1.7.1. Estruturas de controle - Javascript

If-else em Javascript

```
const numberOfSeats = 30;
let numberOfGuests = 25;

if (numberOfGuests == numberOfSeats) {
    // all seats are taken
} else if (numberOfGuests < numberOfSeats) {
    // allow more guests
} else {
    // do not allow any more guests
}
```

- A estrutura de Javascript é idêntica a de Java, com exceção das declarações de tipo
- Porque Javascript tem loucuras com relação à conversão implícita de tipos, `5 == "5"` retorna verdadeiro. Para você comparar se os elementos comparados são do mesmo tipo, você precisa usar `` `5 = "5"` ==, o que retornaria falso.

Switch case em Javascript

```
switch (firstUser.accountLevel) {
    case 'normal':
        console.log('You have a normal account!');
        break;
    case 'premium':
        console.log('You have a premium account!');
        break;
    case 'mega-premium':
        console.log('You have a mega premium account!');
        break;
```

```
    default:
        console.log('Unknown account type!');
}
```

- Switch-case permite que haja menos repetição de código do que if-else, em casos que a comparação é sempre de um único objeto de um mesmo tipo.

1.7.2. Estruturas de controle - Java

If-else em Java

```
public class Main{
    public static void main(String[] args){
        final int numberOfSeats = 30;
        int numberOfGuests = 25;

        if (numberOfGuests == numberOfSeats) {
            // all seats are taken
        } else if (numberOfGuests < numberOfSeats) {
            // allow more guests
        } else {
            // do not allow any more guests
        }
    }
}
```

Switch case em Java

```
public class Main{
    public static void main(String[] args){
        String firstUser = "first";

        switch (firstUser) {
            case "first":
```

```
                System.out.println("You have a normal
account!");
                break;
            case "second":
                System.out.println("You have a premium
account!");
                break;
            case "third":
                System.out.println("You have a mega
premium account!");
                break;
            default:
                System.out.println("Unknown account
type!");
        }
    }
}
```

1.7.3. Estruturas de controle - Python

If-else em Python

```
number_seats = 30;
number_guests = 25;

if number_seats == number_guests:
    pass
    # all seats are taken
elif number_guests < number_seats:
    pass
    # allow more guests
else:
    pass
    # do not allow any more guests
```

Switch case em Python

```
firstUser = "first"
match firstUser:
    case 'first':
        print('You have a normal account!')
    case 'second':
        print('You have a premium account!')
    case 'third':
        print('You have a mega premium account!')
    case _: # igual default em JS e java
        print('Unknown account type!')
```

- Importante: o match-case só surgiu no Python 3.10 – o que significa que muitos sistemas e programadores ainda não fazem uso dele como padrão, por estarem rodando versões mais antigas do Python. Eu, por exemplo, sempre evito utilizá-lo para prevenir problemas de compatibilidade.

1.7.4. Comparando de operadores lógicos e de comparação

Operadores lógicos

Operator	Python	Java	JavaScript
AND	`a and b`	`a && b`	`a && b`
OR	`a or b`	`a || b`	`a || b`
NOT	`not a`	`!a`	`!a`

Operadores de comparação

- É a mesma sintaxe para Python, Java e Javascript

Operator	Syntax

Operator	Syntax
Equal	`a == b`
Not Equal	`a != b`
Greater	`a > b`
Less	`a < b`
Greater Equal	`a >= b`
Less Equal	`a <= b`

1.8.1. Laços de repetição - Javascript

Laços definidos - For

```
const numberOfPassengers = 10;

for (let i = 0; i < numberOfPassengers; i++) {
    console.log('Passenger boarded!');
}
```

Laços definidos - For-in

```
const passengers = [
    'Will Alexander',
    'Sarah Kate',
    'Audrey Simon',
    'Tau Perkington'
]

for (let i in passengers) { //  retorna o índice
    console.log('Boarding passenger ' + passengers[i]);
}
```

- Retorna o índice do elemento

Laços definidos - For-of

```
const passengers = [
    'Will Alexander',
    'Sarah Kate',
    'Audrey Simon',
    'Tau Perkington'
]

for (let passenger of passengers) { // retorna o elemento
em si
    console.log('Boarding passenger ' + passenger);
}
```

- Retorna o elemento em si

Laços indefinidos - while

```
let i = 0

while (i<10){
    i++;
    console.log(i);
}
```

- Em geral, laços em JS são extremamente similares à Java. O `for` e o `while` possui a mesma estrutura, of `for-of` de JS tem a mesma funcionalidade do `for-each` em Java

1.8.2. Laços de repetição - Java

Laços definidos - for

```
public class Main{
    public static void main(String[] args){
```

```
        for (int i = 1; i<=10; i++){
            System.out.println(i);
        }
    }
}
```

Laços definidos - for-each

```
import java.util.ArrayList;

public class Main{
    public static void main(String[] args){
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(9);

        for (int number:numbers){
            System.out.println(number);
        }
    }
}
```

- Retorna o elemento em si

Laços indefinidos - while

```
public class Main{
    public static void main(String[] args){
        int i = 0;
        while (i<10){
            i++;
            System.out.println(i);
        }
    }
}
```

1.8.3. Laços de repetição - Python

Laços definidos - For

```
for i in range(1,11):
    print(i)
```

Pegando o índice e valor ao mesmo tempo

```
numeros = [1,5,3,7]

for i, numero in enumerate(numeros):
    print(i, numero)
```

- Diferentemente da maioria das linguagens, o `for` do Python tem o comportamento de retornar os valores em si → o que acontece com o for-each ou for-of de outras linguagens

Laços indefinidos - while

```
i = 0
while i<10:
    i+=1
    print(i)
```

1.9.1. Funções - Javascript

```
function calculateAverage(numbers){
    let sum = 0;
    let length = 0;
```

```
    for(let number of numbers){
        sum += number;
        length++;
    }

    let avg = sum/length;
    return avg;
}

console.log(calculateAverage([1,2,3,4]));
```

- A lógica de funções em JS é a mesma de Java e Python. Lembre-se de evitar a utilização de variáveis globais dentro do contexto da função.

Arrow functions

- A versão ES6 em JS introduziu o conceito de arrow functions, uma sintaxe mais breve para definir funções. Ela funciona similarmente às funções lambda em Python e Java.

```
// Traditional function expression
const add = function(a, b) {
    return a + b;
};

console.log(add(2, 3)); // Output: 5

// Arrow function equivalent
const addArrow = (a, b) => a + b;

console.log(addArrow(2, 3)); // Output: 5
```

1.9.2. Funções - Java

- Não existem "funções" em Java. Uma vez que todo o código deve ser contido dentro de classes, Java tecnicamente apenas possui métodos.

```
public class Person{
    private String name;
    private int age;

    public Person(String personName, int personAge){
        this.name = personName;
        this.age = personAge;
    }

    public String getName(){
        return this.name;
    }

    public int getAge(){
        return this.age;
    }

    public static void main(String[] args){
        Person person1 = new Person("Alex", 40);
        System.out.println(person1.getName());
        System.out.println(person1.getAge());
    }
}
```

- Funções possuem a seguinte estrutura:
  - Visibilidade
  - Estático (pertencente à classe) ou não (pertencente ao objeto/instância)
  - O tipo do retorno
  - Nome da função

- Tipo e nome dos argumentos

Tipos de visibilidade

Visibilidade	Definição
`public`	A variável é acessível de qualquer lugar, dentro ou fora da classe.
`protected`	A variável é acessível dentro da mesma classe, de suas subclasses e dentro do mesmo pacote.
`default`	(Sem modificador) A variável é acessível somente dentro do mesmo pacote.
`private`	A variável é acessível apenas dentro da mesma classe.

- 99% das vezes que você não estiver lidando com herança, você coloca a visibilidade como `private`

Tipo de retorno

Tipo de Retorno	Descrição
`void`	Indica que a função não retorna nenhum valor.
`int`	Retorna um valor inteiro.
`double`	Retorna um valor decimal de ponto flutuante.
`boolean`	Retorna um valor verdadeiro (`true`) ou falso (`false`).
`String`	Retorna uma sequência de caracteres.
`char`	Retorna um caractere.
`Object`	Retorna um objeto genérico.
`Array`	Retorna um array de elementos.
`Classe`	Retorna uma instância de uma classe definida.
`Interface`	Retorna uma implementação de uma interface.
`Enum`	Retorna um valor de um conjunto enumerado.

Clean code

Segundo preceitos de clean code, a maioria das funções não deveria ter argumentos. Funções com um argumento são perfeitamente aceitáveis, dois ou três argumentos são aceitáveis, mas deveriam ser evitadas. Funções com mais de três argumentos devem ser evitadas a qualquer custo.

É importante destacar que Uncle Bob escreveu Clean Code usando Java como linguagem base → uma linguagem que é exclusivamente orientada a objetos. Em linguagens com outros paradigmas, os números sugeridos não devem ser tão restritos.

Em um script Python, por exemplo, uma função sem argumentos não faz sentido, porque você estaria mudando variáveis globais dentro do contexto da função, o que viola os princípios de responsabilidade única (SRP) e de isolamento de funcionalidades

1.9.3. Funções - Python

```python
def average(numbers:list) -> float:
    return sum(numbers)/len(numbers)

print(average([1,2,3,4]))
```

E005. Criação de classes

Sua tarefa é criar um sistema de biblioteca simples com classes para livros, autores e uma biblioteca. Você precisará estabelecer

relacionamentos entre essas classes e implementar métodos para gerenciar a coleção da biblioteca.

Exercício: Sistema de Biblioteca

- Crie uma classe Autor:
  - Propriedades: nome, ano_de_nascimento, nacionalidade
  - Métodos: **init**, get_info
- Crie uma classe Livro:
  - Propriedades: título, ano_de_publicação, isbn, autor
  - Métodos: **init**, get_info
- Crie uma classe Biblioteca:
- Propriedades: nome, livros (lista para armazenar instâncias de Livro)
- Métodos: **init**, add_book, remove_book, list_books

Sua tarefa é implementar essas classes com métodos apropriados para gerenciar a coleção de livros e autores da biblioteca. Estabeleça relacionamentos entre as classes e assegure-se de que é possível adicionar, remover e listar livros na biblioteca. Os métodos get_info nas classes Autor e Livro devem retornar informações formatadas sobre autores e livros.

Lembre-se de seguir boas práticas de OOP, como encapsulamento, abstração e uso adequado de relacionamentos entre classes.

E005.3. Python

`author.py`

```python
class Author:
    def __init__(self, name:str, birth_year:int,
nationality:str) -> None:
        self.__name = name
        self.__birth_year = birth_year
        self.__nationality = nationality

    @property
    def name(self):
        return self.__name

    @property
    def birth_year(self):
        return self.__birth_year

    @property
    def nationality(self):
        return self.__nationality

    def __str__(self) -> str:
        return f"{self.name}, born in {self.birth_year},
{self.nationality}"

if __name__=="__main__":
    autor1 = Author("Vinicius Borges", 1995, "Brazilian")
    print(autor1.name)
```

book.py

```python
from author import Author

class Book:
    def __init__(self, title:str, publication_year:int,
isbn:str, author:Author) -> None:
        self.__title = title
        self.__publication_year = publication_year
```

```
        self.__isbn = isbn
        self.__author = author

    @property
    def title(self):
        return self.__title

    @property
    def publication_year(self):
        return self.__publication_year

    @property
    def isbn(self):
        return self.__isbn

    @property
    def author(self):
        return self.__author

    def __str__(self) -> str:
        return f"Book: {self.title}\nPublication year:
{self.publication_year}\nISBN: {self.isbn}\nAuthor:
{self.author}\n"

if __name__=="__main__":
    autor1 = Author("Vinicius Borges", 1995, "Brazilian")
    print(autor1)
    book1 = Book("O Zen do Python", 2023, "1234546515654",
autor1)
    print(book1)
```

`library.py`

```
from author import Author
from book import Book
```

```
class Library:
    def __init__(self, name) -> None:
        self.__name = name
        self.__books = []

    @property
    def name(self):
        return self.__name

    def add_book(self, book:Book):
        self.__books.append(book)

    def remove_book(self, book:Book):
        self.__books.remove(book)

    def list_books(self):
        for book in self.__books:
            print(book)

if __name__=="__main__":
    # Authors
    rowling = Author("J.K. Rowling", 1965, "British")
    orwell = Author("George Orwell", 1903, "British")
    austen = Author("Jane Austen", 1775, "British")
    twain = Author("Mark Twain", 1835, "American")
    marquez = Author("Gabriel García Márquez", 1927,
"Colombian")
    christie = Author("Agatha Christie", 1890, "British")
    kafka = Author("Franz Kafka", 1883, "Czech")

    # Books
    book1 = Book("Harry Potter and the Sorcerer's Stone",
1997, "978-0590353403", rowling)
    book2 = Book("1984", 1949, "978-0451524935", orwell)
    book3 = Book("Pride and Prejudice", 1813, "978-
0141439518", austen)
    book4 = Book("The Adventures of Huckleberry Finn",
1884, "978-0486280615", twain)
```

```
    book5 = Book("One Hundred Years of Solitude", 1967,
"978-0060883287", marquez)
    book6 = Book("Murder on the Orient Express", 1934,
"978-0062073501", christie)
    book7 = Book("The Metamorphosis", 1915, "978-
0143129471", kafka)

    # Library
    library1 = Library("Vienna Library")
    library1.add_book(book1)
    library1.add_book(book2)
    library1.add_book(book3)

    library1.list_books()

    library1.remove_book(book3)
    library1.remove_book(book1)

    library1.list_books()
```