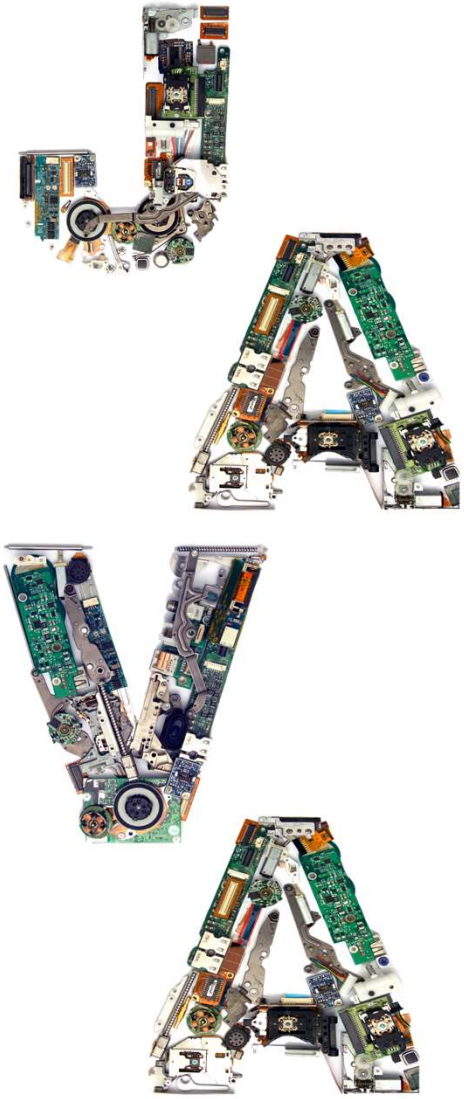
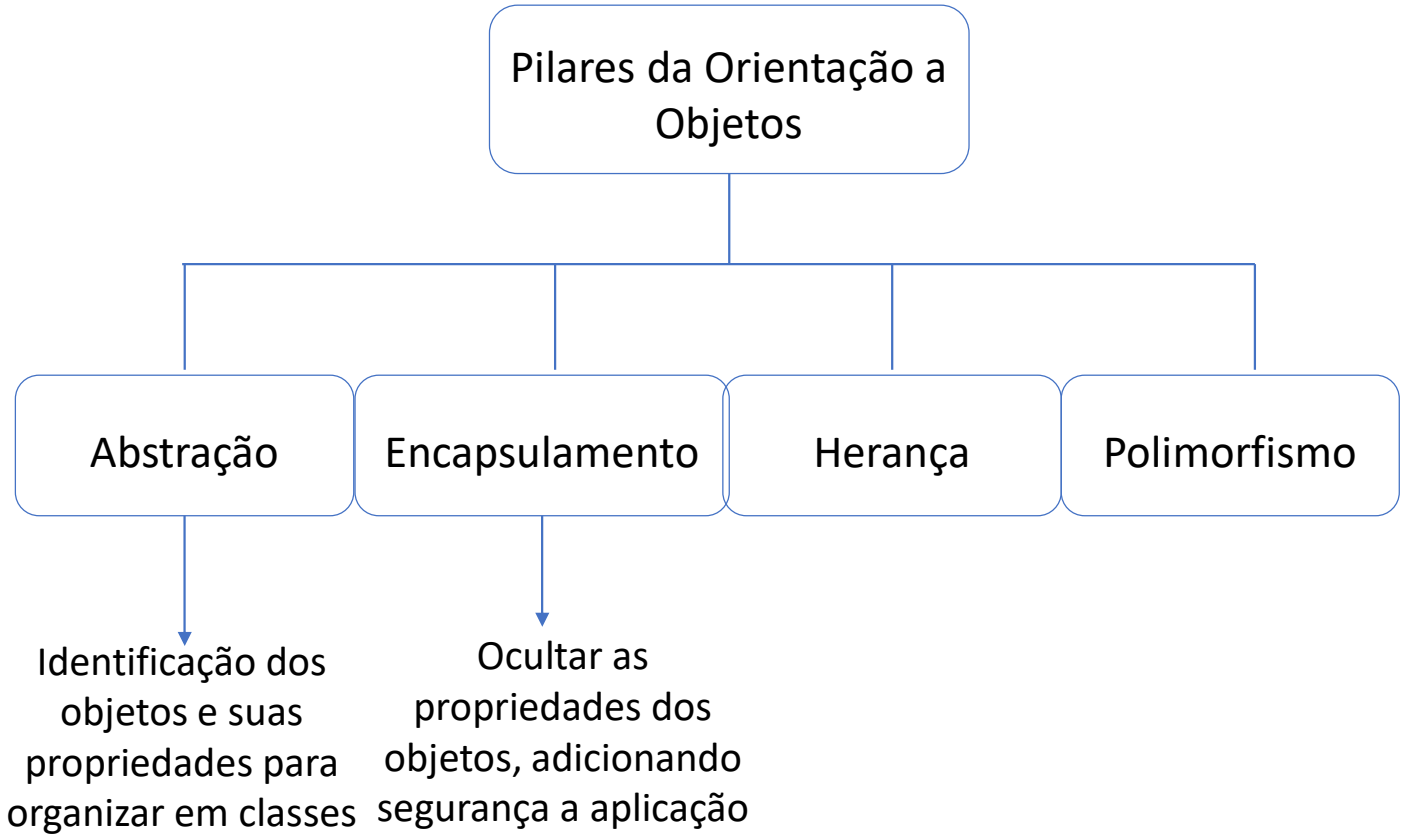
Programação Orientada a Objetos com Java e WEB

Herança



Recapitulando



Recapitulando

Classe

- É um tipo de dado definido pelo desenvolvedor.
- Define as propriedades (atributos) de um objeto.
- Define as operações (comportamento / métodos) de um objeto.

Objeto

- É um elemento representativo de uma classe.
- É gerado a partir de uma classe.
- “Age” de acordo com as propriedades e os métodos definidos na classe.

Introdução

- Herança é um dos mecanismos fundamentais para as linguagens que suportam o paradigma da POO.

Herança é o mecanismo que possibilita a criação de novas classes a partir de uma já existente.

Herança é utilizada como forma de reutilizar os atributos e métodos de classes já definidas.

- Aplicar herança sempre envolve basicamente dois elementos: ***uma superclasse*** (classe mãe) e uma ***subclasse*** (classe filha).

Introdução

- Superclasse é também conhecida como classe ancestral. Apresenta as características genéricas de um conjunto de objetos.

- Subclasse ou classe descendente é aquela que estende a superclasse para incluir suas características.

Classes criadas com o qualificador *final* não podem ser estendidas, ou seja, não podem ser reutilizadas

Introdução

- Quais os benefícios da herança?

  - **Eliminar códigos duplicados.**
  - **Facilidade na manutenção do código. É possível alterar apenas a superclasse e a alteração repercutirá instantaneamente em todas as classes que herdam o seu comportamento. Apenas a superclasse deve ser compilada e redistribuída.**
  - **A herança permite que todas as classes agrupadas sob um supertipo tenham todos os métodos herdáveis do supertipo.**
  - **Aplicar POLIMORFISMO.**

Comando *extends*

- Toda classe criada no Java é estendida a partir da classe ***Object.***
- O comando ***extends*** é utilizado na declaração de uma classe para especificar quem é sua superclasse.
- Caso o comando seja omitido, a classe ***Object*** será assumida como a superclasse da nova classe.
- Sintaxe:

[encapsulamento] [*abstract* | *final*] *class* <subclasse> *extends* <superclasse> {}

ini@fiap.com.br

Comando *extends*

- Exemplo:

```
public class MinhaClasse extends Cliente { }
```

- Neste caso, ***MinhaClasse*** é a subclasse e ***Cliente*** é a superclasse.

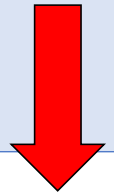
- ***MinhaClasse*** herdará todos os métodos e atributos da classe ***Cliente***, ou seja, herdará todos os membros *public* e não os *private.*

- A classe ***MinhaClasse*** é mais específica em relação à classe ***Cliente.***

Comando *extends*

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

```
public class Aluno extends Pessoa {
    int serie;
}
```



A classe Aluno herda todos os atributos e métodos (não privativos da classe Pessoa.

Construtores em classes estendidas

- Um objeto de uma classe estendida contém variáveis que são herdadas da superclasse, bem como variáveis definidas localmente na classe.
- Para construir um objeto da classe estendida você deve inicializar corretamente o conjunto de variáveis (da subclasse e os herdados da superclasse).
- O construtor da classe estendida lida apenas com as variáveis definidas na classe, e o construtor da superclasse lida com as variáveis herdadas.

Construtores em classes estendidas

- Um construtor da classe estendida pode invocar diretamente um dos construtores da superclasse.
- A referência ***super()*** é utilizada para invocar o construtor da superclasse.

Referência *super*

- Permite que atributos e métodos da superclasse sejam referenciados pelos métodos da subclasse.

- Sintaxe:

  - ***super*.<atributo>;**
  - ***super*.<método>;**

- Caso queira referir-se a um construtor da superclasse, a sintaxe é diferente. Deve ser utilizada apenas a referência seguida de um par de parênteses.
  - ***super*();**

Referência *super* – exemplo

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

```
public class Aluno extends Pessoa {
    int serie;
    public Aluno(String nome, int idade, int serie) {
        super(nome, idade);
        this.serie = serie;
    }
}
```

Sobreposição de métodos – *overriding*

- É a implementação de métodos em subclasses de tal forma que anule o comportamento que ele apresentava em sua superclasse ou apenas acrescente novas instruções.

- Para ocorrer a sobreposição é necessário que o método tenha a mesma assinatura do método da superclasse.

Variáveis de instância não são sobrepostas porque não definem nenhum comportamento;

Métodos definidos como *final* não podem ser sobrepostos

Sobreposição de métodos – *overriding*

- Fazemos uma sobreposição de métodos quando:

  - **Um método da subclasse realize sua tarefa diferente daquela da superclasse.**
  - **Desejamos acrescentar novas instruções à implementação de um método da subclasse.**

Ocorre quando o método herdado apresenta o mesmo nome em relação ao método que está sendo codificado na subclasse

Sobreposição de métodos – *overriding*

```
public class Teste {
    public void imprimir() {
        System.out.println("Teste!!");
    }
}
```

```
public class Novidade extends Teste {
    public void imprimir() {
        System.out.println("Novidade!!");
    }

    public void imprimir(String msg) {
        System.out.println(msg);
    }
}
```

Sobreposição de métodos (métodos devem ter a mesma assinatura)

Sobrecarga de métodos (métodos devem ter tipos ou quantidades de parâmetros diferentes)

Sobreposição de métodos – *overriding*

A sobrecarga não tem relação com a herança e o polimorfismo

- Implementação em uma mesma classe de duas ou mais versões para um mesmo método, de forma que manifestem comportamentos diferentes.

- Não é possível implementar o mesmo método duas vezes com os mesmos tipos de parâmetros (ou a mesma quantidade).

Sobrecarregar um método significa implementar duas ou mais versões do método com tipos de parâmetros distintos

Sobrecarga de métodos – *overload*

Métodos sobrecarregados podem ter retornos diferentes, mas alterar apenas o tipo de retorno não configura a sobrecarga

Sobrecarga se justifica quando se deseja versões alternativas de um método que realiza a mesma tarefa, mas com tipois diferentes de parâmetros.

Sobrecarga de métodos – *overload*

```
public int quadrado(int x, int y) {
    return Math.pow(x, y);
}

public double quadrado(double x, double y) {
    return Math.pow(x, y);
}
```

Sobrecarga de métodos → **dois ou mais métodos com o mesmo nome desde de que haja diferença na lista de parâmetros (variando quantidade ou tipo)**

Relacionamento É-UM

- Quando uma classe herda atributos e métodos de outras classes, diz-se que a subclasse ***estende*** a superclasse.

- Para saber se uma classe estende outra classe, aplique o teste É-UM.

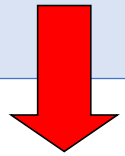
- Exemplos:

  - **Um triângulo É-UMA forma (faz sentido ter herança).**
  - **Um triângulo É-UM quadrado (não faz sentido, não há herança).**
  - **Um banheiro É-UMA banheira (não faz sentido, não há herança).**
  - **Um banheiro TEM-UMA banheira.**

Relacionamento É-UM – exemplo

```
public class Carro {
    //código da classe carro
    //...
}
```

```
public class Subaru extends Carro {
    //código específico da classe Subaru
    //a classe herda membros acessíveis de Carro
    //...
}
```



Todo Subaru É-UM Carro!!!

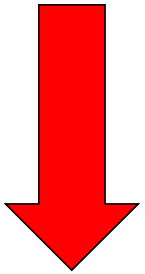
Relacionamento TEM-UM

- Esse tipo de relacionamento é baseado na utilização e não na herança.
- Ocorre quando uma classe tem uma referência a uma instância de outra classe, ou seja, o código da classe A apresenta uma referência a uma instância da classe B.
- Por exemplo, podemos dizer que: Um Cavalo É-UM Animal. Um Cavalo TEM-UMA Rédea.
- A propriedade TEM-UM não está relacionada com a propriedade É-UM, ou seja, não tem nada com a herança de classes.

Relacionamento TEM-UM – exemplo

```
public class Autor {
    String nome, cidade;
    public Autor(String nome, String cidade) {
        this.nome = nome;
        this.cidade = cidade;
    }
}
```

A classe Livro TEM-UMA referência à classe Autor



```
public class Livro {
    String titulo;
    Autor a;
    public Livro(String titulo, String nome, String cidade) {
        this.titulo = titulo;
        a = new Autor(nome, cidade);
    }
}
```

Modificador de acesso *protected*

- É muito semelhante ao encapsulamento ***default*** ou de pacote ou *package*.
- Os comportamentos ***default*** e ***protected*** só diferem quando a herança é aplicada.
- Os membros de uma classe com acesso ***default*** ou de pacote só podem ser acessados por classes que estejam dentro do mesmo pacote.
- Membros definidos como ***protected*** podem ser herdados para subclasses que estejam em pacotes diferentes da superclasse.

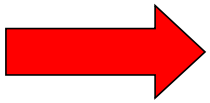
Modificador de acesso *protected*

Protegido = herança, ou seja, membros são herdados e não podem ser acessados.

- A subclasse só pode ver o membro protegido por meio da herança.

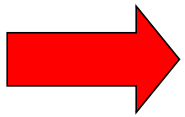
Modificador de acesso *protected* – exemplo

```
package teste;
public class Pai {
    protected int x = 9;
}
```



A variável x é acessível a todas as outras classes dentro do pacote e herdável a todas as subclasses fora do pacote.

```
package outro;
import teste.Pai;
public class Filho extends Pai {
    public void imprimir() {
        System.out.println(x);
    }
}
```



A variável protegida x está sendo acessada através da herança. Subclasses herdam membros da superclasse, mas não podem acessar esses membros.

Modificador de acesso *protected* – exemplo

```
package outro;
import teste.Pai;
public class Filho extends Pai {
    public void imprimir() {
        System.out.println(x);
        Pai p = new Pai();
        System.out.println(p.x);
    }
}
```

Depois que a subclasse de fora do pacote herda o membro *protected*, ele se torna *private* para qualquer código de fora da subclasse, com exceção das subclasses dessa subclasse.

Acesso ao atributo x da classe **Filho** que foi herdado da classe **Pai**.

Acesso ao atributo x da classe **Pai**.

Um membro protegido tem acesso de nível de pacote a todas as classes, exceto as subclasses. **Para uma subclasse de fora do pacote, o membro *protected* só pode ser acessado através da herança.**

Compatibilidade de tipos

- A linguagem de programação Java é *fortemente tipada*, ou seja, ela verifica na maioria das vezes a compatibilidade dos tipos durante a compilação.

- Lembre-se:

Todo objeto da subclasse também é um objeto da superclasse, ou seja, através de uma variável da superclasse pode-se referenciar um objeto da subclasse.

O contrário não é verdadeiro, ou seja, uma variável da subclasse não pode referenciar um objeto da superclasse.

Compatibilidade de tipos (conversão de tipos)

- Uma coerção é usada para indicar ao compilador que uma expressão deve ser tratada como tendo o tipo especificado pela coerção.
- Por exemplo, suponha que a classe ***Professor*** estenda a classe ***Pessoa***:

```
Professor pr = new Professor();
Pessoa pe = new Pessoa();
Pessoa pe2 = (Pessoa) pr;
```

- Você pode testar a classe de um objeto usando o operador ***instanceof***, o qual é avaliado como ***true*** ou ***false.***

```
if(pr instanceof Professor) {}
```

Modificador *final*

- O modificador ***final*** pode ser usado para ***classes***, ***variáveis*** e ***métodos***, tendo um comportamento diferente para cada um.

- Uma classe definida com o modificador final não pode ser estendida, ou seja, não podemos aplicar herança na classe.

- O modificar final aplicado a uma variável indica que a mesma é uma constante, ou seja, tem um valor inicial que não pode ser alterado durante a execução do programa.

- Para métodos, o modificar final indica que o método não pode ser sobrescrito, ou seja, não pode ser redefinido nas subclasses.

Modificador *final*

```
public final class MinhaSuperClasse {
    int meuAtributo;

    public final void meuMetodo(int valor) {
        this.meuAtributo = valor;
    }
}
```

*A classe **MinhaSuperClasse** não pode ser estendida porque foi definida como **final***

```
public class MinhaSubClasse extends MinhaSuperClasse {

}
```

Modificador *final* – exemplo

```
public class MinhaSuperClasse {
    int meuAtributo;

    public final void meuMetodo(int valor) {
        this.meuAtributo = valor;
    }
}
```

O método meuMetodo() não pode ser sobrescrito porque foi definido como final

```
public class MinhaSubClasse extends MinhaSuperClasse {
    int meuSubAtributo;

    public void meuMetodo(int valor) {
        this.meuSubAtributo = 2*valor;
    }
}
```

Modificador *final* – exemplo

```
public class MinhaClasse {
   final int valor = 150;
}
```

O atributo valor não pode ser alterado porque foi definido como final.

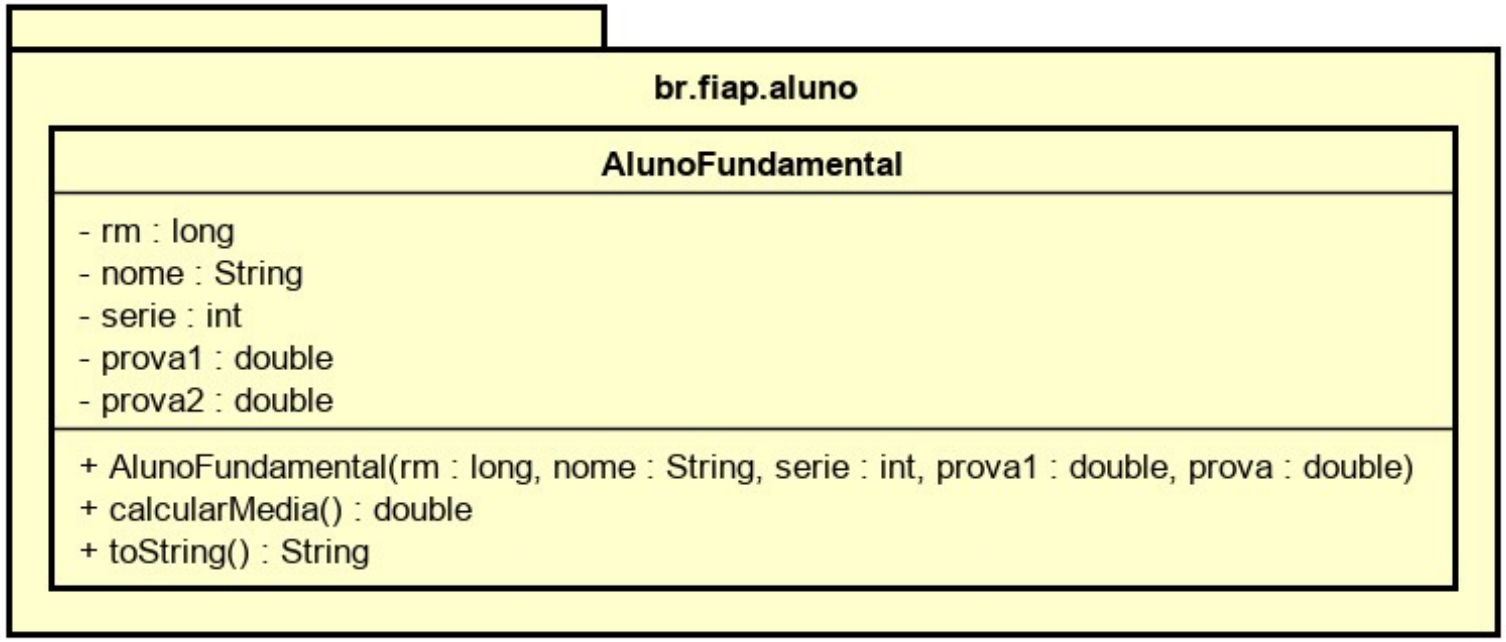
```
public class TesteMinhaClasse {
   public static void main(String[] args) {
      MinhaClasse minhaClasse = new MinhaClasse();
      minhaClasse.valor = 250;

   }
}
```


Prof. Dr. Antonio Marcos SELMINI – selmini@fiap.com.br

Modificador *final* – exemplo

- ***getClass()*** : Esse método retorna informações do objeto atual, como o package e o nome da classe.
- ***clone()*** : Retorna uma referência - ou cópia - de um objeto.
- ***toString()*** : Retorna uma string com o ***package***, nome da classe e um hexadecimal que representa o objeto em questão.
- ***equals(Object object)*** : Faz a comparação entre dois ***Objects***, e retorna *true* se os objetos forem o mesmo, e false se não forem o mesmo. É útil para saber se dois objetos apontam para o mesmo local na memória.
- ***hashCode()*** : Esse método retorna um inteiro único de cada objeto, muito usado em ***Collections.***

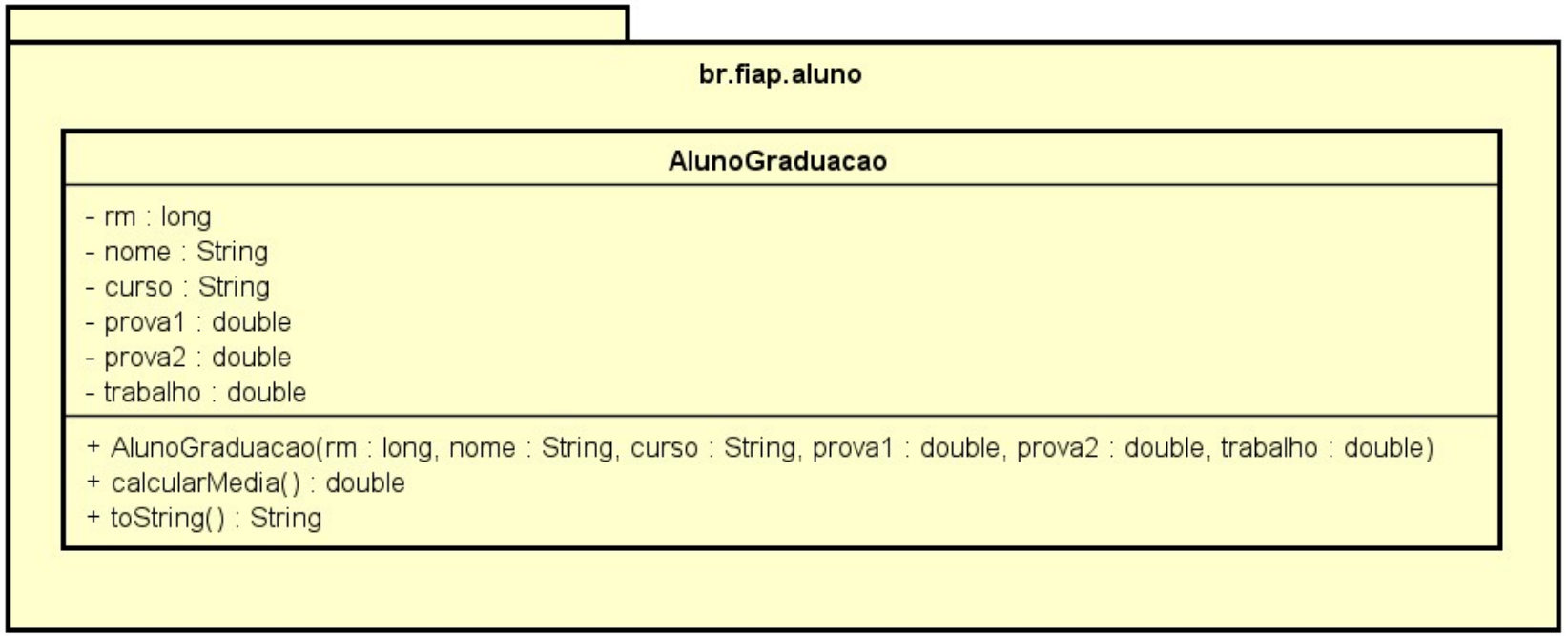
Exercício de programação 1



A média final de um aluno do ensino fundamental é a média aritmética entre a nota de suas duas provas.

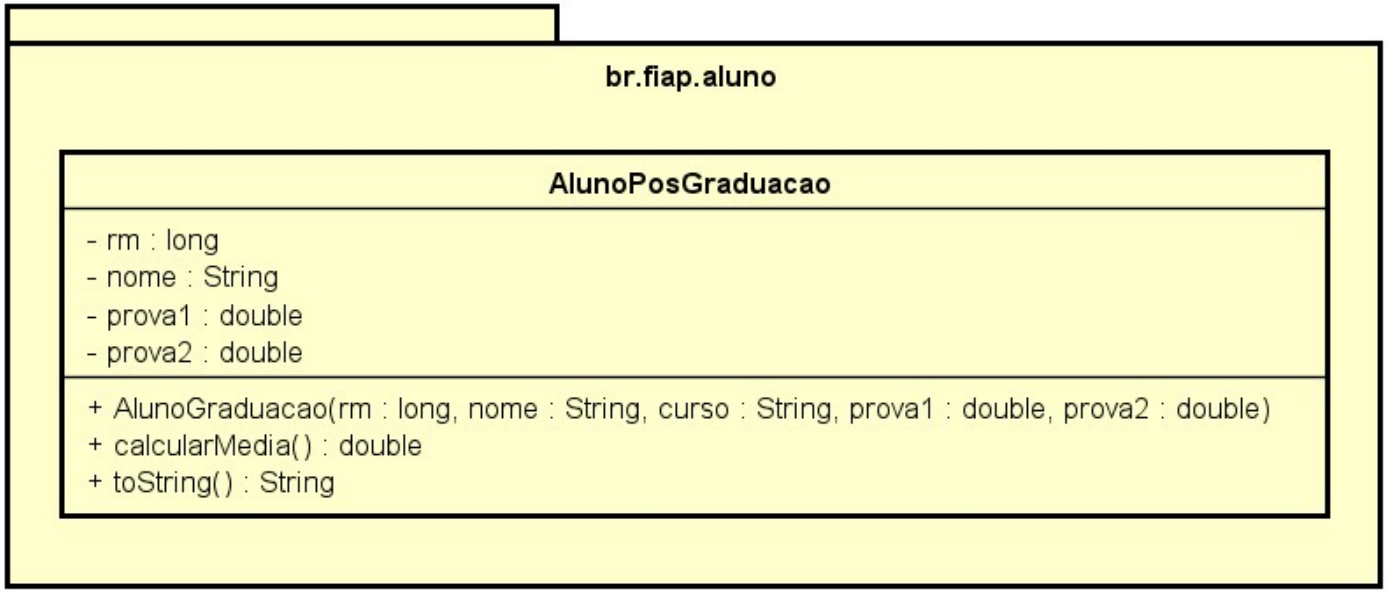
O método ***toString()*** deverá ser sobrescrito para retornar o rm do aluno, seu nome, sua série e a média final.

Exercício de programação 1



- A média final de um aluno da graduação é composta por 70% da média aritmética entre as duas provas e 30% da nota do trabalho.
- O método ***toString()*** deverá ser sobrescrito para retornar o rm do aluno, seu nome, seu curso e a média final.

Exercício de programação 1



- A média final de um aluno da pós graduação é composta por 40% da nota da primeira prova e 60% da nota da segunda prova.
- O método ***toString()*** deverá ser sobrescrito para retornar o rm do aluno, seu nome, e a média final.

Exercício de programação 1

- Escreva um programa em Java que gere pelo menos um objeto de cada classe e armazene em um único array de objetos.

- Imprima os dados de todos os alunos no vídeo.

Exercício de programação 2

EmpregadoComissionado
- matricula : long - nome : String - totalDeVendas : double - comissao : double
+ calcularSalario() : double + toString() : String

EmpregadoHorista
- matricula : long - nome : String - totalDeHorasTrabalhadas : int - valorDaHoraTrabalhada : double
+ calcularSalario() : double + toString() : String

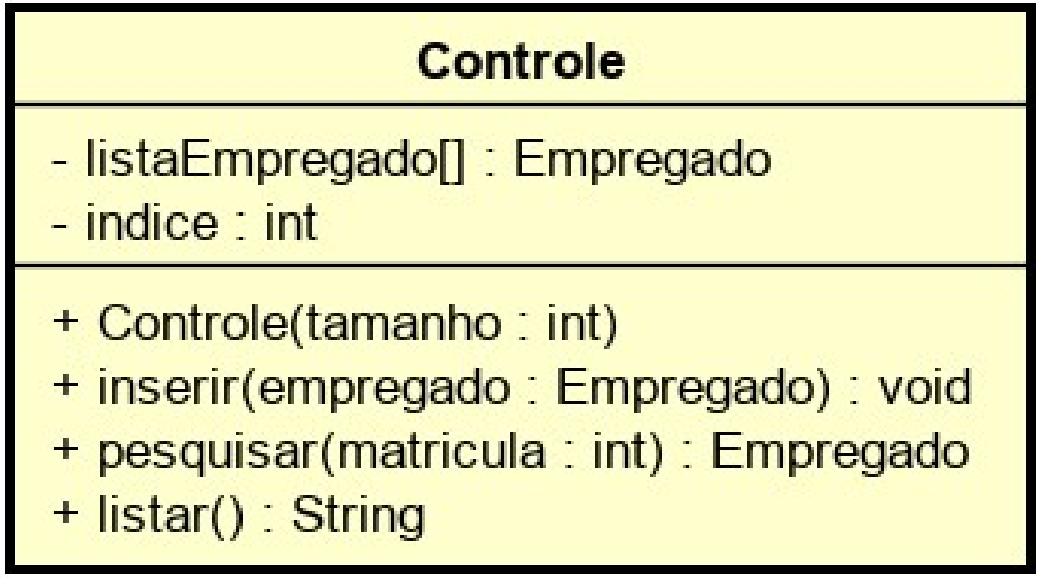
Cálculo do salário:
- totalDeVendas * comissao / 100

Cálculo do salário:
- horasTrabalhadas * valorDaHora

I) Todas as classes devem ter o método construtor;
II) Gerar alguns objetos e armazenar em um ***único array***;

Sistemas de Informação | FIAP
Prof. Dr. Antonio Marcos SELMINI – selmini@fiap.com.br

Exercício de programação 2

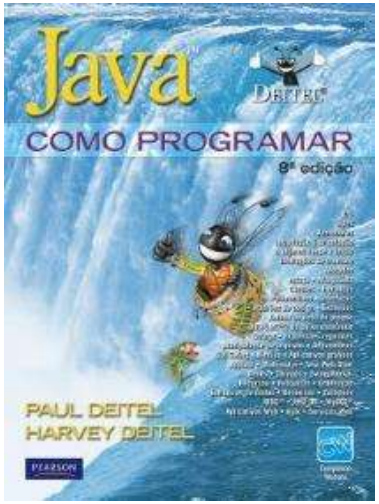


- O método construtor deverá receber como parâmetro o tamanho do array lista e deverá instanciá-lo (no corpo do método).
- O método **inserir()** deverá inserir um empregado na lista. O controle das posições fica a seu critério.
- O método **pesquisar()** deverá retornar um empregado, que por sua vez deverá ser pesquisado pelo número da matrícula.
- O método **listar()** deverá retornar todos os dados de cada empregado, inclusive o valor do salário.

REFERÊNCIAS

- DEITEL, H. M., DEITEL, P. J. **JAVA como programar**. 8ª edição. São Paulo: Prentice-Hall, 2010.

- SCHILDT, H. **Java para Iniciantes – Crie, Compile e Execute Programas Java Rapidamente**. 6ª Edição, Editora Bookman, Porto Alegre, RS, 2015.



REFERÊNCIAS

- KNUDSEN, J., NIEMEYER, P. **Aprendendo Java**. Rio de Janeiro: Editora Elsevier Campus, 2000.
- FLANAGAN, D. **Java – o guia essencial**. Porto Alegre: Editora Bookman, 2006.

REFERÊNCIAS

- ARNOLD, K., GOSLING, J., HOLMES, D., **Java programming language**. 4th Edition, Editora Addison-Wesley, 2005.

- JANDL JUNIOR, P. **Introdução ao Java**. São Paulo: Editora Berkeley, 2002.