Programação Orientada a Objetos

UNIDADE 4

GUIA DE ESTUDO

UNIDADE 4

PROGRAMAÇÃO ORIENTADA A OBJETOS

PARA INÍCIO DE CONVERSA

Olá caro(a) estudante, vamos dar início a unidade 4 do nosso curso de Programação Orientada a Objetos.

Nesse guia de estudo da respectiva disciplina, vamos falar principalmente sobre relacionamento entre classes, polimorfismo e exceções. Veremos também como funciona o sistema de ligação dinâmica.

Espero que tenha contribuído para este momento de sua caminhada acadêmica.

Preparado? Então vamos lá.

RELACIONAMENTO ENTRE CLASSES

Meu caro(a) estudante, em orientação a objetos, vimos que a reutilização de código é bastante sofisticada. Temos, por exemplo, o mecanismo de herança que vimos anteriormente, entre outras formas de reutilizar o código. Esse mecanismo visto é uma das formas das classes se relacionarem.

VOCÊ SABIA?

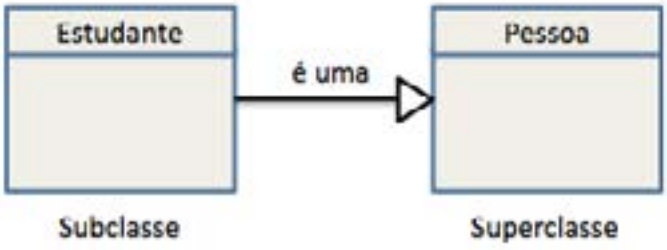
Você sabia que existem outras formas de relacionamento entre as classes de objetos? Pois é, assim como a herança pode facilitar a criação de novas classes, podemos pensar na construção de novas classes a partir de outras que possam ser usadas dentro dessa nova, por exemplo. Mas, então qual o interesse nas relações entre as classes? Basicamente essas relações facilitam a reutilização de códigos.

Com isso temos uma redução do esforço de programação, reduzindo bastante os custos de produção de software. Sem dúvidas uma das principais vantagens da programação orientada a objetos. Na prática, construímos novas classes a partir de classes existentes. Então, objetos de uma classe usam serviços fornecidos por objetos da outra ou contém parte do código da outra.

FIQUE ATENTO!

Fique atento(a), pois veremos aqui os tipos mais comuns de relacionamento entre classes, como por exemplo: Herança, Dependência, Agregação, Composição.

A relação de herança, que já vimos anteriormente, também é conhecida como relação de generalização. Essa relação onde uma classe é uma generalização (especialização) de outra classe. É uma relação do tipo "é um" ou "é uma", ou seja, temos uma classe que pode ser considerada como uma generalização de outra. Por exemplo, "Estudante é uma Pessoa":



Notação UML: Relação de Generalização (Herança)

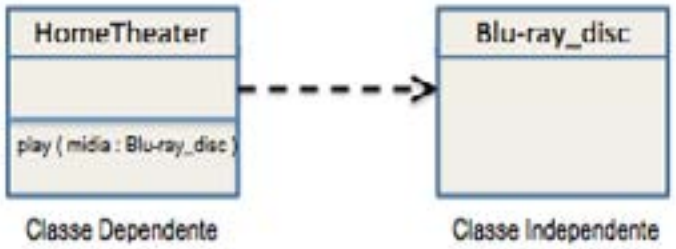
JAVA:

```
public class Pessoa {
. . .
}

public class Estudante extends Pessoa {
. . .
}
```

Meu caro(a), na relação de dependência, uma classe depende de "outra classe", ou seja, usa objetos de "outra classe" (que é uma classe independente). A classe, normalmente, usa temporariamente objetos da classe da outra classe (independente). Essas classes independentes são usadas para declaração de variáveis locais e/ou para declaração de parâmetros de métodos. Sua existência é temporária, só durante a execução dos métodos que as usam. Durante um relacionamento de dependência, quando uma alteração na classe independente é feita, afeta a outra classe. O inverso não é verdade. Nesse contexto, diz-se que uma classe utiliza a outra. É uma relação do tipo "precisa de".

EXEMPLO

Por exemplo: Um classe HomeTheater tem uma relação de dependência com a classe Blu-ray_disc, uma vez que no parâmetro do método play de HomeTheater ele "precisa de" um Blu-ray_disc para ser passado.



Notação UML: Relação de Dependência

JAVA:

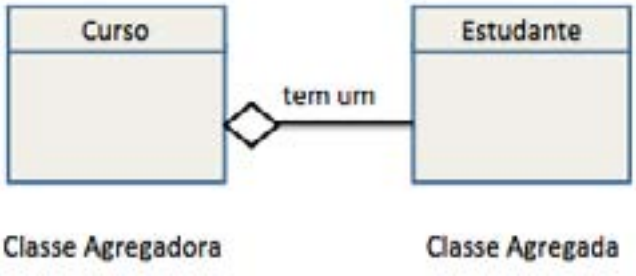
```
public class Blu-ray_disc {
. . .
}

public class HomeTheater {
. . .
public void play ( Blu-ray_disc midia)
}
```

A relação de agregação o objeto de uma classe contém um objeto de outra classe. Ou seja, a classe (agregadora) agrega outra classe (agregada). Nessa relação, o objeto agregado tem existência independente do objeto agregador. Em outras palavras, objeto agregado pode existir após eliminação do objeto agregador e o objeto agregado não pertence ao objeto agregador.

Classe Agregada faz parte da estrutura da Classe Agregadora. Nesse caso o objeto agregado é parte do objeto agregador, é usado como um atributo desse objeto agregador. A classe agregada é usada na declaração de variável de instância (atributo). É uma relação conhecida como do tipo "tem um", ou "tem uma", ou "é parte de".

EXEMPLO

Um Curso tem Estudante.



Notação UML: Relação de Composição

JAVA:

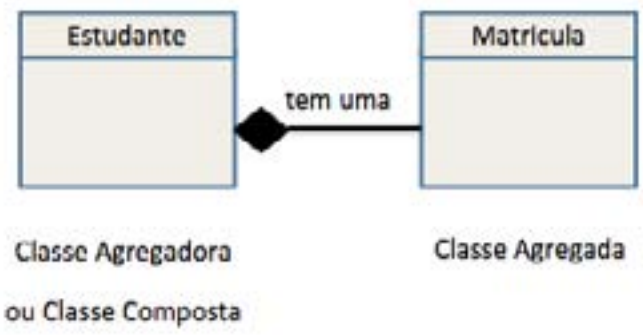
```
public class Curso {
private Estudante [ ] estudantes ;
. . .
}

public class Estudante extends Pessoa {
. . .
}
```

Assim como na relação de agregação, a relação de composição o objeto de uma classe também contém um objeto de outra classe. Nesse caso, a classe (agregadora) agrega outra classe (agregada), porém, nessa relação, o objeto agregado tem existência dependente do objeto agregador. Em outras palavras, objeto agregado não pode existir após eliminação do objeto agregador e o objeto agregado pertence ao objeto agregador. Classe Agregada faz parte da estrutura da Classe Agregadora ou Composta. Da mesma forma que na relação de agregação, a classe agregada é usada na declaração de variável de instância (atributo). Também é uma relação conhecida como do tipo "tem um", ou "tem uma", ou "é parte de".

EXEMPLO

Exemplo: Um Estudante tem uma Matrícula.

Exemplo: Um Curso tem Estudante.



Notação UML: Relação de Composição

JAVA:

```
public class Estudante extends Pessoa {
private Matricula MatriculaEstudante ;
. . .
}

public class Matricula {
. . .
}
```

POLIMORFISMO

PALAVRAS DO PROFESSOR

Querido(a), veremos agora mais um pilar da programação orientada a objetos: o polimorfismo. Até agora, vimos como obter componentes autossuficientes utilizando encapsulamento e como fazer reuso através da herança. O fato é que o software está sempre mudando. Como costumamos dizer, "A única coisa certa é que vai ter mudança!". A necessidade que temos hoje, é que o software seja capaz de se adaptar para satisfazer necessidades futuras. Um software moderno deve se adaptar a requisitos futuros sem alteração. Com isso facilita a realização de mudanças. Isso pode ser obtido através do uso do polimorfismo. Polimorfismo é um conceito abstrato onde algo pode responder de formas diferentes para uma mesma situação.

VOCÊ SABIA?

Você sabia que o polimorfismo apoia-se nos conceitos de encapsulamento e herança, permitindo a construção de programas flexíveis para se adaptar a requisitos futuros? Pois é, o Mecanismo que adiciona grande flexibilidade a programação orientada a objetos. Um mesmo nome pode representar diferentes comportamentos, selecionados através de um mecanismo automático. Permite a uma mesma ação ser implementada de diversas maneiras em objetos diferentes. Ou seja, a mesma invocação pode produzir "muitas formas" de resultados. O domínio dessa técnica é fundamental para se aproveitar ao máximo as vantagens da programação orientada a objetos.

Veremos agora como é feito a implementação usando polimorfismo. Utilizamos herança para implementar polimorfismo. Quando um programa chama um método através de uma variável de superclasse (atributo de superclasse), a versão correta de subclasse do método é chamada com base no tipo da referência armazenada na variável da superclasse. Com o polimorfismo, o mesmo nome e assinatura de método podem ser utilizados para fazer com que ações diferentes ocorram, dependendo do tipo de objeto em que o método é invocado.

Podemos ter várias vantagens com o uso do polimorfismo. Com polimorfismo é permitido que o programador trate de generalidades deixando o ambiente de tempo de execução tratar das especificidades. Os programadores podem codificar objetos a se comportarem de maneiras apropriadas para esses objetos,

sem nem mesmo conhecer os tipos dos objetos (apenas se os objetos pertencerem à mesma hierarquia de herança). O polimorfismo promove extensibilidade: O programa que invoca o comportamento polimórfico é independente dos tipos de objeto para os quais as mensagens são enviadas.

PRATICANDO

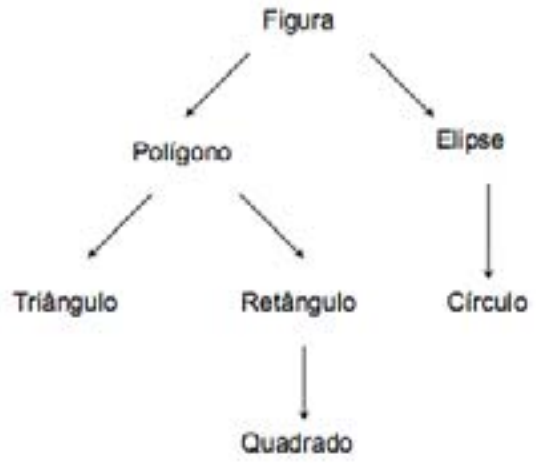
Meu caro(a), para melhor ilustrar o polimorfismo, vamos ver alguns exemplos.

A ação abrir pode ser aplicada a diferentes objetos como: Porta; Conta Bancária; Caixa; Janela.

Podemos codificar a ação "abrir" para ser aplicada a diferentes objetos. Cada objeto pode implementar a ação abrir de uma maneira diferente. A mensagem abrir pode então ser enviada a diferentes objetos, que a realizarão da forma que for mais adequada a cada um deles. Essa realização é feita usando polimorfismo.

O polimorfismo é resolvido dinamicamente (em tempo de execução, através de ligações dinâmicas). Ocorre quando uma classe possui um método com o mesmo nome e assinatura (número, tipo e ordem dos parâmetros) de um método da superclasse. Quando isso acontece, dizemos que a subclasse sobrepõe o método da superclasse.

Considere agora a hierarquia de classes envolvendo herança de figuras geométricas que foram vistas quando falamos de herança:



Para exemplificar o polimorfismo, vamos implementar parte da hierarquia de classe acima. Usaremos as classes: Figura ® Elipse ® Circulo.

Depois vamos criar uma classe (ExemploPolimorfismoFiguras) que será usada para testar o comportamento polimórfico dos objetos criados nesse teste.

Figura.java

```
1  public class Figura {
2          private double x , y ;     // Coordenadas da figura no espaço
3
4          // Construtor da classe Figura sem parâmetros
5          // Definido explicitamente para substituir o
6          // construto vazio padrão do compilador JAVA
7          public Figura ( ) {
```

```
8                       // Vazio, apenas para substituir o construtor padrão !
9               } // Fim do construtor
10
                // Construtor da classe Figura com dois parâmetros, coordena-
11  da X e coordenada Y.
12              public Figura ( double x , double y ) {
13                      this.x = x ;
14                      this.y = y ;
15              } // Fim do construtor
16
17              // Método de acesso para a coordenada X
18              public double getX ( ) {
19                      return x ;
20              } // fim do método getX
21
22              // Método de acesso para a coordenada X
23              public void setX ( double x ) {
24                      this.x = x ;
25              } // fim do método setX
26
27              // Método de acesso para a coordenada Y
28              public double getY ( ) {
29                      return y ;
30              } // fim do método getY
31
32              // Método de acesso para a coordenada Y
33              public void setY ( double y ) {
34                      this.y = y ;
35              }
36
                // Método que exibe na tela de console as informações da figu-
37  ra
38              public void exibe ( ) {
39                      System.out.println ( " --==<< Figura >>==-- " ) ;
                        System.out.println ( " [ " + getX ( ) + ", " + getY ( ) + " ] ")
40  ;
                        System.out.println ( " --==<<========>>==-- \n " )
41  ;
42              } // fim do método exibe
43
44              // Outros métodos . . .
45
46  }  // Fim da classe Figura
```

Elipse.java

```
1  // Classe Elipse herdeira de Figura
2  public class Elipse extends Figura {
3          private double rx , ry ; // Raio X e Raio Y da elipse
4
5          // Construtor da classe Elipse sem parâmetros
6          // Definido explicitamente para substituir o
7          // construto vazio padrão do compilador JAVA
8          public Elipse ( ) {
9                  // Vazio, apenas para substituir o construtor padrão !
10         } // Fim do construtor
11
           // Construtor da classe Elipse com quatro parâmetros: coordenada X , coordenada Y
12 ,
13         // Raio X e Raio Y.
14         public Elipse ( double x , double y , double rx , double ry ) {
15                 super ( x , y ) ;
16                 this.setRx ( rx ) ;
17                 this.setRy ( ry ) ;
18         } // Fim do construtor
19
20         // Método de acesso para o raio X
21         public double getRx ( ) {
22                 return rx ;
23         } // fim do método getRx
24
25         // Método de acesso para o raio X
26         public void setRx ( double rx ) {
27                 if ( rx >= 0 ) {
28                         this.rx = rx ;
29                 }
30                 else {
31                         System.out.println ( " Raio não pode ser negativo ! " ) ;
32                 }
33         } // fim do método setRx
34
35         // Método de acesso para o raio Y
36         public double getRy ( ) {
37                 return ry ;
38         } // fim do método getRy
39
40         // Método de acesso para o raio Y
41         public void setRy ( double ry ) {
```

```
42                  if ( ry >= 0 ) {
43                          this.ry = ry ;
44                  }
45                  else {
46                          System.out.println ( " Raio não pode ser negativo ! " ) ;
47                  }
48          } // fim do método setRy
49
50          // Método que exibe na tela de console as informações da elipse
51          public void exibe ( ) {
52                  System.out.println ( " --==<< Elipse >>==-- " ) ;
53                  System.out.println ( " [ " + getX ( ) + " , " + getY ( ) + " ] " ) ;
54                  System.out.println ( " Raios: [ " + getRx ( ) + " , " + getRy ( ) + " ] " );
55                  System.out.println ( " --==<<========>>==-- \n " ) ;
56          } // fim do método exibe
57
58          // Outros métodos . . .
59
60  }  // Fim da classe Elipse
```

Circulo.java

```
1   // Classe Circulo herdeira de Elipse
2   public class Circulo extends Elipse{
3
4           private double raio;      // Raio do Círculo
5
6           // Construtor da classe Circulo sem parâmetros
7           // Definido explicitamente para substituir o
8           // construto vazio padrão do compilador JAVA
9           public Circulo ( ) {
10                  // Vazio, apenas para substituir o construtor padrão !
11          } // Fim do construtor
12
            // Construtor da classe Circulo com três parâmetros: coordenada X , coordenada Y
13  e Raio.
14          public Circulo ( double x , double y , double raio ) {
15                  super ( x , y , raio , raio ) ;
16                  this.setRaio ( raio ) ;
17          } // Fim do construtor
18
19          // Método de acesso para o raio
20          public double getRaio() {
```

```
21                  return raio;
22          } // fim do método getRaio
23
24          // Método de acesso para o raio
25          public void setRaio ( double raio ) {
26                  if ( raio >= 0 ) {
27                          this.raio = raio ;
28                  }
29                  else {
30                          System.out.println ( " Raio não pode ser negativo ! " ) ;
31                  }
32          } // fim do método setRaio
33
34          // Método que exibe na tela de console as informações do círculo
35          public void exibe ( ) {
36                  System.out.println ( " --==<< Círculo >>==-- " ) ;
37                  System.out.println ( " [ " + getX ( ) + " , " + getY ( ) + " ] " ) ;
38                  System.out.println ( " Raio: [ " + getRaio ( ) + " ] " ) ;
39                  System.out.println ( " --==<<==========>>==-- \n " ) ;
40          } // fim do método exibe
41
42          // Outros métodos . . .
43
44  }  // Fim da classe Circulo
```

ExemploPolimorfismoFiguras.java

```
1   // Classe para testar o exemplo de polimorfismo
2   public class ExemploPolimorfismoFiguras {
3
4           public static void main ( String [ ]  args ) {
5
6                   // Declara três referências para objetos da classe Figura
7                   Figura f = new Figura ( ) ;                 // Instancia como Figura
8                   Figura e = new Elipse ( 2 , 3 , 4 , 2 ) ;   // Instancia como Elipse
9                   Figura c = new Circulo( 5 , 8 , 4 ) ;       // Instancia como Circulo
10
11                  // Chamada ou método exibe () para os três objetos
12                  f.exibe();          // exibe da classe Figura
13                  e.exibe();          // exibe da classe Elipse
14                  c.exibe();          // exibe da classe Circulo
15
16          } // fim do método main
```

```
17
18  } // Fim da classe ExemploPolimorfismoFiguras
```

Saída do console:

```
--==<< Figura >>==--
 [ 0.0, 0.0 ]
 --==<<========>>==--

 --==<< Elipse >>==--
 [ 2.0 , 3.0 ]
 Raios: [ 4.0 , 2.0 ]
 --==<<========>>==--

 --==<< Círculo >>==--
 [ 5.0 , 8.0 ]
 Raio: [ 4.0 ]
 --==<<=========>>==--
```

No método main da classe ExemploPolimorfismoFiguras nós temos um teste onde três referências são declaradas para a classe Figura que é a superclasse que inicia a nossa hierarquia de classes. Temos então nas linhas 7, 8 e 9 as declarações como referências para objetos do tipo Figura. Também nessas linhas cada referência é instanciada usando construtores da própria classe Figura, como outros construtores de classes derivadas de Figura (Elipse e Circulo). Nas linhas 12, 13 e 14 são feitas chamadas ao método exibe( ) das três referências declaradas como Figura.

Porém a execução desse método será feita de acordo com os objetos que foram criados, se comportando da forma adequada para cada chamada do exibe( ). Como pode ser visto na saída do console depois da execução, cada um dos três objetos se exibe da forma correta, chamando o método exibe( ) adequado para cada um.

PARA RESUMIR

Caro(a) aluno(a), podemos então constatar com esse exemplo que, através do polimorfismo, o programa se adaptou automaticamente para executar de forma correta a operação de exibir os dados de cada objeto.

LIGAÇÃO DINÂMICA (DYNAMIC BINDING)

Veremos agora um mecanismo que foi utilizado para viabilizar o polimorfismo. A ligação dinâmica acontece quando associamos um método a um objeto, mas essa realmente só é feita tardiamente durante o período da execução (em tempo de execução). A ligação dinâmica (dynamic binding) também é conhecida com ligação tardia (late binding). Chamamos de ligação tardia, pois é feita depois da compilação do código, deixando para fazer a ligação quando for executar (em tempo de execução).

Essa ligação geralmente acontece quando um método pertencente a uma classe derivada (subclasse) é chamado usando uma referência para sua classe base (superclasse). Nesse caso o código do método da classe derivada (subclasse) será invocado ao invés do método da classe base (superclasse). Isso permite a substituição da implementação mais genérica por uma implementação em particular codificada na classe

derivada (subclasse). A ligação da referência para objeto está sendo feita de forma tardia, ou seja, em tempo de execução. Durante a execução é feita a ligação com o método certo, de acordo com o tipo da classe que foi usada para instanciar o objeto.

Podemos ver como isso acontece no exemplo abaixo:

Pessoa.java

```
1  public class Pessoa {
2
3          private String nome;      // Nome completo da pessoa
4          private int idade;        // Idade da pessoa
           private char sexo;        // Sexo da pessoa: 'M' para masculi-
5  no e 'F' para feminino
           private double altura;    // Altura da pessoa: valro real ex-
6  presso em metros
7
8          // Construtor da classe Pessoa com quatro parâmetros
9          public Pessoa ( String n , int i , char s , double a ) {
10                 nome = n ;
11                 setIdade ( i ) ;
12                 sexo = s ;
13                 setAltura( a ) ;
14         } // Fim do construtor
15
16         // Construtor da classe Pessoa sem parâmetros
17         // Definido explicitamente para substituir o
18         // construto vazio padrão do compilador JAVA
19         public Pessoa ( ) {
20                 // Vazio, apenas para subtituir o construtor padrão !
21         }
22
           // Construtor da classe Pessoa com dois parâmetros, apenas
23 o nome e a idade.
24         public Pessoa ( String nome , int idade ) {
25                 this.nome = nome ;
26                 this.setIdade( idade ) ;
27         } // Fim do construtor
28
29         // Métodos para acesso aos atributos (campos)
30
31         // Método de acesso para o campo nome
32         public String getNome ( ) {
33                 return nome ;
34         } // fim do método getNome
35
```

```
36          // Método de acesso para o campo nome
37          public void setNome ( String nome ) {
38                  this.nome = nome ;
39          } // fim do método setNome
40
41          // Método de acesso para o campo idade
42          public int getIdade ( ) {
43                  return idade ;
44          } // fim do método getIdade
45
46          // Método de acesso para o campo idade
47          public void setIdade ( int idade ) {
48                  if ( idade >= 0 ) {
49                          this.idade = idade ;
50                  }
51                  else {
52                          System.out.println ( " Idade inválida ! " ) ;
53                  }
54          } // fim do método setIdade
55
56          // Método de acesso para o campo sexo
57          public char getSexo ( ) {
58                  return sexo ;
59          } // fim do método getSexo
60
61          // Método de acesso para o campo sexo
62          public void setSexo ( char sexo ) {
63                  this.sexo = sexo ;
64          } // fim do método setSexo
65
66          // Método de acesso para o campo altura
67          public double getAltura ( ) {
68                  return altura ;
69          } // fim do método getAltura
70
71          // Método de acesso para o campo altura
72          public void setAltura ( double altura ) {
73                  if ( altura > 0 ) {
74                          this.altura = altura ;
75                  }
76                  else {
77                          System.out.println ( " Altura inválida ! " ) ;
78                  }
```

```
79          } // fim do método setAltura
80
81
            // Método que exibe na tela de console as informações da
82  pessoa
83          public void exibe ( ) {
                    System.out.println ( " ---- === < Pessoa > === ---- " )
84  ;
85                  System.out.println ( " Nome   : " + nome ) ;
86                  System.out.println ( " Idade  : " + idade ) ;
87                  System.out.println ( " Sexo   : " + sexo ) ;
88                  System.out.println ( " Altura : " + altura + " \n \n " ) ;
89          } // fim do método exibe
90
91
92          // Método que soma mais um ano na idade da pessoa
93          public void aniversario ( ) {
94                  idade = idade + 1 ;
95          } // fim do método aniversario
96
97          // Outros métodos . . .
98
99  } // Fim da classe Pessoa
```

Estudante.java

```
1   public class Estudante extends Pessoa{
2
3           private int matricula;
4
5           // Construtor da classe Estudante
6           public Estudante ( int m ) {
7                   setMatricula ( m ) ;
8           } // Fim do construtor
9
10          // Método de acesso para o campo matricula
11          public int getMatricula() {
12                  return matricula;
13          }// fim do método getMatricula
14
15          // Método de acesso para o campo matricula
16          public void setMatricula(int matricula) {
17                  this.matricula = matricula;
18          } // fim do método setMatricula
```

```
19
20          // Método que exibe na tela de console as informações da Estudante
21          public void exibe ( ) {
22                  System.out.println ( " ---- === < Estudante > === ---- " ) ;
23                  System.out.println ( " Nome      : " + getNome ( ) ) ;
24                  System.out.println ( " Idade     : " + getIdade ( ) ) ;
25                  System.out.println ( " Sexo      : " + getSexo ( ) ) ;
26                  System.out.println ( " Altura    : " + getAltura ( ) ) ;
27                  System.out.println ( " Matrícula : " + getMatricula ( ) + " \n \n " ) ;
28          } // fim do método exibe
29
30          // Outros métodos . . .
31
32  } // Fim da classe Estudante
```

ExemploLigacaoDinamica.java

```
1   public class ExemploLigacaoDinamica {
2
3           public static void main ( String [ ]  args ) {
4
5                   // Declara duas referências para objetos da classe Pessoa
                    Pessoa a = new Pessoa ( "Fulano" , 19 , 'M' , 1.65 ) ;      // Instancia como
6   Pessoa
                    Pessoa b = new Estudante ( 2016100443 ) ;                   // Instancia como
7   Estudante
8
9                   b.setNome ( "Beltrana" ) ;
10                  b.setIdade ( 21 ) ;
11                  b.setSexo ( 'F' ) ;
12                  b.setAltura( 1.70 )         ;
13                  // Chamada ao  método exibe  ( ) para os dois objetos
14                  a.exibe();          // exibe da classe Pessoa
15                  b.exibe();          // exibe da classe Estudante
16
17          } // fim do método main
18
19  }  // Fim da classe ExemploLigacaoDinamica
```

Saída do console:

```
---- === < Pessoa > === ----
 Nome    : Fulano
 Idade   : 19
 Sexo    : M
 Altura  : 1.65


 ---- === < Estudante > === ----
 Nome       : Beltrana
 Idade      : 21
 Sexo       : F
 Altura     : 1.7
 Matrícula  : 2016100443
```

Vamos lá, nesse exemplo vimos duas classes ligadas pela relação de herança. A classe Pessoa é a super-classe de Estudante. Depois foi criada uma classe para testar o exemplo de ligação dinâmica, onde foram declaradas duas referências para objetos do tipo Pessoa. Note que a classe b foi instanciada como um Estudante e que a ligação de seus métodos será feita em tempo de execução. Quando é feito a chamada do método exibe nas linhas 14 e 15, são feitas as ligações com os códigos corretos. Ou seja, o método exibe que é chamado na linha 15 pela referência b, é o código de Estudante mesmo que essa referência tenha sido declarada como Pessoa. Essa ligação como o método exibe não foi feita durante a declaração e sim na hora que foi chamado na linha 15. Nesse momento é definido que o código correto seria o de Estudante, pois esse objeto foi instanciado como um Estudante. Vimos com esse exemplo como funciona a ligação dinâmica (ou ligação tardia).

Pois então meu caro(a), alguma dúvida até agora? Já conversou com seu tutor? Lembre-se ele aguarda sua sinalização para lhe auxiliar no que for preciso.

TRATAMENTO DE EXCEÇÕES

DICA

Como guia para seus estudos, recomendo que você leia o capítulo de tratamento de exceções do livro do DEITEL (DEITEL. Java Como Programar. Ed. Pearson Brasil, 2005.), capítulo 13. Eu espero que goste.

Meu caro(a), exceções são eventos inesperados que ocorrem durante a execução de um programa. Essas podem ser causadas também por erros de programação ou por uso incorreto do software. Para se prevenir disso é necessário fazer um tratamento de exceções. O tratamento de exceções é uma forma de resolver exceções que poderiam ocorrer durante a execução, deixando que o programa continue ou termine elegantemente.

O tratamento de exceções permite que os programadores criem programas mais robustos e tolerantes a falhas.

Temos, de modo geral, os seguintes tipos de erros:

- Erros de programação;
- Erros de execução ou de ambiente;
- Erros de sistema.

Os **Erros de programação** são causados pelo programador, e por ele deve ser corrigido. Se o tratamento desse tipo de erro utilizar exceções é porque ocorreu uma falha de projeto.

Exemplo: Tentativa de acessar elementos de um array fora dos limites válidos desse vetor.

Já os **Erros de execução ou de ambiente** são causados por uso indevido do programa por parte do usuário ou por falhas no ambiente de execução. Apesar de não ser causado pelo programador, é dever do mesmo antecipar a ocorrência desses problemas, escrevendo código para tratar esses erros.

Exemplo: Tentativa de abrir arquivo inexistente, queda da rede, etc.

E os **Erros de sistema** são erros graves, normalmente relacionados ao hardware ou a JVM (Java Virtual Machine – máquina virtual Java). Não há praticamente nada que possa ser feito pelo programador para antecipar e prevenir esses erros.

Exemplo: Falta de memória, bug da JVM, etc.

Em JAVA as exceções são tratadas com uma família de classes cujo objetivo é modelar os tipos de erro que podem ocorrer durante a execução de um programa. A classe Exception define uma família de classes relacionadas a erros menos severos em Java e que devem ser tratados.

EXEMPLO

Temos abaixo alguns exemplos de exceções em JAVA:

ArrayIndexOutOfBoundsException — ocorre quando é feita uma tentativa de acessar um elemento fora dos limites do array (depois do final de um array ou um índice negativo, por exemplo).

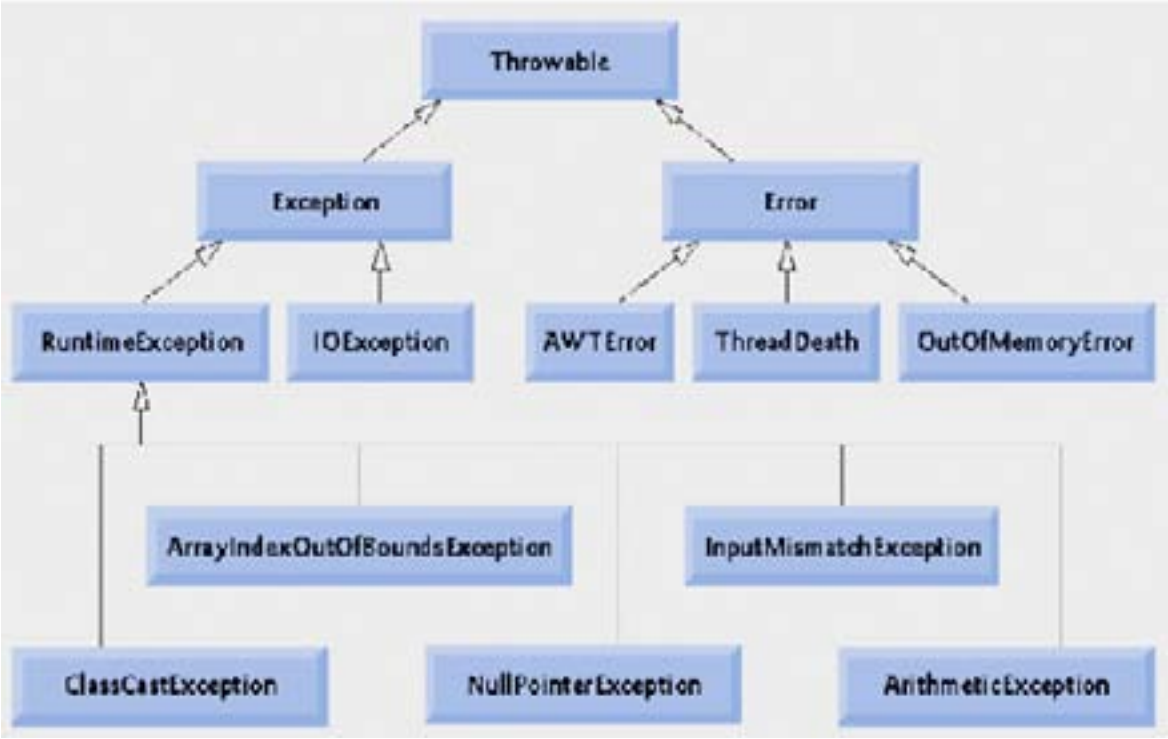
ClassCastException — ocorre quando há uma tentativa de fazer uma coerção em um objeto que não tem um relacionamento "é um" com o tipo especificado no operador de coerção.

NullPointerException — ocorre quando uma referência null é utilizada onde um objeto é esperado. Ou seja, a referência do objeto não foi instanciada, ainda não referencia um objeto.

GUARDE ESSA IDEIA!

Fique atento pois, quando misturamos a lógica do programa com a lógica do tratamento de erros isso pode tornar os programas difíceis de ler, modificar, manter e depurar. Com o uso de tratamento de exceções os programadores podem remover código de tratamento de erro da "linha principal" de execução do programa. Com isso torna os programas mais claros e facilita o processo de modificação do código.

Vemos abaixo a hierarquia de classes de exceções do JAVA. Essa hierarquia começa com a superclasse Throwable e segue com as classes herdeiras.



Parte da hierarquia de herança da classe Throwable.

Fonte: DEITEL. Java Como Programar. Ed. Pearson Brasil, 2005. Capítulo 13. Figura 13.3.

Veremos em seguida uma adaptação das classes **DivideByZeroNoExceptionHandling** e também **DivideByZeroWithExceptionHandling** do livro do DEITEL (DEITEL. **Java Como Programar.** Ed. Pearson Brasil, 2005. Capítulo 13. Figura 13.1 e 13.2). No primeiro exemplo não são tratadas as exceções. Quando executamos e digitamos 0 (zero) no denominador (na linha 20), isso provoca um erro de divisão por zero. Nesse caso o programa é encerrado com erro.

No segundo exemplo (**DivideByZeroWithExceptionHandling**), as exceções são tratadas. Tanto o caso de divisão por zero, quanto o caso onde o usuário tenta digitar valores que não são números inteiros. Veja os resultados das execuções com e sem tratamento das exceções.

DivideByZeroNoExceptionHandling.java

```
1  // Fig. 13.1: DivideByZeroNoExceptionHandling.java - ADAPTADO
2  // Um aplicativo que tenta dividir por zero.
3  import java.util.Scanner;
4
5  public class DivideByZeroNoExceptionHandling
6  {
7     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
8     public static int quotient ( int numerator , int denominator )
9     {
10       return numerator / denominator ; // possível divisão por zero
11    } // fim de método quotient
12
```

```
13    public static void main ( String args [ ] )
14    {
15       Scanner leia = new Scanner ( System.in ) ; // leia para entrada
16
17       System.out.print ( " Digite um inteiro para o numerador: " ) ;
18       int numerador = leia.nextInt ( ) ;
19       System.out.print ( " Digite um inteiro para o denominador: " ) ;
20       int denominador = leia.nextInt ( ) ;
21
22       int result = quotient ( numerador , denominador ) ;
23       System.out.printf (
24          " \n Resultado: %d / %d = %d \n " , numerador , denominador , result ) ;
25    } // fim de main
26 } // fim da classe DivideByZeroNoExceptionHandling
```

Saída do console:

```
 Digite um inteiro para o numerador: 30
 Digite um inteiro para o denominador: 0
Exception in thread "main" java.lang.Arith-
meticException: / by zero
     at poo.DivideByZeroNoExceptionHandling.
quotient(DivideByZeroNoExceptionHandling.
java:10)
     at poo.DivideByZeroNoExceptionHan-
dling.main(DivideByZeroNoExceptionHandling.
java:22)
```

DivideByZeroWithExceptionHandling.java

```
1  // Fig. 13.2: DivideByZeroWithExceptionHandling.java - ADAPTADO
2  // Um exemplo de tratamento de exceções que verifica a divisão por zero.
3  import java.util.InputMismatchException;
4  import java.util.Scanner;
5
6  public class DivideByZeroWithExceptionHandling
7  {
8     // demonstra o lançamento de uma exceção quando ocorre uma divisão por zero
9     public static int quotient ( int numerator , int denominator )
10       throws ArithmeticException
11    {
12       return numerator / denominator ; // possível divisão por zero
13    } // fim de método quotient
14
15    public static void main ( String args [ ] )
16    {
```

```
17      Scanner leia = new Scanner ( System.in ) ; // leia para entrada
18      boolean continueLoop = true ; // determina se mais entradas são necessárias
19
20      do
21      {
22        try // lê dois números e calcula o quociente
23        {
24            System.out.print ( " Digite um inteiro para o numerador: " ) ;
25          int numerador = leia.nextInt ( ) ;
26          System.out.print ( " Digite um inteiro para o denominador: " ) ;
27          int denominador = leia.nextInt ( ) ;
28
29          int result = quotient ( numerador , denominador ) ;
30          System.out.printf ( " \n Resultado: %d / %d = %d \n " , numerador ,
31            denominador , result ) ;
32          continueLoop = false ; // entrada bem-sucedida ; fim de loop
33        } // fim de try
34        catch ( InputMismatchException inputMismatchException )
35        {
36          System.out.printf ( " \n Exceção: %s \n \n " , inputMismatchException ) ;
37          leia.nextLine ( ) ; // descarta entrada para o usuário tentar novamente
38          System.out.println (
39            " Você deve digitar um número inteiro. Por favor, tente novamente. \n \n " ) ;
40        } // fim de catch
41        catch ( ArithmeticException arithmeticException )
42        {
43          System.out.printf ( " \n Exceção: %s \n \n " , arithmeticException ) ;
44          leia.nextLine ( ) ; // descarta entrada para o usuário tentar novamente
45          System.out.println (
46            " Zero é um denominador inválido. Por favor, tente novamente. \n \n " ) ;
47        } // fim de catch
48      } while ( continueLoop ) ; // fim de do...while
49    } // fim de main
50  } // fim da classe DivideByZeroWithExceptionHandling
```

Saída do console:

```
 Digite um inteiro para o numerador: 30
 Digite um inteiro para o denominador: 0

 Exceção: java.lang.ArithmeticException:
/ by zero

  Zero é um denominador inválido. Por fa-
vor, tente novamente.


 Digite um inteiro para o numerador:
```

No caso acima, foi digitado um valor 0 (zero) para o denominador e foi tratado o caso com a mensagem de erro para o usuário e a solicitação novamente dos dados.

Saída do console:

```
 Digite um inteiro para o numerador: 30
 Digite um inteiro para o denominador: texto

 Exceção: java.util.InputMismatchException

  Você deve digitar um número inteiro. Por favor, tente nova-
mente.


 Digite um inteiro para o numerador:
```

Nesse caso agora, foi digitado um "texto" para o denominador e foi tratado o caso com a mensagem de erro para o usuário e a solicitação novamente dos dados.

Com base no exemplo anterior, vamos ver como é feita a captura e tratamento das exceções. Para manipular uma possível exceção foi usado o comando try. Dentro deste bloco devem ser colocadas as linhas do programa que podem gerar exceções.Para capturar e tratar exceções levantadas no bloco try use-se o catch. Como no exemplo, mais de um tipo de exceção ocorreu, usamos mais de um bloco catch. Para garantir que um bloco de código seja (quase) sempre executado, independente de acontecer uma exceção ou não, use o finally para essa finalidade (esse bloco é opcional).

Temos então a seguinte estrutura para os blocos try-catch-finally:

```
try {
        // código que pode levantar uma ou mais exceções
}
catch ( TipoDaExceção1 e ) {
  // código que será executado se TipoDaExceção1 for levantado
}
catch ( TipoDaExceção2 e ) {
  // código que será executado se TipoDaExceção2 for levantado
}
finally {
```

```
  // código (quase) sempre executado, independentemente de ter
  // ocorrido um erro ou não
}
```

PALAVRAS DO PROFESSOR

Bem caro(a) aluno(a), chegamos ao fim da unidade 4.

Para consolidar mais esses assuntos vistos aqui em nossos guias de estudos, recomendo fortemente a leitura dos capítulos do livro do DEITEL (DEITEL. Java Como Programar. Ed. Pearson Brasil) que foram mencionados aqui e que tratam dos assuntos abordados nos nossos guias.

É fundamental que você teste os exemplos vistos aqui e os outros que estão no livro, para fixar melhor os conhecimentos adquiridos durante nosso curso de programação orientada a objetos. Qualquer dúvida entre em contato com nossos tutores.

Espero que você tenha aproveitado o máximo do nosso curso e que esse conhecimento contribua para sua vida acadêmica e profissional.

Obrigado pela participação e até a próxima!