THIAGO FARIA E NORMANDES JR

# ORIENTAÇÃO A OBJETOS

## Conceitos e Aplicação com Java

*Presente de Natal para nossos queridos alunos e seguidores.*

*Dezembro/2016*

algaworks

# Orientação a Objetos - Conceitos e Aplicação com Java

por Thiago Faria e Normandes Junior

1ª Edição, 24/12/2016



AlgaWorks Softwares, Treinamentos e Serviços Ltda
www.algaworks.com
contato@algaworks.com
+55 (11) 2626-9415

# Sobre os autores

## Thiago Faria de Andrade

Fundador e instrutor da AlgaWorks. Certificado como programador Java pela Sun, autor e co-autor de diversos livros de Java, Java EE, JPA, JSF e PrimeFaces e palestrante da maior conferência de Java do mundo (JavaOne San Francisco). Thiago iniciou seu interesse por programação aos 14 anos de idade (1995), quando desenvolveu o primeiro jogo de truco online e multiplayer do mundo. Já foi sócio e trabalhou em outras empresas de software como programador, gerente e diretor de tecnologia, mas nunca deixou de programar.

LinkedIn: https://www.linkedin.com/in/thiagofa

## Normandes Junior

Sócio e instrutor da AlgaWorks. Graduado em Engenharia Elétrica pela Universidade Federal de Uberlândia e detentor das certificações LPIC-1, SCJP e SCWCD. Palestrante da maior conferência de Java do mundo (JavaOne San Francisco), autor e co-autor de vários livros e instrutor de cursos de Java, JPA, TDD, Design Patterns, Spring Framework, etc.

LinkedIn: https://www.linkedin.com/in/normandesjr

# Antes de começar...

Antes que você comece a ler esse livro, nós gostaríamos de combinar algumas coisas com você, para que tenha um excelente aproveitamento do conteúdo. Vamos lá?

## Como obter ajuda?

Durante os estudos, é muito comum surgir várias dúvidas. Nós gostaríamos muito de te ajudar pessoalmente nesses problemas, mas infelizmente não conseguimos fazer isso com todos os leitores do livro, afinal, ocupamos grande parte do dia ajudando os alunos de cursos online na AlgaWorks.

Então, quando você tiver alguma dúvida e não conseguir encontrar a solução no Google ou com seu próprio conhecimento, nossa recomendação é que você poste na nossa Comunidade Java no Facebook. É só acessar:

http://alga.works/comunidadejava/

## Como sugerir melhorias ou reportar erros sobre este livro?

Se você encontrar algum erro no conteúdo desse livro ou se tiver alguma sugestão, vamos ficar muito felizes se você puder nos dizer.

Envie um e-mail para livros@algaworks.com.

# Ajude na continuidade desse trabalho

Escrever um livro dá muito trabalho, por isso, esse projeto só faz sentido se muitas pessoas tiverem acesso a ele.

Ajude a divulgar esse livro especial de Natal para seus amigos que também se interessam por programação Java. Compartilhe no Facebook e Twitter!


# Sumário

Capítulo 1

# Introdução a orientação a objetos

## 1.1. Para quem é esse livro?

Parabéns por se interessar em aprender orientação a objetos.

Esse é um dos principais investimentos que qualquer desenvolvedor precisa fazer, porque tentar aprender frameworks, ferramentas e bibliotecas sem conhecer muito bem a base, pode ser muito frustrante.

Mas antes de começar, é importante que você saiba que esse livro foi escrito para um público que já conhece pelo menos os fundamentos de Java.

Se você nunca instalou a JDK (Java Development Kit), nunca compilou uma classe Java ou não conhece a sintaxe da linguagem, esse livro **ainda** não é pra você.

Nesse caso, é melhor você estudar por algum outro material que inclua também os conceitos básicos, antes de ler esse livro.

Agora, se você já estudou Java na faculdade ou já instalou na sua máquina e executou alguns exemplos básicos, com certeza esse livro vai te ajudar bastante.

Bons estudos!. :)

## 1.2. O que é POO?

Programação Orientada a Objetos (POO) é um paradigma de programação que ajuda a definir a estrutura de programas de computadores, baseado nos conceitos do mundo real, sejam eles reais ou abstratos. A ideia é simular as coisas que existem e acontecem no mundo real, no mundo virtual.

O termo foi usado pela primeira vez na linguagem Smalltalk, criada por Alan Kay, mas algumas ideias já eram usadas na década de 1960 na linguagem Simula 67, criada por Ole Johan Dahl e Kristen Nygaard.

Atualmente, diversas linguagens de programação utilizam este paradigma, como por exemplo: Java, VB.NET, C#, C++, Object Pascal, Ruby, Python, etc.

Dentre as vantagens que a OO proporciona, podemos destacar o aumento de produtividade, reuso de código, redução das linhas de código programadas, separação de responsabilidades, encapsulamento, polimorfismo, componentização, maior flexibilidade do sistema, etc.

A Orientação a Objetos (OO) permite criar programas componentizados, separando as partes do sistema por responsabilidades e fazendo que essas partes se comuniquem entre si, por meio de mensagens. O programador é responsável por definir como os objetos devem se comportar e como eles devem interagir entre si.

## 1.3. Classes e objetos

Uma classe, nada mais é do que a descrição de um conjunto de entidades (reais ou abstratas) do mesmo tipo e com as mesmas características e comportamentos.

As classes definem a estrutura e o comportamento dos objetos daquele determinado tipo.

Podemos dizer que as classes são, na verdade, modelos de objetos do mesmo tipo. Por falar nisso, é comum usarmos também o termo "tipo" ao falar de "classe", apesar de que "tipo" é mais abrangente.

Para exemplificar, vamos imaginar a classe "Carro" do mundo real e trazer para o mundo virtual.

Um carro tem rodas, pneus, lanternas, portas, motor, pára-choque, etc.

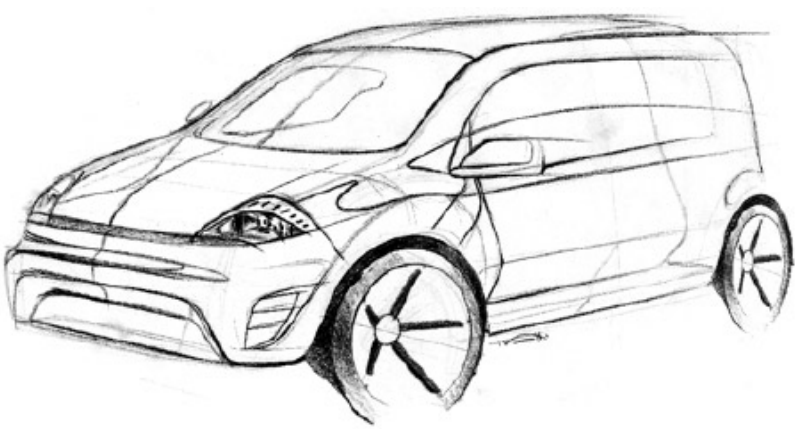
Qual é o modelo do carro? Qual é a cor do carro? Qual é a marca dos pneus? As lanternas estão ligadas? O motor do carro está funcionando? Não sabemos!

Na verdade, não é possível responder essas perguntas, pois ao pensar na classe "Carro", nós definimos apenas **o que o carro pode ter e o que ele faz**, e não o que realmente ele é.

Para ficar mais claro, pense como se não existissem carros no mundo. Você é o grande inventor dos carros!

Como qualquer invenção, você precisará fazer algum desenho, escrever um rascunho, criar algum projeto ou apenas formar uma ideia em sua cabeça de como "a coisa" deve funcionar.

Isso são as classes! Ideias, modelos, protótipos, rascunhos… algo que ainda não existe, mas que, a partir da ideia, alguém pode começar a construir o produto final (que existe de verdade).



Ao criar a classe "Carro", precisamos pensar em o que a classe deve ter (características/propriedades) e o que ela poderá fazer (comportamentos/ações).

Devemos pensar apenas no essencial para que o problema que estamos tentando resolver seja solucionado.

Por exemplo, podemos pensar que o carro deve ter:

- Fabricante
- Modelo
- Cor
- Tipo de combustível
- Ano de fabricação
- Valor de mercado

E agora os comportamentos do carro (o que ele pode fazer):

- Ligar
- Desligar
- Mudar marcha
- Acelerar
- Frear

Agora já temos a classe "Carro", mas ainda não podemos ligar, acelerar ou frear um carro, e nem mesmo saber o fabricante, modelo ou a cor dele, pois ainda não existe um carro. Tudo que existe é uma ideia de um carro, uma especificação!

Nós podemos construir um ou vários carros a partir da classe "Carro", e isso se chama **instanciação**, na orientação a objetos.


Quando usamos a classe para instanciar algo, o resultado final é um **objeto** do tipo da classe.

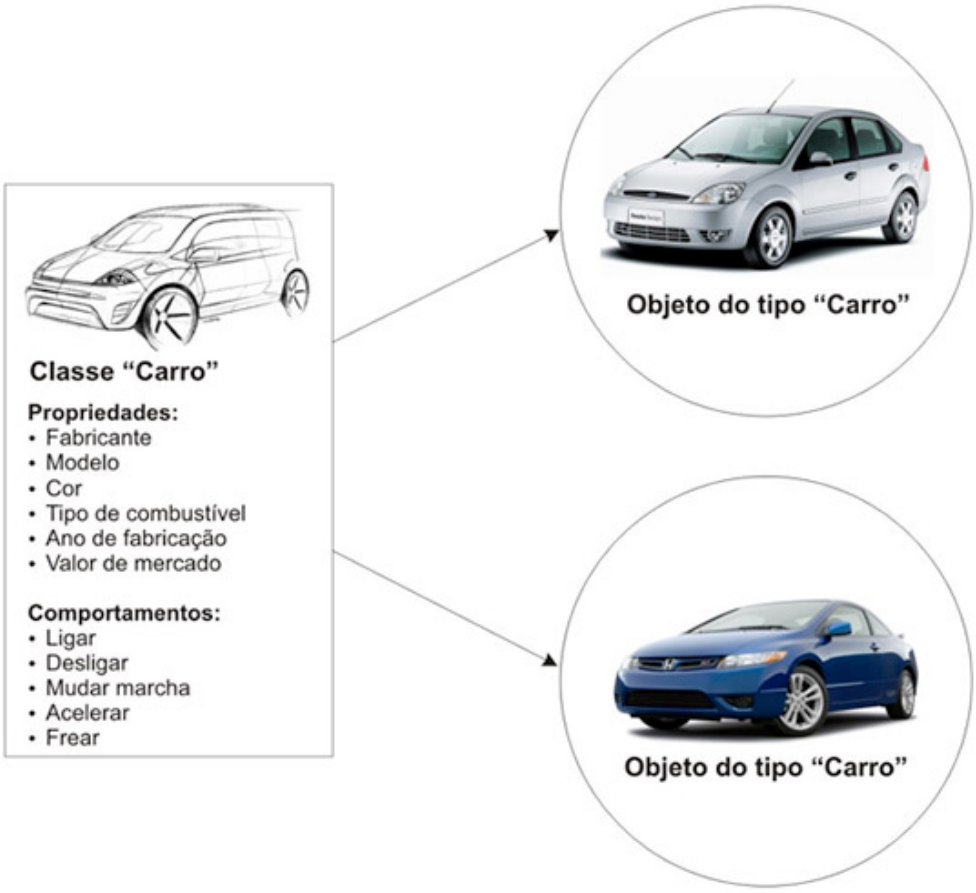
No caso do tipo "Carro", instanciamos um objeto que, errr… é um carro, mas agora real, que ocupa lugar no espaço, que acelera, freia, tem cor, modelo, fabricante, etc.

Um objeto nada mais é que uma instância particular de um tipo de dado específico (classe), ou seja, em outras palavras, objeto é uma entidade, do mundo computacional, que representa uma entidade do mundo real especificamente.

Os objetos possuem estados (atributos ou propriedades), comportamentos (métodos ou ações) e identidade (cada objeto é único).

O objeto que "instanciamos mentalmente" (pois ainda não estamos programando) é um Fiesta Sedan da Ford, tem cor prata, bicombustível e podemos pisar no acelerador e sair passeando por aí.

O Ford Fiesta que instanciamos é único no mundo! Podem existir outros muito parecidos (praticamente iguais), mas o carro da imagem acima tem uma identidade única, um chassi único, um único dono (ou dona) e o estado atual dele é só dele e de mais nenhum outro.




Os objetos se comunicam entre si por meio de mensagens (chamadas aos métodos, que veremos em breve).

Por exemplo, o João (um objeto do tipo “Motorista”) pode bater o carro dele (um objeto do tipo “Carro”) em seu carro (que é outro objeto do tipo “Carro”) se você (um objeto do tipo “Motorista”) parar bruscamente seu carro na frente.

Se você usar bem a imaginação (não é tão difícil assim), perceberá que você comunica com seu carro (acelerando, freando, etc), da mesma forma que o João também comunica com o carro dele, e os dois carros também podem se comunicar (através de uma colisão, por exemplo).

Podemos pensar no mundo real como um conjunto de “objetos” que interagem uns com os outros.

Por exemplo, em um ambiente de negócios, o cliente interage com o fornecedor, ambos interagem com produtos, com transportadoras, notas fiscais, que também interagem com órgãos do governo, e outros milhares de objetos.

A programação orientada a objetos é uma forma de criar sistemas "mapeando" as características e comportamentos de entidades do mundo real e como elas se comunicam, porém com um escopo mais limitado (apenas o que faz sentido para o domínio do problema).

## 1.4. Criando uma classe com atributos

Agora que você já sabe o que é uma classe, vamos criar a classe `Carro` em Java.

```
class Carro {

}
```

O código acima é referente ao tipo "Carro", mas não incluímos os atributos e métodos, por isso o carro que definimos não tem nenhuma característica e nem comportamentos.

Vamos agora incluir os atributos de um carro, também conhecidos como propriedades, características ou ainda variáveis de instância.

No caso do carro, os atributos que definimos são:

- Fabricante
- Modelo
- Cor
- Tipo de combustível
- Ano de fabricação
- Valor de mercado

O código Java fica da seguinte forma:

```
class Carro {

    String fabricante;
```

```
    String modelo;
    String cor;
    String tipoCombustivel;
    int anoDeFabricacao;
    double valorDeMercado;

}
```

Os atributos são variáveis com o escopo do objeto. Eles são acessíveis e estão disponíveis enquanto o objeto "possuir vida", e são iniciados durante a criação de seu objeto.

Neste caso, podemos chamá-los de variáveis de instância, pois só existirão valores para as variáveis quando existirem objetos (instâncias).

Os tipos dos atributos podem ser um tipo primitivo (como `int`, `double`, etc) ou um tipo de classe.

Por exemplo, `String` é uma classe (não é um tipo primitivo) que representa uma cadeia de caracteres, mas poderíamos definir um atributo do tipo `Motorista` ou `Proprietario` (se essas classes existissem) para outros atributos, se fosse o caso.

Lembre-se que a classe `Carro` que acabamos de criar não representa nenhum carro real (objeto), mas apenas especifica como os carros devem ser quando forem instanciados.

## 1.5. Instanciando objetos

O código-fonte da classe `Carro` compila com sucesso, mas não é possível executá-lo. Se você tentar executar a classe pelo Eclipse, receberá a mensagem de erro:

```
Editor does not contain a main type
```

Você pode ficar tentado a colocar um método `main()` na classe `Carro` para executá-la, mas isso é muito, mas muito errado.

A classe `Carro` não pode ter a responsabilidade de iniciar a execução de um programa, pois ela é apenas um dos muitos componentes que podemos ter em um sistema completo.

Quando você estiver programando uma aplicação real, você terá centenas ou milhares de classes que especificam pequenas partes do sistema (componentes). Essas classes se comunicam entre si para fornecer as funcionalidades completas do sistema.

Normalmente, você terá apenas uma classe com o método `main()` para iniciar o sistema. Esse método deve ser responsável por instanciar objetos de outras classes existentes e a partir daí esses objetos darão continuidade na execução do sistema.

Para desenvolver o método `main()`, é melhor criarmos uma classe que não tem nada a ver com o domínio de nosso problema (e muito menos com orientação a objetos). Vamos dar o nome `Principal` para essa classe.

```
class Principal {

    public static void main(String[] args) {

    }

}
```

Ainda não colocamos código dentro do método `main()`. Precisamos nos perguntar: o que queremos fazer?

Para começar, vamos apenas instanciar um objeto do tipo `Carro` e armazená-lo em uma variável.

```
class Principal {

    public static void main(String[] args) {
        Carro meuCarro; // declaração de variável
        meuCarro = new Carro(); // instanciação de um carro
    }

}
```

O código do exemplo acima apenas declara uma variável chamada `meuCarro` do tipo `Carro` e depois instancia um objeto do tipo `Carro` e atribui à variável `meuCarro`.

Usamos uma palavra `new` para indicar que desejamos instanciar um novo objeto do tipo `Carro`.

As variáveis não guardam os objetos, mas sim uma referência para a área de memória onde os objetos são alocados.

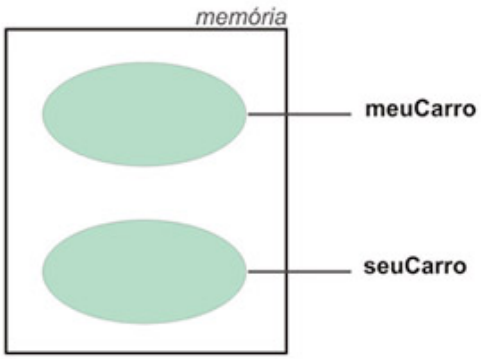
Se criarmos duas instâncias da classe `Carro` e atribuirmos cada instância para variáveis diferentes, `meuCarro` e `seuCarro`, temos o código abaixo.

```
class Principal {

    public static void main(String[] args) {
        Carro meuCarro = new Carro();
        Carro seuCarro = new Carro();
    }

}
```

Perceba que agora nós declaramos e instanciamos cada variável em apenas uma linha.

Quando executamos o último código, parece que nada acontece. Na verdade, não conseguimos visualizar nada na tela, pois não incluímos nenhuma instrução para isso, mas dois objetos são instanciados e referenciados pelas variáveis de nomes `meuCarro` e `seuCarro`.

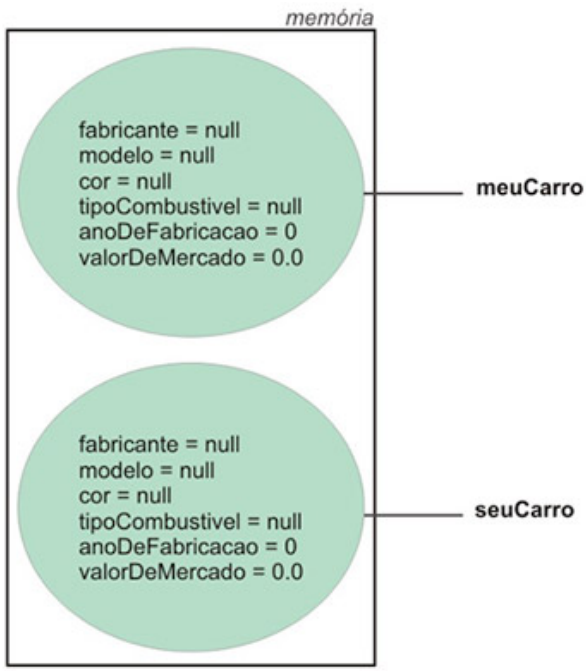
Os objetos instanciados são alocados na memória da máquina virtual Java, e os nomes de variáveis apenas apontam para esses objetos.




Veja que as variáveis referenciam objetos diferentes, apesar de serem do mesmo tipo.

Você pode estar se perguntando agora: Qual é a cor do carro? Qual é o preço de mercado? E o ano de fabricação?

Como nós não atribuímos valores para essas variáveis, elas receberam valores padrão. Por exemplo, o ano de fabricação recebeu o valor 0, valor de mercado recebeu o valor 0.0 e todas as variáveis do tipo `String` receberam o valor `null` (que indica "nada", ou seja, sem referência).



Veremos no próximo tópico como acessar os atributos de objetos para atribuição e leitura.

## 1.6. Acessando atributos de objetos

Quando instanciamos um objeto, é comum precisarmos acessar os valores de seus atributos para realizar cálculos, tomar decisões ou simplesmente mostrá-los na tela.

Para acessá-los, basta digitarmos o nome da variável que contém o objeto seguido por `.` (ponto) e o nome do atributo da instância. Veja um exemplo:

```
class Principal {

    public static void main(String[] args) {
        Carro meuCarro = new Carro();
        System.out.println("Modelo: " + meuCarro.modelo);
        System.out.println("Ano: " + meuCarro.anoDeFabricacao);
    }

}
```

A saída da execução do código acima é:

```
Modelo: null
Ano: 0
```

Nós conseguimos acessar os atributos `modelo` e `anoDeFabricacao` de `meuCarro` e imprimir na tela, porém não existia uma descrição do modelo e nem um ano de fabricação para o carro instanciado, por isso tudo que vimos foram os valores padrão.

Se desejarmos atribuir (modificar) os valores das variáveis da instância `meuCarro`, basta usarmos o operador de atribuição = (igual), conforme o exemplo abaixo:

```
Carro meuCarro = new Carro();
meuCarro.anoDeFabricacao = 2011;
meuCarro.cor = "Prata";
meuCarro.fabricante = "Fiat";
```

```
meuCarro.modelo = "Palio";
meuCarro.tipoCombustivel = "Bicombustível";
meuCarro.valorDeMercado = 30000;

System.out.println("Modelo: " + meuCarro.modelo);
System.out.println("Ano: " + meuCarro.anoDeFabricacao);
```

No exemplo acima, modificamos os valores de todas as variáveis de instância de `meuCarro` e depois acessamos `modelo` e `anoDeFabricacao` para exibir na tela.

A execução do código tem a seguinte saída na tela:

```
Modelo: Palio
Ano: 2011
```

Para que não restem dúvidas sobre o funcionamento das variáveis de instância, vamos fazer mais um exemplo, declarando e instanciando duas variáveis do tipo `Carro`.

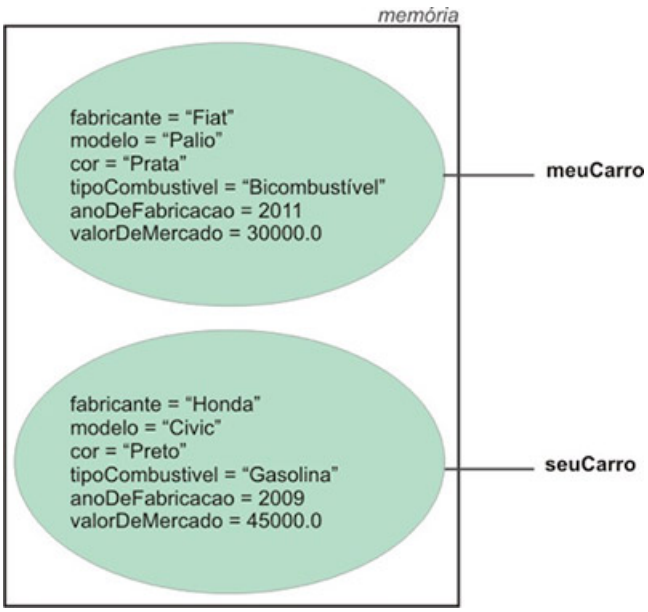
```
Carro meuCarro = new Carro();
meuCarro.anoDeFabricacao = 2011;
meuCarro.cor = "Prata";
meuCarro.fabricante = "Fiat";
meuCarro.modelo = "Palio";
meuCarro.tipoCombustivel = "Bicombustível";
meuCarro.valorDeMercado = 30000;

Carro seuCarro = new Carro();
seuCarro.anoDeFabricacao = 2009;
seuCarro.cor = "Preto";
seuCarro.fabricante = "Honda";
seuCarro.modelo = "Civic";
seuCarro.tipoCombustivel = "Gasolina";
seuCarro.valorDeMercado = 45000;

System.out.println("Meu carro");
System.out.println("----------------------");
System.out.println("Modelo: " + meuCarro.modelo);
System.out.println("Ano: " + meuCarro.anoDeFabricacao);

System.out.println();

System.out.println("Seu carro");
```

```
System.out.println("----------------------");
System.out.println("Modelo: " + seuCarro.modelo);
System.out.println("Ano: " + seuCarro.anoDeFabricacao);
```

O exemplo acima é bem simples, mas geralmente confunde algumas pessoas que estão iniciando os estudos em orientação a objetos.

Nós instanciamos dois carros, `meuCarro` e `seuCarro`, e atribuímos valores diferentes às variáveis de instância de cada um.

Veja abaixo uma representação das instâncias para entender melhor.




Veja que `meuCarro` e `seuCarro` têm os mesmos atributos (pois são do mesmo tipo), porém os valores de cada atributo podem ser diferentes.

É como na vida real! Todos os carros possuem esses mesmos atributos, mas os valores desses atributos normalmente são diferentes entre o seu carro, o carro do seu vizinho, o carro do seu pai, etc.

Agora que você entendeu, deve ficar claro para você que a execução do último exemplo exibe na console:

```
Meu carro
---------------------
Modelo: Palio
Ano: 2011

Seu carro
---------------------
Modelo: Civic
Ano: 2009
```

## 1.7. Composição de objetos

Imagine que desejamos incluir atributos do proprietário do carro na classe `Carro`. O código ficaria assim:

```
class Carro {

    String fabricante;
    String modelo;
    String cor;
    String tipoCombustivel;
    int anoDeFabricacao;
    double valorDeMercado;

    // atributos do proprietário
    String nomeProprietario;
    String cpfProprietario;
    int idadeProprietario;
    String logradouroProprietario;
    String bairroProprietario;
    String cidadeProprietario;

}
```

O exemplo acima até compila, mas não é bem visto por programadores de software mais experientes, pois a classe `Carro` passou a ter baixa coesão.

Dizemos que uma classe não está coesa quando ela tem responsabilidades demais, como é o caso do último código.

A classe `Carro` passou a guardar não só informações sobre o carro, mas também sobre o proprietário do carro e seu endereço.

No mundo real, um carro não tem nome, CPF, idade, logradouro, bairro ou cidade, mas sim o proprietário do carro. Por isso, vamos criar uma nova classe chamada `Proprietario` e incluir esses atributos.

```
class Proprietario {

    String nome;
    String cpf;
    int idade;
    String logradouro;
    String bairro;
    String cidade;

}
```

Agora alteramos a classe `Carro` para incluir um atributo que faz referência a `Proprietario`.

```
class Carro {

    String fabricante;
    String modelo;
    String cor;
    String tipoCombustivel;
    int anoDeFabricacao;
    double valorDeMercado;
    Proprietario dono;

}
```

A variável de instância `dono` recebeu esse nome apenas para não coincidir com o nome da classe, mas não teria nenhum problema se a variável tivesse o nome `proprietario`. Neste caso, achamos melhor os nomes não coincidirem para você entender que o nome da classe não tem nada a ver com o nome do atributo.

O que acabamos de fazer foi uma composição de objetos. Composição é uma forma de combinar objetos simples em objetos mais complexos.

Essa técnica traz grandes benefícios para a reutilização de objetos e legibilidade do código, além do código-fonte expressar melhor o que está acontecendo, de acordo com o mundo real.

Normalmente, podemos dizer que objetos compostos fazem parte de um relacionamento do tipo "tem um". Por exemplo, podemos dizer que o carro tem um proprietário.

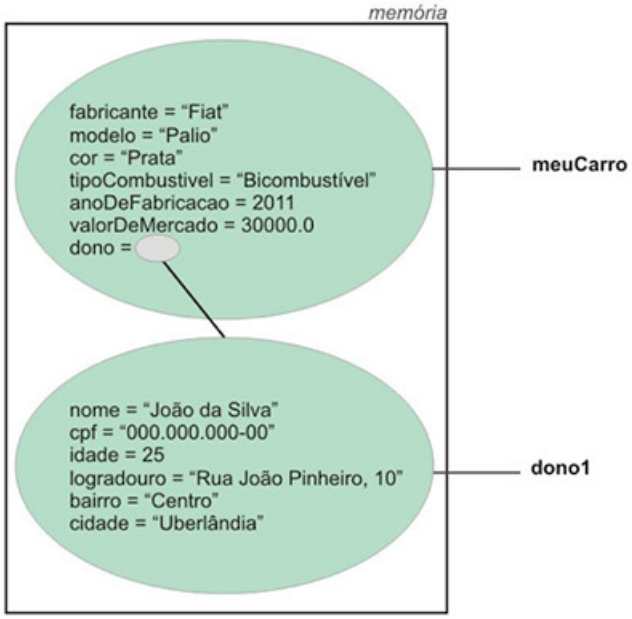
Olhando para a classe `Proprietario`, poderíamos ainda pensar em criar outra classe chamada `Endereco` para incluir o logradouro, bairro e cidade e a classe `Proprietario` incluiria um atributo do tipo `Endereco`.

Não há nada de errado em fazer isso, mas vamos deixar como está, pois no momento não faz muito sentido para nossos exemplos.

Agora vamos instanciar um carro e um proprietário dentro do método `main`:

```
Proprietario dono1 = new Proprietario();
dono1.nome = "João da Silva";
dono1.cpf = "000.000.000-00";
dono1.idade = 25;
dono1.logradouro = "Rua João Pinheiro, 10";
dono1.bairro = "Centro";
dono1.cidade = "Uberlândia";

Carro meuCarro = new Carro();
meuCarro.anoDeFabricacao = 2011;
meuCarro.cor = "Prata";
meuCarro.fabricante = "Fiat";
meuCarro.modelo = "Palio";
meuCarro.tipoCombustivel = "Bicombustível";
meuCarro.valorDeMercado = 30000;
meuCarro.dono = dono1; // atribuímos o dono do carro
```

Veja no exemplo acima que instanciamos um `Proprietario` e depois um `Carro` isoladamente, e apenas na última linha que atribuímos `dono1` à variável `dono` de `meuCarro`.

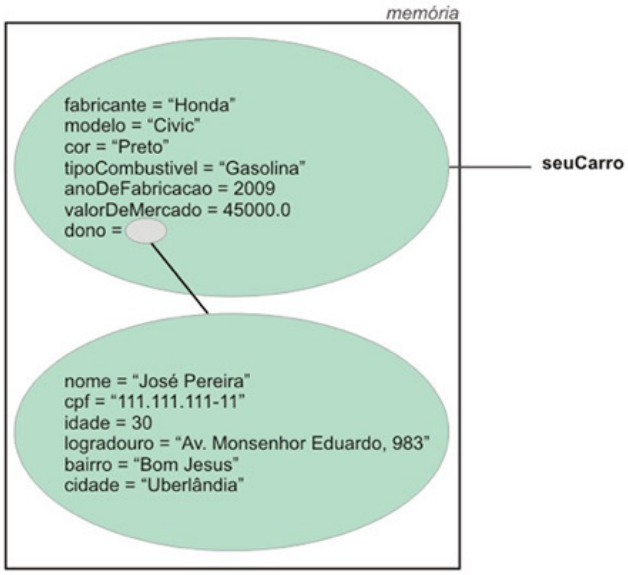



Outra forma comum de atribuir valores às variáveis de `dono` é como no exemplo abaixo:

```
Carro seuCarro = new Carro();
seuCarro.anoDeFabricacao = 2009;
seuCarro.cor = "Preto";
seuCarro.fabricante = "Honda";
seuCarro.modelo = "Civic";
seuCarro.tipoCombustivel = "Gasolina";
seuCarro.valorDeMercado = 45000;

// atribuindo valores às variáveis do dono do seuCarro
seuCarro.dono = new Proprietario();
seuCarro.dono.nome = "José Pereira";
seuCarro.dono.cpf = "111.111.111-11";
seuCarro.dono.idade = 30;
seuCarro.dono.logradouro = "Av. Monsenhor Eduardo, 983";
seuCarro.dono.bairro = "Bom Jesus";
seuCarro.dono.cidade = "Uberlândia";
```

Veja que no exemplo acima nós instanciamos `Proprietario` e atribuímos diretamente à variável `dono` de `seuCarro`.

O resultado final é o mesmo, apenas não usamos outra variável intermediária para armazenar o proprietario instanciado.




## 1.8. Valores padrão

Você já estudou que, quando instanciamos um objeto usando o operador `new`, as variáveis de instância são inicializadas com valores padrão (*default*). Vamos fazer uma pequena revisão para lembrar os valores que cada tipo de variável recebe:

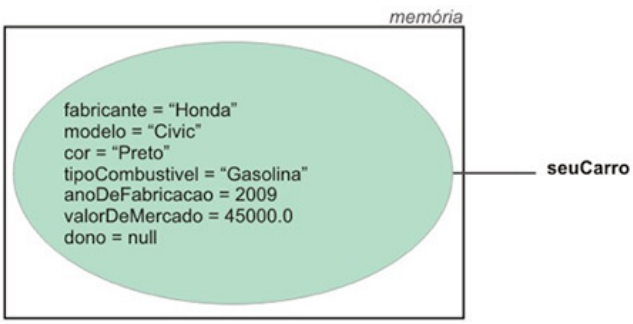
- Tipos numéricos primitivos recebem `0` ou `0.0`
- Tipo booleano recebe `false`
- Referências a objetos (`String`, `Proprietario`, `Carro`, etc) recebem `null`

No exemplo abaixo, tentamos atribuir um valor à variável `nome` em `dono` de `seuCarro`.

```
Carro seuCarro = new Carro();
seuCarro.anoDeFabricacao = 2009;
seuCarro.cor = "Preto";
seuCarro.fabricante = "Honda";
seuCarro.modelo = "Civic";
seuCarro.tipoCombustivel = "Gasolina";
```

```
seuCarro.valorDeMercado = 45000;

// lança erro em tempo de execução
seuCarro.dono.nome = "José Pereira";
```

Veja na representação do objeto referenciado em `seuCarro` que não existe um proprietário (dono) associado:




A execução do código do exemplo acima gera uma exceção (um erro) em tempo de execução chamado `NullPointerException`.

```
Exception in thread "main" java.lang.NullPointerException
        at Principal3.main(Principal3.java:11)
```

É importante você entender que não podemos atribuir um valor em uma referência nula (que não possui objeto associado).

Se quisermos evitar o `NullPointerException`, podemos deixar a variável `dono` com um valor padrão, instanciando um novo `Proprietario`.

Vamos aproveitar e inicializar também as outras variáveis da classe `Carro`.

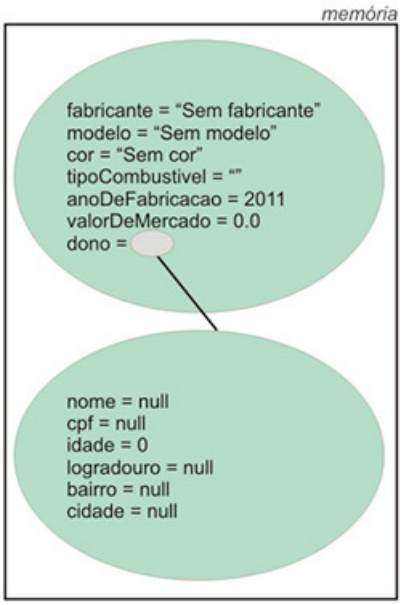
```
class Carro {

    String fabricante = "Sem fabricante";
    String modelo = "Sem modelo";
    String cor = "Sem cor";
    String tipoCombustivel = "";
    int anoDeFabricacao = 2011;
    double valorDeMercado = 0;
    Proprietario dono = new Proprietario();
```

```
}
```

Para inicializar as variáveis de instância, basta incluir o símbolo = (igual) seguido pelo valor ou pela instanciação do objeto logo após a declaração da variável.

No caso da classe `Carro`, não faz muito sentido inicializar as variáveis com esses valores, mas mesmo assim fizemos a título de exemplo.

Agora, quando chamamos `new Carro()`, o novo objeto instanciado tem a seguinte representação (automaticamente):




Agora é seguro executar o código abaixo sem se preocupar com a exceção `NullPointerException`.

```
Carro seuCarro = new Carro();
seuCarro.dono.nome = "José Pereira";
```

Apesar de parecer mais fácil sempre instanciar os objetos com valores padrão nas variáveis de instância de uma classe, conceitualmente pode estar incorreto, pois neste caso, seria o mesmo que dizer que todos os carros que acabaram de ser fabricados já possuem um dono automaticamente (o que pode não ser verdade).

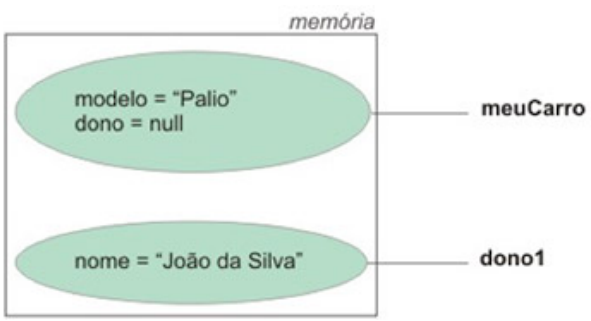
## 1.9. Variáveis referenciam objetos

Quando instanciamos um objeto e atribuímos a uma variável, a variável não armazena o objeto, mas **faz referência** ao objeto. Esse conceito é muito importante para a programação orientada a objetos.

Veja os exemplos abaixo e as representações dos objetos na memória da máquina virtual. Não incluímos atribuições para algumas variáveis apenas para deixar o exemplo mais curto e fácil de entender.

Primeiramente, instanciamos um `Proprietario` e um `Carro` e atribuímos às variáveis `dono1` e `meuCarro`, respectivamente.

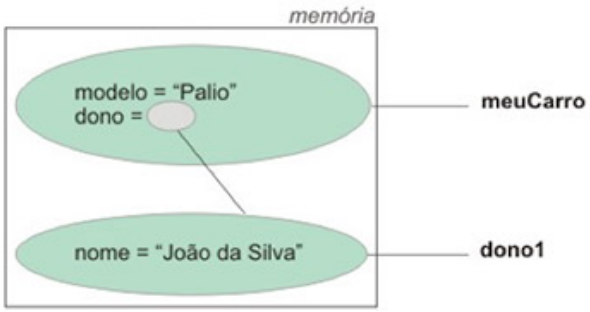
```
Proprietario dono1 = new Proprietario();
dono1.nome = "João da Silva";


Carro meuCarro = new Carro();
meuCarro.modelo = "Palio";
```




Veja que os dois objetos não possuem nenhuma associação. O `meuCarro` não possui um dono (pois a variável `dono` está nula).

Quando atribuímos `dono1` em `meuCarro.dono`, o proprietário "João da Silva" passa a ser referenciado também pela variável `dono` de `meuCarro`.

```
meuCarro.dono = dono1;
```




É importante entender que o objeto não é copiado (duplicado), mas apenas uma nova referência é feita até ele. Dessa forma, é correto dizer que:

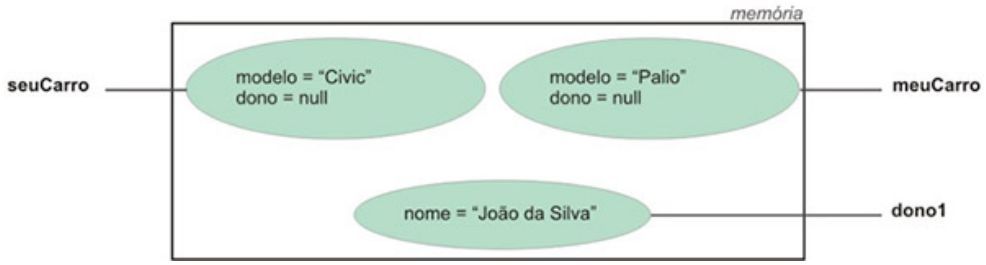
```
// a linha abaixo
meuCarro.dono.nome = "Maria Joaquina";

// tem o mesmo efeito que
dono1.nome = "Maria Joaquina";
```

Quando dizemos que as duas instruções têm o mesmo efeito, quer dizer que o nome "João da Silva" será substituído por "Maria Joaquina" no mesmo objeto, pois só existe um objeto do tipo `Proprietario`, que é o que representa o "João da Silva".

Para ficar ainda mais claro, vamos a mais um exemplo.

```
Proprietario dono1 = new Proprietario();
dono1.nome = "João da Silva";

Carro meuCarro = new Carro();
meuCarro.modelo = "Palio";

Carro seuCarro = new Carro();
seuCarro.modelo = "Civic";
```

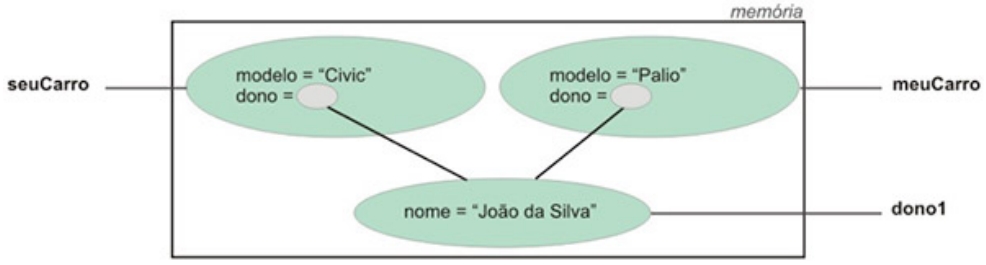
Instanciamos um `Proprietario` e dois objetos do tipo `Carro`, porém ainda não associamos nenhum deles.




Agora associamos a variável `dono1` à variável `dono` de `meuCarro` e `seuCarro`.

```
meuCarro.dono = dono1;
seuCarro.dono = dono1;
```

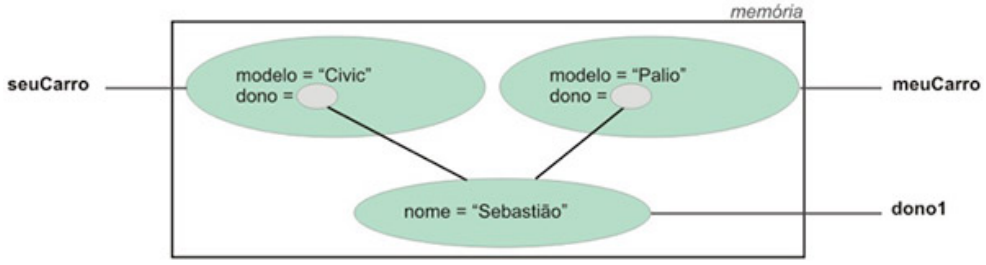
Quando fazemos isso, estamos dizendo que `meuCarro` e `seuCarro` possuem o mesmo `dono` (objeto do tipo `Proprietario`), que é o "João da Silva".




Chegamos a uma parte muito importante. Vamos mudar o nome do `dono` do `meuCarro` para "Sebastião".

```
meuCarro.dono.nome = "Sebastião";
```

Veja agora a representação dos objetos.



Quando mudamos o nome do dono do `meuCarro` para "Sebastião", na verdade estávamos mudando também o nome do dono do `seuCarro`, pois é o mesmo objeto, ou seja, é o mesmo dono, exatamente como acontece no mundo real.

Imagine se no mundo real o João tem dois carros, um do modelo Palio e outro do modelo Civic. Se por algum motivo estranho, o João conseguir mudar seu nome para Sebastião, ele ainda é o dono dos dois carros, porém agora ele tem um novo nome.

Se o João ficou doente, é o dono do Civic ou o dono do Palio que está doente? Ahãaa… agora você entendeu!

## 1.10. Criando e chamando métodos

Os comportamentos das classes são definidos através de métodos. Métodos são trechos de código que são referenciados por um nome e podem ser chamados por alguma outra parte do software através de seu nome.

Os métodos podem receber argumentos (valores e objetos de entrada) e retornar um valor ou objeto para quem o chamou.

A sintaxe dos métodos é semelhante às das funções de um programa procedural, mas em orientação a objetos, não é comum chamá-los de funções.

Nós definimos anteriormente que os comportamentos do carro devem ser:

- Ligar
- Desligar
- Mudar marcha
- Acelerar
- Frear

Cada comportamento dessa lista deve se tornar um método na classe `Carro`. Primeiramente, vamos implementar o método `ligar`:

```
void ligar() {
    System.out.println("Ligando carro " + modelo);
}
```

No método `ligar`, usamos a palavra-reservada `void` para dizer que o método não tem retorno, ou seja, nenhuma informação será retornada para quem chamar (invocar) o método. Estudaremos outros tipos de retorno adiante.

Após o nome do método, abrimos e fechamos os parênteses para dizer que esse método não recebe argumentos, ou seja, quem invocar este método não terá opção de enviar informações (valores ou objetos) a ele.

O código da programação do método deve estar dentro de um bloco, delimitado pela abertura e fechamento de chaves, mesmo que seja apenas uma linha, como é o caso do exemplo acima.

Para não complicar neste momento, programamos o método `ligar` para apenas exibir uma mensagem dizendo que o carro de um determinado modelo está sendo ligado. Preste atenção que `modelo` é uma variável de instância da classe `Carro`.

Para chamar o método `ligar`, precisamos de uma instância de carro. Vamos programar a chamada no método `main` da classe `Principal`.

```
public static void main(String[] args) {
    Carro seuCarro = new Carro();
    seuCarro.modelo = "Civic";
    // atribuição de outras variáveis de instância aqui...

    // chamando o método ligar
    seuCarro.ligar();
}
```

Veja que para chamar um método, precisamos indicar o nome da variável que aponta para o objeto (neste caso, `seuCarro`), seguido por ponto e o nome do método. A abertura e fechamento de parênteses indicam que não será passado nenhum argumento ao método.

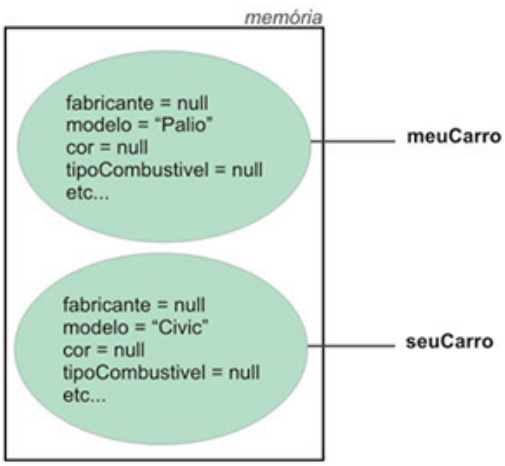
Ao executar a classe `Principal`, a saída na console é:

```
Ligando carro Civic
```

Agora vamos alterar a classe `Principal` para instanciar dois objetos do tipo `Carro` e chamar o método `ligar` em cada um deles.

```
public static void main(String[] args) {
    Carro seuCarro = new Carro();
    seuCarro.modelo = "Civic";

    Carro meuCarro = new Carro();
    meuCarro.modelo = "Palio";

    seuCarro.ligar();
    meuCarro.ligar();
}
```

Veja a imagem abaixo que representa os objetos que acabamos de instanciar no código-fonte acima. Perceba que `meuCarro` é um Palio e `seuCarro` é um Civic.




Milagrosamente, ao executarmos a classe `Principal`, temos seguinte saída:

```
Ligando carro Civic
Ligando carro Palio
```

Muitas pessoas iniciantes em programação orientada a objetos "travam" exatamente aqui. É comum elas perguntarem "Mas como pode? Palio não substituiu Civic quando atribuí à variável?".

Lembre-se que estamos programando orientado a objetos. Pense no mundo real!

Se o meu carro é um Palio e o seu é um Civic (ambos são do tipo `Carro`, não é mesmo?), ao sair de casa para o trabalho com meu carro, qual deles eu vou ligar? Claro que é o Palio, que é meu. O mesmo acontece com você… ao ligar o seu carro, é o Civic que levará você onde quiser.

## 1.11. Nomeando métodos

Os métodos em Java podem conter letras, dígitos, _ (underscore) e `$` (dólar), porém eles não podem ser iniciados por um dígito e não podem ser palavras reservadas.

Veja alguns nomes de métodos válidos para o compilador Java:

```
void conferirEstoque() // pode iniciar com letras minúsculas
void despacha_produtos() // pode ter underscore
void DESLIGAR() // pode ser todo em letras maiúsculas
void ExibirSaldo() // pode ter letras maiúsculas e minúsculas
void $salvardados() // pode iniciar com dólar
void _exibirFoto() // pode iniciar com underscore
void listar_alunos_nota_10() // pode ter dígitos
```

Agora alguns nomes de métodos inválidos (que nem compila):

```
void 2calcular() // não pode iniciar com dígitos
void ver extrato() // não pode ter espaços
void new() // new é uma palavra reservada do Java
```

Apesar da linguagem suportar letras maiúsculas e minúsculas, underscore, dólar e dígitos nos nomes dos métodos, a convenção de código Java diz que eles devem ser nomeados com a inicial em letra minúscula e as demais iniciais das outras palavras em letras maiúsculas.

Além disso, os métodos devem usar verbos (se possível no infinitivo), pois assim fica fácil ler e entender a ação que ele se propõe a fazer. Veja alguns exemplos:

```
void verExtrato()
void listarAlunosNota10()
```

```
void despacharPedido()
void exibirFotoEmMiniatura()
```

É uma boa prática escrever as frases e palavras completas quando vamos declarar métodos em Java. Abreviações devem ser usadas somente se forem muito conhecidas no domínio do negócio. Por exemplo, você deve evitar:

```
void verificar() // verifica o que? Que tal verificarEstoque?
void lstAlu() // listar alunos? Até que seu cérebro interprete isso...
void sCliente() // que isso? O que se pretende fazer com o cliente?
void calcQtdCliInat() // não é natural ler isso
```

Ao programar, lembre-se dessas convenções. Os próximos programadores que forem pegar o seu código para dar manutenção agradecem. :)

## 1.12. Métodos com retorno

A execução de um método retorna para o código que o chamou, quando acontece qualquer uma das seguintes situações:

- O bloco de código do método chega ao fim
- Uma instrução `return` é encontrada
- Uma exceção é lançada

Para um método poder retornar um valor, precisamos informar o tipo de retorno na declaração dele.

A classe `Paciente`, definida no exemplo abaixo, possui um método `calcularIndiceDeMassaCorporal` que retorna um valor do tipo `double`, indicando o IMC do objeto (paciente).

```
class Paciente {

    double peso;
    double altura;

    double calcularIndiceDeMassaCorporal() {
        double imc = peso / (altura * altura);
```

```
        // retornamos o valor calculado do tipo double
        return imc;
    }

}
```

A instrução `return` no exemplo acima é obrigatória. Quando um método define um tipo de retorno, o método **deve** retornar sempre algum valor.

O tipo do valor retornado deve sempre coincidir com o tipo do retorno declarado no método, ou pelo menos ser possível fazer uma conversão implícita.

Por exemplo, um método declarado para retornar um tipo `int` não pode devolver um valor do tipo `long`, `double` ou `float`, mas pode retornar um valor do tipo `int`, `short`, `char` ou `byte`.

O método de cálculo do IMC de um paciente pode ser chamado através de outra classe, que pode obter o retorno e usá-lo como desejar. Veja o exemplo abaixo:

```
public static void main(String[] args) {
    Paciente p = new Paciente();
    p.peso = 70.5;
    p.altura = 1.75;

    // invocamos o método e guardamos o retorno em uma variável
    double imcCalculado = p.calcularIndiceDeMassaCorporal();

    System.out.println("IMC do paciente: " + imcCalculado);
}
```

O exemplo do método de cálculo do IMC retorna um tipo primitivo `double`, mas podemos também retornar objetos.

Por exemplo, imagine que ao invés do método de cálculo do IMC retornar apenas o índice (um número), queremos retornar várias informações que obtemos durante a análise do IMC. Para isso, criamos uma classe chamada IMC com alguns atributos de instância. Veja:

```
class IMC {

    double indice;
```

```
    boolean abaixoDoPesoIdeal;
    boolean pesoIdeal;
    boolean obeso;
    String grauObesidade;

}
```

Agora o método de cálculo foi alterado para retornar um objeto do tipo IMC contendo dados já analisados, baseado no índice obtido.

```
IMC calcularIndiceDeMassaCorporal() {
    // instanciamos um objeto do tipo IMC
    IMC imc = new IMC();

    // calculamos o índice de massa corporal
    double indice = peso / (altura * altura);

    // analisamos o índice do paciente
    if (indice < 18.5) {
        imc.abaixoDoPesoIdeal = true;
    } else if (indice < 25) {
        imc.pesoIdeal = true;
    } else {
        imc.obeso = true;

        if (indice < 30) {
            imc.grauObesidade = "Acima do peso";
        } else if (indice < 35) {
            imc.grauObesidade = "I";
        } else if (indice < 40) {
            imc.grauObesidade = "II";
        } else {
            imc.grauObesidade = "III";
        }
    }

    // retornamos o objeto do tipo IMC com a análise feita
    return imc;
}
```

Ao chamar o método `calcularIndiceDeMassaCorporal`, recebemos não só o índice, mas também dados analisados indicando se o índice representa obesidade, peso abaixo do ideal ou peso ideal.

Essa é uma boa prática de programação, pois reduz a possibilidade de ter essa mesma lógica para analisar o IMC em diversos pontos do seu código-fonte.

Agora é só chamar o método e verificar os atributos do objeto.

```
public static void main(String[] args) {
    Paciente p = new Paciente();
    p.peso = 100;
    p.altura = 1.65;

    IMC imc = p.calcularIndiceDeMassaCorporal();

    System.out.println("Abaixo do peso ideal: "
        + imc.abaixoDoPesoIdeal);
    System.out.println("Peso ideal: " + imc.pesoIdeal);
    System.out.println("Obeso: " + imc.obeso);
    System.out.println("Grau de obesidade: " + imc.grauObesidade);
}
```

O resultado da execução do código acima é:

```
Abaixo do peso ideal: false
Peso ideal: false
Obeso: true
Grau de obesidade: II
```

Métodos que declaram `void` como seu tipo de retorno não podem retornar nenhum valor. Nesse caso, nós não precisamos incluir uma instrução `return`.

Mesmo assim, se em algum momento desejarmos sair da execução do método atual e voltar para quem chamou, podemos simplesmente usar a instrução `return` sem nenhum valor. Veja um exemplo:

```
void ligar() {
    if (modelo == null) {
        return;
    }

    System.out.println("Ligando carro " + modelo);
}
```

O método acima pára de ser executado se a variável de instância modelo for nula, pois não faz sentido ligar o carro se ele não tiver um modelo.

Apesar de parecer interessante, na maioria das vezes não é nada elegante usar o `return` para sair no meio de um método. As boas práticas dizem que um método deve ter um único ponto de saída, ou seja, apenas uma instrução `return`.

Podemos fazer um refactoring no exemplo acima para deixar o código mais legível e dentro das boas práticas de programação. Veja abaixo:

```
void ligar() {
    if (modelo != null) {
        System.out.println("Ligando carro " + modelo);
    }
}
```

## 1.13. Passando argumentos para métodos

Ao declarar métodos em Java, podemos definir alguns argumentos que devem ser passados para executá-los.

No exemplo abaixo, criamos um método que calcula o valor do salário de um funcionário baseado no número de horas normais e horas extras trabalhadas e nos valores em reais combinados por hora.

```
class FolhaPagamento {

    double calcularSalario(int horasNormais, int horasExtras,
        double valorHoraNormal, double valorHoraExtra) {

        // cálculo do salário
        double valorHorasNormais = horasNormais * valorHoraNormal;
        double valorHorasExtras = horasExtras * valorHoraExtra;

        return valorHorasNormais + valorHorasExtras;
    }

}
```

Um termo que falamos bastante quando queremos especificar um determinado método, juntamente com o seu tipo de retorno e seus parâmetros é "assinatura do método". A assinatura do método do último exemplo é a seguinte:

```
double calcularSalario(int, int, double, double)
```

O método `calcularSalario` recebe 4 argumentos, sendo os dois primeiros do tipo `int` e os dois últimos do tipo `double`, e retorna um valor do tipo `double`.

Os parâmetros de um método devem ter seus nomes logo após a definição de seus tipos. Os nomes não podem repetir e também não podem coincidir com nomes de variáveis locais do método.

Para chamar um método que possui parâmetros, basta informar o nome dele seguido pelos argumentos, que devem ficar entre parênteses e separados por vírgulas. Veja:

```
public static void main(String[] args) {
    FolhaPagamento folha = new FolhaPagamento();
    double salario = folha.calcularSalario(160, 12, 32.5, 40.2);

    System.out.println("Salário calculado: " + salario);
}
```

A saída da execução do código acima é:

```
Salário calculado: 5682.4
```

Você pode usar qualquer tipo para um parâmetro de um método, incluindo tipos primitivos ou tipos de classes.

Apesar de, tecnicamente, não existir um número máximo de parâmetros que possa ser usado em métodos, não é recomendado que você defina mais que 2 ou 3 parâmetros, pois ao passar muitos argumentos, a legibilidade do código é afetada e normalmente indica um mal cheiro (*Code smell*).

## 1.14. Argumentos por valor ou por referência

Quando passamos argumentos de tipos primitivos para um método, eles são passados por valor.

Isso quer dizer que qualquer alteração nos valores dos argumentos recebidos tem efeito apenas no escopo do método, mas não fora dele. Veja um exemplo:

```
class Produto {

    void definirPreco(double precoCusto) {
        // adiciona 20% de impostos
        precoCusto = precoCusto * 1.20;

        // faz várias outras coisas...
    }

}
```

O método `definirPreco` do código-fonte acima altera o valor da variável `precoCusto`, que foi recebida como parâmetro do método.

O método poderia fazer várias outras coisas para ter algum sentido, mas queremos apenas testar a passagem de parâmetro, por isso ignoramos o restante do código.

O código abaixo instancia um `Produto` e invoca o método `definirPreco`, passando o valor da variável `preco` como argumento.

```
public static void main(String[] args) {
    double preco = 140;

    Produto p = new Produto();
    p.definirPreco(preco);

    System.out.println("Preço: " + preco);
}
```

Ao executar esse exemplo, a saída é:

```
Preço: 140.0
```

Veja que o método `definirPreco` alterou o valor do argumento, que teve efeito apenas dentro do método, pois foi passado por valor. Não existem outras possibilidades quando estamos trabalhando com tipos primitivos.

Quando passamos um objeto como argumento para um método, uma referência do objeto é passada. Isso quer dizer que o objeto passado por quem chama o método é o mesmo objeto que é acessado dentro do corpo do método. Sendo assim, qualquer alteração nos valores dos atributos do objeto dentro da execução do método reflete também do lado de fora (de quem chamou), pois na realidade o objeto é o mesmo.

Para exemplificar, criamos uma classe `Preco` que possui vários atributos que fazem a composição de um preço de venda.

```
class Preco {

    double valorCustos;
    double valorImpostos;
    double valorLucro;
    double precoVenda;

}
```

O método `definirPreco` da classe `Produto` receberá um objeto do tipo `Preco` como parâmetro, além do percentual de impostos e margem de lucro a serem usados para calcular o preço de venda.

Veja no código-fonte abaixo que as variáveis de instância de `preco` estão sendo alteradas a partir dos cálculos feitos.

```
class Produto {

    void definirPreco(Preco preco, double percentualImpostos,
            double margemLucro) {
        preco.valorImpostos = preco.valorCustos
            * (percentualImpostos / 100);
        preco.valorLucro = preco.valorCustos
            * (margemLucro / 100);
        preco.precoVenda = preco.valorCustos + preco.valorImpostos
            + preco.valorLucro;
```

```
        // faz várias outras coisas...
    }

}
```

A classe principal, que possui o método `main`, instancia um objeto do tipo `Preco` e atribui um valor para a variável de instância `valorCustos`.

Depois disso, chama o método `definirPreco` da classe `Produto`. O objeto `preco` é passado como parâmetro, e ao executar o método, as variáveis desse objeto são alteradas, refletindo no resultado visualizado pelo usuário.

```
public static void main(String[] args) {
    Preco preco = new Preco();
    preco.valorCustos = 140;

    Produto produto = new Produto();
    produto.definirPreco(preco, 20, 15);

    System.out.println("Valor custos: " + preco.valorCustos);
    System.out.println("Valor impostos: " + preco.valorImpostos);
    System.out.println("Valor lucro: " + preco.valorLucro);
    System.out.println("Preço venda: " + preco.precoVenda);
}
```

A execução do último exemplo exibe na saída:

```
Valor custos: 140.0
Valor impostos: 28.0
Valor lucro: 21.0
Preço venda: 189.0
```

O método `main` passou uma referência de um objeto do tipo `Preco` para o método `definirPreco`. Esse objeto foi alterado dentro do método, e quando a execução voltou para o `main`, o objeto encontrava-se alterado.

Vamos fazer uma analogia com o mundo real. Imagine que você tenha um carro de cor vermelha, e você o emprestou para um amigo. Se o seu amigo mandar pintar seu carro de preto, mesmo que ele ainda não tenha devolvido para você, na prática, qual é a cor do seu carro? Quando você pegar o carro de volta, qual será a cor dele? Preto, claro! Isso quer dizer que você passou seu carro com todos

os atributos para seu amigo, e ele pode fazer o que quiser com ele (quase tudo), e isso refletirá no seu carro quando você o ver novamente.

## 1.15. Métodos que alteram variáveis de instância

Os métodos declarados em uma classe podem, além de alterar variáveis locais e as recebidas como parâmetro (que também possuem o escopo local), alterar as variáveis de sua própria instância.

Veja a classe `Aeronave`, declarada abaixo:

```
class Aeronave {

    int totalAssentos;
    int assentosReservados;

    void reservarAssentos(int numeroAssentos) {
        assentosReservados += numeroAssentos;
    }

    int calcularAssentosDisponiveis() {
        return totalAssentos - assentosReservados;
    }

}
```

Esta classe representa uma aeronave, e para simplificar, criarmos apenas dois atributos, `totalAssentos` e `assentosReservados`.

O método `reservarAssentos` recebe como parâmetro o número de assentos que devem ser reservados e altera a variável de instância `assentosReservados`, incrementando o valor dessa variável.

Criamos também um método `calcularAssentosDisponiveis`, que calcula o número de assentos livres na aeronave.

Agora vamos para a parte mais legal: programar um código que instancia aeronaves e usa os métodos que criamos.

```
public static void main(String[] args) {
    Aeronave aviaoGol = new Aeronave();
    aviaoGol.totalAssentos = 100;

    Aeronave aviaoLatam = new Aeronave();
    aviaoLatam.totalAssentos = 130;

    aviaoGol.reservarAssentos(10);
    aviaoLatam.reservarAssentos(5);

    int assentosGol = aviaoGol.calcularAssentosDisponiveis();
    int assentosLatam = aviaoLatam.calcularAssentosDisponiveis();

    System.out.println("Assentos disponíveis - Gol: " + assentosGol);
    System.out.println("Assentos disponíveis - LATAM: " + assentosLatam);
}
```

No código acima, instanciamos duas aeronaves, sendo uma da Gol e outra da LATAM.

A primeira tem capacidade de 100 passageiros, enquanto a segunda tem capacidade de 130 passageiros. Reservamos 10 assentos na aeronave da Gol e 5 assentos na aeronave da LATAM, por isso, ao executarmos o código, a saída na console deve ser a seguinte:

```
Assentos disponíveis - Gol: 90
Assentos disponíveis - LATAM: 125
```

Dá pra perceber que, quando invocamos um método que altera as variáveis de sua instância, isso é refletido exatamente no objeto em que chamamos o método, certo? Por isso que métodos, como os que declaramos, são chamados de métodos de instância.

## 1.16. O objeto this

A palavra reservada `this` faz referência ao próprio objeto, quando usado dentro de um método, por exemplo.

No código-fonte da classe `Aeronave`, vamos criar um método chamado `alterarTotalAssentos`, que receberá um argumento com o novo número de passageiros a ser atribuído à variável de instância `totalAssentos`.

```
class Aeronave {

    int totalAssentos;
    int assentosReservados;

    void reservarAssentos(int assentos) {
        assentosReservados += assentos;
    }

    int calcularAssentosDisponiveis() {
        return totalAssentos - assentosReservados;
    }

    void alterarTotalAssentos(int totalAssentos) {
        totalAssentos = totalAssentos;
    }

}
```

No código acima, existe um erro que pode passar despercebido. O método `alterarTotalAssentos` não tem efeito algum sobre a variável `totalAssentos`.

Veja que o nome do argumento é o mesmo da variável de instância. Quando fazemos a atribuição, estamos dizendo que o valor da variável local (recebida como parâmetro) deve ser alterado para ele mesmo!

Claro que, uma forma bem fácil de resolver isso, seria alterar o nome da variável do parâmetro e evitar essa confusão, mas muitas vezes, usar um nome diferente dificultaria a legibilidade do código.

Vamos resolver esse problema usando a palavra-chave `this`.

```
void alterarTotalAssentos(int totalAssentos) {
    this.totalAssentos = totalAssentos;
}
```

Agora sim! Estamos dizendo que queremos atribuir o valor da variável local à variável de instância. Quando usamos `this`, estamos deixando essa informação explícita.

## 1.17. Sobrecarga de métodos

Sobrecarga de métodos é um conceito simples da orientação a objetos, que permite a criação de vários métodos com o mesmo nome, mas com parâmetros diferentes.

Para exemplificar, vamos alterar a classe `Aeronave` para incluir a possibilidade de reserva de assentos normais e especiais. Para isso, criamos novas variáveis de instância.

```
class Aeronave {

    int totalAssentosNormais;
    int totalAssentosEspeciais;
    int assentosNormaisReservados;
    int assentosEspeciaisReservados;

    void reservarAssentos(int assentos) {
        this.assentosNormaisReservados += assentos;
    }

    int calcularAssentosDisponiveis() {
        return totalAssentosNormais - assentosNormaisReservados
            + totalAssentosEspeciais -assentosEspeciaisReservados;
    }

}
```

Veja que refatoramos os métodos `reservarAssentos` e `calcularAssentosDisponiveis` para usar as novas variáveis que criamos.

Já temos um método para fazer reserva de assentos, que reserva assentos normais na aeronave, mas e se quisermos reservar assentos especiais?

Poderíamos criar um método chamado `reservarAssentosEspeciais`, mas não vamos fazer isso. Vamos sobrecarregar o método `reservarAssentos`, incluindo uma nova versão com parâmetros adicionais. Veja:

```
void reservarAssentos(int assentosNormais, int assentosEspeciais) {
    this.assentosNormaisReservados += assentosNormais;
    this.assentosEspeciaisReservados += assentosEspeciais;
}
```

O método acima recebe, além do número de assentos normais, o total de assentos especiais a serem reservados. Isso é sobrecarga de métodos!

Temos duas versões de métodos com o nome `reservarAssentos`. Vamos ver as assinaturas desses métodos?

```
void reservarAssentos(int)
void reservarAssentos(int, int)
```

Sobrecarga de métodos é algo simples de ser feito, mas tem uma restrição que a própria linguagem impõe. Não é possível ter duas versões de métodos com a mesma assinatura.

Por exemplo, seria impossível ter o método a seguir na classe `Aeronave`:

```
void reservarAssentos(int assentosEspeciais) {
    this.assentosEspeciaisReservados += assentosEspeciais;
}
```

O código acima seria uma tentativa de criar uma versão do método `reservarAssentos`, para reservar assentos especiais, mas não funcionaria (nem compilaria), porque a classe `Aeronave` já possui um método `reservarAssentos` que recebe um `int`.

Capítulo 2

# Construtores e encapsulamento

## 2.1. Construtores

Os construtores fazem a função de iniciação dos objetos instanciados. Toda classe Java tem pelo menos um construtor.

Quando não declaramos explicitamente um construtor, a classe recebe um padrão, também conhecido como *construtor default*.

```
class Cliente {

    // esta classe tem construtor default

}
```

A sintaxe dos construtores é parecida com a de métodos, mas eles não possuem retorno e o nome deve ser **exatamente** igual ao da classe.

```
class Cliente {

    Cliente() {
        // isso é um construtor
    }

}
```

Os dois primeiros exemplos possuem construtores que não recebem argumentos e também não fazem nada! A diferença é que, no primeiro código, não

declaramos um construtor, e por isso foi usado o construtor padrão. No segundo exemplo, declaramos um construtor padrão (sem parâmetros), que substituiu o construtor implícito, mas que também não faz nada.

Agora, vamos imprimir uma mensagem dentro do construtor.

```
class Cliente {

    Cliente() {
        System.out.println("Construindo cliente...");
    }

}
```

Para chamar um construtor, basta instanciarmos um objeto da classe, com a palavra-chave `new`.

```
class TesteConstrutor {

    public static void main(String[] args) {
        Cliente cliente = new Cliente();
    }

}
```

A execução da classe `TesteConstrutor` exibirá na saída a mensagem "Construindo cliente…".

Podemos adicionar parâmetros aos construtores, assim como fazemos em métodos.

```
class Cliente {

    String nome;

    Cliente(String nome) {
        this.nome = nome;
    }

}
```

Quando criamos um construtor com parâmetros, como o exemplo acima, não conseguimos mais instanciar um objeto da classe usando o construtor sem parâmetros, pois o *construtor default* é anulado.

```
// não compila
Cliente cliente = new Cliente();
```

Agora, para instanciar um objeto do tipo `Cliente`, somos obrigados a passar como parâmetro o nome dele.

```
// compila
Cliente cliente = new Cliente("João das Couves");
```

Podemos ainda criar uma sobrecarga de construtores, adicionando um construtor sem parâmetros e outros com parâmetros diversos.

```
class Cliente {

    String nome;
    String cpf;

    Cliente() {
    }

    Cliente(String nome) {
        this.nome = nome;
    }

    Cliente(String nome, String cpf) {
        this.nome = nome;
        this.cpf = cpf;
    }

}
```

Agora podemos escolher qual construtor queremos usar na instanciação de objetos do tipo `Cliente`.

```
// usa o construtor padrão (sem parâmetros)
Cliente cliente1 = new Cliente();

// usa o construtor que recebe o nome
```

```
Cliente cliente2 = new Cliente("João das Couves");

// usa o construtor que recebe o nome e cpf
Cliente cliente3 = new Cliente("João das Couves", "12312312312");
```

Um construtor pode chamar outro, para aproveitar algum processamento. No código abaixo, o construtor `Cliente(String, String)` chama `Cliente(String)`, que então chama `Cliente()`.

```
class Cliente {

    String nome;
    String cpf;

    Cliente() {
        System.out.println("Construindo cliente...");
    }

    Cliente(String nome) {
        this();
        System.out.println("Cliente com nome " + nome);
        this.nome = nome;
    }

    Cliente(String nome, String cpf) {
        this(nome);
        System.out.println("Cliente com CPF " + cpf);
        this.cpf = cpf;
    }

}
```

A chamada de outro construtor, através do comando `this()`, deve ser a primeira instrução do construtor.

Para testar, vamos instanciar um novo cliente, usando o construtor que recebe o nome e CPF do cliente.

```
class TesteConstrutor {

    public static void main(String[] args) {
        Cliente cliente = new Cliente("João das Couves",
            "12312312312");
```

```
    }

}
```

A execução do último exemplo exibe na saída:

```
Construindo cliente...
Cliente com nome João das Couves
Cliente com CPF 12312312312
```

## 2.2. Modificadores de acesso public e private

Os modificadores de acesso em Java, também conhecidos como modificadores de visibilidade, permitem controlar o acesso a classes, atributos, métodos e construtores.

Existem 4 tipos diferentes de modificadores de acesso, mas por enquanto, estudaremos apenas o `public` e `private`.

### Modificadores em classes

Quando o modificador de acesso `public` é usado em uma classe, ela fica visível (acessível) para todas as outras classes.

Talvez seja um pouco difícil entender o que isso significa nesse momento, mas ficará mais claro no futuro, quando você aprender sobre os outros modificadores (*default* e *protected*, principalmente).

```
public class Produto {

}
```

A classe acima pode ser instanciada por *qualquer* outra classe, sem restrições.

Talvez você esteja se perguntando: "mas já não era assim?". Não! Quando não especificamos `public` na declaração da classe, ela não pode ser instanciada por qualquer outra classe. Existem restrições!

```
// esta classe não pode ser instancida por qualquer outra classe
class Produto {

}
```

Você pode ficar curioso(a), mas ainda não chegou a hora de você aprender mais sobre o modificador de acesso padrão, quando nenhum é especificado. Em um outro capítulo, você descobrirá a diferença.

O modificador `private` não pode ser usado em declarações de classes, com exceção de classes aninhadas, que não abordaremos neste livro.

## Modificadores em atributos, métodos e construtores

O modificador de acesso `public` pode ser usado em atributos, métodos e construtores.

```
public class Produto {

    public int estoque;

    public Produto(int estoque) {
        this.estoque = estoque;
    }

    public void zerarEstoque() {
        this.estoque = 0;
    }

}
```

No exemplo acima, o atributo `estoque`, o construtor e o método `zerarEstoque` podem ser acessados por qualquer outra classe.

```
Produto produto = new Produto(10);
produto.estoque = 20;
produto.zerarEstoque();
```

Mais uma vez, você entenderá melhor quando estudarmos outros modificadores.

O modificador `private` torna o membro privado, incapaz de ser acessado por outras classes, a não ser a própria classe a qual o membro pertence.

```
public class Produto {

    private int estoque;

    public Produto(int estoque) {
        this.estoque = estoque;
    }

    public void zerarEstoque() {
        this.estoque = 0;
    }

}
```

Veja que alteramos o modificador do atributo `estoque` para *private*. A partir de agora, não conseguimos mais acessar o atributo a partir de uma classe externa, mas o acesso pelo construtor e pelo método `zerarEstoque` são válidos, pois pertencem à mesma classe.

```
public class UmaClasseQualquer {

    public static void main(String[] args) {
        Produto produto = new Produto(10);
        produto.estoque = 20; // não compila
    }

}
```

Quando não queremos que um construtor seja usado por classes externas, podemos torná-lo privado, mas temos que deixar pelo menos um construtor acessível. Se não fizermos isso, tornamos impossível a instanciação de um objeto da classe.

```
public class Produto {

    private int estoque;

    public Produto() {
        this(10);
```

```
    }

    private Produto(int estoque) {
        this.estoque = estoque;
    }

    public void zerarEstoque() {
        this.estoque = 0;
    }

    public void imprimirEstoque() {
        System.out.println("Estoque: " + this.estoque);
    }

}
```

No último código, o único construtor público chama um construtor privado, que inicializa a quantidade em estoque com 10 unidades.

```
// chama construtor público
Produto produto = new Produto();

// imprime "Estoque: 10" na saída
produto.imprimirEstoque();
```

O código abaixo não compila, porque o construtor que tentamos usar é privado.

```
// não compila
Produto produto = new Produto(10);
```

Um método privado também só pode ser acessado pela própria classe, e é útil para, principalmente, reutilizar código em mais de um método.

```
public class Produto {

    private int estoque;

    public Produto(int estoque) {
        this.estoque = estoque;
        this.imprimirEstoque();
    }

    public void zerarEstoque() {
```

```
        this.estoque = 0;
        this.imprimirEstoque();
    }

    private void imprimirEstoque() {
        System.out.println("Estoque: " + this.estoque);
    }

}
```

No exemplo acima, o método `imprimirEstoque` foi definido como privado, e por isso, só pode ser acessado pela própria classe.

O construtor e o método `zerarEstoque` chamam o método `imprimirEstoque` depois de alterar a quantidade em estoque.

```
Produto produto = new Produto(20);
produto.zerarEstoque();
```

A execução do último exemplo exibe na saída:

```
Estoque: 20
Estoque: 0
```

O código abaixo não compila, pois `imprimirEstoque` é privado.

```
Produto produto = new Produto(20);
produto.zerarEstoque();
produto.imprimirEstoque(); // não compila
```

## 2.3. Encapsulamento

Encapsulamento é um dos conceitos fundamentais da orientação a objetos. É uma técnica que torna os atributos da classe privados e fornece acesso a eles através de métodos públicos.

Na seção anterior, acabamos usando conceitos de encapsulamento, sem mesmo citar sobre isso.

Como você já sabe, atributos privados só podem ser acessados pela própria classe, portanto, eles ficam "protegidos" de outras classes, e a única forma de acesso é pelos métodos públicos.

```
public class Produto {

    private int estoque;

    public void adicionarEstoque(int estoque) {
        this.estoque += estoque;
        this.verificarEstoqueMinimo();
    }

    public void retirarEstoque(int estoque) {
        this.estoque -= estoque;
        this.verificarEstoqueMinimo();
    }

    private void verificarEstoqueMinimo() {
        if (this.obterEstoque() < 5) {
            System.out.println("Abaixo do estoque mínimo: "
                + this.obterEstoque());
        }
    }

    public int obterEstoque() {
        return this.estoque;
    }

}
```

Na classe `Produto` acima, encapsulamos o atributo `estoque` e centralizamos o controle de estoque nos métodos `adicionarEstoque` e `retirarEstoque`.

Essa centralização facilita a manutenção do código, pois outras classes que precisarem adicionar ou retirar itens do estoque, não precisam se preocupar com a lógica por trás disso.

Se alguma classe precisar saber a quantidade de itens que um produto possui no estoque, poderá ainda usar o método `obterEstoque`.

```
Produto produto = new Produto();

produto.adicionarEstoque(10);
System.out.println(produto.obterEstoque());

produto.retirarEstoque(8);
System.out.println(produto.obterEstoque());
```

### Encapsulamento no mundo real

No mundo real, podemos notar o uso de encapsulamento em quase todos os objetos que usamos.

Por exemplo, a TV de sua casa. Existem milhares de componentes dentro de um aparelho de TV, mas nós interagimos apenas com uma interface pública, através do controle remoto ou botões no próprio equipamento.

Se quisermos ligar a TV, não precisamos conhecer os detalhes eletrônicos que estão por trás, não há necessidade de pegarmos uma chave de fenda e abrirmos a TV para ligar os circuitos que ativam a imagem e som. Basta apertarmos o botão "ligar", que é público. Os detalhes técnicos estão encapsulados, o botão "ligar" sabe o que deve ser feito tecnicamente, e dentro do próprio aparelho, existem muitos outros níveis de encapsulamento, para facilitar a manutenção e extensão das funcionalidades do equipamento.

## 2.4. JavaBeans

JavaBeans são classes Java reutilizáveis, que encapsulam as variáveis de instância em um único objeto (o bean).

Para uma classe Java ser considerada um JavaBean, ela deve ter um construtor padrão (sem argumentos) e permitir o acesso às variáveis de instância através de métodos acessores, conhecidos como *getters* e *setters*.

```
public class Produto {

    private String nome;
    private int estoque;
```

```
    public String getNome() {
        return this.nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public int getEstoque() {
        return this.estoque;
    }

    public void setEstoque(int estoque) {
        this.estoque = estoque;
    }

}
```

A classe `Produto` é um JavaBean! Veja que expomos os atributos `nome` e `estoque` através dos métodos *getters* e *setters*.

```
Produto produto = new Produto();
produto.setNome("Mouse sem fio");
produto.setEstoque(10);

System.out.println(produto.getNome() + " - " + produto.getEstoque());
```

Embora o padrão diga que os atributos devem ser expostos pelos métodos de acessibilidade, você não é obrigado a expor tudo!

Por exemplo, nossa classe `Produto` poderia expor os métodos de alteração de estoque usando formas mais orientadas a objetos, sem seguir a risca o padrão.

```
public class Produto {

    private String nome;
    private int estoque;

    public String getNome() {
        return this.nome;
    }
```

```
    public void setNome(String nome) {
        this.nome = nome;
    }

    public int getEstoque() {
        return this.estoque;
    }

    public void adicionarEstoque(int estoque) {
        this.estoque += estoque;
    }

    public void retirarEstoque(int estoque) {
        this.estoque -= estoque;
    }

}
```

Antes de sair gerando os *getters* e *setters* para todos os atributos, pense bem na necessidade disso.

Diversos *frameworks* de desenvolvimento exigem que os objetos que eles interagem sejam JavaBeans. Por isso, esse padrão é muito importante em Java.

Capítulo 3

# Pacotes e outros modificadores

## 3.1. Organizando os projetos em pacotes

Quando estamos desenvolvendo projetos reais, geralmente, temos centenas ou milhares de classes que se comunicam para executar o que o software se propõe. Surge então a necessidade de organizarmos nossas classes em diretórios diferentes, para separar sistemas, módulos ou responsabilidades.

Em Java, existe o conceito de *packages* (pacotes), que são diretórios e subdiretórios que organizam as classes.

Além de organizar as classes de projetos, os pacotes são importantes para evitar conflitos de nomes de classes.

Por exemplo, imagine se você pegar uma biblioteca de classes de um colega que não organizou em pacotes. Dentro dessa biblioteca de classes, existe uma com o nome `Deducao`. Se você precisar criar uma classe sua, também com o nome `Deducao`, será impossível se não colocá-la dentro de um pacote, pois os nomes entrariam em conflito.

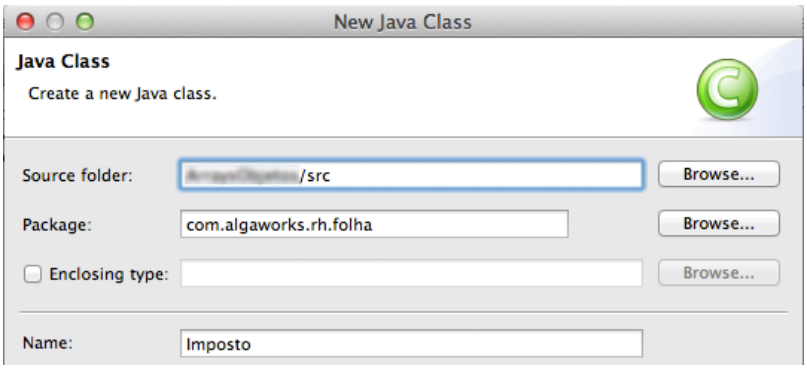
Por isso, é uma boa prática colocar suas classes **sempre** dentro de pacotes, e não só isso, mas nomear os pacotes de acordo com um padrão usado no mundo todo, que veremos daqui a pouco.

```
package com.algaworks.rh.folha;
```

```
public class Deducao {

    private String nome;
    private double valor;

    public String getNome() {
        return this.nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public double getValor() {
        return this.valor;
    }

    public void setValor(double valor) {
        this.valor = valor;
    }

}
```

Colocamos a classe `Deducao` dentro do pacote `com.algaworks.rh.folha` usando a palavra-chave `package` no início do arquivo de código-fonte. Isso quer dizer que a classe deve ficar dentro do diretório *com/algaworks/rh/folha*.

Você não precisa criar os diretórios manualmente. No Eclipse, quando você está criando uma classe, pode informar o nome do pacote no campo **Package**.



A separação de nomes de subpacotes é feita por um ponto, mas o Eclipse criará os diretórios automaticamente para você. Na verdade, funciona assim em qualquer IDE, incluindo o NetBeans.

Existe uma convenção que diz que nomes de pacotes devem iniciar com o domínio da empresa na internet ao contrário, e deve ser escrito tudo em letras minúsculas.

No caso do pacote `com.algaworks.com.rh.folha`, veja que *algaworks.com* é o domínio da empresa na internet.

É importante seguir esse padrão, pois assim, torna-se praticamente impossível que duas classes com o mesmo nome entrem em conflito.

Depois do nome do domínio ao contrário, geralmente, usa-se o nome do sistema, módulo e submódulo. Por exemplo:

```
com.algaworks.nomedosistema.nomedomodulo.nomedosubmodulo
com.algaworks.rh.folha
com.algaworks.rh.demisao
com.algaworks.rh.admissao
com.algaworks.financeiro.contaspagar
com.algaworks.financeiro.fluxocaixa
com.algaworks.seguranca.web
```

Se sua empresa ou você não tiver um domínio registrado na internet, pense em usar o domínio do site onde você hospeda o código-fonte de seu projeto. Por exemplo, se seu projeto estiver no GitHub, use `com.github.nomeusuario.nomeprojeto`.

Você irá perceber que as classes de APIs Java não usam um nome de domínio ao contrário em nomes de pacotes. Não há nada de errado nisso, pois o padrão foi criado apenas para terceiros usarem.

## Usando classes de pacotes

Uma classe pode usar todas as outras classes do mesmo pacote sem precisar especificar o nome do pacote.

```
package com.algaworks.rh.folha;

public class Holerite {

    private String nomeFuncionario;
    private double salarioBruto;
    private Deducao[] deducoes;

    public String getNomeFuncionario() {
        return nomeFuncionario;
    }

    public void setNomeFuncionario(String nomeFuncionario) {
        this.nomeFuncionario = nomeFuncionario;
    }

    public double getSalarioBruto() {
        return salarioBruto;
    }

    public void setSalarioBruto(double salarioBruto) {
        this.salarioBruto = salarioBruto;
    }

    public Deducao[] getDeducoes() {
        return deducoes;
    }

    public void setDeducoes(Deducao[] deducoes) {
        this.deducoes = deducoes;
    }

    public double getSalarioLiquido() {
        double salarioLiquido = getSalarioBruto();

        for (Deducao deducao : getDeducoes()) {
            salarioLiquido -= deducao.getValor();
        }

        return salarioLiquido;
    }

}
```

A classe `Holerite` referencia a classe `Deducao`, mas não foi necessário informar ao compilador onde ele pode encontrar a classe `Deducao`, pois ela está no mesmo pacote.

Agora, vamos criar uma classe em outro pacote e usar `Deducao` e `Holerite`.

```
package com.algaworks.rh;

public class TesteFolha {

    public static void main(String[] args) {
        com.algaworks.rh.folha.Deducao inss =
            new com.algaworks.rh.folha.Deducao();
        inss.setNome("INSS");
        inss.setValor(64);

        com.algaworks.rh.folha.Deducao valeTransporte =
            new com.algaworks.rh.folha.Deducao();
        valeTransporte.setNome("Vale transporte");
        valeTransporte.setValor(48);

        com.algaworks.rh.folha.Holerite holerite =
            new com.algaworks.rh.folha.Holerite();
        holerite.setNomeFuncionario("João das Couves");
        holerite.setSalarioBruto(800);
        holerite.setDeducoes(new com.algaworks.rh.folha.Deducao[]
            { inss, valeTransporte });

        System.out.println("Salário líquido: "
            + holerite.getSalarioLiquido());
    }

}
```

Note que, para usar as classes de outro pacote, tivemos que informar o *nome do pacote + nome da classe* em todas as referências. Isso é chamado de *Fully Qualified Name*, ou *Nome totalmente qualificado*.

Quando uma classe está dentro de um pacote, o nome completo dela passa a ser o nome do pacote mais o nome da classe.

Usar nomes totalmente qualificados nas referências de classes pode deixar o código ilegível. Podemos importar as classes do outro pacote, para usar apenas os nomes simples nas referências.

```
package com.algaworks.rh;

import com.algaworks.rh.folha.Deducao;
import com.algaworks.rh.folha.Holerite;

public class TesteFolha {

    public static void main(String[] args) {
        Deducao inss = new Deducao();
        inss.setNome("INSS");
        inss.setValor(64);

        Deducao valeTransporte = new Deducao();
        valeTransporte.setNome("Vale transporte");
        valeTransporte.setValor(48);

        Holerite holerite = new Holerite();
        holerite.setNomeFuncionario("João das Couves");
        holerite.setSalarioBruto(800);
        holerite.setDeducoes(new Deducao[] { inss, valeTransporte });

        System.out.println("Salário líquido: "
            + holerite.getSalarioLiquido());
    }

}
```

Usando a palavra-chave `import`, importamos as classes `Deducao` e `Holerite` para serem usadas por `TesteFolha`.

As declarações de importação devem vir depois da declaração do nome do pacote no arquivo (se houver).

Podemos ainda, importar todas as classes de um pacote.

```
package com.algaworks.rh;

import com.algaworks.rh.folha.*;
```

```
public class TesteFolha {

    ...

}
```

Quando usamos o coringa *, importamos tudo do pacote. Você não precisa se preocupar com a performance de seu programa por estar importando de um pacote classes que nem serão usadas. Isso não degrada a performance, pois o compilador é inteligente e não carrega classes que você não precisa.

Todavia, é uma boa prática importar as classes uma por uma, pois deixando os nomes explícitos, a legibilidade é melhorada e conflitos de nomes são evitados.

O `java.lang` é um pacote fundamental da linguagem Java, por isso, as classes dele não precisam ser importadas explicitamente. Por exemplo, a classe `java.lang.String` pode ser usada sem fazer um `import java.lang.String` antes.

## 3.2. Modificador de acesso default

Quando não especificamos nenhum modificador de acesso em nossas classes, atributos, construtores ou métodos, esses elementos recebem o modificador de acesso *default* (padrão).

```
package com.algaworks.estoque;

class Produto {
}
```

A classe `Produto` possui o modificador padrão, por isso, fica visível apenas para classes do mesmo pacote.

```
package com.algaworks.comercial;

import com.algaworks.estoque.Produto;

class FechamentoPedido {

    public static void main(String[] args) {
```

```
        // não compila, pois não tem acesso
        Produto produto = new Produto();
    }

}
```

Para conseguirmos compilar a classe `FechamentoPedido`, precisamos incluir o modificador de acesso `public` na classe `Produto` ou mover a classe `FechamentoPedido` para o pacote `com.algaworks.estoque`.

```
package com.algaworks.estoque;

public class Produto {

    private String nome;
    private int estoque;

    String getNome() {
        return this.nome;
    }

    void setNome(String nome) {
        this.nome = nome;
    }

    int getEstoque() {
        return this.estoque;
    }

    void adicionarEstoque(int estoque) {
        this.estoque += estoque;
    }

    void retirarEstoque(int estoque) {
        this.estoque -= estoque;
    }

}
```

Agora a classe `Produto` é pública, mas todos os métodos não possuem modificadores de acesso (ou seja, são *default*).

```
package com.algaworks.comercial;

import com.algaworks.estoque.Produto;

class FechamentoPedido {

    public static void main(String[] args) {
        Produto produto = new Produto();

        // não compila, pois não tem acesso
        produto.setNome("Mouse sem fio");
        produto.adicionarEstoque(10);

        System.out.println(produto.getNome() + " - "
            + produto.getEstoque());
    }

}
```

Continuamos sem conseguir compilar a classe `FechamentoPedido`, pois agora todos os métodos são visíveis apenas para o pacote `com.algaworks.estoque`. Podemos torná-los públicos ou mover a classe `FechamentoPedido` para o mesmo pacote.

```
package com.algaworks.estoque;

class FechamentoPedido {

    public static void main(String[] args) {
        Produto produto = new Produto();

        // Agora sim compila, está no mesmo pacote
        produto.setNome("Mouse sem fio");
        produto.adicionarEstoque(10);

        System.out.println(produto.getNome() + " - "
            + produto.getEstoque());
    }

}
```

Agora que movemos a classe `FechamentoPedido` para o pacote `com.algaworks.estoque`, conseguimos compilar normalmente.

A melhor solução, muitas vezes, não é mover as classes para o mesmo pacote, mas tornar público os membros que necessitam de acesso por classes de outros pacotes.

## 3.3. Membros de classe

Quando diversos objetos são instanciados a partir de uma mesma classe, cada objeto possui suas variáveis de instâncias com valores distintos.

```
public class Usuario {

    private String nome;

    public Usuario(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

}

public class TesteUsuario {

    public static void main(String[] args) {
        Usuario usuario1 = new Usuario("joao");
        Usuario usuario2 = new Usuario("sebastiao");

        System.out.println(usuario1.getNome());
        System.out.println(usuario2.getNome());
    }

}
```

Em determinadas situações, precisamos de variáveis globais, comuns para todos os objetos instanciados. Por exemplo, gostaríamos de contar o número de usuários *logados*.

```
public class Usuario {
```

```
    private int totalUsuariosLogados;
    private String nome;

    public Usuario(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }

    public void logar() {
        this.totalUsuariosLogados++;
    }

    public void deslogar() {
        this.totalUsuariosLogados--;
    }

    public int getTotalUsuariosLogados() {
        return totalUsuariosLogados;
    }

}
```

A variável `totalUsuariosLogados` ainda não é comum a todos os objetos, ela é uma variável de instância!

```
Usuario usuario1 = new Usuario("joao");
Usuario usuario2 = new Usuario("sebastiao");

usuario1.logar();
usuario2.logar();

System.out.println(usuario1.getTotalUsuariosLogados());
System.out.println(usuario2.getTotalUsuariosLogados());
```

Quando executamos o código acima, a saída é:

```
1
1
```

Isso quer dizer que cada instância de `Usuario` armazenou um total de usuários logados.

Para tornar uma variável comum a todos os objetos, precisamos transformá-la em uma variável de classe, adicionando a palavra-chave `static` na declaração.

```
public class Usuario {

    private static int totalUsuariosLogados;

    ...

}
```

Se executarmos novamente o teste anterior, a saída será:

```
2
2
```

Agora, a variável `totalUsuariosLogados` é da classe, e não mais da instância, por isso, o valor dela é compartilhado com todos os objetos.

Se a variável `totalUsuariosLogados` fosse pública, poderíamos acessá-la diretamente pela classe.

```
public class Usuario {

    public static int totalUsuariosLogados;

    ...

}
```

Veja o exemplo:

```
Usuario usuario1 = new Usuario("joao");
Usuario usuario2 = new Usuario("sebastiao");

usuario1.logar();
usuario2.logar();

System.out.println(Usuario.totalUsuariosLogados);
```

Note que acessamos a variável usando o nome da classe mais o nome da variável, sem usar uma instância de `Usuario`.

Deixar a variável `totalUsuariosLogados` pública pode não ser muito legal, pois quem controla o incremento e decremento dela são os métodos da classe `Usuario`. Vamos colocar novamente o modificador `private` nela e tornar o método `getTotalUsuariosLogados` estático.

```
public class Usuario {

    private static int totalUsuariosLogados;

    ...

    public static int getTotalUsuariosLogados() {
        return totalUsuariosLogados;
    }

}
```

Agora, para acessarmos o total de usuários logados, basta chamarmos o método estático (da classe).

```
Usuario usuario1 = new Usuario("joao");
Usuario usuario2 = new Usuario("sebastiao");

usuario1.logar();
usuario2.logar();

System.out.println(Usuario.getTotalUsuariosLogados());
```

Capítulo 4

# Orientação a objetos avançada

## 4.1. Herança e sobrescrita

Em um sistema de banco, temos uma classe que representa uma transferência entre contas.

```
public class Transferencia {

    private String descricao;
    private double valor;
    private String data;
    private String contaOrigem;
    private String contaDestino;

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("========================");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println();
    }

    // getters e setters
```

```
}
```

Declaramos o atributo `data` com o tipo `String` apenas porque não é foco desse livro estudar sobre datas.

Agora, precisamos também de algo que represente uma transação de pagamento de boleto. Podemos tentar adicionar alguns atributos na classe `Transferencia` e até mesmo alterar o nome dela para `Transacao`, para ficar mais genérico.

```
public class Transacao {

    // atributos comuns
    private String descricao;
    private double valor;
    private String data;

    // atributos de uma transferência
    private String contaOrigem;
    private String contaDestino;

    // atributos de um pagamento de boleto
    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("========================");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());

        if (this.getContaOrigem() != null) {
            System.out.println("Conta de origem: "
                + this.getContaOrigem());
            System.out.println("Conta de destino: "
                + this.getContaDestino());
        } else {
            System.out.println("Linha digitável: "
                + this.getLinhaDigitavel());
            System.out.println("Data de vencimento: "
                + this.getDataVencimento());
```

```
            System.out.println("Cedente: " + this.getCedente());
        }

        System.out.println();
    }

    // getters e setters

}
```

A classe `Transacao` que acabamos de criar ficou muito feia. Fizemos uma gambiarra! Uma classe que representa duas coisas ao mesmo tempo só poderia resultar nisso.

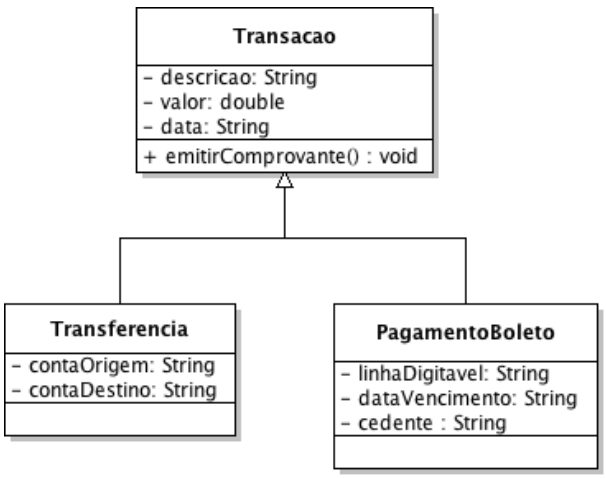
Vamos esquecer essa tentativa e tentar separar em duas classes, `Transferencia` (a primeira versão que criamos) e `PagamentoBoleto`.

```
public class PagamentoBoleto {

    private String descricao;
    private double valor;
    private String data;
    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("========================");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
        System.out.println("Linha digitável: "
            + this.getLinhaDigitavel());
        System.out.println("Data de vencimento: "
            + this.getDataVencimento());
        System.out.println("Cedente: " + this.getCedente());
        System.out.println();
    }

    // getters e setters

}
```

Conseguimos melhorar um pouco a estrutura de nossas classes, eliminando a bagunça que teríamos que tratar com tudo junto em uma única classe, porém, estamos repetindo bastante código.

Podemos deixar nossas classes muito mais reutilizáveis com orientação a objetos. Para isso, usaremos o conceito de herança.

Herança adiciona a capacidade de uma classe filha estender/herdar atributos e métodos de uma classe mãe.




A classe `Transacao` possuirá apenas atributos e métodos que representam uma transação genérica.

```
public class Transacao {

    private String descricao;
    private double valor;
    private String data;

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("========================");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
    }
```

```
    // getters e setters

}
```

As classes `Transferencia` e `PagamentoBoleto` herdam a classe `Transacao`. A herança é declarada através da palavra-chave `extends`.

```
public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;

    // getters e setters

}

public class PagamentoBoleto extends Transacao {

    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    // getters e setters

}
```

Agora, instâncias de objetos do tipo `Transferencia` e `PagamentoBoleto` possuem os membros de `Transacao` também.

```
public class TesteTransacao {

    public static void main(String[] args) {
        PagamentoBoleto pagtoBoleto = new PagamentoBoleto();
        pagtoBoleto.setDescricao("Condomínio");
        pagtoBoleto.setValor(450);
        pagtoBoleto.setData("26/08/2013");
        pagtoBoleto.setLinhaDigitavel("1234 1234 1234");
        pagtoBoleto.setDataVencimento("27/08/2013");
        pagtoBoleto.setCedente("Edifício da Praça Redonda");
        pagtoBoleto.emitirComprovante();

        Transferencia transferencia = new Transferencia();
        transferencia.setDescricao("Aluguel");
        transferencia.setValor(1500);
```

```
        transferencia.setData("10/08/2013");
        transferencia.setContaOrigem("0001000123");
        transferencia.setContaDestino("0001000965");
        transferencia.emitirComprovante();
    }

}
```

Podemos dizer que `PagamentoBoleto` **é uma** `Transacao`, assim como `Transferencia` também é.

Também é comum dizer que `Transacao` é uma **superclasse** e `Transferencia` e `PagamentoBoleto` são **subclasses** de `Transacao`.

Java não permite herança múltipla, ou seja, uma classe pode herdar de apenas uma outra classe.

O problema agora é que o método `emitirComprovante` não tem acesso aos atributos e métodos das classes filhas, para imprimir na saída algumas informações específicas dessas subclasses. Por exemplo, a execução da classe acima exibe na saída comprovantes genéricos.

```
Comprovante da transação
========================
Descrição: Condomínio
Data: 26/08/2013
Valor: 450.0
Comprovante da transação
========================
Descrição: Aluguel
Data: 10/08/2013
Valor: 1500.0
```

Podemos usar o conceito de **sobrescrita** e declarar o método `emitirComprovante` nas classes filhas, substituindo totalmente o método da classe mãe. Por enquanto, vamos fazer isso apenas na classe `Transferencia`.

```
public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;
```

```
    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("========================");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());

        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println();
    }

    // getters e setters

}
```

Agora a execução de nossa classe de teste exibe na saída o comprovante de transferência que implementamos no método sobrescrito na classe filha.

```
Comprovante da transação
========================
Descrição: Condomínio
Data: 26/08/2013
Valor: 450.0

Comprovante da transação
========================
Descrição: Aluguel
Data: 10/08/2013
Valor: 1500.0
Conta de origem: 0001000123
Conta de destino: 0001000965
```

Em nosso exemplo, os cabeçalhos dos comprovantes são idênticos para todos os tipos de transações, por isso, seria desnecessário escrever esse código novamente em todas as classes filhas.

Mesmo substituindo um método da classe mãe, podemos chamá-lo se acharmos necessário, usando a palavra-chave `super`.

```
public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;

    public void emitirComprovante() {
        super.emitirComprovante();

        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println();
    }

    // getters e setters

}
```

Pronto! Agora vamos fazer a mesma coisa com a classe `PagamentoBoleto`.

```
public class PagamentoBoleto extends Transacao {

    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    public void emitirComprovante() {
        super.emitirComprovante();

        System.out.println("Linha digitável: "
            + this.getLinhaDigitavel());
        System.out.println("Data de vencimento: "
            + this.getDataVencimento());
        System.out.println("Cedente: " + this.getCedente());
        System.out.println();
    }

    // getters e setters

}
```

A chamada `super.emitirComprovante()` irá invocar o método da superclasse e depois imprimir na saída os detalhes específicos da transação, como uma transferência ou pagamento de boleto.

```
Comprovante da transação
========================
Descrição: Condomínio
Data: 26/08/2013
Valor: 450.0
Linha digitável: 1234 1234 1234
Data de vencimento: 27/08/2013
Cedente: Edifício da Praça Redonda

Comprovante da transação
========================
Descrição: Aluguel
Data: 10/08/2013
Valor: 1500.0
Conta de origem: 0001000123
Conta de destino: 0001000965
```

Como você pôde ver, herança é um recurso poderoso da orientação a objetos, mas você deve usar com muito cuidado.

É comum ver programadores iniciantes em orientação a objetos usar herança apenas para reaproveitar código, e isso é muito errado.

Para evitar este erro, sempre que pensar em usar herança, faça uma pequena análise.

Suponhamos que você queira que uma classe `Radio` herde `Televisor` para reaproveitar membros comuns. Pergunte-se a si mesmo: *"rádio é um televisor?"*.

Claro que a resposta é **não**, rádio não é um tipo específico de televisor, por isso, `Radio` não deve estender `Televisor`.

## java.lang.Object

Todas as classes em Java, inclusive as que você mesmo criou, herdam de `java.lang.Object`.

```
// herda Object implicitamente
public class MinhaClasse {

}

// herda Object explicitamente
public class MinhaClasse extends Object {
}
```

Se uma classe herdar outra, apenas a classe pai que pode herdar `Object` diretamente.

```
// herda Object implicitamente
public class MinhaClasse {
}

public class OutraClasse extends MinhaClasse {
}
```

A classe `OutraClasse` não herda `Object` diretamente, mas ainda podemos dizer que `OutraClasse` é um `Object`, pois toda a hierarquia de classes é herdada por `OutraClasse`.

## 4.2. Modificador de acesso protected

O modificador de acesso `protected` diz que o membro só pode ser acessado pelo próprio pacote (igual o nível *default*) e por subclasses, mesmo estando em outros pacotes.

Este modificador pode ser usado em atributos, métodos e construtores de uma classe.

```
package com.algaworks.comum;

public class Equipamento {

    protected boolean ligado;

    public boolean isLigado() {
        return this.ligado;
```

```
    }

}
```

Se uma classe de outro pacote tentar atribuir um valor para o atributo `ligado`, o código não compilará.

```
package com.algaworks.simulacao;

import com.algaworks.comum.Equipamento;

public class TesteProtected {

    public static void main(String[] args) {
        Equipamento equipamento = new Equipamento();

        // não compila
        equipamento.ligado = true;
    }

}
```

O código acima não compila! A classe `TesteProtected` está em um pacote diferente da classe `Equipamento`, e o atributo `ligado` não está visível.

Criaremos agora uma classe que estende `Equipamento`, em um pacote diferente, mas usa o atributo `ligado`.

```
package com.algaworks.audio;

import com.algaworks.comum.Equipamento;

public class Radio extends Equipamento {

    public void ligar() {
        // compila!
        this.ligado = true;
    }

}
```

A classe `Radio` compila normalmente, pois um atributo protegido é visível para subclasses.

```
package com.algaworks.simulacao;

import com.algaworks.audio.Radio;

public class TesteProtected {

    public static void main(String[] args) {
        Radio radio = new Radio();

        radio.ligar();
        System.out.println(radio.isLigado());
    }

}
```

Deixar atributos protegidos para serem acessados diretamente por outras classes pode influenciar você e outros programadores a alterarem valores de atributos sem passar por um método, "roubando" a responsabilidade da classe de gerenciar o seu estado. Isso não é uma boa prática, por isso, tome bastante cuidado.

## 4.3. Polimorfismo

Polimorfismo é a capacidade de um objeto tomar várias formas, que decorre diretamente do mecanismo de herança. Ao estendermos ou especializarmos uma classe, não perdemos compatibilidade com a superclasse.

Voltando ao código do tópico sobre herança, vamos instanciar um objeto do tipo `Transferencia`, mas atribuir a referência a uma variável do tipo `Transacao`.

```
Transacao transferencia = new Transferencia();
```

Quando fazemos isso, o objeto que instanciamos não é convertido para um novo tipo. Ele continua sendo uma transferência, mas **enxergamos** ele apenas como uma transação, por isso, não conseguimos invocar os métodos específicos de uma transferência.

```
Transacao transferencia = new Transferencia();
transferencia.setDescricao("Aluguel");
```

```
transferencia.setValor(1500);
transferencia.setData("10/08/2013");

// não compila
transferencia.setContaOrigem("0001000123");
transferencia.setContaDestino("0001000965");

transferencia.emitirComprovante();
```

Talvez você ache que não faz sentido um objeto de um tipo ser enxergado pelo supertipo, pois restringe os métodos que podem ser chamados. Esse é um pensamento comum em pessoas que estão iniciando com orientação a objetos, mas não se engane, o poder do polimorfismo é fantástico!

Vamos criar um construtor na classe `Transferencia`, para ficar mais fácil nosso exemplo.

```
public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;

    public Transferencia(String descricao, double valor, String data,
            String contaOrigem, String contaDestino) {
        super(descricao, valor, data);

        this.contaOrigem = contaOrigem;
        this.contaDestino = contaDestino;
    }

    public void emitirComprovante() {
        super.emitirComprovante();

        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println();
    }

    // getters e setters

}
```

Veja uma novidade aí! Chamamos o construtor da classe mãe pela instrução `super(descricao, valor, data)`. Esse construtor ainda não existe, portanto, precisamos criá-lo também.

```
public class Transacao {

    private String descricao;
    private double valor;
    private String data;

    public Transacao(String descricao, double valor, String data) {
        this.descricao = descricao;
        this.valor = valor;
        this.data = data;
    }

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("========================");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
    }

    // getters e setters

}
```

Voltando ao polimorfismo, vamos instanciar novamente uma `Transferencia`, passando os dados pelo construtor criado, e invocar o método `emitirComprovante`.

```
Transacao transferencia = new Transferencia("Aluguel", 1500,
    "10/08/2013", "0001000123", "0001000965");
transferencia.emitirComprovante();
```

O código acima exibe na saída:

```
Comprovante da transação
========================
Descrição: Aluguel
Data: 10/08/2013
Valor: 1500.0
```

```
Conta de origem: 0001000123
Conta de destino: 0001000965
```

Veja que o método chamado foi da classe `Transferencia`.

Precisamos criar um construtor na classe `PagamentoBoleto` para continuar nossos exemplos.

```
public class PagamentoBoleto extends Transacao {

    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    public PagamentoBoleto(String descricao, double valor,
            String data, String linhaDigitavel, String dataVencimento,
            String cedente) {
        super(descricao, valor, data);

        this.linhaDigitavel = linhaDigitavel;
        this.dataVencimento = dataVencimento;
        this.cedente = cedente;
    }

    public void emitirComprovante() {
        super.emitirComprovante();

        System.out.println("Linha digitável: "
            + this.getLinhaDigitavel());
        System.out.println("Data de vencimento: "
            + this.getDataVencimento());
        System.out.println("Cedente: " + this.getCedente());
        System.out.println();
    }

    // getters e setters

}
```

Agora, vamos criar uma classe emissora de comprovantes, que desconhece qualquer tipo específico de transação.

```
public class EmissorDeComprovantes {

    public void emitirComprovantes(Transacao... transacoes) {
        for (Transacao transacao : transacoes) {
            transacao.emitirComprovante();
        }
    }

}
```

Repare que o programador da classe `EmissorDeComprovantes` não precisa conhecer os tipos de transações que existem. Ele só precisa conhecer o que uma transação é capaz de fazer. Dessa forma, deixamos o código mais desacoplado e facilitamos a manutenção do software.

Instanciamos nosso emissor de comprovantes e invocamos o método `emitirComprovantes`.

```
Transferencia transferencia = new Transferencia("Aluguel", 1500,
    "10/08/2013", "0001000123", "0001000965");

PagamentoBoleto pagtoBoleto = new PagamentoBoleto("Condomínio", 450,
    "26/08/2013", "1234 1234 1234", "27/08/2013",
    "Edifício da Praça Redonda");

EmissorDeComprovantes emissor = new EmissorDeComprovantes();
emissor.emitirComprovantes(transferencia, pagtoBoleto);
```

A JVM chama o método apropriado para cada objeto, de acordo com o tipo real dele, e não do tipo da variável.

A execução do último código exibe na saída:

```
Comprovante da transação
========================
Descrição: Aluguel
Data: 10/08/2013
Valor: 1500.0
Conta de origem: 0001000123
Conta de destino: 0001000965

Comprovante da transação
```

```
========================
Descrição: Condomínio
Data: 26/08/2013
Valor: 450.0
Linha digitável: 1234 1234 1234
Data de vencimento: 27/08/2013
Cedente: Edifício da Praça Redonda
```

## 4.4. Casting de objetos e instanceof

Um objeto tem o tipo da classe usada para instanciá-lo.

```
Transacao transacao = new PagamentoBoleto("Condomínio", 450,
    "26/08/2013", "1234 1234 1234", "27/08/2013",
    "Edifício da Praça Redonda");

Object obj = pagtoBoleto;
```

No código acima, o objeto referenciado pelas variáveis `transacao` e `obj` é do tipo `PagamentoBoleto`.

A classe `PagamentoBoleto` é filha de `Transacao`, que é filha de `Object`. Podemos dizer que `PagamentoBoleto` é uma `Transacao` e também um `Object`.

Dessa forma, sempre que você precisar de uma `Transacao` ou `Object`, poderá passar um objeto do tipo `PagamentoBoleto`.

Pense no mundo real: se o gerente de sua conta pedir para você falar qual transação você quer gerar uma segunda via do comprovante, fica subentendido que você pode dizer os dados de um pagamento de boleto ou de uma transferência, além das diversas outras transações.

Se um pedinte na rua pedir para você qualquer coisa, por exemplo, "pode me dar qualquer coisa?", você poderia simplesmente entregar um pagamento de boleto para ele, uma caneta, seu carro, uma nota fiscal, uma moeda ou qualquer outra coisa, pois o que foi solicitado é um tipo muito genérico, um `Object` do mundo real.

Até agora, estudamos o caminho mais simples, onde temos um tipo específico sendo atribuído a um tipo mais genérico.

O reverso exige um pouco de cuidado! Uma `Transacao` pode ser um `PagamentoBoleto`, mas também pode não ser.

```
Transacao transacao = new PagamentoBoleto("Condomínio", 450,
    "26/08/2013", "1234 1234 1234", "27/08/2013",
    "Edifício da Praça Redonda");

// não compila
PagamentoBoleto pagtoBoleto = transacao;
```

Sabemos que no código acima, instanciamos um `PagamentoBoleto`, mas o compilador não sabe. Por isso, não conseguimos compilar.

Precisamos dizer ao compilador que queremos fazer um *casting* do objeto! Quando fazemos isso, nada é convertido e nada no objeto é alterado, apenas assumimos todos os riscos de tentar enxergar um objeto mais genérico como um tipo mais específico.

```
Transacao transacao = new PagamentoBoleto("Condomínio", 450,
    "26/08/2013", "1234 1234 1234", "27/08/2013",
    "Edifício da Praça Redonda");

// compila!
PagamentoBoleto pagtoBoleto = (PagamentoBoleto) transacao;
System.out.println(pagtoBoleto.getLinhaDigitavel());
```

Conseguimos fazer o *casting* escrevendo `(PagamentoBoleto)` na frente do nome da variável.

Agora, vamos instanciar uma `Transferencia`, declarando a variável com o tipo `Transacao`, e depois tentar atribuir a uma variável do tipo `PagamentoBoleto`.

```
Transacao transacao = new Transferencia("Aluguel", 1500,
    "10/08/2013", "0001000123", "0001000965");

// não compila
PagamentoBoleto pagtoBoleto = transacao;
```

Precisamos deixar explícito o *casting* que queremos fazer, ou o código não irá compilar.

```
Transacao transacao = new Transferencia("Aluguel", 1500, "10/08/2013",
    "0001000123", "0001000965");

// compila, mas experimente rodar (hehehe)
PagamentoBoleto pagtoBoleto = (PagamentoBoleto) transacao;
```

Fizemos o *casting* do objeto para o tipo `PagamentoBoleto`, e o código compila, pois é possível que o objeto referenciado pela variável `transacao` seja do tipo `PagamentoBoleto`, já que declaramos essa variável com o tipo `Transacao`.

Nós sabemos que, na verdade, é uma transferência, mas o compilador não sabe disso, e acredite, muitas vezes nem mesmo você saberá, por exemplo, quando você recebe objetos como parâmetro de um método.

Apesar do código compilar, quando executamos, recebemos uma mensagem de erro em tempo de execução.

```
Exception in thread "main" java.lang.ClassCastException:
         Transferencia cannot be cast to PagamentoBoleto
    at TesteCasting.main(TesteCasting.java:8)
```

## Operador de comparação instanceof

Você pode fazer uma comparação lógica para verificar se um objeto é de um tipo específico, usando o operador `instanceof`.

```
Object obj = new Transferencia("Aluguel", 1500, "10/08/2013",
    "0001000123", "0001000965");

if (obj instanceof PagamentoBoleto) {
    PagamentoBoleto pagtoBoleto = (PagamentoBoleto) obj;
    System.out.println(pagtoBoleto.getLinhaDigitavel());
} else if (obj instanceof Transacao) {
    Transacao transacao = (Transacao) obj;
    System.out.println(transacao.getDescricao());
}
```

Com essa verificação, você fica livre de erros em tempo de execução pelo motivo de *castings* inválidos.

## 4.5. Classes e métodos abstratos

No exemplo de transações de uma conta em um banco, que fizemos nas últimas seções, criamos a classe `Transacao` apenas para reaproveitar código e tomar vantagem do polimorfismo. Não faz sentido algum instanciar uma `Transacao`.

```
Transacao transacao = Transacao("Convênio médico", 400, "05/08/2013");
```

A transação que instanciamos é muito genérica! Não é um pagamento de boleto, uma transferência ou qualquer outra coisa. É apenas uma transação.

Se analisarmos o mundo real, realmente, não existe o conceito concreto do que é uma transação em um banco. Sempre existe algo mais específico para definir melhor, neste caso.

Podemos pensar em outro exemplo do mundo real: uma fabricante de carros fabrica apenas veículos ou carros?

Se alguém disser que quer oferecer um veículo para você, com certeza você deve imaginar que pode ser uma moto, um carro, um avião, mas nunca apenas um veículo, sem um tipo específico. Sabe porque? Veículo é muito abstrato! Não existem objetos simplesmente do tipo `Veiculo` no mundo real. Usamos a palavra "veículo" para conceituar um meio de transporte abstrato.

Continuando o raciocínio sobre nosso exemplo, seria interessante dizer que `Transacao` é uma classe abstrata, portanto, não pode ser instanciada. Para isso, usamos a palavra-chave `abstract` na declaração da classe.

```
public abstract class Transacao {

    private String descricao;
    private double valor;
    private String data;

    public Transacao(String descricao, double valor, String data) {
```

```
        this.descricao = descricao;
        this.valor = valor;
        this.data = data;
    }

    public void emitirComprovante() {
        System.out.println("Comprovante da transação");
        System.out.println("========================");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Data: " + this.getData());
        System.out.println("Valor: " + this.getValor());
    }

    // getters e setters

}
```

Agora, não conseguimos instanciar uma `Transacao`.

```
// não compila
Transacao transacao = Transacao("Convênio médico", 400, "05/08/2013");
```

Você quer saber qual é a utilidade da classe `Transacao`, já que não podemos instanciá-la? Polimorfismo, principalmente, e reaproveitamento de código pelas subclasses.

## Métodos abstratos

Suponha que exista a necessidade de emitir comprovantes com conteúdos totalmente diferentes. Por exemplo, um comprovante de transferência pode incluir um cabeçalho com os números das contas de origem e destino, antes mesmo de imprimir o valor, etc.

Neste caso, podemos remover o método `emitirComprovante` da classe `Transacao`, pois não conseguiremos reaproveitá-lo.

```
public abstract class Transacao {

    private String descricao;
    private double valor;
    private String data;
```

```
    public Transacao(String descricao, double valor, String data) {
        this.descricao = descricao;
        this.valor = valor;
        this.data = data;
    }

    // getters e setters

}
```

Agora, alteramos a implementação dos métodos de emissão de comprovante nas classes `Transferencia` e `PagamentoBoleto`.

```
public class PagamentoBoleto extends Transacao {

    private String linhaDigitavel;
    private String dataVencimento;
    private String cedente;

    public PagamentoBoleto(String descricao, double valor,
            String data, String linhaDigitavel,
            String dataVencimento, String cedente) {
        super(descricao, valor, data);

        this.linhaDigitavel = linhaDigitavel;
        this.dataVencimento = dataVencimento;
        this.cedente = cedente;
    }

    public void emitirComprovante() {
        System.out.println("Comprovante da pagamento de boleto");
        System.out.println("==================================");
        System.out.println("Linha digitável: "
            + this.getLinhaDigitavel());
        System.out.println("----------------------------------");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Cedente: " + this.getCedente());
        System.out.println("Valor: " + this.getValor());
        System.out.println("Data de vencimento: "
            + this.getDataVencimento());
        System.out.println("Data do pagamento: " + this.getData());
        System.out.println();
```

```
    }

    // getters e setters

}

public class Transferencia extends Transacao {

    private String contaOrigem;
    private String contaDestino;

    public Transferencia(String descricao, double valor, String data,
            String contaOrigem, String contaDestino) {
        super(descricao, valor, data);

        this.contaOrigem = contaOrigem;
        this.contaDestino = contaDestino;
    }

    public void emitirComprovante() {
        System.out.println("Comprovante de transferência");
        System.out.println("============================");
        System.out.println("Conta de origem: "
            + this.getContaOrigem());
        System.out.println("Conta de destino: "
            + this.getContaDestino());
        System.out.println("----------------------------");
        System.out.println("Descrição: " + this.getDescricao());
        System.out.println("Valor: " + this.getValor());
        System.out.println("Data da operação: " + this.getData());
        System.out.println();
    }

    // getters e setters

}
```

Você se lembra da classe `EmissorDeComprovantes`? Será que temos algum problema nela, depois dessas alterações?

```
public class EmissorDeComprovantes {

    public void emitirComprovantes(Transacao... transacoes) {
        for (Transacao transacao : transacoes) {
```

```
            // não compila
            transacao.emitirComprovante();
        }
    }

}
```

Puxa! A classe `EmissorDeComprovantes` não está mais compilando, pois ela utiliza polimorfismo e tenta chamar o método `emitirComprovante` em um tipo `Transacao`, mas nós removemos o método.

Para resolver isso, poderíamos criar um método `emitirComprovante` na classe `Transacao`, que não implementa nada, mas isso não é uma boa ideia, já que poderíamos esquecer de sobrescrever o método quando criarmos novas subclasses.

Felizmente, podemos resolver isso criando um método abstrato!

Em Java, uma classe abstrata pode ter métodos abstratos. Métodos abstratos devem ser, obrigatoriamente, implementados por uma subclasse.

Para dizer que um método é abstrato, basta declararmos ele usando a palavra-chave `abstract`.

```
public abstract class Transacao {

    private String descricao;
    private double valor;
    private String data;

    public Transacao(String descricao, double valor, String data) {
        this.descricao = descricao;
        this.valor = valor;
        this.data = data;
    }

    public abstract void emitirComprovante();

    // getters e setters

}
```

Repare que não incluímos corpo no método `emitirComprovante`. Métodos abstratos não possuem implementação, por isso, finalizamos a declaração dele com um ponto e vírgula.

Agora a classe `EmissorDeComprovantes` continua compilando, pois o compilador tem certeza que o método `emitirComprovante` será implementado por classes concretas que herdarem `Transacao`.

## 4.6. Interfaces

Existem diversas situações na vida real que precisamos definir padrões de como as coisas devem funcionar, e os fabricantes assinam ou concordam com um contrato que diz como os componentes se interagem. Cada fabricante é capaz de fornecer seus componentes/peças sem conhecer como as outras partes funcionam.

Para exemplificar, vamos pensar em como os carros funcionam, sem entrar em muitos detalhes técnicos, pois nós (autores) não conhecemos a fundo sobre esse assunto, e acredito que você também não (a não ser que seja um mecânico ou apaixonado pela mecânica dos carros).

Para termos um carro funcionando, primeiramente, as peças devem ser fabricadas. Somente depois da fabricação de todas as peças que o carro é montado.

Durante a fabricação das peças, os diversos fabricantes conhecem as especificações das outras partes que irão interagir com as peças de sua responsabilidade, em termos do que elas devem ter, e não como elas funcionam exatamente.

Por exemplo, o fabricante de pneus deve conhecer o aro da roda e também que ela deve girar, mas não faz a mínima ideia de como funciona a combustão do motor ou se o carro tem freios ABS ou não.

O fabricante do painel do carro sabe que em determinado lugar, deve existir o espaço para o painel do computador de bordo, mas não tem mais nenhum detalhe a mais sobre isso.

O computador de bordo pode ser fabricado por dezenas de outros fabricantes, inclusive os paralelos, e funcionar normalmente no carro, desde que siga o padrão.

Essas especificações ou padrões, nós chamamos em Java de interfaces. Uma interface especifica o que alguma coisa deve ter, mas não define como deve fazer para alcançar o que é necessário.

Pense nos diversos componentes do mundo real que seguem interfaces: TVs e seus componentes, computadores e as placas e acessórios que se encaixam perfeitamente, independente do fabricante, funcionários em sua empresa, que possuem cargos bem definidos e interagem com outros departamentos e pessoas, etc. O mundo todo é baseado em interfaces!

Interfaces é o recurso da orientação a objetos que potencializa o polimorfismo em nível máximo. É fantástico o que podemos fazer com a combinação desses conceitos!

Agora vamos implementar algo em Java usando o conceito de interfaces. Precisamos programar um checkout (fechamento de compra), que aceita pagamentos por cartão de crédito de uma operadora qualquer e imprime os dados da compra em uma impressora qualquer.

Veja bem a palavra "qualquer". Neste momento, não importa muito qual é a operadora de cartão e qual é a impressora.

Vamos começar definindo o contrato de uma impressora.

```
package com.algaworks.impressao;

public interface Impressora {

    void imprimir(Imprimivel imprimivel);

}
```

Declaramos uma interface usando a palavra-chave `interface`.

Uma interface possui métodos públicos abstratos (sem implementação). Quando não declaramos que o método é público e abstrato, ele recebe implicitamente `public abstract`.

Na interface `Impressora`, dizemos que qualquer classe que tenha interesse em se tornar uma impressora, deve implementar o método `imprimir`.

Note que não especificamos como deve ser feita essa impressão. Pode ser jato de tinta, laser, matricial, etc.

O método `imprimir` recebe um objeto do tipo `Imprimivel`.

```
package com.algaworks.impressao;

public interface Imprimivel {

    String getCabecalhoPagina();
    String getCorpoPagina();

}
```

Um imprimível é algo capaz de ser impresso, e para isso, definimos como requisitos mínimos que ele deve ter os dois métodos declarados, um que retorna o cabeçalho da página e outro que retorna o corpo da página.

Até agora, não temos nenhuma impressora e nenhum imprimível. Temos apenas contratos!

Deixaremos para implementar classes concretas mais pra frente, para estimular a programação baseada em interfaces, sem conhecer o que está por trás da implementação, que é uma grande dificuldade para quem está iniciando em programação orientada a objetos.

Agora, criaremos a interface `Operadora`, que define o que uma operadora de cartão de crédito deve ter.

```
package com.algaworks.pagamento;

public interface Operadora {
```

```
    boolean autorizar(Autorizavel autorizavel, Cartao cartao);

}
```

Veja que uma operadora deve ter o método `autorizar`, que recebe um objeto do tipo `Autorizavel` e um `Cartao`.

```
package com.algaworks.pagamento;

public interface Autorizavel {

    double getValorTotal();

}
```

```
package com.algaworks.pagamento;

public class Cartao {

    private String nomeTitular;
    private String numeroCartao;

    public String getNomeTitular() {
        return nomeTitular;
    }

    public void setNomeTitular(String nomeTitular) {
        this.nomeTitular = nomeTitular;
    }

    public String getNumeroCartao() {
        return numeroCartao;
    }

    public void setNumeroCartao(String numeroCartao) {
        this.numeroCartao = numeroCartao;
    }

}
```

`Autorizavel` é uma interface, que define o que uma classe deve implementar para ser autorizável, ou seja, para ser processada por alguma operadora de cartão de crédito.

A classe `Cartao` possui os dados básicos do cartão de crédito a ser autorizado/cobrado.

Agora chegou a hora de criar a primeira classe que implementa `Autorizavel`.

Programaremos a classe `Compra`, que representa a compra que está sendo feita por um cliente. Uma compra é autorizável, ou seja, pode ser cobrada por cartão de crédito.

```
package com.algaworks.caixa;

import com.algaworks.pagamento.Autorizavel;

public class Compra implements Autorizavel {

    private String nomeCliente;
    private double valorTotal;
    private String produto;

    public String getNomeCliente() {
        return nomeCliente;
    }

    public void setNomeCliente(String nomeCliente) {
        this.nomeCliente = nomeCliente;
    }

    public double getValorTotal() {
        return valorTotal;
    }

    public void setValorTotal(double valorTotal) {
        this.valorTotal = valorTotal;
    }

    public String getProduto() {
        return produto;
    }
```

```
    public void setProduto(String produto) {
        this.produto = produto;
    }

}
```

Uma classe implementa uma interface a partir da palavra-chave `implements`. A classe é obrigada a implementar todos os métodos declarados na interface.

Neste caso, implementamos o método `getValorTotal`, que é o único método exigido pela interface `Autorizavel`.

Classes podem implementar diversas interfaces. Incluiremos a interface `Imprimivel` na declaração da classe `Compra`.

```
package com.algaworks.caixa;

import com.algaworks.impressao.Imprimivel;
import com.algaworks.pagamento.Autorizavel;

public class Compra implements Autorizavel, Imprimivel {

    // atributos e métodos

}
```

Quando implementamos a interface `Imprimivel`, nosso código deixa de compilar, pois somos obrigados a implementar os métodos definidos pela nova interface que adicionamos.

```
package com.algaworks.caixa;

import com.algaworks.impressao.Imprimivel;
import com.algaworks.pagamento.Autorizavel;

public class Compra implements Autorizavel, Imprimivel {

    // atributos e métodos

    public String getCorpoPagina() {
        return this.getProduto() + " = " + this.getValorTotal();
```

```
    }

    public String getCabecalhoPagina() {
        return this.getNomeCliente();
    }

}
```

A próxima classe será de nome `Checkout`. Esta classe receberá uma `Operadora` e uma `Impressora` no construtor, e terá um método `fecharCompra`, que receberá uma `Compra` e um `Cartao` para fechar a compra (autorizar e imprimir o cupom).

```
package com.algaworks.caixa;

import com.algaworks.impressao.Impressora;
import com.algaworks.pagamento.Cartao;
import com.algaworks.pagamento.Operadora;

public class Checkout {

    private Operadora operadora;
    private Impressora impressora;

    public Checkout(Operadora operadora, Impressora impressora) {
        this.operadora = operadora;
        this.impressora = impressora;
    }

    public void fecharCompra(Compra compra, Cartao cartao) {
        boolean autorizado = this.operadora.autorizar(compra, cartao);

        if (autorizado) {
            this.impressora.imprimir(compra);
        } else {
            System.out.println("Pagamento negado!");
        }
    }

}
```

Até aqui, este poderia ser o seu trabalho em um projeto em Java. As implementações de operadoras de cartões e impressoras, poderiam ser terceirizadas ou de responsabilidade de outro programador de sua equipe.

Ainda não conseguimos executar nosso projeto, pois precisamos de pelo menos uma classe que implementa `Impressora` e outra que implementa `Operadora`.

```
public class TesteCheckout {

    public static void main(String[] args) {
        // precisamos de implementações de Operadora e Impressora
        Operadora operadora = ...
        Impressora impressora = ...

        Cartao cartao = new Cartao();
        cartao.setNomeTitular("João M Couves");
        cartao.setNumeroCartao("456");

        Compra compra = new Compra();
        compra.setNomeCliente("João Mendonça Couves");
        compra.setProduto("Sabonete");
        compra.setValorTotal(2.5);

        Checkout checkout = new Checkout(operadora, impressora);
        checkout.fecharCompra(compra, cartao);
    }

}
```

Criaremos uma classe `ImpressoraEpson`, que implementa a interface `Impressora`.

```
package com.algaworks.impressao;

public class ImpressoraEpson implements Impressora {

    public void imprimir(Imprimivel imprimivel) {
        System.out.println("* * * * * * * * * * * * * * * * * *");
        System.out.println(imprimivel.getCabecalhoPagina());
        System.out.println("* * * * * * * * * * * * * * * * * *");
        System.out.println(imprimivel.getCorpoPagina());
        System.out.println("- - - - - - - - - - - - - - - - - -");
        System.out.println("==            EPSON            ==");
        System.out.println("- - - - - - - - - - - - - - - - - -");
    }

}
```

E agora a classe `Cielo`, que implementa a interface `Operadora` e fornece a integração com a processadora de cartões e crédito Cielo.

```
package com.algaworks.pagamento;

public class Cielo implements Operadora {

    public boolean autorizar(Autorizavel autorizavel, Cartao cartao) {
        return cartao.getNumeroCartao().startsWith("123");
    }

}
```

Para efeito de testes, a implementação da Cielo autorizará apenas cartões com números que comecem com "123".

Podemos agora instanciar um `Checkout`, passando como parâmetro do construtor os objetos das classes que implementam `Impressora` e `Operadora`.

```
public class TesteCheckout {

    public static void main(String[] args) {
        Operadora operadora = new Cielo();
        Impressora impressora = new ImpressoraEpson();

        Cartao cartao = new Cartao();
        cartao.setNomeTitular("João M Couves");
        cartao.setNumeroCartao("456");

        Compra compra = new Compra();
        compra.setNomeCliente("João Mendonça Couves");
        compra.setProduto("Sabonete");
        compra.setValorTotal(2.5);

        Checkout checkout = new Checkout(operadora, impressora);
        checkout.fecharCompra(compra, cartao);
    }

}
```

Tente executar a classe `TesteCheckout`, alterando o número do cartão.

Até agora, parece que o uso de interfaces não trouxe nenhum benefício.

Imagine que sua empresa tenha necessidade de instalar uma nova impressora, e também fechou um contrato com a Redecard, para ter uma alternativa no processamento de pagamentos com cartões de crédito.

Você pode ficar responsável pela implementação da nova impressora e passar para outro programador a responsabilidade da integração com a Redecard. Basta usar as interfaces!

```
package com.algaworks.impressao;

public class ImpressoraXingling implements Impressora {

    public void imprimir(Imprimivel imprimivel) {
        System.out.println(imprimivel.getCabecalhoPagina());
        System.out.println("-=-=-=-=-=-=-=-=-=-=-");
        System.out.println(imprimivel.getCorpoPagina());
        System.out.println("====================");
        System.out.println("**Xingling Printer**");
    }

}
```

A classe `ImpressoraXingling` exibe na saída os mesmos dados do cabeçalho e corpo da página, mas em um formato diferente.

```
package com.algaworks.pagamento;

public class Redecard implements Operadora {

    public boolean autorizar(Autorizavel autorizavel, Cartao cartao) {
        return cartao.getNumeroCartao().startsWith("456")
                    && autorizavel.getValorTotal() < 200;
    }

}
```

A classe `Redecard` autoriza apenas cartões que os números iniciem com “456” e o valor total seja menor que R$200.

Para usar as novas classes, basta substituir a instanciação em `TesteCheckout`.

```
public class TesteCheckout {

    public static void main(String[] args) {
        Operadora operadora = new Redecard();
        Impressora impressora = new ImpressoraXingling();

        Cartao cartao = new Cartao();
        cartao.setNomeTitular("João M Couves");
        cartao.setNumeroCartao("456");

        Compra compra = new Compra();
        compra.setNomeCliente("João Mendonça Couves");
        compra.setProduto("Sabonete");
        compra.setValorTotal(2.5);

        Checkout checkout = new Checkout(operadora, impressora);
        checkout.fecharCompra(compra, cartao);
    }

}
```

Faça diversos testes, alternando as implementações de impressoras e operadoras e mudando o valor total da compra e número do cartão.

O uso de interfaces deixou nosso código muito mais extensível e desacoplado.

A classe `Checkout` não conhece as implementações de impressoras e operadoras, mas apenas os contratos (as interfaces).

Da mesma forma, as implementações de impressoras não conhecem o que está sendo impresso, e as classes que implementam operadoras não conhecem o que está sendo autorizado. Tudo é baseado em interfaces.

Capítulo 5

# Conclusão

Se você chegou até aqui sem pular as páginas, parabéns! Você está entre os poucos que conseguem se dedicar em um assunto do início ao fim!

Caso você ainda esteja pensando em ler o livro, nossa sugestão é que você comece agora e deixe de procrastinar! :)

## 5.1. Próximos passos

Embora nós tenhamos nos dedicado bastante para escrever esse livro, o conteúdo que você aprendeu nele é só a ponta do iceberg!

É claro que você não perdeu tempo com o que acabou de estudar, o que queremos dizer é que há muito mais coisas para aprofundar.

Caso você tenha interesse em mergulhar fundo em conteúdos ainda mais avançados e aprender com vídeos passo a passo, recomendo que você dê uma olhada no nosso curso online:

Java e Orientação a Objetos
http://alga.works/ebook-oo-com-java-cta

Se você acha que já está preparado para aprender os famosos frameworks para desenvolvimento Java, veja nossos outros cursos aqui:

http://www.algaworks.com/

Além de tudo que você vai aprender, nós ainda oferecemos suporte para as suas dúvidas!

THIAGO FARIA E NORMANDES JR

algaworks