

Aula 20: Interfaces

Professor(a): Virgínia Fernandes Mota

<http://www.dcc.ufmg.br/~virginiaferm>

ALGORITMOS E ESTRUTURAS DE DADOS - SETOR DE INFORMÁTICA



- 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.
- Repare que o método de autenticação de cada tipo de Funcionario pode variar muito.

```
1 class Diretor extends Funcionario {
2     public boolean autentica(int senha) {
3         // verifica aqui se a senha confere com a recebida
4           como parametro
5     }
6 }
```

```
1 class Gerente extends Funcionario {
2     public boolean autentica(int senha) {
3         // verifica aqui se a senha confere com a recebida
4           como parametro
5         // no caso do gerente verifica também se o
6           departamento dele
7         // tem acesso
8     }
9 }
```

Aumentando nosso exemplo

- 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.

```
1  class SistemaInterno {  
2  
3      void login(Funcionario funcionario) {  
4          // invocar o método autentica?  
5          // não da! Nem todo Funcionario tem  
6      }  
7  }
```

Aumentando nosso exemplo

- Uma solução: criar dois métodos login no SistemaInterno - um para receber Diretor e outro para receber Gerente.

```
1  class SistemaInterno {  
2  
3      // design problemático  
4      void login(Diretor funcionario) {  
5          funcionario.autentica(...);  
6      }  
7  
8      // design problemático  
9      void login(Gerente funcionario) {  
10         funcionario.autentica(...);  
11     }  
12  
13 }
```

- **sobrecarga** (overloading) de método.

Aumentando nosso exemplo

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

```
1 class FuncionarioAutenticavel extends Funcionario {  
2  
3     public boolean autentica(int senha) {  
4         // faz autenticacao padrão  
5     }  
6  
7     // outros atributos e métodos  
8  
9 }
```

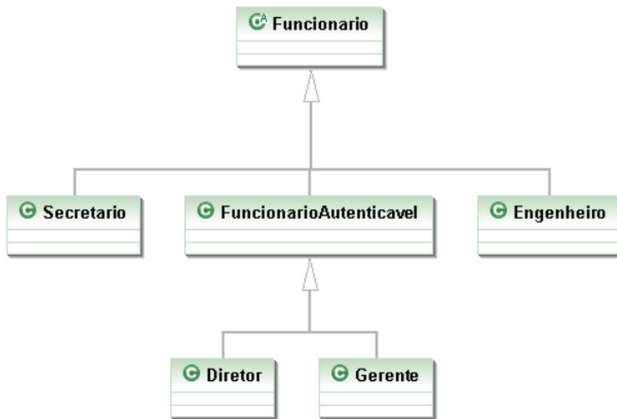
- As classes Diretor e Gerente passariam a estender de FuncionarioAutenticavel.

Aumentando nosso exemplo

- SistemaInterno receberia referências desse tipo:

```
1  class SistemaInterno {  
2  
3      void login(FuncionarioAutenticavel fa) {  
4  
5          int senha = //pega senha de um lugar, ou de um  
                        scanner de polegar  
6  
7          // aqui eu posso chamar o autentica!  
8          // Pois todo FuncionarioAutenticavel tem  
9          boolean ok = fa.autentica(senha);  
10  
11      }  
12  }
```

Aumentando nosso exemplo



- **FuncionarioAutenticavel** é uma forte candidata a classe abstrata. O método autentica poderia ser um método abstrato.

Aumentando nosso exemplo

- Problemas?
- E se precisarmos que todos os clientes também tenham acesso ao SistemaInterno?

Aumentando nosso exemplo

- Cliente definitivamente **não** é FuncionarioAutenticavel! → herança sem sentido não resolve!
- Como resolver essa situação?

Aumentando nosso exemplo

- Problema: Arranjar uma forma de referenciar Diretor, Gerente e Cliente de uma mesma maneira.
- Sabemos: Toda classe define o que uma classe faz e como ela faz.
- Queremos: criar um "contrato" que define tudo o que uma classe deve fazer se quiser ter um determinado status.
 - *contrato Autenticavel: quem quiser ser Autenticavel precisa saber autenticar dada uma senha, devolvendo um booleano*

- Solução: Interface!

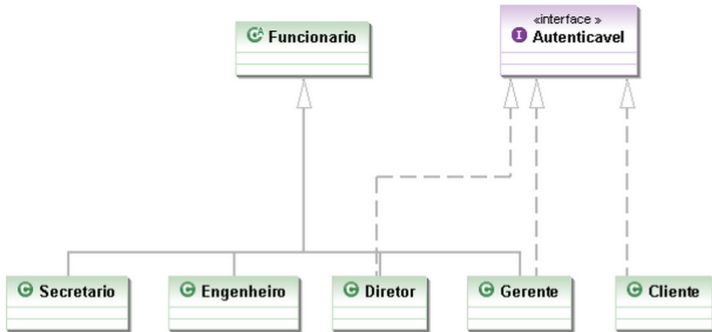
```
1 interface Autenticavel {  
2  
3     boolean autentica(int senha);  
4  
5 }
```

- 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!

Interfaces

```
1 class Gerente extends Funcionario implements Autenticavel {
2
3     private int senha;
4
5     // outros atributos e métodos
6
7     public boolean autentica(int senha) {
8         if(this.senha != senha) {
9             return false;
10        }
11        // pode fazer outras possíveis verificações, como saber
12           se esse
13        // departamento do gerente tem acesso ao Sistema
14
15        return true;
16    }
17 }
```

Interfaces



- Podemos tratar um Gerente como sendo um Autenticavel. Ganhamos mais polimorfismo!

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

- Quando crio uma variável do tipo Autenticavel, estou criando uma referência para qualquer objeto de uma classe que implemente Autenticavel.

- Nosso SistemaInterno fica então:

```
1
2 class SistemaInterno {
3
4     void login(Autenticavel a) {
5         int senha = // pega senha de um lugar, ou de um
6                     scanner de polegar
7         boolean ok = a.autentica(senha);
8
9         // aqui eu posso chamar o autentica!
10        // não necessariamente é um Funcionario!
11        // Mais ainda, eu não sei que objeto a
12        // referência "a" está apontando exatamente!
13        Flexibilidade.
14    }
15 }
```


- 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.
- 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).

- Diferentemente das classes, uma interface pode herdar de mais de uma interface.
- *Você não herda métodos e atributos, mas sim responsabilidades.*

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!
- Faremos uma `ConnectionFactory` no terceiro trimestre!

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

```
1 interface AreaCalculavel {  
2     double calculaArea();  
3 }
```

- Crie algumas classes que são AreaCalculavel: Quadrado, Retangulo, Círculo.
- Crie uma classe Teste com o main.

- 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:

```
1 interface Tributavel {  
2     double calculaTributos();  
3 }
```

- 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.
- Crie a classe SeguroDeVida, claro ;)
- Crie uma classe TestaTributavel com um método main.

- 3. Crie um GerenciadorDeImpostoDeRenda, que recebe todos os tributáveis de uma pessoa e soma seus valores e inclua nele um método para devolver seu total.
- Obs: Use o método printf para imprimir o saldo com exatamente duas casas decimais:

```
1      System.out.printf("O saldo é: %.2f", cc.getSaldo())  
      ;
```

- Vimos que utilizando interfaces você pode especificar comportamentos semelhantes para classes possivelmente não relacionadas.
- Há uma preocupação "atual" com as pegadas de carbono (emissões anuais de gás carbônico na atmosfera) a partir de instalações que queimam vários tipos de combustíveis para aquecimento, veículos que queimam combustíveis para se mover e assim por diante.

- 1. Nesse cenário:
 - Crie três pequenas classes não relacionadas por herança - Building, Car e Bicycle. Dê a cada classe alguns atributos e comportamentos únicos que ela não tem em comum com as outras classes.
 - Building: número de pessoas (int), uso de energia renovável (boolean), número de lâmpadas (int)...
 - Car: Combustível, cilindrada...
 - Escreva uma interface CarbonFootprint com um método getCarbonFootprint. Faça cada uma das suas classes implementar essa interface, para que seu método getCarbonFootprint calcule uma pegada de carbono apropriada a cada classe.
 - Crie a classe Teste.

- 2. Modifique o código tornando Building uma classe abstrata e implementando duas novas subclasses concretas House e School.

Na próxima aula...

Exceções e controle de erros