Introdução

*“Não existe trabalho ruim. Ruim é ter que trabalhar”*

**Sr. Madruga**

Olá alunos, assustados com a frase do Sr. Madruga?

Antes de começar com nossa primeira disciplina de programação, vamos refletir um pouco sobre a frase preguiçosa que acabamos de ler?

Pois bem, caros alunos programadores, a preguiça é o pior e o melhor do bom programador. Confuso? Nem tanto. A roda foi inventada há milhares de anos antes de Cristo certamente impulsionada pela preguiça. Imaginem só se toda vez que fosse preciso carregar algum peso de um lado para outro fosse necessário usar a força humana para essa tarefa? Finalmente, um preguiçoso decidiu usar a cabeça ao invés da força. Então, a preguiça poderia ser considerada o melhor do bom programador, pois usar a preguiça para desenvolver soluções e diminuir o retrabalho, agiliza as tarefas do cotidiano.

Mas, porquê "Ruim é ter que trabalhar" ? Aqui então passaremos a discutir sobre o fundamento desta disciplina: Aprender a produzir ao invés de trabalhar. Sim, são verbos diferentes com significados diferentes. Usando o exemplo da criação da roda, podemos afirmar que trabalho é o que gera a preguiça e produtividade é o que gera motivos para determinada ação: O desenvolvimento da roda.

Trabalhar gera preguiça e produzir, MOTIVAÇÃO !

Nossas disciplinas sob o escopo da programação serão motivadas pela produtividade de criar e usar a "roda" e aprendermos que jamais uma roda será recriada novamente, mas sim, melhorada. Uma roda há mais de 5 mil anos atrás não foi criada para sustentar um avião, mas foi melhorada até ser utilizada para tal.

Para finalizar essa leitura e partirmos ao aprendizado, sugerimos alguns pré requisitos para esta disciplina:

- Motivação

- Curiosidade

- Prática

- Autocrítica

Somente um programador *motivado* se torna *curioso* para aprender coisas novas e praticá-las e somente a *prática* torna o programador um bom *crítico* de seu próprio trabalho e o prepara para **trabalhar em equipe**.

E sobre o trabalho em equipe (o que mais faremos durante nossa disciplina), fique com uma importante frase dita pelo ícone do basquete mundial, ***Michael Jordan***:

*“O talento vence jogos, mas só o trabalho em equipe ganha campeonatos.”*

Faça um bom proveito desta disciplina. Bons estudos!

Algoritmos e Orientação a Objetos

Unidade 1 - Introdução e conceitos

Subunidade 1 - Noções de algoritmos

Na certeza de que um algoritmo não será a grande novidade a ser lida nas próximas páginas, optamos por identificar alguns conceitos importantes do processo de desenvolvimento de um bom algoritmo.

O processo de desenvolvimento de um bom algoritmo está em saber como começa-lo. O início do desenvolvimento de qualquer algoritmo que você for construir na vida, sempre, será reconhecer o problema. Para isso, você precisa aprender a refletir e usar seu conhecimento de mundo para chegar à resolução correta do problema.

1.1.1 Antes de reconhecer o problema

Creio que você já deve ter recebido seu primeiro salário como aluno, por isso vamos a um exemplo prático do dia a dia: Sacar dinheiro em um caixa eletrônico.

Antes de olhar para a representação de algoritmo abaixo, reflita sobre as tarefas necessárias para “Sacar dinheiro” usando sua experiência pessoal. Refletiu? Então, observe:

**Algoritmo**: Sacar dinheiro

INÍCIO

1. Ir até o caixa eletrônico
2. Colocar o cartão
3. Digitar a senha
4. Solicitar o saldo
5. Se o saldo for maior ou igual à quantia desejada, sacar a quantia desejada; caso contrário, sacar o valor do saldo
6. Retirar dinheiro
7. Retirar o cartão

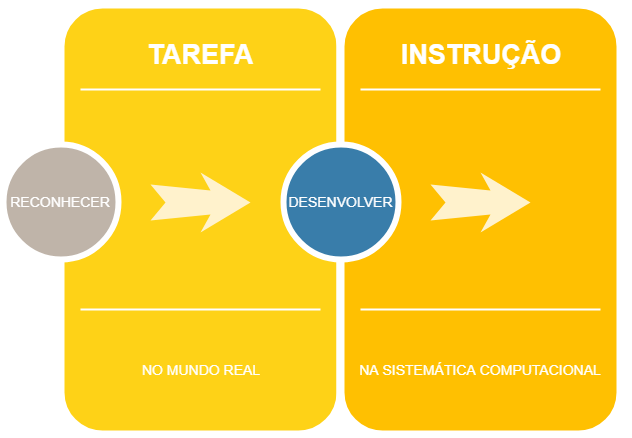
FIM

Perceba que nessa representação nós definimos as tarefas necessárias para *sacar dinheiro* ao separamos nosso algoritmo em: **Início**, **meio** - *que se parece com uma lista tarefas do cotidiano* - e **fim** para atingirmos o nosso objetivo final.

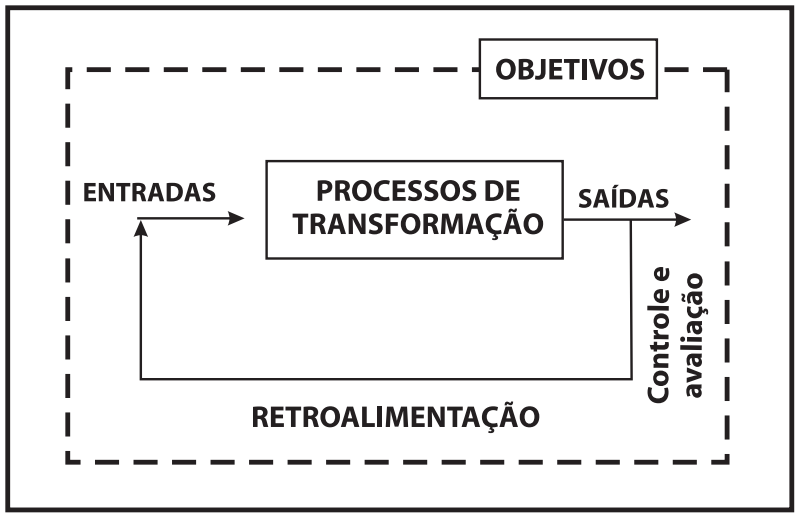
A representação de algoritmo acima tem a única responsabilidade de sacar dinheiro, e dentro desse algoritmo, vamos descobrir outros algoritmos com responsabilidades também específicas para atingir outros problemas. Mas primeiro, aprenderemos a reconhecer o problema que motiva o desenvolvimento de um algoritmo.

1.1.2 Reconhecendo o problema

No mundo real, nós identificamos as tarefas do cotidiano a serem traduzidas em instruções para serem executadas por um algoritmo na sistemática computacional. Como é sugerido abaixo:

**

Reconhecer um problema é descobrir que um objetivo precisa ser alcançado através da solução de algumas tarefas específicas, como representado a seguir:



Essa imagem esclarece a maneira como um objetivo é alcançado: Através do controle e avaliação de instruções de tarefas que transformam entradas em saídas.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Qual objetivo? **Sacar dinheiro**

Como atingir o objetivo? **Reconhecendo e resolvendo as tarefas necessárias**

Como resolver as tarefas do objetivo? **Controlando e avaliando cada instrução da tarefa. Através de um processo de transformação que precisa gerar uma entrada em uma saída válida**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

*Sacar dinheiro reconhecendo e resolvendo as tarefas necessárias. Controlando e avaliando cada instrução da tarefa. Através de um processo de transformação que precisa gerar uma entrada em uma saída válida.*

1.1.3 O que é um algoritmo?

Depois de se pensar no mundo real, precisamos aprender a pensar na leitura sistemática computacional através de um algoritmo.

Um algoritmo é um conjunto de instruções que resolvem um problema específico. É um “caminho” para a solução de um problema e, em geral, existem muitos caminhos que levam a uma solução satisfatória para resolver o mesmo problema.

Um problema pode ser solucionado através de vários algoritmos, mas um algoritmo resolve apenas um único problema.

Passos para a resolução do problema:

1. Entendimento do problema (saindo do mundo real para a leitura sistemática computacional)
2. Criação de uma sequência de instruções
3. Execução dessa sequência (desempenhada pelo computador)
4. Verificação da adequação da solução

A parte mais importante em resolver um problema[[1]](#footnote-1) é o processo de transformação das entradas em saídas válidas. E esse processo de transformação - *entradas em saídas válidas* - acontece em cada instrução[[2]](#footnote-2) desenvolvida para cada tarefa que foi previamente definida para o meu objetivo em questão.

Olhando para o algoritmo “Sacar dinheiro” nós vemos o nosso objetivo “sacar dinheiro” e as tarefas necessárias para sacar dinheiro. Em cada tarefa nós iremos identificar um ou mais problemas específicos que precisam sistematizados na forma de algoritmos para serem, então, resolvidos.

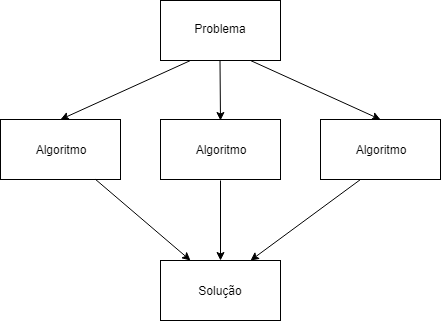
1.1.4 Avaliando um problema

Abaixo iremos avaliar a tarefa 2 do algoritmo: Colocar o cartão.

Para que essa tarefa seja realizada, a tarefa imediatamente anterior a ela precisa ser concluída com sucesso para que se dê início à tarefa seguinte.

*Quando um cartão para saque é confeccionado, é programado nele uma informação que identifica o banco emissor. E cada caixa eletrônico só consegue ler cartões cujo banco proprietário do caixa seja o mesmo banco que o emissor do cartão*. Então chegamos ao nosso primeiro problema para resolver: Validar o cartão inserido.

1.1.5 Resolvendo um problema

**

Observe o fluxo de instruções abaixo e perceba como cada instrução é importante para concluirmos a tarefa “Colocar o cartão”.

Propriedades do algoritmo

1. Composto por ações simples e bem definidas (não pode haver ambiguidade, ou seja, cada instrução representa uma ação que deve ser entendida e realizada)
2. Sequência ordenada de ações
3. Conjunto finito de passos

**Objetivo**: Sacar dinheiro

**Tarefa**: Colocar o cartão

**Problema**: Validar o cartão inserido

**Algoritmo**: Validar cartão

**INÍCIO**

1. INSERIR o código do banco emissor do cartão
2. LER o código do banco emissor do cartão
3. COMPARAR o código do banco do cartão com o código do banco do caixa eletrônico
   1. SE o código do banco do cartão for IGUAL ao código do banco proprietário do caixa eletrônico ENTÃO
      1. PERMITA PROSSEGUIR
   2. SENÃO
      1. NÃO PERMITA PROSSEGUIR

**FIM**

1.1.6 Princípio da responsabilidade única

Em nosso algoritmo anterior, cada instrução recebe uma função, ou responsabilidade única. Por exemplo, a instrução “Ler o código do banco emissor do cartão” só sabe ler um código, apenas isso. Ela apenas lê uns caracteres que representam um código e registra essa entrada - *o código do cartão -* para ser processada pela próxima instrução que compara um código com outro e recebe o nome “Comparar código”.

Então, após esse entendimento, podemos caracterizar como princípio da responsabilidade única - *no contexto da leitura sistemática computacional* - um algoritmo que recebe uma entrada específica, processa essa entrada e gera uma **única saída**, pois essa responsabilidade está relacionada a um problema específico e não conseguiria resolver mais do que um único problema[[3]](#footnote-3).

1.1.7 Identificando as entradas e saídas

Para que o nosso problema fosse resolvido, um valor foi recebido, processado e uma saída foi gerada.

Observe que instrução 2 do algoritmo “Validar cartão” leu um código e o retornou como entrada para o passo 3 que então o recebeu e o processou comparando com outro código - *o do banco do caixa eletrônico* - gerando uma saída que permitia prosseguir ou não com o fluxo de instruções.

No passo 3 *- instrução “comparar códigos -* foram identificadas as entradas “Código do banco do cartão” e “Código do banco do caixa eletrônico”. E as saídas foram: “Permita prosseguir” ou “Não permita prosseguir”.

1.1.8 Avaliando as entradas e saídas

Avaliando nosso algoritmo principal “Sacar dinheiro”, percebemos ser impossível executarmos, por exemplo, a tarefa “Solicitar o saldo” sem que o cartão seja inserido e validado com sucesso. Para isso, as saídas do algoritmo “Validar cartão” - *algoritmo criado para resolver um problema específico relacionado à tarefa “colocar o cartão” -*  precisam ser saídas válidas. Após a resposta de sucesso das saídas desse algoritmo e de todos os demais algoritmos necessários para resolver outros problemas ainda na tarefa “Colocar o cartão” - *se houver* - poderemos então, seguir para a próxima tarefa: “Digitar a senha”.

Ao identificarmos a saída de “Validar cartão”, a gente percebe que a única saída válida de sucesso é “Permita prosseguir”. E o que acontece depois?

Se existirem outros algoritmos a serem resolvidos ainda nessa mesma tarefa, eles precisarão de saídas válidas para que a toda a tarefa seja dada como realizada e então a próxima tarefa do objetivo poderá ser atingida.

A tarefa seguinte, “digitar senha”, desencadeará em uma série de outras entradas para a solução de problemas específicos. Por exemplo, ao digitar a senha, é preciso que a senha digitada seja utilizada como entrada para que haja comparação com a senha cadastrada no cartão. Esses dados e alguns outros, tornam-se necessários como entrada para algoritmos que serão desenvolvidos para resolverem problemas específicos da tarefa “digitar senha”, por exemplo: Validar a senha do usuário.

E se a saída de “validar o cartão” não for uma saída válida?

Nesse caso, nenhuma tarefa posterior a “Colocar o cartão” poderá ser executada e o fluxo é interrompido, pois a tarefa não conseguiu cumprir integralmente com sua responsabilidade de validar o cartão.

Diretrizes para a elaboração de um algoritmo

1. Identificação do problema: Determinar o que se quer resolver ou qual objetivo a ser atingido
2. Identificação das entradas: Quais informações estarão disponíveis
3. Identificação das saídas: Quais informações deverão ser geradas como resultado
4. Definir os passos a serem realizados: Determinar a sequência de ações que leve a solução do problema (transforma as entradas em saídas)
5. Identificar as regras e limitações do problema
6. Identificar as limitações do computador
7. Determinar as ações possíveis de serem realizadas pelo computador
8. Concepção do algoritmo: Registrar a sequência de comandos, utilizando uma das formas de representação de algoritmos
9. Teste da solução: Execução manual de cada passo do algoritmo, seguindo o fluxo estabelecido, para detectar possíveis erros.

Formas de representação de um algoritmo

Não existe uma determinação quanto a melhor forma de se representar um algoritmo. O importante é que a representação seja clara e não induza ao erro ou à ambiguidade de informações.

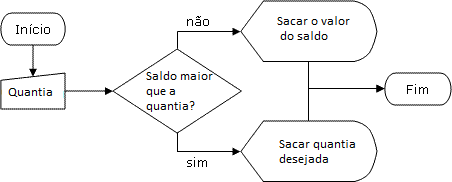
Segue abaixo as representações mais utilizadas:

*Descrição narrativa*

O algoritmo “Sacar dinheiro”foi apresentado sob a forma de descrição narrativa. Cada tarefa é narrada de forma impositiva, diretamente na linguagem natural.

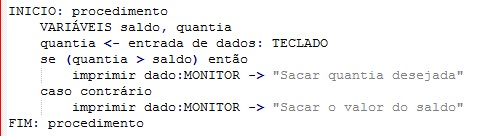
*Fluxograma convencional*

Reescrevendo os passos 4 e 5 do nosso algoritmo principal “Sacar dinheiro” em um fluxograma convencional fica assim:



Pseudocódigo

O mesmo exemplo anterior em pseudocódigo fica representado assim:

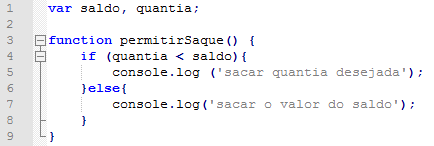


Execução de um algoritmo

Todas as formas de representação de um algoritmo são alternativas para a compreendermos, no fim das contas, como iremos executar, na leitura sistemática computacional, um algoritmo que irá atingir determinado objetivo da vida real.

A execução de um algoritmo na leitura sistemática computacional ocorre através de uma linguagem de programação que será utilizada ao longo de nosso material didático.

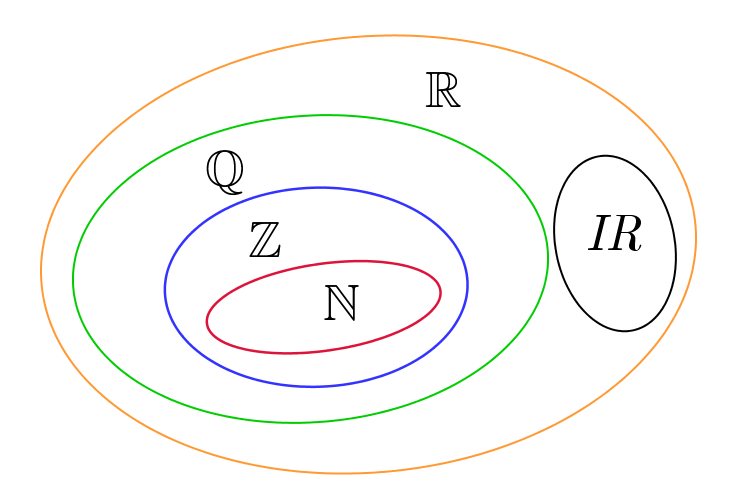
O algoritmo em pseudocódigo que acabamos de representar poderá ser desenvolvido em uma linguagem de programação[[4]](#footnote-4) como no exemplo abaixo:



Subunidade 2 - Conceitos de variáveis e funções

1.2.1 Conjunto dos números reais

O conjunto dos números *Reais* (R) engloba 4 conjuntos de números: Naturais (N), Inteiros (Z), Racionais (Q) e Irracionais (I). O conjunto dos números *Racionais* (Q) é formado pelo conjunto dos Números *Naturais* (N) e dos Números *Inteiros*(Z). Por isso, todo Número *Inteiro* (Z) é *Racional* (Q), ou seja, Z está contido em Q.

**

1.2.2 Tipos de dados da computação

Um tipo de dado nada mais é do que algo do mundo real que pode ser representado computacionalmente. Por exemplo, os números que pertencem ao conjunto dos números inteiros, os números que pertencem ao conjunto dos números reais, letras, caracteres especiais, acentuação, pontuação, palavras, etc.

O objetivo principal de qualquer computador é a resolução de problemas através da manipulação de dados que pode ocorrer de várias maneiras. Em computação, dado é o que pode ser representado, armazenado e manipulado.

Os tipos de dados na computação são divididos em 2 grupos:

*Tipos primitivos*

Os tipos de dados primitivos são os tipos básicos que devem ser implementados por todas as linguagens de programação, como os números reais, inteiros, booleanos e caracteres.

*Inteiro*

Pertence ao conjunto dos números inteiros positivos, negativos e nulos. Ex: 5, -10, 0.

*Real*

Pertence aos números reais. O conjunto dos números inteiros está contido no conjunto dos números reais. A casa decimal, na computação, é representada por ponto (.) e não vírgula (,). Ex: 5.2, 0.0, 0, 10, -1.

*Booleano (lógico)*

Informação que pode assumir apenas 2 valores: Verdadeiro (V) ou Falso (F).

*Caracter*

Um caractere pode ser uma letra (maiúscula ou minúscula), um ponto final, ponto de interrogação, colchetes, enfim, símbolos que normalmente encontramos num teclado de um computador. Para que o computador identifique um caractere, deve-se representá-lo entre aspas simples (apóstrofos). Ex: ‘C’, ‘b’, ‘ ‘, ‘5’, ‘%’

*Tipos derivados*

Os tipos de dados não primitivos, normalmente são os vetores, matrizes, classes, enumerações[[5]](#footnote-5) e conjuntos de caracteres, que costumam ser estruturas de dados mais complexas do que os tipos de dados primitivos. Elas derivam dos tipos de dados primitivos.

*String*

Não é um tipo de dados primitivo, mas sim derivado. Uma string é um conjunto de caracteres. Ex: “Aluno”, “Professor”, “Casa 4”. Portanto, uma palavra, uma frase ou um texto são representados por uma String.

A categorização de um tipo de dados em tipos primitivos ou não primitivos se dá em relação dos fatores a seguir:

1. Natureza
2. Tamanho
3. Representação
4. Imagem ou Faixa de representação

A **natureza** caracteriza o tipo representado, que pode ser, por exemplo, um caractere, um número inteiro, um número real ou uma cadeia de caracteres. O **tamanho** determina o tamanho (a quantidade) em bits necessário para armazenar os valores do tipo. A **representação** determina a forma como os bits armazenados devem ser interpretados pelo computador. A **Imagem** ou **faixa** **de representação** determina a faixa de valores[[6]](#footnote-6) válidos para o tipo. A expressão usada para identificar um tipo de dado é chamada de **especificador de tipo**.

1.2.3 O uso de funções na computação

Para que o conceito de funções na computação seja melhor assimilado, é necessário que você revise “Princípio da responsabilidade única”.

Agora que você já revisou, segue um exemplo do uso da responsabilidade única para melhor compreensão, utilizando os conceitos que já aprendemos até aqui:

Ao longo da história da programação *- e até hoje -* os programadores desenvolvem soluções que auxiliam o desenvolvimento de sistemas. Muitas dessas soluções tiveram sua origem para poupar o desenvolvimento de cálculos matemáticos[[7]](#footnote-7) mais complexos. Tais soluções tinham a responsabilidade de fornecer o resultado de expressões matemáticas sem que o programador precisasse escrever a solução por ele mesmo.

A disponibilização de funções desenvolvidas para resolverem problemas específicos, facilitou a programação de expressões matemáticas a serem utilizadas em qualquer projeto.

Uma função é a maneira de se escrever algoritmos especializados em resolver um problema específico utilizando uma linguagem de programação e permite que programadores a utilize sem precisar reescrever o algoritmo que está dentro dela. Uma função é um algoritmo criado para ser executado pelo computador. Para entender melhor, observe abaixo:

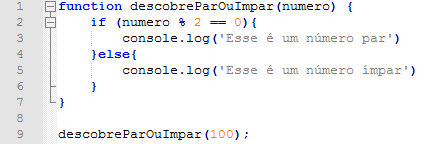
1. *Você precisa descobrir quais desses números a seguir são números pares: 425, 238, 200, 0*
2. *Sem recorrer a uma calculadora, você precisaria desenvolver um algoritmo para descobrir*
3. *Então, iríamos trazer para a leitura sistemática computacional uma expressão básica da matemática: A divisão.*
4. *Nesse caso, na matemática, seria simples: Dividir cada número por 2 e então, aqueles números cujo resto for igual a 0 serão considerados pares. E em um algoritmo, como poderíamos resolver?*

**Objetivo**: Descobrir números pares

**Tarefa**: Dividir um número por 2

**Problema**: Como descobrir o resto de uma divisão

**Algoritmo**: Descobre par ou ímpar



Na linha 9, ocorre a chamada da função passando o número 100 como parâmetro[[8]](#footnote-8)

Ao descobrirmos um problema, desenvolvemos o algoritmo com a responsabilidade única de informar se determinado número é ímpar ou par. Escrevemos essa responsabilidade dentro de uma função computacional. Uma função computacional é a maneira de representar a responsabilidade única de um algoritmo utilizando uma leitura sistemática computacional. Ou seja, a função computacional encapsula a solução de um problema.

Ao utilizarmos uma função computacional, o programador não precisa tomar conhecimento do código escrito dentro dela, apenas pode utilizar a função quando quiser.

1.2.4 O uso de variáveis na computação

“Em computação, dado é o que pode ser representado, armazenado e manipulado”. Essa menção refere-se ao conceito de dado em computação aprendido quando estudamos tipos de dados.

No nosso algoritmo “Sacar dinheiro” a gente pode perceber alguns dados (entradas e saídas) que são fundamentais para o processamento das instruções de cada tarefa do mundo real representada no algoritmo. Vamos a alguns exemplos deles:

Em “Colocar o cartão”, desenvolvemos um algoritmo que comparou o saldo com a quantia de dinheiro desejada para o saque. Nessa tarefa no mundo real nós temos 2 componentes importantes que serão utilizados para definir se o objetivo de “sacar dinheiro” será alcançado ou não: O saldo e a quantia.

No nosso algoritmo[[9]](#footnote-9), nós representamos - *em uma linguagem de programação -* esses dados da vida real e conseguimos fazer a comparação. Primeiro foi necessário declará-los no algoritmo, para que depois se tornasse possível atribuir um valor de “saldo” e “saque” e então manipular esses valores posteriormente. Tanto “valor” quanto “saque” foram utilizados para entrada, manipulados nos processos de transformação e exibidos em uma saída.

1.2.5 Acesso a variáveis na computação

Você já deve ter percebido que uma variável representa um dado do mundo real e esse dado, na computação, para ser manipulado, precisa, primeiramente, estar acessível.

Quando a tarefa 2 “Digitar a senha”, do nosso algoritmo principal, for executada pelo computador presente dentro do caixa eletrônico, ele receberá como entrada os caracteres da senha digitada para comparar com a senha cadastrada no cartão. Isso significa que a o dado “senha digitada” foi comparado ao dado “senha cadastrada” e, se as senhas coincidirem, gerar-se-ia uma saída válida que permitisse prosseguir para tarefa seguinte, “solicitar o saldo”.

Então, a associação de variáveis a entradas e saídas de um algoritmo está correta. Toda entrada e saída de um algoritmo se dá através de variáveis que são declaradas dentro de um algoritmo e precisam ser acessíveis na hora do processo de transformação das entradas em saídas válidas.

1.2.6 Identificando o escopo de acessos às variáveis

Ainda observando a tarefa 2 “Digitar a senha”, perceba que um problema precisa ser solucionado: A comparação da senha digitada com a senha cadastrada. Para isso, um algoritmo precisa receber uma entrada e gerar uma saída. O problema que esse algoritmo irá resolver é simples, ele deverá ter a responsabilidade única de comparar a senha digitada com a senha cadastrada no cartão. Os únicos dados a serem acessíveis por esse algoritmo são: “Senha digitada” e “Senha cadastrada”. Reflita e veja se esse algoritmo precisa acessar o dado “Saldo” para alguma coisa. Volte e observe as tarefas do nosso algoritmo principal e pense um pouco.

O princípio da responsabilidade única fará com que variáveis específicas sejam criadas para serem utilizadas como entrada para algoritmos que irão resolver problemas específicos. Baseado nessa lógica, o escopo de variáveis irá trazer o benefício de só acessar as variáveis necessárias para a solução do algoritmo, o que trará mais agilidade na execução[[10]](#footnote-10) e clareza de código.

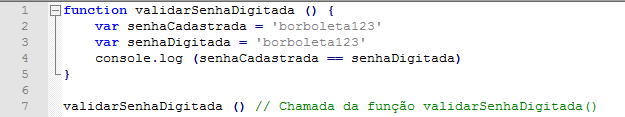
Para ficar claro, abaixo nós temos 2 algoritmos com responsabilidades (funções) totalmente distintas: O primeiro verifica se a senha que o usuário digitou no caixa eletrônico é igual a senha cadastrada para o cartão. O segundo, verifica se é possível ser sacada a quantia que o usuário deseja. Observe:

**Objetivo**: Assegurar que a senha digitada seja a mesma senha que o cliente cadastrou no cartão

**Tarefa**: Comparar senha digitada com a senha cadastrada no cartão

**Problema**: Como comparar duas senhas?

**Algoritmo**: Validar senha digitada

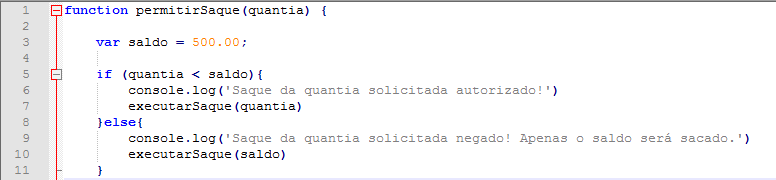


**Objetivo**: Permitir que o cliente saque somente a quantia que possuir

**Tarefa**: Comparar a quantia do saque com o saldo da conta

**Problema**: Como garantir que o saque seja menor ou igual ao saldo da conta?

**Algoritmo**: Permitir saque



Podemos observar algoritmos totalmente diferentes. Se executarmos os 2 ao mesmo tempo, um algoritmo não conseguirá acessar os dados do outro, justamente porque os dados estão encapsulados dentro de seu próprio algoritmo, dentro da *function* (função).

Um experimento a se fazer agora seria tentar acessar “senhaDigitada” dentro do algoritmo validarSaque(). O resultado desse teste resultaria em erro[[11]](#footnote-11) justamente porque os algoritmos cumprem papéis diferentes e as variáveis de cada um foram declaradas somente no escopo local da função.

1.2.7 Escopo local e escopo global

Acabamos de conhecer que o escopo[[12]](#footnote-12) de uma variável é o que define o local de acesso a seu valor. Quando desejamos proteger o valor de uma variável e deixá-la acessível somente dentro de uma função, utilizaremos o escopo local. Quando há a necessidade de compartilhá-la[[13]](#footnote-13) para que outras funções a utilizem como entrada, é fundamental que se utilize o escopo global.

1.2.8 Variável que não varia

A idéia que o nome “variável” sugere refere-se justamente ao processo de transformação que uma variável poderá sofrer dentro de um algoritmo. Temos visto vários exemplos de variáveis em nossos exemplos, mas até agora não identificamos nenhum caso em que uma entrada em um algoritmo não sofra alterações. Por isso, vamos retornar ao nosso algoritmo principal, tarefa 2*.*

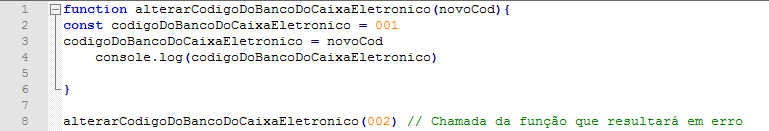
Em “colocar o cartão” espera-se que seja validado o código do banco do cartão ao ser comparado com o código do banco do caixa eletrônico. Esse algoritmo torna-se necessário para resolver o problema que poderia ocorrer caso o usuário, por engano, pegue um cartão de um banco errado. Dessa maneira, seria fundamental que o algoritmo não permita que o código do banco do caixa eletrônico sofra alteração. Para casos como esses em que não se permite alteração no valor de uma variável, utiliza-se uma *constante,* que sugere justamente o oposto de “variável”: Não varia!

**Objetivo**: Hackear o código do banco cadastrado no caixa eletrônico

**Tarefa**: Alterar o código do banco cadastrado

**Problema**: Substituir o código do banco cadastrado pelo código inserido pelo hacker

**Algoritmo**: Alterar código do banco do caixa eletrônico



Subunidade 3 - Operadores

O computador é uma ferramenta tecnológica que foi inventada inicialmente para realizar cálculos matemáticos complexos realizando tarefas repetitivas e mais seguras contra erros. Essas operações matemáticas realizadas na vida real foram representadas na leitura sistemática computacional para executarem algoritmos que já aprendemos a construir nessa disciplina. Nesta unidade iremos aprender como a matemática pode ser interpretada e executada em nossos algoritmos. Alguns dos operadores que serão exibidos aqui já apareceram em algum algoritmo já realizado por nós, mas agora iremos nos aprofundar nos 3 tipos de operadores existentes na computação: Operadores aritméticos, relacionais e lógicos.

1.3.1 Operadores aritméticos

São os operadores mais primitivos e conhecidos da matemática:

**

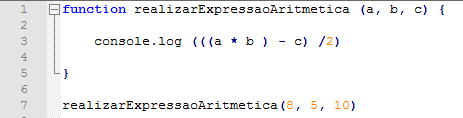
Tal como na matemática, a computação também prevê prioridade entre os operadores no momento da execução de cálculos matemáticos. A prioridade da matemática foi mantida. Dentro da mesma prioridade as operações são executadas da esquerda para a direita. Para alterar a prioridade, deve-se utilizar parênteses para que a expressão ganhe prioridade.

**Objetivo**: Realizar uma expressão aritmética em um computador

**Tarefa**: Criar um algoritmo que realize uma expressão aritmética

**Problema**: Receber 3 números como parâmetro e utilizar operadores aritméticos em uma expressão aritmética

**Algoritmo**: Realizar expressão aritmética

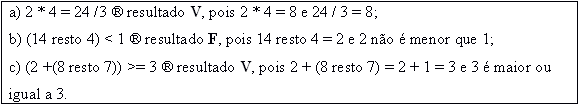


Operadores relacionais

Operadores relacionais são sinais utilizados para realizar comparações entre dois valores de mesmo tipo. Os valores comparados podem ser constantes, variáveis ou expressões aritméticas.



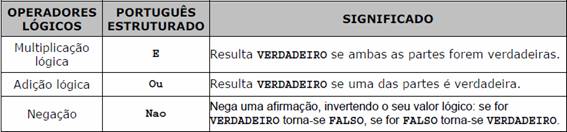
O resultado obtido sempre será um valor lógico. Exemplo: Operação *a+b = c*. O resultado será verdadeiro se o valor da expressão aritmética *a + b* for igual[[14]](#footnote-14) ao conteúdo da variável *c*. Caso o resultado da expressão *a + b* seja diferente de *c*, o resultado será falso.



Operadores lógicos

A álgebra booleana é uma ferramenta básica para construção de sistemas lógicos e serve como base para a operação de circuitos computacionais. Na álgebra booleana são considerados apenas 2 números, o zero (0) e o um (1).

As operações realizadas na álgebra booleana são operações que que utilizam operadores lógicos e trabalham com valores booleanos, com o objetivo de avaliar expressões cujo valor pode ser Verdadeiro (1) ou Falso (0), ou seja, implementando a lógica booleana.

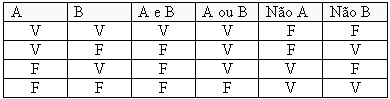


Uma expressão de multiplicação lógica (*E*) é representada pelo operador lógico **^**, já uma expressão de adição lógica (*OU*) é representada pelo operador lógico **+**.

É possível unir o operador lógico *NÃO* antes do operador relacional *IGUAL*, sugerindo que um elemento seja DIFERENTE de outro.

1.3.2 Tabela verdade

Demonstra os resultados da aplicação de operadores lógicos em expressões, conforme o valor dos operandos envolvidos:



Uma expressão de negação[[15]](#footnote-15) também é representada por **—** (barra horizontal sobre o termo).

Unidade 2 - Controle de fluxo e repetição



Aprenderemos nesta unidade o conceito de tomada de decisão que será desenvolvido em um algoritmo através do controle de fluxo e operação.

Subunidade 1 - Estruturas de desvio de fluxo

O fluxo de um algoritmo é executado conforme a sequência de instruções é desenvolvida. No item 1.1.1 onde foram descritas as tarefas necessárias para “Sacar dinheiro” a ordem em que as tarefas aparecem define o fluxo de execução para que o objetivo seja atingido. As tarefas não poderiam ser invertidas, porque a tarefa seguinte precisa da conclusão da tarefa anterior. Não é possível “solicitar o saldo” sem “digitar a senha” antes.

O controle de fluxo é o conceito em programação que irá definir a ordem de execução de cada instrução em um algoritmo. Sem a definição de um controle de fluxo, as instruções serão executadas conforme forem escritas no algoritmo.

O algoritmo “Validar cartão” , estudado na unidade anterior, define uma estrutura de desvio de fluxo que realiza duas ações possíveis dada uma condição. Observe o trecho desse algoritmo logo abaixo:

**3.** COMPARAR o código do banco do cartão com o código do banco do caixa eletrônico

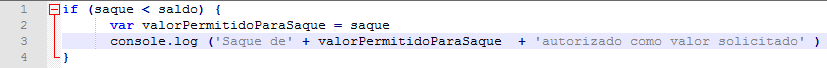
* 1. SE o código do banco do cartão for IGUAL ao código do banco proprietário do caixa eletrônico ENTÃO
     1. PERMITA PROSSEGUIR
  2. SENÃO
     1. NÃO PERMITA PROSSEGUIR

Podemos observar algumas definições que o trecho de algoritmo acima implementou para compor a estrutura de desvio de fluxo:

1. **Condicional** - É quando se define a lógica que precisará ser executada para atingir o próximo passo da execução. A lógica é executada desenvolvendo:
   1. O uso da palavra **Se[[16]](#footnote-16)** que precede a lógica de decisão a ser implementada
   2. O uso da palavra **Então** que sucede a lógica de decisão e precede a lógica de execução[[17]](#footnote-17)
   3. O uso da palavra **Senão** que precede a lógica de execução alternativa a ser satisfeita caso a lógica anterior não seja executada

2.1.1 Lógica de decisão

Utiliza operadores relacionais, na maioria das vezes[[18]](#footnote-18), para comparar variáveis, testar valores, dentre outras situações. É a implementação lógica fundamental de uma estrutura de desvio de fluxo. Sem uma lógica de decisão, não existiria nenhuma lógica alternativa.



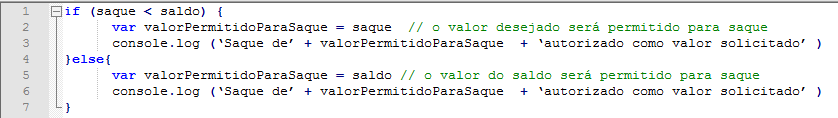
2.1.2 Lógica de execução

Parte do código que é executada quando a condição da lógica de decisão é satisfeita (retorna verdadeiro).

Ex1:



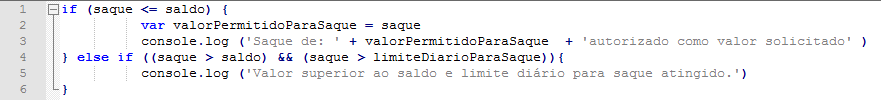
Ex2: Entre chaves { }



2.1.3 Lógica alternativa

É uma lógica de decisão. Porém, diz respeito a uma decisão alternativa. Uma condicional pode haver várias decisões alternativas. Se a lógica de decisão anterior não retornar um valor verdadeiro, a lógica de decisão alternativa é verificada.

Ex:



2.1.4 Switch Case

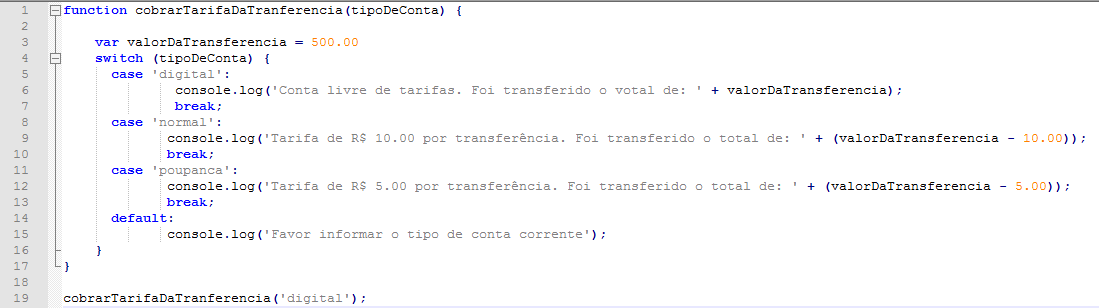
A utilização do *if* e *else* é mais adequada quando a lógica de decisão recebe como entrada valores desconhecidos pelo programador. O trecho de código da lógica decisão exposta nesta unidade do algoritmo “Validar cartão” não prevê um código de cartão específico a ser utilizado como entrada, pelo contrário, a condicional espera qualquer entrada de código a ser comparada com o código do banco. Por outro lado, quando há a previsibilidade das entradas, o Switch Case (controle de fluxo de seleção) torna-se mais adequado por criar uma estrutura que define os valores previstos para a entrada. Observe abaixo:

**Objetivo**: Estipular tarifas específicas de transferências de dinheiro entre contas correntes baseadas em um tipo de conta

**Tarefa**: Descontar o valor das tarifas das transferências

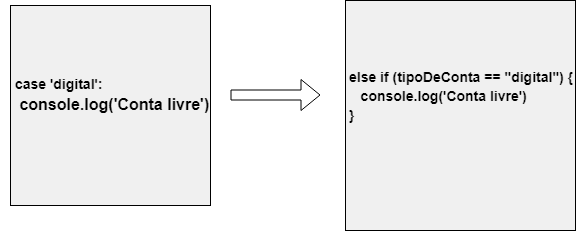
**Problema**: Como garantir descontar valores diferentes na transferência de dinheiro entre contas diferentes?

**Algoritmo**: Cobrar tarifa da transferência

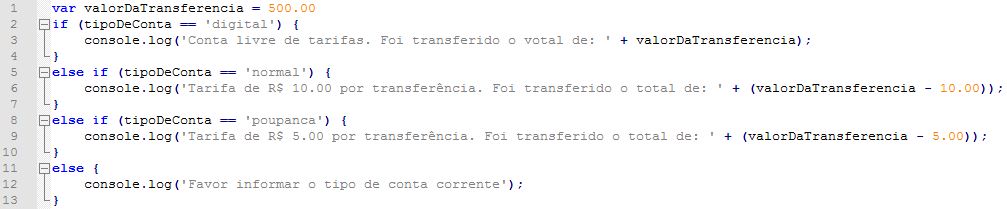


Perceba que todos os *case (*casos previsíveis) definem o que espera-se como entrada *- momento em que realizam a lógica de decisão-* e no interior dos *case*  é executada a lógica de execução se a lógica de decisão for satisfeita.

Você já deve ter percebido que a estrutura do Switch Case é concebida através da definição de várias lógicas condicionais alternativas, como se fossem vários *else if,* um no lugar de cada *case.* Observe abaixo:



Perceba abaixo a comparação entre as duas abordagens:



2.1.5 Diretivas *break, continue e default*

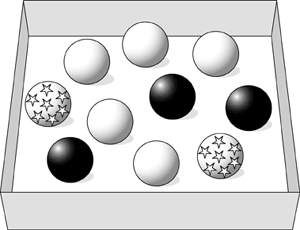
A execução de cada *case* é realizada em cascata, isto é, executando de cima para baixo.

A diretiva ***break***  precisa constar no fim da lógica de execução para que, se a lógica de decisão for satisfeita e a lógica de execução for resolvida, o fluxo de execução não prossiga para os *case* seguintes.

A diretiva ***continue*** seria utilizada para o oposto da diretiva break. Ela permite o fluxo para a lógica de decisão seguinte independente se a lógica de decisão em que ela esteja inserida seja satisfeita.

A diretiva ***default*** é utilizada se nenhuma das entradas previstas nos *case* anteriores forem recebidas. Então a lógica de execução de um *default* realiza um código padrão definido pelo programador.

Subunidade 2 - Estruturas de repetição (simples e condicional)

**

Uma estrutura de repetição (ou controle de fluxo de repetição), seja ela simples ou condicional, está ligada a alguma ação. Ação indica movimento e movimento sugere que um corpo deixa um lugar para ocupar outro lugar. Claro, esse não é um curso de física. Mas, a computação sem a matemática e a física não existiria.

2.2.1 Ação e movimento

Para utilizarmos a ilustração acima que exibe uma caixa de bolinhas de gude precisamos criar uma ação: Tirar bolinhas de gude da caixa. Essa ação, só é possível através de um movimento que resultará em ter menos bolinhas de gude na caixa e mais fora dela. Reflita sobre esse movimento.

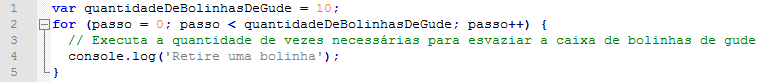
2.2.2 Repetição simples e condicional

Ainda utilizando a ilustração desta subunidade, podemos observar que o movimento sugerido na ação exemplificada anteriormente precisa de uma repetição contínua até que não haja mais bolinhas de gude na caixa. Para esse tipo de repetição, iremos utilizar a denominação *Repetição Simples.*

Alterando um pouco a ação de repetição simples, vamos acrescentar uma condição que não irá mudar a ação de tirar bolinhas de gude da caixa, apenas irá impor uma condicional que irá determinar quais tipos de bolinhas serão retiradas da caixa. Nesse caso, o objetivo não é simplesmente retirar bolinhas de gude, mas sim, retirar algumas bolinhas de gude. Se no primeiro objetivo o final da ação seria atingido quando não houvesse mais bolinhas na caixa, o segundo objetivo o final seria atingido quando não houvesse mais bolinhas de gude de “estrelinha”, por exemplo - *nesse ponto você já deve ter notado que as bolinhas de gude possuem características que podem diferenciar um grupo de outro.*

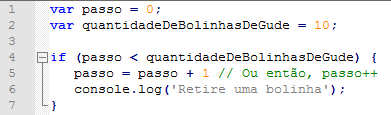
2.2.3 Estruturas de repetição do tipo: For

A palavra *for* define uma estrutura de controle de fluxo que implementa um laço de repetição. Ou seja, o objetivo de um laço de repetição é executar uma ação repetidas vezes. A ação de repetição simples mencionada acima poderia ser representada em um algoritmo da seguinte maneira:

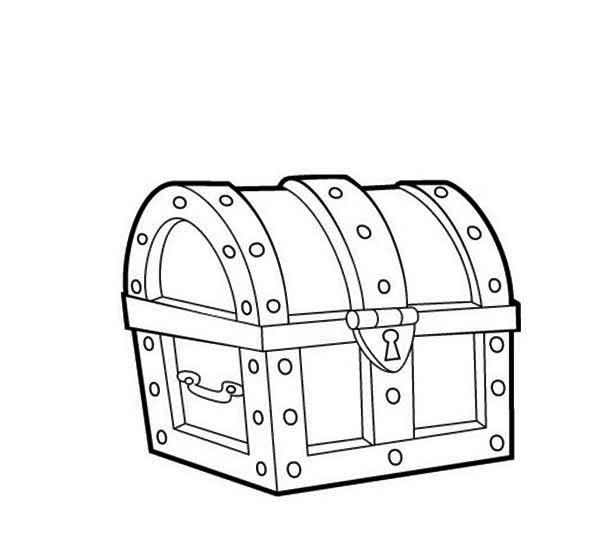


O que pode ser observado na estrutura[[19]](#footnote-19) do algoritmo acima é que um laço de repetição precisa sempre saber **como iniciar** e **como terminar**, ou seja, torna-se necessária a criação de uma variável de controle[[20]](#footnote-20) de repetições. Essa variável será testada - *em uma condicional -*  todas as vezes que houver repetição, pois a cada execução do laço ela irá servir como um contador que informa a quantidade de vezes executadas. Para o exemplo acima, a repetição do laço não poderá exceder a quantidade de bolinhas de gude na caixa, por esse motivo que só poderão ser realizadas 10 repetições já que só existem 10 bolinhas na caixa.

Todo laço de repetição implementa uma condicional que irá comparar o valor atual da variável de controle criada com o valor máximo de execuções que podem ser realizadas. Podemos reescrever[[21]](#footnote-21) o laço do algoritmo acima o exemplificando na estrutura de desvio de fluxo condicional abaixo:

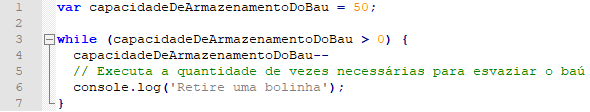


2.2.4 Estruturas de repetição do tipo: While e Do While

**

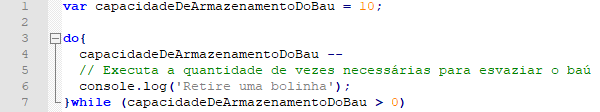
Avaliando um laço de repetição implementado com o uso do *for* percebemos que é sabido de antemão quantas vezes o laço será repetido. No caso da caixa de bolinhas de gude, poderá haver no máximo 10 repetições, uma para cada bolinha.

Supondo que ao invés de uma caixa, as bolinhas de gude fossem depositadas em um baú com a capacidade de armazenar centenas de bolinhas. Nesse caso, não seria razoável contar bolinha por bolinha. Por esse motivo seria mais prático o laço de repetição executar enquanto houver bolinhas no baú. Para casos como esse, um laço *while*torna-se mais adequado, pois dispensa o uso da variável de controle para determinar a quantidade de vezes em que o looping seria executado e passa a usar a idéia de verificar o estado do “todo” para saber se o looping chegou ao fim. Observe a transcrição do laço *for* já escrito, abaixo:



Algumas diferenças podemos observar nessa estrutura [[22]](#footnote-22) de repetição. Foi trocada a variável *quantidadeDeBolinhasDeGude* pela variável *capacidadeDeArmazenamentoDoBau*, justamente porque a referência que temos do “todo”, ou seja, da quantidade de vezes que precisaremos executar o laço de repetição, é bem maior quando se trata da capacidade de armazenamento de um baú.

A estrutura *do while* tem uma diferença importante da estrutura While: A lógica de execução é realizada antes da lógica de decisão. Observe abaixo:



Unidade 3 - Estruturas de dados

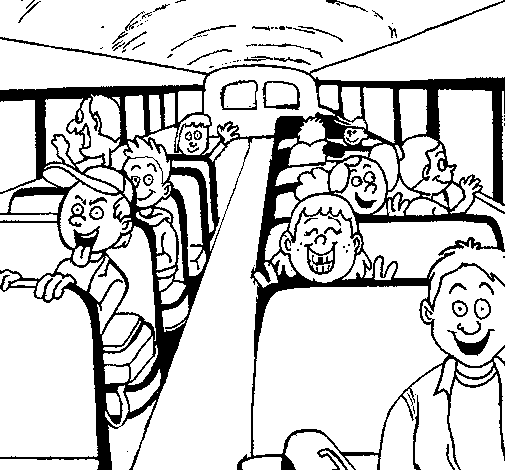
Aprendemos na unidade 2 sobre tipos de dados, que representa um dado da vida real, dentro de uma leitura sistemática computacional. Esse dado, na computação é armazenado juntamente com um valor dentro da memória do computador. Essa armazenagem de dados é realizada através de uma estrutura que permite a leitura e serviços de manipulação de dados em memória.

Subunidade 1 - Estruturas lineares

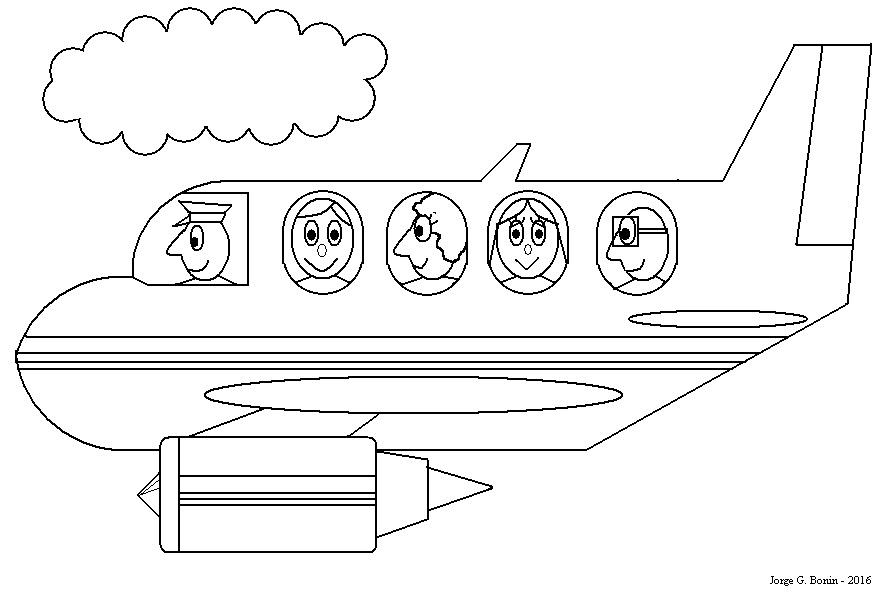
3.1.1 Array

Um Array é uma estrutura de dados na computação utilizada para armazenagem e manipulação de dados em memória de tal forma que cada um dos elementos armazenados possa ser facilmente identificado.

Um array pode ser observado como os assentos de um ônibus. Assentos vagos podem ser comparados a espaços em memória vazios e os assentos ocupados, podem ser comparados a espaços em memórias que possuem dados armazenados. Quando um passageiro novo entra no ônibus ele ocupa um espaço anteriormente vazio. Da mesma forma, quando um novo dado é armazenado, passa a ocupar um espaço em memória vazio.



Arrays são mais comumente criados para armazenagem de dados de mesmo tipo e o acesso a cada dado inserido é realizado através da posição que ocupa dentro da estrutura. Como ocorre em um avião, quando localiza-se um passageiro através de sua poltrona.

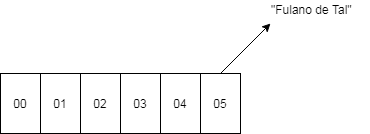


Ex: Passageiro “Fulano de Tal” ocupa poltrona “6A”.

No exemplo acima, é necessário conhecer a posição da poltrona para se obter o nome do passageiro. Da mesma forma, os itens individuais de um array são encontrados através de seus índices que utilizam uma sequência de números inteiros.

Ex1: Array aviao[6] - Aqui estamos criando um array de nome “aviao” que possui apenas 6 lugares.

Ex2:



A estrutura de armazenamento utilizada no exemplo acima é um array e demonstra bem uma importante limitação: Tamanho fixo. Ou seja, se for necessário a inserção de um novo elemento nesse array, alguma das 6 posições precisaria estar vazia.

A forma de acesso a um elemento em um array é direta, ou seja, para se obter o valor de um elemento é preciso conhecer previamente o seu posicionamento no array através de seu índice. No exemplo acima o índice 05 armazena a string “Fulano de Tal”.

Estruturas lineares são estruturas criadas para armazenar dados de forma linear (sequencial).

A manipulação de dados é realizada de forma ordenada, utilizando como informação o endereço do dado armazenado na sequência (ordem) em que foi inserido.

Estruturas de dados mantêm seus itens de informação de forma independente de seus valores. A única informação utilizada pela estrutura é a posição do item.

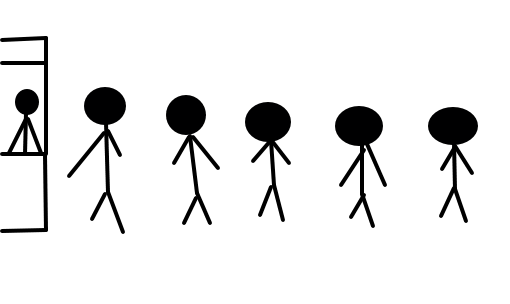
Um Array é estático, ou seja, não há como modificar sua dimensão após a criação. No momento em que a estrutura é criada em memória, é especificada a quantidade de espaço necessária para a alocação. Por esse motivo, se um elemento do meio da estrutura for retirado dela, seu lugar na estrutura permanecerá definido, só que vazio. O exemplo do ônibus citado acima, anteriormente comparado a um array, mostra exatamente essa limitação. Se um passageiro desce do ônibus, o espaço não é removido do ônibus, ele apenas se torna vago. Um ônibus, assim como um array, é uma estrutura de dados linear por disponibilizar uma quantidade específica de espaço livre para alocação.

3.1.2 Implementações de estruturas lineares

Uma estrutura de dados linear define um armazenamento ordenado em posições contíguas em memória o que torna altamente eficiente as operações de armazenagem e remoção de dados pela previsibilidade dada pela alocação sequencial de dados. As implementações de estruturas lineares podem se dar através do uso de arranjos ou ponteiros.

A estrutura de dados *Array*, estudada no item anterior é um exemplo de uma estrutura de dados desenvolvida exclusivamente para o uso de arranjos. As demais estruturas a serem estudadas abaixo, são estruturas que são implementadas mais comumente pelas linguagens de programação através do uso de ponteiros, o que as torna mais flexíveis quanto ao tamanho da estrutura, pois cada elemento da estrutura aponta para outro elemento aumentando ou diminuindo dinamicamente seu tamanho. É importante frisar que o uso de arranjos em filas, pilhas, listas encadeadas e duplamente encadeadas é opcional. Cada estrutura, com sua particularidade, implementará operações específicas para gerência de dados. Acompanhe abaixo o estudo de cada uma.

3.1.3 Operações em fila

**

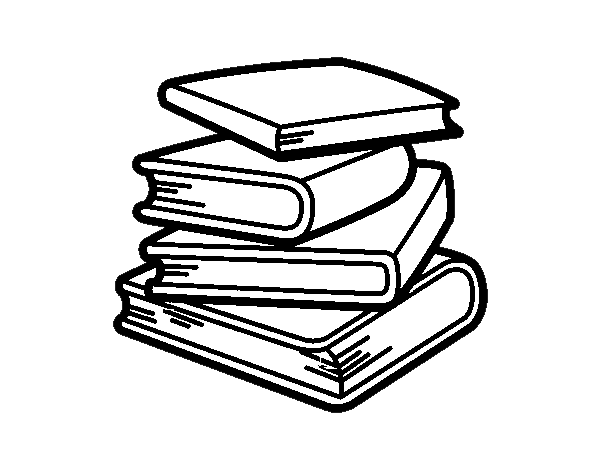
Uma estrutura de armazenamento em fila é linear, exatamente como nos sugere uma fila do mundo real. Os dados são armazenados na fila criando uma ordenação em que o primeiro a entrar na fila será o primeiro a ser removido.

Em uma fila, por padrão, toda manipulação de remoção de dados é realizada pelo início da fila (cabeça) e inserção de dados realizada pelo fim da fila (cauda). Com isso, o último elemento da fila deve esperar até que todos os elementos a sua frente sejam removidos da fila.

As operações utilizadas para manipulação de dados em uma lista podem ser:

* Criar uma fila vazia
* Enfileirar o item no final da fila
* Desenfileirar, retorna o item do início da fila e o retira da estrutura
* Verificar se a fila está vazia

3.1.4 Operações em pilha

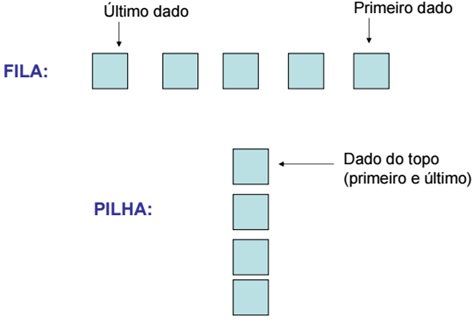
**

Uma estrutura de dados em pilha sugere que o último dado a ser inserido será o primeiro a ser removido da pilha, pois todas as inserções e remoções são feitas pelo topo da pilha.

As operações utilizadas para manipulação de dados em uma pilha podem ser:

* Criar uma pilha vazia
* Verifica se a pilha está vazia
* Empilha elemento (push)
* Desempilha elemento (pop)
* Verificar o tamanho atual da pilha

Um exemplo da diferença de manipulação de dados entre fila e pilha:



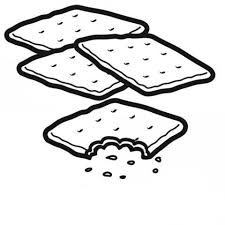
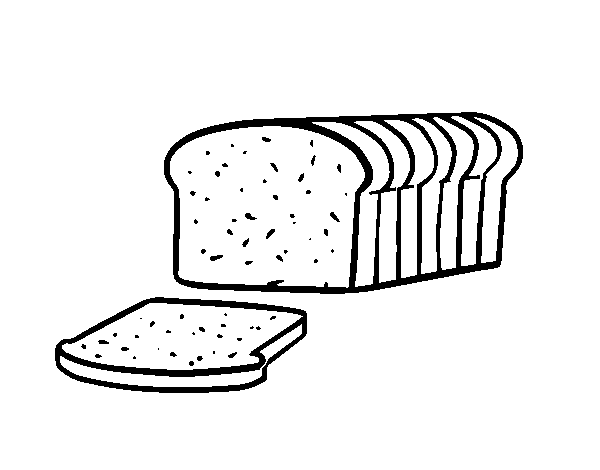
3.1.5 Manipulação de dados sob demanda

Vimos que uma importante característica das estruturas de dados lineares é a alocação contígua de dados em memória e, que a implementação de arranjos traria o prejuízo de estipular um tamanho fixo de elementos a serem armazenados e que o uso de ponteiros trás dinamismo de tamanho para a inserção e remoção de elementos da estrutura.

A previsibilidade de uma estrutura linear trás o benefício de agilizar a manipulação de dados por tornar claro como os elementos entram e saem da estrutura seja por especificar a quantidade exata de elementos armazenados no caso de arranjos ou por criar uma política de inserção e remoção de elementos específica de forma ordenada.

A manipulação de dados sob demanda é uma importante necessidade a ser levada em conta na hora do desenvolvimento de softwares modernos que recebem cada vez mais uma grande quantidade de interações de diferentes usuários ao mesmo tempo, consumindo bastante recursos de máquina. Acompanhe um breve estudo sobre alocação estática de memória a seguir.

3.1.6 Alocação estática de dados em memória



Imagine que, em um supermercado, você tem apenas 3 itens (pão, leite, biscoito) para ensacar. Você possui 3 sacos e decide categorizá-los (1 saco para pães, 1 saco para laticínios, 1 saco para biscoitos). Os sacos são grandes e caberiam pelo menos 5 itens, mas infelizmente você os categorizou para itens específicos. Podemos perceber então um desperdício de espaços não utilizados em cada sacola.

Em uma estrutura linear utilizando arranjos, como bem vimos, não podemos alocar mais dados do que a quantidade definida na criação da estrutura, além disso, é definida uma sequência rígida de alocação e manipulação de dados de mesmo tipo cujas alocações são feitas de forma estática (um dado do lado do outro) em memória e trazem o malefício de desperdiçar espaços vazios em memória se todo o espaço definido previamente não for utilizado. Em contrapartida, estruturas que conseguem dinamizar a alocação de dados em memória tem como limitação apenas a quantidade de espaço livre em memória, ou seja, por não definirem a quantidade de dados a serem alocados de forma prévia, quanto maior o espaço de armazenamento em memória, maior poderá ser a estrutura de dados.

Essa alternativa ao uso de arranjos se dá através do uso de listas encadeadas ou duplamente encadeadas.

3.1.7 Listas encadeadas

Estruturas lineares apresentadas até aqui são listas por definição, pois implementam sua estrutura com tipos de dados bem definidos e de forma sequencial, porém cada estrutura, como bem vimos, implementa sua própria estratégia para manipulação de dados através de operações.

A utilização de Listas Encadeadas em estruturas lineares é uma importante técnica[[23]](#footnote-23) para dinamizar a alocação em memória através do uso de ponteiros, pois permitem que cada estrutura aloque dados sem seguir a rigidez de ocupar um espaço pré definido em memória. Ocupando e desocupando espaço quando for necessário conforme a estrutura aumenta ou diminui.

Uma alocação dinâmica em memória propõe uma ordem virtual dos dados - *e não mais física -* , apenas indicando para cada dado qual o próximo elemento da lista. Essa implementação só é possível porque cada dado é rotulado para ser mais facilmente encontrado pelo seu antecessor.

Cada dado em uma lista encadeada é chamado de nó e carrega consigo a informação além de um ponteiro que indica o próximo na lista. O último nó de uma lista encadeada aponta para NULL (endereço do próximo nó é nulo).



As vantagens de uma lista linear são:

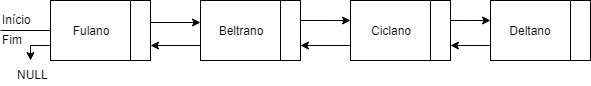
* A inserção ou remoção de um elemento na lista não implica a mudança de lugar de outros elementos
* Não é necessário definir, no momento da criação da lista, o número máximo de elementos que esta poderá ter. Ou seja, é possível alocar memória "dinamicamente", apenas para o número de nós necessários.
* Quanto maior for o espaço para o armazenamento físico, maior será o tamanho da lista

Já as desvantagens, são:

* A manipulação torna-se mais "perigosa" uma vez que, se o encadeamento (ligação) entre elementos da lista for mal feito, toda a lista pode ser perdida
* Para aceder ao elemento na posição *n* da lista, deve-se percorrer os *n - 1* anteriores

3.1.8 Listas duplamente encadeadas

Como evolução das listas encadeadas simples, as listas duplamente encadeadas apontam não só para o nó seguinte, mas também, para o nó anterior.



Subunidade 2 - Estruturas não lineares

Estruturas de dados não lineares propõem uma metodologia diferente de estrutura para armazenamento de dados em memória.

Até aqui, nós pudemos observar a linearidade física da implementação de arranjo e a linearidade virtual com a utilização de ponteiros em estruturas lineares.

Uma estrutura não linear prevê um conjunto de dados finitos e de mesmo tipo. A finitude dos dados pode nos levar a crer que torna uma pesquisa nesse conjunto uma tarefa fácil, na realidade, a finitude pode ser impossível de parecer finita para o ser humano. Por exemplo, todos nós sabemos que nosso país possui um número limitado de ruas e bairros, certo? É humanamente possível lembrarmos a localização de todas as ruas de nosso país? - *Já seria humanamente impossível decorar todas as ruas de nossa cidade, imagine decorar as ruas de nosso país.*

Para essa tarefa de pesquisa, foi definido que toda rua receberia uma identificação única que tornaria uma consulta a uma rua uma tarefa mais rápida e prática. Essa identificação seria um CEP, por exemplo, ou então um número que represente a rua geograficamente no espaço como a latitude e longitude. Ex: Av. Nossa Senhora de Copacabana - CEP: 22010-121 e latitude -22,9666 longitude -43,1796

Para se buscar dentre todas as ruas do país ou do mundo, a pesquisa por “Av. Nossa Senhora de Copacabana” seria extremamente custosa, até mesmo para um computador. Por essa razão, para facilitar as operações de busca em um conjunto muito grande de dados, um localizador único torna-se indispensável.

Dado que um conjunto é um aglomerado de elementos que são únicos e que, para isso, precisam obter uma identificação única, a finalidade de poder consultar determinado que item se encontra nesse conjunto torna-se computacionalmente eficiente.

3.2.1 Problema do vocabulário

Um vocabulário é formado por um conjunto de palavras usadas na língua portuguesa.

Um requisito sobre as palavras que formam um vocabulário é que cada uma seja única. Não tem sentido existir palavras repetidas em um vocabulário. Por outro lado, não há a necessidade de as palavras ficarem em uma determinada sequência. Se as palavras estiverem espalhadas sem estar numa sequência específica, elas ainda podem formar o vocabulário.

Uma tarefa difícil seria inserir ou remover uma nova palavra no vocabulário sem ter o custo de percorrer todo o vocabulário para não permitir inserção de palavras já existentes. Esse tipo de operação deveria de ser rápida para ser eficaz. E para isso, algum tipo de organização especial precisa ser realizado para se obter uma boa performance.

Uma boa opção seria agrupar palavras em estruturas de dados organizadas em conjuntos.

3.2.2 Conjuntos

As operações de um Conjunto são basicamente estas:

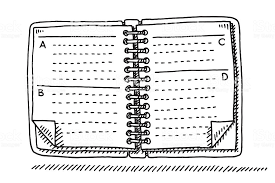
* Adicionar um dado elemento
* Verificar se dado elemento está ou não no Conjunto
* Recuperar todos os elementos
* Informar o número de elementos

Essas operações não são muito diferentes das que podem ser realizadas com as listas que já conhecemos, porém, para se realizar operações de busca em listas é preciso percorrer toda a lista, o que torna o processo bem ineficiente para o caso de listas muito grandes (vocabulário). O consumo de tempo destinado para uma operação de busca em listas é linear em relação ao número de elementos que ela possui.

Ao invés de percorrer todos os elementos, a alternativa ideal seria restringir o espaço de busca, eliminando o maior número possível de elementos sem que seja preciso “olhar” para eles.

Diferentemente das Listas, Pilhas e Filas, os Conjuntos não precisam manter os elementos em sequência. Então tentaremos uma abordagem bem diferente aqui. Vamos armazenar os elementos do Conjunto espalhados! Isso mesmo, espalhados, mas não aleatoriamente e sim com uma certa lógica que facilitará buscá-los.

A ideia é separar os elementos em categorias de acordo com certas características chaves do próprio elemento. Desta forma, para buscar um elemento, basta verificar qual é a categoria dele e assim eliminar todos os outros que estão nas outras categorias. Parecido como uma agenda de telefones.



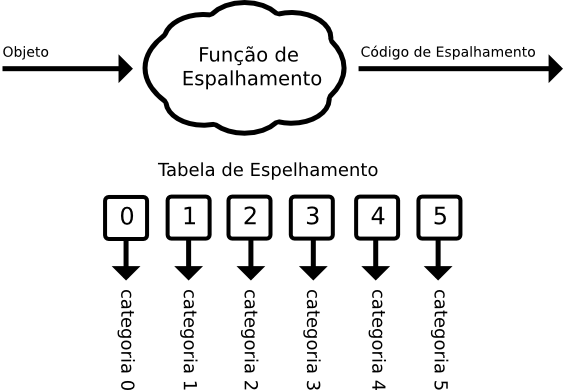
Nas agendas de telefones, os contatos são separados por categorias, normalmente, o que define a categoria do contato é a primeira letra do nome dele. Há uma página para cada letra do alfabeto. Desta forma, quando é preciso buscar um contato, vamos direto para a página referente a primeira letra do nome dele e assim eliminamos todos os contatos das outras páginas.

A técnica que estamos falando aqui é a técnica de *Espalhamento*. Ela consiste basicamente de duas partes: a *Função de Espalhamento* ("Função de Hash") e a *Tabela de Espalhamento* ("Tabela de Hash").

A Função de Espalhamento deve descobrir qual é a categoria de um determinado elemento. Para isso, ela deve analisar as características chave do próprio elemento. Por exemplo, na agenda de telefone, a Função deveria ver qual é a primeira letra do nome do contato e devolver o número da página referente àquela letra.

A Tabela de Espalhamento deve armazenar as categorias e cada categoria deve ter um índice. Este índice, gerado com a Função de Espalhamento, é usado para recuperar a categoria rapidamente.

A Função deve ser determinística, ou seja, toda vez que aplicada a um mesmo elemento ela deve gerar o mesmo índice, caso contrário procuraríamos um nome na página errada da agenda.



3.2.3 Mapas

Uma estrutura de dados do tipo mapa, é um conjunto de dados armazenados que implementa para cada dado uma chave (índice) única. Como no exemplo do CEP da rua, essa implementação de mapa é muito utilizada quando se torna necessária uma pesquisa por um dado em um conjunto de dados muito grande.

As operações possíveis de serem realizadas em um mapa, são:

* Adicionar uma associação
* Pegar um valor dado uma chave.
* Remover uma associação dada uma chave
* Verificar se existe uma associação para uma determinada chave
* Informar a quantidade de associações.

Unidade 4 - Boas práticas de desenvolvimento de algoritmo

A filosofia de Boas Práticas de desenvolvimento de algoritmo foi criada por programadores ao longo do desenvolvimento das linguagens de programação e de diferentes tipos de projetos sugerindo a implementação de técnicas e conceitos que diminuem a carga de trabalhos repetitivos, a má compreensão de códigos e tornam as tarefas, desempenhadas por um programador, mais produtivas.

De certa forma, já tentamos introduzir alguns conceitos que se referem a uma boa prática de desenvolvimento de algoritmos, por exemplo quando nesta disciplina nos referimos ao “princípio da responsabilidade única”. Repare:

Ao solicitar um saque de uma determinada quantia em um caixa eletrônico a gente espera que o valor solicitado seja, finalmente, retirado do caixa. Certo? Mas até que o dinheiro saia do caixa, alguns passos são realizados *- tarefas do algoritmo “Sacar dinheiro”*. Mas a maior complexidade está em desenvolver algoritmos focados em resolver apenas um único problema.

Subunidade 4 - Boas práticas de desenvolvimento

4.1.1 Análise de algoritmos

Para “sentirmos na pele” a complexidade em se desenvolver algoritmos focados que cumpram com a finalidade de sua responsabilidade única, é sugerido abaixo a análise de 2 algoritmos que cumprem a mesma função, só que escritos de formas diferentes.

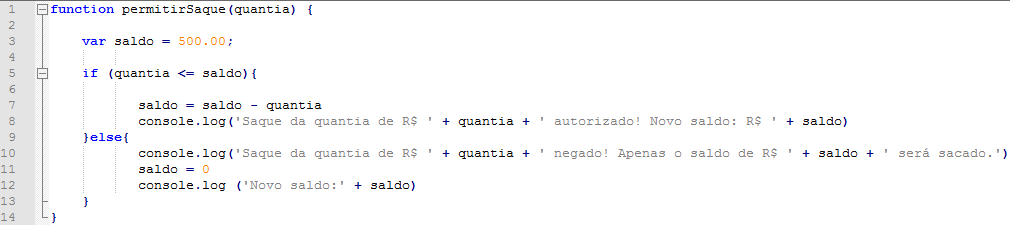
Para realizar o desafio de analisar algoritmos e interpretá-los, comente abaixo seu entendimento do algoritmo escrito a seguir:

Objetivo: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tarefa: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Problema: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Algoritmo: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



* ***Atenção!*** *Existe uma consideração importante a se fazer sobre a relação entre o nome desse algoritmo e sua responsabilidade única. Ela não será dita pelo instrutor até o fim desta disciplina. Identifique você mesmo e leve o questionamento para toda a classe antes que a consideração seja revelada. Quem sabe você não ganha um prêmio?*

Para esse desafio de analisar um algoritmo já escrito, você precisará observar as lógicas desenvolvidas e então interpretar os problemas que motivaram o programador a desenvolver esse algoritmo. Encontrando as lógicas você encontrará os problemas.

A primeira lógica que podemos observar é a comparação da quantia solicitada com o saldo disponível. Somente após a comparação dessas entradas (lógica de decisão) é que se torna possível executar as ações (lógica de execução) de aprovar o saque e executar o desconto da quantia aprovada do saldo.

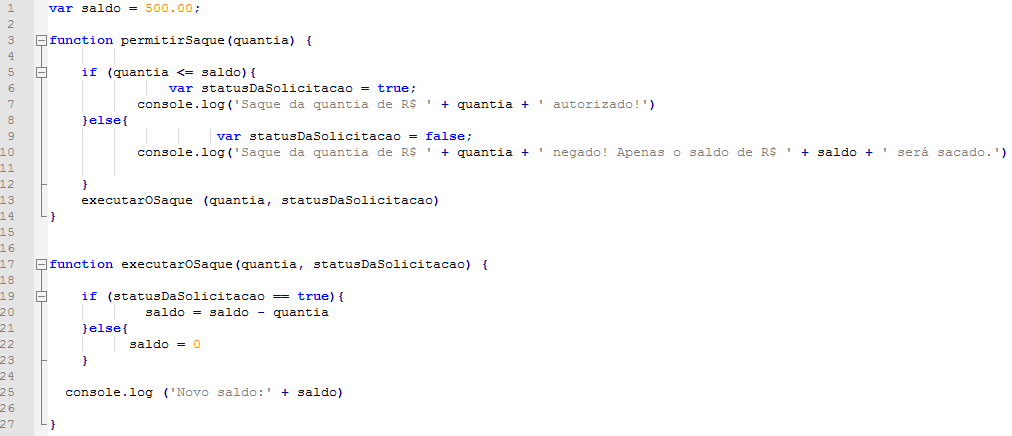
A segunda lógica consiste em não permitir o saque solicitado, pois o valor solicitado para saque é superior ao saldo disponível e então realiza o saque integral do saldo.

Ao analisarmos as lógicas podemos extrair 2 problemas, são eles:

1. Como permitir o saque de um valor solicitado
2. Como executar o saque do valor solicitado
   1. Saque de uma quantia menor ou igual ao saldo
   2. Saque de uma quantia maior que o saldo

Se 2 problemas puderam ser extraídos desse algoritmo, não é muito difícil dizer que ele não cumpre integralmente com a função de resolver apenas um único problema, ou seja, não cumpre com o Princípio da Responsabilidade Única.

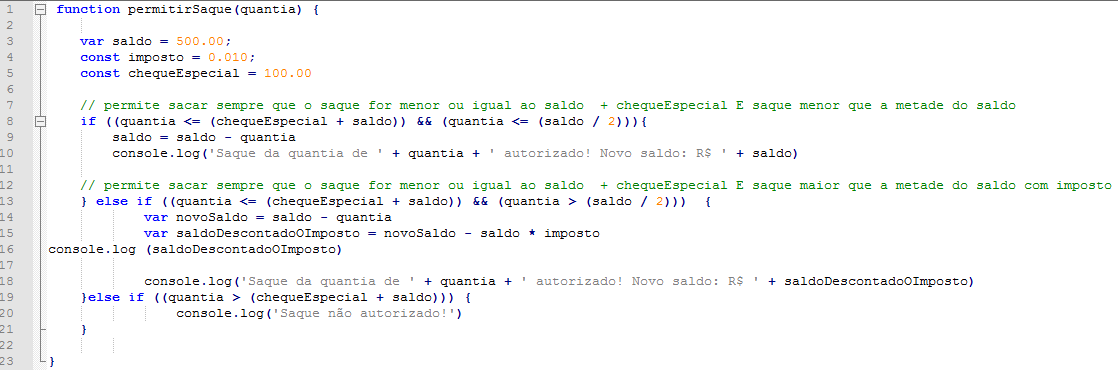
Dessa maneira, chegamos a conclusão de que ao invés de 1 algoritmo para 2 responsabilidades, precisaremos desenvolver no mínimo 2 algoritmos e cada um com 1 responsabilidade: Permitir o saque e executar o saque, respectivamente.



4.2.1 Princípio da responsabilidade única

Vamos supor que as regras de saque agora mudaram e para cada saque realizado partir de 50% do saldo, será cobrada uma taxa de 0.010% de imposto sobre o saldo inicial. Além disso, a conta corrente passará a possuir um limite de cheque especial no valor de R$ 100,00. Ou seja, as solicitações agora poderão ser maiores que o saldo até o limite do cheque especial (saldo negativo de R$ -100,00). Nem precisamos pensar muito que isso acrescenta mais dois problemas em nosso estudo de caso. Observe o desenvolvimento de um algoritmo para a solução dos novos requisitos acima:





*Esse algoritmo mereceu o selo “Go Horse Process”. Uma expressão informal que é utilizada entre nós programadores para nos referirmos ao desenvolvimento desleixado, ou sem pretensões em desenvolver boas práticas e certamente torna o algoritmo grande, difícil de ser entendido por outro programador e de ser executado pelo computador.*

*Um algoritmo que precisaria conhecer apenas o saldo e a quantia solicitada agora realiza cálculo de imposto, desconta o imposto do saldo, desconta o saque do saldo, não permite saldo inferior ao limite do cheque especial, faz seu imposto de renda, trás a pessoa amada em 3 dias, faz pipoca …*

4.3.1 Refatoração

*“Alteração feita na estrutura interna do software para torná-lo mais fácil de ser entendido e menos custoso de ser modificado sem alterar seu comportamento observável” - Martin Fowler*

Dado o algoritmo acima que foi desenvolvido para solucionar os novos requisitos solicitados, perceba na solução abaixo, a mudança na estrutura interna do algoritmo e compare o comportamento das duas soluções e julgue se pode ser observado alguma alteração perceptível pelo usuário.



* ***Atenção!*** *Existe uma consideração importante a se fazer sobre o comportamento das soluções desenvolvidas para os novos requisitos se comparado ao comportamento de uma conta corrente com limite de cheque especial definido. Como todo algoritmo representa um comportamento do mundo real, teste os casos dessa última solução e, se chegar a alguma consideração, explane para toda a sala de aula. Quem sabe você não ganha um prêmio se conseguir?*

Na primeira impressão você pode achar que, por criarmos um código com maior quantidade de linhas, a melhor solução seria a primeira. Mas, ledo engano. O código refatorado extraiu todas as lógicas de decisão do primeiro algoritmo e as encapsulou em funções (algoritmos) que cumprem, cada um, uma única responsabilidade. Dessa forma, você consegue testar facilmente lógica de decisão de cada um e, se precisar realizar alguma de regra no futuro, saberá mais facilmente onde alterar o código, pois cada função revela sua real intenção.

4.4.1 Clean Code

Até agora nessa disciplina, pudemos observar como um problema pode ser resolvido utilizando algoritmos escritos de diversas formas. Tanto a primeira quanto a segunda solução desenvolvida funcionam e cumprem o papel de resolver os novos requisitos, ou seja, o resultado final (comportamento) é o mesmo para as duas soluções e o cliente conseguirá sacar o dinheiro e ficará feliz com o resultado, pois o resultado não gerou erros que impedissem o sucesso do saque.

O *Clean Code* é a definição de código limpo que é desenvolvida pensando no programador, não no usuário. É muito comum um programador depois de desenvolvido um algoritmo *- 1 dia depois já é suficiente -* não fazer idéia do que significa um trecho de código que ele mesmo desenvolveu. Por esse motivo, a seguir serão listadas algumas dicas que deverão ser levadas em consideração na hora de desenvolver um algoritmo.

Observe a seguir:

var valor1 = 500.00;

var valor2 = 0.010;

Você já sabe que acima foram definidas duas variáveis que recebem valores distintos. Ok. Mas em qual contexto as duas variáveis foram definidas? No exemplo acima, em nenhum contexto. Sendo assim, qualquer operação poderá ser feita com essas duas variáveis, pois elas não representam entradas para nenhum problema específico. Se você quiser somar os dois valores, ok. Se você quiser dividir um valor pelo outro, ok também. Mas e se eu lhe disse que valor1 se refere ao saldo existente em uma conta corrente e valor2 representa um imposto a ser pago sobre o saldo, você mudaria sua forma de enxergar as duas variáveis? Reflita um pouco sobre isso.

Agora que possuímos um contexto definido, observe as alterações sugeridas pelo Clean Code:

var saldo = 500.00;

var imposto = 0.010;

Com o contexto definido e a declaração das variáveis com os nomes sugestivos a ele, é possível interpretar a operação que precisaria ser realizada, por exemplo:

saldo = saldo - (saldo \* imposto)

4.4.2 Quanto menos *else/if* melhor

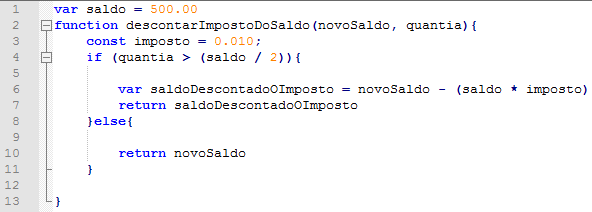
O *if ternário[[24]](#footnote-24)* já consiste em uma forma elegante de se escrever um código com menos linhas e tornar o algoritmo mais claro para ser interpretado, com ele uma estrutura condicional é representada assim:

quantia < saldo ? console.log (‘Saque autorizado’) : console.log (‘Saque não autorizado’)

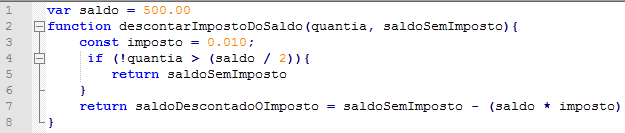
O *Clean Code* sugere, “ao pé da letra”, um código limpo. Isso significa uma menor quantidade de linhas utilizadas para desenvolver uma solução, uma indentação de código que facilite a compreensão, não repetição de códigos, dentre várias dicas que poderão ser estudadas no livro *Clean Code de Robert C. Martin.*

O Else, como já aprendemos nesta disciplina, representa uma lógica de execução alternativa. Dessa forma, o resultado da lógica de execução principal do algoritmo *- lógica que retorna o comportamento que condiz com o propósito do algoritmo -* deve ser retornada por padrão por um algoritmo e as lógicas alternativas (else) deverão ser encapsuladas em lógicas condicionais específicas. Observe o código a seguir e a refatoração seguindo a sugestão do Clean Code:

**Código a ser refatorado**



**Código refatorado**



No código refatorado podemos observar a ausência da condicional alternativa *Else* implementando uma técnica simples: Negar a lógica de decisão e retornar como valor padrão o que o propósito do algoritmo exige que seja retornado*.* Um algoritmo denominado “Descontar imposto do saldo” tem o propósito de retornar um saldo descontado o imposto, logo, espera-se como padrão que seja retornado *saldoDescontatoOImposto* e tudo o que não for padrão seja retornado em lógicas condicionais específicas, nesse caso, *saldoSemImposto*  é uma alternativa retornada quando não é preciso aplicar o imposto ao valor do saldo.

4.4.3 Código auto explicativo

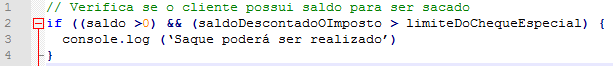
Assim como foi aprendido a dar nomes para variáveis que sejam compatíveis com o contexto do problema do algoritmo, nomes de funções e códigos sem repetição de lógicas dispensam que sejam realizados comentários para que o programador explique para outro programador o que foi desenvolvido. Por exemplo:



Mas, nem todos os comentários são dispensáveis.



Não se explique com comentários



Se explique com código



4.5.1 Teste em algoritmos

O *Clean Code* é uma filosofia de desenvolvimento criada para programadores. Como já foi visto, um algoritmo que implementa ou não alguma técnica de *Clean Code* é transparente para o usuário, desde que o algoritmo cumpra com seu papel de executar com perfeição os requisitos que o usuário deseja.

O teste de código não só traz segurança para o programador desenvolver outras funcionalidades ou corrigir bugs no mesmo algoritmo, quanto também para o usuário final, pois ao realizar um desenvolvimento orientado a testes, os requisitos são lidos pelo programador e testados antes mesmo que um código seja escrito.

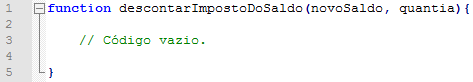
Em um desenvolvimento orientado a testes, vamos definir aqui alguns passos a serem rigorosamente seguidos:

1. Escrever um teste, sem mesmo ter escrito o código real a ser testado (Figure o que deseja testar).
2. Executar os testes e acompanhar a falha
3. Escrevemos a funcionalidade do sistema que iremos testar
4. Testar novamente, agora para passar (Se não passou algo saiu errado, faça novamente o passo 3)
5. Refatore sua funcionalidade e a escreva por completo
6. Passe para o próximo problema

Escrever primeiro o teste garante que o requisito solicitado seja desenvolvido simulando a realidade. Dessa maneira, há uma garantia de que a implementação do requisito foi criada com base no propósito do algoritmo que reflete o requisito solicitado.

Passo 1) Identificar o problema e declarar a função vazia

**Problema** (propósito): Retornar o valor do novo saldo descontado o imposto sobre o saldo se a quantia for maior que a metade do saldo



Passo 2) Descrever todos os testes possíveis com entradas simulando a realidade seguindo o modelo:

*dado a(s) entrada(s) \_\_\_\_\_\_\_\_\_ espero o retorno \_\_\_\_\_\_\_\_\_\_\_*

**Teste1:** Dado a(s) entrada(s) QUANTIA = 100.00 e NOVO SALDO = 400.00 espero o retorno de SALDO SEM IMPOSTO de 400.00



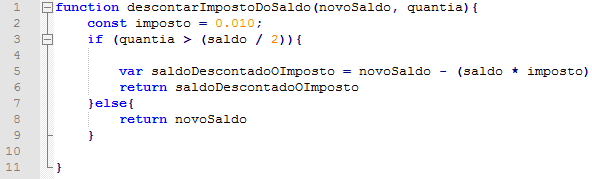
**Teste2:** Dado a(s) entrada(s) QUANTIA = 250 .00 e NOVO SALDO = 250.00 espero o retorno de SALDO SEM IMPOSTO de 250.00



**Teste3:** Dado a(s) entrada(s) QUANTIA = 260.00 e NOVO SALDO = 240.00 espero o retorno de SALDO SEM IMPOSTO de 235.00



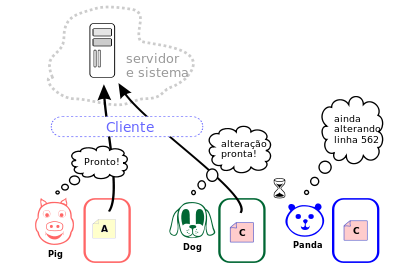
Passo 3) A execução de cada teste desenvolvido, primeiramente, não retorna valor algum. Dessa maneira, o primeiro passo (falhar o teste, pois não há código desenvolvido) será atingido. O passo a seguir será desenvolver a lógica para se obter o retorno que o teste espera. Uma vez desenvolvida a lógica, o teste precisará ser executado novamente até que o teste, consiga ser executado sem erros e com o retorno esperado. Segue abaixo o algoritmo solucionado com todas as lógicas necessárias para execução correta dos testes levantados.



Passo 4) A refatoração é o passo seguinte ao teste ter funcionado corretamente.

Passo 5) Após a refatoração, o programador poderá seguir para o desenvolvimento do próximo requisito.

4.6.1 Versionamento de código



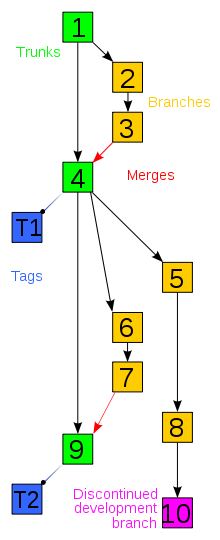
Ainda na filosofia de boas práticas, na verdade … Versionamento de código é a prática obrigatória para o bom programador controlar as versões de seus códigos. Impossível um projeto hoje em dia ser desenvolvido sem o auxílio de uma ferramenta de versionamento de código.

O versionamento de código funciona como a imagem acima sugere. Programadores diferentes possuem seu próprio local físico de trabalho, realizam suas alterações em arquivos locais (cliente) e então submetem essas alterações para um servidor central que irá gerenciar todas as alterações realizadas por qualquer programador.

Essa interação entre o código cliente (máquina local) e o código servidor (máquina servidor) se dá através de alguns passos importantes:

1. Um repositório (**no servidor** para salvar os arquivos versionados) precisa existir
2. O programador então cria um clone em seu ambiente de trabalho desse repositório para desenvolver seus arquivos localmente
3. Toda alteração que o programador fizer ocorre primeiramente em seu ambiente local
4. Toda vez que o programador concluir a correção de algum bug, ou desenvolver alguma nova funcionalidade ou refatorar algum código, etc … ele conclui a edição dos arquivos do trabalho e gera uma versão para esse trabalho.
5. Quando o programador desejar inserir suas alterações no servidor para que todos os outros programadores tenham acesso a elas, ele cria uma versão local e a submete para o servidor que se encarrega de integrar a versão recebida com a versão que já existia no servidor.

4.6.2 Ambiente de versionamento



No primeiro momento do versionamento, quando é realizado um *clone* do projeto principal - *no caso de projetos já existentes[[25]](#footnote-25) no servidor -* todos os arquivos existentes no servidor são copiados para a máquina local do programador, momento em que o trabalho local está idêntico[[26]](#footnote-26) ao trabalho existente no servidor.

Inicialmente, a versão do projeto que foi recebida do servidor é idêntica à versão que consta no servidor, criando então uma cópia de trabalho (*working copy*) sem nome sob a forma de um “tronco” (*trunk*).

O Trunk pode ser considerado como uma linha principal do projeto e cada alteração nessa linha principal surge uma versão mais moderna que é atribuída ao projeto local e posteriormente ao projeto principal (servidor). Se o programador está trabalhando exclusivamente no trunk, ele vai conter a versão mais atualizada e moderna do projeto local, porém geralmente pode ser a versão mais instável.

Uma nova abordagem seria então a criação de um “ramo” ou versão (*branch*) do *trunk* e implementar as mudanças neste branch e depois mesclar (*merge*) as alterações de volta no trunk assim que esse ramo estiver com uma versão estável do projeto.

Em um sistema de controle de versões, o *trunk* deve ser transparente para o programador, pois cada alteração realizada em seu trabalho local deve ocorrer em um *branch* e então mesclado com o *trunk* antes de o trabalho ser integrado ao projeto principal no servidor.

No ambiente local do programador, denomina-se convencionalmente como *master* o *branch* que possui a versão idêntica à versão do projeto no servidor . Esse *branch* contém, inicialmente, o *clone* do projeto principal e deve ser utilizado apenas para receber e enviar para o servidor o trabalho final do projeto desenvolvido pelo programador.

Também por convenção, denominamos um *branch* a ser criado como cópia do *branch* *master*: O *branch desenv*,que será utilizado pelo programador como o principal *branch* de desenvolvimento. Mesmo sendo o principal *branch* de desenvolvimento, essa convenção não impede [[27]](#footnote-27)que outros *branches* sejam criados a partir do *master* ou a partir do *desenv*.

No final das alterações desenvolvidas pelo programador, todos os branches criados, devem ser integrados ao branch principal de desenvolvimento (desenv) e então integrados com o branch master para que a versão do projeto final seja integrada ao projeto principal do servidor.

4.6.3 Histórico de envio

No momento em que um programador está editando um arquivo, ele o edita, necessariamente em um *branch*. Cada alteração em um ou mais arquivos é refletida diretamente no *branch* ativo do momento. Para que essas alterações sejam registradas no histórico de alterações do *branch*, é necessário que o programador realize um *commit* que pode conter uma mensagem para orientar os demais programadores a respeito da alteração realizada.

Cada *commit* gera um histórico de alteração que especifica exatamente o que o programador alterou no arquivo. Um *commit* pode fazer parte de um trabalho maior, que impõe que vários *commits* precisem ser realizados. Só então, ao término de todas as tarefas imputadas ao programador, uma versão nova será criada e integrada à linha de trabalho principal (*trunk*).

Quando um programador decide que suas tarefas foram concluídas e a versão do *branch* está pronta para ser integrada ao *trunk*, ele executa um procedimento *push* que se encarregará de integrar a versão do *branch* no *trunk*.

Conclusão

*"Sabemos o que somos, mas não sabemos o que poderemos ser"*

**Willian Shakespeare**

Chegamos ao final desta disciplina e a certeza que fica é de que nunca saberemos onde esses conhecimentos adquiridos poderão nos levar. Onde eles irão te levar não sabemos, mas, com certeza você está apto a responder às seguintes perguntas:

* Como criar um algoritmo?
* O que é a responsabilidade única de um algoritmo?
* Refatorar algoritmos usando boas práticas de programação.
* Desenvolver testes para um algoritmo.
* Versionar os códigos de um projeto.

Ainda restam dúvidas? Releia sua apostila, dê um "Google" ou leve sua dúvida para o responsável pelo galpão que o instrutor da disciplina irá sanar sua dúvida.

Grande abraço!

"*Ensinar não é transferir conhecimento, mas criar as possibilidades para a sua própria produção ou a sua construção*"

**Paulo Freire**

1. Uma saída válida é um resultado que permite que a próxima instrução seja iniciada. Exemplo: Em “Sacar dinheiro” , Tarefa 3, espera-se que uma senha válida seja digitada para que então possa ser comparada com a senha cadastrada no cartão. [↑](#footnote-ref-1)
2. Para uma melhor compreensão, vamos fazer uma distinção aqui: Toda vez que o termo “instrução” for lido, estamos nos referindo aos passos sistemáticos de um algoritmo específico. Toda vez que o termo “tarefa” for lido, estamos nos referindo aos passos no mundo real para atingirmos um objetivo específico. [↑](#footnote-ref-2)
3. É só lembrar de um espremedor de laranjas. Ele só sabe espremer laranjas. Tente espremer bananas em um espremedor de laranjas e valide o resultado. [↑](#footnote-ref-3)
4. Todos os algoritmos a partir de agora, enquanto estiverem sendo representados em uma linguagem de programação, serão escritos na linguagem nativa da web: Javascript.. Para maiores informações visite: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript> [↑](#footnote-ref-4)
5. Nomenclaturas que ainda serão vistas nesta disciplina, exceto “enumerações” que será tratada na disciplina “Linguagem de programação orientada a objetos” [↑](#footnote-ref-5)
6. Faixa de valores indica os limites de cada tipo de dado. Por exemplo, a faixa de um tipo int é de -2.147.483.648 até 2.147.483.647. Acima disso, o número já precisa ser declarado como o tipo long. [↑](#footnote-ref-6)
7. Em Javascript, a função sqrt() encontra a raiz quadrada de um número. Ex: sqrt(9) tem como resultado o número 9 [↑](#footnote-ref-7)
8. Um parâmetro é um valor que é atribuído a uma variável que será utilizada no interior da função. Mas nem toda função necessita de um parâmetro. Uma função que não recebe parâmetro irá declarar e atribuir um valor para a variável dentro da própria função ou está utilizando uma variável global (escopo de visualização de variável que ainda será visto nesta unidade). [↑](#footnote-ref-8)
9. A representação de um dado na leitura sistemática computacional dar-se-á através da declaração de uma variável. Em Javascript - *linguagem utilizada para executar os algoritmos dessa disciplina* - a declaração inicia-se com a palavra “var”. [↑](#footnote-ref-9)
10. Quanto mais variáveis criadas, mais espaço em memória será utilizado. Para um sistema computacional que utilize muitas variáveis que ocupem um grande espaço em memória, a execução do algoritmo poderá se tornar crítica. Para solucionar esse problema, nunca se esqueça do Princípio da Responsabilidade Única já estudado nesta disciplina. Sempre crie algoritmos específicos para solucionarem problemas específicos que receberão entradas específicas e retornarão saídas específicas. [↑](#footnote-ref-10)
11. *Uncaught ReferenceError: senhaDigitada is not defined at permitirSaque()*  [↑](#footnote-ref-11)
12. Fique atento à forma como as linguagens de programação implementam o escopo de uma variável. Por exemplo, no Javascript - após o ECMAScript 6 - a declaração de uma variável local dar-se-á através da palavra “let”. Já a declaração de uma variável global, dar-se-á através da palavra “var”. [↑](#footnote-ref-12)
13. O compartilhamento de variáveis é uma prática aceita na programação mas é preciso cuidado. Já que uma variável representa um dado no mundo real e é utilizada como entrada para um algoritmo que possui uma responsabilidade única definida, poucos são os casos em que se torna-se necessário o compartilhamento de variáveis. [↑](#footnote-ref-13)
14. É preciso fazer uma distinção importante aqui. O operador relacional que sugere igualdade é o *- já conhecido na matemática -* sinal de “igual” (=). Já utilizamos esse sinal nesta disciplina para declarar variáveis em um algoritmo. No momento da declaração de uma variável uma igualdade é realizada de forma diferente que a mostrada nesta unidade. Quando se declara uma variável, estamos atribuindo um valor a uma variável. Ou seja, uma variável “recebe” o valor especificado. Somente após essa atribuição é que a variável passaria a ser igual ao valor, ou seja, poderemos dizer que em *var a = 8;* a variável *a* passaria a ser *igual a 8* somente quando o computador interpretasse essa atribuição de valor. Nas linguagens de programação a igualdade ensinada nesta unidade é representada como == se quisermos comparar só valor ou === se quisermos comparar valor e tipo de dado. [↑](#footnote-ref-14)
15. Em Javascript e na maioria das demais linguagens de programação o ! (exclamação) é o operador de negação. [↑](#footnote-ref-15)
16. A execução de uma estrutura de desvio de fluxo se dá através de algumas palavras reservadas: O “Se”, por exemplo, é definido pela palavra *If* pelas linguagens de programação. [↑](#footnote-ref-16)
17. A realização da lógica de execução ocorre a partir do término de um *if* (lógica de decisão) ou a partir da definição da palavra Else ou Else If (lógica alternativa). A definição do Else (senão) prevê uma execução alternativa igualmente ao Else If. A diferença entre eles é que o *Else* implementa implicitamente o oposto da lógica de decisão anterior, já o *Else If*, implementa uma nova lógica de decisão que não necessariamente é o oposto da lógica de decisão anterior. [↑](#footnote-ref-17)
18. É muito comum lógicas de decisão testarem uma variável sem utilizar operadores relacionais. Como por exemplo, para testar se uma variável existe ou possui algum valor. Ex:  
      
    if (variavel) {   
     console.log (‘variável existe’)   
    } [↑](#footnote-ref-18)
19. A leitura que devemos fazer dessa uma estrutura *for* é: Para passo de zero até nove, retire uma bolinha de gude. [↑](#footnote-ref-19)
20. A variável de controle terá seu valor alterado após cada execução da instrução contida dentro de um laço de repetição. [↑](#footnote-ref-20)
21. Que fique claro, não é possível substituir um laço de repetição por uma estrutura de desvio de fluxo condicional, apenas foi exemplificado como seria a condicional do laço de repetição utilizando uma estrutura de desvio de fluxo condicional. [↑](#footnote-ref-21)
22. Um laço de repetição *while* deve ser lido como: Enquanto houver bolinhas de gude no baú, retire uma bolinha de gude. Já um laço de repetição *do while* deve ser lido como: Retire uma bolinha de gude, enquanto houver bolinhas de gude no baú [↑](#footnote-ref-22)
23. Listas encadeadas não são uma estrutura em si, mas sim uma técnica de alocação em memória para ser utilizada em estruturas lineares [↑](#footnote-ref-23)
24. Pode ser percebido que *?* corresponde a *“Então”*, em português estruturado. Prepara para a lógica de execução. E o caractere : corresponde a *“Senão”*, em português estruturado. Prepara para a lógica de execução alternativa. O *if ternário* é comumente utilizado para condicionais simples, com pouco código de lógica de execução, por esse motivo, não consegue substituir uma estrutura *else if,* apenas uma estrutura *if else.* [↑](#footnote-ref-24)
25. Quando o programador está iniciando um novo projeto, o repositório ainda não existe no servidor. Dessa forma, é comum o programador fazer um *clone* de um repositório vazio e então começar a desenvolver seus códigos dentro dele. [↑](#footnote-ref-25)
26. Em um projeto com uma equipe grande, as versões no servidor são modificadas com grande frequência, tornando obsoleta rapidamente a versão local dos programadores. [↑](#footnote-ref-26)
27. É uma prática muito comum, programadores criarem, a partir do *branch master*, o *branch desenv* e a partir desse, outros *branches* para desenvolverem tarefas específicas como: desenvolvimento de um novo requisito, correção de um *bug*, refatoração de código, escrita de testes, etc … [↑](#footnote-ref-27)