Interfaces

Linguagem de Programação

Objetivos da seção

- Apresentar o conceito de Tipo Abstrato de Dado (“interface”, em Java)
- Mostrar como interfaces permitem o desacoplamento entre interface e implementação
- Mostrar como interfaces criam uma hierarquia de tipos, e permite polimorfismo

Saber quais métodos escrever, com que parâmetros e com que valor de retorno é uma *arte* que os alunos vão adquirir com experiência

Mas existem regras que devem ser aprendidas bem cedo!

Iremos aprender regras de ouro neste módulo!

Definição

Interfaces definem os métodos com seus parâmetros e tipos de retorno, mas sem implementação alguma

- Depois veremos que isso é quase levar o conceito de classes abstratas ao extremo

Quando estamos trabalhando com objetos declarados de uma dada interface sabemos quais são os métodos que podemos chamar, o que eles recebem como parâmetros de entrada e o que eles retornam

- Mas não sabemos como foi implementado
- Não há implementação associada
- É como se fosse um acordo de uso (um protocolo)

Vejamos a implementação da interface que poderia representar um objeto voador em um software de simulação

```
package objetos.voadores;


public interface Voador {

        public double voar();

        public void planar();

        public boolean pousar();

}
```

Note que temos uma palavra reservada nova: **`interface`**

Como não existe implementação, não é possível instanciar objetos da interface (usando *new*), pois a JVM não saberia que código executar (já que cada métodos não tem associação com a implementação dos mesmos)

O que fizemos até agora foi apenas **definir um tipo**

- **Isso só é útil se fizermos mais duas coisas**
  - **i) Fornecer uma ou mais implementações para esta interface**
  - **ii) Usar o tipo definido (e não suas implementações ao definir as nossas referências a objetos)**

Vamos então criar uma classe que implementa esta interface?

```
package objetos.voadores;


public class GalinhaVoadora implements Voador {
	public void comeMilho() {
	System.out.println("Come milho galinha");
	}

	@Override
	public double voar() {
		System.out.println("Voa voa galinha");
		return 0;
	}

	@Override
	public void planar() {
		System.out.println("Plana galinha");
	}

	@Override
	public boolean pousar() {
		System.out.println("Pousa galinha");
		return true;
```

```
    }

}
```

Veja como é simples implementar uma interface (você literalmente diz que quer que a classe implemente a interface)

- Acrescentamos a cláusula **`implements Voador`** que significa que esta classe implementa a interface Voador
- **`implements`** é uma palavra reservada
- **Isso nos obriga a implementar cada método que pertence à interface**
  i) O compilador (através da IDE) ajudará você a garantir isso

Qualquer objeto da classe GalinhaVoadora pode ser tratado como se fosse do tipo Voador

Interfaces são recursos muito poderosos, utilizados em linguagens orientadas a objeto, para "obrigar" um determinado grupo de classes a ter métodos em comum

Quando uma classe implementa os métodos de uma interface, objetos desta classe passam a "se comportar como a interface" e assim, é possível substituir objetos do tipo da interface pelos objetos das classes que as implementam

O Princípio de Substituição de Liskov diz que objetos podem ser substituídos por seus subtipos sem que isso afete a execução correta do programa. Ao implementar a interface, o objeto da classe que implementa a interface passa a estar hierarquicamente abaixo da interface e assim o princípio da substituição de Liskov é válido

Observe também que há métodos implementados pela classe que não fazem parte da interface (quais, por exemplo?)

Agora vamos criar um programa que faz um objeto voador voar

```
package objetos.voadores;
```

```
import java.util.Random;


public class VoadoresMain {
	public static void main(String[] args) {
		Voador qqVoador = new GalinhaVoadora();
		qqVoador.voar();//chamada polimórfica
		qqVoador.planar();//chamada polimórfica
		qqVoador.pousar();//chamada polimórfica
	}
}
```

Manutenção de programas

Vamos fazer “manutenção” no nosso programa

- Manutenção é uma atividade extremamente comum feita por programadores
- Software não é uma coisa estática que não muda depois de feita
- Há *sempre* mudanças a fazer em programas que são utilizados
- Programas que não precisam mudar mais é porque ninguém os está utilizando

Nosso problema de manutenção: o usuário deseja manipular novos tipos de objetos voadores

- Vamos manipular drones

A primeira coisa que fazemos é implementar a classe Drone:

```
package objetos.voadores;
```

```java
public class Drone implements Voador {

	private String tipo;

	public Drone(String tipo) {
		super();
		this.tipo = tipo;
	}
	public String getTipo() {
		return tipo;
	}

	@Override
	public double voar() {
		System.out.println("Voa voa drone");
		return 0;
	}

	@Override
	public void planar() {
		System.out.println("Plana drone");
	}

	@Override
	public boolean pousar() {
```

```
            System.out.println("Pousa drone");
            return true;
        }
}
```

Você vê que os métodos definidos na interface Voador estão todos marcados com @Override? Isto significa que a assinatura destes métodos – tipos de parâmetros de entrada e de retorno – devem ser tais quais estão definidas na interface

Agora vamos mudar o programa que faz a galinha voar para um que faz um drone voar

```
package objetos.voadores;


import java.util.Random;


public class VoadoresMain {
        public static void main(String[] args) {
            Voador qqVoador = new Drone("do tipo bom!");
            qqVoador.voar();//chamada polimórfica
            qqVoador.planar();//chamada polimórfica
            qqVoador.pousar();//chamada polimórfica
        }
}
```

Mudamos completamente o comportamento do programa, trocando apenas a palavra GalinhaVoadora pela palavra Drone!!!! (e incluindo o parâmetro de entrada do construtor do drone)

Por que essa mudança foi tão simples!

- As interfaces definem novos tipos
- As interfaces participam de uma hierarquia de tipos – agora o drone e também a galinha se comportam como voadores e podem ser usados sempre que um Voador for declarado
- Definimos boas interfaces (definimos bem o vocabulário do problema)
- Implementamos a interface
- Usamos o tipo (interface) ao declarar objetos e não a classe que implementa a interface

→ **Escrevemos chamadas polimórficas**

A variável qqVoador foi definida como sendo do tipo Voador, que é uma interface

Ela pode referenciar qualquer objeto de uma classe que implemente a interface Voador, seja Drone, GalinhaVoadora, Helicoptero, AviaoDePapel ou qualquer outra classe que em sua definição tenha a declaração de que a classe implementa a interface Voador

As chamadas a voar, planar e pousar são ditas polimórficas porque o que estas chamadas fazem muda de acordo com a implementação referenciada pela variável qqVoador

Se recebe um drone voar() é o voar do drone, mas se recebe uma galinha, voar() é o voar da galinha – basta mudar a implementação usada para mudar o comportamento do método

Agora vamos mudar o programa para deixar ainda mais claro isso. Agora queremos que ele sorteie um objeto voador para voar.

```
package objetos.voadores;


import java.util.Random;


public class VoadoresMain {
```

```
    public static void main(String[] args) {
        Voador qqVoador = recebeVoador();
        qqVoador.voar();
        qqVoador.planar();
        qqVoador.pousar();
    }

    public static Voador recebeVoador() {
        Random r = new Random();
        if(r.nextBoolean())
            return new GalinhaVoadora();
        else
            return new Drone("do tipo bom!");
    }
}
```

Qual é a implementação dos métodos voar(), planar() e pousar() que serão executadas neste programa acima?

Difícil de saber, não é? Mais uma vez: são chamadas polimórficas e vai depender de que tipo a referência qqVoador está referenciando

Só saberemos isso em tempo de execução

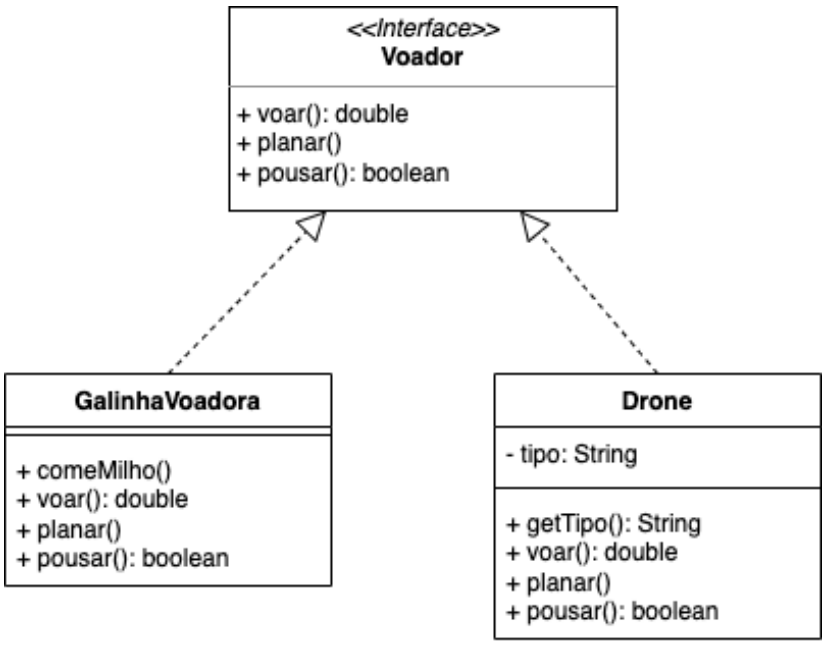
Diagrama de classes UML

As interfaces participam de uma hierarquia de tipos – agora o drone e também a galinha se comportam como voadores e podem ser usados sempre que um Voador for declarado

Podemos usar UML (ver figura abaixo) para representar este relacionamento hierárquico entre a interface Voador e as classes que a implementam (GalinhaVoadora e Drone)

Usamos um <<estereótipo>> de UML para indicar que temos uma interface – está é a forma mais completa de apresentar a interface porque requer que a gente indique que métodos a interface define

Usamos uma ligação de implementação que começa com um triângulo na interface a ser implementada e usa linha tracejada até chegar na classe de implementação



Ao ver este diagrama entendemos que Voador é uma interface que existem no momento duas classes que a implementam: GalinhaVoadora e Drone. É possível ver métodos nestas classes que não estão na interface e assim fazem parte apenas da classe e não da interface

Um repositório de voadores

Agora queremos criar uma abstração que é um repositório de objetos voadores

Deve ser possível adicionar e remover voadores neste repositório

Deve ser possível botar quaisquer objetos voadores para voar, pousar e planar

Vamos ao código?

- Como esta classe deve ser chamada?
- Que atributos ela deve ter?
- Que métodos queremos escrever?

```
package objetos.voadores;


import java.util.ArrayList;

import java.util.List;


public class RepositorioDeVoadores {

	private List<Voador> voadores = new ArrayList<>();


	public boolean adicionaVoador(Voador voador) {

		return voadores.add(voador);

	}


	public Voador removeVoador(int posicao) {

		return voadores.remove(posicao);

	}


	public void fazVoar(int posicao) {

		voadores.get(posicao).voar();

	}


	public void fazPousar(int posicao) {

		voadores.get(posicao).pousar();
```

```
        }

        public void fazPlanar(int posicao) {
                voadores.get(posicao).planar();
        }

        public void fazTodosVoarem() {
                for (Voador voador : voadores) {
                        voador.voar();
                }
        }

        public void fazTodosPousarem() {
                for (Voador voador : voadores) {
                        voador.pousar();
                }
        }

        public void fazTodosPlanarem() {
                for (Voador voador : voadores) {
                        voador.planar();
                }
        }
}
```

Note que em nenhum momento acoplamos o repositório à implementação de qualquer tipo específico de voador

- O repositório é capaz de gerenciar qualquer voador: os que já existem… e todos os que estão por vir!
- Isso é possível porque fizemos uso de polimorfismo, você é capaz de identificar no código do repositório onde estão as chamadas polimórficas?
- Toda vez que chamados os métodos da interface Voador, ali temos chamadas polimórficas
    - Não sabemos de que voador específico virá a implementação

Novos voadores podem passar a existir... e autometicamente nosso repositório sem mudança alguma está pronto para lidar com ele. Quer ver?

Vamos agora escrever um novo voador que é uma Libélula

```
package objetos.voadores;


public class Libelula implements Voador {

	private String cor;


	public Libelula(String cor) {

		super();

		this.cor = cor;

	}


	public String getCor() {

		return cor;

	}


	@Override

	public double voar() {

		System.out.println("Voa voa libelula " + getCor());

		return 0;

	}
```

```
        @Override
        public void planar() {
                System.out.println("Plana libelula " + getCor());
        }

        @Override
        public boolean pousar() {
                System.out.println("Pousa libelula " + getCor());
                return true;
        }
}
```

O que precisa mudar na classe RepositorioDeVoadores para que a nova voadora do tipo Libelula também possa ser armazenada no repositório?

- Absolutamente NADA!

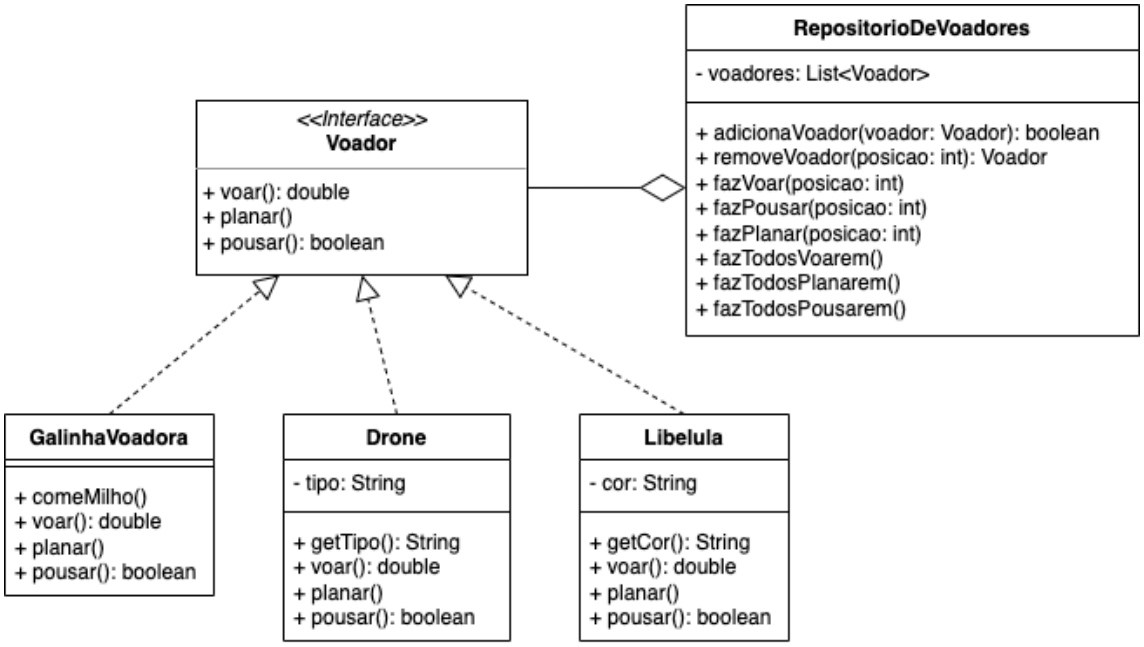
Poderíamos pensar em outra hierarquia de classes, faladores por exemplo?

Representação em UML

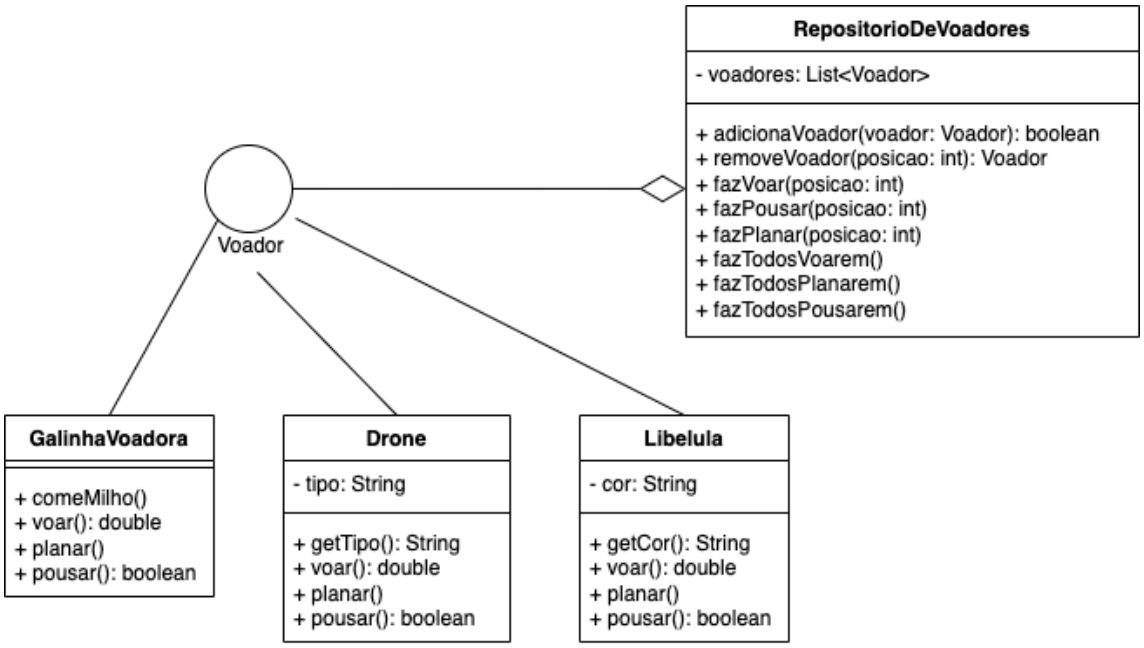
Em UML, interfaces são representadas de duas formas (já vimos uma delas)

- Com definição completa, como “classe” usando o estereótipo <<interface>>
- De forma resumida, usando uma notação gráfica específica (bola)

Exemplo: nosso último programa, agora com o repositório



Exemplo: nosso último programa mas usando a forma abreviada de interface



Interfaces populares do Java

List<E> - uma interface que define todos os métodos que objetos que se comportam como coleções do tipo lista devem ter – basicamente, são coleções cujos elementos podem ser acessados pela sua posição

- É uma boa prática de programação declarar a coleção como List<E>, e instanciar o objeto a partir de uma das implementações de List, sendo uma das mais populares o ArrayList<E> que representa uma lista do tipo array
- Isso é interessante porque em uma manutenção, se for decidido que outra implementação de List é mais eficiente para o software em desenvolvimento ou em manutenção, a mudança para a outra implementação é rápida e não vai causar bugs

Comparable<E> – quando uma classe implementa esta interface, então objetos desta classe passam a ser "Comparáveis" e podem ser ordenados pelo método static Collections.sort() da classe Collections

- Apenas um método precisa ser implementado por esta interface, que é o método compareTo(E e). Sendo x e y da classe E (a classe que está implementando a interface), então os seguintes resultados são possíveis
    - Se x == y → x.compareTo(y) deve retornar 0
    - Se x < y → x.compareTo(y) deve retornar um valor menor que 0, tipicamente retorna-se -1
    - Se x > y → x.compareTo(y) deve retornar um valor maior que 0, tipicamente retorna-se 1

Qual é a diferença entre uma classe abstrata e uma interface?

- Uma classe abstrata pode ter implementação de alguns métodos e a interface não pode
- Uma classe abstrata também define uma interface (seus métodos públicos)
- Para reusar o contrato de uma classe abstrata temos que estendê-la e implementar os métodos abstratos e/ou sobrescrever os métodos que devem apresentar comportamento diferente do herdado
  - Isto causa uma relação “é um” entre a classe abstrata e a classe derivada dela
  - Muitas vezes esta relação é muito forte para ser usada!
- Para reusar o contrato de uma interface temos que implementar os métodos que ela define
  - Isto causa um relacionamento “é como um” entre a interface e a classe que a implementa
  - A nova classe age como um …
    i) Raquel age como uma mãe…
    ii) Raquel age como uma filha…
    iii) Raquel age como uma professora…
    iv) Raquel age como uma aluna… (na aula de francês!)
    v) Raquel pode acumular vários papéis ao mesmo tempo
- Peculiaridade de Java:
  - Em se tratando da hierarquia de classes apenas herança simples é permitida
  - Em se tratando da hierarquia de tipos abstratos (interfaces) herança múltipla é permitida
    i) Uma interface pode estender várias interfaces simultaneamente ou

ii) Uma interface pode herdar de várias interfaces simultaneamente
iii) Isto é simples porque não há implementação. Assim, se dois métodos tiverem a mesma assinatura, apenas uma implementação será oferecida

- Como sugerido no exemplo dos papéis acumulados por Raquel, uma classe pode implementar várias interfaces simultaneamente
- Vejamos o código disso!

```
package p2.exemplos;

public interface MaeIF {

    public void manda();

}
```

```
package p2.exemplos;

public interface FilhaIF {

    public void obedece();

}
```

```
package p2.exemplos;

public interface ProfessoraIF {

    public void daAula();
    public void preparaAula();

}
```

```
package p2.exemplos;

public interface AlunaIF {
```

```
    public void estudaParaProva();
    public void participaDeAula();
    public void fazProva();

}
```

```
package p2.exemplos;

public class MulherOcupada extends Pessoa
implements MaeIF, FilhaIF, ProfessoraIF,
        AlunaIF {

    public MulherOcupada(String nome) {
        super(nome);
    }

    public void manda() {
        System.out.println("Mandando como
mae!");
    }

    public void obedece() {
        System.out.println("Obedecendo como
filha!");
    }

    public void daAula() {
        System.out.println("Professora dando
aula.");
    }

    public void preparaAula() {
        System.out.println("Professora
preparando aula.");
    }

    public void estudaParaProva() {
        System.out.println("Aluna estuda para a
prova!");
    }
```

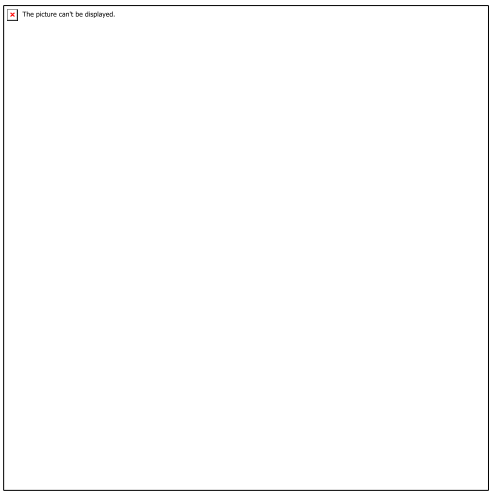
```
    public void participaDeAula() {
        System.out.println("Aluna participa de
aula.");
    }

    public void fazProva() {
        System.out.println("Aluna fazendo
prova.");
    }

}
```

```
package p2.exemplos;

public class DiaDaMulherOcupada {

    private static void manda(MaeIF mae) {
        mae.manda();
    }

    private static void obedece(FilhaIF filha) {
        filha.obedece();
    }

    private static void preparaAula(ProfessoraIF
professora) {
        professora.preparaAula();
    }

    private static void estudaParaProva(AlunaIF
aluna) {
        aluna.estudaParaProva();
    }

    public static void main(String[] args) {
        MulherOcupada raquel = new
MulherOcupada("Raquel");
        manda(raquel); // trata raquel como uma
MaeIF
        obedece(raquel); // trata raquel como
uma FilhaIF
        preparaAula(raquel); // trata raquel
como uma ProfessoraIF
```

```
        estudaParaProva(raquel); // trata raquel
como uma AlunaIF
    }

}
```

- Em situações como essa, uma flexibilidade que as interfaces nos oferecem é a de poder fazer o *upcast* para mais de um tipo base
  - No exemplo acima fizemos o *upcast* do mesmo objeto para quatro diferentes tipos base!
- Como exemplo, podemos ter uma interface que estende várias outras
  - No código acima, poderíamos ter criado a interface da mulher ocupada que estende MaeIF, FilhaIF, ProfessoraIF e AlunaIF
  - Depois implementaríamos apenas esta interface. Também teria dado certo. Tente isto em casa!

Princípio importante de projeto orientado a objetos



Program to an interface, not to an implementation!

Erich Gamma

Grande parte deste material que segue foi retirado da entrevista concedida por Erich Gamma a Tim Bernners (aqui).

- O que você acha que isto significa? Vamos ver?

- Quando reusamos estabelecemos dependências
- Usar interfaces faz você depender de interfaces e não de implementações, o que é uma dependência saudável
- Mas o que significa usar uma interface?
  - Significa declarar a variável como sendo do tipo da interface, mas ao fazer o new, obviamente você irá usar uma implementação da interface
- Uma interface define a colaboração entre objetos
  - Uma interface não tem detalhes de execução/implementação
  - Uma interface define o "vocabulário" da colaboração
  - Uma vez que você entende as interfaces, você entende mais do sistema porque você compreende o vocabulário do problema
  - Vocabulário = abstrações + métodos (mensagens)
- Toda classe faz parte de uma hierarquia de tipos
  - A idéia é que quanto mais alto estivermos na hierarquia, mais abstrato é o tipo (até chegarmos nas interfaces!)
  - Quanto mais descemos na hierarquia, mais concretas vão ficando as classes, associadas a implementações
  - Esta regra diz que você deve declarar seus objetos (suas referências, na realidade!) como sendo do nível mais alto possível da hierarquia. Vá até o nível mais alto onde você possa chegar!!!
  - Assim você estará desacoplado de implementações específicas. Novas implementações deste mesmo tipo abstrato podem ser facilmente incorporadas em seu programa
- Apesar de tudo que foi dito, cuidado para não sair criando interfaces sem necessidade!
  - Refactoring!

- Design patterns interessantes de serem estudados neste momento: Strategy e Factory Method!