


# Java

## Orientação a Objetos





**Java já nasceu orientado  
o objetos.**



**Tudo em Java é associado a classes e objetos. Juntamente com seus atributos e métodos.**

**Por exemplo, na vida real um carro é um objeto tem atributos como peso, cor, quantidade de passageiros. Bem como métodos, como acelerar, freia, etc**



```
package JavaPOO;
```

```
public class ClassePessoa {  
    String nome;  
    public void comer(){  
        System.out.println("Está comendo");  
    }  
  
}
```

```
package JavaPOO;
```

```
public class Pessoa {
```

```
    String nome;
```

```
    public void comer(){
```

```
        System.out.println("Está comendo");
```

```
    }
```

```
}
```



```
package JavaPOO;
```

```
public class ClasseTeste {
```

```
    public static void main(String[] args) {  
        ClassePessoa aluno01 = new ClassePessoa();  
        aluno01.nome = "Wellington";  
        System.out.println(aluno01.nome);  
        aluno01.comer();  
    }  
}
```



```
package JavaPOO;
```

```
public class ClasseTeste {
```

```
    public static void main(String[] args) {  
        ClassePessoa aluno01 = new ClassePessoa();  
        aluno01.nome = "Wellington";  
        System.out.println(aluno01.nome);  
        aluno01.comer();  
    }  
}
```


```
package JavaPOO;
```

```
public class SegundaClasse {  
    public static void main(String[] args) {  
        ClassePessoa aluno01 = new ClassePessoa();  
        ClassePessoa aluno02 = new ClassePessoa();  
        aluno01.nome="João";  
        aluno02.nome = "Maria";  
        System.out.println(aluno01.nome);  
        System.out.println(aluno02.nome);  
    }  
}
```



```
package JavaPOO;
```

```
public class SegundaClasse {  
    public static void main(String[] args) {  
        ClassePessoa aluno01 = new ClassePessoa();  
        ClassePessoa aluno02 = new ClassePessoa();  
        aluno01.nome="João";  
        aluno02.nome = "Maria";  
        alunos01.comer();  
        alunos02.comer();    }  
}
```



**É comum trabalharmos com variáveis, contudo as variáveis dentro de uma classe são chamadas de atributos ou campos**



```
package JavaPOO;
```

```
public class Atributos {
```

```
    String fname = "Wellington";
```

```
    String lname = "Oliveira";
```

```
    int idade = 42;
```

```
}
```

```
package JavaPOO;
```

```
public class UsoClasseAtributos {
```

```
    public static void main(String[] args) {  
        AtributosClasses cliente01 = new AtributosClasses();  
        System.out.println(cliente01.fnome);  
        System.out.println(cliente01.lnome);  
        System.out.println(cliente01.idade);  
        System.out.printf("O aluno %s %s tem %d anos",  
        cliente01.fnome, cliente01.lnome, cliente01.idade);  
  
    }
```

**Um método é um bloco de código  
que só é executado quando é chamado.**

**Você pode passar dados, conhecidos  
como parâmetros, para um método.**


**Os métodos são usados para  
executar certas ações e também  
são conhecidos como funções.**

**Por que usar métodos?**

**Para reutilizar o código: defina o  
código uma vez e use-o várias vezes.**

## Criar um método

Um método deve ser declarado dentro de uma classe. É definido com o nome do método, seguido de parênteses (). Java fornece alguns métodos predefinidos, como **System.out.println()**, mas você também pode criar seus próprios métodos para executar determinadas ações:



```
public class Main {  
    static void meuMetodo() {  
        // Código do método  
    }  
}
```




**package Metodos;**

**public class JavaMetodos {**

**public int idade() {  
 return 25;  
}**

**public double valor() {  
 return 25.50;  
}**





```
String nome() {  
    return "Wellington";  
}  
boolean estado() {  
    return true;  
}  
}
```



# **Assinatura de um método**

**A assinatura do método em Java consiste no nome do método e a lista de parâmetros.**

**A assinatura do método não inclui o tipo de retorno do método.**

**Uma classe não pode ter dois métodos com a mesma assinatura.**

**Se tentarmos declarar dois métodos com a mesma assinatura, você obterá um erro de tempo de compilação.**

```
package Metodos;  
public class CalcularMetodos {  
    int somar (int a, int b) {  
        return a + b;  
    }  
    int somar (int a, int b , int c ) {  
        return a + b + c;  
    }  
}
```

# Construtores

Também conhecido como constructors, os construtores são os responsáveis por criar o objeto em memória, ou seja, instanciar a classe que foi definida. Eles são obrigatórios e são declarados conforme a seguir.

# Construtores

```
public class Carro{
```

```
/* CONSTRUTOR DA CLASSE Carro  
*/
```

```
public Carro(){
```

```
//Faça o que desejar na construção  
do objeto
```

```
}
```

# Construtores

**Por padrão, o Java já cria esse construtor sem parâmetros para todas as classes, então você não precisa fazer isso se utilizará apenas construtores sem parâmetros.**

# Construtores

**Por outro lado, se você quiser, poderá criar mais de um construtor para uma mesma classe. Ou seja, posso criar um construtor sem parâmetros, com dois parâmetros e outro com três parâmetros, como vemos no exemplo**



# Construtores

```
public class Carro{  
  
    public String cor;  
    public double preco;  
    public String modelo;  
  
    /* CONSTRUTOR PADRÃO */  
    public Carro(){  
  
    }  
}
```

# Construtores

```
* CONSTRUTOR COM 2 PARÂMETROS */  
public Carro(String modelo, double preco){  
//Se for escolhido o construtor sem a COR do  
veículo  
// definimos a cor padrão como sendo PRETA  
this.cor = "PRETA";  
this.modelo = modelo;  
this.preco = preco;  
}
```

# Construtores

```
/* CONSTRUTOR COM 3 PARÂMETROS */  
public Carro(String cor, String modelo,  
double preco){  
    this.cor = cor;  
    this.modelo = modelo;  
    this.preco = preco;  
}  
}
```

# Construtores

**Temos agora três construtores padrões, ou seja, podemos criar um novo Carro sem definir sua Cor, ou podemos criar um novo carro definindo todos os seus atributos, como no exemplo da Listagem**

# **Construtores**

**Criando Vários carros com os  
construtores**

# Construtores

```
public class LojaDeCarro {  
  
    public static void main(String[] args) {  
        //Construtor sem parâmetros  
        Carro prototipoDeCarro = new Carro();  
  
        //Construtor com 2 parâmetros  
        Carro civicPreto = new Carro("New Civic",40000);  
  
        //Construtor com 3 parâmetros  
        Carro golfAmarelo = new Carro("Preto","Golf", 38000);  
    }  
  
}
```

# Construtores

**Nós também podemos utilizar construtores da classe pai com a palavra reservada “super()”.**

**Assim, dentro do seu construtor chamamos o construtor da classe Pai**

# Construtores

```
public class Honda extends Carro{
```

```
public String motor;
```

```
/* CONSTRUTOR PADRÃO */
```

```
public Honda(){
```

```
}
```



# Construtores

```
/* CONSTRUTOR COM PARÂMETROS */  
public Honda(String modelo, String cor,  
double preco){  
    super(modelo, cor, preco);  
  
}  
  
}
```

# Construtores

```
public class Aplicacao2 {  
  
    public static void main(String[] args) {  
        //Construtor sem parâmetros  
        Honda hondaFitPreto = new Honda("2.0  
Flex", "Honda Accord", 60000);  
    }  
}
```

# Construtores

```
public class Aplicacao {  
  
    public static void main(String[] args) {  
        //Construtor sem parâmetros  
        Honda hondaFitPreto = new Honda("2.0  
Flex", "Honda Accord", 60000);  
    }  
}
```

# Classes e membros

Uma classe é um molde do qual podemos criar objetos. Definimos na classe o comportamento e os estados que terá o objeto do seu tipo. A classe pode e geralmente representa coisas do mundo real, como por exemplo, um carro.

# Classes e membros

Todo programa Java é composto por uma coleção de objetos que podem se comunicar entre si chamando métodos uns dos outros. Além disso, cada objeto tem seu tipo, e o que determina o tipo do objeto é a classe da qual ele foi instanciado. Os objetos são criados quando uma classe é instanciada, o que faz com que um novo modelo da classe fique disponível na memória. Então podemos atribuir valores às suas variáveis e chamar seus métodos.

# Modificadores de acesso

Os modificadores de acesso são palavras-chave na linguagem Java. Eles servem para definir a visibilidade que determinada classe ou membro terá diante das outras. Visibilidade neste caso tem o mesmo significado que acesso, pois se não está visível não pode ser acessado. Para entender como o controle de acesso é feito, primeiramente devemos estudar dois conceitos: níveis de acesso e modificadores de acesso.

# Modificadores de acesso

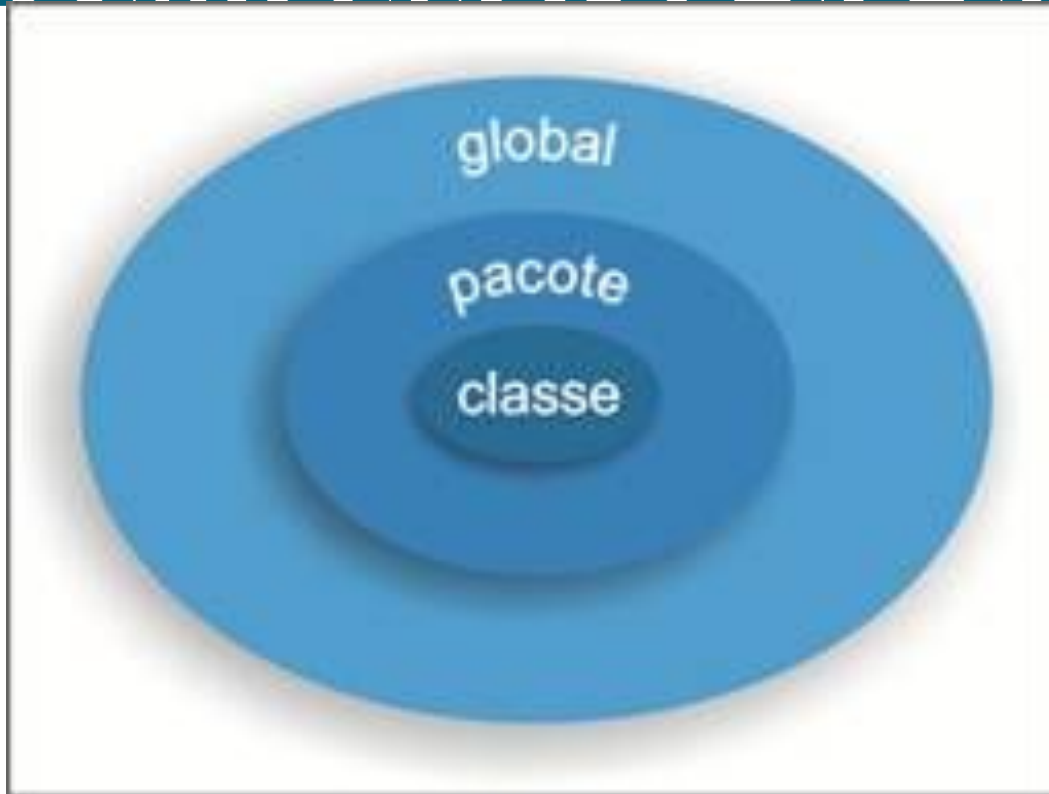
Níveis de acesso são conhecidos por ***public***, ***private***, ***protected*** e ***default***. E os modificadores são apenas três: **public**, **private**, **protected**. O nível de acesso default (padrão) não exige modificador. Quando não se declara nenhum modificador, o nível default é implícito.

# Modificadores de acesso

Para entender como os modificadores trabalham, devemos imaginar a classe ou membros inseridos no ambiente do programa, onde podem existir outras classes e membros. Por exemplo, temos um projeto e nele podemos ter vários pacotes. E cada pacote pode conter várias classes. Então na visão de uma classe, ela está inserida num pacote que está dentro do projeto



# Modificadores de acesso



# Modificadores de acesso

## *public*

Uma declaração com o modificador `public` pode ser acessada de qualquer lugar e por qualquer entidade que possa visualizar a classe a que ela pertence.

# Modificadores de acesso

## *private*

Os membros da classe definidos como ***private*** não podem ser acessados ou usados por nenhuma outra classe. Esse modificador não se aplica às classes, somente para seus métodos e atributos. Esses atributos e métodos também não podem ser visualizados pelas classes herdadas.

# Modificadores de acesso

## *protected*

O modificador `protected` torna o membro acessível às classes do mesmo pacote ou através de herança, seus membros herdados não são acessíveis a outras classes fora do pacote em que foram declarados.

# Modificadores de acesso

## *default (padrão):*

A classe e/ou seus membros são acessíveis somente por classes do mesmo pacote, na sua declaração não é definido nenhum tipo de modificador, sendo este identificado pelo compilador.

# Modificadores de acesso

## *final*

Quando é aplicado na classe, não permite extendê-la, nos métodos impede que o mesmo seja sobrescrito (overriding) na subclasse, e nos valores de variáveis não pode ser alterado depois que já tenha sido atribuído um valor.

# Modificadores de acesso

## ***abstract***

Esse modificador não é aplicado nas variáveis, apenas nas classes. Uma classe abstrata não pode ser instanciada, ou seja, não pode ser chamada pelos seus construtores. Se houver alguma declaração de um método como `abstract` (abstrato), a classe também deve ser marcada como `abstract`.

# Modificadores de acesso

## *static*

É usado para a criação de uma variável que poderá ser acessada por todas as instâncias de objetos desta classe como uma variável comum, ou seja, a variável criada será a mesma em todas as instâncias e quando seu conteúdo é modificado numa das instâncias, a modificação ocorre em todas as demais. E nas declarações de métodos ajudam no acesso direto à classe, portanto não é necessário instanciar um objeto para acessar o método.



# Modificadores de acesso

	private	default	protected	public
mesma classe	sim	sim	sim	sim
mesmo pacote	não	sim	sim	sim
pacotes diferentes (subclasses)	não	não	sim	sim
pacotes diferentes (sem subclasses)	não	não	não	sim

# **Membros de classes x membros de instâncias**

```
package Membros;
```

```
public class DataNascimento {  
    static int dia;  
    int mes;  
    int ano;  
}
```

# Membros de classes x membros de instâncias

```
package Membros;
```

```
public class UsoMembros {  
    public static void main(String[] args) {  
        DataNascimento dt = new DataNascimento();  
        DataNascimento dt2 = new DataNascimento();  
        dt.dia=12;  
        dt.mes=02;  
        dt.ano=1991;  
    }  
}
```

# Membros de classes x membros de instâncias

```
dt2.dia=25;
dt2.mes=01;
dt2.ano=1992;
System.out.printf("A data de Nascimento "
    + "é %d %d %d \n", dt.dia, dt.mes, dt.ano );

System.out.printf("A data de Nascimento "
    + "é %d %d %d", dt2.dia, dt2.mes, dt2.ano );
    }
}
```

# Herança

***Herança é um mecanismo onde uma classe recebe todos os comportamentos e estados de uma classe pai ou super classe.***

# Herança

*A herança é um princípio da POO que permite a criação de novas classes a partir de outras previamente criadas. Essas novas classes são chamadas de subclasses, ou classes derivadas; e as classes já existentes, que deram origem às subclasses, são chamadas de superclasses, ou classes base.*

```
import java.util.Date;  
public class Pessoa {  
    public String nome;  
    public String cpf;  
    public Date data_nascimento;  
  
    public Pessoa(String nome, String cpf, Date data) {  
        this.nome = nome;  
        this.cpf = cpf;  
        this.data_nascimento = data;  
    }  
}
```




**;**

***public class Pessoa {  
 public String nome;  
 public String cpf;  
 public String telefone;***

***public Pessoa(String nome, String cpf, String  
telefone) {  
 this.nome = nome;  
 this.cpf = cpf;  
 this.telefone = data;  
 }}***





***Neste primeiro código, vemos que a classe pessoa possui nome, CPF, e data de nascimento como atributos; além de um construtor, que recebe estes três dados como parâmetro, e assim preenche os atributos do objeto. Na criação de um objeto Pessoa, o programa deve fornecer seus dados.***

```
public class Aluno extends Pessoa {  
    public Aluno(String nome, String cpf,  
String telefone) {  
        super(nome, cpf, telefone);  
    }  
    public String matricula;  
}
```

```
public class Professor extends Pessoa {  
    public Professor(String nome, String  
    cpf, Date data) {  
        super( nome, cpf, telefone);  
    }  
    public double salario;  
    public String disciplina;  
}
```

```
public class Funcionario extends Pessoa {  
    public Funcionario(String nome, String  
    cpf, Date data) {  
        super(nome, cpf, telefone);  
    }  
    public double salario;  
  
    public String cargo;  
}
```

***As novas classes criadas possuem suas características (atributos e métodos) próprias, mas possuem também propriedades comuns: os atributos nome, data de nascimento e CPF. Podemos ver que cada construtor das novas classes possui uma chamada super(\_nome, \_cpf, \_data);***

***A palavra **super** representa uma chamada de método ou acesso a um atributo da superclasse, por isso tem esse nome. No nosso caso, estamos usando o super para invocar construtor da superclasse Pessoa, que recebe os três parâmetros e preenche os atributos do objeto.***

***Então, quando criarmos um objeto do tipo Aluno, por exemplo, utilizando “new Aluno(“nome”, “cpf”, new Date())”, a classe Aluno invocará o construtor Pessoa(String, String, Date), e então seus atributos serão preenchidos com os dados enviados por parâmetro***

```
public class TestePessoa {  
    public static void main(String[] args) {  
        Aluno i = new Aluno("Jose Francisco",  
"123.456.789-00", new Date());  
        System.out.println("Veja como os atributos  
foram preenchidos\n\nNome: " + i.nome);  
        System.out.println("CPF: " + i.cpf);  
        System.out.println("Data de nascimento: " +  
i.data_nascimento.toString());  
    }  
}
```



# Polimorfismo

***Polimorfismo significa "muitas formas", é o termo definido em linguagens orientadas a objeto, como por exemplo Java, C# e C++, que permite ao desenvolvedor usar o mesmo elemento de formas diferentes.***

# Polimorfismo

*Polimorfismo denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem. No Polimorfismo temos dois tipos:*

*Polimorfismo **Estático** ou Sobrecarga*

*Polimorfismo **Dinâmico** ou Sobreposição*

# Polimorfismo Sobrecarga

*O Polimorfismo Estático se dá quando temos a mesma operação implementada várias vezes na mesma classe. A escolha de qual operação será chamada depende da assinatura dos métodos sobrecarregados.*

# **Polimorfismo (Sobrescrita)**

***O Polimorfismo Dinâmico acontece na herança, quando a subclasse sobrepõe o método original. Agora o método escolhido se dá em tempo de execução e não mais em tempo de compilação. A escolha de qual método será chamado depende do tipo do objeto que recebe a mensagem.***

# Polimorfismo

***Crie um Classe que exemplifica os conceitos de polimorfismo de sobrescrita e sobrecarga***

# Polimorfismo

Desenvolva uma a classe chamada **Imovel**, a qual possuirá como atributos um endereço e um preço.

Em seguida, crie uma classe **Novo**, a qual herda Imovel e possui um adicional no preço, crie os métodos de acesso e impressão deste valor adicional.

Após, crie uma classe **Velho**, que herda Imovel e possui um desconto no preço, crie os métodos de acesso e impressão para este desconto. Na classe principal, o programa deverá pedir ao usuário que digite 1 para Imóvel novo e 2 para Imóvel velho. Conforme a definição do usuário, o programa deverá imprimir o endereço e valor final do imóvel

# Encapsulamento

*Encapsulamento é um processo de envolver dados e código em uma única unidade.*

*É como uma capsula que possui uma mistura de diversos medicamentos, é uma técnica que ajuda a manter as variáveis de instância protegidas.*

*Essa proteção pode ser conquistada utilizando o modificador de acesso private, que indica que a variável ou dado não pode ser acessado de fora da classe. Para acessar estados privados de modo seguro, temos que providenciar métodos getters e setters públicos*

# **Vantagens do encapsulamento em Java**

***Podemos fazer uma classe somente leitura ou somente escrita. Para uma classe somente leitura, temos que informar apenas os métodos getters. Para uma classe somente escrita, devemos informar apenas os métodos setters.***

***Controle sobre os dados: podemos controlar os dados adicionando lógica nos métodos setters, assim como fizemos para evitar que o gerenciador de estoques definisse valores negativos nos exemplos acima.***



# **Vantagens do encapsulamento em Java**

***Proteção dos dados: outras classes não podem acessar membros privados de uma classe diretamente.***

# Praticando

***Crie um novo pacote chamado encapsulamento***

***Crie uma classe Numeros***

***Esta classe deve ter um atributo privado do tipo inteiro.***

***crie 2 métodos:***

***o primeiro método deve retornar o conteúdo do atributo***

***o segundo deve alterar o conteúdo do atributo***

***Crie uma classe em Java chamada fatura para uma loja de suprimentos de informática. A classe deve conter quatro variáveis – o número (String), a descrição (String), a quantidade comprada de um item (int) e o preço por item (double).***

***A classe deve ter um construtor e um método get e set para cada variável de instância. Além disso, forneça um método chamado getTotalFatura que calcula o valor da fatura e depois retorna o valor como um double. Se o valor não for positivo, ele deve ser configurado como 0. Se o preço por item não for positivo, ele deve ser configurado como 0.0.***

***Escreva um aplicativo de teste chamado FaturaTeste (em outro arquivo) que demonstra as capacidades da classe Fatura.***

### ***Classe animal de estimação***

Crie uma classe para representar um tipo Pet em java, que deverá ter os seguintes campos:

***nome***. Variável de instância do tipo String que contém o nome de um animal de estimação.

***tipo***. Variável de instância do tipo String que contém o tipo de animal que é um animal de estimação.

***idade***. A variável de instância do tipo int contém a idade do animal de estimação.

A classe Pet também deve ter os seguintes métodos:**construtor para esta classe**. O construtor deve aceitar um argumento para cada um dos campos.

***setName***. O método setName armazena um valor no campo de nome.

***setTipo***. O método setAnimal armazena um valor no campo animal.

***setIdade***. O método setIdade armazena um valor no campo idade.

***getNome***. O método getNome retorna o valor do campo de nome.

***getTipo***. O método getAnimal retorna o valor do campo animal.

***getIdade***. O método getIdade retorna o valor do campo idade.

# Praticando

***Crie uma classe chamada Retângulo, receba 2 valores ( Base e Altura) através do construtor. Esses atributos devem ser privados.***

***Em outra classe instancie a classe Retangulo e chamando os métodos, calcule a area e perimetro***

# Praticando

*crie uma classe chamada Retângulo, com Construtor default esses atributos devem ser privados. usando get e set calcule e mostre a área e perímetro do retângulo*

# Classes abstratas

***Pode-se dizer que as classes abstratas servem como “modelo” para outras classes que dela herdem, não podendo ser instanciada por si só. Para ter um objeto de uma classe abstrata é necessário criar uma classe mais especializada herdando dela e então instanciar essa nova classe. Os métodos da classe abstrata devem então serem sobrescritos nas classes filhas.***

# Interfaces

***As interfaces são padrões definidos através de contratos ou especificações. Um contrato define um determinado conjunto de métodos que serão implementados nas classes que assinarem esse contrato. Uma interface é 100% abstrata, ou seja, os seus métodos são definidos como abstract, e as variáveis por padrão são sempre constantes (static final).***



# Interfaces

*Uma interface é definida através da palavra reservada “**interface**”. Para uma classe implementar uma interface é usada a palavra “implements”,*

# Interfaces

***Como a linguagem Java não permite herança múltipla, as interfaces ajudam nessa questão, pois bem se sabe que uma classe pode ser herdada apenas uma vez, mas pode implementar inúmeras interfaces. As classes que forem implementar uma interface terão de adicionar todos os métodos da interface ou se transformar em uma classe abstrata***

# Interfaces

```
interface Conta{  
    void depositar(double valor);  
    void sacar(double valor);  
    double getSaldo();  
}
```

# Interfaces

```
public class ContaCorrente implements  
Conta {  
    private double saldo;  
    @Override  
    public void deposita(double valor) {  
        this.saldo += valor - taxaOperacao;  
    }  
}
```

# Interfaces


**@Override**

```
public double getSaldo() {  
    return this.saldo;  
}
```


**@Override**

```
public void sacar(double valor) {  
    this.saldo -= valor + taxaOperacao;  
}}
```

**Crie uma variável para armazenar cada um dos seguintes tipos de dados: inteiro, ponto flutuante, caractere e uma string. Atribua valores a essas variáveis e imprima-os.**




**Escreva um programa que leia  
dois números inteiros do  
usuário, some esses números  
e exiba o resultado.**




**Escreva um programa que leia a idade de uma pessoa e exiba uma mensagem dizendo se ela é menor de idade, adulta ou idosa (considere 18 e 65 anos como os limites).**







**Escreva um programa que  
imprima todos os números de  
1 a 50.**



**Escreva um programa que peça ao usuário para digitar um número e continue pedindo até que o número digitado seja zero**



**Escreva um programa que peça ao usuário para digitar um número e continue pedindo até que o número digitado seja zero**



**Crie um array de 5 inteiros,  
preencha com valores lidos do  
usuário e exiba a soma desses  
valores.**

**Escreva um método que receba dois números inteiros como parâmetros e retorne o maior entre eles. Utilize este método no método `main` para testar com diferentes valores**

**Crie uma classe `Pessoa` com os atributos `nome` e `idade`. Crie um construtor para inicializar esses atributos e um método para exibir as informações da pessoa. No método `main`, crie um objeto da classe `Pessoa` e exiba suas informações.**