# Desenvolvimento de Software

Introdução a OO

Encapsulamento

Prof. Rhafael Freitas da Costa

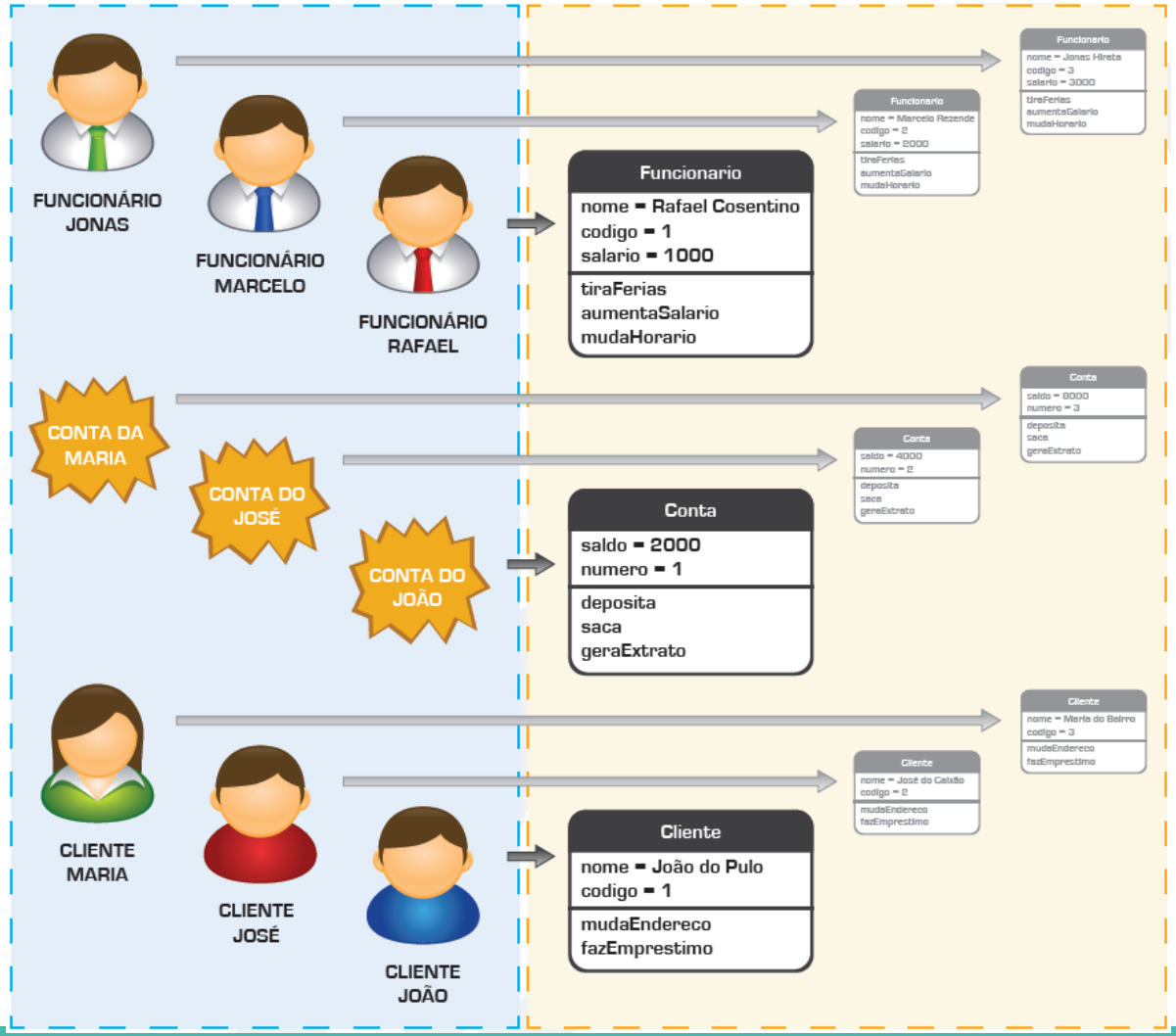
# Conteúdo

- Compreender o conceito de encapsulamento.
- Entender a importância do encapsulamento na programação orientada a objetos.
- Aprender a aplicar encapsulamento em classes e objetos.

# Revisão

# Orientação a Objetos

- No mundo da Orientação a Objetos (POO), os objetos são a base fundamental para a construção de software robusto e flexível. Mas o que exatamente é um objeto? E quais características o definem?

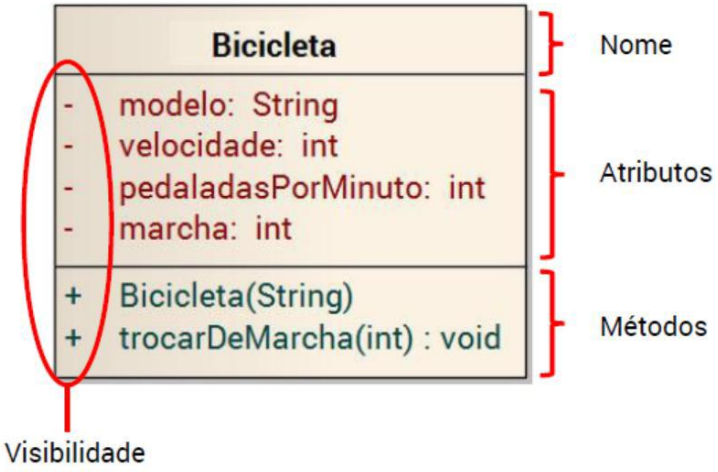

FUNCIONÁRIO JONAS
FUNCIONÁRIO MARCELO
FUNCIONÁRIO RAFAEL
Funcionario
nome = Rafael Cosentino
codigo = 1
salario = 1000
tiraFerias
aumentaSalario
mudaHorario
Funcionario
nome = Marcelo Rezende
codigo = 2
salario = 2000
tiraFerias
aumentaSalario
mudaHorario
Funcionario
nome = Jonas Hirata
codigo = 3
salario = 3000
tiraFerias
aumentaSalario
mudaHorario
CONTA DA MARIA
CONTA DO JOSÉ
CONTA DO JOÃO
Conta
saldo = 2000
numero = 1
deposita
saca
geraExtrato
Conta
saldo = 4000
numero = 2
deposita
saca
geraExtrato
Conta
saldo = 8000
numero = 3
deposita
saca
geraExtrato
CLIENTE MARIA
CLIENTE JOSÉ
CLIENTE JOÃO
Cliente
nome = João do Pulo
codigo = 1
mudaEndereco
fazEmprestimo
Cliente
nome = José do Caixão
codigo = 2
mudaEndereco
fazEmprestimo
Cliente
nome = Maria do Bairro
codigo = 3
mudaEndereco
fazEmprestimo

# Orientação a Objetos

- **Classes:** São modelos/entidades que possuem atributos (características) e métodos (funções) para a criação e manipulação dos objetos e seus dados; define elementos de mesma natureza. Modela características comuns a estes elementos. As classes são definições estáticas que possibilitam o entendimento e comportamento de um grupo de objetos.

- **Objetos:** São elementos/representações criados(as) a partir de uma classe que agrupam, armazenam e manipulam os dados relacionados ao objeto; uma instância de uma classe. Os objetos são abstrações de entidades que existem no mundo real.

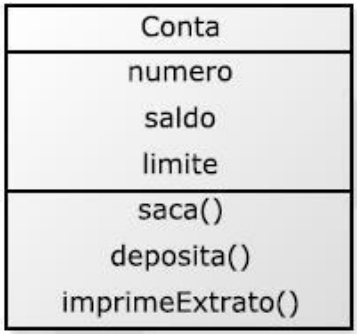
# Classe (é um tipo evoluído personalizado)



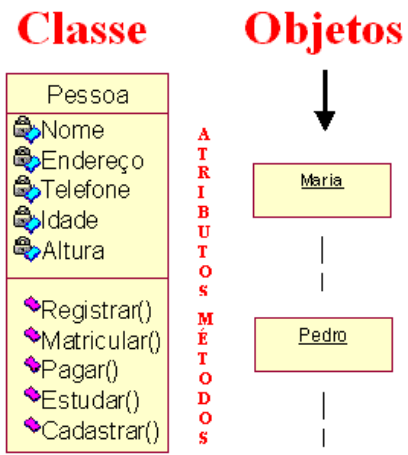

```java
class Bicicleta {
    private String modelo;
    private int velocidade =  0;
    private int pedaladasPorMinuto =  0;
    private int marcha =  1;

    Bicicleta(String modelo) {
        this.modelo = modelo;
    }
    void trocarDeMarcha(int novaMarcha) {
        marcha = novaMarcha;
    }
}
```

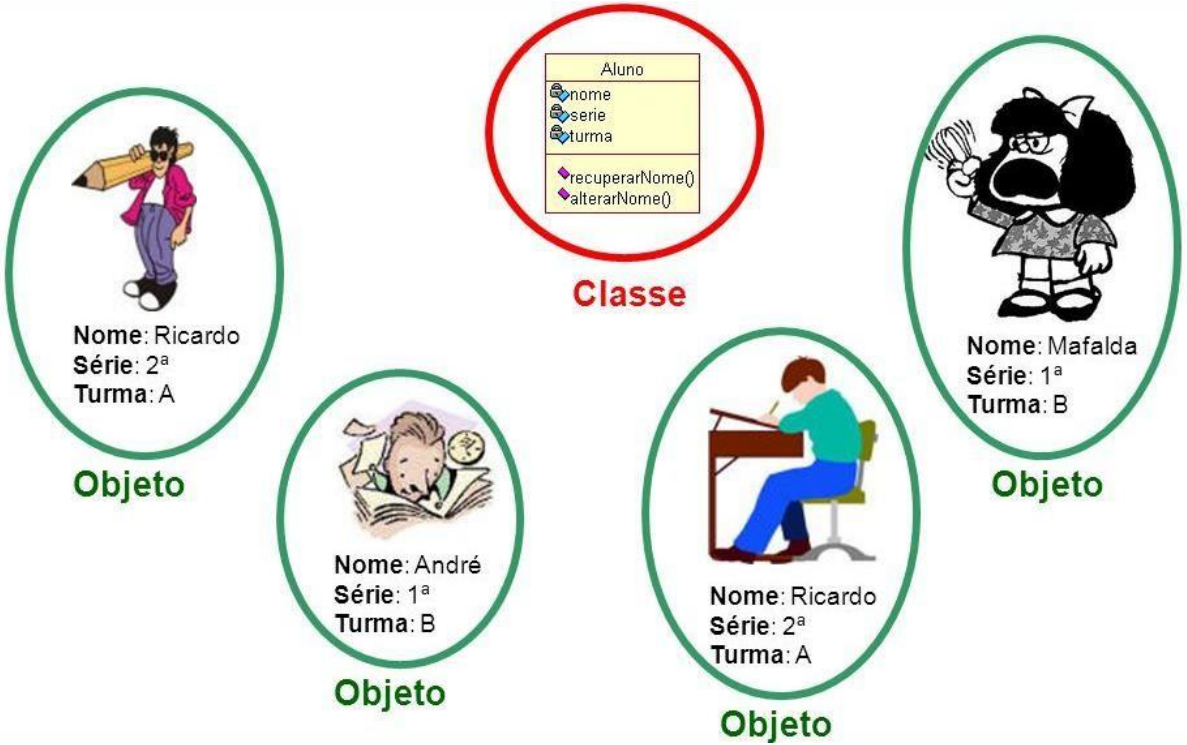
# Orientação a objetos

- **Classes:** Antes de um objeto ser criado, devemos definir quais serão os seus atributos e métodos. Essa definição é realizada através de uma **classe** elaborada por um programador. A partir de uma classe, podemos construir objetos na memória do computador que executa a nossa aplicação.
- Representação de uma classe com diagrama UML:
  1. Nome da classe
  2. Atributos
  3. Métodos



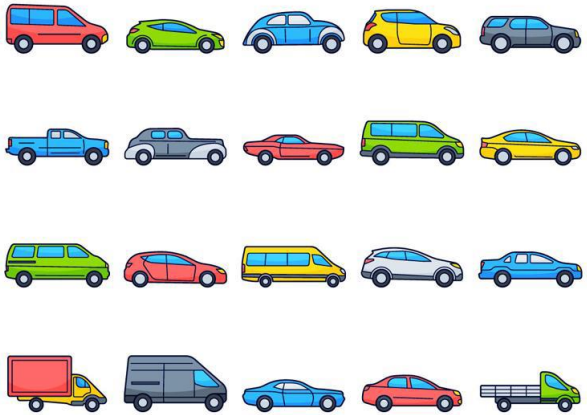
# Orientação a objetos



# Abstração

- Construção de um modelo para representar algo da realidade;
- Foco apenas em aspectos essenciais;
- Preservação da simplicidade do projeto.



# Modularidade

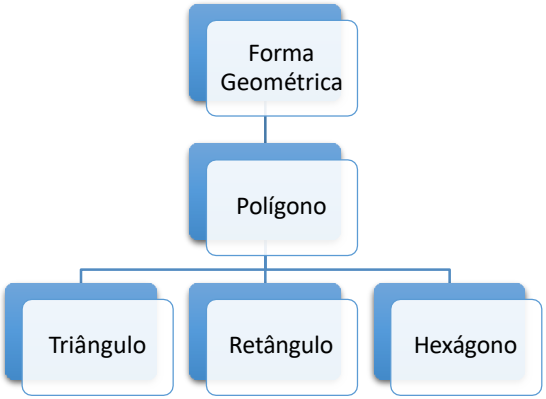
- Quebrar algo complexo em partes menores;
- Facilita o entendimento;
- Cada parte pode ser desenvolvida separadamente.

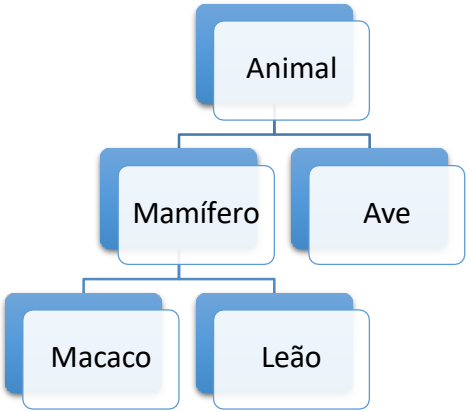
# Encapsulamento

- As informações do objeto ficam encapsuladas, como se fosse uma caixa protegida;
- Evita que as informações sejam corrompidas por entidades externas;
- Mudanças internas não impactam os clientes;
- Manutenções mais baratas e fáceis.

# Hierarquia

- Define níveis de abstração;
- Base conceitual para que o software seja extensível;
- Permite reuso de código/comportamento.




# Revisão

- Conceito de objeto, entidade mais próxima do mundo real;
- Objeto deve ser abstrato, modular, encapsulado, hierárquico;
- **Abstrato**, foca em aspectos essenciais ao projeto, e nada mais;
- **Modular**, pois o complexo é quebrado em partes menores;
- **Encapsulado**, pois o objeto é responsável por seus próprios dados;
- **Hierárquico**, pois permite a construção de modelos extensíveis;

# Encapsulamento

# O que é Encapsulamento?

- O encapsulamento é um pilar fundamental da Orientação a Objetos em Java. Ele se baseia na ideia de **ocultar os detalhes de implementação** de um objeto, protegendo seus dados e métodos internos contra acessos indesejados. Isso significa que apenas os métodos da própria classe podem manipular seus atributos diretamente, garantindo a **integridade e segurança** do código.

# Ocultando detalhes de implementação

- Imagine uma classe `Pessoa` que armazena o nome e a idade de um indivíduo. O encapsulamento nos permite **esconder como esses dados são armazenados** internamente, seja em um array, um map ou qualquer outra estrutura de dados. O que realmente importa é a **interface pública** da classe, que define como os usuários podem interagir com ela através de métodos como `getNome()` e `setIdade()`.

# Interface pública

- A interface pública de uma classe define os métodos e atributos que podem ser acessados por outras classes. Ela é como um **contrato** que estabelece como o objeto pode ser utilizado, **promovendo modularidade e reutilização de código**.

# Modificadores de acesso

- Modificadores de acesso são palavras-chave em Java que controlam a visibilidade de classes, atributos, métodos em relação a outras classes. Eles definem quem pode acessar esses elementos do código, garantido **segurança, modularidade e flexibilidade**. Existem quatro modificadores de acesso em Java:

# Modificador de acesso public

- O modificador **`public`** torna a classe, atributo, método ou construtor acessível de qualquer lugar, ou seja, de qualquer classe ou pacote.

# Modificador de acesso private

- O modificador **`private`** restringe o acesso à classe, atributo, método ou construtor apenas à própria classe em que está definido. Nenhuma outra classe, mesmo dentro do mesmo pacote, pode acessá-lo diretamente.

# Modificador de acesso protected

- O modificador **`protected`** torna a classe, atributo, método ou construtor acessível dentro do mesmo pacote ou através de herança. Isso significa que outras classes no mesmo pacote ou subclasses podem acessá-lo.

# Modificador de acesso default

- O modificador de acesso (**`default ou package-private`**) é considerado o acesso padrão. Isso significa que a classe, atributo, método ou construtor é acessível apenas dentro do mesmo pacote.

# Modificadores de acesso

# Por que Encapsular?

- O encapsulamento é uma ferramenta poderosa que contribui para a criação de **códigos mais seguros, robustos, flexíveis, fáceis de manter e com interfaces públicas intuitivas**. Essa técnica é fundamental para o desenvolvimento de software de alta qualidade em Java.

# Segurança de Dados

- Ao **encapsular** os atributos de uma classe e torná-los privados, é possível **controlar como esses dados** são acessados e modificados. Métodos `getters` e `setters` podem ser usados para garantir que os dados sejam acessados e modificados de maneira segura e controlada. Isso ajuda a prevenir modificações não autorizadas nos dados, garantindo a integridade dos mesmos.

# Ocultação de Detalhes de Implementação

- O encapsulamento permite ocultar os detalhes de implementação de uma classe, expondo apenas uma interface pública. Isso significa que outros componentes do sistema podem interagir com a classe apenas através de métodos públicos, sem precisar conhecer os detalhes internos da implementação. Isso facilita a manutenção do código, já que as mudanças internas na classe não afetam o restante do sistema, desde que a interface pública permaneça inalterada.

# Facilidade de Manutenção

- Ao encapsular os atributos e métodos de uma classe, torna-se mais fácil realizar alterações na implementação interna da classe sem afetar outros componentes do sistema. Isso promove um código mais modular e coeso, facilitando a manutenção e a evolução do software ao longo do tempo.

# Redução do Acoplamento

- O encapsulamento ajuda a reduzir o acoplamento entre as diferentes partes de um sistema, promovendo um design mais flexível e extensível. Alterações em uma classe encapsulada geralmente têm um impacto mínimo em outras partes do sistema, desde que a interface pública permaneça consistente. Isso torna o código mais fácil de entender, modificar e estender.

# Exemplo

```
public class Pessoa {
   private String nome;
   private int idade;
   // Getter para o atributo nome
   public String getNome() {
      return nome;
   }
   // Setter para o atributo nome
   public void setNome(String nome) {
      this.nome = nome;
   }
   // Getter para o atributo idade
   public int getIdade() {
      return idade;
   }
   // Setter para o atributo idade
   public void setIdade(int idade) {
      this.idade = idade;
   }
}
```

# Boas Práticas de Encapsulamento

- **Manter Atributos Privados**: Sempre que possível, declare os atributos como privados para proteger os dados da classe contra acesso não autorizado.

# Boas Práticas de Encapsulamento

- **Utilizar Métodos Getters e Setters**: Em vez de permitir acesso direto aos atributos, forneça métodos getters e setters para acessar e modificar os valores dos atributos. Isso ajuda a controlar como os dados são manipulados e permite a validação dos valores.

# Boas Práticas de Encapsulamento

- **Evitar Exposição Excessiva**: Não exponha mais do que o necessário na interface pública da classe. Mantenha apenas os métodos essenciais para a interação com objetos da classe, ocultando detalhes de implementação desnecessários.

# Praticar

# Exercício 1

- **Classe de Conta Bancária:**

- Crie uma classe `ContaBancaria` com campos privados para o saldo (`saldo`) e métodos públicos para depositar (`depositar`), sacar (`sacar`) e obter o saldo (`obterSaldo`). No método depositar, verifique se o valor depositado é positivo antes de atualizar o saldo. No método sacar, verifique se há saldo suficiente antes de efetuar o saque. Garanta que o saldo nunca seja diretamente acessível fora da classe.

# Exercício 2

- **Classe de Produto:**

- Crie uma classe `Produto` com campos privados para o nome (`nome`), preço (`preco`) e quantidade em estoque (`quantidadeEstoque`). Implemente métodos públicos para atualizar o preço (`atualizarPreco`) e a quantidade em estoque (`atualizarEstoque`), garantindo que os novos valores sejam válidos (por exemplo, preço não negativo, quantidade não negativa). Forneça métodos para obter informações sobre o produto.

# Exercício 3

- **Classe de Ponto 2D:**

- Implemente uma classe `Ponto2D` com campos privados para as coordenadas `x` e `y`. Forneça métodos públicos para definir as coordenadas (`definirCoordenadas`) e obter essas coordenadas (`obterCoordenadas`). Verifique se as coordenadas estão dentro de limites aceitáveis, como valores reais.

# Exercício 4

- **Classe de Pessoa:**

- Crie uma classe `Pessoa` com campos privados para o nome (`nome`), idade (`idade`) e gênero (`genero`). Implemente métodos públicos para definir e obter esses atributos, garantindo que a idade seja um número positivo e o gênero seja um valor válido (por exemplo, masculino, feminino, não-binário). Considere adicionar validações adicionais, como limites de idade ou restrições no formato do nome.

# Exercício 5

- **Classe de Carro:**

- Implemente uma classe `Carro` com campos privados para a marca (`marca`), modelo (`modelo`), ano de fabricação (`anoFabricacao`) e quilometragem (`quilometragem`). Forneça métodos públicos para atualizar a quilometragem (`atualizarQuilometragem`) e obter informações sobre o carro. Garanta que a quilometragem não possa ser definida como um valor negativo. Considere adicionar métodos para realizar a manutenção do carro, como troca de óleo ou revisão.

# Dúvidas

# Obrigado!

Prof. Rhafael Freitas da Costa