

[S203]

# Arquitetura e Desenho de Software

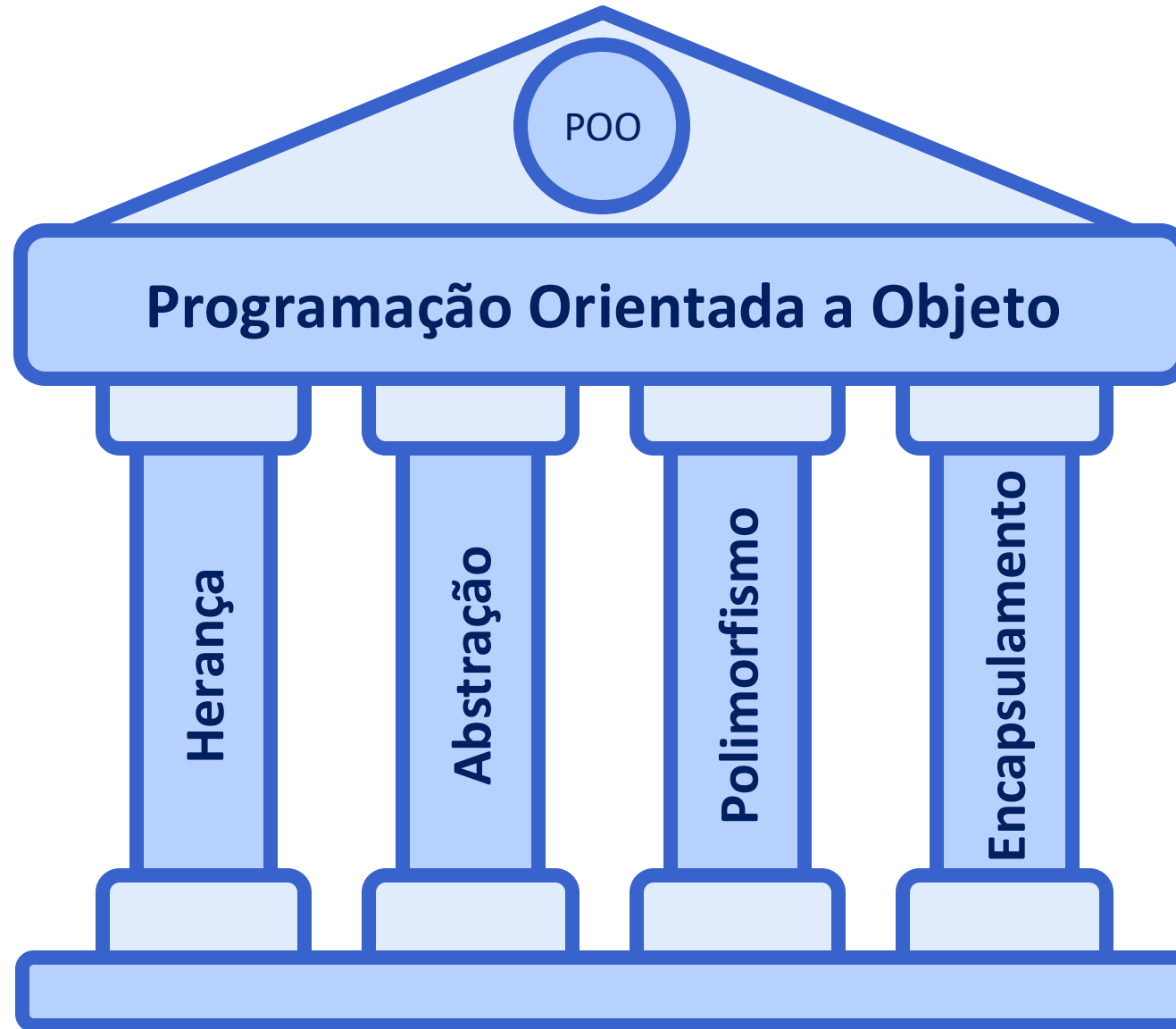


*Cap 00*

*Revisão*

# ***Orientação a Objeto***

# Pilares da Programação Orientada a Objetos





**RevisaoOO**

Use default location

Location: C:\LABS\_ADS\ADS\_VitorFigueiredo\workspace\_eclipse\RevisaoOO

JRE

☒ Use an execution environment JRE: **JavaSE-17**

☐ Use a project specific JRE: jre

☐ Use default JRE 'jre' and workspace compiler preferences

Project layout

☐ Use project folder as root for sources and class files

☒ Create separate folders for sources and class files

Working sets

☐ Add project to working sets

Working sets:

Module

☐ Create module-info.java file

Module name:

☐ Generate comments

**Finish**

## 1. Abrir o Eclipse.

(Verificar que se ele está apontando para **workspace\_eclipse**)

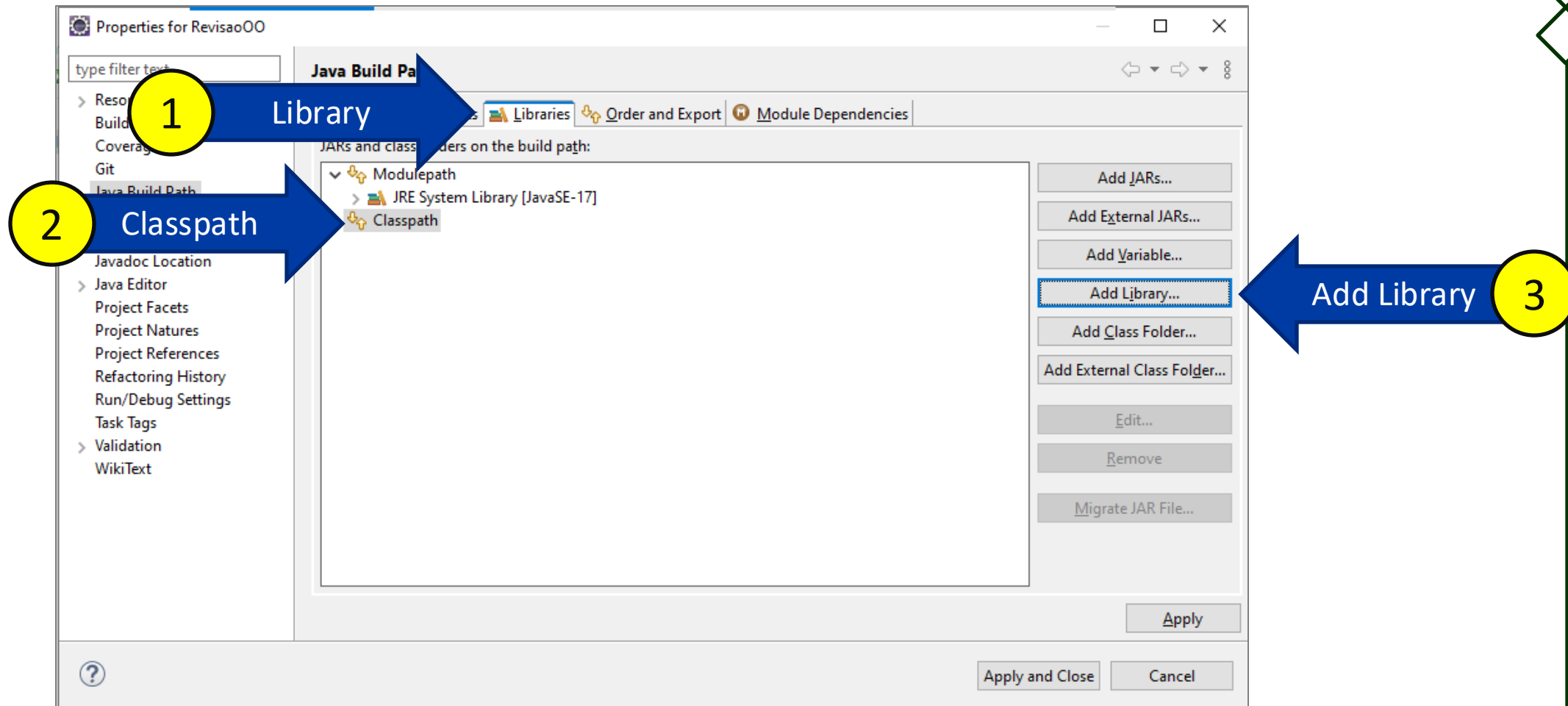
## 2. Criar o projeto Java: RevisaoOO

- Selecionar **Java-SE 17**
- Desmarcar **Create module-info.java file**
- Clicar **Finish**



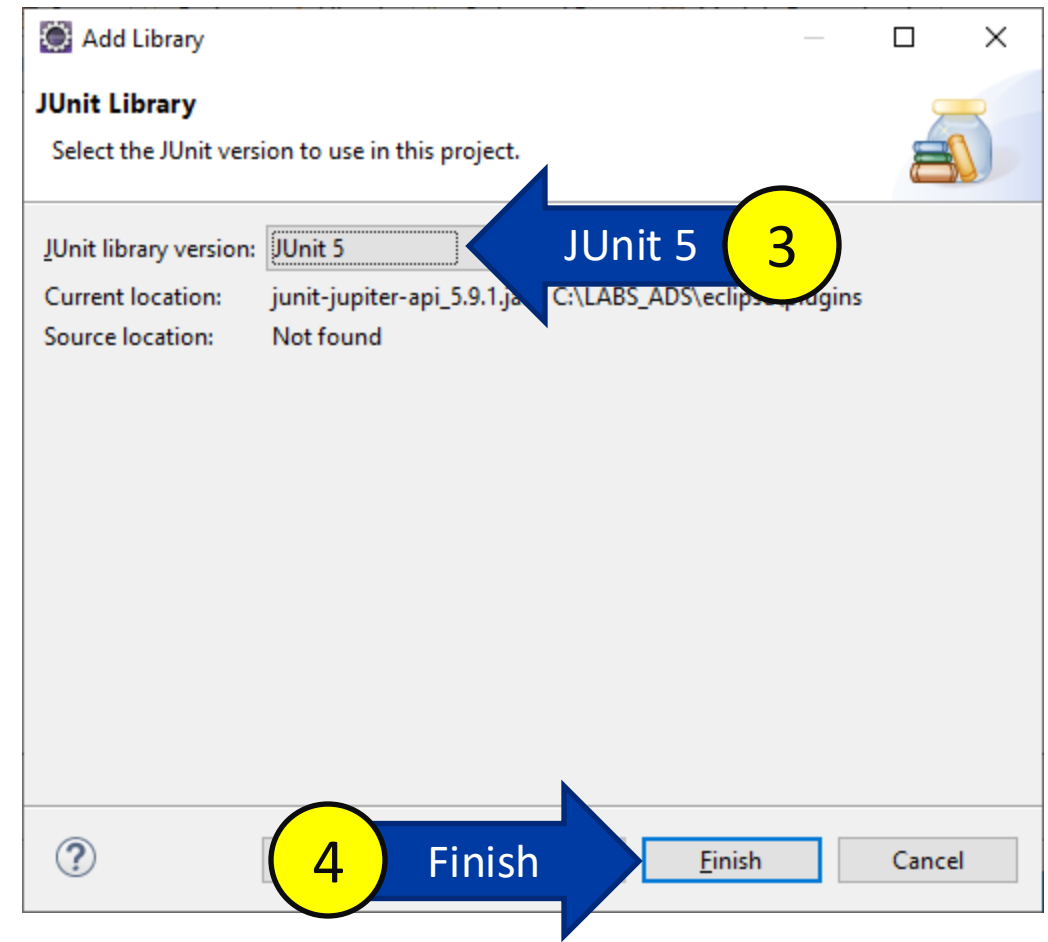
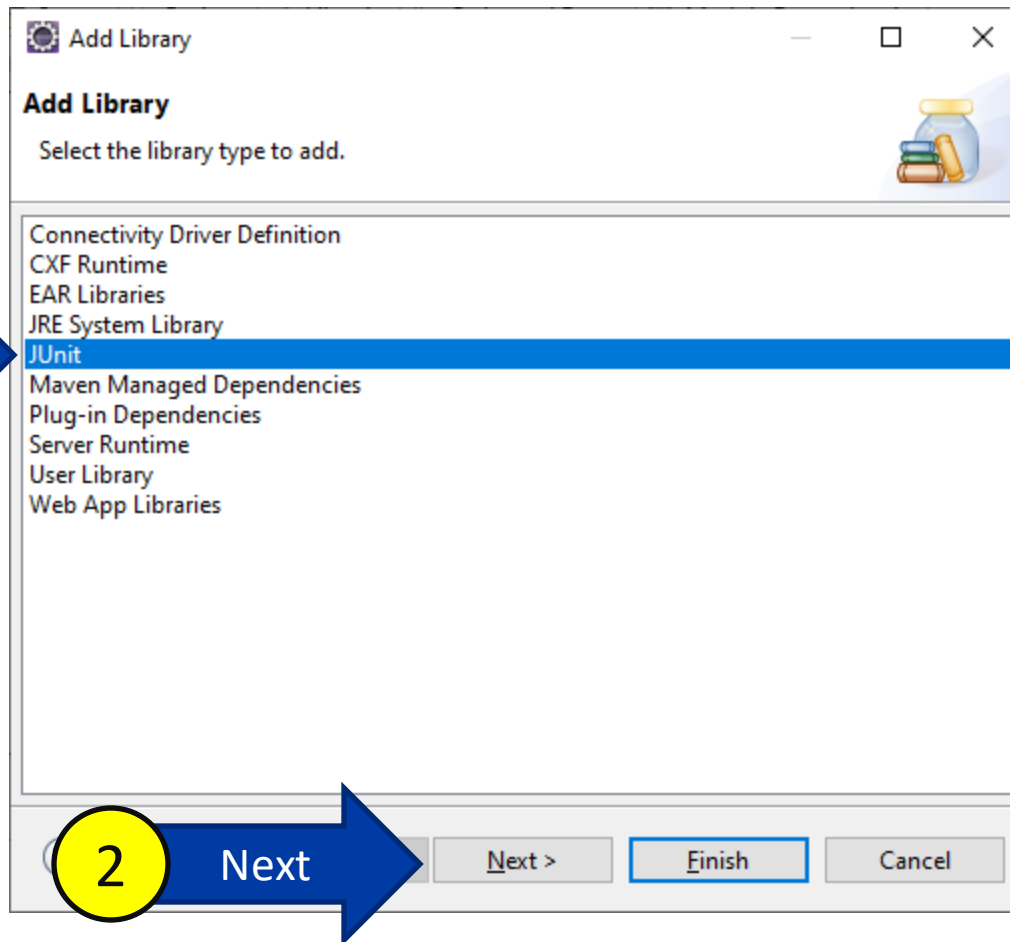
## 2.1. Adicionar JUnit como Library:

- Clique da direita no projeto: **Build Path** > **Configure Build Path**



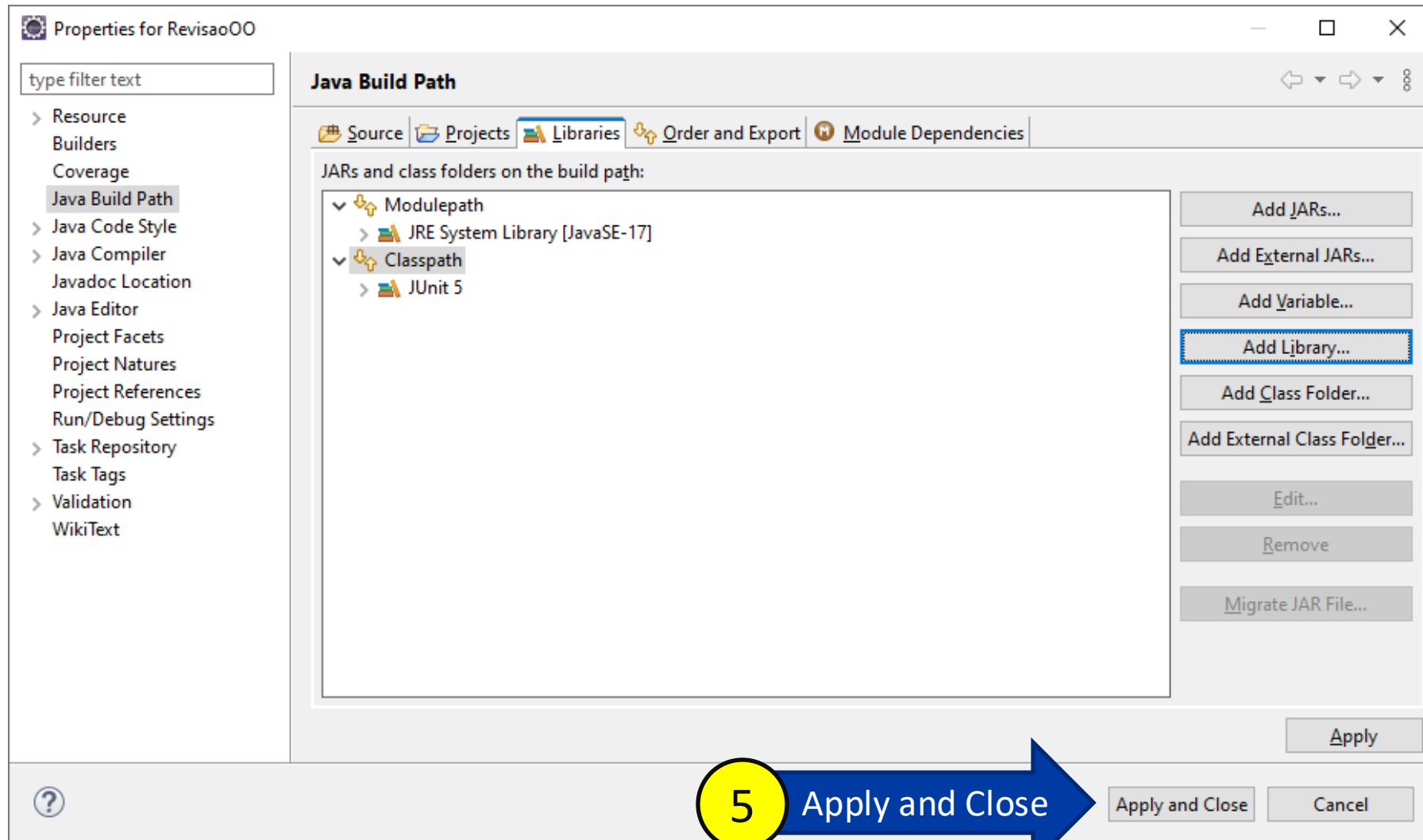


## 2.2. Selecionar JUnit 5





## 2.3. Clicar **Apply and Close**



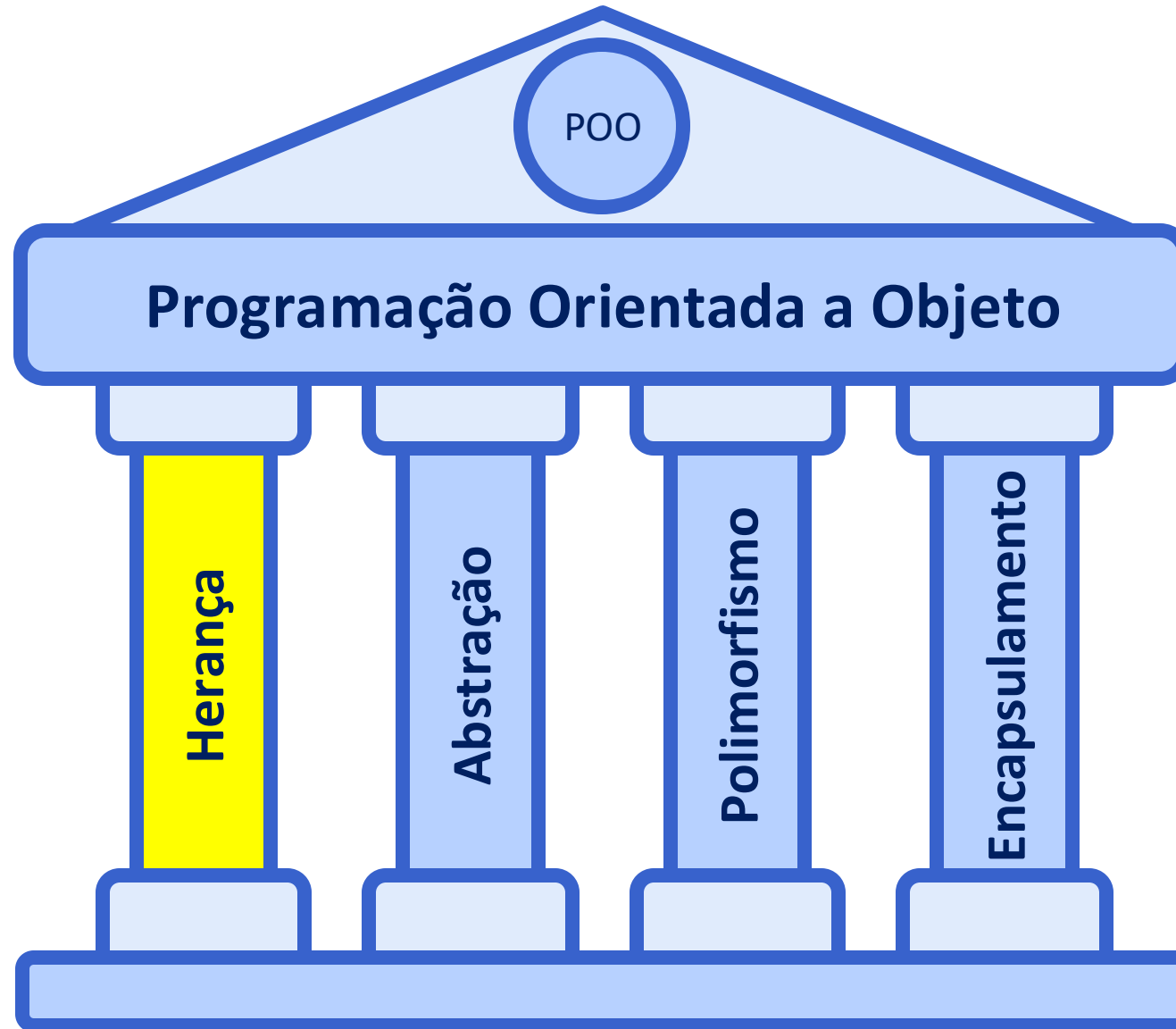


# Os Pilares da Programação Orientada a Objetos

- A programação Orientada a Objetos é bem requisitada no contexto das aplicações mais atuais, que o mercado demanda.
- Entre os motivos dessa preferência, estão a possibilidade de reutilização de código e a capacidade de representação do sistema ser muito mais próximo do que vivenciamos no mundo real.
- Os quatro pilares da programação Orientada a Objetos:
  - **Herança;**
  - **Abstração;**
  - **Encapsulamento;**
  - **Polimorfismo;**

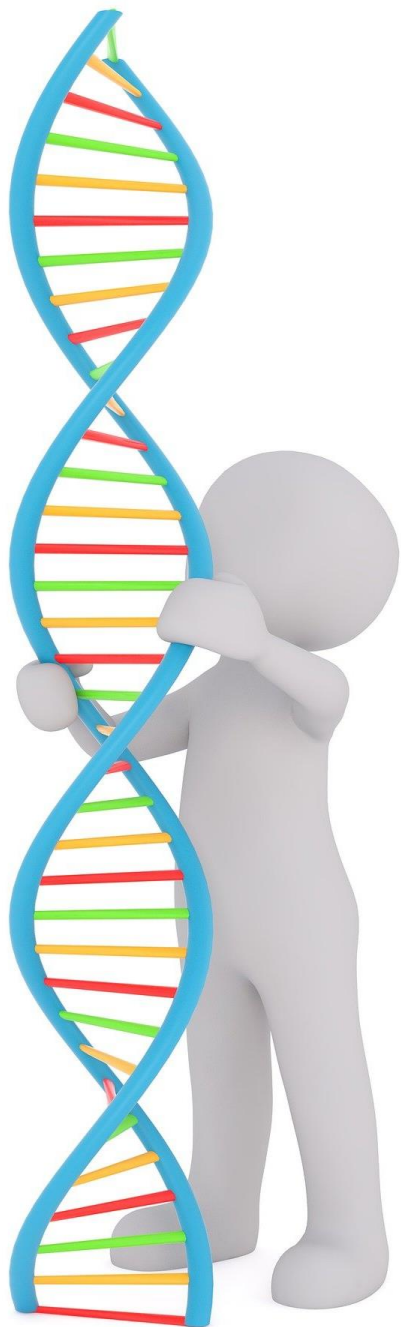


# Pilares da Programação Orientada a Objetos





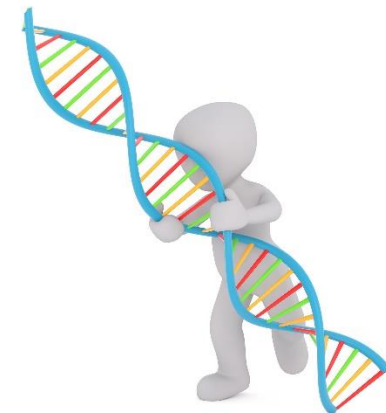
## Herança



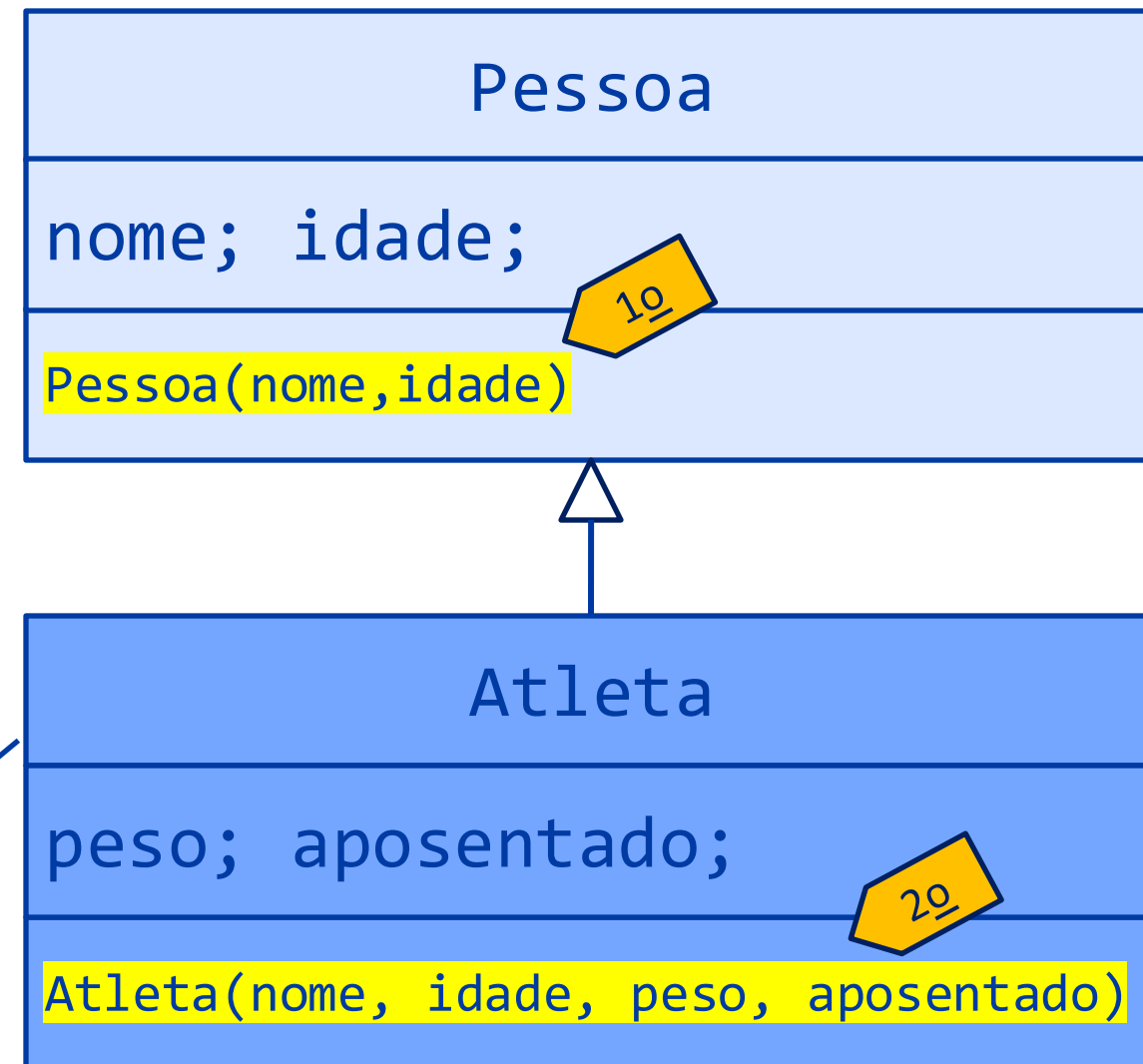
- O **reuso de código** é uma das grandes vantagens da programação orientada a objetos.
- Muito disso se dá por uma questão que é conhecida como **Herança**.
- Essa característica otimiza a produção da aplicação em tempo e linhas de código.
- É um mecanismo da Orientação a Objetos que permite criar novas classes a partir de classes já existentes, aproveitando-se das características existentes na classe a ser estendida

# Herança

- Com a herança, conseguimos **estender** os atributos e métodos de uma classe.
- Assim, evita-se **reescrever** atributos e método comuns a mais de um grupo de classes.
- Desta forma, temos um **agrupamento** de objetos que possuem um conjunto de propriedades e operações **em comum**.
- Com isso, temos a definição de um novo **tipo** de objeto.



- Construtores são **métodos especiais** invocados automaticamente pelo operador **new**
- Construtores podem ser **reinvocados pela sub-classe**.
- Construtores são invocados de **cima-para-baixo**:



Ao instanciar Atleta, primeiro será instanciado Pessoa

```
public class Pessoa extends Object {
```

```
    String nome;
```

```
    Integer idade;
```

```
    public Pessoa(String nome, Integer idade) {  
        super();  
        this.nome = nome;  
        this.idade = idade;  
    }
```

```
    void envelhecer() {
```

```
    }
```

```
}
```

```
public class Atleta extends Pessoa {
```

```
    Integer peso;
```

```
    Boolean aposentado;
```

```
    public Atleta(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade);  
        this.peso = peso;  
        this.aposentado = aposentado;  
    }
```

```
    void aquecer() {  
        System.out.println("Atleta aquecido");  
    }
```

```
    void aposentar() {  
        this.aposentado = true;  
    }
```

```
public class Pessoa extends Object {
```

```
    String nome;
```

```
    Integer idade;
```

```
    public Pessoa(String nome, Integer idade) {  
        super();  
        this.nome = nome;  
        this.idade = idade;  
    }
```

```
    void envelhecer() {
```

```
    }
```

```
}
```

```
public class Atleta extends Pessoa {
```

```
    Integer peso;
```

```
    Boolean aposentado;
```

```
    public Atleta(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade);  
        this.peso = peso;  
        this.aposentado = aposentado;  
    }
```

```
    void aquecer() {  
        System.out.println("Atleta aquecido");  
    }
```

```
    void aposentar() {  
        this.aposentado = true;  
    }
```

A primeira linha de um construtor é a invocação do **super-construtor** (mesmo que implícito)



1. Criar pacote **esporte**
2. Criar a super-classe **Pessoa**

```
public class Pessoa extends Object {  
  
    String nome;  
  
    Integer idade;  
  
    public Pessoa(String nome, Integer idade) {  
        super();  
        this.nome = nome;  
        this.idade = idade;  
    }  
  
    void envelhecer() {  
        this.idade++;  
    }  
}
```



## 3. Criar a sub-classe **Atleta**

```
public class Atleta extends Pessoa {  
  
    Integer peso;  
  
    Boolean aposentado;  
  
    public Atleta(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade);  
        this.peso = peso;  
        this.aposentado = aposentado;  
    }  
  
    void aquecer() {  
        System.out.println("Atleta aquecido");  
    }  
  
    void aposentar() {  
        this.aposentado = true;  
    }  
}
```





## 4. Escrever **AtletaTest**:

(executar em modo debug e analisar quais construtores são executados)

```
@Test
void testarConstrutor() {

    Pessoa pessoaEdson = new Pessoa("Edson", 21);

    System.out.println( pessoaEdson.nome );
    System.out.println( pessoaEdson.idade );

    Atleta atletaEdson = new Atleta("Edson", 21, 80, false);

    System.out.println( atletaEdson.nome );
    System.out.println( atletaEdson.idade );
    System.out.println( atletaEdson.peso );
    System.out.println( atletaEdson.aposentado );
}
```

- Podemos criar  
CLASSES ainda mais  
especializadas:

```
public class Nadador extends Atleta {  
    public Nadador(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade, peso, aposentado);  
    }  
  
    void nadar() {  
        System.out.println("Nadador nadando");  
    }  
}
```

```
public class Ciclista extends Atleta {  
    public Ciclista(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade, peso, aposentado);  
    }  
  
    void pedalar() {  
        System.out.println("Ciclista pedalando");  
    }  
}
```

```
public class Corredor extends Atleta {  
    public Corredor(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade, peso, aposentado);  
    }  
  
    void correr() {  
        System.out.println("Corredor correndo");  
    }  
}
```

- Uma classe ao extender outra, ela herda os **atributos** e **métodos**
- Isso significa **reutilização de código**

```
Corredor corredorJoao = new Corredor("João", 25, 75, false);  
  
corredorJoao.aquecer(); // metodo de Atleta  
corredorJoao.correr();  // metodo de Corredor
```

No entanto:

**Construtores NÃO são herdados,**  
por isso precisam ser declarados em cada sub-classe

```
public class Nadador extends Atleta {  
    public Nadador(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade, peso, aposentado);  
    }  
}
```

```
public class Ciclista extends Atleta {  
    public Ciclista(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade, peso, aposentado);  
    }  
    void nadar() {  
        System.out.println("Nadador nadando");  
    }  
    void pedalar() {  
        System.out.println("Ciclista pedalando");  
    }  
}
```

```
public class Corredor extends Atleta {  
    public Corredor(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade, peso, aposentado);  
    }  
    void correr() {  
        System.out.println("Corredor correndo");  
    }  
}
```



1. Declarar a sub-classe **Nadador** que herda de **Atleta**

\*codar o método **nadar()**

```
public class Nadador extends Atleta {  
    public Nadador(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade, peso, aposentado);  
    }  
  
    void nadar() {  
        System.out.println("Nadador nadando");  
    }  
}
```



2. Declarar a sub-classe **Ciclista** que herda de **Atleta**

\*codar o método **pedalar()**

```
public class Ciclista extends Atleta {  
    public Ciclista(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade, peso, aposentado);  
    }  
    void pedalar() {  
        System.out.println("Ciclista pedalando");  
    }  
}
```



3. Declarar a sub-classe **Corredor** que herda de **Atleta**

\*codar o método **correr()**

```
public class Corredor extends Atleta {  
    public Corredor(String nome, Integer idade, Integer peso, Boolean aposentado) {  
        super(nome, idade, peso, aposentado);  
    }  
    void correr() {  
        System.out.println("Corredor correndo");  
    }  
}
```



4. Na classe **AtletaTest**, escrever o teste **testarNadador()**:

```
@Test
void testarNadador() {

    Nadador nadador = new Nadador("Gustavo", 35, 90, false);
    nadador.nadar();

    nadador.envelhecer();
    System.out.println( nadador.idade );

    nadador.aposentar();
    System.out.println( nadador.aposentado );
}
```





5. Na classe **AtletaTest**, escrever o teste **testarCiclista()**:

```
@Test
void testarCiclista() {
    Ciclista ciclista = new Ciclista("Tiago", 44, 85, false);
    System.out.println("Está aposentado");
    if (ciclista.aposentado) {
        System.out.println("Sim");
    } else {
        System.out.println("Não");
    }

    ciclista.pedalar();
    ciclista.envelhecer();
    ciclista.aposentar();
    System.out.println("Agora já está aposentado?");
    System.out.println( ciclista.aposentado ? "Sim" : "Não" );
}
```

## Para saber mais: Herança Múltipla

- Herança múltipla é a capacidade de uma classe herdar propriedades de **várias classes** simultaneamente.
- Apesar de parecer um recurso útil e poderoso, em sistemas grandes e complexos a herança múltipla pode trazer **problemas grandes**
- Por exemplo: **numa classe que usa herança múltipla, nada impede que existam 2 métodos idênticos em super-classes diferentes. Quando a classe invocar este método, qual super-classe será responsável pelo método?**
- Pelos possíveis efeitos colaterais, **Java NÃO suporta herança múltipla.**



# Desafio: Herança



**Inatel**

1. Criar o pacote **cidadania**
2. Escreva uma classe chamada **Pessoa** com os atributos: **nome** (String), **sexo** (String), **idade** (Integer)
3. Escreva agora outra classe chamada **Cidadao**, que é uma pessoa (extende da classe Pessoa) a qual possui o atributo **cpf** (String). Os construtores das duas classes devem exigir valores para todos os seus atributos.

**Obs.: o construtor da classe filha deve invocar o construtor da sua super classe.**

O programa deve solicitar os dados ao usuário para se instanciar um objeto **Cidadao**. Deve-se exibir todos os atributos do objeto criado.

## Dicas:

- Use a classe `java.util.Scanner` para entrar receber entradas do usuário:

```
Scanner scanner = new Scanner( System.in );
```

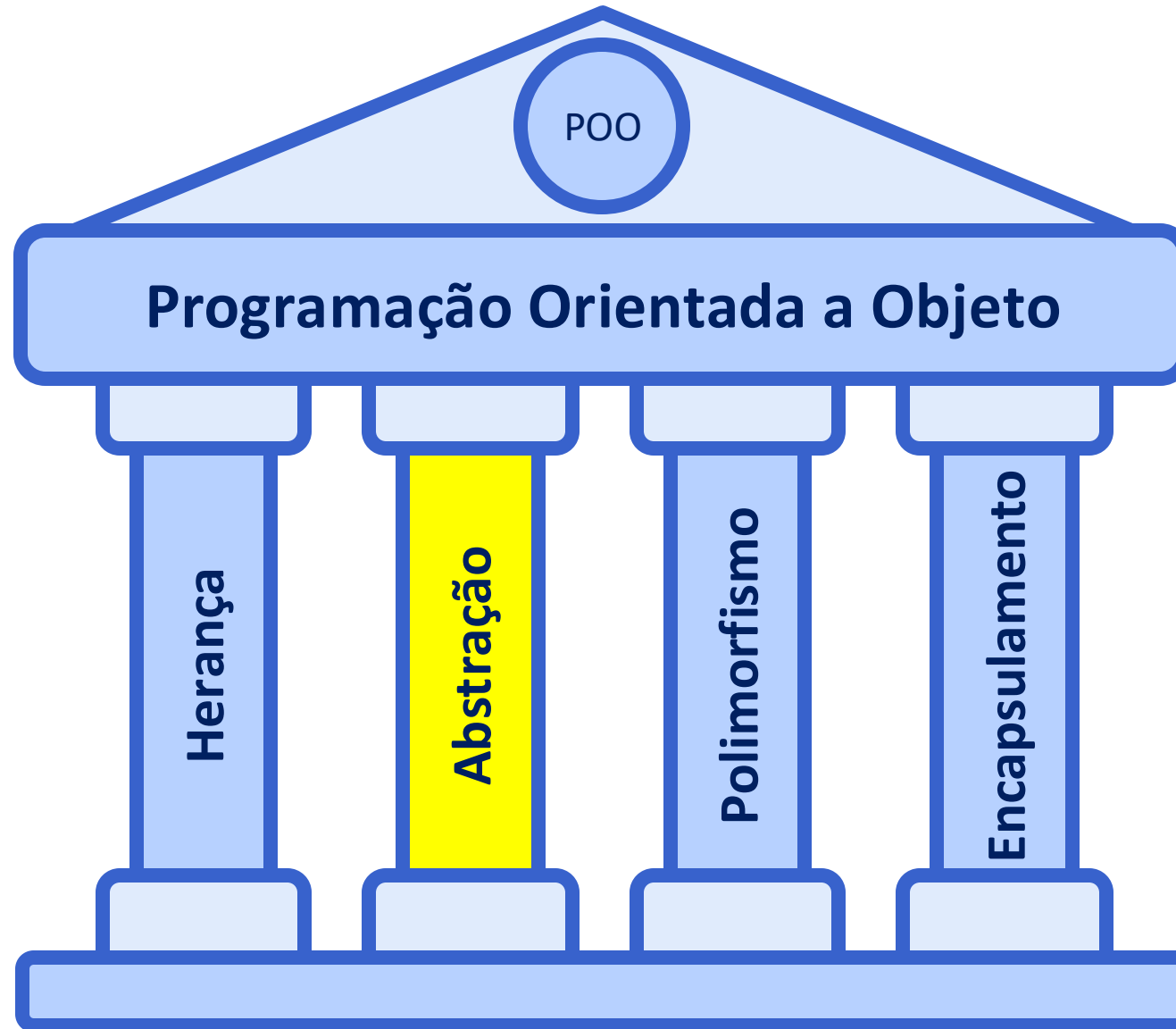
- Para solicitar os dados do tipo String, use:

```
System.out.println("Entre com o nome: ");  
String nome = scanner.nextLine();
```

- Para solicitar os dados do tipo Integer, use:

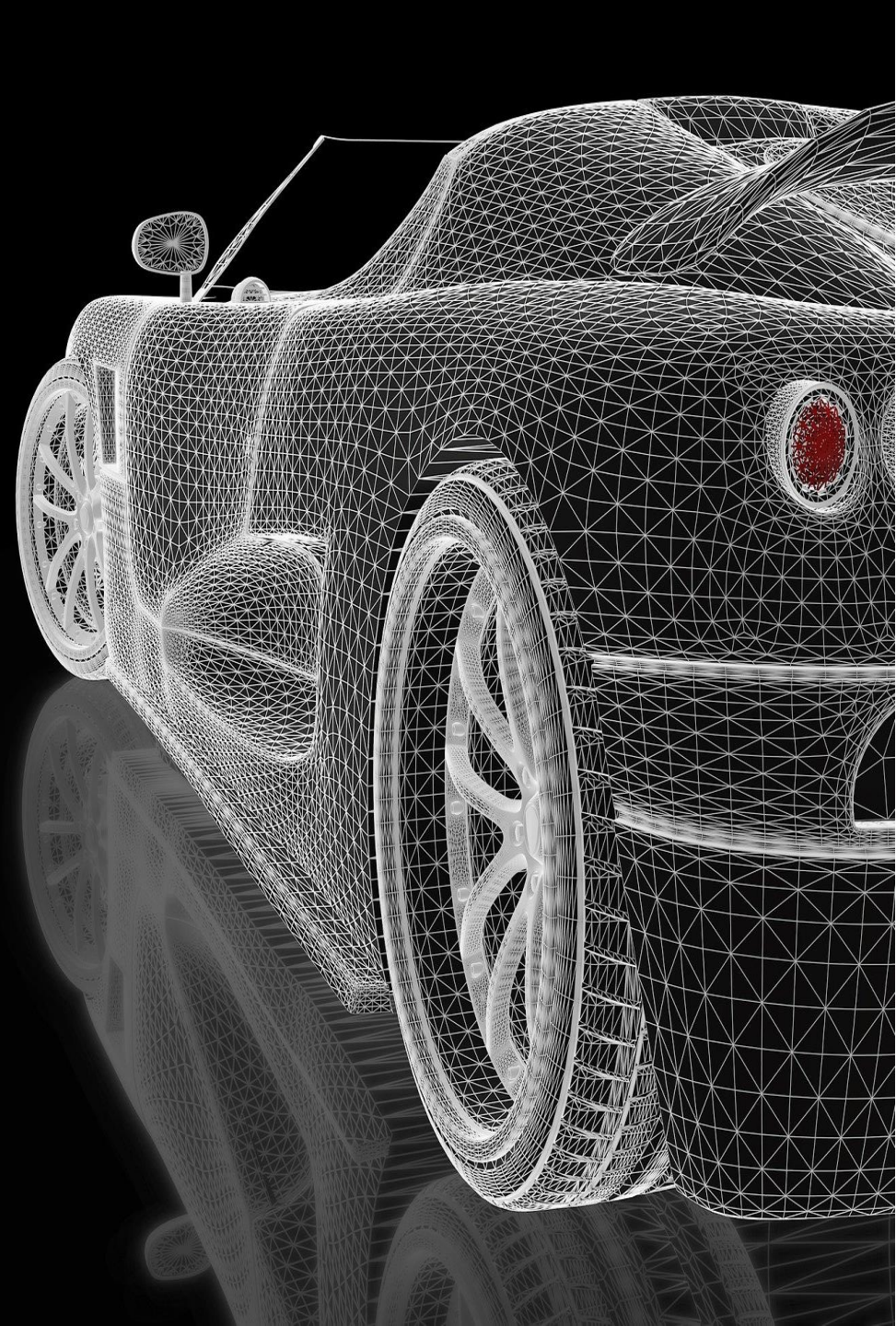
```
Integer idade = scanner.nextInt();
```

# Pilares da Programação Orientada a Objetos



# Abstração

- *“Ação de abstrair, de analisar isoladamente um aspecto, contido num todo, sem ter em consideração sua relação com a realidade.”*
- A Abstração é a **habilidade** e a **capacidade** de se modelar conceitos, entidades, elementos, problemas e características do mundo real, de um domínio do problema em questão, **levando-se em conta apenas os detalhes importantes** para a resolução do problema e desprezando coisas que não têm importância no contexto.



# Abstração

- Com esse pilar, uma classe **deve expor para outras classes apenas a ideia geral de uma propriedade ou funcionalidade, sem entrar nos detalhes.**
- **Por exemplo, um liquidificador possui 3 velocidades:**
  - Não sabemos como a velocidade da hélice é calculada, nem como internamente o apertar de um botão faz ela rodar, mas sabemos que ao apertar um botão a hélice irá girar de acordo com a velocidade escolhida.
- Podemos alcançar abstração através de **classes abstratas** e **interfaces**, que funcionam como uma espécie de contrato.
- Interfaces e classes abstratas **apenas definem** o que uma classe **deve fazer**, enquanto o **dever da classe** é implementar **como a classe irá fazer**.



# Classe Abstrata

- Em Java, usamos a palavra reservada **abstract** para indicar que uma classe é abstrata. Usamos a mesma palavra para indicar que um método é abstrato.
- Tomemos o exemplo para classe abstrata **Human**.
  - Por se tratar de uma classe abstrata, a operação eat() ainda não será definida ainda e deixaremos seus detalhes para as sub-classes:

```
public abstract class Human {  
  
    Integer refeicoesDiarias;  
  
    public Human(Integer refeicoesDiarias) {  
        super();  
        this.refeicoesDiarias = refeicoesDiarias;  
    }  
  
    abstract void eat();  
  
}
```

# Implementando Classe Abstrata

- Duas classes podem implementar a Human:
  - **Adulto:** cuja alimentação é baseada em alimento sólido
  - **Criança:** cuja alimentação é baseada em leite materno

```
public class Adult extends Human {  
  
    public Adult(Integer refeicoesDiarias) {  
        super(refeicoesDiarias);  
    }  
  
    @Override  
    void eat() {  
        String dica = String.format("Alimento sólido ao menos %s ao dia", this.refeicoesDiarias);  
        System.out.println( dica );  
    }  
}
```

```
public class Child extends Human {  
  
    public Child(Integer refeicoesDiarias) {  
        super(refeicoesDiarias);  
    }  
  
    @Override  
    void eat() {  
        String recomendacao = String.format("Amamentar %s vezes ao dia", this.refeicoesDiarias);  
        System.out.println( recomendacao );  
    }  
}
```



# Exemplo Classe Abstrata



**Inatel**

```
Adult adulto = new Adult(5);  
adulto.eat();
```

```
Child crianca = new Child(3);  
crianca.eat();
```

Console X

<terminated> HumanTest [JUnit] C:\LABS\_ADS\eclipse\plugins\org.eclipse.jst

Alimento sólido ao menos 5 ao dia

Amamentar 3 vezes ao dia

**Classe abstrata NÃO pode ser instanciada**

```
Human humano = new Human( 4 ); //erro de compilação
```

**A variável pode usar a classe abstrata**

```
Human adulto_2 = new Adult( 5 );
```

```
Human crianca_2 = new Child( 4 );
```



## Desafio: Classe Abstrata



**Inatel**

Criar um sistema em Java para **gerenciar contas bancárias**:

(1/3)

- No pacote **banco**, implementar uma classe **Conta** tornando-a abstrata através da definição de método **rentabilizar()**.
- Além disso, a classe Conta deve ter os métodos concretos: **depositar()** e **sacar()**, bem com o atributo **saldo** (Double) com valor inicial zero.
- Utilizando herança, criar duas classes a partir de Conta: **ContaCorrente** e **ContaPoupanca**. Nestas classes, implementar o método **rentabilizar()** de acordo com *regra de negócio*:
  - Conta corrente: deve rentabilizar-se com uma taxa fixa de 1%;
  - Conta poupança: deve rentabilizar-se com uma taxa fixa de 2%;



# Desafio: Classe Abstrata

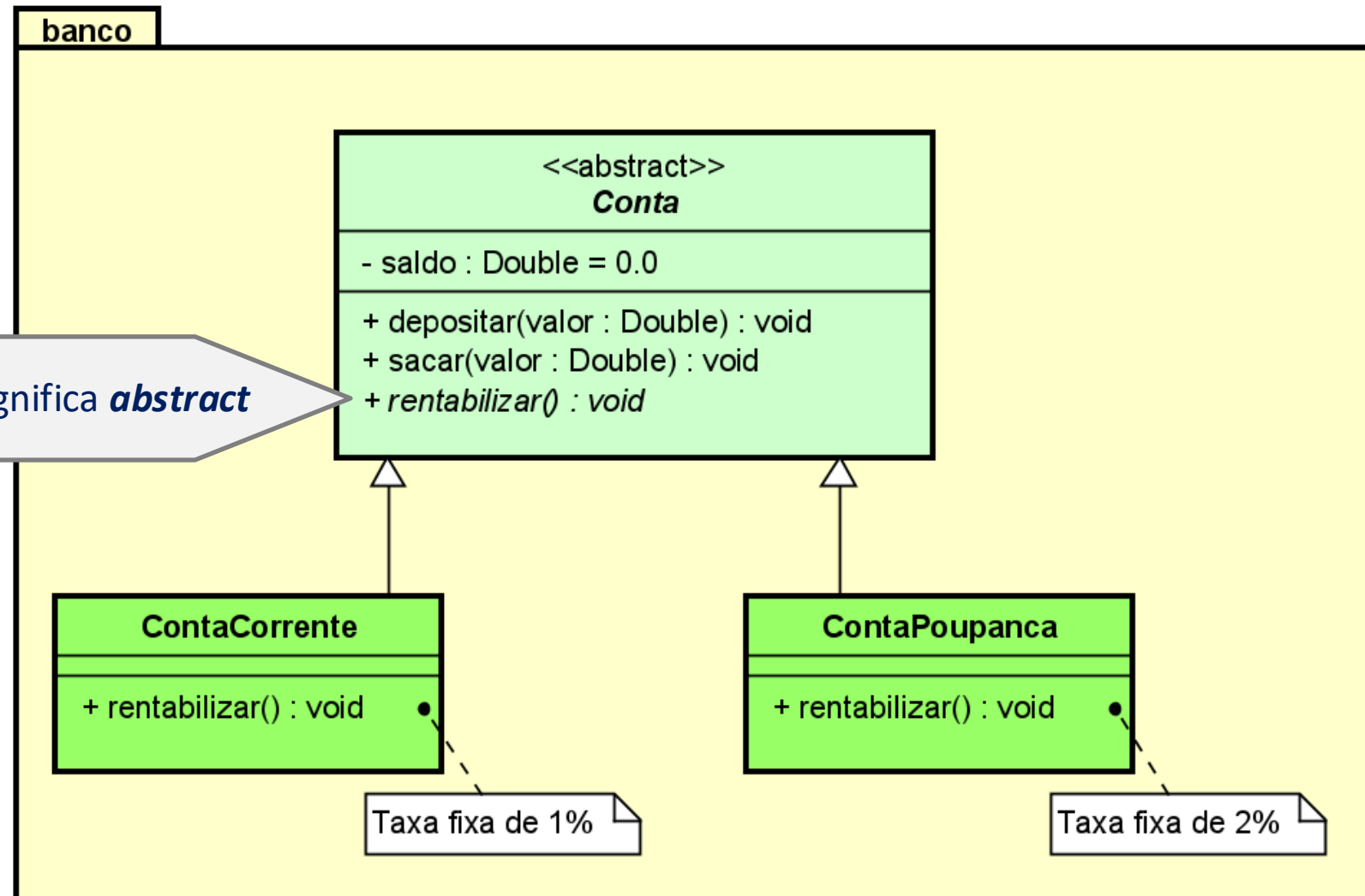


**Inatel**

Criar um sistema em Java para **gerenciar contas bancárias**:

(2/3)

Na UML, formatação *ITALIC* significa **abstract**





## Desafio: Classe Abstrata



**Inatel**

Criar um sistema em Java para **gerenciar contas bancárias**:

(3/3)

- Criar a classe **ContaTest**, com os métodos **testaContaCorrente()** e **testaContaPoupanca()** onde você deve instanciar objetos das novas classes criadas, invocar **depositar()**, **sacar()**, e então método **rentabilizar()** e exibir o novo saldo ao usuário.
- Finalmente, fazer a confirmação que o valor do saldo atual é o valor esperado

(\*) Dica: Usar **assertEquals()** do JUnit. Buscar na documentação

## Classes Abstratas

- Podem ter métodos concretos
- Podem ter atributos
- Sub-classe “***extende***” classe abstrata

## Interfaces

- Tem apenas **métodos abstratos**
- Todos atributos automaticamente são **constantes**
- Sub-classe “***implementa***” a interface

- É tão somente um conjunto de contrato

- Nome do método
- Lista de parâmetros
- Tipo de retorno

- Todas as classes que implementam a interface são obrigadas a “dar corpo” a estes contratos.

- Interfaces servem também para marcarem uma **familiaridade** entre classes.

# Interfaces :: Exemplo

Num sistema de contabilidade, cada tipo de empresa é passível de ser tributada mensalmente pelo IR e ISS. Cada tipo de empresa tem sua própria taxa.

```
public class EmpresaMEI {  
  
    Double faturamentoMensal;  
  
}
```

```
public class EmpresaSimples {  
  
    Double faturamentoMensal;  
  
}
```

Podemos declarar uma interface para obrigar as respectivas empresas a calcular seus impostos. Vamos chamar esta interface **Tributavel** com o métodos getValorIR() e getValorISS()



# Interfaces :: Exemplo



**Inatel**

```
public interface Tributavel {  
  
    Double getValorIR();  
  
    Double getValorISS();  
}
```

As classes passam a implementar **Tributavel**

```
public class EmpresaMEI implements Tributavel {  
  
    Double faturamentoMensal;  
  
    Double getValorIR() {  
        return /*calcula IR aqui*/;  
    }  
  
    Double getValorISS() {  
        return /*calcula ISS aqui*/;  
    }  
}
```

```
public class EmpresaSimples implements Tributavel {  
  
    Double faturamentoMensal;  
  
    Double getValorIR() {  
        return /*calcula IR aqui*/;  
    }  
  
    Double getValorISS() {  
        return /*calcula ISS aqui*/;  
    }  
}
```



1. Criar pacote **contabilidade**
2. Criar classe **EmpresaMEI**
3. Criar classe **EmpresaSimples**

```
public class EmpresaMEI {  
  
    Double faturamentoMensal;  
  
    public EmpresaMEI(Double faturamentoMensal) {  
        super();  
        this.faturamentoMensal = faturamentoMensal;  
    }  
}
```

```
public class EmpresaSimples {  
  
    Double faturamentoMensal;  
  
    public EmpresaSimples(Double faturamentoMensal) {  
        super();  
        this.faturamentoMensal = faturamentoMensal;  
    }  
}
```



4. Declarar a interface **Tributavel** com os 2 métodos de cálculo:

```
public interface Tributavel {  
  
    Double getValorIR();  
  
    Double getValorISS();  
  
}
```





5. Refatorar classe **EmpresaMEI** para implementar **Tributavel**

\* Com o comando do Eclipse, declarar os métodos da interface

\* Usar a tabela abaixo:

Imposto	Taxas
IR	27,5%
ISS	0%



6. Refatorar classe **EmpresaSimples** para implementar **Tributavel**

\* Com o comando do Eclipse, declarar os métodos da interface

\* Usar a tabela abaixo:

Imposto	Taxas
IR	15%
ISS	8%



7. Escrever a classe de teste **ContabilidadeTest** com os métodos:

```
void testEmpresaMEI_valorIR()
```

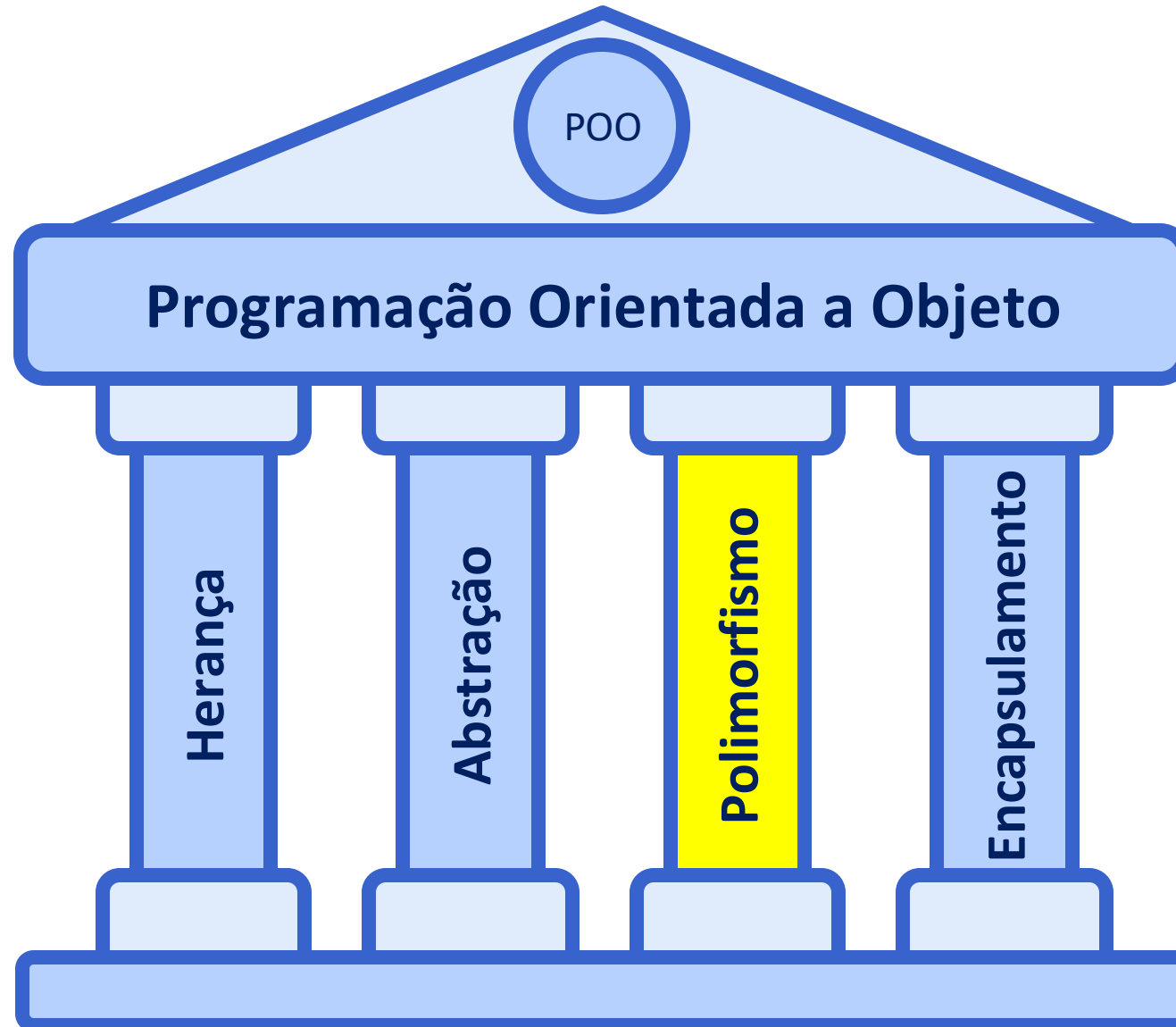
```
void testEmpresaMEI_valorISS()
```

```
void testEmpresaSimples_valorIR()
```

```
void testEmpresaSimples_valorISS()
```



# Pilares da Programação Orientada a Objetos



- Polimorfismo é capacidade de **alterar a execução** de um método conforme a sub-classe usada

Nota: Apesar de usar “morfismo” (*forma*, do grego), não é a forma que é “poli”, e sim a **execução**



# Polimorfismo :: Exemplo



**Inatel**

- Vamos usar **Conta**, **ContaCorrente** e **ContaPoupanca**

```
//Este flag define qual será a instancia de Conta.  
boolean flagContaCorrenteComoDefault = true;  
  
Conta conta = null;  
if (flagContaCorrenteComoDefault) {  
    conta = new ContaCorrente();  
}else {  
    conta = new ContaPoupanca();  
}  
  
conta.depositar(1000.00);//sempre é invocado de Conta  
  
conta.rentabilizar();//pode ser invocado de ContaCorrente ou ContaPoupanca  
//isso é polimorfismo  
  
System.out.println( conta.getClass().getName() );//qual instancia  
System.out.println( conta.saldo );
```

# Polimorfismo :: Exemplo



**Inatel**

- Vamos usar **Conta**, **ContaCorrente** e **ContaPoupanca**

```
//Este flag define qual será a instancia de Conta.  
boolean flagContaCorrenteComoDefault = true;  
  
Conta conta = null;  
if (flagContaCorrenteComoDefault) {  
    conta = new ContaCorrente();  
}  
else {  
    conta = new ContaPoupanca();  
}  
  
conta.depositar(1000.00);  
  
conta.rentabilizar();  
  
System.out.println( conta.getClass().getName() );  
System.out.println( conta.saldo );
```

Console X

<terminated> ContaTest.testPolimorfismo

banco.ContaCorrente  
1010.0

# Polimorfismo :: Exemplo



**Inatel**

- Vamos usar **Conta**, **ContaCorrente** e **ContaPoupanca**

```
//Este flag define qual será a instancia de Conta.
```

```
boolean flagContaCorrenteComoDefault = false;
```

```
Conta conta = null;
```

```
if (flagContaCorrenteComoDefault) {
```

```
    conta = new ContaCorrente();
```

```
} else {
```

```
    conta = new ContaPoupanca();
```

```
}
```

```
conta.depositar(1000.00); //sempre é invocado de Conta
```

```
conta.rentabilizar(); //pode ser invocado de ContaCorrente ou ContaPoupanca
```

```
//isso é polimorfismo
```

```
System.out.println( conta.getClass().getName() ); //qual instancia
```

```
System.out.println( conta.saldo );
```

O que acontece se o valor da variável for false?

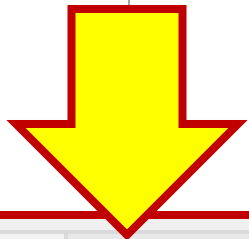
# Polimorfismo :: Exemplo



**Inatel**

- Vamos usar **Conta**, **ContaCorrente** e **ContaPoupanca**

```
//Este flag define qual será a instancia de Conta.  
boolean flagContaCorrenteComoDefault = false;  
  
Conta conta = null;  
if (flagContaCorrenteComoDefault) {  
    conta = new ContaCorrente();  
} else {  
    conta = new ContaPoupanca();  
}  
  
conta.depositar(1000.00); //sempre é invocado de Conta  
  
conta.rentabilizar(); //pode ser invocado de ContaCorrente ou  
//isso é polimorfismo  
  
System.out.println( conta.getClass().getName() ); //qual inst  
System.out.println( conta.saldo );
```



```
Console X  
<terminated> ContaTest.testPolimorfism  
banco.ContaPoupanca  
1020.0
```



1. Na classe de teste **ContaTest**, declarar novo metodo de teste:

```
@Test
void testPolimorfismo() {

    //Este flag define qual será a instancia de Conta.
    boolean flagContaCorrenteComoDefault = true;

    Conta conta = null;
    if (flagContaCorrenteComoDefault) {
        conta = new ContaCorrente();
    } else {
        conta = new ContaPoupanca();
    }

    conta.depositar(1000.00); //sempre é invocado de Conta

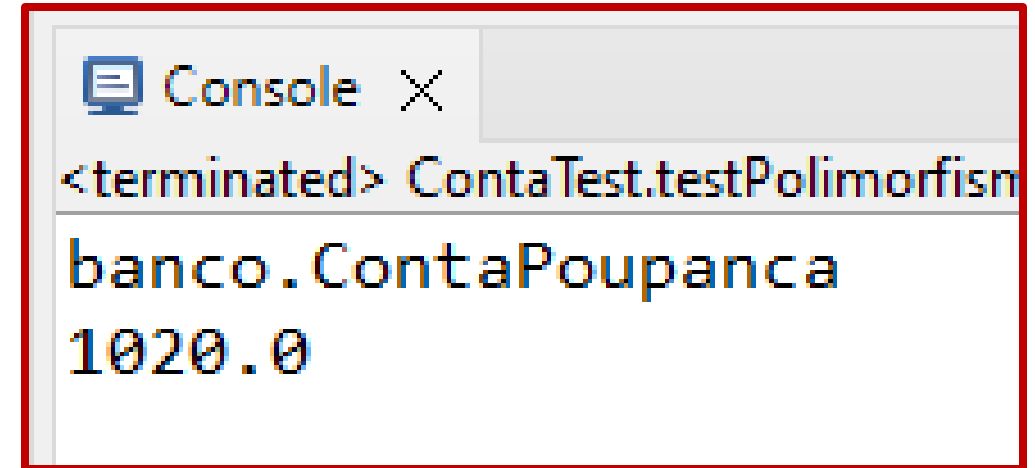
    conta.rentabilizar(); //pode ser invocado de ContaCorrente ou ContaPoupanca
    //isso é polimorfismo

    System.out.println( conta.getClass().getName() ); //qual instancia
    System.out.println( conta.saldo );
}
```



2. Agora, vamos trocar o valor do `flagContaCorrenteComoDefault` para **false**

3. Executar o teste e ver o resultado



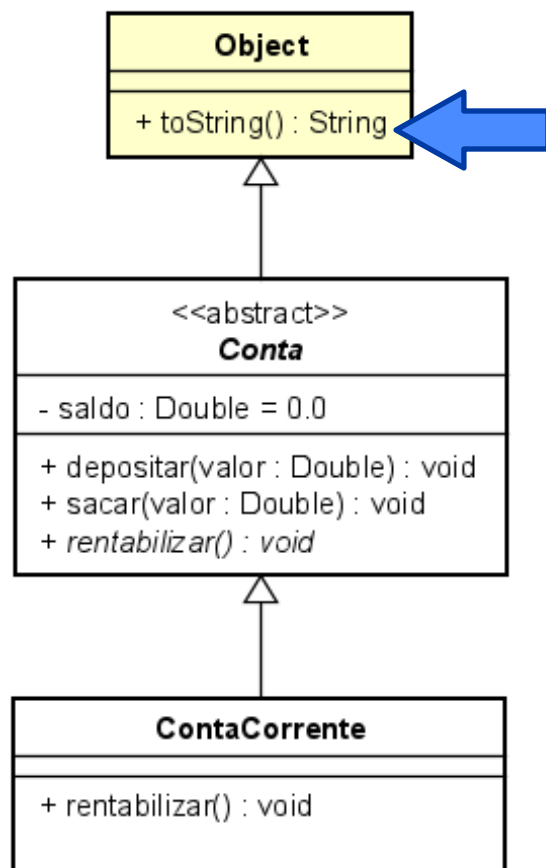
```
Console X
<terminated> ContaTest.testPolimorfismo
banco.ContaPoupanca
1020.0
```

Outro  
resultado é  
esperado

# Polimorfismo em Object



**Inatel**



- Já vimos que toda classe em Java implicitamente estende de **Object**
- **Object** tem vários métodos prontos propósito geral que pode ser reescritos (*overrides*).
- **toString()** é um deles e sua ideia é representar uma instância na forma de *string*:

```
Conta conta = new ContaCorrente();
conta.depositar( 100.00 );
conta.sacar( 10.0 );

String str = conta.toString(); //toString() de Object

System.out.println( str );
```

```
Console X
<terminated> ContaTest.testToString [JUnit] C:\LABS_
banco.ContaCorrente@5677323c
```

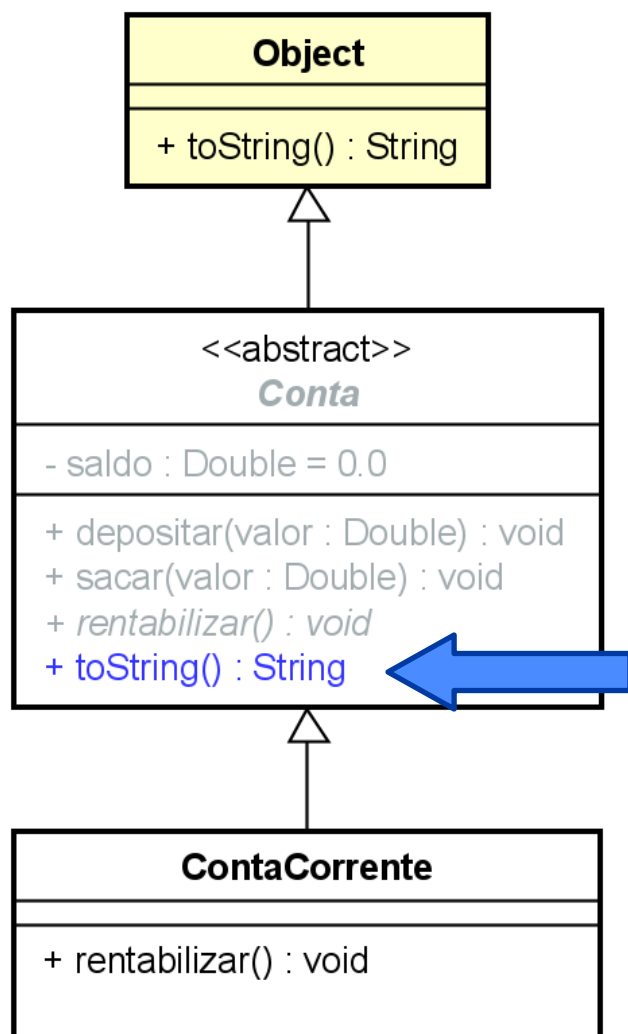
String sem significado

# Polimorfismo em Object: reescrevendo toString()



# Inatel

- Mas podemos dar significado à string retornada por toString(), reescrevendo-o em Conta



```
public abstract class Conta {

    Double saldo = 0.0;

    @Override
    public String toString() {
        return "Conta [saldo=" + saldo + "]";
    }
}
```

Agora a string é a concatenação do nome da classe e o saldo



# Polimorfismo em Object: reescrevendo toString()



# Inatel

## ...executando mesmo código

```
Conta conta = new ContaCorrente();
conta.depositar( 100.00 );
conta.sacar( 10.0 );

String str = conta.toString(); //toString() de Conta

System.out.println( str );
```

## ..teremos outro resultado

```
Console X
<terminated> ContaTest.testToString [Unit]
Conta [saldo=90.0]
```

Uma string significativa



# Polimorfismo: **toString()**

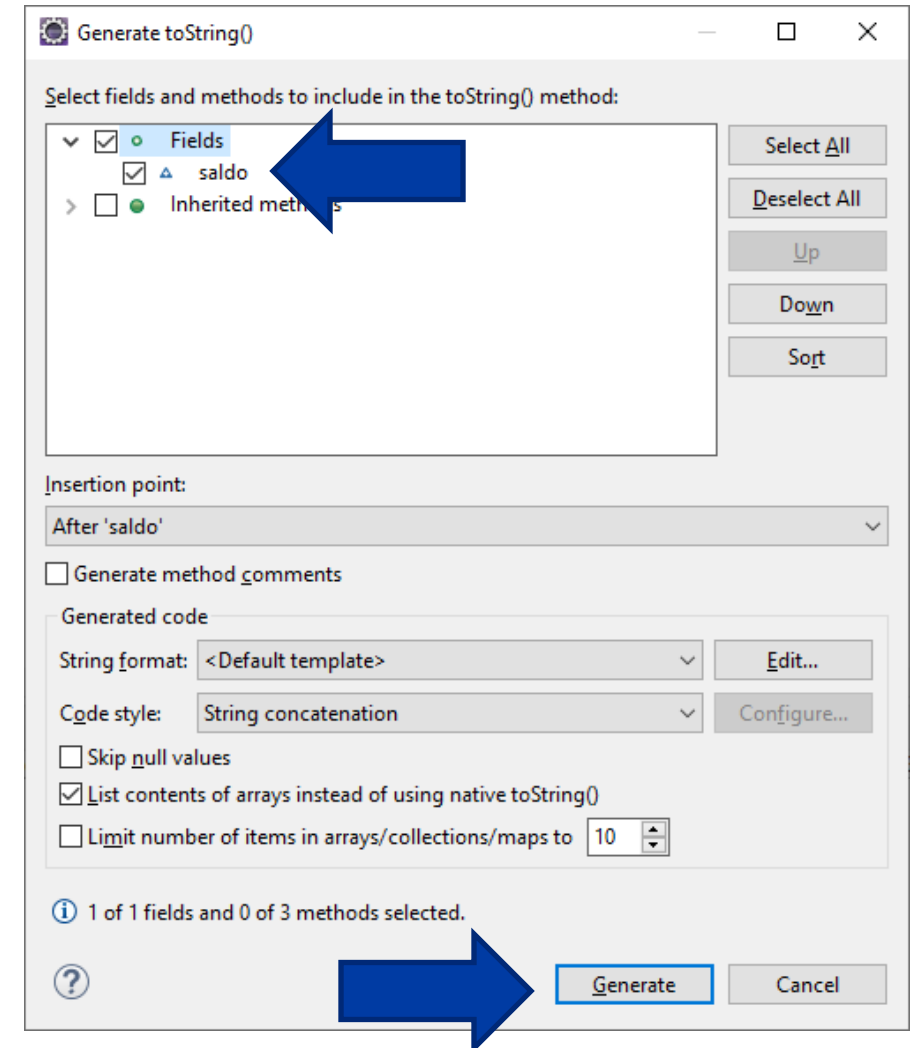


**Inatel**

1. Na classe **Conta**, dar o comando para o Eclipse gerar o método toString():

\*Ctrl+3: **tostr** + Enter

\*No diálogo, apenas clicar **Generate**





2. Na classe de teste **ContaTest**, declarar novo metodo de teste:

```
@Test
void testConta_toString() {

    Conta conta = new ContaCorrente();
    conta.depositar( 100.00 );
    conta.sacar( 10.0 );

    String str = conta.toString(); //toString() de Conta

    System.out.println( str );
}
```

3. Executar e ver o console

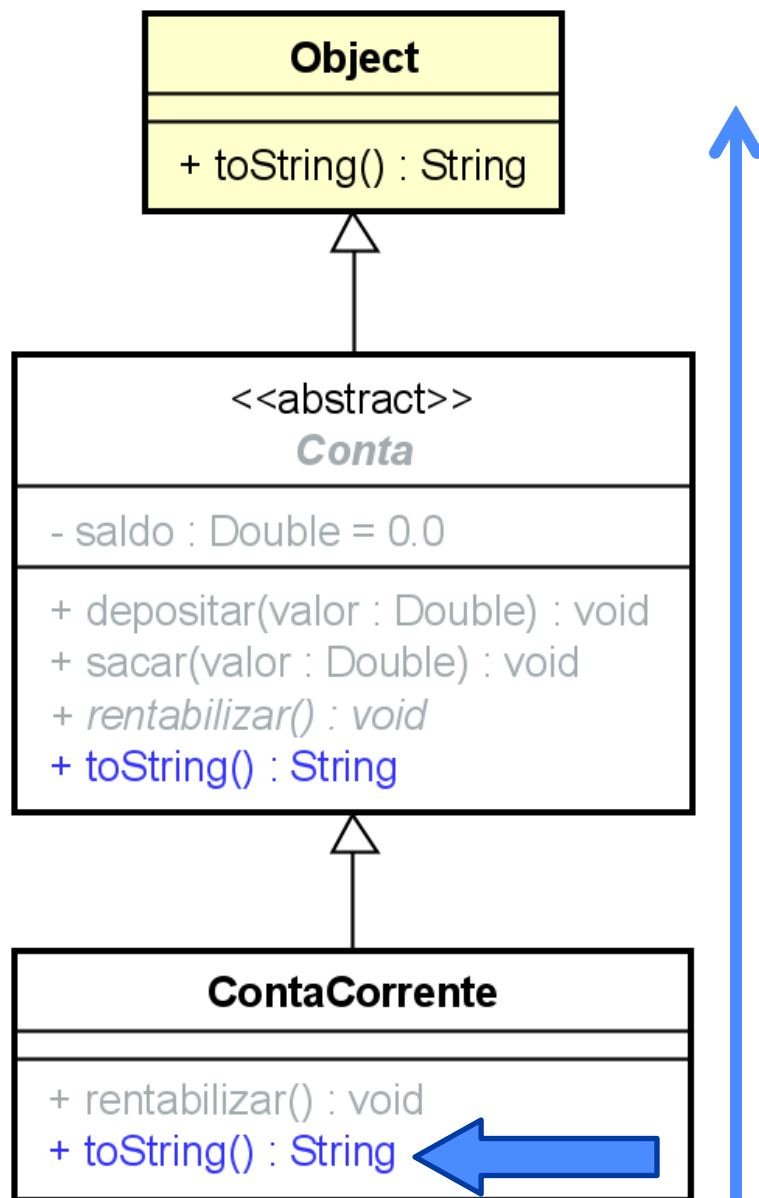




# Polimorfismo: **toString()**



**Inatel**

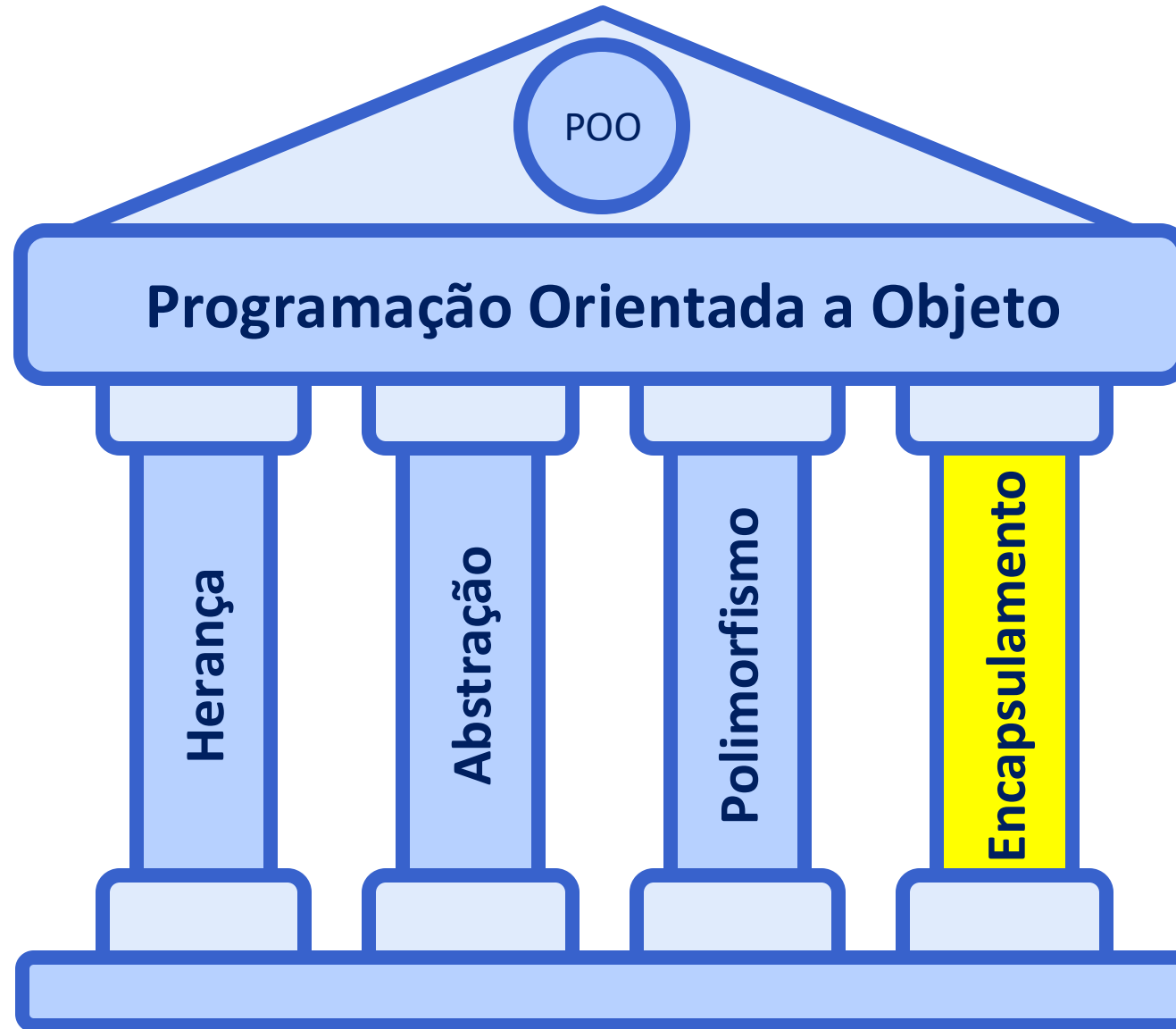


O que aconteceria que na classe **ContaCorrente** também reescrevêssemos o toString() com outra lógica?

O resultado seria o mesmo?

Para saber, vamos gerar toString() (com um código diferente) em **ContaCorrente** e ver o resultado

# Pilares da Programação Orientada a Objetos



- As classes devem ser **guardiãs** de uma lógica específica de negócio
- Muitas vezes esta lógica deve ser interna e, portanto, **inacessível** ao resto do sistema
- Por isso, as classes devem ser estruturas de software do tipo **caixa preta** e cujo acesso devem ser **o mais restrito possível**.

- No Java, existem palavras reservadas denominadas **modificadores de acesso** e servem para o encapsulamento de atributos e métodos

Modificador de acesso	Nível de restrição
<b>private</b>	Máximo: acessível somente dentro da classe
<b>protected</b>	Médio: acessível pelas sub-classes
<b>public</b>	Mínimo: acessível de fora da classe

# Encapsulamento :: Regra geral



**Inatel**

Como regra geral, deixamos:

- os atributos sempre **private**
- os métodos sempre **public**

Raramente, usamos **protected**



- JavaBean é um **padrão de código** sobre encapsulamento.
- Foi proposta pela criadora do Java (*Sun Microsystems*)

## Regras:

1. Não declarar construtor (construtor default será gerado pelo compilador)
2. Declarar todos os **atributos** como **private**
3. Para cada atributo, declarar um **par** de métodos **public**:
  - \* um de **leitura** com prefixo get
  - \* um de **escrita** com prefixo set

## *Para saber mais*

1

É comum usar o termo **POJO** (*Plain Old Java Object*) para descrever classes JavaBean.

2

**DTO** (*Data Transfer Object*) também um Design Pattern que usa a idéia de JavaBeans



1. Declarar pacote **estoque**
2. Declarar a classe **Produto** com os atributos privados:
  - \***descricao** (String)
  - \***valorCompra** (Double)
  - \***dataValidade** (LocalDate)
3. Com o Eclipse, gerar os métodos **getter** e **setters**  
**Ctrl+3: gg**as (Generate Getters And Setters)





```
public class Produto {  
  
    private String descricao;  
  
    private Double valorCompra;  
  
    private LocalDate dataValidade;  
  
    //Ctrl + 3: ggas (Generate Getters And Setters)
```





## 4. Declarar a classe de teste **ProdutoTest** e escrever **testeProdutoComoJavaBean()**

```
@Test
void testeProdutoComoJavaBean() {

    Produto p = new Produto();
    p.setDescricao("Queijo Minas 1Kg");
    p.setValorCompra( 50.00 );
    p.setDataValidade( LocalDate.of(2022, Month.DECEMBER, 30) );

    System.out.println("Dados do produto:");
    System.out.println( p.getDescricao() );
    System.out.println( p.getValorCompra() );
    System.out.println( p.getDataValidade() );
}
```

