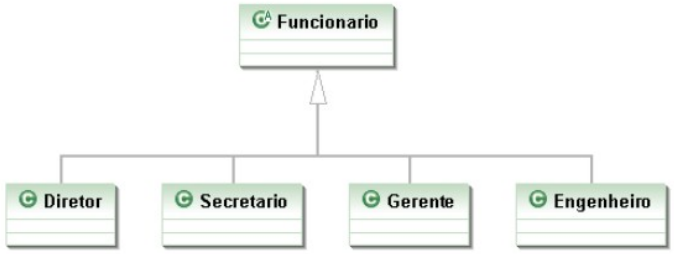
# INTERFACES

Para explicar o conceito de INTERFACES na Programação Orientada a Objetos e ver como se programa uma interface no Java vamos diretamente a um exemplo.

Imagine que um Sistema de Controle do Banco pode ser acessado, além de pelos Gerentes, pelos Diretores, Secretarios e Engenheiros do Banco. Então, teríamos a seguinte estrutura de classes em UML:




Tínhamos uma classe Gerente que era subclasse de Funcionario e tinha um método para autenticar o Gerente para o acesso ao sistema:

```
class Gerente extends Funcionario {
  private int senha;

  public boolean autentica(int senha) {
    if (this.senha == senha) {
      System.out.println("Acesso Permitido!");
      return true;
    } else {
      System.out.println("Acesso Negado!");
      return false;
    }
  }
}
```

Agora vamos supor que temos também a subclasse Diretor que também tem um método para autenticar o Diretor no sistema:

```
class Diretor extends Funcionario {
   private int senha1;
   private int senha2;

  public boolean autentica(int senha) {
    if (this.senha == senha1) && (this.senha ==senha2) {
      System.out.println("Acesso Permitido!");
      return true;
    } else {
      System.out.println("Acesso Negado!");
      return false;
    }
  }
}
```

E temos também a subclasse Secretario que não tem o método autentica():

```
class Secretario extends Funcionario {

}
```

O método de autenticação de cada tipo de Funcionario pode variar muito. Isso gera problemas para a programação. Considere o SistemaInterno e seu controle: precisamos receber um Diretor ou Gerente ou um Secretario 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). O que fazer então?

```
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. 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(...);
  }

}
```

Cada vez que criarmos uma nova subclasse de Funcionario que é autenticável, precisaríamos adicionar um novo método de login no SistemaInterno.

Métodos com mesmo nome

Em Java, métodos podem ter o mesmo nome desde que não sejam ambíguos, isto é, que exista uma maneira de distinguir no momento da chamada.

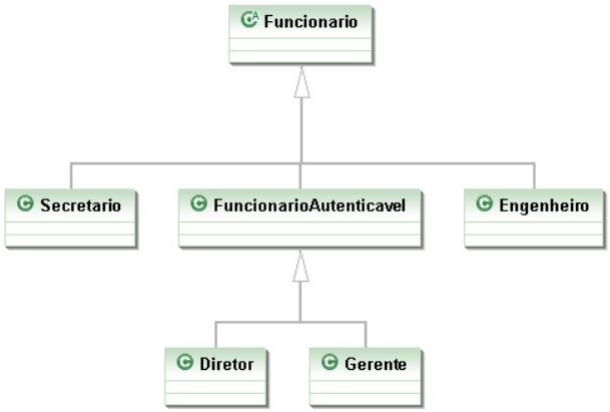
Isso se chama **sobrecarga de método (Overloading)** . Não confundir com o conceito que já vimos chamado *Overriding* que é sobrescrever um método , que é um conceito muito mais poderoso).

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
}
```

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

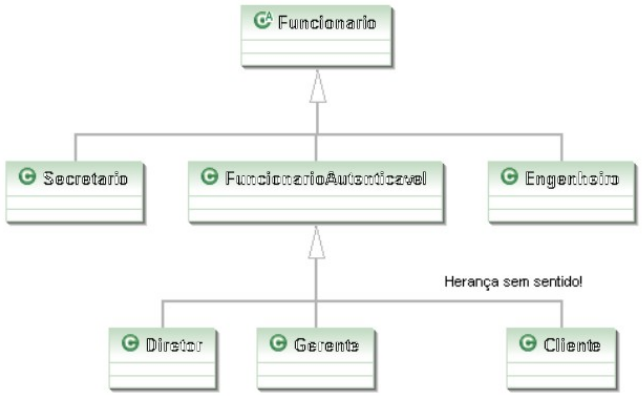
```
class SistemaInterno {

  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);

  }
}
```



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.

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 "é um".




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 dois 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)

**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.**

Podemos criar esse contrato em Java!

```
interface Autenticavel {

  boolean autentica(int senha);
}
```

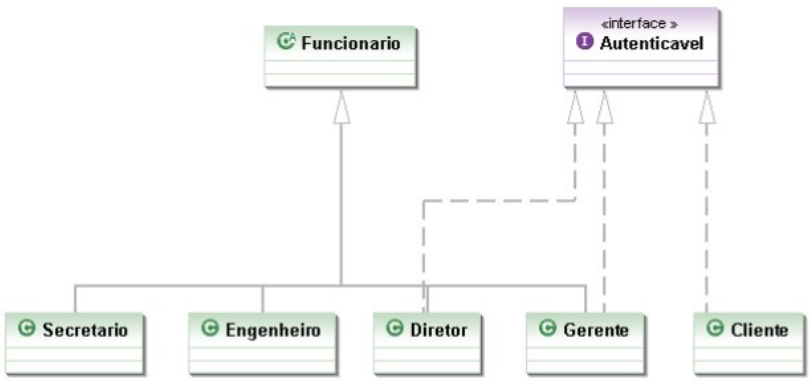
**Chama-se interface pois é a maneira pela qual poderemos conversar com um Autenticavel. Interface é a maneira através da qual conversamos com um objeto.**

**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.**

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:

```
class Gerente extends Funcionario implements Autenticavel {

  private int senha;
  // outros atributos e métodos
  public boolean autentica(int senha) {
    if (this.senha == senha) {
      System.out.println("Acesso Permitido!");
      return true;
    } else {
      System.out.println("Acesso Negado!");
      return false;
    }
    // pode fazer outras possíveis verificações, como saber se esse
    // departamento do gerente tem acesso ao Sistema
    return true;
  }
                                                              }
```



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:

```
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.
  }

}
```

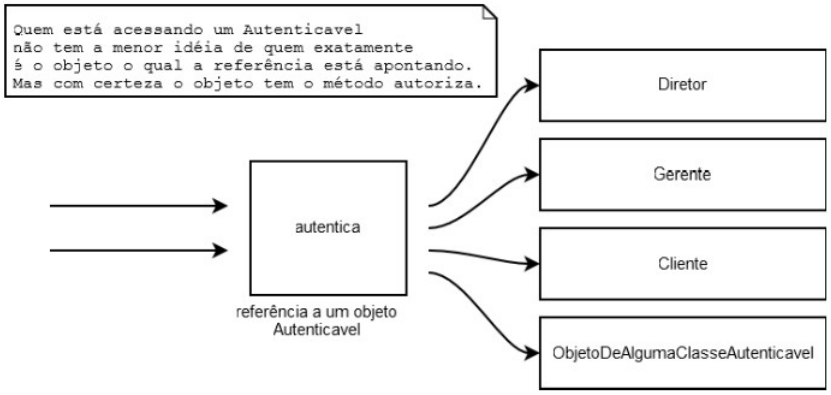
Pronto! E já podemos passar qualquer Autenticavel para o SistemaInterno. Então precisamos fazer com que o Diretor também implemente essa interface.

```
class Diretor extends Funcionario implements Autenticavel {

  // métodos e atributos, além de obrigatoriamente ter o autentica

}
```

E a classe Cliente também teria o método autentica():

```
class Cliente implements Autenticavel {
    private int departamento;

    if (this.departamento == senha) {
```

```
      System.out.println("Acesso Permitido!");
      return true;
    } else {
      System.out.println("Acesso Negado!");
      return false;
    }

  // métodos e atributos, além de obrigatoriamente ter o autentica

}
```




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();
```

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
  }

}
```

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).

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. Esta é uma das ideias principais do bom desenvolvimento de software, que é fazer software manutenível.