Criação de classes

Prof. Ítalo Assis

Ajude a melhorar este material =]

Encontrou um erro? Tem uma sugestão?

Envie e-mail para italo.assis@ufersa.edu.br

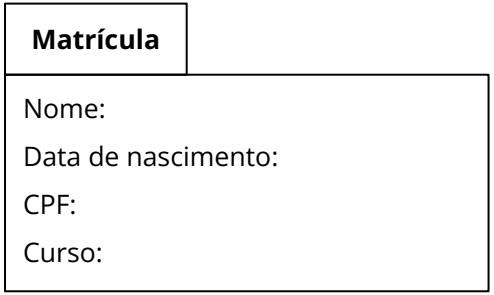
Agenda

- Classes, atributos e métodos
- Objetos e construtores
- Escopo e encapsulamento
- Sobrecarga de métodos
- Atributos e métodos estáticos
- Classes e métodos genéricos

Classes, atributos e métodos

Classes

- Programadores POO criam e usam objetos a partir de classes
- Classes são estruturas que contém, para determinado modelo, os dados que devem ser representados e as operações que devem ser efetuadas com estes dados
- Exemplos de classes: *Pessoa*, *Imovel*, *Produto*...



Classes

- Uma classe em Java é declarada com a palavra-chave *class* seguida do nome da classe
  - *class NomeDaClasse*
- O nome da classe:
  - não pode conter espaços
  - deve sempre ser iniciado por uma letra
  - não deve conter acentos
  - pode conter números
  - não podem ser exatamente iguais às palavras reservadas de Java
- Tradicionalmente os nomes de classes possuem caracteres maiúsculos no ínicio da cada palavra

Classes

- O conteúdo das classes é delimitado pelas chaves (caracteres { e })
  - todos os campos e operações da classe devem estar entre estes caracteres

```
class CadastroDeFuncionariosDeSupermercado {
    // aqui virão os dados e operações
}
```

- Cada arquivo deve conter apenas uma classe e ter o mesmo nome dessa classe
- **Prática:** Crie um arquivo que defina uma classe para representar uma data

Atributos

- Os dados contidos em uma classe são conhecidos como campos ou atributos daquela classe
- Para declarar um atributo em uma classe basta declarar o tipo de dado, seguido do nome do atributo
- O tipo deve ser:
  - nativo da linguagem Java (*int*, *float*...)
  - uma classe existente na linguagem (*String*, *Scanner*...)
  - ou uma classe definida pelo programador (*Data, Pessoa...)*

Atributos

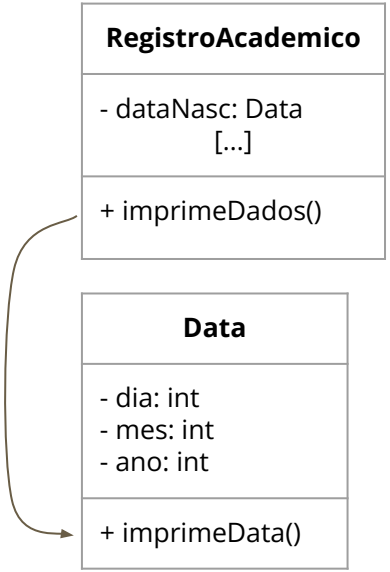
- Tradicionalmente os nomes de classes possuem caracteres maiúsculos no ínicio da cada palavra exceto a primeira

```
class CadastroDeFuncionariosDeSupermercado {
    String nome;
    int matricula;
    float salario;
    Data dataDeNascimento;
    // aqui virão as operações
}
```

- **Prática:** Adicione atributos à classe que representa uma data

Métodos

- As operações contidas em uma classe são chamadas de métodos desta classe
- O processamento de dados que deve ser feito para um modelo será feito dentro dos métodos, que eventualmente poderão chamar outros métodos, da mesma classe a que pertencem ou de outras classes
- Muitos autores preferem usar o termo **troca de mensagens** para se referir à chamada de métodos
- Métodos podem opcionalmente receber argumentos e retornar um valor



Métodos

- Métodos não podem ser criados dentro de outros métodos, nem fora da classe a qual pertencem
- Podemos declarar um método da seguinte forma:
  - *tipoOuClasseDeRetorno nomeDoMetodo (listaDeArgumentos)*
    - Nomes de métodos seguem as mesmas regras de nomes de campos
    - O tipo do retorno ***void*** indica que o método não retorna nada
    - Métodos que retornam um valor diferente de *void* devem ter a palavra-chave ***return*** seguida de constante/variável do tipo/classe do retorno do método

Métodos

```
class CadastroDeFuncionariosDeSupermercado {
        String nome;
        int matricula;
        float salario;
    Data dataDeNascimento;

        void mostraDados() {
    System.out.println("Nome: "+ nome);
  System.out.println("Matricula: "+ matricula);
    System.out.println("Salario: "+ salario);
                                    }
}
```

- **Prática:** Adicione os métodos, *dataEhValida, inicializaData* e *mostraData* à classe que representa uma data

Classe executável

- Para que uma classe seja executável, ela deve conter um método com a seguinte assinatura:
  - *public static void main(String[] args) { }*
- Os demais métodos são chamados a partir do método *main*
- É comum ter uma classe apenas para conter o método *main*
- **Prática:** crie um arquivo com uma classe executável na mesma pasta da classe que representa uma data
  - Neste ponto, a execução deve apenas imprimir uma frase de boas vindas

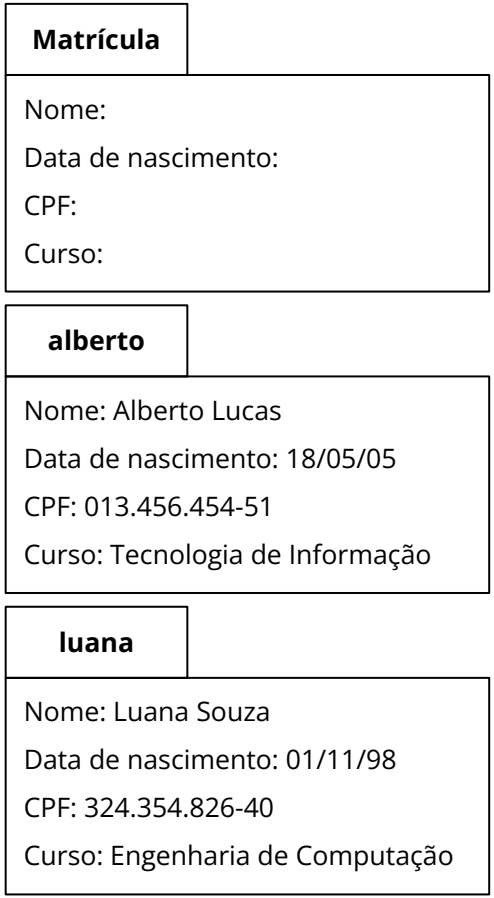
HelloWorld.java

```
public class HelloWorld {

  public static void main(String[] args) {
    System.out.println(“Hello World!”);
  }

}
```

Objetos e construtores

Objetos e referências

- Para representação de dados específicos usando classes é necessária a criação de objetos (ou instâncias) desta classe
- São uma **materialização da classe** e, assim, podem ser usados para representar dados e executar operações
- Para que possam ser manipulados, é necessária a criação de **referências** a estes objetos, que são basicamente variáveis do “tipo” da classe
- Cada instância de uma classe possui os mesmos atributos da classe



Prática

- Modele uma classe que represente um aluno em uma disciplina da UFERSA. Dê um exemplo de objeto dessa classe.

[classe] Aluno
- nome: texto - matricula: texto - nota1: real - nota2: real - nota3: real - notaFinal: real
+ obterNome(): texto + obterMatricula(): texto - calcularNotaFinal(): vazio + informarNotas(real, real, real): vazio + obterNotaFinal(): real

[objeto da classe Aluno] aluno01
- nome: Milton Nascimento - matricula: 16845379 - nota1: 7.5 - nota2: 8.5 - nota3: 9.5 - notaFinal: 8.5
+ obterNome(): texto + obterMatricula(): texto - calcularNotaFinal(): vazio + informarNotas(real, real, real): vazio + obterNotaFinal(): real

Criação de referências

- Similar a criação de variáveis de tipos primitivos:
  - *NomeDaClasse nomeDoObjeto;*
- Exemplo:
  - *Matricula mat;*
    - Cria uma variável do tipo *Matricula*
    - *mat* é uma referência para um objeto da classe *Matricula*
    - Declara a referência mas não cria o objeto

Operador *new*

- Em Java, objetos são criados usando o operador *new*
  - *Matricula mat = new Matricula();*
    OU
  - *Matricula mat;*
    mat = new Matricula();
- O operador *new* cria uma instância da classe e retorna a referência do novo objeto.
  - Aloca memória para o novo objeto;
  - Chama um método especial de inicialização da classe denominado **construtor**;
  - Retorna a referência para o novo objeto.

Operador *new*

- Detalhando:
  - *Matricula mat;*
    - Como apenas com a declaração da variável o objeto ainda não existe, o conteúdo inicial dela será o valor nulo (*null*)
    - O *null* é um valor válido em Java e permite comparações como: *if (cont == null)*
  - *mat = new Matricula();*
    - O objeto é criado na inicialização. A partir disso, é possível acessar os atributos e métodos do objeto

Prática

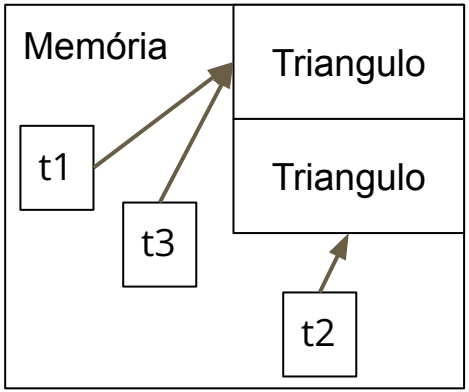
- Escreva um classe *Triangulo* com atributos para representar seus lados e uma descrição textual
- A classe deve conter também um método para inicializar os seus atributos e um método para calcular o perímetro do triângulo
- Crie uma classe executável de teste que:
  - faça a declaração das variáveis t1, t2 e t3 do tipo *Triangulo*;
  - crie t1 e t2 com o operador *new* e os inicialize com os mesmos parâmetros
  - faça t3 = t1
- t1 == t2? t1 == t3? Escreva o código para testar.

Prática

TestaTriangulo.java

```
public class TestaTriangulo {

  public static void main(String[] args) {
    Triangulo t1, t2, t3;
    t1 = new Triangulo();
    t2 = new Triangulo();
    t3 = t1;
    System.out.println(t1 == t2);
    System.out.println(t1 == t3);
  }
}
```

false
true



- Deve-se criar métodos (ou utilizar um existente) para compara objetos
  - Para comparar objetos do tipo *String*, podemos utilizar o método *equals*:
    - *str1.equals(str2)*

Utilizando objetos

- Uma vez um objeto tendo sido criado, seus métodos e atributos públicos podem ser acessados utilizando sua referência através do operador ponto:
  - <identificador>.<método>
  - <identificador>.<atributo>
- Exemplos:
  - *t1.calculaPerimetro()*
  - *t1.desc*

Prática

- Escreva a classe *RegistroAcademico* com os atributos *nome*, *matricula*, *codigoCurso* e *percentualDeCobranca*, além dos métodos *inicializaRegistroAcademico* e *calculaMensalidade*
  - A mensalidade é igual a 100 x *codigoCurso* x *percentualDeCobranca*
- Crie a classe executável *DemoRegistroAcademico* onde o objeto *michael*, do tipo *RegistroAcademico*, é criado e seus atributos são inicializados com o método *inicializaRegistroAcademico*. Por fim, calcule e informe o valor da mensalidade de michael.

Prática

- O que acontece se criarmos outro objeto, *roberto*, da classe *RegistroAcademico* e, em seguida, executarmos o método *calculaMensalidade*?

- A classe *RegistroAcademico* permite que seus métodos sejam executados sem garantir a inicialização dos atributos. Esse fator pode gerar problemas como no exemplo acima.

Caso os campos de uma instância não sejam inicializados, os seguintes valores serão adotados:

boolean	*false*
char	[espaço]
inteiros e pontos flutuantes	zero
instâncias de classes	*null*

Construtores

- **Construtores** são métodos especiais chamados automaticamente quando instâncias são criadas através da palavra-chave **new**
- São executados antes de qualquer outro método
- Devem ter exatamente o mesmo nome da classe
- Devem ser declarados sem tipo de retorno
- Sintaxe:
  - *NomeDaClasse(argumentos) { ... }*

Prática

- Crie uma classe *EventoAcademico* com os atributos *nomeDoEvento*, *localDoEvento* e *numeroDeParticipantes*
- A classe deve também conter um método construtor e o método *mostraEvento*

A palavra-chave *this*

- Existe, internamente para cada instância, uma “auto-referência”, ou seja, uma referência à própria instância
- Esta referência é representada pela palavra-chave *this*
- Dessa forma, podemos explicitar que estamos acessando um atributo ou método da própria instância

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

- **Prática:** No exemplo anterior, nomeie os argumentos do construtor com os mesmos nomes dos atributos da classe. Utilize a palavra-chave *this* para evidenciar o uso dos atributos da classe.

Escopo e encapsulamento

Escopo

- O escopo dos campos e variáveis dentro de uma classe determina a sua visibilidade
- Variáveis declaradas dentro de métodos possuem escopo igual ao de um programa estruturado
  - O escopo é delimitado pelo bloco ({...}) no qual a variável se encontra
  - A declaração deve ser feita antes do uso do variável
- Os atributos da classe são visíveis por toda a classe
  - Não importa em que ponto estão declarados
  - É recomendado declarar no começo da classe

Contador.java

```
public class Contador {
  int valor;

  void setValor(int valor) {
    int inicial = 20;
    valor = inicial + valor + correcao ;
  }

  int correcao = -5;
}
```

Encapsulamento

- Uma das principais vantagens do paradigma de orientação a objetos é a possibilidade de encapsular campos e métodos capazes de manipular estes campos em uma classe
- À propriedade de proteger a estrutura interna de uma classe escondendo-a de observadores externos dá-se o nome de **encapsulamento**
- É desejável que os campos das classes sejam ocultos ou escondidos dos programadores usuários das classes, para evitar que os dados sejam manipulados diretamente
- Exemplo: A classe *Data* que criamos permite que seja definida uma data inválida através do acesso direto aos atributos

Modificadores de acesso

- Permitem a restrição ao acesso a campos e métodos em classes
- São declarados dentro das classes, antes dos métodos e campos
  - modificador-de-acesso tipo-ou-classe nome-do-campo;
  - modificador-de-acesso tipo-ou-classe-de-retorno nome-do-metodo(lista-de-argumentos);
- Existem quatro modificadores de acesso:
  - ***public***: garante que o campo ou método poderá ser acessado a partir de **qualquer classe**
  - ***private***: campos e métodos só podem ser acessados por métodos da **mesma classe**
  - ***protected***: similar ao *private* exceto que **classes herdeiras** também terão acesso ao campo ou método
  - ***package*** ou *friendly*: campos e métodos declarados sem modificadores. Esses campos e métodos serão visíveis para todas as classes pertencentes a um **mesmo pacote**

Encapsulamento

- Ao criar classes, o programador de classes deve implementar uma política de acesso a dados e métodos internos
- Algumas regras básicas:
  - Todos os campos de uma classe devem ser declarados com o modificador *private* (ou *protected*)
  - Métodos que devem ser acessíveis devem ser declarados explicitamente com o modificador *public*
  - Métodos que permitam a manipulação controlada dos valores dos campos devem ser escritos e utilizar o modificador *public*
  - Métodos auxiliares podem ser declarados com o modificador *private*. Esses métodos poderão ser executados por outros métodos dentro da mesma classe

Prática

- Vamos revisar as classes que criamos nesta aula e adicionar os modificadores de acesso adequados
  - As classes criadas foram:
    - *Data*
    - *Triangulo*
    - *RegistroAcademico*
    - *EventoAcademico*

Métodos *get* e *set*

- Se o atributo fosse *public*, qualquer cliente da classe poderia ver e fazer o que quisesse com os dados
- *set*: usado para validar tentativas de modificações nos dados *private*
  - pode ser programado rejeitar qualquer tentativa de definir os dados como valores inválidos
    - temperatura corporal negativa
    - dia em março fora do intervalo de 1 a 31
    - código de produto que não está no catálogo da empresa
- *get*: usado para controlar como os dados são apresentados para o chamador
  - pode apresentar os dados de uma forma diferente
    - uma classe *Avaliacao* pode armazenar uma nota como um *int* entre 0 e 100, mas um método *getNota* pode retornar uma classificação como uma *String*, por exemplo, "A" para as notas entre 90 e 100, "B" para as notas entre 80 e 89...

Prática

- Crie uma classe para representar uma pessoa, com os atributos privados de nome e altura. Além do construtor, crie os métodos de acesso, *set*'s e *get*'s e também um método para retornar uma representação textual (*toString*) dos dados de uma pessoa.

Sobrecarga de métodos

Sobrecarga de métodos

- Em algumas ocasiões pode ser útil executar um método em uma classe passando argumentos de diferentes tipos e/ou em diferentes quantidades
- O Java, entre outras linguagens, permite a criação de métodos com nomes iguais, porém com assinaturas diferentes
- Técnica comumente utilizada para construtores

Soma.java

```
public class Soma {

  public int Soma(int a, int b) {
    return a+b;
  }

  public float Soma(float a, float b) {
    return a+b;
  }

}
```

Prática

- Crie a classe *ContaBancaria* com os atributos nome, saldo e o status da conta (especial ou não) e um método para imprimir os valores de seus atributos.
  Na maioria dos casos, contas serão abertas com saldo zerado e não serão contas especiais - se o construtor exige que os argumentos sejam passados, teremos que especificá-los todas as vezes que formos criar instâncias para esta classe. Por isso, crie duas versões do construtor:
  - uma para a qual se deve passar somente o nome do correntista, sendo que os outros atributos devem receber valores pré-determinados;
  - e outra versão para a qual se devem passar todos os dados.
- Por fim, crie uma classe executável que demonstre o uso da classe *ContaBancaria*

Atributos e métodos estáticos

Atributos estáticos

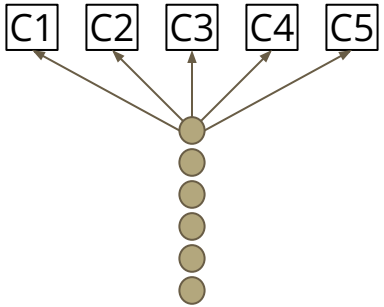
- Compartilhados por todas as instâncias dessa classe
- Somente um valor é armazenado em um campo estático
  - Caso esse valor seja modificado por uma instância, a modificação será refletida às outras
- Utilidades:
  - manter uma informação que possa ser modificada ou acessada por qualquer instância
  - armazenar valores que não serão modificados nem serão variáveis por instâncias de classe
- São declarados com o modificador *static*
  - *public static int qtd;*
- Podem ser acessados através da própria classe:
  - *NomeDaClasse.atributoEstatico*

Métodos estáticos

- Métodos estáticos podem ser chamados sem a necessidade de criação de instâncias das classes às quais pertencem
- Adequados para implementar rotinas que sejam independentes de dados armazenados nas instâncias da classe
- São declarados com o modificador *static* antes do tipo de retorno do método
  - *public static void main(String[] args) { ... }*
- Podem ser acessados através da própria classe:
  - *NomeDaClasse.metodoEstatico()*
- Métodos e atributos que pertencem à mesma classe de um método estático, e são utilizados por este, devem ser estáticos

Prática

- Vamos escrever um programa de gerenciamento de uma fila de banco com clientes sendo atendidos em 5 caixas.
- Para isso, vamos criar a classe *CaixaBanco* com os atributos *clientesAtendidos* e *numeroCaixa*, além do método *iniciaAtendimento*.



Prática

- Crie a classe *Circulo* que possui como atributos o raio e um valor aproximado de π
- A classe também deve ter um método para calcular o perímetro do círculo
- Por fim, crie uma classe executável para demonstrar o uso de *Circulo*

Prática

- Uma das aplicações mais frequentes de métodos estáticos é a criação de **bibliotecas de métodos**: classes que contêm somente métodos estáticos, geralmente agrupados por função.
- Para exemplificar, vamos construir uma classe composta por métodos estáticos que realizem conversões de unidades
  - Polegadas para centímetros
    - *polegadas*2.54*
  - Pés para centímetros
    - *pes*30.48*
  - Milhas para quilômetros
    - *milhas*1.609*

Fábrica de instâncias

- Métodos estáticos que retornem novas instâncias de classes são conhecidos como **fábricas de instâncias**.
- Fábricas de instâncias são úteis para a criação simples e rápida de instâncias que sejam bem características de uma classe.

Prática

- Vamos desenvolver um novo método para a classe que representa uma data. Este método deve receber um ano e retornar uma instância de uma data equivalente ao Natal daquele ano

Classes e métodos genéricos

Métodos genéricos

- Métodos sobrecarregados são frequentemente utilizados para realizar operações semelhantes em tipos diferentes de dados
- Podemos substituir os tipos em cada destaque nos métodos por um único tipo genérico e declarar um método para somar números *Integer* ou *Double*
- Nesse caso, os métodos sobrecarregados podem ser codificados mais compactamente com um **método genérico**

Soma.java

```
public class Soma {

  public Integer Soma(Integer a, Integer b) {
    Integer res = a+b;
    return res;
  }

  public Double Soma(Double a, Double b) {
    Double res = a+b;
    return a+b;
  }

}
```

Métodos genéricos

- Métodos genéricos têm uma **seção de parâmetros de tipo** que precede o tipo de retorno do método
  - *public static* ***<T>*** *T maximo(T valor1, T valor2)*
  - Contém um ou mais parâmetros de tipo, separados por vírgulas
  - Podem ser utilizados para declarar o tipo de retorno, tipos de parâmetro e tipos de variáveis locais
  - Atuam como marcadores de lugar para os tipos dos argumentos passados
  - Podem representar somente tipos por referência (não tipos primitivos como *int* e *double*)

Prática

- Escreva uma versão genérica simples do método *ehIgual* que compara seus dois argumentos com o método *equals* e retorna *true* se forem iguais e *false* caso contrário
- Utilize esse método genérico em um programa que chama *ehIgual* com uma variedade de tipos predefinidos (*Double*, *String*, *Integer*...).

Classes genéricas

- Também conhecidas como classes parametrizadas ou tipos parametrizados
- Algumas classes podem ser entendidas independentemente do tipo de elemento que elas manipulam
  - Estruturas de dados, como uma pilha, são exemplos
- **Classes genéricas** fornecem um meio de descrever o conceito de uma classe de uma maneira independente do tipo
- Podemos então instanciar objetos específicos de tipo da classe genérica
- Genéricos fornecem uma boa oportunidade para reutilização de software

Classes genéricas

- Uma vez que há uma classe genérica, você pode indicar o(s) tipo(s) que deve(m) ser utilizado(s) no lugar do(s) parâmetro(s) de tipo da classe
  - *public class Pilha<T> {*

    ...

    public void push(T valor) {

    elementos.add(valor);

    }

    }
  - *Pilha<Double> doubleStack = new Pilha<>();*

    Pilha<Integer> integerStack = new Pilha<>();

Prática

- Escreva uma classe genérica *Tupla* que tem dois parâmetros de tipo, *F* e *S*, representando o tipo do primeiro e segundo elemento do par, respectivamente
- Adicione métodos *get* e *set* para cada elemento do par

Os códigos relacionados a esta aula estão disponíveis em

https://github.com/italoaug/Programacao-Orientada-a-Objetos/tree/main/codigos/classe

Referências

BATISTA, Rogério da Silva; MORAES, Rafael Araújo de. **Introdução à Programação Orientada a Objetos**. 2013. Disponível em: http://proedu.rnp.br/handle/123456789/611. Acesso em: 18 ago. 2021.

SANTOS, R. **Introdução à programação orientada a objetos usando JAVA**. 2. ed. Rio de Janeiro: Campus, 2013. 336p.

DEITEL, Paul; DEITEL, Harvey. **Java: como programar**. 10. ed. São Paulo: Pearson Education do Brasil, 2017.

BACALÁ JÚNIOR, Sílvio. **Revisão de POO em Java: lista de exercícios 1**. 2022. Disponível em: http://www.facom.ufu.br/~bacala/POO/lista1.pdf. Acesso em: 30 mar. 2022.