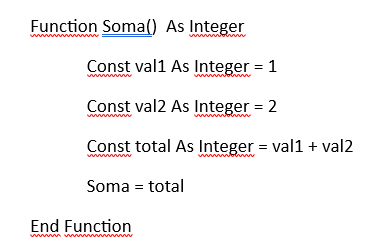
# VB6 – CONSTANTES

1. Uma constante dentro da função só vai ser acessível dentro do escopo da mesma, a não ser que eu retorne seu valor, usando o nome da própria função para fazer isso.



val1, val2 e total não poderão ser acessadas fora da função. Apenas Soma (retorno da função) estará disponível.

1. Constante Global: Posso declarar uma constante fora do escopo de alguma função, sub, para que seu valor seja acessado dentro do mesmo módulo/form
2. Constantes Públicas: Para acessar uma constante em outros formulários ou módulos, posso usar as constantes públicas:

* Public Const nomeGlobal As String = "Constante global"

1. Constantes não podem ser sobrescritas
2. Constantes devem ter seu tipo definido na declaração

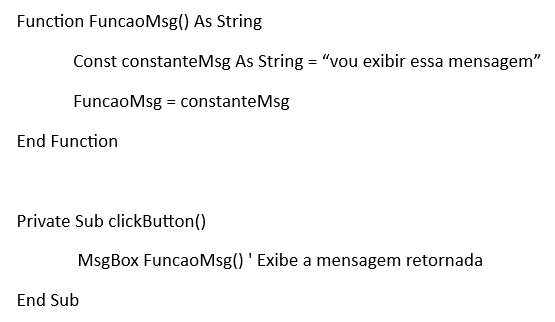
# VB6 – VARIÁVEIS

Comportamento semelhante ao das constantes, com algumas diferenças:

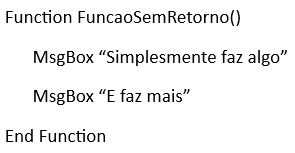
1. Variáveis podem ser sobrescritas
2. Variáveis não precisam ter seu tipo definido na declaração, sem a definição, seu tipo é Variant, que pode contar qualquer tipo de dados.
3. Posso usar Option Explicit para que as variáveis tenham que ser declaradas. E, ao usar, todas as variáveis devem ser declaradas.

# VB6 – FUNÇÕES

1. Uma função em VB6 pode ter um valor retornado, quando atribuo um valor ao seu nome, exemplo:



1. Também posso fazer uma função sem valor retornado



1. Ao definir uma função como pública, posso usá-la em outros forms, um uso comum é criar a função em um módulo e chamar ela (Call função) nos formulários que preciso usar, o que é útil para organizar o código ou reaproveitar a mesma função

# ARRAYS

Declaração:

* Dim Numeros(4) As Integer – de 0 a 4
* Dim Numeros(1 To 4) As Integer – de 1 a 4
* Dim Numeros() As Integer; –> ReDim Numeros(3); = redefinindo o tamanho depois de declarar.
* Ao usar ReDim Preserve Numeros(5) = ele mantém os dados que tem salvos no array anteriormente.
* Dim Matriz(2, 3) As Integer = array com linhas e colunas

Percorrer:

Usamos geralmente For e For Each.

# Escopos

**Public:** declaro em módulos, é acessível de qualquer lugar do projeto (form ou módulo).

*Exemplo:*

1. Arquivo mód1 = Public NomeUsuario As String ' Pode ser acessado em qualquer form ou módulo
2. Form1 = NomeUsuario = "João"
3. Form2 = MsgBox NomeUsuario ' Exibe "João"

**Private:** declaro em módulos ou forms, só pode ser usada dentro do próprio módulo ou form.

**Local**: Acessível apenas dentro do procedimento ou função onde foi declarada.

**Static**: O valor da variável é preservado entre chamadas (usando Static).

* Mantém o valor mesmo depois que a função termina. Diferente do Dim, que apaga a variável quando a função termina.

# Sub x Function:

* Sub executa ações sem retorno.
* Function executa ações e pode retornar valores.

# Form x Módulo

Form é usado para o que precisamos ver e usar, é a interface gráfica, diferente do módulo, que é usado para criar as funções e variáveis globais.

* O form pode conter código, mas não é ideal para armazenar funções reutilizáveis.
* O módulo é ideal para a lógica, funções reutilizáveis e globais, e, as variáveis globais também.

# Tipos de Dados

Em JS temos o number, em VB6 temos o Double – que são a mesma coisa. Como já estou acostumado com os tipos em JS, “traduzi” os tipos para VB6 a fim aprender como é a sintaxe no VB6.

|  |  |  |
| --- | --- | --- |
| JS | VB6 | EXEMPLO/EXPLICAÇÃO |
| number | Double | tipo numérico, números inteiros ou de ponto flutuante. |
| undefined | Ñ existe diretamente | Em JS, um tipo especial para valores indefinidos; em VB6, se não inicializados, variáveis e objetos podem ser Empty ou Nothing |
| null | Null / Nothing | Null para variants e Nothing para objetos, sendo a segunda para objetos que não estão referenciando nada |
| array | Variant | Array no VB6 (dinâmicos ou fixos) são do tipo ‘Variant’ = podem armazenar diferentes tipos de dados. |
| symbol | Não existe | identificadores únicos em JS |
| Function | Function | |  | | --- | |  |  |  | | --- | | **JS**:  function soma(a, b) {  return a + b; }  **VB6**:  Function Soma(a As Integer, b As Integer) As Integer  Soma = a + b  End Function | |
| Object | Object | **JS**:  let pessoa = { nome: "João", idade: 25 };  **VB6**:  Dim pessoa As Object  Set pessoa = CreateObject("Scripting.Dictionary")  pessoa.Add "nome", "João"  pessoa.Add "idade", 25 |

# MSFLEXGRID

MSFLEXGRID: possui as 2 propriedades: “cols” e “rows”, que são usadas para definir as colunas e linhas do grid

nameMsFLexGrid.ColWidth: define a largura da coluna

nameMsFLexGrid.TextMatrix (0, 0) = “conteúdo a ser recebido pela linha, na coluna”

* (linha, coluna)

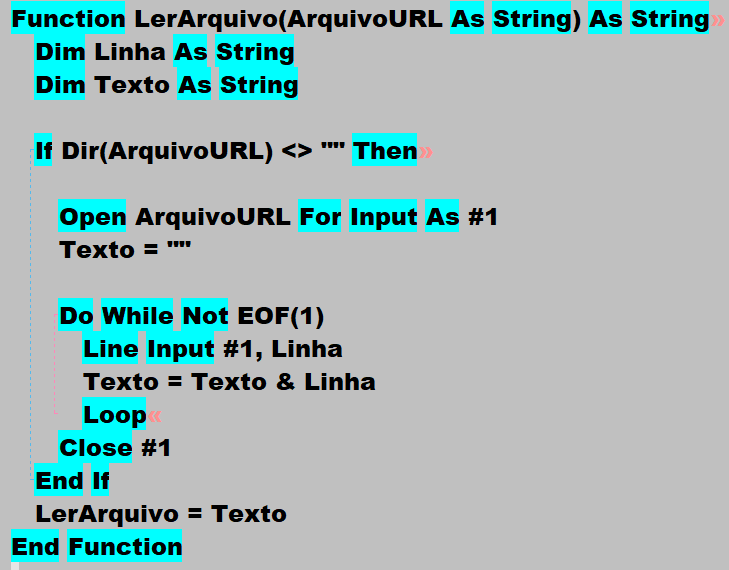
# TIMER

Timer executa alguma ação repetidamente após um intervalo de tempo. Usei para carregar arquivos e gerar um retorno visual, onde ele lia o arquivo 1 e exibia “carregando arquivo 1” para o cliente, logo em seguida - depois de 1seg, lê o 2.º arquivo e mostra em tela “carregando o arquivo 2”. (incrementando o Timer)

# ABRIR E LER ARQUIVOS

Para abrir um arquivo, basta: Open "caminho\_do\_arquivo" For MODO As #NÚMERO

* O caminho do arquivo pode ser uma variável com o caminho ou simplesmente o caminho mesmo
* Possuem diversos MODOS para abrir o arquivo, sendo eles:
  + **For Input**: Abre para leitura.
  + **For Output**: Abre para escrita e sobrescreve o arquivo.
  + **For Append**: Abre para escrita e adiciona no final do arquivo.
  + **For Binary**: Abre para leitura/escrita em binário.
* O comando Line Input #. serve para ler uma linha do arquivo



Para ler esse arquivo, primeiro verifiquei se ele era vazio, caso tenha conteúdo, vou ler. Então, leio enquanto o arquivo ainda tem conteúdo: ‘Do While Not EOF(1)’

Leio a linha e salvo na variável Linha, depois, concateno cada linha na variável Texto.

Assim que o conteúdo acabar e não tiverem linhas a serem lidas, o EOF será = 1, então terminará o loop e ‘Close #1’ fecha o arquivo.

# URL DO BANCO

Para a conexão ao banco de dados, ter um arquivo semelhante ao “Caminho\_BD” é importante para não precisar alterar em código toda vez que mudar o caminho do banco, tendo um arquivo txt para que o VB6 possa ler esse caminho, é só alterar o txt para que ele faça a comparação em código, com isso, ao declarar o caminho do banco, podemos concatenar ler o arquivo com o caminho, que gerará uma string semelhante a “C:\Lojas\BD\_Lojas.mdb”, que usamos para definir a conexão com o banco de dados.

# ADODB

Usando o ADODB.Connection – objeto da biblioteca ADO (ActiveX Data Objects) posso me conectar com um banco de dados e fazer comandos SQL

* Ele abre a conexão com o banco.
* Realiza select, insert, update, delete.
* Gerencia transações com commit e rollback
  + Commit: Confirma as alterações feitas durante uma transação, assim que eu confirmo, as alterações no BD se tornam permanentes e não tem volta.
  + Rollback: Desfaz as alterações feitas durante uma transação que ainda não foi confirmada, restaurando o banco de dados ao seu estado anterior

ADODB suporte qualquer banco compatível com OLE DB ou ODBC: Access, SQL Server, MySQ, Oracle... um jeito genérico de acessar vários bancos.

# CursorLocation

Define onde os dados serão manipulados, no cliente ou server

* adUseServer (Valor = 2 - Padrão): Usa o cursor no servidor (Access, SQL Server etc.)
* adUseClient (Valor = 3): Usa o cursor no cliente (VB6)

adUseClient:

* Os dados ficam na memória do VB6, não no banco.
* O banco só processa a consulta, e o VB6 faz o restante.
* Permite desconectar a conexão e continuar manipulando os dados.
* É mais rápido para consultas pequenas, mas pode consumir mais RAM com consultar grandes.
* Evita várias idas ao servidor (oq é ruim em redes lentas)
* Filtro, manipulação e ordenação de dados localmente

adUseServer:

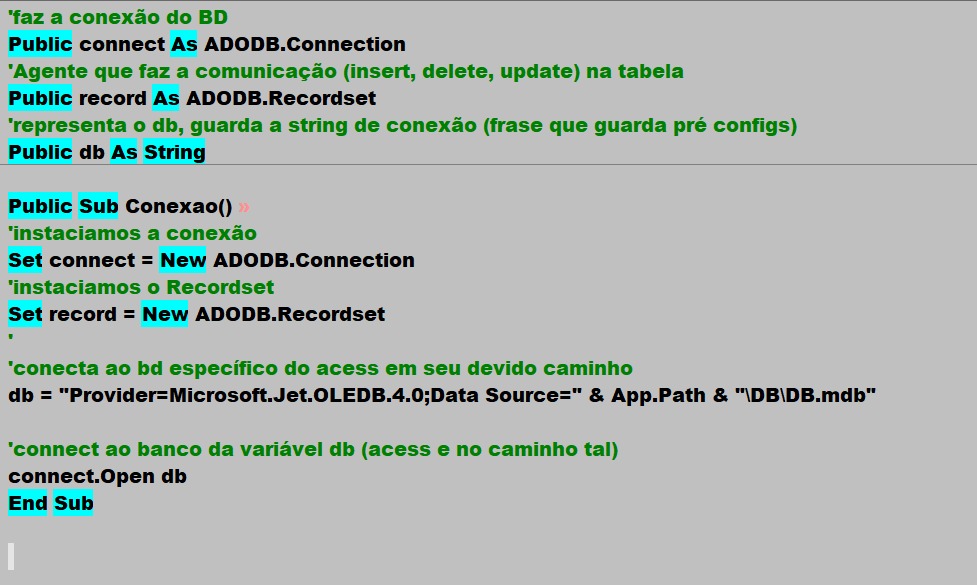
* O banco de dados gerencia bloqueios e concorrência.
* Melhor para consultas grandes

# KeyAscii

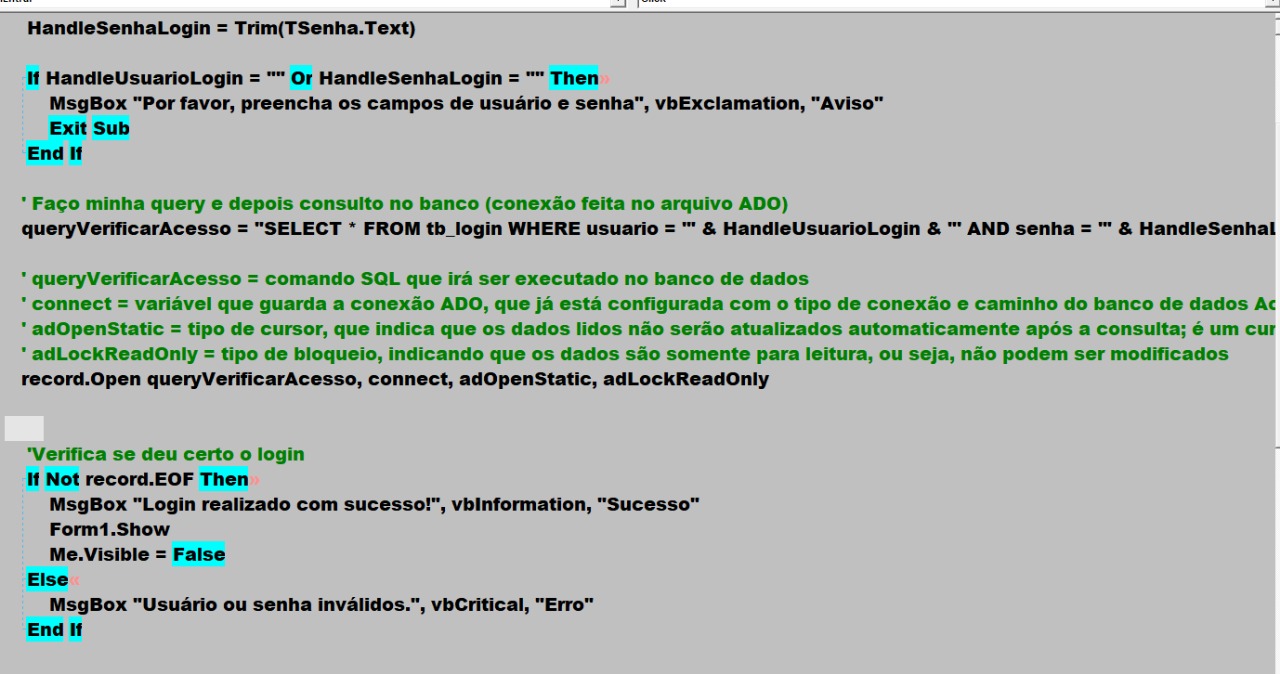
Temos códigos para as teclas do Windows, onde em uma calculadora, não permiti a entrada de nada que não fosse as teclas referentes aos números de 0 à 9 e a tecla backspace. Usando o site da Microsoft, tabelas do Google ou IA, conseguimos saber qual é a KeyAscii correspondente à tecla do Windows, onde por exemplo, ‘KeyAscii = 8’ corresponde ao backspace.

# TELA DE LOGIN

**Figura 1 – Função para conexão ao BD local feita em um módulo.**

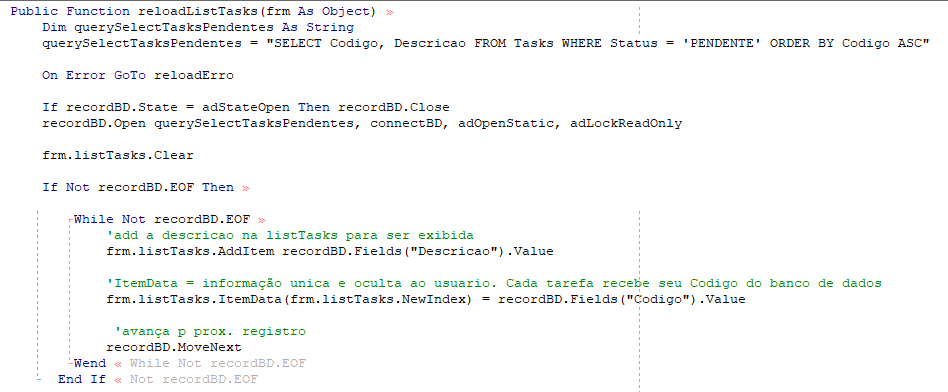


**Figura 2 - Código do FORM - parate que faz a conexão e retorna se deu certo ou não**



# SELECT, INSERT UPDATE E DELETE

## SELECT



‘reloadListTasks’ é usada após alguma operação de adicionar nova tarefa, concluir tarefa ou excluir tarefa, para que sejam recarregadas as tarefas pendentes de forma automática após qualquer processo e a lista esteja sempre correta.

‘On Error GoTo NomeDoErro’ faz com que eu possa capturar detalhes do erro, assim como seu número e a descrição, e, caso eu queira, exibir ao usuário. Se o código deve continuar rodando, posso usar o Resume Next ou posso encerrar com Exit. Também temos o ‘On Error Resume Next’, que ignora o erro e continua executando normalmente.

Salvo a query do select em querySelectTasksPendentes e executo:

* recordBD.Open querySelectTasksPendentes, connectBD, adOpenStatic, adLockReadOnly
* É um select onde defino que será aberta a conexão de forma estática (exibirá as informações do momento da consulta apenas) e que será uma conexão apenas para leitura (adLockReadOnly)

Depois, faço a limpeza da listTasks, e, no While, insiro todas que estão pendentes na listTasks até que não restem tarefas pendentes a serem adicionadas.

Como resultado, após qualquer operação de adição de tarefa, marcar como concluída ou deletar = o reload é feito as tarefas pendentes são exibidas sempre atualizadas na lista.

INSERT

A maior diferença foi ao executar o comando SQL, onde para consulta foi aberto o recordSet e definido o tipo de consulta e etc... enquanto aqui, foi simplesmente executado o comando no banco de dados.

1. queryAddTask = "INSERT INTO Tasks (descricao, status) VALUES ('" & newTask & "', 'PENDENTE')"
2. connectBD.Execute queryAddTask

## UPDATE

## DELETE