

Orientação a Obejtos Classica

Namom Alves Alencar



INTERFACES

- O que é uma interface e as diferenças entre herança e implementação;
- Vamos escrever uma interface em Java;
- Utilizá-las como um poderoso recurso para diminuir acoplamento entre as classes

AUMENTANDO NOSSO EXEMPLO

Imagine que um Sistema de Controle do Banco pode ser acessado, além de pelos Gerentes, pelos Diretores do Banco. Então, teríamos uma classe Diretor:

```
class Diretor extends Funcionario {  
    public boolean autentica(int senha) {  
        // verifica aqui se a senha confere com a recebida como  
        parametro  
    }  
}
```

AUMENTANDO NOSSO EXEMPLO

E a classe Gerente:

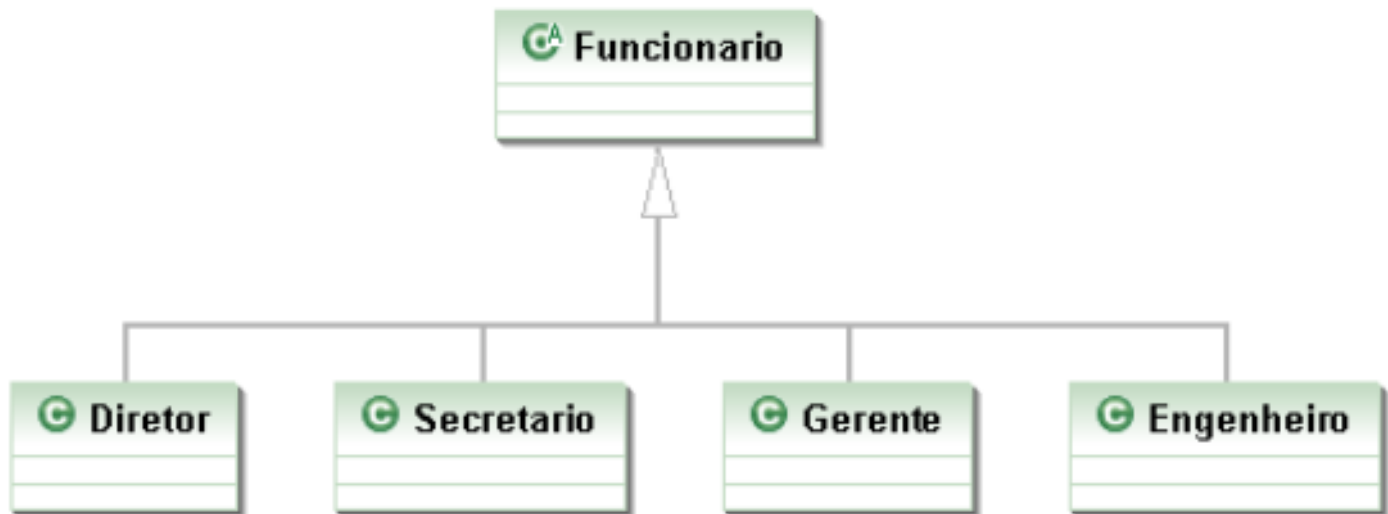
```
class Gerente extends Funcionario {  
    public boolean autentica(int senha) {  
        // verifica aqui se a senha confere com a recebida como  
        parametro  
        // no caso do gerente verifica também se o departamento  
        dele  
        // tem acesso  
    }  
}
```

AUMENTANDO NOSSO EXEMPLO

Repare que o método de autenticação de cada tipo de Funcionario pode variar muito. Mas vamos aos problemas. Considere o SistemaInterno e seu controle: precisamos receber um Diretor ou Gerente como argumento, verificar se ele se autentica e colocá-lo dentro do sistema.

```
class SistemaInterno {  
    void login(Funcionario funcionario) {  
        // invocar o método autentica?  
        // não da! Nem todo Funcionario tem  
    }  
}
```

AUMENTANDO NOSSO EXEMPLO



AUMENTANDO NOSSO EXEMPLO

O SistemaInterno aceita qualquer tipo de Funcionario, tendo ele acesso ao sistema ou não, mas note que nem todo Funcionario possui o método autentica. Isso nos impede de chamar esse método com uma referência apenas a Funcionario (haveria um erro de compilação). O que fazer então?

```
class SistemaInterno {  
    void login(Funcionario funcionario) {  
        funcionario.autentica(...); // não compila  
    }  
}
```

AUMENTANDO NOSSO EXEMPLO

Uma possibilidade é criar dois métodos login no SistemaInterno: um para receber Diretor e outro para receber Gerente. Já vimos que essa não é uma boa escolha. Por quê?

```
class SistemaInterno {  
    // design problemático  
    void login(Diretor funcionario) {  
        funcionario.autentica(...);  
    }  
    // design problemático  
    void login(Gerente funcionario) {  
        funcionario.autentica(...);  
    }  
}
```


AUMENTANDO NOSSO EXEMPLO

Cada vez que criarmos uma nova classe de Funcionario que é autenticável, precisaríamos adicionar um novo método de login no SistemaInterno?!

AUMENTANDO NOSSO EXEMPLO

Uma solução mais interessante seria criar uma classe no meio da árvore de herança, `FuncionarioAutenticavel`:

```
class FuncionarioAutenticavel extends Funcionario {  
    public boolean autentica(int senha) {  
        // faz autenticacao padrão  
    }  
    // outros atributos e métodos  
}
```

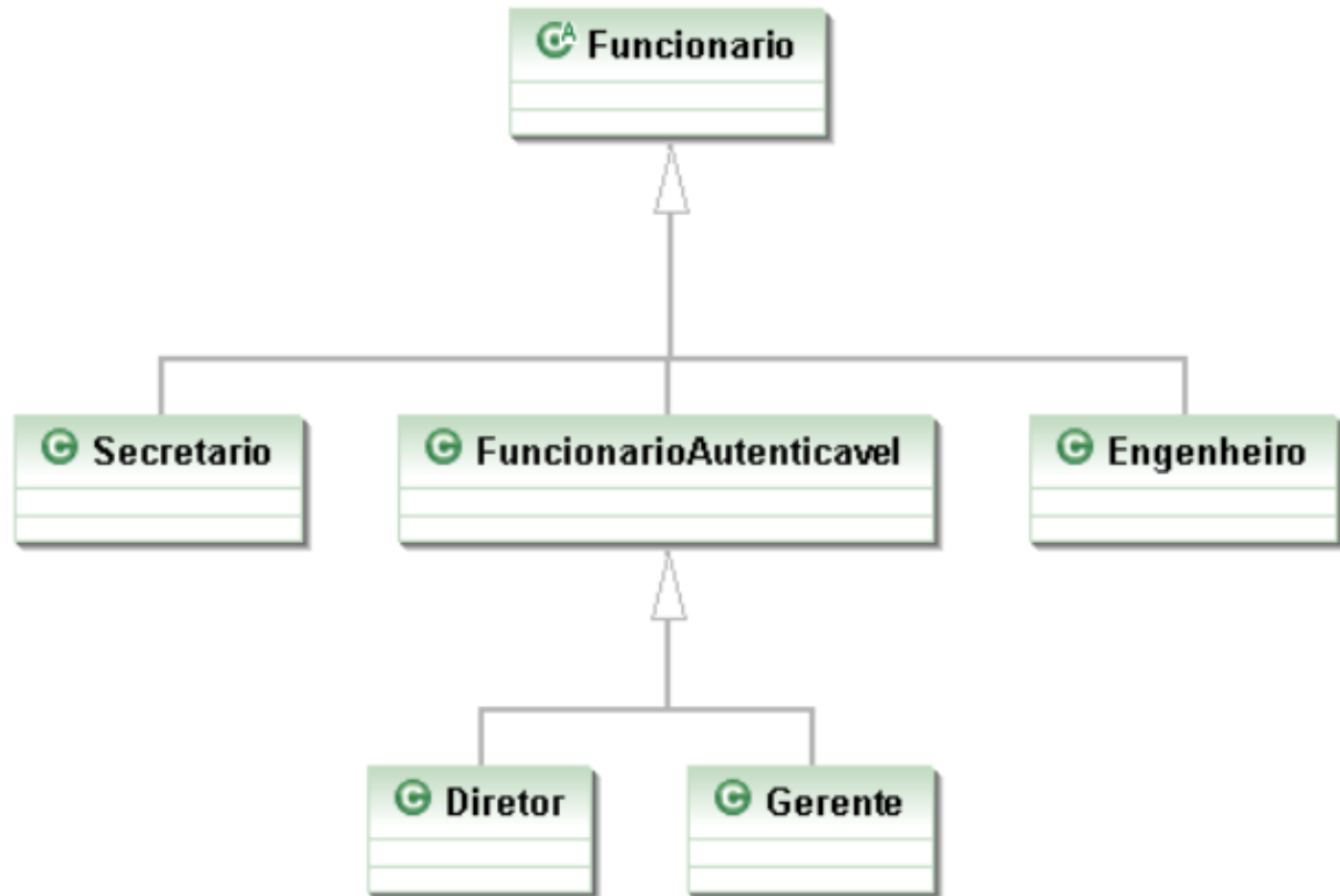
AUMENTANDO NOSSO EXEMPLO

As classes Diretor e Gerente passariam a estender de FuncionarioAutenticavel, e o SistemaInterno receberia referências desse tipo, como a seguir:

```
class SistemaInterno {  
    void login(FuncionarioAutenticavel fa) {  
        int senha = //pega senha de um lugar, ou de um scanner de  
        polegar  
        // aqui eu posso chamar o autentica!  
        // Pois todo FuncionarioAutenticavel tem  
        boolean ok = fa.autentica(senha);  
    } }  

```

AUMENTANDO NOSSO EXEMPLO



AUMENTANDO NOSSO EXEMPLO

Repare que FuncionarioAutenticavel é uma forte candidata a classe abstrata. Mais ainda, o método autentica poderia ser um método abstrato.

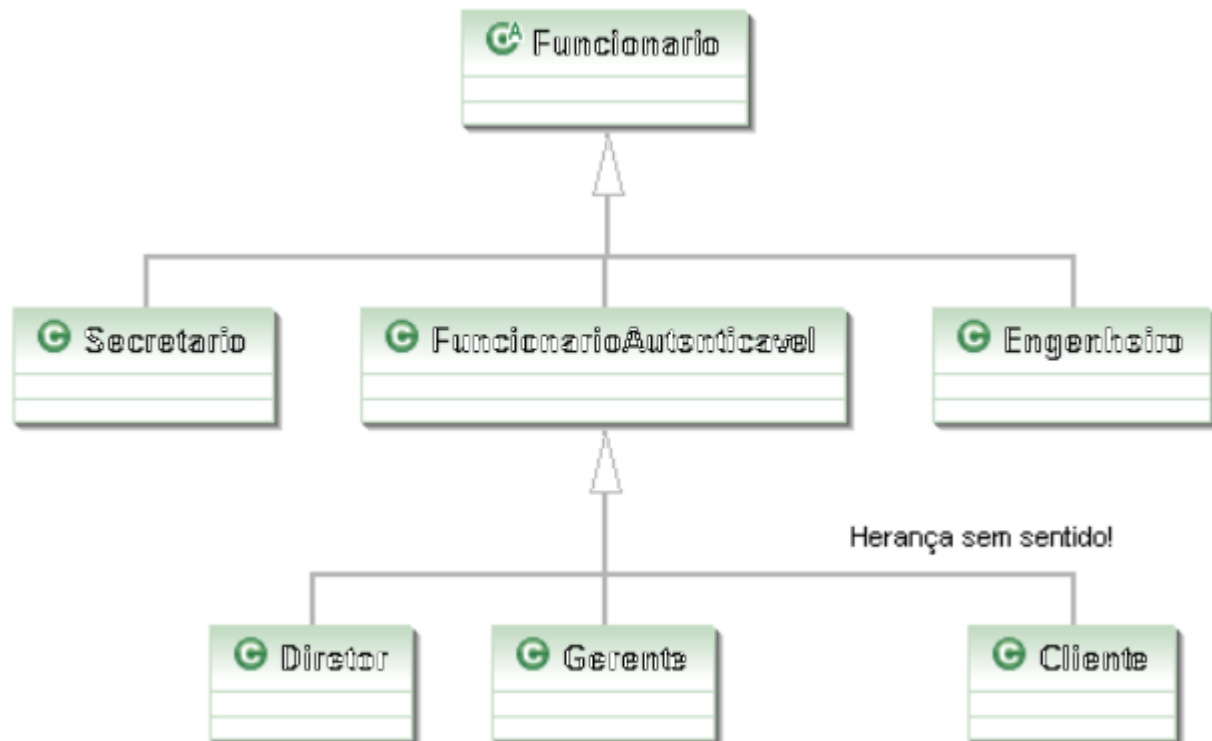
O uso de herança resolve esse caso, mas vamos a uma outra situação um pouco mais complexa: precisamos que todos os clientes também tenham acesso ao SistemaInterno. O que fazer? Uma opção é criar outro método login em SistemaInterno: mas já descartamos essa anteriormente.

AUMENTANDO NOSSO EXEMPLO

Uma outra, que é comum entre os “novatos”, é fazer uma herança sem sentido para resolver o problema, por exemplo, fazer Cliente extends FuncionarioAutenticavel. Realmente, resolve o problema, mas trará diversos outros. Cliente definitivamente não é FuncionarioAutenticavel. Se você fizer isso, o Cliente terá, por exemplo, um método getBonificacao, um atributo salario e outros membros que não fazem o menor sentido para esta classe! Não faça herança quando a relação não é estritamente forte.

AUMENTANDO NOSSO EXEMPLO

Como resolver essa situação? Note que conhecer a sintaxe da linguagem não é o suficiente, precisamos estruturar/desenhar bem a nossa estrutura de classes.



INTERFACES

O que precisamos para resolver nosso problema? Arranjar uma forma de poder referenciar Diretor, Gerente e Cliente de uma mesma maneira, isto é, achar um fator comum.

Se existisse uma forma na qual essas classes garantissem a existência de um determinado método, através de um contrato, resolveríamos o problema.

Toda classe define 2 itens:

- 1 - o que uma classe faz (as assinaturas dos métodos)
- 2 - como uma classe faz essas tarefas (o corpo dos métodos e atributos privados)

INTERFACES

Podemos criar um "contrato" que define tudo o que uma classe deve fazer se quiser ter um determinado status. Imagine:

contrato Autenticavel:

quem quiser ser Autenticavel precisa saber fazer:

1. autenticar dada uma senha, devolvendo um booleano

Quem quiser, pode "assinar" esse contrato, sendo assim obrigado a explicar como será feita essa autenticação. A vantagem é que, se um Gerente assinar esse contrato, podemos nos referenciar a um Gerente como um Autenticavel.

INTERFACES

Podemos criar esse contrato em Java!

```
interface Autenticavel {  
  
    boolean autentica(int senha);  
  
}
```

Chama-se interface pois é a maneira pela qual poderemos conversar com um Autenticavel. Interface é a maneira através da qual conversamos com um objeto.

INTERFACES

Lemos a interface da seguinte maneira: "quem desejar ser autenticável precisa saber autenticar dado um inteiro e retornando um booleano". Ela é um contrato onde quem assina se responsabiliza por implementar esses métodos (cumprir o contrato).

Uma interface pode definir uma série de métodos, mas nunca conter implementação deles. Ela só expõe o que o objeto deve fazer, e não como ele faz, nem o que ele tem. Como ele faz vai ser definido em uma implementação dessa interface.

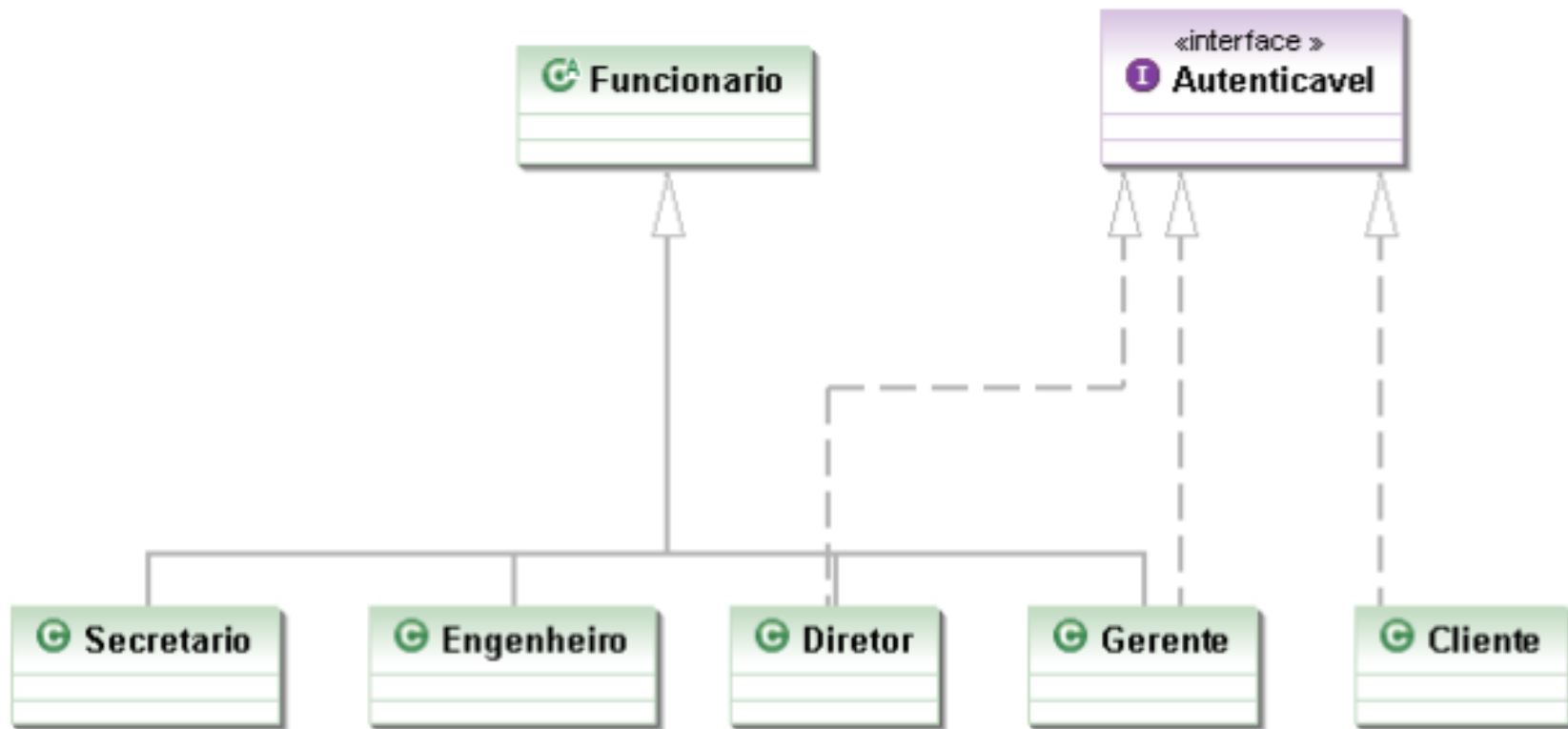
INTERFACES

E o Gerente pode "assinar" o contrato, ou seja, implementar a interface. No momento em que ele implementa essa interface, ele precisa escrever os métodos pedidos pela interface (muito parecido com o efeito de herdar métodos abstratos, aliás, métodos de uma interface são públicos e abstratos, sempre). Para implementar usamos a palavra chave implements na classe:

INTERFACES

```
class Gerente extends Funcionario implements Autenticavel {  
  
    private int senha;  
    // outros atributos e métodos  
    public boolean autentica(int senha) {  
        if(this.senha != senha) {  
            return false;  
        }  
        // pode fazer outras possíveis verificações, como saber se esse  
        // departamento do gerente tem acesso ao Sistema  
        return true;  
    }  
}
```

INTERFACES



INTERFACES

O implements pode ser lido da seguinte maneira: "A classe Gerente se compromete a ser tratada como Autenticavel, sendo obrigada a ter os métodos necessários, definidos neste contrato".

A partir de agora, podemos tratar um Gerente como sendo um Autenticavel. Ganhamos mais polimorfismo! Temos mais uma forma de referenciar a um Gerente. Quando crio uma variável do tipo Autenticavel, estou criando uma referência para qualquer objeto de uma classe que implemente Autenticavel, direta ou indiretamente:

INTERFACES

```
Autenticavel a = new Gerente();
```

```
// posso aqui chamar o método autentica!
```

Novamente, a utilização mais comum seria receber por argumento, como no nosso SistemaInterno:

```
class SistemaInterno {
```

```
    void login(Autenticavel a) {
```

```
        int senha = // pega senha de um lugar, ou de um scanner de polegar
```

```
        boolean ok = a.autentica(senha);
```

```
        // aqui eu posso chamar o autentica!
```

```
        // não necessariamente é um Funcionario!
```

```
        // Mais ainda, eu não sei que objeto a
```

```
        // referência "a" está apontando exatamente! Flexibilidade.
```

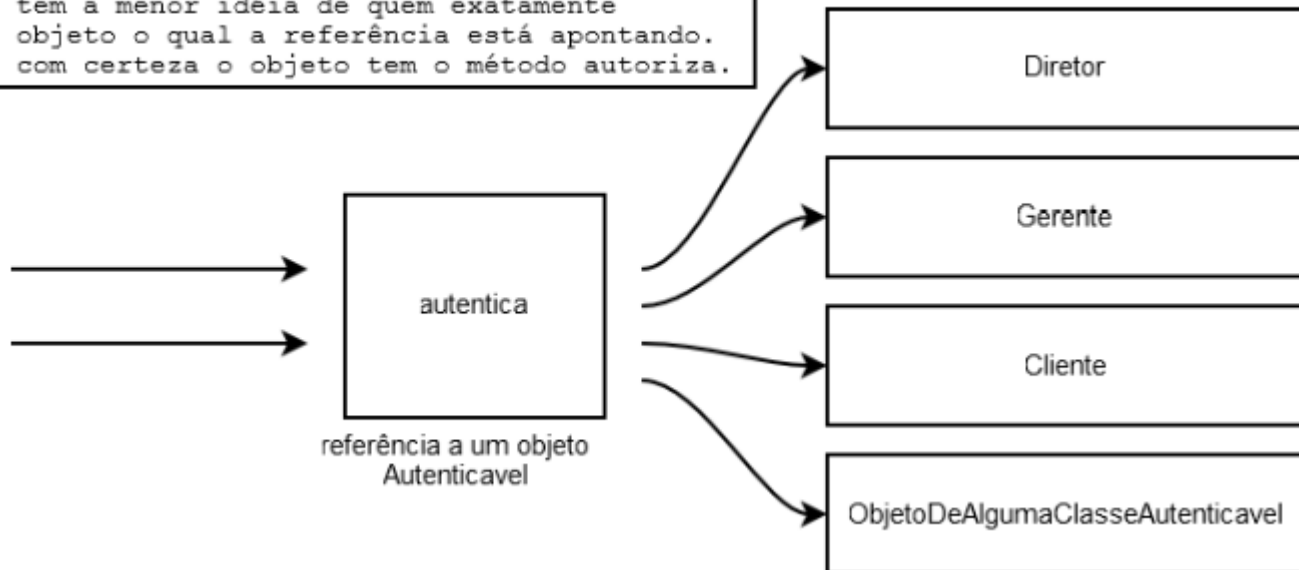
```
    } }
```


INTERFACES

Pronto! E já podemos passar qualquer Autenticavel para o SistemaInterno. Então precisamos fazer com que o Diretor também implemente essa interface.

class Diretor **extends** Funcionario **implements** Autenticavel {
 // métodos e atributos, além de obrigatoriamente ter o autentica
}

Quem está acessando um Autenticavel não tem a menor idéia de quem exatamente é o objeto o qual a referência está apontando. Mas com certeza o objeto tem o método autoriza.



INTERFACES

Podemos passar um Diretor. No dia em que tivermos mais um funcionário com acesso ao sistema, basta que ele implemente essa interface, para se encaixar no sistema.

Qualquer Autenticavel passado para o SistemaInterno está bom para nós. Repare que pouco importa quem o objeto referenciado realmente é, pois ele tem um método autentica que é o necessário para nosso SistemaInterno funcionar corretamente. Aliás, qualquer outra classe que futuramente implemente essa interface poderá ser passada como argumento aqui.

```
Autenticavel diretor = new Diretor();  
Autenticavel gerente = new Gerente();
```

INTERFACES

Ou, se achamos que o Fornecedor precisa ter acesso, basta que ele implemente Autenticavel. Olhe só o tamanho do desacoplamento: quem escreveu o SistemaInterno só precisa saber que ele é Autenticavel.

```
class SistemaInterno {
```

```
    void login(Autenticavel a) {
```

```
        // não importa se ele é um gerente ou diretor
```

```
        // será que é um fornecedor?
```

```
        // Eu, o programador do SistemaInterno, não me preocupo
```

```
        // Invocarei o método autentica
```

```
    }
```

```
}
```

INTERFACES

Não faz diferença se é um Diretor, Gerente, Cliente ou qualquer classe que venha por aí. Basta seguir o contrato! Mais ainda, cada Autenticavel pode se autenticar de uma maneira completamente diferente de outro.

Lembre-se: a interface define que todos vão saber se autenticar (o que ele faz), enquanto a implementação define como exatamente vai ser feito (como ele faz).

INTERFACES

A maneira como os objetos se comunicam num sistema orientado a objetos é muito mais importante do que como eles executam. O que um objeto faz é mais importante do que como ele faz. Aqueles que seguem essa regra, terão sistemas mais fáceis de manter e modificar. Como você já percebeu, esta é uma das ideias principais que queremos passar e, provavelmente, a mais importante de todo esse curso.

EXERCICIOS

1- A sintaxe do uso de interfaces pode parecer muito estranha, à primeira vista.

Vamos começar com um exercício para praticar a sintaxe. Crie um projeto interfaces e crie a interface AreaCalculavel:

```
interface AreaCalculavel {  
    double calculaArea();  
}
```

Crie as classes: **Quadrado** e **Triangulo**.

Repare que, aqui, se você tivesse usado herança, não iria ganhar muito, já que cada implementação é totalmente diferente da outra: um Quadrado, um Retangulo e um Circulo têm atributos e métodos bem diferentes.

EXERCICIOS - EXPLICAÇÃO

Mas, mesmo que eles tivessem atributos em comum, utilizar interfaces é uma maneira muito mais elegante de modelar suas classes. Elas também trazem vantagens em não acoplar as classes. Uma vez que herança através de classes traz muito acoplamento, muitos autores renomados dizem que, na maioria dos casos, herança quebra o encapsulamento.

EXERCICIOS

2 - Crie a seguinte classe de Teste. Repare no polimorfismo. Poderíamos passar esses objetos como argumento para alguém que aceitasse `AreaCalculavel` como argumento:

```
class Teste {  
    public static void main(String[] args) {  
        AreaCalculavel a = new Retangulo(3,2);  
        System.out.println(a.calculaArea());  
    }  
}
```

Opcionalmente, crie a classe `Circulo`:

```
class Circulo implements AreaCalculavel {  
    // ... atributos (raio) e métodos (calculaArea)  
}
```

Utilize `Math.PI * raio * raio` para calcular a área.

DIFICULDADE NO APRENDIZADO DE INTERFACES

Interfaces representam uma barreira no aprendizado do Java: parece que estamos escrevendo um código que não serve pra nada, já que teremos essa linha (a assinatura do método) escrita nas nossas classes implementadoras. Essa é uma maneira errada de se pensar. O objetivo do uso de uma interface é deixar seu código mais flexível e possibilitar a mudança de implementação sem maiores traumas. Não é apenas um código de prototipação, um cabeçalho!

Os mais radicais dizem que toda classe deve ser "interfaceada", isto é, só devemos nos referir a objetos através de suas interfaces. Se determinada classe não tem uma interface, ela deveria ter. Os autores deste material acham tal medida radical demais, porém o uso de interfaces em vez de herança é amplamente aconselhado. Você pode encontrar mais informações sobre o assunto nos livros Design Patterns, Refactoring e Effective Java.

DIFICULDADE NO APRENDIZADO DE INTERFACES

No livro Design Patterns, logo no início, os autores citam 2 regras "de ouro". Uma é "evite herança, prefira composição" e a outra, " programe voltado a interface e não à implementação".

Veremos o uso de interfaces quando falarmos de coleções, o que melhora o entendimento do assunto. O exemplo da interface Comparable também é muito esclarecedor, onde enxergamos o reaproveitamento de código através das interfaces, além do encapsulamento. Para o método `Collections.sort()`, pouco importa quem vai ser passado como argumento. Para ele, basta que a coleção seja de objetos comparáveis. Ele pode ordenar Elefante, Conexao ou ContaCorrente, desde que implementem Comparable.

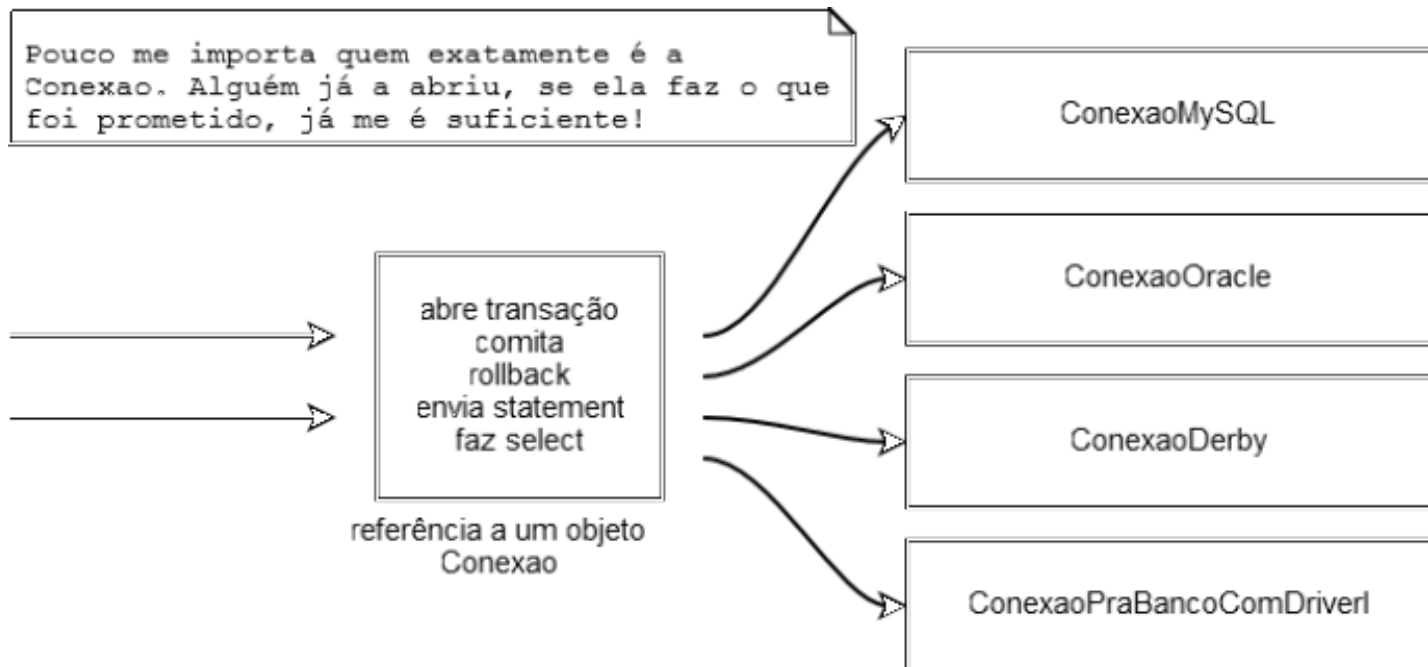
EXEMPLO INTERESSANTE: CONEXÕES COM O BANCO DE DADOS

Como fazer com que todas as chamadas para bancos de dados diferentes respeitem a mesma regra? Usando interfaces!

Imagine uma interface `Conexao` contendo todos os métodos necessários para a comunicação e troca de dados com um banco de dados. Cada banco de dados fica encarregado de criar a sua implementação para essa interface.

Quem for usar uma `Conexao` não precisa se importar com qual objeto exatamente está trabalhando, já que ele vai cumprir o papel que toda `Conexao` deve ter. Não importa se é uma conexão com um Oracle ou MySQL.

EXEMPLO INTERESSANTE: CONEXÕES COM O BANCO DE DADOS



EXEMPLO INTERESSANTE: CONEXÕES COM O BANCO DE DADOS

Apesar do `java.sql.Connection` não trabalhar bem assim, a ideia é muito similar, porém as conexões vêm de uma factory chamada `DriverManager`.

EXERCICIOS

2 - Nosso banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso, vamos criar uma interface no nosso projeto banco já existente:

```
interface Tributavel {  
    double calculaTributos();  
}
```

Lemos essa interface da seguinte maneira: "todos que quiserem ser tributável precisam saber calcular tributos, devolvendo um double".

Alguns bens são tributáveis e outros não, ContaPoupanca não é tributável, já para ContaCorrente você precisa pagar 1% da conta e o SeguroDeVida tem uma taxa fixa de 42 reais.

EXERCICIOS

3 - Vamos criar uma classe TestaTributavel com um método main para testar o nosso exemplo:

```
class TestaTributavel {  
    public static void main(String[] args) {  
        ContaCorrente cc = new ContaCorrente();  
        cc.deposita(100);  
        System.out.println(cc.calculaTributos());  
        // testando polimorfismo:  
        Tributavel t = cc;(A linha em que atribuímos cc a um Tributavel é apenas para você enxergar que é possível fazê-lo. Nesse nosso caso, isso não tem uma utilidade. Essa possibilidade será útil para o próximo exercício.)  
        System.out.println(t.calculaTributos());  
    }  
}
```

Tente chamar o método getSaldo através da referência t, o que ocorre? Por quê?

EXERCICIOS

4 - Crie um GerenciadorDeImpostoDeRenda, que recebe todos os tributáveis de uma pessoa e soma seus valores e inclua nele um método para devolver seu total:

```
class GerenciadorDeImpostoDeRenda {  
    private double total;  
    void adiciona(Tributavel t) {  
        System.out.println("Adicionando tributavel: " + t);  
        this.total += t.calculaTributos();  
    }  
    public double getTotal() {  
        return this.total;  
    }  
}
```


EXERCICIOS

5 - Crie um main para instanciar diversas classes que implementam Tributavel e passar como argumento para um GerenciadorDeImpostoDeRenda. Repare que você não pode passar qualquer tipo de conta para o método adiciona, apenas a que implementa Tributavel. Além disso, pode passar o SeguroDeVida.

6 - Use o método printf para imprimir o saldo com exatamente duas casas decimais:

EXPLICAÇÃO

Repare que, de dentro do `GerenciadorDeImpostoDeRenda`, você não pode acessar o método `getSaldo`, por exemplo, pois você não tem a garantia de que o `Tributavel` que vai ser passado como argumento tem esse método. A única certeza que você tem é de que esse objeto tem os métodos declarados na interface `Tributavel`.

É interessante enxergar que as interfaces (como aqui, no caso, `Tributavel`) costumam ligar classes muito distintas, unindo-as por uma característica que elas tem em comum. No nosso exemplo, `SeguroDeVida` e `ContaCorrente` são entidades completamente distintas, porém ambas possuem a característica de serem tributáveis.

EXPLICAÇÃO

Se amanhã o governo começar a tributar até mesmo PlanoDeCapitalizacao, basta que essa classe implemente a interface Tributavel! Repare no grau de desacoplamento que temos: a classe GerenciadorDeImpostoDeRenda nem imagina que vai trabalhar como PlanoDeCapitalizacao. Para ela, o único fato que importa é que o objeto respeite o contrato de um tributável, isso é, a interface Tributavel. Novamente: programe voltado à interface, não à implementação.

Quais os benefícios de manter o código com baixo acoplamento?

EXERCICIOS++

Vamos criar um novo projeto chame de contaInterface
1 - Transforme a classe Conta em uma interface.

```
public interface Conta {  
    public double getSaldo();  
    public void deposita(double valor);  
    public void saca(double valor);  
    public void atualiza(double taxaSelic);  
}
```

Adapte ContaCorrente e ContaPoupanca para essa modificação:

Algum código vai ter de ser copiado e colado? Isso é tão ruim?

EXERCICIOS++

2 - Às vezes, é interessante criarmos uma interface que herda de outras interfaces: essas, são chamadas subinterfaces. Essas, nada mais são do que um agrupamento de obrigações para a classe que a implementar.

```
interface ContaTributavel extends Conta, Tributavel {  
}
```

Dessa maneira, quem for implementar essa nova interface precisa implementar todos os métodos herdados das suas superinterfaces (e talvez ainda novos métodos declarados dentro dela):

EXERCICIOS++

```
class ContaCorrente implements ContaTributavel {  
    // métodos  
}
```

```
Conta c = new ContaCorrente();  
Tributavel t = new ContaCorrente();
```

Repare que o código pode parecer estranho, pois a interface não declara método algum, só herda os métodos abstratos declarados nas outras interfaces.

Ao mesmo tempo que uma interface pode herdar de mais de uma outra interface, classes só podem possuir uma classe mãe (herança simples).

Bons Estudos

Namom Alves Alencar

