

Programação Orientada a Objetos

Professor Filipe Dwan Pereira

Aula 8 – Interfaces

“Toda palavra de Deus é pura; escudo é para os que confiam Nele”

– Provérbios 30:5

Objetivo desta aula

- Ao término desta aula você será capaz de:
 - Dizer o que é uma interface e as diferenças entre herança e implementação;
 - escrever uma interface em Java;
 - utilizá-las como um poderoso recurso para diminuir acoplamento entre as classes.

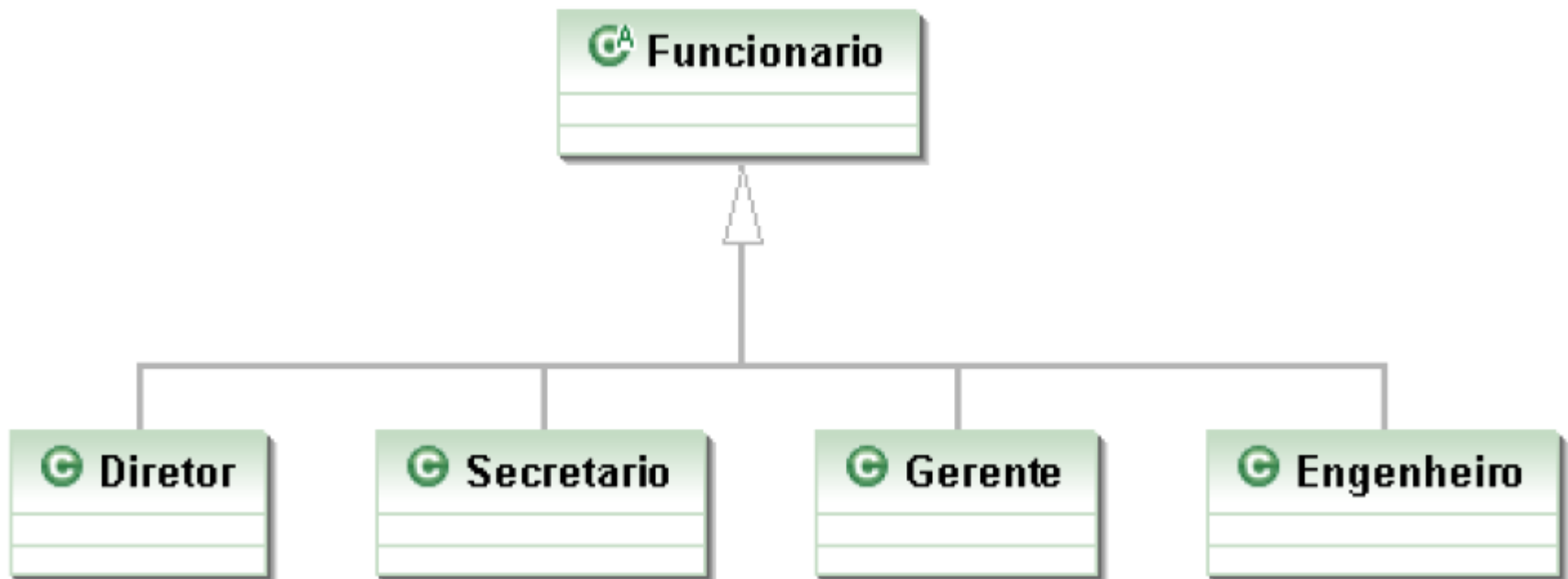
Aumentando nosso exemplo

- 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  
    }  
  
}
```

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  
    }  
}
```



- Perceba que o método de autenticação de cada tipo de Funcionario pode variar muito.
- 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  
    }  
}
```

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).

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

- Uma possibilidade é criar dois métodos login no SistemaInterno: um para receber Diretor e outro para receber Gerente, ou seja uma sobrecarga de métodos (overloading, não confundir com overriding).

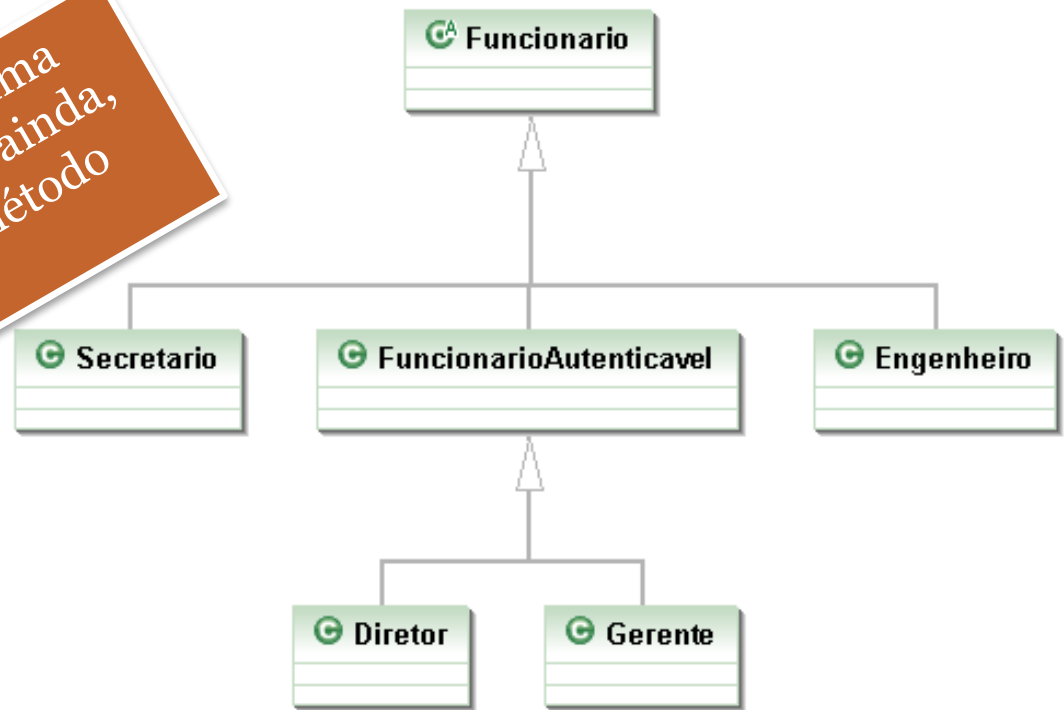
```
class SistemaInterno {  
    void login(Diretor funcionario) {  
        funcionario.autentica(...);  
    }  
    void login(Gerente funcionario) {  
        funcionario.autentica(...);  
    }  
}
```

Essa não é uma boa escolha, pois cada vez que criarmos uma nova classe de Funcionario que é autenticável, precisaríamos adicionar um novo método de login no SistemaInterno.

Que tal nós criarmos 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  
}
```

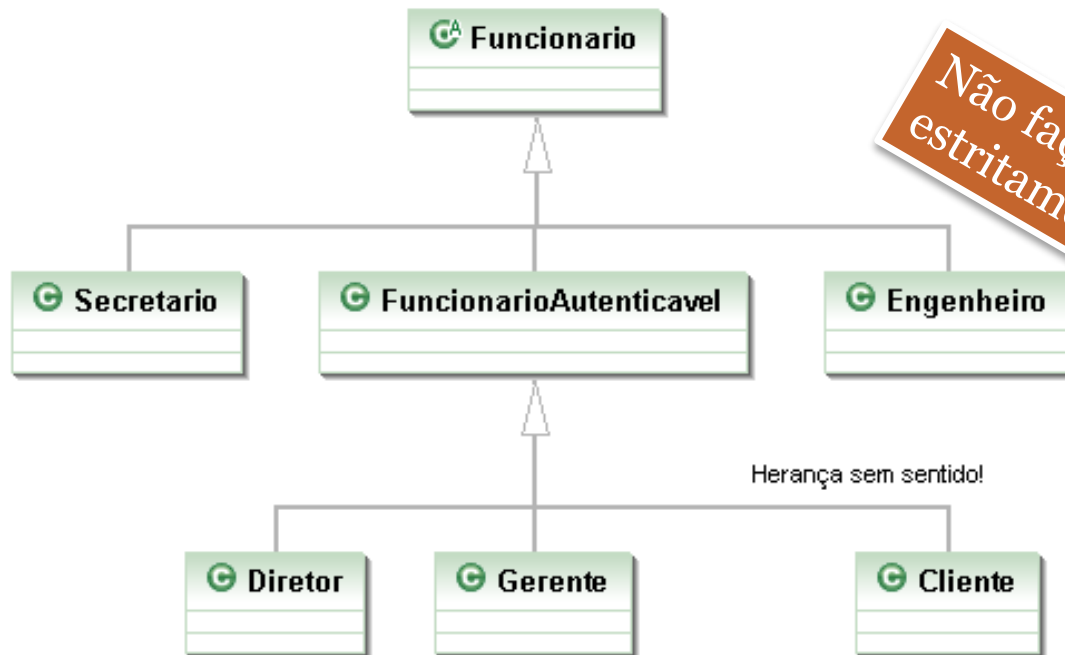
Repare que FuncionarioAutenticavel é uma forte candidata a classe abstrata. Mais ainda, o método autentica poderia ser um método abstrato.



O SistemaInterno receberia referências desse tipo:

```
class SistemaInterno {  
    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);  
  
}  
}
```

- Precisamos que todos os clientes também tenham acesso ao SistemaInterno. O que fazer?
 - Fazer Cliente extends FuncionarioAutenticavel.
 - Isso trará diversos problemas. Cliente **não é um** 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 "é um"

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.
- Toda classe define os itens:
 - o que uma classe faz (as assinaturas dos métodos)
 - como uma classe faz essas tarefas (o corpo dos métodos e atributos privados)

- Podemos criar um “contrato” que define tudo o que uma classe deve fazer

`contrato Autenticavel:`

`quem quiser ser Autenticavel precisa saber fazer:`

`1.autenticar dada uma senha, devolvendo um booleano`

- Quem quiser, pode “assinar” esse contrato, sendo 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.
- Podemos criar esse contrato em Java!

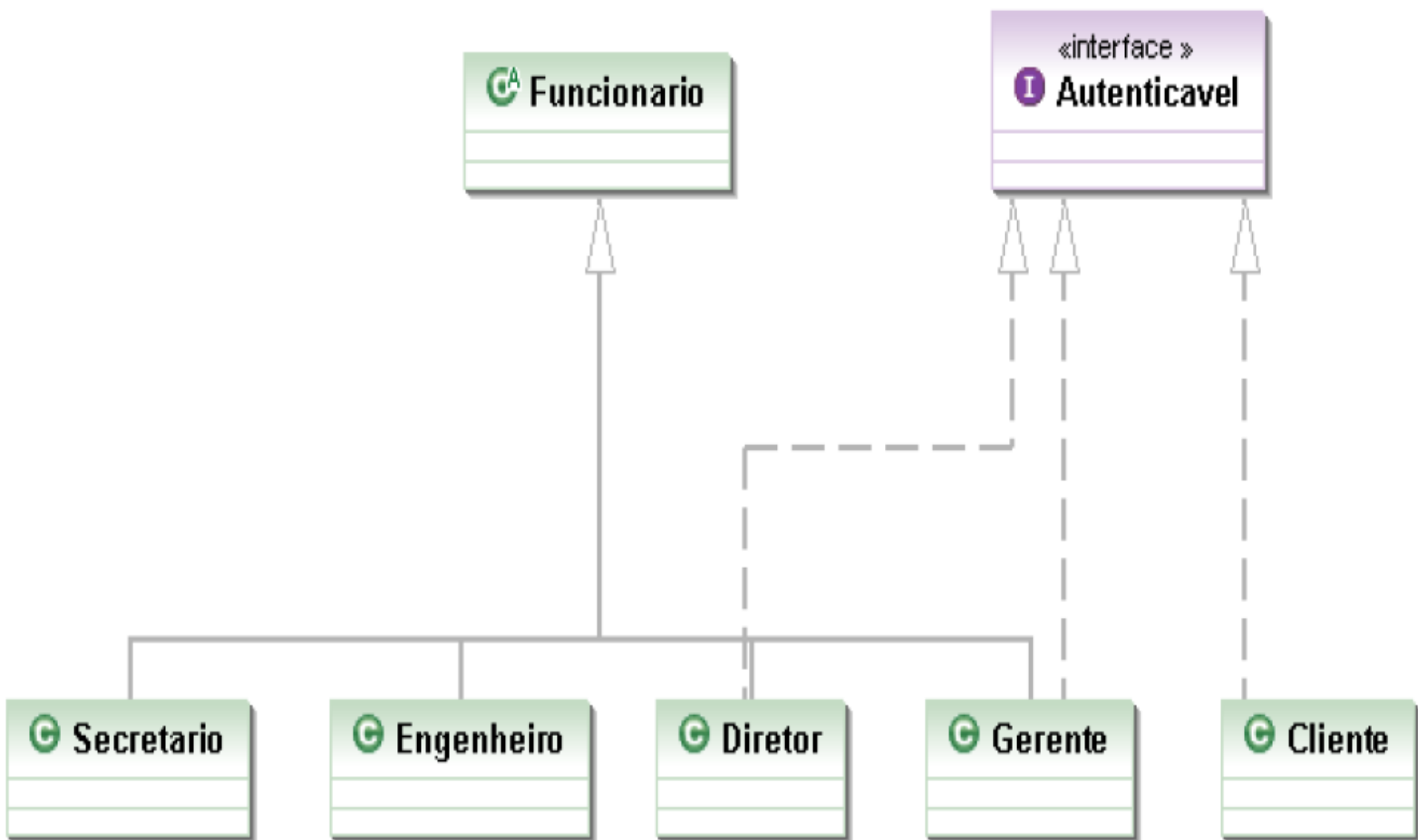
```
interface Autenticavel {  
    boolean autentica(int senha);  
}
```

Interfaces

- Uma interface pode definir uma série de métodos, mas não conter implementação deles.
- A interface 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.
- E o Gerente pode “assinar” o contrato, ou seja, **implementar** a interface.
- **Obs.: a partir do Java8 é possível colocar a implementação (como ele faz) dos métodos conforme veremos.**

Para implementar usamos a palavra chave **implements** na classe:

```
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;  
    }  
}
```



- 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**:

```
Autenticavel a = new Gerente();  
// posso aqui chamar o método autentica!
```

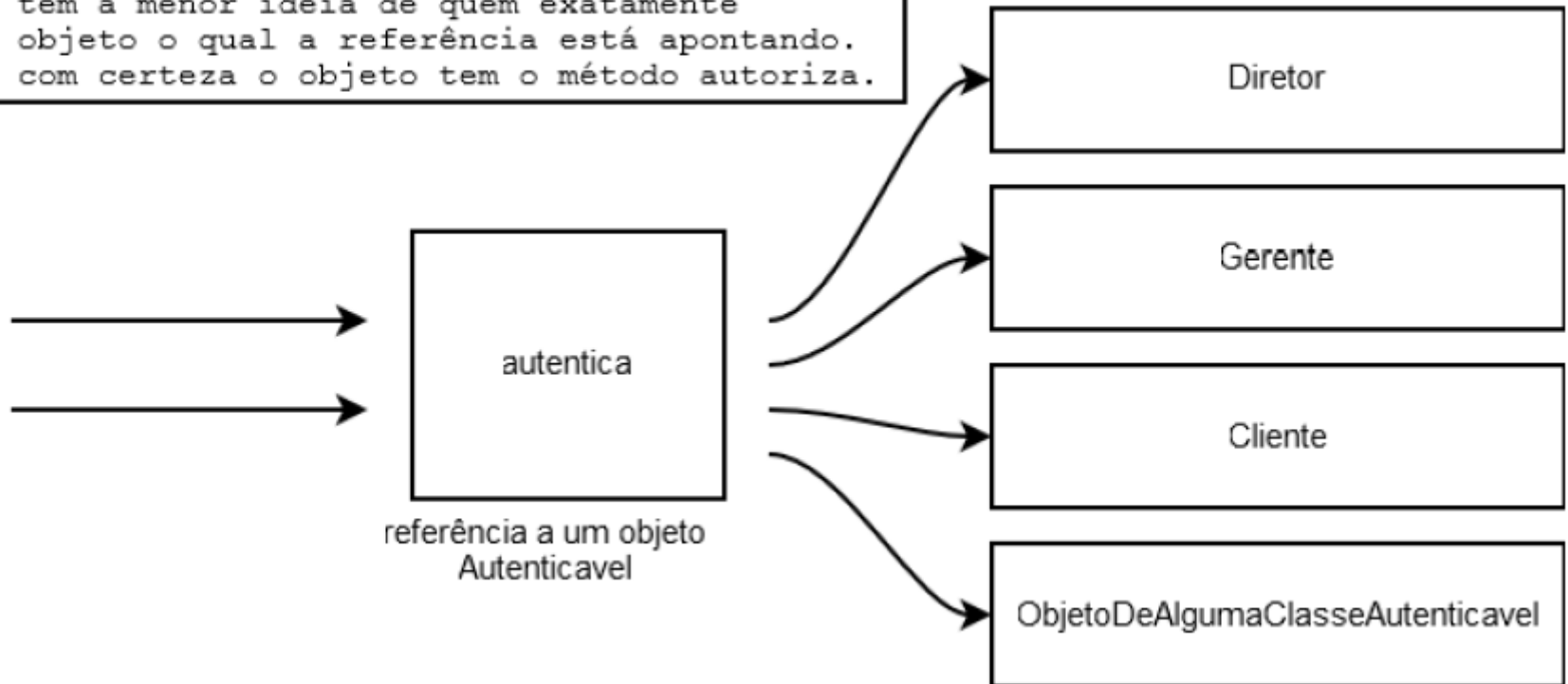

A utilização mais comum seria receber por argumento, como no nosso SistemaInterno:

```
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.  
    }  
  
}
```

Diretor também implemente essa interface.

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

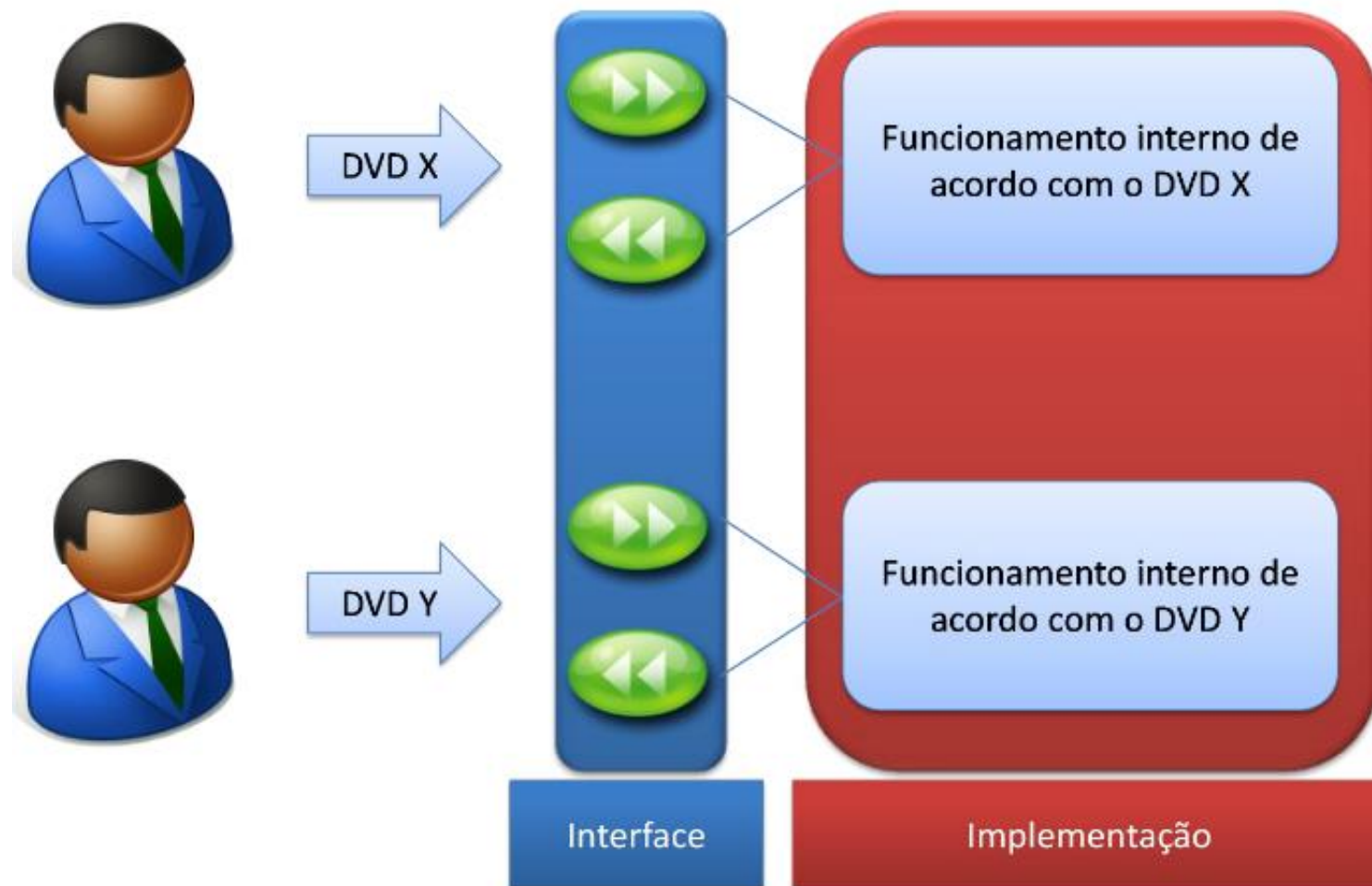
Quem está acessando um Autenticavel não tem a menor idéia de quem exatamente é o objeto o qual a referência está apontando. Mas com certeza o objeto tem o método autoriza.



- E se o Fornecedor também precisar ter acesso ao sistema interno?
 - 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  
    }  
  
}
```

Interface - outro exemplo



Um pouco mais...

```
public interface Contrato1 {  
    public void metodoDoContrato1();  
}
```

```
public interface Contrato2 {  
    public void metodoDoContrato2();  
}
```

```
public interface Contrato3 extends Contrato2, Contrato1 {  
    public void metodoDoContrato3();  
}
```

```
public class SegueContrato implements Contrato3 {  
    public void metodoDoContrato1() {  
        //...  
    }  
    public void metodoDoContrato2() {  
        //...  
    }  
    public void metodoDoContrato3() {  
        //...  
    }  
}
```

```
public class SegueDoisContratos implements Contrato1, Contrato2 {  
    public void metodoDoContrato1() {  
        //...  
    }  
    public void metodoDoContrato2() {  
        //...  
    }  
}
```

Um pouco mais:

É possível que uma classe abstrata implemente uma Interface.

```
public abstract class Teste implements Contrato1, Contrato2 {  
  
    @Override  
    public abstract void metodoContrato2();  
  
    @Override  
    public abstract void metodoContrato1();  
  
}
```

Nesse caso a Classe Abstrata delegou a obrigação de implementar os métodos dos Contrato's para as filhas. Perceba que a Classe Abstrata poderia também implementá-los.

```
public class TesteFilha extends Teste {  
  
    @Override  
    public void metodoContrato2() {  
        System.out.println("Metodo implementado do Contrato 2");  
    }  
  
    @Override  
    public void metodoContrato1() {  
        System.out.println("Metodo implementado do Contrato 1");  
    }  
  
}
```

Um pouco mais

- 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.
- Essa é uma das ideias mais importantes em programação orientada a objetos.

Interfaces no Java8

- Em uma famosa entrevista em 2005, Erick Gamma afirma:
 - In Java when you add a new method to an interface, you break all your clients.
 - When you have an abstract class, you can add a new method and provide a default implementation in it. All the clients will continue to work.
 - As always there is a trade-off, an interface gives you freedom with regard to the base class, an abstract class gives you the freedom to add new methods later.

Vamos discutir sobre isso...

Interfaces no Java 8

- *A interface define métodos, mas não os implementa*
 - Com **exceção** de métodos que usam os modificadores **default** e **static**
- Por via de regra a implementação é de responsabilidade de quem implementa a interface

```
public interface AreaCalculavel {  
    public double calcularArea();  
}
```

Ao invés de *class*,
interface é utilizada

Numa interface, nenhum
método é implementado

Interfaces não possuem
atributos (só constantes)

Métodos Default

- Uma interface pode definir métodos com o modificador default
- Neste caso, o método é implementado diretamente na interface
- Este recurso surgiu no Java 8, a fim de permitir o suporte à expressões lambda em interfaces que já faziam parte da linguagem

Definindo Métodos Default

```
public interface Calculator {  
  
    double calculate();  
  
    default double calculatePow(double x, int y) {  
        return Math.pow(x, y);  
    }  
}
```

O método default é implementado na interface

```
public class MyCalculator implements Calculator {  
    public double calculate() {  
        //...  
    }  
}
```

```
MyCalculator my = new MyCalculator();  
double x = my.calculatePow(10, 3);
```


Funciona como um método qualquer

Métodos estáticos

- Interfaces também podem implementar métodos definidos com o modificador **static**. A partir do Java 8, também.
- O método é acessível diretamente pela interface, sem precisar que ocorra a criação de objetos


Definindo Métodos Estáticos

```
public interface Calculator {  
  
    double calculate();  
  
    static double calculatePow(double x, int y) {  
        return Math.pow(x, y);  
    }  
}
```



O método estático é implementado na interface

```
Calculator.calculatePow(10, 3);
```



Método chamado diretamente na interface *Calculator*

Exemplo Interfaces Java 8:

```
public interface Interfacel {  
  
    public double calcula();  
  
    public default double potencia(double base, double expoente) {  
        return Math.pow(base, expoente);  
    }  
  
    public static int fatorial (int n) {  
        int fat = 1;  
        if (n==0 || n==1) return 1;  
        for (int i=n; i>1; i--) {  
            fat = fat*i;  
        }  
        return fat;  
    }  
}
```

```
public class ImplementeInterfacel implements Interfacel {  
  
    @Override  
    public double calcula() {  
        System.out.println("Calcula alguma coisa...");  
        return 0;  
    }  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        ImplementeInterfacel teste = new ImplementeInterfacel();  
        teste.calcula();  
        System.out.println("2^4 = "+teste.potencia(2, 4));  
        System.out.println("Fatorial de 5 = "+Interfacel.fatorial(5));  
    }  
}
```

Calcula alguma coisa...
2^4 = 16.0
Fatorial de 5 = 120

Herança entre interfaces

- Diferentemente das classes, uma interface pode herdar de mais de uma interface. É como um contrato que depende que outros contratos sejam fechados antes deste valer.
- Você não herda métodos e atributos, mas sim responsabilidades.

Outras Considerações

- Classes podem **estender** outra classe, mas apenas podem **implementar** interfaces
- Uma classe pode implementar uma ou mais interfaces

Classes Abstratas ou Interfaces?

- A escolha entre classes abstratas ou interfaces tem dois aspectos
 - **Conceitual**
 - Classes abstratas são classes que não podem ter instâncias
 - Interfaces determinam como um objeto será exposto
 - **Prático**
 - Uma classe pode implementar mais de uma interface
 - Uma classe abstrata pode conter atributos
- Classes abstratas e interfaces têm o objetivo comum de favorecer o uso do polimorfismo

Dificuldade no aprendizado de Interfaces

- No início parece que estamos escrevendo um código inútil.
- Mas o objetivo do uso de uma interface é deixar seu código mais flexível e possibilitar a mudança de implementação sem maiores traumas.
- O uso de interfaces em vez de herança é amplamente aconselhado.

Exercício em sala

- Cria a interface **Automovel** com os métodos: **irParaEsquerda()**, **irParaDireita()**, **frear()**, **acelerar()**, **buzinar()**.
 - Cria a classe **Ferreri**, **Porsche** e **Fusca**. O métodos devem dizer por exemplo: **Fusca** indo para esquerda ou **Ferrari** indo para esquerda...
- Crie a classe **Rota** com o método:
 - **irParaPosto (Automovel automovel)**. No método **ir** você deverá: acelerar, irParaEsquerda, irParaDireita e depois frear.
 - **irParaUfrj (Automovel automovel)**. No método **ir** você deverá: acelerar, irParaDireita, irParaEsquerda, frear e depois buzinar.
- Crie uma interface chamada **ItemCaro** que deverá ter o método **getPreco**. A **Ferreri** deverá implementar essa interface.
 - Crie a Classe **ReligioRolex** que também deverá implementar a interface **ItemCaro**.
 - Crie uma classe **VendedorDeLuxo** com o método:
 - **mostrarPreco (ItemCaro)**.
- Crie agora uma **interface** chamada **Possante** que **herdará** de **Automóvel**. Crie na interface **Possante** o método **sobre()**. Faça o **Fusca** implementar essa interface.
- Agora crie o método **derrapar()** na interface **Automovel**. Para derrapar ele precisará acelerar, acelerar e frear. Perceba esse método será **default** ou **static**

Sugestão de Leitura

- Entrevista com Erich Gamma sobre programar orientado a interfaces (e não a implementações) e sobre o uso de composição de objetos e herança.
 - <http://www.artima.com/lejava/articles/designprinciples.html>
- Leia o capítulo 10 da apostila fj11 – Herança, reescrita e polimorfismo:
 - <http://www.caelum.com.br/apostila-java-orientacao-objetos/>
- Leia o capítulo 6– Interfaces e Classes Internas:
 - HORSTMANN, CORNELL. Core Java Volume I – Fundamentos, 8º Edição. São Paulo, Pearson Education, 2010.

Referências Bibliográficas

- DEITEL, Harvey M. e DEITEL, Paul J. Java - Como Programar, 8ª edição. Pearson. 2010.
- BLOCH, Joshua. Effective Java, 2ª edição. Addison-Wesley, 2008.
- CAELUM. Java e Orientação a Objetos. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/>
- SOFTBLUE. Professor Carlos Eduardo Gusso Tosin. Fundamentos de Java. <http://www.softblue.com.br/>.
- K19. Java e Orientação a Objetos. Disponível em: <http://www.k19.com.br/cursos/orientacao-a-objetos-em-java>.
- HORSTMANN, CORNELL. Core Java Volume I – Fundamentos, 8ª Edição. São Paulo, Pearson Education, 2010.
- BRAUDE, E. J. Projeto de software - da programação à arquitetura: uma abordagem baseada em Java. Porto Alegre: Bookman, 2005.
- SANTOS, R. Introdução à Programação Orientada a Objetos usando Java. São Paulo: Campus, 2003.
- Slides do Professor Doutor Horácio Fernandes da UFAM.