

Exercícios com exceções – parte 1.

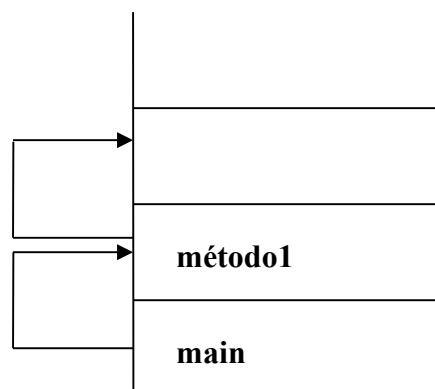
Exceção → uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

Exercícios para fixar os conceitos:

- 1) Teste o seguinte código você mesmo:

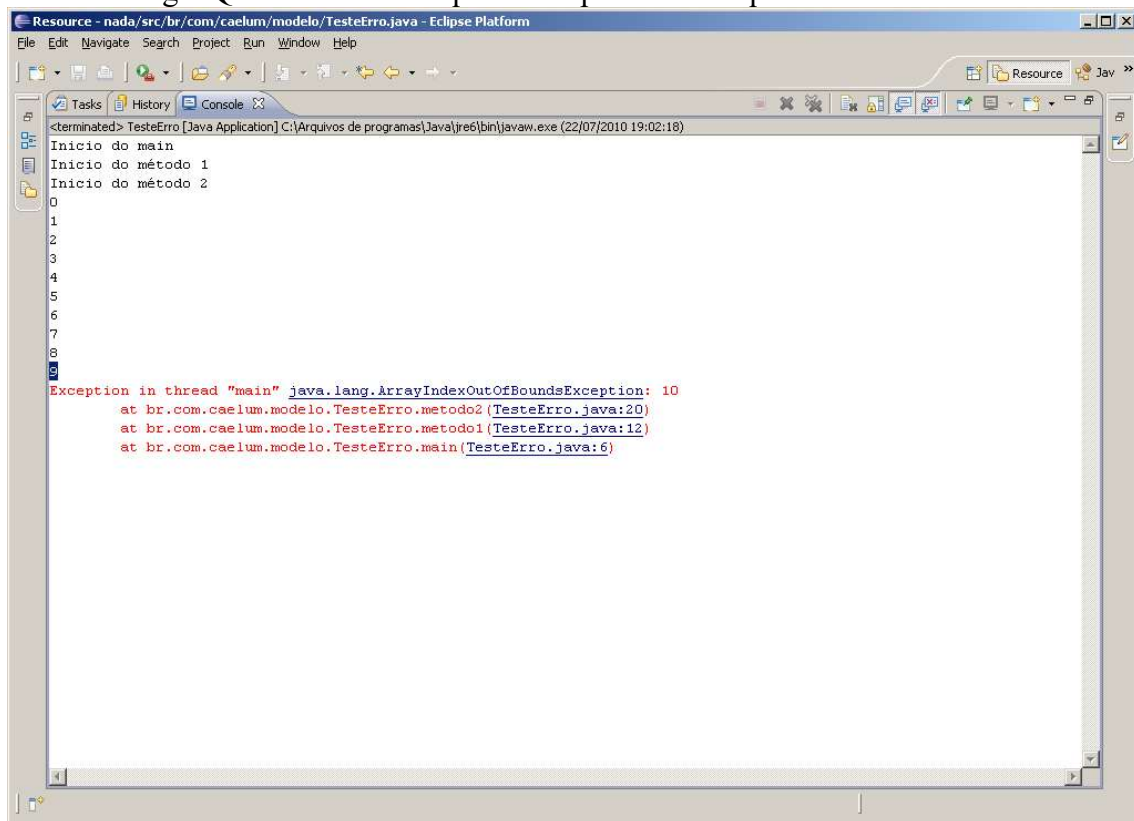
```
public class TesteErro {  
    public static void main(String[] args) {  
        System.out.println("Inicio do main");  
        metodo1();  
        System.out.println("Fim do main");  
    }  
  
    static void metodo1() {  
        System.out.println("Inicio do método 1");  
        metodo2();  
        System.out.println("Fim do método 1");  
    }  
  
    static void metodo2() {  
        System.out.println("Inicio do método 2");  
        int[] array = new int[10];  
        for (int i = 0; i <= 15; i++) {  
            array[i]=i;  
            System.out.println(i);  
        }  
        System.out.println("Fim do método 2");  
    }  
}
```

O método main chama o método1 que chama o método2. Cada um destes métodos pode ter suas próprias variáveis locais sendo que, por exemplo, o método1 não enxerga as variáveis declaradas no método main. Como o Java (e muitas outras linguagens) faz isso? Toda invocação de método é empilhada em uma estrutura de dados que isola a área de memória de cada um. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da pilha de execução (stack). Basta jogar fora um gomo da pilha (stackframe):



Propositalmente nosso método2 tem um enorme problema: está acessando um índice de array indevido. O índice estará fora dos limites do array quando chegar em 10!

Rode o código. Qual é a saída? O que isso representa? O que ela indica?



The screenshot shows the Eclipse IDE interface. The top bar indicates the file path: 'Resource - nada/src/br/com/caelum/modelo/TesteErro.java - Eclipse Platform'. The menu bar includes 'File', 'Edit', 'Navigate', 'Search', 'Project', 'Run', 'Window', and 'Help'. The toolbar contains various icons for file operations and running. The 'Console' tab is active, displaying the following output:

```
<terminated> TesteErro [Java Application] C:\Arquivos de programas\Java\jre6\bin\javaw.exe (22/07/2010 19:02:18)
Inicio do main
Inicio do método 1
Inicio do método 2
0
1
2
3
4
5
6
7
8
9
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at br.com.caelum.modelo.TesteErro.metodo2 (TesteErro.java:20)
    at br.com.caelum.modelo.TesteErro.metodo1 (TesteErro.java:12)
    at br.com.caelum.modelo.TesteErro.main (TesteErro.java:6)
```

O que você vê acima é conhecido como **rastro da pilha** (stacktrace). Essa é uma saída importantíssima para o programador. O rastro é tão importante que em qualquer fórum ou lista de discussão é comum programadores enviarem, juntamente com a descrição do problema, essa stacktrace.

Por que isso ocorreu? Quando uma exceção é lançada a JVM entra em estado de alerta e vai ver se o método atual toma alguma precaução ao tentar executar esse trecho de código. Como podemos ver o método2 não toma nenhuma medida.

Como o método2 não está **tratando** esse problema, a JVM para a execução abruptamente, sem esperar ele terminar, e volta um stackframe para baixo, onde será feita nova verificação: o método1 está se precavendo de um problema chamado *ArrayIndexOutOfBoundsException*? Não.... volta para o main, onde também não há proteção, então a thread corrente morre.

O erro aqui foi proposital. Para arrumar isso bastaria fazer com que o array navegasse no máximo até o seu length.

Para entender o controle de fluxo da exceção, vamos colocar o código que vai **tentar** (try) executar o bloco perigoso e, caso o problema seja do tipo *ArrayIndexOutOfBoundsException*, ele será **pego** (catched). Perceba que é interessante que cada exceção tenha um tipo... ela pode ter atributos e métodos.

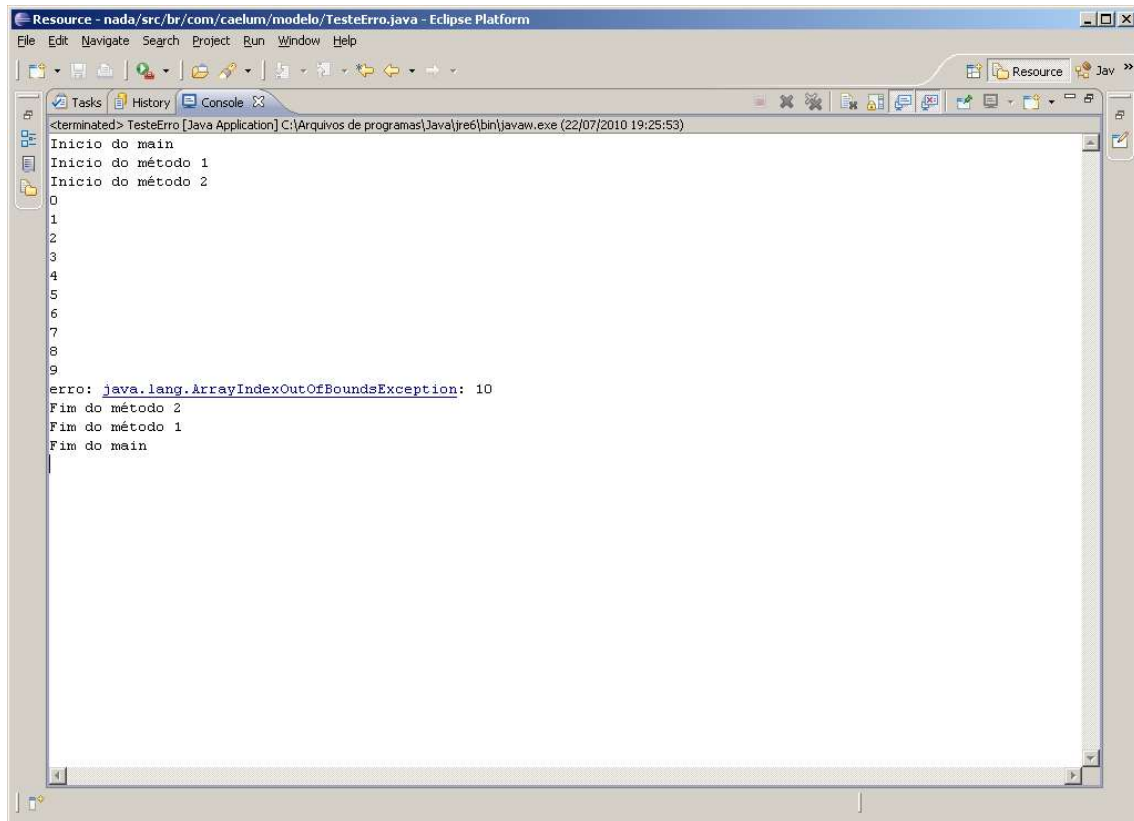
- 2) Adicione um try/catch em volta do for, pegando *ArrayIndexOutOfBoundsException*.
O que o código imprime agora?

```

try {
    for (int i = 0; i <= 15; i++) {
        array[i] = i;
        System.out.println(i);
    }
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("erro: " + e);
}

```

Qual é a diferença?

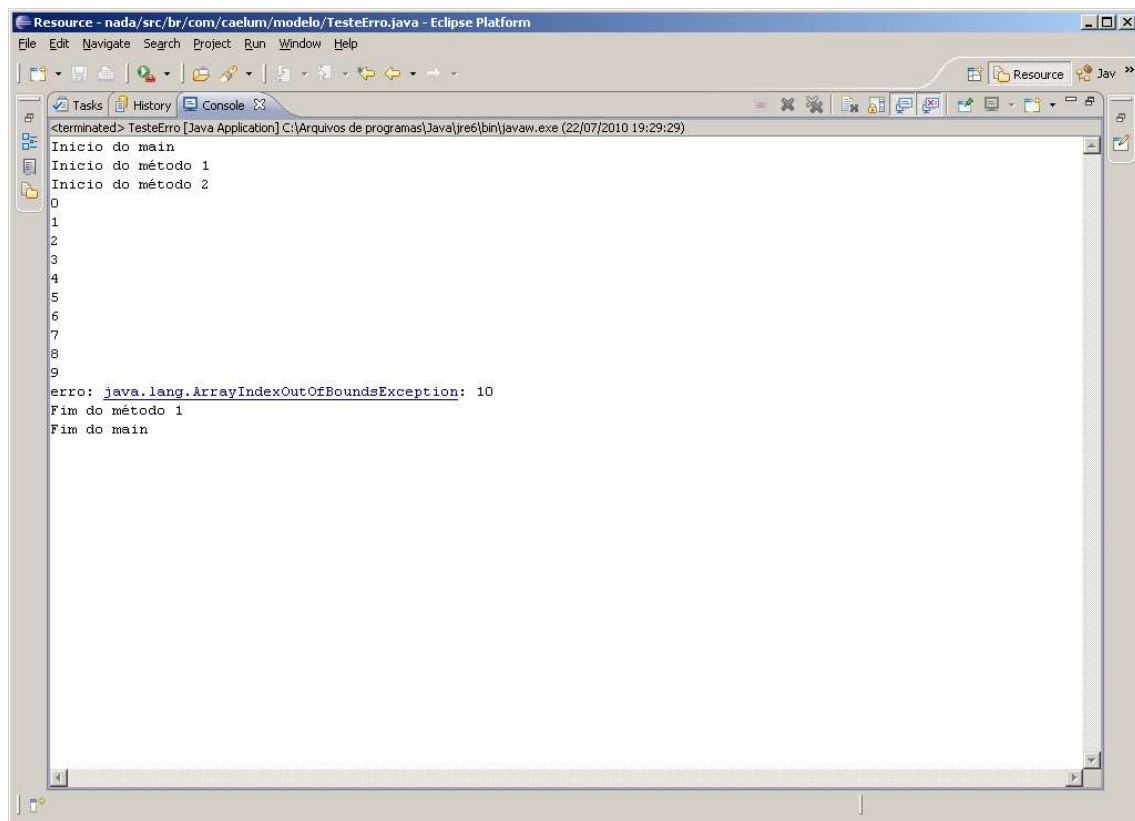


Agora retire o try/catch e coloque em volta da chamada do método2.

```

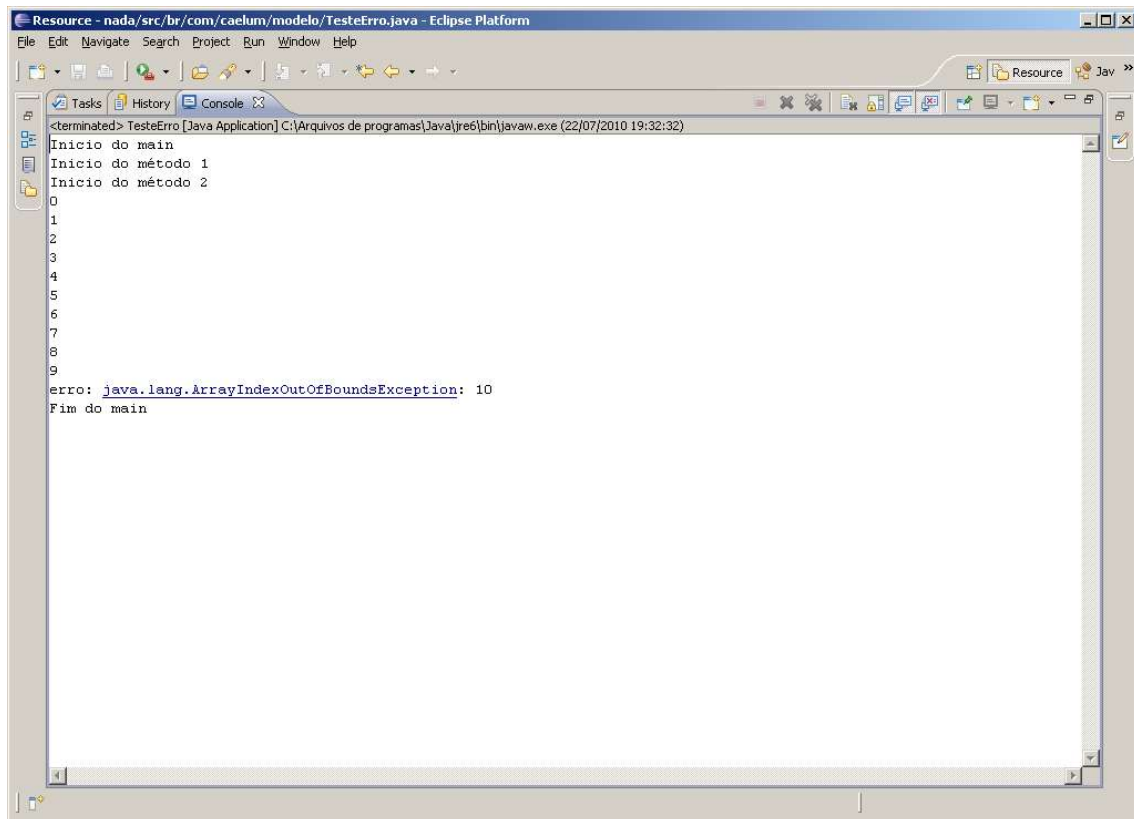
static void metodo1() {
    System.out.println("Início do método 1");
    try {
        metodo2();
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("erro: " + e);
    }
    System.out.println("Fim do método 1");
}

```



Faça a mesma coisa retirando o try/catch novamente e colocando em volta da chamada do método1. Rode os códigos, o que acontece?

```
System.out.println("Início do main");  
try {  
    metodo1();  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("erro: " + e);  
}  
System.out.println("Fim do main");
```



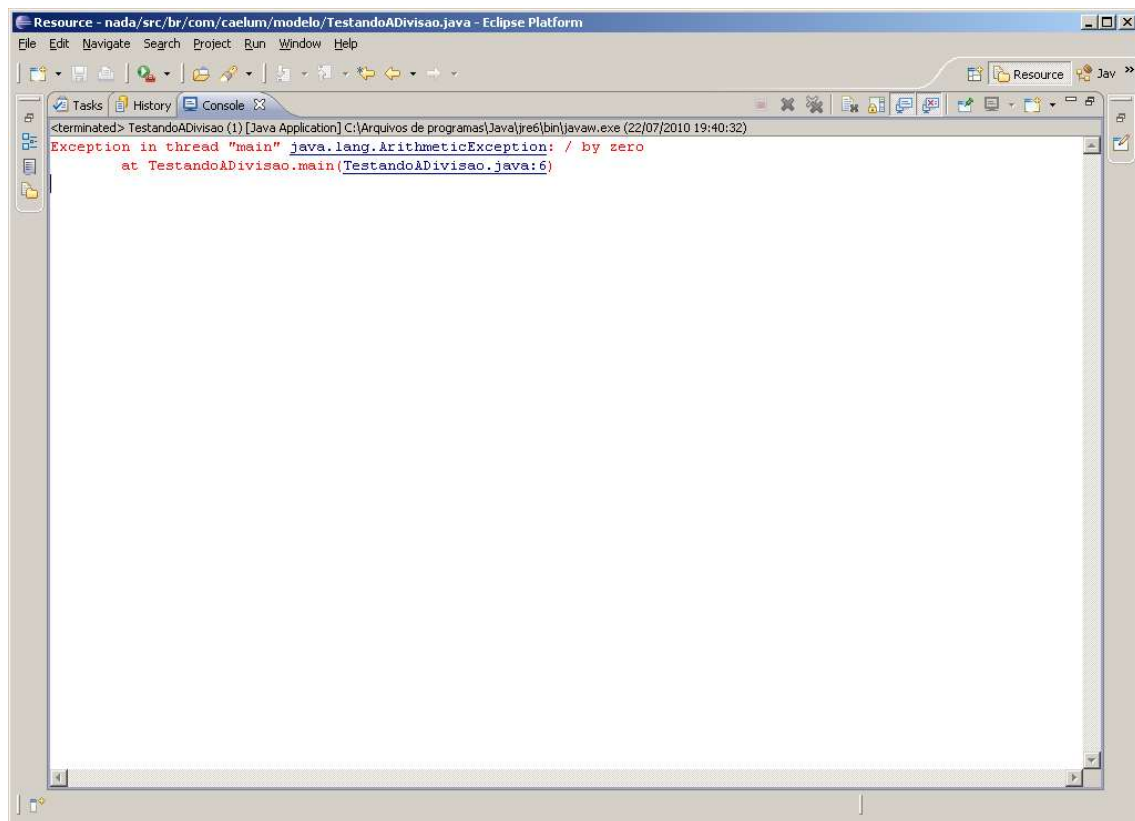
Repare que, a partir do momento que uma exception foi caught (pega, tratada, handled), a execução volta ao normal a partir daquele ponto.

Exceções de Runtime mais comuns

Tente dividir um número por zero. Será que o computador consegue fazer algo que nós definimos que não existe?

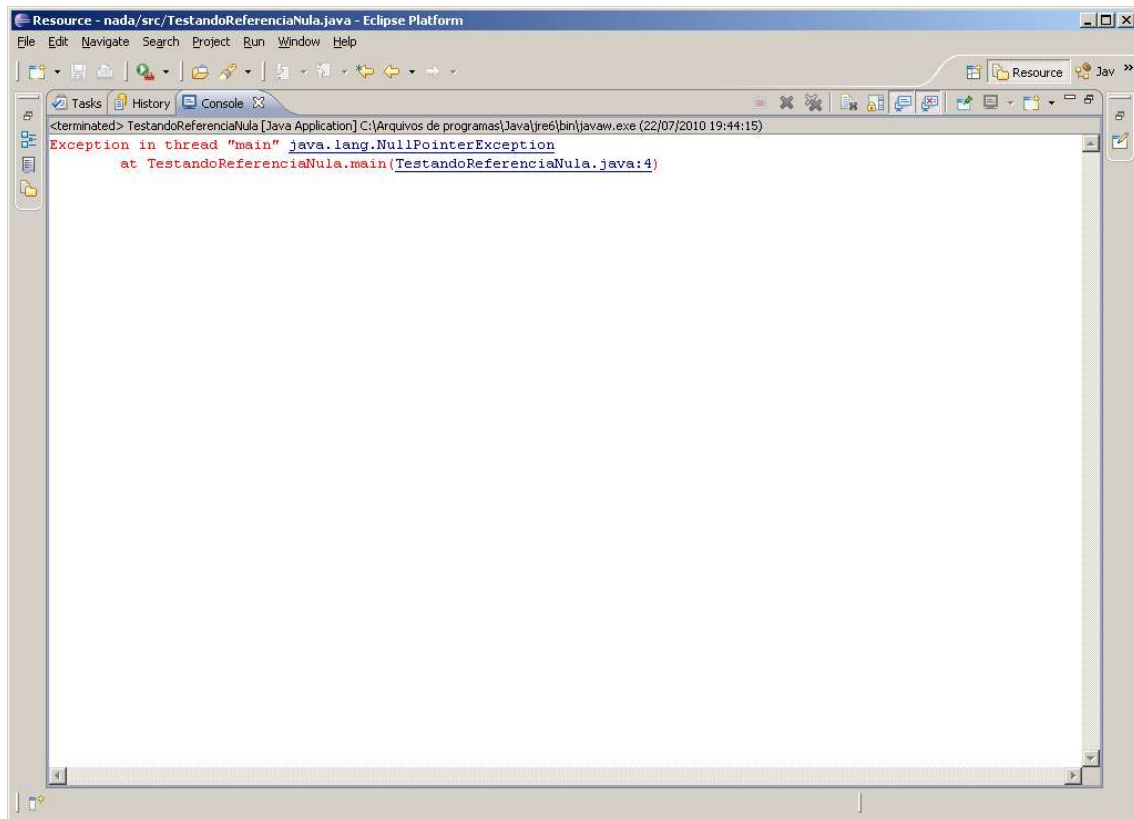
```
public class TestandoADivisao {  
    public static void main(String[] args) {  
        int i = 5000;  
        i = i / 0;  
        System.out.println("O resultado é " + i);  
    }  
}
```

Tente executar o programa acima. O que acontece?



```
public class TestandoReferenciaNula {  
    public static void main(String[] args) {  
        Conta c = null;  
        System.out.println("Saldo atual " + c.getSaldo());  
    }  
}
```

E agora? O que acontece?



Outro caso comum é quando um cast errado é feito. Veremos mais adiante. Em todos os casos, tais erros provavelmente poderiam ser evitados pelo programador. É por esse motivo que o Java não te obriga a dar try/catch nessas exceptions. Chamamos essas exceções de unchecked. Em outras palavras, o compilador não checka se você está tratando essas exceções.

Checked Exceptions:

Todos os exemplos que vimos até agora poderiam ter ficado sem o try/catch que iriam compilar e rodar. Sem usar o try/catch o erro terminou o programa, usando o try/catch o erro foi tratado. Mas, no Java não existe só esse tipo de exceção onde o tratamento é opcional. Existe também um tipo de exceção que obriga quem chama o método ou construtor a tratá-la. Chamamos esse tipo de exceção de **checked**, pois o compilador irá checkar se ela está sendo devidamente tratada.

Um exemplo clássico é o de abrir um arquivo para leitura, onde pode ocorrer o erro de o arquivo não existir (veremos como tratar arquivos mais adiante, não se preocupe com isso agora):

```
public static void metodo() {  
    new java.io.FileInputStream("arquivo.txt");  
}
```

O código não compila. O compilador avisa que é necessário tratar o FileNotFoundException que pode ocorrer.

Para compilar e fazer o programa funcionar, precisamos tratar o erro de um dos dois jeitos. O primeiro, é tratá-lo com o try e catch do mesmo jeito que usamos no exemplo anterior, com uma array:

```

    public static void metodo() {
        try {
            new java.io.FileInputStream("arquivo.txt");
        } catch (FileNotFoundException e) {
            System.out
                .println("Não foi possível abrir o arquivo
para leitura.");
        }
    }

```

A outra forma de tratar esse erro é delegar ele para quem chamou nosso método, ou seja, passar adiante.

```

    public static void metodo() throws FileNotFoundException{
        new java.io.FileInputStream("arquivo.txt");
    }

```

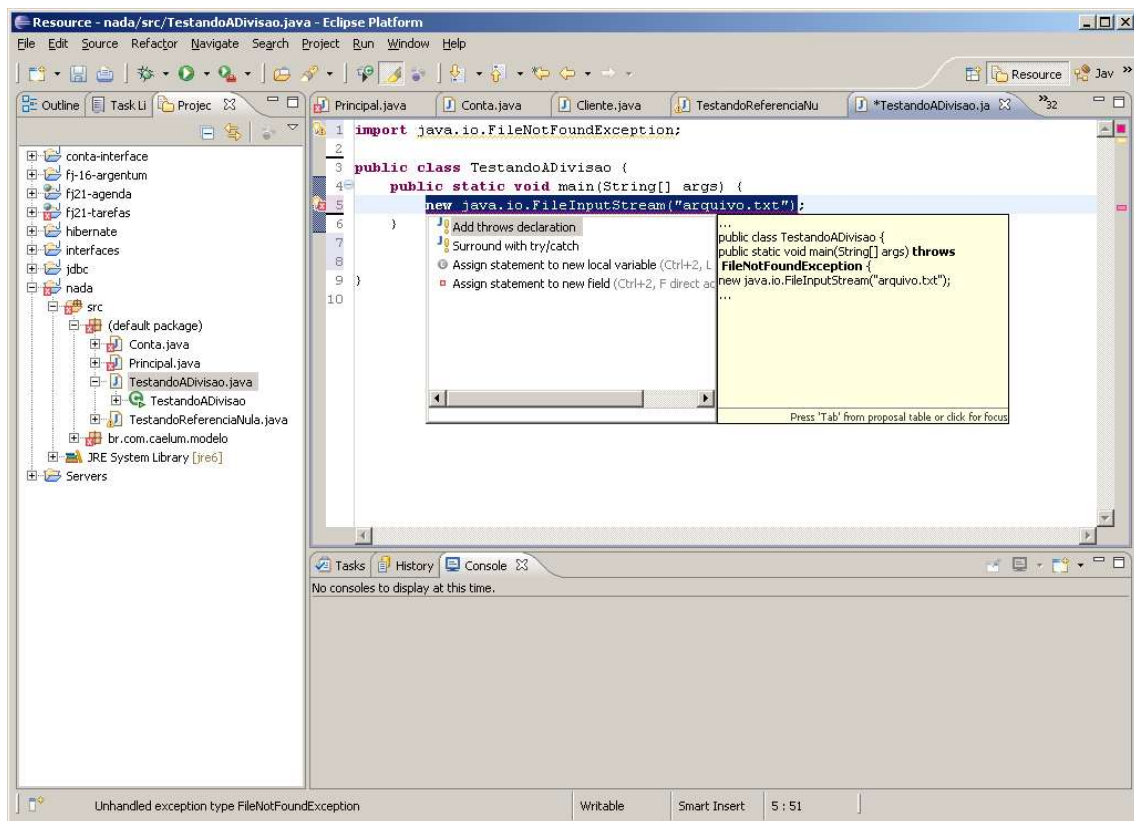
Usando o Eclipse, fica bem simples fazer tanto o try/catch como um throws: Digite esse código no Eclipse:

```

    public static void main(String[] args) {
        new java.io.FileInputStream("arquivo.txt");
    }

```

O Eclipse vai reclamar:



E te oferecer duas opções:

1) Add throws declaration que vai gerar:

```
public static void main(String[] args) throws
FileNotFoundException {
    new java.io.FileInputStream("arquivo.txt");
}
```

2) Surround with try/catch, que vai gerar:

```
public static void main(String[] args) {
    try {
        new java.io.FileInputStream("arquivo.txt");
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

No início, a tentação de sempre passar o erro para frente para outros tratarem dele é grande. Dependendo do caso até faz sentido, mas não até o **main**, por exemplo. Quem tentou abrir um arquivo não sabe como lidar com um problema na leitura. Quem chamou um método no começo do programa não saber resolver isso é muito ruim. Pode ser até pior: quem chamou o método no começo do programa pode estar tentando abrir uns 4 ou 5 arquivos diferentes e não saberá qual deles teve um problema.

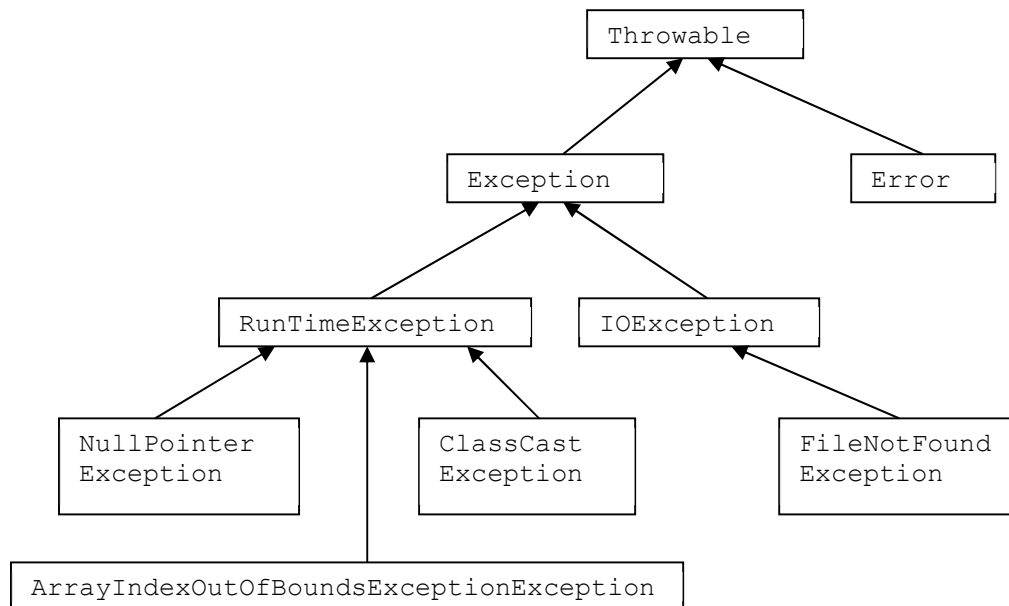
Não existe uma regra para decidir em que momento do programa você vai tratar determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação àquele erro. Enquanto não for o momento, provavelmente você vai preferir passar para frente, delegando a responsabilidade para o método que te invocou.

No endereço abaixo você vai encontrar um artigo que discute boas práticas em relação ao tratamento de exceções.

<http://blog.caelum.com.br/2006/10/07/lidando-com-exceptions/>

A família Throwable:

Eis uma pequena parte da família Throwable:



Aprendendo a lidar com mais de um erro:

É possível tratar mais de uma exceção quase que ao mesmo tempo:

- Com o try/catch:

```

public static void main(String[] args) {
    try {
        objeto.metodoQuePodeLancarException();
    } catch (IOException e) {
        //...
    } catch (SQLException e) {
        //...
    }
}

```

- Com o throws:

```

public void abre(String arquivo) throws IOException, SQLException{
    //...
}

```

- Tratando algumas exceções e declarando outras no throws:

```

public void abre(String arquivo) throws IOException{
    try {
        objeto.metodoQuePodeLancarIOeSQLException();
    } catch (SQLException e) {
        //...
    }
}

```

Declarar, no throws, as exceptions que são unchecked, é desnecessário, porém é permitido e, às vezes, facilita a leitura e a documentação do seu código.

Lançando Exceções:

Lembra do nosso método `saca` da classe `Conta`? Conseguindo ou não sacar, ele devolve um `boolean`:

```
public boolean saca(double quantidade) {
    if ((this.saldo) >= quantidade) {
        this.saldo -= quantidade;
        return true;
    } else {
        return false;
    }
}
```

Caso ocorra algo inesperado, como um valor negativo para saque, podemos lançar uma `Exception`, o que é extremamente útil. Assim, resolvemos o problema de alguém poder esquecer-se de fazer um `IF` no retorno do método.

A palavra chave `throw` lança uma `Exception`. Não é o mesmo caso que `throws`, que apenas avisa da possibilidade daquele método lançá-la, obrigando o outro método, que chamou este, a se preocupar com a referida `exception`.

```
public void saca(double quantidade) {
    if ((this.saldo) >= quantidade) {
        this.saldo -= quantidade;
    } else {
        throw new RuntimeException();
    }
}
```

No código acima estamos lançando uma exceção do tipo `unchecked`. `RunTimeException` é a exceção mãe de todas as `exceptions unchecked`. A desvantagem, nesse caso, é que ela é muito genérica. Quem receber esse erro não saberá dizer exatamente qual foi o problema. Podemos então usar uma `Exception` mais específica:

```
public void saca(double quantidade) {
    if ((this.saldo) >= quantidade) {
        this.saldo -= quantidade;
    } else {
        throw new IllegalArgumentException();
    }
}
```

`IllegalArgumentException` já esclarece um pouco mais: algo foi passado como argumento e seu método não gostou. Ela é uma `Exception unchecked`, pois estende de `RunTimeException`. Quando um argumento sempre é inválido (números negativos, referências nulas, etc), `IllegalArgumentException` é a melhor escolha!

Por fim, para pegar esse erro, não usaremos um `IF/else` e sim um `try/catch`, porque faz mais sentido já que a falta de saldo é uma exceção:

```
Conta cc = new ContaCorrente();
cc.deposita(100);
try {
    cc.saca(300);
} catch (IllegalArgumentException e) {
    System.out.println("saldo insuficiente.");
}
```

Podemos melhorar ainda mais o código, passando para o construtor de `IllegalArgumentException` o motivo da exceção.

```
public void saca(double quantidade) {
    if ((this.saldo) >= quantidade) {
        this.saldo -= quantidade;
    } else {
        throw new IllegalArgumentException("saldo
insuficiente.");
    }
}
```

O método `getMessage()` definido na classe `Throwable` (mãe de todos os tipos de erros e exceptions) vai retornar a mensagem que passamos ao construtor da `IllegalArgumentException`.

```
try {
    cc.saca(300);
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

O que devo colocar dentro do try?

Imagine que vamos sacar dinheiro de diversas contas:

```
Conta cc = new ContaCorrente();
cc.deposita(100);

Conta cp = new ContaPoupanca();
cp.deposita(100);

cc.saca(50);
System.out.println("Conseguí sacar da conta corrente.");

cp.saca(50);
System.out.println("Conseguí sacar da conta poupança.");
```

Onde colocar a mensagem “Conseguí sacar”?

Sempre que temos algo que depende do sucesso da linha de cima para ser correto, devemos agrupá-lo no try:

```
try {
    cc.saca(300);
    System.out.println("Conseguí sacar da conta
corrente.");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}

try {
    cp.saca(300);
    System.out.println("Conseguí sacar da conta
poupança.");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

Ainda há uma outra opção: imagine que, para o nosso sistema, uma falha ao sacar da conta poupança deve parar o processo de saques e nem tentar sacar da conta corrente. Para isso, agruparíamos mais ainda:

```
try {
    cc.saca(300);
    System.out.println("Conseguir sacar da conta corrente.");

    cp.saca(300);
    System.out.println("Conseguir sacar da conta poupança.");
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage());
}
```

O que você vai colocar dentro do try influencia demais na execução do programa! Pense bem nas linhas que dependem uma da outra para a execução correta de sua lógica de negócios.

Como criar seu próprio tipo de Exceção?

Para controlar melhor o uso de suas exceções, é bem comum criar uma própria classe de Exceção. Dessa forma, podemos passar valores específicos para ela carregar que sejam úteis de alguma forma. Vamos criar nossa classe de Exceção SaldoInsuficienteException:

```
public class SaldoInsuficienteException extends RuntimeException {
    public SaldoInsuficienteException(String message) {
        super(message);
    }
}
```

Agora ao invés de lançar uma IllegalArgumentException, vamos lançar nossa própria exceção, com a seguinte mensagem: “Saldo Insuficiente.”

```
public void saca(double quantidade) {
    if ((this.saldo) >= quantidade) {
        this.saldo -= quantidade;
    } else {
        throw new SaldoInsuficienteException("saldo insuficiente.");
    }
}
```

Para testar, crie uma classe que deposite um valor e tente sacar um valor maior:

```
public static void main(String[] args) {
    Conta cc = new ContaCorrente();
    cc.deposita(10);

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

Podemos também transformar essa exceção de unchecked para checked, obrigando quem chama esse método a dar try/catch, ou throws:

```
public class SaldoInsuficienteException extends Exception {  
    public SaldoInsuficienteException(String message) {  
        super(message);  
    }  
}
```