Programação Modular

Modularidade e Fundamentos de Programação Orientada por Objetos

Prof. João Caram

Programação modular

- Técnica de projeto (*design*) de software.
- Decomposibilidade de software.
- Cada ***módulo*** é uma unidade independente que provê uma funcionalidade específica de um programa.
- Módulos se comunicam.

Programação modular

- Imprescindível para a construção de sistemas de software modernos
  - Flexibilidade
  - Reutilização
  - Manutenção

Orientação por objetos

- Orientação por objetos abstrai o mundo real utilizando objetos que interagem entre si.

- Utiliza o princípio da decomposibilidade para desenvolver sistemas modulares
  - Quais são os componentes de um jogo digital?
  - Quais são os componentes de uma nota fiscal eletrônica?

Orientação por objetos

- Análise Orientada para Objetos (OOA/AOO):
  - Examina os requisitos de um sistema de uma perspectiva de classes e objetos, usando o vocabulário do domínio do problema

Orientação por objetos

- Projeto Orientado por Objetos (OOD/DOO):
  - Projeto no qual o processo da decomposibilidade em objetos é utilizado e modelos físicos e lógicos de objetos são descritos.

Orientação por objetos

- Programação Orientada por Objetos (OOP/POO):
  - Implementação de programas organizados como coleções de objetos cooperativos

Orientação por objetos

- Análise orientada por objetos

- Projeto orientado por objetos

- Programação orientada por objetos

Programação orientada por objetos

- *"I'm sorry that I long ago coined the term 'objects' for this topic because it gets many people to focus on the lesser idea. The big idea is 'messaging'."* (Alan Kay[1])

1 – Pai dos conceitos de POO, GUI, notebooks e da linguagem Smalltalk. Guitarrista e compositor profissional de jazz, cenógrafo profissional e organista clássico amador

- POO, segundo Alan Kay
  - troca de mensagens;
  - proteção e retenção local e ocultamento do estado ou processo;
  - associação tardia e dinâmica de tudo o que for possível.

Linguagens OO

- Java, C#, Python, Eiffel, Object Pascal.
  - Porém: tipos básicos

- C++, Ada, Perl, PHP entre outras:
  - incluem o conceito de classes e objetos, mas não são consideradas LPs OO em um sentido mais rigoroso.

Classes e Objetos

Classe

- *Descrição* de um Tipo Abstrato de Dados (TAD)

- Constituída por *atributos* (dados/características) e *métodos* (ações/comportamento)

Classe

- Um conjunto de entidades semelhantes compõem uma classe de objetos.

- Ex: Classe *automóvel*
  - **Características**: placa, velocidade atual, km percorridos, combustível no tanque
  - **Ações**: Alterar velocidade, atualizar km percorridos, verificar combustível no tanque, abastecer

Objetos

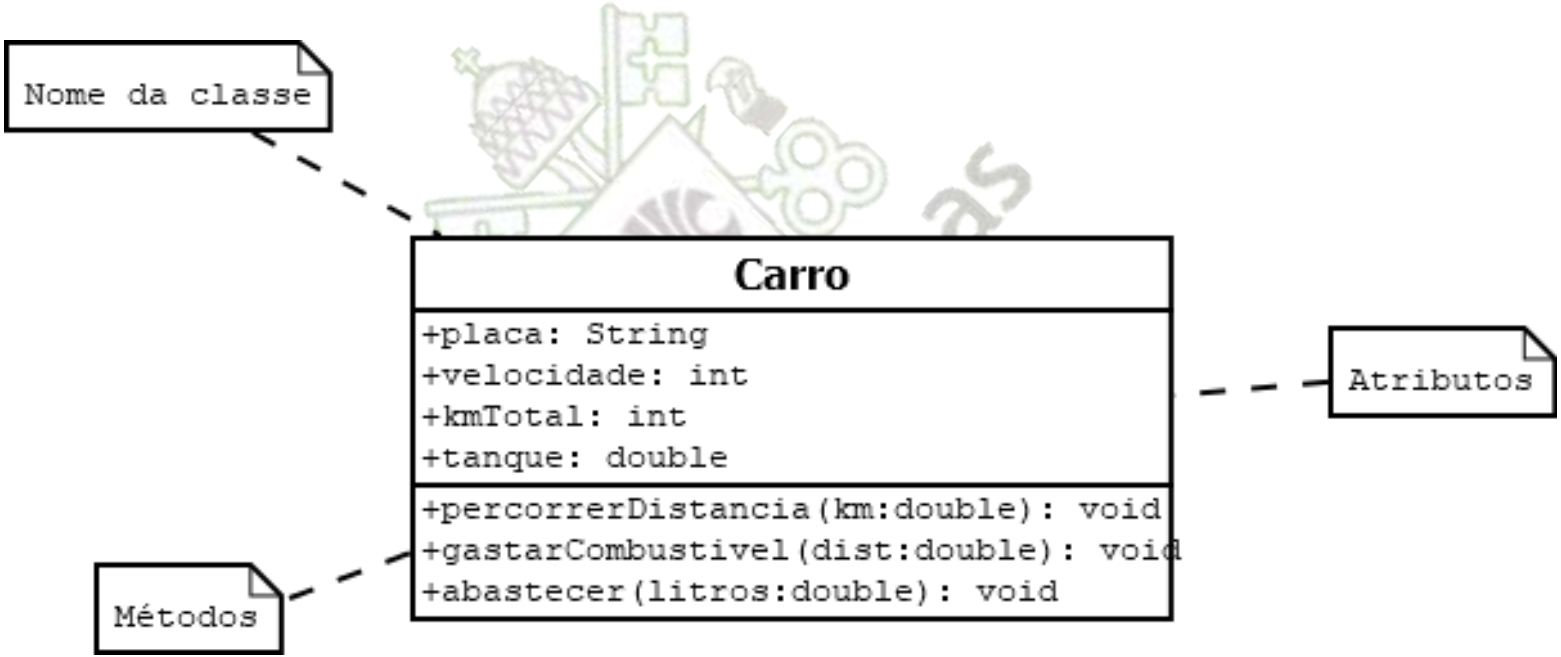
- Um objeto representa uma entidade referenciável (identificada) de uma classe.
  - *Instância* de uma classe

```
GAL0013
84 km/h
1908 km
36,7 litros
```

```
PUC2022
47 km/h
25798 km
11,2 litros
```

Classes: representação UML

- Classe *carro*

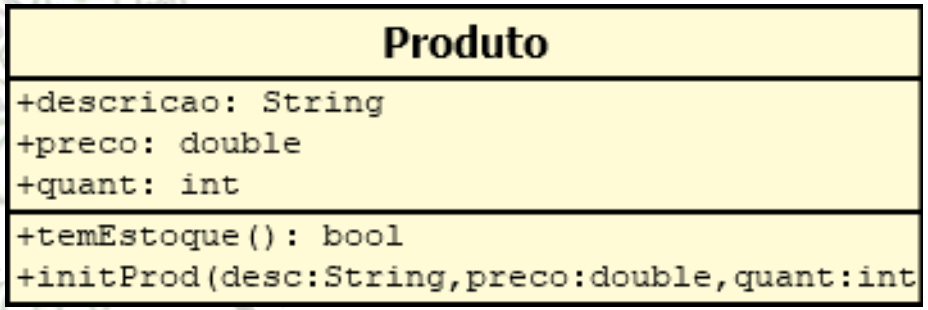


Classes

- Exemplo/exercício: crie uma classe *Produto* para um sistema de gerenciamento de estoque

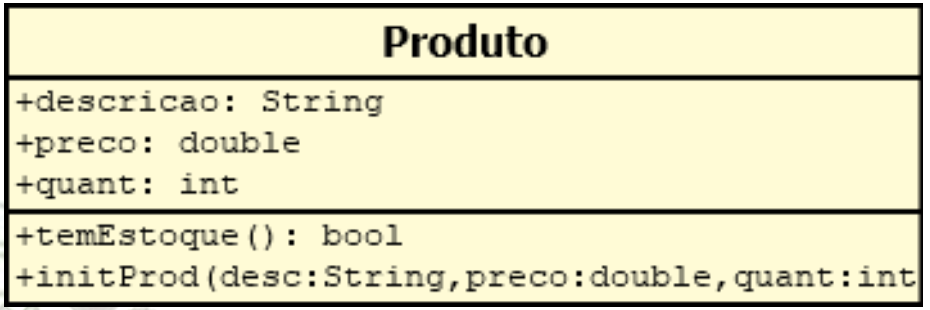
Classes

- Classe *Produto* para gerenciamento de estoque
- Atributos:
  - descricao: String
  - preco: double
  - quant: int
- Métodos:
  - temEstoque(): boolean
  - inicializarProduto(String, double, int)



Produto

```
class Produto {
        String descricao;
        double preco;
        int quant;

        boolean temEstoque(){
                return ( quant > 0 );
        }

        void initProd(String desc, double preco, int quant){
                this.descricao = desc;
                this.preco = preco;
                this.quant = quant;
        }
}
```



Usando a classe Produto

```
class Aplicacao {
        public static void main(String args[]){
                Produto p = new Produto();
                p.descricao = "Xulambs";
                p.preco = 1.99;
                p.quant = 200;
                System.out.println("Produto: " + p.descricao);
                System.out.println("Preço : " + p.preco);
                System.out.println("Estoque : " + p.quant);
                if(p.temEstoque())
                        System.out.println("Produto em estoque.");
                else
                        System.out.println("Produto em falta.");
        }
}
```

Usando a classe Produto

```
class Aplicacao {
	public static void main(String args[]){
		Produto p = new Produto();
		p.descricao = "Xulambs";
		p.preco = 1.99;
		p.quant = 200;
		System.out.println("Produto: " + p.descricao);
		System.out.println("Preço : " + p.preco);
		System.out.println("Estoque : " + p.quant);
		if(p.temEstoque())
			System.out.println("Produto em estoque.");
		else
			System.out.println("Produto em falta.");
	}
}
```

Usando a classe Produto

```
class Aplicacao {
        public static void main(String args[]){
                Produto p = new Produto();
                p.descricao = "Xulambs";
                p.preco = 1.99;
                p.quant = 200;
                System.out.println("Produto: " + p.descricao);
                System.out.println("Preço : " + p.preco);
                System.out.println("Estoque : " + p.quant);
                if(p.temEstoque())
                        System.out.println("Produto em estoque.");
                else
                        System.out.println("Produto em falta.");
        }
}
```

Modularidade - Coesão

Coesão

- Coesão: Qualidade de uma coisa em que todas as partes estão ligadas umas às outras[1]
- Objetivo de um módulo em POO:
  - alta coesão interna;
  - dependência intramodular;
  - utilizar informações internas de forma coerente para resolver um problema específico.

1 - Dicionário Priberam da Língua Portuguesa, 2008-2022. Disponível em ‹https://dicionario.priberam.org/coes%C3%A30›, acessado em Fev/2022

Alta coesão

- Induz a independência funcional.
- Facilita a manutenção.
- Reduz efeitos colaterais e propagação de erros.

Coesão e projeto OO

- POO, segundo Alan Kay
  - troca de mensagens
  - proteção, ***retenção local*** e ocultamento do estado
  - associação tardia e dinâmica de tudo o que for possível

Modularidade - Acoplamento

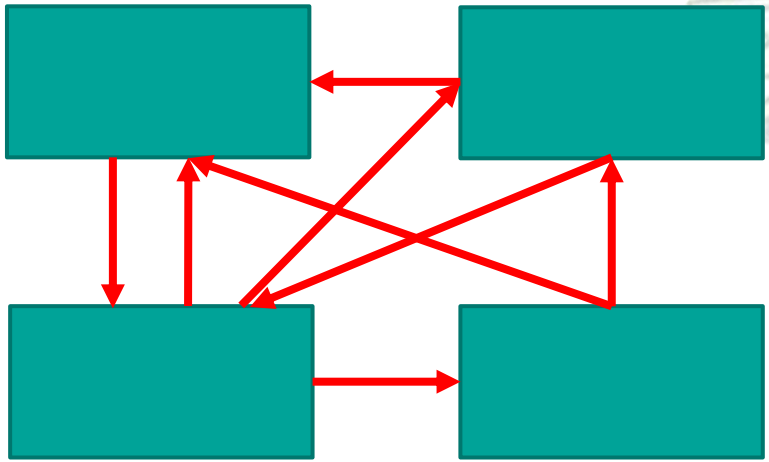
Independência funcional

- Módulo: *"grupo de comandos com uma função bem definida e o mais independente possível em relação ao resto do algoritmo"*

- A dependência pode ser medida pela quantidade de conexões entre os elementos de software

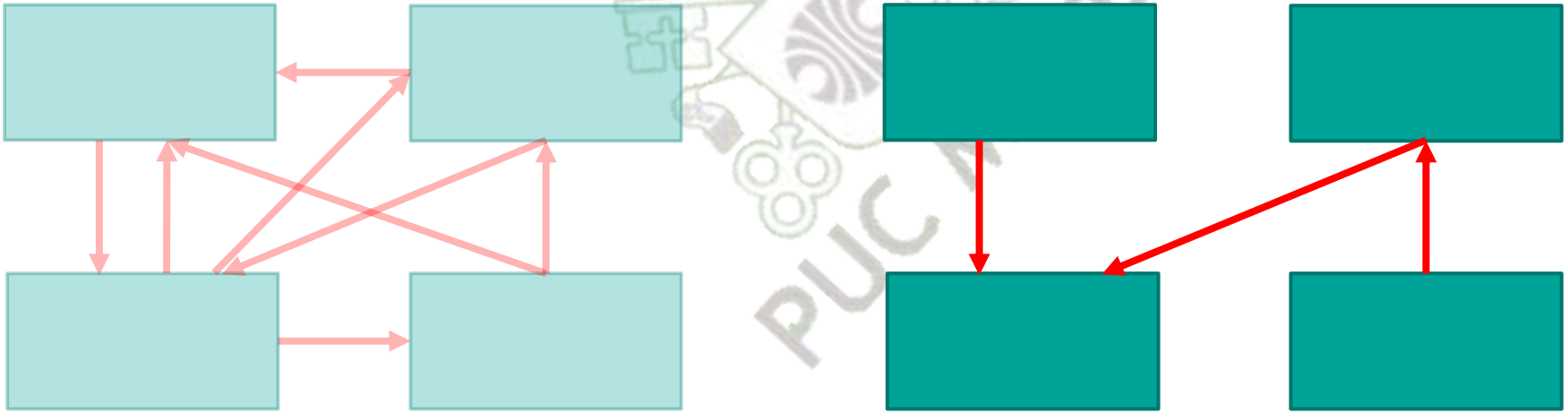
Acoplamento

- Medida da interconexão entre elementos de software



Acoplamento

- Situação ideal: baixo acoplamento



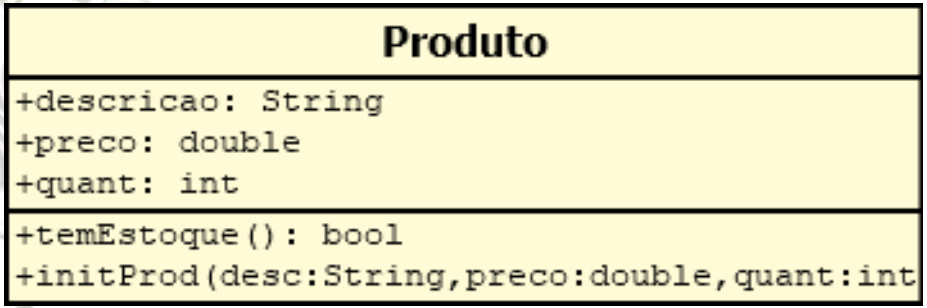
Baixo acoplamento: indicadores

- Tamanho
  - Quantidade parâmetros e métodos públicos
- Visibilidade
  - Uso de parâmetros x Uso de variáveis globais
- Flexibilidade
  - Facilidade na chamada (abordaremos no futuro)

Modularidade - Abstração

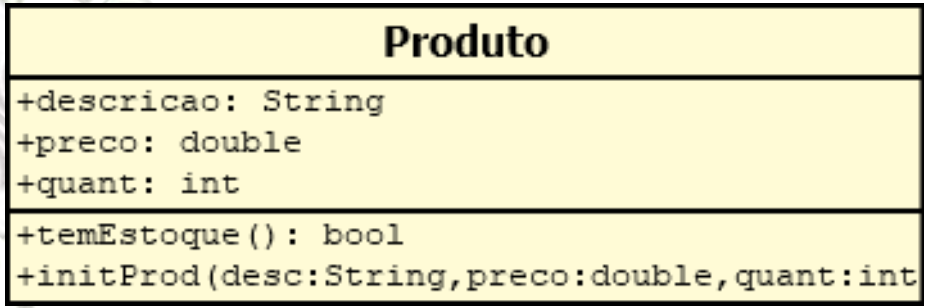
Modelando classes

- Uma classe *Produto*



Modelando classes

- Quem define como é modelado o produto?
  - Inclui o fabricante?
  - Tem data de validade?
  - Indica a unidade?
  - Se é vendido, precisa calcular ICMS?



Abstração

"Capacidade de ver uma operação complexa de uma forma simplificada"

"Visualização ou representação de uma entidade que inclui somente os atributos de importância em um contexto particular" (Sebesta, 2000)

Abstração

- Um módulo deve prover uma boa abstração da função pela qual é responsável
  - Pessoa → cadastro de cidadãos (identidade)
  - Pessoa → cadastro acadêmico
  - Pessoa → cadastro na *Epic Store*

- Além disso...

Abstração

- Conceito da caixa-preta:
  - Entrada e saída bem conhecidas
  - Detalhes ocultos

- Ideia principal: não é necessário saber detalhes do funcionamento de um objeto para utilizá-lo

Abstração e projeto OO

- POO, segundo Alan Kay
  - troca de mensagens
  - proteção, retenção local e *ocultamento do estado*
  - associação tardia e dinâmica de tudo o que for possível

MODULARIDADE - ENCAPSULAMENTO

Encapsulamento

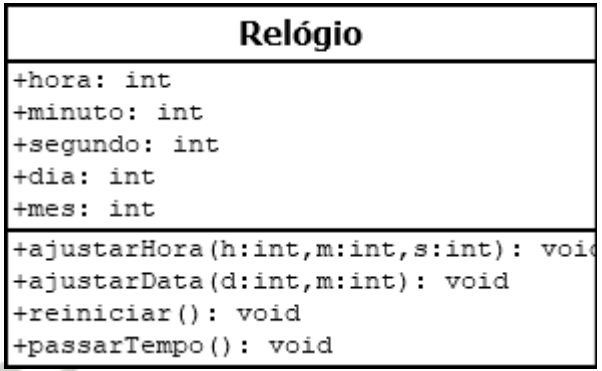
- *Encapsular*: *"incluir ou proteger em uma cápsula ou como em uma cápsula"*[1]

- Em POO, faz parte do princípio da *ocultação de informação* e do conceito de caixa preta.

1-Dicionário Michaelis Online. Disponível em ‹ https://michaelis.uol.com.br/busca?id=kQQD›, acessado em Fev/2022

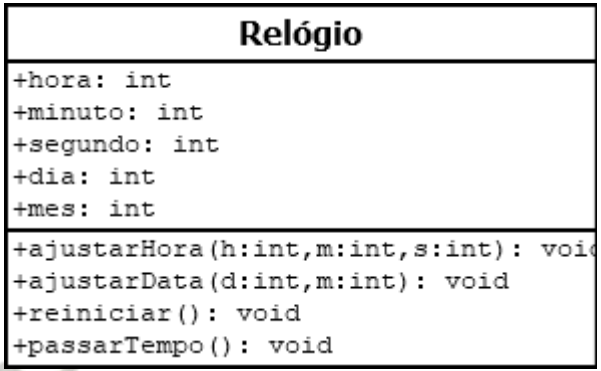
Ex: um relógio e encapsulamento

```
class Relogio {
   public int hora;
   public int minuto;
   ....
}

Relogio meuRelogio = new Relogio();
meuRelogio.hora = 16;
meuRelogio.minuto = 93;
```



Ex: um relógio e encapsulamento

```
class Relogio {
   public int hora;
   public int minuto;
   ....
}

Relogio meuRelogio = new Relogio();
meuRelogio.hora = 16;
meuRelogio.minuto = 93;
meuRelogio.passarTempo();
```



Encapsulamento

- Separar aspectos visíveis de um objeto ou classe de seus detalhes de implementação

Encapsulamento

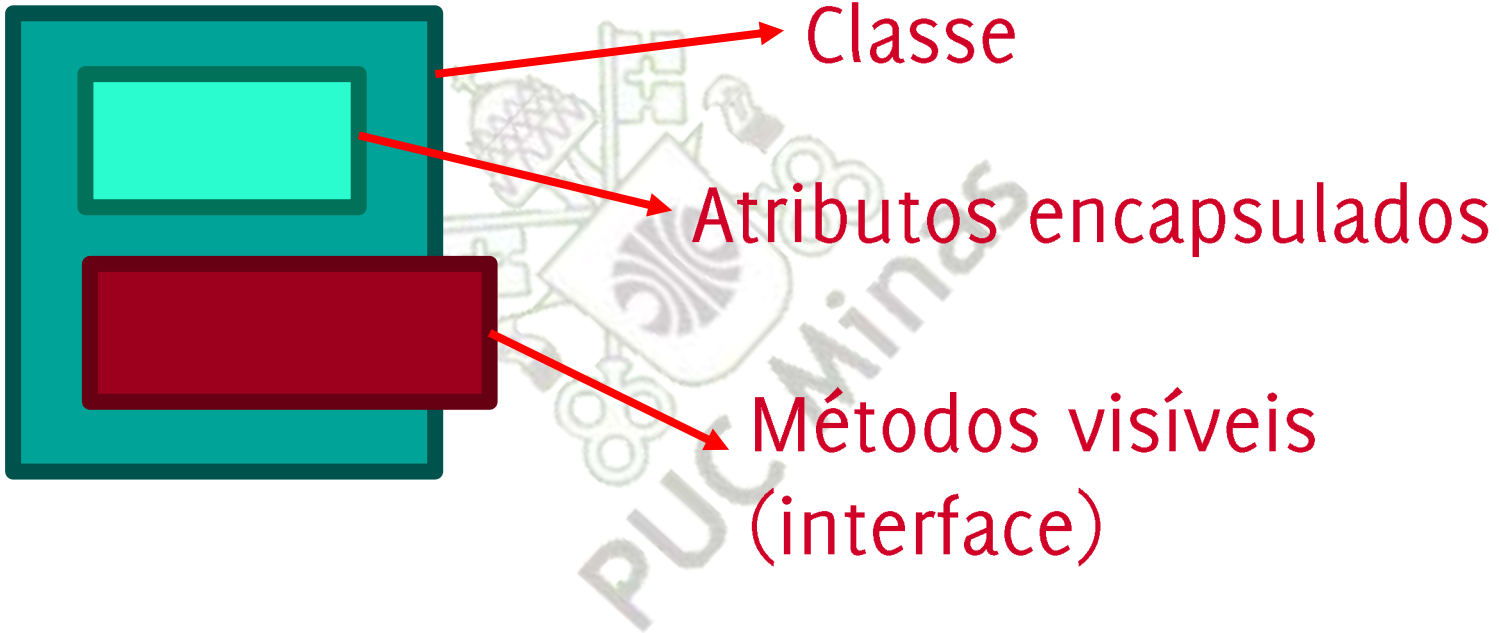
- Separar aspectos visíveis de um objeto ou classe de seus detalhes de implementação

- Detalhes ocultos, interface exposta
  - Interface: aquilo que o usuário do objeto vê

Encapsulamento

- Permite alteração na implementação sem impactos em outros módulos do sistema
- Favorece a abstração
- Protege dados/atributos do acesso direto a partir de um código externo

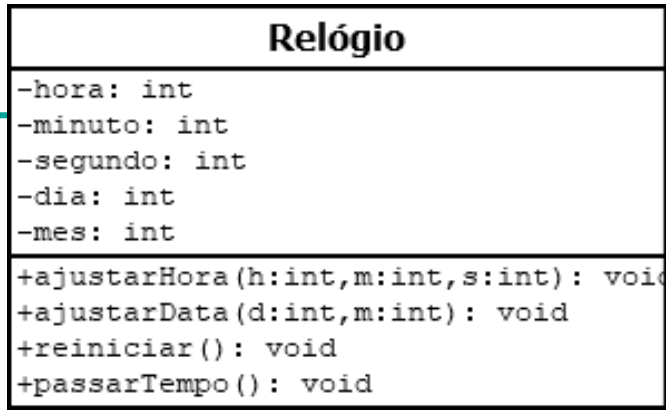
Encapsulamento



Encapsulamento e projeto OO

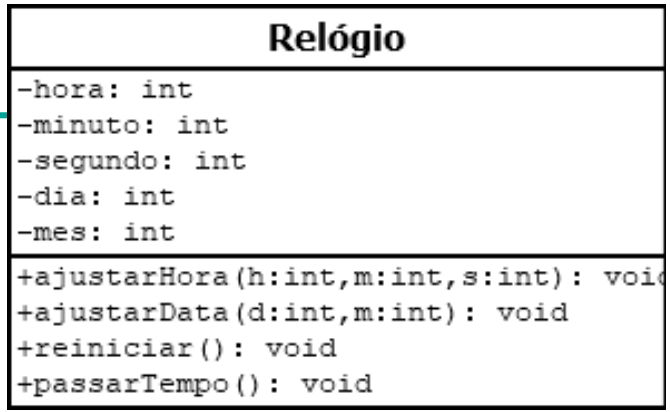
- POO, segundo Alan Kay
  - troca de mensagens
  - *proteção*, retenção local e *ocultamento do estado*
  - associação tardia e dinâmica de tudo o que for possível

Encapsulamento




```
class Relogio {
  private int hora;
  private int minuto;
  ....
      public void ajustarHora(....);
}

Relogio meuRelogio = new Relogio();
meuRelogio.ajustarHora(19,55,00);
```

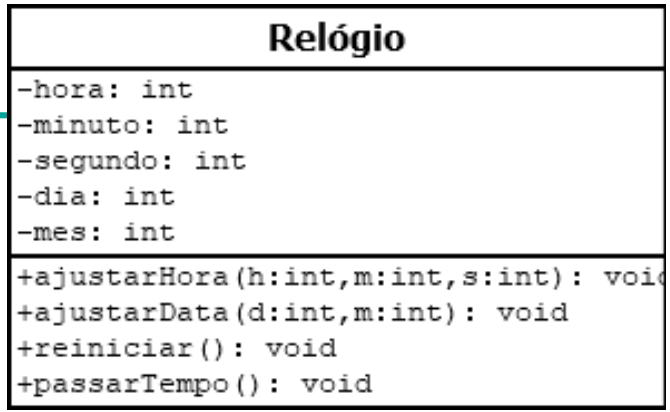
Encapsulamento




```
class Relogio {
   private int hora;
   private int minuto;
   ....
        public void ajustarHora(....);
}

Relogio meuRelogio = new Relogio();
meuRelogio.ajustarHora(18,51,93);
```

Encapsulamento

```
class Relogio {
   private int hora;
   private int minuto;
   ....
        public void ajustarHora(....);
}

Relogio meuRelogio = new Relogio();
meuRelogio.ajustarHora(18,51,93);
```

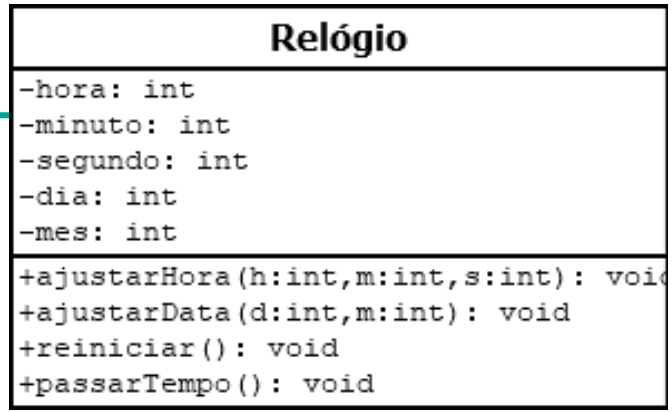


Encapsulamento

- Internamente, a classe pode se referir a seus componentes utilizando a palavra-chave *this*
  - Clareza
  - Desambiguação

Encapsulamento

```
class Relogio {
 public void ajustarHora(....){
            if(h>=0 && h <=23)
                  this.hora = h;
            else
                  this.hora = 0;
 }
 ....
}
meuRelogio.ajustarHora(18,51,93);
```



Encapsulamento

- Os atributos de uma classe só deveriam ser acessados e modificados por meio de seus métodos.

- Um método de uma classe só deve existir se ele tiver relação com os atributos da classe.

Encapsulamento

- Na criação de uma classe, o encapsulamento de atributos e métodos se dá pela definição de suas visibilidades (acessibilidades ou níveis de acesso)

Níveis de acesso e modificadores

- Os níveis de acesso típico em linguagens OO são:
  - Público
  - Privado
  - Protegido
- As diferentes linguagens usam palavras-chaves específicas (*modificadores de acesso*), bem como podem implementar níveis particulares

Níveis de acesso

- Público (acesso irrestrito)
  - Classe, atributos ou métodos visíveis em qualquer parte
- Privado (acesso totalmente restrito)
  - Classe, atributos ou métodos visíveis somente para quem os declarou

Níveis de acesso

- Protegido
  - Classe, atributos ou métodos visíveis para quem o declarou e para módulos derivados deste
  - Aplicação na *especialização com herança* *(veremos mais adiante)*

Modificadores (Java)

- Em Java, temos os modificadores:
  - Public
  - Private
  - Protected
  - *Default (package)*

Níveis de acesso (Java)

- *Default (package)*
  - Não é necessário declarar no Java
  - Classe, atributos e métodos visíveis para quem todos que fazem parte do mesmo pacote Java
    - *Não há visibilidade para subclasses de outros pacotes*

Níveis de acesso (Java)

Modificador	Classe	Pacote	Subclasse	Mundo
public	sim	sim	sim	sim
protected	sim	sim	sim	não
default	sim	sim	não	não
private	sim	não	não	não

Getters and setters

- Se os atributos devem ser privados:
  - Como atribuir valor a eles?
  - Como ler seus valores?

Getters and setters

- Métodos *get:* servem "apenas" para acessar o valor de um atributo privado
- Métodos *set:* servem "apenas" para atribuir um valor a um atributo privado
- Por recomendação, em Java os *getters* e *setters* usam o prefixo e o nome do atributo original

Getters and setters (Java)

```
public class Hortifruti {
      private String _nome;
      private String _unidade;
      private int _quantidade;
      private boolean _organico;
      private double _preco;
...
```

Getters and setters (Java)

```
public double getPreco() {
	return _preco;
}

public void setPreco(double novoPreco) {
	if(novoPreco>0)
		_preco = novoPreco;
}
```

Getters and setters

- Debate:
  - É uma boa idéia termos indiscriminadamente métodos *getters* e *setters* na classe?

Obrigado.

Dúvidas?