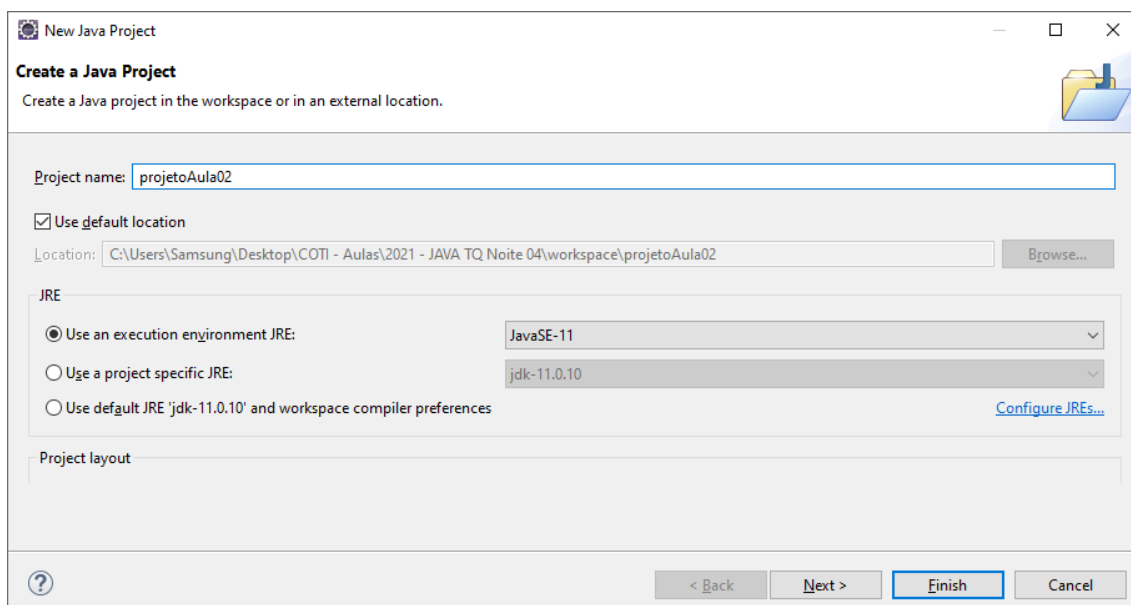
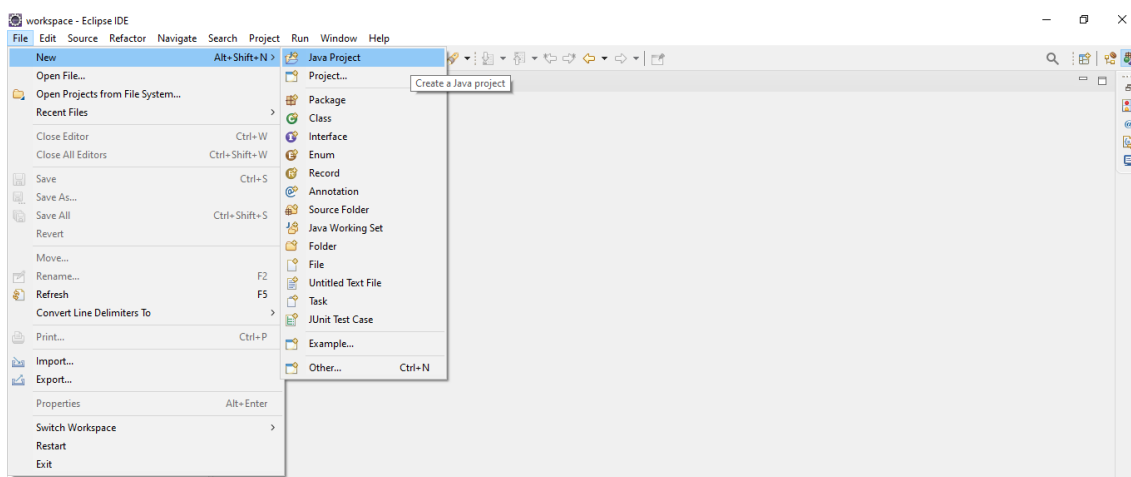


## Novo projeto:



## JAVABEANS

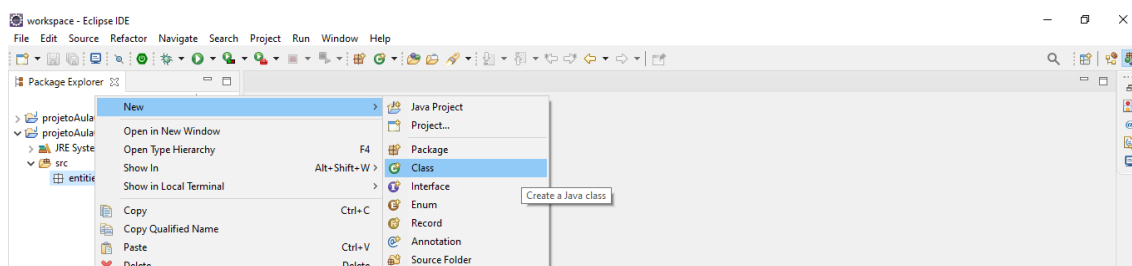
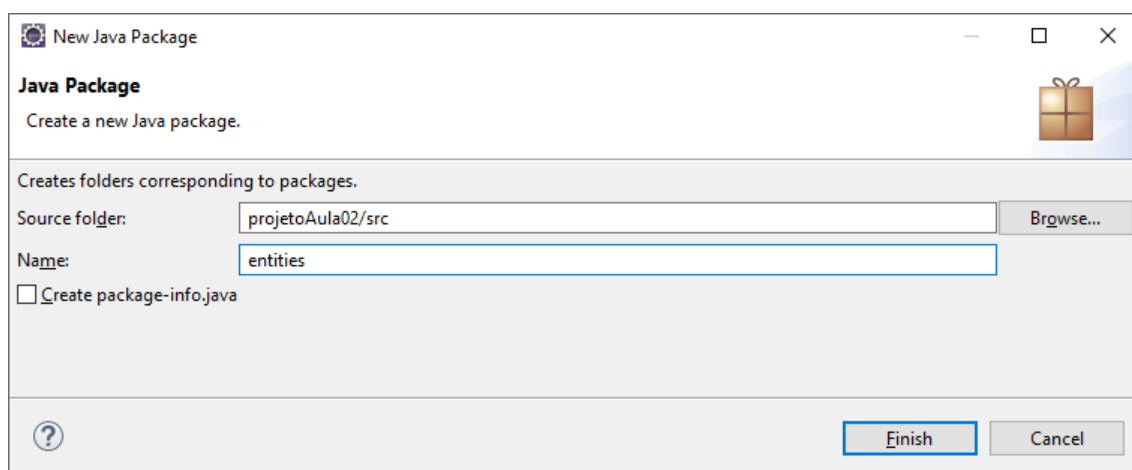
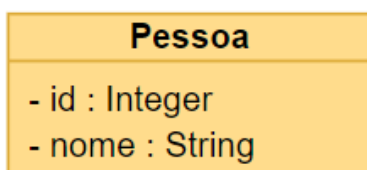
São classes utilizadas como **modelos de dados** em projetos Java, geralmente representam as entidades de um projeto.

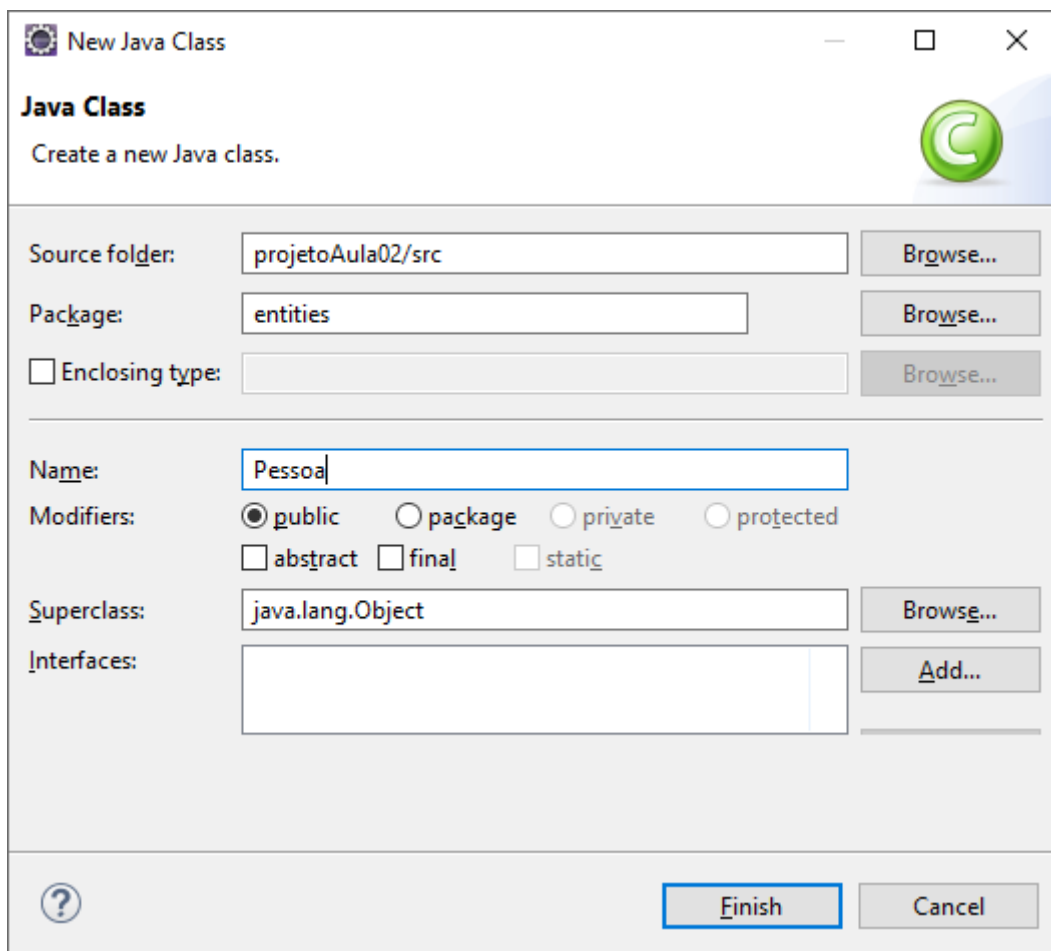
Características de uma classe JavaBean:

- Atributos privados.
- Métodos de encapsulamento para fazer a entrada e saída de dados de cada atributo, esses métodos são chamados de set e get.
- Sobrecarga de construtores
  - Método construtor sem argumentos
  - Método construtor recebendo como argumentos todos os atributos da classe.
- Sobrescrita de métodos
  - equals
  - hashCode
  - toString

Exemplo:

### /entities





**New Java Class**

Java Class  
Create a new Java class.

Source folder: projetoAula02/src Browse...

Package: entities Browse...

☐ Enclosing type: Browse...

Name: Pessoa

Modifiers: ☒ public ☐ package ☐ private ☐ protected  
☐ abstract ☐ final ☐ static

Superclass: java.lang.Object Browse...

Interfaces: Add...

? Finish Cancel

```
package entities;
```

```
//definição da classe
```

```
public class Pessoa {
```

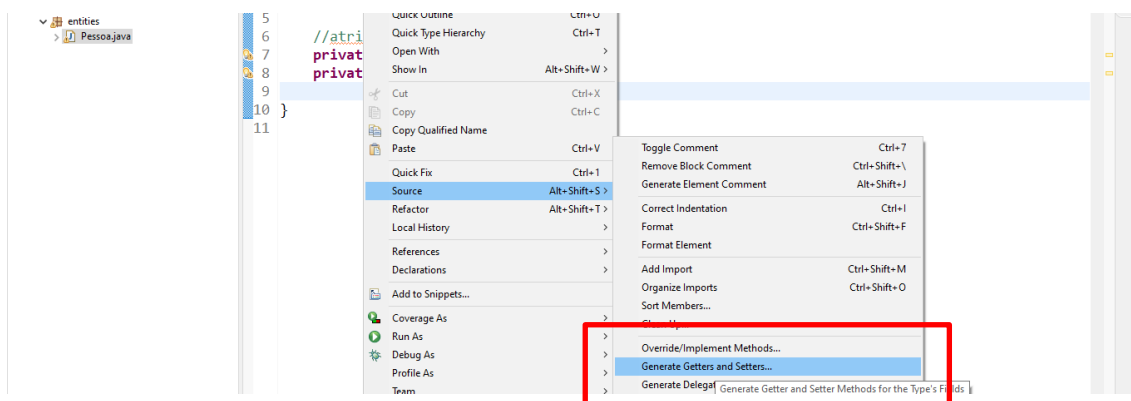
```
    //atributos (campos)
```

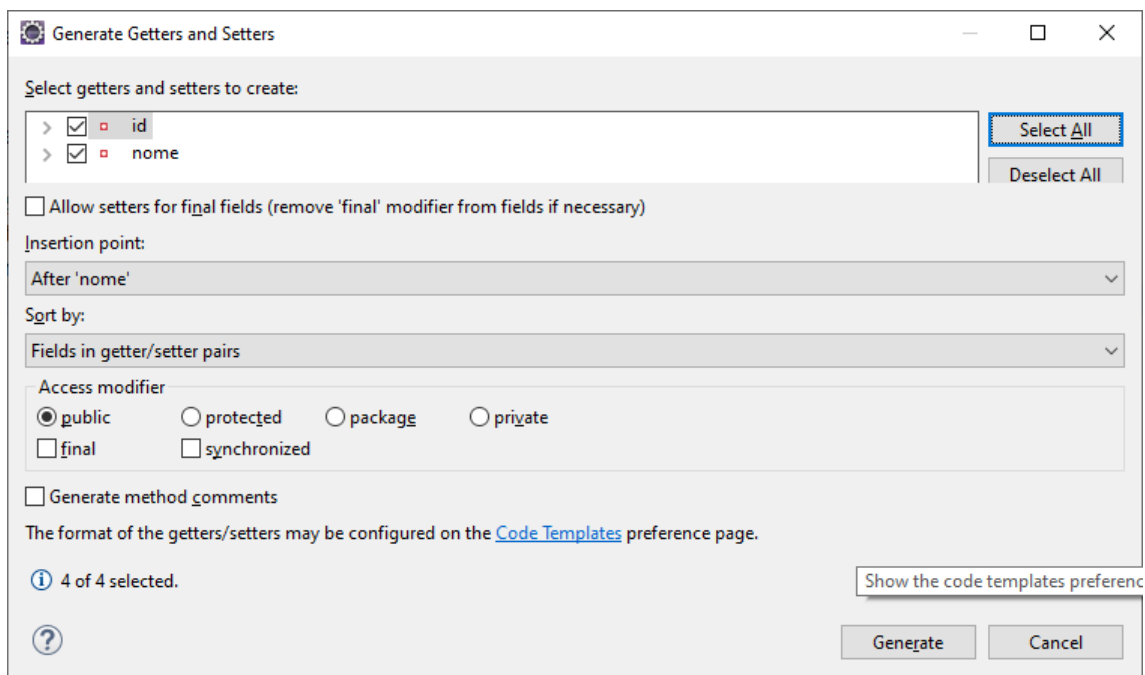
```
    private Integer id;
```

```
    private String nome;
```

```
}
```

**Gerando os métodos set e get:**





**package** entities;

//definição da classe

```
public class Pessoa {

    // atributos (campos)
    private Integer id;
    private String nome;

    public Integer getId() {
        return id;
    }

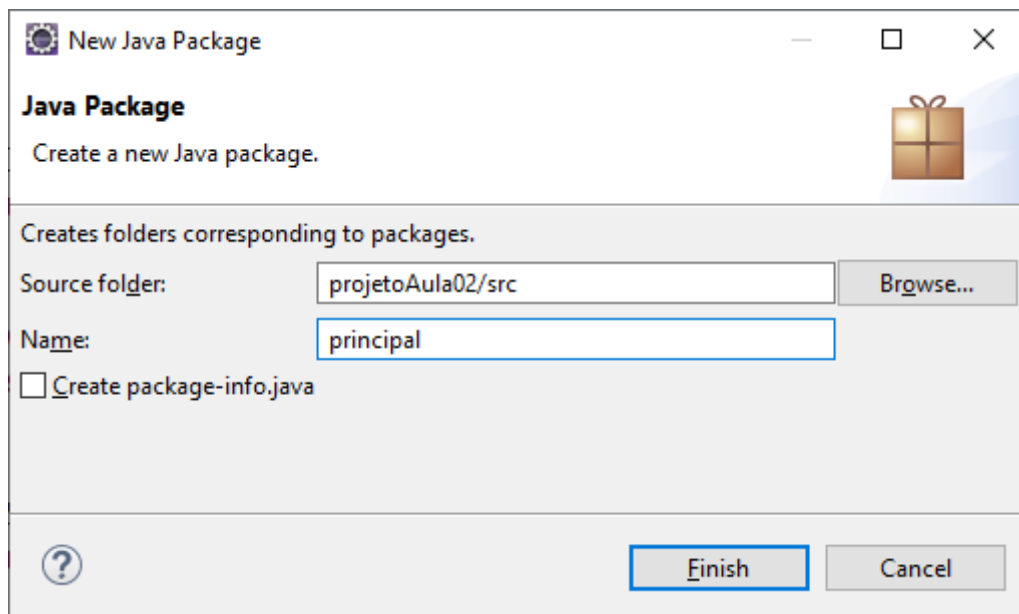
    public void setId(Integer id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

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

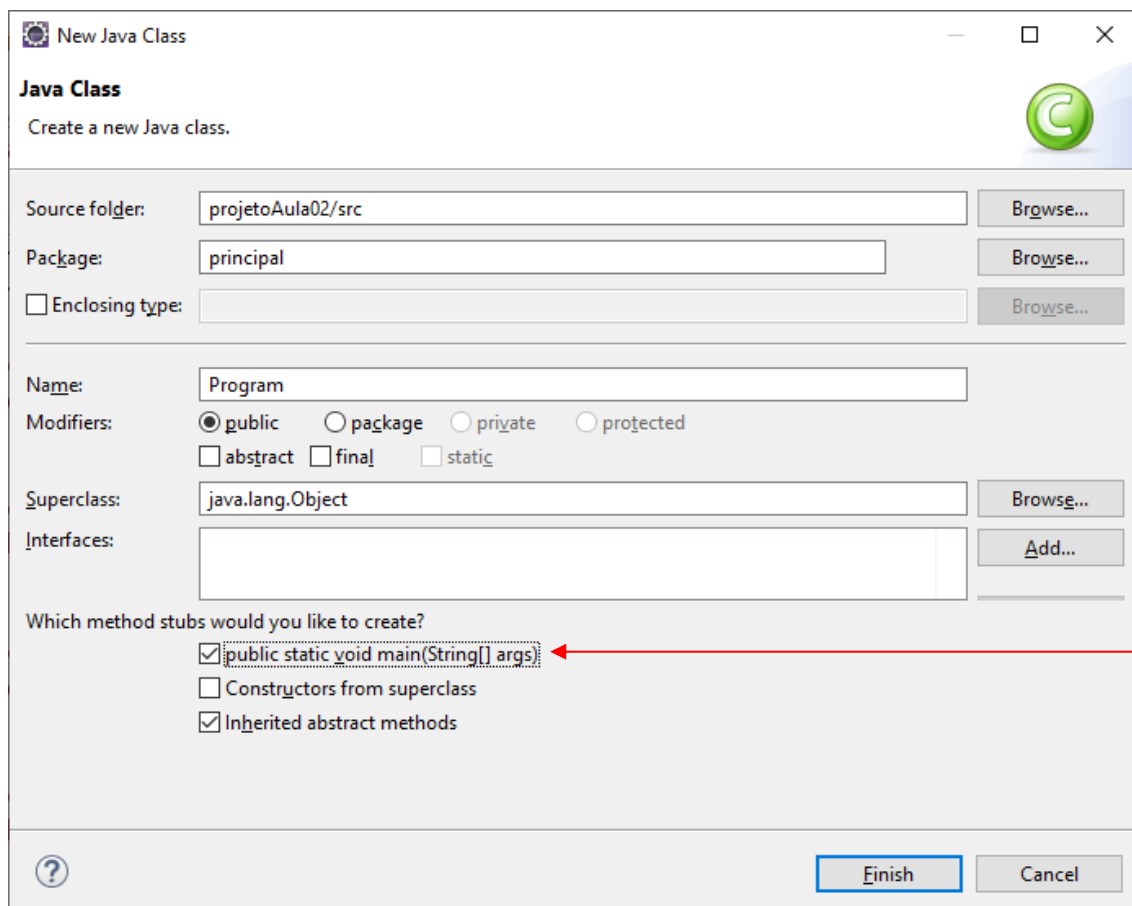
Iremos criar em seguida uma classe para que possamos executar o projeto.  
Esta classe deverá ter o método **void main()**

Criando o pacote:



/principal/**Program.java**

Já podemos criar esta classe com o método main().



## Variável de instância (**Objeto**)

Consiste em uma variável possui um tipo de Classe definido e é inicializada a partir do espaço de memória de uma classe. Por exemplo:

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

[Classe]      [Variável]      [Inicializando a variável]

Testando o uso dos métodos set e get:

```
package principal;
```

```
import entities.Pessoa;
```

```
public class Program {
```

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

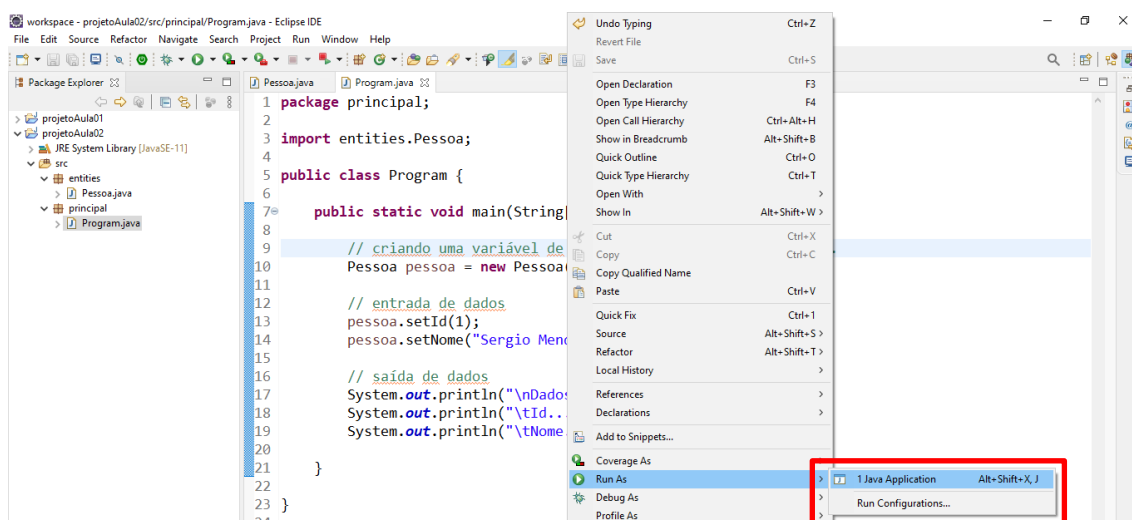
```
        // criando uma variável de
        // instância para a classe Pessoa..
        Pessoa pessoa = new Pessoa();
```

```
        // entrada de dados
        pessoa.setId(1);
        pessoa.setNome("Sergio Mendes");
```

```
        // saída de dados
        System.out.println("\nDados de Pessoa:\n");
        System.out.println("\tId.....: " + pessoa.getId());
        System.out.println("\tNome....: " + pessoa.getNome());
```

```
    }
}
```

## Executando:



## Método construtor

Consiste em um método utilizado pela classe para inicializar duas variáveis de instancia. Sempre que utilizamos a palavra reservada **new** estamos chamando o método construtor de uma classe.

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

[Classe]      [Variável]      [Método Construtor]

Toda classe já possui um método construtor implícito (não precisa estar escrito no código). Mas também podemos escrevê-lo.

Este construtor é chamado de **construtor sem argumentos**, pois não recebe nenhum parâmetro para fazer a inicialização da variável de instância.

Criando o método construtor na classe Pessoa:

```
package entities;
```

```
//definição da classe
```

```
public class Pessoa {
```

```
    // atributos (campos)
```

```
    private Integer id;
```

```
    private String nome;
```

```
    // construtor sem argumentos
```

```
    public Pessoa() {
```

```
        // TODO Auto-generated constructor stub
```

```
    }
```

```
    public Integer getId() {
```

```
        return id;
```

```
    }
```

```
    public void setId(Integer id) {
```

```
        this.id = id;
```

```
    }
```

```
    public String getNome() {
```

```
        return nome;
```

```
    }
```

```
    public void setNome(String nome) {
```

```
        this.nome = nome;
```

```
    }
```

```
}
```

## Sobrecarga de métodos (**OVERLOADING**)

Prática em POO (Programação Orientada a Objetos) onde declaramos em uma classe métodos com o mesmo nome, porém com entrada de argumentos diferentes. Por exemplo:

```
class Calculo {

    //método
    public int somar(int num1, int num2) {
        return num1 + num2;
    }

    //método
    public int somar(int num1, int num2, int num3) {
        return num1 + num2 + num3;
    }

}
```

A diferença entre os métodos não é o seu nome (ambos possuem o mesmo nome) mas sim a entrada de argumentos:

```
class Calculo {

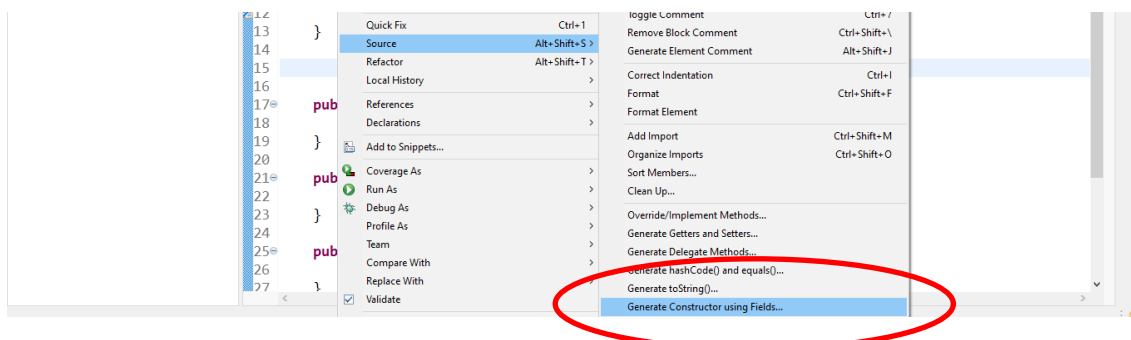
    //método
    public int somar(int num1, int num2) {
        return num1 + num2;
    }

    //método
    public int somar(int num1, int num2, int num3) {
        return num1 + num2 + num3;
    }

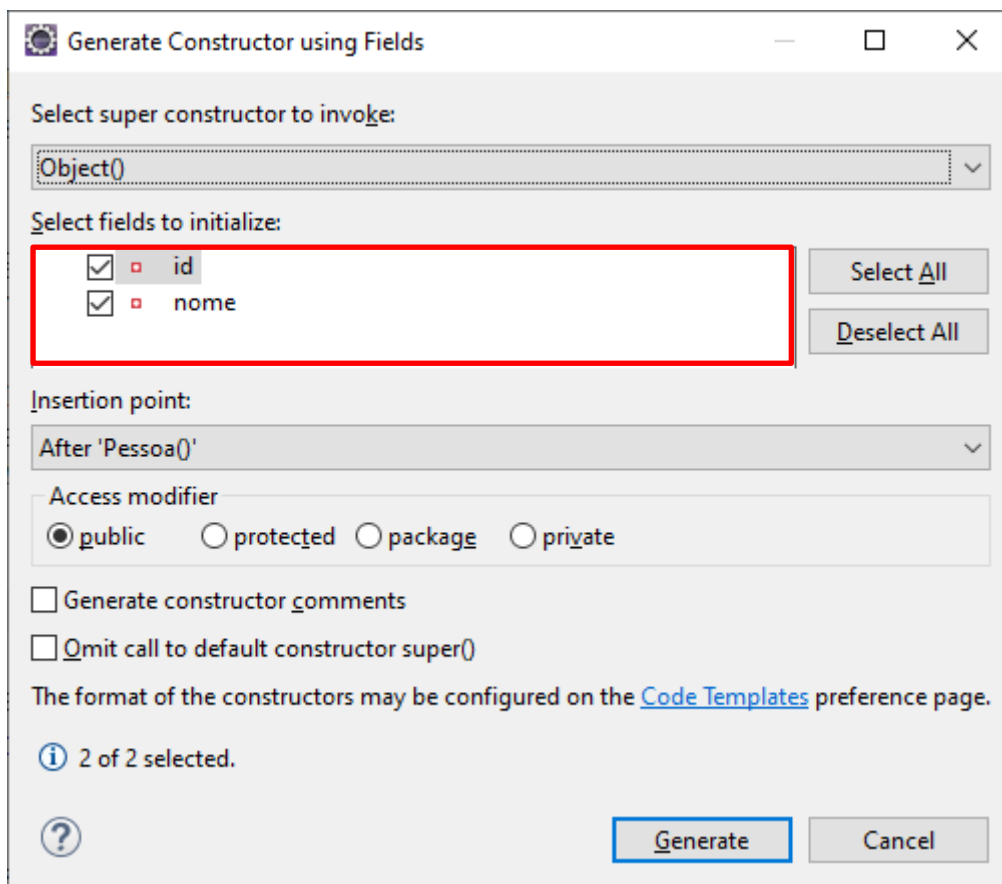
}
```

Uma classe JAVABEAN pode ter vários construtores, utilizando o conceito de **sobrecarga de métodos**, teríamos então uma sobrecarga de métodos construtores, começando pelo primeiro **construtor sem argumentos** e criando depois **construtores recebendo argumentos**.

### Exemplo:







`package` entities;

//definição da classe

`public class` Pessoa {

    // atributos (campos)

`private` Integer id;

`private` String nome;

    // construtor sem argumentos

`public` Pessoa() {

        // TODO Auto-generated constructor stub

    }

    // construtor com entrada de argumentos (sobrecarga de métodos)

`public` Pessoa(Integer id, String nome) {

`super`();

`this.id` = id;

`this.nome` = nome;

    }

`public` Integer getId() {

`return` id;

    }

```
public void setId(Integer id) {  
    this.id = id;  
}  
  
public String getNome() {  
    return nome;  
}  
  
public void setName(String nome) {  
    this.nome = nome;  
}  
}
```

Abaixo, temos uma **sobrecarga de métodos construtores** onde declaramos:

- Primeiro construtor sem argumentos
- Segundo construtor recebendo como argumentos todos os atributos da classe (id e nome)

```
package entities;
```

```
//definição da classe
```

```
public class Pessoa {
```

```
    // atributos (campos)
```

```
    private Integer id;
```

```
    private String nome;
```

```
    // construtor sem argumentos
```

```
    public Pessoa() {
```

```
        // TODO Auto-generated constructor stub
```

```
    }
```

```
    // construtor com entrada de argumentos
```

```
    // (sobrecarga de métodos)
```

```
    public Pessoa(Integer id, String nome) {
```

```
        super();
```

```
        this.id = id;
```

```
        this.nome = nome;
```

```
    }
```

```
    public Integer getId() {
```

```
        return id;
```

```
    }
```

```
    public void setId(Integer id) {
```

```
        this.id = id;
```

```
    }
```

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

## Testando o uso do construtor com entrada de argumentos:

```
package principal;  
  
import entities.Pessoa;  
  
public class Program {  
  
    public static void main(String[] args) {  
  
        // criando uma variável de instância  
        // para a classe Pessoa..  
        Pessoa pessoa = new Pessoa(1, "Sergio Mendes");  
  
        // saída de dados  
        System.out.println("\nDados de Pessoa:\n");  
        System.out.println("\tId.....: " + pessoa.getId());  
        System.out.println("\tNome....: " + pessoa.getNome());  
  
    }  
}
```

```
package entities;  
  
//definição da classe  
public class Pessoa {  
  
    // atributos (campos)  
    private Integer id;  
    private String nome;  
  
    // construtor sem argumentos  
    public Pessoa() {  
        // TODO Auto-generated constructor stub  
    }  
  
    // construtor com entrada de argumentos (sobrecarga de métodos)  
    public Pessoa(Integer id, String nome) {  
        super();  
        this.id = id;  
        this.nome = nome;  
    }  
}
```

```
public Integer getId() {  
    return id;  
}  
  
public void setId(Integer id) {  
    this.id = id;  
}  
  
public String getNome() {  
    return nome;  
}  
  
public void setNome(String nome) {  
    this.nome = nome;  
}  
}
```

De acordo com o padrão **JAVABEAN**, a classe Pessoa já atende aos seguintes requisitos:

- Atributos privados.
- Métodos de encapsulamento para fazer a entrada e saída de dados de cada atributo, esses métodos são chamados de set e get.
- Sobrecarga de construtores
  - Método construtor sem argumentos
  - Método construtor recebendo como argumentos todos os atributos da classe.

```
package entities;
```

```
//definição da classe
```

```
public class Pessoa {
```

```
    // atributos (campos)
```

```
    private Integer id;
```

```
    private String nome;
```

```
    // construtor sem argumentos
```

```
    public Pessoa() {
```

```
        // TODO Auto-generated constructor stub
```

```
    }
```

```
    // construtor com entrada de argumentos (sobrecarga de métodos)
```

```
    public Pessoa(Integer id, String nome) {
```

```
        super();
```

```
        this.id = id;
```

```
        this.nome = nome;
```

```
    }
```

```
    public Integer getId() {
```

```
        return id;
```

```
    }
```

```

public void setId(Integer id) {
    this.id = id;
}

public String getNome() {
    return nome;
}

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

```

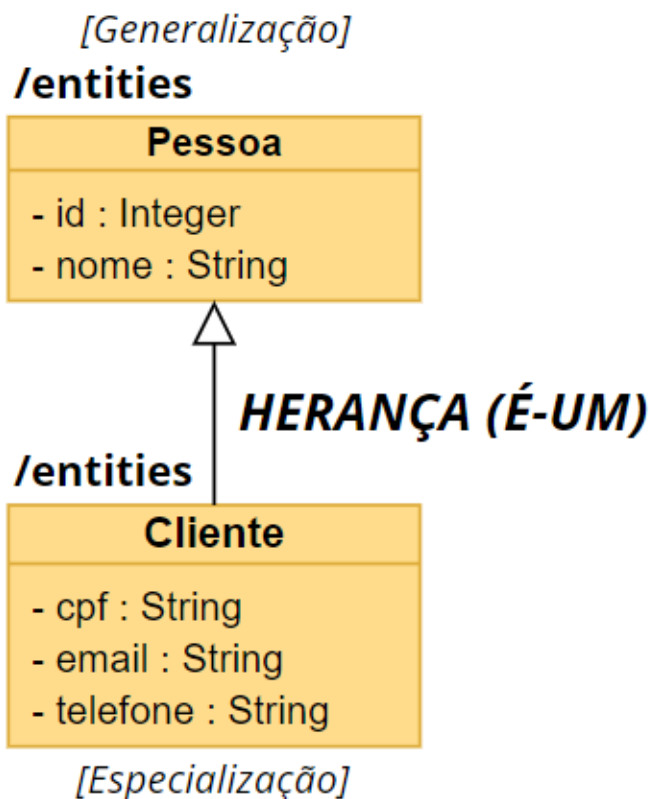
## Relacionamentos entre classes:

Em POO, podemos relacionar classes de 2 maneiras: HERANÇA ou ASSOCIAÇÃO. Basicamente a diferença é a seguinte:

### HERANÇA (É-UM)

É um tipo de relacionamento entre classes que define o uso do verbo "SER". Define uma relação de superclasse / subclasse, ou seja, generalização / especialização.

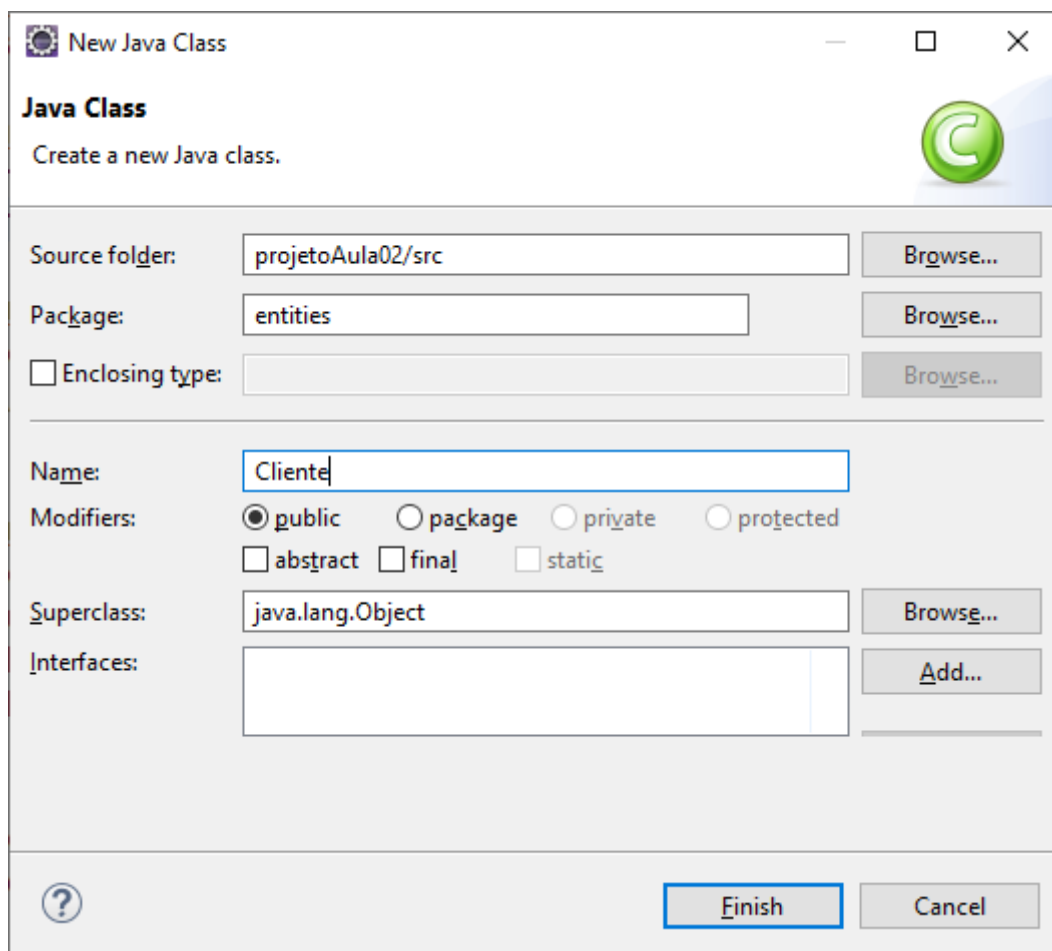
- Exemplo:**



## Criando um JAVABEAN Cliente:

Como um JAVABEAN, a classe Cliente deverá ter:

- Atributos privados.
- Métodos de encapsulamento para fazer a entrada e saída de dados de cada atributo, esses métodos são chamados de set e get.
- Sobrecarga de construtores
  - Método construtor sem argumentos
  - Método construtor recebendo como argumentos todos os atributos da classe.



The screenshot shows the 'New Java Class' dialog box. The 'Source folder' is 'projetoAula02/src', 'Package' is 'entities', 'Name' is 'Cliente', 'Modifiers' are 'public', 'Superclass' is 'java.lang.Object', and 'Interfaces' is empty. The 'Finish' button is highlighted.

```
package entities;
```

```
public class Cliente {
```

```
    // atributos
```

```
    private String cpf;
```

```
    private String telefone;
```

```
    private String email;
```

```
    // construtor sem argumentos
```

```
    public Cliente() {
```

```
        // TODO Auto-generated constructor stub
```

```
    }
```

```
public Cliente(String cpf, String telefone, String email) {
    super();
    this.cpf = cpf;
    this.telefone = telefone;
    this.email = email;
}

public String getCpf() {
    return cpf;
}

public void setCpf(String cpf) {
    this.cpf = cpf;
}

public String getTelefone() {
    return telefone;
}

public void setTelefone(String telefone) {
    this.telefone = telefone;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}
}
```

Definindo a herança entre as classes:

**extends**

```
package entidades;
```

```
public class Cliente extends Pessoa {
```

```
    // atributos
    private String cpf;
    private String telefone;
    private String email;

    // construtor sem argumentos
    public Cliente() {
        // TODO Auto-generated constructor stub
    }

    public Cliente(String cpf, String telefone, String email) {
        super();
        this.cpf = cpf;
        this.telefone = telefone;
    }
}
```

```
        this.email = email;
    }

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

---

## Regras sobre HERANÇA:

- Em Java, não podemos fazer HERANÇA MULTIPLA entre classes, ou seja, uma subclasse só pode ter uma única superclasse.

Exemplo:

```
class A {
}

class B {
}

class C extends A, B {
}
```

↓  
Java não permite herança múltipla entre classes.



- Em Java, se uma classe é declarada com a palavra reservada **final** ela não pode ser herdada, ou seja, não pode ter subclasses:

```
class A {
}

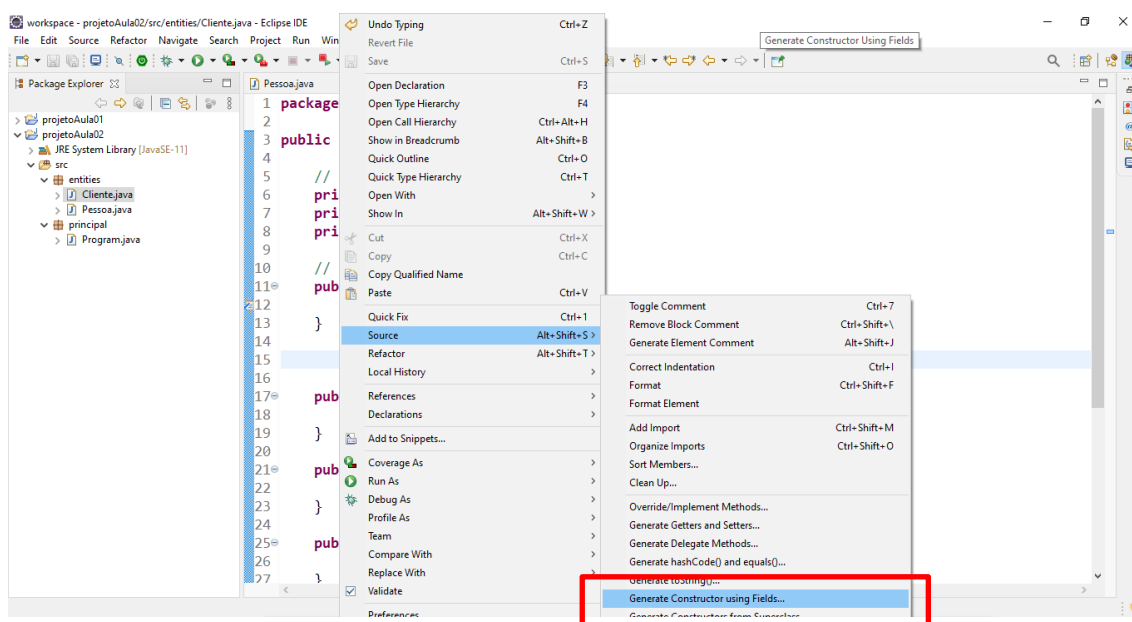
final class B extends A{
}

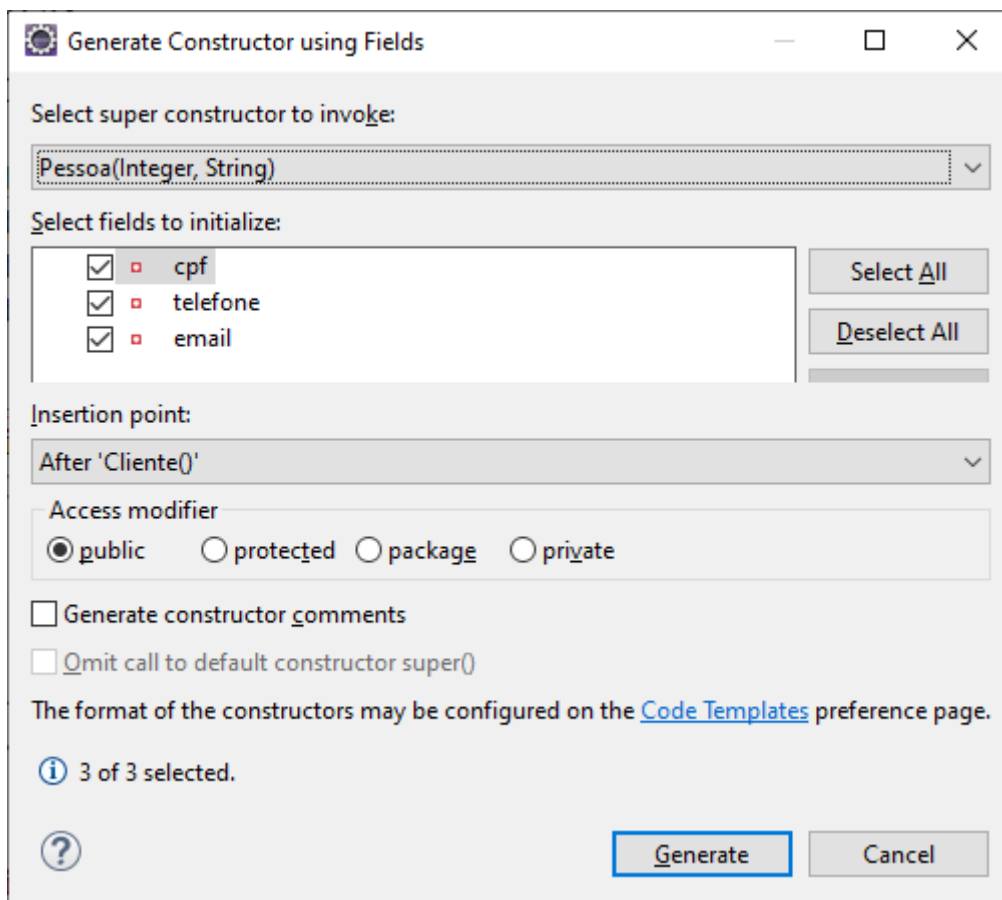
class C extends B{
}
```

A classe C não pode herdar a classe B, pois a classe B foi declarada como **final**, ou seja, a classe B não pode ter subclasses.

Quando utilizamos HERANÇA, também podemos fazer com que os métodos construtores de uma subclasse possam herdar os métodos construtores da sua superclasse.

## Exemplo:





package entities;

public class Cliente extends Pessoa {

// atributos

private String cpf;

private String telefone;

private String email;

// construtor sem argumentos

public Cliente() {

// TODO Auto-generated constructor stub

}

// construtor com entrada de argumentos

public Cliente(Integer id, String nome, String cpf,  
String telefone, String email) {

super(id, nome);

this.cpf = cpf;

this.telefone = telefone;

this.email = email;

}

```
public String getCpf() {  
    return cpf;  
}  
  
public void setCpf(String cpf) {  
    this.cpf = cpf;  
}  
  
public String getTelefone() {  
    return telefone;  
}  
  
public void setTelefone(String telefone) {  
    this.telefone = telefone;  
}  
  
public String getEmail() {  
    return email;  
}  
  
public void setEmail(String email) {  
    this.email = email;  
}  
}
```

Note que, o construtor com entrada de argumentos está fazendo uma chamada para o construtor da superclasse, através da palavra reservada **super**:

## super

Palavra reservada que faz uma chamada a métodos que estão na superclasse (Classe pai).

```
package entities;
```

```
public class Cliente extends Pessoa {  
  
    // atributos  
    private String cpf;  
    private String telefone;  
    private String email;  
  
    // construtor sem argumentos  
    public Cliente() {  
        // TODO Auto-generated constructor stub  
    }  
  
    // construtor com entrada de argumentos  
    public Cliente(Integer id, String nome, String cpf,  
                    String telefone, String email) {  
        super(id, nome);  
    }  
}
```

```
        this.cpf = cpf;
        this.telefone = telefone;
        this.email = email;
    }

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

---

Testando o uso da classe Cliente:

/principal/**Program.java**

```
package principal;

import entities.Cliente;

public class Program {

    public static void main(String[] args) {

        // criando uma variável de instância
        // para a classe Cliente..
        Cliente cliente = new Cliente(1, "Ana Paula",
            "123.456.789-00", "(21) 99999-9999",
            "anapaula@gmail.com");
    }
}
```

## Sobrescrita de métodos (**OVERRIDE**)

Ocorre quando uma subclasse substitui / sobrepõe métodos da sua superclasse, de forma a redefinir o comportamento destes métodos.

Exemplo:

Note que, no programa abaixo, a classe B pode executar o método imprimir() da classe A pois o mesmo é acessado por herança:

```
package principal;
```

```
class A {
```

```
    public void imprimir() {  
        System.out.println("Hello, A!");  
    }
```

```
}
```

```
class B extends A{
```

```
}
```

```
public class Test {
```

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

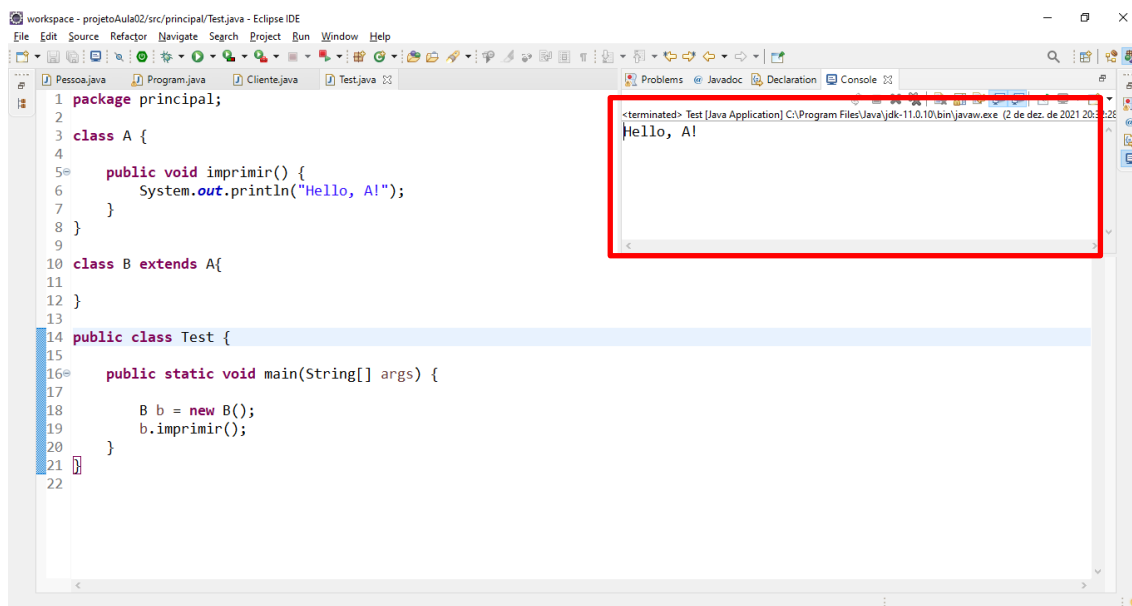
```
        B b = new B();
```

```
        b.imprimir();
```

```
    }
```

```
}
```

## Resultado da execução:



A classe B pode **sobrescrever o método** imprimir() herdado da classe A, dando a ele um novo comportamento. Por exemplo:

```
package principal;

class A {

    public void imprimir() {
        System.out.println("Hello, A!");
    }
}

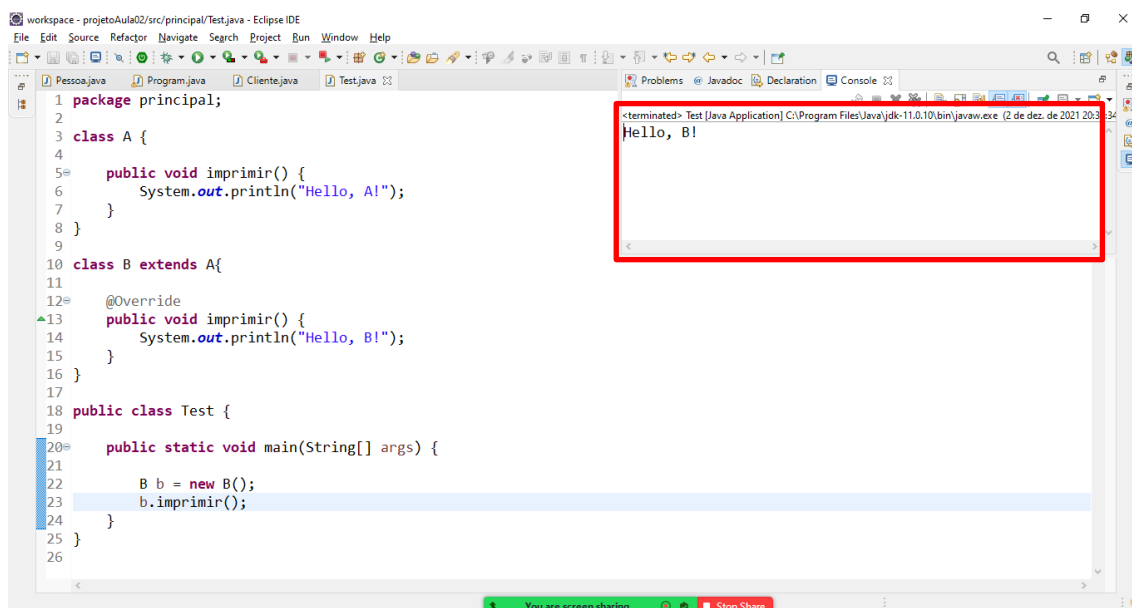
class B extends A{

    @Override
    public void imprimir() {
        System.out.println("Hello, B!");
    }
}

public class Test {

    public static void main(String[] args) {

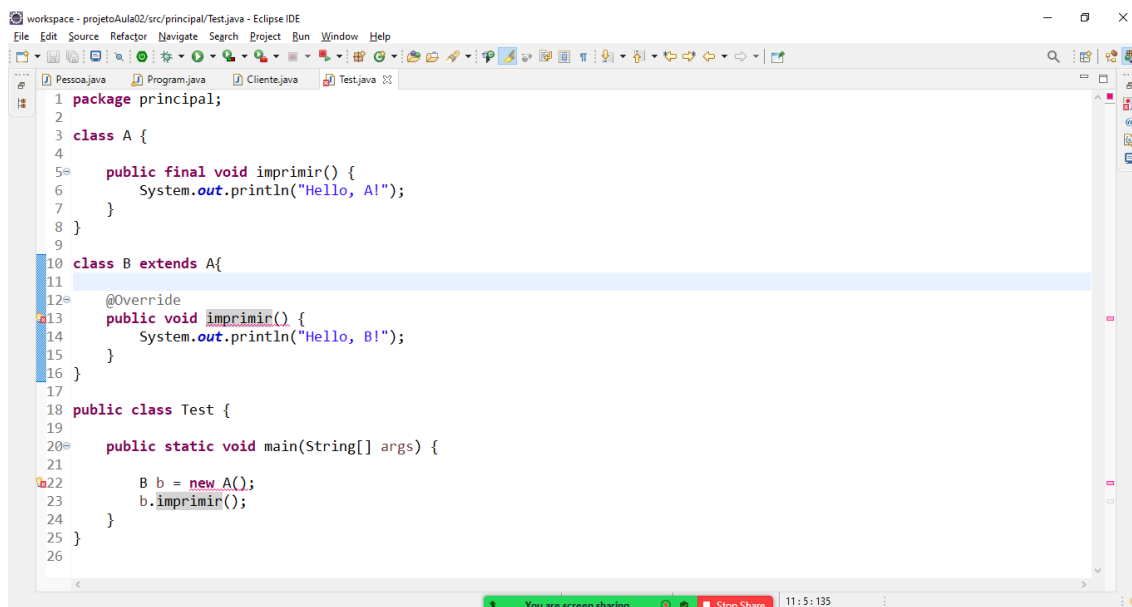
        B b = new B();
        b.imprimir();
    }
}
```



Note que no resultado acima, a classe B substituiu o método imprimir() da classe A, dando a ele uma nova implementação. Esta ação é chamada de **Sobrescrita de métodos**.

## Curiosidade:

Se uma superclasse quiser proibir um dos seus métodos de ser sobrescrito por uma subclasse, basta ela definir o método como **final**.



```

1 package principal;
2
3 class A {
4
5     public final void imprimir() {
6         System.out.println("Hello, A!");
7     }
8 }
9
10 class B extends A{
11
12     @Override
13     public void imprimir() {
14         System.out.println("Hello, B!");
15     }
16 }
17
18 public class Test {
19
20     public static void main(String[] args) {
21
22         B b = new A();
23         b.imprimir();
24     }
25 }
26
  
```

```
package principal;
```

```
class A {
```

```
    public final void imprimir() {
        System.out.println("Hello, A!");
    }
```

```
}
```

```
class B extends A{
```

```
@Override
```

```
public void imprimir() {
```

```
    System.out.println("Hello, B!");
```

```
}
```

```
}
```

```
public class Test {
```

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

```
        B b = new A();
```

```
        b.imprimir();
```

```
    }
```

```
}
```

Exemplo de um código que contém **SOBRECARGA** de métodos e também **SOBRESCRITA** de métodos:

```
package principal;

class Pagamento {

    public void realizarPagamento(Double valor) {
        System.out.println("Pagamento realizado: " + valor);
    }
}

class PagamentoParcelado extends Pagamento {

    @Override //Sobrescrita de método
    public void realizarPagamento(Double valor) {
        Double parcela = valor / 3;
        System.out.println("Pagamento realizado em
                           3 parcelas de: " + parcela);
    }

    //Sobrecarga de método (argumentos diferentes
    //do método anterior)
    public void realizarPagamento(Double valor, Integer qtd) {
        Double parcela = valor / qtd;
        System.out.println("Pagamento realizado em "
                           + qtd + " parcelas de: " + parcela);
    }
}

public class Test {

    public static void main(String[] args) {

        PagamentoParcelado pagamento = new PagamentoParcelado();
        pagamento.realizarPagamento(900.0, 6);
    }
}
```

---

Em JAVA, toda classe criada implicitamente HERDA uma superclasse denominada **Object** (**java.lang.Object**)

### Object

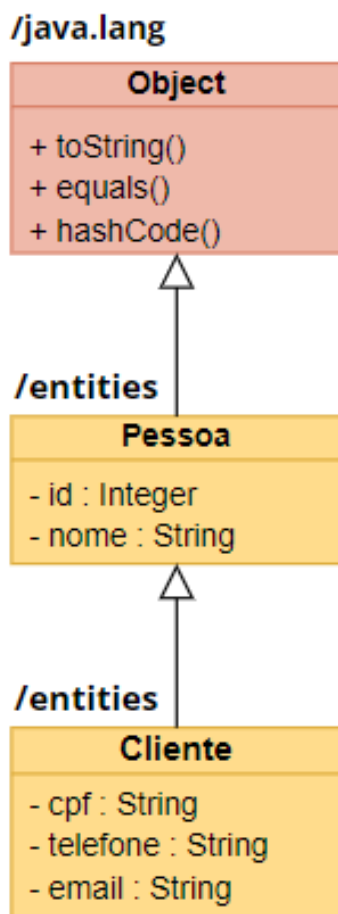
É o nome da classe PAI de todas as classes criadas em JAVA.

Portanto, toda classe Java já possui métodos como: toString(), equals(), hashCode(), clone(), etc...



Portanto, nós podemos **sobrescrever** métodos da classe **Object**, tais como:

- **toString()**  
Método utilizado para retornar todos os atributos da classe em uma única linha de texto, voltado para impressão dos dados.
- **equals()**  
Método utilizado para comparar se dois objetos de uma mesma classe são iguais baseado em um critério definido pelo programador.
- **hashCode()**  
Método utilizado para fazer a ordenação de muitos objetos de uma classe quando estes estiverem sendo agrupados por uma collection do java.



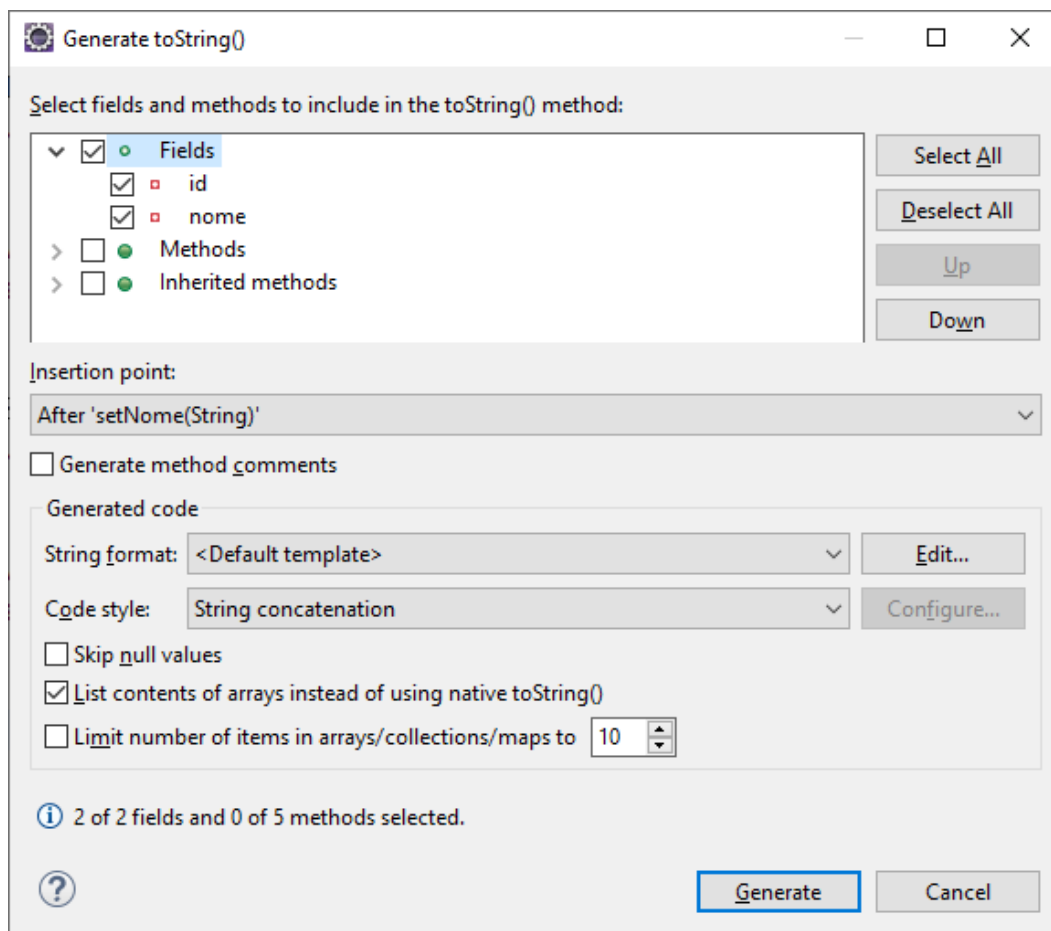
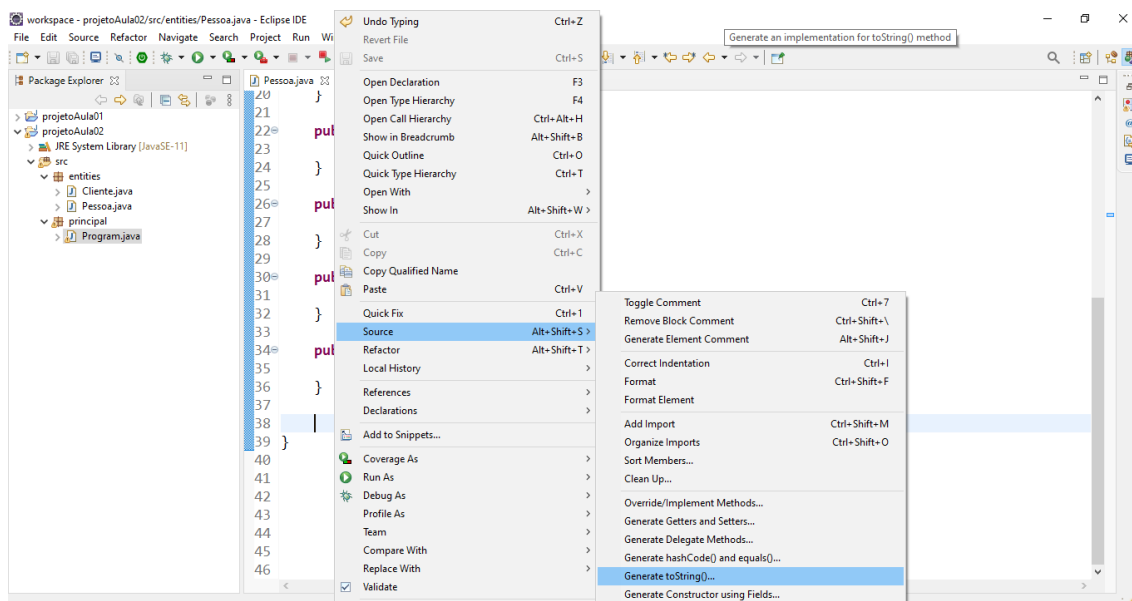
Uma das características de uma classe JAVABEAN é sobrescrever os métodos da classe **Object**, tais como `toString()`, `equals()` e `hashCode()`.

Neste exemplo iremos sobrescrever o método `toString()` da classe **Object** com o objetivo de criar uma função que imprima em uma única linha de texto todos os atributos da classe.

## Sobrescrevendo o método toString()

O método toString() é utilizado para imprimir os dados de uma classe em uma única linha de texto.

/entities/**Pessoa.java**



- Atributos privados.
- Métodos de encapsulamento para fazer a entrada e saída de dados de cada atributo, esses métodos são chamados de set e get.
- Sobrecarga de construtores
  - Método construtor sem argumentos
  - Método construtor recebendo como argumentos todos os atributos da classe.
- Sobrescrita de métodos
  - toString

```
package entities;
```

```
//definição da classe
```

```
public class Pessoa {
```

```
    // atributos (campos)
```

```
    private Integer id;
```

```
    private String nome;
```

```
    // construtor sem argumentos
```

```
    public Pessoa() {
```

```
        // TODO Auto-generated constructor stub
```

```
    }
```

```
    // construtor com entrada de argumentos (sobrecarga de métodos)
```

```
    public Pessoa(Integer id, String nome) {
```

```
        super();
```

```
        this.id = id;
```

```
        this.nome = nome;
```

```
    }
```

```
    public Integer getId() {
```

```
        return id;
```

```
    }
```

```
    public void setId(Integer id) {
```

```
        this.id = id;
```

```
    }
```

```
    public String getNome() {
```

```
        return nome;
```

```
    }
```

```
    public void setNome(String nome) {
```

```
        this.nome = nome;
```

```
    }
```

```
    @Override
```

```
    public String toString() {
```

```
        return "Pessoa [id=" + id + ", nome=" + nome + "];
```

```
    }
```

```
}
```

```
package entities;

public class Cliente extends Pessoa {

    // atributos
    private String cpf;
    private String telefone;
    private String email;

    // construtor sem argumentos
    public Cliente() {
    }

    // construtor com entrada de argumentos
    public Cliente(Integer id, String nome, String cpf,
                    String telefone, String email) {
        super(id, nome);
        this.cpf = cpf;
        this.telefone = telefone;
        this.email = email;
    }

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    @Override
    public String toString() {
        return super.toString() + ", Cliente [cpf=" + cpf
            + ", telefone=" + telefone + ", email=" + email + "];"
    }
}
```

Testando na classe **Program.java**:

```
package principal;

import entities.Cliente;

public class Program {

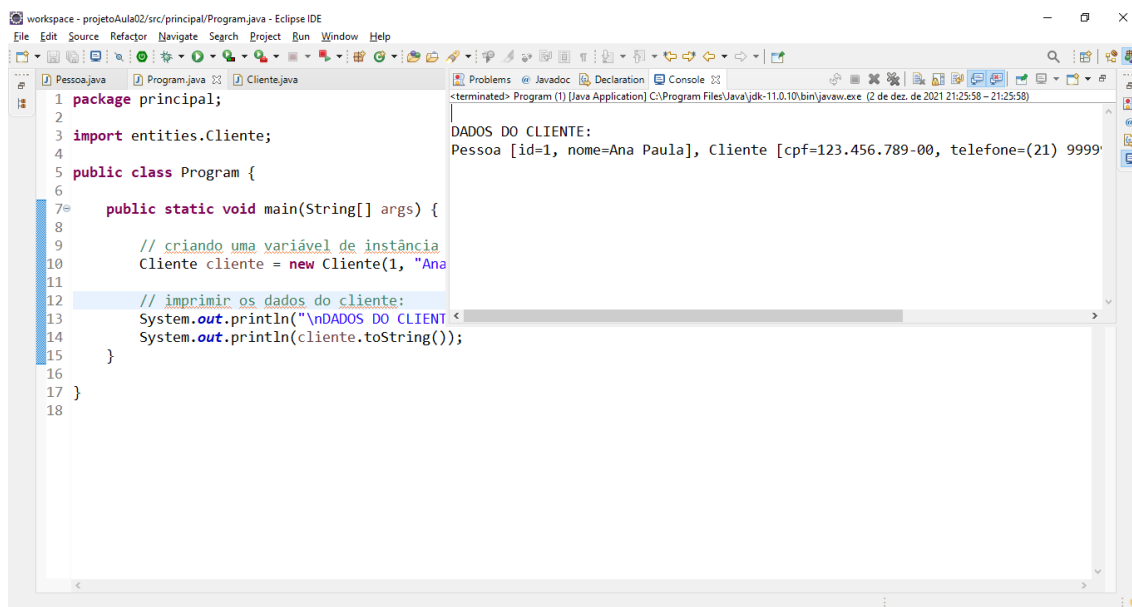
    public static void main(String[] args) {

        // criando uma variável de instância
        // para a classe Cliente..
        Cliente cliente = new Cliente(1, "Ana Paula",
                                     "123.456.789-00", "(21) 99999-9999",
                                     "anapaula@gmail.com");

        // imprimir os dados do cliente:
        System.out.println("\nDADOS DO CLIENTE:");
        System.out.println(cliente.toString());

    }
}
```

**Resultado:**



The screenshot shows the Eclipse IDE with the 'Program.java' file open. The code in the editor matches the one provided in the previous block. The console window on the right shows the output of the program:

```
<terminated> Program (1) [Java Application] C:\Program Files\Java\jdk-11.0.10\bin\javaw.exe (2 de dez. de 2021 21:25:58)

DADOS DO CLIENTE:
Pessoa [id=1, nome=Ana Paula], Cliente [cpf=123.456.789-00, telefone=(21) 9999
```

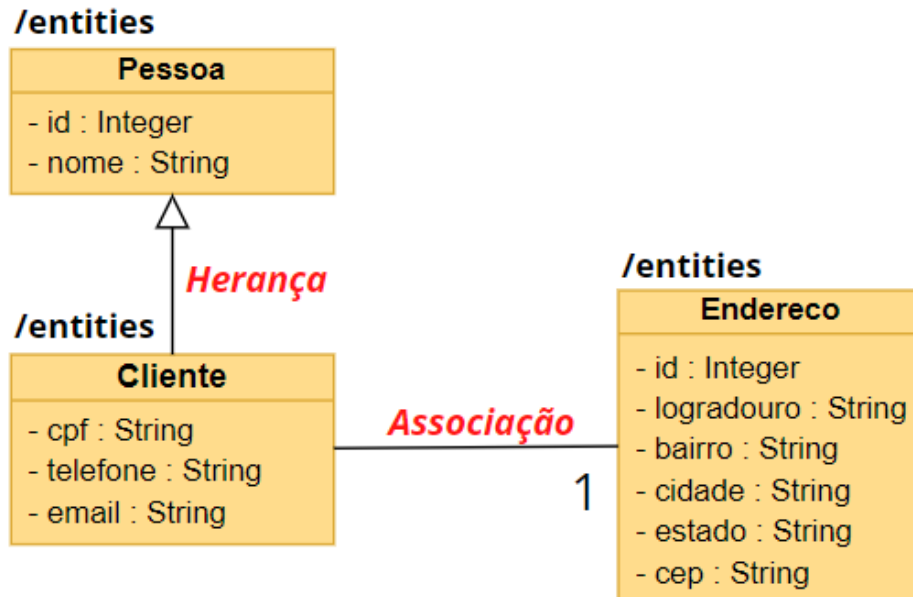
**DADOS DO CLIENTE:**

Pessoa [id=1, nome=Ana Paula], Cliente  
[cpf=123.456.789-00, telefone=(21) 99999-9999,  
email=anapaula@gmail.com]

## Relacionamentos entre classes:

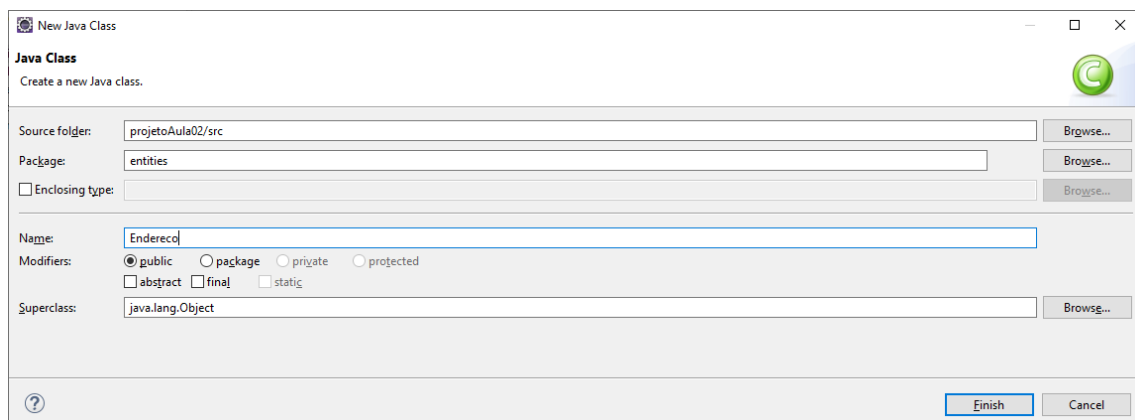
### ASSOCIAÇÃO (TER)

É um tipo de relacionamento entre classes que define um vínculo de agregação ou composição entre classes (utilização) que pode ser do tipo **TER-1** ou **TER-MUITOS**.



Criando a classe **Endereco** como um **JAVABEAN** com as seguintes características:

- **Atributos privados.**
- **Métodos de encapsulamento** para fazer a entrada e saída de dados de cada atributo, esses métodos são chamados de `set` e `get`.
- **Sobrecarga** de construtores
  - Método **construtor sem argumentos**
  - Método **construtor recebendo argumentos**
- **Sobrescrita** de métodos
  - **`toString`**



```
package entities;

public class Endereco {

    private Integer id;
    private String logradouro;
    private String bairro;
    private String cidade;
    private String estado;
    private String cep;

    public Endereco() {
        // TODO Auto-generated constructor stub
    }

    public Endereco(Integer id, String logradouro,
        String bairro, String cidade, String estado, String cep) {
        super();
        this.id = id;
        this.logradouro = logradouro;
        this.bairro = bairro;
        this.cidade = cidade;
        this.estado = estado;
        this.cep = cep;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getLogradouro() {
        return logradouro;
    }

    public void setLogradouro(String logradouro) {
        this.logradouro = logradouro;
    }

    public String getBairro() {
        return bairro;
    }

    public void setBairro(String bairro) {
        this.bairro = bairro;
    }
}
```

```

public String getCidade() {
    return cidade;
}

public void setCidade(String cidade) {
    this.cidade = cidade;
}

public String getEstado() {
    return estado;
}

public void setEstado(String estado) {
    this.estado = estado;
}

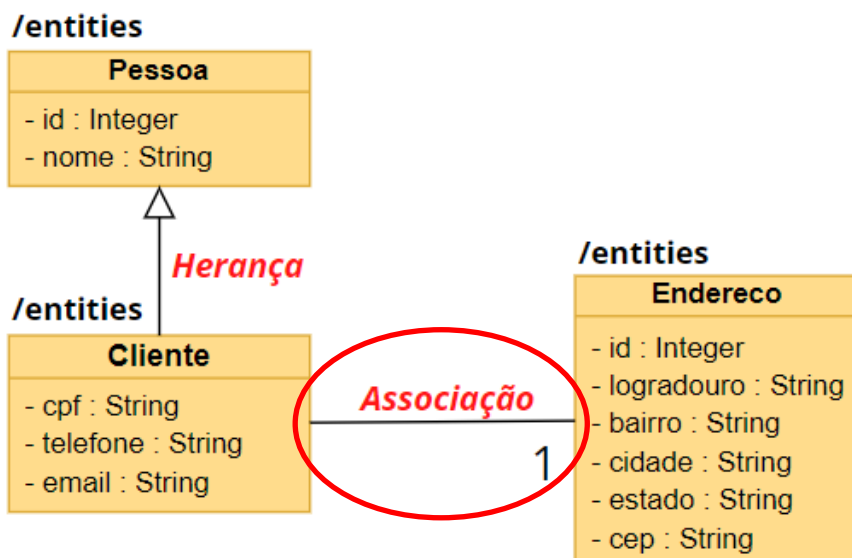
public String getCep() {
    return cep;
}

public void setCep(String cep) {
    this.cep = cep;
}

@Override
public String toString() {
    return "Endereco [id=" + id + ", logradouro=" + logradouro
        + ", bairro=" + bairro + ", cidade=" + cidade
        + ", estado=" + estado + ", cep=" + cep + "];"
}
}

```

## Fazendo o relacionamento de associação de Cliente com Endereco:





/entities/**Cliente.java**

```
package entities;

public class Cliente extends Pessoa {

    // atributos
    private String cpf;
    private String telefone;
    private String email;
    private Endereco endereco;

    // construtor sem argumentos
    public Cliente() {
        // TODO Auto-generated constructor stub
    }

    // construtor com entrada de argumentos
    public Cliente(Integer id, String nome, String cpf,
        String telefone, String email) {
        super(id, nome);
        this.cpf = cpf;
        this.telefone = telefone;
        this.email = email;
    }

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public String getTelefone() {
        return telefone;
    }

    public void setTelefone(String telefone) {
        this.telefone = telefone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

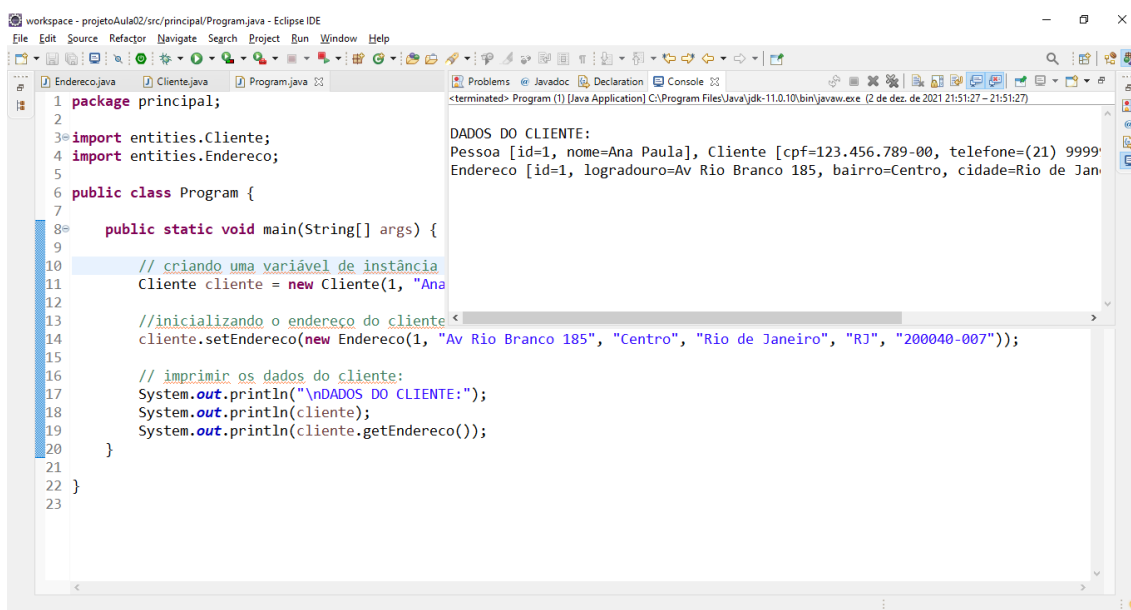
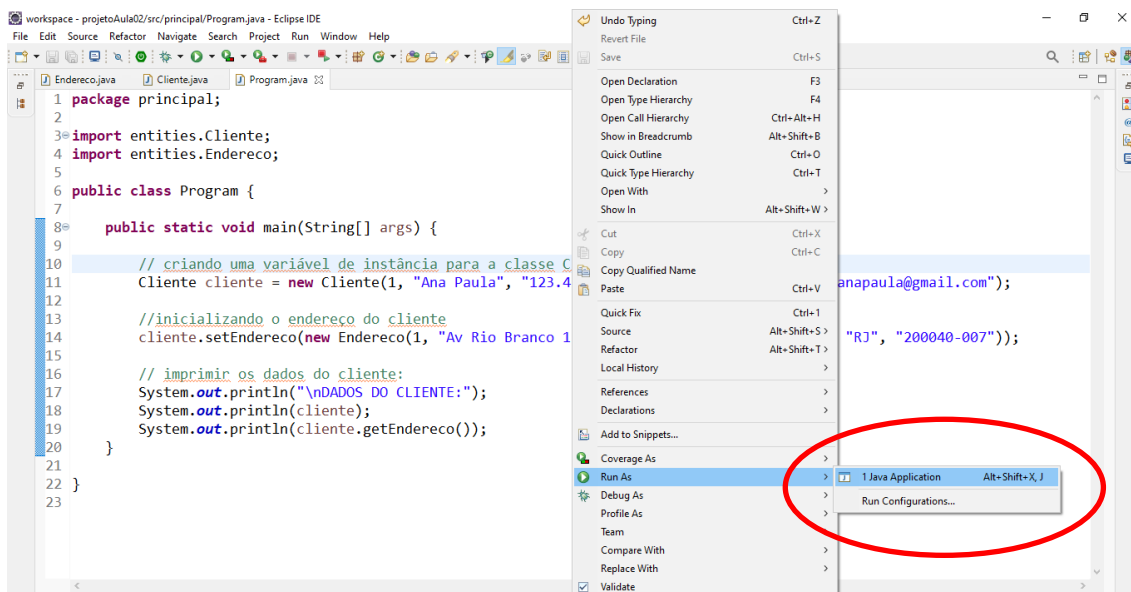
```
public Endereco getEndereco() {  
    return endereco;  
}  
  
public void setEndereco(Endereco endereco) {  
    this.endereco = endereco;  
}  
  
@Override  
public String toString() {  
    return super.toString() + ", Cliente [cpf=" + cpf  
        + ", telefone=" + telefone + ", email=" + email + "];  
}  
}
```

---

Voltando na classe **Program.java**

```
package principal;  
  
import entities.Cliente;  
import entities.Endereco;  
  
public class Program {  
  
    public static void main(String[] args) {  
  
        // criando uma variável de instância  
        // para a classe Cliente..  
        Cliente cliente = new Cliente(1, "Ana Paula",  
            "123.456.789-00", "(21) 99999-9999",  
            "anapaula@gmail.com");  
  
        //inicializando o endereço do cliente  
        cliente.setEndereco(new Endereco(1, "Av Rio Branco 185",  
            "Centro", "Rio de Janeiro", "RJ", "200040-007"));  
  
        // imprimir os dados do cliente:  
        System.out.println("\nDADOS DO CLIENTE:");  
        System.out.println(cliente);  
        System.out.println(cliente.getEndereco());  
    }  
}
```

## Executando:



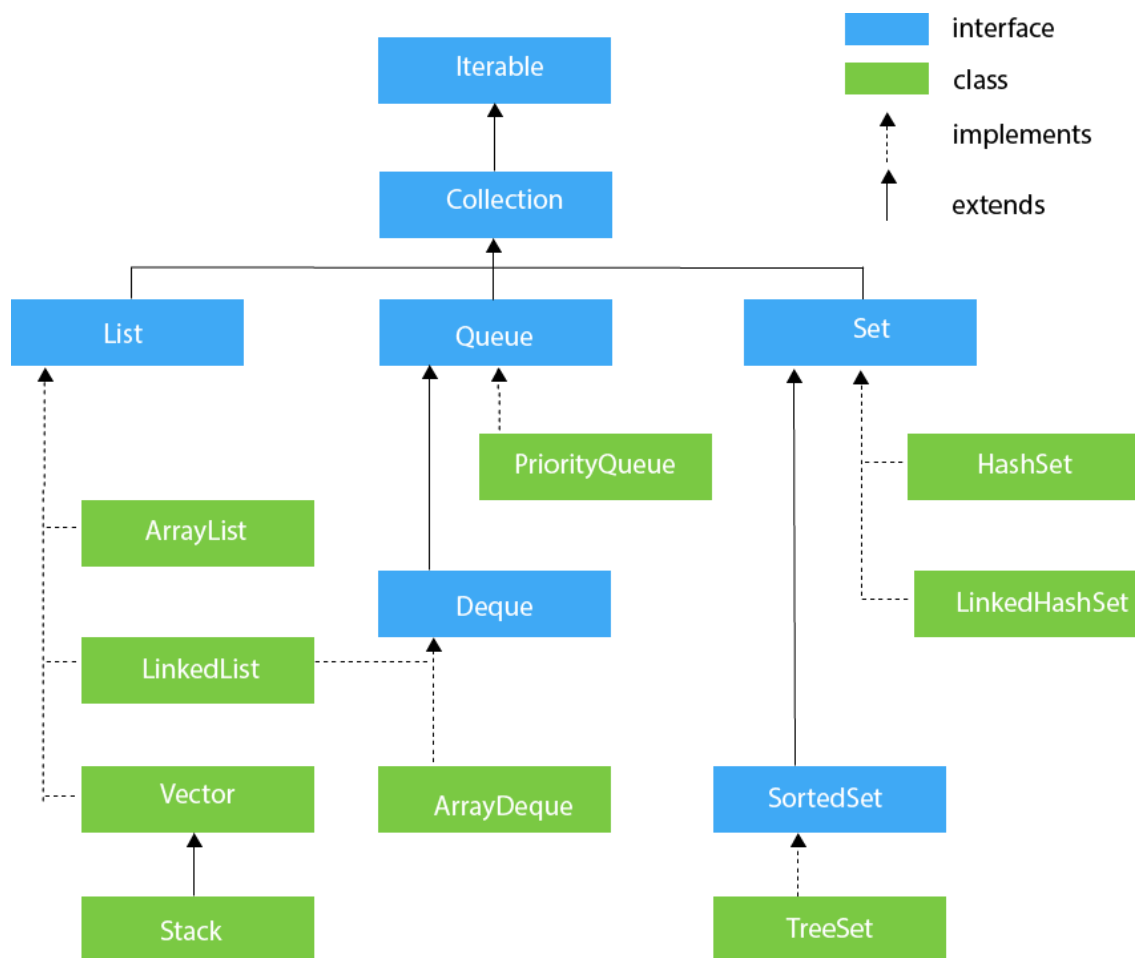
## DADOS DO CLIENTE:

Pessoa [id=1, nome=Ana Paula], Cliente  
[cpf=123.456.789-00, telefone=(21) 99999-9999,  
email=anapaula@gmail.com]

Endereco [id=1, logradouro=Av Rio Branco 185,  
bairro=Centro, cidade=Rio de Janeiro, estado=RJ,  
cep=200040-007]

Para criarmos um vínculo de associação do tipo TER-MUITOS precisamos utilizar algum tipo de Collection do Java (Coleções de objetos) tais como:

- **List** (listas de objetos)
- **Set** (listas que não permitem valor duplicado)
- **Queue** (filas de objetos – FIFO FIRST IN FIRST OUT)
- **Map** (mapas – chave / valor)



Continua...