



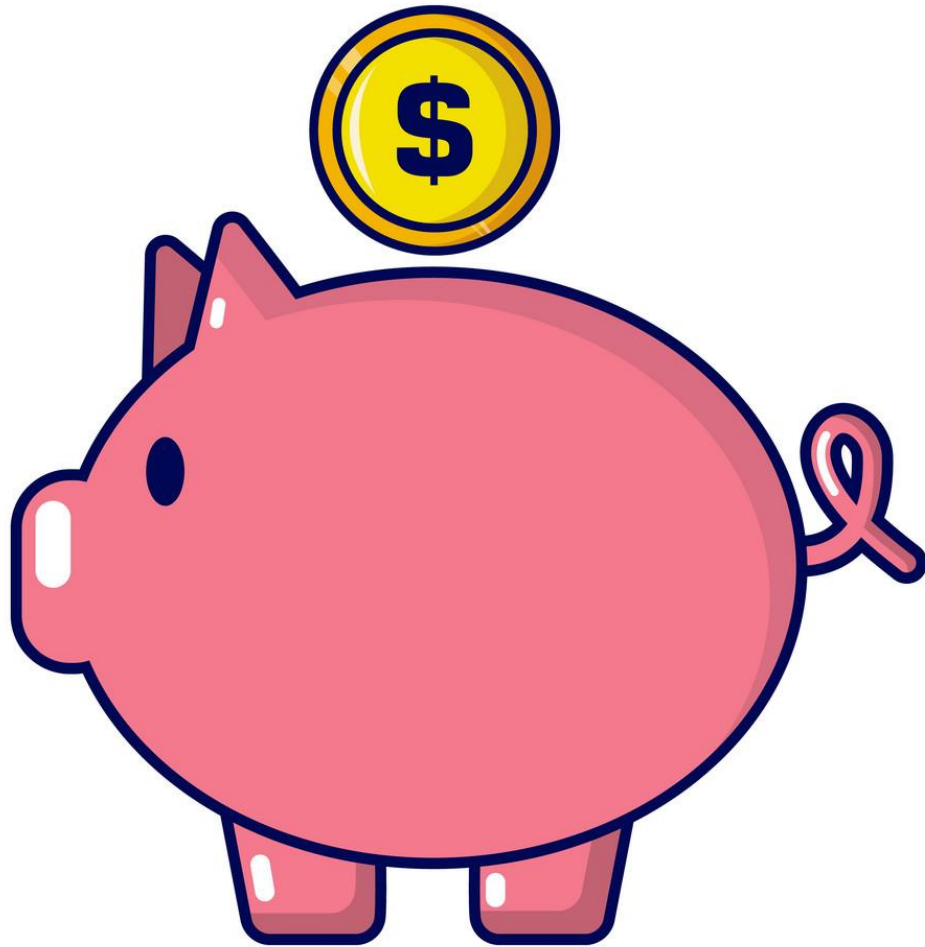
C125/C206 – Programação Orientada a Objetos
com Java

Exceções e Controle de Erros

Prof. Phyllipe Lima
phyllipe@inatel.br



☕ Vamos resgatar nossas classes Conta e Cliente (originais)





```
public class Conta {  
  
    private double saldo;  
    private double limite;  
    private Cliente[] clientes;  
  
    public Conta(double saldo, double limite) {  
        this.saldo = saldo;  
        this.limite = limite;  
        this.clientes = new Cliente[3];  
    }  
  
    public boolean sacar(double quantia) {  
        this.saldo -= quantia;  
    }  
}
```



```
public class Cliente {  
  
    private String nome;  
    private int cpf;  
  
    public Cliente(String nome, int cpf) {  
        this.nome = nome;  
        this.cpf = cpf;  
    }  
}
```



Saldo Insuficiente?

- ☕ Observe o método **sacar()** na classe Conta
- ☕ O que acontece se tentarmos sacar um valor com saldo menor que o disponível?

```
public static void main(String[] args) {  
  
    Conta conta = new Conta("Tiozin", 300, 50);  
  
    conta.sacar(400);  
  
    System.out.println(conta.getSaldo());  
    //Vai imprimir -100  
  
}
```

-100.0



Como Avisar?

- ☕ A situação do slide anterior é um ***desastre absoluto***. Imagine um caixa eletrônico permitindo essa situação?
- ☕ O fato de estarmos utilizando o ***modificador private*** já é um avanço e protege o membro ***saldo***. Mas ainda não está 100% protegido, como já vimos em aulas passadas
- ☕ Como corrigir essa situação e ainda ***avisar quem chamou*** que não foi possível fazer a operação?
- ☕ Se, ao tentar sacar, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar que o saque não foi feito.



Fazendo Teste?

- ☕ Uma solução simples que já vimos nas aulas passadas é fazer o teste se o saldo é suficiente e retornar um **booleano** informando **true** caso o saque tenha sido feito com sucesso, ou **false** caso contrário.
- ☕ Esse ajuste precisa ser feito no próprio método **sacar()**.

```
public boolean sacar(double quantia) {  
    if(quantia < (saldo + limite)) {  
        this.saldo -= quantia;  
        return true;  
    }else {  
        System.out.println("Saldo Insuficiente.");  
        return false;  
    }  
}
```



Verificando!

☕ Essa solução parece boa. Mas ***quem chama*** precisa verificar se o retorno foi ***true*** ou ***false***.

☕ Considere o exemplo abaixo, onde ***quem chama*** o método ***sacar()*** faz o teste

```
Conta conta = new Conta("Tiozin", 300, 50);
```

```
boolean sacou = conta.sacar(400);
```

```
if(sacou) {  
    System.out.println("Deu Certo");  
    //Segue o programa  
}else {  
    System.out.println("Deu ruim :(");  
}
```




Esqueci de Verificar?

☕ Mas **quem chama** não é obrigado a verificar! Nada o está forçando a fazer isso.

☕ Imagine que **quem chamou** o método **sacar()** simplesmente ignore o **booleano** que este método retorna e siga a vida normalmente.

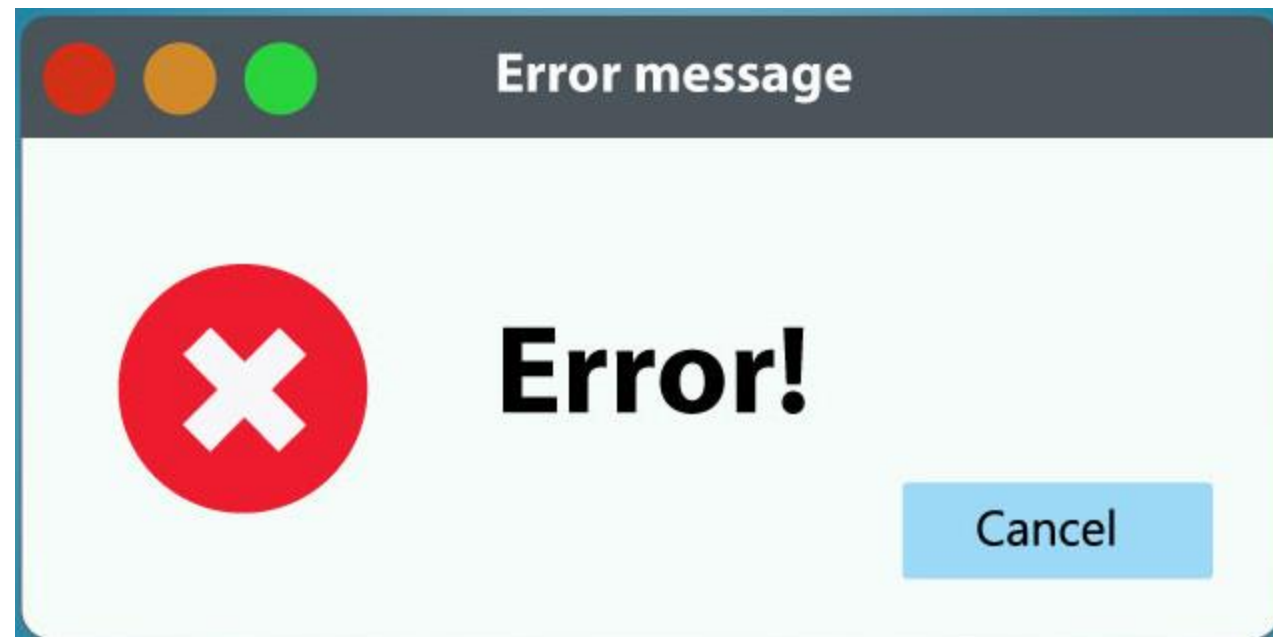
```
Conta conta = new Conta("Tiozin", 300, 50);
```

```
conta.sacar(400);  
//O saque não deu certo  
//Mas eu não testei o retorno  
//E o programa continua executando como se o saldo tivesse sido  
//atualizado. Poderia chamar agora algum sistema de caixa eletrônico  
//para emitir R$ 400 reais.  
//Outro desastre
```

```
//Agora vamos fazer um deposito  
//Ignorando o que aconteceu antes  
conta.deposita(100);
```



- ☕ Por esses e outro motivos que o Java oferece um mecanismo próprio para tratamento de erro. Chamamos de Exceção!
- ☕ É uma ***exceção à regra!***
- ☕ Representa uma situação que ***normalmente (regra)*** não ocorre, é algo inesperado e estranho no sistema



PARA! PARA!
PARA! PARA! PARA!
PARA! PARA!



- ☕ Antes de resolvermos esse problema, vamos verificar como o Java lida com algumas situações ***estranhas***
- ☕ Considere o código na classe Main no próximo slide



```
public class Main {  
  
    public static void main(String[] args) {  
        System.out.println("Estamos no inicio main");  
        metodo1();  
        System.out.println("Fim da main");  
    }  
    public static void metodo1() {  
  
        System.out.println("Estamos no inicio do método 1");  
        metodo2();  
        System.out.println("Estamos no fim do método 1");  
    }  
    public static void metodo2() {  
        System.out.println("Estamos no inicio do método 2");  
        Conta conta = null;  
        conta.deposita(250);  
        System.out.println("Estamos no fim do método 2");  
    }  
}
```



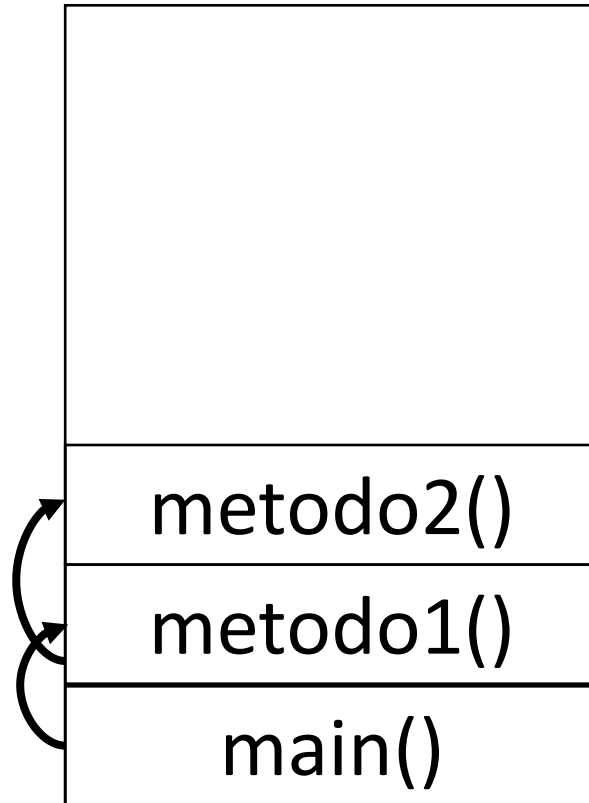
Pilha de Execução

- ☕ Observe que temos três métodos
- ☕ O método ***main*** chama o método ***metodo1()***
- ☕ O método ***método1()*** chama o método ***metodo2()***
- ☕ O objetivo é verificar como o Java, e várias outras linguagens de programação, coordenam a execução de métodos através da ***pilha de execução*** ou ***stack***
- ☕ Cada método tem sua própria área de memória, isto é, suas variáveis locais somente são visíveis dentro do próprio método. Esse é conceito de ***escopo das variáveis***.
- 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).



Pilha de Execução

Stack



- ☕ A main **chama** o **metodo1()**. Assim, ele é empilhado por cima da **main**.
- ☕ O **metodo1()**, por sua vez, **chama** o **metodo2()**. Este é empilhada também.
- ☕ Quando o **metodo2()** termina sua execução, o programa retorna para o ponto onde se encontrava no **metodo1()**.
- ☕ E assim por diante
- ☕ Mas e o erro que tem no **metodo2()**?



Apontando Para o Além!

```
public static void metodo2() {  
    System.out.println("Estamos no inicio do método 2");  
    Conta conta = null;  
    conta.deposita(250);  
    System.out.println("Estamos no fim do método 2");  
}
```

- ☕ A variável **conta** esta nula.
- ☕ Logo em seguida tentamos fazer um depósito
- ☕ Qual será o resultado ao executar esse código?



Apontando Para o Além!

```
Estamos no inicio main  
Estamos no inicio do método 1  
Estamos no inicio do método 2  
Exception in thread "main" java.lang.NullPointerException  
    at br.inatel.cdg.Main.metodo2(Main.java:29)  
    at br.inatel.cdg.Main.metodo1(Main.java:19)  
    at br.inatel.cdg.Main.main(Main.java:11)
```

- ☕ Aqui temos o erro/exceção mais tradicional no Java. A ***NullPointerException***
- ☕ O resultado em vermelho na figura nos mostra a ***stracktrace*** (rastro da pilha)
- ☕ Como esse rastro é utilizado?



Desempilhando!

- ☕ Quando uma exceção é lançada, a JVM entra em alerta, e começa a **percorrer a pilha(stack)** verificando se alguém trata o erro.
- ☕ Ela começa primeiro no próprio **metodo2()** (pois está no topo da pilha)
- ☕ A JVM pergunta “Ei metodo2() você está tratando algum NullPointerException?”. Sabemos que não. Assim a JVM para a execução desse método, **desempilha** a stack e vai até quem o chamou, nesse exemplo o **metodo1()**

```
Estamos no inicio main
Estamos no inicio do método 1
Estamos no inicio do método 2
Exception in thread "main" java.lang.NullPointerException
    at br.inatel.cdg.Main.metodo2(Main.java:29)
    at br.inatel.cdg.Main.metodo1(Main.java:19)
    at br.inatel.cdg.Main.main(Main.java:11)
```





Desempilhando!

☕ A JVM vai investigar se o ***metodo1()*** tem algum mecanismo de tratar esse erro. Também sabemos que não. Portanto ela encerra a execução do ***metodo1()*** faz o ***desempilhamento*** e segue para o método da pilha, que é o ***main()***.

```
Estamos no início main  
Estamos no início do método 1  
Estamos no início do método 2  
Exception in thread "main" java.lang.NullPointerException  
    at br.inatel.cdg.Main.metodo2(Main.java:29)  
    at br.inatel.cdg.Main.metodo1(Main.java:19)  
    at br.inatel.cdg.Main.main(Main.java:11)
```





Desempilhando!

☕ A **main()** também não trata nada!

☕ Assim a JVM encerra o programa! Fim! Acabou! Já era

☕ Explosões



```
Estamos no início main
Estamos no início do método 1
Estamos no início do método 2
Exception in thread "main" java.lang.NullPointerException
    at br.inatel.cdg.Main.metodo2(Main.java:29)
    at br.inatel.cdg.Main.metodo1(Main.java:19)
    at br.inatel.cdg.Main.main(Main.java:11)
```





Forçando a barra!

- ☕ Obviamente forçamos a barra nesse exemplo!
- ☕ É simples resolver essa situação fazendo um condicional.





Tentando Pegar a Exceção!

- ☕ Para entendermos o controle de fluxo de uma **Exceção**, vamos colocar o código que vai **tentar** (**try**) executar o bloco perigoso e, caso o problema seja especificamente do tipo **NullPointerException**, ele será **pego** (**catch**). Repare que cada exceção no Java tem um tipo. Ela pode ter membros e métodos.
- ☕ Faremos esse tratamento no **metodo2()**



Tentando Pegar a Exceção!

```
public static void metodo2() {  
    System.out.println("Estamos no inicio do método 2");  
    try {  
        Conta conta = null;  
        conta.deposita(250);  
    } catch (NullPointerException e) {  
        System.out.println("Erro: " + e);  
    }  
  
    System.out.println("Estamos no fim do método 2");  
}
```

☕ Vamos executar o código e observar a saída



Tentando Pegar a Exceção!

```
Estamos no inicio main
Estamos no inicio do método 1
Estamos no inicio do método 2
Erro: java.lang.NullPointerException
Estamos no fim do método 2
Estamos no fim do método 1
Fim da main
```

- ☕ Observe a mensagem de erro. Isso significa que **pegamos** a exceção
- ☕ Veja que as demais mensagens do **metodo1()** e **main()** foram impressas!
- ☕ A JVM não abortou o programa dessa vez, pois **tratamos a exceção**



Andando na pilha!

☕ Vamos agora tratar a exceção no ***metodo1()***.

☕ Fazemos isso colocando ***try-catch*** antes de chamar o ***metodo2()***

```
public static void metodo1() {  
    System.out.println("Estamos no inicio do método 1");  
    try {  
        metodo2();  
    } catch (NullPointerException e) {  
        System.out.println("Erro: " + e);  
    }  
    System.out.println("Estamos no fim do método 1");  
}
```




Andando na pilha!

☕ Observe a saída após a execução!

```
Estamos no inicio main
Estamos no inicio do método 1
Estamos no inicio do método 2
Erro: java.lang.NullPointerException
Estamos no fim do método 1
Fim da main
|
```

☕ A mensagem “fim do método 2” não foi chamada.

☕ Isso porque a JVM abortou a execução do **metodo2()** pois ele não tratou a exceção.

☕ Com isso a JVM avançou na pilha e verificou que o **metodo1()** tratou.

☕ Assim o **metodo1()** em diante foi executado normalmente

Andando na pilha!



☕ Repare que, a partir do momento que uma **Exceção** foi **tratada (catch, pega, handled)**, a execução volta ao normal a partir daquele ponto!



Exceções *Unchecked*!

☕ Vamos dividir um número por zero! Como a JVM resolve isso?

```
public static void main(String[] args) {  
  
    int x = 10;  
    int y = 0;  
  
    System.out.println(x/y);  
  
}
```

☕ Temos aqui a ***ArithmeticException***

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at br.inatel.cdg.Main.main(Main.java:13)
```



Exceções *Unchecked*!

- ☕ Essa última **exceção** poderia facilmente ser tratada com um simples ***if-else***.
- ☕ A ***NullPointerException*** também. Bastava verificar se a variável ***conta*** era nula.
- ☕ Nesses casos, tais problemas provavelmente poderiam ser evitados pelo programador. É por esse motivo que o java não te obriga a dar o ***try/catch*** nessas exceções.
- ☕ Chamamos essas exceções de ***unchecked***. Em outras palavras, o compilador não checa se você está tratando.

Exercício - 1



- ☕ Na classe Conta, crie um método ***mostraInfo()***.
- ☕ Esse método deverá mostrar as informações da conta (saldo, limite) e também as informações dos clientes.
- ☕ A Conta pode ter até no máximo 4 clientes.
- ☕ Adicione apenas 2 clientes, deixando o restante do array nulo.
- ☕ Dentro do método ***mostraInfo()*** faça um ***for-each*** no array de Clientes e trate o **NullPointerException** utilizando o ***try-catch*** quando for apresentar os dados do clientes. Faça isso dentro do método ***mostraInfo()***. Lembre-se, apenas 2 clientes foram adicionadas, mas o ***for-each*** irá varrer todo o array
- ☕ No final do método ***main()*** adicione uma mensagem para garantir que o programa executou até o final



Checked Exceptions

- ☕ Vimos que as ***Unchecked Exceptions*** ou ***Runtime Exceptions*** não obrigam o programador a trata-las. Isto é, não é obrigatório o uso do ***try/catch***.
- ☕ O código irá compilar sem problemas.
- ☕ Mas existem também as ***Checked Exceptions*** onde o programador é obrigado a tratar a potencial exceção.



Abrindo um Arquivo

- ☕ Considere o código abaixo onde tentamos acessar um arquivo chamado configuração.txt.
- ☕ Não se preocupe em como abrir arquivos ainda veremos isso no curso.

```
public static void main(String[] args) {  
    FileInputStream file = new FileInputStream("configuracao.txt");  
}
```

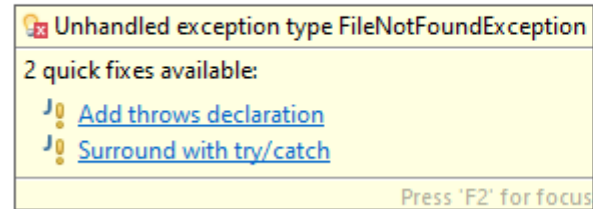
- ☕ Observe a linha vermelha. Isso significa que esse código ***não compila***



O Arquivo Existe?

☕ O Eclipse nos ajuda a verificar a razão desse código não estar compilando.

```
FileInputStream file = new FileInputStream("configuracao.txt");
```



- ☕ Ele diz “Unhandled Exception”. Isso significa que existe uma **exceção** que não estamos tratando. Mas essa **exceção** é do tipo **checked** então precisamos trata-la **obrigatoriamente**.
- ☕ Repare o nome da exceção que pode ser lançada, **FileNotFoundException**. Isso significa que o arquivo pode não existir.



O Arquivo Existe?

☕ Vamos utilizar a segunda opção ***Surround with try/catch***.

```
try {  
    FileInputStream file = new FileInputStream("configuracao.txt");  
} catch (FileNotFoundException e) {  
    //Iremos simplesmente imprimir o erro  
    e.printStackTrace();  
}
```

☕ O próprio Eclipse já colocou o ***try/catch*** para nós. E se acontecer a exceção ***FileNotFoundException*** o código que está dentro do ***catch*** será executado.

☕ Nesse exemplo estamos apenas imprimindo o erro.



O Arquivo Não Existe

☕ Vamos executar o código abaixo e ver a saída

```
System.out.println("Tentando abrir o arquivo");
try {
    FileInputStream file = new FileInputStream("configuracao.txt");
} catch (FileNotFoundException e) {
    //Iremos simplesmente imprimir o erro
    e.printStackTrace();
}
System.out.println("Depois de tentar abrir o arquivo");
```

Tentando abrir o arquivo

```
java.io.FileNotFoundException: configuracao.txt (O sistema não pode encontrar o arquivo especificado)
    at java.io.FileInputStream.open0(Native Method)
    at java.io.FileInputStream.open(Unknown Source)
    at java.io.FileInputStream.<init>(Unknown Source)
    at java.io.FileInputStream.<init>(Unknown Source)
    at br.inatel.cdg.Main.main(Main.java:15)
```

Depois de tentar abrir o arquivo



O Arquivo Não Existe

- ☕ Ao executar o código, uma mensagem de erro foi impressa (a ***stacktrace***) informando que o arquivo não foi localizado. Por isso a ***exceção*** foi lançada.
- ☕ Mas como ela foi tratada através do ***try/catch*** o programa não é interrompido de forma abrupta. Observe que a mensagem “Depois de tentar abrir o arquivo” também foi impressa. Isso reforça que o programa ***tratou a exceção*** pois continuou sua execução.
- ☕ Naturalmente para uma aplicação final, colocaríamos uma mensagem mais elegante dentro do bloco ***catch*** para apresentar ao usuário. Por exemplo, considerando a classe Conta, podemos informar que o saldo é insuficiente.



Jogando o Erro para o Outro

☕ Vamos tentar a outra opção que o Eclipse forneceu “**Add throws declaration**”.

```
public static void main(String[] args) throws FileNotFoundException {  
    FileInputStream file = new FileInputStream("configuracao.txt");
```

☕ Observe que ao lado do método **main()** surgiu o código “**throws FileNotFoundException**”.

☕ Isso significa que quem invoca o método **main()** será obrigado a tratar essa exceção.



Jogando o Erro para o Outro

- ☕ Dentro do método ***main()*** temos um código que tenta abrir um arquivo, e portanto precisa ***obrigatoriamente*** tratar uma ***checked exception***.
- ☕ De fato essa exceção necessita ser tratada ***obrigatoriamente***. Mas podemos simplesmente delegar isso para ***quem nos invoca***. Ou seja, ***alguém será obrigado a tratar isso***. Mas quem?
- ☕ O método ***que invoca*** o método que pode lançar (***throws***) a exceção deverá então tratá-la. Ou ele pode, por sua vez, também delegar para quem o invoca. E assim por diante, até chegar na ***main()***. Todos correndo da responsabilidade de tratar.
- ☕ Porém essa é a melhor opção? Ficar simplesmente delegando para outros métodos?

Jogando o Erro para o Outro



- ☕ A princípio pode parecer tentador simplesmente **delegar** a exceção para **quem invoca** através do **throws**. Mas, normalmente, quem deveria tratar o erro seria quem está tentando executar a operação.
- ☕ Quem tenta abrir um arquivo **deveria saber** como lidar com um problema na leitura.
- ☕ Quem chamou esse método no começo do programa pode não saber ou, pior ainda, tentar abrir cinco arquivos diferentes e não saber qual deles teve um problema!
- ☕ Um caso óbvio para não usar o **throws** é no método **main()**. Pois quem o chama é a própria JVM. Assim ela jamais será tratada e o programa será interrompido.



Jogando o Erro para o Outro

☕ Um exemplo mais adequado do uso do ***throws*** seria algo como no código abaixo.

```
public static void main(String[] args){  
    try {  
        metodo1();  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    }  
}  
  
private static void metodo1() throws FileNotFoundException {  
    FileInputStream file = new FileInputStream("configuracao.txt");  
}
```



Jogando o Erro para o Outro

- ☕ Observe que o ***metodo1()*** tenta abrir um arquivo. Ele é ***obrigado*** a tratar essa chamada. Ou através do ***try/catch*** ou ***delegando*** através do ***throws***.
- ☕ Optando pelo ***throws***, quem invoca o ***metodo1()*** deverá obrigatoriamente tratar essa possível ***exceção*** através do ***try/catch*** ou ainda, ***delegando*** para quem invoca o ***metodo1()***.
- ☕ Mas quem invoca o ***metodo1()*** é o próprio método ***main()***. Assim, não faz sentido utilizar o ***throws*** e a saída é utilizar o ***try/catch***.



Erros Recuperáveis

- ☕ Um uso muito comum das ***checked exceptions*** é quando queremos tratar situações onde ainda pode ser possível recuperar o erro afim do programa ***tentar*** de novo. Ou seja, que o código cliente consiga ***recuperar***.
- ☕ Um exemplo seria o do “saldo insuficiente”. É esperado que o código que invoca o método ***sacar()*** consiga lidar com o erro e ***tentar*** sacar um valor cujo saldo seja suficiente.
- ☕ Mas ao pensar em um ***NullPointerException*** estamos falando de um objeto nulo, o que pode tornar impossível recuperar isso (depois de ocorrido). Além disso, é algo totalmente inesperado. Por isso ela é uma ***Unchecked Exception***. São exceções que, provavelmente, não tem como o cliente tentar de novo.



Checked vs Unchecked

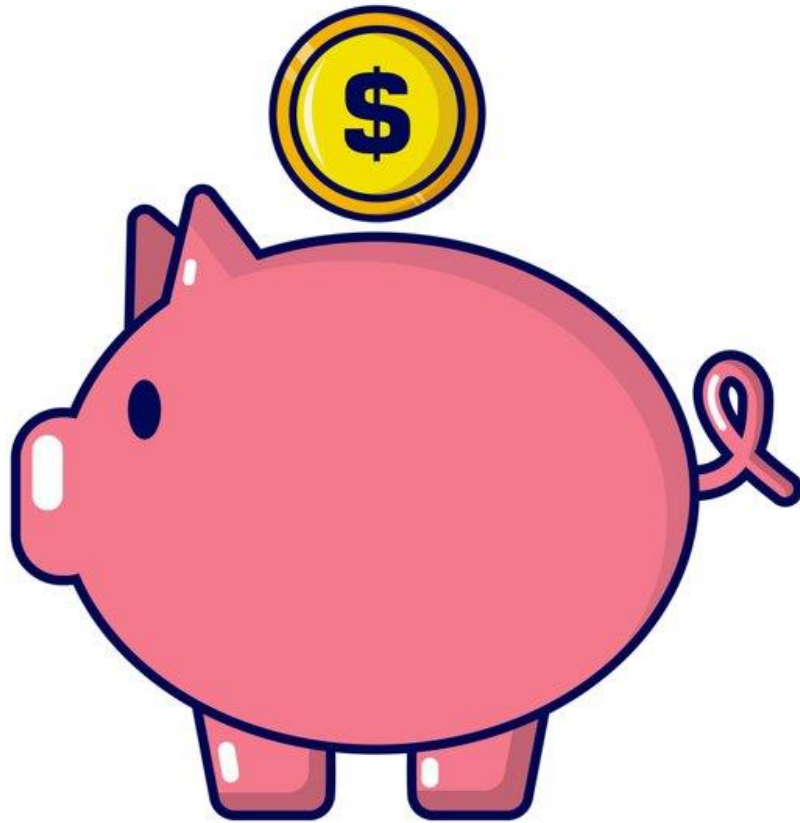
- ☕ Para concluir, as ***checked exception*** são para erros em ***tempo de compilação***, isto é, enquanto estamos programando. A própria IDE irá nos alertar.
- ☕ As ***unchecked exceptions*** são para erros em ***tempo de execução***. Ou seja, são erros lançados apenas durante a execução do programa. Por isso são chamadas também de ***Runtime Exceptions***.



Checked vs Unchecked

- ☕ Mas algumas exceções podem ser confusas. Considere a ***ArithmeticException*** que é lançada quando erros aritméticos ocorrem, como por exemplo, divisão por zero. Ela é uma ***Unchecked Exception***.
- ☕ Mas vimos que uma das características das ***Checked Exceptions*** é quando o cliente ***pode conseguir recuperar***. Nesse caso, passar um divisor que não seja zero.
- ☕ Porém, se ela fosse ***Checked Exception*** em todo lugar que existe uma divisão seríamos ***obrigados*** a colocar um ***try/catch*** ou ***throws***. Isso poderia deixar o código desnecessariamente complicado.
- ☕ Além disso, é possível que o divisor seja obtido através de outros cálculos, assim deixa de ser trivial simplesmente pedir para o usuário não entrar com um divisor 0 (zero)

☕ Vamos resgatar nossas classes Conta e Cliente





Saldo Insuficiente?

☕ Considere o método *sacar()*

```
public boolean sacar(double quantia) {  
  
    if(quantia < (saldo + limite)) {  
        this.saldo -= quantia;  
        return true;  
    }else {  
        System.out.println("Saldo Insuficiente.");  
        return false;  
    }  
}
```

☕ O problema é quem o *invoca* pode simplesmente não verificar o retorno.

☕ E isso pode trazer consequências desastrosas, como já visto.

- ☕ A primeira opção seria lançar um erro, pois mesmo que o código cliente não faça o teste, o programa será interrompido. Evitando um estado inconsistente. Podemos até deixar o método **void**.
- ☕ Podemos fazer isso lançando uma exceção através do **throw**.
- ☕ A palavra chave **throw**, lança uma **Exception** . Isto é bem diferente de **throws** que apenas avisa da possibilidade daquele método lançar um exceção.

```
public void sacar(double quantia) {  
    if(quantia < (saldo + limite)) {  
        this.saldo -= quantia;  
    }else {  
        throw new RuntimeException();  
    }  
}
```



☕ Se o saldo for insuficiente, o código dentro do ***else*** será executado. E lá estamos lançando uma exceção do tipo ***Runtime***. Logo ela não precisa ser tratada.

☕ Vamos testar esse código passando um saldo insuficiente.

```
public static void main(String[] args){
```

```
    Conta c1 = new Conta(100, 100);
```

```
    c1.sacar(300);
```

```
}
```

```
Exception in thread "main" java.lang.RuntimeException  
    at br.inatel.cdg.banco.Conta.sacar(Conta.java:20)  
    at br.inatel.cdg.Main.main(Main.java:12)
```

☕ Já tivemos um grande avanço. Pois, ao menos, o código foi interrompido e impediu um erro de ser propagado. Mas a mensagem ficou muito genérica.

☕ Vamos utilizar a ***IllegalArgumentException*** para sermos um pouco mais específico. Ela diz que algo foi passado como argumento e seu método não recebeu bem.

```
public void sacar(double quantia) {  
    if(quantia < (saldo + limite)) {  
        this.saldo -= quantia;  
    } else {  
        throw new IllegalArgumentException();  
    }  
}
```



```
Exception in thread "main" java.lang.IllegalArgumentException  
    at br.inatel.cdg.banco.Conta.sacar(Conta.java:20)  
    at br.inatel.cdg.Main.main(Main.java:12)
```

☕ Ao executarmos o código será lançada a ***IllegalArgumentException*** e, para o programador familiarizado, ele saberá que algum argumento não foi adequado.

☕ Dado que é uma ***Unchecked Exception*** não somos obrigados a trata-la. Mas podemos usar o ***try/catch*** como já conhecemos.

```
public static void main(String[] args){
```

```
    Conta c1 = new Conta(100, 100);
```

```
    try {  
        c1.sacar(300);  
    } catch (IllegalArgumentException e) {  
        System.out.println("Saldo Insuficiente");  
    }  
}
```





Criando nossa própria Exceção

☕ Para sermos mais específicos, podemos criar a nossa própria exceção. Basta herdar da classe ***RuntimeException*** ou de ***Exception*** (para ser do tipo ***checked***).

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

☕ No próprio construtor já podemos receber a mensagem de erro como parâmetro. Repare também que uma exceção nada mais é do que uma classe (que criamos). Como boa prática colocamos o nome ***Exception*** no final



Criando nossa própria Exceção

```
public void sacar(double quantia) {  
    if(quantia < (saldo + limite)) {  
        this.saldo -= quantia;  
    }else {  
        throw new SaldoInsuficienteException("Saldo Insuficiente");  
    }  
}
```

```
Conta c1 = new Conta(100, 100);
```

```
try {  
    c1.sacar(300);  
} catch (SaldoInsuficienteException e) {  
    System.out.println(e.getMessage());  
}
```



Criando nossa própria Exceção

☕ Se quisermos alterar para que seja uma ***CheckedException*** basta fazermos a nossa classe ***herdar*** de ***Exception***.

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

☕ Agora ***alguém*** terá de tratar o ***SaldoInsuficienteException***



Tratando Vários Erros

☕ É possível tratar vários erros por vez. O bloco **catch** pode ser repetido para cada exceção ou elas podem ser acomodadas no mesmo **catch**.

```
try {  
    FileInputStream f = new FileInputStream("config.txt");  
    c1.sacar(300);  
} catch (FileNotFoundException e) {  
    System.out.println("Exceção 1");  
} catch (SaldoInsuficienteException e) {  
    System.out.println(e.getMessage());  
}
```

☕ Observe o código dentro do **try**. Ele pode lançar duas exceções. Quando não encontrar o arquivo, ou com saldo insuficiente.

☕ Repare como a exceção que criamos pode ser integrada as demais exceções do próprio Java



Tratando Vários Erros

☕ Usando um único ***catch***. Com as exceções separadas por “|” (pipe – barra vertical)

```
try {  
    FileInputStream f = new FileInputStream("config.txt");  
    c1.sacar(300);  
} catch (FileNotFoundException | SaldoInsuficienteException e) {  
    System.out.println(e.getMessage());  
}
```



Finalmente!

- ☕ Os blocos ***try/catch*** podem conter uma terceira cláusula chamada ***finally*** que indica o que deve ser feito após o término do bloco ***try*** ou de um ***catch*** qualquer. Esse bloco ***sempre será executado***.
- ☕ O caso mais comum é o de liberar um recurso no ***finally***, como um arquivo ou conexão com banco de dados. Falaremos mais disso na parte de arquivos.

```
try {  
    c1.sacar(300);  
} catch (SaldoInsuficienteException e) {  
    System.out.println(e.getMessage());  
} finally {  
    System.out.println("Sempre serei executado!");  
}
```

Exercício - 2




- ☕ Utilizando o código do Exercício 1, faça com o que o método **sacar()** lance uma exceção ***SaldoInsuficienteException*** do tipo ***checked***.
- ☕ Faça o tratamento dessa exceção delegando para o método que invoca o método **sacar()**. Mas não faça esse método continuar delegando, ele deverá tratar.
- ☕ Apenas imprima uma mensagem de erro.
- ☕ Use o próprio construtor da exceção para receber a mensagem.



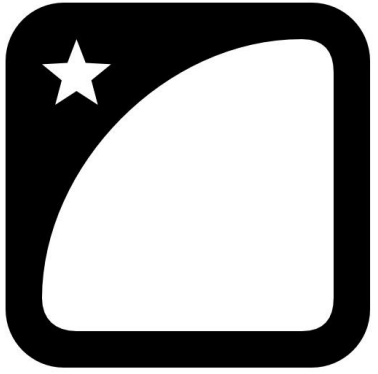
Vídeo Aula



 <https://youtu.be/oHI7ZJZyu7Y>

 Obs: (2020/1)

Material Complementar



☕ Capítulo 12 da apostila FJ-11

☕ Exceções e Controle de Erros

☕ Item 12.4

Resolução dos Exercícios



<https://github.com/phillima-inatel/C125>





C125/C206 – Programação Orientada a Objetos
com Java

Exceções e Controle de Erros

Prof. Phyllipe Lima
phyllipe@inatel.br

