Capítulo 7

Interface

Além dos mais diversos tipos de `Livros`, nossa livraria também trabalhará com `Revistas` e futuramente outros produtos. Podemos criar uma nova classe para representá-la, como a seguir:

```
public class Revista {

    private String nome;
    private String descricao;
    private double valor;
    private Editora editora;

    // getters e setters

    public boolean aplicaDescontoDe(double porcentagem) {
        if (porcentagem > 0.1) {
```

```
            return false;
        }
        double desconto = getValor() * porcentagem;
        setValor(getValor() - desconto);
        return true;
    }
}
```

Repare que, além de um `nome`, `descricao` e `valor`, uma revista também possui uma regra de desconto e é composta pela classe `Editora`. Essa é uma outra classe bastante simples:

```
public class Editora {

    private String nomeFantasia;
    private String razaoSocial;
    private String cnpj;

    // getters e setters
}
```

Precisamos agora evoluir nosso `CarrinhoDeCompras` para que seja possível, além de `Livro`s, adicionar `Revista`s. Uma solução seria duplicar seu método `adiciona`:

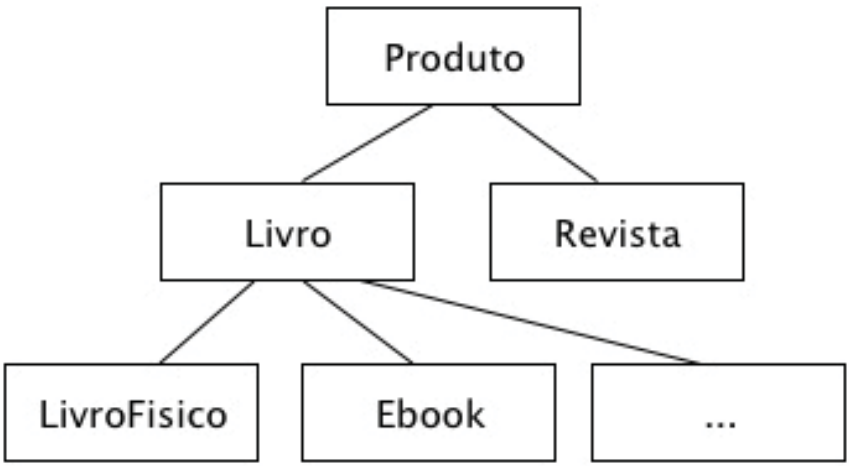
```
public class CarrinhoDeCompras {

    private double total;

    public void adiciona(Livro livro) {
        System.out.println("Adicionando: " + livro);
        livro.aplicaDescontoDe(0.05);
        total += livro.getValor();
    }

    public void adiciona(Revista revista) {
        System.out.println("Adicionando: " + revista);
        revista.aplicaDescontoDe(0.05);
        total += revista.getValor();
```

```
    }

    public double getTotal() {
        return total;
    }
}
```

Mas note como os dois métodos ficaram bastante parecidos, há muita repetição de código! Além disso, para cada novo produto precisaríamos criar um novo método, o que tornaria trabalhosa a evolução dessa classe.

Poderíamos fazer a classe `Revista` herdar de `Livro`, de modo que o *polimorfismo* seria aplicável e teríamos um único método adiciona. Mas ao fazer isso toda `Revista` teria um `Autor`, `ISBN` além dos demais atributos e comportamentos que só se aplicam para os `Livro`s, uma verdadeira bagunça.

Sim, podemos criar um nível a mais na hierarquia de nossas classes adicionando a classe `Produto`, assim `Livro` e `Revista` herdariam de `Produto` e poderíamos utilizar esse tipo no *polimorfismo*. A hierarquia de nossas classes seria:



O grande problema dessa abordagem é que aumentaria demais o **acoplamento entre as classes**, ou seja, o quanto uma classe depende da outra. A herança cria uma relação muito forte entre as classes. Ao mudar a classe `Produto`, por exemplo, todas as *subclasses* (e também *subclasses* das *subclasses*) seriam influenciadas.

E se eu quiser aplicar a mudança apenas para algumas classes? Teríamos que sobrescrever esse novo comportamento de uma forma desnecessária em todas as outras, assim como fizemos com a classe `MiniLivro`. Mesmo sem existir desconto para essa classe, ela foi obrigada a sobrescrever o método `aplicaDescontoDe` para apenas retornar `false`.

Com o passar do tempo, o uso da herança faz com que nossas classes fiquem cada vez mais acopladas e isso dificulta sua evolução.

7.1 O contrato Produto

Em Java, há uma outra forma para se tirar proveito de todos os benefícios do *polimorfismo* sem ter que acoplar tanto as suas classes com vários níveis de herança. Você pode estabelecer um fator em comum entre as classes, criando uma espécie de contrato.

Para esse contrato, não importa a forma como será implementado, a única coisa que importa é que seus métodos (*cláusula*s) sejam implementados de alguma forma. Isso lembra algo? Sim, é bastante parecido com um *método abstrato* cujo corpo você só define na *superclasse* para que todas as *subclasses* herdem a obrigação de implementá-lo.

Esse tipo de contrato Java é conhecido como *Interface*.

> **Não se trata de uma interface gráfica**
>
> É muito comum confundir no inicio, mas não estamos falando de uma interface gráfica de usuário (*GUI*). Também não estamos falando da *interface da classe*, seus métodos públicos. Você perceberá no decorrer do capítulo que se trata de um recurso diferente, um **contrato Java**.

Podemos criar uma interface `Produto`, que por enquanto terá um único método abstrato estabelecendo que todo produto deve ter o método

`getValor`. Uma interface se parece bastante com uma classe abstrata que tenha apenas métodos abstratos, mas no lugar de declará-la como uma classe, utilizamos a palavra reservada `interface`:

```
public interface Produto {

    public abstract double getValor();
}
```

Como todo método *sem corpo* de uma interface é abstrato, o uso do modificador `abstract` é opcional. Não precisamos também adicionar o modificador `public`, pois seus métodos também são públicos por padrão. Podemos simplificar a escrita da interface `Produto` deixando apenas:

```
public interface Produto {

    double getValor();
}
```

Uma interface não pode ter atributos e, até a versão 1.7 da linguagem, também não pode ter nenhum método concreto, ou seja, com implementação. Veremos que, a partir do Java 1.8, isso mudou um pouco.

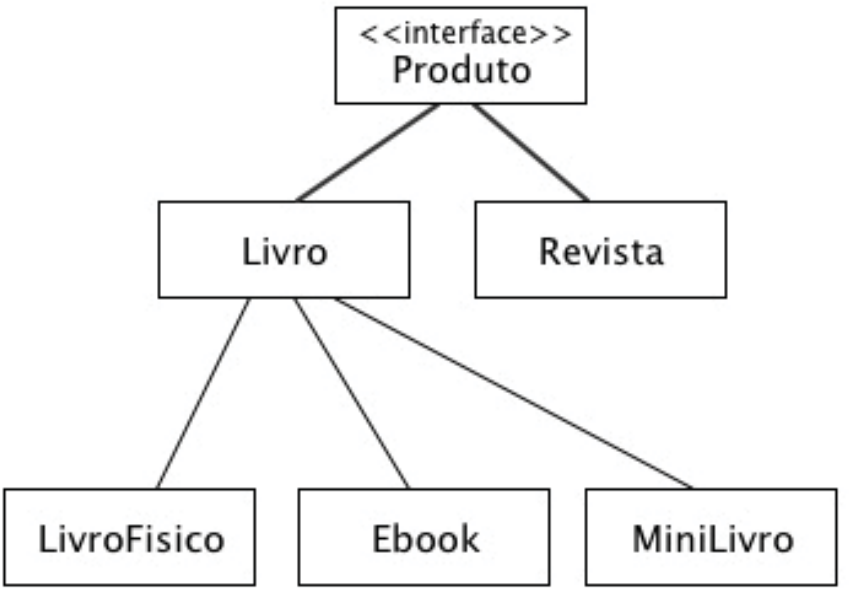
Podemos agora fazer com que todas as classes que idealizam *produtos* de nossa livraria assinem o contrato `Produto`. Para fazer isso, basta adicionar a palavra-chave `implements` seguida do nome da interface que deve ser implementada na declaração das classes, veja:

```
public abstract class Livro implements Produto {
    // atributos e métodos omitidos
}

public class Revista implements Produto {
    // atributos e métodos omitidos
}
```

Como todas essas classes já possuem o método `getValor` declarado, você não perceberá nenhuma diferença. Nosso código passa a funcionar como esperado. Mas é importante perceber que se apagarmos o método

`getValor` a classe deixará de compilar, afinal toda classe que implementa a interface `Produto` é obrigada a implementar seus métodos abstratos.

Veja que com a interface o diagrama de nossas classes fica assim:



Como todos que implementam uma `interface` podem ser referenciados por este tipo, podemos usar *polimorfismo* com *interfaces*. Por exemplo, no método `adiciona` do `CarrinhoDeCompras` podemos receber um `Produto` como parâmetro:

```
public void adiciona(Produto produto) {
    System.out.println("Adicionando: " + produto);
    produto.aplicaDescontoDe(0.16);
    total += produto.getValor();
}
```

O *polimorfismo* funcionará, mas o problema desse código é que nem todo `Produto` tem o método `aplicaDescontoDe`, mas apenas os filhos da classe

`Livro`. Sim, poderíamos mover o método abstrato `aplicaDescontoDe` para a interface `Produto`, mas se nem todos os produtos têm um desconto, devemos evitar isso. Por enquanto, removeremos esse desconto, mas logo veremos uma forma mais interessante de resolver o problema. Nosso método deve ficar assim:

```
public void adiciona(Produto produto) {
    System.out.println("Adicionando: " + produto);
    total += produto.getValor();
}
```

Rode a classe `RegistroDeVendas` para ver que tudo está funcionando como esperado. A classe continua assim:

```
public class RegistroDeVendas {

    public static void main(String[] args) {

        Autor autor = new Autor();
        autor.setNome("Mauricio Aniche");

        LivroFisico fisico = new LivroFisico(autor);
        fisico.setNome("Test-Driven Development");
        fisico.setValor(59.90);

        Ebook ebook = new Ebook(autor);
        ebook.setNome("Test-Driven Development");
        ebook.setValor(29.90);

        CarrinhoDeCompras carrinho = new CarrinhoDeCompras();

        carrinho.adiciona(fisico);
        carrinho.adiciona(ebook);

        System.out.println("Total " + carrinho.getTotal());
    }
}
```

Ao executá-la, o resultado será:

```
Adicionado: LivroFisico@4f23c55
Adicionado: Ebook@21a33b44
Total 89.8
```

7.2 Diminuindo acoplamento com Interfaces

O uso da interface já nos ajudou a resolver o problema de *polimorfismo* de nossos produtos. Agora há uma forma simples e flexível para representar qualquer `Produto` de nossa livraria. Mas ainda precisamos resolver o problema do método `aplicaDescontoDe`.

Ao adicionar esse método abstrato na classe `Livro`, obrigamos todas as suas *subclasses* a implementá-lo, mas não é bem isso que precisamos. Essa solução deixa de ser interessante quando nem todos os `Livro`s tenham esse comportamento, como é o caso do `MiniLivro`, que não pode ter um desconto.

Outra questão é que apenas os filhos da classe `Livro`s têm essa obrigação, sendo que uma `Revista` também deve possuir o método `aplicaDescontoDe`. Podemos diminuir esse acoplamento de uma forma bem simples, criando uma nova interface! Vamos chamá-la de `Promocional`:

```
public interface Promocional {

    boolean aplicaDescontoDe(double porcentagem);
}
```

Agora podemos remover o método `aplicaDescontoDe` abstrato da classe `Livro` e dizer que apenas as classes promocionais, que possuem desconto, implementam essa nova interface:

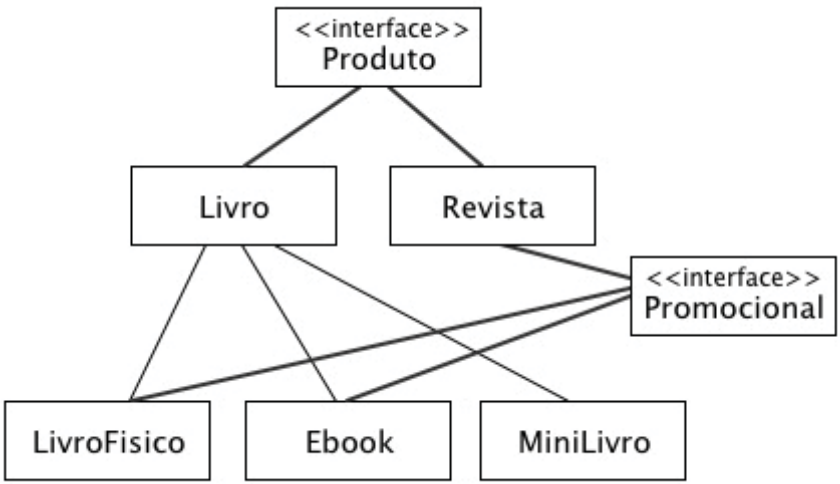
```
public class LivroFisico extends Livro implements Promocional {
    // atributos e métodos omitidos
}

public class Ebook extends Livro implements Promocional {
        // atributos e métodos omitidos
}
```

```
public class Revista implements Produto, Promocional {
    // atributos e métodos omitidos
}
```

Assim como podemos assinar diversos contratos ao longo de nossas vidas, uma classe também pode implementar diversas interfaces. Repare que a classe `Revista` agora implementa duas interfaces, utilizando uma vírgula em sua declaração.

A grande vantagem de trabalhar com interfaces é que apenas as classes que a implementam são obrigadas a implementar seus métodos, portanto, se eu não quero que `MiniLivro` tenha desconto, basta não implementar a interface `Promocional`.

Observe como a estrutura de nossas classes está mais flexível. Qualquer nova classe pode passar a ser promocional, sem herdar nenhuma outra obrigação. O mesmo ocorre com o `Produto`: tudo que uma classe precisa fazer para ter o tipo `Produto` é assinar esse contrato e implementar seu método `getValor`, sem nenhum efeito colateral indesejado.



Você sempre pode e deve favorecer interfaces para criar polimorfismo entre suas classes, seu código fica muito mais flexível e com menor acoplamento.

7.3 Novas regras da interface no Java 8

default methods

Desde o Java 8, uma *interface* pode ter métodos concretos. Com isso, suas implementações não são obrigadas a reescrevê-los. Esse novo recurso é conhecido como *default method*.

Basta adicionar a palavra reservada `default` no início da declaração de um método de interface para que ele possa ter código implementado, por exemplo:

```
public interface Promocional {

    boolean aplicaDescontoDe(double porcentagem);

    default boolean aplicaDescontoDe10Porcento() {
        return aplicaDescontoDe(0.1);
    }
}
```

Dessa forma, toda classe que implementar a interface `Promocional` terá um novo método `aplicaDescontoDe10Porcento`, sem a obrigação de implementar nenhuma linha de código.

Podemos testar essa mudança na classe `RegistroDeVendas`, chamando o método `aplicaDescontoDe10Porcento` no `LivroFisico`:

```
if (fisico.aplicaDescontoDe10Porcento()) {
    System.out.println("Valor agora é " + fisico.getValor());
}
```

Ao executar a classe, teremos como resultado:

```
Valor agora é 53.91

Adicionado: LivroFisico@4f23c557
```

```
Adicionado: Ebook@21a33b44
Total 83.81
```

Mais à frente veremos que diversos *default methods* foram adicionados nas *interfaces* da API do Java. Foi possível evoluir seu comportamento sem quebrar compatibilidade com suas implementações já existentes e bastante utilizadas.

Interface funcional

Uma interface não precisa ter um único método abstrato, mas essa é uma estrutura bem comum. Normalmente, trabalhar com interfaces menores é uma estratégia interessante, afinal temos mais flexibilidade. Se alguém implementar aquela interface é porque realmente precisa do comportamento que ela estabelece.

A partir do Java 8, as *interfaces* que obedecem essa regra de ter um único método abstrato podem ser chamadas de `interface funcional`. Mesmo sem ter esse propósito, nossas interfaces `Produto` e `Promocional` possuem essa estrutura e se enquadram no perfil de uma interface funcional.

Logo veremos que a *interface funcional* é a chave para outros novos recursos da linguagem, que são as expressões *lambda* e *method reference*s.

A anotação `@FunctionalInterface`

Podemos marcar uma interface como funcional explicitamente, para que o fato de ela ser uma *interface funcional* não seja pela simples coincidência de ter um único método abstrato. Para fazer isso, usamos a anotação `@FuncionalInterface`:

```
@FunctionalInterface
public interface Promocional {

    boolean aplicaDescontoDe(double porcentagem);

    default boolean aplicaDescontoDe10Porcento() {
        return aplicaDescontoDe(0.1);
    }
}
```

Ao fazer essa alteração, note que nada mudou. Ela continua compilando e, executando nossas classes de teste, o resultado será o mesmo. Mas diferente do caso de não termos anotado nossa interface com `@FunctionalInterface`, tente alterá-la da seguinte forma, adicionando um novo método *abstrato*:

```
@FunctionalInterface
public interface Promocional {

    boolean aplicaDescontoDe(double porcentagem);
    boolean naoSouMaisUmaInterfaceFuncional();

    default boolean aplicaDescontoDe10Porcento() {
        return aplicaDescontoDe(0.1);
    }
}
```

O código deixará de compilar, com a seguinte mensagem:

```
Invalid '@FunctionalInterface' annotation;
    Promocional is not a functional interface
```

E ao tentar compilar fora do Eclipse, com o `javac` como fizemos no primeiro capítulo, a mensagem seria ainda mais explicativa:

```
java: Unexpected @FunctionalInterface annotation
  Promocional is not a functional interface
    multiple non-overriding abstract methods found
    in interface Promocional
```

Outro detalhe que você já deve ter percebido é que podemos ter um ou mais *default method*s declarados em nossa interface e isso não influencia o fato de ela ser ou não uma `interface funcional`, apenas métodos abstratos são considerados.

Herança múltipla no Java 8?

Métodos defaults foram adicionados para permitir que interfaces evoluam sem quebrar código existente. Essa frase foi bastante repetida na

lista de discussão da especificação dessa nova versão da linguagem. Eles não foram criados para permitir alguma variação de herança múltipla ou de *mix-ins*. Vale lembrar que há uma série de restrições para esses métodos. Em especial, eles não podem acessar atributos de instância, até porque isso não existe em interfaces! Em outras palavras, não há herança múltipla ou compartilhamento de estado.