

INSTITUTO MAUÁ DE TECNOLOGIA



Linguagens I

Interface

Prof. Tiago Sanches da Silva
Prof. Murilo Zanini de Carvalho

Interface

Antes de qualquer coisa

O que é interface de um objeto?

A interface de um objeto é um conjunto de operações públicas que ele pode realizar.

O objeto da classe Lâmpada, por exemplo, tem como interface as operações:

- acender()
- apagar()

Os objetos se comunicam através desses métodos públicos, qualquer outra operação é considerada inválida.

Interface

A entidade **Interface** então determina métodos obrigatórios para a classe que deseja implementá-la.

Assim como classes abstratas uma interface também define métodos que deverão ser implementados por classes que venham a implementar a interface.

Também como uma classe abstrata, uma interface não pode ser instanciada.

Interface

O nível de acesso de todos os métodos declarados em uma Interface é **public**. Faz sentido?

SIM! Já que Interface (entidade) define parte da interface de um objeto!



Interface

Para criar uma interface basta utilizar a palavra-chave **Interface**, e no corpo definir as assinaturas dos métodos.

Para implementar a **Interface** em um classe, basta utilizar a palavra-chave **implements**.

```
public interface Figura {  
    public abstract double calcularArea( );  
}
```

```
public class Quadrado implements Figura {  
    double lado;  
    public Quadrado(double lado) {  
        this.lado = lado;  
    }  
  
    public double calcularArea( ) {  
        double area = 0;  
        area = lado * lado;  
        return area;  
    }  
}
```

```
public class Circulo implements Figura {  
    double raio;  
  
    public Circulo (double raio) {  
        this.raio = raio;  
    }  
  
    public double calcularArea( ) {  
        double area = 0;  
        area = 3.14 * raio * raio;  
        return area;  
    }  
}
```

Interface

Já que as classes **Quadrado** e **Circulo** implementam a interface **Figura**, logo precisam obrigatoriamente implementar o método **calcularArea()**.

Interface é como um contrato, em que quem assina se responsabiliza por implementar os métodos definidos na Interface (cumprir o contrato).

Uma Interface expõe **o que** o objeto deve fazer, e **não como** ele deve fazer. Como o objeto irá fazer é definido na implementação dessa interface, por parte da classe que a implementará.

Interface

```
public interface Figura {  
    public abstract double calcularArea( );  
}
```

```
public class Quadrado implements Figura {  
    double lado;  
    public Quadrado(double lado) {  
        this.lado = lado;  
    }  
    public double calcularArea( ) {  
        double area = 0;  
        area = lado * lado;  
        return area;  
    }  
}
```

```
public class Circulo implements Figura {  
    double raio;  
    public Circulo (double raio) {  
        this.raio = raio;  
    }  
    public double calcularArea( ) {  
        double area = 0;  
        area = 3.14 * raio * raio;  
        return area;  
    }  
}
```

```
new Quadrado(4);  
new Circulo(2);
```


Interface

```
public class Main {
```

```
    public static void main(String[] args) {
```

```
        Figura f1 = new Quadrado(4);
```

```
        Figura f2 = new Circulo(2);
```

```
        System.out.println("Área da Figura 1 é: "
            + f1.calcularArea() + "\n"
            + "Área da Figura 2 é: "
            + f2.calcularArea());
```

```
    }
```

```
}
```


```
new Quadrado(4);
new Circulo(2);
```

Observe que uma **interface** não pode ser **instanciada**, mas é possível um objeto, declarado como sendo do tipo definido por uma interface, **receber** objetos de classes que **implementam tal interface**.

A variável f1 é uma referência para um objeto que implementa Figura, portanto ele só enxergará os métodos descritos nessa Interface.



Interfaces x Classes Abstratas

- Classes abstratas **podem conter métodos não abstratos**, isto é, que contêm implementação e que podem ser herdados e utilizados por instancias das subclasses.
 - Interfaces não podem conter nenhum método com implementação, **TODOS OS MÉTODOS SÃO IMPLICITAMENTE abstract E public** e não possuem corpo.
 1. Os modificadores public e abstract podem ser omitidos sem problemas.
- 

Interfaces x Classes Abstratas

- Um diferença essencial entre classes abstratas e interfaces em Java é que uma subclasse somente pode herdar de uma única classe (abstrata ou não) enquanto qualquer classe **pode implementar várias interfaces** simultaneamente.
- Interfaces são, portanto, um mecanismo simplificado de implementação de “herança múltipla” em Java, que possibilita que mais de uma interface determine os métodos que uma classe herdeira deve implementar.

Múltiplas Interfaces

Uma classe pode implementar mais de uma interface, assumindo assim vários comportamentos.

```
public interface Impressora {  
    public void imprime(Documento d);  
}
```

```
public interface Fax {  
    public void transmite(Documento d);  
}
```

```
public class FaxImpressora implements Impressora, Fax {  
    public void imprime(Documento d) {  
        ...  
    }  
    public void transmite(Documento d) {  
        ...  
    }  
}
```

Múltiplas Interfaces

```
public interface Radio {  
  
    public void liga();  
  
    public void desliga();  
  
    public void trocaEstacao(  
        int frequencia);  
  
}
```

```
public interface Relogio {  
  
    public String getHoras();  
  
}
```

```
public class RadioRelogio  
    implements Radio, Relogio {  
  
    public void liga() {  
        // Implementação  
    }  
  
    public void desliga() {  
        // Implementação  
    }  
  
    public void trocaEstacao  
        (int frequencia) {  
        // Implementação  
    }  
  
    public String getHoras() {  
        // Implementação  
        return null;  
    }  
  
}
```

A parte difícil

Aprender a sintaxe de como cria uma Interface e como implementa através de uma classe é a parte fácil.

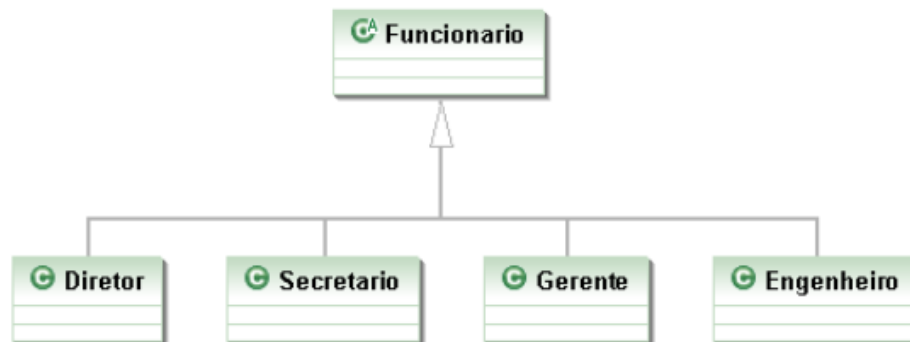
Muitas vezes o difícil é entender o conceito e as vantagens de sua utilização.

Uma interface estabelece uma espécie de **contrato** que é obedecido pelas classes que a implementam.

Sendo assim, quando uma classe implementa uma interface, garante-se que todas as funcionalidades especificadas pela interface serão oferecidas pela classe.

Mais exemplos

Imagine que um Sistema de Controle do Banco pode ser acessado, além de pelos Gerentes, pelos Diretores do Banco.



```
class Diretor extends Funcionario {

    public boolean autentica(int senha) {
        // verifica aqui se a senha confere com a recebida como parametro
    }

}

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
    }

}
```

Mais exemplos

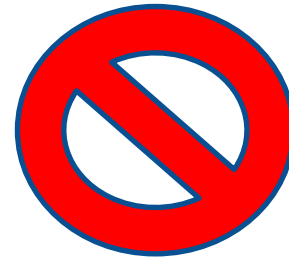
Repare que o método de autenticação de cada tipo de **Funcionario** pode variar muito. Mas vamos aos problemas. Considere o **SistemaInterno** e seu controle: precisamos receber um **Diretor** ou **Gerente** como argumento, verificar se ele se autentica e colocá-lo dentro do sistema.

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


Mais exemplos

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(...);  
    }  
}
```

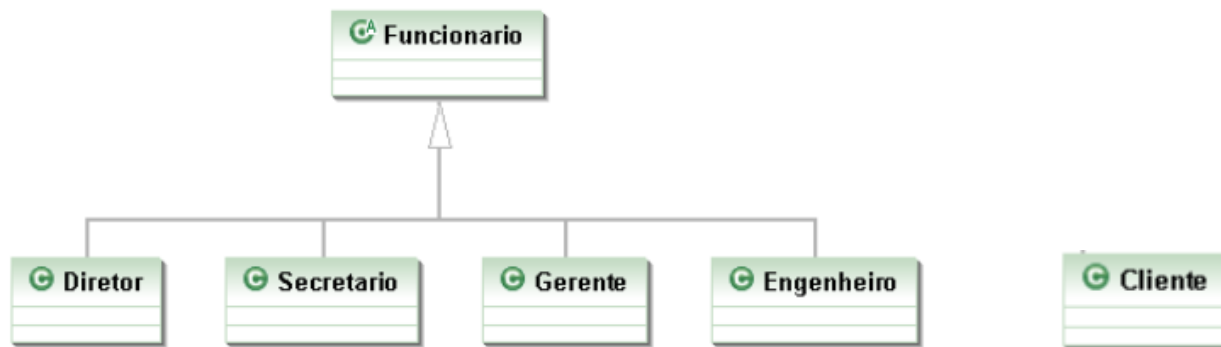


Cada vez que criarmos uma nova classe de **Funcionario** que é autenticável, precisaríamos adicionar um novo método de **login** no **SistemaInterno**.

Mais exemplos

Imagine ainda que existe a classe cliente e que ele também acessará o sistema do banco, afinal ele precisa poder acessar as informações da própria conta.

E com certeza ele não pertence a essa arvore de funcionários.



O que fazer?????

A parte difícil

Vamos criar um contrato para que todos que queriam ser “autenticáveis” assinem. Vamos criar uma Interface!

contrato Autenticavel:

quem quiser ser Autenticavel precisa saber fazer:
1.autenticar dada uma senha, devolvendo um booleano

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

A parte difícil

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

“Quem desejar ser autenticável precisa saber autenticar dado um inteiro e retornando um booleano”.

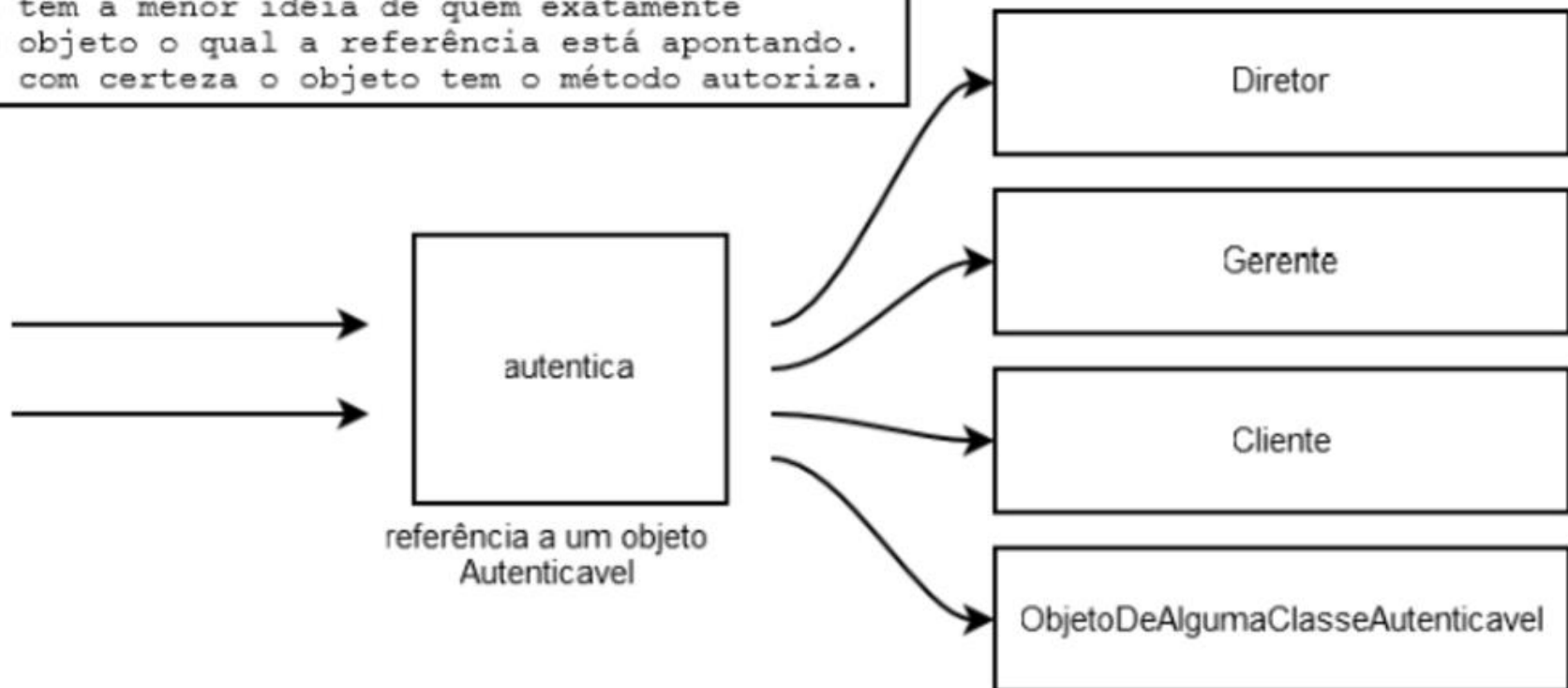
```
class Gerente extends Funcionario implements Autenticavel {  
  
    private int senha;  
  
    // outros atributos e métodos  
  
    public boolean autentica(int senha) {  
        if(this.senha != senha) {  
            return false;  
        }  
        // pode fazer outras possíveis verificações,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

Assim como no Diretor
e no Cliente.

A parte difícil

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

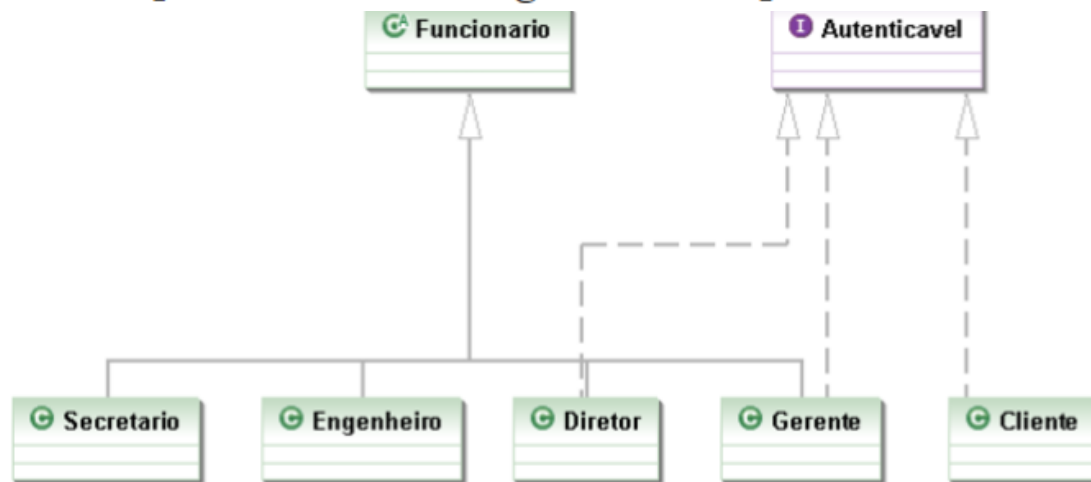
Quem está acessando um Autenticavel não tem a menor idéia de quem exatamente é o objeto o qual a referência está apontando. Mas com certeza o objeto tem o método autoriza.



A parte difícil

```
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!
```

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.

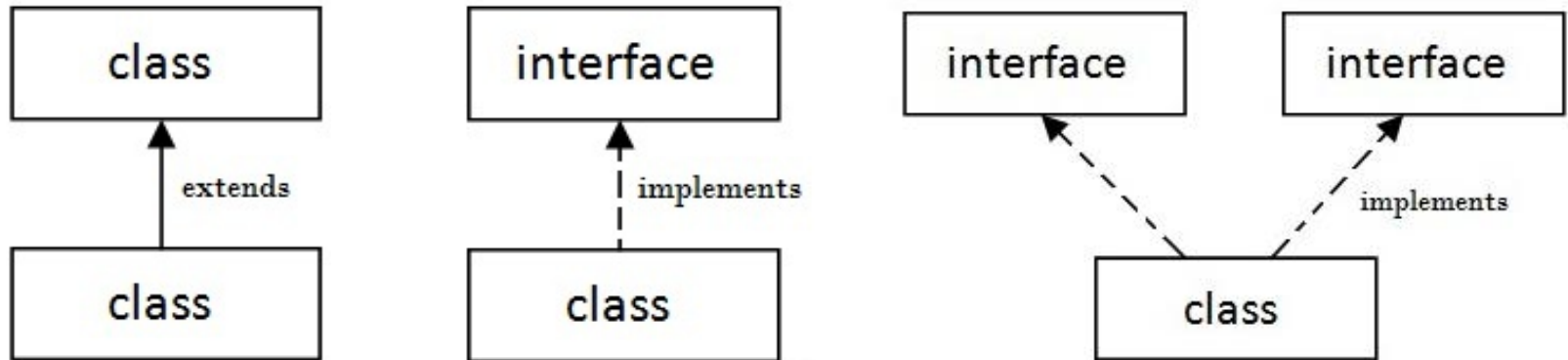


A parte difícil

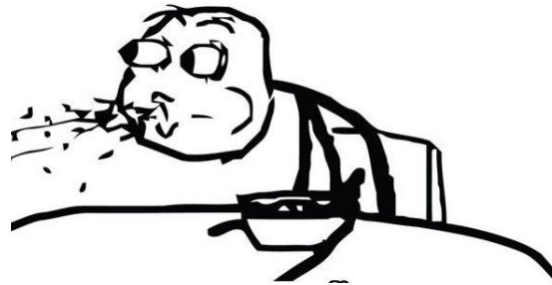
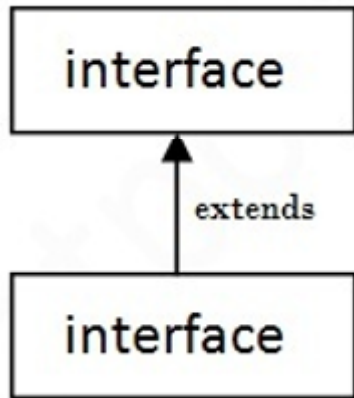
Qualquer `Autenticavel` passado para o `SistemaInterno` está bom para nós. Repare que pouco importa quem o objeto referenciado realmente é, pois ele tem um método `autentica` que é o necessário para nosso `SistemaInterno` funcionar corretamente. Aliás, qualquer outra classe que futuramente implemente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

```
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  
    }  
  
}
```

Relações possíveis entre classes e interfaces



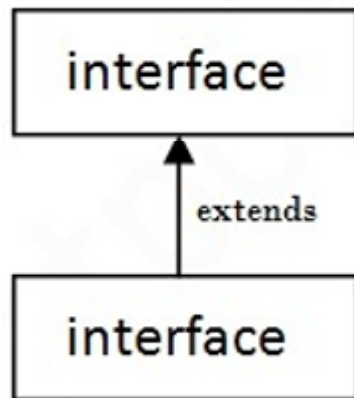
Relações possíveis entre classes e interfaces



É possível que uma interface herde outra interface, mas no que isso implica?

Basicamente, que para uma classe implementar a interface herdeira, deve implementar também a Interface pai.

Relações possíveis entre classes e interfaces



```
interface Printable{
```

```
void print();
```

```
}
```

```
interface Showable extends Printable{
```

```
void show();
```

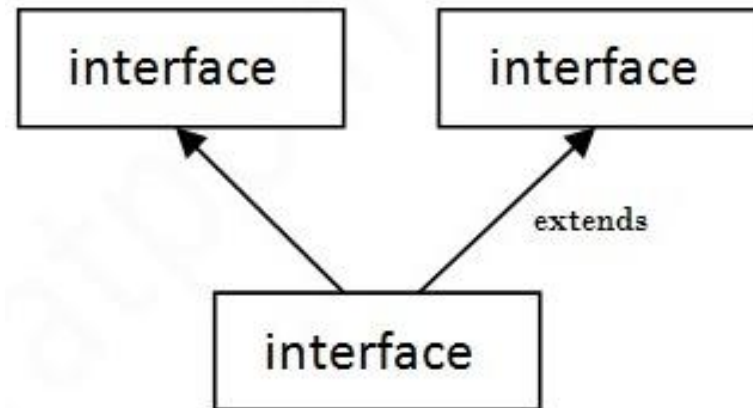
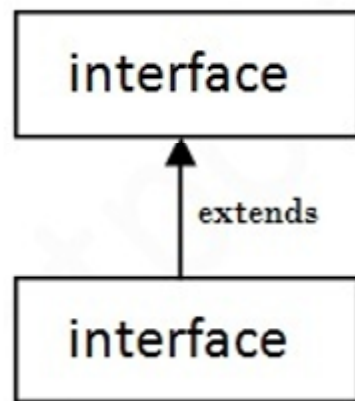
```
}
```

```
class TestInterface4 implements Showable{
```

```
public void print(){System.out.println("Hello");}
```

```
public void show(){System.out.println("Welcome");}
```

Relações possíveis entre classes e interfaces



Método default do Java 8

Desde Java 8, podemos ter corpo de método na interface. Mas precisamos torná-lo método padrão.

```
interface Drawable{  
    void draw();  
    default void msg(){System.out.println("default method");}  
}  
  
class Rectangle implements Drawable{  
    public void draw(){System.out.println("drawing rectangle");}  
}  
  
class TestInterfaceDefault{  
    public static void main(String args[]){  
        Drawable d=new Rectangle();  
        d.draw();  
        d.msg();  
    }  
}
```

Em Resumo (Adaptado de *Head First*):

- Fazer uma **CLASSE** quando ela não estender ninguém (não herdar de ninguém a não ser de *Object*) e quando a classe não passar no teste É-UM para nenhuma outra classe.
- Fazer uma **SUBCLASSE** (herdar de outra classe) quando você precisar fazer uma versão mais específica dela, sobrescrevendo ou adicionando novos métodos.
- Usar **CLASSES ABSTRATAS** quando for definir um modelo para um grupo de subclasses e você possui alguma implementação de código comum que todas as subclasses devem possuir. Torne uma classe abstrata quando desejar garantir que ninguém vai criar instâncias (criar objetos) dela.
- Use uma **INTERFACE** quando você quer definir uma funcionalidade que algumas de suas classes podem assumir, mesmo sem implementar aquele árvore de herança.

Perguntas?

Como podemos melhorar nosso programa com o que aprendemos hoje?