




Java WebDeveloper - Aula 03 - Noite Seg, Qua, Sex - Noite

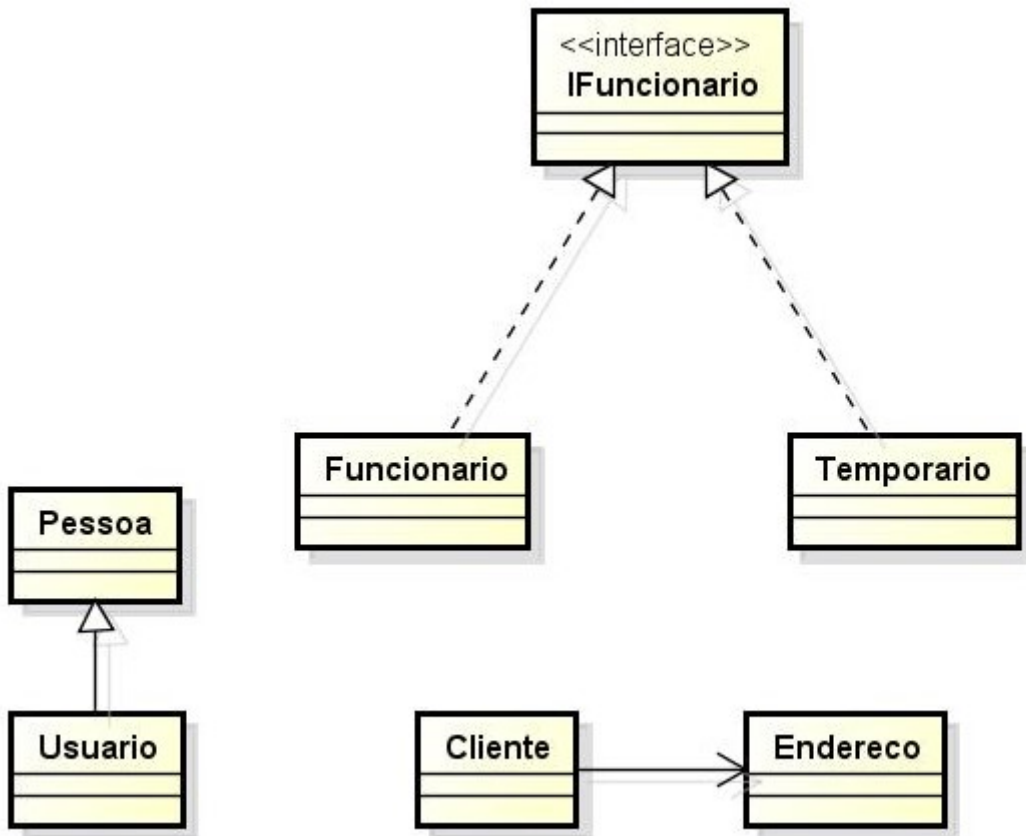
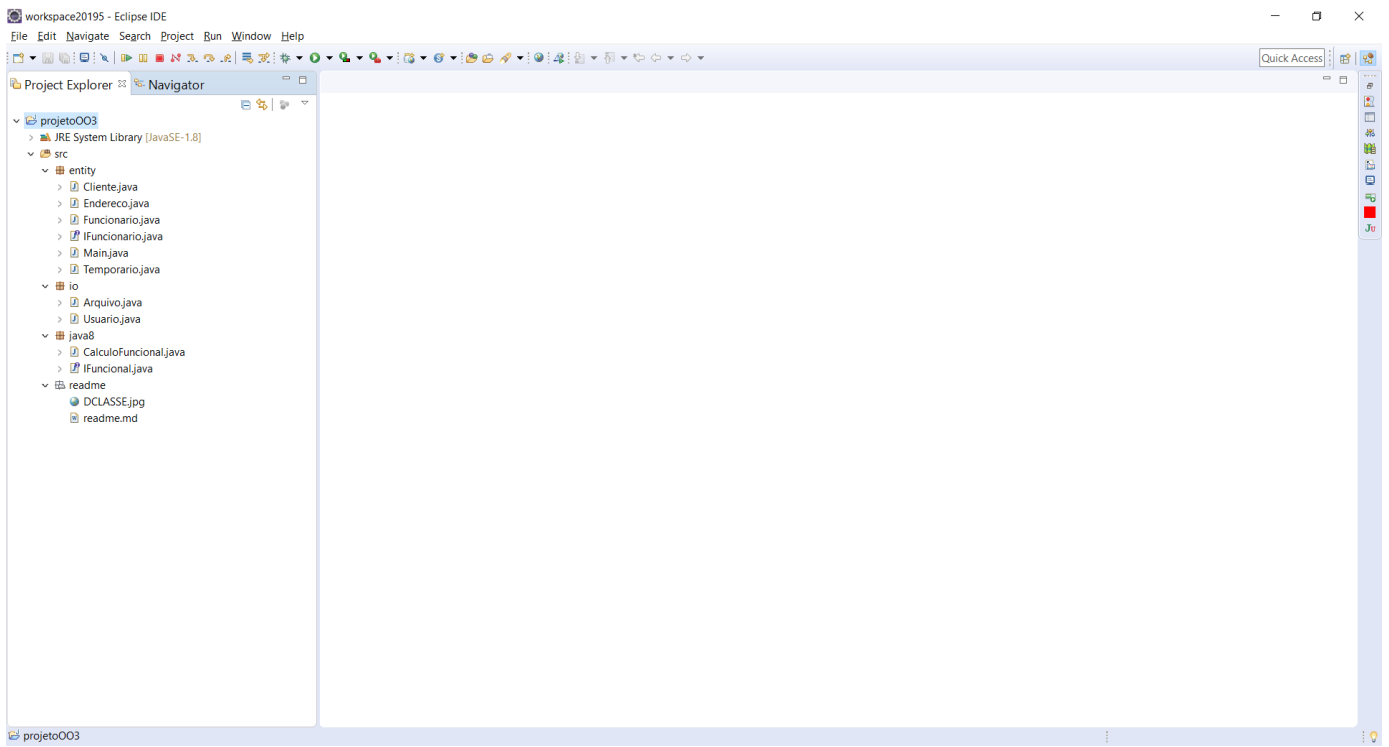


Tema da aula:

Orientação a Objetos, UML, Relacionamento OnetoOne, Interface, Sobrescrita, SimpleDateFormat Formatação de Data, Gravar arquivo em arquivo, Arquivo de log, Chat, Lista, Programação Funcional

—  Prof Edson Belém - profedsonbelem@gmail.com (<mailto:profedsonbelem@gmail.com>) 🕒 Sexta, Mar 13, 2020

👁 Estrutura do projeto depois de finalizado:



</> Readme.md (<http://Readme.md>)

```

package entity;

#public interface IFuncionario {

    //throws Exception (citar Erros Fortes)
    //Erros que Herdam RuntimeException, erros fracos
    //Erro numero Throwable ...
    //citar
#public void gerarSalario() throws Exception;
    //IllegalArgumentException Erro Fraco ...
    //nao precisa nem de throws
    //throws (grava)

    if (getValor()<800){
        throw new Exception("Salario invalido");
    }

#}
    //Cada Classe que Implementa a interface, aplica a regra de
    //negocio da interface ...
    //EJB (todo método negocio) interface ...
    //throws e throw (lançar)
//try (tentar) e (catch)

```

</> **Cliente.java**

```
1 package entity;
2
3 public class Cliente {
4
5     private Integer idCliente;
6     private String nome;
7     private String email;
8     // Associacao Unidirecional
9     // Cliente HasOne ...
10    private Endereco endereco;
11
12    public Cliente() {
13    }
14
15    public Cliente(Integer idCliente, String nome, String email) {
16        super();
17        this.idCliente = idCliente;
18        this.nome = nome;
19        this.email = email;
20    }
21
22    @Override
23    public String toString() {
24        return "Cliente [idCliente=" + idCliente + ", nome=" + nome
25        + ", email=" + email + "];"
26    }
27
28    public Integer getIdCliente() {
29        return idCliente;
30    }
31
32    public void setIdCliente(Integer idCliente) {
33        this.idCliente = idCliente;
34    }
35
36    public String getNome() {
37        return nome;
38    }
39
40    public void setNome(String nome) {
41        this.nome = nome;
42    }
43
44    public String getEmail() {
45        return email;
46    }
47
48    public void setEmail(String email) {
49        this.email = email;
50    }
51
52    public Endereco getEndereco() {
53        return endereco;
54    }
```

```
55
56     public void setEndereco(Endereco endereco) {
57         this.endereco = endereco;
58     }
59
60     public static void main(String[] args) {
61         // Rodar Endereco no Cliente
62
63         Cliente c = new Cliente();
64         c.setNome("Luciana");
65         c.setEndereco(new Endereco("Centro", "Rio de Janeiro"));
66         // Endereco faz parte do Cliente
67         // Cliente tem Endereco
68         // Endereco está sendo executado dentro do Cliente ...
69         // Relacionamento mais forte sempre é composição
70         // Agregação é forte mais não tanto ...
71     }
72 }
```

</> Endereco.java

```
1  package entity;
2
3  public class Endereco {
4
5      private String bairro;
6      private String cidade;
7
8      public Endereco() {
9
10     }
11
12     public Endereco(String bairro, String cidade) {
13         super();
14         this.bairro = bairro;
15         this.cidade = cidade;
16     }
17
18     @Override
19     public String toString() {
20         return "Endereco [bairro=" + bairro + ", cidade=" + cidade + "];"
21     }
22
23     public String getBairro() {
24         return bairro;
25     }
26
27     public void setBairro(String bairro) {
28         this.bairro = bairro;
29     }
30
31     public String getCidade() {
32         return cidade;
33     }
34
35     public void setCidade(String cidade) {
36         this.cidade = cidade;
37     }
38
39 }
```

Interface

Interface in Java

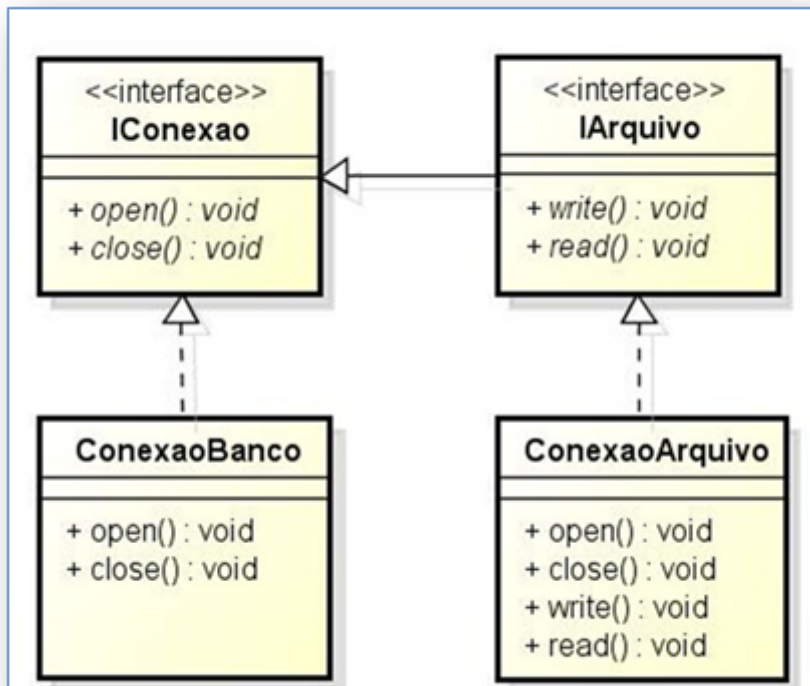


Em algumas linguagens de programação, termo interface é uma referência à característica que permite a construção de interfaces que isolam do mundo exterior os detalhes de implementação de um componente de software.

A Interface tem como objetivo a padronização de métodos para sua aplicação e o comando para representar a interface é o `implements`. Ela não pode ser instanciada e sim implementada por outra classe, entretanto a classe que a implementa tem que ser concreta. Uma classe pode implementar varias interfaces. A Interface `extends` outra interface e uma classe concreta `implements` uma ou mais interfaces.

A Interface é um recurso muito utilizado em Java, bem como na maioria das linguagens orientadas a objeto, para “obrigar” a um determinado grupo de classes a ter métodos ou propriedades em comum para existir em um determinado contexto, contudo os métodos podem ser implementados em cada classe de uma maneira diferente. Pode-se dizer, a grosso modo, que uma interface é um contrato que quando assumido por uma classe deve ser implementado.

Dentro das interfaces existem somente assinaturas de métodos e propriedades, cabendo à classe que a utilizará realizar a implementação das assinaturas, dando comportamentos práticos aos métodos.



```

1 package entity1;
2
3 public interface IConexao {
4
5     public void open() throws Exception;
6     public void close() throws Exception;
7 }

```

```

1 package entity1;
2
3 public interface IArquivo extends IConexao {
4
5     public void write() throws Exception;
6
7     public String read() throws Exception;
8
9 }

```



```

1  package entity1;
2
3  public class ConexaoBanco implements IConexao {
4
5      @Override
6      public void open() throws Exception {
7          System.out.println("Abrir Conexao Banco de Dados");
8      }
9
10     @Override
11     public void close() throws Exception {
12         System.out.println("Fechar Conexao Banco de Dados");
13     }
14 }

```

```

1  package entity1;
2
3  public class ConexaoArquivo implements IArquivo{
4
5      @Override
6      public void open() throws Exception {
7          System.out.println("Abrir Conexao Arquivo");
8      }
9
10     @Override
11     public void close() throws Exception {
12         System.out.println("Fechar Conexao Arquivo");
13     }
14
15     @Override
16     public void write() throws Exception {
17         System.out.println("Escrever no Arquivo");
18     }
19
20     @Override
21     public String read() throws Exception {
22         return "Ler do Arquivo";
23     }
24 }

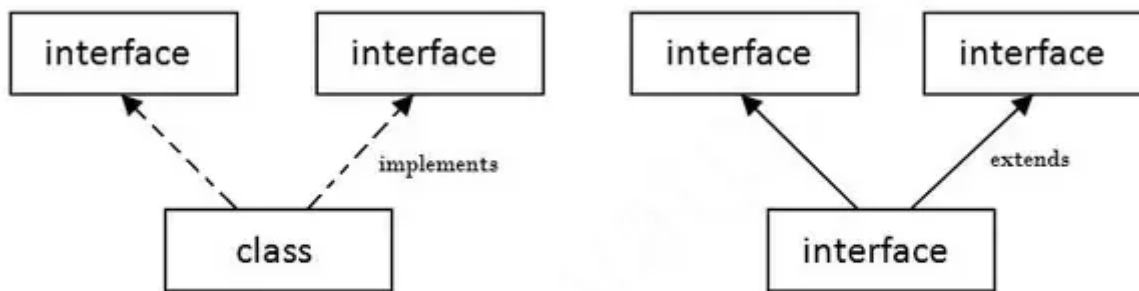
```

Sintaxe

public class nome_classe implements nome_interface

Onde

- nome_classe – Nome da classe a ser implementada.
- nome_Interface – Nome da interface a se implementada pela classe.



Multiple Inheritance in Java

Utilização

Um exemplo clássico de utilização de interfaces é o do sistema operacional que, através de uma interface de programação de aplicativos, permite que os programas utilizem os recursos do sistema (memória, CPU e etc) sem que os seus detalhes de implementação sejam conhecidos do programador. Este esquema isola e protege o sistema operacional de eventuais erros cometidos pela aplicação.

Os componentes de software utilizam interfaces padronizadas para criar uma camada de abstração que facilite a reutilização e a manutenção do software. Neste cenário, a interface de um módulo de software deve ser mantida em separado da sua implementação e qualquer outro módulo, que interaja com (cliente de), deve ser forçado a fazê-lo apenas através da interface. Este mecanismo permite que no caso de uma alteração em, o módulo continue funcionando; desde que a utilização do módulo pelo módulo satisfaça as especificações da interface. (Ver também o princípio da substituição de Liskov).

Uma interface disponibiliza tipos variados de acesso entre componentes, como por exemplo: constantes, tipos de dado, procedimentos, especificação de exceções e assinaturas de métodos. Em alguns casos é mais apropriado definir as variáveis como parte das interfaces. As interfaces também especificam a funcionalidade disponibilizada através de comentários ou através de declarações lógicas formais (assertions).

Linguagens

O princípio da interface é um alicerce da programação modular que, por sua vez, é precursora e parte da programação orientada a objeto. Na programação orientada a objeto, a interface de um objeto consiste de um conjunto de métodos que um objeto deve suportar. É importante notar que as variáveis de instância não fazem parte da interface de um objeto pois devem ser acessadas somente pelos "métodos de acesso". Historicamente, as interfaces são derivadas dos arquivos de cabeçalho da Linguagem C (normalmente arquivos com extensão ".h") que separam o contexto sintático de um módulo (ou protótipos de funções) da sua implementação.

Algumas linguagens de programação orientadas a objeto exigem que a interface do objeto seja especificada de forma separada da implementação do objeto, enquanto outras não fazem esta exigência. Por exemplo, em linguagens de programação como Objective-C, a classe do objeto define a sua interface e é declarada em um arquivo de cabeçalho (header em inglês) e, por outro lado, a implementação da classe é mantida em um arquivo chamado de "arquivo fonte". Devido à tipagem dinâmica existente na Objective-C, que permite o envio de mensagens para qualquer objeto, a interface de uma classe é importante para determinar para quais métodos um objeto de uma classe responde.

A linguagem de programação Java, que recebeu influência da Objective-C, utiliza outra abordagem para o conceito de interface, assim como outras linguagens orientadas a objeto, onde a interface especifica um conjunto de métodos ou funcionalidades comuns a um conjunto de classes. Ver interface na linguagem Java.

Algumas linguagens de programação como D, Java e Logtalk, por exemplo, permitem a definição de "hierarquias de interfaces". A linguagem Logtalk também suporta a implementação "privada" e "protegida" dos métodos de uma interface.

A linguagem Eiffel inclui na interface de uma classe as pré e pós-condições para execução dos seus métodos. Esta característica é essencial para a metodologia do projeto por contrato, e pode ser entendida como uma extensão das condições impostas pelos tipos dos argumentos. Estas regras podem ser especificadas na implementação da classe ou em uma classe genérica (classe mãe) que não precisa implementar os seus métodos. Elas são extraídas por processadores da linguagem e podem ser vistas em uma ambiente de desenvolvimento além de gerarem verificações em tempo de execução. A linguagem garante que classes derivadas obedeçam aos contratos definidos nas classes que servem de base.

</> IFuncionario.java

```

1  package entity;
2
3  public interface IFuncionario {
4
5      // throws Exception (citar Erros Fortes)
6      // Erros que Herdam RuntimeException, erros fracos
7      // Erro numero Throwable ...
8      // citar
9      public void gerarSalario() throws Exception;
10     // IllegalArgumentException Erro Fraco ...
11     // nao precisa nem de throws
12     // throws (grava)
13
14 }
15 //Cada Classe que Implementa a interface, aplica a regra de
16 //negocio da interface ...
17 //EJB (todo mÃ©todo negocio) interface ...
18 //throws e throw (lançar)
19 //try (tentar) e (catch)

```

Sobrescrita de Métodos(Override)



A sobrescrita (ou override) está diretamente relacionada à orientação a objetos, mais especificamente com a herança. Com a sobrescrita, conseguimos especializar os métodos herdados das superclasses, alterando o seu comportamento nas subclasses por um mais específico. A sobrescrita de métodos consiste basicamente em criar um novo método na classe filha contendo a mesma assinatura e mesmo tipo de retorno do método sobrescrito.

Quando mencionado anteriormente que o método deve possuir a mesma assinatura, significa dizer que o método deve possuir o mesmo nome, a mesma quantidade e o mesmo tipo de parâmetros utilizado no método sobrescrito. Com relação ao tipo de retorno, este pode ser um subtipo do tipo de retorno do método sobrescrito, por exemplo: se o método da superclasse retornar um List, é permitido que o novo método retorne um ArrayList ou qualquer outro List. No entanto o oposto não é permitido, gerando um erro de compilação.

A sobrescrita de métodos ocorre quando tem uma subclasse herdado uma classe ou implementado uma interface. Esse método caracteriza-se pela assinatura do método ser igual ao método da Superclasse ou da Interface ou pela anotação @Override (Sobrescrita) acima do método.

```
1 package entity;
2
3 // A classe Object está implícita na classe Cliente
4 //public class Cliente extends Object
5 public class Cliente {
6
7     private Integer idCliente;
8     private String nome;
9
10    @Override
11    public String toString() {
12        return "Cliente [idCliente=" + idCliente + ", nome=" + nome + "];"
13    }
14
15    public Integer getIdCliente() {
16        return idCliente;
17    }
18    public void setIdCliente(Integer idCliente) {
19        this.idCliente = idCliente;
20    }
21    public String getNome() {
22        return nome;
23    }
24    public void setNome(String nome) {
25        this.nome = nome;
26    }
27 }
28
```

Quando uma Subclasse herda uma Superclasse. Nesse caso está sendo sobrescrito o método imprimirDescricao do Funcionario e reescrito no FuncionarioCLT.

```
1 package entity;
2
3 public class Funcionario {
4
5     private Integer idFuncionario;
6     private String nome;
7     private String descricao;
8
9     public Funcionario() {
10    }
11
12    public Funcionario(Integer idFuncionario, String nome,
13        String descricao) {
14        super();
15        this.idFuncionario = idFuncionario;
16        this.nome = nome;
17        this.descricao = descricao;
18    }
19
20    @Override
21    public String toString() {
22        return "Funcionario [idFuncionario=" + idFuncionario +
23            ", nome=" + nome + ", descricao=" + descricao + "];"
24    }
25
26    public Integer getIdFuncionario() {
27        return idFuncionario;
28    }
29    public void setIdFuncionario(Integer idFuncionario) {
30        this.idFuncionario = idFuncionario;
31    }
32    public String getNome() {
33        return nome;
34    }
35    public void setNome(String nome) {
36        this.nome = nome;
37    }
38    public String getDescricao() {
39        return descricao;
40    }
41    public void setDescricao(String descricao) {
42        this.descricao = descricao;
43    }
44    public String imprimirDescricao(){
45        return "Funcionario Padrao";
46    }
47 }
```

```

1 package entity;
2
3 public class FuncionarioCLT extends Funcionario {
4
5     public FuncionarioCLT() {
6     }
7
8     public FuncionarioCLT(Integer idFuncionario, String nome) {
9         super(idFuncionario, nome);
10    }
11
12    @Override
13    public String imprimirDescricao() {
14        return "Funcionario CLT";
15    }
16 }

```

.

```

1 package main;
2
3 import entity.Funcionario;
4 import entity.FuncionarioCLT;
5
6 public class Main {
7
8     public static void main(String[] args) {
9
10        Funcionario f = new Funcionario(1, "Joao");
11        System.out.println("Funcionario: " +
12            f + "\n Descricao: " + f.imprimirDescricao());
13        FuncionarioCLT f1 = new FuncionarioCLT(2, "Luiz");
14        System.out.println("\nFuncionario: " + f1 +
15            "\n Descricao: " + f1.imprimirDescricao());
16    }
17 }

```

Quando uma classe implementa uma Interface.

```

1 package entity;
2
3 public interface IArquivo {
4
5     public void open() throws Exception;
6     public void close() throws Exception;
7 }

```

.

```
1 package entity;
2
3 public class Arquivo implements IArquivo {
4
5     @Override
6     public void open() throws Exception {
7         System.out.println("Abrir o Arquivo");
8     }
9
10    @Override
11    public void close() throws Exception {
12        System.out.println("Fechar o Arquivo");
13    }
14 }
```

```
1 package main;
2
3 import entity.Arquivo;
4
5 public class Main {
6
7     public static void main(String[] args) {
8         Arquivo a = new Arquivo();
9         try {
10             a.open();
11             a.close();
12         } catch (Exception e) {
13             System.out.println("Error" + e.getMessage());
14         }
15     }
16 }
```

</> Funcionario.java


```
1 package entity;
2
3 //classe Entidade
4 public class Funcionario implements IFuncionario {
5
6     // desconto 500 programador, 800 arquiteto, 300 teste
7     // atributos
8
9     private Integer idFuncionario;
10    private String nome;
11    private Double salario;
12    private String cargo;
13    private Double desconto;
14
15    public Funcionario() {
16    }
17
18    public Funcionario(Integer idFuncionario, String nome,
19        Double salario, String cargo, Double desconto) {
20        super();
21        this.idFuncionario = idFuncionario;
22        this.nome = nome;
23        this.salario = salario;
24        this.cargo = cargo;
25        this.desconto = desconto;
26    }
27
28    @Override
29    public String toString() {
30        return "Funcionario [idFuncionario=" + idFuncionario +
31            ", nome=" + nome + ", salario=" + salario +
32            ", cargo=" + cargo + ", desconto=" + desconto + "]";
33    }
34
35    // gerarSalario () _ Sobrescrevendo o Método
36
37    @Override
38    public void gerarSalario() throws Exception {
39        switch (this.cargo) {
40            case "teste":
41                setDesconto(300.);
42                break;
43            case "programador":
44                setDesconto(500.);
45                break;
46            case "arquiteto":
47                setDesconto(800.);
48                break;
49            default:
50                throw new Exception("Cargo nao Definido");
51        }
52        this.setSalario(this.getSalario() - getDesconto());
53    }
54
```

```
55 //Interface
56 //      (Classe que tem as regras de negocio que serão implementadas)
57 //Quando vira classe A Interfafce implementa a Regra de Negocio...
58
59     public Integer getIdFuncionario() {
60         return idFuncionario;
61     }
62
63     public void setIdFuncionario(Integer idFuncionario) {
64         this.idFuncionario = idFuncionario;
65     }
66
67     public String getNome() {
68         return nome;
69     }
70
71     public void setNome(String nome) {
72         this.nome = nome;
73     }
74
75     public Double getSalario() {
76         return salario;
77     }
78
79     public void setSalario(Double salario) {
80         this.salario = salario;
81     }
82
83     public String getCargo() {
84         return cargo;
85     }
86
87     public void setCargo(String cargo) {
88         this.cargo = cargo;
89     }
90
91     public Double getDesconto() {
92         return desconto;
93     }
94
95     public void setDesconto(Double desconto) {
96         this.desconto = desconto;
97     }
98 }
```

</> Temporario.java

```
1 package entity;
2
3 public class Temporario implements IFuncionario {
4
5     // cargo _ estagiario desconto ...(500)
6     // professor trainee _ desconto (1000.)
7
8     private Integer idFuncionario;
9     private String nome;
10    private Double salario;
11    private String cargo;
12    private Double desconto;
13
14    @Override
15    public void gerarSalario() throws Exception {
16        switch (cargo) {
17            case "estagiario":
18                setDesconto(500.);
19                break;
20            case "professor trainee":
21                setDesconto(1000.);
22                break;
23            default:
24                throw new Exception("Tu nao existe nao ...");
25        }
26        this.setSalario(this.getSalario() - this.getDesconto());
27    }
28
29    public Temporario() {
30    }
31
32    public Temporario(Integer idFuncionario, String nome, Double salario,
33        String cargo, Double desconto) {
34        super();
35        this.idFuncionario = idFuncionario;
36        this.nome = nome;
37        this.salario = salario;
38        this.cargo = cargo;
39        this.desconto = desconto;
40    }
41
42    @Override
43    public String toString() {
44        return "Temporario [idFuncionario=" + idFuncionario + ", nome="
45            + nome + ", salario=" + salario + ", cargo=" + cargo +
46            ", desconto=" + desconto + "]\n";
47    }
48
49    public Integer getIdFuncionario() {
50        return idFuncionario;
51    }
52
53    public void setIdFuncionario(Integer idFuncionario) {
54        this.idFuncionario = idFuncionario;
55    }
56 }
```

```
55     }
56
57     public String getNome() {
58         return nome;
59     }
60
61     public void setNome(String nome) {
62         this.nome = nome;
63     }
64
65     public Double getSalario() {
66         return salario;
67     }
68
69     public void setSalario(Double salario) {
70         this.salario = salario;
71     }
72
73     public String getCargo() {
74         return cargo;
75     }
76
77     public void setCargo(String cargo) {
78         this.cargo = cargo;
79     }
80
81     public Double getDesconto() {
82         return desconto;
83     }
84
85     public void setDesconto(Double desconto) {
86         this.desconto = desconto;
87     }
88 }
```

</> Main.java

```

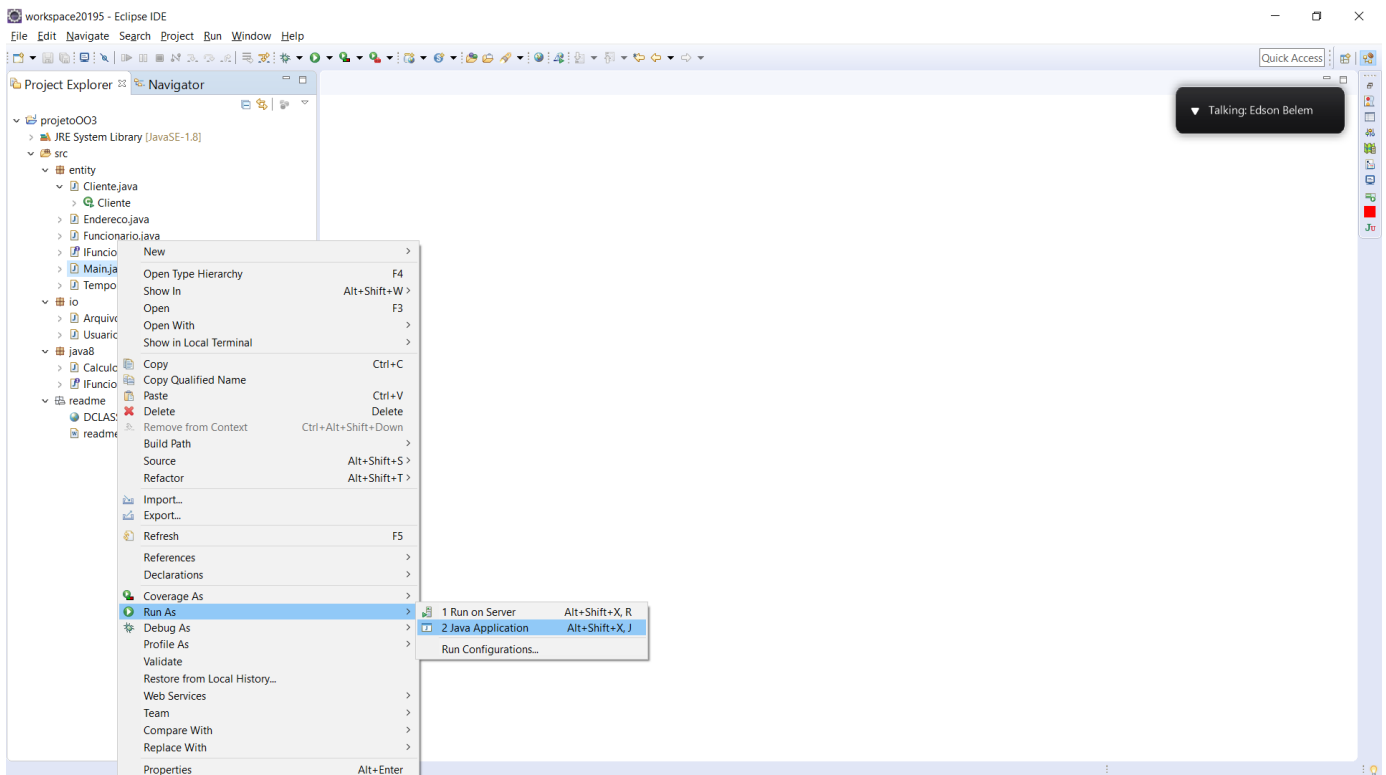
1 package entity;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         Funcionario ifu = new Funcionario();
8         try {
9             // throws te Obriga a tralhar dentro de try e catch
10            ifu.setIdFuncionario(5);
11            ifu.setNome("rafael");
12            ifu.setCargo("programador");
13            ifu.setSalario(7000.);
14
15            ifu.gerarSalario(); // toda vez que rodo throws
16
17            System.out.println(ifu.getNome());
18            System.out.println(ifu.getCargo());
19            System.out.println(ifu.getSalario());
20        } catch (Exception ex) {
21            ex.printStackTrace();
22        }
23    }
24 }

```

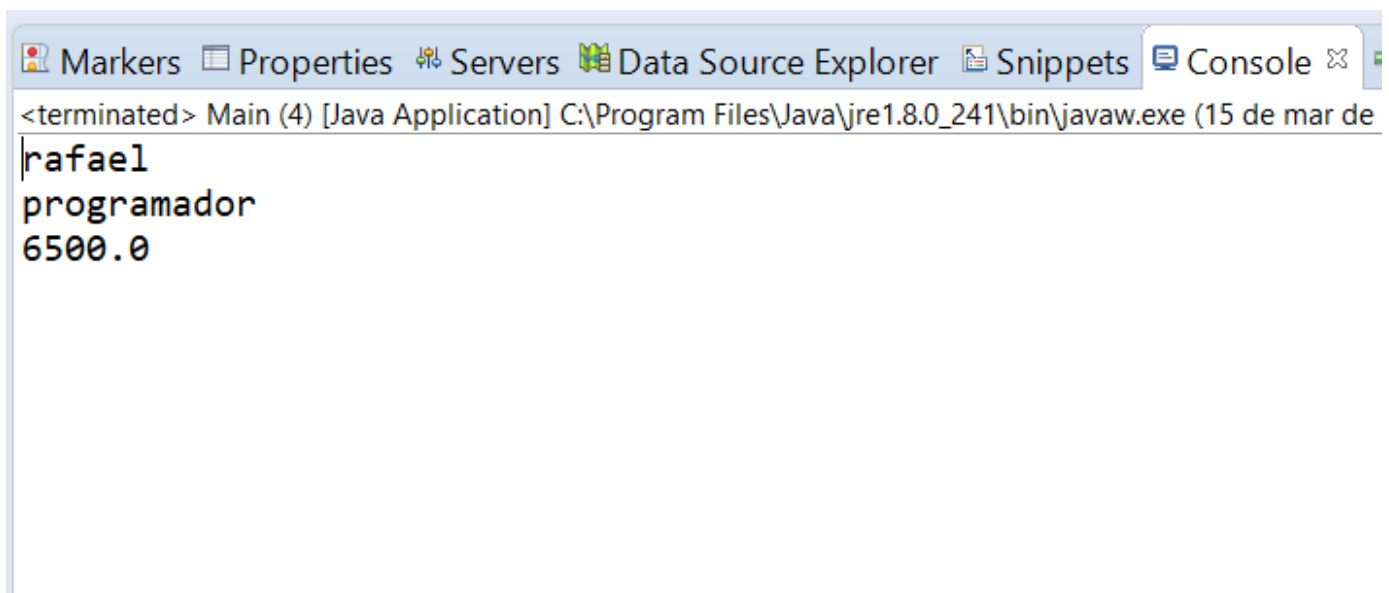


Para rodar a classe:

Clique na classe com o botão direito ➡ run as ➡ java application



👁 Resultado no console:



The screenshot shows the 'Console' tab of an IDE. The title bar of the console window reads: '<terminated> Main (4) [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (15 de mar de 2016 15:00:00)'. The console output consists of three lines: 'rafael', 'programador', and '6500.0'. The text is in a monospaced font.

```
<terminated> Main (4) [Java Application] C:\Program Files\Java\jre1.8.0_241\bin\javaw.exe (15 de mar de 2016 15:00:00)
rafael
programador
6500.0
```

</> Usuario.java

```
1 package io;
2
3 import java.text.SimpleDateFormat;
4 import java.util.Date;
5
6 public class Usuario {
7
8     // Formatar a Data ...
9     // mes-dia-ano
10    // formatar é como eu quero ver ...
11
12    static SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy
13    hh:mm:ss");
14    private String nome;
15    private String mensagem;
16
17    private Date data = new Date(); // data de agora ... now()
18
19    public Usuario() {
20    }
21
22    public Usuario(String nome, String mensagem) {
23        this.nome = nome;
24        this.mensagem = mensagem;
25    }
26
27    @Override
28    public String toString() {
29        return "Usuario [nome=" + nome + ", mensagem=" + mensagem +
30        " ,data=" + sdf.format(data) + " ]";
31    }
32
33    public String getNome() {
34        return nome;
35    }
36
37    public void setNome(String nome) {
38        this.nome = nome;
39    }
40
41    public String getMensagem() {
42        return mensagem;
43    }
44
45    public void setMensagem(String mensagem) {
46        this.mensagem = mensagem;
47    }
48
49    public Date getData() {
50        return data;
51    }
52
53    public void setData(Date data) {
54        this.data = data;
```

```
55     }  
56  
57 }
```

</> **Arquivo.java**


```

1  package io;
2
3  import java.io.BufferedReader;
4  import java.io.FileReader;
5  import java.io.FileWriter;
6  import java.util.ArrayList;
7  import java.util.List;
8
9  public class Arquivo {
10      // Gravar
11      FileWriter fw;
12
13      public void gravar(Usuario u) throws Exception {
14          fw = new FileWriter
15          ("/run/user/1000/gvfs/smb-share:server=coti204maq00,share=rede/"
16          + "chat1.log", true); // adicionar
17
18      // (false
19      // sobrescrever)
20      // gravar nome, mensagem, data
21
22          fw.write(u.toString() + "\n"); // enter
23          fw.close(); // fechar
24      }
25
26      FileReader fr;
27
28      public List<String> ler() throws Exception {
29          // Buscar a lista de linhas (gravadas)
30          List<String> lista = new ArrayList<String>();
31          // Apontando para o Arquivo
32          //ATENÇÃO PARA O CAMINHO DA REDE DA SUA MAQUINA
33          fr = new FileReader("/run/user/1000/gvfs" +
34          "/smb-share:server=coti204maq00,share=rede/chat1.log");
35          // Trazendo o Arquivo ...
36          BufferedReader bf = new BufferedReader(fr);
37          String s = "";
38          // Buscando a linha do arquivo (senao chegou a fim)
39          while ((s = bf.readLine()) != null) {
40              // !=null nao chegou
41              lista.add(s + "\n");
42          }
43          bf.close();
44          fr.close();
45          return lista;
46      }
47
48      public static void main(String[] args) {
49      try {
50
51          Arquivo arq = new Arquivo();
52          System.out.println("... Mensagens gravadas...");
53          System.out.println(arq.ler());
54          // Leitura do que está gravado

```

```

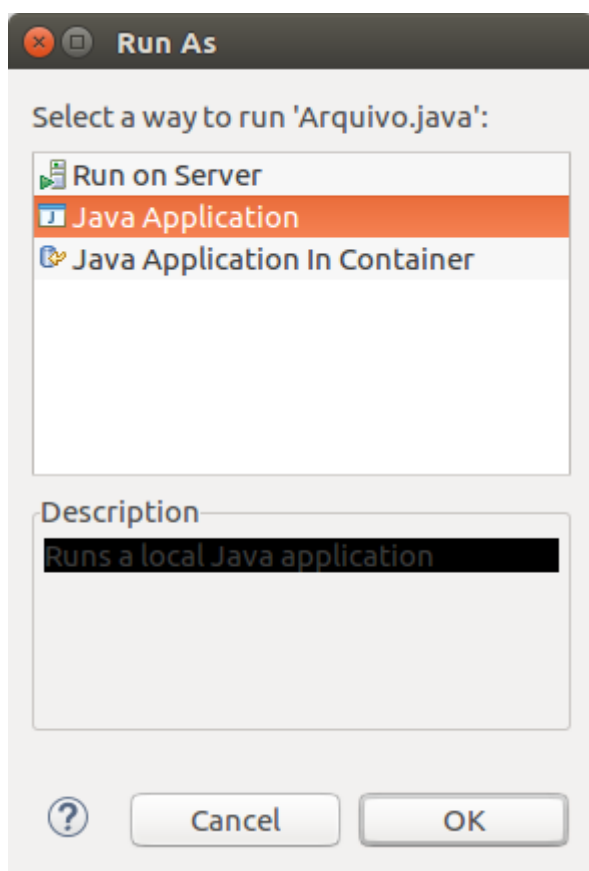
55
56     System.out.println("Gravando ...");
57     Usuario usu = new Usuario("belem", "Sexta Treze... Tadam..");
58     arq.gravar(usu); // gravo a Mensagem
59
60     System.out.println("... Novas Mensagens ...");
61     System.out.println(arq.ler()); // Atualizo a Listagem ...
62 } catch (Exception ex) {
63     ex.printStackTrace();
64 }
65     }
66 }

```



Para rodar a classe:

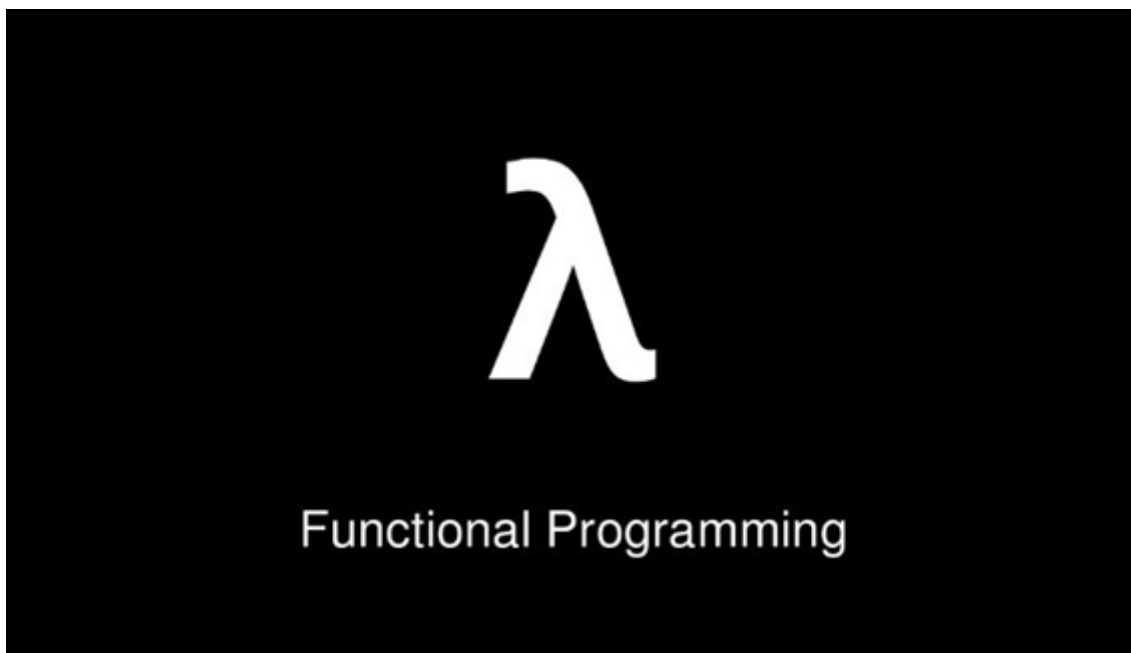
Clique na classe com o botão direito → run as → java application



👁 Resultado no console:

```
<terminated> Arquivo [Java Application] /opt/jre1.8.0_101/bin/java (15 de mar de 2020 10:13:46)
... Mensagens gravadas...
[Usuario [nome=mercador, mensagem=Coringa2, data=13/03/2020 08:24:52]
, Usuario [nome=Lucas, mensagem=CAVERA NAO TEM MEDO DE CORONA, data=13/03/2020 09:17:46]
, Usuario [nome=mercador, mensagem=Coringa3, data=13/03/2020 08:25:24]
, Usuario [nome=pessoa, mensagem=Boa Noite... ,data=13/03/2020 09:23:07]
, Usuario [nome=Davi, mensagem=
metade = (a) -> "Metade: " + (a/2);
parOuImpar = (a) -> (a % 2 == 0)?"par:"+a:"Impar: "+a;
raiz = (a) -> "Raiz: " + (Math.sqrt(a));
quadrado = (a) -> "Quadrado: " + (Math.pow(a, 2));
cubo = (a) -> "Cubo: " + (Math.pow(a, 3));,data=13/37/2020 09:37:19]
]
Gravando ...
... Novas Mensagens ...
[Usuario [nome=mercador, mensagem=Coringa2, data=13/03/2020 08:24:52]
, Usuario [nome=Lucas, mensagem=CAVERA NAO TEM MEDO DE CORONA, data=13/03/2020 09:17:46]
, Usuario [nome=mercador, mensagem=Coringa3, data=13/03/2020 08:25:24]
, Usuario [nome=pessoa, mensagem=Boa Noite... ,data=13/03/2020 09:23:07]
, Usuario [nome=Davi, mensagem=
metade = (a) -> "Metade: " + (a/2);
parOuImpar = (a) -> (a % 2 == 0)?"par:"+a:"Impar: "+a;
raiz = (a) -> "Raiz: " + (Math.sqrt(a));
quadrado = (a) -> "Quadrado: " + (Math.pow(a, 2));
cubo = (a) -> "Cubo: " + (Math.pow(a, 3));,data=13/37/2020 09:37:19]
, Usuario [nome=belem, mensagem=Sexta Treze... Tadam.. ,data=15/03/2020 10:13:46]
]
```

Programação Funcional



É um paradigma de programação, tal como Programação Orientado a Objetos(POO) e Programação Imperativa (existem outros paradigmas, mas esses são os mais famosos). Entende-se como paradigma uma forma de fazer algo. Ou seja, paradigma de programação é o nome que se dá a maneira como se programa, a orientação que seus códigos irão ter. POO, por exemplo, vai me ensinar a modelar meus códigos e algoritmos pensando em entidades que possuem características e comportamentos, vulgo objetos. Em Imperativa, vamos dar ao nosso programa uma sequência de passos para resolver determinado problema.

De maneira simples: código funcional é um código composto de múltiplas funções que se compõem para resolver um problema. Pense da seguinte forma: eu tenho um dado de entrada e preciso transformá-lo em um dado de saída. Usando PF eu vou abstrair as lógicas de transformações do meu código em funções, e usá-las no momento oportuno para transformar este meu dado.

Imutabilidade, pra que te quero?

Lá vai a primeira orientação: não use variáveis, use constantes! Sim, isso mesmo que eu falei, você não vai ter código com uma pancada de variáveis, você vai ter um código mais sucinto com constantes que, via de regra, não irão mudar! Parece coisa de maluco, mas vou te dizer alguns motivos de porquê um código imutável pode ser tão bom.

A imutabilidade faz sentido dentro da programação funcional pelo seu viés matemático. Nela, um número sempre será aquele valor, independente de onde esteja ou como está sendo usado. É importante também entender que nas expressões matemáticas, para um mesmo valor passado a uma variável, teremos o mesmo retorno da função. Ele nunca muda. Se você tem uma expressão como $f(x) = x + 2$, você pode passar o número 3 quantas vezes quiser, esta função sempre retornará 5. Um último ponto é que o número 3 passado para x , não irá mudar seu valor, ou seja, ele permanece inalterado após o seu uso na função.

Abstração acima de tudo!

Abstração é a capacidade de separar o essencial do que não é essencial.

Primeiramente, o que seria, em termos de código, uma abstração? Seria a implementação de uma função para um determinado bloco de código a ser utilizado e reutilizado. Quais as vantagens da abstração?

- Primeiro, você estará reaproveitando código. Menos código com mais informações. Isso é bom.
- O processo todo em que as funções foram abstraídas irão fazer sentido para você. Isso, alinhado com um bom nome para as funções, deixarão o seu código muito legível e de fácil compreensão.

Funções, funções, funções...

Não é a toa que programação funcional é a programação "orientada a funções". O conceito de função matemática permeia este paradigma, como expliquei quando falei de imutabilidade. Agora, se torna necessário eu trazer alguns conceitos importantes quando falamos de funções.

Funções puras

Bem, o conceito de funções puras tem tudo a ver com a não existência de efeitos colaterais. Funções puras são funções que não modificam o escopo ao redor delas. Quando eu aprendi um pouco da Linguagem C, aprendi um conceito muito interessante: ponteiro. E aprendi que é através deles que as funções em C podiam modificar variáveis fora de seu escopo interno. Bem, nós já vimos um exemplo de uma função impura e também um exemplo de uma função pura. Mas quais os princípios por trás? Uma função pura:


- Recebe ao menos um parâmetro e trabalha com ele.
- Ela retorna alguma coisa. É interessante que uma função, como nos ensinam, via de regra, é uma sequencia de procedimentos a serem executados. Até ai, tudo bem, porem, em programação funcional uma função sempre deve retornar um valor. Por que? Encadeamento de operações.

</> IFuncional.java

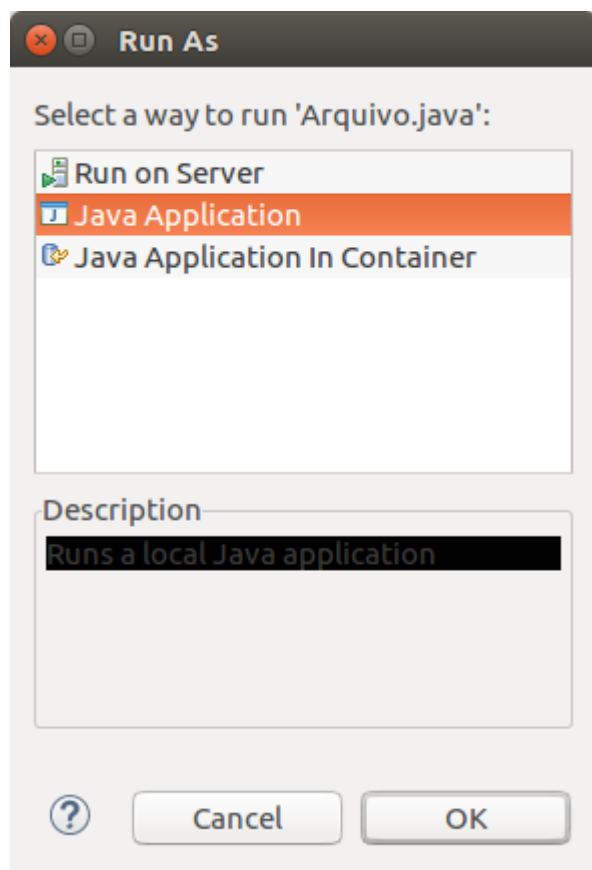
```
1 package java8;
2
3 @FunctionalInterface
4 public interface IFuncional {
5     // Programacao Funcional (MATEMATICA)
6     // saida texto //1 numero Entrada
7     public String operacao(Integer numero);
8 }
```

</> CalculoFuncional.java

```
1 package java8;
2
3 public class CalculoFuncional {
4     //Funcional (OO) Lambda
5
6     public static IFuncional metade = (a) -> "Metade:" + (a / 2);
7
8     public static IFuncional parouimpar = (a) -> (a % 2 == 0) ?
9         "par:" + a : "Impar :" + a;
10
11     public static IFuncional raiz = (a) -> "Raiz:" + (Math.sqrt(a));
12
13     public static void main(String[] args) {
14
15         System.out.println(metade.operacao(10));
16         System.out.println(parouimpar.operacao(4));
17         System.out.println(raiz.operacao(9));
18     }
19 }
```

 Para rodar a classe:

Clique na classe com o botão direito → run as → java application



👁 Resultado no console:

