Sumário

Introdução

Quando uma linguagem de programação é compilada, o código que foi escrito será passado para um programa chamado `compilador` para ser transformado em código de baixo nível. Essa linguagem de baixo nível é conhecida como `assembly` . Ou seja, linguagens como C ou C++ tem seu código compilado para `assembly` . Por sua vez o `assembly` é
transformado em linguagem de máquina por um `assembler` (uma tradução direta é montar ou reunir — basta lembrar de
Vingadores Ultimato, quando o Capitão América pega o Martelo do Thor e diz: "Avengers! Assemble"), um programa que vai transformar a linguagem de baixo nível em linguagem de máquina propriamente dita (na verdade ainda existem vários detalhes aí, só que por questão de simplicidade vamos deixar como está).

Entretanto cada arquitetura de computador (por enquanto você pode entender isso apenas como cada família de processador) possui instruções diferentes, ou seja, para cada arquitetura nós temos um `assembly` diferente. Isso fez com que há alguns anos um programa compilado funcionava somente em alguns computadores, mas não em outros. Ou seja, para garantir que um determinado programa fosse executado no máximo de máquinas possíveis, ele deveria ser adaptado e compilado várias vezes, por diferentes compiladores.

Diante dessa dificuldade, os projetistas da linguagem `Java` aplicaram o conceito de `máquina virtual` e criaram a `JVM` ou *Java Virtual Machine* (Máquina Virtual Java). O compilador `javac` transforma o código em arquivos `.class` no formato `bytecode`. A JVM executa o `bytecode`. Ou seja, a execução de um programa em `Java` não depende mais da arquitetura do computador. Isso ficou conhecido como "*Write once, run everywhere*" (traduzindo: escreva uma vez, execute em qualquer lugar).

Isso também significa que, qualquer linguagem que tenha algum compilador para `bytecode` pode ser executada por uma `JVM`. Exemplos: `Scala`, `JRuby`, `Jython`, `Clojure`.

Para que a `JVM` consiga executar um programa apropriadamente, ela precisa do `JRE` ou *Java Runtime Environment* (Ambiente de Tempo de Execução Java). O `JRE` possui as bibliotecas padrões do Java, inclusive o arquivo executável, o qual executa a classe que tenha o método `main`.

Para programar em `Java` você precisa baixar e instalar o `JDK` *Java Development Kit* (Kit de Desenvolvimento Java). O `JDK` fornece ferramentas de desenvolvimento, e também o `JRE`. O `JRE`, como já vimos, fornece as bibliotecas padrões, e também uma `JVM`.

Atualmente a linguagem `Java` é propriedade da `Oracle`. Temos então duas alternativas: instalar o `JDK` com a licença da `Oracle` (o que possivelmente implica em você pagar honorários em determinadas situações, além de receber suporte deles), ou o `OpenJDK`, a versão gratuita e livre.

Programação Orientada a Objetos

A **Programação Orientada a Objetos** (`POO`) é um `paradigma` de linguagens de programação baseado no conceito de objetos.

Os `paradigmas` são uma maneira de classificar as linguagens de programação de acordo com suas características.

Tipo Abstrato de Dados

Primeiro vamos lembrar de alguns tipos de dados `primitivos` :

- Inteiro;
- Float;
- Char.

Tipos primitivos descrevem valores únicos e básicos: uma nota, um peso, uma idade, quantos objetos em um local, etc.

Como descrever um conjunto de valores? Podemos utilizar de `estruturas de dados abstratos` , `homogêneos` ou `heterogêneos` . Por exemplo, o conjunto das notas da primeira avaliação pode ser armazenado em um `array` , também conhecido como `vetor` .

```
double notas = [9, 4.5, 6.7, 8.9, 6.2, 7.5];
```

Como o `array` só possui valores do tipo `double` , ele é um conjunto de dados homogêneo. O armazenamento das notas das duas avaliações pode extender o vetor para 2 `dimensões` e então temos uma `matriz` .

```
notas = [9, 4.5, 6.7, 8.9, 6.2, 7.5; 8.4, 7.3, 6.9, 8.7, 7.0, 7.2]

9.0 4.5 6.7 8.9 6.2 7.5
8.4 7.3 6.9 8.7 7.0 7.2
```

Uma vez que um conjunto de dados homogêneo só tem dados do mesmo tipo, obviamente um conjunto de dados heterogêneo possui dados de mais de um tipo. Outro exemplo utilizando uma `matriz` :

```
alunos_notas = ["João", "Maria", "José", "Rosicleide", "Cleyton", "Bervely";9, 4.5,
        9, 8.7, 7.0, 7.2]

"João" "Maria" "José" "Rosicleide" "Cleyton" "Bervely"
9.0    4.5      6.7    8.9         6.2       7.5
8.4    7.3      6.9    8.7         7.0       7.2
```

Percebam que com estruturas de dados heterogêneas começamos a descrever coisas mais complexas. Começamos a `abstrair` conceitos da realidade. No exemplo temos

pessoas e para cada pessoa duas notas associadas.

Vamos descrever uma pessoa de forma simplificada:

- Nome;
- Peso;
- Idade;
- Altura;
- CPF.

Então temos a primeira pessoa descrita da seguinte forma:

- Nome: "João";
- Peso: 75 kg;
- Idade: 22 anos;
- Altura: 1.76 m;
- CPF: 123.456.789-00

E se tivermos outra pessoa? Vejamos a descrição:

- Nome: "Maria";
- Peso: 58 kg;
- Idade: 21 anos;
- Altura: 1.62;
- CPF: 234.567.890-11

São pessoas diferentes, mas descritas com os mesmos `atributos`. Então agora podemos dizer que temos um tipo abstrato de dados chamado `pessoa`. Porém as pessoas podem falar e fazer atividades. Então podemos acrescentar às pessoas a possibilidade das seguintes atitudes:

- Dizer o nome;
- Dizer o peso;
- Dizer a idade;
- Dizer a altura;
- Dizer o CPF;
- Conversar;
- Caminhar;
- Correr;

- Praticar um esporte.

Agora temos a descrição não somente dos `atributos` de uma pessoa como também a forma como elas podem ser "manipuladas" e manipular o mundo. O termo técnico para isso é uma `classe` . Ou seja, uma `classe` é uma descrição de alguma coisa através de `atributos` e `métodos` pelos quais essa coisa pode manipular o mundo ou ser manipulada.

A descrição é geral, mas podemos ter `instâncias` dessas descrições, ou seja, `objetos` "tangíveis", a "encarnação" da classe. Por exemplo, toda nova pessoa existente é a `instância` da `classe` pessoa. Portanto, `objetos` são a "personificação" de alguma descrição abstrata.

Classes

Como acabamos de ver as `classes` podem ser vistas como estruturas de dados abstratos e heterogêneos. Mas, mais do que isso, as `classes` permitem ações. Os `objetos` de cada classe é que irão interagir entre si através dessas ações.

A seguir, como seria a `classe` pessoa em `Java` :

```
public class Pessoa{
    String nome;
    double peso;
    int idade;
    double altura;
    String cpf;

    public String getNome(){
        return nome;
    }

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

    public double getPeso(){
        return peso;
    }
```

```
    public void setPeso(double peso){
        this.peso = peso;
    }

    //...

    public void conversar(String assunto){
        System.out.println("Eu gosto muito desse " + assunto);
    }

    //...
}
```

Vamos destrinchar a classe que acabamos de criar.

Em `Java`, para sinalizar que estamos criando uma classe, escrevemos a palavra reservada `class` e depois o nome da `classe`; no nosso caso foi Pessoa. Por padrão os nomes das classes começam com letra maiúscula. Se o nome for composto, é tudo escrito junto e a primeira letra da nova palavra é escrita em maiúsculo também. Isso é chamado de `camel case`. Ex.: MinhaClasse.

Em `Java` o conteúdo de todos os blocos fica contido entre chaves `{ }`. Dentro da classe que acabamos de criar declaramos seus atributos, com seus respectivos tipos. Após declarar os atributos passamos a declarar os métodos da classe, ou seja, como ela vai interagir.

Por padrão os métodos que permitem consultar e modificar os valores dos atributos são os `getters` e `setters`. Com o método `get` nós conseguimos acessar o valor de um determinado atributo, e com o método `set` nós modificamos o valor daquele atributo. Percebe que no nosso caso, nos métodos `setters` temos a requisição de `parâmetros`, ou seja, para que o método seja executado, precisamos passar algum valor, e esse valor tem de ser do tipo especificado.

Outra coisa a se prestar atenção na declaração dos métodos são as palavras `String`, `double` e `void`. Isso acontece porque o Java exige que toda declaração de método seja explicitado o tipo de retorno. Portanto, quando criamos o método `double getPeso()`, significa que esperamos receber um valor do tipo `double` ao executar esse método. A mesma coisa acontece para os demais tipos. Existe porém um tipo "especial", o `void`. Quando utilizamos esse tipo dizemos que o método não vai retornar nada.

Agora vamos ver dentro dos métodos. Naqueles em que esperamos algum tipo de retorno nós utlizamos a palavra reservada `return`. É assim que "entregamos" o valor que está sendo esperado receber. Logo, se usamos o método `getNome()`, receberemos a `String` que está armazenada no atributo `nome`.

Por fim temos a palavra reservada `this`. Quando usamos essa palavra nós estamos explicitando que o atributo que queremos manipular é o atributo daquela classe. Perceba que no método `setNome()` nós recebemos por parâmetro uma `String` chamada `nome`, que por coincidência é exatamente o mesmo nome do atributo. Então para deixar claro ao compilador que estamos modificando o atributo da classe, utilizamos o `this`. Quando o método for executado, aquele atributo passará a ter o valor do parâmetro.

Objeto

Reforçando, um `objeto` é uma instância de uma `classe`. Enquanto uma `classe` é uma descrição abstrata de alguma coisa e o que essa coisa pode fazer, o objeto seria a "encarnação" dessa classe ("encarnação" não é o termo adequado, mas ajuda a entender).

Outra forma de entender o que é um `objeto` é vê-lo como uma espécie de `variável` de um `tipo abstrato de dado`. Por exemplo, se declaramos `int i = 2`, sabemos que `i` é uma variável do tipo primitivo `Inteiro` e que estará armazenando o valor 2. A operação correspondente a um objeto é "trazê-lo à existência". Exemplo:

```
Pessoa pessoa = new Pessoa();
```

No exemplo dado estamos criando um `objeto` chamado `pessoa` do tipo `Pessoa`. Para que haja a criação propriamente dita é necessário escrever explicitamente com a palavra reservada `new`, seguido da classe da qual o `objeto` deve ser.

Quando declaramos uma `variável` de algum `tipo primitivo` o que acontece "por detrás dos panos" é que será procurada na memória um espaço suficiente para armazenar aquele dado. Portanto, se o `int` for armazenado com 32 bits, assim que declaramos a variável `i`, será procurada na memória um espaço de 32 bits, o qual será vinculado à variável `i`. Como demos o valor 2 a essa variável, nesse endereço de memória que foi vinculado a ela, estará armazenado lá o valor 2.

Quando criamos um `objeto` acontece algo similar. Explicando de forma **extremamente simplifaca**: uma vez que um `objeto` é um `tipo abstrato de dado`, geralmente heterogêneo, quando ele é criado é buscado na memória um espaço suficiente para armazenar todos os atributos. O mesmo acontece com os métodos, ou seja, são armazenados. Se você quiser uma explicação mais acurada basta clicar nesse linke também nesse aqui.

Antes de continuarmos é necessário um breve aprofundamento sobre a criação de objetos. Perceba que, para um novo objeto ser criado, foi necessária a utilização da palavra reservada `new`. O que acontece então se no código estiver escrito assim?

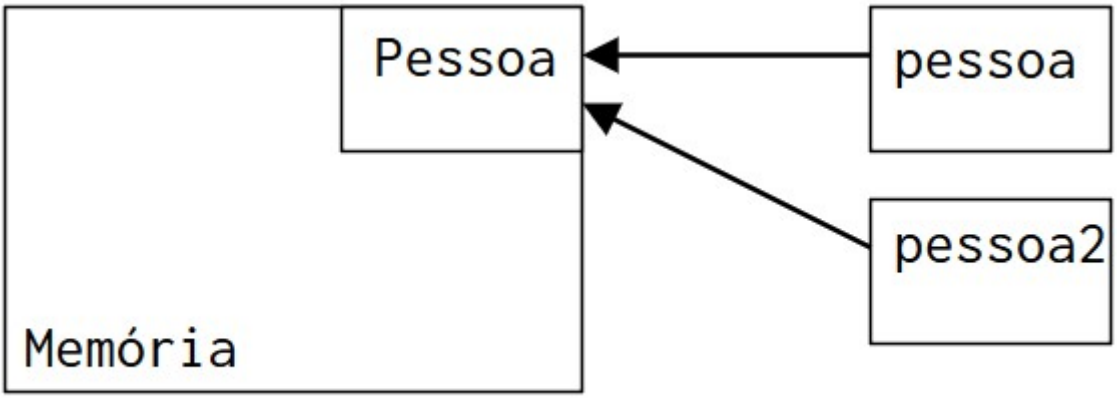
```
Pessoa pessoa2;
```

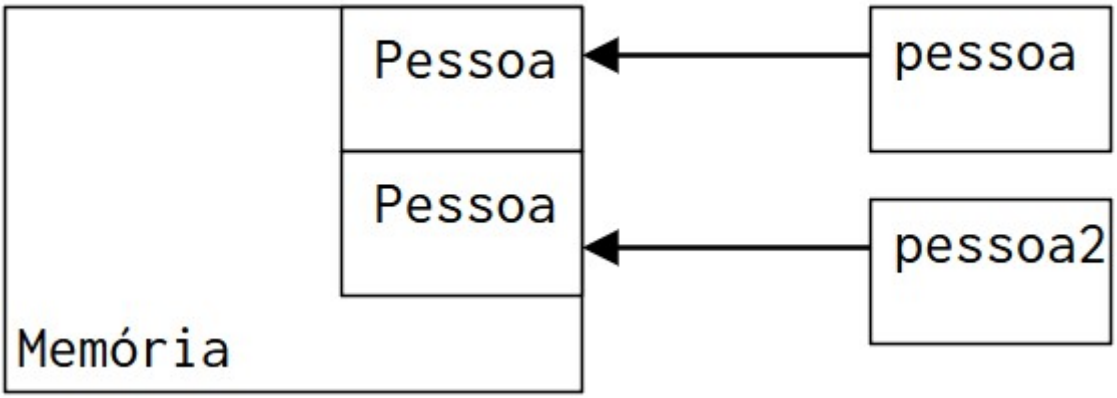
Nesse caso temos apenas uma referência para um **possível** objeto. Nenhum objeto foi criado. Agora, o que acontece se estiver escrito assim?

```
Pessoa pessoa = new Pessoa();
Pessoa pessoa2 = pessoa;
```

Neste caso apenas **um** `objeto` foi criado, porém nós temos duas referência para ele. Ou seja, podemos acessar aquele `objeto` tanto através de `pessoa` quanto através de `pessoa2`. E se eu quiser dois objetos, como fazer? Então teremos de criar dois objetos de fato:

```
Pessoa pessoa = new Pessoa();
Pessoa pessoa2 = new Pessoa();
```





Interação entre Objetos

Os `objetos` podem interagir entre si, inclusive um fazendo parte do outro. Anteriormente já havíamos criado e `classe` Pessoa. Agora vamos criar a `classe` ContaBancaria:

```
public class ContaBancaria{
    Pessoa titular;
    String codigoBanco;
    String codigoConta;
    double saldo;

    public Pessoa getTitular(){
        return titular;
    }

    public void setTitular(Pessoa titular){
        this.titular = titular;
    }

    public String getCodigoBanco(){
```

```
        return codigoBanco;
    }

    public void setCodigoBanco(String codigoBanco){
        this.codigoBanco = codigoBanco;
    }

    public String getCodigoConta(){
        return codigoConta;
    }

    public void setCodigoConta(String codigoConta){
        this.codigoConta = codigoConta;
    }

    public double getSaldo(){
        return saldo;
    }

    public void sacar(Double valor){
        this.saldo -= valor;
    }

    public void depositar(Doube valor){
        this.saldo += valor;
    }
}
```

Perceba que a construção da `classe` ContaBancaria é bem parecida com a da `classe` Pessoa, ou seja, inciamos declarando os `atributos` e depois os `métodos`. Entretanto nós temos um `atributo` do tipo Pessoa. Como isso é possível?

Isso só é possível se tivermos feito a `classe` Pessoa. Como ela já existe, então podemos dizer que a `classe` ContaBancaria deve ter um `objeto` do tipo Pessoa entre seus atributos.

Além desse novo `atributo` você deve ter percebido que não foi criado o método `setSaldo`. Isso se deu porque o `atributo` `saldo` pode ser manipulado de duas formas, seja através de um saque, seja através de um depósito. Cada `método` vai modificar o `saldo` de uma forma específica. Ou seja, é como se o `setSaldo` tivesse sido dividido em dois. A partir disso você deve entender que, apesar de ser padrão a existência de `getters` e `setters`, eles não são obrigatórios.

Voltemos agora ao `atributo titular`. O método `getTitular` informa que terá como retorno uma Pessoa. Como Pessoa é um `objeto`, então o método deve retornar um `objeto` do tipo Pessoa. Dentro da `classe` ContaBancaria esse `objeto` é referenciado a partir do `atributo titular`. Lembra quando tivemos `pessoa` e `pessoa2` como duas referências para o mesmo objeto? Com o `titular` nós temos o mesmo caso.

Supondo que já havíamos criado um `objeto` do tipo Pessoa, chamado `pessoa`, e que também tenhamos criado um `objeto` do tipo ContaBancaria chamada `conta`. Como podemos vincular o `objeto` do tipo Pessoa ao `objeto` do tipo ContaBancaria? Isso acontece através dos métodos:

```
Pessoa pessoa = new Pessoa();
ContaBancaria conta = new ContaBancaria();

conta.setTitular(pessoa);
```

Lembre que na `classe` ContaBancaria, o `método setTitular` exige por parâmetro algum `objeto` do tipo Pessoa. A partir disso o `atributo titular` pode ser vinculado àquele `objeto`. Foi exatamente isso que acabamos de fazer no exemplo acima. O `objeto pessoa` já existia e o passamos por `parâmetro` ao `método` do `objeto conta`.

Construtores e Destrutores

Até então vimos como criar um novo objeto. Por exemplo:

```
Pessoa pessoa = new Pessoa();
```

Assim que criamos um objeto, qual são os valores dos atributos?

São os chamados valores padrão ou `default`. Por exemplo, para tipos numéricos é o valor `0`. Mas e se eu quiser criar um objeto com valores que eu tenha especificado? Por exemplo, e se eu quiser criar uma pessoa já dando o nome para ela?

Neste caso nós temos de usar um método chamado `construtor`. Toda `classe` que é criada em `Java` já possui um `construtor` padrão implícito. Quando escrevemos `new`

`Pessoa()` estamos utilizando o `construtor` padrão da `classe` Pessoa. Entretanto é possível acrescentarmos outros `construtores` . Exemplo:

```
public class Pessoa{
    String nome;
    double peso;
    int idade;
    double altura;
    String cpf;

    public Pessoa(){
    }

    public Pessoa(String nome){
        this.nome = nome;
    }

    public Pessoa(String nome, double peso){
        this.nome = nome;
        this.peso = peso;
    }

    public Pessoa(String nome, double peso, int idade, double altura, String cpf){
        this.nome = nome;
        this.peso = peso;
        this.idade = idade;
        this.altura = altura;
        this.cpf = cpf;
    }

    // getters e setters
}
```

No exemplo acima fizemos **quatro** construtores para a `classe` Pessoa. Observe que o primeiro não tem qualquer parâmetro e não faz nada. Por que ele está ali?

Esse é o `construtor` padrão. Se você não criar qualquer `construtor` ele vai existir implicitamente. Porém a partir do momento em que você declara algum `construtor` ele deixa de existir. Logo, se você quiser que ele continue existindo, terá de declará-lo explicitamente.

Outra coisa que você deve ter notado: todos os métodos `construtores` têm o **mesmo nome**. Essa é uma `regra sintática` do `Java`, ou seja, é obrigatório que os `construtores` tenham o mesmo nome da `classe`. Além disso, você pode criar quantos `construtores` você quiser, desde que cada um tenha um conjunto de `parâmetros` diferentes. Ou seja, o seguinte exemplo:

```
public Pessoa(double peso){
    this.peso = peso;
        }

public Pessoa(double altura){
    this.altura = altura;
        }
```

Não vai funcionar, pois para o `compilador` é como se fosse o mesmo `construtor` só que duplicado. Lembre que por mais que nós consigamos diferenciar as palavras *peso* de *altura*, o `compilador` só enxerga *variável do tipo double*, portanto, para ele acaba sendo o mesmo `construtor`.

E quanto ao `destrutor`? Ele é o contrário do `construtor`. Ou seja, enquanto o método `construtor` instancia um `objeto` de uma `classe`, o `destrutor` encerrará ou destruirá esse `objeto`. Em `Java` o `destrutor` funciona automaticamente. Quando algum `objeto` já não está sendo mais utilizado entra em cena o `coletor de lixo` ou `garbage collector`, que destroi os objetos inúteis e libera memória para a continuação do programa.

Exercícios

Adaptado de :

Victor Perin - Lista 2

Victor Perin - Lista 3

1. Implementar uma classe Caneta que deve possuir como características **marca**, **cor** e **tamanho**. Nesta classe devem ser implementados os métodos construtores, *getters* e *setters*. Em outra classe, chamada CanetaTeste (onde deve estar o

método **main**), deverá ser criado um objeto do tipo Caneta, atribuir valores e exibir os dados deste objeto.

2. Implementar uma classe Lampada que deve possuir como características o **tipo** (led, fluorescente, etc.), **voltagem**, **cor**, **marca**, **preço**, **potência** e **status** (*boolean*, verdadeiro para aceso e falso para apagado). Em outra classe chamada LampadaTeste devem ser criados dois objetos do tipo Lampada, atribuir valores e exibir os dados deste objeto. O programa deverá informar também qual das duas lâmpadas possui maior potência e também qual é a mais cara.
3. Implementar uma classe Apolice com os seguintes atributos: **nome do segurado**, **idade do segurado** e **valor do prêmio da apólice**. Nesta classe Apolice devem ser implementados os métodos:
    - imprimir() - este método não retorna valor e deverá mostrar na tela todos os atributos da classe Apolice.
    - calcularPremioApolice() - este método não retorna valor e deverá calcular o valor do prêmio seguinto as seguintes regras:
        - Caso a idade seja maior ou igual a 18 e menor ou igual a 25 anos, use a fórmula: valorPremio += (valorPremio*20)/100.
        - Quando a idade for superior a 25 e menor ou igual a 36 anos, use a fórmula: valorPremio += (valorPremio*15)/100.
        - Quando a idade for superior a 36 anos use a fórmula: valorPremio += (valorPremio*10)/100.
    - oferecerDesconto() - este método não retorna valor, mas recebe o parâmetro cidade, que irá conter o nome da cidade para o cálculo do desconto. Caso a cidade seja Teresina, dê um desconto no valor do prêmio de 20%. Caso seja Timon, dê um desconto de 15%. Caso seja Fortaleza dê um desconto de 10% e, se for São Luiz, dè um desconto no valor do prêmio de 5%.

   Logo após implementar a classe Apolice, implemente em uma classe chamada ApoliceTeste uma sequência de instruções para testar as funcionalidades da classe Apolice.
4. Implementar uma classe chamada CanetaTesteVetor, onde será criado um vetor para armazenar no máximo 50 objetos do tipo Caneta. O programa deverá exibir o seguinte menu para o usuário:
    - 1 - Cadastrar caneta;
    - 2 - Exibir todas as canetas;
    - 3 - Exibir a quantidade de canetas cadastradas.

    - 4 - Consultar a quantidade de canetas de uma determinada cor (escolhida pelo usuário).
    - 0 - Sair.
5. Implementar uma classe chamada LampadaTesteVetor para armazenar 30 objetos do tipo Lampada. O programa deverá exibir o seguinte menu para o usuário:
    - 1 - Cadastrar lâmpada.
    - 2 - Exibir todas as lâmpadas.
    - 3 - Exibir a quantidade de lâmpadas cadastradas.
    - 4 - Consultar a quantidade de lâmpadas de uma determinada potência (escolhida pelo usuário).
    - 5 - Exibir os dados das lâmpadas com preço menor do que o preço médio das lâmpadas cadastradas.
    - 6 - Exibir a quantidade de lâmpadas acesas e apagadas.
    - 0 - Sair.
6. Implementar uma classe Data que deve possuir como atributos: **dia**, **mês** e **ano** (todos do tipo inteiro). Nesta classe, além dos métodos construtores, *getters* e *setters*, deve constar o método **validarData(int, int, int): boolean** que deverá verificar se as informações passadsas por parâmetros são verdadeiras ou não. Este método deverá inclusive verificar se é ano bissexto. Por fim deve ser implementado o método **toString** que deverá retornar uma String no seguinte formato "dia/mês/ano", por exemplo: "01/setembro/2023".

Os próximos exercícios foram traduzidos e adaptados de w3resource

1. Escreva um programa com a classe Pessoa com os atributos **nome** e **idade**. Crie duas instâncias da classe Pessoa, forneça os valores de seus atributos através de um construtor e imprima seus nomes e idades.
2. Escreva um progama com a classe Cachorro com os atributos **nome** e **raça**. Crie duas instâncias da classe Cachorro utilizando o construtor e modifique os atributos usando o método *set* e depois imprima os valores atualizados.
3. Escreva um programa em Java com umas classe chamada **Retangulo** com os atributos **largura** e **altura**. Calcule a área e o perímetro do retângulo.
4. Escreva um programa com a classe Circulo com o atributo **raio**, que pode ser modificado. Calcule a área e a circunferência do círculo.
5. Escreva um programa com uma classe chamada Livro, com os atributos **titulo**, **autor** e **ISBN**, com métodos para adicionar e remover livros de um *array*.

6. Escreva um programa com a classe Empregado, com os atributos **nome**, **cargo** e **salário**. Deve haver também um método para atualizar o salário.
7. Escreva um programa com a classe Banco que possua uma *array* de contas, e métodos para adicionar e remover contas, além de depositar e sacar dinheiro. Crie também a classe Conta para armazenar os detalhes do cliente da conta.
8. Crie uma classe chamada Semaforo com os atributos **cor** e **duração**, e métodos para modificar a cor e verificar se o sinal está verde ou vermelho.
9. Escreva um programa com uma classe chamada Empregado, com os atributos **nome**, **salário** e **data de admissão**. Deve haver um método para calcular há quantos anos o empregado está na empresa.
10. Escreva um programa com a classe Aluno com os atributos **nome**, **notas** e **cursos**. Deve haver métodos para acrescentar e remover cursos.
11. Escreva uma classe chamada Biblioteca, que terá uma coleção de livros e métodos para adicionar e remover livros.
12. Escreva um programa com a classe Avião com os atributos **número do voo**, **destino**, **horário de saída**, e métodos para verificar o status do voo (no horário ou atrasado).
13. Escreva um programa com a classe Inventário, com uma coleção de produtos e métodos para adicionar e remover produtor, e verificar se o inventário está com poucos produtos.
14. Escreva uma classe Escola com atributos para alunos, professores e disciplinas, e métodos para adicionar e removê-los.
15. Crie uma classe Album com um conjunto de músicas e métodos para adicionar e remover músicas, e tocar alguma música aleatória.
16. Escreva uma classe Filme, com atributos para título, diretor, atores e avaliações, com métodos para adicionar e remover itens e calcular a avaliação média.
17. Escreva uma classe Restaturante com atributos para itens no menu, preços e avaliações, com métodos para adicionar e remover itens e calcular a média de avaliação.