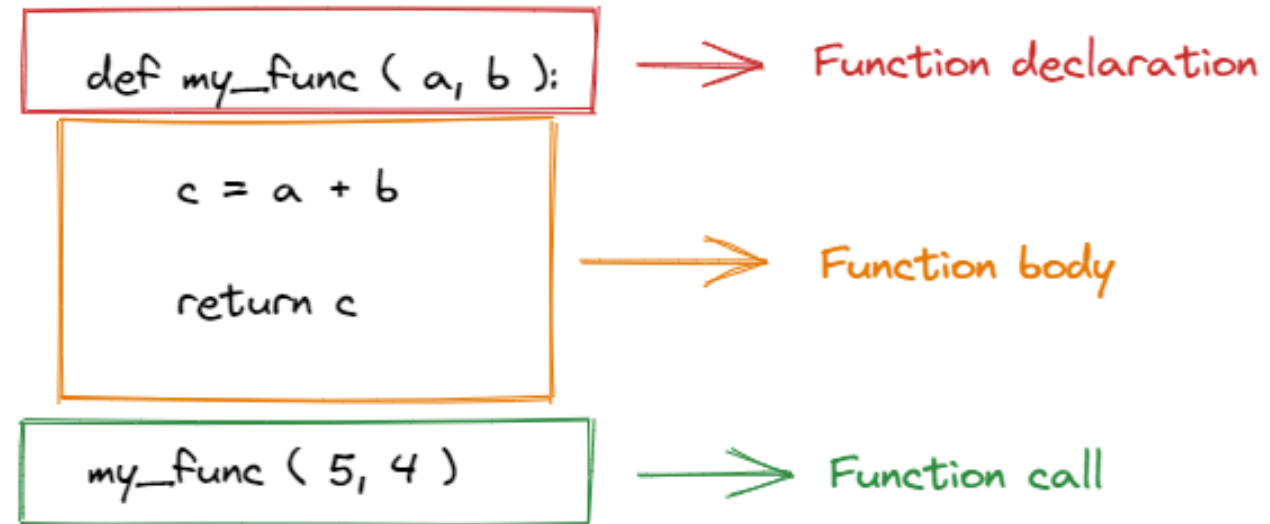


# TÓPICO 04 - FUNÇÕES

Clean Code - Professor Ramon Venson - SATC 2025

# Funções

Funções são blocos de código que realizam uma tarefa específica. Elas podem receber parâmetros, realizar cálculos e retornar um resultado.



```
public class CustomerProcessor {  
  
    public void processCustomerData(Customer customer) {  
        // Passo 1: Valida customer data  
        if (customer.getName() == null || customer.getName().isEmpty()) {  
            return System.out.println("Error: Customer name is missing.");  
        }  
        if (customer.getEmail() == null || customer.getEmail().isEmpty()) {  
            return System.out.println("Error: Customer email is missing.");  
        }  
  
        // Passo 2: checa customer status  
        boolean isReturningCustomer = checkIfReturningCustomer(customer);  
        if (isReturningCustomer) {  
            System.out.println("Returning customer found: " + customer.getName());  
        } else {  
            System.out.println("New customer: " + customer.getName());  
        }  
  
        // Passo 3: processa o endereço  
        String customerAddress = customer.getAddress();  
        if (customerAddress == null || customerAddress.isEmpty()) {  
            System.out.println("Error: Customer address is missing.");  
            return;  
        }  
        System.out.println("Customer address validated: " + customerAddress);  
    }  
}
```

```
public class CustomerProcessor {  
    public void processCustomerData(Customer customer) {  
        validateCustomerData(customer);  
        checkCustomerStatus(customer);  
        processCustomerAddress(customer);  
    }  
}
```

## Argumentos vs Parâmetros

The diagram shows a function definition and a function call. In the function definition, the parameters `param1` and `param2` are highlighted with orange arrows pointing to the label **Parameters**. In the function call `sum(5, 6);`, the arguments `5` and `6` are highlighted with blue arrows pointing to the label **Arguments**.

```
function sum(param1, param2){  
  return param1 + param2;  
}  
  
sum(5, 6);
```

- **Argumentos** : Valores passados para uma função quando ela é chamada.
- **Parâmetros** : Variáveis declaradas em uma função para receber os argumentos.

## Funções vs Métodos

A diferença entre funções e métodos é que os métodos são funções que pertencem a uma classe ou objeto específico.

- O objeto é um dos parâmetros de cada método ( `self` / `this` )
- Uma função pode ser chamada de qualquer lugar do código.
- Um método pode ser chamado apenas dentro da classe ou objeto em que foi definido.

# Single Responsibility Principle (SRP)

O Princípio da Responsabilidade Única (SRP) faz parte dos princípios SOLID.

O SRP diz que uma classe/método deve ter apenas uma responsabilidade.

Ao invés de:

```
public class PokemonCreator {  
    PokemonBattle battle;  
    PokemonRepository pokemonRepository;  
    public void generateWildPokemon(PokemonType type, int level) {  
        Pokemon pokemon = new Pokemon(type, level);  
        pokemonRepository.save(pokemon);  
        System.out.println("Wild " + pokemon.getName() + " appeared!");  
        battle.startBattle(pokemon);  
    }  
}
```



Use:

```
public class PokemonCreator {  
    PokemonRepository pokemonRepository;  
    public void generateWildPokemon(PokemonType type, int level) {  
        return new Pokemon(type, level);  
    }  
  
    public void savePokemon(Pokemon pokemon) {  
        pokemonRepository.save(pokemon);  
    }  
}
```

Removemos algumas responsabilidades dessa classe e dividimos o método `generateWildPokemon()` em dois métodos separados.

## Ordem de Leitura

O código-fonte deve contar a história de forma lógica e linear.

Pode-se usar diferentes estratégias, como ler o código de cima para baixo (top-down) ou de baixo para cima (bottom-up).

Ao invés de:

```
public class GerenciadorDePedidos {  
    private void enviarConfirmacao(String pedido) {}  
    private void gerarNotaFiscal(String pedido, double valor) {}  
  
    public void processarPedido(String pedido) {  
        if (validarPedido(pedido)) {  
            double valor = calcularValorTotal(pedido);  
            gerarNotaFiscal(pedido, valor);  
            enviarConfirmacao(pedido);  
        } else {  
            System.out.println("Pedido inválido.");  
        }  
    }  
  
    private double calcularValorTotal(String pedido) {}  
    private boolean validarPedido(String pedido){}  
}
```

Ao invés de:

```
public class GerenciadorDePedidos {  
    public void processarPedido(String pedido) {  
        if (validarPedido(pedido)) {  
            double valor = calcularValorTotal(pedido);  
            gerarNotaFiscal(pedido, valor);  
            enviarConfirmacao(pedido);  
        } else {  
            System.out.println("Pedido inválido.");  
        }  
    }  
  
    private boolean validarPedido(String pedido){}  
    private void gerarNotaFiscal(String pedido, double valor) {}  
    private double calcularValorTotal(String pedido) {}  
    private void enviarConfirmacao(String pedido) {}  
}
```

## Switches

O uso de `switch` é geralmente desencorajado por questões de legibilidade e manutenção.

Assim como um conjunto de `if`, o `switch` pode ser substituído por uma tabela de decisão.

Ao invés de:

```
public Money calculatePay(Employee e) throws InvalidEmployeeType {  
    switch (e.type) {  
        case COMMISSIONED:  
            return calculateCommissionedPay(e);  
        case HOURLY:  
            return calculateHourlyPay(e);  
        case SALARIED:  
            return calculateSalariedPay(e);  
        default:  
            throw new InvalidEmployeeType(e.type);  
    }  
}
```

Use:

```
public abstract class Employee {  
    public abstract boolean isPayday();  
    public abstract Money calculatePay();  
    public abstract void deliverPay(Money pay);  
}
```

## Nomes Significativos

Nomes de métodos ou funções são geralmente verbos ou frases verbais, como: `post_tweet` , `delete_user` , `calculate_total` .



## Parâmetros em Funções

Segundo o livro "Clean Code", a quantidade ideal de parâmetros de uma função deve ser 0, seguida de 1, 2 ou 3.

É importante compreender que quanto mais parâmetros uma função tiver, mais difícil se torna identificar todas as possibilidades de uso.

```
createSchedule(class1Name, class1Credits, class2Name,  
class2Credits, class3Name, class3Credits, class4Name,  
class4Credits, class5Name, class5Credits, class6Name,  
class6Credits, class7Name, class7Credits, class8Name,  
class8Credits, class9Name, class9Credits, class10Name,  
class10Credits) {
```

Ao invés de:

```
public void sendEmail(String to, String from, String subject, String body, boolean isImportant) {}
```

Use:

```
public void sendEmail>Email(email) {}
```

## Parâmetros Lógicos

Robert C. Martin:

| "Esses parâmetros são feios"

O uso de um booleano como parâmetro de uma função é um sinal de que a função faz mais do que uma coisa.

Ao invés de:

```
public void render(Boolean isSilent)
```

Use:

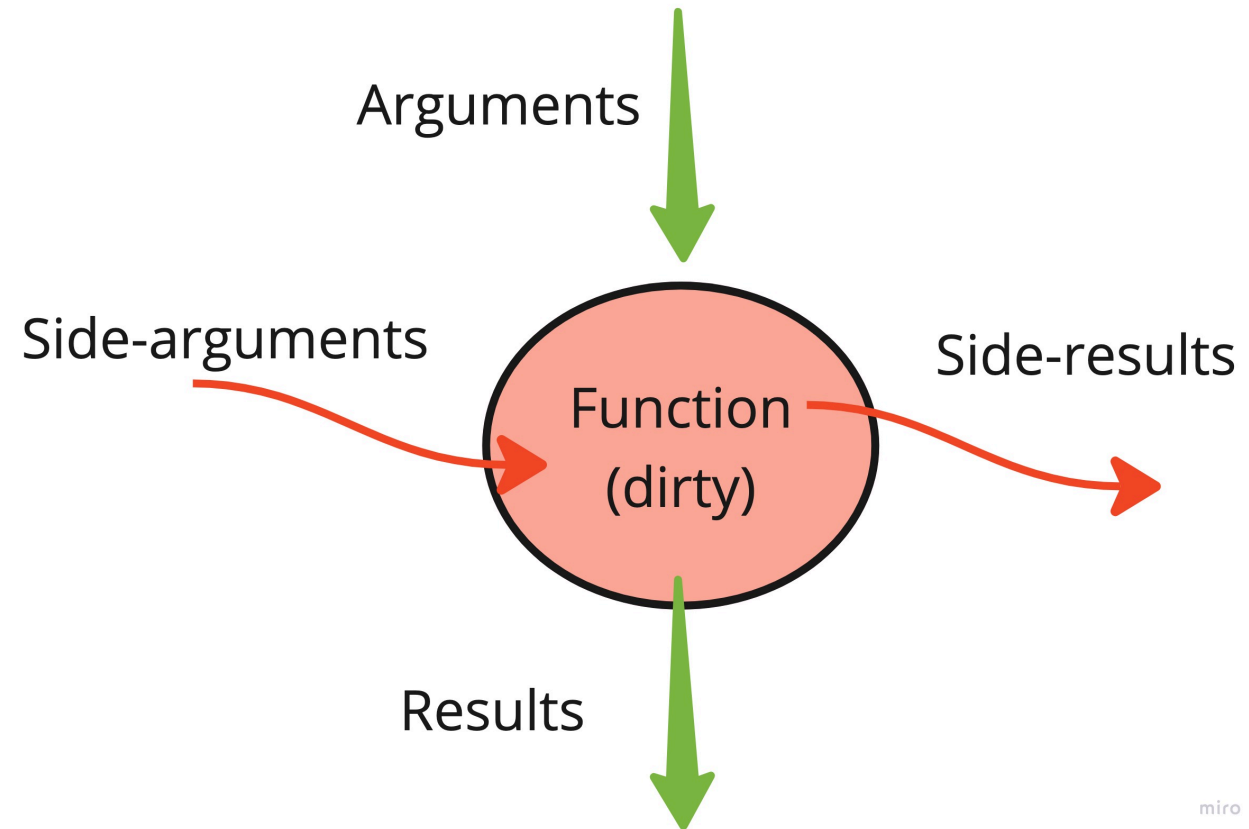
```
public void render()  
public void renderInSilentMode()
```

Nesse caso, ainda que uma decisão precise ser tomada, ela não será tomada dentro da função, mas em outro lugar do programa.

## Efeitos Colaterais

Efeitos colaterais são ações que acontecem fora do escopo da função, ou seja, alteram o estado do programa ou recebem dados de forma indireta.

Uma função que altera o estado de uma variável global é um exemplo de efeito colateral que é difícil de rastrear e testar.



Ao invés de:

```
public void updatePhysics() {  
    // Código que atualiza a posição do objeto  
    RenderServer.update(this);  
}
```



Use:

```
public void updatePhysics() {  
    // Código que atualiza a posição do objeto  
}  
  
public void updateRender() {  
    // Código que atualiza a renderização do objeto  
}
```

## Parâmetros de Saída

Parâmetros são geralmente interpretados como **entradas** de uma função.

Parâmetros de saída são alterados pela função e seu uso é desencorajado, especialmente em linguagens orientadas a objetos.

```
private static bool MethodWithoutParam(int id, out string message)
{
    if (id > 0 && id <= 10)
    {
        message = "Normal Response";
        return true;
    }
    else
    {
        message = "Invalid id";
        return false;
    }
}
```

Ao invés de:

```
public void add_to_list(List<String> list, String value) {  
    list.add(value);  
}
```

Use:

```
list.add(value);
```

## **Separação comando-consulta**

Uma função deve executar uma modificação ou definir uma consulta, mas não ambas.

Ao invés de:

```
public Boolean setUsername(String username) {  
    this.username = username;  
    return this.username;  
}  
  
String newName = setUsername(username)
```

A junção de comandos e consultas é um sinal de que a função faz mais do que uma coisa.

Use:

```
public void setUsername(String username) {  
    this.username = username;  
}  
  
public void getUsername() {  
    return this.username;  
}
```

A separação entre comandos e consultas é uma forma de evitar efeitos colaterais.

## Exception handling: try/catch vs Result

Which is best?

```
Future<Location>
getLocationFromIP(String ipAddress) async {
  final uri = Uri.parse('https://geo.com/$ipAddress/');
  final response = await http.get(uri);
  switch (response.statusCode) {
    case 200:
      final data = json.decode(response.body);
      return Location.fromMap(data);
    default:
      throw Exception(response.reasonPhrase);
  }
}
```

```
Future<Result<Exception, Location>>
getLocationFromIP(String ipAddress) async {
  final uri = Uri.parse('https://geo.com/$ipAddress/');
  final response = await http.get(uri);
  switch (response.statusCode) {
    case 200:
      final data = json.decode(response.body);
      return Success(Location.fromMap(data));
    default:
      return Error(Exception(response.reasonPhrase));
  }
}
```

Throw an exception or return an error?

## Erros vs Exceções

Funções que retornam valores de erros são difíceis de testar e podem ser substituídas por exceções.



Ao invés de:

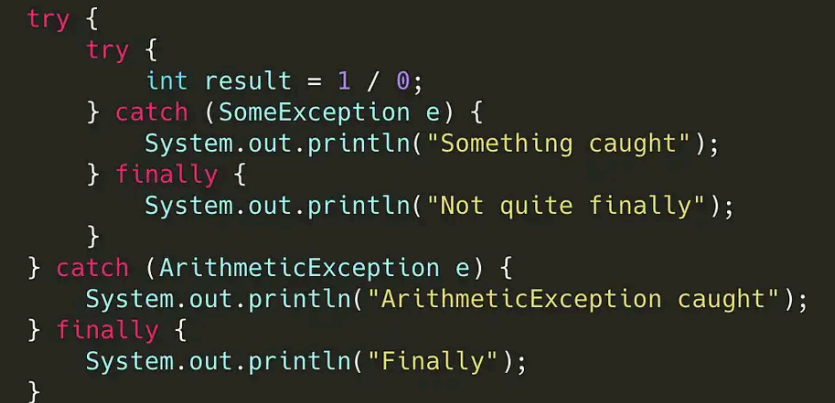
```
public int divide(int a, int b) {  
    if (b == 0) {  
        return -1;  
    }  
    return a / b;  
}
```

Use:

```
public int divide(int a, int b) {  
    if (b == 0) {  
        throw new IllegalArgumentException("Divisor cannot be zero");  
    }  
    return a / b;  
}
```

## Extraia Exceções

Blocos de `try-catch` são difíceis de ler. Dessa forma, pode ser interessante extrair exceções para uma função separada, que realiza apenas o tratamento de erros.



```
try {  
    try {  
        int result = 1 / 0;  
    } catch (SomeException e) {  
        System.out.println("Something caught");  
    } finally {  
        System.out.println("Not quite finally");  
    }  
} catch (ArithmeticException e) {  
    System.out.println("ArithmeticException caught");  
} finally {  
    System.out.println("Finally");  
}
```

Ao invés de:

```
public String readFile(String path) {  
    try {  
        File file = new File(path);  
        Scanner scanner = new Scanner(file);  
        String line = "";  
        while (scanner.hasNextLine()) {  
            line = scanner.nextLine();  
            scanner.close();  
        }  
        return line;  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("Arquivo não encontrado");  
    }  
    catch (IOException e) {  
        System.out.println("Erro ao ler o arquivo");  
    }  
}
```

Use:

```
public void readFile(String path) throws FileNotFoundException, IOException {  
    File file = new File(path);  
    Scanner scanner = new Scanner(file);  
    String line = "";  
    while (scanner.hasNextLine()) {  
        line = scanner.nextLine();  
        scanner.close();  
    }  
    return line;  
}
```

```
public void getFileContent(String path) {  
    try {  
        readFile(path);  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("Arquivo não encontrado");  
    }  
    catch (IOException e) {  
        System.out.println("Erro ao ler o arquivo");  
    }  
}
```

## Evite Repetição

Códigos duplicados são difíceis de manter. Generalizar funções muito parecidas pode ser uma alternativa viável para evitar a repetição de código.

## Conclusão

- Funções devem ser pequenas, com no máximo 20 linhas.
- Funções devem ser bem nomeadas.
- Funções devem ter apenas uma responsabilidade.
- Funções devem ter poucos parâmetros.
- Funções devem ser testáveis.
- Funções devem evitar efeitos colaterais.
- Funções devem evitar parâmetros de saída.
- Funções devem ser separadas em comandos e consultas.



## Material de Apoio

- [Daniel Wisky](#)
- [Introdução a Monadas](#)
- [seydialkan](#)