

# Lista 2 - Conceitos Básicos de Orientação a Objetos

Resolução

**Vinicius Takeo Friedrich Kuwaki**

Universidade do Estado de Santa Catarina

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

# Introdução

- A Lista 2, aborda os seguintes tópicos:
  - Criação de classes e objetos;
  - Encapsulamento;
  - Estrutura orientada a objetos;
- Todos os códigos fontes estarão disponíveis a partir do dia 29/07/2020 nesse [link](#).
- Os alunos que quiserem corrigir as questões da sua lista, tem até dia 28/07/2020 as 23h59 para reenviar no Moodle.
- Caso deseje, avance para a resolução:
  - Exercício 1 (slide 7);
  - Exercício 2 (slide 22);
  - Exercício 3 (slide 43);
  - Exercício 4 (slide 71);
  - Exercício 5 (slide 99);
  - Exercício 6 (slide 122);

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

## Exercício 1

- Defina as classes abaixo contendo ao menos três atributos que os representam adequadamente:
  - Pessoa;
  - Comida;
  - Animal;
  - Cidade;
  - Filme;

# Seções

Introdução

Exercício 1

**Exercício 1 - Resolução**

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

## Exercício 1 - Resolução

- O objetivo desse exercício é representar esses objetos computacionalmente;
- Isto é, definir alguns atributos para essas classes e formas de acessá-los (métodos getters e setters);
- Em Java temos quatro modificadores de acesso:
  - private: apenas a própria classe pode acessar;
  - public: todos podem acessar;
  - protected: apenas a própria classe e filhas podem acessar (veremos isso mais a frente na disciplina);
  - default: apenas membro do mesmo package podem acessar;
- Esses modificadores de acessos valem para métodos e atributos das classes;
- Em Java, por padrão, todos os atributos de uma classe devem ser **privados**!

## Exercício 1 - Resolução

- Mas se os atributos são privados, qual a utilidade deles se não podem ser acessados por ninguém exceto a própria classe?
- É aí que entram os métodos getters e setters;
- Eles atuam como a “ponte” entre os atributos privados e o mundo de fora da classe;
- Por padrão:
  - getters: retornam o valor de um atributo;
    - são do mesmo tipo do atributo;
    - Não recebem parâmetros;
  - setters: modificam o valor do atributo;
    - são do tipo void (raras exceções retornam valores);
    - Recebem o valor do atributo que será modificado, e por consequência esse valor é do mesmo tipo do atributo;



## Exercício 1 - Resolução

- E como construir esses objetos para usarmos os getters e setters?
- Ai entra os construtores;
- Cada classe pode ter um ou vários construtores;
- A única regra que vale é a de que todos tem o nome da classe;
- Não podem existir construtores que recebem os mesmo tipos;
- Por exemplo:
  - `public Casa(String cor);`
  - `public Casa(String nomeRua);`
- Para o compilador cor e nomeRua é a mesma coisa;
- Agora, se forem tipos diferentes é possível:
  - `public Casa(String cor);`
  - `public Casa(int cepRua);`
- Construtores podem ter modificadores de acesso também: private, public, etc...

## Exercício 1 - Resolução

- Vamos aplicar esses conceitos para classe Pessoa:
- Primeiramente, vamos definir três atributos que pessoas podem possuir:
  - Um nome;
  - Uma idade;
  - E um CPF;
- Lembre-se que os atributos devem ser privados!

```
public class Pessoa {  
  
    private String nome;  
    private int idade;  
    private int cpf;  
  
}
```

## Exercício 1 - Resolução

- Vamos criar um construtor vazio, que não faz nada e não recebe nada;
- Declará-lo e não declará-lo dá no mesmo:
- Como nossos atributos são privados, quem tentar definir um nome, idade ou cpf a uma pessoa não irá conseguir;
- Para isso vamos criar métodos getters e setters;

```
public class Pessoa {  
  
    private String nome;  
    private int idade;  
    private int cpf;  
  
    public Pessoa() {  
    }  
  
}
```

## Exercício 1 - Resolução

- Vamos começar pelo setter do atributo nome;
- Esse método recebe uma String (mesmo tipo do atributo nome);
- E joga no atributo nome da classe;
- Note o uso da palavra **this**;
- Ela se refere ao objeto (instância da classe);
- Quando fazemos **this.nome**, estamos acessando o atributo nome da classe Pessoa;

```
public class Pessoa {  
    ...  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

## Exercício 1 - Resolução

- Agora que podemos alterar o valor do atributo, vamos criar um método para ver qual o valor está lá;
- O getter realiza isso, ele retorna o valor do atributo;
- Por isso seu retorno deve ser do mesmo tipo do atributo;
- Como o nome é uma String, o retorno também deve ser:

```
public class Pessoa {  
    ...  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
    public String getNome() {  
        return this.nome;  
    }  
}
```

## Exercício 1 - Resolução

- Agora podemos repetir o mesmo processo para os demais atributos:
- É possível gerar getters e setters pela própria IDE;
- Assim, o processo não se torna tão repetitivo;

```
public class Pessoa {  
  
    ...  
  
    public int getIdade() {  
        return this.idade;  
    }  
  
    public void setIdade(int idade) {  
        this.idade = idade;  
    }  
  
    public int getCpf() {  
        return this.cpf;  
    }  
  
    public void setCpf(int cpf) {  
        this.cpf = cpf;  
    }  
  
}
```

## Exercício 1 - Resolução

- Mais tarde veremos como instanciar objetos da classe Pessoa;
- Mas adiantando, da forma como criamos a classe Pessoa, para definir seus atributos vamos ter que chamar todos os setters;
- Podemos eliminar esse trabalho todo e no próprio construtor setar os valores dos atributos;
- Vamos adicionar outro construtor a classe Pessoa;

```
public class Pessoa {  
    ...  
    public Pessoa(String nome, int idade, int  
    cpf) {  
        this.nome = nome;  
        this.idade = idade;  
        this.cpf = cpf;  
    }  
    ...  
}
```

## Exercício 1 - Resolução

- Por convenção, em Java os construtores ficam localizados abaixo dos atributos;
- Que devem ser os primeiros a serem declarados dentro de uma classe;
- Logo abaixo dos construtores, os demais métodos

```
public class Pessoa {  
    ... // Atributos acima  
  
    public Pessoa() {  
    }  
  
    public Pessoa(String nome, int idade, int  
    cpf) {  
        this.nome = nome;  
        this.idade = idade;  
        this.cpf = cpf;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    ... // Metodos abaixo  
}
```



## Exercício 1 - Resolução

- Terminada a modelagem da classe Pessoa, as outras seguem a mesma lógica;
- Primeiro definimos os atributos e para cada atributo criamos um get e um set;
- Algumas considerações a respeito:
  - Nada impede que um get ou set, ou até mesmo um construtor seja `private`;
  - Em aulas futuras veremos casos em que teremos que fazer isso.
- Como todas as classes repetem o mesmo processo, e o intuito desse slide é comentar a respeito da resolução, não vamos fazer todas as demais classes aqui;
- Os códigos-fontes estarão disponíveis no dia 29/07/2020, no [link](#).
- Para as demais classes, utilizei os seguintes atributos:

## Exercício 1 - Resolução

- Animal:
  - String cor;
  - String especie;
  - int idade;
- Cidade:
  - String nome;
  - String estado;
  - String pais;
- Comida:
  - String nome;
  - float calorias;
  - float peso;
- Filme:
  - String titulo;
  - int anoLancamento;
  - String genero;

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

**Exercício 2**

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

## Exercício 2

- Para cada uma das classes descritas no exercício anterior, instancie pelo menos três objetos distintos de cada uma delas e exiba no console o retorno do método `toString()`. Implemente o método `toString()` para cada uma das classes.

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

## Exercício 2 - Resolução

- Os métodos **toString()** são maneiras de converter toda a informação contida em um objeto em uma string;
- É a maneira de manter o encapsulamento da classe;
- Nem sempre é possível "imprimir no console" as informações;
- Na verdade é uma pessima prática de programação manter "prints" dentro de métodos das classes;
- Para isso definimos os métodos **toString()**;
- Por padrão toda classe já possui um método **toString()**, mas ela apenas retorna o endereço do objeto;
- Através de herança é possível sobreescrevê-la (veremos isso mais a frente na disciplina);
- Por enquanto basta saber que ele segue a seguinte assinatura  
`public String toString();`

## Exercício 2 - Resolução

- Trabalhar com Strings pode ser uma tarefa bem pesada para o processador;
- Já que em Java, cada String é única, e fica salva em uma área específica do programa;
- Por isso, ficar concatenando pode ser custoso caso hajam muitas coisas a serem concatenadas, pois cada vez que um pedaço é adicionado ao fim da String, o programa cria um nova e mantém ambas;
- Não vamos nos preocupar muito com isso no momento;
- Mas é uma boa prática usar a classe **StringBuilder** para "montar"Strings em Java.
- Vamos começar com a classe Pessoa, vamos adicionar o método **toString()**:

## Exercício 2 - Resolução

- É convenção também que o método **toString()** seja o último declarado na classe.
- Observe ao lado que utilizamos o operador **+** para "somar" as Strings;
- Fazemos isso para todos os atributos da classe Pessoa;

```
public class Pessoa {  
  
    ...  
  
    public String toString() {  
        return "Nome: " + nome + " - Idade: " +  
            idade + " - CPF: " + cpf;  
    }  
}
```



## Exercício 2 - Resolução

- Agora que vimos como criar uma classe, vamos instanciar objetos dela;
- Para isso, vamos criar um método **main()** em uma classe diferente;
- Sempre crie o método `main()` dentro de uma classe própria para ele.

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

## Exercício 2 - Resolução

- Vamos instanciar um objeto da classe Pessoa e utilizar seus setters para atribuir valores a eles;
- Para criar um objeto, utilizamos o operador **new()** que invoca o construtor da classe;
- Os objetos são variáveis, assim como inteiros, floats, etc...

```
public class Main {  
    public static void main(String [] args) {  
        Pessoa p1 = new Pessoa();  
    }  
}
```

## Exercício 2 - Resolução

- Para acessar os métodos, utilizamos a sintaxe do "."(ponto);
- Semelhante a acessar campos de structs em C;
- Vamos utilizar os métodos setters da classe Pessoa:
- Veja quanto código utilizamos para modificar apenas três atributos;

```
public class Main {  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa();  
        p1.setNome("Vinicius");  
        p1.setIdade(19);  
        p1.setCpf(111);  
    }  
}
```

## Exercício 2 - Resolução

- Uma forma de reduzir isso é utilizar o segundo construtor que definimos, que já atribui os valores aos atributos da classe;

```
public class Main {  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa();  
        p1.setNome("Vinicius");  
        p1.setIdade(19);  
        p1.setCpf(111);  
  
        Pessoa p2 = new Pessoa("Fernanda", 15, 2222);  
        Pessoa p3 = new Pessoa("Luciano", 21, 3333);  
    }  
}
```

## Exercício 2 - Resolução

- Agora vamos ver o porquê de termos criado o método **toString()** na classe Pessoa;
- Quando chamamos o método **println()** da classe System.out, ele automaticamente invoca o **toString()** do objeto, exibindo no console o resultado;

## Exercício 2 - Resolução

- Basta chamar esse método dentro da **main()**;

```
public class Main {  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa();  
        p1.setNome("Vinicius");  
        p1.setIdade(19);  
        p1.setCpf(111);  
  
        Pessoa p2 = new Pessoa("Fernanda", 15, 2222);  
        Pessoa p3 = new Pessoa("Luciano", 21, 3333);  
  
        System.out.println(p1);  
        System.out.println(p2);  
        System.out.println(p3);  
    }  
}
```

## Exercício 2 - Resolução

- Ao executar, o resultado obtido é:

```
Nome: Vinicius — Idade: 19 — CPF: 111  
Nome: Fernanda — Idade: 15 — CPF: 2222  
Nome: Luciano — Idade: 21 — CPF: 3333
```

## Exercício 2 - Resolução

- Mencionamos anteriormente e agora vamos ver como usar a classe `StringBuilder` para concatenar Strings;
- Para isso utilizamos o método **`append()`**;
- A classe `StringBuilder`, vai aos poucos concatenando a String;
- (O mesmo que "somar" duas Strings, só que de forma otimizada);
- Vamos definir um **`toString()`** para a classe `Filme`, utilizando essa abordagem:



## Exercício 2 - Resolução

- A classe Filme, possuía três atributos: titulo, ano de lançamento e gênero;
- Para cada "pedaço" da String, vamos usar o método **append()**;
- E ao final, chamamos o método **toString()** da classe **StringBuilder**, que vai nos retornar a String montada;

```
public String toString() {  
    StringBuilder filme = new StringBuilder();  
  
    filme.append(titulo);  
    filme.append(" (");  
    filme.append(anoLancamento);  
    filme.append(") — ");  
    filme.append(genero);  
  
    return filme.toString();  
}
```

## Exercício 2 - Resolução

- Vamos instanciar alguns objetos da classe Filme no método **main()** dos slides anteriores:

```
public class Main {  
    public static void main(String[] args) {  
        ...  
        Filme f1 = new Filme("10 coisas que eu odeio em voce", 1999, "Romance");  
        Filme f2 = new Filme("Interestelar", 2014, "Ficcao Cientifica");  
        Filme f3 = new Filme("Rei Leao", 1994, "Animacao");  
    }  
}
```

## Exercício 2 - Resolução

- Um dos pilares da orientação a objetos é a reutilização de código;
- E o método **toString()** é um dos exemplos disso;
- Veja a seguir como faríamos para exibir as informações do filme sem utilizar o método **toString()**:

```
public class Main {  
    public static void main(String[] args) {  
        ...  
        System.out.println("Titulo: " + f1.getTitulo());  
        System.out.println("Ano de Lancamento: " + f1.getAnoLancamento());  
        System.out.println("Genero: " + f1.getGenero());  
    }  
}
```

## Exercício 2 - Resolução

- Não é nem um pouco prático ficar utilizando todos os getters de um objeto quando estamos criando uma interface console;
- No nosso caso, a classe possui só três atributos, mas e se tivesse uma lógica maior, encapsulada dentro da classe Filme?
- Por exemplo um array de atores, um array de cenários, cada qual com uma localização, uma descrição, etc...
- Se torna inviável deixar que o método **main()** conheça essa lógica;

## Exercício 2 - Resolução

- Por isso utilizamos métodos **toString()** nas classes:

```
public class Main {  
    public static void main(String[] args) {  
        ...  
        System.out.println("Titulo: " + f2.getTitulo());  
        System.out.println("Ano de Lançamento: " + f2.getAnoLancamento());  
        System.out.println("Genero: " + f2.getGenero());  
  
        System.out.println(f3);  
    }  
}
```

## Exercício 2 - Resolução

- Veja a saída no console:

```
Titulo: 10 coisas que eu odeio em voce  
Ano de Lancamento: 1999  
Genero: Romance
```

```
Titulo: Interestelar  
Ano de Lan amento: 2014  
Genero: Ficcao Cientifica
```

```
Rei Leao (1994) – Animacao
```

## Exercício 2 - Resolução

- Assim como feito no exercício anterior, os principais conceitos foram apresentados com exemplos;
- Os demais métodos **toString()** seguem a mesma lógica desses apresentados aqui, só que para os respectivos atributos de suas classes;
- Os códigos-fontes completos estarão no [link](#) no dia 29/07/2020.

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

**Exercício 3**

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução



## Exercício 3

- Crie uma classe que modele um carro, que utilize outras 5 classes que modelem as principais partes dele. Por exemplo, um carro possui rodas, tanque de combustível, assentos, vidros, etc. Esses objetos devem possuir pelo menos 3 atributos que os definem (dimensões, quantidades, etc).

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

**Exercício 3 - Resolução**

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

## Exercício 3 - Resolução

- O objetivo desse exercício é modelar uma classe composta de objetos de outras classes;
- Muitos alunos entenderam que o objetivo era modelar a criação de objetos, quando na verdade era menos complexo que isso, era apenas definir como a classe Carro é: seus atributos e métodos.
- Nem era necessário criar um método **main()** e instanciar objetos;

## Exercício 3 - Resolução

- Para a minha solução eu criei as classes:
  - Assento, que possui largura, comprimento e altura;
  - Roda, que possui aro (tamanho), marca e calota (um booleano para dizer se possui ou não);
  - Vidro, que possui largura e comprimento, além de um atributo booleano para identificar se o vidro tem película ou não.
  - Porta, que possui largura, comprimento e uma instância de um vidro que representa a janela da porta;
  - E por fim um enumerado que representa o tipo de combustível do carro;
- Vamos começar pelo tipo de combustível;

## Exercício 3 - Resolução

- Enumerados são classes especiais em Java que servem para enumerar coisas;
- Isto é, definir um conjunto finito de coisas;
- No meu exemplo eu utilizei o combustível para isso;
- Um carro, na minha solução pode ser movido a:
  - Etanol;
  - Gasolina;
  - Diesel;
  - Gás;
- E para definir um enumerado em Java é muito simples, basta utilizar a palavra reservada **enum** no lugar de **class** e definir entre virgulas os possíveis valores;

## Exercício 3 - Resolução

- É convensão em Java declarar os itens de um enumerados todos em letra maiúscula;
- É possível criar métodos e atributos dentro de um enumerado, mas não veremos isso agora;

```
public enum TipoCombustivel {  
    ETANOL, GASOLINA, DIESEL, GAS;  
}
```

## Exercício 3 - Resolução

- Para a classe Assento:

```
public class Assento {  
  
    private float largura;  
    private float comprimento;  
    private float altura;  
  
    ... // Construtor, getters e setters  
  
    public String toString() {  
  
        StringBuilder assento = new StringBuilder();  
  
        assento.append(largura);  
        assento.append(" largura, ");  
        assento.append(comprimento);  
        assento.append(" comprimento, ");  
        assento.append(altura);  
        assento.append(" altura");  
  
        return assento.toString();  
    }  
}
```

## Exercício 3 - Resolução

- Para a classe Vidro:
- Veja que a película só é incluída no **toString()** caso o vidro tenha-a;

```
public class Vidro {  
  
    private float largura;  
    private float comprimento;  
    private boolean pelicula;  
  
    ... // Construtor, getters e setters  
  
    public String toString() {  
  
        StringBuilder vidro = new StringBuilder();  
  
        vidro.append(largura);  
        vidro.append(" largura, ");  
        vidro.append(comprimento);  
        vidro.append(" comprimento");  
  
        if (pelicula) {  
            vidro.append(" , possui pelicula");  
        }  
  
        return vidro.toString();  
    }  
}
```



## Exercício 3 - Resolução

- Para a classe Porta:
- A janela mesma coisa, somente caso ela exista;

```
public class Porta {  
  
    private float comprimento;  
    private float largura;  
    private Vidro janela;  
  
    ... // Construtor, getters e setters  
  
    public String toString() {  
  
        StringBuilder porta = new StringBuilder();  
  
        porta.append(largura);  
        porta.append(" largura, ");  
        porta.append(comprimento);  
        porta.append(" comprimento, ");  
  
        if (janela != null) {  
            porta.append(" janela: ");  
            porta.append(janela.toString());  
        }  
  
        return porta.toString();  
    }  
}
```

## Exercício 3 - Resolução

- Para a classe Roda:
- Calota, mesma história da película e janela;

```
public class Roda {  
  
    private int aro;  
    private String marca;  
    private boolean calota;  
  
    ... // Construtor, getters e setters  
  
    public String toString() {  
  
        StringBuilder roda = new StringBuilder();  
  
        roda.append("aro ");  
        roda.append(arro);  
        roda.append(" — ");  
        roda.append(marca);  
  
        if (calota) {  
            roda.append(" — possui calota");  
        }  
  
        return roda.toString();  
    }  
}
```

## Exercício 3 - Resolução

- Agora finalmente a classe Carro;
- O objetivo dela é modelar um Carro;
- Ou seja, ela vai ter rodas, portas, assentos e um tipo de combustível;
- E para tais atributos, é necessário ter getters e setters;

## Exercício 3 - Resolução

- Veja os atributos da classe Carro:
  - Quatro rodas e portas;
  - Cinco assentos;
  - E um tipo de combustível;

```
public class Carro {  
  
    private Porta[] portas = new Porta[4];  
    private Roda[] rodas = new Roda[4];  
    private Assento[] assentos = new Assento[5];  
  
    private TipoCombustivel tipoCombustivel;  
  
}
```

## Exercício 3 - Resolução

- Para o array de portas, um get e um set;

```
public class Carro {  
    ...  
    public Porta[] getPortas() {  
        return this.portas;  
    }  
    public void setPortas(Porta[] portas) {  
        this.portas = portas;  
    }  
}
```

## Exercício 3 - Resolução

- O mesmo para os assentos e rodas;

```
public class Carro {  
  
    ...  
  
    public Roda[] getRodas() {  
        return this.rodas;  
    }  
  
    public void setRodas(Roda[] rodas) {  
        this.rodas = rodas;  
    }  
  
    public Assento[] getAssentos() {  
        return this.assentos;  
    }  
  
    public void setAssentos(Assento[] assentos) {  
        this.assentos = assentos;  
    }  
  
}
```

## Exercício 3 - Resolução

- E o tipo de combustível também:

```
public class Carro {  
    ...  
    public TipoCombustivel getTipoCombustivel() {  
        return this.tipoCombustivel;  
    }  
    public void setTipoCombustivel(TipoCombustivel tipo)  
    {  
        this.tipoCombustivel = tipo;  
    }  
}
```

## Exercício 3 - Resolução

- O enunciado só pedia isso, caso deseje, avance para o slide 69(exercício 4).
- Mas já que muitos alunos criaram um método **main()** e instanciaram várias coisas, vamos fazer também;
- Mas o método **main()** não deve ficar dentro da classe Carro;
- Ele deve possuir uma classe própria para ele;
- Antes disso, vamos fazer um método **toString()** para a classe Carro;



## Exercício 3 - Resolução

- Vamos utilizar o `StringBuilder` novamente;
- E vamos primeiro iterar sobre todas as portas do carro;
- Como é um array, vamos utilizar a sintaxe do `forEach`;
- Para cada porta "p" do array `portas` (o `this` é opcional), vamos dar um `append` no `String` que será retornada e adicionar uma quebra de linha;

```
public String toString() {  
    StringBuilder carro = new StringBuilder();  
  
    carro.append("Portas do carro:\n");  
    for (Porta p : portas) {  
        carro.append(p.toString());  
        carro.append("\n");  
    }  
    ...  
}
```

## Exercício 3 - Resolução

- Mesma coisa para as rodas:

```
public String toString() {  
  
    ...  
  
    carro.append("Rodas do carro:\n");  
    for (Roda r : rodas) {  
        carro.append(r.toString());  
        carro.append("\n");  
    }  
  
    ...  
}
```

## Exercício 3 - Resolução

- Para o assento:

```
public String toString() {  
    ...  
    carro.append("Assentos do carro:\n");  
    for (Assento a : assentos) {  
        carro.append(a.toString());  
        carro.append("\n");  
    }  
    ...  
}
```

## Exercício 3 - Resolução

- E para o tipo de combustível;
- Ao final podemos retornar a String construída;
- Note que utilizamos o método **name()** do enum;
- Ele retorna o valor enumerado;
- Só que como a conversão é utilizar letras maiúsculas, precisamos usar o método **toLowerCase()** da classe String para transformar em letras minúsculas;

```
public String toString() {  
    ...  
    carro.append("Carro movido a ");  
    carro.append(tipoCombustivel.name().  
toLowerCase());  
    return carro.toString();  
}
```

## Exercício 3 - Resolução

- Agora vamos criar uma classe Main, contendo um método **main()**;
- E já vamos instanciar um objeto Carro;

```
public class Main {  
    public static void main(String [] args) {  
        Carro carro = new Carro();  
    }  
}
```

## Exercício 3 - Resolução

- Vamos definir um array de assentos, e setar como atributo do Carro;
- Nos slides o código do construtor com os atributos não foi mostrado, mas o construtor pede: largura, comprimento e altura, respectivamente.
- Vamos criar 5 assentos e definir as dimensões deles;
- Utilizando o método **setAssentos()** da classe Carro;

```
public class Main {  
    public static void main(String [] args) {  
        Carro carro = new Carro();  
        Assento [] assentos = new Assento [5];  
        for (int i = 0; i < 5; i++) {  
            assentos[i] = new Assento(30, 20,  
60);  
        }  
        carro.setAssentos(assentos);  
    }  
}
```

## Exercício 3 - Resolução

- O mesmo para as rodas: aro, marca e calota;

```
public class Main {  
    public static void main(String [] args) {  
        ...  
        Roda[] rodas = new Roda[4];  
        for (int i = 0; i < 4; i++) {  
            rodas[i] = new Roda(20, "sem marca",  
true);  
        }  
        carro.setRodas(rodas);  
    }  
}
```

## Exercício 3 - Resolução

- E para as portas:  
comprimento, largura e  
janela (pode ser null pois é  
um objeto do tipo vidro);
- Note que criamos o Vidro  
direto no construtor da  
Porta;

```
public class Main {  
    public static void main(String [] args) {  
        ...  
        Porta [] portas = new Porta [4];  
        for (int i = 0; i < 4; i++) {  
            portas[i] = new Porta(40, 50, new  
Vidro(20, 20, true));  
        }  
        carro.setPortas(portas);  
    }  
}
```



## Exercício 3 - Resolução

- Para o tipo de combustível, basta definir o tipo;
- Diretamente no set:

```
public class Main {  
    public static void main(String [] args) {  
        ...  
        carro.setTipoCombustivel(TipoCombustivel  
            .GASOLINA);  
    }  
}
```

## Exercício 3 - Resolução

- E após tudo isso, vamos exibir o objeto no console:

```
public class Main {  
    public static void main(String [] args) {  
        ...  
        System.out.println(carro);  
    }  
}
```

## Exercício 3 - Resolução

Portas do carro:

50.0 largura , 40.0 comprimento , janela: 20.0 largura , 20.0 comprimento , possui  
película

50.0 largura , 40.0 comprimento , janela: 20.0 largura , 20.0 comprimento , possui  
película

50.0 largura , 40.0 comprimento , janela: 20.0 largura , 20.0 comprimento , possui  
película

50.0 largura , 40.0 comprimento , janela: 20.0 largura , 20.0 comprimento , possui  
película

Rodas do carro:

aro 20 — sem marca — possui calota

aro 20 — sem marca — possui calota

aro 20 — sem marca — possui calota

aro 20 — sem marca — possui calota

Assentos do carro:

30.0 largura , 20.0 comprimento , 60.0 altura

30.0 largura , 20.0 comprimento , 60.0 altura

30.0 largura , 20.0 comprimento , 60.0 altura

30.0 largura , 20.0 comprimento , 60.0 altura

30.0 largura , 20.0 comprimento , 60.0 altura

Carro movido a gasolina

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

**Exercício 4**

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

## Exercício 4

- Implemente duas classes em Java para representar uma turma de estudantes, uma das classes deve ser o Aluno e a outra deve ser a Turma.
- Cada aluno tem um nome e cinco notas e deve possuir também um método para calcular a média do aluno.
- A classe Turma tem um número n de alunos definido no construtor e deve conter um método para adicionar alunos na turma.
- A classe Turma também deve possuir um método para listar os alunos aprovados (média acima de 7,0).

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

## Exercício 4 - Resolução

- Agora que vimos os conceitos básicos de orientação a objetos, podemos definir algoritmos dentro das classes;
- Vamos começar implementando a classe Aluno;
- Nessa classe, vamos ter um método para adicionar as notas e um para calcular a média;

## Exercício 4 - Resolução

- Mais a frente na disciplina veremos como usar coleções, e isso irá facilitar muito os algoritmos;
- Mas agora faremos os trabalhos na mão;
- Para lidar com as notas, vamos implementar uma pilha;
- Temos que ter uma variável topo para podemos adicionar notas;
- Essa variável vai controlar onde devemos colocar a nota;

```
public class Aluno {  
    private String nome;  
  
    private float [] notas = new float [5];  
    private int topo = 0;  
}
```

- Vamos colocar um atributo nome para o Aluno;
- E também o array de notas;
- Vamos definir um número fixo 5 para a quantidade máxima de notas;



## Exercício 4 - Resolução

- No construtor vamos pedir o nome do aluno:

```
public class Aluno {  
    private String nome;  
  
    private float[] notas = new float[5];  
    private int topo = 0;  
  
    public Aluno(String nome) {  
        this.nome = nome;  
    }  
}
```

## Exercício 4 - Resolução

- Agora precisamos de um método para adicionar notas;
- Sempre vamos adiciona a nota no "topo" da nossa "pilha";
- A variável topo controla o lugar onde devemos colocar o valor passado como parâmetro;
- E cada vez que adicionamos uma nota, devemos prepará-la para a próxima inserção;
- Incrementando essa variável topo.
- Faremos esse método retornar um valor booleano.
- Só podemos adicionar a nota se estiver espaço no vetor, é claro!
- Alguns alunos trataram essa impossibilidade de adição;
- Mas exibiram no console uma mensagem quando não era possível adicionar um valor;

## Exercício 4 - Resolução

- Essa abordagem não é tão eficiente, porque assim a aplicação pode perder a sua continuidade;
- Isso é, já fizemos em algumas aulas sistemas que ficam dentro de um while até que o usuário queira sair;
- Imagine que estivessemos fazendo isso agora, se o método apenas exibir uma mensagem na tela, como fazemos para o método **main()** saber que a nota não foi adicionada?
- Simples. Vamos fazer ela retornar um booleano caso isso ocorra;
- Mais a frente da disciplina veremos formas melhores de fazer isso.

## Exercício 4 - Resolução

- Vamos então seguir essa lógica;
- O método será do tipo boolean;
- E só podemos adicionar uma nota caso o topo não esteja em 5, porque se não já estaríamos no limite do array.

```
public boolean adicionarNota(float nota) {  
    if (topo < 5) {  
    } else {  
    }  
}
```

## Exercício 4 - Resolução

- Caso o array esteja cheio, vamos direto retornar **false**

```
public boolean adicionarNota(float nota) {  
    if (topo < 5) {  
    } else {  
        false;  
    }  
}
```

## Exercício 4 - Resolução

- Agora, caso poderemos adicionar, vamos colocar essa nota na posição **topo** do array;
- E vamos incrementar tal variável para que na próxima inserção o método já saiba onde deve ser colocado.

```
public boolean adicionarNota(float nota) {  
    if (topo < 5) {  
        notas[topo] = nota;  
        topo++;  
    } else {  
        false;  
    }  
}
```

## Exercício 4 - Resolução

- E está feito o método de adicionar;
- Vamos fazer o **setNotas()**, usando esse método;
- Como não sabemos o que vai existir dentro do array de notas enviado pelo método (lembre-se que o método **setNotas** deve ser do mesmo tipo do atributo) vamos iterar sob todos os floats;
- Então temos que garantir que todas as notas do array serão adicionadas e o topo deslocado corretamente.

```
public void setNotas(float [] notas) {  
    for (int i = 0; i != notas.length || topo !=  
        5; i++) {  
        adicionarNota(notas[i]);  
    }  
}
```

- Para isso iteramos sob o array passado como parâmetro, parando apenas quando o array inteiro for iterado ou quando o topo for 5;
- O método **adicionarNotas()** já verifica o topo, mas temos que garantir que esse método não chame o **adicionarNotas()** se o atributo de notas estiver cheio.

## Exercício 4 - Resolução

- Agora vamos calcular a média;
- Esse método vai retornar um float;

```
public float calcularMedia() {  
}
```



## Exercício 4 - Resolução

- A lógica é muito simples, precisamos iterar sobre as notas e acumular esse valor em uma variável temporária;
- E ao final retornar a divisão dessa variável pela quantidade de notas, que definimos como 5:

```
public float calcularMedia() {  
    float media = 0;  
    for (float nota : notas) {  
        media += nota;  
    }  
    return media / 5;  
}
```

## Exercício 4 - Resolução

- Vamos só criar um método **toString()** para podermos usar mais tarde;
- Seguindo a mesma lógica do exercício 2 (slide 32), caso esteja em dúvida, revise-o.

```
public String toString() {  
    StringBuilder aluno = new StringBuilder();  
  
    aluno.append("Nome: ");  
    aluno.append(nome);  
    aluno.append("\n");  
  
    for (float nota : notas) {  
        aluno.append(nota);  
        aluno.append(" ");  
    }  
  
    aluno.append("Media: ");  
    aluno.append(calcularMedia());  
  
    return aluno.toString();  
}
```

## Exercício 4 - Resolução

- Agora que terminamos a classe Aluno, podemos avançar para a classe Turma.
- Nela teremos um array de alunos e faremos o mesmo que fizemos com as notas;
- Criaremos uma pilha de alunos com uma variável topo para controlar as inserções;
- Por causa dessa abordagem, o método **listarAprovados()** será um pouco trabalhoso;
- Mais a frente na disciplina aprenderemos sobre coleções, e será bem mais fácil fazer o que iremos fazer nesse método.
- Como o método de adicionar alunos segue a mesma lógica da classe anterior (slide 74), vamos apenas apresentar o código:
- A única diferença é que só iremos adicionar alunos que não forem ponteiros **null**.

## Exercício 4 - Resolução

- A classe Turma possui uma quantidade n de alunos definida no construtor;
- Mantemos esse valor em um atributo, para podermos saber o máximo de alunos que podemos ter;
- E também alocamos o array no construtor a partir desse valor.

```
public class Turma {  
  
    private int n;  
    private Aluno[] alunos;  
    private int topo = 0;  
  
    public Turma(int n) {  
        this.n = n;  
        alunos = new Aluno[n];  
    }  
  
    public boolean adicionarAluno(Aluno aluno) {  
  
        if (topo < n && aluno != null) {  
            alunos[topo] = aluno;  
            topo++;  
            return true;  
        } else {  
            return false;  
        }  
    }  
}
```

## Exercício 4 - Resolução

- Vamos agora construir o algoritmo que irá “filtrar” os alunos aprovados da turma;
- Como estamos lidando com arrays, precisamos primeiro adicionar a uma pilha auxiliar de alunos, os que possuem média superior a 7;
- Depois precisamos retornar um array com exatamente o número de alunos aprovados, para isso iremos alocar um terceiro array com essa quantidade;
- Como nem sempre a turma vai estar com todos os  $n$  alunos, alguns ponteiros podem estar nulos, precisamos tomar cuidado com isso também.

## Exercício 4 - Resolução

- Vamos começar definindo o retorno do método e alocando as variáveis necessárias para a pilha auxiliar de alunos:

```
public Aluno[] listarAprovados() {  
    Aluno[] auxiliar = new Aluno[n];  
    int auxiliarTopo = 0;  
}
```

## Exercício 4 - Resolução

- Agora precisamos iterar até o topo da pilha:

```
public Aluno[] listarAprovados() {  
    Aluno[] auxiliar = new Aluno[n];  
    int auxiliarTopo = 0;  
    for (int i = 0; i < topo; i++) {  
    }  
}
```

## Exercício 4 - Resolução

- Caso a média do aluno que está sendo iterado seja maior que 7, vamos fazer a mesma abordagem dos métodos de adição e colocá-lo na pilha auxiliar:
- Utilizaremos o método **calcularMedia()** do Aluno;

```
public Aluno[] listarAprovados() {  
    Aluno[] auxiliar = new Aluno[n];  
    int auxiliarTopo = 0;  
  
    for (int i = 0; i < topo; i++) {  
        if (alunos[i].calcularMedia() > 7) {  
            auxiliar[auxiliarTopo] = alunos[i];  
            auxiliarTopo++;  
        }  
    }  
}
```



## Exercício 4 - Resolução

- Agora que temos os alunos aprovados em uma pilha auxiliar, precisamos retornar um array que contenha somente esse alunos, sem os espaços em branco restantes;
- Pois veja, caso tenhamos 50 alunos em uma turma e só 5 deles forem aprovados, vamos retornar um array com 45 posições vazias;
- Para isso vamos alocar um novo array de acordo com o tamanho do topo da pilha auxiliar;

## Exercício 4 - Resolução

- Alocando o array....
- Depois disso, precisamos iterar novamente sobre todos os alunos desse array auxiliar e passá-los para o array a ser retornado;

```
public Aluno[] listarAprovados() {  
  
    Aluno[] auxiliar = new Aluno[n];  
    int auxiliarTopo = 0;  
  
    for (int i = 0; i < topo; i++) {  
        if (alunos[i] != null) {  
            auxiliar[auxiliarTopo] = alunos[i];  
            auxiliarTopo++;  
        }  
    }  
  
    Aluno[] aprovados = new Aluno[auxiliarTopo];  
  
    for (int i = 0; i < auxiliarTopo; i++) {  
        aprovados[i] = auxiliar[i];  
    }  
  
    return aprovados;  
  
}
```

## Exercício 4 - Resolução

- Agora terminamos a classe Turma;
- A grande maioria dos alunos, apenas exibiu o aluno que era aprovado, dentro do próprio método;
- Mas e caso quiséssemos fazer outra coisa com os alunos aprovados, por exemplo, criar a disciplina 2 daquela matéria?
- Não vamos fazer isso agora. Vamos apenas criar um método **main()** para ver os resultados;
- Novamente, essa questão não faz parte do exercício, então caso deseje, avance para o exercício 5 (slide 96).

## Exercício 4 - Resolução

- Vamos criar uma turma de 5 alunos e colocar três alunos dentro dela;
- Cada um com 5 notas.

```
public class Main {  
  
    public static void main(String [] args) {  
  
        Turma turma = new Turma(5);  
  
        Aluno a1 = new Aluno("Joao");  
        a1.setNotas(new float [] { 7, 6, 8, 6, 8 });  
        turma.adicionarAluno(a1);  
  
        Aluno a2 = new Aluno("Julia");  
        a2.setNotas(new float [] { 8, 6, 8, 6, 8 });  
        turma.adicionarAluno(a2);  
  
        Aluno a3 = new Aluno("Lucas");  
        a3.setNotas(new float [] { 7, 6, 10, 6, 8 });  
        turma.adicionarAluno(a3);  
  
    }  
  
}
```

## Exercício 4 - Resolução

- Vamos agora iterar sobre o array retornado pelo método **listarAprovados()** da turma, exibindo as informações dos alunos aprovados.

```
public class Main {  
    public static void main(String[] args) {  
        ...  
        Aluno a3 = new Aluno("Lucas");  
        a3.setNotas(new float[] { 7, 6, 10, 6, 8 });  
        turma.adicionarAluno(a3);  
  
        Aluno[] aprovados = turma.listarAprovados();  
  
        for (Aluno aluno : aprovados) {  
            System.out.println(aluno);  
        }  
    }  
}
```

## Exercício 4 - Resolução

- O resultado obtido:

Nome: Julia

8.0 6.0 8.0 6.0 8.0 M dia: 7.2

Nome: Lucas

7.0 6.0 10.0 6.0 8.0 M dia: 7.4

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

## Exercício 5

- Crie uma classe Carta contendo os atributos que desejar.
- Implemente outra classe chamada Baralho que contém um número n de cartas (podendo ser fixo).
- A classe Baralho deve conter o método **embaralhar()** responsável por distribuir as n cartas de forma aleatória.



# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

**Exercício 5 - Resolução**

Exercício 6

Exercício 6 - Resolução

## Exercício 5 - Resolução

- O objetivo desse exercício é criar uma classe Carta e outra classe Baralho, responsável por embaralhar um conjunto de cartas;
- Sendo o método de embaralhar independente de como as cartas são;
- Alguns alunos fizeram o método **embaralhar()** ser dependente de algum atributo da Carta;
- Não faremos dessa forma, pois queremos que o método **embaralhar()** independa dos atributos da Carta;
- Utilizaremos um array de cartas dentro do Baralho e os indexes desse array para embaralhá-las.
- Vamos começar definindo uma classe Carta contendo apenas um atributo: cor.

## Exercício 5 - Resolução

- Para esse atributo, vamos definir um get;
- Não vamos definir um set porque faremos o seguinte, a partir do momento que uma carta é criada, ela não pode ser modificada.

```
public class Carta {  
    private String cor;  
    public Carta(String cor) {  
        this.cor = cor;  
    }  
    public String getCor() {  
        return this.cor;  
    }  
}
```

## Exercício 5 - Resolução

- Para o Baralho vamos fazer a mesmo que no exercício anterior: um array de objetos (no nosso caso Cartas) e uma variável topo para controlar as inserções;
- Também vamos definir um atributo n, assim como requisitado no enunciado;
- Esse valor n vai controlar a quantidade máxima de cartas.

## Exercício 5 - Resolução

- Utilizaremos a classe Random para gerar valores aleatórios;
- Também vamos criar um construtor que pede o máximo de cartas e já aloca o array a partir desse valor.
- Além também de um get para o array;

```
import java.util.Random;

public class Baralho {

    private int n;
    private Carta[] cartas;
    private int topo = 0;

    public Baralho(int n) {
        this.n = n;
        this.cartas = new Carta[n];
    }

    public Carta[] getCartas() {
        return this.cartas;
    }

}
```

## Exercício 5 - Resolução

- O método de adicionarCartas segue a mesma lógica dos métodos do exercício anterior (veja explicação detalhada no slide 74);
- Incrementa o topo ao final, preparando-o para a próxima vez;
- E colocando a carta no array;
- Além de retornar true se tudo funcionou e false se não foi possível adicionar.

```
import java.util.Random;

public class Baralho {

    ...

    public Carta[] getCartas() {
        return this.cartas;
    }

    public boolean adicionarCarta(Carta c) {
        if (topo < n) {
            cartas[topo] = c;
            topo++;
            return true;
        } else {
            return false;
        }
    }
}
```

## Exercício 5 - Resolução

- Antes de criarmos o método que de fato irá embaralhar as cartas, vamos criar um método para “puxar” uma carta do meio do array;
- Mais a frente na disciplina utilizaremos coleções para fazer isso, o que torna muito mais simples, pois as coleções possuem um método para embaralhar chamado: **shuffle()**.
- Mas por enquanto vamos fazer na mão.
- Vamos implementar então o método **tirarCarta()**, ele vai receber a posição do array que gostaríamos de tirar a carta e vai retornar a carta daquela posição, puxando todas as demais uma posição atrás.
- Uma analogia a uma pilha de livros, se você tira um livro do meio, o buraco é preenchido com os livros que estavam em cima.

## Exercício 5 - Resolução

- Primeiro vamos verificar se a posição que o método recebeu existe.
- Se ela for menor que o topo, vamos construir a lógica;
- Caso não, vamos retornar null.

```
import java.util.Random;

public class Baralho {

    ...

    public boolean adicionarCarta(Carta c) {
        ...
    }

    public Carta tirarCarta(int pos) {
        if (pos < topo) {
            ...
        }
        return null;
    }
}
```



## Exercício 5 - Resolução

- Agora vamos declarar uma carta “c” que será a carta que retornaremos;
- Não podemos retorná-la direto, pois precisamos “puxar” todas as cartas a frente dela no array em uma posição.

```
public Carta tirarCarta(int pos) {  
    if (pos < topo) {  
        Carta c = cartas[pos];  
  
        ...  
  
        return c;  
    }  
  
    return null;  
}
```

## Exercício 5 - Resolução

- Para isso vamos iterar sob todas as cartas seguintes a que salvamos na variável “c”;
- Puxando cada uma delas uma posição no array;

```
public Carta tirarCarta(int pos) {  
    if (pos < topo) {  
        Carta c = cartas[pos];  
        for (int i = pos; i < topo - 1; i++) {  
            cartas[i] = cartas[i + 1];  
        }  
        ...  
        return c;  
    }  
    return null;  
}
```

## Exercício 5 - Resolução

- Agora, temos que tirar 1 do topo, pois vamos tirar uma carta do array;
- E definir a última carta do array como `null`, pois no momento temos duas cartas repetidas ao final.
- E está feito o algoritmo de tirar cartas.

```
public Carta tirarCarta(int pos) {  
    if (pos < topo) {  
        Carta c = cartas[pos];  
  
        for (int i = pos; i < topo - 1; i++) {  
            cartas[i] = cartas[i + 1];  
        }  
        topo--;  
        cartas[topo] = null;  
  
        return c;  
    }  
    return null;  
}
```

## Exercício 5 - Resolução

- Agora vamos fazer o algoritmo de embaralhar;
- Para isso, ao invés de reordenar o próprio array, vamos criar um array temporário e jogar cartas de posições aleatórias nela;

## Exercício 5 - Resolução

- Esse método não vai retornar nada;
- Utilizaremos a classe Random para gerar os valores aleatório das posições que vamos tirar as cartas;
- E criaremos um array temporário e uma variável para controlar as inserções, tal como fizemos para o método de adicionar.

```
public void embaralhar() {  
    Random random = new Random();  
    Carta[] nova = new Carta[n];  
    int topoNova = 0;  
    ...  
}
```

## Exercício 5 - Resolução

- Nosso algoritmo vai ao final substituir o array das cartas pelo temporário.
- E vamos “passar” as cartas de um array para o outro, utilizando um laço `while` que só vai parar quando não houver mais cartas no array principal (que é o atributo da classe Baralho).
- Sabemos que não há mais cartas, quando o topo é 0.

```
public void embaralhar() {  
    Random random = new Random();  
    Carta [] nova = new Carta[n];  
    int topoNova = 0;  
    while (topo != 0) {  
        ...  
    }  
    this.topo = topoNova;  
    this.cartas = nova;  
}
```

- Também precisamos passar o valor final do topo novo para o topo que é atributo da classe, visto que ao final do laço de repetição ele é 0.

## Exercício 5 - Resolução

- Para gerar o número aleatório vamos utilizar o objeto `Random`;
- Passando para ele o topo como parâmetro no método **`nextInt()`**, assim ele vai gerar um valor aleatório entre 0 e o topo.

```
public void embaralhar() {  
    Random random = new Random();  
    Carta[] nova = new Carta[n];  
    int topoNova = 0;  
    while (topo != 0) {  
        int posicaoAleatoria = random.nextInt(  
topo);  
    }  
    this.topo = topoNova;  
    this.cartas = nova;  
}
```

## Exercício 5 - Resolução

- Agora para passar a carta para o array temporário, vamos chamar o método **tirarCarta()**, passando a posição que geramos aleatoriamente;
- E precisamos incrementar o topo do array temporário também, preparando o while para a próxima iteração (topoNova++);
- Pronto, finalizamos o algoritmo.

```
public void embaralhar() {  
    Random random = new Random();  
    Carta [] nova = new Carta[n];  
    int topoNova = 0;  
    while (topo != 0) {  
        int posicaoAleatoria = random.nextInt(  
topo);  
        nova[topoNova] = tirarCarta(  
posicaoAleatoria);  
        topoNova++;  
    }  
    this.topo = topoNova;  
    this.cartas = nova;  
}
```



## Exercício 5 - Resolução

- Orientação a objetos preza a reutilização de código;
- Por isso evite ao máximo criar classes que dependem de informações de outras, por exemplo, o Baralho e Carta;
- Se o método embaralhar() usasse atributos da Carta para isso, e quiséssemos alterar a implementação da Carta, quantas alterações precisariam ser feitas no código da classe Baralho?
- Na implementação que fizemos da classe Baralho, se a classe Carta não tivesse mais nenhum atributo, o código continuaria funcionando?

## Exercício 5 - Resolução

- Como o exercício está completo, caso deseje, avance para o exercício 6 (slide 120).
- Vamos criar uma classe Main e um método **main()** para testarmos.
- Criaremos também um baralho com cinco cartas, com as cores:
  - Azul;
  - Vermelho;
  - Verde;
  - Roxo;
  - Amarelo;

## Exercício 5 - Resolução

- Passamos 5 no construtor da classe Baralho;
- E instanciamos cinco cartas.

```
public class Main {  
    public static void main(String[] args) {  
        Baralho baralho = new Baralho(5);  
  
        baralho.adicionarCarta(new Carta("Azul"));  
        baralho.adicionarCarta(new Carta("Vermelho"));  
;  
        baralho.adicionarCarta(new Carta("Verde"));  
        baralho.adicionarCarta(new Carta("Roxo"));  
        baralho.adicionarCarta(new Carta("Amarelo"));  
    }  
}
```

## Exercício 5 - Resolução

- Vamos iterar sob o array de cartas e exibi-las;
- Poderíamos ter criado um método **toString()** para isso na classe Baralho!

```
public static void main(String[] args) {  
    Baralho baralho = new Baralho(5);  
  
    baralho.adicionarCarta(new Carta("Azul"));  
    baralho.adicionarCarta(new Carta("Vermelho"));  
;  
    baralho.adicionarCarta(new Carta("Verde"));  
    baralho.adicionarCarta(new Carta("Roxo"));  
    baralho.adicionarCarta(new Carta("Amarelo"));  
  
    for (Carta c : baralho.getCartas()) {  
        if (c != null) {  
            System.out.println(c.getCor());  
        }  
    }  
}
```

## Exercício 5 - Resolução

- Agora vamos chamar o método **embaralhar()** e depois exibir as cartas novamente.

```
public static void main(String[] args) {  
    ...  
    baralho.embaralhar();  
    System.out.println(">>> Embaralhado");  
    for (Carta c : baralho.getCartas()) {  
        if (c != null) {  
            System.out.println(c.getCor());  
        }  
    }  
}
```

## Exercício 5 - Resolução

- Executando a **main()** temos:

```
Azul  
Vermelho  
Verde  
Roxo  
Amarelo  
>>> Embaralhado  
Vermelho  
Azul  
Amarelo  
Verde  
Roxo
```

# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

**Exercício 6**

Exercício 6 - Resolução

## Exercício 6

- Criar a classe Registrador e a classe CPU, onde o Registrador possui uma limitação de 8 bits;
- Criar os métodos da classe CPU:
  - **void addRegistrador(Registrador registrador):** esse método adiciona o registrador ao array de registradores da cpu;
  - **void getRegistradores():** esse método retorna os registradores da cpu;
  - **Registrador operacaoOr(Registrador r1, Registrador r2):** esse método irá fazer  $r3 = r1 \text{ or } r2$ ;
  - **Registrador operacaoAnd(Registrador r1, Registrador r2):** esse método irá fazer  $r3 = r1 \text{ and } r2$ .
- O enunciado completo está disponível nesse [link](#).



# Seções

Introdução

Exercício 1

Exercício 1 - Resolução

Exercício 2

Exercício 2 - Resolução

Exercício 3

Exercício 3 - Resolução

Exercício 4

Exercício 4 - Resolução

Exercício 5

Exercício 5 - Resolução

Exercício 6

Exercício 6 - Resolução

## Exercício 6 - Resolução

- Vamos começar pela classe Registrador;
- Como temos que representar um array de 0's e 1's, vamos utilizar um array de booleanos;
- Com o tamanho fixo em 8;
- E para visualizarmos isso, criaremos um método **toString()** que indentará os booleanos lado a lado, representando-os com 0's e 1's.

## Exercício 6 - Resolução

- Vamos definir esse array de booleanos e seu get e set:

```
public class Registrador {  
    private boolean[] bits = new boolean[8];  
  
    public boolean[] getBits() {  
        return this.bits;  
    }  
  
    public void setBits(boolean[] bits) {  
        this.bits = bits;  
    }  
}
```

## Exercício 6 - Resolução

- Agora vamos fazer um set e um get para posições específicas do array;
- Sempre tomando cuidado para que não ultrapássemos o tamanho máximo do array que é 8:

```
public class Registrador {  
  
    ...  
  
    public void setBit(int pos, boolean bit) {  
        if (pos < 8) {  
            this.bits[pos] = bit;  
        }  
    }  
  
    public boolean getBit(int pos) {  
        if (pos < 8) {  
            return this.bits[pos];  
        }  
        return false;  
    }  
}
```

## Exercício 6 - Resolução

- E por fim o toString;
- Vamos utilizar a classe `StringBuilder` que utilizamos nos exercícios anteriores (ver slide 32).
- Iteraremos sob o array e adicionaremos 0 ou 1 de acordo com o valor;

```
public class Registrador {  
    ...  
    public String toString() {  
        StringBuilder builder = new StringBuilder();  
        for (boolean bit : bits) {  
            if (bit) {  
                builder.append("1 ");  
            } else {  
                builder.append("0 ");  
            }  
        }  
        return builder.toString();  
    }  
}
```

## Exercício 6 - Resolução

- Vamos agora para a classe CPU;
- Utilizaremos a mesma técnica dos exercícios anteriores para adicionar Registradores em um array de Registrador;
- Como já fizemos três vezes isso, não iremos detalhar;
- Caso deseje, retorne ao slide 74 e aplique a mesma lógica das notas para o método adicionarRegistrador().

## Exercício 6 - Resolução

- Então nossa classe CPU só terá dois atributos responsáveis pelo array de Registradores;
- Além do método get para esse array e o método de adicionar nesse array:

```
public class CPU {  
  
    private Registrador[] registradores = new Registrador[8];  
    private int topo = 0;  
  
    public boolean adicionarRegistrador(Registrador r) {  
        if (topo < 8) {  
            registradores[topo] = r;  
            topo++;  
            return true;  
        } else {  
            return false;  
        }  
    }  
  
    public Registrador[] getRegistradores() {  
        return this.registradores;  
    }  
}
```

## Exercício 6 - Resolução

- Agora vamos construir o método **operacaoOr()**;
- Para isso, teremos que iterar sob os registradores recebidos;
- Se um dos bits (a cada iteração) for **true**, podemos setar o valor do terceiro registrador como true;
- Se não, setaremos como false;
- Faremos essa verificação dos dois valores usando um **or**.



## Exercício 6 - Resolução

- Precisamos do index do array, por isso iteramos com um for utilizando uma variável 'i':
- Passando, a cada iteração, a posição 'i' para o método **getBit()** de ambos registradores;

```
public class CPU {  
    ...  
    public Registrador operacaoOr(Registrador r1, Registrador  
r2) {  
        Registrador r3 = new Registrador();  
        for (int i = 0; i < 8; i++) {  
            if (r1.getBit(i) || r2.getBit(i)) {  
                r3.setBit(i, true);  
            } else {  
                r3.setBit(i, false);  
            }  
        }  
        return r3;  
    }  
}
```

## Exercício 6 - Resolução

- Para o método **operacao-And()**, a lógica é a mesma, só precisamos substituir o operador de dentro do if para o **and**:

```
public class CPU {  
    ...  
    public Registrador operacaoAnd(Registrador r1, Registrador  
r2) {  
        Registrador r3 = new Registrador();  
        for (int i = 0; i < 8; i++) {  
            if (r1.getBit(i) && r2.getBit(i)) {  
                r3.setBit(i, true);  
            } else {  
                r3.setBit(i, false);  
            }  
        }  
        return r3;  
    }  
}
```

## Exercício 6 - Resolução

- Feito isso, vamos criar uma classe Main com um método **main()** para realizarmos testes;
- Instanciaremos um CPU e dois Registradores (apenas para testes, pois o exercício está completo).

```
public class Main {  
    public static void main(String[] args) {  
        CPU cpu = new CPU();  
  
        Registrador r1 = new Registrador();  
        r1.setBits(new boolean[] { true, true, false, true, false, true, false,  
false });  
  
        Registrador r2 = new Registrador();  
        r2.setBits(new boolean[] { false, true, false, false, true, true, true,  
false });  
    }  
}
```

## Exercício 6 - Resolução

- Agora salvaremos o resultado da chamada dos métodos em dois registradores r3 e r4, e depois exibiremos o resultado:

```
public class Main {  
    public static void main(String[] args) {  
        ...  
  
        Registrador r3 = cpu.operacaoAnd(r1, r2);  
  
        System.out.println(r1);  
        System.out.println(r2);  
        System.out.println(r3);  
  
        Registrador r4 = cpu.operacaoOr(r1, r2);  
  
        System.out.println(r1);  
        System.out.println(r2);  
        System.out.println(r4);  
    }  
}
```


## Exercício 6 - Resolução

- Executando temos isso:
- Para a operação and:

```
1 1 0 1 0 1 0 0
0 1 0 0 1 1 1 0
1 1 0 1 1 1 1 0
```

- E para a operação or:

```
1 1 0 1 0 1 0 0
0 1 0 0 1 1 1 0
1 1 0 1 1 1 1 0
```

 KUWAKI, V. T. F. Modelo de slides udesc lattex. In: . [S.l.]: Disponível em: <<https://github.com/takeofriedrich/slidesUdescLattex>>. Acesso em: 5 jun. 2020.

Duvidas:  
Vinicius Takeo Friedrich Kuwaki  
vtkwki@gmail.com  
[github.com/takeofriedrich](https://github.com/takeofriedrich)