Para se programar bem em Java, é necessário conhecer orientação para Objetos.

Você vai ficar surpreso, mas OO têm muito a ver com Inteligência Artificial, e as duas se iniciaram a partir dos mesmos princípios. O conceito de objetos, utilizado em linguagens OO, tem muita relação com o estudo dos processos cognitivos que o ser humano utiliza no dia a dia. Se parece confuso, vamos utilizar um exemplo prático.

Você se senta no seu veículo e vai iniciar sua jornada diária até a empresa. Como você sabe o que deve fazer? Alguém ensinou em algum momento que você deve colocar a chave no contato, girar para dar a partida, e se for um veículo com câmbio mecânico, sabe que deve pisar na embreagem para engatar uma marcha (normalmente a mais baixa) e depois vagarosamente ir soltando a embreagem enquanto acelera para que o veículo se movimente. Bom existe uma parte do processo conhecido e outra parte do processo que pode nem ser conhecido, como por exemplo como o motor funciona, como as velas geram uma faísca que irá explodir o combustível dentro do cilindro e gerar movimento. Esta segunda parte não é necessária para você saber dirigir. Você só precisa conhecer a "interface" que utiliza para interagir com o veículo. O mesmo com uma calculadora, você não precisa saber como ela calcula a raiz quadrada ou uma soma, basta saber a interface para obter o resultado que deseja.

Neste ponto já temos um conceito de orientação para objetos, que é a interface. Também temos o objeto, que é o veículo em si. Como todos os veículos de um fabricante saem de uma mesma "forma", esta forma é chamada de classe. Uma

classe molda as características de um objeto. Você armazena classes de coisas durante a vida e conhece as interfaces para lidar com elas. Já parou para pensar porque em todos os veículos o volante, acelerador e freio estão na mesma posição? Pode parecer uma pergunta meio infantil, mas o que os fabricantes fazem é fornecer uma mesma "interface" para os vários produtos que estão oferecendo. Cada veículo de um fabricante diferente tem a sua classe, ou jeito de fabricar. Alguns são movidos a álcool, outros a gasolina, alguns SUV e outros são sedan. Mas todos compartilham uma mesma interface de interação com o usuário. Já pensou como seria incômodo se cada veículo tivesse os pedais, marcha, volante e bancos em posições diferentes? Precisaríamos reaprender tudo a cada momento que fôssemos utilizar um novo veículo (ou objeto).

O conceito mais importante de uma linguagem orientada para objetos relacionado a isto que estamos conversando se chama polimorfismo. A palavra polimorfismo significa muitas formas (do grego "poli"+"mophus"). Polimorfismo é a capacidade de manipular objetos diferentes de uma mesma forma, ou utilizando uma mesma interface. Basicamente tudo o que fazemos na vida, como escovar os dentes, abrir uma porta, dirigir um veículo, utilizar um garfo, faca, copo ou qualquer coisa que possa imaginar utiliza o polimorfismo.

O cérebro humano sabe bem menos do que imaginamos. Quando saímos de uma implementação comum e olhamos para dentro do objeto dizemos que estamos especializando. É o que o mecânico faz quando vai consertar seu veículo.

Quando saímos de uma implementação muito complexa

e nos atemos ao funcionamento básico dela estamos abstraindo.

Veja, para abrir uma porta, sabemos que devemos interagir com uma interface que é a fechadura, utilizando um objeto chamado chave. Sabemos que devemos colocar a chave na fechadura, e normalmente girar (nem sempre). Pare por um momento e faça um exercício mental. Você consegue listar todos os tipos de fechadura que já viu na vida? Aposto que esta será uma tarefa difícil. Simplesmente porque o cérebro humano ignora na maioria das vezes as especialidades, e guarda somente as abstrações. Sabemos o básico necessário para abrir uma porta (a interface). Tudo que é específico, não gastamos espaço em nosso cérebro. Se você estiver em frente uma porta corta fogo (aquela com uma alavanca que deve ser empurrada para abrir), sabemos que devemos colocar uma chave, mas como aquela é uma porta comum não há chave. Então sabemos que devemos girar, mas nada há para ser girado nesta porta. Então nosso cérebro tenta especializar, e já que não podemos girar, tentamos outros movimentos, como empurrar. Ou seja, abrimos uma porta com uma "implementação" totalmente diferente, mas utilizamos a interface que já sabíamos, ou adaptamos alguns passos para que a interface conhecida funcionasse.

O cérebro economiza espaço quando guarda abstrações ou interfaces, e somente precisa se esforçar quando está diante de um desafio que é aplicar o genérico a um contexto específico (especializar). Talvez por isto programar seja uma atividade tão difícil. Exige uma boa dose de especialização a todo momento.

Bons programadores têm facilidade em especializar, mas excelentes programadores OO tem a facilidade de especializar e generalizar quando necessário.

Percebe como tudo isto tem muito a ver com inteligência artificial? A essência dos estudos em IA é entender o funcionamento do cérebro para que possa de alguma forma sintetizar estes conhecimentos através de paradigmas computacionais (bem, ao menos uma das vertentes de IA).

Mas como isto se aplica a uma linguagem? Veja, se nossa vida é mais simples lidando apenas com interfaces e abstração, o mesmo deve ser válido para uma linguagem de programação (e na realidade é). Vamos utilizar um exemplo muito comum para explicação em OO, que é um programa para desenho de figuras geométricas.

Você foi incumbido de criar um programa de pintura de imagens, como o paint, do Windows. Ele deve fazer inicialmente quadrados e círculos, e estes quadrados e círculos tem uma cor, tamanho e posição na tela. Para este exemplo vou utilizar um pseudocódigo para ficar mais fácil de entender. Como poderemos criar vários quadrados e vários círculos, precisaremos de coleções para armazená-los. Algo como:

```
var quadrados[];
var círculos[];
```

também precisaremos de uma rotina para criar quadrados e

uma rotina para criar círculos. Também precisaremos de uma rotina para desenhar quadrados e uma rotina para desenhar círculos. Uma rotina para salvar quadrados e uma rotina para salvar círculos. Está percebendo o padrão? Basicamente qualquer funcionalidade que desejarmos precisaremos criar rotinas duplicadas para quadrados e para círculos.

```
function cria_quadrado()
function cria_circulo()
function desenha_quadrados()
function desenha_circulos()
function salva_quadrados()
function salva_circulos()
```

Podemos melhorar um nível, e criar apenas uma rotina para criar quadrados e círculos, uma para desenhar e uma para salvar. A rotina para desenhar todos os objetos ficaria, portanto:

```
function desenha()
{
    para cada quadrado desenha;
    para cada circulo desenha;
}
```

Para criar poderia ser

```
function cria(tipo)
{
    se tipo igual a quadrado cria quadrado e retorna
    se tipo igual a circulo cria circulo e retorna
}
```

Desta forma, "encapsulamos" particularidades dentro dos métodos, e olhando de fora ficou mais simples, embora a complexidade só tenha se movido para dentro do método.

Mas ainda existe um problema complicado. Imagine que o programa está todo pronto e funcionando, e o usuário está testando. Daí ele solta a pérola que é: **"puxa, mas seria legal se o programa também desenhasse triângulos, não é mesmo? ".** Pedido muito inocente, matador, e pior, faz sentido. O problema é que todos os métodos precisam ser reajustados para incorporar o tratamento do novo objeto.

```
function desenha()
{
    para cada quadrado desenha;
    para cada circulo desenha;
    para cada triângulo desenha;
}
```

Se você acha simples, imagine que depois de pronto existirão métodos para desenhar, salvar, pintar, mudar de cor, aumentar, reduzir, preencher e tudo mais que puder imaginar. Definitivamente não é algo simples de se fazer. Fizemos o programa desta forma porque não pensamos em implementar como o cérebro funciona no dia a dia.

Vamos melhorar mais um nível. Agora entra em cena o polimorfismo. Imagine o cenário, você vai a uma festa e precisa interagir com as pessoas. Para conseguir isto, você precisa identificar a pessoa, acessar uma parte do cérebro que coleta informações sobre ela, ou seja, especializar. Daí você faz a pergunta: **"olá Paula, como estão suas filhas? "**. Mas imagine outro cenário onde você vai a uma festa onde

todos devem entrar com uma mesma máscara que cobre todo o rosto. Naturalmente você pode conversar com cada pessoa, mas as perguntas seriam do tipo: **"puxa está calor aqui"** ou **"este poncho está muito bom mesmo!"**. O segundo cenário é mais fácil de lidar, pois não precisamos especializar. Para isto existem as interfaces em Java. Elas funcionam como máscaras que escondem as complexidades e só deixam à mostra generalidades. No caso do programa de desenho, teríamos que extrair tudo que é comum entre um quadrado e um circulo. Poderíamos chamar esta abstração ou interface de forma geométrica. Todas as formas geométricas têm algumas coisas em comum:

```
package meulivro;
public interface FormaGeometrica {
	public void desenha(); // todas sabem desenhar
	public float calculaArea(); // todas tem uma área
	public void salva(); // todas precisam se salvar no disco
	// outros métodos comuns que todas as formas têm
}
```

Perceba que uma interface não tem implementação, apenas os elementos com os quais os usuários interagem. Depois teremos as classes, afinal as formas geométricas existirão de verdade no programa. As interfaces são apenas abstrações do mundo real.

```
package meulivro;
public class Quadrado implements FormaGeometrica {
	private float lado;
	public Quadrado(float lado)
	{
		this.lado = lado;
	}
	public void desenha() {
```

```
            // rotina para desenhar vai aqui
      }
      public float calculaArea() {
            return lado * lado;
      }
      public void salva() {
            // rotina para salvar vai aqui
      }
}
```

Neste exemplo de classe já temos a implementação. Claro, assim como um modelo específico de veículo tem suas definições prontas, cada forma geométrica tem sua maneira de funcionar. Veja o método **calculaArea**. Ele pega o lado e multiplica por ele mesmo para calcular a área. Isto somente acontece em um quadrado, porque no circulo é diferente.
O Circulo já poderia ser algo assim:

```
package meulivro;
public class Circulo implements FormaGeometrica {
      private float raio;
      public Circulo(float raio)
      {
            this.raio = raio;
      }
      public void desenha() {
      }
      public float calculaArea() {
            return (float)(Math.PI * raio * raio);
      }
      public void salva() {
            // rotina para salvar vai aqui
      }
}
```

Algumas explicações até agora. Um método com o mesmo nome da classe em Java é chamado de Construtor. Ele inicializa as variáveis e deixa o objeto pronto para uso, ou seja,

é chamado quando criamos o Objeto (apenas uma vez na vida do Objeto). Assim como um veículo uma vez construído nunca mais retorna à fabrica (deveria ser assim).

Repare no método **calculaArea** da classe Circulo e da classe Quadrado. Embora os dois tenham o mesmo nome, cada um deles "implementa" uma coisa diferente. Ambos retornam a área, mas cada um o faz de uma forma. Não lembra muito a história do abrir a porta com fechadura e abrir a porta corta fogo?
Na verdade, quando criamos um objeto, fazemos desta forma:

```
FormaGeometrica f = new Quadrado(10);
```

Veja o destacado, você criou um Quadrado, mas atribuiu a uma variável do tipo **FormaGeometrica**. Isto é **exatamente** como se você chegasse em uma festa que comentamos anteriormente, e colocasse uma máscara igual a de todos os outros na festa (a interface é a máscara). A interface faz com que todo o restante do programa não saiba se o objeto com o qual está lidando é um Quadrado ou um Circulo. Para todo o restante do programa basta saber que o objeto é uma **FormaGeométrica**. Agora, ao invés de declarar uma coleção para cada elemento (Quadrados ou Circulos), como fizemos lá no começo, apenas fazemos:

```
FormaGeometrica figuras[];
```

Dentro desta coleção podem existir Quadrados ou Circulos, não importa. **"Ei, mas eu preciso saber o que tem na posição 2! "**. Este é um erro comum do programador. Por

que você precisaria saber disto? Lembre-se que abstrair é um esforço mental em programação, e devemos sempre que possível tratar objetos da forma mais genérica possível. Como fica a rotina de desenho de todos os Objetos?

```
public void desenhaTodos(FormaGeometrica figuras[])
{
      for (int i = 0; i < figuras.length; i++) {
            figuras[i].desenha();
      }
}
```

Olha só, você itera sobre a máscara ou interface, mesmo sabendo que por baixo de cada máscara existe um objeto real, Quadrado ou Circulo. Como a máscara declara o método desenha, e todos os objetos implementam este método, tudo deve funcionar. Não é possível ter um objeto criado a partir de uma classe que não implementa todos os métodos de uma interface. Pense sobre isto: Na hora em que formos chamar o método, a linguagem irá olhar atrás da máscara para chama-lo, e se não estiver presente, cairemos em um buraco negro sem solução. A linguagem obriga a isto. Uma forma interessante de pensar é como se você colocasse uma folha de papel sobre a impressão do código fonte da classe, com pequenas aberturas para visualizar apenas a assinatura do método. Olhando as duas folhas, uma sobre a outra, você só consegue ver a assinatura dos métodos, e não o que ele faz realmente. Isto é polimorfismo!
Lembra do desejo do usuário lá em cima de ter um Triangulo como forma geométrica? Basta criar uma classe:

```
package meulivro;
public class Triangulo implements FormaGeometrica {
	private float base;
	private float altura;
	public Triangulo(float base, float altura){
		this.base = base;
		this.altura = altura;
	}
	public void desenha() {
		// aqui vai a rotina de desenho
	}
	public float calculaArea() {
		return (base * altura) / 2 ;
	}
	public void salva() {
		// aqui vai a rotina de salvar
	}
}
```

O novo objeto precisa implementar a interface (**implements**) **FormaGeometrica**, ou seja, deve ser capaz de usar a máscara na festa.
Na verdade, fora criar a classe, existe um outro ponto que deverá ser mudado, que é a rotina de criação.

```
	public FormaGeometrica cria(int tipo, float valor1, float valor2)
	{
		// 0 é quadrado
		if (tipo == 0)
			return new Quadrado(valor1);
		// 1 é circulo
		else if (tipo == 1)
			return new Circulo(valor1);
		// 2 é triangulo
		else if (tipo ==2 )
			return new Triangulo(valor1, valor2);
		return null;
	}
```

A rotina de criação sabe que tipo de objeto deverá criar (claro, a Ford ou a Fiat sabem como construir seus veículos), mas quando saem da fábrica os carros saem com uma interface que esconde todo o funcionamento interno do usuário final. Este é o único ponto onde o objeto real está sendo conhecido. O operador new cria um novo objeto ou instância a partir de uma classe real. Estamos criando um objeto específico, mas veja lá em cima logo após o **public**, estamos dizendo que o retorno desta função é **FormaGeometrica** (ou seja, estamos colocando a máscara na frente do objeto real para retorna-lo escondido atrás da máscara). O código poderia bem ser:

```
public FormaGeometrica cria(int tipo, float valor1, float valor2)
{
      FormaGeometrica retorno = null; // sem objeto dentro
      // 0 é quadrado
      if (tipo == 0)
            return new Quadrado(valor1);
      // 1 é circulo
      else if (tipo == 1)
            retorno = new Circulo(valor1);
      // 2 é triangulo
      else if (tipo ==2 )
            retorno = new Triangulo(valor1, valor2);
      return retorno;
}
```

Se conseguirmos realizar a proeza de nunca precisarmos tirar a máscara da frente, teremos um programa completamente genérico, que pode adicionar novas classes sob demanda. Todos os métodos para desenhar, pintar, ajustar tamanho, salvar, calcular área, e tudo mais que puder imaginar ficará encapsulado nas classes.

Na prática algumas vezes precisamos no meio do

programa retirar a máscara e acessar direto o objeto, por alguma razão específica. Imagine uma rotina hipotética para contar quantos círculos existem no desenho.

```
public int contaCirculos(FormaGeometrica figuras[])
{
      int contador =0;
      for (int i = 0; i < figuras.length; i++) {
            if (figuras[i] instanceof Circulo)
                  contador++;
      }
      return contador;
}
```

Olhe o teste do **if**. O operador **instanceof** testa se o que está por trás da interface (ou máscara) é algo que você está procurando. Se for, podemos retirar a máscara assim:

```
Circulo c = (Circulo)figuras[i];
```

O Circulo entre parênteses é chamado **typecast**, e força o objeto a assumir uma determinada forma. Mas aqui não dá para blefar, ou seja, não dá para transformar um objeto que não é do tipo que ele foi criado. Assim com não dá para transformar uma porta comum em uma porta corta fogo. Isto geraria uma exceção ou erro em Java. Por isto testamos primeiro com **instanceof**. Ok, mas o que fazer se precisar tirar a máscara? Estamos quebrando um princípio fundamental da OO, mas a vida não é perfeita. Na programação do dia a dia você precisa as vezes acessar especializações, e não dá para ficar olhando para todos os objetos apenas utilizando a máscara.

Neste caso, você pode "isolar" esta funcionalidade em um local onde todas as manutenções sejam facilitadas. Mesmo que precisemos modificar ou acrescentar mais um objeto como **Poligono**, por exemplo, o número de pontos de modificação é bem menor do que no caso do exemplo lá no começo da explicação.

Neste momento você pode estar pensando: **"esta é a maior maluquice que já ouvi na vida e nunca precisei nem precisarei de nada disto para programar em Java"**. É muito possível. A maior parte de nossa vida como programadores utilizamos o ponto de vista do usuário do polimorfismo, ou a pessoa que interage com a máscara. Muitas vezes não criamos o que está por trás da máscara.

Não temos a especialização para criar um veículo novo, mas teremos a capacidade de utilizá-lo, quando for criado. Talvez nunca precisemos olhar dentro do capô do motor do Websphere, e nem é desejável mesmo. Um código parecido com este você já deve ter digitado ou visto, se já programou em JEE:

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/MeuBanco");
Connection c = ds.getConnection();
PreparedStatement stm= c.prepareStatement("select * from banco");
ResultSet rs = stm.executeQuery();
// restante do código
```

Você está obtendo uma conexão para um banco de dados, e executando uma query. Mas pergunto, qual banco? O **ctx.lookup** volta um objeto genérico, e colocamos nele a

máscara de **DataSource** (veja o typecast). Ele está na verdade retornando uma classe implementada pelo banco de dados em uso, seja DB2, Oracle, MySql. Cada um dos provedores ou fabricantes de bancos de dados recebe um conjunto de máscaras (interfaces) de quem criou a especificação, e as implementa de forma que possam ser utilizadas de forma polimórfica.

Tanto **PreparedStatement** como **ResultSet** são interfaces que serão implementadas por cada fabricante, mas raramente precisaremos nos preocupar de quem é a implementação. Na verdade, se o fizermos, estaremos complicando nossa vida quando desejarmos mudar para outro banco de dados. Quer outro exemplo? Quando você implementa um **Servlet**:

```
package meulivro;
// imports removidos
@WebServlet("/MeuServlet")
public class MeuServlet extends HttpServlet {
    public MeuServlet() {
    }
      protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
            // implementação do GET
      }
      protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
            // implementação do POST
      }
}
```

Olha lá o **extends HttpServlet**. Ele obriga você a implementar algumas coisas como o método **doGet** e **doPost** (na verdade neste caso já é uma classe concreta para você não precisar

implementar nada, mas poderia ser uma interface). Aqui você está construindo um objeto específico, que o servidor de aplicações usará a interface `HttpServlet` para acessá-lo, e não importa muito para o servidor o que mais existir além disto. Não importa se este código irá executar no Tomcat, Liberty ou WebLogic, pois todos reconhecem `HttpServlet`.

O conhecimento do código específico pertence a sua aplicação, e não ao servidor de aplicações.

O que chamamos de padrão JEE é um conjunto imenso de interfaces, que todos os fabricantes de servidores de aplicação recebem, e recheiam os códigos abaixo dele. Da mesma forma, os programadores criam códigos que interagem com o outro lado da interface, ficando a interface como se fosse a linha divisória entre desenvolvedor e fabricante. Desde que todo mundo respeite as interfaces, tanto do lado de quem usa (o programador JEE) como do lado do fabricante (a IBM) teremos a certeza de que as coisas irão colar ao final. Isto é a base do que chamamos de programação por contrato. Tanto a IBM quanto o desenvolvedor respeitam um contrato, que é a interface. Isto funciona muito bem e está permeado por todas as implementações em Java e JEE.

Ainda existe um outro conceito importante, que é o encapsulamento. O veículo funciona da forma que você espera, desde que não abra o capô e arranque alguns fios lá de dentro. Se você fizer isto, o objeto não funciona como esperado quando interagimos com a interface. Veja lá nas classes **Quadrado**, **Circulo** e **Triangulo**, o **private** na declaração de variáveis. Eles garantem que ninguém de fora

do objeto terá a capacidade de alterá-lo.

Os conceitos de encapsulamento derivam dos antigos castelos feudais da idade média. Todo o castelo era cercado por muros, e as únicas formas de acesso aos castelos era através de portões fortemente fiscalizados. Se a fiscalização pudesse garantir que nenhum baderneiro entrasse no castelo, provavelmente a convivência dentro do castelo seria saudável (o encapsulamento não protege de problemas originados dentro do castelo).

Podemos com relativa segurança afirmar que, se tudo o que é fundamental para execução do objeto estiver protegido, caso ele apresente algum problema, este certamente deve ter sido causado por algo dentro da classe. Em resumo, o encapsulamento ajuda a juntar coisas comuns dentro de uma classe, mas também facilita a encontrar problemas quando uma classe não funciona corretamente.

Embora não seja totalmente verdadeiro, há uma expectativa de que se todos os objetos funcionarem como projetados, o sistema como um todo também deverá funcionar sem problemas.

Bem, apenas para efeito de comparação, JavaScript não pode ser considerada uma linguagem OO, porque não força o polimorfismo nem implementa o encapsulamento.

Podemos criar um método em um objeto JS, mas depois de criado podemos adicionar novos métodos, modificar o comportamento dos existentes e o mesmo com variáveis. Ou seja, não podemos garantir que a máscara que colocaríamos na frente do objeto funcionaria, portanto JS nem precisa do conceito de interface. O mesmo em relação aos atributos. Não

existem atributos protegidos em JS, qualquer coisa pode ser alterada de fora do objeto.

Se por um lado, Java é engessado porque força você a pensar bem antes de criar os objetos, por outro lado o JS é mais flexível porque permite você adaptar enquanto programa. Isto pode ser positivo ou negativo, depende do ponto de vista.

Eu sugiro fortemente que você pense em utilizar estes princípios aí em cima, mesmo quando estiver programando em JavaScript.

Algumas coisas simples, como evitar modificar os métodos em prototypes após criados e atribuídos, ou mesmo colocando novos métodos ou atributos. Defina como será seu protótipo desde o início e sempre a utilize desta forma.

Isto irá manter todos os objetos de um mesmo tipo iguais. Ainda existe uma outra boa razão para fazer isto em JS. Quando você faz new em um método, ele tem associado a ele um **prototype**, que é como se fosse uma classe, com o esqueleto da classe, e que poderá ser utilizado depois para clonar um objeto. Se você adiciona um método novo ao protótipo, por definição o JS irá criar outro **prototype**, o que dificulta em termos de processamento e aumenta a quantidade de memória. Isto será mais explorado no capítulo seguinte.

Se declarar variáveis, utilize a convenção de prefixar todas as variáveis com um ou dois underline (__carro). Isto não provoca o encapsulamento, mas torna visível que uma variável não deveria ser alterada de fora da instância do método.

Coisas simples assim irão facilitar bastante sua vida com uma linguagem mais flexível como JavaScript.