

# EXCEÇÕES

Profª Paola Accioly

Material baseado nos slides de Thaís Alves Burity Rocha



# Introdução

**Agora que conhecemos os conceitos OO, vamos nos preocupar em desenvolver sistemas mais robustos...**

- ❑ Sistemas robustos devem...
- ❑ Se recuperar de falhas de forma eficiente
- ❑ Validar os dados
- ❑ Fornecer mensagens de erro apropriadas
- ❑ Garantir a consistência das operações
  - ▣ Retornar à um estado **seguro** e **estável**, permitindo que o usuário execute outros comandos
  - ▣ Prevenir a perda de dados



# Sistema bancário

```
public void debitar(double valor) {  
    saldo -= valor;  
}
```

- ❑ **Como evitar que o saldo fique negativo?**
- ❑ O compilador se preocupa com o uso correto da linguagem
- ❑ A lógica fica por conta do programador



# Solução 1: Desconsiderar a operação

```
public void debitar(double valor) {  
    if (valor <= saldo) {  
        saldo -= valor;  
    }  
}
```

## ❑ Problemas

- ▣ Não se sabe facilmente se a operação foi realizada
- ▣ Nenhuma informação é dada ao usuário do sistema



# Solução 2: Mostrar mensagem de erro

```
public void debitar(double valor) {  
    if (valor <= saldo){  
        saldo -= valor;  
    }else {  
        System.out.println("Saldo insuficiente!");  
    }  
}
```

## ❑ Problemas

- ❑ O erro não é sinalizado para o código que invocou “debitar”, somente para o usuário
- ❑ A notificação do erro está vinculada à interface com o usuário



# Solução 3: Retornar código de erro

```
public boolean debitar(double valor) {  
    boolean debitou = false;  
    if (valor <= saldo) {  
        saldo = saldo - valor;  
        debitou = true;  
    }  
    return debitou;  
}
```

**Prática antiga!**

## ❑ Problemas

- ❑ Dificulta a definição e o uso do método
- ❑ Se o método já retornasse um valor, o que seria feito?



# Exceções

- ❑ Significam uma **condição excepcional**, que altera o fluxo de execução normal do programa
- ❑ Surgem em **tempo de execução**
- ❑ Podem ser geradas por diferentes causas
  - ▣ Erro lógico, como saldo insuficiente
  - ▣ Limitações físicas, como memória insuficiente
  - ▣ Erro do programador, como índice inválido em vetor



# Exceções em POO

- ❑ São objetos que encapsulam informações relevantes sobre o erro ocorrido
  - ▣ Mensagem explicativa
  - ▣ Rastro da execução até a ocorrência do erro
  - ▣ Localização do erro (arquivo e linha)
- ❑ Quando um evento excepcional ocorre é dito que “uma exceção foi lançada”
- ❑ Há uma clara **separação** entre o código que **lança** a exceção e o código que **trata** a exceção





# Declaração e lançamento de exceção

- É necessário criar um **objeto de exceção**

```
public void debitar(double valor) throws Exception {  
    if (valor > saldo){  
        throw new Exception();  
    }  
    else{  
        saldo -= valor;  
    }  
}
```

Declaração

Lançamento

- **Exception** é o tipo mais genérico de exceção
- O tipo de exceção lançada tem que ser compatível com o tipo declarado



# Tratamento de exceções

- ❑ Transfere a execução de um programa para contornar a ocorrência de uma exceção, de maneira apropriada
- ❑ Logo, **exceções caracterizam situações recuperáveis**
- ❑ Uma forma de tratar exceções é usar blocos **try/catch** para capturá-las
- ❑ É possível usar um mesmo código para tratar uma grande variedade de exceções (**polimorfismo**)



# Bloco try/catch

Chamada  
de código  
“arriscado”

```
5 public class Main {  
6  
7     public static void main(String[] args) {  
8         System.out.println("INICIO");  
9         Conta conta = new Conta("12345", 100);  
10        try {  
11            conta.debitar(200);  
12        } catch (Exception e) {  
13            System.out.println("Saldo insuficiente!");  
14            e.printStackTrace();  
15        }  
16        System.out.println("FIM");  
17    }  
18 }
```

Só executa se  
uma exceção  
é lançada

```
1  
2 public class Conta {  
3  
4     private String numero;  
5     private double saldo;  
6  
7     public Conta(String numero, double saldo) {  
8         this.numero = numero;  
9         this.saldo = saldo;  
10    }  
11  
12    public void debitar(double valor) throws Exception {  
13        if(valor > saldo) throw new Exception();  
14        else saldo -= valor;  
15    }  
16 }
```

INICIO

Saldo insuficiente!

java.lang.Exception

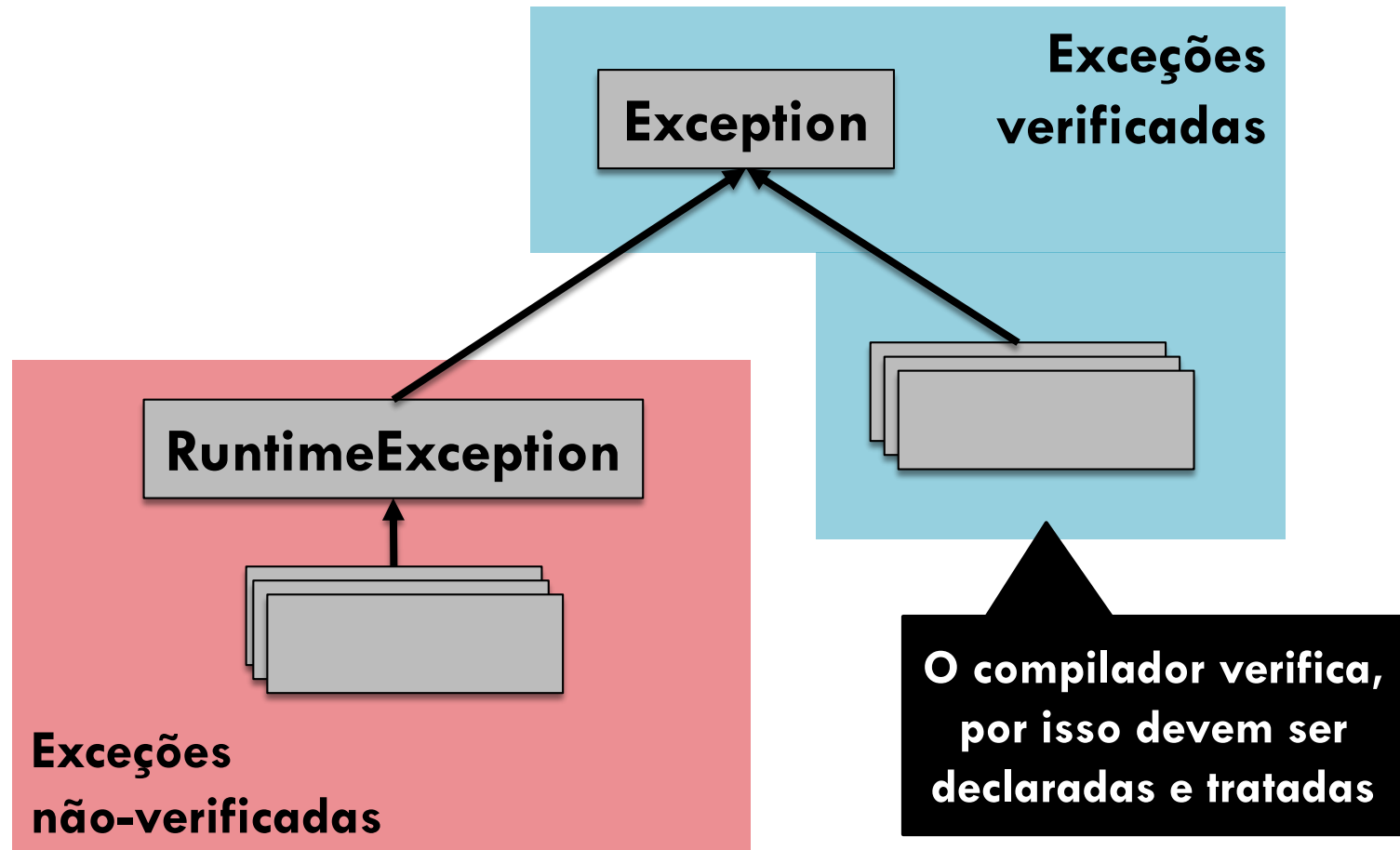
at Conta.debitar(Conta.java:13)

at Main.main(Main.java:11)

FIM



# Hierarquia de exceções



# Exceções verificadas

- ❑ Exception e subclasses que não são RuntimeException
- ❑ **Representam situações que não podem ser evitadas e que precisam de tratamento especial**
- ❑ Precisam ser declaradas para serem lançadas
  - ▣ Na assinatura do método
- ❑ Precisam ser tratadas
  - ▣ A chamada de debitar não compila sem try/catch
- ❑ A API define várias



# FileNotFoundException

```
public void exhibirLinha(String nomeArquivo) {
```

CHAMADA

```
obj.exibirLinha("arquivo.txt");
```

SAÍDA

arquivo.txt (O sistema não pode encontrar o arquivo especificado)

```
        scanner.close();  
    } catch (FileNotFoundException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

**Assinatura do construtor da classe Scanner:**

```
public Scanner(File source) throws FileNotFoundException
```




# Risco de problemas

- ❑ Operações de leitura e escrita de arquivos são críticas porque um mesmo arquivo pode ser requisitado por diferentes programas e o SO precisa gerenciar
  - ▣ Proteger o recurso para evitar inconsistências
- ❑ O mais seguro é liberar o arquivo após usá-lo
  - ▣ Chamada do método **close** de Scanner
- ❑ No código anterior, em caso de lançamento de exceção isso não é feito



# O desvio da execução impede...

```
public void exhibirLinha(String nomeArquivo){  
    File arquivo = new File(nomeArquivo);  
    try {  
        Scanner scanner = new Scanner(arquivo);  
        if(scanner.hasNextLine()){  
            String linha = scanner.nextLine();  
            System.out.println(linha);  
        }  
        scanner.close();  
    } catch (FileNotFoundException e) {  
        System.out.println(e.getMessage());  
    }  
}
```





# Bloco finally (opcional)

É executado  
sempre, havendo  
lançamento de  
exceção ou não

```
public void exibirLinha(String nome){  
    File arquivo = new File(nome);  
    Scanner scanner = null;  
    try {  
        scanner = new Scanner(arquivo);  
        if(scanner.hasNextLine()){  
            String linha = scanner.nextLine();  
            System.out.println(linha);  
        }  
    } catch (FileNotFoundException e) {  
        System.out.println(e.getMessage());  
    } finally {  
        if(scanner != null){  
            scanner.close();  
        }  
    }  
}
```



# Sintaxe alternativa

## ❑ Fechamento automático de recursos

```
public void exibirLinha(String nome) {  
    File arquivo = new File(nome);  
    try (Scanner scanner = new Scanner(arquivo)) {  
        if (scanner.hasNextLine()) {  
            String linha = scanner.nextLine();  
            System.out.println(linha);  
        }  
    } catch (FileNotFoundException e) {  
        e.printStackTrace();  
    }  
}
```

Se houver mais  
de um recurso,  
separar por ;



# Exceções não-verificadas

- ❑ RuntimeException e subclasses
- ❑ **Representam erros de programação, que podem ser evitados por princípio**
  - ▣ Se acontecem, devem mesmo quebrar a execução
- ❑ Não precisam ser declaradas nem tratadas
- ❑ Exemplos da API de Java
  - ▣ ArithmeticException: divisão por 0, dentre outros
  - ▣ ArrayIndexOutOfBoundsException: índice inválido em vetor
  - ▣ ClassCastException: conversão inválida de referência
  - ▣ NullPointerException: usar referência sem inicializá-la



# ArithmeticException

- ❑ A exceção não é explicitamente declarada nem lançada
- ❑ O código compila mesmo sem tratamento de exceção
- ❑ Quando lançada, a exceção interrompe a execução

```
1 public class DivisaoPorZero {  
2     public static int dividir (int a, int b){  
3         return a/b;  
4     }  
5 }
```

## INICIO

Exception in thread "main" java.lang.ArithmeticException: / by zero  
at DivisaoPorZero.dividir(DivisaoPorZero.java:3)  
at DivisaoPorZero.main(DivisaoPorZero.java:7)

```
9     }  
10 }
```



# Múltiplas exceções

- Um método pode lançar mais de uma exceção

```
public void metodo() throws Tipo1, Tipo2, Tipo3 { }
```

- É possível capturar todas com um único catch

```
catch (Exception ex) { ... }
```

- Ou definir múltiplos blocos catch

```
try{...}  
catch(Tipo1 ex){...}  
catch(Tipo2 ex){...}  
catch(Tipo3 | Tipo4 ex) {...}
```

Só um catch  
captura a exceção

- A JVM procura um catch adequado, em sequência
- Blocos devem ser ordenados do menos para o mais genérico



# Novos tipos de exceção

- ❑ Normalmente usamos as exceções da API de Java
- ❑ Mas pode ser útil definirmos novas exceções
  - ▣ Lidar com exceções específicas da aplicação
  - ▣ Criar classes para outros programadores usarem
- ❑ Mecanismo: Herdar de alguma exceção existente
- ❑ O mais comum
  - ▣ Exceção verificada herda de Exception
  - ▣ Exceção não-verificada herda de RuntimeException



# SaldoInsuficienteException

Boa prática: Sufixo Exception

```
public class SaldoInsuficienteException extends Exception {  
    private double saldo, valor;  
  
    public SaldoInsuficienteException(double s, double v) {  
        super("Saldo insuficiente!\nValor disponível: " +  
            s + "; valor desejado: "+v);  
        this.saldo = s;  
        this.valor = v;  
    }  
  
    public double getSaldo() { return saldo; }  
    public double getValor() { return valor; }  
}
```



# Usando SaldoInsuficienteException

```
public abstract class ContaAbstrata {  
    ...  
    public abstract void debitar(double valor)  
        throws SaldoInsuficienteException;  
}
```

Método abstrato pode  
declarar exceção

```
public class Conta extends ContaAbstrata {  
    ...  
    @Override  
    public void debitar(double valor) throws SaldoInsuficienteException {  
        if (valor > saldo){  
            throw new SaldoInsuficienteException(saldo, valor);  
        }  
        else{ saldo -= valor; }  
    }  
}
```






# Desvio de exceção

- A chamada de um método que lança exceção verificada deve ser tratada

```
public void transferir(ContaAbstrata conta, double v) {  
    this.debitar(v) ;   
    conta.creditar(v) ;  
}
```

- Se a chamada não é feita em um bloco try/catch, o método deve declarar a exceção

```
public void transferir(ContaAbstrata conta, double v)  
    throws SaldoInsuficienteException {   
    this.debitar(v) ;  
    conta.creditar(v) ;  
}
```



# Até quando é possível desviar?

- ❑ O desvio pode ocorrer inúmeras vezes, retardando o tratamento da exceção
- ❑ No máximo, a exceção chega no método main
- ❑ Como todo método, o main também pode declarar exceção
- ❑ Nesse caso, quem irá tratar a exceção?
  - ▣ A JVM, encerrando a execução do programa



# Exceções no sistema bancário

- ❑ Existem várias situações excepcionais a tratar
  - ▣ Não há saldo suficiente para debitar ou transferir
  - ▣ Tentar criar um cadastro de conta que já existe
  - ▣ Tentar acessar um cadastro de conta que não existe
  - ▣ Tentar criar um cadastro de cliente que já existe
  - ▣ Tentar acessar um cadastro de cliente que não existe
- ❑ Sugestão: Criar duas hierarquias de exceção, uma para **conta** e outra para **cliente**



# Boas práticas

- ❑ Não deixar o catch vazio
- ❑ Definir um catch muito genérico é conveniente, mas causa “perda” de informação

```
try {...} catch(Exception ex) {...}
```
- ❑ Informações específicas da exceção lançada só podem ser acessadas mediante casting
- ❑ Não lançar RuntimeException por conveniência
- ❑ Quando criar exceções, organize-as em hierarquias e pacotes

