Universidade Federal Fluminense
Centro Tecnológico
Escola de Engenharia
Curso de Engenharia de Telecomunicações
Programa de Educação Tutorial
Grupo PET-Tele

# Tutorial de Programação Orientada a Objeto

(Versão: 2k9)

Niterói - RJ
Julho / 2009

# Sumário


**1 Introdução** **2**
1.1 Paradigmas de programação . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Justificativa para o paradigma OOP . . . . . . . . . . . . . . . . . . 3

**2 Conceitos básicos** **5**
2.1 Objeto . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.2 Dados . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.3 Métodos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.4 Dados privados e métodos privados e públicos . . . . . . . . . . . . 6
2.5 Declaração X Implementação de métodos . . . . . . . . . . . . . . . 6

**3 Classes, Instâncias e Mensagens** **7**
3.1 Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
3.2 Instâcias . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
3.3 Mensagens . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
3.4 Encapsulamento . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.5 Reusabilidade . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
3.6 Manutenibilidade . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.7 Eficiência e Eficácia na programação . . . . . . . . . . . . . . . . . 10

**4 Herança, Polimorfismo e Extensibilidade** **11**
4.1 Herança . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
4.2 Hierarquia . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
4.3 Polimorfismo . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
4.4 Ligação Dinâmica . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
4.5 Extensibilidade . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

# Capítulo 1

# Introdução

A programação orientada a objetos é um paradigma de programação, isto é, uma forma, nem mais certa ou errada do que as outras, de escrever um software.

A origem da OOP é encontrada nas linguagens de programação LISP e ALGOL. A primeira linguagem a utilizar idéias de Programação Orientada a Objeto foi a Simula 67, uma linguagem proveniente do ALGOL criada por Ole Johan Dahl e Kristen Nygaard em 1967 utilizada em simulações.

Uma linguagem de programação é considerada orientada a objeto, a partir do momento que passa a oferecer mecanismos para explorar a encapsulação. A encapsulação contribui para a redução dos efeitos que as mudanças causam sobre os dados existentes. A fim de proteger estes dados, colocamos uma "parede" de código em volta deles. Assim, toda a interação com estes dados será manipulada por procedimentos criados para mediar o acesso a eles.

Praticamente todas as linguagens vem explorando a encapsulação de código, porém, de maneiras distintas. A importância da encapsulação se dá pela necessidade de decompor sistemas grandes em pequenos subsistemas encapsulados, facilitando assim, o seu desenvolvimento, manutenção e portabilidade.

Na programação convencional, o usuário deve escolher o método de acesso (interação) que seja compatível com o tipo de dado a ser manipulado. O intuito do OOP é retirar do usuário esta responsabilidade, e colocá-la no próprio dado e no criador destes métodos de acesso.

Temos que a OOP é uma evolução da arte de escrita e desenvolvimento de um código de programação.

## 1.1 Paradigmas de programação

Como já foi dito, um paradigma de programação é uma forma nem mais certa ou errada do que as outras, de escrever um software. Algumas linguagens foram desenvolvidas para suportar um paradigma específico, enquanto outras linguagens suportam múltiplos paradigmas.

Muitas vezes, os paradigmas de programação são diferenciados pelas técnicas de programação que proíbem ou permitem. Por exemplo, a programação estruturada não permite o uso de goto. Esse é um dos motivos pelo qual novos

paradigmas são considerados mais rígidos que estilos tradicionais.

Existem muitos paradigmas de programação, podemos citar os mais importantes:

**Paradigma Procedural**

Paradigma baseado no conceito de chamadas a procedimentos, que simplesmente possuem passos computacionais a serem executados.

**Paradigma Funcional**

Paradigma baseado no conceito de funções, que podem ser construídas, manipuladas e resolvidas, como qualquer outro tipo de expressão matemática, usando leis algébricas. Pode-se pensar neste paradigma como a avaliação de expressões. Temos que o significado de uma expressão é seu valor, que é exatamente o que o computador obtém.

**Paradigma Lógico**

Paradigma baseado no uso da lógica matemática.

**Paradigma OOP**

Paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos.

Baseado na aproximação do mundo virtual do mundo real, simulando o mundo real no computador, consiste em abordar a resolução de um problema, analisando suas entidades e o relacionamento entre elas, dentro do contexto onde se situa o problema. Após esta análise, se constroi um Modelo de Resolução representando, da melhor maneira possível, o que acontece no mundo real.

A análise e projeto orientados a objetos identificam o melhor conjunto de objetos para descrever um sistema de software. O funcionamento deste sistema se dá através do relacionamento e troca de mensagens entre estes objetos. É uma forma mais adequada de pensar em como fazemos cálculos, manipulamos dados dentro do computador e de que forma abstraímos o mundo real para o virtual.

## 1.2 Justificativa para o paradigma OOP

Ao decorrer do tempo, vários programadores construiram aplicações semelhantes, que resolviam os mesmos problemas. Para que os trabalhos dos programadores pudessem ser utilizados por outros programadores, foi criado a Programação Orientado a Objeto. Desta maneira, possuímos a reutilizão do código já desenvolvido, poupando tempo e trabalho.

Um ponto importante na Programação Orientada a Objeto é possuir uma divisão de código um pouco mais lógica e melhor encapsulada do que a empregada

em linguagens de programação utilizando outros paradigmas, tornando assim, a manutenção e a extensão do código mais fácil e com a probabilidade de obter bugs no código, menor. Tal divisão facilita o desenvolvimento do software quando este é feito em grupos, podendo cada membro do grupo ficar responsável por desenvolver uma parte do software.

# Capítulo 2

# Conceitos básicos

## 2.1 Objeto

Como visto até então, no paradigma de Programação Orientada a Objeto, tentamos aproximar o mundo virtual do real. Tendo isto em mente, podemos considerar o objeto como uma representação de algo que realmente existe, como por exemplo: um carro, cliente, relógio. Devemos abstrair o objeto que realmente existe (objeto real), e representá-lo computacionalmente, como por exemplo construir o objeto carro.

Exemplo 1:

Podemos considerar o carro Vectra como um elemento que possui uma série de características: a cor, o modelo ou a marca. Além destas características, o objeto também possui uma série de funcionalidades associadas: andar, parar, freiar.

Temos que:

- O Vectra seria o objeto.
- As propriedades seriam as características (dados como a cor, modelo ou marca).
- Os métodos seriam as funcionalidades (andar, parar ou freiar).

Exemplo 2:

Podemos considerar a matriz A como um elemento que possui uma série de características: as linhas, as colunas e os elementos da matriz. Além destas características, o objeto também possui uma série de funcionalidades associadas: Inicialização, leitura, multiplicação por um escalar, inversão, multiplicação por um vetor.

Temos que:

- O objeto seria a Matriz A
- As propriedades seriam os dados - Linhas, colunas e elementos da matriz.
- Os métodos seriam a inicialização, a leitura, a multiplicação por um escalar, a inversão, a multiplicação por um vetor.

Generalizando, temos que um objeto possui um estado, comportamento e identidade.

- Estado - valores dos dados.
- Comportamento - é definido pelos métodos, ou seja, como o objeto age e/ou reage.
- Identidade - O que diferencia um objeto de outro.

## 2.2 Dados

São as características referentes ao objeto, no exemplo citado do objeto Vectra temos que os dados são: a cor, o modelo ou a marca.

## 2.3 Métodos

Os métodos de um objeto caracterizam o comportamento do mesmo, ou seja, o conjunto de ações que o objeto pode realizar. As ações podem depender dos valores dos dados, portanto, temos que as ações ou métodos estão estritamente ligados aos dados.

No exemplo do objeto Vectra, os métodos são: andar, parar, freiar.

## 2.4 Dados privados e métodos privados e públicos

Os dados e métodos privados são aqueles que não podem ser acessados por partes do programa que se encontram fora do objeto, somente o objeto pode acessá-los e alterá-los.

Os dados e métodos públicos são aqueles que estão disponíveis para tudo e todos acessarem, desde que tenham visibilidade para o objeto.

## 2.5 Declaração X Implementação de métodos

Temos que a declaração dos métodos descreve suas características externas, ou seja, sua parte visível, como seu nome, tipos dos parâmetros e do valor retornado. Enquanto que a implementação descreve o código efetivo para a operação do método utilizado, é uma sequência de procedimentos podendo incluir a chamada de outro método pertencente ao objeto.

# Capítulo 3

# Classes, Instâncias e Mensagens

## 3.1 Classes

Em uma aplicação de programação é possível, e muito provável, que existam diversos objetos com as mesmas características. Com o objetivo de evitar uma redundância que ocorreria se as propriedades de um objeto fossem declaradas individualmente, as linguagens orientadas a objeto permitem a criação de um modelo a partir do qual são criados os objetos. Esse modelo é denominado **classe**.

Como exemplo, tomemos como comparação o Vectra, Linea e Palio. Todos são carros diferentes, porém, todos são carros. Possúimos portanto, a classe carro e os objetos pertencentes a esta classe: Vectra, Linea e Palio.

Considere também, por exemplo, os personagens Tom e Garfield. São indivíduos diferentes, porém, possuem características em comum, pois ambos são gatos.

Uma classe é um grupamento de objetos que possuem semelhanças entre si, tanto no aspecto estrutural quanto funcional.

Toda classe possui atributos e serviços. Um atributo é uma característica comum aos objetos da classe e um serviço é algum procedimento ou operação que pode ser solicitado ao objeto. Cada serviço corresponde a um método.

## 3.2 Instâcias

Em um programa, todos os objetos criados a partir de uma classe, irão compartilhar as características descritas por essa classe. Portanto, definimos que todo objeto é uma **instância** de uma classe.

À partir da definição de classe, podemos definir protocolo como sendo o conjunto de métodos correspondentes a uma classe. Portanto, todas as instâncias de uma classe (objetos) obedecem ao mesmo protocolo, podendo assim, executar os mesmos métodos descritos pela classe.

Da mesma maneira, temos que cada instância possui uma coleção de dados privados, assim como uma coleção de valores para estes dados. Assim, os dados privados são chamados de variáveis de instância.

Podemos definir variáveis de instância como sendo os elementos que compõem a coleção de dados privados de uma classe.

Por exemplo, a classe carro possui as variáveis de instância cor, modelo ou marca. Os objetos Vectra, Linea e Palio devem ter seus próprios valores para essas variáveis.

Como dito na definição de objeto, este possui uma identidade que é o que o diferencia de outros objetos. Mesmo que objetos diferentes possuam o mesmo conjunto de valores, não podemos afirmar que estes objetos são iguais. Podemos então definir **estado interno** como sendo o conjunto de valores assumidos pelas variáveis de instância de um objeto. Assim, os objetos que possuem o mesmo conjunto de valores não são iguais, possuem o mesmo estado interno.

Cada vez que o valor armazenado em uma variável de instância de um objeto é alterado, alteramos o seu estado interno. Seria importante que cada objeto tivesse um valor de inicialização conhecido para suas variáveis de instância, o que definimos como sendo o **estado inicial** do objeto.

Na programação orientada a objeto, necessitamos que toda classe possua, em seu protocolo, métodos que inicializem e finalizem objetos, estes, respectivamente, são denominados construtores e destrutores.

Os construtores são utilizados para inicializar os valores das variáveis de instância, impedindo que estas assumam valores inválidos. Já os destrutores são utilizados para liberar memória caso alguma das variáveis de instância seja do tipo ponteiro ou simplesmente quando um objeto não precisa ser mais utilizado.

## 3.3 Mensagens

Um método só é ativado através do envio de uma **mensagem** ao objeto, o que corresponde a uma chamada de procedimento. Toda mensagem é composta por um seletor, que deve ser igual ao nome de algum dos métodos presentes no protocolo do objeto, e um conjunto de parâmetros que deve corresponder, em tipo e em número, à declaração da interface desse método.

Podemos não somente enviar mensagens iguais para objetos diferentes, assim como é possível enviar a mesma mensagem para instâncias de classes diferentes, tendo em vista que estas podem possuir métodos com a mesma interface.

Quando uma mensagem é enviada a um objeto, ocorre a ativação de um método. Após a execução deste método obtemos um resultado que é denominado **resposta**. É importante salientar que um método pode não retornar um valor como resposta, podendo simplesmente alterar os valores de algumas variáveis de instância do objeto, alterando assim seu estado interno.

No contexto de Mensagens, temos o **comportamento** de um objeto que é determinado pela forma como ele responde às mensagens que recebe. Logo, todas as instâncias de uma classe devem apresentar comportamento idêntico, visto que respondem às mensagens ativando os mesmos métodos.

## 3.4 Encapsulamento

Pode-se definir **encapsulamento** como sendo a capacidade de um objeto possuir uma parte privada, construindo assim, uma "muralha" ao redor do objeto, impedindo o cliente de enxergar os detalhes de sua implementação. Assim, é permitido ao cliente utilizar o obejto ignorando a forma como está organizada a sua estrutura interna e como as operações são realizadas. O cliente não terá conhecimento do interior do objeto, e o acesso ao mesmo será feito através do envio de mensagens.

O encapsulamento é muito importante na programação orientada a objeto, porém, esta importância não se dá somente para que o cliente não enxergue o interior do objeto. Na verdade, precisamos de objetos que sejam independentes da aplicação, ou seja, objetos que sejam altamente reusáveis. Assim, temos que o projetista não pode ver o que está externo ao objeto.

O uso do encapsulamento nos permite que novas classes sejam projetadas e desenvolvidas como unidades individuais e independentes de qualquer aplicação particular, assim, com uma capacidade de reutilização em diversas situações. Desta maneira, o foco principal é o de projetar os detalhes internos do objeto, garantindo que seu comportamento seja independente do ambiente ao redor.

Diante das informações esclarecidas, podemos evidenciar duas regras do encapsulamento:

**Regra 1**

Apenas os métodos devem ser capazes de modificar o estado interno de um objeto.

**Regra 2**

A resposta a uma mensagem deve ser completamente determinada pelo estado interno do objeto receptor.

De acordo com a regra 1, podemos mudar o estado interno de um objeto por meio do envio de uma mensagem. Porém, não significa que toda mensagem enviada ao objeto irá mudar seu estado interno, podendo somente retornar um valor como resposta.

De acordo com a regra 2, se dois objetos possuem o mesmo estado interno, então devem responder a mensagens iguais de forma idêntica, garantindo assim, que a resposta a uma mensagem não dependa de fatores externos ao objeto.

Temos então, por exemplo, que não podemos ter o acesso a variáveis globais da aplicação no interior de um método, pois isto causa problemas de interdependências entre módulos, o que queremos evitar.

## 3.5 Reusabilidade

Com o encapsulamento, o funcionamento de cada classe é independente do resto da aplicação, tornando assim, os objetos reusáveis, permitindo que sejam

reaproveitados diversas vezes sem que seja necessária uma mudança em sua implementação.

Então, temos a proposta da **Reusabilidade**, que nada mais é que a diminuição dos custos de projetos, através do reuso de componentes, objetos por exemplo, tornando o desenvolvimento do software mais rápido, sem diminuir obviamente, a qualidade do mesmo.

Entretanto, o desenvolvimento de classes e objetos reusáveis se torna mais complexo pois não desejamos um objeto que satisfaça os requisitos de uma aplicação particular, e sim uma classe genérica que possa ser utilizada em uma grande variedade de contextos. Assim, o projetista deve gastar uma boa parcela de tempo no planejamento de uma nova classe que satisfaça a proposta da reusabilidade, e também é de extrema importância abandonar certos hábitos da programação convencional, e se direcionar para os conceitos da programação orientada a objeto.

## 3.6 Manutenibilidade

A **manutenibilidade** da aplicação nada mais é que um benfício adicional proveniente do encapsulamento pois este, garante que os detalhes de implementação permaneçam ocultos, dando liberdade ao projetista de fazer as modificações na parte privada ao objeto, garantindo que as alterações no mesmo tenham efeitos mínimos e bem localizados, impedindo a propagação de certos erros para todo o sistema.

Vale ressaltar que a manutenibilidade só é bem definida e utilizada caso as regras do encapsulamento sejam respeitadas.

## 3.7 Eficiência e Eficácia na programação

As metodologias de programação visavam produzir programadores eficientes. A eficiência do programador era medida pelo número de linhas do programa que o programador conseguia escrever por unidade de tempo.

A metodologia de programação orientada a objeto pretende produzir programadores eficazes. A eficácia do programador é medida pela sua habilidade de utilizar os objetos disponíveis.

O trabalho do programador orientado a objeto passa a ser mais concentrado nas exigências dos usuários do que nos detalhes da implementação. Sua tarefa portanto, é coordenar os esforços individuais de diversas classes de objetos para atingir o resultado desejado.

# Capítulo 4

# Herança, Polimorfismo e Extensibilidade

No decorrer do desenvolvimento de um software, podemos nos deparar com a seguinte situação: Não temos disponível na biblioteca uma classe com todas as características que necessitamos.

É comum encontrarmos classes com características próximas das quais necessitamos, porém, na maioria das vezes é necessária a inclusão de certas características particulares exigidas pela aplicação.

Para resolver este problema, poderíamos ter acesso ao código-fonte do objeto e modificá-lo de forma a atender nossas necessidades, o que não seria possível dentro do contexto de programação orientada a objeto pois iria contrariar os conceitos de abstração, encapsulamento e a relação projetista-cliente.

Caso tivéssimos somente uma unidade pré-compilada, então teríamos que refazer todo o trabalho, ainda que tivéssimos acesso a grande parte da implementação da classe. Entretanto, isto também não poderia ser possível pois iria contrariar o princípio da reusabilidade.

Para obtermos uma solução para o caso em questão, as linguagens de programação orientadas a objeto fazem uso do conceito de herança.

## 4.1 Herança

A **herança** é um mecanismo que permite que uma nova classe consiga herdar propriedades de uma outra classe existente. A nova classe a ser criada é chamada de **subclasse** e a classe existente que será utilizada para descrever a nova classe é chamada de **superclasse**. Temos que a subclasse herda os métodos e variáveis de instância da superclasse.

A subclasse é capaz de adicionar novos métodos e variáveis de instância à superclasse ou alterá-los de maneira a atender as necessidades do programador. Então para definir uma subclasse, dizemos em que ela difere da superclasse.

Temos também a possibilidade de redefinir métodos herdados pela subclasse, ou seja, refazer sua implementação, o que é chamado de ***overriding***.

É importante ressaltar que a subclasse envolve a adição ou a modificação de características, mas nunca sua subtração. Assim, temos que toda instância de uma subclasse é também um membro da superclasse.

Como por exemplo, temos a classe carro com os objetos Vectra, Palio e Linea e como métodos temos parar, andar e freiar. Podemos agora definir uma nova classe, a classe das bicicletas por exemplo e adicionar o método de passar a marcha.

No caso, utilizaremos a superclasse carro e criaremos uma subclasse bicicleta adicionando um método e utilizando os métodos e variáveis de instância da classe carro, ficando da seguinte maneira.

**Superclasse - carro**

Propriedades: cor, modelo, marca.

Métodos: andar, parar, freiar.

**Subclasse - bicicleta**

Mesmas propriedades.

Método adicional: passar_marcha

Existem linguagens de programação que fazem uso do mecanismo de **Herança Múltipla**, onde uma nova classe pode herdar características de duas ou mais classes que não estão relacionadas como superclasse e subclasse uma da outra.

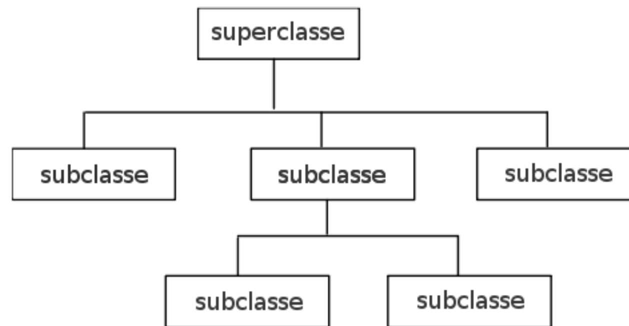
## 4.2 Hierarquia

A **hierarquia de classes** pode ser definida como a organização na qual as classes mais genéricas ficam próximas ao topo e as mais específicas na parte inferior, com classes intermediárias fazendo a ligação. Assim, as classes de níveis mais baixos herdam as características das classes acima delas.

A relação existente entre a superclasse e a subclasse é denominada "**é-um**" (*is-an*) pois dizemos que uma instância da subclasse **é um** membro da superclasse.

Os termos superclasse e subclasse podem ser considerados ambíguos, podendo ser utilizados para designar herança imediata ou através de outras classes. Por isso, autores preferem utilizar uma notação diferente, onde classes são relacionadas como pai e filho (indicando herança imediata) e descendente (indicando herança indireta) ou então, utilizar o termo "**espécie-de**" (*kind-of*) para definir o relacionamento entre subclasses e superclasse, prefirindo utilizar o "é-um" para o relacionamento entre instância e classe.

Abaixo temos uma hierarquia de classes na OOP.




## 4.3 Polimorfismo

O **polimorfismo** permite o envio de uma mesma mensagem para diferentes objetos, e que cada objeto responda da maneira mais apropriada para sua classe. Assim, vários objetos podem conter métodos implementados em diferentes níveis de hierarquia de classes, utilizando nomes idênticos. Temos em cada classe, um comportamento específico para o método.

Um conceito importante a ser definido é o de *overloading* (sobrecaga), que é definido como a habilidade de criar métodos com os mesmos nomes, mas diferindo-se entre si pelos dados de entrada (parâmetros). Sobrecarga não pode ser confundido com polimorfismo.

## 4.4 Ligação Dinâmica

Quando um procedimento é chamado, a determinação do endereço para o qual a execução do programa deve ser desviada pode ser feita em tempo de compilação. Porém, quando o objeto para o qual a mensagem é enviada é polimórfico, não é possível determinar qual será o endereço. A classe a qual pertence o objeto só será determinada em tempo de execução, assim como o método chamado em resposta à mensagem enviada. Denomina-se **ligação dinâmica** (*dynamic binding* ou *late binding*) o mecanismo que permite a resolução dos endereços em tempo de execução. O caso contrário, quando a determinação do endereço é feita em tempo de compilação, o mecanismo é denominado **ligação estática** (*static binding* ou *early binding*).

## 4.5 Extensibilidade

Pode-se considerar a **extensibilidade** como uma vantagem adicional obtida com a herança e o polimorfismo.

Temos que na programação convencional, quando um código não é útil para uma nova aplicação, fazemos uso do reuso para revisões e adaptações, o que não é muito agradável pois torna o software sujeito a erros que podem se propagar pelo programa. Na programação orientada a objeto, a extensibilidade nos permite criar uma nova subclasse para incorporar os requisitos específicos de uma nova

aplicação à ser desenvolvida, assim como podemos incluir novos tipos de objetos nas aplicações, com um mínimo de mudanças e erros pelo código.

# Referências Bibliográficas

[1] FERREIRA, Marcelo; JARABECK, Flávio , *Programação Orientada ao Objeto com Clipper 5.0, São Paulo, Makron Books, 1991*

[2] Namir Shammas , *Programação Orientada ao Objeto com TURBO PASCAL 5.5, Mc Graw-Hill, 1991*