Programação Orientada a Objetos

O desenvolvimento de software é extremamente amplo. Nesse mercado, existem diversas linguagens de programação, que seguem diferentes paradigmas. Um desses paradigmas é a Orientação a Objetos, que atualmente é o mais difundido entre todos. Isso acontece porque se trata de um padrão que tem evoluído muito, principalmente em questões voltadas para segurança e reaproveitamento de código, o que é muito importante no desenvolvimento de qualquer aplicação moderna.

A Programação Orientada a Objetos (POO) diz respeito a um padrão de desenvolvimento que é seguido por muitas linguagens, como C# e Java. A seguir, iremos entender as diferenças entre a POO e a Programação Estruturada, que era muito utilizada há alguns anos, principalmente com a linguagem C. Esse padrão se baseia em quatro pilares que veremos ao longo desse artigo. Além disso, a POO diversas vantagens em sua utilização, que também serão vistas e explicadas.

Na Figura 1 vemos uma comparação muito clara entre a programação estruturada e a programação orientada a objetos no que diz respeito aos dados. Repare que, no paradigma estruturado, temos procedimentos (ou funções) que são aplicados globalmente em nossa aplicação. No caso da orientação a objetos, temos métodos que são aplicados aos dados de cada objeto. Essencialmente, os procedimentos e métodos são iguais, sendo diferenciados apenas pelo seu escopo.

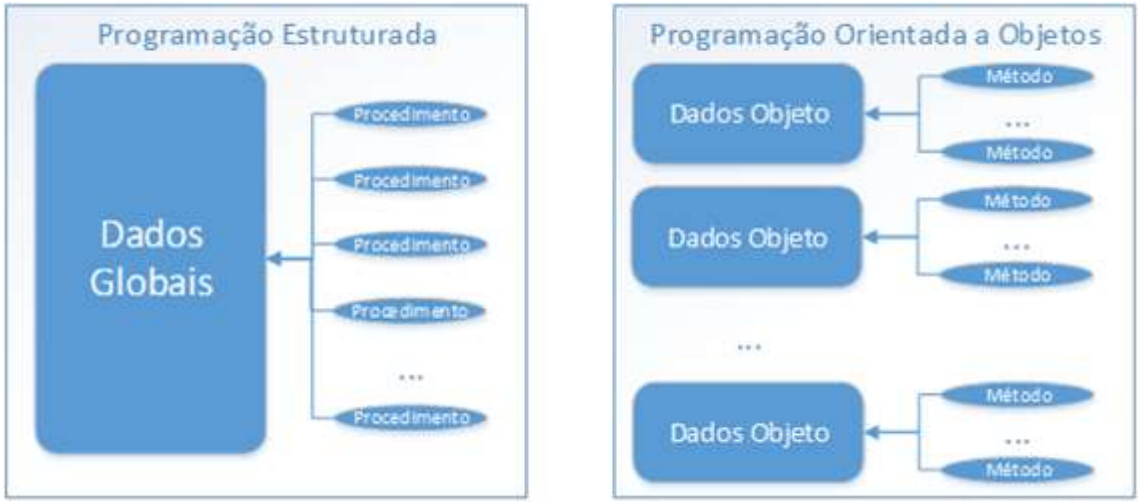


Figura 1. Estruturada x Orientação a Objetos

A linguagem C é a principal representante da programação estruturada. Se trata de uma linguagem considerada de baixo nível, que atualmente não é utilizada para projetos muito grandes. A sua principal utilização, devido ao baixo nível, é em programação para sistemas embarcados ou outros em que o conhecimento do hardware se faz necessário para um bom programa.

Essa colocação nos traz a um detalhe importante: a programação estruturada, quando bem feita, possui um desempenho superior ao que vemos na programação orientada a objetos. Isso ocorre pelo fato de ser um paradigma sequencial, em que cada linha de código é executada após a outra, sem muitos desvios, como vemos na POO. Além disso, o paradigma estruturado costuma permitir mais liberdades com o hardware, o que acaba auxiliando na questão desempenho.

Entretanto, a programação orientada a objetos traz outros pontos que acabam sendo mais interessantes no contexto de aplicações modernas. Como o desempenho das aplicações não é uma das grandes preocupações na maioria das aplicações (devido ao poder de processamento dos computadores atuais), a programação orientada a objetos se tornou muito difundida.

Essa difusão se dá muito pela questão da reutilização de código e pela capacidade de representação do sistema muito mais perto do que veríamos no mundo real.

Veremos em detalhes esses e outros pontos que dizem respeito a programação orientada a objetos. Como desenvolvedores, é nossa missão entender quais são as vantagens e desvantagens de cada um dos paradigmas de programação e escolhermos o melhor para nossa aplicação. A escolha da linguagem também deve estar presente nessa escolha.

O termo Programação Orientada a Objetos foi criado por Alan Kay, autor da linguagem de programação Smalltalk. Mas mesmo antes da criação do Smalltalk, algumas das ideias da POO já eram aplicadas, sendo que a primeira linguagem a realmente utilizar estas idéias foi a linguagem Simula 67, criada por Ole Johan Dahl e Kristen Nygaard em 1967.

Note que este paradigma de programação já é bastante antigo, mas só agora vem sendo aceito realmente nas grandes empresas de desenvolvimento de Software. Alguns exemplos de linguagens modernas utilizadas por grandes empresas em todo o mundo que adotaram essas idéias: Java, C#, C++, Object Pascal (Delphi), Ruby, Python, Lisp, ...

A maioria delas adota as idéias parcialmente, dando espaço para o antigo modelo procedural de programação, como acontece no C++ por exemplo, onde temos a possibilidade de usar POO, mas a linguagem não força o programador a adotar este paradigma de programação, sendo ainda possível programar da forma procedural tradicional. Este tipo de linguagem segue a idéia de utilizar uma linguagem previamente existente como base e adicionar novas funcionalidades a ela.

Outras são mais "puras", sendo construidas do zero focando-se sempre nas idéias por trás da orientação a objetos como é o caso das linguagens Smalltalk, Self e IO, onde TUDO é orientado a objetos.

Idéias Básicas da POO

A POO foi criada para tentar aproximar o mundo real do mundo virtual: a idéia fundamental é tentar simular o mundo real dentro do computador. Para isso, nada mais natural do que utilizar Objetos, afinal, nosso mundo é composto de objetos, certo?!

Na POO o programador é responsável por moldar o mundo dos objetos, e explicar para estes objetos como eles devem interagir entre si. Os objetos "conversam" uns com os outros através do envio de mensagens, e o papel principal do programador é especificar quais serão as mensagens que cada objeto pode receber, e também qual a ação que aquele objeto deve realizar ao receber aquela mensagem em específico.

Uma mensagem é um pequeno texto que os objetos conseguem entender e, por questões técnicas, não pode conter espaços. Junto com algumas dessas mensagens ainda é possível passar algumas informações para o objeto (parâmetros), dessa forma, dois objetos conseguem trocar informações entre si facilmente.

Ficou confuso? Vamos a um exemplo prático: imagine que você está desenvolvendo um software para uma locadora e esta locadora tem diversos clientes. Como estamos tentando modelar um sistema baseado no sistema real, nada mais obvio do que existirem objetos do tipo Clientesdentro do nosso programa, e esses Clientes dentro do nosso programa nada mais serão do que objetos que "simulam" as características e ações no mundo virtual que um cliente pode realizar no mundo real.

Vamos apresentar exemplo mais detalhadamente mais para frente, mas antes precisamos especificar mais alguns conceitos.

O que é uma Classe, Atributo e Método? E pra que serve o Construtor?

Uma classe é uma abstração que define um tipo de objeto e o que objetos deste determinado tipo tem dentro deles (seus atributos) e também define que tipo de ações esse tipo de objeto é capaz de realizar (métodos).

É normal não entender isso logo de cara, mas os conceitos de classes e subclasses são relativamente simples: Tente pensar no conceito de Classes utilizado na Biologia: um animal é uma classe, e tem suas características: é um ser vivo capaz de pensar, precisa se alimentar para viver, etc, etc. O Ser Humano é uma sub-classe dos animais.

Ele tem todas as características de um animal, mas também tem algumas peculiaridades suas, não encontradas nas outras sub-classes de animais. Os pássaros também são animais, mas possuem características próprias.

Note que uma Classe não tem vida, é só um conceito. Mas os Objetos (animais, serem humanos, pássaros, etc) possuem vida.

O seu cachorro rex é um Objeto (ou instância) da classe Cachorro. A classe Cachorro não pode latir, não pode fazer xixi no poste, ela apenas especifica e define o que é um cachorro. Mas Objetos do tipo Cachorro, estes sim podem latir, enterrar ossos, ter um nome próprio, etc.

A criação de uma nova Classe é dividida em duas partes: os seus atributos e os seus métodos. Os atributos são variáveis que estarão dentro de cada um dos objetos desta classe, e podem ser de qualquer tipo. Por exemplo, a classe Cachorro poderá ter o atributo nome que será do tipo String. Assim, cada Objeto desta classe terá uma variável própria chamada nome, que poderá ter um valor qualquer (Rex, Frodo, Atila, ...).

Métodos serão as ações que a Classe poderá realizar. Quando um objeto desta classe receber uma mensagem de algum outro objeto contendo o nome de um método, a ação correspondente a este método será executada. Por exemplo, caso um objeto da classe Dono envie uma mensagem para um objeto do tipo Cachorro falando "sente", o cachorro irá interpretar esta mensagem e conseqüentemente irá executar todas as instruções que foram especificadas na classe Cachorro dentro do método sente.

Um construtor tem uma função especial: ele serve para inicializar os atributos e é executado automaticamente sempre que você cria um novo objeto.

Quando você especifica os atributos de uma classe, você apenas diz ao sistema algo como "objetos desta classe Pessoa vão ter uma variável chamada Nome que é do tipo String, uma variável chamada idade que é do tipo inteiro, etc, etc". Mas estas variáveis não são criadas, elas só serão criadas no construtor. O construtor também pode receber parâmetro, desta forma, você pode passar para o construtor uma String contendo o nome da Pessoa que você está criando, sua idade, etc. Normalmente, a sintaxe é algo parecido com isto:

Pessoa joao := new Pessoa("João", 13);

Este código gera um novo objeto chamado joao que é do tipo Pessoa e contem os valores "João" como nome e 13 como idade.

Caso você tenha entendido o texto acima, você entendeu 99% do que esta por trás da orientação a objetos. O mais importante é entender como funciona a comunicação entre os objetos, que sempre segue o seguinte fluxo:

-Um objeto A envia uma mensagem para o objeto B.

-Objeto B verifica qual foi a mensagem que recebeu e executa sua ação correspondente. Esta ação está descrita no método que corresponde a mensagem recebida, e está dentro da Classe a qual este objeto pertence.

Exemplo

Voltando ao exemplo da locadora: Você está desenvolvendo um software para uma locadora. Esta locadora terá diversos clientes. Poderímos então criar uma classe explicando para o computador o que é um Cliente: para isso precisaríamos criar uma classe chamada Cliente, e com as seguintes características (atributos):

- Nome
- Data de Nascimento
- Profissão

Mas um cliente é mais do que simples dados. Ele pode realizar ações! E no mundo da POO, ações são descritas através da criação de métodos.

Dessa forma, objetos da nossa classe Cliente poderá por exemplo executar as seguintes ações (métodos):

- AlugarFilme

- DevolverFilme
- ReservarFilme

Note o tempo verbal empregado ao descrever os métodos. Fica fácil perceber que trata-se de ações que um Cliente pode realizar.

É muito importante perceber as diferenças entre Atributo e Método. No começo é normal ficar um pouco confuso, mas tenha sempre em mente:

- Atributos são dados.
- Métodos descrevem possíveis ações que os objetos são capazes de realizar.

Assim, nosso sistema pode ter vários Objetos do tipo Cliente. Cada um destes objetos possuirá seu próprio nome, data de nascimento e profissão, e todos eles poderão realizar as mesmas ações (AlugarFilme, RevolverFilme ou ReservarFilme).

Herança

Voltando a idéia das classes na Biologia: um ser humano é um animal. Ele tem todas as características (atributos) e pode realizar todas as ações (métodos) de um animal. Mas além disso, ele tem algumas características e ações que só ele pode realizar.

Em momentos como este, é utilizado a herança. Uma classe pode estender todas as características de outra e adicionar algumas coisas a mais. Desta forma, a classe SerHumano será uma especialização (ou subclasse) da classe Animal. A classe Animal seria a classe pai da serHumano, e logicamente, a classe SerHumano seria a classe filha da Animal.

Uma classe pode sempre ter vários filhos, mas normalmente as linguagens de programação orientadas a objetos exigem que cada classe filha tenha apenas uma classe pai. A linguagem C++ permite que uma classe herde as características de varias classes (herança múltipla), mas C++ não é um bom exemplo quando se está falando sobre conceitos de POO.

Um exemplo um pouco mais próximo da nossa realidade: vamos supor que estamos desenvolvendo um sistema para um banco. Nosso banco possui clientes que são pessoas físicas e pessoas jurídicas.

Poderiamos criar uma classe chamada Pessoa com os seguintes atributos:

- Nome
- Idade

Em seguida, criamos 2 classes que são filhas da classe Pessoa, chamadas PessoaFisica e PessoaJuridica. Tanto a classe PessoaFisia como a PessoaJuridica herdariam os atributos da classe Pessoa, mas poderiam ter alguns atributos a mais.

A classe PessoaFisica pode ter por exemplo o atributo RG enquanto a classe PessoaJuridica poderia ter o atributo CNPJ.

Dessa forma, todos os objetos da classe PessoaFisica terá como atributos:

- Nome
- Idade
- RG

E todos os objetos da classe PessoaJuridica terão os seguintes atributos:

- Nome
- Idade

- CNPJ

Os métodos são análogos: poderíamos criar alguns métodos na classe Pessoa e criar mais alguns métodos nas classes PessoaJuridica e PessoaFisica. No final, todos os objetos teriam os métodos especificados na classe Pessoa, mas só os objetos do tipo PessoaJuridica teriam os métodos especificados dentro da classe PessoaJuridica, e objetos do tipo PessoaFisica teriam os métodos especificados na classe PessoaFisica.

Polimorfismo

Um dos conceitos mais complicados de se entender, e também um dos mais importantes, é o Polimorfismo. O termo polimorfismo é originário do grego e significa "muitas formas".

Na orientação a objetos, isso significa que um mesmo tipo de objeto, sob certas condições, pode realizar ações diferentes ao receber uma mesma mensagem.

Ou seja, apenas olhando o código fonte não sabemos exatamente qual será a ação tomada pelo sistema, sendo que o próprio sistema é quem decide qual método será executado, dependendo do contexto durante a execução do programa.

Desta forma, a mensagem "fale" enviada a um objeto da classe Animal pode ser interpretada de formas diferentes, dependendo do objeto em questão.

Para que isto ocorra, é preciso que duas condições sejam satisfeitas: exista herança de uma classe abstrata e casting (outras situações também podem resultar em polimorfismo, mas vamos nos centrar neste caso).

Herança nós já definimos previamente, mas o que é uma classe abstrata? E o que é esse tal de casting?

Uma classe abstrata é uma classe que representa uma coleção de características presentes em vários tipos de objetos, mas que não existe e não pode existir isoladamente. Por exemplo, podemos criar uma classe abstrata chamada Animal.

Um Animal tem diversas características (atributos) e podem realizar diversas ações (métodos) mas não existe a possibilidade de criarmos objetos do tipo Animal.

O que existem são objetos das classes Cachorro, Gato, Papagaio, etc. Essas classes, estendem a classe Animal herdando todas as suas características, e adicionando algumas coisas a mais. "Animal" é só uma entidade abstrata, apenas um conjunto de características em comum, nada mais.

Você pode olhar para um objeto da classe Cachorro e falar "isto é um animal, pois estende a classe Animal", mas você nunca vai ver um objeto que seja apenas da classe Animal, pois isso não existe! É como eu olhar para você e falar "você é um objeto da classe SerVivo".

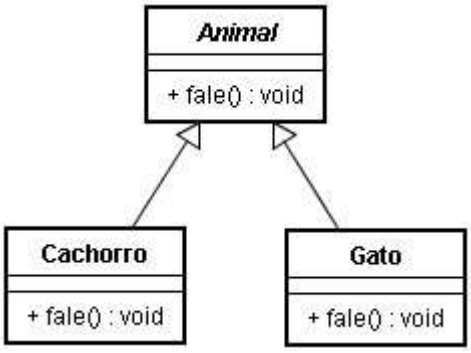
Essa afirmação está correta, mas você na verdade é um objeto da classe SerHumano, que por sua vez herda todas as características da classe SerVivo (que por sua vez é uma classe abstrata, já que não podemos criar algo que seja apenas classificado como "SerVivo", sempre vamos classifica-lo de forma menos genérica).

Resumindo: uma classe abstrata é um conjunto de informações a respeito de uma coleção de outras classes.

Uma classe abstrata sozinha é completamente inútil, já que não podemos instanciar um objeto desta classe, podemos apenas instanciar objetos de classes que estendem a classe abstrata inicial. Ela serve apenas para simplificar o sistema, juntando em um único lugar diversas características que são comuns a um grupo de classes.

Nunca esqueça disso: você nunca vai poder criar um objeto do tipo de uma classe abstrata. Sempre crie objetos das classes que estendem esta classe abstrata.

Já o termo casting é utilizado quando nos forçamos o sistema a ver um certo objeto como sendo de um determinado tipo que não é o seu tipo original. Supondo que temos a situação a seguir:



Essa é uma representação na notação UML (Unified Modeling Language) que nos informa que existe a definição de três classes em nosso sistema: existe a classe abstrata Animal (note que ela está em itálico, isso significa na notação UML que trata-se de uma classe abstrata) e existem também outras duas classes chamadas Cachorro e Gato, que são filhas da classe Animal.

Ou seja, todo objeto das classes Cachorro e Gato vão ter todas as características (atributos e métodos) presentes na classe Animal, e mais algumas características próprias.

Imagine que vamos criar um sistema de cadastro de animais. Vamos, por questões didáticas, supor que todos os animais de nosso sistema fiquem armazenados em um array. Então existiria um array contendo objetos dos tipos Gato e do tipo Cachorro. Mas armazenar diversos tipos diferentes de objetos em um único array não é uma boa idéia, pois depois, para extrair essas informações de volta é bastante complicado. Mas pare e pense por um instante: objetos do tipo Cachorro e objetos do tipo Gato são também objetos do tipo Animal, correto? Bom, então podemos criar um array capaz de armazenar Animais! Assim nossa vida fica bem mais fácil, bastando atribuir ao array os objetos (do tipo Cachorro e Gato - que estendem a classe Animal) que queremos guardar. Em forma algorítmica, seria mais ou menos:

Animal[] listaDeAnimais = new Animal[100]; // Criamos um array com 100 posições que armazena objetos do tipo Animal.

listaDeAnimais[0] = new Cachorro("Frodo"); // Criamos um novo objeto do tipo Cachorro com o nome Frodo e armazenamos no array

listaDeAnimais[1] = new Gato("Alan"); // Criamos um novo objeto do tipo Gato com o nome Alan e armazenamos no array.....

Certo! Agora temos um array com vários objetos do tipo Animal. Agora vamos fazer um looping por todos esses objetos, enviando para cada um deles a mensagem "fale". O que iria acontecer?

Inicialmente, vamos supor que a classe abstrata Animal possui o método "fale", e que ele seja implementado (de forma algorítmica) da seguinte forma:

```
Classe Animal {

 método fale() {

 imprimaNaTela(" Eu sou mudo! ");

 }

}
```

Desta forma, todo objeto que de alguma classe que estenda a classe Animal vai ter automaticamente o método "fale", e isso inclui todos os objetos das classes Cachorro e Gato. Mas todos eles, atualmente, ao receber a mensagem "fale" vão responder imprimindo na tela a mensagem "Eu sou mudo!".

Mas Gatos e Cachorros podem falar! O que podemos fazer é sobreescrever o método fale para cada uma das classes, substituindo então seu conteúdo pelo comportamento que queremos que cada sub-classe tenha.

Por exemplo, poderíamos escrever na classe Gato do seguinte método:

```
Classe Gato {

 Método fale() {

 imprimaNaTela(" Miaaaaaauuuuuu! ");

 }

}
```

Para a classe Cachorro, poderíamos fazer de forma semelhante:

```
Classe Cachorro {

 Método fale() {

 imprimaNaTela(" Au au au! ");

 }

}
```

Agora, se fizermos um looping entre todos os objetos contidos em nosso array criado anteriormente enviando para cada objeto a mensagem "fale", cada um deles irá ter um comportamento diferente, dependendo se é um Cachorro ou um Gato. Nosso looping entre todos os animais cadastrado no nosso sistema seria mais ou menos assim:

```
int cont;

 para cont de 0 a 100 faça {

 listaDeAnimais[cont].fale();

 }
```

Isto é polimorfismo! Uma mesma mensagem é enviada para diferentes objetos da mesma classe (Animal) e o resultado pode ser diferente, para cada caso. :-)

Vantagens da POO

- Os sistemas, em geral, possuem uma divisão de código um pouco mais lógica e melhor encapsulada do que a empregada nos sistemas não orientados a objetos. Isto torna a manutenção e extensão do código mais fácil e com menos riscos de inserção de bugs. Também é mais fácil reaproveitar o código.

- É mais fácil gerenciar o desenvolvimento deste tipo de software quando temos uma equipe grande. Podemos fazer uma especificação UML antes de iniciar o desenvolvimento do software em si, e em seguida dividirmos o sistema em classes e pacotes, e cada membro da equipe pode ficar responsável por desenvolver uma parte do sistema.

Desvantagens da POO

- Na minha opinião, o aprendizado do paradigma de programação orientada a objetos é bem mais complicado no início do que os velhos sistemas procedurais. ‘

- Para começar a programar é necessário ter estabelecido uma série de conceitos bastante complexos. Já na programação procedural tradicional, basta decorar meia dúzia de comandos e você já consegue fazer um programa simples.

• Dificilmente uma linguagem orientada a objetos conseguirá ter um desempenho em tempo de execução superior a linguagens não orientadas a objetos.

A programação Orientada a objetos (POO) é uma forma especial de programar, mais próximo de como expressaríamos as coisas na vida real do que outros tipos de programação.

Com a POO temos que aprender a pensar as coisas de uma maneira distinta, para escrever nossos programas em termos de objetos, propriedades, métodos e outras coisas que veremos rapidamente para esclarecer conceitos e dar uma pequena base que permita soltarmos um pouco com este tipo de programação.

Motivação

Durante anos, os programadores se dedicaram a construir aplicações muito parecidas que resolviam uma vez ou outra, os mesmos problemas. Para conseguir que os esforços dos programadores possam ser utilizados por outras pessoas foi criado a POO.

Esta é uma série de normas de realizar as coisas de maneira com que outras pessoas possam utilizá-las e adiantar seu trabalho, de maneira que consigamos que o código possa se reutilizar.

A POO não é difícil, mas é uma forma especial de pensar, às vezes subjetiva de quem a programa, de forma que a maneira de fazer as coisas possa ser diferente segundo o programador. Embora possamos fazer os programas de formas distintas, nem todas elas são corretas, o difícil não é programar orientado a objetos e sim, programar bem. Programar bem é importante porque assim podemos aproveitar todas as vantagens da POO.

Como se Pensa em Objetos

Pensar em termos de objetos é muito parecido a como faríamos na vida real. Por exemplo, vamos pensar em um carro para dar um modelo em um esquema de POO. Diríamos que o carro é o elemento principal que tem uma série de características, como poderiam ser a cor, o modelo ou a marca. Ademais tem uma série de funcionalidades associadas, como podem ser andar, parar ou estacionar.

Então em um esquema POO o carro seria o objeto, as propriedades seriam as características como a cor ou o modelo e os métodos seriam as funcionalidades associadas como andar ou parar.

Por dar outro exemplo, vamos ver como faríamos um modelo em um esquema POO de uma fração, ou seja, essa estrutura matemática que tem um numerador e um denominador que divide ao numerador, por exemplo, 3/2.

A fração será o objeto e terá duas propriedades, o numerador e o denominador. Logo, poderia ter vários métodos como simplificar, somar com outra fração ou número, subtrair com outra fração, etc.

Estes objetos poderão ser utilizados nos programas, por exemplo, em um programa de matemáticas seria feito o uso de objetos fração e em um programa que providencie uma oficina de carros, seria utilizado o uso de objeto carro. Os programas Orientados a objetos utilizam muitos objetos para realizar as ações que se desejam realizar e eles mesmos também são objetos. Ou seja, a oficina de carros será um objeto que utilizará objetos carro, ferramenta, mecânico, trocas, etc.

Classes em POO

As classes são declarações de objetos, também se poderiam definir como abstrações de objetos. Isto quer dizer que a definição de um objeto é a classe. Quando programamos um objeto e definimos suas características e funcionalidades na verdade o que estamos fazendo é programar uma classe. Nos exemplos anteriores, na verdade falávamos das classes carro ou fração porque somente estivemos definindo, embora por alto, suas formas.

Propriedades em Classes

As propriedades ou atributos são as características dos objetos. Quando definimos uma propriedade normalmente especificamos seu nome e seu tipo. Podemos ter a idéia de que as propriedades são algo assim como as variáveis onde armazenamos os dados relacionados com os objetos.

Métodos nas Classes

São as funcionalidades associadas aos objetos. Quando estamos programando as classes as chamamos de métodos. Os métodos são como funções que estão associadas a um objeto.

Objetos em POO

Os objetos são exemplares de uma classe qualquer. Quando criamos um exemplar temos que especificar a classe a partir da qual se criará. Esta ação de criar um objeto a partir de uma classe se chama instance (que significa em inglês exemplificar). Por exemplo, um objeto da classe fração é por exemplo, 3/5. O conceito ou definição de fração seria a classe, mas quando já estávamos falando de uma fração em concreto 4/7, 8/1000 ou qualquer outra a chamamos de objeto.

Para criar um objeto temos que escrever uma instrução especial que possa ser distinta dependendo da linguagem de programação que se empregue, mas será algo parecido a isto.

meuCarro = new Carro()

Com a palavra new especificamos que se tem que criar uma instance da classe que continua a seguir. Dentro dos parênteses poderíamos colocar parâmetros com os quais se inicia o objeto da classe carro.

Estados em Objetos

Quando temos um objeto suas propriedades tomam valores. Por exemplo, quando temos um carro a propriedade cor tomará um valor em concreto, como por exemplo, vermelho, cinza. O valor concreto de uma propriedade de um objeto se chama estado.

Para acessar a um estado de um objeto para ver seu valor ou mudá-lo se utiliza o operador ponto.

meuCarro.cor = vermelho

O objeto é meuCarro, logo colocamos o operador ponto e por último o nome da propriedade a qual desejamos acessar. Neste exemplo, estamos mudando o valor do estado da propriedade do objeto a vermelho com uma simples atribuição.

Mensagens em Objetos

Uma mensagem em um objeto é a ação de efetuar uma chamada a um método. Por exemplo, quando dizemos a um objeto carro para andar, estamos lhe passando a mensagem "ande".

Para mandar mensagens aos objetos utilizamos o operador ponto, seguido do método que desejamos utilizar.

meuCarro.andar()

Neste exemplo, passamos a mensagem andar(). Deve-se colocar parênteses assim como com qualquer chamada a uma função, dentro iriam os parâmetros.

Outras Coisas

Ainda há muito o que conhecer da POO já que somente fizemos referência às coisas mais básicas. Também existem mecanismos como a herança e o polimorfismo que são umas das possibilidades mais potentes da POO.

A herança serve para criar objetos que incorporem propriedades e métodos de outros objetos. Assim, poderemos construir uns objetos a partir de outros sem ter que reescrevê-lo todo.

O polimorfismo serve para que não tenhamos que nos preocupar sobre o que estamos trabalhando, e abstrairmos para definir um código que seja compatível com objetos de vários tipos.

São conceitos avançados que custa explicar nas linhas deste artigo. Não se deve esquecer que existem livros inteiros dedicados à POO e aqui só pretendemos dar uma idéia a algumas coisas para que os

lembrem quando tenham que estar diante delas nas linguagens de programação que deve conhecer um programador do web.

Tratamento de Exceção

As exceções ocorrem quando algo imprevisto acontece, elas podem ser provenientes de erros de lógica ou acesso a recursos que talvez não estejam disponíveis.

Alguns possíveis motivos externos para ocorrer uma exceção são:

- Tentar abrir um arquivo que não existe.
- Tentar fazer consulta a um banco de dados que não está disponível.
- Tentar escrever algo em um arquivo sobre o qual não se tem permissão de escrita.
- Tentar conectar em servidor inexistente.

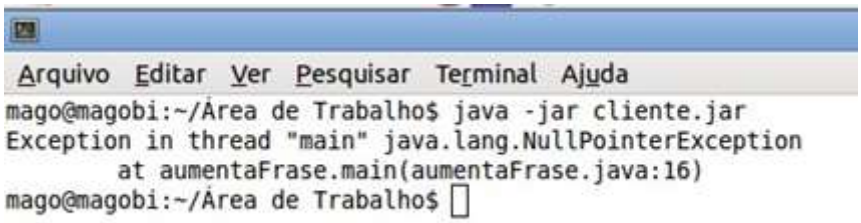


Figura 1. Ocorrência de exceção apontada no console

Alguns possíveis erros de lógica para ocorrer uma exceção são:

- Tentar manipular um objeto que está com o valor nulo.
- Dividir um número por zero.
- Tentar manipular um tipo de dado como se fosse outro.
- Tentar utilizar um método ou classe não existentes.

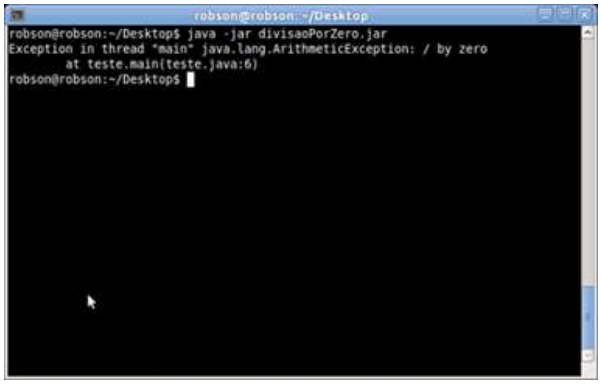


Figura 2. Exemplo de exceção por divisão por zero

Tratando Exceções

Uma maneira de tentar contornar esses imprevistos é realizar o tratamento dos locais no código que podem vir a lançar possíveis exceções, como por exemplo, campo de consulta a banco de dados, locais em que há divisões, consulta a arquivos de propriedades ou arquivos dentro do próprio computador.

Para tratar as exceções em Java são utilizados os comandos try e catch.

Sintaxe:

```
1  try
2  {
3     //trecho de código que pode vir a lançar uma exceção
4  }
5  catch(tipo_exceçao_1 e)
6  {
7     //ação a ser tomada
8  }
9  catch(tipo_exceçao_2 e)
10 {
11    //ação a ser tomada
12 }
13 catch(tipo_exceçao_n e)
14 {
15    //ação a ser tomada
16 }
```

Listagem 1. Sintaxe de uso d try-catch

Onde:

• try{ … } - Neste bloco são introduzidas todas as linhas de código que podem vir a lançar uma exceção.

• catch(tipo_excessao e) { … } - Neste bloco é descrita a ação que ocorrerá quando a exceção for capturada.

Exemplificando uma exceção

Imagine uma classe que tem um método principal main que tem como seu único objetivo alterar todas as letras de um frase para maiúsculas utilizando o método toUpperCase() da classe String, caso a frase esteja nula e se tente usar o método toUpperCase() na mesma será lançada uma exceção de NullPointerException.

Primeiro vamos ver como ficaria a tal classe sem a utilização do try/catch.

```
1  public class aumentaFrase {
2     public static void main(String args[])
3     {
4        String frase = null;
5        String novaFrase = null;
6        novaFrase = frase.toUpperCase();
7        System.out.println("Frase antiga: "+frase);
8        System.out.println("Frase nova: "+novaFrase);
9     }
10 }
```

Listagem 2. Exemplo de código sem try-catch

Quando este código for executado, o mesmo lançará uma NullPointerException, como poder ser visto na saída do console quando executamos tal programa.

```
1  Exception in thread "main" java.lang.NullPointerException
2  at aumentaFrase.main(aumentaFrase.java:15)
```

Listagem 3. Saída gerada pelo programa sem try-cach

Ou seja, o mesmo tentou acessar um atributo de um objeto que estava nulo. Para ajudar a melhorar a situação, deve-se usar o try/catch.

```
1  public static void main(String args[])
2  {
3     String frase = null;
4     String novaFrase = null;
5     try
6     {
7        novaFrase = frase.toUpperCase();
8     }
9     catch(NullPointerException e) //CAPTURA DA POSSÍVEL exceção.
10    {
11       //TRATAMENTO DA exceção
12       System.out.println("O frase inicial está nula,
13       para solucional tal o problema, foi lhe atribuito um valor default.");
14       frase = "Frase vazia";
15       novaFrase = frase.toUpperCase();
16    }
17    System.out.println("Frase antiga: "+frase);
18    System.out.println("Frase nova: "+novaFrase);
19 }
```

Listagem 4. Reformulação do código com try-catck

Quando este código for executado, o mesmo lançará uma NullPointerException, porém esta exceção será tratada desta vez, sendo a mesma capturada pelo catch{} e dentro deste bloco as devidas providências são tomadas. Neste caso é atribuído um valor default à variável frase. A saída deste programa seria a seguinte:

```
1  array4
```

Listagem 5. Saída do programa reformulado

Comando Finally

Imagine a seguinte situação: foi aberta uma conexão com o banco de dados para realizar determinada ação, e no meio deste processo seja lançada alguma exceção, como por exemplo, NullPointerException ao tentar manipular um determinado atributo de um objeto. Neste caso seria necessário que mesmo sendo lançada uma exceção no meio do processo a conexão fosse fechada. Um outro exemplo bom seria a abertura de determinado arquivo para escrita no mesmo, e no meio deste processo é lançada uma exceção por algum motivo, o arquivo não seria fechado, o que resultaria em deixar o arquivo aberto.

Quando uma exceção é lançada e é necessário que determinada ação seja tomada mesmo após a sua captura, utilizamos a palavra reservada finally.

Sintaxe:

```
1  try
2  {
3     //trecho de código que pode vir a lançar uma exceção
4  }
5  catch(tipo_exceçao_1 e)
6  {
7     //ação a ser tomada
8  }
9  catch(tipo_exceçao_2 e)
10 {
11    //ação a ser tomada
12 }
13 catch(tipo_exceçao _n e)
14 {
15    //ação a ser tomada
16 }
17 finally
18 {
19    //ação a ser tomada
20 }
```

Listagem 6. Sintaxe de uso do bloco finally

Exemplo:

```
1  public class aumentaFrase {
2     public static void main(String args[])
3     {
4        String frase = null;
5        String novaFrase = null;
6        try
7        {
8           novaFrase = frase.toUpperCase();
9        }
10       catch(NullPointerException e)
11       {
12          System.out.println("A frase inicial está nula, para
13          solucional tal o problema, foi lhe atribuito um valor default.");
14          frase = "Frase vazia";
15       }
16       finally
17       {
18          novaFrase = frase.toUpperCase();
19       }
20       System.out.println("Frase antiga: "+frase);
21       System.out.println("Frase nova: "+novaFrase);
22    }
23 }
```

Listagem 7. Programa aumetaFrase com bloco finally

Quando este código fosse executado, o mesmo lançaria uma NullPointerException, porém esta exceção será tratada desta vez, sendo a mesma capturada pelo catch{} e dentro deste bloco as devidas providências são tomadas. Neste caso é atribuído um valor default à variável frase. Neste exemplo, mesmo o código lançando uma exceção durante a sua execução e a mesma sendo capturada pelo catch, uma determinada ação será tomada no bloco finally, neste caso tanto com a exceção ou não será executada a linha “ novaFrase = frase.toUpperCase();, tornando todas letras da frase maiúsculas. A saída deste programa seria a seguinte:

```
1  array4
```

Listagem 8. Saída do programa com bloco finally

Comandos Throw e Throws

Imagine uma situação em que não é desejado que uma exceção seja tratada na própria classe ou método, mas sim em outro que venha lhe chamar. Para solucionar tal situação utilizamos o comando throws na assinatura do método com a possível exceção que o mesmo poderá a vir lançar.

Sintaxe:

```
1  tipo_retorno nome_metodo() throws tipo_exceção_1, tipo_exceção_2, tipo_exceção_n
2  {
3
4  ...
5
6  }
```

Listagem 9. Sintaxe de declaração de método com definição de exceções

Onde:

• tipo_retorno – Tipo de retorno do método.

• nome_metodo() - Nome do método que será utilizado.

• tipo_exceção_1 a tipo_exceção_n – Tipo de exceções separadas por virgula que o seu método pode vir a lançar.

Exemplo:

```
1  public class TesteString {
2      private static void aumentarLetras() throws NullPointerException //lançando excessão
3      {
4          String frase = null;
5          String novaFrase = null;
6          novaFrase = frase.toUpperCase();
7          System.out.println("Frase antiga: "+frase);
8          System.out.println("Frase nova: "+novaFrase);
9      }
10
11     public static void main(String args[])
12     {
13         try
14         {
15             aumentarLetras();
16         }
17         catch(NullPointerException e)
18         {
19             System.out.println("Ocorreu um
20             NullPointerException ao executar o método aumentarLetras() "+e);
21         }
22     }
23 }
```

Listagem 10. Definição de exceções que um método pode gerar

Neste exemplo será lançada uma exceção no método aumetarLetras():

```
1  private static void aumentarLetras() throws NullPointerException
```

Listagem 11. Definição da exceção gerada pelo método aumentarLetras

E o mesmo será tratado no método main().

```
1  ...
2  try
3  {
4     aumentarLetras();
5  }
6  catch(NullPointerException e)
7  {
8     System.out.println("Ocorreu um NullPointerException ao
9     executar o método aumentarLetras() "+e);
10 }
11 ...
```

Listagem 12. Aplicação da exceção definida

Saída:

```
1  Ocorreu um NullPointerException ao executar o método
2  aumentarLetras() java.lang.NullPointerException.
```

Listagem 13. Saída do programa atualizado

Agora imagine o caso em que seja necessário lançar uma exceção padrão ao invés de uma específica. Para resolver este problema, utilizamos o comando throw dentro do bloco catch que desejamos converter a exceção.

Sintaxe:

```
1  try
2  {
3     //...
4  }
5  catch(tipoExcessão_1 e)
6  {
7     throw new novoTipoExceçao(e);
8  }
```

Listagem 14. Sintaxe de uso do comando throw

Onde:

• tipoExcessão_1 e – Tipo de exceção que pode ser capturada pelo bloco catch.

• NovoTipoExceçao – Tipo de exceção que será lançada.

Exemplo:

```
1  public class TesteString {
2      private static void aumentarLetras() throws Exception //lançando exceção
3      {
4          String frase = null;
5          String novaFrase = null;
6          try
7          {
8              novaFrase = frase.toUpperCase();
9          }
10         catch(NullPointerException e)
11         {
12             throw new Exception(e);
13         }
14         System.out.println("Frase antiga: "+frase);
15         System.out.println("Frase nova: "+novaFrase);
16     }
17     public static void main(String args[])
18     {
19         try
20         {
21             aumentarLetras();
22         }
23         catch(Exception e)
24         {
25             System.out.println("Ocorreu uma exceção ao
26             executar o método aumentarLetras() "+e);
27         }
28     }
29 }
```

Listagem 15. Exemplo de uso do comando throw

Neste exemplo será lançada uma NullPointerException e a mesma será convertida para Exception e relançada como Exception no método aumentarLetras() e, por fim, a mesma é tratada no método main().

Saida:

```
1  Ocorreu uma exceção ao executar o método aumentarLetras()
2  java.lang.Exception: java.lang.NullPointerException
```

Listgem 16. Saída do programa atualizada

Criando Exceções

Assim como qualquer objeto, em Java também é possível criar suas próprias exceções. Imagine um cenário em que nenhuma exceção existente faça sentido para ser lançada por você.

Por exemplo, imagine que por algum motivo você precisa que uma exceção seja lançada quando a letra B ou b não existe e determinada frase, como não existe nenhuma exceção específica para este caso será necessário criar uma exceção.

Criando uma exceção para ser lançada toda vez que uma letra B ou B não é encontrada em uma determinada frase.

```
1  public class SemLetraBException extends Exception {
2      @Override
3      public String getMessage(){
4          return "Não existe letra B em sua frase";
5      }
6  }
```

Listagem 17. Exemplo de exceção customizada

Toda exceção criada deve estender Exception, neste exemplo foi sobrescrito o método getMessage(), que é exibida no prompt toda vez que a exceção é lançada.

Utilizando a Exceção

Abaixo segue um exemplo que é utilizada a exceção criada acima.

```
1  public class TesteExcecao {
2      public static void main(String args[]) throws SemLetraBException
3      {
4          String frase = "Sou um teste!";
5          if(!frase.contains("b") || !frase.contains("B"))
6          throw new SemLetraBException();
7      }
8  }
```

Listagem 18. Utilizando a exceção customizada

Quando o programa acima fosse executado, uma exceção do tipo SemLetraBException() seria lançada. Abaixo está a saída exibida no prompt:

```
1 Exception in thread "main" SemLetraBException: Não existe letra B ou b em
2 sua frase at TesteExcecao.main(TesteExcecao.java:8)
```

Listagem 19. Saída do método com a mensagem da exceção customizada