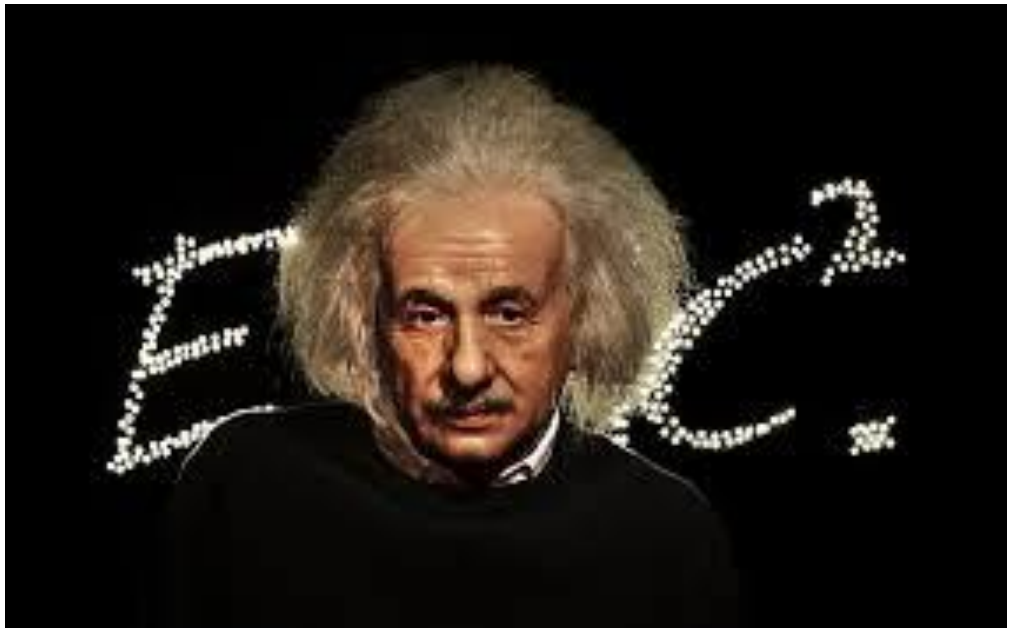
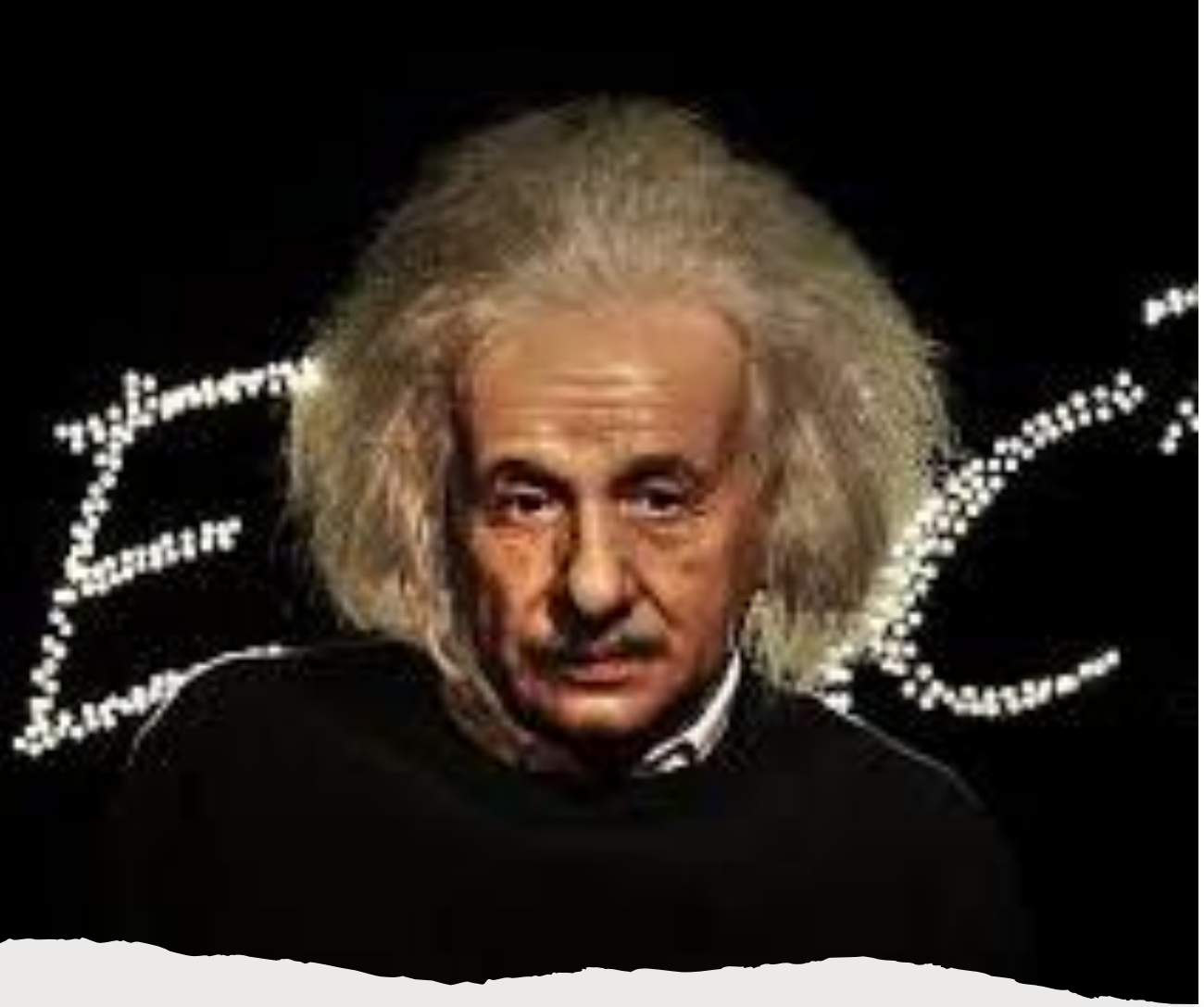




JAVA

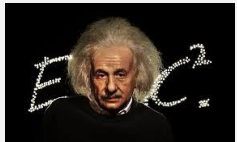
Saiba mais sobre a Linguagem Java

Primeiro....

Existem duas formas principais para se interagir com uma aplicação.

1ª -> acontece passando-se argumentos no momento da execução do programa.

2ª -> se dá por meio da leitura de valores em tempo real



Saiba mais sobre a Linguagem Java

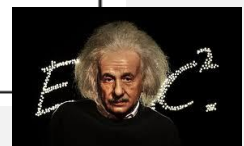
Assinatura do ponto de entrada da aplicação

String[] - Vetor de strings que permite a passagem de quantos argumentos quisermos

args - parâmetro que recebe valores no momento da execução do código.

```
public class ArgsLinhaDeComandoBasico {
    public static void main(String[] args) {
        System.out.printf("qtd de argumentos = %d%n",
args.length);
        for (int i = 0; i < args.length; i++) {
            System.out.printf("\targs[%d] = %s%n", i, args[i]);
        }
    }
}
```

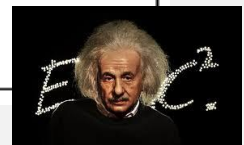
Especificadores de formato



Saiba mais sobre a Linguagem Java

Sequências de scape

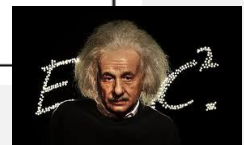
Sequência de Escape	Descrição	Exemplo de utilização
\n	Insere nova linha.	`System.out.print("Introdução\na\nProgramação\ncom\nJava");`
\t	Insere tabulação na horizontal.	`System.out.print("Col A\tCol B\tCol C\tCol D");`
\	Insere barra invertida.	`System.out.print("C:\\Windows\\system32");`
\"	Insere aspa dupla.	`System.out.print("Nome do livro \"Dom Quixote\" de Miguel de Cervantes");`
\r	Realiza retorno do carro.	`System.out.print("Texto Não Mostrado \rEsse Texto Aparece\n");`



Saiba mais sobre a Linguagem Java

Especificadores de formato

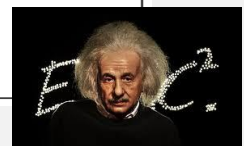
Especificador	Descrição	Exemplo de Utilização	Saída
%d	Valor inteiro em decimal com sinal (pode ser usado para *byte*, *short*, *int* e *long*).	`System.out.print("%d", 127);`	127
%o	Valor inteiro em octal com sinal.	`System.out.print("%o", 127);`	177



Saiba mais sobre a Linguagem Java

Especificadores de formato

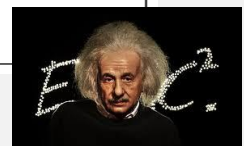
Especificador	Descrição	Exemplo de Utilização	Saída
%x	Valor inteiro em hexadecimal com sinal (minúsculo).	`System.out.printf("%x", 127);`	7f
%X	Valor inteiro em hexadecimal com sinal (maiúsculo).	`System.out.printf("%X", 127);`	7F
%f	Valor real (*float* ou *double*).	`System.out.printf("%f", 3.141592);`	3,141592
%e	Valor real (notação exponencial) (minúsculo).	`System.out.printf("%e", 3.14);`	3,14e+00
%E	Valor real (notação exponencial) (maiúsculo).	`System.out.printf("%E", 3.14);`	3,14E+00
%X	Valor inteiro em hexadecimal com sinal (maiúsculo).	`System.out.printf("%X", 10);`	7F
%b	Valor lógico (*boolean*) (minúsculo).	`System.out.printf("%b", 3 > 2);`	true
%B	Valor lógico (*boolean*) (maiúsculo).	`System.out.printf("%B", 2 > 3);`	FALSE



Saiba mais sobre a Linguagem Java

Exemplo - especificador de formato

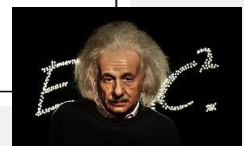
```
1  /**
2   * @Leonardo Rocha
3   */
4
5  public class DefinePeso{
6
7  public static void main(String[] args) {
8
9      String nome = "Leonardo";
10     int idade = 30;
11     double peso = 95.5;
12     System.out.printf("%s, %d anos, pesa %.2f kg!", nome, idade, peso);
13     }
14 }
```



Saiba mais sobre a Linguagem Java

Estrutura de pasta - visão geral

Diretório	Descrição do Conteúdo
src/	Código-fonte da aplicação.
test/	Código de teste unitário (não utilizado neste livro).
lib/	Dependências do projeto.
dist/	Arquivos de distribuição como .jar e suas dependências.
build/	Arquivos gerados pelo processo de compilação.

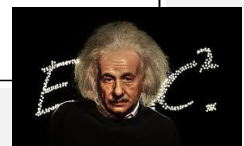


Saiba mais sobre a Linguagem Java

Leitura de dados em tempo real

A leitura de dados em tempo real pode ser feita via classe Scanner, que fornece métodos de leitura com sintaxes diferentes.

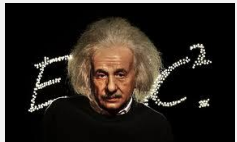
```
import java.util.Scanner;
public class ExemploLeituraDados {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Digite um valor inteiro (int): ");
        int entradaInt = scan.nextInt();
        System.out.print("Digite um valor real (double): ");
        double entradaDouble = scan.nextDouble();
        System.out.print("Digite um valor lógico (boolean): ");
        boolean entradaBoolean = scan.nextBoolean();
        System.out.print("Digite uma string (uma palavra): ");
        String entradaPalavra = scan.next();
```



Saiba mais sobre a Linguagem Java

Construir um código legível e bem documentado é extremamente importante, pois ele necessitará de atualizações e poderá ser lido por outros programadores.

A forma mais simples de documentar um código se dá por meio de comentários

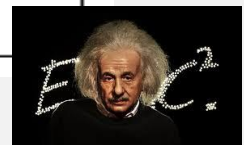


Saiba mais sobre a Linguagem Java

Tipos de comentários

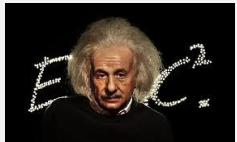
Uma única linha - utiliza-se o comando **//** para inserção do comentário;

Em blocos - utiliza-se o comando **/* conteúdo */** para inserção de comentários.



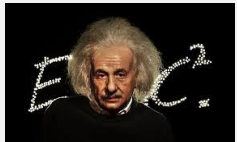
Argumentos variáveis – Varargs

A linguagem Java nos permite chamar um método diretamente passando n valores e os parâmetros enviados são automaticamente adicionados em um Array de mesmo tipo.



Argumentos variáveis – Varargs

Em Java, Varargs (abreviação de "variable arguments") é uma característica que permite a um método aceitar um número variável de argumentos do mesmo tipo. Isso significa que você pode chamar um método com diferentes quantidades de argumentos sem precisar definir múltiplas versões do método com diferentes números de parâmetros.



Argumentos variáveis – Varargs

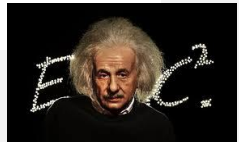
Exemplo para entender ...

Varargs.java

```
1  package explicacao.varargs;
2
3  public class Varargs {
4
5      public static void main(String[] args) {
6
7          int resultado1 = somar(1, 2);// Resultado: 3
8          int resultado2 = somar(1, 2, 3);// Resultado: 6
9          int resultado3 = somar(1, 2, 3, 4, 5);// Resultado: 15
10
11     }
12
13     public static int somar(int... numeros) {
14         int soma = 0;
15         for (int numero : numeros) {
16             soma += numero;
17         }
18         System.out.println(soma);
19         return soma;
20     }
21
```

Problems | Javadoc | Declaration | Console

```
<terminated> Varargs [Java Application] C:\Program Files\RedHat\java-1.8.0-openjdk-1.8.0.212-3\bin\javaw.exe  (01/04/2024 19:52:50 – 19:52:51)
3
6
15
```



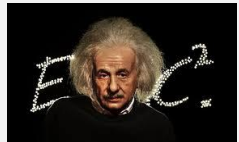
Argumentos variáveis – Varargs

```
    public static int somar(int... numeros) {
        int soma = 0;
        for (int numero : numeros) {
            soma += numero;
        }
        System.out.println(soma);
        return soma;
    }

}
```

É aqui que os Varargs entram em jogo.

Em vez de definir várias versões da função com diferentes quantidades de parâmetros, você pode usar Varargs para lidar com qualquer quantidade de números.

Veja como ficaria:



Argumentos variáveis – Varargs

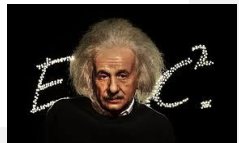
Varargs.java

```java
package explicacao.while_varargs;

public class Varargs {

    public static void main(String[] args) {

        int resultado1 = somar(1, 2);// Resultado: 3
        int resultado2 = somar(1, 2, 3);// Resultado: 6
        int resultado3 = somar(1, 2, 3, 4, 5);// Resultado: 15

    }

    public static int somar(int... numeros) {
        int soma = 0;
        for (int numero : numeros) {
            soma += numero;
        }
        System.out.println(soma);
        return soma;
    }

}
```

Problems | Javadoc | Declaration | Console

<terminated> Varargs [Java Application] C:\Program Files\RedHat\java-1.8.0-openjdk-1.8.0.212-3\bin\javaw.exe (01/04/2024 19:52:50 – 19:52:51)

```
3
6
15
```

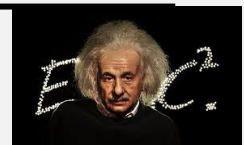


Java – Orientação Objeto

Um paradigma de programação que aproxima a manipulação das estruturas de um programa ao manuseio das coisas no mundo real.

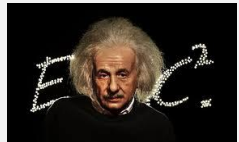
Pode ser considerado uma filosofia que guia todo o desenvolvimento de código.

Considerada a principal diferença para OO, a programação procedural é uma forma de desenvolvimento de código que se caracteriza por utilizar, principalmente, funções e procedimentos como núcleo de organização estrutural.



Paradigmas de programação

Paradigma de programação, a grosso modo, é a forma de escrita (estrutura) de um determinado código, desde que ele a aceite. Existem linguagens que aceitam mais de um paradigma de programação. Um deles, a programação orientada a objetos apresenta como uma das principais características: a reutilização de código, isso é possível, por exemplo, pela construção de classes, outro exemplo é o paradigma estruturado tem como principal característica seguir sequência.

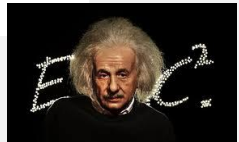


Paradigmas de programação

POO

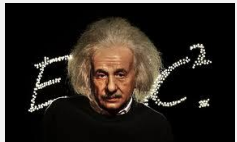
A programação orientada a objetos tem como base a:

1 - Abstração
2 - Encapsulamento
3 - Herança
4 - Polimorfismo



Java - abstração

Imagine que você está construindo um carro. Para a maioria das pessoas, um carro é uma coisa bem simples de entender: tem rodas, um motor, um volante, etc. Mas se você começar a pensar sobre como um carro é realmente construído e como ele funciona, você vai perceber que há muitos detalhes complicados e coisas acontecendo sob o capô.

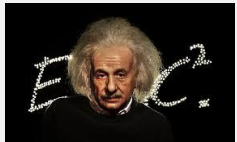


Java - abstração

A abstração em Java é como uma simplificação desse processo. Em vez de lidar com todos os detalhes internos de como as coisas funcionam, você pode pensar em termos mais simples e abstratos.

Por exemplo, vamos pensar em um carro em termos de uma classe em Java. Uma classe em Java é como um modelo ou um plano para criar objetos.

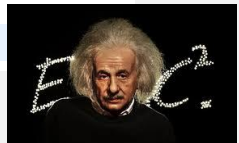
Vamos criar uma classe Carro:



Java - abstração

Carro.java

```java
package explicacao.abstracao;

public class Carro {
	String marca;
	String modelo;
	int ano;

	public Carro(String marca, String modelo, int ano) {
		this.marca = marca;
		this.modelo = modelo;
		this.ano = ano;
	}

	public void ligar() {
		System.out.println("O carro está ligado!");
	}

	public void acelerar() {
		System.out.println("O carro está acelerando!");
	}

}
```

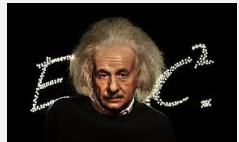


Java - abstração

Aqui, Carro é uma abstração de um carro real. Ele tem atributos como marca, modelo e ano, e métodos como ligar() e acelerar().

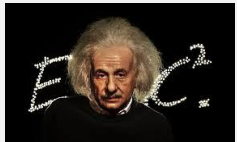
Você pode pensar nessa classe como uma versão simplificada de um carro real.

A abstração permite que você se concentre nos aspectos importantes de um objeto ou sistema sem se preocupar com todos os detalhes internos.



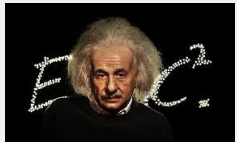
Java - abstração

Por exemplo, quando você usa um carro, não precisa entender como o motor funciona internamente; você só precisa saber como ligar e dirigir o carro. Da mesma forma, ao usar uma classe em Java, você só precisa saber quais métodos ela oferece e como usá-los, sem precisar entender todos os detalhes de como a classe foi implementada internamente.



Java - abstração

Em resumo, a abstração em Java é sobre simplificar as coisas, concentrando-se nos aspectos importantes e ignorando os detalhes internos complicados. Isso torna mais fácil entender e trabalhar com objetos e sistemas complexos.

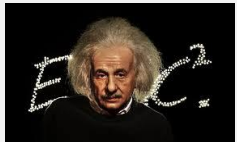


Java - encapsulamento

Encapsulamento em Java é um conceito fundamental da programação orientada a objetos que permite proteger os dados de uma classe, controlando o acesso a eles por meio de métodos públicos.

Imagine que você tem uma caixa, e dentro dessa caixa você guarda algumas coisas importantes.

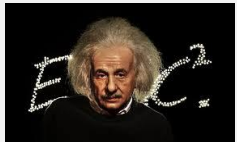
Você não quer que qualquer pessoa possa acessar diretamente o conteúdo da caixa, então você coloca uma tampa nela.



Java - encapsulamento

Agora, para acessar o conteúdo da caixa, as pessoas têm que usar a tampa - elas não podem simplesmente pegar as coisas de dentro da caixa.

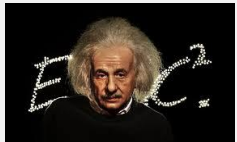
Em Java, uma classe é como essa caixa, e os dados que ela armazena são como as coisas dentro dela. O encapsulamento é como a tampa da caixa - ele protege os dados da classe, controlando o acesso a eles.



Java - encapsulamento

Você faz isso definindo os atributos da classe como privados e fornecendo métodos públicos para acessá-los e modificá-los, se necessário. Esses métodos públicos são conhecidos como métodos getters e setters.

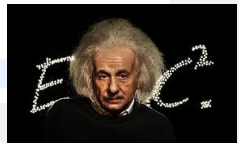
Um método getter é usado para recuperar o valor de um atributo privado, enquanto um método setter é usado para definir ou modificar o valor de um atributo privado. Dessa forma, você tem controle sobre como os dados da classe são acessados e modificados.



Java - encapsulamento

Pessoa.java

```java
package explicacao.encapsulamento;

public class Pessoa {
    private String nome;
    private int idade;

    // Método getter para o atributo 'nome'
    public String getNome() {
        return nome;
    }

    // Método setter para o atributo 'nome'
    public void setNome(String novoNome) {
        this.nome = novoNome;
    }

    // Método getter para o atributo 'idade'
    public int getIdade() {
        return idade;
    }

    // Método setter para o atributo 'idade'
    public void setIdade(int novaIdade) {
        if (novaIdade >= 0) { // Verifica se a idade é positiva
            this.idade = novaIdade;
        } else {
            System.out.println("Idade inválida");
        }
    }
}
```

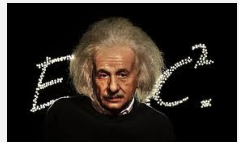


Java - enccapsulamento

A classe **Pessoa** tem dois atributos privados: nome e idade.

Os métodos getters (getNome() e getIdade()) permitem acessar os valores desses atributos, enquanto os métodos setters (setNome() e setIdade()) permitem modificar esses valores, seguindo as regras de validação necessárias (no caso da idade, garantindo que seja um número positivo).

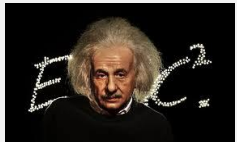
Dessa forma, o encapsulamento ajuda a proteger os dados da classe e a manter a integridade do objeto.



Java - herança

Imagine que você está construindo uma cidade de robôs, e cada robô tem suas próprias características e habilidades especiais. Por exemplo, alguns robôs podem voar, outros podem nadar, alguns podem escavar e assim por diante.

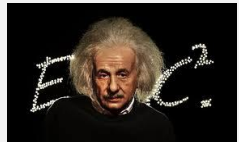
Agora, em vez de construir cada tipo de robô do zero, você pode usar a herança.



Java - herança

Em vez de criar uma nova classe para cada tipo de robô, você pode criar uma classe base, chamada por exemplo de Robo, que contém características e habilidades comuns a todos os robôs.

Em seguida, você pode criar subclasses específicas, como RoboVoador, RoboNadador, RoboEscavador, etc., que herdam todas as características e habilidades da classe Robo, mas também podem ter suas próprias características e habilidades exclusivas.



Java - herança

Em Java, a herança é implementada usando a palavra-chave **extends**

*Robo.java

```java
package explicacao.heranca;

//Classe base
class Robo {
    void mover() {
        System.out.println("O robô está se movendo.");
    }


    // main apenas para executar
    public static void main(String[] args) {

        RoboVoador robozinho = new RoboVoador();
        robozinho.mover();
        robozinho.voar();

    }
}

```

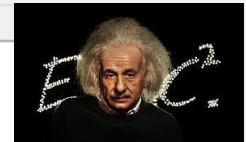
RoboVoador.java

```java
package explicacao.heranca;

//Subclasse que herda de Robo
class RoboVoador extends Robo {
    void voar() {
        System.out.println("O robô está voando.");
    }
}
```

Problems | Javadoc | Declaration | Console

<terminated> Robo [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 16:35:58 – 16:35:58)

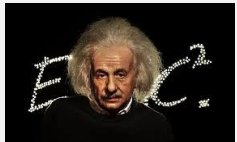
```
O robô está se movendo.
O robô está voando.
```



Java - herança

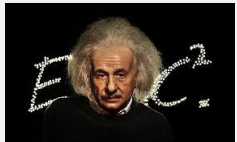
Neste exemplo, a classe RoboVoador é uma subclasse de Robo, o que significa que ela herda todos os métodos e atributos de Robo. Assim, um objeto da classe RoboVoador também pode chamar o método mover(), assim como um objeto da classe Robo, mas também tem seu próprio método exclusivo voar().

Essa abordagem torna o código mais organizado e reutilizável.



Java - herança

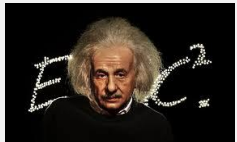
Em resumo, a herança em Java é um mecanismo que permite que uma classe herde características e comportamentos de outra classe, promovendo a reutilização de código e a organização do design do programa.



Java - polimorfismo

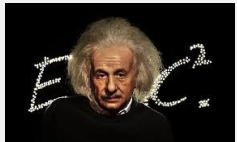
Polimorfismo é um conceito importante na programação orientada a objetos que nos permite tratar objetos de diferentes classes de maneira uniforme.

Imagine que você tem vários tipos de animais em uma fazenda: vacas, galinhas, cavalos e assim por diante. Cada animal faz um som diferente. Por exemplo, uma vaca faz "muu", uma galinha faz "cocoricó" e um cavalo faz "hihihi".



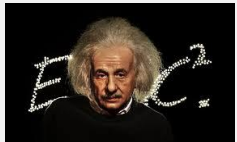
Java - polimorfismo

Agora, se você quisesse escrever um programa para fazer esses animais "falarm", como você faria? Você poderia criar um método falar() em cada classe de animal, mas isso pode ser trabalhoso e complicado de gerenciar. É aqui que o polimorfismo entra em cena.



Java - polimorfismo

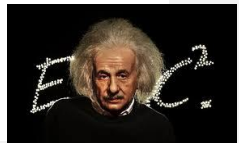
Em Java, o polimorfismo nos permite tratar objetos de diferentes classes da mesma maneira, usando uma referência comum. Por exemplo, podemos ter uma classe base Animal com um método falar(), e então podemos criar subclasses como Vaca, Galinha, Cavalo, cada uma sobrescrevendo o método falar() com seu próprio comportamento.



Java - polimorfismo

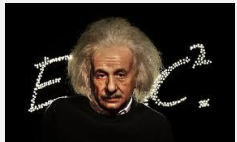
*Animal.java

```java
package explicacao.polimorfismo;

//Classe base
class Animal {
	public void falar() {
		System.out.println("Este animal emite algum som.");
	}
}

//Subclasse Vaca
class Vaca extends Animal {
	@Override
	public void falar() {
		System.out.println("Muu");
	}
}

//Subclasse Galinha
class Galinha extends Animal {
	@Override
	public void falar() {
		System.out.println("Cocoricó");
	}
}

//Subclasse Cavalo
class Cavalo extends Animal {
	@Override
	public void falar() {
		System.out.println("Hihihi");
	}
}
```



Java - polimorfismo

Agora, podemos criar objetos de cada uma dessas subclasses e tratá-los como objetos da classe Animal. Quando chamamos o método falar() em cada objeto, o método apropriado é executado, dependendo do tipo real do objeto.



Java - polimorfismo

Animal.java

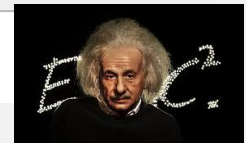
```java
package explicacao.polimorfismo;

//Classe base
class Animal {
    public void falar() {
        System.out.println("Este animal emite algum som.");
    }
}

//Subclasse Vaca
class Vaca extends Animal {
    @Override
    public void falar() {
        System.out.println("Muu");
    }
}

//Subclasse Galinha
class Galinha extends Animal {
    @Override
    public void falar() {
        System.out.println("Cocoricó");
    }
}

//Subclasse Cavalo
class Cavalo extends Animal {
    @Override
    public void falar() {
        System.out.println("Hihihi");
    }
}
```

TestePolimorfismo.java

```java
package explicacao.polimorfismo;

public class TestePolimorfismo {
    public static void main(String[] args) {
        Animal animal1 = new Vaca();
        Animal animal2 = new Galinha();
        Animal animal3 = new Cavalo();

        animal1.falar(); // Saída: Muu
        animal2.falar(); // Saída: Cocoricó
        animal3.falar(); // Saída: Hihihi
    }
}
```

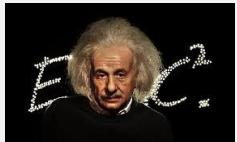
Problems | Javadoc | Declaration | Console

```
<terminated> TestePolimorfismo [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 16:40:44 – 16:40:44)
Muu
Cocoricó
Hihihi
```



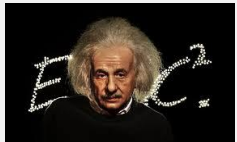
Java - polimorfismo

Essa capacidade de tratar objetos de diferentes classes de maneira uniforme é o polimorfismo em ação. Ele nos permite escrever código mais flexível, reutilizável e fácil de entender, promovendo uma melhor organização e manutenção do código.



Java – classes e objetos

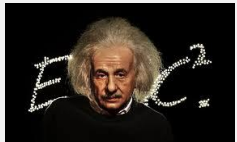
Imagine que você está construindo uma casa. Você pode pensar em uma casa como um modelo ou plano para construir muitas casas semelhantes. O modelo da casa descreve suas características básicas, como o número de quartos, banheiros, a cor das paredes, e assim por diante.



Java – classes e objetos

Em Java, uma classe é como esse modelo ou plano para criar objetos. Uma classe é uma estrutura de programação que define o comportamento e as características de um objeto. Ela define quais informações um objeto pode armazenar (atributos) e quais ações ele pode realizar (métodos).

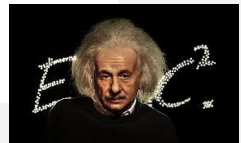
Por exemplo, vamos criar uma classe Carro em Java:



Java – classes e objetos

*Carro.java

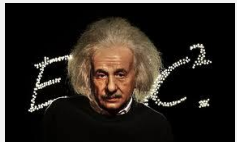
```java
package explicacao.classes_e_objetos;

public class Carro {
    // Atributos
    String marca;
    String modelo;
    int ano;

    // Método
    void ligar() {
        System.out.println("O carro está ligado!");
    }
}
```



Java – classes e objetos

A classe Carro define três atributos: marca, modelo e ano, que representam as características de um carro. Além disso, a classe possui um método chamado ligar(), que representa uma ação que um carro pode realizar.

Agora, para usar essa classe e criar um objeto (ou instância) dela, você pode fazer assim:



Java – classes e objetos

Carro.java

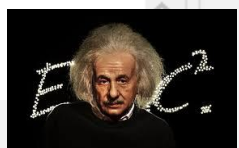
```java
package explicacao.classes_e_objetos;

public class Carro {
    // Atributos
    String marca;
    String modelo;
    int ano;

    // Método
    void ligar() {
        System.out.println("O carro está ligado!");
    }
}
```

TesteCarro.java

```java
package explicacao.classes_e_objetos;

public class TesteCarro {
    public static void main(String[] args) {
        // Criando um objeto da classe Carro
        Carro meuCarro = new Carro();

        // Atribuindo valores aos atributos
        meuCarro.marca = "Toyota";
        meuCarro.modelo = "Corolla";
        meuCarro.ano = 2022;

        // Chamando um método do objeto
        meuCarro.ligar(); // Saída: O carro está ligado!
    }
}
```

Problems | Javadoc | Declaration | Console

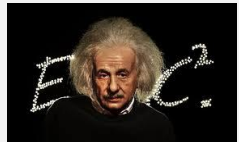
```
<terminated> TesteCarro [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 16:44:45 – 16:44:47)
O carro está ligado!
```



Java – classes e objetos

Neste exemplo, criamos um objeto meuCarro da classe Carro usando a palavra-chave new. Em seguida, atribuímos valores aos atributos marca, modelo e ano do objeto. Por fim, chamamos o método ligar() do objeto.

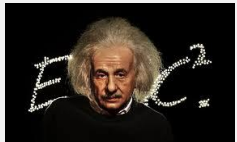
Em resumo, uma classe em Java é como um modelo ou plano para criar objetos. Os objetos são instâncias específicas de uma classe, que possuem seus próprios valores de atributos e podem realizar as ações definidas pelos métodos da classe. Classes e objetos são fundamentais na programação orientada a objetos e ajudam a organizar e estruturar o código de maneira eficiente.



Java – atributos e métodos estáticos

Você está em uma sala de aula e quer contar quantos alunos já passaram por ali.

Você pode criar uma variável para armazenar esse número, mas essa informação é comum a todos os alunos, então faz sentido que seja uma característica estática da sala de aula, e não de cada aluno individualmente.

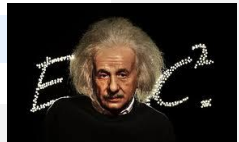


Java – atributos e métodos estáticos

Em Java, um atributo estático é uma variável que pertence à classe como um todo, em vez de pertencer a instâncias individuais (objetos) da classe. Isso significa que todos os objetos da classe compartilham o mesmo valor do atributo estático. Você declara um atributo estático usando a palavra-chave static.

*SalaDeAula.java

```
1  package explicacao.tributos_metodosEstaticos;
2
3  public class SalaDeAula {
4      static int numeroAlunos = 0;
5  }
6
```

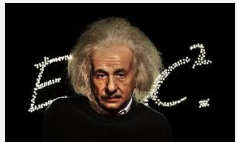


Java – atributos e métodos estáticos

Neste exemplo, **numeroAlunos** é um atributo estático da classe SalaDeAula.

Isso significa que você pode acessá-lo sem criar um objeto da classe **SalaDeAula**.

Por exemplo:



Java – atributos e métodos estáticos

SalaDeAula.java

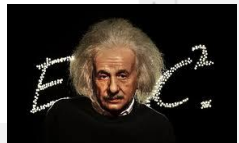
```java
package explicacao.tributos_metodosEstaticos;

public class SalaDeAula {

    static int numeroAlunos = 0;


}

```

*Main_SaladeAula.java

```java
package explicacao.tributos_metodosEstaticos;

public class Main_SaladeAula {

    public static void main(String[] args) {

        SalaDeAula.numeroAlunos++;

        System.out.println("Número de alunos na sala: " + SalaDeAula.numeroAlunos);

    }

}

```

Problems | Javadoc | Declaration | Console

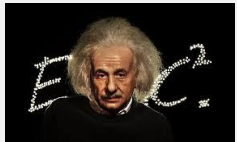
```
<terminated> Main_SaladeAula [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 16:51:11 – 16:51:12)
Número de alunos na sala: 1
```



Java – atributos e métodos estáticos

Métodos Estáticos -> imagine que você quer calcular a média das idades de todos os alunos da escola. Este é um cálculo que não depende de um aluno específico, então faz sentido que seja um método estático.

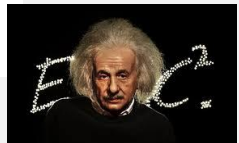
Em Java, um método estático é um método que pertence à classe como um todo, em vez de pertencer a instâncias individuais (objetos) da classe. Isso significa que você pode chamar o método sem criar um objeto da classe. Você declara um método estático usando a palavra-chave static.



Java – atributos e métodos estáticos

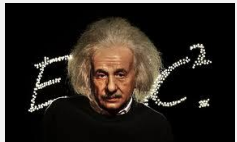
Aluno.java

```java
package explicacao.tributos_metodosEstaticos;

public class Aluno {
    private int idade;

    static int calcularMediaIdades(Aluno[] alunos) {
        int soma = 0;
        for (Aluno aluno : alunos) {
            soma += aluno.idade;
        }
        return soma / alunos.length;
    }
}
```



Java – atributos e métodos estáticos

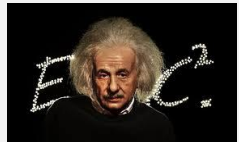
Em resumo, atributos e métodos estáticos em Java pertencem à classe como um todo, em vez de pertencerem a instâncias individuais (objetos) da classe. Isso permite compartilhar dados e funcionalidades entre todos os objetos da classe sem a necessidade de criar uma instância da classe.



Java - Métodos construtores

declaração de um construtor é obrigatória e deve sempre possuir o mesmo nome da classe. Não existe nenhum tipo de retorno, nem mesmo a palavra void pode ser especificada. É invocado no momento da criação do objeto através do operador new e é responsável por criá-lo em memória, ou seja, instanciar a classe que foi definida. O retorno desse operador , é uma referência para objeto recém-criado. É possível criar mais de um construtor dentro da mesma classe.

Em java, apenas as Interfaces não possuem construtores.

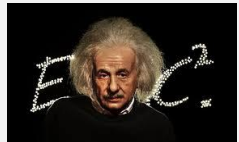


Java - Métodos construtores

você está construindo uma casa e, quando terminar, gostaria de decorá-la com móveis.

Você pode definir alguns móveis básicos que todas as casas devem ter, como uma cama, uma mesa e uma cadeira.

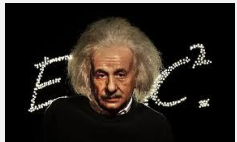
Em vez de colocar esses móveis um por um toda vez que construir uma casa, você pode criar um método que automaticamente coloca esses móveis na casa assim que ela for construída.



Java - Métodos construtores

Em Java, um método construtor é um método especial que é chamado automaticamente quando um objeto de uma classe é criado.

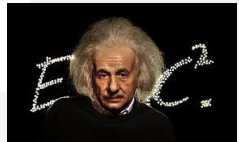
Ele é usado para inicializar os atributos do objeto e realizar outras tarefas de inicialização necessárias.



Java - Métodos construtores

*Casa.java

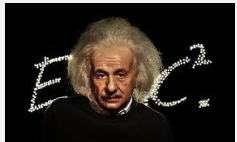
```java
package explicacao.construtores;

public class Casa {
    String cor;
    int numeroComodos;

    // Método construtor
    public Casa(String cor, int numeroComodos) {
        this.cor = cor;
        this.numeroComodos = numeroComodos;
    }
}
```



Java - Métodos construtores

criamos um método construtor para a classe Casa. Este método tem o mesmo nome da classe (Casa) e não possui tipo de retorno. Ele recebe parâmetros (como a cor e o número de cômodos da casa) e os utiliza para inicializar os atributos correspondentes (cor e numeroComodos).

Agora, quando criamos um objeto Casa, podemos passar os valores desejados para o método construtor:



Java - Métodos construtores

Casa.java

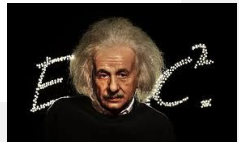
```java
package explicacao.construtores;

public class Casa {
    String cor;
    int numeroComodos;

    // Método construtor
    public Casa(String cor, int numeroComodos) {
        this.cor = cor;
        this.numeroComodos = numeroComodos;
    }
}
```

TesteCasa.java

```java
package explicacao.construtores;

public class TesteCasa {
    public static void main(String[] args) {
        // Criando um objeto Casa usando o método con
        Casa minhaCasa = new Casa("Azul", 3);

        // Acessando os atributos da Casa
        System.out.println("Cor da casa: " + minhaCas
        System.out.println("Número de cômodos da casa
    }
}
```

Problems | Javadoc | Declaration | Console

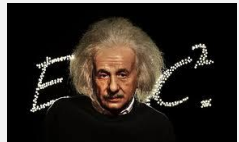
```
<terminated> TesteCasa [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:04:03 – 17:04:04)
Cor da casa: Azul
Número de cômodos da casa: 3
```



Java - Métodos construtores

Quando você cria um novo objeto Casa usando new Casa("Azul", 3), o método construtor é chamado automaticamente e os atributos cor e numeroComodos são inicializados com os valores fornecidos.

Em resumo, métodos construtores em Java são usados para inicializar os atributos de um objeto quando ele é criado. Eles fornecem uma maneira conveniente de configurar o estado inicial de um objeto e são chamados automaticamente quando o objeto é instanciado.

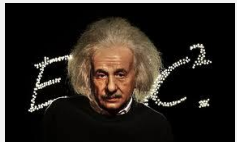


Java - Sobrecarga de métodos

A sobrecarga (overload) consiste num conceito de polimorfismo que é empregado na criação de variações de um mesmo método, ou seja, a criação de dois ou mais métodos com nomes totalmente iguais em uma classe.

O que os difere são seus argumentos.

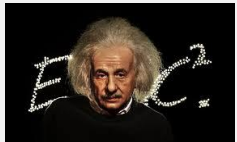
Só dessa forma é feita a separação destes.



Java - Sobrecarga de métodos

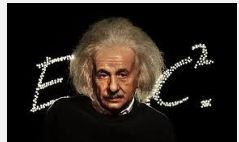
Sua principal característica é possuir mais uma classe.

Nesse sentido, consiste na primeira herdar a segunda, na sobreposição é importante fazer o uso da annotation @Override para indicar que o método foi sobreposto.



Java - Sobrecarga de métodos

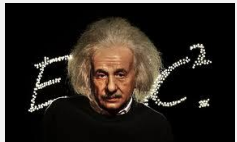
Você tem uma calculadora e quer adicionar dois números. Você pode criar um método chamado adicionar que aceita dois números como argumentos e retorna a soma deles. Mas e se você quiser adicionar três números? Ou quatro? Seria útil ter diferentes versões do método adicionar que aceitem diferentes números de argumentos, certo?



Java - Sobrecarga de métodos

Em Java, a sobrecarga de métodos permite fazer isso. Ela permite definir vários métodos com o mesmo nome em uma classe, desde que tenham diferentes tipos de parâmetros ou números de parâmetros. O compilador Java pode distinguir entre esses métodos com base nos tipos ou números de argumentos passados.

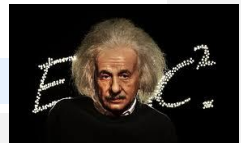
Por exemplo, vamos criar uma classe Calculadora em Java com vários métodos adicionar sobrecarregados:



Java - Sobrecarga de métodos

Calculadora.java

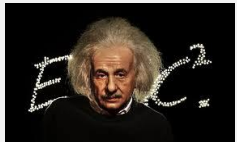
```java
package explicacao.sobrecarga;

public class Calculadora {
	// Método para adicionar dois números inteiros
	public int adicionar(int a, int b) {
		return a + b;
	}

	// Método para adicionar três números inteiros
	public int adicionar(int a, int b, int c) {
		return a + b + c;
	}

	// Método para adicionar dois números decimais
	public double adicionar(double a, double b) {
		return a + b;
	}
}
```



Java - Sobrecarga de métodos

Neste exemplo, temos três versões do método adicionar na classe Calculadora. Uma versão aceita dois argumentos inteiros, outra aceita três argumentos inteiros e a terceira aceita dois argumentos decimais (do tipo double).

Agora podemos usar esses métodos adicionar de acordo com nossas necessidades:



Java - Sobrecarga de métodos

Calculadora.java

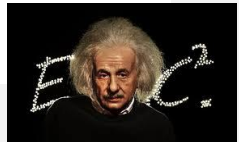
```java
package explicacao.sobrecarga;

public class Calculadora {
    // Método para adicionar dois números inteiros
    public int adicionar(int a, int b) {
        return a + b;
    }

    // Método para adicionar três números inteiros
    public int adicionar(int a, int b, int c) {
        return a + b + c;
    }

    // Método para adicionar dois números decimais
    public double adicionar(double a, double b) {
        return a + b;
    }
}
```

TesteCalculadora.java

```java
package explicacao.sobrecarga;

public class TesteCalculadora {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();

        // Usando o método adicionar para somar dois números inteiros
        System.out.println("Soma de 5 e 7: " + calc.adicionar(5, 7));

        // Usando o método adicionar para somar três números inteiros
        System.out.println("Soma de 5, 7 e 10: " + calc.adicionar(5, 7, 10));

        // Usando o método adicionar para somar dois números decimais
        System.out.println("Soma de 3.5 e 2.5: " + calc.adicionar(3.5, 2.5));
    }
}
```

Problems | Javadoc | Declaration | Console

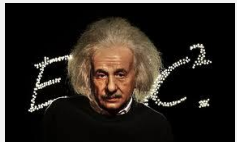
```
<terminated> TesteCalculadora [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:09:34 – 17:09:34)
Soma de 5 e 7: 12
Soma de 5, 7 e 10: 22
Soma de 3.5 e 2.5: 6.0
```



Java - Operador ternário

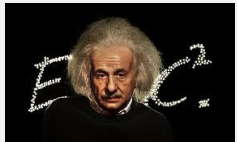
Cada chamada do método adicionar acima chama uma versão diferente do método, dependendo do número e tipo de argumentos passados.

Em resumo, a sobrecarga de métodos em Java permite definir vários métodos com o mesmo nome em uma classe, desde que tenham diferentes tipos ou números de parâmetros. Isso proporciona flexibilidade e conveniência ao trabalhar com métodos que têm comportamentos semelhantes, mas diferentes requisitos de entrada.



Java - Sobreposição de métodos

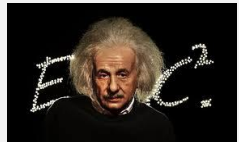
Imagine que você tem uma classe base chamada Animal com um método fazerSom(), que representa o som que um animal faz. Agora, você quer criar subclasses específicas, como Cachorro, Gato e Vaca, e cada uma delas faz um som diferente. Em vez de criar métodos diferentes para cada animal, você pode usar a sobreposição de métodos.



Java - Sobreposição de métodos

A sobreposição de métodos em Java ocorre quando uma classe filha fornece uma implementação específica de um método que já está sendo usado em sua classe pai. Isso significa que a classe filha "substitui" a implementação do método na classe pai com sua própria implementação.

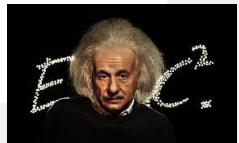
Por exemplo, vamos criar uma classe Animal com um método fazerSom() e subclasses como Cachorro, Gato e Vaca que sobrepõem esse método:



Java - Sobreposição de métodos

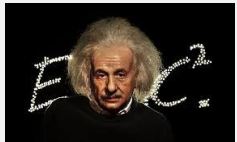
Animal.java

```java
package explicacao.sobreposicao;

//Classe base
class Animal {
	void fazerSom() {
		System.out.println("Este animal emite algum som.");
	}
}

//Subclasse Cachorro
class Cachorro extends Animal {
	@Override
	void fazerSom() {
		System.out.println("Au Au");
	}
}

//Subclasse Gato
class Gato extends Animal {
	@Override
	void fazerSom() {
		System.out.println("Miau");
	}
}

//Subclasse Vaca
class Vaca extends Animal {
	@Override
	void fazerSom() {
		System.out.println("Muu");
	}
}

```



Java - Sobreposição de métodos

Agora, se criarmos objetos dessas subclasses e chamarmos o método fazerSom(), obteremos o som específico de cada animal:



Java - Sobreposição de métodos

Animal.java

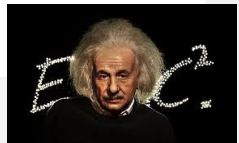
```java
package explicacao.sobreposicao;

//Classe base
class Animal {
    void fazerSom() {
        System.out.println("Este animal emite algum som.");
    }
}

//Subclasse Cachorro
class Cachorro extends Animal {
    @Override
    void fazerSom() {
        System.out.println("Au Au");
    }
}

//Subclasse Gato
class Gato extends Animal {
    @Override
    void fazerSom() {
        System.out.println("Miau");
    }
}

//Subclasse Vaca
class Vaca extends Animal {
    @Override
    void fazerSom() {
        System.out.println("Muu");
    }
}
```

TesteAnimais.java

```java
package explicacao.sobreposicao;

public class TesteAnimais {
    public static void main(String[] args) {
        Animal cachorro = new Cachorro();
        Animal gato = new Gato();
        Animal vaca = new Vaca();

        cachorro.fazerSom(); // Saída: Au Au
        gato.fazerSom();     // Saída: Miau
        vaca.fazerSom();     // Saída: Muu
    }
}
```

Problems | Javadoc | Declaration | Console

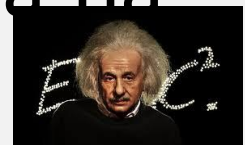
```
<terminated> TesteAnimais [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:15:20 – 17:15:20)
Au Au
Miau
Muu
```



Java - Sobreposição de métodos

Neste exemplo, cada objeto Cachorro, Gato e Vaca substitui a implementação do método fazerSom() da classe Animal com sua própria implementação específica.

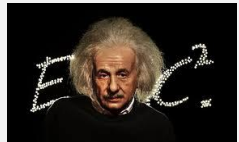
Em resumo, a sobreposição de métodos em Java permite que as subclasses forneçam uma implementação específica de um método definido em sua classe pai, substituindo a implementação original. Isso é útil quando queremos que objetos de diferentes subclasses se comportem de maneiras diferentes, mas mantenham uma interface comum definida na classe pai.



Java - IF e IF-ELSE

A principal estrutura de decisão é o comando if e if-else. Esses comandos também podem ser concatenados sucessivamente por outros ifs, como em if-elseif-else, etc.

você está tomando uma decisão sobre se deve sair de casa hoje ou não. Você pode ter algumas condições que influenciam sua decisão, como o clima. Se estiver ensolarado, você sairá. Se estiver chovendo, você pode decidir ficar em casa. Em Java, você pode expressar essas decisões usando a estrutura if.



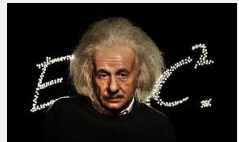
Java - IF e IF-ELSE

IF:

O if é uma estrutura de controle de fluxo em Java que permite executar um bloco de código se uma condição específica for verdadeira.

Se a condição não for verdadeira, o bloco de código não é executado.

Por exemplo, vamos usar o if para expressar a decisão de sair de casa com base no clima:



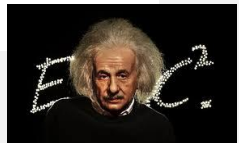
Java - IF e IF-ELSE

If.java

```java
package explicacao.if_else;

public class If {

    public static void main(String[] args) {
        boolean estaEnsolarado = true;

        if (estaEnsolarado) {
            System.out.println("Hoje está ensolarado! Vou sair de casa.");
        }

    }

}
```

Problems @ Javadoc Declaration Console

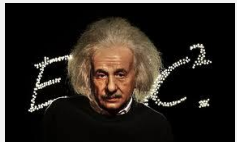
```
<terminated> If [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:20:39 – 17:20:39)
Hoje está ensolarado! Vou sair de casa.
```



Java - IF e IF-ELSE

IF-ELSE:

Às vezes, você pode ter mais de uma opção ou ramificação em sua decisão. Por exemplo, se estiver chovendo, você pode decidir ficar em casa, mas se estiver ensolarado, você vai sair. Nesse caso, você pode usar a estrutura if-else.



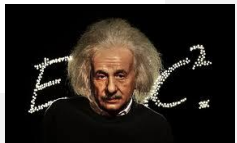
Java - IF e IF-ELSE

If.java

```java
package explicacao.if_else;

public class If {

    public static void main(String[] args) {
        boolean estaChovendo = false;

        if (estaChovendo) {
            System.out.println("Hoje está chovendo. Vou ficar em casa.");
        } else {
            System.out.println("Hoje não está chovendo. Vou sair de casa.");
        }

    }

}
```

Problems | Javadoc | Declaration | Console

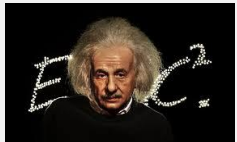
```
<terminated> If [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:21:16 – 17:21:18)
Hoje não está chovendo. Vou sair de casa.
```



Java - IF e IF-ELSE

Neste exemplo, o bloco de código dentro do if é executado se estaChovendo for verdadeiro, caso contrário, o bloco de código dentro do else é executado.

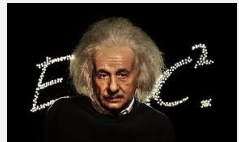
Em resumo, o if é usado para executar um bloco de código se uma condição for verdadeira, enquanto o if-else é usado para executar um bloco de código se uma condição for verdadeira e outro bloco de código se a condição for falsa. Essas estruturas de controle de fluxo permitem que seu programa tome decisões com base em certas condições.



Java - Operador ternário

Considerada uma forma simplificada de construção de comandos do tipo if-else. O nome deve-se ao fato de que o comando é quebrado em três partes separadas pelos símbolos de interrogação (?) e de dois pontos (:).

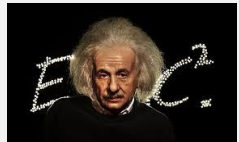
O operador ternário em Java é uma forma concisa de escrever uma estrutura condicional if-else em apenas uma linha. Ele permite que você avalie uma expressão condicional e retorne um valor com base nessa avaliação. Vou explicar de forma didática como funciona.



Java - Operador ternário

Imagine que você tem uma condição simples e deseja atribuir um valor a uma variável dependendo dessa condição. Por exemplo, vamos considerar uma situação em que queremos verificar se uma pessoa é maior de idade e, em seguida, atribuir um valor de acordo com essa verificação.

Sem o operador ternário, você teria que usar uma estrutura if-else, assim:

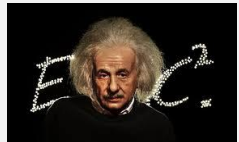


Java - Operador ternário

Exemplo

Comando: if-else	Comando: operador-ternário

`if (ExpLógicaA) {`
`  ComandoA;`
`} else {`
`  ComandoB;`
`}`
	`ExpLógicaA ? ComandoA : ComandoB;`



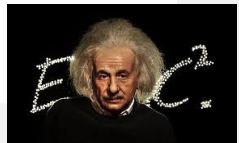
Java - Operador ternário

OperadorTernario.java

```java
package explicacao.operadorTernario;

public class OperadorTernario {

    public static void main(String[] args) {
        int idade = 20;
        String status;

        if (idade >= 18) {
            status = "Maior de idade";
        } else {
            status = "Menor de idade";
        }

        System.out.println("Status: " + status);

    }
}

```

Problems | Javadoc | Declaration | Console

```
<terminated> OperadorTernario [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe  (01/04/2024 17:24:04 – 17:24:04)
Status: Maior de idade
```



Java - Operador ternário

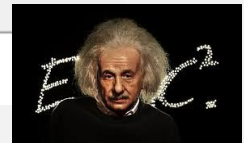
Com o operador ternário, você pode fazer a mesma coisa em uma única linha de código:

OperadorTernario.java

```java
package explicacao.operadorTernario;

public class OperadorTernario {

    public static void main(String[] args) {
        int idade = 20;
        String status = (idade >= 18) ? "Maior de idade" : "Menor de idade";

        System.out.println("Status: " + status);


    }
}
```

Problems | Javadoc | Declaration | Console

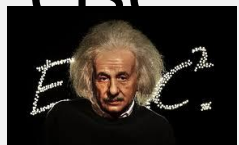
```
<terminated> OperadorTernario [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:24:47 – 17:24:49)
Status: Maior de idade
```



Java - Operador ternário

Neste exemplo, a expressão (idade >= 18) é avaliada primeiro. Se for verdadeira, o valor após o ? é retornado (nesse caso, "Maior de idade"). Se for falsa, o valor após o : é retornado (nesse caso, "Menor de idade").

O operador ternário é especialmente útil quando você tem uma condição simples e deseja atribuir um valor com base nessa condição em uma única linha de código. No entanto, é importante usá-lo com moderação para manter o código legível. Quando a lógica fica muito complexa, é melhor usar uma estrutura if-else convencional.

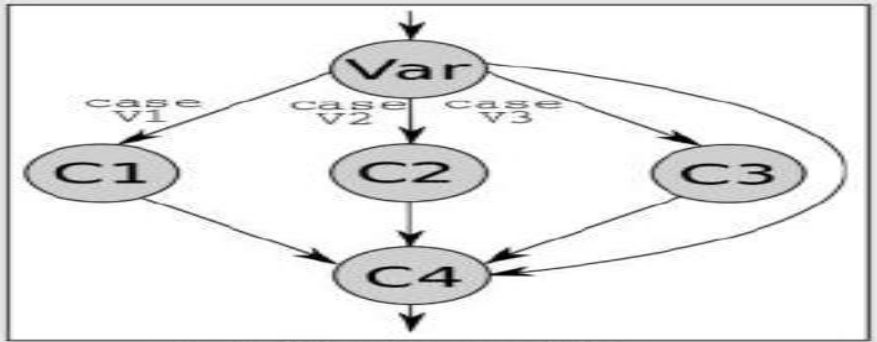


Java - Switch-case | Switch-case-default

Esse comando provê o que chamamos de estrutura de seleção múltipla baseada em alguma variável de controle.

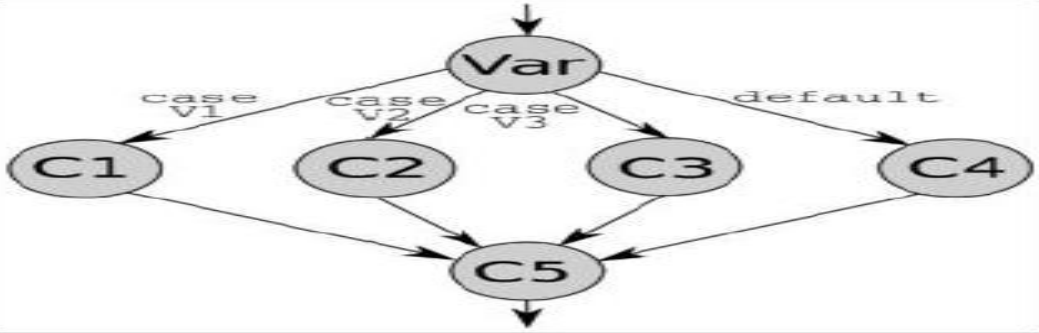
Exemplo

```
switch (Var) {
    case V1:
        C1; break;
    case V2:
        C2; break;
    case V3:
        C3; break;
}
C4
```

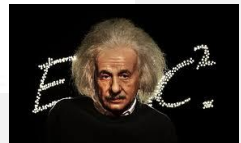


switch-case

```
switch (Var) {
    case V1:
        C1; break;
    case V2:
        C2; break;
    case V3:
        C3; break;
    default:
        C4;
}
C5
```

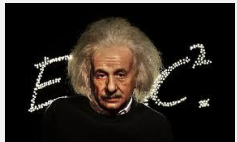


switch-case-default



Java - Switch-case | Switch-case-default

você está em uma sorveteria e quer escolher um sabor de sorvete. Você tem várias opções de sabores e deseja tomar uma decisão com base no sabor escolhido. O switch-case em Java é uma estrutura de controle de fluxo que permite tomar decisões com base no valor de uma variável.

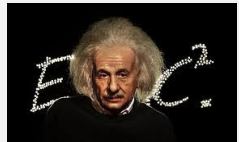


Java - Switch-case | Switch-case-default

Switch-case:

O switch-case é uma estrutura que permite avaliar uma expressão e, dependendo do resultado dessa avaliação, executar um bloco de código específico. Ele oferece uma alternativa mais clara e concisa do que encadear vários if-else quando você tem muitas condições para avaliar.

Por exemplo, imagine que você quer escolher um sabor de sorvete e seu sabor favorito é baunilha:



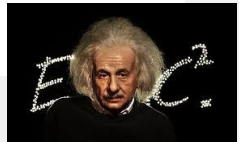
Java - Switch-case | Switch-case-default

SwitchCase.java

```java
package explicacao.switchCase;

public class SwitchCase {

    public static void main(String[] args) {
        String sabor = "baunilha";

        switch (sabor) {
        case "baunilha":
            System.out.println("Sabor escolhido: Baunilha");
            break;
        case "morango":
            System.out.println("Sabor escolhido: Morango");
            break;
        case "chocolate":
            System.out.println("Sabor escolhido: Chocolate");
            break;
        default:
            System.out.println("Sabor não reconhecido");
        }

    }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> SwitchCase [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:29:58 – 17:29:58)
Sabor escolhido: Baunilha
```

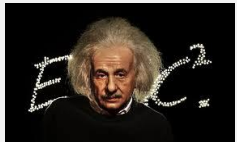


Java - Switch-case | Switch-case-default

Switch-case-default:

O bloco default é opcional e é usado para lidar com situações em que nenhum dos casos corresponde ao valor da expressão. Ele é executado quando nenhum dos outros casos foi acionado.

Por exemplo, se você tentar escolher um sabor de sorvete que não esteja na lista:



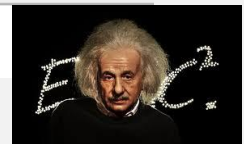
Java - Switch-case | Switch-case-default

SwitchCase.java

```java
package explicacao.switchCase;

public class SwitchCase {

    public static void main(String[] args) {
        String sabor = "";

        switch (sabor) {
        case "baunilha":
            System.out.println("Sabor escolhido: Baunilha");
            break;
        case "morango":
            System.out.println("Sabor escolhido: Morango");
            break;
        case "chocolate":
            System.out.println("Sabor escolhido: Chocolate");
            break;
        default:
            System.out.println("Sabor não reconhecido");
        }

    }
}
```

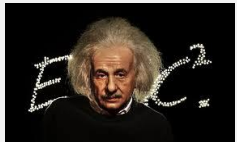
Problems | Javadoc | Declaration | Console

```
<terminated> SwitchCase [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:30:57 – 17:30:58)
Sabor não reconhecido
```



Java - Switch-case | Switch-case-default

Em resumo, o switch-case em Java é uma estrutura de controle de fluxo que permite avaliar uma expressão e tomar decisões com base nos diferentes valores dessa expressão. O bloco default é opcional e é executado quando nenhum dos outros casos corresponde ao valor da expressão.

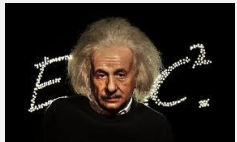


Java - Repetição com while / for / do while

1. Repetição com while:

O while é uma estrutura de repetição que permite executar um bloco de código enquanto uma condição específica for verdadeira. Ele verifica a condição antes de cada execução do bloco de código.

Vamos ver um exemplo onde queremos contar de 1 a 5 usando um while:



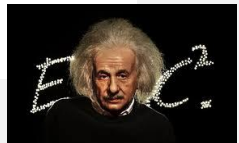
Java - Repetição com while / for / do while

Laco_While.java

```java
package explicacao.while_for_doWhile;

public class Laco_While {

    public static void main(String[] args) {

        int contador = 1;

        while (contador <= 5) {
            System.out.println(contador);
            contador++;
        }

    }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> Laco_While [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:35:00 – 17:35:02)
1
2
3
4
5
```

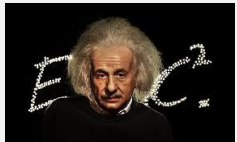


Java - Repetição com while / for / do while

2. Repetição com for:

O for é outra estrutura de repetição em Java. Ele é frequentemente usado quando o número de iterações é conhecido.

Aqui está o mesmo exemplo anterior de contagem de 1 a 5, mas desta vez usando um for:



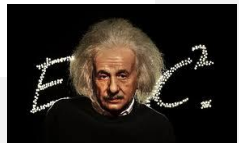
Java - Repetição com while / for / do while

Laco_While.java | Laco_For.java

```
1  package explicacao.while_for_doWhile;
2
3  public class Laco_For {
4
5      public static void main(String[] args) {
6
7          for (int i = 1; i <= 5; i++) {
8              System.out.println(i);
9          }
10
11     }
12
13 }
14
```

Problems | Javadoc | Declaration | Console

```
<terminated> Laco_For [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe  (01/04/2024 17:36:18 – 17:36:19)
1
2
3
4
5
```

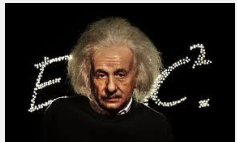


Java - Repetição com while / for / do while

3. Repetição com do-while:

O do-while é semelhante ao while, mas a condição é verificada após a execução do bloco de código, garantindo que o bloco seja executado pelo menos uma vez.

Vamos considerar um exemplo onde pedimos ao usuário para fornecer um número positivo usando do-while:



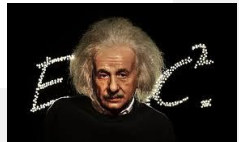
Java - Repetição com while / for / do while

Laco_While.java | Laco_For.java | *Laco_DoWhile.java

```java
package explicacao.while_for_doWhile;

import java.util.Scanner;

public class Laco_DoWhile {

    public static void main(String[] args) {

        int numero;

        do {
            System.out.println("Digite um número positivo: ");
            Scanner ler = new Scanner(System.in);
            numero = ler.nextInt();
        } while (numero <= 0);
        System.out.println("Número positivo fornecido: " + numero);
    }

}
```

Problems | Javadoc | Declaration | Console

```
<terminated> Laco_DoWhile [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:45:23 – 17:45:28)
Digite um número positivo:
1
Número positivo fornecido: 1
```

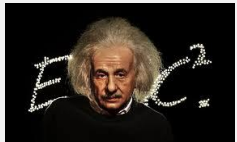


Java – Tratamento de Exceção

O que são exceções?

Em Java, uma exceção é um evento que ocorre durante a execução de um programa e interrompe o fluxo normal de execução.

As exceções podem ser causadas por erros de programação, erros de hardware, entrada de usuário incorreta, entre outras situações imprevistas.

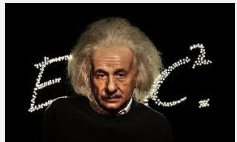


Java – Tratamento de Exceção

Tratamento de exceções em Java:

O tratamento de exceções em Java permite lidar com essas situações imprevistas de forma controlada, evitando que o programa seja encerrado abruptamente.

Existem duas principais partes no tratamento de exceções em Java: try-catch e finally.

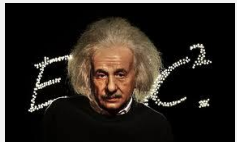


Java – Tratamento de Exceção

1. Bloco try-catch:

O bloco try-catch é usado para envolver o código que pode gerar uma exceção. Dentro do bloco try, você coloca o código que pode lançar uma exceção, e dentro do bloco catch, você trata a exceção caso ela ocorra.

Vamos modificar o exemplo anterior para incluir um bloco try-catch:



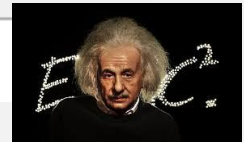
Java – Tratamento de Exceção

*Try_catth.java

```
1  package explicacao.try_catth;
2
3  public class Try_catth {
4
5      public static void main(String[] args) {
6          int dividendo = 10;
7          int divisor = 0;
8
9          try {
10             int resultado = dividendo / divisor;
11             System.out.println(resultado1);
12         } catch (ArithmeticException e) {
13             System.out.println("Ocorreu uma exceção: " + e.getMessage());
14         }
15
16     }
17 }
18
```

Problems | Javadoc | Declaration | Console

```
<terminated> Try_catth [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:50:48 – 17:50:48)
Ocorreu uma exceção: / by zero
```

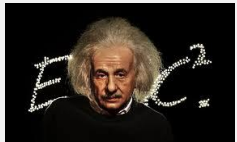


Java – Tratamento de Exceção

2. Bloco finally:

O bloco finally é opcional e é usado para executar código que deve ser executado independentemente de ocorrer uma exceção ou não. É útil para liberar recursos ou realizar limpeza, por exemplo.

Vamos adicionar um bloco finally ao exemplo anterior:



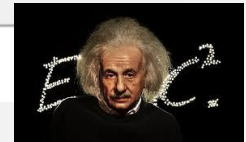
Java – Tratamento de Exceção

Try_catth.java | Try_catch_finally.java

```java
package explicacao.try_catth;

public class Try_catth {

    public static void main(String[] args) {
        int dividendo = 10;
        int divisor = 0;

        try {
            int resultado = dividendo / divisor;
            System.out.println(resultado);
        } catch (ArithmeticException e) {
            System.out.println("Ocorreu uma exceção: " + e.getMessage());
        } finally {
            System.out.println("Execução do bloco finally.");
        }

    }
```

Problems | Javadoc | Declaration | Console

```
<terminated> Try_catth [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:52:10 – 17:52:10)
Ocorreu uma exceção: / by zero
Execução do bloco finally.
```



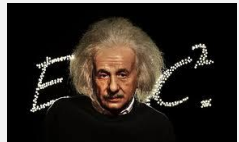
Java – Tratamento de Exceção

Exceções

ArithmeticException: lançada quando uma condição aritmética excepcional ocorre, como, por exemplo, uma divisão por zero de números inteiros.

IndexOutOfBoundsException: lançada para indicar que um índice de algum tipo, como um vetor, uma string ou uma matriz, está fora do intervalo.

ArrayIndexOutOfBoundsException: lançada para indicar que um vetor foi acessado com um índice ilegal, como valor negativo ou maior ao tamanho do vetor.



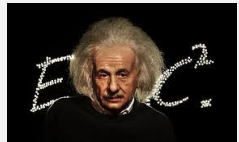
Java – Tratamento de Exceção

Exceções

IllegalArgumentException: lançada para indicar que um método recebeu um argumento ilegal ou inapropriado.

NumberFormatException: lançada para indicar que a aplicação tentou converter um valor em algum tipo numérico, mas o valor não possui o formato apropriado.

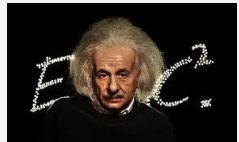
NullPointerException: lançada quando uma aplicação tenta usar um objeto null quando uma instância de objeto é necessária.



Java - For Each

O for-each em Java é uma forma simplificada de iterar sobre elementos de um array ou coleção, sem a necessidade de usar índices ou contadores. É uma estrutura de loop conveniente e legível, especialmente ao lidar com coleções de objetos.

Imagine que você tem um array de números e deseja percorrer cada elemento para realizar alguma operação com eles. Em vez de usar um for tradicional com índices, você pode usar o for-each para percorrer o array de uma maneira mais simples e limpa.



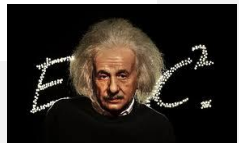
Java - For Each

For_each.java

```java
package explicacao.for_each;

public class For_each {

    public static void main(String[] args) {
        int[] numeros = { 1, 2, 3, 4, 5 };

        for (int numero : numeros) {
            System.out.println(numero);
        }

    }

}
```

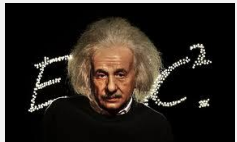
Problems @ Javadoc Declaration Console

```
<terminated> For_each [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 17:57:38 – 17:57:38)
1
2
3
4
5
```



Java - For Each

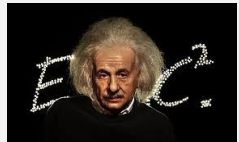
Neste exemplo, o for-each percorre cada elemento do array numeros e armazena temporariamente o valor de cada elemento na variável numero. Em cada iteração do loop, o valor de numero é atualizado para o próximo elemento do array, e o bloco de código dentro do loop é executado.



Java - For Each

O for-each também pode ser usado com coleções, como listas, conjuntos e mapas. Aqui está um exemplo de como usar o for-each com uma lista de strings:

Neste exemplo, o for-each itera sobre cada elemento da lista nomes e armazena temporariamente o valor de cada elemento na variável nome, que é uma string. O bloco de código dentro do loop é executado para cada elemento da lista.



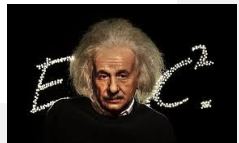
Java - For Each

For_each.java | For_each_Strings.java

```
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class For_each_Strings {
7
8      public static void main(String[] args) {
9
10         List<String> nomes1 = new ArrayList<>();
11         nomes1.add("Alice");
12         nomes1.add("Bob");
13         nomes1.add("Carol");
14
15         for (String nome : nomes1) {
16             System.out.println(nome);
17         }
18
19
```

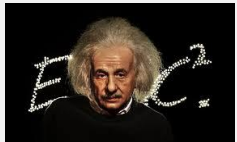
Problems | Javadoc | Declaration | Console

```
<terminated> For_each_Strings [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe  (01/04/2024 17:59:51 – 17:59:53)
Alice
Bob
Carol
```



Java - For Each

O for-each é uma maneira conveniente e eficiente de percorrer arrays e coleções em Java, reduzindo a quantidade de código necessário e tornando o código mais legível. É uma boa prática usar o for-each sempre que possível, especialmente ao lidar com iterações simples sobre coleções de objetos



Java - Interfaces

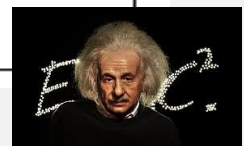
Interfaces

Algumas interfaces implementadas:

Comparable: é utilizada para impor uma ordem nos objetos de uma determinada classe que a implementa.

Runnable: é utilizada para especificar alguma tarefa a ser realizada.

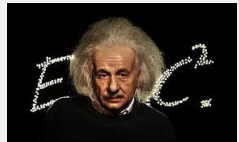
Serializable: é utilizada para identificar classes em que os objetos podem ser gravados (também chamados de serializados) ou lidos (também chamados de desserializados) de algum dispositivo de armazenamento, como HD.



Java - Interfaces

Interfaces em Java são como contratos que especificam um conjunto de métodos que uma classe deve implementar se quiser cumprir o contrato. De forma mais didática, imagine que você tem um manual de instruções para um brinquedo. Esse manual diz que o brinquedo deve ter botões específicos, fazer determinados sons e realizar certos movimentos quando acionado.

Nesse caso, o manual é como uma interface e o brinquedo é a classe que implementa essa interface.

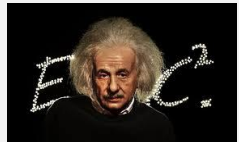


Java - Interfaces

*Animal.java

```
1  package explicacao.interfaces;
2
3  public interface Animal {
4
5      void fazerSom();
6
7      void mover();
8  }
9
```

Animal é a nossa interface. Ela especifica que qualquer classe que a implementar deve ter dois métodos: fazerSom() e mover().

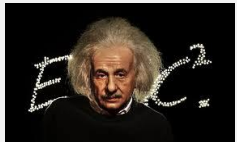


Java - Interfaces

vamos criar uma classe Cachorro que implementa essa interface:

Animal.java | *Cachorro.java

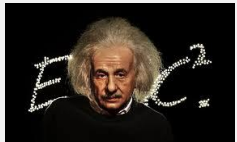
```
1  package explicacao.interfaces;
2
3  public class Cachorro implements Animal {
4
5      @Override
6      public void fazerSom() {
7          System.out.println("Au au!");
8      }
9
10     @Override
11     public void mover() {
12         System.out.println("Correndo...");
13     }
14 }
15
```



Java - Interfaces

A classe Cachorro implementa a interface Animal e fornece implementações para os métodos fazerSom() e mover(). Isso significa que um cachorro, que é um tipo de animal, deve ser capaz de fazer som e se mover.

Agora, podemos criar uma instância de Cachorro e chamá-la:



Java - Interfaces

Animal.java

```java
package explicacao.interfaces;

public interface Animal {

    void fazerSom();

    void mover();
}
```

Cachorro.java

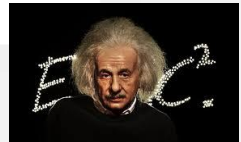
```java
package explicacao.interfaces;

public class Cachorro implements Animal {

    @Override
    public void fazerSom() {
        System.out.println("Au au!");
    }

    @Override
    public void mover() {
        System.out.println("Correndo...");
    }
}
```

TesteInterface.java

```java
package explicacao.interfaces;

public class TesteInterface {

    public static void main(String[] args) {

        Cachorro cachorro = new Cachorro();
        cachorro.fazerSom(); // Saída: Au au!
        cachorro.mover(); // Saída: Correndo...

    }

}
```

Console

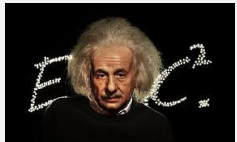
```
<terminated> TesteInterface [Java Application] C:\Program Files\RedHat\java-1.8.0-openjdk-1.8.0.212-3\bi
Au au!
Correndo...
```



Java - Interfaces

Ou seja, interfaces em Java permitem que você defina um conjunto de métodos que as classes devem implementar, o que promove um alto nível de abstração e modularidade em seu código.

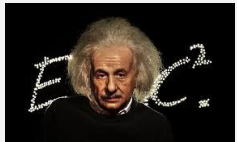
Isso é útil especialmente quando você deseja definir um contrato para diferentes classes, garantindo que elas cumpram certos requisitos.



Java - JAVAFX

A linguagem Java possui um suporte muito bom para a criação de Interfaces Gráficas de Usuário ou, em inglês, Graphical User Interface (GUI), e existem várias formas de se fazer a criação de interfaces gráficas em Java. Ao longo da evolução dessa linguagem, diversas bibliotecas gráficas foram criadas, como:

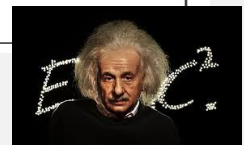
Abstract Window Toolkit (AWT), Swing, Standard Widget Toolkit (SWT), Apache Pivot, SwingX, JGoodies, QtJambi e JavaFX.



Java - JAVAFX

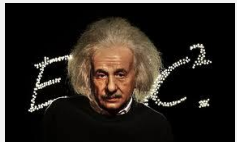
Componentes JAVAFX

Nome do Componente	Classe do Componente	Descrição do Componente	Aspecto Visual do Componente
Botão	Button	Simples botão de controle.	Button
Rótulo	Label	Campo que permite exibir um texto não editável.	Label
Campo de Texto	TextField	Campo que permite a inserção de texto em uma única linha.	TextField
Área de Texto	TextArea	Campo que permite a inserção de texto com múltiplas linhas.	TextArea TextArea
Botão de Opção	RadioButton	Seleção de opções mutuamente excludentes.	RadioButton
Caixa de Seleção	CheckBox	Seleção de opções não mutuamente excludentes.	CheckBox
Caixa de Combinação	ComboBox	Seleção de opções mutuamente excludentes em forma de lista.	Segunda



Java - Vetores ou Arrays Unidimensionais

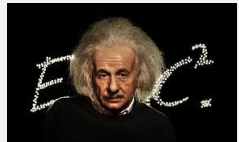
Vetores ou Arrays Unidimensionais -> Imagine que você tem uma prateleira (ou uma "linha") com várias caixas. Cada caixa contém um valor, como um número, uma palavra, um objeto ou qualquer outra coisa. Agora, suponha que você queira armazenar esses valores em seu programa Java. Você poderia criar uma variável para cada valor, mas isso pode ficar muito confuso e difícil de gerenciar, especialmente se você tiver muitos valores.



Java - Vetores ou Arrays Unidimensionais

É aqui que os vetores entram em cena. Um vetor é como uma prateleira virtual onde você pode armazenar vários valores do mesmo tipo. Ao invés de criar uma variável para cada valor, você pode criar um vetor para armazenar todos eles juntos.

Em Java, um vetor é uma estrutura de dados que permite armazenar uma coleção ordenada de elementos do mesmo tipo. Para criar um vetor em Java, você declara o tipo dos elementos que deseja armazenar, seguido por colchetes [] e o nome do vetor.

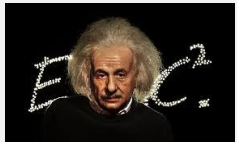


Java - Vetores ou Arrays Unidimensionais

Por exemplo, se você quisesse armazenar uma lista de números inteiros em um vetor, você poderia fazer assim:

```
1  package vetores;
2
3  public class Vetores {
4
5      int[] numeros;
6
7  }
8
```

Isso cria um vetor chamado números que pode armazenar números inteiros.

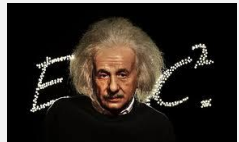


Java - Vetores ou Arrays Unidimensionais

Você também pode inicializar o vetor com valores iniciais ao declará-lo:

```
1  package vetores;
2
3  public class Vetores {
4
5      int[] numeros = {1, 2, 3, 4, 5};
6
7  }
8
```

Agora, numeros é um vetor que contém os números de 1 a 5.



Java - Vetores ou Arrays Unidimensionais

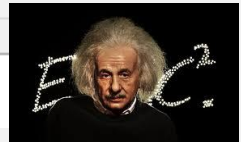
Para acessar um elemento específico em um vetor, você usa um índice. Os índices em Java começam em 0. Por exemplo, para acessar o primeiro elemento do vetor numeros, você faria assim:

Vetores.java

```
1  package vetores;
2
3  public class Vetores {
4
5      public static void main(String[] args) {
6          int[] numeros = {1, 2, 3, 4, 5};
7
8          int primeiroNumero = numeros[0];
9
10         System.out.println(primeiroNumero);
11     }
12
13 }
14
```

Problems @ Javadoc Declaration Console

```
<terminated> Vetores [Java Application] C:\Program Files\Java\jre-1.8\bin\javaw.exe (01/04/2024 16:11:32 – 16:11:33)
1
```

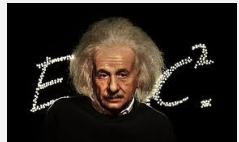


Java - Vetores ou Arrays Unidimensionais

Foi atribuído o valor 1 à variável primeiroNumero, porque 1 é o primeiro elemento do vetor números.

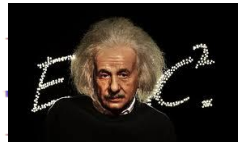
Em resumo, vetores em Java são como prateleiras virtuais onde você pode armazenar uma coleção ordenada de elementos do mesmo tipo.

Eles são úteis quando você precisa lidar com múltiplos valores semelhantes e quer manter seu código organizado e fácil de gerenciar.



Java - Vetores ou Arrays Unidimensionais

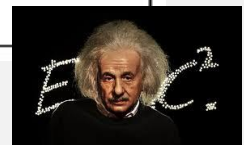
```
Inicializa um array de inteiros com tamanho 5
int[] numero = new int[5];
System.out.println(numero);
-----------------------------------------------------------------------------------------
inicializando Array
int[] numeros = new int[5]; // Inicializa um array de inteiros com tamanho 5
-----------------------------------------------------------------------------------------
Atribuindo valores Array
numeros[0] = 10;
numeros[1] = 20;
numeros[2] = 30;
numeros[3] = 40;
numeros[4] = 50;
-----------------------------------------------------------------------------------------
Iterando sobre um Array
for (int i = 0; i < numeros.length; i++) {
    System.out.println(numeros[i]);
}
-----------------------------------------------------------------------------------------
inicialização de Array com Valores iniciais
int[] primos = { 2, 3, 5, 7, 11 }; // Inicializa um array com valores iniciais
-----------------------------------------------------------------------------------------
Copiando Arrays
int[] copia = Arrays.copyOf(numeros, numeros.length); // Copia o array "numeros" para "copia"
-----------------------------------------------------------------------------------------
ordenando um Array
Arrays.sort(numeros); // Ordena o array "numeros"
-----------------------------------------------------------------------------------------
Pesquisando e ordenando um Array
int indice = Arrays.binarySearch(numeros, 30); // Procura o valor 30 no array ordenado "numeros"
-----------------------------------------------------------------------------------------
Verificando se um Valor Existe em um Array Não Ordenado
boolean existe = Arrays.asList(numeros).contains(30); // Verifica se o valor 30 existe no array
-----------------------------------------------------------------------------------------
```



Java - Vetores ou Arrays Unidimensionais

Criação

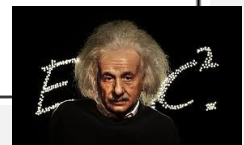
Operação sobre Vetores	Exemplo de Código
Declaração de um vetor.	`tipo nomeVetor[];`
Alocação de espaço de um vetor.	`nomeVetor = new tipo[tamanho];`
Declaração e alocação de um vetor.	`tipo nomeVetor[] =` `new tipo[tamanho];`
Acesso de uma posição do vetor.	`nomeVetor[indice]`
Atribuição de um valor a uma posição.	`nomeVetor[indice] = valor;`
Acesso ao tamanho de um vetor.	`nomeVetor.length`



Java - Vetores ou Arrays Unidimensionais

Exemplo - Vetor

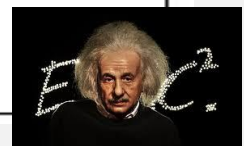
```
1  int vet[] = new int[5];
2  vet[0] = 6;
3  vet[1] = 3;
4  vet[2] = 7;
5  vet[3] = 4;
6  vet[4] = 2;
7  for (int i = 0; i < vet.length; i++) {
8      System.out.println("Array[" + i + "]: " + vet[i]);
9  }
```



Java - Vetores ou Arrays Unidimensionais

Formas de criar e inicializar

Tipo do Vetor	Exemplos usando a forma 1	Exemplos usando a forma 2
Inteiro	`int vet[] = {1, 5, 3, 4};`	`int[] vet = {1, 5, 3, 4};`
Real	`double vet[]={2.0, 5.3, 3.8};`	`double[] vet={2.0, 5.3, 3.8};`
Caractere	`char vet[] = {'a', 'e', 'i'};`	`char[] vet = {'a', 'e', 'i'};`



Java - Vetores ou Arrays Unidimensionais

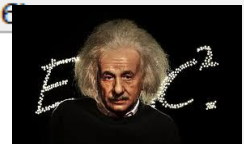
Arrays.java | Arrays.java

```java
package explicacao.arrays.unidimensionais;

public class Arrays {

    public static void main(String[] args) {
        // Declarando e inicializando um array unidimensional de inteiros
        int[] numeros = new int[5];

        // Preenchendo o array com alguns valores
        for (int i = 0; i < numeros.length; i++) {
            numeros[i] = i * 2; // Preenchendo com o dobro do índice
        }

        // Exibindo os valores do array
        System.out.println("Array:");
        for (int i = 0; i < numeros.length; i++) {
            System.out.print(numeros[i] + " ");
        }
    }

```

Problems | Javadoc | Declaration | Console

<terminated> Arrays [Java Application] C:\Program Files\RedHat\java-1.8.0-openjdk-1.8.0.212-3\bin\javaw.exe (01/04/2024 20:43:15 – 20:43:16

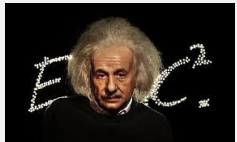
```
Array:
0 2 4 6 8
```



Java - Arrays Multidimensionais

Arrays multidimensionais em Java são arrays que contêm outros arrays como elementos.

Isso permite representar dados em várias dimensões, como matrizes ou tabelas.

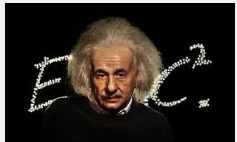


Java - Arrays Multidimensionais

Vamos começar com um exemplo simples de uma matriz bidimensional:

Neste exemplo, matriz é um array bidimensional de inteiros com dimensões 3x3. Isso significa que temos três linhas e três colunas.

Para inicializar e acessar elementos de uma matriz bidimensional, podemos fazer o seguinte:



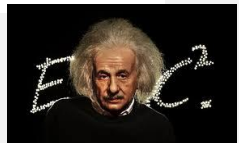
Java - Arrays Multidimensionais

Arrays.java

```java
package explicacao.arrays.multidomensionais;

public class Arrays {

    public static void main(String[] args) {

        int[][] matriz = new int[3][3];

        // Inicializando a matriz
        matriz[0][0] = 1;
        matriz[0][1] = 2;
        matriz[0][2] = 3;
        matriz[1][0] = 4;
        matriz[1][1] = 5;
        matriz[1][2] = 6;
        matriz[2][0] = 7;
        matriz[2][1] = 8;
        matriz[2][2] = 9;

        // Acessando elementos da matriz
        int primeiroElemento = matriz[0][0]; // Retorna 1
        System.out.println(primeiroElemento);

        int segundoElemento = matriz[1][2]; // Retorna 6
        System.out.println(segundoElemento);
    }
}
```

Problems | Javadoc | Declaration | Console

```
<terminated> Arrays (1) [Java Application] C:\Program Files\RedHat\java-1.8.0-openjdk-1.8.0.212-3\bin\javaw.exe  (01/04/2024 20:35:04 – 20:35:04)
1
6
```

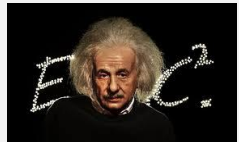


Java - Arrays Multidimensionais

Podemos pensar em uma matriz bidimensional como uma tabela com linhas e colunas, onde cada elemento pode ser acessado especificando o índice da linha e da coluna.

Além de matrizes bidimensionais, é possível ter arrays multidimensionais com mais dimensões. Por exemplo, uma matriz tridimensional seria declarada assim:

```
int[][][] cubo = new int[3][3][3];
```

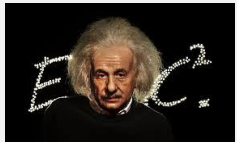


Java - Arrays Multidimensionais

Isso cria um cubo com 3x3x3 elementos.

A ideia é a mesma, podemos acessar cada elemento especificando os índices de cada dimensão.

Em resumo, os arrays multidimensionais em Java são úteis para representar estruturas de dados mais complexas, como matrizes, tabelas ou volumes, e podem ser acessados e manipulados de maneira semelhante aos arrays unidimensionais



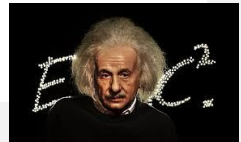
Java - Arrays Multidimensionais

Arrays.java | Arrays.java

```java
package explicacao.arrays.multidomensionais;

public class Arrays {

    /** EXEMPLO DE ARRAY MULTIDOMENSIONAL */
     public static void main(String[] args) {
            // Declarando e inicializando uma matriz bidimensional de inteiros
            int[][] matriz = new int[3][3];

            // Preenchendo a matriz com alguns valores
            for (int i = 0; i < matriz.length; i++) {
                for (int j = 0; j < matriz[i].length; j++) {
                    matriz[i][j] = i + j; // Preenchendo com a soma dos índices da linha e da coluna
                }
            }

            // Exibindo os valores da matriz
            System.out.println("Matriz:");
            for (int i = 0; i < matriz.length; i++) {
                for (int j = 0; j < matriz[i].length; j++) {
                    System.out.print(matriz[i][j] + " ");
                }
                System.out.println(); // Adicionando uma quebra de linha após cada linha da matriz
            }
        }
    }
```

Problems | Javadoc | Declaration | Console

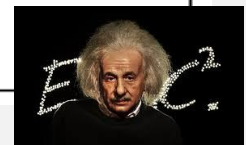
```
<terminated> Arrays (1) [Java Application] C:\Program Files\RedHat\java-1.8.0-openjdk-1.8.0.212-3\bin\javaw.exe  (01/04/2024 20:40:56 – 20:40:56)
Matriz:
0 1 2
1 2 3
2 3 4
```



Java - Exemplo - Matriz

Exemplo - Matriz

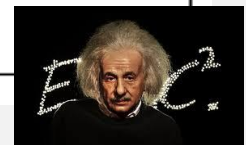
```
1  double mat[][] = {{1.5, 5.2}, {3.6, 4.9}, {2.4, 8.1}};
2  for (int i = 0; i < mat.length; i++) {
3      for (int j = 0; j < mat[i].length; j++) {
4          System.out.println("M[" +i+ "][" +j+ "]: " + mat[i][j]);
5      }
6  }
```



Java - Exemplo - Utilizando arrays

Exemplo - Utilizando arrays

```
 1 package sp1_aula4;
 2
 3 public class SP1_aula4 {
 4
 5     public static void main(String[] args) {
 6
 7
 8         String[] vet = new String[]{"leonardo", "João", "Maria"};
 9         System.out.println("Os nomes informados foram: ");
10         for(int cont = 0; cont < vet.length; cont++)
11         {
12             System.out.println(vet[cont]);
13         }
14     }
15 }
```

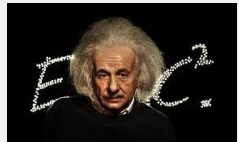


Java - Strings

A classe String possui uma importante característica, que é ser imutável.

Isso quer dizer que uma vez atribuído um valor literal para a variável, existirá uma memória, só que sem nenhum objeto apontando para si.

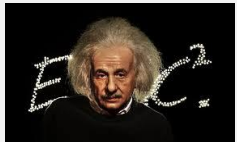
Assim, se você construir uma aplicação que modifique constantemente o valor da variável, você terá um desempenho ruim em termos de gastos de memória e de processamento .



Java – StringBuilder e StringBuffer

StringBuilder - Outra classe que existe para manipular strings e é uma boa alternativa à classe String se desejar que o conteúdo da string seja mutável.

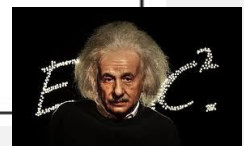
StringBuffer - é uma boa alternativa às outras duas classes se desejar que o conteúdo seja mutável e necessitar de uma aplicação thread safe.



Java – StringBuilder e StringBuffer

Exemplo

```
1  String str1 = new String("Orientação a objetos");
2  String str2 = "Orientação a objetos";
3  StringBuilder str3 = new StringBuilder("Orientação a objetos");
4  StringBuilder str4 = "Orientação a objetos"; //não é aceito (dá
   erro)
5  StringBuffer str5 = new StringBuffer("Orientação a objetos");
6  StringBuffer str6 = "Orientação a objetos"; //não é aceito (dá
   erro)
```

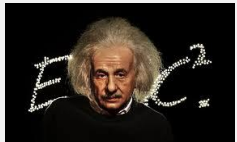


Java - Heap x String Pool

Heap -> O Heap é uma região de memória onde objetos são alocados durante a execução de um programa Java.

É uma área de memória dinâmica que é compartilhada por todas as partes do programa.

Os objetos criados dinamicamente usando a palavra-chave new são alocados no heap.

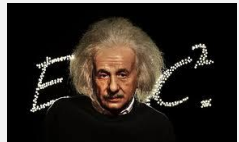


Java - Heap x String Pool

String Pool -> O String Pool é uma área específica da memória do heap que armazena literais de string.

Quando você cria uma string literal em seu código Java (por exemplo, String str = "hello";), ela é armazenada no String Pool.

O String Pool é uma otimização na JVM para economizar espaço de memória, reutilizando strings idênticas em vez de criar novas instâncias.



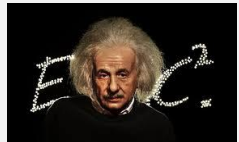
Java - Heap x String Pool

Aqui está uma explicação mais detalhada sobre a diferença entre o Heap e o String Pool:

Quando você cria uma string literal, como "hello", o Java verifica se uma string com esse valor já existe no String Pool.

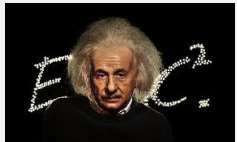
Se a string já existe no String Pool, uma nova referência para essa string é retornada.

Se a string não existe no String Pool, uma nova string é criada e adicionada ao String Pool.



Java - Heap x String Pool

Por outro lado, quando você cria uma string usando a palavra-chave new, como String str = new String("hello");, uma nova instância de string é sempre criada no heap, independentemente de já existir uma string com o mesmo valor no String Pool.

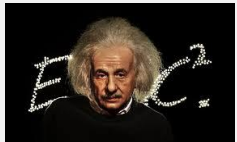


Java - Heap x String Pool

Portanto, mesmo que duas strings tenham o mesmo valor, elas podem estar localizadas em diferentes áreas de memória:

Se ambas as strings forem criadas como literais de string, elas serão armazenadas no String Pool.

Se pelo menos uma delas for criada usando new, ela será armazenada no heap, mesmo que o valor já exista no String Pool.

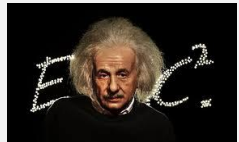


Java - Heap x String Pool

Em suma,

- o String Pool é uma parte do heap que armazena literais de string para otimizar o uso de memória, enquanto

- o heap é a área geral de memória onde todos os objetos são alocados durante a execução do programa Java.



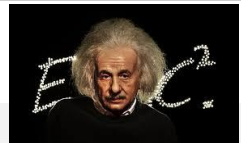
Java - Heap x String Pool

HeapStringPool.java

```java
package explicacao.HeapStringPool;

public class HeapStringPool {
    public static void main(String[] args) {
        // Criando strings literais
        String str1 = "hello";
        String str2 = "hello";

        // Criando strings usando a palavra-chave 'new'
        String str3 = new String("hello");
        String str4 = new String("hello");

        // Verificando se as strings literais estão no mesmo local na memória (String
        // Pool)
        System.out.println("str1 e str2 referenciam o mesmo objeto? " + (str1 == str2)); // Deve imprimir true

        // Verificando se as strings criadas com 'new' estão no mesmo local na memória
        // (heap)
        System.out.println("str3 e str4 referenciam o mesmo objeto? " + (str3 == str4)); // Deve imprimir false

        /**
        Neste exemplo:
                    str1 e str2 são strings literais e, portanto, são armazenadas no String Pool.
                    Como são iguais, elas referenciam o mesmo objeto no String Pool.

                    str3 e str4 são criadas usando a palavra-chave new, portanto,
                    mesmo que tenham o mesmo valor "hello", elas são instâncias separadas de objetos no heap.
                    Portanto, elas não referenciam o mesmo objeto.
         * */
    }

}
```

Problems | Javadoc | Declaration | Console

```
<terminated> HeapStringPool [Java Application] C:\Program Files\RedHat\java-1.8.0-openjdk-1.8.0.212-3\bin\javaw.exe  (01/04/2024 20:56:28 – 20:56:29)
str1 e str2 referenciam o mesmo objeto? true
str3 e str4 referenciam o mesmo objeto? false
```




Obrigado Pessoal!