR CORRETAMENTE O TRATAMENTO DE EXCEÇÕES

8.4. TRATAMENTO DE EXCEÇÕES


```
public class Exemplo4Tratado2 {
    // Código para ler facilamente valores do teclado
    private static BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));

    public static void main(String[] args) {
        try {
            long n1 = leLong();
            int n2 = leBinario();
            System.out.println(n1 + "+" + n2 + "=" + (n1 + n2));
        } catch (IOException e) {
            System.err.println("Ocorreu um erro de leitura");
        } catch (NumberFormatException e) {
            System.err.println("Formato inválido");
        }
    }

    private static long leLong() throws IOException {
        System.out.print("Introduza um long (base 10): ");
        String num = teclado.readLine();
        long valor = Long.parseLong(num);
        return valor;
    }

    private static int leBinario() throws IOException {
        System.out.print("Introduza um inteiro (base 2): ");
        String num = teclado.readLine();
        int valor = Integer.parseInt(num, 2);
        return valor;
    }
}
```

8.4. TRATAMENTO DE EXCEÇÕES

ALTERE O MÉTODO `leLong()` DA CLASSE `Exemplo4` A FIM DE LER UM `long` COMPREENDIDO ENTRE UM VALOR MÍNIMO E MÁXIMO

- O QUE TEVE DE ALTERAR?

8.4. TRATAMENTO DE EXCEÇÕES


```
public class Exemplo4Tratado3 {
    // Código para ler facilamente valores do teclado
    private static BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in));

    public static void main(String[] args) {
        try {
            long n1 = leLong(1, 19);
            int n2 = leBinario();
            System.out.println(n1 + "+" + n2 + "=" + (n1 + n2));
        } catch (IOException e) {
            System.err.println("Ocorreu um erro de leitura");
        } catch (NumberFormatException e) {
            System.err.println("Formato inválido");
        }
    }

    private static long leLong(long min, long max) throws IOException {
        System.out.print("Introduza um long (base 10) entre " + max + " e " + min + ": ");
        long valor = min - 1;
        String num;
        do {
            try {
                num = teclado.readLine();
                valor = Long.parseLong(num);
            } catch (NumberFormatException e) {
                System.err.println("Formato inválido");
            }
        }
        while (valor < min || valor > max);
        return valor;
    }
    …
}
```

8.5. CLASSE Throwable

A classe `Throwable` é classe base de todas as exceções e fornece os seguintes métodos:

- `String getMessage()` - devolve a mensagem da exceção
- `void printStackTrace()` - escreve todo o caminho percorrido (pilha de métodos invocados) até à origem da exceção
- `String toString()` - devolve uma `String` com a descrição completa da exceção

8.5. CLASSE Throwable

EXEMPLO:

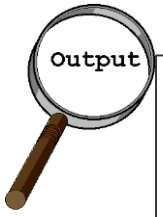
```
public class SuperClasseThrowable {
    public static void main(String[] args) {
        try {
            throw new Exception("Vou lançar uma exceção");
        } catch (Exception e) {
            System.err.println("Apanhei a exceção");
            System.err.println("e.getMessage(): " + e.getMessage());
            System.err.println("e.toString(): " + e.toString());
            System.err.println("e.printStackTrace():");
            e.printStackTrace();
        }
    }
}
```

8.5. CLASSE Throwable

Todas as exceções (subclasses de `Throwable`) vão herdar os métodos anteriores

Para lançar uma exceção utiliza-se a palavra `throw`

O resultado da aplicação anterior é o seguinte:




```
Apanhei a exceção
e.getMessage(): Vou lançar uma exceção
e.toString(): java.lang.Exception: Vou lançar uma exceção
e.printStackTrace():
java.lang.Exception: Vou lançar uma exceção
     at cap8_excecoes_streams.SuperClasseThrowable.main(SuperClasseThrowable.java:6)
```

8.6. LANÇAMENTO DE EXCEÇÕES

Um método lança uma exceção utilizando o **throw**

- a sintaxe para criar uma nova exceção é:

```
throw new NomeDaExceção();
```

ou

```
throw new NomeDaExceção("descrição");
```

- a sintaxe para relançar uma exceção já criada é:

```
throw instanciaSubclasseThrowable;
```

8.6. LANÇAMENTO DE EXCEÇÕES

Assim que a exceção é lançada, a execução do programa é interrompida e tenta encontrar na pilha de métodos invocados um bloco `try-catch` que permita tratar esse tipo de exceção

Se, ao chegar ao fim da pilha, não tiver encontrado nenhum bloco `try-catch` que trate esse tipo de exceção o programa aborta escrevendo a exceção e o respetivo percurso

8.7. ENCAMINHAMENTO DE EXCEÇÕES


```java
public class EncaminhaDemo {
    public static void main(String[] args) {
        try {
            metodo1();
        } catch (Exception e) {
            System.err.println("Exceção apanhada no método main:\n" + e + "\nPercurso:");
            e.printStackTrace(); // Escreve o percurso da exceção
        }
    }

    private static void metodo1() {
        metodo2();
    }

    private static void metodo2() {
        try {
            metodo3();
        } catch (Exception e) {
            System.err.println("Exceção apanhada no método 2:" + e.getMessage());
            throw new RuntimeException(e.getMessage() + "->transformada"); // Lança uma nova exceção
        }
    }

    private static void metodo3() throws Exception {
        try {
            throw new Exception("Demonstração");
        } catch (Exception e) {
            System.err.println("Exceção apanhada no método 3");
            throw e; // Lança novamente a exceção
        }
    }
}
```

8.7. ENCAMINHAMENTO DE EXCEÇÕES

NO CASO DO EXEMPLO ANTERIOR, A PILHA DE MÉTODOS INVOCADOS (*call stack*) ATÉ OCORRER A PRIMEIRA EXCEÇÃO É A SEGUINTE:

SÓ SE CONSEGUE GARANTIR A ORDEM DO OUTPUT UTILIZANDO

`System.err.println(…)`

O resultado do exemplo anterior seria:

```
Exceção apanhada no método 3
Exceção apanhada no método 2:Demonstração
Exceção apanhada no método main:
java.lang.RuntimeException: Demonstração->transformada
Percurso:
java.lang.RuntimeException: Demonstração->transformada
     at cap8_excecoes_streams.EncaminhaDemo.metodo2(EncaminhaDemo.java:22)
     at cap8_excecoes_streams.EncaminhaDemo.metodo1(EncaminhaDemo.java:14)
     at cap8_excecoes_streams.EncaminhaDemo.main(EncaminhaDemo.java:6)
```

8.8. PROPAGAÇÃO DE EXCEÇÕES

Como já foi referido anteriormente, um método declara os tipos de exceção que pode lançar através do `throws`

Um método apenas deve declarar os tipos de exceção que são verificadas pelo compilador

- No entanto, sempre que for importante dar conhecimento de exceções não verificadas (do tipo `RuntimeException`), estas podem ser declaradas (ex: `NumberFormatException`)

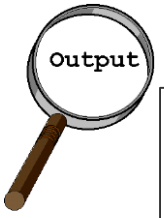
8.8. PROPAGAÇÃO DE EXCEÇÕES


```
public class PropagaDemo {
    public static void main(String[] args) {
        metodo1();
        metodo2();
    }

    // Versão possível mas inapropriado
    private static void metodo1() thro  RuntimeException {
        throw new RuntimeException("Exceção não verificada");
    }

    // Versão correta
    private static void metodo2() {
        throw new RuntimeException("Exceção não verificada");
    }
}
```

8.9. BLOCO FINALLY

O bloco `finally` é um bloco especial que é executado sempre que a execução de um bloco `try` termina

```
public class FinallyDemo1 {
    public static void main(String[] args) {
        try {
            metodo1();
        } catch (Exception e) {
            System.err.println("Apanhei a exceção:\n" + e);
        } finally {
            System.err.println("Fim do programa");
        }
    }

    private static void metodo1() throws Exception {
        try {
            System.out.println("Início método 1");
            throw new Exception("Método1");
        } finally {
            System.err.println("Fim método 1");
        }
    }
}
```




```
Início método 1
Fim método 1
Apanhei a exceção:
java.lang.Exception: Método1
Fim do programa
```

8.9. BLOCO FINALLY


```
public class FinallyDemo2 {
    public static void main(String[] args) {
        metodo1(-1);
        metodo1(3);
    }

    private static void metodo1(int a) {
        try {
            if (a < 0) {
                throw new Exception("Valor negativo");
            }
        } catch (Exception e) {
            System.err.println("Apanhei a exceção: " + e.getMessage());
        } finally {
            System.err.println("Passo 1");
        }
        try {
            if (a > 0) {
                throw new Exception("Valor positivo");
            }
            return;
        } catch (Exception e) {
            System.err.println("Apanhei a exceção: " + e.getMessage());
        } finally {
            System.err.println("Passo 2");
        }
    }
}
```

???

8.9. BLOCO FINALLY


```
public class FinallyDemo2 {
    public static void main(String[] args) {
        metodo1(-1);
        metodo1(3);
    }

    private static void metodo1(int a) {
        try {
            if (a < 0) {
                throw new Exception("Valor negativo");
            }
        } catch (Exception e) {
            System.err.println("Apanhei a exceção: " + e.getMessage());
        } finally {
            System.err.println("Passo 1");
        }
        try {
            if (a > 0) {
                throw new Exception("Valor positivo");
            }
            return;
        } catch (Exception e) {
            System.err.println("Apanhei a exceção: " + e.getMessage());
        } finally {
            System.err.println("Passo 2");
        }
    }
}
```


```
Apanhei a exceção: Valor negativo
Passo 1
Passo 2
Passo 1
Apanhei a exceção: Valor positivo
Passo 2
```

8.9. BLOCO FINALLY

SUPONHA QUE UM ROBOT PODE:

- EMBATER NUM OBSTÁCULO ENQUANTO EFETUA O SEU PERCURSO
- PERDER A ORIENTAÇÃO

```
public class Robot {
    private boolean ligado = false;

    public void liga() {
        ligado = true;
    }

    public void desliga() {
        ligado = false;
    }

    public void efetuaPercurso() {
        …
    }
}
```

```
public class Fabrica {
    public static void main(String[] args) {
        Robot r = new Robot();
        try {
            r.liga();
            r.efetuaPercurso();
            r.desliga();
        } catch (ObstaculoException e) {
            System.err.println("Embateu num obstáculo");
            r.desliga();
        } catch (SemOrientaçãoException e) {
            System.err.println("Perdeu a orientação");
            r.desliga();
        }
    }
}
```

8.9. BLOCO FINALLY

O OBJETIVO É DESLIGAR O ROBOT NO FINAL DO PERCURSO

OU SEMPRE QUE OCORRA UM ERRO

```
public class Fabrica {
    public static void main(String[] args) {
        Robot r = new Robot();
        try {
            r.liga();
            r.efetuaPercurso();
        } catch (ObstaculoException e) {
            System.err.println("Embateu num obstáculo");
        } catch (SemOrientaçãoException e) {
            System.err.println("Perdeu a orientação");
        } finally {
            r.desliga();
        }
    }
}
```

O BLOCO `finally` GARANTE A LIBERTAÇÃO DOS RECURSOS RESERVADOS DENTRO DE UM BLOCO `try-catch` MESMO QUE OCORRAM ERROS

8.10. TRY-WITH-RESOURCES

`try-with-resources` é uma forma de `try` que declara um ou mais rescursos

Um recurso é um objeto que deve ser fechado antes do programa terminar

`try-with-resources` assegura que cada recurso é fechado no fim desse `try`

Qualquer objeto que implemente `java.lang.AutoCloseable` pode ser usado como recurso

Até à versão 7 do java, utilizava-se o bloco `finally` para fechar os recursos, independentemente do `try` terminar normal ou anormalmente

8.10. TRY-WITH-RESOURCES

EXEMPLO COM try SIMPLES

```
BufferedReader br = null;

try {

    String line;

    br = new BufferedReader(new FileReader("C:\\testing.txt"));

    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }

} catch (IOException e) {
    e.printStackTrace();
} finally {
    try {
        if (br != null) br.close();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

RECURSOS FECHADOS

NO BLOCO finally

8.10. TRY-WITH-RESOURCES

EXEMPLO COM try-with-resources

```
try (BufferedReader br = new BufferedReader(new FileReader("C:\\testing.txt"))) {

    String line;

    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }

} catch (IOException e) {
    e.printStackTrace();
}
```

BLOCO finally

DESNECESSÁRIO

RECURSOS FECHADOS

APÓS BLOCO try

8.11. DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO

CENÁRIO DE APLICAÇÃO:

IMPLEMENTAÇÃO DE UMA CLASSE QUE REPRESENTA UMA LISTA LIGADA E QUE POSSUI OS SEGUINTES MÉTODOS:

- `objectAt(int n)`
- `firstObject()`
- `indexOf(Object objeto)`

8.11. DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO

QUE TIPO DE ERROS PODEM OCORRER?

- `objectAt(int n)` - O VALOR DE `n` PODE SER INFERIOR A `0` OU MAIOR QUE O NÚMERO DE ELEMENTOS CONTIDOS NA LISTA
- `firstObject()` - A LISTA PODE ESTAR VAZIA
- `indexOf(Object objeto)` - O OBJETO `objeto` PODE NÃO EXISTIR NA LISTA

8.11. DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO

QUE TIPOS DE EXCEÇÃO SE DEVEM UTILIZAR?

- **UTILIZAR AS EXCEÇÕES INCLUÍDAS NA API DO JAVA**
- **DEFINIR NOVAS EXCEÇÕES**

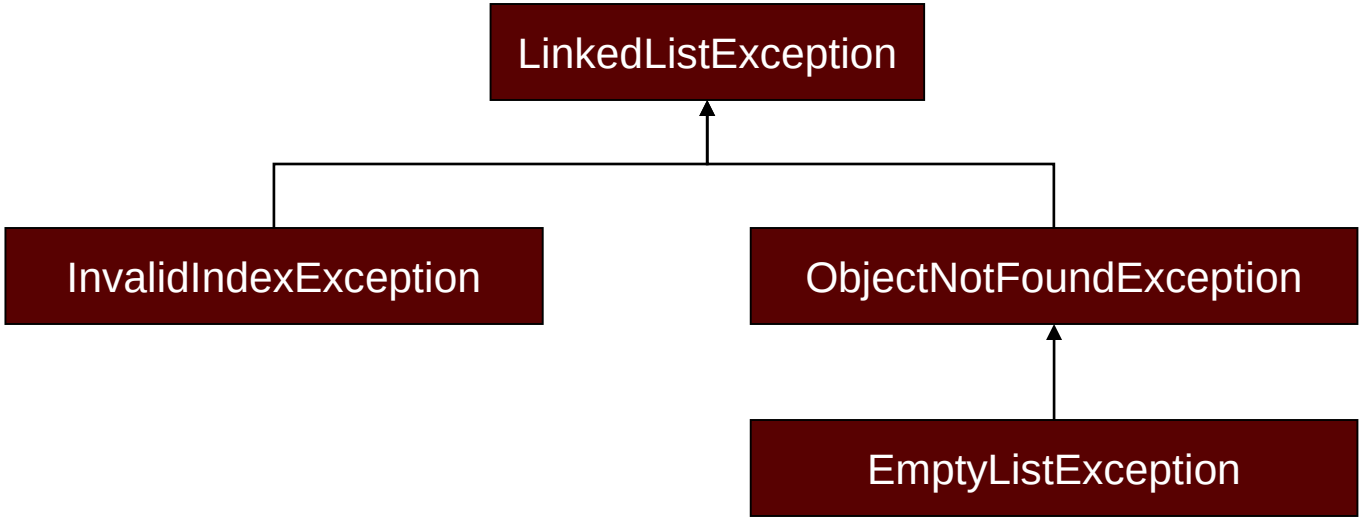
8.11. DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO

DEVEM DEFINIR-SE NOVOS TIPOS DE EXCEÇÃO SE:

- **O TIPO DE EXCEÇÃO QUE QUEREMOS UTILIZAR NÃO EXISTE NA API DO JAVA**
- **A DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO TORNA A APRENDIZAGEM DA NOVA CLASSE MAIS FÁCIL**
- **A CLASSE LANÇA VÁRIOS TIPOS DE EXCEÇÃO RELACIONADOS**

8.11. DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO

SOLUÇÃO POSSÍVEL:



8.11. DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO

QUE TIPO DE CLASSE BASE DEVEMOS ESCOLHER?

- TODAS AS EXCEÇÕES DEVEM SER, EM ÚLTIMA INSTÂNCIA, OBJETOS DO TIPO `Throwable`
- NO ENTANTO, NÃO É CORRETO ESCOLHER COMO CLASSE BASE A CLASSE `Throwable`
- A CLASSE `Error` DEVE SER UTILIZADA APENAS PARA ERROS RELACIONADOS COM O SISTEMA OPERATIVO OU A JVM

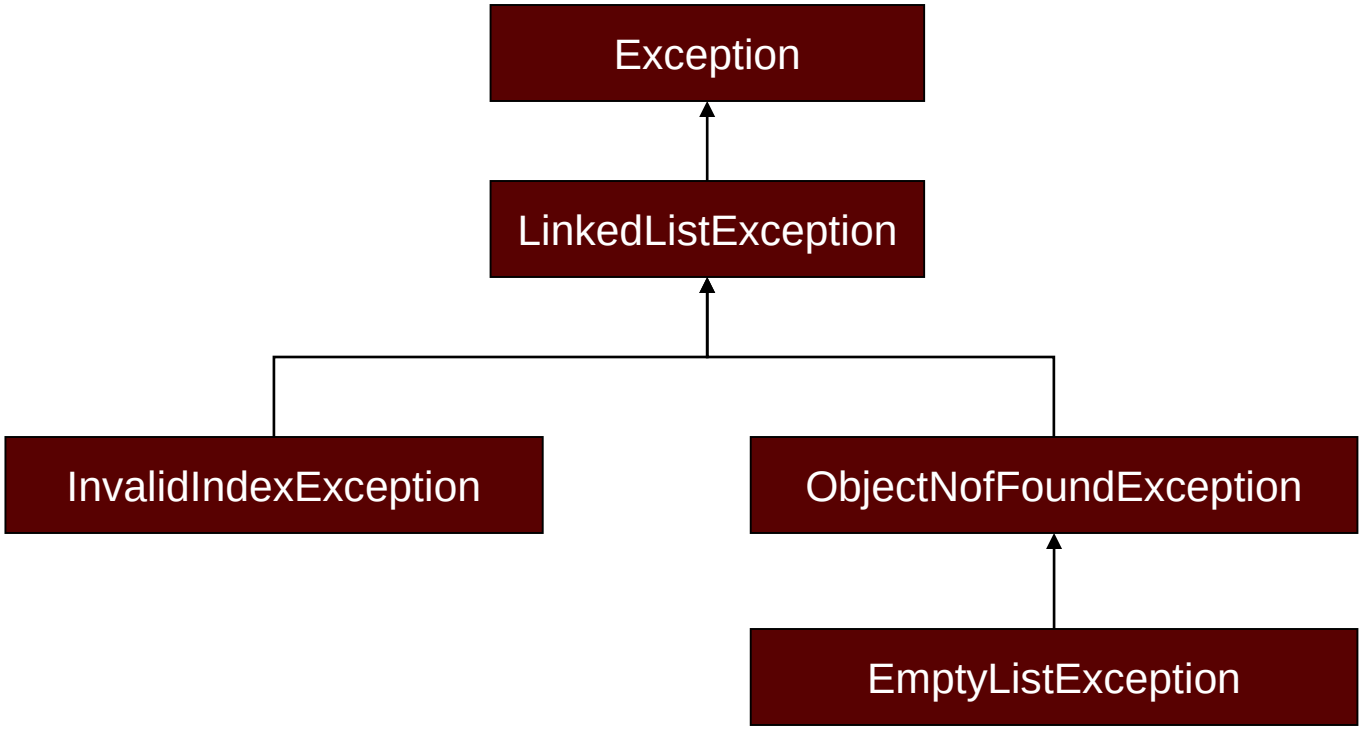
8.11. DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO

PARA UM NOVO TIPO DE EXCEÇÃO DEVE ESCOLHER-SE COMO CLASSE BASE UMA DAS SEGUINTES:

- **`Exception`** - **SE A EXCEÇÃO DEVE SER VERIFICADA**
- **`RuntimeException`** - **SE NÃO É IMPORTANTE OBRIGAR A VERIFICAR A EXCEÇÃO (EX: OCORRE DE VEZ EM QUANDO)**

8.11. DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO

SOLUÇÃO COMPLETA:



8.11. DEFINIÇÃO DE NOVOS TIPOS DE EXCEÇÃO

QUANDO É QUE SE DEVE ESCOLHER, COMO CLASSE BASE, A CLASSE `RuntimeException`?

- **QUANDO O TIPO DE ERRO ESTÁ RELACIONADO COM A JVM E NÃO COM A LÓGICA DA OPERAÇÃO (REFERÊNCIAS NULAS, ÍNDICES INVÁLIDOS, SEGURANÇA, ETC)**

STREAMS

8.12. CONCEITO DE STREAM

O conceito de stream é, muitas vezes, confundido incorretamente com o conceito de ficheiro

A comunicação com o sistema de entradas e saídas do java – ficheiros, a consola, ligações de rede, memória, entre processos, etc – é feita utilizando as bibliotecas `io` (input/output) e `nio` (new io)

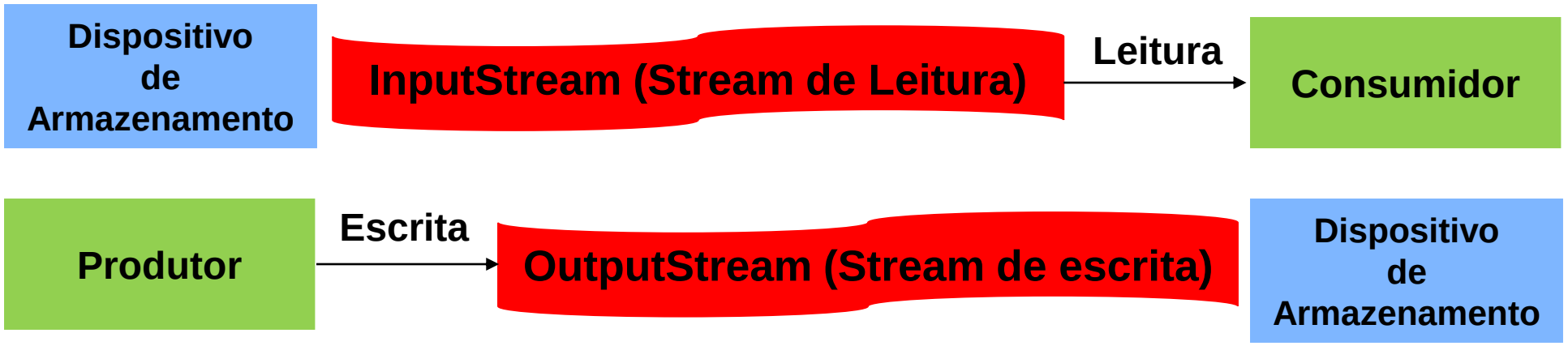
De uma forma geral, as streams podem ser agrupadas em duas categorias:

- Streams de texto
- Streams binárias

8.12. CONCEITO DE STREAM

Conceito que representa o fluxo de informação entre um dispositivo de armazenamento e um processo produtor/consumidor

- **dispositivo de armazenamento pode estar: num ficheiro em disco, em memória ou algures na rede**



8.13. A CLASSE File

Um objeto do tipo `File` representa um ficheiro no disco e permite realizar as seguintes operações:

- ver as propriedades do ficheiro (tamanho, data de criação, permissões de escrita/leitura)
- alterar alguns dos atributos de um ficheiro (`lastModified`, `readOnly`, ...)
- verificar se o ficheiro existe
- obter informações mais detalhadas sobre o caminho (*path*) do ficheiro
- criar um ficheiro
- apagar um ficheiro
- renomear um ficheiro

8.13. A CLASSE File

Um objeto do tipo `File` não permite:

- ler de um ficheiro
- escrever para um ficheiro

Um objeto do tipo `File` também pode representar uma diretoria no disco e permite:

- criar/apagar uma diretoria
- obter o conteúdo de uma diretoria

A classe `File` possui um conjunto de métodos estáticos que permitem:

- criar ficheiros temporários
- obter a lista de unidades do sistema de ficheiros

8.13. A CLASSE File

EXEMPLO

```
File idFicheiro = new File("C:\\Users\\nome\\workspace\\projetoAtual");

if (idFicheiro.exists()) {
    if (idFicheiro.isDirectory()) {
        for (File ficheiro : idFicheiro.listFiles()) {
            System.out.println("Nome: " + ficheiro.getName() +
                               " tamanho: " + ficheiro.length() +
                               " data: " + ficheiro.lastModified());
        }
    }
}

// Apenas se cria uma representação de um ficheiro ou uma diretoria
// Não se cria nada em disco
File novaDiretoria = new File("mytemp");

if (!novaDiretoria.exists()) {
    novaDiretoria.mkdir();
}

if (novaDiretoria.exists()) {
    novaDiretoria.delete();
}
```

8.14. A INTERFACE FilenameFilter

PERMITE DEFINIR FILTROS PARA OS SEGUINTES MÉTODOS DA CLASSE `File`

```
String[] list(FilenameFilter f);
File[] listFiles(FilenameFilter f);
```

```
public class Filtro implements FilenameFilter {
    private String extensao;

    public Filtro(String extensao) {
        this.extensao = extensao;
    }

    public boolean accept(File dir, String name) {
        return name.endsWith(extensao);
    }
}
```

```
public class FilterDemo {
    public static void main(String[] args) {
        File idFicheiro = new File("C:\\Users\\nome\\workspace\\projetoAtual");

        System.out.println("Classes:");
        String[] lista = idFicheiro.list(new Filtro(".class"));
        for (String nome : lista) {
            System.out.println(nome);
        }
        System.out.println("\nCódigo fonte:");
        lista = idFicheiro.list(new Filtro(".java"));
        for (String nome : lista) {
            System.out.println(nome);
        }
    }
}
```

8.15. CLASSIFICAÇÃO DE STREAMS

Tipo de dados que manipulam:

- **Streams de texto (*character streams*) - manipulam carateres (2 bytes)**
- **Streams binárias - manipulam bytes**

Tipo de operações que efetuam:

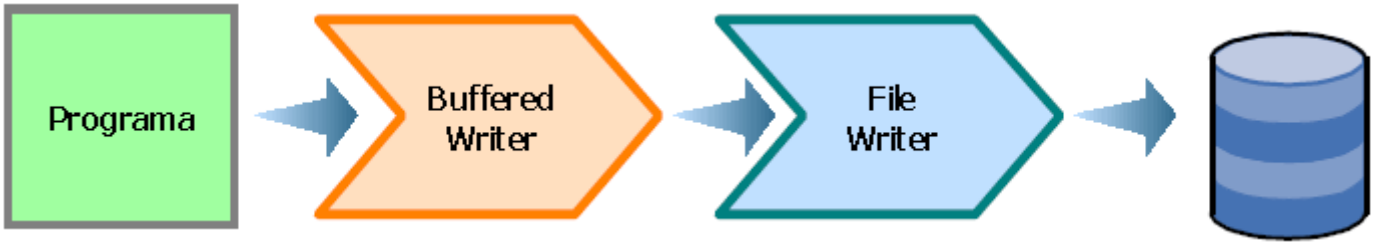
- **Streams simples - streams de baixo nível que não alteram os dados**
- **Streams processadas - streams de alto nível que efetuam um processamento adicional aos dados**

8.15. CLASSIFICAÇÃO DE STREAMS

As streams simples são consideradas de baixo nível visto que não modificam nem processam os dados

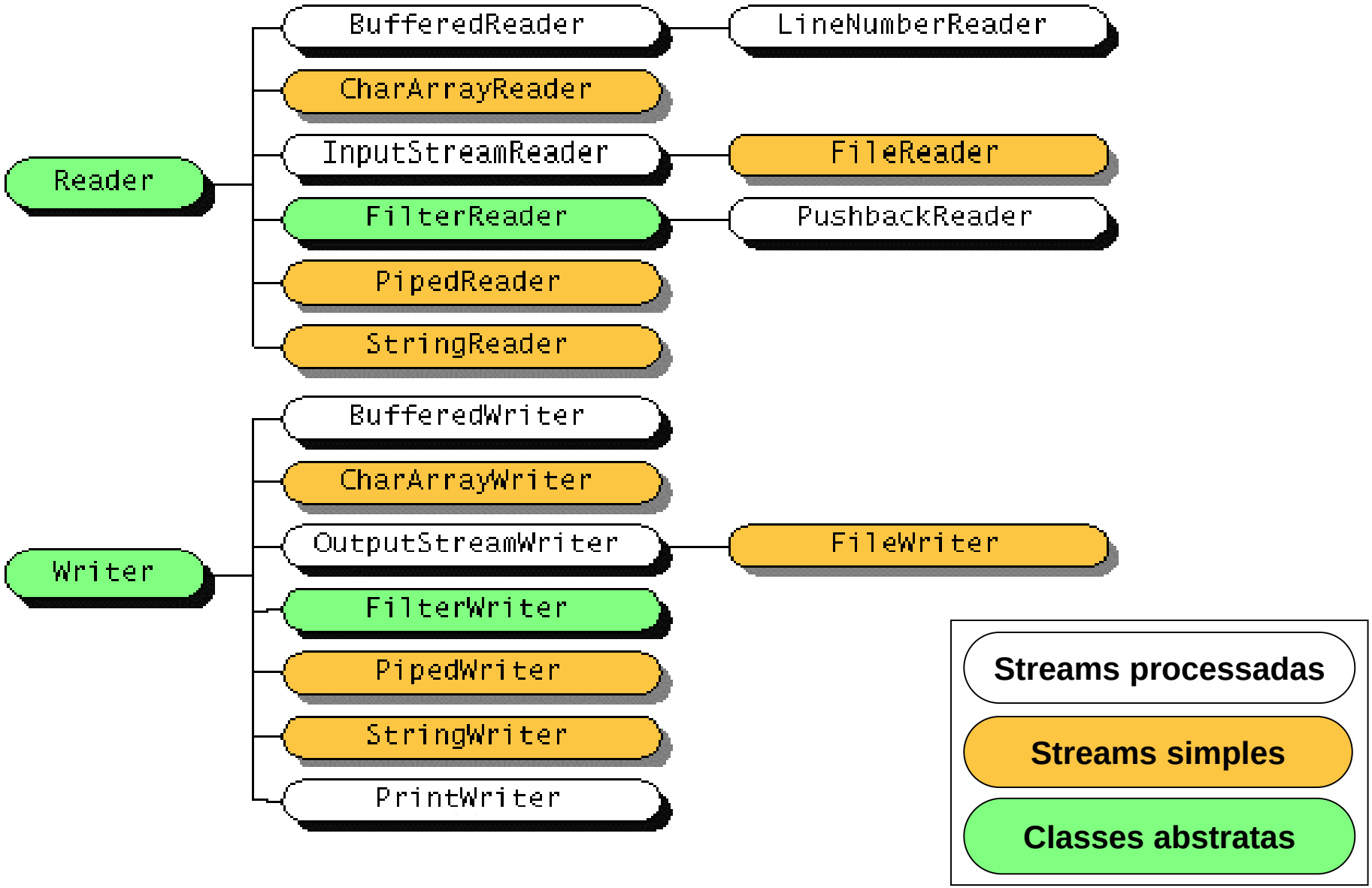
As streams processadas tratam os dados e disponibilizam métodos para obter os dados tratados

Exemplo usual: recurso a um *buffer* para ler/escrever de/para ficheiro



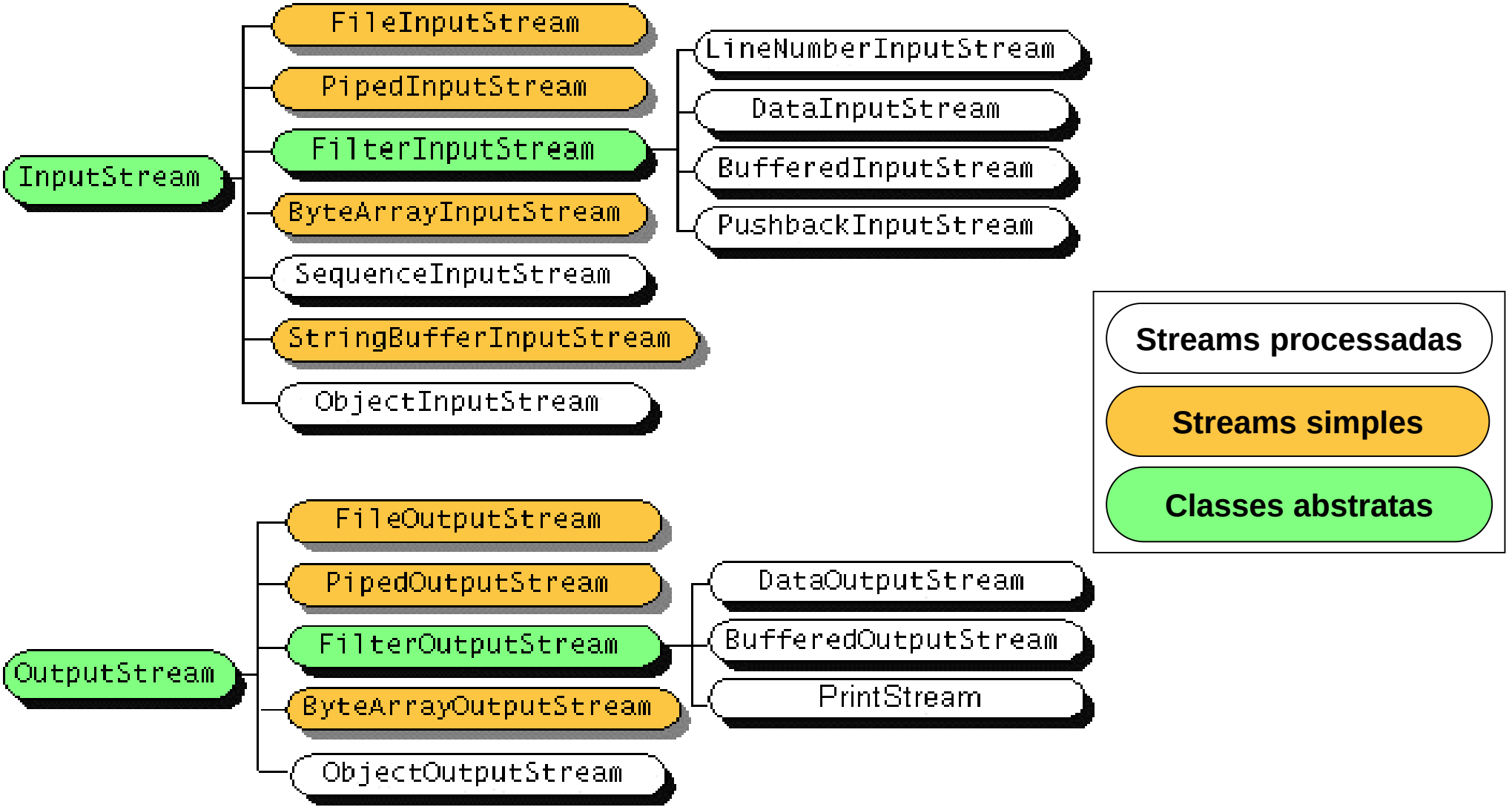
8.15. CLASSIFICAÇÃO DE STREAMS

STREAMS DE TEXTO



8.15. CLASSIFICAÇÃO DE STREAMS

STREAMS BINÁRIAS



8.15. CLASSIFICAÇÃO DE STREAMS

STREAMS SIMPLES

Tipo de Armazenamento	Streams de texto	Streams binárias
Memória	`CharArrayReader` `CharArrayWriter`	`ByteArrayInputStream` `ByteArrayOutputStream`
	`StringReader` `StringWriter`	`StringBufferInputStream`
Pipe	`PipedReader` `PipedWriter`	`PipedInputStream` `PipedOutputStream`
Ficheiro	`FileReader` `FileWriter`	`FileInputStream` `FileOutputStream`

8.15. CLASSIFICAÇÃO DE STREAMS

STREAMS PROCESSADAS

Processo	Streams de texto	Streams binárias
Utilização de um buffer	**BufferedReader** **BufferedWriter**	**BufferedInputStream** **BufferedOutputStream**
Contabiliza o número de linhas	**LineNumberReader**	**LineNumberInputStream**
Leitura antecipada	**PushbackReader**	**PushbackInputStream**
Escrita formatada	**PrintWriter**	**PrintStream**
Concatenação de streams	-	**SequenceInputStream**
Conversão entre streams binárias e streams de texto	**InputStreamReader** **OutputStreamWriter**	-
Conversão de dados	-	**DataInputStream** **DataOutputStream**
Armazenamento de objectos	-	**ObjectInputStream** **ObjectOutputStream**

8.16. HIERARQUIA DE STREAMS

As superclasses `Reader`, `Writer`, `InputStream` e `OutputStream` são classes abstratas

- As classes `Reader` e `InputStream` representam as streams de leitura
- As classes `Writer` e `OutputStream` representam as streams de escrita

As streams são automaticamente abertas quando são criadas

Todas as streams podem/devem ser fechadas utilizando o método `close()`

8.17. SERIALIZAÇÃO

A abordagem mais simples para escrever um objeto num ficheiro seria escrever cada um dos seus atributos

Para ler esse objeto, teríamos de ler cada atributo e (re)construir o objeto

O objeto recuperado será o mesmo que foi escrito?

E quando o objeto a gravar tiver uma lista de outros objetos?

8.17. SERIALIZAÇÃO

A serialização é um mecanismo que permite ler/escrever objetos

Por omissão os objetos não são serializáveis

Assim, de modo a indicar que os objetos de uma classe são serializáveis, esta deve implementar a interface `Serializable` que funciona como simples marcador

8.17. SERIALIZAÇÃO

Para ler/escrever objetos serializáveis utilizam-se as streams binárias:

- `ObjectInputStream`
- `ObjectOutputStream`

Os seguintes atributos não são processados:

- atributos da classe (marcados com `static`)
- atributos marcados com a palavra `transient`

8.17. SERIALIZAÇÃO

Classe	Funcionalidade
ObjectInputStream	**Classe utilizada para efetuar a leitura de um objeto (e respetivo estado) a partir de uma stream . Esta classe implementa a interface `ObjectInput` que define, entre outros, o seguinte método:** `Object readObject() throws IOException, ClassNotFoundException` **As exceções anteriores têm o seguinte significado:** **`IOException` – erro de leitura** **`ClassNotFoundException` – não encontra a classe que implementa o objeto que vai ler**
ObjectOutputStream	**Classe utilizada para escrever um objeto (e respetivo estado) numa stream. Esta classe implementa a interface `ObjectOuput` que define, entre outros, o seguinte método:** `void writeObject(Object o) throws IOException` **As excepções devolvidas por esta classe (incluindo a mencionado no método) são:** **`IOException` – erro de escrita** **`NotSerializableException` – este objeto não pode ser armazenado**

8.18. ENTRADA/SAÍDA PADRÃO

A CLASSE `System` CONTÉM ATRIBUTOS QUE PERMITEM REPRESENTAR AS STREAMS PADRÃO:

ENTRADA: `System.in` (OBJETO DO TIPO `InputStream`)

SAÍDA: `System.out` (OBJETO DO TIPO `PrintStream`)

SAÍDA DE ERRO: `System.err` (OBJETO DO TIPO `PrintStream`)

8.19. OUTRAS CLASSES

Classe	Funcionalidade
`StreamTokenizer`	Permite dividir um ficheiro de texto em palavras (*tokens*) de acordo com um conjunto de regras
`CheckedInputStream`	Permite verificar a integridade de uma stream de leitura através do cálculo de um valor de verificação (*checksum*)
`CheckedOutputStream`	Permite garantir a integridade de uma stream de escrita através do cálculo de um valor de verificação (*checksum*)
`InflaterInputStream`	Classe base para as classes de descompressão (**`ZipInputStream`** e **`GZIPInputStream`**)
`DeflaterOutputStream`	Classe base para as classes de compressão (**`ZipOutputStream`** e **`GZIPOutputStream`**)
`ZipOutputStream`/ `ZipInputStream`	Classe que permite a compressão/descompressão de dados utilizando o formato Zip
`GZIPOutputStream`/ `GZIPInputStream`	Classe que permite a compressão/descompressão de dados utilizando o formato GZIP (extensão .gz)