Fluxo de execução procedural

Durante toda a disciplina Algoritmos e Introdução à Programação nós utilizamos o paradigma estruturado para evoluirmos nossos conhecimentos sobre programação. Com isso, o Javascript foi a linguagem escolhida para nos apoiar nesse aprendizado.

```
1   var saldo = 500.00;
2
3   function sacar(quantia){
4
5     let mensagemSaqueAutorizado = "Saque autorizado com sucesso! Saldo: "
6                                   + descontarSaqueDoSaldo(quantia)
7
8     let mensagemSaqueNegado = "Saque negado! Saldo: " + saldo
9
10    if (permitirSaque(quantia) === false){
11      console.log(mensagemSaqueNegado)
12      return false
13    }
14    console.log(mensagemSaqueAutorizado)
15
16  }
17
18  function permitirSaque(quantia) {
19
20    if (quantia > saldo){
21      return false
22    }
23    return descontarSaqueDoSaldo(quantia)
24
25  }
26
27  function descontarSaqueDoSaldo(quantia) {
28      return saldo - quantia
29  }
30
31  sacar(100) //quantiaMenorQueSaldo: DADO saque = 100 ESPERO saldo = 400
32  sacar(500) //quantiaIgualQueSaldo: DADO saque = 500 ESPERO saldo = 0
33  sacar(550) //quantiaMaiorQueSaldo: DADO saque = 550 ESPERO saldo = 500
34
```

```
CONSOLE ×
Saque autorizado com sucesso! Saldo: 400
Saque autorizado com sucesso! Saldo: 0
Saque negado! Saldo: 500
```

Estudo de caso

Em uma abordagem estruturada a execução do nosso programa se dá respeitando as chamadas de funções que compõem toda a estrutura. Utilizando variáveis em diferentes escopos para representar as entradas a serem processadas nos diferentes algoritmos desenvolvidos.

No nosso algoritmo de estudo de caso acima podemos observar o total de 3 funções cada uma com uma responsabilidade única bem definida:

1. **Sacar**: Iniciar o fluxo de funções com um valor de saque e imprimir as mensagens de “mensagemSaqueAutorizado” e “mensagemSaqueNegado”.
2. **Permitir Saque**: Decide se o saque deve ser permitido ou não dado um valor de quantia
3. **Descontar Saque do Saldo**: Desconta um valor de saque do saldo em conta corrente.

A leitura sistemática computacional do nosso programa é realizada em seu fluxo de execução padrão: De cima para baixo. A cada chamada de função encontrada durante a leitura do nosso programa, a função correspondente é procurada por toda a estrutura do programa e executada. Observe a seguir:

- Na linha 6 do código abaixo uma função é declarada como parte do valor da variável *mensagemSaqueAutorizado*

```
5   let mensagemSaqueAutorizado = "Saque autorizado com sucesso! Saldo: "
6                               + descontarSaqueDoSaldo(quantia)
```

- Quando a leitura do programa chega na linha 14, no momento em que a variável *mensagemSaqueAutorizado* é utilizada pelo programa, o retorno da função *desconarSaqueDoSaldo()* é concatenado ao texto definido na variável.

```
14  console.log(mensagemSaqueAutorizado)
```

- Na linha 10, também há uma declaração de função e, quando o fluxo do programa chega na referida linha, a função *permitirSaque()* é executada pela lógica de decisão.

```
10  if (permitirSaque(quantia) === false){
```

Dados os exemplos acima, podemos concluir que, o paradigma estruturado realiza a leitura sistemática computacional baseado na execução das funções declaradas ao longo do programa, iniciando sempre sua leitura de forma ordenada de cima para baixo.

Responsabilidade única e teste das funções

Quando os requisitos para um programa são bem definidos, é esperado que cada requisito esteja definido dentro de uma mais funções, sempre se atentando para que cada função tenha 1 responsabilidade atendida e que haja, antes de tudo um teste de comportamento definido.

As linhas 30, 31 e 32 possuem, em formato de comentário, cenários de testes que realizam uma chamada da função principal do programa passando diferentes valores por parâmetro para que o valor retornado pela execução do programa possa ser comparado com o resultado esperado pelo teste. Observe:

```
//quantiaMenorQueSaldo: DADO saque = 100 ESPERO saldo = 400
//quantiaIgualQueSaldo: DADO saque = 500 ESPERO saldo = 0
```

O primeiro comentário espera que o valor 100 seja utilizado como entrada para o programa e espera o retorno de “saldo” com valor 400.

O segundo comentário espera que o valor 500 seja utilizado como entrada para o programa e espera o retorno de “saldo” com valor de 0.

Ambos os cenários acima podem ser avaliados observando o que será impresso pelo programa quando a linha 14 do código for executada:

```
14    console.log(mensagemSaqueAutorizado)
```

Já o teste abaixo espera que o valor de 550 seja utilizado como entrada para o programa e espera o retorno de “saldo” com valor de 500. O que se considera como saque não autorizado.

```
//quantiaMaiorQueSaldo: DADO saque = 550 ESPERO saldo = 500
```

O cenário acima pode ser avaliado observando o que será impresso pelo programa quando a linha 11 do código for executada:

```
11      console.log(mensagemSaqueNegado)
```

Os cenários de teste descritos para esse programa (linhas 31,32 e 33) são avaliados com a chamada da função de início do programa: “sacar”.

Mas o que é impresso por essa função é apenas um valor que é retornado pelas funções que correspondem com cada responsabilidade de cada teste definido.

Na linha 10 (abaixo) é possível observar uma lógica de decisão que compara o valor retornado por *permitirSaque().*

```
10    if (permitirSaque(quantia) === false){
```

No momento em que a chamada da referida função for executada, o bloco de código da função *permitirSaque()* será executado.

```
18  function permitirSaque(quantia) {
19
20    if (quantia > saldo){
21      return false
22    }
23    return descontarSaqueDoSaldo(quantia)
24
25  }
```

A função *permitirSaque()* tem a única responsabilidade de permitir ou não o saque. Na linha 20, a lógica de decisão escrita compara se o valor de “quantia” é maior que “saldo”. Se o resultado dessa lógica for *true* então a o fluxo do programa será desviado para a função principal *sacarDinheiro(),* pois *return false* interrompe o fluxo natural de execução do programa atribuindo o valor *false* para aquela função. Ou seja, quando um valor *false* é retornado dentro de uma função, não é mais necessário do programa permanecer dentro dela.

A lógica de execução escrita no interior da lógica de decisão realiza 2 operações. Como pode ser observado abaixo:

```
11      console.log(mensagemSaqueNegado)
12      return false
```

A linha 11 imprime o valor de *mensagemSaqueNegado* e, mesmo após a impressão do valor, o programa mantém o fluxo natural de leitura (de cima para baixo) e, somente na linha 12, ao encontrar *return false*, o fluxo do programa é interrompido.

Você deve estar se perguntando qual a razão de existir o *return false* para interromper o fluxo do programa. Que tal você realizar esse teste? Remova a linha 12 do programa e perceba que a saída do programa será modificada imprimindo também o valor de *mensagemSaqueAutorizado.* Um erro grave! Isso desorientaria o usuário que solicitou um saque. Afinal, ele teria ou não seu saque autorizado?

Como o valor de *mensagemSaqueAutorizado* seria impresso logo após, na linha 14, pelo fluxo natural “de cima para baixo” da leitura sistemática estruturada do programa, o fluxo precisou ser, então, interrompido na linha 12 com o *return false.*

Proteção das responsabilidades

Nosso estudo de caso é um cenário que representa a ação de sacar dinheiro de uma conta corrente. Para isso, trabalhamos com algumas entradas fundamentais em nosso programa, são elas: Saldo e quantia.

Já entendemos a responsabilidade de *permitirSaque()*, e quanto a responsabilidade de *descontarSaqueDoSaldo()*?

O nome sugestivo da função já sugere o que é realizado no interior: Um valor de *saque* é retirado do valor de *saldo*. O resultado dessa subtração é retornado pela função. Se fossemos testar o comportamento dessa função seria preciso apenas 2 valores para serem subtraídos. Essa função apenas subtrai um número por outro. Com isso, o cenário de teste abaixo seria totalmente possível com apenas alguns ajustes. Observe:

```
26  function descontarSaqueDoSaldo(quantia) {
27      console.log(saldo - quantia)
28  }
29
30  // quantiaMaiorQueSaldo: dado saque = 1000 saldo = 500 espero -500
31  descontarSaqueDoSaldo(1000)
```

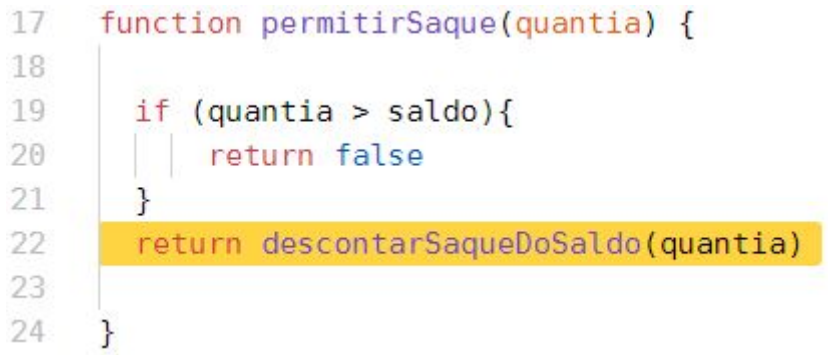
O cenário de teste acima realiza o teste de comportamento da função *descontarSaqueDoSaldo()* para valores de saque superiores ao saldo disponível em conta corrente.

Apenas alguns ajustes foram necessários de serem feitos: Linha 27, ao invés de *return* o *console.log()* foi utilizado para imprimir o resultado no console.

O valor de quantia de *1000* foi passado na linha 31 durante a chamada da função e o valor de *saldo,* está público para todo o programa ao ser declarado fora de qualquer função na linha 1 do código com o valor de *500.00.*

O cenário realiza a subtração desses 2 valores mencionados acima resultando em um valor negativo. Bem, você já sabe que o resultado de valores negativos, para nosso estudo de caso resultaria em uma falha muito grave de segurança. Mas, nós já não testamos o programa passando valores de *saque* maiores que *saldo* e o comportamento correto pode ser observado. Então, qual o problema do nosso programa? Onde está o erro?

Se a responsabilidade que uma função precisa ter é o que esperamos do comportamento da função, a função *descontarSaqueDoSaldo()* está correta. Afinal, o que garante o correto funcionamento do programa? O trecho de código a seguir:



Podemos observar que, a função *permitirSaque()* se encarrega de interromper o fluxo do programa quando o valor de *quantia* for maior que o valor de *saldo*. Além disso, se encarrega de, executar a função *descontarSaqueDoSaldo()* quando o valor de *quantia* for menor ou igual ao valor de *saldo.*

Você deve estar se questionando se há uma quebra de responsabilidade única nessa função. Mas já lhe respondo que não. Estamos diante de uma lógica de decisão que realiza *return false* caso o resultado da decisão seja *true* e *return descontarSaqueDoSaldo(quantia)* caso o resultado da decisão seja *false.*

Mas o mais interessante a se esclarecer não é nem se há quebra de responsabilidade na função *permitirSaque(),* mas sim por que a função *descontarSaqueDoSaldo()* sozinha apresenta um comportamento e quando integrada ao comportamento do programa, se iniciarmos o programa com a função *sacar(),* a gente desconfia que o comportamento muda por não mais permitir valores de *saldo* negativo?

Essa percepção da função “isolada” e a função “integrada” nos sugere que nenhuma função, em um paradigma estruturado, deverá existir sozinha. Toda função estará inserida de forma integrada dentro de um contexto de programa. Por isso, que devemos sempre garantir a proteção do comportamento de cada “unidade” ou “função” desse programa e, quando integrada, a “unidade” ou “função” não poderá alterar o comportamento esperado para o programa.

Falha de proteção do programa

Na leitura anterior percebemos que devemos nos atentar para o correto funcionamento das unidades do sistema, o qual chamamos também de função. Igualmente importante ao comportamento das unidades de forma isolada, precisamos garantir o funcionamento das unidades de forma integrada, formando assim, um programa que funcione com o comportamento adequado aos requisitos de software.

Para nosso estudo de caso, precisamos garantir que somente a quantia solicitada para saque possa ser realizada, ou seja, jamais poderemos receber *100* como entrada para o valor para *saque* e esse valor ser alterado para *500* em alguma parte do nosso programa. Ou, até mesmo, jamais poderemos permitir que um valor de *saque* seja maior que um valor de *saldo.*

Em um paradigma estruturado, todas as funções e variáveis estão visíveis por todo o programa e podem ser utilizadas e manipuladas por todo o programa. Claro, no caso de variáveis temos um nível de proteção de *escopo* e, no caso de funções, só pode ser executada a função que existir no momento da leitura da linha do código, por exemplo:

```
1  function a(){
2    let variavelA
3  }
4
5  function b(){
6    let variavelb
7  }
```

A *variavelA* só é reconhecida em *function a()* e a *variavelb* só é reconhecida em *function b()*

Assim como no momento em que a leitura sistemática executar a linha 3 desse código, a *function b()* ainda não existiria para o programa, uma vez que a leitura é de “cima para baixo”. Mas no momento em que a *function b()* for executada, o programa já terá sabido da existência da *functionA().*

Com todo esse esclarecimento, que tal uma pequena alteração em nosso programa? Comente as linhas 10,11,12 e 13 do programa:

```
9   /**
10  if (permitirSaque(quantia) === false){
11    console.log(mensagemSaqueNegado)
12    return false
13  }
14  */
```

Feito isso, vamos observar se algo mudou no comportamento esperado para o teste escrito abaixo:

```
//quantiaMaiorQueSaldo: DADO saque = 550 ESPERO saldo = 500
sacar(550)
```

Nossa! Uma falha grave de segurança pode ser percebida. Um simples comentário de código pode trazer sérias consequências indesejadas. Achou que esse erro não poderia ser realizado por um programador por ser muito “fácil de perceber”? Ok. E a seguinte alteração?

```
10    if (permitirSaque(quantia) === true){
11      console.log(mensagemSaqueNegado)
12      return false
13    }
```

Descobriu tão facilmente qual a alteração foi realizada no código igual à alteração que comentou o trecho de código?

Na linha 10, houve, por engano (ou intencional), uma modificação de *false* para *true* que expôs todo o funcionamento do nosso programa ao risco de permitir um valor de *saque* maior que o valor de *saldo* disponível. Para ambas as alterações de código acima, o resultado a seguir é esperado:

```
30   sacar (550)
```

```
CONSOLE ×

Saque autorizado com sucesso! Saldo: -50
```

Podemos concluir que, o paradigma estruturado, por mais que seja permitido o desenvolvimento de testes de unidade ou “funções isoladas” e de integração ou “funções integradas”, o fato das funções estarem sempre visíveis por toda a estrutura do programa, o comportamento correto do programa esperado pelo requisito de software pode ser prejudicado por erros intencionais ou não.