# Introdução à Programação
Orientada a Objetos com Java

Exceções

Paulo Borba
Centro de Informática
Universidade Federal de Pernambuco

## Classe de Contas: Definição

```
class Conta {
  private String numero;
  private double saldo;
  /* ... */
  void debitar(double valor) {
    saldo = saldo - valor;
  }
}
```

*Como evitar débitos acima do limite permitido?*

## Desconsiderar Operação

```
class Conta {
  private String numero;
  private double saldo;
  /* ... */
  void debitar(double valor) {
    if (valor <= saldo)
      saldo = saldo - valor;
  }
}
```

## Desconsiderar Operação

- Problemas:
  - quem solicita a operação não tem como saber se ela foi realizada ou não
  - nenhuma informação é dada ao usuário do sistema

## Mostrar Mensagem de Erro

```
class Conta {
  static final String erro =
    "Saldo Insuficiente!";
  private String numero;
  private double saldo;
  /* ... */
  void debitar(double valor) {
    if (valor <= saldo)
      saldo = saldo - valor;
    else System.out.print(erro);
  }
}
```

## Mostrar Mensagem de Erro

- Problemas:
  - informação é dada ao usuário do sistema, mas nenhuma sinalização de erro é fornecida para métodos que invocam `debitar`
  - há uma forte dependência entre a classe `Conta` e sua interface com o usuário
  - não há uma separação clara entre código da camada de negócio e código da camada de interface com o usuário

## Retornar Código de Erro

```
class Conta {
  private String numero;
  private double saldo;
  /* ... */
  boolean debitar(double valor) {
    boolean r = false;
    if (valor <= saldo) {
      saldo = saldo - valor; r = true;
    } else r = false;
    return r;
  }
}
```

## Retornar Código de Erro

- Problemas:
  - dificulta a definição e o uso do método
  - métodos que invocam `debitar` têm que testar o resultado retornado para decidir o que deve ser feito
  - A dificuldade é maior para métodos que retornam valores:
    - Se `debitar` já retornasse um outro valor qualquer? O que teria que ser feito?

## Código de Erro: Problemas

```
class Conta {
  /* ... */
  boolean transferir(Conta c, double v) {
    boolean r = false;
    boolean b = this.debitar(v);
    if (b) {
      c.creditar(v); r = true;
    } else r = false;
    return r;
  }
}
```

## Código de Erro: Problemas

```
class CadastroContas {
  /* ... */
  int debitar(String n, double v) {
    int r = 0;
    Conta c = contas.procurar(n);
    if (c != null) {
      boolean b = c.debitar(v);
      if (b) r = 0; else r = 2;
    } else r = 1;
    return r;
  }
}
```

## Exceções

- Ao invés de códigos, teremos exceções...
- São objetos comuns, portanto têm que ter uma classe associada
- Classes representando exceções herdam e são subclasses de `Exception` (pré-definida)
- Define-se subclasses de `Exception` para
  - oferecer informações extras sobre a falha, ou
- distinguir os vários tipos de falhas

## Definindo Exceções

```
class SIException extends Exception {
  private double saldo;
  private String numero;
  SIException (double s, String n) {
    super ("Saldo Insuficiente!");
    saldo = s;
    numero = n;
  }
  double getSaldo() {return saldo;}
  /* ... */
}
```

## Definindo Métodos com Exceções

```
class Conta {
  /* ... */
  void debitar(double v)
    throws SIException {
    if (v <= saldo) saldo = saldo - v;
    else {
      SIException e;
      e = new SIException(saldo,numero);
      throw e;
    }
}
```

## Definindo e Levantando Exceções

- *Res metodo*(*Pars*) throws *E1,...,EN*
- Todos os tipos de exceções levantadas no corpo de um método devem ser declaradas na sua assinatura
- Levantando exceções: `throw` *obj-exceção*
- Fluxo de controle e exceção são passados para a chamada do código que contém o comando `throw`, e assim por diante...

## Usando Métodos com Exceções

```
class Conta {
  /* ... */
  void transferir(Conta c, double v)
    throws SIException {
    this.debitar(v);
    c.creditar(v);
  }
}
```

*Exceções levantadas indiretamente também devem ser declaradas!*

## Usando e Definindo Métodos com Exceções

```
class CadastroContas {
  /* ... */
  void debitar(String n, double v)
    throws SIException, CNEException {
    Conta c = contas.procurar(n);
    if (c != null) c.debito(v);
    else throw new CNEException(n);
  }
}
```

## Tratando Exceções

```
/* Antes...*/
try {
   /*...*/ banco.debitar("123-4",90.00);
   /*...*/
} catch (SIException e) {
   System.out.print(sie.getMessage());
   System.out.print(" Conta/saldo: ");
   System.out.print(e.getNumero()+
     " / " + e.getSaldo());
} catch (CNEException e) {/*...*/}
/* Depois...*/
```

## Tratando Exceções

- A execução do `try` termina assim que uma exceção é levantada
- O *primeiro* `catch` de um supertipo da exceção é executado e o fluxo de controle passa para o código seguinte ao último `catch`
- Se não houver nenhum `catch` compátivel, a exceção e o fluxo de controle são passados para a chamada do código com o `try/catch`

## Tratando Exceções: Forma Geral

```
try {...
} catch (E1 e1) {
   ...
}
   ...
} catch (En en) {
   ...
} finally {...}
```

## Tratando Exceções

- O bloco `finally` é sempre executado, seja após a terminação normal do `try`, após a execução de um `catch`, ou até mesmo quando não existe nenhum `catch` compatível
- Quando o `try` termina normalmente ou um `catch` é executado, o fluxo de controle é passado para o código seguindo o bloco `finally` (depois deste ser executado)

## Exceções no Cadastro de Contas

```
class CadastroContas {/* ... */
  void cadastrar(Conta c) throws
    CEException,IllegalArgumentException {
    if (c != null) {
      String n = c.getNumero();
      if (!contas.existe(n))
        contas.inserir(c);
      else throw new CEException(n);
    } else throw new
            IllegalArgumentException();
  }
```

## Exceções no Cadastro de Contas

```
   void debitar(String n, double v)
     throws SIException, CNEException {
     Conta c = contas.procurar(n);
     c.debitar(v);
   }
}
```

## Exceções no Conjunto de Contas

```
class ConjuntoContas {/*...*/
  static final IllegalArgumentException
    e = new IllegalArgumentException();
  void inserir(Conta c)
    throws IllegalArgumentException {
    if (c != null) {
      contas[indice] = c;
      indice = indice + 1;
    } else throw e;
  }
```

## Exceções no Conjunto de Contas

```
Conta procurar(String n)
  throws CNEException {
    Conta c = null;
    int i = this.procurarIndice(n);
    if (i == indice)
      throw new CNEException(n);
    else c = contas[i];
    return c;
}
```