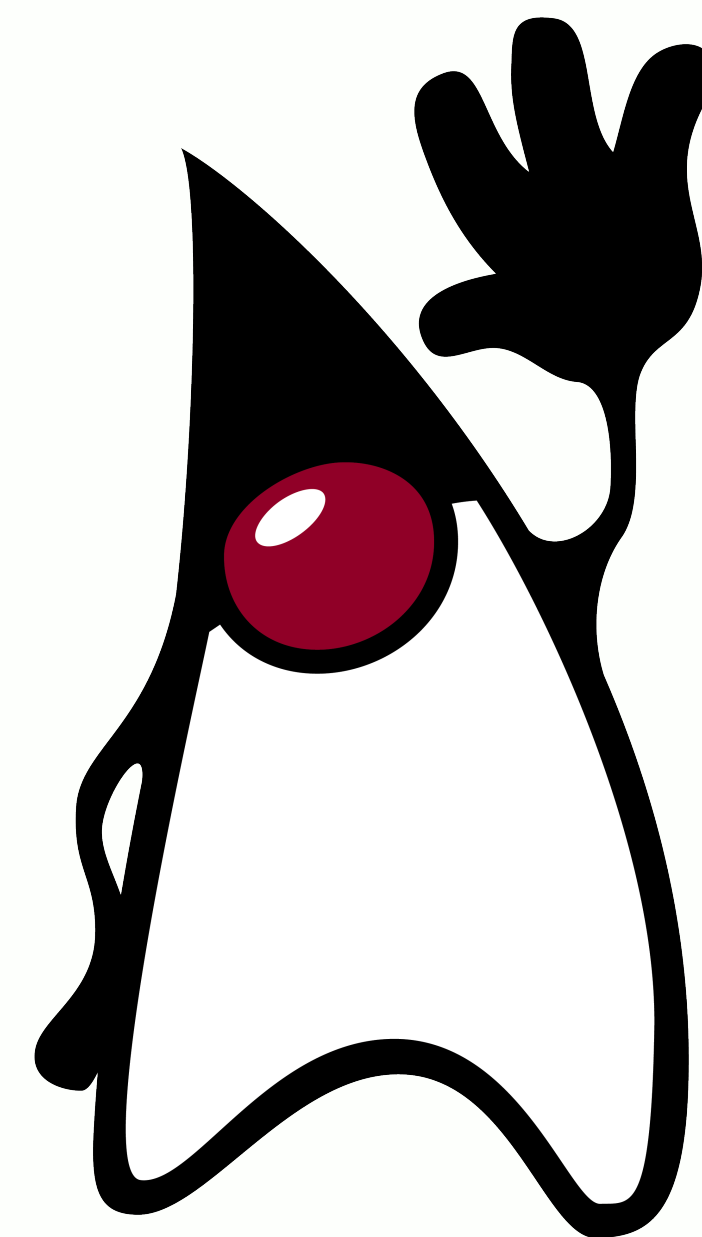




UNIVERSIDADE
FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Tratamento de Exceções

QXD0007 - Programação Orientada a Objetos



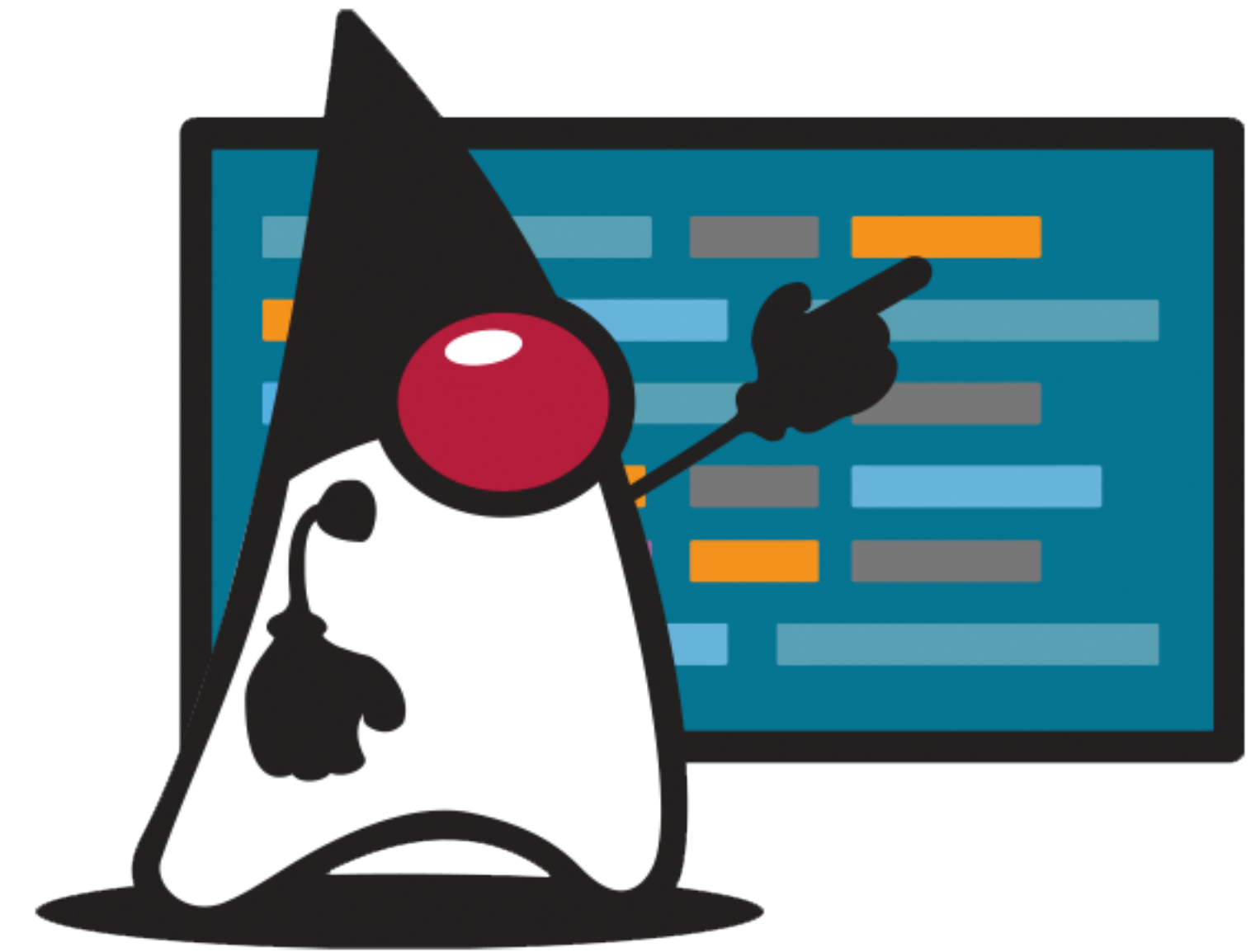
Prof. Bruno Góis Mateus (brunomateus@ufc.br)

Conteúdo

- Introdução
- Lidando com erros
- Error
- Runtime Exceptions
- Checked Exceptions
- Tratando exceções
- Lançando exceções



Introdução



Introdução

“Program testing can be used to show the presence of bugs, but never to show their absence!”

— Edsger W. Dijkstra

— Edsger W. Dijkstra

Introdução

- Assumindo que seu código tem a habilidade de detectar uma condição de erro, é possível tratá-lo de diversas maneiras:
 - Ignorar o problema - **Má ideia**
 - Verificar potenciais problemas e abortar o programa caso algo seja detectado
 - Verificar potenciais problemas, capturá-los os erros, e tentar corrigir o problema
 - **Lançar e tratar uma exceção**

Introdução

- Ignorar o problema

```
const input = 'X100';  
const num = parseInt(input);  
  
console.log(num); // NaN
```

- A princípio uma aplicação nunca deve quebrar
- Logo, simplesmente ignorar os erros é a receita do desastre
- Quando o erro pode acontecer e sua aplicação irá terminar de maneira inesperada
- A aplicação por continuar executando porém realizando operações erroneamente
- Do que adianta detectar os erros se você vai ignorá-los?

Introdução

- Verificar os problemas e abortar a aplicação
 - A aplicação pode mostrar uma mensagem indicando que algo está errado
 - Não terminaria de forma inesperada
 - O usuário pode ficar se perguntando que deu errado
- Do ponto de vista do sistema operacional está opção é bem melhor do que ignorar o problema

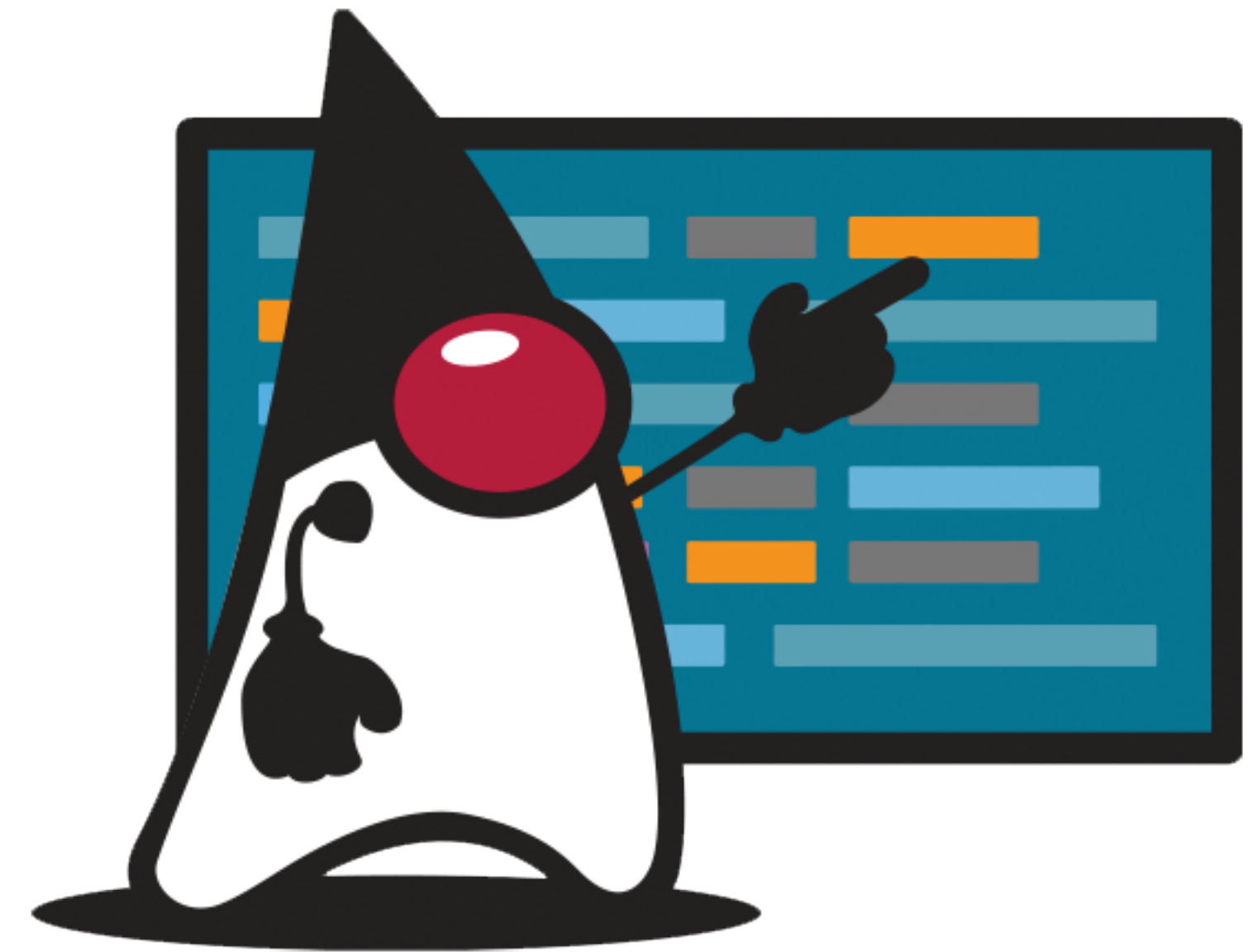
```
const input = 'X100';  
const num = parseInt(input);  
  
if (isNaN(num)) {  
    process.exit(0)  
}  
  
console.log(num);
```


Introdução

- Verificar os problemas, identificá-los e tentar corrigi-los
 - Solução bem superior em relação a identificar o problema e abortar a execução
 - O problema é detectado em código e a própria aplicação tenta corrigi-lo
- Funciona bem em algumas situações

```
const input = 'X100';  
const num = parseInt(input);  
  
if (isNaN(num)) {  
    num = 0;  
}  
  
console.log(num); // 0
```

Lidando com Erros



Lidando com Erros

Exemplo prático

```
public int transferir(Conta destino, double valor) {  
    if (!isPositive(valor)) {  
        System.out.println("Valor de inválido");  
        return -1;  
    }  
  
    if (sacar(valor)) {  
        destino.depositar(valor);  
        System.out.println("Operacao realizada com sucesso");  
        return 0;  
    } else {  
        System.out.println("Saldo insuficiente");  
        return -2;  
    }  
}
```

Valor negativo

Valor inválido

- É muito que quem saiba tratar o erro é aquele que chamou o método e não o próprio método
- Como avisar ao método que chamou que algo aconteceu?

Não é obrigatório testar o retorno de um método

Tratando o retorno de um método

Lidando com Erros

- Códigos de Erro
 - Confusão com o domínio do retorno
 - Interpretação do retorno pelo código chamador
 - Mistura entre código funcional vs tratamento de erro
- Mensagens de Erro
 - São úteis apenas para seres humanos
 - O código chamador não consegue "entender"

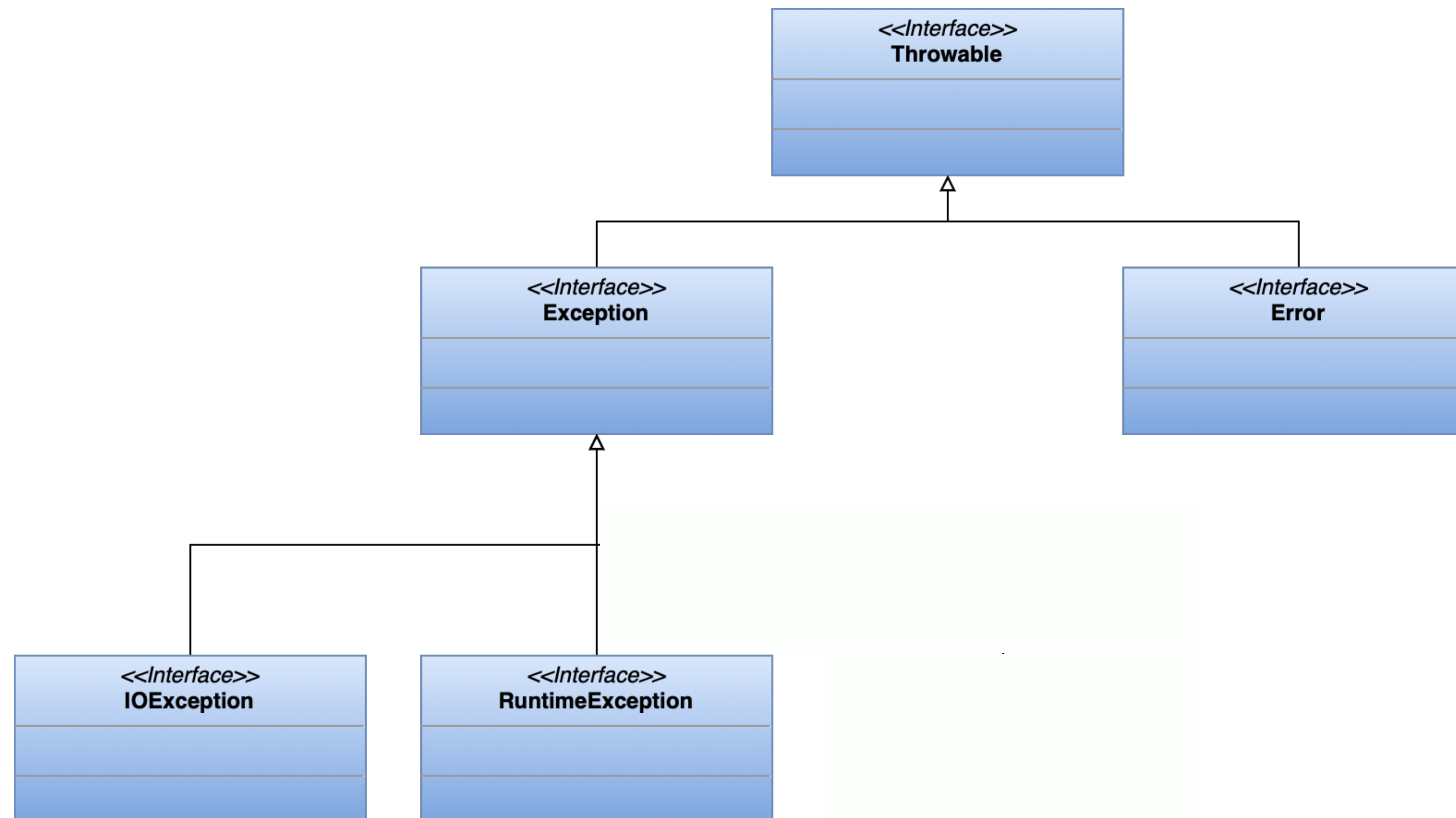
Lidando com Erros

- Ao invés de códigos e mensagens de erro, teremos exceções

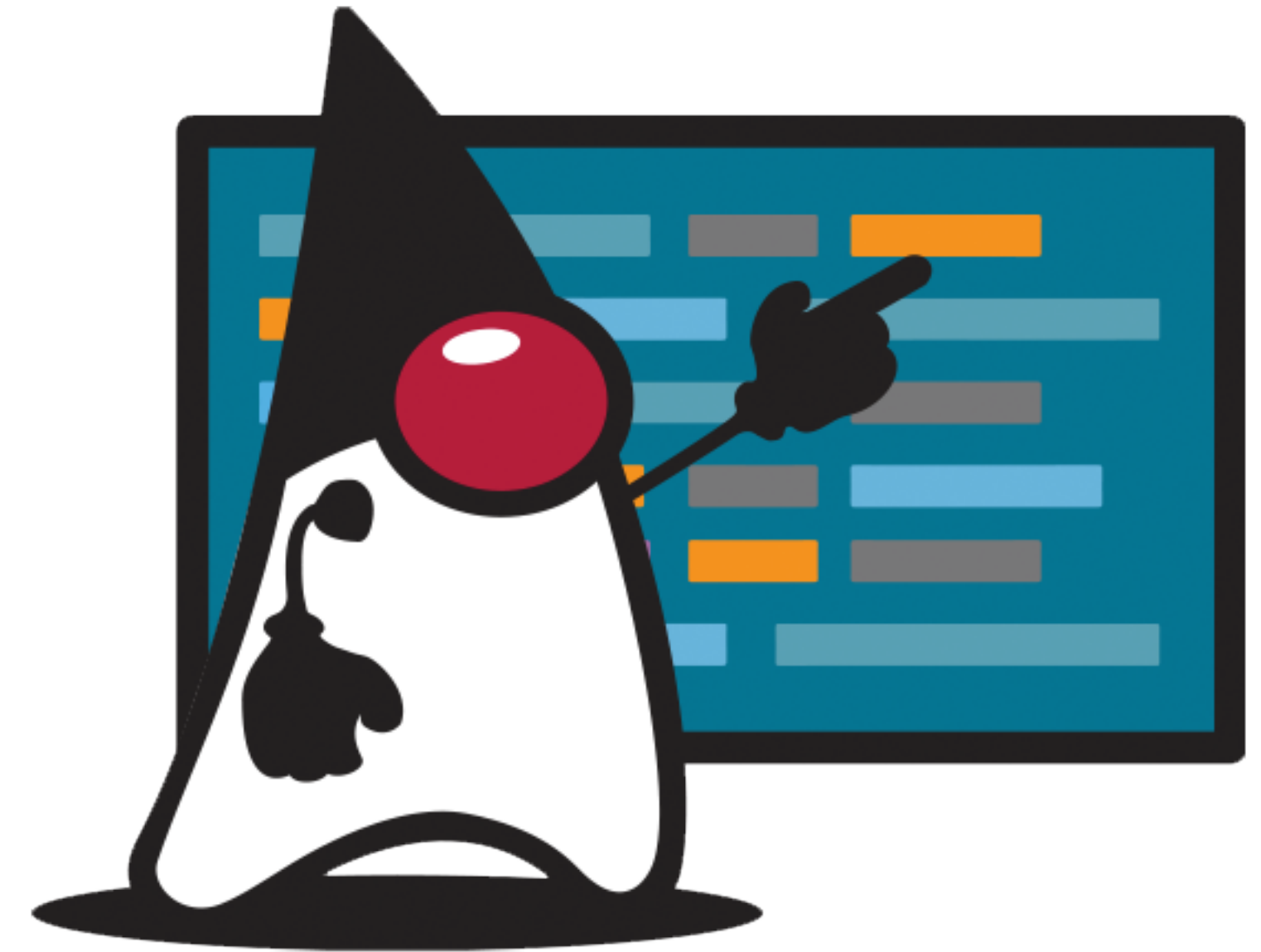
“Uma exceção é um evento que ocorre durante a execução de um programa e que interrompe seu fluxo normal de execução.”

execução.”

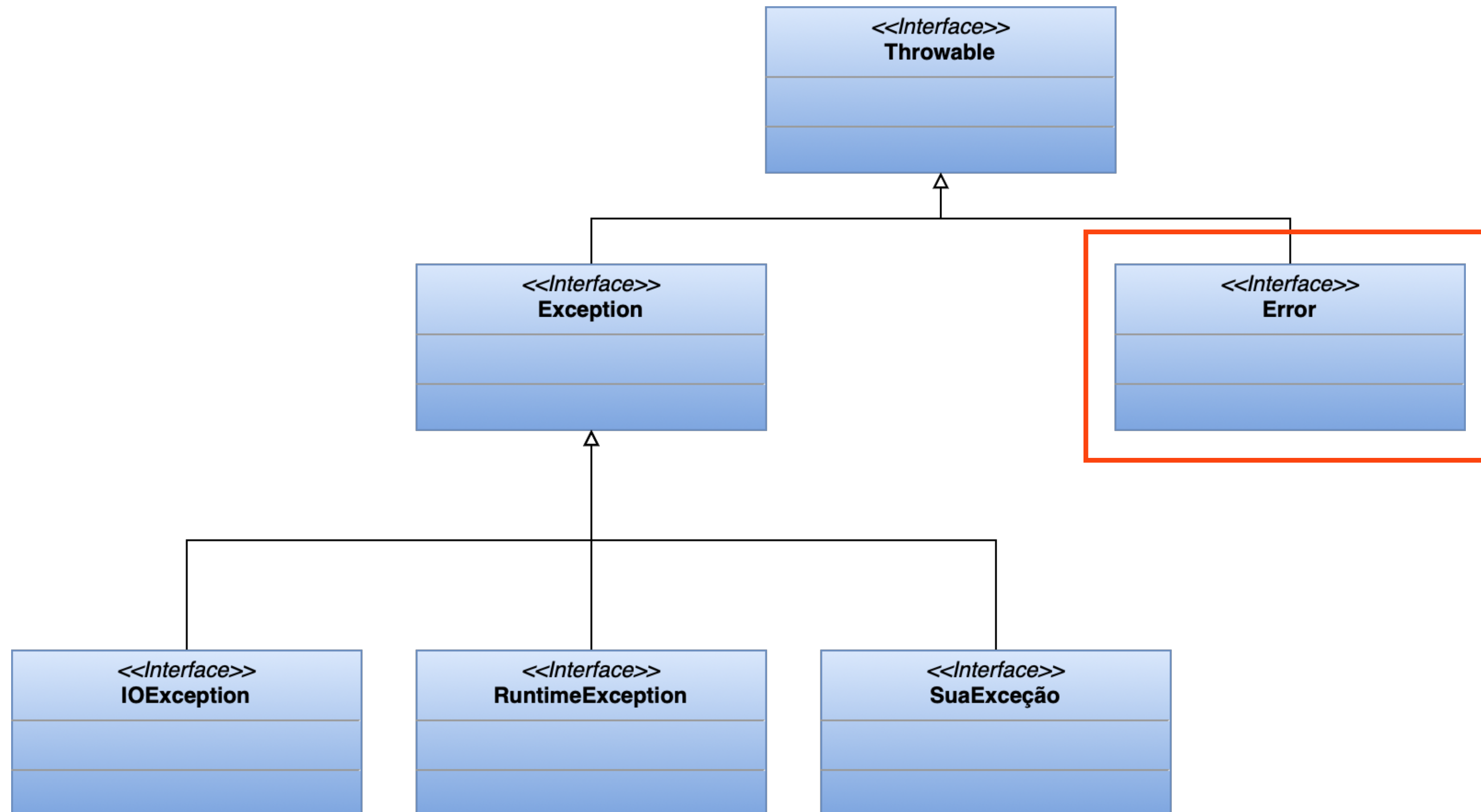
Lidando com Erros



Error



Error

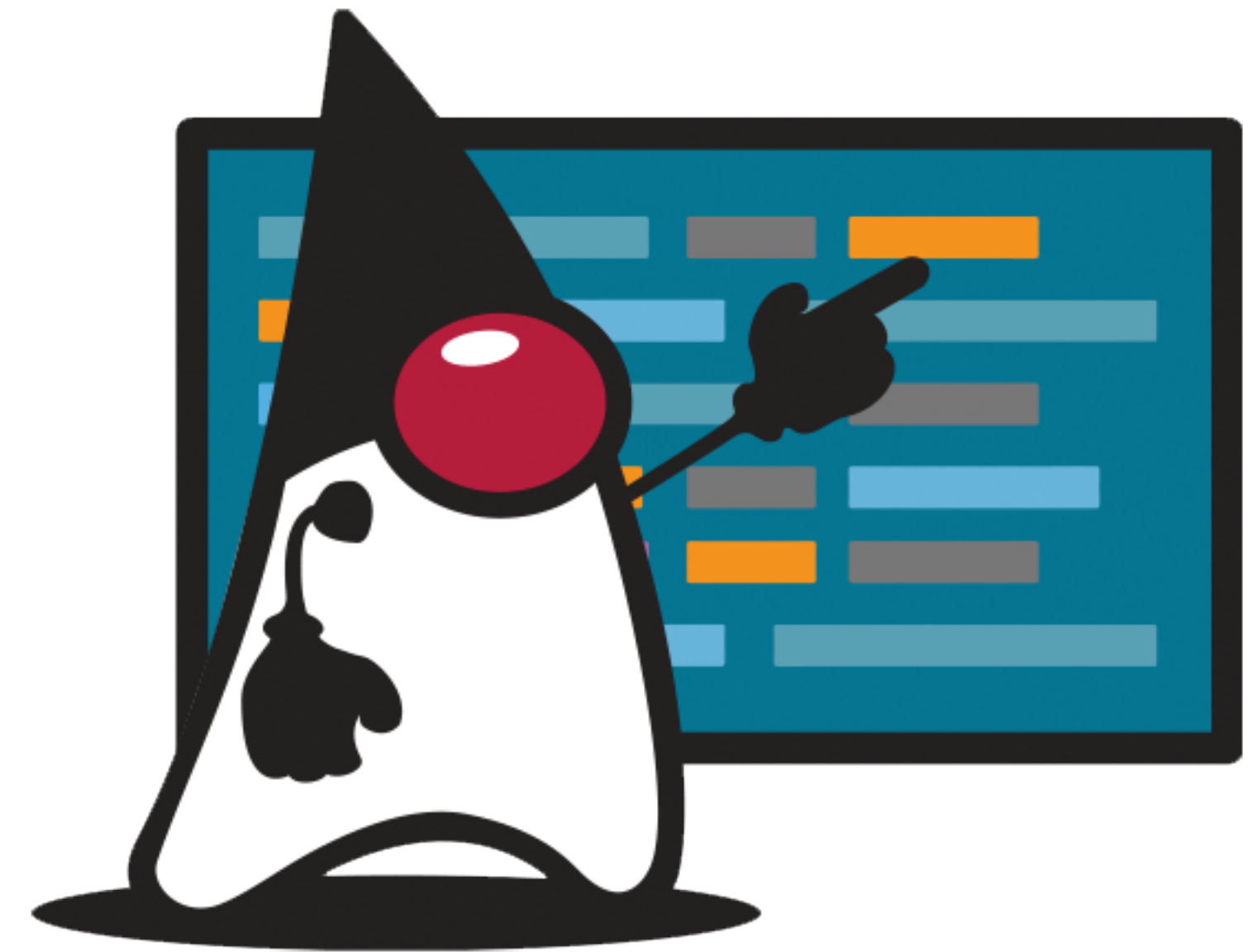


Error

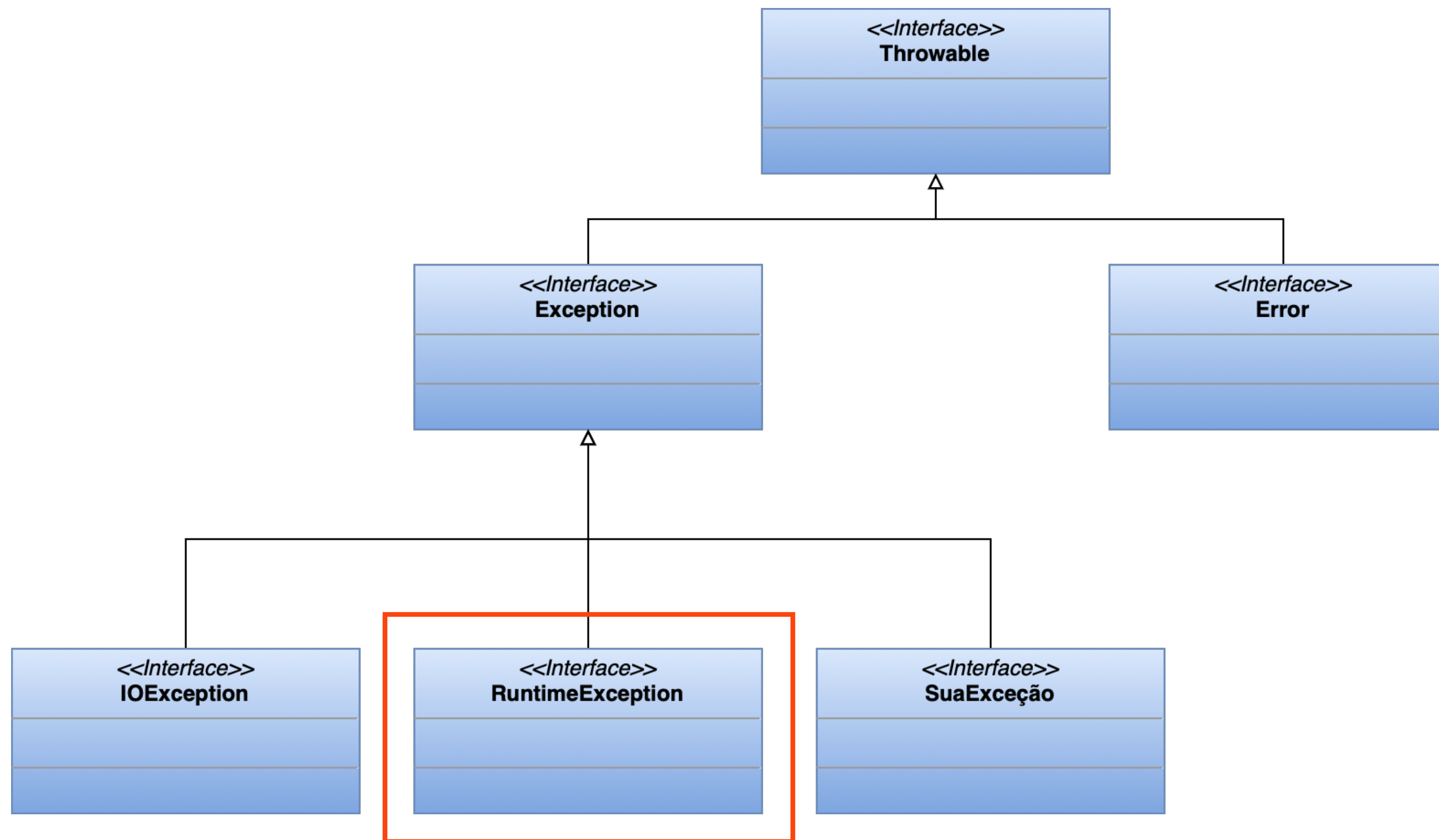
Error

- São exceções que indicam que algo sério aconteceu
- Aplicação não deve tentar tratar
- Ex: `OutOfMemoryError` e `StackOverflowError`

Runtime Exceptions



Runtime Exceptions



Runtime Exceptions

- O que acontece se executarmos o código abaixo ?

```
public class TestandoADivisao {  
    public static void main(String args[]) {  
        int i = 5571;  
        i = i / 0;  
        System.out.println("O resultado " + i);  
    }  
}
```

- Será lançada uma **ArithmeticException**, na linha responsável por realizar a divisão

Runtime Exceptions

- O que acontece se executarmos o código abaixo ?

```
public class Example {  
    public void doSomething() {  
        Integer number = null;  
  
        if (number > 0) {  
            System.out.println("Positive number");  
        }  
    }  
}
```

- Será lançada uma **NullPointerException** quando programa tentar acessar o objeto que contém **null**

Runtime Exceptions

- O que acontece se executarmos o código abaixo ?

```
public class Example {  
    public void processArray() {  
        List names = new ArrayList<>();  
        names.add("Eric");  
        names.add("Sydney");  
  
        return names.get(5);  
    }  
}
```

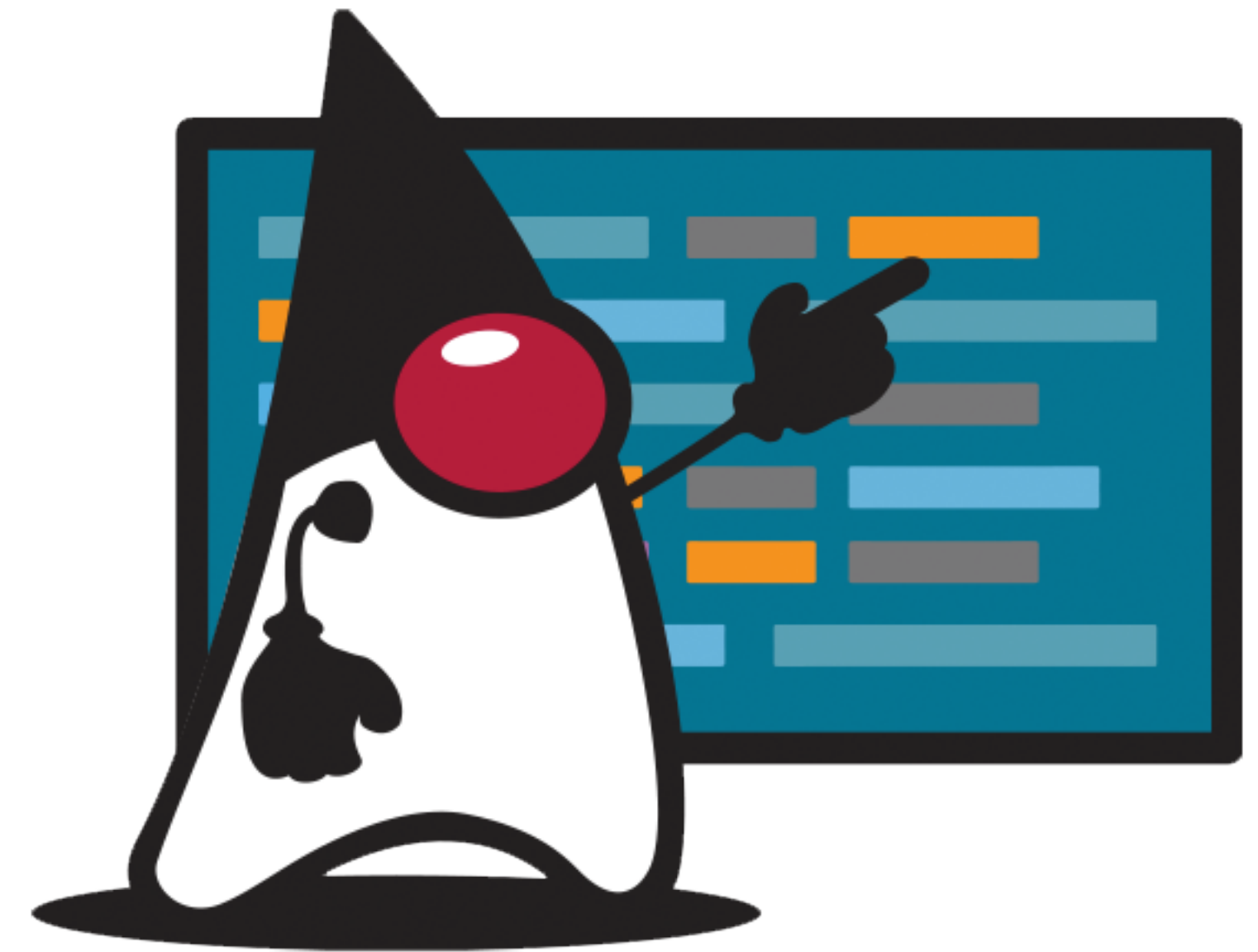
- Será lançada uma **ArrayIndexOutOfBoundsException** quando programa tentar acessar uma posição que não existe na lista

Runtime Exceptions

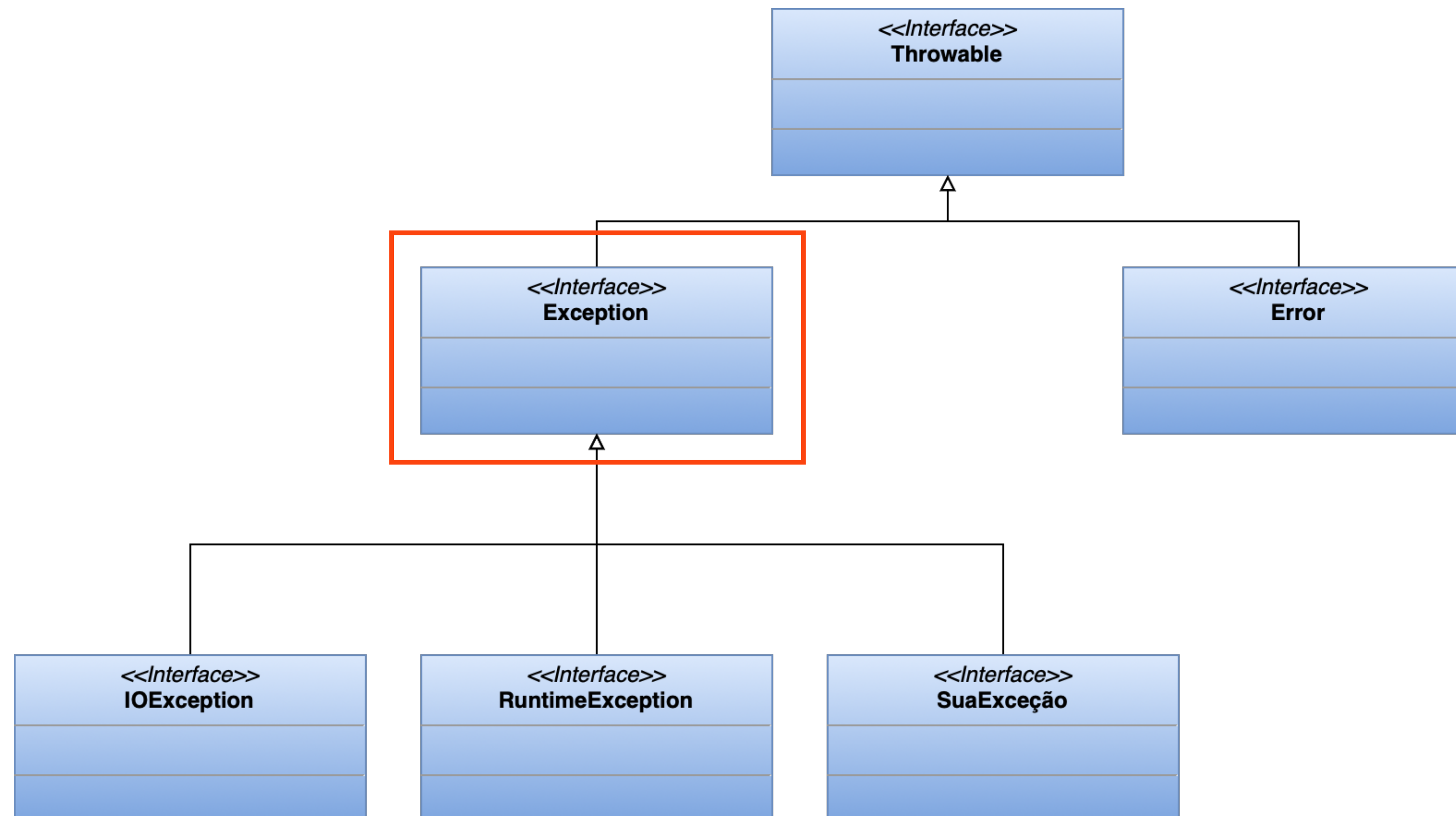
Runtime Exceptions

- Nos casos anteriores os problemas poderiam ser facilmente contornados com adição um if
- Por esse motivo o Java não obriga tratar esses tipos de exceções
- **Unchecked exceptions** ou exceções não checadas
- O compilador não verifica se você está tratando essas exceções

Checked Exceptions



Checked Exceptions



Checked Exceptions


Checked Exceptions

- Obrigam quem chama o método ou construtor tratar a exceção
- O compilador realizar uma verificação para saber se a exceção está sendo tratada
- De acordo com Oracle, devem utilizadas quando o método chamador tem chances de se recuperar
- Exceções mais comuns:
 - IOException e FileNotFoundException

Checked Exceptions

```
public int getPlayerScore(String playerFile){  
    Scanner contents = new Scanner(new File(playerFile));  
    return Integer.parseInt(contents.nextLine());  
}
```

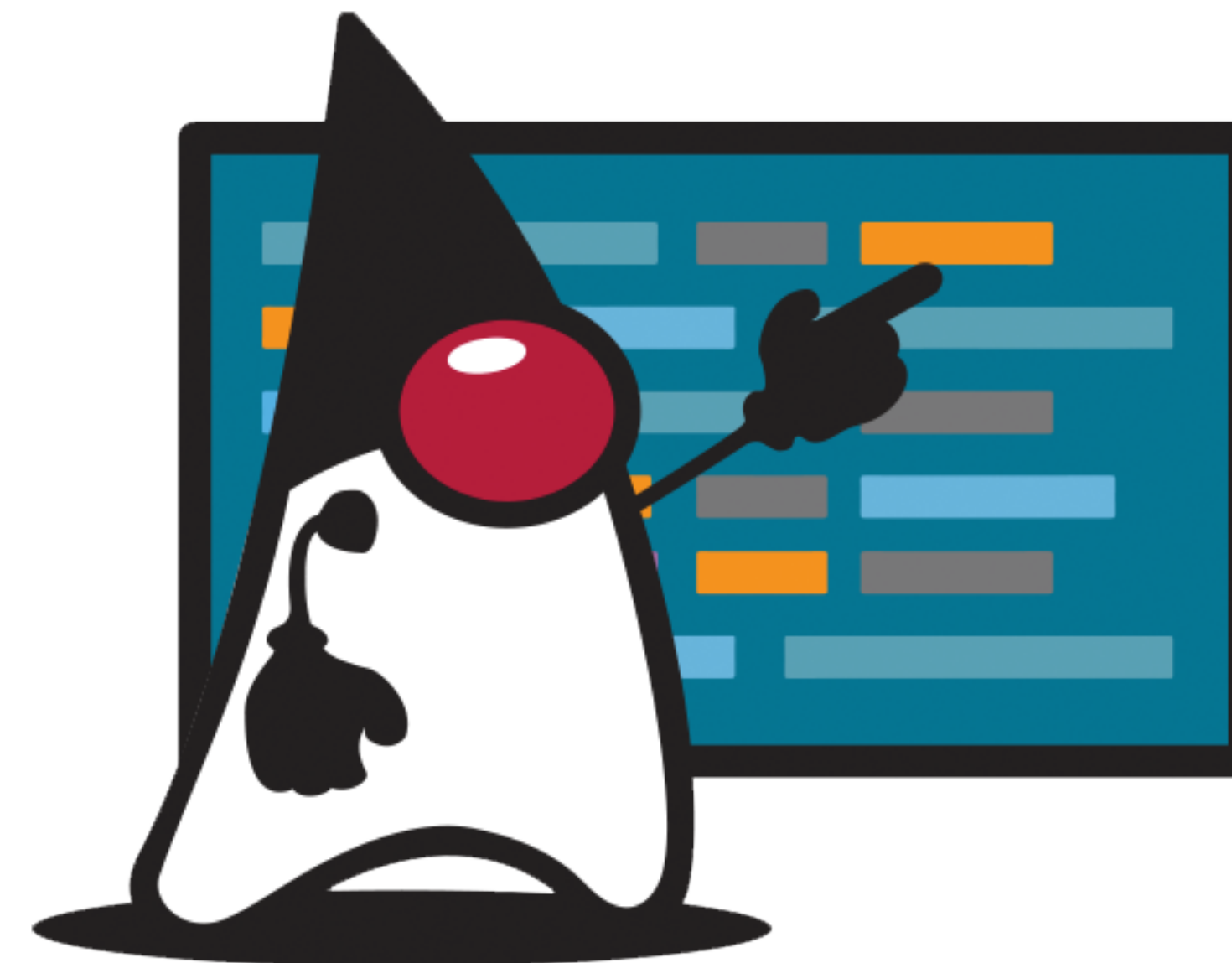
`FileNotFoundException`



```
public int getPlayerScore(String playerFile) throws FileNotFoundException {  
    Scanner contents = new Scanner(new File(playerFile));  
    return Integer.parseInt(contents.nextLine());  
}
```

```
public int getPlayerScore(String playerFile) {  
    try {  
        Scanner contents = new Scanner(new File(playerFile));  
        return Integer.parseInt(contents.nextLine());  
    } catch (FileNotFoundException noFile) {  
        logger.warn("File not found, resetting score.");  
    }  
}
```

Tratando exceções



Tratando Exceções

- Evita que nossos programas quebrem devido a exceções

```
try {  
    // attempt to execute this code  
} catch (exception e) {  
    // this code handles exceptions  
} finally {  
    // this code always gets executed  
}
```

- try
 - Obrigatória
 - Usada para delimitar o bloco de código que pode gerar a exceção

Tratando Exceções

```
try {  
    // attempt to execute this code  
} catch (exception e) {  
    // this code handles exceptions  
} finally {  
    // this code always gets executed  
}
```

- **catch**
 - O bloco interno **é executado caso a exceção ocorra**
 - A cláusula **interrompe a propagação do erro**
 - É possível acessar a exceção lançada

Tratando Exceções

```
try {  
    // attempt to execute this code  
} catch (exception e) {  
    // this code handles exceptions  
} finally {  
    // this code always gets executed  
}
```

- **finally**
 - É opcional
 - O seu bloco de código é **SEMPRE** executado
 - Independentemente da exceção ser lançada
 - Mesmo que o bloco **try** possua um **return**

```
boolean foo() {  
    try {  
        return true;  
    } finally {  
        return false;  
    }  
}
```

Tratando Exceções

Múltiplas exceções

- É possível que um trecho de código possa ocasionar mais de uma exceção
- Se ambas forem **checked**, precisaremos tratar ambas
- Nesse caso também teríamos duas opções
 - Adicionar um bloco **try/catch**
 - Delegar o tratamento usando **throws**
- Podemos ainda tratar alguma delas e delegar o tratamento de outra

Tratando Exceções

Múltiplas exceções

- Tratando exceções de maneiras distintas

```
try {  
    objeto.metodoQuePodeLancarIOeSQLException();  
} catch (IOException e) {  
    // ..  
} catch (SQLException e) {  
    // ..  
}
```

- **Multicatch:** a partir do **Java 7**

```
try {  
    objeto.metodoQuePodeLancarIOeSQLExceptioneTimeoutException();  
} catch (SQLException | IOException e) {  
    logger.log(e);  
} catch (TimeoutException e) {  
    logger.severe(e);  
}
```

Tratando Exceções

Try with resources

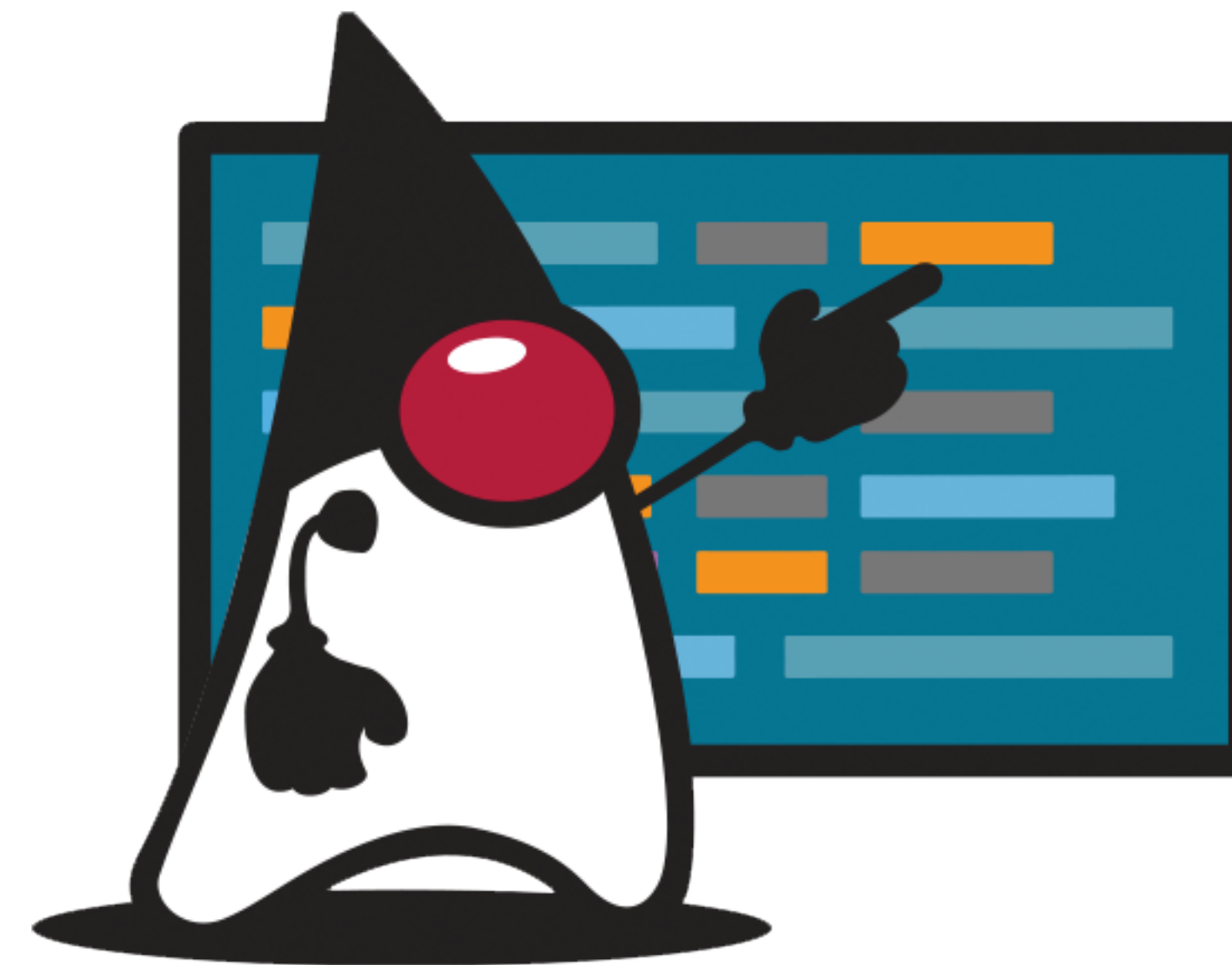
```
public int getPlayerScore(String playerFile) {  
    Scanner contents;  
    try {  
        contents = new Scanner(new File(playerFile));  
        return Integer.parseInt(contents.nextLine());  
    } catch (FileNotFoundException noFile) {  
        logger.warn("File not found, resetting score.");  
        return 0;  
    } finally {  
        try {  
            if (contents != null) {  
                contents.close();  
            }  
        } catch (IOException io) {  
            logger.error("Couldn't close the reader!", io);  
        }  
    }  
}
```

Tratando Exceções

Try with resources

```
public int getPlayerScore(String playerFile) {  
    try (Scanner contents = new Scanner(new File(playerFile))) {  
        return Integer.parseInt(contents.nextLine());  
    } catch (FileNotFoundException e) {  
        logger.warn("File not found, resetting score.");  
        return 0;  
    }  
}
```

Exceções e a pilha de execução



Exceções e a pilha de execução

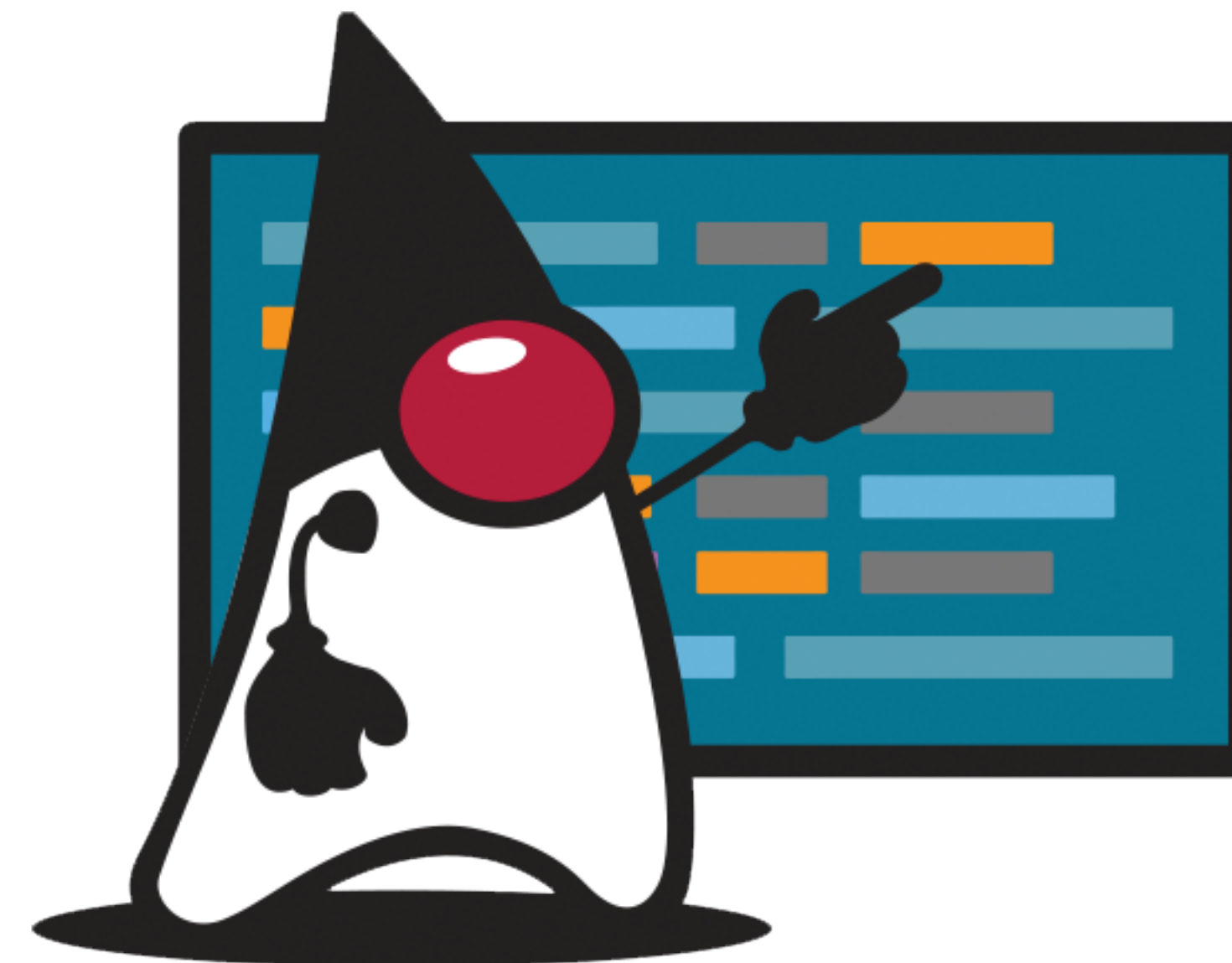
```
class TesteErro {
    public static void main(String[] args) {
        System.out.println("inicio do main");
        metodo1();
        System.out.println("fim do main");
    }

    static void metodo1() {
        System.out.println("inicio do metodo1");
        metodo2();
        System.out.println("fim do metodo1");
    }

    static void metodo2() {
        System.out.println("inicio do metodo2");
        int[] array = new int[10];
        for (int i = 0; i <= 15; i++) {
            array[i] = i;
            System.out.println(i);
        }
        System.out.println("fim do metodo2");
    }
}
```

- O que acontece se executarmos o código anterior ?
- E se colocarmos o laço dentro de um bloco **try** ?
- E se colocarmos apenas a instrução interna do laço dentro do bloco **try** ?
- E se colocarmos o bloco **try** em volta da chamada ao **método2**?
- E se colocarmos o bloco **try** em volta da chamada ao **método1**?

Lançando exceções



Lançando exceções

- É possível, **lançarmos exceções** no momento que acharmos mais adequado
- Para isso usamos a palavra reservada **throw**
 - **throw**, imperativo. Lança um **Exception**
 - **throws** presente do indicativo. **Avisa sobre a possibilidade de ocorrer um Exception**

Lançando exceções

```
public class Conta {  
    void sacar(double valor) {  
        if (this.saldo < valor) {  
            throw new RuntimeException();  
        } else {  
            this.saldo -= valor;  
        }  
    }  
}
```

- **RuntimeException** é uma exceção não checada
 - Mãe de todas as unchecked exceptions
- **Muito genérica**, dificulta o tratamento do erro

Lançando exceções


```
public class Conta {  
    void sacar(double valor) {  
        if (this.saldo < valor) {  
            throw new IllegalArgumentException();  
        } else {  
            this.saldo -= valor;  
        }  
    }  
}
```

- **IllegalArgumentException**, um pouco mais específica
 - Melhor opção para lidar com argumentos inválidos
- Também é uma **unchecked exception** já que herda de **RuntimeException**

Lançando exceções

```
public class Conta {  
    void sacar(double valor) {  
        if (this.saldo < valor) {  
            throw new IllegalArgumentException("Saldo Insuficiente");  
        } else {  
            this.saldo -= valor;  
        }  
    }  
}
```

```
try {  
    cc.sacar(100);  
} catch (IllegalArgumentException e) {  
    System.out.println(e.getMessage());  
}
```



Definido na interface Throwable

Lançando exceções

- Apesar das mudanças realizadas, o tratamento da exceção continua **não sendo obrigatório**
- Para tornar obrigatório, vamos lançar uma Exception

```
public class Conta {  
    void sacar(double valor) {  
        if (this.saldo < valor) {  
            throw new Exception("Saldo Insuficiente");  
        } else {  
            this.saldo -= valor;  
        }  
    }  
}
```

Lançando exceções

Criando sua própria exceção

- É uma prática bastante comum criar nossas próprias exceções
- Checked ou Unchecked exception ?
 - Criar uma classe que herde de uma classe do tipo Exception

Lançando exceções

```
public class SaldoInsuficienteException extends Exception {  
    SaldoInsuficienteException(int valorDoSaque, int saldo) {  
        super("Saldo insuficiente[" + saldo+ "], tente uma valor menor que "+ valorDoSaque);  
    }  
}
```

Lançando exceções

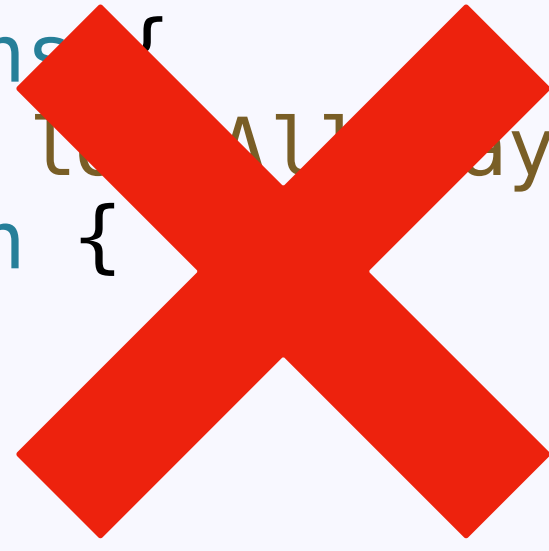
Exceções vs Herança

- Quando um método da superclasse possui a palavra chave throws, ele impacta nas subclasses que o sobreescrevem
 - Na subclasse o método pode ser menos “perigosos”
 - Lançar menos exceções
 - O método sobrescrito nunca pode ser mais “perigoso”

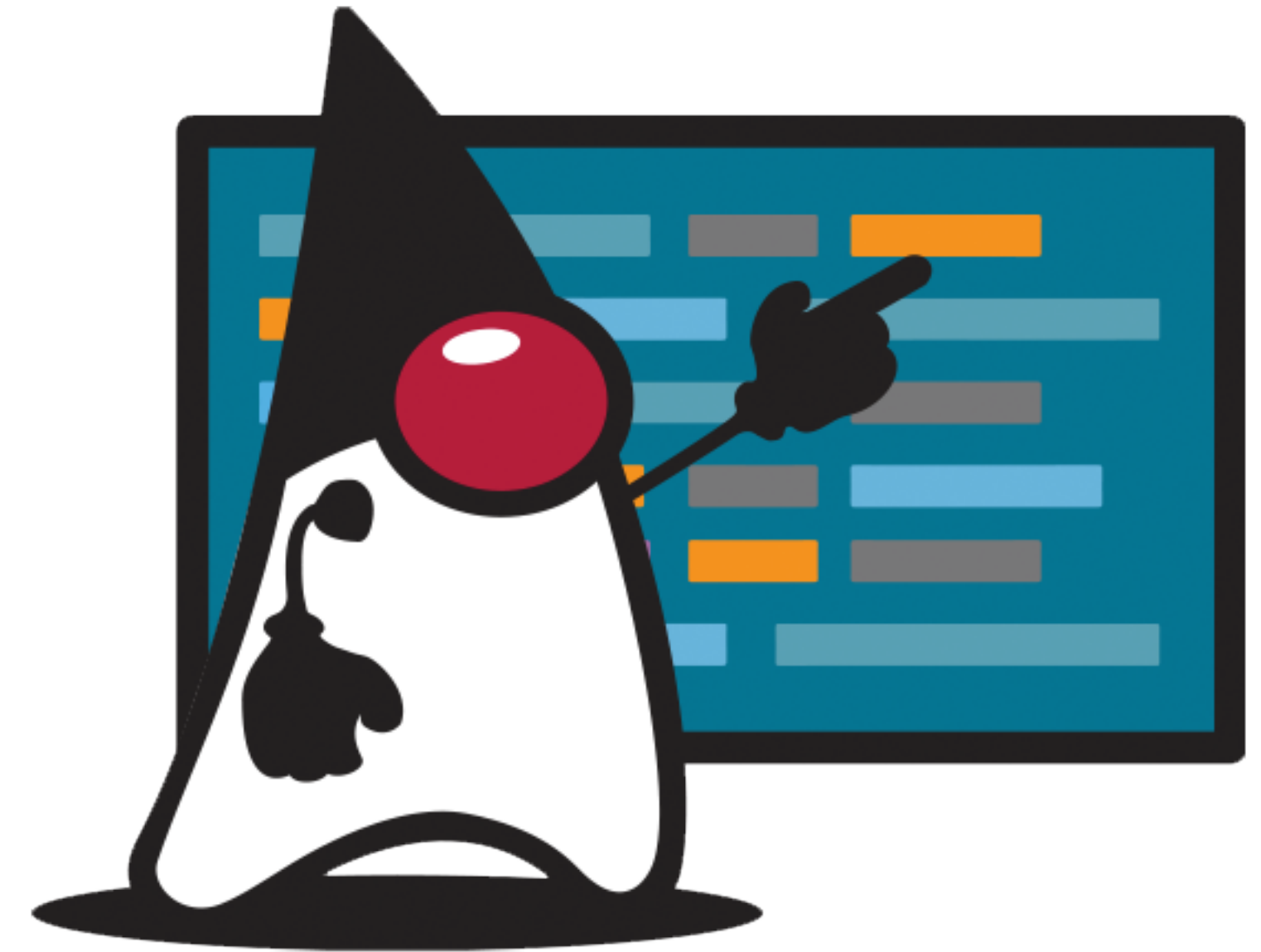
```
public class Exceptions {  
    public List<Player> loadAllPlayers(String playersFile)  
        throws TimeoutException {  
        // ...  
    }  
}
```

```
public class Exceptions {  
    public List<Player> loadAllPlayers(String playersFile){  
        // ...  
    }  
}
```

```
public class Exceptions {  
    public List<Player> loadAllPlayers(String playersFile)  
        throws IOException {  
        // ...  
    }  
}
```



Más prácticas



Más prácticas

Engolindo exceções (Swallowing Exceptions)

```
public int getPlayerScore(String playerFile) {  
    try {  
        // ...  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return 0;  
}
```


Más prácticas

Usando return no bloco finally

```
public int getPlayerScore(String playerFile) {  
    int score = 0;  
    try {  
        throw new IOException();  
    } finally {  
        return score; // <== the IOException is dropped  
    }  
}
```

Más prácticas

Usando throw no bloco finally

```
public int getPlayerScore(String playerFile) {  
    try {  
        // ...  
    } catch ( IOException io ) {  
        throw new IllegalStateException(io); // <== eaten by the finally  
    } finally {  
        throw new OtherException();  
    }  
}
```

Por hoje é só

