Curso

Orientação a Objetos em JAVA

Atualizado até o Java 21 &
Eclipse 2023-09

Prof. Msc. Antonio B. C. Sampaio Jr
ENGENHEIRO DE SOFTWARE & PROFESSOR

@abctreinamentos
@amazoncodebr

www.abctreinamentos.com.br
www.amazoncode.com.br

REPOSITÓRIO GITHUB

antonio-sampaio-jr / **orientacao-objetos**

CONTEÚDO PROGRAMÁTICO

- UNIDADE 1 – FUNDAMENTOS DO JAVA
- UNIDADE 2 – RÁPIDA REVISÃO DE ALGORITMOS
- UNIDADE 3 - PROGRAMAÇÃO ORIENTADA A OBJETOS EM JAVA (PARTE 1)
- UNIDADE 4 - PROGRAMAÇÃO ORIENTADA A OBJETOS EM JAVA (PARTE 2)
- UNIDADE 5 – NOVOS RECURSOS - JAVA 8 ao JAVA 21
  - Evolução das Interfaces
  - Novos Métodos String

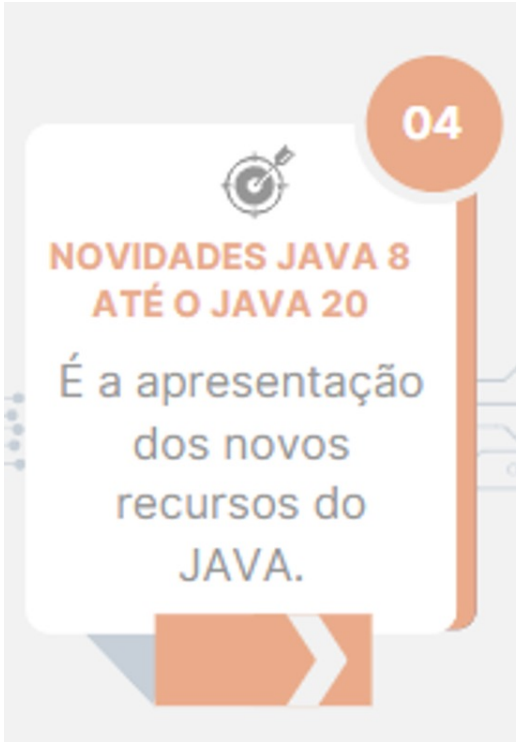
CONTEÚDO PROGRAMÁTICO

- UNIDADE 5 – NOVOS RECURSOS - JAVA 8 ao JAVA 21
  - Records
  - Sealed Classes

UNIDADE 5

NOVOS RECURSOS
JAVA 8 ao JAVA 21

A Nossa Trilha DEV JAVA FULL STACK



Curso

Domine as Inovações do Java com Spring Boot & MongoDB

As Mudanças mais Importantes do Java 8 até o Java 20

Prof. Msc. Antonio B. C. Sampaio Jr
ENGENHEIRO DE SOFTWARE & PROFESSOR

@abctreinamentos
@amazoncodebr

www.abctreinamentos.com.br
www.amazoncode.com.br

Evolução das **INTERFACES**

INTERFACES

- **Definição**
  - Até a versão do Java 7, todos os métodos de uma interface deveriam ser públicos e abstratas, isto é, sem implementação.

```java
// Definindo uma interface chamada "Veiculo"
interface Veiculo {
    void acelerar(int velocidade);
    void frear();
}
```

```java
// Implementando a interface "Veiculo" em uma classe "Carro"
class Carro implements Veiculo {
    private int velocidadeAtual = 0;

    // Implementação do método "acelerar" da interface
    @Override
    public void acelerar(int velocidade) {
        velocidadeAtual += velocidade;
        System.out.println("Carro acelerando para " + velocid
    }

    // Implementação do método "frear" da interface
    @Override
    public void frear() {
        velocidadeAtual = 0;
        System.out.println("Carro freando e parando");
    }
}
```

INTERFACES

- **Definição**
  - Desta maneira, uma interface é como um contrato, que dita quais operações devem estar disponíveis para obter-se um conjunto específico de funcionalidade (e de interoperabilidade, portanto). Isto é particularmente importante, pois a questão aqui é como os objetos podem ser utilizados, e não como foram implementados.

  - Desta maneira, classes pertencentes a diferentes hierarquias de objetos, mas que implementam uma interface comum, podem, polimorficamente, serem tratadas como de um mesmo tipo que corresponde a tal interface.

  - A modelagem de sistemas por meio da definição de interfaces é uma prática reconhecidamente boa, também conhecida como programação por contratos.

MÉTODOS DEFAULT

- **Interfaces com Implementação de Métodos**
  - A partir do Java 8 se tornou possível realizar alterações numa interface existente, mantendo a compatibilidade com suas versões anteriores, isto é, sem propagar as alterações, simplificando muito o trabalho de manutenção do código. Para isto foram introduzidos os métodos default e estáticos às interfaces.

  - Os métodos default são aqueles declarados com o especificador de visibilidade public e com o novo modificador default. Além disso, tais métodos devem ser implementados na própria interface, pois não são abstratos, o que evita a propagação da modificação.

MÉTODOS DEFAULT

- **Métodos Default - Exemplo**

```java
// Atualizando a interface "Veiculo" com um método padrão
interface Veiculo {
    void acelerar(int velocidade);
    void frear();

    // Adicionando um método padrão para obter a velocidade atual
    default int getVelocidadeAtual() {
        return 0; // Valor padrão para a velocidade
    }
}
```

```java
// Implementando a interface "Veiculo" em uma classe "Carro"
class Carro implements Veiculo {
    private int velocidadeAtual = 0;

    // Implementação do método "acelerar" da interface
    @Override
    public void acelerar(int velocidade) {
        velocidadeAtual += velocidade;
        System.out.println("Carro acelerando para " + velocidadeA
    }

    // Implementação do método "frear" da interface
    @Override
    public void frear() {
        velocidadeAtual = 0;
        System.out.println("Carro freando e parando");
    }
}
```

MÉTODOS ESTÁTICOS

- **Definição**
  - De forma análoga, os métodos estáticos são aqueles declarados com o especificador de visibilidade public e com o conhecido modificador static, o que também exige sua própria interface, pois, como qualquer elemento estático, pertencem a seu tipo e não a suas instâncias. Da mesma forma sua adição não propaga qualquer efeito nas classes que já realizam a interface modificada.

```
// Atualizando a interface "Veiculo" com um método estático
interface Veiculo {
    void acelerar(int velocidade);
    void frear();

    // Adicionando um método padrão para obter a velocidade atual
    default int getVelocidadeAtual() {
        return 0; // Valor padrão para a velocidade
    }

    // Adicionando um método estático para verificar se um veículo
    static boolean estaEmMovimento(int velocidade) {
        return velocidade > 0;
    }
}
```

MÉTODOS DEFAULT E ESTÁTICOS

- **Definição**
  - As duas alternativas, dos métodos default e dos métodos estáticos, possibilitam a evolução de interfaces existentes, sem propagação de alterações e com garantia da compatibilidade binária com suas versões antigas.

REFERÊNCIA DE MÉTODOS

- **Definição**
  - A Referência de Métodos pode ser utilizada para fazer uso de uma função já existente, no lugar de uma Expressão Lambda.

  - No Java 8 é possível a passagem por referência de métodos via operador `::` .Sendo assim, uma Expressão Lambda pode ser escrita da seguinte forma:

```
Function<String,Integer> f = s -> s.length();
Function<String,Integer> f = String::length;
```

  - Fica implícito que deverá ser chamado o método length do objeto String passado como parâmetro.

REFERÊNCIA DE MÉTODOS

- **Resumo**

Method Reference Type	Syntax	Example
Static	ClassName::StaticMethodName	String::valueOf
Constructor	ClassName::new	String::new
Specific object instance	objectReference::MethodName	System.out::println
Arbitrary object of a particular type	ClassName::InstanceMethodName	String::toUpperCase

MÉTODOS PRIVADOS EM INTERFACES

- **Definição**
  - A adição da possibilidade de declarar métodos privados em interfaces, embora estranha à primeira vista, é um complemento para auxiliar o programador no uso dos métodos default e métodos estáticos nas interfaces.

  - Os novos métodos privados são declarados com o especificador de visibilidade private e, opcionalmente, com o modificar static. Como qualquer membro privado, só podem ser acessados pelos demais membros do seu tipo.

  - Sendo assim, o uso de métodos privados, estáticos ou não, numa interface tem como objetivo permitir que o programador defina operações auxiliares para os demais métodos default e estáticos públicos, sem expor tal operação.

 http://tecnopode.blogspot.com/2017/11/java-9-metodos-privados-em-interfaces.html

MÉTODOS PRIVADOS EM INTERFACES

- **Benefícios**
  - Os métodos privados servem principalmente como métodos auxiliares para serem usados apenas por outros métodos na mesma interface.

  - Os métodos privados em interfaces têm as seguintes características:
    - **Acesso Restrito:** Eles só podem ser chamados por outros métodos na mesma interface. Não podem ser chamados por classes que implementam a interface.
    - **Encapsulamento:** São úteis para encapsular lógica repetitiva ou complexa em um método privado, tornando o código mais legível e manutenível.
    - **Reutilização de Código:** Permitem a reutilização de código dentro da interface, evitando duplicação de lógica.
    - **Escondem Detalhes de Implementação:** Eles ocultam detalhes de implementação interna da interface.

MÉTODOS PRIVADOS EM INTERFACES

- **Exemplo**

```
interface Animal {
    void comer();

    default void dormir() {
        System.out.println("O animal está dormindo.");
    }

    default void fazerBarulho() {
        System.out.println("O animal faz um som.");
        // Chamando um método privado para adicionar detalhes ao som.
        fazerSomDetalhado();
    }

    private void fazerSomDetalhado() {
        System.out.println("Som detalhado do animal.");
    }
}
```

Novos Métodos **STRING**

NOVOS MÉTODOS STRING

- **Definição**
  - No Java 11, foram adicionados 04 novos métodos à classe String:

  - ***isBlank()*** verifica se a string está vazia ou consiste apenas em caracteres de espaço em branco. Ele retorna *true* se a string estiver vazia ou contiver apenas espaços em branco; caso contrário, retorna *false*. Isso é útil para verificar se uma string é composta apenas por espaços em branco ou não.

```
String str1 = ""; // String vazia
String str2 = "   "; // String com espaços em branco

System.out.println(str1.isBlank()); // true
System.out.println(str2.isBlank()); // true
```

NOVOS MÉTODOS STRING

- **Definição**

  - ***strip()*** remove espaços em branco do início e do fim de uma string. É semelhante ao método *trim()*, mas trata espaços em branco não apenas no padrão ASCII, mas também em outros conjuntos de caracteres Unicode. Isso é útil para normalização de entrada de texto.

```
String str = "   Olá, mundo!   ";
String resultado = str.strip();

System.out.println(resultado); // "Olá, mundo!"
```

NOVOS MÉTODOS STRING

- **Definição**

  - ***lines()*** retorna um fluxo de linhas da string. Cada linha é considerada como uma sequência de caracteres terminada por uma quebra de linha (ou final da string). Isso é útil para dividir uma string em suas linhas componentes.

```
String texto = "Primeira linha\nSegunda linha\nTerceira linha";
texto.lines().forEach(System.out::println);
```

```
Primeira linha
Segunda linha
Terceira linha
```

NOVOS MÉTODOS STRING

- **Definição**

  - ***repeat()*** permite criar uma nova string que é uma repetição da string original, quantas vezes especificado pelo argumento count.

```
String str = "Java ";
String repeticao = str.repeat(3);

System.out.println(repeticao); // "Java Java Java "
```

NOVOS MÉTODOS STRING

- **Definição**
  - No Java 12, foram adicionados 02 novos métodos à classe String:

  - ***indent()*** ajusta a indentação de cada linha da String de acordo com o argumento passado.

```
String multilineStr = "This is\na multiline\nstring.";
String outputStr = "   This is\n   a multiline\n   string.\n";

String postIndent = multilineStr.indent(3);
System.out.println(postIndent);
System.out.println(outputStr);
```

```
This is
a multiline
string.

This is
a multiline
string.
```

NOVOS MÉTODOS STRING

- **Definição**

  - ***transform()*** é usado para modificar uma string com base em uma função lambda e retornar o resultado dessa transformação.

```
String result = "hello".transform(input -> input + " world!");
```

```
hello world!
```

RECORDS

RECORDS

- **Definição**
  - A partir do Java 14, já é possível criar classes imutáveis em Java de forma concisa.

  - Os Registros (*Records*) simplificam a criação de classes que representam dados e incluem automaticamente métodos como '*equals()*', '*hashcode()*', '*toString()*', e métodos de acesso.

```java
public record Pessoa(String nome, int idade) {}

// Criando uma instância do registro
Pessoa pessoa = new Pessoa("Alice", 30);

// Acessando os campos do registro
String nome = pessoa.nome();
int idade = pessoa.idade();

// Usando o método gerado automaticamente toString()
System.out.println(pessoa); // Saída: Pessoa[nome=Alice, idade=30]
```

RECORDS

- **Classe Pessoa**
  - Muito “verboso”.

```java
import java.util.Objects;

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

    // Construtor
    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }

    // Métodos de acesso para o campo nome
    public String getNome() {
        return nome;
    }
```

```java
    public void setNome(String nome) {
        this.nome = nome;
    }

    // Métodos de acesso para o campo idade
    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }
```

RECORDS

- **Classe Pessoa**
  - Muito “verboso”

```java
// Método equals para comparar objetos Pessoa
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    Pessoa pessoa = (Pessoa) o;
    return idade == pessoa.idade &&
            Objects.equals(nome, pessoa.nome);
}

// Método hashCode para gerar um código hash para objetos Pessoa
@Override
public int hashCode() {
    return Objects.hash(nome, idade);
}
```

```java
// Método toString para retornar uma representação de string da Pessoa
@Override
public String toString() {
    return "Pessoa{" +
            "nome='" + nome + '\'' +
            ", idade=" + idade +
            '}';
}
```

CLASSES vs RECORDS

- **Tabela Comparativa**

Aspecto	Classes	Records
Definição	Define um novo tipo de objeto com campos e métodos personalizados.	Define uma nova forma de dados imutáveis, principalmente para armazenamento de dados.
Mutabilidade	Pode ser mutável ou imutável, dependendo da implementação.	Sempre imutável por padrão.
Herança	Pode estender outras classes e implementar interfaces.	Não pode estender outras classes, mas pode implementar interfaces.
Método Adicional	Pode ter qualquer número de métodos com comportamento personalizado.	Pode ter métodos adicionais, mas geralmente usados para funcionalidades básicas relacionadas aos dados.
Acesso a Membros	Pode ter controle granular sobre acesso aos seus membros (encapsulamento).	Sempre transparente - componentes (campos) são sempre acessíveis diretamente.
Controle de Comportamento Adicional	Pode ter comportamentos personalizados adicionais em seus métodos.	Comportamentos adicionais são normalmente associados à manipulação de dados.
Equals, HashCode e ToString	Precisa ser implementado manualmente para garantir a lógica de igualdade correta.	Gerado automaticamente a partir dos componentes (campos).

SEALED CLASSES

SEALED CLASSES

- **Definição**
  - As *sealed classes* (classes “seladas”) são uma nova característica introduzida com preview a partir do Java 15 e que permite controlar quais classes podem ser subclasse de uma classe específica, tornando o mecanismo de herança mais controlado e explícito.

```
public sealed class Pessoa permits PessoaFisica, PessoaJuridica{

    private String nome;
    private int idade;

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

SEALED CLASSES

- As subclasses devem ser ‘final’

```java
public final class PessoaFisica extends Pessoa {

    private String cpf;

    public PessoaFisica(String nome, int idade) {
        super(nome, idade);
        // TODO Auto-generated constructor stub
    }
```

```java
public final class PessoaJuridica extends Pessoa {

    private String cnpj;

    public PessoaJuridica(String nome, int idade) {
        super(nome, idade);
        // TODO Auto-generated constructor stub
    }
```

SEALED CLASSES

- **As subclasses devem ser ‘final’**
  - No exemplo apresentado, Pessoa é uma classe selada e tem duas subclasses finais: PessoaFisica e PessoaJuridica. Isso significa que não é permitido criar novas subclasses delas, visto que o código não será compilado. Isso garante que PessoaFisica e PessoaJuridica são as últimas classes na hierarquia de herança desta linha específica.

SEALED CLASSES

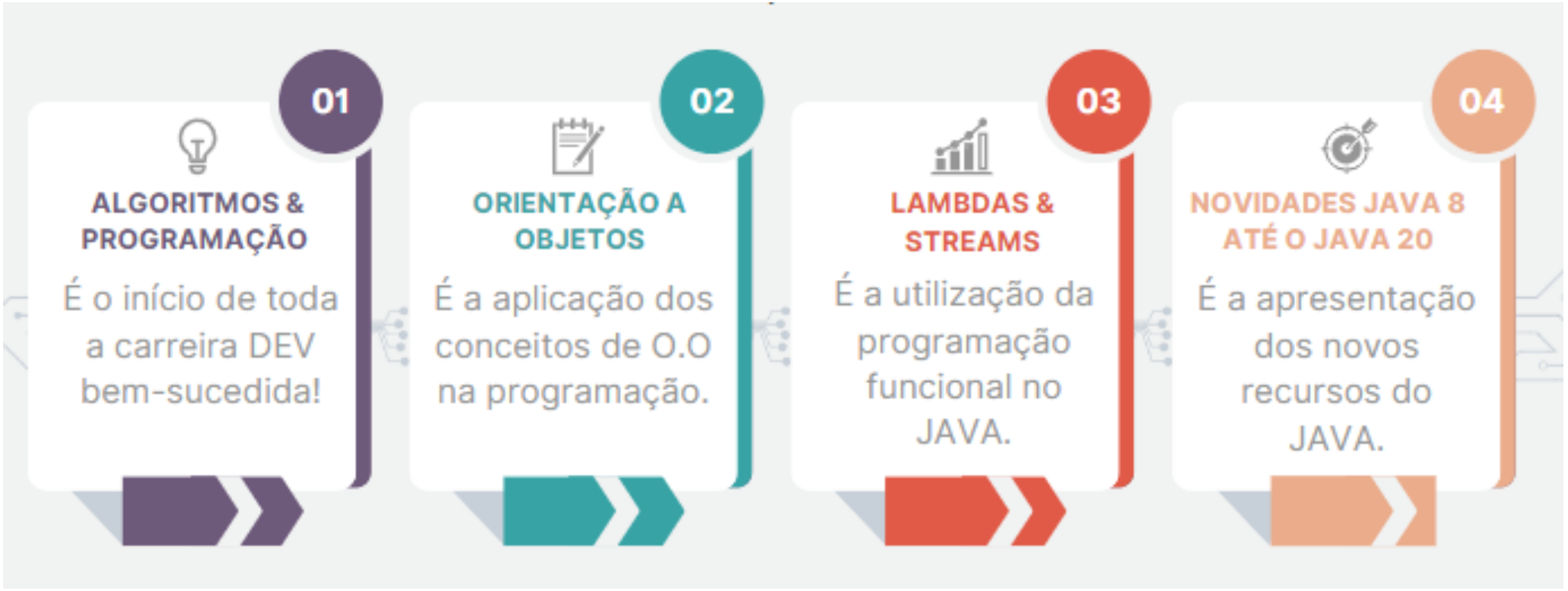
- **Considerações**
  - A principal vantagem das classes seladas é que elas fornecem um controle mais rígido sobre a hierarquia de herança. Isso pode ser útil para evitar que terceiros estendam arbitrariamente classes de códigos de terceiros, o que pode levar a problemas de compatibilidade e complexidade.

  - Além disso, as classes seladas podem ser usadas em conjunto com o recurso de "*pattern matching*" introduzido no Java 16 para simplificar o código que lida com tipos específicos em uma hierarquia de classes seladas.

  - No entanto, é importante notar que as classes seladas não impedem que você crie objetos das classes seladas diretamente. Elas apenas restringem quais classes podem ser subclasse direta da classe selada.

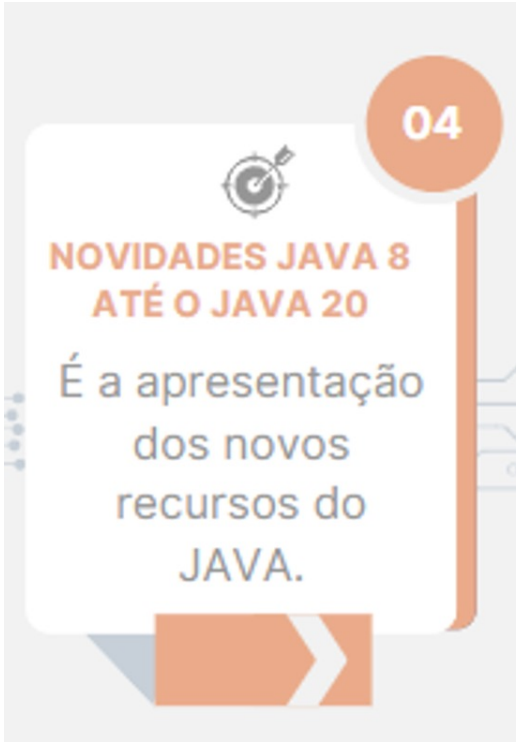
Conclusão

A Nossa Trilha DEV JAVA FULL STACK



A Nossa Trilha DEV JAVA FULL STACK

A Nossa Trilha DEV JAVA FULL STACK



Curso

Domine as Inovações do Java com Spring Boot & MongoDB

As Mudanças mais Importantes do Java 8 até o Java 20

Prof. Msc. Antonio B. C. Sampaio Jr
ENGENHEIRO DE SOFTWARE & PROFESSOR

@abctreinamentos
@amazoncodebr

www.abctreinamentos.com.br
www.amazoncode.com.br

A Nossa Trilha DEV JAVA FULL STACK

Curso

Aplicações JAVA com SPRING BOOT

Prof. Msc. Antonio B. C. Sampaio Jr
ENGENHEIRO DE SOFTWARE & PROFESSOR

@abctreinamentos
@amazoncodebr

www.abctreinamentos.com.br
www.amazoncode.com.br

A Nossa Trilha DEV JAVA FULL STACK

Curso

Aprenda a Resolver Questões Java para Seleção no Exterior

Resolução de 18 Questões de Java

Prof. Msc. Antonio B. C. Sampaio Jr
ENGENHEIRO DE SOFTWARE & PROFESSOR

@abctreinamentos
@amazoncodebr

www.abctreinamentos.com.br
www.amazoncode.com.br

PROF. SAMPAIO

MUITO OBRIGADO!

www.abctreinamentos.com.br

CEL/ZAP: (91) 98216-1883