# Orientação a objetos

Origem: Wikipédia, a enciclopédia livre.

**Programação Orientada a Objetos** (também conhecida pela sua sigla **POO**) é um modelo de análise, projeto e programação de *software* baseado na composição e interação entre diversas unidades chamadas de 'objetos'.[1] A POO é um dos 4 principais paradigmas de programação (as outras são programação imperativa, funcional e lógica). Os objetos são operados com o conceito de 'this' (isso) ou 'self' (si), de forma que seus métodos (muitas vezes) modifiquem os dados da própria instância. Os programas são arquitetados através de objetos que interagem entre si. Dentre as várias abordagens da POO, as baseadas em classes são as mais comuns: objetos são instâncias de classes, o que em geral também define o tipo do objeto. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.[2] A alternativa mais usual ao uso de classes é o uso de protótipos. Neste caso, objetos são copias de outros objetos, não instâncias de classes. Javascript e Lua são exemplos de linguagens cuja POO é realizada por protótipos. A diferença prática mais evidente é que na POO baseada em protótipos apenas a herança simples é implementada pela cópia do objeto. Assim, na POO, implementa-se um conjunto de classes passíveis de serem instanciadas como objetos, e.g. Python e C++ (ou objetos protótipos que são copiados e alterados, e.g. JavaScript e VimL).

Em alguns contextos, o termo modelagem orientada ao objeto (MOO) é preferível ao termo POO. De fato, o paradigma "orientado ao objeto" tem origem nos estudos da cognição e influenciou a inteligência artificial e a linguística, dada a relevância para a abstração de conceitos do mundo real. A MOO é considerada a melhor estratégia para diminuir o "gap semântico" (o hiato entre o mundo real e a representação dele), e facilita a comunicação das partes interessadas no modelo ou software (e.g. o modelador e o usuário final) na medida em que conceitos, terminologia, símbolos, grafismo e estratégias, são, potencialmente, mais óbvios, intuitivos, naturais e exatos.[1]

Muitas das linguagens de programação mais utilizadas atualmente (talvez a maioria) são multi-paradigma com suporte à POO. C++, C#, VB.NET, Java, Object Pascal, Objective-C, Python, SuperCollider, Ruby e Smalltalk são exemplos de linguagens de programação orientadas a objetos. ActionScript, ColdFusion, Javascript, PHP (a partir da versão 4.0), Perl (a partir da versão 5), Visual Basic (a partir da versão 4), VimL (ou Vim script) são exemplos de linguagens de programação com suporte a orientação a objetos. Vivace[3] é um exemplo de linguagem sem suporte à POO.

## Índice

**Em direção à POO**

**POO básica**
- Legado de paradigmas anteriores
- Novas características

**Críticas à POO**
- Críticas de programadores notórios

**Conceitos típicos da POO**
- Cliente, servidor e mensagens
- Responsabilidades
- Modularidade centrada nos dados
- Reúso
- Ação nos objetos

**Fenômenos e conceitos**

**Documentação**

**Vocabulário essencial**

**Tópicos avançados**

**Ver também**

**Referências**

**Bibliografia**

**Ligações externas**

# Em direção à POO

Além da programação estruturada, a POO incorpora as seguintes diretrizes[1]:

- Aborde o hiato entre o problema e a linguagem compreendida pela máquina: preencha bottom-up.
- Use os dados como unidades básicas para construção do programa
  - Dados e suas inter-relações são mais estáveis do que ações sobre os dados.
- Empacote os dados com suas operações naturais/intrínsecas
  - Construa sob as ideias de tipos abstratos de dados
  - Consolide os construtos que encapsulam os dados
- Concentre nos conceitos e fenômenos a serem manipulados
  - Faça uso de teorias existentes de fenômenos e conceitos (bastante útil, por exemplo, para a computação natural e a inteligência artificial em geral).
  - Forme novos conceitos a partir dos existentes
- Utilize um estilo de programação que permita o colapso da programação dos objetos

# POO básica

Tipo de dados (TD ou 'tipo') é um conjunto de valores com propriedades em comum. Tipo abstrato de dados (TAD) é um TD em que o foco recai sobre as operações nos valores do tipo, em contraste com a representacão destes valores. Estas operações escondem e protegem a representação dos dados enquanto facilitam a sua manipulação. A POO é frequentemente compreendida e descrita como calcada nos TADs.[1]

## Legado de paradigmas anteriores

Linguagens pré-POO (em especial as que comportavam a programação estruturada) forneceram o seguinte legado à POO:

- Variáveis portam informação formatada dentre um número pequeno de TDs embutidos/padrão/default e estruturas de dados derivados destes TDs básicos.
- Procedimentos/métodos/funções/rotinas, que podem receber uma entrada, retornar uma saída, e manipular dados.
- Uso das estruturas de controle.
- Modularidade: procedimentos agrupados em arquivos e módulos, com espaços distintos de nomes para que os identificadores não colidam (i.e., uso de namespaces).

## Novas características

Os atributos e métodos podem ser referentes a uma classe (e todas as suas instâncias) ou a uma única instância. O vínculo dos atributos aos métodos, de forma a manter uma interface bem definida para operação sobre os dados, e a evitar corrupção dos dados, é chamado de *encapsulamento*. O encapsulamento foi responsável pelo conceito de 'ocultamento de dados', central para a POO. O encapsulamento pode ser realizado através de convenções (em Python, underscores demarcam métodos e atributos protegidos e privados), ou via recursos da linguagem (em Java ou C++, um método privado só é acessado pela própria classe). Encapsulamento incentiva o desacoplamento.

Quando um objeto contém outros objetos, diz-se que há *composição de objetos*. A composição de objetos é usada para representar uma relação 'tem um', usualmente uma meronímia. Já a *herança* (quase sempre suportada pelas linguagens que utilizam classes) apresenta relações 'é um' (i.e. 'é um tipo de'), ou seja, relações de hiperonímia cujo resultado final é a *árvore taxonômica*. Na herança, tipicamente todos os atributos e métodos da classe pai/mãe estão também disponíveis na classe filha, embora seja comum que algumas características sejam substituídas. Assim, a herança permite reúso facilitado de características e muitas vezes reflete relações do mundo real de forma intuitiva. Ambas a 'composição de objetos' e a 'herança' constituem hierarquias entre as classes e objetos na POO.

*Herança múltipla* ocorre quando uma classe é filha de mais de uma classe. *Mixin* pode ser considerado um tipo especifico de herança, embora haja uma diferença crucial: a classe da qual são herdadas as características não é considerada pai/mãe.

*Polimorfismo* é quando alguma rotina pode ser chamada para objetos diferentes. Por exemplo, assuma que a função retornaCorPrincipal() possa ser chamada tanto em um objeto da classe Imagem quanto da classe Video. O polimorfismo é um tipo de abstração que simplifica códigos externos à hierarquia de classes e uma separação forte das *responsabilidades* (separation of concerns).

Há orientações diversas para a POO, muitas de reconhecida complexidade. Por exemplo, o *princípio da composição sobre herança* advoca que a composição de objetos é preferida à herança. O *princípio aberto/fechado* advoca que classes e funções deve ser 'abertas p extensão mas fechadas para modificação'.

*Subtipos comportamentais fortes* (ou *princípio de substituição de Liskov*, LSP do inglês *Liskov substitution principle*) são filhos que mantém todas as características dos pais. Dito de outra forma, seja 's' tal que todas as propriedades dos objetos t da classe T estão também presentes no objeto s da classe S, em que S é subtipo de T. Tal 's' é chamado de subtipo comportamental forte, subtipo forte, subtipo de Liskov, e variações semelhantes.

# Críticas à POO

As críticas mais recorrentes à POO podem ser resumidas em: 1) não satisfazer os objetivos de reusabilidade e modularidade, e 2) a ênfase demasiada em design e modelamento de software em detrimento de outros aspectos (computabilidade/algoritmos).

Não há consenso de que a POO realmente seja útil para modelar a realidade, tampouco se modelar a realidade é um objetivo pertinente. A premissa de que 'a POO é apropriada para modelar sistemas complexos com comportamentos complexos', contrasta com o renomado princípio KISS. Linguagens naturais não compartilham da estratégia usual na POO: 'priorizar coisas ao invés de ações', o que leva muitas vezes a representações contorcidas.[4] A POO muitas vezes é realizada sem uma representação transparente do fluxo de controle (ordem das instruções), o que tem se tornado uma desvantagem com a relevância, acentuada nos últimos anos, do processamento paralelo. Outras críticas usuais incluem: a POO é 'intrinsecamente menos eficiente' que o código procedural, em geral demora mais para compilar, e tende a ser extremamente complexa.

Veja também: PE vs POO.

### Críticas de programadores notórios

Joe Armstrong: "O problema com linguagens orientadas a objetos é que você tem todo esse ambiente implícito que elas levam consigo. Você queria uma banana mas o que você conseguiu foi um gorila segurando a banana e uma selva inteira."[5]

Alexander Stepanov: "Dizer que tudo é um objeto é simplesmente dizer nada."

Steve Yegge: "Por que alguém iria tão longe para colocar um só aspecto da fala [substantivos] em um pedestal?"[6]

Eric Steven Raymond: "A POO encoraja programas com muitas camadas e destrói a transparência."[7]

Rob Pike: "A POO constitui os números romanos da computação."[8]

# Conceitos típicos da POO

### Cliente, servidor e mensagens

Da passagem de mensagens entre objetos: o cliente (um objeto) emite uma mensagem para o servidor (outro objeto) em que solicita um serviço (i.e. que uma rotina qualquer seja executada) que pode implicar a emissão/retorno de uma resposta ao cliente. Um servidor pode ser cliente de outros (sub-)servidores. Esta resposta pode ser síncrona (quando o cliente aguarda a resposta do servidor para executar qualquer outra instrução) ou assíncrona (envolvendo paralelismo de atividades).

Um exemplo simples: um instituto (cliente), solicita um buffet para uma empresa especializada (servidor). Esta empresa (agora cliente) solicita salgados e sucos de outras empresas (sub-servidores). O instituto pode ficar paralisado enquanto não chega o buffet (envio síncrono de mensagem) ou continuar suas atividades (envio assíncrono de mensagem). Neste último caso, o instituto deve ser capaz de lidar com a *interrupção* das atividades (ou parte delas) assim que chegar o buffet.

A existência/espera de uma resposta pode ser inerente ao modelo de envio da mensagem ou ser realizada através do disparo de uma nova mensagem pelo servidor.

## Responsabilidades

A um servidor e seus métodos são atribuídas *responsabilidades* específicas. No exemplo anterior, a empresa de buffet não deve entregar um brinquedo, e a comida deve ser apropriada (não apenas várias barras de chocolate, por exemplo).

Uma responsabilidade que todos os servidores tem é a de manter seus dados consistentes. Idealmente, não deve ser possível tornar inconsistentes os dados de um objeto. Um objeto tem a responsabilidade de manter uma boa interface para se comunicar com outros objetos.

Outra responsabilidade genérica e usual é a de que as operações tenham um sentido razoável. Por exemplo, se um cliente envia uma mensagem que satisfaz alguma pré-condição, é obrigação do método ativado retornar um resultado que reflita a pós-condição.

## Modularidade centrada nos dados

A modularidade de um programa é uma medida que reflete o quanto ele é composto por partes separadas (chamadas de módulos). Na POO, a modularidade é preferencialmente (ou basicamente) centrada das TADs, o que pode ser chamado de *modularidade centrada nos dados* e possui as seguintes características:

- um módulo é construído encapsulando dados que representam um único conceito.
- Alto grau de coesão.
- A visibilidade (dos atributos e métodos) pode ser controlada.
- O módulo pode ser utilizado com um tipo de dado.

## Reúso

Enquanto na PE há bibliotecas de rotinas (procedure libraries), na POO há bibliotecas de classes, que podem (até certo ponto, em geral, etc) serem compreendidas com conjuntos de TADs reutilizáveis. Desafios para a reusabilidade:

- Descobrimento: onde está o componente e como acessá-lo?
- Entendimento: o que o componente oferece e como encaixa em um programa?
- Modificação: quando e como o componente deve ser modificado?
- Integração: como organizar e usar o componente em conjunto com outros componentes.

Nota: a modificação deve ser usada com cuidado. A modificação direta da rotina ou objeto implica dificuldades quando alguma versão nova da biblioteca é utilizada. Preferencialmente, as modificações devem ser feitas de forma modular, i.e. separando as contribuições locais das originais. Na POO, a herança ameniza este problema. Veja também o aberto/fechado (https://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objetos#Novas_caracter%C3%ADsticas%7Cprincípio).

## Ação nos objetos

As ações devem sempre ser realizadas através de algum objeto.

```
'Não pergunte o que um sistema faz, pergunte o que ele faz a que' (Meyer[2])
```

- Ações em geral:
  - implementadas por chamadas de procedimentos.
  - com frequência, mas não sempre, parametrizadas (com parâmetros de entrada).
- Ações em objetos:

- ativadas via mensagens.
  - uma mensagem sempre possui um objeto receptor
  - uma mensagem é similar a uma chamada de procedimento com ao menos um parâmetro
- uma mensagem ativa uma operação (um método).
  - o objeto receptor identifica a operação apropriada (busca/lookup de método)

# Fenômenos e conceitos

Metáforas são recorrentes em computação: mensagens, inteiros, etc. Classes e objetos são, em parte, inspirados na teoria de conceitos e fenômenos:[9]

- Um fenômeno é algo que possui uma existência definida na realidade ou na mente, é qualquer coisa real em si.
- Um conceito é uma coleção de fenômenos com propriedades em comum. Os atributos característicos de um conceito são:
  - **Nome do conceito**: também chamado 'designação', pode não ser único.
  - **Intenção**: a coleção de propriedades que caracterizam os fenômenos do conceito
    - Sob a ótica aristotélica, as propriedades da intenção são divididas em definidoras e características. Cada fenômeno do conceito necessariamente possui todas as propriedades definidoras, o que implica a distinção objetiva dos fenômenos associados a um conceito aristotélico.
    - Sob a ótica fuzzy, as propriedades na intenção são (grosso modo) apenas exemplos de propriedades possíveis. Neste caso, a intenção é também caracterizada por fenômenos prototípicos e (em geral) não há distinção objetiva entre fenômenos pertencentes ou não a um conceito fuzzy.
  - **Extensão**: a coleção de fenômenos do conceito.

Muitas classes representam conceitos da vida real pois muitos programas administram coisas do mundo real. No entanto, é comum que se faça uso também de conceitos imaginados (e.g. uma tabela hash). Ambos estes fatos favorecem a utilização da lógica fuzzy, mas na prática conceitos aristotélicos são convenientes para a construção de classes.

O estabelecimento de conceitos a partir de conjuntos de fenômenos é chamado de **classificação**. A intenção sendo as propriedades definidoras que são compartilhadas dentre os fenômenos do conceito. A **exemplificação** é o oposto da classificação: um subconjunto da extensão do conceito.

**Agregar** é formar um conceito que possui um número de partes (outros conceitos), i.e. é formar uma holonímia. **Decompor** é partir um conceito em um número de partes (outros conceitos), é formar meronímias. A intenção de um conceito agregado pode ser a intersecção das intenções de suas partes, pode ser a soma de suas intenções, ou pode possuir propriedades adicionais (em que o todo é mais que a soma das partes). Este último caso é útil na escrita de sistemas que envolvem sistemas complexos, e.g. para otimização bioinspiradas como por colonia de formigas, ou por enxame de partículas.

**Generalização** é formar um conceito mais amplo a partir de um (ou mais) conceitos mais estritos, é formar um hiperônimo. **Especialização** é formar um conceito mais estrito a partir de um conceito mais geral, é formar um hipônimo. A extensão de uma especialização é um subconjunto de sua generalização. A intenção de uma especialização (aristotélica) inclui a intenção de sua generalização. A especialização de um conceito é naturalmente expressa como herança de uma classe por outra, ou pela instanciação de uma classe.

# Documentação

Na POO, é/são documentado(s) o(s):

- uso:
  - descreve como o programa é usado.
  - orientado para o usuário final do programa.
- Tipos (para interfaceamento com rotinas externas)
  - descreve as propriedades externas de cada TAD.
  - orientado para programadores de clientes.
- Tipos e programas (para propósitos internos)
  - descreve as propriedades internas de um programa ou TAD.
  - orientado para os desenvolvedores (presentes e futuros) do programa ou TAD.

A documentação pode ser escrita antes, durante ou depois da programação (e tipicamente, para um mesmo software, é escrita durante estes três momentos). A documentação pode ser distinta do programa, mas pode estar dentro do programa ou o programa dentro dela.

# Vocabulário essencial

Os seguintes verbetes foram contextualizados no restante deste artigo:

- Classe representa um conjunto de objetos com características afins. Uma classe define o comportamento dos objetos através de seus métodos, e quais estados ele é capaz de manter através de seus atributos.
  - Subclasse é uma nova classe que herda características de sua(s) classe(s) ancestral(is).
- Objeto / instância de uma classe. Um objeto é armazena estados através de seus atributos e reage a mensagens enviadas a ele, se relaciona e envia mensagens a outros objetos. Exemplo de objetos da classe Humanos: João, Maria, José.
- Atributo é uma característica do objeto. Basicamente a estrutura de dados da classe. Exemplos: para um objeto 'Funcionário': nome, endereço, telefone, CPF; 'Carro': nome, marca, ano, cor; 'Livro': autor, editora, ano. Os atributos possuem valores. Por exemplo, o atributo cor pode conter o valor azul. O conjunto de valores dos atributos de um determinado objeto é chamado de **estado**.
- Método define habilidades do objeto. Bidu é uma instância da classe Cachorro, portanto tem habilidade para latir, implementada através do método 'latir'. Em geral, um método em uma classe é apenas uma definição e a ação só ocorre quando o método é invocado através do objeto, no caso Bidu. A utilização de um método deve afetar apenas um objeto em particular: todos os cachorros podem latir, mas apenas Bidu emite o latido quando seu método é executado. Normalmente, uma classe possui diversos métodos, que no caso da classe Cachorro podem ser coma, morda, corra, etc.
- Mensagem é uma chamada a um objeto para executar um de seus métodos. Também pode ser direcionada diretamente a uma classe.
- Herança é o mecanismo pelo qual uma classe pode estender outra classe ou ser estendida por outra classe. O mecanismo de herança permite que uma classe (subclasse) compartilhe o código-fonte outra classe (superclasse), aproveitando seus comportamentos (métodos) e variáveis possíveis (atributos). As grandes vantagens deste mecanismo são: 1) organização do software; 2) evitar a duplicação desnecessária de código, o que pode levar a reduzir o tempo gasto para desenvolver o projeto. **Generalização** é o processo de herança, no qual é criada uma superclasse, a partir de subclasses já existentes. **Especialização** é o processo no qual é criada uma subclasse a partir de superclasse(s) já existentes. Neste último caso, a herança é **simples**, quando uma subclasse herda características de uma única superclasse; a herança é **múltipla**, quando a subclasse herda características de mais de uma superclasse. Um exemplo de herança: Mamífero é super-classe de Humano. Ou seja, um Humano **é um** mamífero. **Observação**: Nem todas as linguagens orientadas a objetos suportam a herança múltipla (Ex.: Java).[10][11]
- Associação é o mecanismo pelo qual um objeto utiliza os recursos de outro. Pode tratar-se de uma associação simples "usa um" ou de um acoplamento "parte de". Por exemplo: Um humano usa um telefone. A tecla "1" é parte de um telefone. A composição de objetos é um tipo de associação.
- Encapsulamento consiste na separação de aspectos internos e externos de um objeto. Este mecanismo é utilizado amplamente para impedir o acesso direto ao estado de um objeto (seus atributos), disponibilizando externamente os métodos que acessam (getters) e alteram (setters) estes estados. Exemplo: você não precisa conhecer os detalhes dos circuitos de um telefone para utilizá-lo. A carcaça do telefone encapsula esses detalhes, provendo a você uma interface mais amigável (os botões, o monofone e os sinais de tom).
- Abstração é a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais. Na POO, uma classe é uma abstração de entidades existentes no domínio do software.
- Polimorfismo consiste no princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe. De acordo com a literatura, existem quatro tipos de polimorfismo que uma linguagem de programação pode ter (nem toda linguagem orientada a objeto tem suporte para os quatro tipos de polimorfismo):
  - **Universal**:
    - Inclusão: um ponteiro para a classe mãe pode apontar para uma instância de uma classe filha (exemplo em Java: "List lista = new LinkedList();" (tipo de polimorfismo mais básico que existe).
    - Paramétrico: se restringe ao uso de *templates* (C++, por exemplo) e *generics* (Java/C#).
  - **Ad-Hoc**:
    - Sobrecarga: duas funções/métodos com o mesmo nome mas assinaturas diferentes
    - Coerção: a linguagem que faz as conversões implicitamente (como por exemplo atribuir um int a um float em C++, isto é aceito mesmo sendo tipos diferentes pois a conversão é feita implicitamente).
- Interface: é um contrato entre a classe e o mundo externo. Quando uma classe implementa uma interface, ela está comprometida a fornecer o comportamento publicado pela interface[12]
- Pacotes (ou Namespaces): são referências para a organização lógica de classes e interfaces[12]

# Tópicos avançados

- Design orientado aos dados (centrada dos TADs) VS orientado à responsabilidades (classes são definidas através de contratos, centradas em responsabilidades e nas informações que compartilha).
- Diretrizes SOLID e GRASP.
- Padrões de design, GoF.
- Armadilhas da POO.
- Problema do círculo e elipse (difícil programar utilizando herança).
- Separação dos domínios (separation of concerns).
- Despacho dinâmico e múltiplo, busca de método.
- Padrões de especificação (e.g. prioridade para perguntas 'o quê?' em detrimento de 'como?')

# Ver também

- Padrões de projeto de software
- Framework
- Classe
- Arquitetura de dados
- Administração de dados
- Modelagem de dados
- Programação estruturada

# Referências


1. Object-oriented Programming in C# for C and Java programmers. Kurt Nørmark, 2011. http://people.cs.aau.dk/~normark/oop-csharp/html/notes/theme-index.html
2. Bertrand Meyer (2009). *Touch of Class: Learning to Program Well with Objects and Contracts*. [S.l.]: Springer Science & Business Media. ISBN 978-3-540-92144-8
3. https://arxiv.org/abs/1502.01312
4. Yegge, Steve (30 de março de 2006). «Execution in the Kingdom of Nouns» (http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html). steve-yegge.blogspot.com. Consultado em 3 de julho de 2010
5. Armstrong, Joe. In *Coders at Work: Reflections on the Craft of Programming.* Peter Seibel, ed. Codersatwork.com (http://www.codersatwork.com/), Accessed 13 November 2009.
6. Stevey's Blog Rants (http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html)
7. Eric S. Raymond (2003). «The Art of Unix Programming: Unix and Object-Oriented Languages» (http://www.catb.org/esr/writings/taoup/html/unix_and_oo.html). Consultado em 6 de agosto de 2014
8. Pike, Rob (2 de março de 2004). «[9fans] Re: Threads: Sewing badges of honor onto a Kernel» (http://groups.google.com/group/comp.os.plan9/msg/006fec195aeeff15). *comp.os.plan9* (Lista de grupo de correio). Consultado em 17 de novembro de 2016
9. A Conceptual Framework for Programming Languages: Jørgen Lindskov Knudsen and Kristine Stougaard Thomsen, Department of Computer Science, Aarhus Universitet, PB-192, April 1985.
10. Deitel, Paul (2010). *Java Como Programar - Oitava Edição*. São Paulo: Prentice Hall
11. Barnes, David J. (2009). *Programação Orientada a Objetos com Java - 4ª Edição*. São Paulo: Prentice Hall
12. «Lesson: Object-Oriented Programming Concepts» (http://download.oracle.com/javase/tutorial/java/concepts/) (em inglês). Oracle Corporation. Consultado em 20 de novembro de 2010


# Bibliografia


- Pablo Dall'Oglio (2007). *PHP Programando com Orientação a Objetos* (http://www.adianti.com.br/phpoo). Inclui Design Patterns 1 ed. São Paulo: Novatec. ISBN 978-85-7522-137-2
- James Martin, Princípios de Análise e Projeto Baseado em Objetos, 1994, Editora Campus, ISBN 85-7001-872-X


# Ligações externas

- cfoop.org OOP para ColdFusion (http://www.cfoop.org/) (em inglês)
- Conceitos de OOP em Java (http://java.sun.com/docs/books/tutorial/java/concepts/index.html) (em inglês)
- Guia do Hardware: Programação Orientada a Objetos (http://www.guiadohardware.net/artigos/programacao-orientada-objetos/) (em português)
- PHP Orientado a Objetos (https://figuraweb.com/web/php-orientado-abjetos-aprenda) (em português)

---