Excel マクロ入門

事務職員協会パソコン研修会 2018

KCS 鹿児島情報専門学校

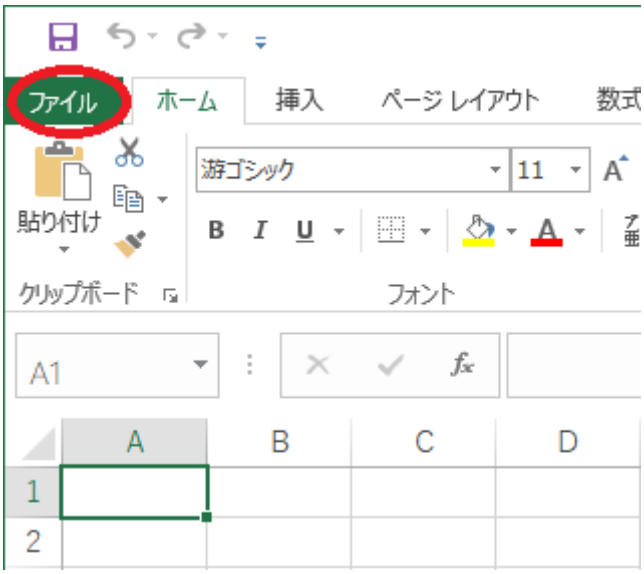
目次

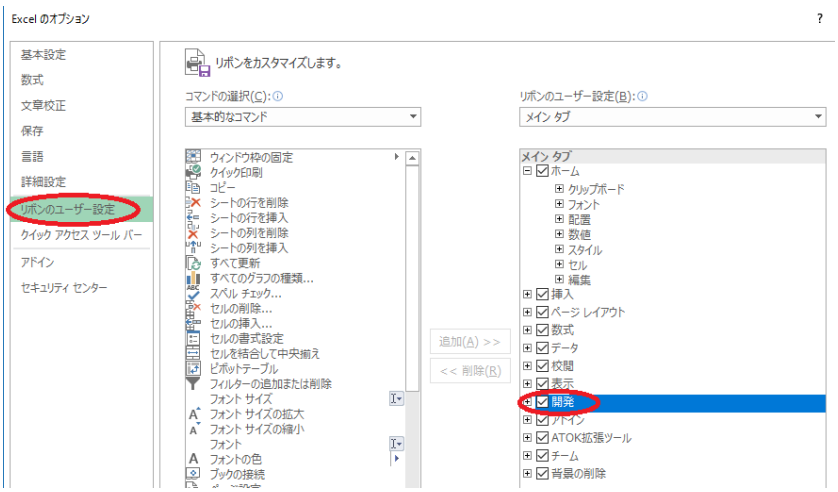
準備

資料の画面は Excel 2016 となっています。

Excel での準備

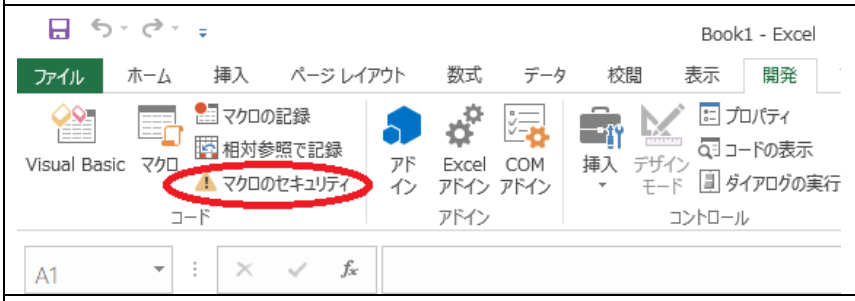
「開発」タブの表示設定を行います。

	「ファイル」タブをクリックします。
	「オプション」をクリックします。

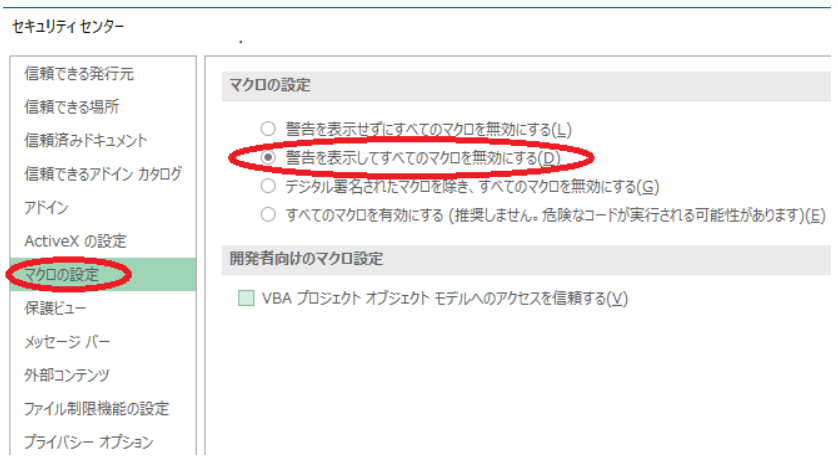


「リボンのユーザー設定」をクリックします。
「開発」にチェックを入れます。
「OK」をクリックします。

「開発」タブが追加されます。



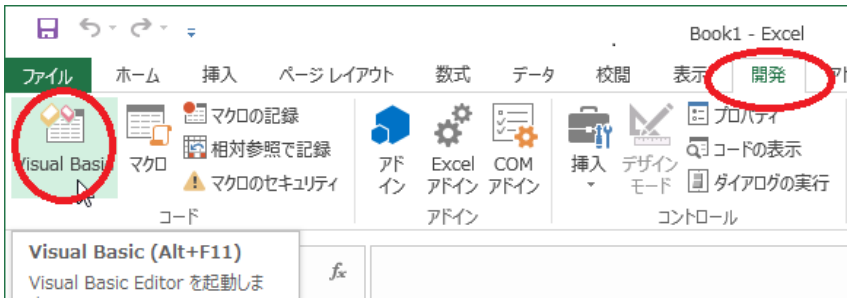
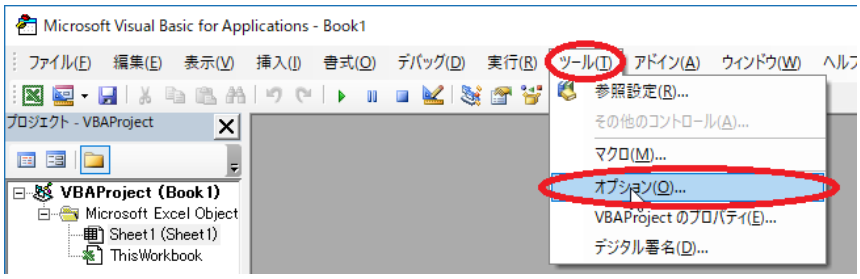
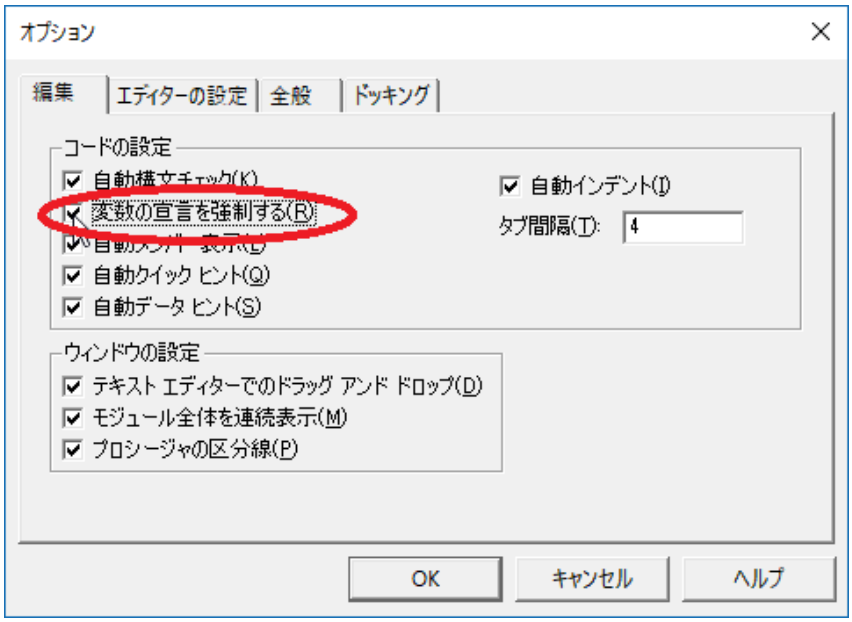
「開発」タブをクリックします。
マクロのセキュリティをクリックします。



「マクロの設定」をクリックします。
「警告を表示してすべてのマクロを無効にする」にチェックを入れます。
「OK」をクリックします。

Visual Basic での準備

Visual Basic の設定を行います。

	開発タブをクリックします。 Visual Basic をクリックして Visual Basic を起動します。
	メニューから「ツール」をクリックし 「オプション」をクリックします。
	オプションダイアログの「編集」タブ をクリックします。 変数の宣言を強制するをチェックし ます。 OK をクリックします。
	Visual Basic の×をクリックして閉 じます。

ファイルの保存形式について

保存形式を説明します。

種類	拡張子	マクロの保存	備考
 Excel ブック	.xlsx	×	Excel 標準関数など使用した Excel ファイル
 Excel マクロ有効ブック	.xlsm	○	マクロを含めた Excel ファイル
 Excel97-2003 ブック	.xls	○	過去のバージョンとの互換性を重視する場合。

※標準では拡張子が xlsx という office2007 で導入された新しいファイル形式で保存されます。

マクロを含める場合は xlsm でないとマクロが保存できません。

記録しているマクロが全て削除されます。

VBE で対象となるモジュールでエクスポートすることで別途保存できます。

モジュール名.bas

復元する際には VBE でインポートしてください。

マクロとは

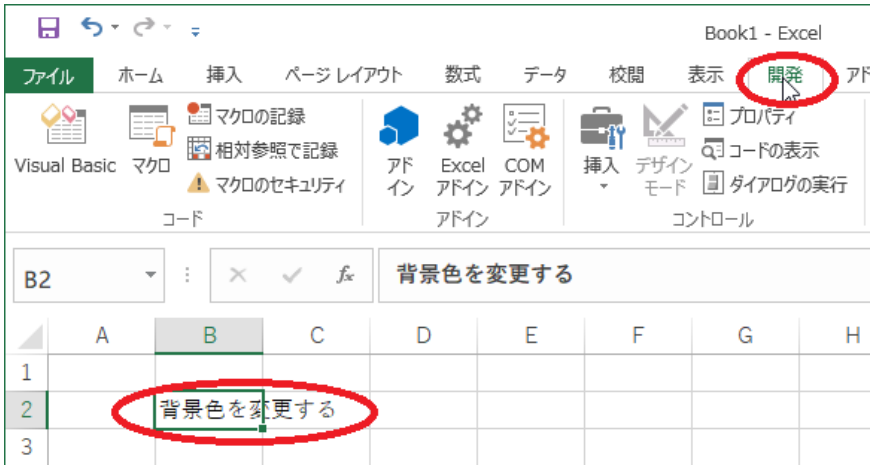
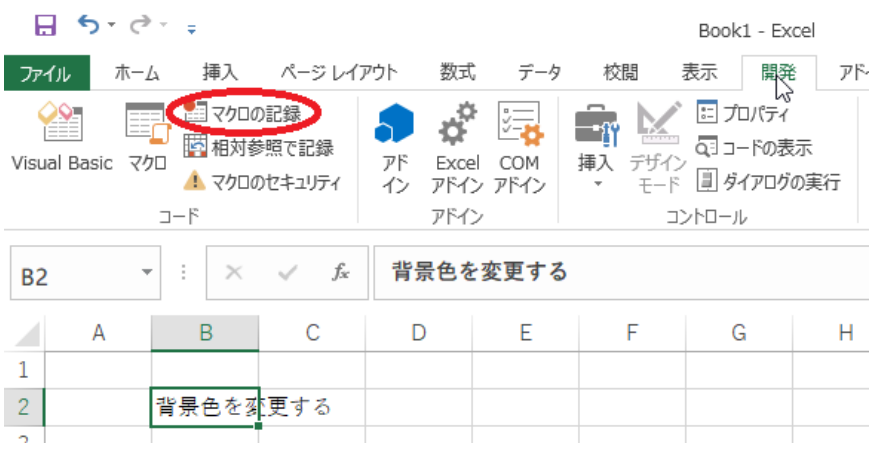
マクロは VBA を使って作成したプログラムです。では VBA とは何かと言えば、Microsoft Visual Basic for Application と言い Office 製品でマクロを開発するために使うプログラミング言語です。

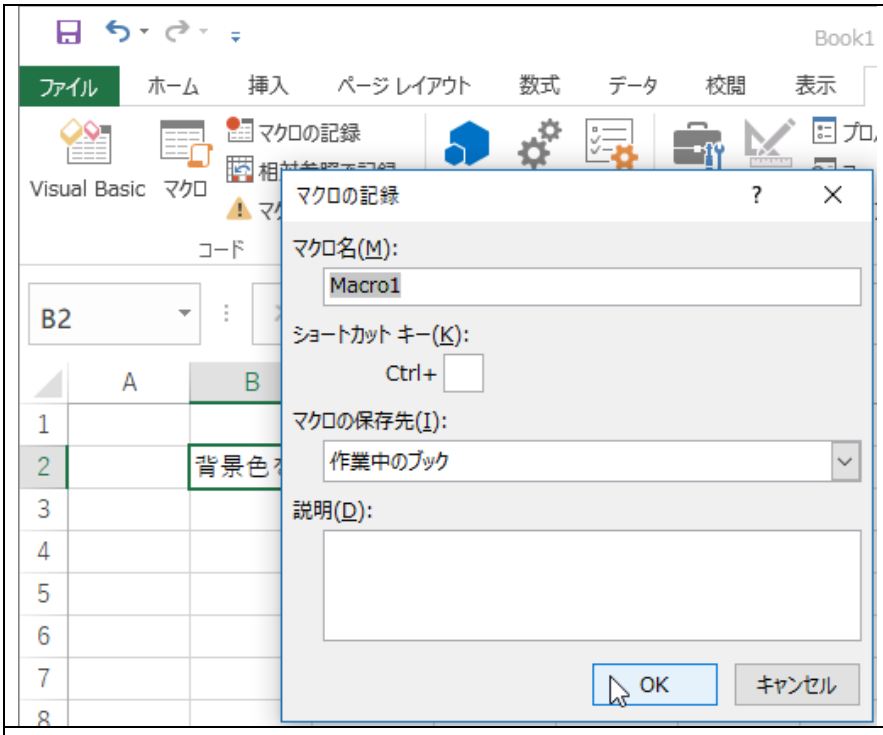
エクセルマクロとは

エクセルでデータを修正するときに、やることが決まっているのに、そのたびに何度も同じ操作を繰り返す時があります。決まりきった操作をマクロを使って自動化しようということです。
Excel をはじめ Microsoft の主要アプリケーションには共通マクロ言語として VBA を利用しています。
Excel でマクロを習得すれば、他のアプリケーションにも応用がききます。

マクロの記録機能

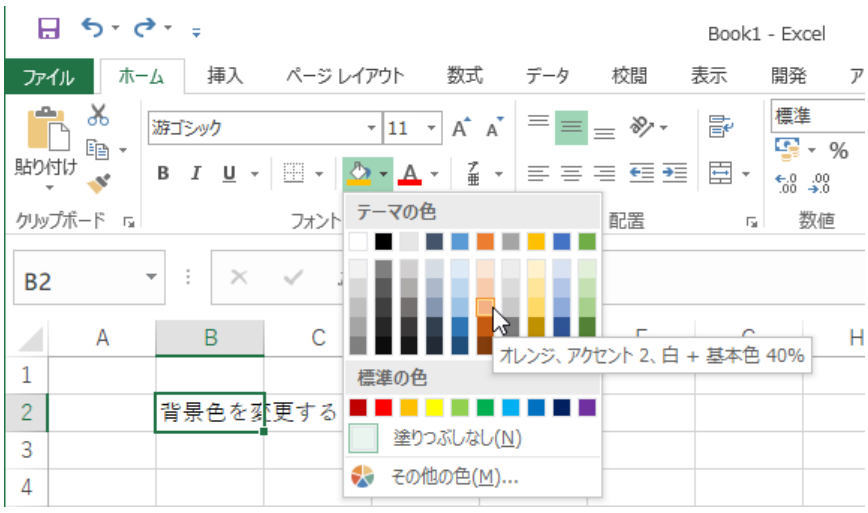
エクセルのワークシートで実行できる操作は、ほとんどがマクロとして記録することができます。
手始めに「セルの背景色の変更」ということを記録してみましょう。

	「開発」タブをクリックします。 B2 セルに文字を入力します。 ここでは「背景色を変更する」と入力しました。
	「マクロの記録」をクリックします。

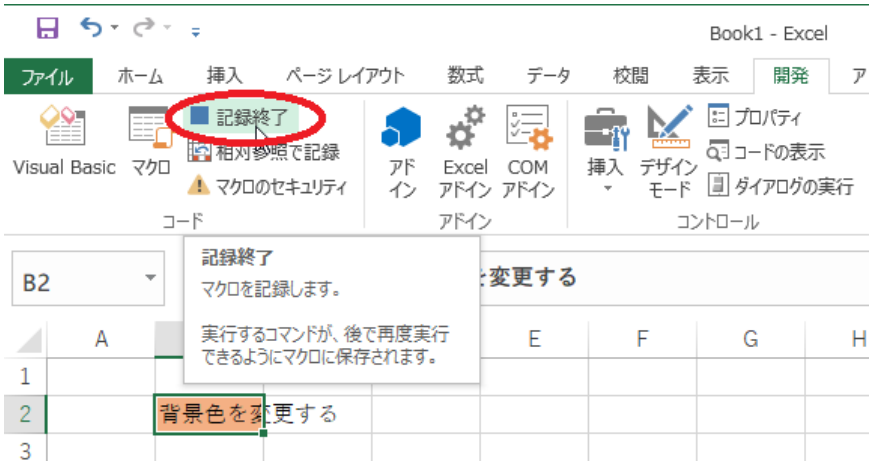


「マクロの記録」ダイアログが開きます。今回はこのまま「OK」ボタンをクリックします。

マクロ名：Macro1
マクロの保存先：作業中のブック



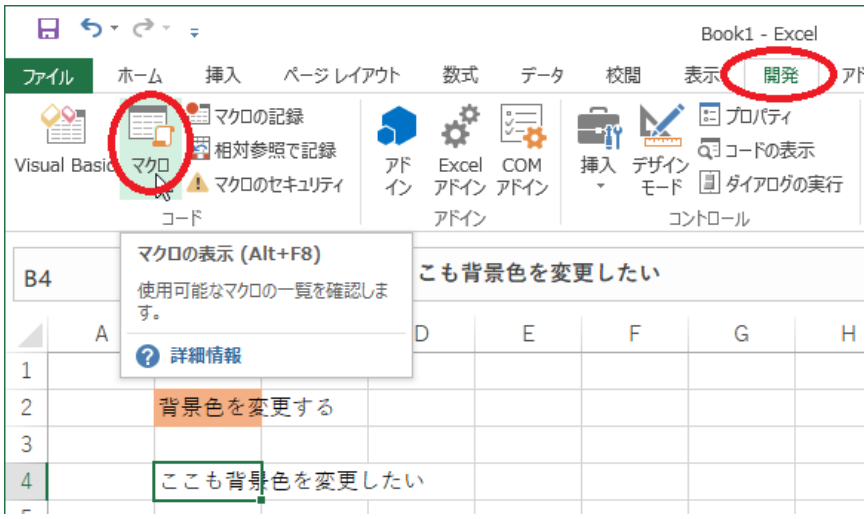
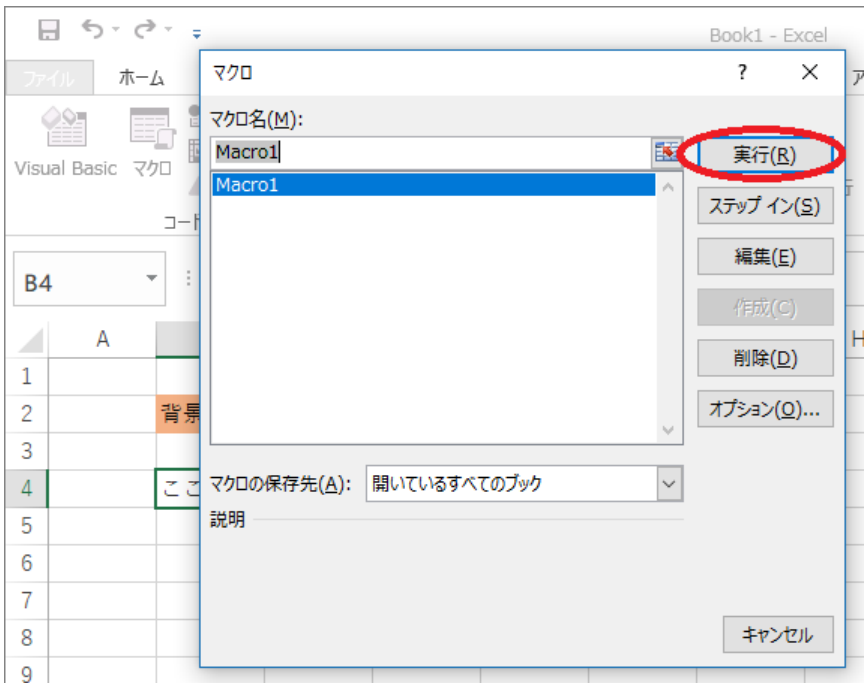
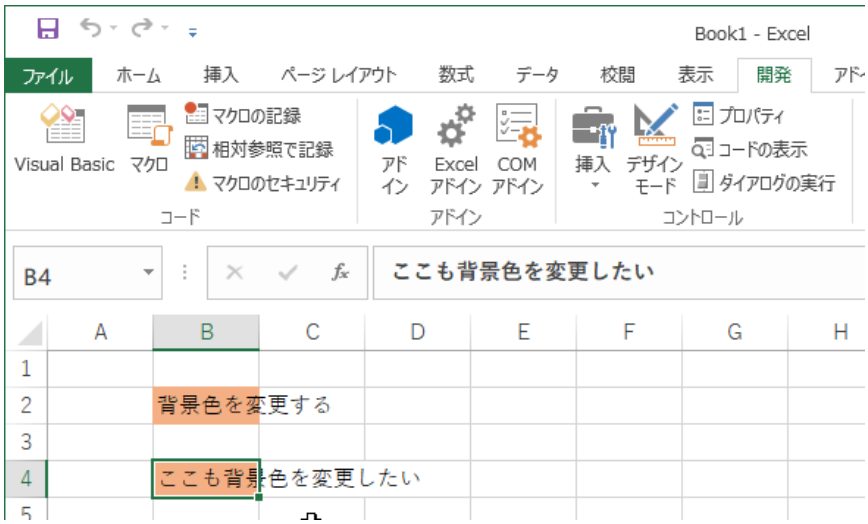
「ホーム」タブをクリックします。
フォントグループで塗りつぶしの色を選択します。



「開発」タブをクリックします。
「記録終了」をクリックします。
これで選択したセルの背景色を変更するマクロの記録が完了しました。
どこに記録されたか？
最初にマクロの保存先は、「作業中のブック」になっています。つまり現在作業中のブックに記録されています。

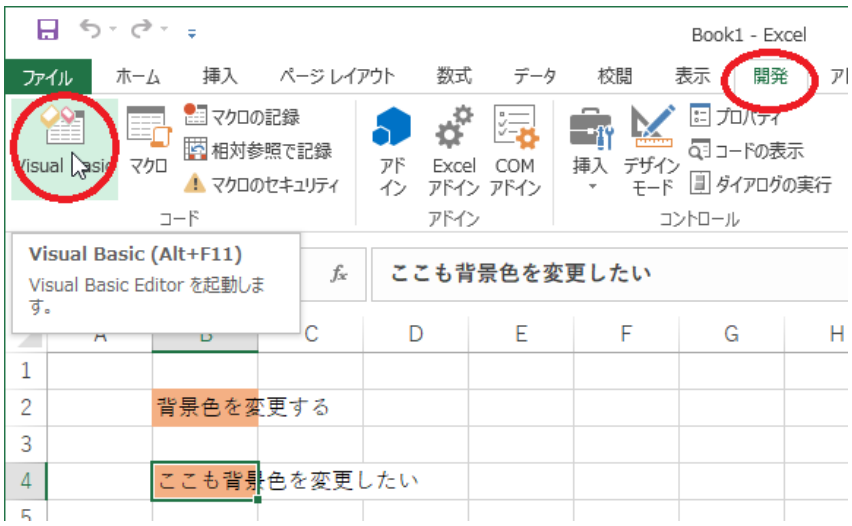
記録したマクロを実行してみる

作成したマクロを実行してみましょう。

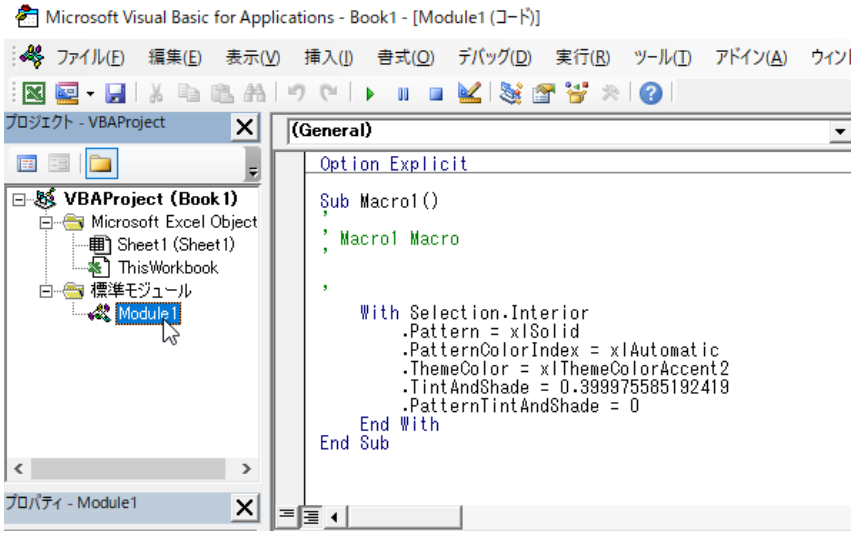
<table>
<tr>
<td> 
</td>
<td>他のセルに文字を入力します。
ここでは B4 セルに
「ここも背景色を変更したい」と入力しました。
「開発」タブの「マクロ」をクリックします。</td>
</tr>
<tr>
<td> 
</td>
<td>マクロダイアログが表示されます。
先ほど記録したマクロ「Macro1」を選択して実行をクリックします。</td>
</tr>
<tr>
<td> 
</td>
<td>選択されたセルの背景色が塗りつぶしされました。</td>
</tr>
</table>

記録したマクロを参照してみる

記録したマクロを参照してみましょう。



「開発」タブをクリックします。
「Visual Basic」をクリックします。



「Visual Basic Editor」が開きます。

メニューからファイル→終了して Microsoft Excel に戻る を選択して閉じます。
作業中のブックを「マクロの記録.xlsm」
マクロ有効ブックで保存します。

練習

「マクロの記録」を使って他にも記録してみましょう。

※作業したブックは上書保存せずに廃棄してください。

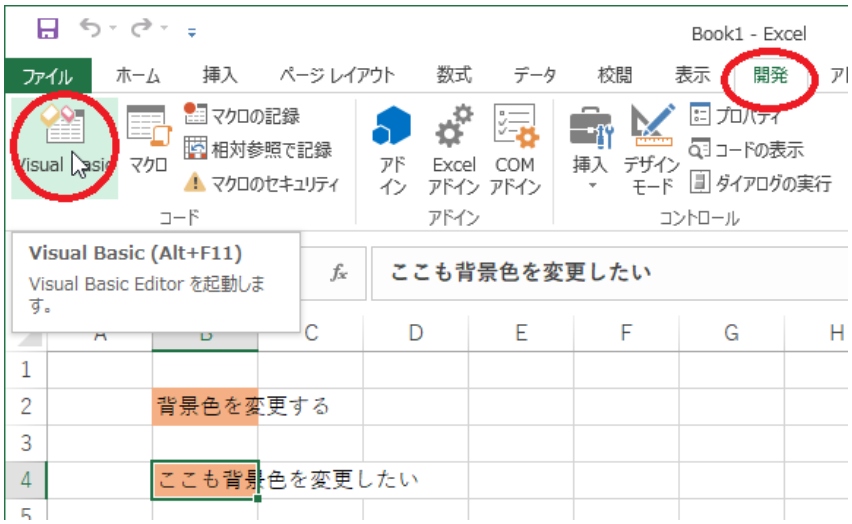
メモ

「マクロの記録開始」から「記録終了」までの全ての操作がマクロとして記録されます。
通常この記録されたマクロをそのまま使うことはありません。記録されたマクロを利用して
目的のマクロを作成します。ある操作がどのようなマクロになるとか調べたりするのに利用します。
まずは、「マクロの記録」機能を使ってマクロ（VBA）の雰囲気を掴みましょう。

記録したマクロの説明

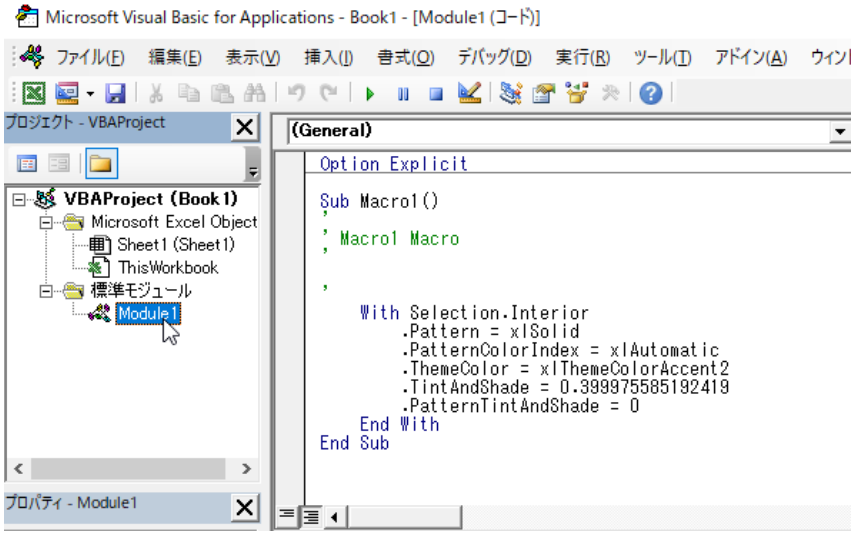
「マクロの記録」で生成したコードを Visual Basic Editor（VBE）で確認します。

マクロの記録.xlsm を開いてください。



「開発」タブをクリックします。

「Visual Basic」をクリックします。



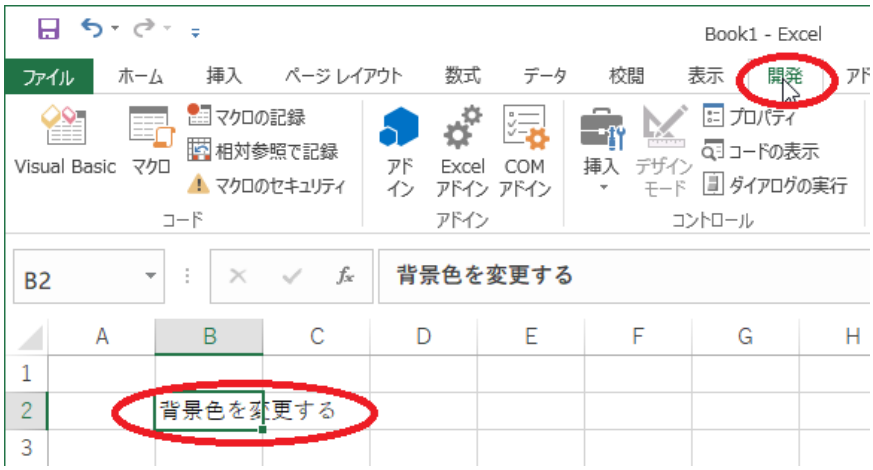
下記参照

```
Sub Macro1()
' Macro1 Macro
'  これはコメント。
    With Selection.Interior
        これは、選択した Inderior プロパティに対して
    .Pattern = xlSolid
        これは、「Pattern プロパティー」を「xlSolid」にする
    .PatternColorIndex = xlAutomatic
        これは、「PatternColorIndex プロパティー」を 「xlAutomatic」にする
    .ThemeColor = xlThemeColorAccent2
        これは、ThemeColor を xlThemeColorAccent2 にする
    .TintAndShade = 0.399975585192419
        これは、TintAndShade（色の明るさ）を 0.399975585192419 にする
    .PatternTintAndShade = 0
        これは、Pattern TintAndShade を 0 にする
```

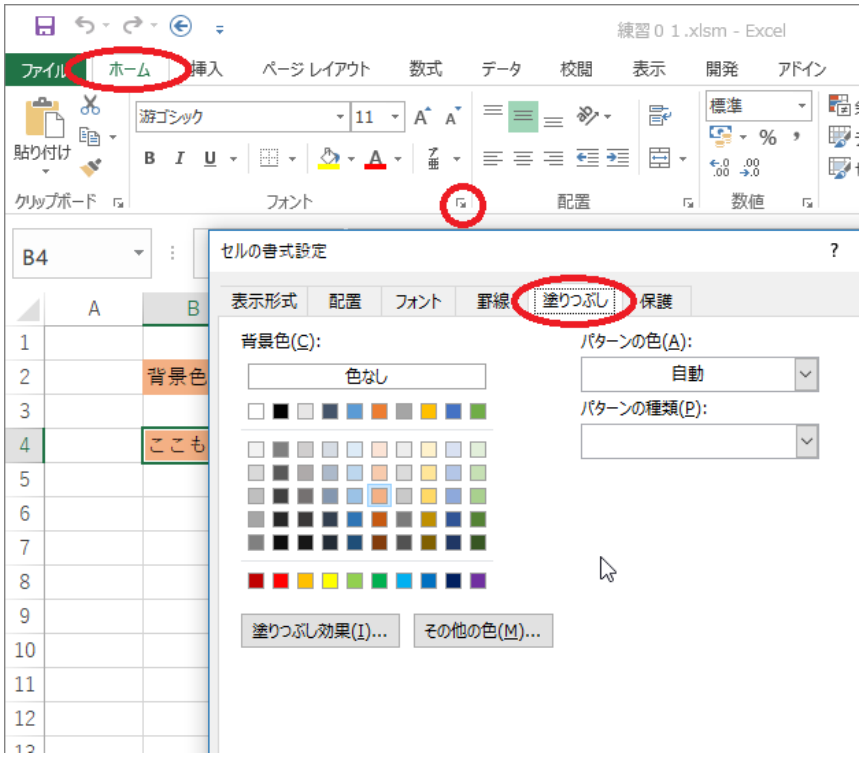
End With

これは、With で始まったステートメントを終わる

End Sub



セルの色を変更しただけなのに、随分コードが多いと感じたと思いますが、マクロの記録では、余分なコードも記載されてしまいます。



Interior（ぬりつぶし属性）には、背景色の他にもセットされている項目があるからです。

ホームタブのフォントグループの右下の矢印をクリックしてセルの書式設定ダイアログを表示させてみてください。

「塗りつぶし」タブをクリックすると、設定項目が複数あることが確認できます。

マクロの記録では設定しない項目も、設定しない(初期値)という設定になります。

プログラムの基本

変数

変数とは数値や文字列などを一時的に格納する入れ物と言えるでしょう。
コード内で変数を利用する場合にはまず、変数を宣言しておきます。変数の宣言には“Dim”を使用します。

```
Dim 変数名
```

変数に値を代入するには、”=”（イコール）を使用します。

```
変数名 = 値
```

使用例
2 つの変数の値を交換する。

```
sub swap()
    Dim a
    Dim b
    Dim swap
    a = 10
    b = 15
    MsgBox “a=” & a & “b=” & b
    swap = a
    a = b
    b = swap
    MsgBox “a=” & a & “b=” & b
End Sub
```

疑問？
2 つの変数の値を交換するプログラムを見て何か気付いたでしょうか？

```
    a = b
    b = a
```

でダメでしょうか？
他のプログラミング言語でも同じですが、代入は全て上書きです。

```
  a = b
```

で変数 a の値が変数 b の値で上書きされ、変数 a の値が消えてしまいます。

```
  b = a
```

で変数 a の値（変数 b の値）で変数 b を上書きしてしまいます。
結果、変数 a と変数 b の値は同じ（最初の変数 b の値）です。
なので交換する場合は退避用の変数を準備して 3 つの変数で 2 つの変数の交換を行います。

変数のデータ型

宣言した変数にはデータ型を指定することができます。データ型とは、変数に格納できるデータの種類をいいます。データ型を指定しない場合、変数は自動的にバリアント型(Variant)になります。

```
Dim 変数名 As データ型
```

数値型

データ型	名称	消費メモリ	格納できる範囲
Byte	バイト型	1 バイト	0 ～ 255
Integer	整数型	2 バイト	-32,768 ～ 32,767
Long	長整数型	4 バイト	-2,147,483,648 ～ 2,147,483,647
Single	単精度 浮動小数点型	4 バイト	-3.402823E38 ～ -1.401298E-45(負の値) 1.401298E-45 ～ 3.402823E38(正の値)
Double	倍精度 浮動小数点型	8 バイト	-1.79769313486232E308 ～ -4.94065645841247E-324(負の値) 4.94065645841247E-324 ～ 1.79769313486232E308(正の値)
Currency	通貨型	8 バイト	-922,337,203,685,477.5808～922,337,203,685,477.5807

数値を記述する場合は、そのまま記述します。例えば「10」とか「12.34」とかです。

```
Sub Test01()
    Dim x As Integer
    Dim y As Double
    x = 10
    y = 1234.567
End Sub
```

文字列型

データ型	名称	消費メモリ	格納できる範囲
String	文字列型	２バイト	最大約２０億文字まで

文字を記述する場合には文字をダブルクォーテーション("")で囲って記述します。

```
Sub Test02()
    Dim x As String
    x = "鹿児島県鹿児島市"
End Sub
```

日付型

データ型	名称	消費メモリ	格納できる範囲
Date	日付型	8バイト	西暦 100 年 1 月 1 日～西暦 9999 年 12 月 31 日までの日付と時刻

値の指定方法としては、日付の形をした文字列を指定するか、シャープ(#)で囲んだ形で日付を表す文字を指定するかのどちらかとなります。

```
Dim x As Date

x = "2006 年 8 月 23 日"
x = #2006 年 8 月 23 日#
x = #2006/8/23#
x = #8/23/2006#
x = #11:26:39 AM#
```

オブジェクト型

データ型	名称	消費メモリ	格納できる範囲
Object	オブジェクト型	4バイト	オブジェクトを参照するデータ型

バリアント型

データ型	名称	消費メモリ	格納できる範囲
Variant	バリアント型	１６バイト	文字列型の範囲と同じ。

※万能であるがゆえに他のデータ型よりメモリの消費が大きい

真偽値型

データ型	名称	消費メモリ	格納できる範囲
Boolean	ブール型	２バイト	真(True)または偽(False)

算術演算子

計算を行うには算術演算子を使用します。変数に値を代入するには、”=”(イコール)を使用します。

```
変数名 = 変数名１ 演算子 変数名２
```

演算子	意味	使用例	結果
＋	加算	8 + 5	13
－	減算	10 – 4	6
*	乗算	3 * 5	15
／	除算	8 / 5	1.6
¥	除算の商	8 ¥ 5	1
Mod	除算の余り	8 Mod 5	3
^	べき乗	6 ^ 2	36

使い方は変数に値を格納する時と同じです。

```
Sub Test03()
    Dim x As Integer
    x = 5 * 4
End Sub
```

比較演算子

値と値を比較するには、比較演算子を使用します。「大きい」とか「未満」などです。
比較した結果はブール型の値として「真(True)」または「偽(False)」の値を取ることになります。

演算子	意味	使用例	結果
<	小さい	8 < 5	False
<=	以下	3 <= 8	True
>	大きい	8 > 5	True
>=	以上	3 >= 8	False
=	等しい	3 = 8	False
<>	等しくない	3 <> 8	True

比較演算子は条件判定や繰り返し処理の中で使われることがほとんどです。無理やり使ってみると次のようになります。

```
Sub Test04()
    Range("A1").Value = 10 > 20
End Sub
```

上記を実行すると、セル(A1)に「FALSE」という値が表示されます。

論理演算子

「A かつ B」とか「A または B」などのように条件を組み合わせて使う場合に使用します。
比較した結果はブール型の値として「真(True)」または「偽(False)」の値を取ることになります。

演算子	意味	使用例	結果
And	論理積	8 > 4 And 2 <= 3	True
Or	論理和	8 > 4 Or 4 <= 1	True
Not	論理否定	Not 8 > 4	False

上記の例でも簡単に書いていますが論理演算子を使う場合には次のように記述します。

```
(条件式 1) And (条件式 2)
(条件式 1) Or (条件式 2)
Not (条件式 1)
```

「And」は、左辺及び右辺にある条件式がどちらも「True」の場合だけ結果として「True」を返します。
「Or」は、左辺または右辺のどちらかの条件式が「True」の場合に「True」を返します。
「Not」は、右辺の条件式が「True]の場合には「False」を、「False」の場合には「True」を返します。

文字列の結合

2 つの文字列を 1 つの文字列として結合する方法について確認していきます。
文字列と文字列を結合するにはアンパサンド(&)を使う方法とプラス(+)を使う方法の 2 つがあります。

```
Dim a As String
Dim b As String
Dim c As String
Dim d As String
a = "鹿児島県"
b = "鹿児島市"
c = a & b
d = a + b
```

上記では変数「c」と変数「d」のどちらにも「鹿児島県鹿児島市」と変数「a」と変数「b」の文字列が結合された文字列が格納されます。
また文字列と文字列の結合だけではなく、文字列と数値を結合して新しい文字列とすることもできますし、数値と数値を結合して文字列とすることもできます。
「&」と「+」のどちらを使ってもいいのですが、「+」の方は数値と数値で使うと加算が実行されてしまい、単に結合とはなりません。また数値と文字、文字と数値では実行時エラーが発生してしまいますので注意が必要です。

オブジェクトとは

データ型の 1 つにオブジェクト型というものがあります。オブジェクトとは、ブックやシート、そしてセルなど VBA で何か操作をしようとする対象となるものです。

主なオブジェクトには次のようなものがあります。

要素名	オブジェクト名
アプリケーション	Application
ブック	Workbook
ワークシート	WorkSheet
セル	Range

他にもグラフやフォームなどを表すオブジェクトがあります。
また例えばワークシートであれば、1 つのブックオブジェクトの中に複数のワークシートオブジェクトが含まれています。このような同じオブジェクトの集まりのことをコレクションと言います。(コレクション自体もオブジェクトの 1 つです)。

例えば次のようなコレクションがあります。

要素名	オブジェクト名	説明
ブック	Workbooks	開いている全てのブック
ワークシート	WorkSheets	ブックに含まれる全てのワークシート

コレクションは例えばあるブックに含まれる全てのワークシートを管理しているもので、ワークシートコレクションから含まれるワークシートオブジェクトを 1 つ 1 つ取り出していくということが可能になります。

また、コレクションは階層構造になっています。

```
Application
  |
  +- Workbooks
       |
       +- Worksheets
            |
            +- Range
```

最上位には「Application」オブジェクトがあり、順にブック、シート、セル、と階層が下がっていくことになります。例えばシートはブックの中に含まれていますし、セルはシートの中に含まれているものですから、この階層構造は理解できるかと思います。
VBA ではオブジェクトに対して色々な操作を行います。例えばシートを開くとかセルに文字を入力するとかなど操作の対象となるものがオブジェクトです。使用例でも「Range("A1").Value = 10」などと書いてきました。これは「A1 の位置にあるセルの値を 10 に設定する」という意味になります。

変数にオブジェクト型を使用する

オブジェクトもデータ型の 1 つですので、各オブジェクトをデータ型とした変数を定義することができます。

```
Dim testBook As Workbook
Dim testSheet As Worksheet
Dim testRange As Range
```

使い方は他のデータ型と基本的には同じですが、変数に値を格納する時、左辺に変数、右辺に値、を記述して単にイコール(=)で結ぶだけではなく「Set」を付ける必要があります。

```
Dim testRange As Range
Set testRange = Range("A1")
testRange.Value = 20
```

いったん変数にオブジェクトを格納したら、後はオブジェクトを直接書く代わりに変数を使って記述することが出来ます。このあたりは他のデータ型の場合と同じです。

プロパティとメソッド

オブジェクトにはセルやワークシートがありますが、各オブジェクトにはプロパティとメソッドを使って操作を行います。

オブジェクトが持っている各種情報を取得するにはオブジェクトのプロパティを使います。例えばセルのオブジェクトを例にすればセルに現在入力されている値や背景色など対象のセルの状態を表す各種情報を保持しているのがプロパティです。

プロパティは次のような構文となります。

```
オブジェクト.プロパティ名
```

どのようなプロパティが用意されているかはオブジェクトにもよりますが、例えばオブジェクトに含まれている値を表すプロパティは「Value」となります。例としてセル("A1")に含まれている値は下記のようにして変更する事が出来ます。

```
Dim testrange As Range
Set testrange = Range("A1")
testrange.Value = 10
```

上記の場合は、セル("A1")を表す Range オブジェクトを取り出し、そのオブジェクトの「Value」というプロパティに 10 という値を設定しています。「Value」プロパティはオブジェクトに格納されている値を管理しているプロパティですので、結果としてセル("A1")に 10 という値が表示されます。

メソッドはオブジェクトに対する動作を指定します。例えばセルを削除するとか、セルを選択するとか、などです。

メソッドは次のような構文となります。

```
オブジェクト.メソッド名
```

どのようなメソッドが用意されているかはオブジェクトにもよりますが、セルを削除するメソッドとして「Delete」が用意されており、例としてセル A1 を削除する場合には次のように記述します

```
Dim testrange As Range
Set testrange = Range("A1")
testrange.Delete
```

またメソッドによっては引数を取るものがあります。引数を記述する場合は次のような構文となります。

```
オブジェクト.メソッド名 引数名 1:=引数 1, 引数名 2:=引数 2, ...
```

制御構造

ここから VBA でプログラムの流れを制御するための構文を見て行きます。

条件分岐型

条件によって処理を分けると言うのは、例えば計算結果としてこうだったらこの処理をしたいけど、違ったら別の処理をしたいなどその時の条件の結果によって処理を分けたい場合に使います。条件分岐には「if」を使います。
基本的な構文は下記の通りです。

```
If 条件式 Then
    (条件式が True の時に行う処理 1)
    (条件式が True の時に行う処理 2)
End If
```

「if」文は「If」で始まり「End If」で終わります。そして「If」の後に処理を行うかどうかを判別する条件式を記述し、その条件を満たす場合に行いたい処理を「If」の行と「End If」の間に記述します。処理は複数の行を記述できますので条件を満たす場合に行いたい処理を複数記述することができます。

まず例で見てみましょう。変数 x の値が 10 より大きい場合にセルに「大きい」と表示したい場合には次のように記述します。

```
Dim x As Integer
x = 12
If x > 10 Then
    Range("A1").Value = "大きい"
End If
```

ここで条件式の記述方法について確認しておきます。条件式には比較演算子を使って数値の大きさを比較したり、文字列が他の文字列と同じかどうかなどを判断します。比較演算子は結果として「True」または「False」を返しますので、条件式に記述した式が結果として「True」を返すような場合が条件を満たす場合ということになり、その後の処理を実行します。

先ほどの例の場合であれば「x＞10」という部分が条件式になります。変数「x」には「12」と言う数値が格納されていますので条件式は「True」を返します。その為、この条件分岐は実行され「Range("A1").Value＝"大きい"」という処理が実行されるわけです。変数「x」に格納されている値が例えば「8」とかの数値であれば、

条件分岐は「False」になるため処理は実行されません。

```
Sub TEST05()
    Dim x As Integer
    x = 12
    If x > 10 Then
        Range("A1").Value = "変数は 10 より大きい"
    End If
End Sub
```

先ほどの例では if 文の条件式が「True」になる場合の処理だけを記述していましたが、同じ条件式で条件が満たされなかった場合に行われる処理を記述することも可能です。

構文は下記の通りです。

```
If 条件式 Then
    (条件式が True の時に行う処理 1)
    (条件式が True の時に行う処理 2)
Else
    (条件式が False の時に行う処理 1)
    (条件式が False の時に行う処理 2)
End If
```

基本的な構文は同じですが、条件式が「False」の場合に行う処理を「Else」の後に記述します。またこの場合には条件式が「True」の場合に行われる処理は「If」の行から「Else」の行までの間に記述します。

まず例で見てみましょう。変数 x の値が 10 より大きい場合にセルに「大きい」と表示し、10 よりも小さい場合には「小さい」と表示する場合には次のように記述します。

```
Dim x As Integer
x = 8
If x > 10 Then
    Range("A1").Value = "大きい"
Else
    Range("A1").Value = "小さい"
End If
```

今回の場合、変数「x」には「8」という数値が格納されています。その為条件式は「False」となりますので「Range("A1").Value = "小さい"」が実行されることになります。

```
Sub TEST06()
    Dim x As Integer
    Dim y As Integer
    x = 12
    y = 8
    If x > 10 Then
        Range("A1").Value = "変数 x は 10 より大きい"
    Else
        Range("A1").Value = "変数 x は 10 より小さい"
    End If
    If y > 10 Then
        Range("A2").Value = "変数 y は 10 より大きい"
    Else
        Range("A2").Value = "変数 y は 10 より小さい"
    End If
End Sub
```

変数「x」に対するの条件判定は「True」となり、変数「y」に対するの条件判定は「False」となるため、それぞれ対応した処理が行われます。

条件は○か×かのようにどちらかだけを選択する場合だけではありません。例えば変数に格納された文字列を色々な値を比較する場合など複数の条件判断を行う場合もあります。このような時には 1 つの if 文の中に複数の条件分岐を記述することが出来ます。

構文は下記の通りです。

```
If  条件式 1 Then
    (条件式 1 が True の時に行う処理 1)
    (条件式 1 が True の時に行う処理 2)
ElseIf  条件式 2 Then
    (条件式 2 が True の時に行う処理 1)
    (条件式 2 が True の時に行う処理 2)
ElseIf  条件式 3 Then
    (条件式 3 が True の時に行う処理 1)
    (条件式 3 が True の時に行う処理 2)
Else
    (どの条件式も False の時に行う処理 1)
    (どの条件式も False の時に行う処理 2)
End If
```

複数の条件を記述する場合、最初の条件については「If」の後ろに記述、それ以降の条件については「ElseIf」の後ろに記述します。また最後に「Else」を記述した場合は全ての条件式で「False」になった場合の処理を記

述します。条件式は何個書いても構いません。
複数の条件を記述する場合、上から順番に条件の確認が行われて行きます。
まず「条件式 1」が確認され「True」であればその下に書かれた処理を実行します。実行後は「End If」の次の行に処理が移ります。
次に「条件式 1」が「False」の場合には、次の条件式である「条件式 2」が確認され「True」であればその下に書かれて処理を実行します。実行後は「End If」の次の行に処理が移ります。
次に「条件式 1」と「条件式 2」が共に「False」の場合には、次の条件式である「条件式 3」が確認され「True」であればその下に書かれて処理を実行します。実行後は「End If」の次の行に処理が移ります。
最後に「条件式 1」「条件式 2」「条件式 3」が全て「False」の場合にはその下に書かれて処理を実行します。

```
Sub TEST07()
    Dim x As String
    x = "鹿児島"
    If x = "福岡" Then
        Range("A1").Value = "お住まいは福岡です"
    ElseIf x = "大分" Then
        Range("A1").Value = "お住まいは大分です"
    ElseIf x = "鹿児島" Then
        Range("A1").Value = "お住まいは鹿児島です"
    Else
        Range("A1").Value = "お住まいは分かりません"
    End If
End Sub
```

今回は文字列型の変数に格納された値を、色々な値と比較しています。結果として変数に格納されている値は「鹿児島」ですので「Range("A1").Value = "お住まいは鹿児島です"」という処理だけが実行されています。

もう一度 if 文の基本的な構文を見てみます。

```
If 条件式 1 Then
    (条件式 1 が True の時に行う処理 1)
    (条件式 1 が True の時に行う処理 2)
Else
    (条件式 1 が False の時に行う処理 1)
    (条件式 1 が False の時に行う処理 2)
End If
```

条件が「True」の場合や「False」の場合に行いたい処理は特別なものではありませんのでどのような処理でも記述ができます。その為、条件式が「True」や「False」の場合に行われる処理のところに別の if 文を記述することも可能です。

```
If 条件式 1 Then
    (条件式 1 が True の時に行う処理 1)
    If 条件式 2 Then
        (条件式 2 が True の時に行う処理 1)
        (条件式 2 が True の時に行う処理 2)
    Else
        (条件式 2 が False の時に行う処理 1)
        (条件式 2 が False の時に行う処理 2)
    End If
    (条件式 1 が True の時に行う処理 2)
Else
    (条件式 1 が False の時に行う処理 1)
    (条件式 1 が False の時に行う処理 2)
End If
```

この場合は特に特別な構文と言うわけではなく、行われる処理の中に、処理の 1 つとして別の if 文を記述しただけです。ですので、どれだけ階層を深くして記述しても構いません。

```
If 条件式 1 Then
    (条件式 1 が True の時に行う処理 1)
    If 条件式 2 Then
        (条件式 2 が True の時に行う処理 1)
        If 条件式 3 Then
            (条件式 3 が True の時に行う処理 1)
        Else
            (条件式 3 が False の時に行う処理 1)
        End If
        (条件式 2 が True の時に行う処理 2)
    Else
        (条件式 2 が False の時に行う処理 1)
        (条件式 2 が False の時に行う処理 2)
    End If
    (条件式 1 が True の時に行う処理 2)
Else
    (条件式 1 が False の時に行う処理 1)
    (条件式 1 が False の時に行う処理 2)
End If
```

```
Sub TEST08()
    Dim x As Integer
    x = 9
    If x > 0 Then
        If x Mod 2 = 0 Then
            Range("A1").Value = "値は正の偶数です"
        Else
            Range("A1").Value = "値は正の奇数です"
        End If
    Else
        Range("A1").Value = "値は負の数です"
    End If
End Sub
```

条件によって処理を分ける方法として、if 文の他に select 文があります。select 文はある 1 つの値を複数の他の値を比較する場合に使います。

基本的な構文は下記の通りです。

```
Select Case 比較対象
Case Is 条件式 1
    (条件式 1 が True の時に行う処理 1)
    (条件式 1 が True の時に行う処理 2)
Case Is 条件式 2
    (条件式 2 が True の時に行う処理 1)
    (条件式 2 が True の時に行う処理 2)
Case Is 条件式 3
    (条件式 3 が True の時に行う処理 1)
    (条件式 3 が True の時に行う処理 2)
Case Else
    (どの条件式も False の時に行う処理 1)
    (どの条件式も False の時に行う処理 2)
End Select
```

if 文と似ていますが、if 文の場合は複数の条件式を記述する場合に別々の条件式を記述することが出来ましたが、select 文の場合には比較する対象が 1 つあり、その値に対する条件式を記述していきます。また「Case Else」は不要であれば記述しなくても構いません。

具体的な例で見てください。下記の例は変数「x」に格納されている値を色々な値と比較しています。

```
If x = "東京" Then
    Range("A1").Value = "Tokyo"
ElseIf x = "大阪" Then
    Range("A1").Value = "Osaka"
ElseIf x = "名古屋" Then
    Range("A1").Value = "Nagoya"
ElseIf x = "札幌" Then
    Range("A1").Value = "Sapporo"
ElseIf x = "福岡" Then
    Range("A1").Value = "Fukuoka"
ElseIf x = "鹿児島" Then
    Range("A1").Value = "Kagoshima"
End If
```

これを select 文を使って書くと次のようになります。

```
Select Case x
Case "東京"          ※Is =は省略可
    Range("A1").Value = "Tokyo"
Case "大阪"
    Range("A1").Value = "Osaka"
Case "名古屋"
    Range("A1").Value = "Nagoya"
Case "札幌"
    Range("A1").Value = "Sapporo"
Case "福岡"
    Range("A1").Value = "Fukuoka"
Case "鹿児島"
    Range("A1").Value = "Kagoshima"
End Select
```

select 文を使う場合は、まず比較対象として変数「x」を指定し、条件を「Case」の値に続けて記載します。条件の部分に何も演算子が書かれていない場合には値が同じなのかどうかを判別する条件式となります。
if 文で書けるものをなぜ select 文を使って書くのかと言うと、select 文の場合は比較対象となる値が 1 つあり、それを色々な値と比較するということが明確になるため、後でプログラムを読み返した時に分かりやすくなるためです。

select 文で書ける条件分岐のプログラムは全て if 文を使って書き直すことができます。ただし select 文では条件式の対象となる値が 1 つに決まっているため次のような if 文を select 文を使って書き直すことはできません。

```
If x> 10 Then
ElseIf y > 8 Then
End If
```

この場合は、最初の条件式で変数「x」を他の値と比較していますが、2 番目の条件式では変数「y」を他の値と比較しています。このように比較対象となる値が異なるような条件式を記述するには if 文を使うしかありません。

```
Sub TEST09()
    Dim x As String
    x = "鹿児島"
    Select Case x
    Case "東京"
        Range("A1").Value = "Tokyo"
    Case "大阪"
        Range("A1").Value = "Osaka"
    Case "名古屋"
        Range("A1").Value = "Nagoya"
    Case "札幌"
        Range("A1").Value = "Sapporo"
    Case "福岡"
        Range("A1").Value = "Fukuoka"
    Case "鹿児島"
        Range("A1").Value = "Kagoshima"
    End Select
End Sub
```

また、ここまでのサンプルでは同じ値かどうかだけを判別する書き方をしていましたが、比較対象となる値が数値の場合に、大きいとか小さいとかの判別を行う事も可能です。その場合は次のように記述します。

```
Dim x As Integer
x = 10
Select Case x
Case Is < 5
    Range("A1").Value = "5 より小さい"
Case Is >= 20
    Range("A1").Value = "20 以上"
Case Else
    Range("A1").Value = "5 以上 20 より小さい"
```

```
End Select
```

if 文の場合と違うのは条件式の記述が「x＜5」などと書かずに「＜5」とだけ記述する事です。

繰り返し型

ここでは決められた回数や、条件を満たす間繰り返し処理を行う方法について確認していきます。

条件式が True の間繰り返す

繰り返し処理とは同じような処理を何度も記述する場合に便利な構文です。

例えば数字の 1 から始めて 2，3，4，5 と順に加えていき 10 まで加えるというプログラムを作成してみます。

単純に書けば次のようになります。

```
Dim sum As Integer
sum = 0
sum = sum + 1
sum = sum + 2
sum = sum + 3
sum = sum + 4
sum = sum + 5
sum = sum + 6
sum = sum + 7
sum = sum + 8
sum = sum + 9
sum = sum + 10
Range("A1").Value = sum
```

1 から 10 までならこのように書けますけど、1 から 100 までの合計だったら同じような文がずらずらっと並んで記述するのも大変です。このような場合に繰り返し処理を使うと便利です。

繰り返し処理を行うには色々な構文がありますが、まず「Do While ... Loop」文を見ていきましょう。構文は下記のようになります。

```
Do While 条件式
    (条件式が True の時に行う処理 1)
    (条件式が True の時に行う処理 2)
Loop
```

この構文の場合、条件式が「True」の間は「Do」と「Loop」の間に書かれた処理を繰り返し行います。

例えば先ほどの例を書き換えてみましょう。

```
Dim sum As Integer
Dim x As Integer
sum = 0
x = 1
Do While x <= 10
    sum = sum + x
    x = x + 1
Loop
```

```
Range("A1").Value = sum
```

まず変数「sum」と「x」を用意します。変数「sum」は合計した値を格納する変数、変数「x」は次に加える値を格納する変数です。
次に繰り返し処理の中身を見ていきます。まず条件式「x <= 10」を評価します。
最初の時点では変数「x」には 1 が入っていますので条件式は「True」です。よって「Do」の行と「Loop」の行の間に書かれた処理が実行されます。つまり変数「sum」に変数「x」の値が加算された値が改めて変数「sum」に格納されます。つまり変数「sum」には 1 が格納されます。そして変数「x」の値に 1 が加えられた値が変数「x」に格納されます。つまり変数「x」には 2 が格納されます。
「Do」の行と「Loop」の行の間に書かれた処理が全て実行されるとまた繰り返し処理の先頭に戻り改めて条件式が評価されます。
今度は変数「x」には 2 が格納されていますので今度も条件式は「True」です。よって今度は変数「sum」には 2 が加えられて変数「sum」の値は 3 となり、変数「x」の値は今度は 3 が格納されます。
このように何度も繰り返し処理が行われていき、変数「x」に 11 が格納された後に条件式が評価された時に「False」となります。条件式が「False」となった時に繰り返し処理は終了し「Loop」の後の行に処理が移ります。つまり「Range("A1").Value = sum」が実行されます。
繰り返し処理を使うことで 1 から 1000 まで追加するようなプログラムでも簡単に記述することが出来ます。先ほどの例で言えば条件式を「x <= 1000」に変更するだけです。

```
Sub TEST10()
    Dim sum As Integer
    Dim x As Integer
    sum = 0
    x = 1
    Do While x <= 100
        sum = sum + x
        x = x + 1
    Loop
    Range("A1").Value = sum
End Sub
```

今回のサンプルでは 1 から順に 100 まで加算した値をセルに表示しています。

条件式が False の間繰り返す

前のページでは条件を満たす間繰り返しを行う方法について見ましたが、同じような構文で条件を満たさない間繰り返しを行う構文も用意されています。

構文は下記のようになります。

```
Do Until 条件式
    (条件式が False の時に行う処理 1)
    (条件式が False の時に行う処理 2)
Loop
```

この構文の場合、条件式が「False」の間は「Do」と「Loop」の間に書かれた処理を繰り返し行います。
例えば前のページのサンプルは次のように記述することが出来ます。

```
Dim sum As Integer
Dim x As Integer
sum = 0
x = 1
Do Until x > 10
    sum = sum + x
    x = x + 1
Loop
Range("A1").Value = sum
```

動作などは「Do While ... Loop」の場合と同様です。異なるのは条件式が「False」の間だけ繰り返しが実行されるということです。

ただ、条件式が「False」の間だけ実行したい場合には論理演算子を使って次のように記述する事もできます。

```
Dim sum As Integer
Dim x As Integer
sum = 0
x = 1
Do While Not x > 10
    sum = sum + x
    x = x + 1
Loop
Range("A1").Value = sum
```

論理演算子の「Not」はその後に書かれた条件式の結果を反転させます。その為、「Do Until ... Loop」構文で記述できる内容は「Do While ... Loop」構文でも記述することが出来ます。

「Do While」と「Do Until」の両方を使うとケアレスミスの原因となりますので、条件式の記述で「False」の場合だけ実行するというような記述の方が望ましい場合には論理演算子の「Not」を使って記述したほうがいいのではと思います。

```
Sub TEST11()
    Dim sum As Integer
    Dim x As Integer
    sum = 0
    x = 1
    Do Until x > 10
        sum = sum + x
        x = x + 1
    Loop
    Range("A1").Value = sum
End Sub
```

今回のサンプルでは 1 から順に 10 まで加算した値をセルに表示しています。

1 回以上の繰り返し

次に同じような構文ですが「Do ... Loop While」構文について見ていきます。

構文は下記のようになります。

```
Do
    (条件式が True の時に行う処理 1)
    (条件式が True の時に行う処理 2)
Loop While  条件式
```

「Do While ... Loop」との違いは条件式の評価がループを一度行った後に行われるということです。つまり「Do」の行と「Loop」の行の間に書かれた処理は少なくとも必ず 1 回は実行されます。最初に 1 回実行された後は条件式を評価されて「True」だった場合には先頭に戻ってまだ処理が実行されます。

```
Sub TEST()
    Dim x As Integer
    x = 1
    Do
        x = x * 3
    Loop while x < 100
    Range("A1").Value = x
End Sub
```

あまり良い例でもないのですけど、3 を累乗していって 100 を最初に超えた数値を求めています。
また「Do ... Loop While」構文にも条件式が「False」の場合のみ繰り返しを実行する「Do ... Loop Until」構文が用意されています。

構文は下記のようになります。

```
Do
    (条件式が False の時に行う処理 1)
    (条件式が False の時に行う処理 2)
Loop Until 条件式
```

「Do ... Loop While」構文との違いは繰り返しを継続する条件が、条件式が「False」の場合になるという点だけです。

決まった回数繰り返す

今までの構文では条件式によって繰り返しを継続するかどうかを決めていました。今回は決まった回数だけ繰り返しを行う場合の構文について見ていきます。

構文は下記のようになります。

```
For 変数 = 初期値 To 最終値
    (実行する処理 1)
    (実行する処理 2)
Next 変数
```

この構文の場合、まず回数をカウントするための整数型の変数を 1 つ用意します。そしてカウントする回数の初期値の値から最終値まで順に 1 ずつ増加させていき、その回数だけ実行します。具体的に書くと次のようになります。

```
Dim i As Integer
For i = 1 To 10
    (実行する処理 1)
    (実行する処理 2)
Next i
```

この場合、カウントが 1 から 10 まで増加する間繰り返しが行われますので、10 回実行されることになります。

例えば 1 から 10 までの数を足した値を求めるプログラムは次のようになります。

```
Dim sum As Integer
Dim x As Integer
Dim i As Integer
sum = 0
x = 1
For i = 1 To 10
    sum = sum + x
    x = x + 1
Next i
```

またカウンター用の変数は、回数をカウントするだけではなく、実際にその変数に現在のカウントしている値が含まれています。その為、このカウンター用の変数に含まれている値を利用することもできます。例えば先ほどのサンプルは次のように書きかえれます。

```
Dim sum As Integer
Dim i As Integer
sum = 0
For i = 1 To 10
    sum = sum + i
Next i
```

カウンター用の変数「i」には、1 から順に 10 までの数値が繰り返しが行われる時に格納されています。つまりこのサンプルは次のようなことを行っているのと同じです。

```
Dim sum As Integer
Dim i As Integer
sum = 0
i = 1
sum = sum + i
i = 2
sum = sum + i
i = 3
sum = sum + i
i = 4
sum = sum + i
i = 5
sum = sum + i
i = 6
sum = sum + i
i = 7
sum = sum + i
i = 8
sum = sum + i
i = 9
sum = sum + i
i = 10
sum = sum + i
```

このように一定の回数繰り返す時に利用するだけではなく、ある変数の値を順に変化させて利用したい場合に便利な構文となっています。

```
Sub TEST13()
    Dim sum As Integer
    Dim i As Integer
    sum = 0
    For i = 1 To 10
        sum = sum + i
    Next i
    Range("A1").Value = sum
End Sub
```

デフォルトではカウンターは 1 ずつ増加しますが、増加する量を 1 ではなく違う値にすることもできます。この場合は次の構文を使います。

```
For 変数 = 初期値 To 最終値 Step 増加量
    (実行する処理 1)
    (実行する処理 2)
Next 変数
```

例えば 2 から開始して 10 まで増加させる場合で、増加する量が 1 ずつではなく 2 ずつ増加させる場合は次のようになります。

```
Dim i As Integer
For i = 2 To 10 Step 2
    (実行する処理 1)
    (実行する処理 2)
Next i
```

この場合カウンターに入る値は「2、4、6、8、10」の各数値となり、繰り返し処理は 5 回行われることになります。

配列

まず配列とは何かを確認しておきます。例えば変数を使っていくつかの文字列を格納する場合を考えて見てください。

```
Dim pref1 As String
Dim pref2 As String
Dim pref3 As String
Dim pref4 As String
Dim pref5 As String
Dim pref6 As String
Dim pref7 As String
pref1 = "東京都"
pref2 = "大阪府"
pref3 = "愛知県"
pref4 = "福岡県"
pref5 = "北海道"
pref6 = "鹿児島県"
pref7 = "広島県"
```

都道府県の名前を格納するために変数を用意した場合、格納したい都道府県の数だけ別々の変数が必要になります。同じ目的で利用しているのに変数名は全部異なりますし、数が多くなってきた場合変数を定義するだけでも大変です。

このように同じ型の変数を同じような目的で利用する場合に便利なのが配列です。配列は変数を 1 つ 1 つ定義するのではなく、まとめて定義することができます。例えば下記のような感じです。

```
Dim pref(6) As String
pref(0) = "東京都"
pref(1) = "大阪府"
pref(2) = "愛知県"
pref(3) = "福岡県"
pref(4) = "北海道"
pref(5) = "鹿児島県"
pref(6) = "広島県"
```

簡単に説明しますと、「pref」という名前を持つ変数に全部で 7 つの値を格納できるよう領域を確保します。それぞれの領域を区別するために変数名の後に括弧の中に番号をふって区別します。このようなものを配列といいます。そして配列の 1 つ 1 つの領域を配列の要素と言います。

配列は同じ目的で利用する変数を別々に定義するのではなく、1 つの変数名に対して複数の要素を確保することでまとめて管理するためのものです。
配列は変数の定義が簡単と言うメリットだけではありません。配列の 1 つ 1 つの要素はインデックス番号を使って区別できますので、次のようなことが可能になります。

例としてある文字列が都道府県名と同じかどうかを判別するプログラムを書くことを考えてみましょう。配列を使わない場合には例えば次のようになります。

```
Dim pref1 As String
Dim pref2 As String
Dim pref3 As String
Dim src As String
Dim msg As String
pref1 = "東京都"
pref2 = "大阪府"
pref3 = "愛知県"
src = "茨城県"
If src = pref1 Then
    msg = "一致しました"
ElseIf src = pref2 Then
    msg = "一致しました"
ElseIf src = pref3 Then
    msg = "一致しました"
End If
```

調べたい文字列を If 文を使って順に比較していきます。Select 文を使っても同じような形になります。

今度は配列を使った場合を考えてみましょう。

```
Dim pref(2) As String
Dim i As Integer
Dim src As String
Dim msg As String
pref(0) = "東京都"
pref(1) = "大阪府"
pref(2) = "愛知県"
src = "茨城県"
For i = 0 To 2
    If src = pref(i) Then
        msg = "一致しました"
    End If
Next i
```

配列の各要素はインデックス番号を変えることで最初から最後までの要素を順に取り出すことができます。インデックス番号は整数ですので、For 文を使ってインデックス番号を変えて繰り返し処理することで、配列に格納された各要素と順に比較するといったプログラムを記述する事ができます。

今回の例のように比較する対象が 3 つしかない場合にはどちらでもよさそうですけど、これが 100 個とかになった場合を考えれば配列のメリットがお分かり頂けるかと思います。

配列の宣言

配列の宣言方法です。構文は下記のようになっています。

```
Dim 変数名(配列の要素数 - 1) As データ型
```

通常の変数と異なる点は、1 つの変数に対してデータを格納できる領域を何個確保するかを指定することです。1 つ 1 つの領域を配列の要素といいますが、例えば 10 個の要素を持つ配列を作成する場合には、変数名(9)と記述します。

なぜ実際の要素数よりも 1 を引くかと言いますと、配列の要素を区分するためのインデックス番号は 0 から始まるためです。例えば 3 個の要素を持つ配列の場合はインデックス番号は「0、1、2」となります。配列の宣言の時には 0 から何番目のインデックス番号までを使うかを表す数値を指定するため、実際の要素数よりも 1 つ少ない数値を指定します。

例として String 型で要素を 10 個持つ配列変数「str」を宣言する場合は次のようになります。

```
Dim str(9) As String
```

次に配列の各要素を使う方法です。変数名自体は配列全体を表していますので、配列に含まれる 1 つ 1 つの要素は、変数名(インデックス番号)という形で使用します。

```
変数名(インデックス番号) = 格納する値
```

先に書きましたとおりインデックス番号は 0 から開始されますので、例えば次のような使い方となります。

```
Dim str(3) As String
str(0) = "山田"
str(1) = "佐藤"
str(2) = "伊藤"
```

変数(インデックス番号)で変数を表す点以外は通常の変数と同じように使うことができます。

```
Sub TEST14()
    Dim pref(3) As String
    Dim i As Integer
    Dim src As String
    Dim msg As String
    pref(0) = "東京都"
    pref(1) = "神奈川県"
    pref(2) = "千葉県"
    pref(3) = "埼玉県"
    src = "茨城県"
    msg = "関東以外の県です"
    For i = 0 To 3
        If src = pref(i) Then
            msg = "関東の県です"
        End If
    Next i
    Range("A1").Value = msg
    src = "埼玉県"
    msg = "関東以外の県です"
    For i = 0 To 3
        If src = pref(i) Then
            msg = "関東の県です"
        End If
    Next i
    Range("A2").Value = msg
End Sub
```

インデックスの範囲を変更

デフォルトの場合、配列の要素を識別するために使うインデックス番号は 0 から開始されますが、これを好きな数字に変更することが出来ます。

構文は下記のようになっています。

```
Dim 変数名(最小値 To 最大値) As データ型
```

例えばインデックス番号として 3 番から 6 番を使う場合には次のように記述します。

```
Dim str(3 To 6) As String
```

このように配列変数を宣言した場合には、各要素を識別するのに使うインデックス番号も 3 から 6 になります。

```
Dim str(3 To 6) As String
str(3) = "山田"
str(4) = "佐藤"
str(5) = "伊藤"
str(6) = "斉藤"
```

配列によって開始インデックス番号を変更すると、メンテナンスの時に混乱するかもしれませんので統一しておいた方がいいと思います。

```
Sub TEST15()
    Dim pref(3 To 6) As String
    Dim i As Integer
    Dim src As String
    Dim msg As String
    pref(3) = "東京都"
    pref(4) = "神奈川県"
    pref(5) = "千葉県"
    pref(6) = "埼玉県"
    Range("A1").Value = msg
    src = "埼玉県"
    msg = "関東以外の県です"
    For i = 3 To 6
        If src = pref(i) Then
            msg = "関東の県です"
        End If
    Next i
    Range("A2").Value = msg
End Sub
```

配列を使った場合に範囲を超えたインデックス番号を使った場合には「実行時エラー」となります。コンパイルしただけではエラーとなりませんので注意して下さい。これはコンパイルの段階では配列のインデックスが範囲内なのかどうかをチェックしていないためです。実際に実行してみて始めてエラーとなります。

プロシージャ

プロシージャはマクロとしてユーザーから直接呼び出されるものの他に、プロシージャの中から呼ばれるプロシージャがあります。ここでは Sub プロシージャ及び Function プロシージャについて確認します。

Sub プロシージャの構文は下記となっています。

```
Sub プロシージャ名()
End Sub
```

プロシージャは「Sub」で始まり「End Sub」で終わります。この間に実際のプログラムを記述します。そしてプロシージャを識別するために「Sub」の後ろにプロシージャ名を記述します。

プロシージャ名はアルファベット、ひらがな、漢字などを使えます。数字も使えますがプロシージャ名の先頭には文字を使わなければなりません。またアンダーバーは使えますが記号やスペースなどは使えません。

Sub プロシージャを定義したら、「Sub」から「End Sun」の間に実際に Excel に行わせたい処理を表すコードを記述していきます。

コードはプロシージャ内に複数記述することができます。

```
Sub プロシージャ名()
    Range("A1").Value = 10
    Range("A2").Value = 20
    Range("A3").Value = 30
End Sub
```

3 行のコードが書かれていますが、1 行毎に 1 つの処理を行っています。そして、プロシージャが呼び出された時、コードは上から順番に実行されていきます (順次型)。例えば上記の例で言えばまず「Range("A1").Value = 10」が実行され、次に「Range("A2").Value = 20」が実行され、最後に「Range("A3").Value = 30」が実行されます。

改行までが 1 つの処理となるため、例え 1 つの処理を記述するのに長くなってしまったとしても途中で改行してはいけません。例えば下記のように記述するのは誤りです。

```
Sub プロシージャ名()
    Range("A1").Value
        = 10
End Sub
```

この場合、「Range("A1").Value」と「= 10」は別の処理と見なされます。そしてどちらも文法的に間違っているためエラーとなってしまいます。

どうしても途中で改行したい場合には、処理が次の行にまたがっていることを表すため半角スペースの後にアンダーバーを記述します。

```
Sub プロシージャ名()
    Range("A1").Value _
        = 10
End Sub
```

Sub プロシージャは今まで使ってきたとおりマクロとして呼び出すため使いました。

```
Sub TEST16()
    Dim sum As Integer
    Dim i As Integer
    sum = 0
    For i = 1 To 10
        sum = sum + i
    Next i
    Range("A1").Value = sum
End Sub
```

例えば上記はとても簡単な Sub プロシージャですが、目的としては現在のワークシートのセル(A1)に、1 から 10 までを合計した値を表示するためのものです。このように Sub プロシージャはある目的を行うために記述されたプログラムの集合のようなものです。

ここで Sub プロシージャ内に記述したプログラムの中で繰り返し使われるような部分などを別の Sub プロシージャに分けることができます。例えば次のプログラムを見てください。

```
Sub TEST17()
    Dim sum As Integer
    Dim i As Integer
    sum = 0
    For i = 1 To 10
        sum = sum + i
    Next i
    Range("A1").Value = sum
    Range("A2").Value = Range("A1").Value * 2
    Range("A3").Value = Range("A2").Value * 2
End Sub
```

このプログラムは 1 から 10 までを加えた値をまずセル(A1)に表示します。その後でセル(A2)にはセル(A1)の値を 2 倍したものを表示し、セル(A3)にはセル(A2)の値を 2 倍したものを表示するものです。

このサンプルプログラムを次のように記述することが出来ます。

```
Sub TEST18()
    Dim sum As Integer
    Dim i As Integer
    sum = 0
    For i = 1 To 10
        sum = sum + i
    Next i
    Range("A1").Value = sum
    otherCellSet
End Sub

Sub otherCellSet()
    Range("A2").Value = Range("A1").Value * 2
    Range("A3").Value = Range("A2").Value * 2
End Sub
```

今度の場合は、セル(A2)とセル(A3)に値をセットする部分を別の目的を行うための 1 つのまとまりとして別の Sub プロシージャに独立させています。そしてこの別に作成した Sub プロシージャを元の Sub プロシージャ内から呼び出しています。

別のプロシージャを呼び出すときは、呼び出すプロシージャ名を記述するだけです。
「TEST18」プロシージャはマクロとしてユーザーから直接呼び出されるために作成していますが、「otherCellSet」プロシージャは「TEST18」プロシージャから呼び出されるために作成しています。もちろん「otherCellSet」プロシージャと「TEST18」プロシージャは構成的に違いはありませんので「otherCellSet」プロシージャもユーザーから直接呼び出すことも可能です。
このようにプロシージャ内から別のプロシージャを呼び出すことも可能です。

別のプロシージャに分けるメリット
別のプロシージャに分けるメリットはいくつかあります。
1 つ目は同一のプロシージャ内で同じ処理が何度も発生する場合です。例えばある複雑な計算を何度もする場合、同じコードを何度も記述する代わりに計算だけをするプロシージャを用意しておき、必要に応じて呼び出すようにすれば 1 度プロシージャを記述するだけで済みます。(別のページで実際に試してみます)。

2 つ目は 1 つのプロシージャ内にあまりにも多くのプログラムが記述されていると分かりにくくなってしまうため、個々の目的別にプロシージャを分けることで、この部分は何を目的としたプログラムなのかということが明確になることです。

3 つ目は複数のプロシージャで同じことをするプログラムがあった場合、共通のプログラムを記述した汎用的なプロシージャを 1 つ用意して、各プロシージャから呼び出すことで全体のコード量を削減することも出来ますし、新規にプログラムを作成する場合に既に用意してある小さな目的別のプロシージャを利用することで開

発期間が短くすることができます。

ここでは複数のプロシージャから共通して利用される汎用プログラムのサンプルを試してみます。

```
Sub TEST19()
    Dim sum As Integer
    Dim i As Integer
    sum = 0
    For i = 1 To 10
        sum = sum + i
    Next i
    Range("A1").Value = sum
    otherCellSet
End Sub

Sub TEST20()
    Dim multiply As Integer
    Dim i As Integer
    multiply = 1
    For i = 1 To 5
        multiply = multiply * i
    Next i
    Range("A1").Value = multiply
    otherCellSet
End Sub

Sub otherCellSet()
    Range("A2").Value = Range("A1").Value * 2
    Range("A3").Value = Range("A2").Value * 2
End Sub
```

上記にはユーザーから呼び出される「TEST19」プロシージャと「TEST20」プロシージャがあり、そしてこの2つのプロシージャから呼び出される「otherCellSet」プロシージャがあります。

まず「TEST19」プロシージャをマクロとして呼び出してみます。
「TEST19」プロシージャではまず 1 から 10 までを加えた数をセル(A1)に表示し、その後で「otherCellSet」プロシージャを呼び出しセル(A2) とセル(A3)に値を表示しています。

次に「TEST20」プロシージャをマクロとして呼び出してみます。
「TEST20」プロシージャではまず 1 から 5 までを掛け算した数をセル(A1)に表示し、その後で「otherCellSet」プロシージャを呼び出しセル(A2) とセル(A3)に値を表示しています。
このように共通して利用されるプログラムは別のプロシージャとして分離してうえで、色々なプロシージャから呼び出す事が出来ます。

別のプロシージャを呼び出した時の処理の流れ

他のプロシージャを呼び出した時の処理の流れがどうなるのかを確認しておきます。次のサンプルで見ていきます。

```
Sub TEST21()
    Range("A1").Value = "東京都"
    sample
    Range("A3").Value = "福岡県"
End Sub

Sub sample()
    Range("A2").Value = "大阪府"
End Sub
```

上記のサンプルは、まず「Range("A1").Value = "東京都"」が実行されます。

そして「sample」プロシージャが呼び出されますのでいったん「sample」プロシージャへ処理が移り「Range("A2").Value = "大阪府"」が実行されます。

「sample」プロシージャ内の処理が全て終わると元々プロシージャを呼び出した「sample」と書かれた行の次の行へ処理が移りますので「Range("A3").Value = "福岡県"」が実行されます。

このように別のプロシージャを呼び出した場合、いったん呼び出したプロシージャへ処理が移りますが、そのプロシージャ内の処理が全て終わると、元のプロシージャ内で別のプロシージャを呼び出した行の次の行へ処理が移ります。

他のプロシージャを呼び出す場合は、単に呼び出したいプロシージャ名を記述するだけで呼び出す事が出来ます。

```
Sub TEST22()
    Dim sum As Integer
    Dim i As Integer
    sum = 0
    For i = 1 To 10
        sum = sum + i
    Next i
    Range("A1").Value = sum
    otherCellSet
End Sub

Sub otherCellSet()
    Range("A2").Value = Range("A1").Value * 2
    Range("A3").Value = Range("A2").Value * 2
End Sub
```

この場合、プロシージャ名だけが書かれているのでプロシージャを呼び出しているのかどうか一見すると分かりにくいです。(プロシージャ名を、プロシージャ名であるとはっきり分かるような名前の付け方をしておけば

いいかもしれません)。

そこでプロシージャを呼び出す別の方法を見ていきます。具体的には「Call」ステートメントを使って別のプロシージャを呼び出します

```
Sub TEST23()
    Dim sum As Integer
    Dim i As Integer
    sum = 0
    For i = 1 To 10
        sum = sum + i
    Next i
    Range("A1").Value = sum
    Call otherCellSet
End Sub

Sub otherCellSet()
    Range("A2").Value = Range("A1").Value * 2
    Range("A3").Value = Range("A2").Value * 2
End Sub
```

「Call」ステートメントの後にプロシージャ名を記述することでプロシージャを呼び出すことができます。わざわざ「Call」を付けるのは無駄なようですけど、別のプロシージャを呼び出していることがはっきりと分かりますので、後でメンテナンスをする場合にプログラムが見やすくなります。出来るだけ分かりやすく記述することが望ましいので、今後は「Call」を使って呼び出すことにします。

値渡しで引数を渡す

前のページのサンプルでは単にプロシージャの中から別のプロシージャを呼び出していました。実はプロシージャを呼ぶときに値を渡す事ができます。

例えば 10 が 2 で割り切れるかどうかを表示するサブプロシージャを考えてみます。

```
Sub TEST24()
    Call warikireCheck
End Sub

Sub warikireCheck()
    Dim num As Integer
    num = 10
    If num Mod 2 = 0 Then
        Range("A1").Value = "割り切れます"
    Else
        Range("A1").Value = "割り切れません"
    End If
End Sub
```

このプロシージャを呼び出せば確かに 10 が 2 で割り切れるかどうかを判別できますが、それ以外の使い道はありません。10 という数値が固定に書かれているからです。

このプロシージャを、呼び出す時に指定した数値が 2 で割り切れるかどうかを判別するプロシージャに変更してみます。

```
Sub TEST25()
    Call warikireCheck2(10)
End Sub

Sub warikireCheck2(ByVal num As Integer)
    If num Mod 2 = 0 Then
        Range("A1").Value = "割り切れます"
    Else
        Range("A1").Value = "割り切れません"
    End If
End Sub
```

今度のサンプルでは「warikireCheck2」プロシージャを呼び出す時に、あわせて 10 と言う数値を指定しています。

```
Call warikireCheck2(10)
```

このようにプロシージャ名呼び出す時に、括弧()で囲んだ中に値や変数などを記述すると、プロシージャに値を渡す事ができます。つまりこの場合は「warikireCheck2」プロシージャを呼び出すと同時に「warikireCheck2」プロシージャに 10 という値を渡しています。

※「Call」を使わずにプロシージャを呼び出す場合には括弧を使わずにプロシージャ名の後にスペースを 1 つ入れてその後ろに値や変数を記述します。

```
warikireCheck2 10
```

呼び出された方の処理

次に値が渡されて来た方のプロシージャ側の処理です。呼び出し側から値が渡されて来ますのでそれを受け取らなくてはなりません。

値を受け取る側では、渡され来た値を格納するための変数を用意します。構文は次のようになります。

```
Sub プロシージャ名(ByVal 変数名 As データ型)
```

プロシージャ名の後の括弧()の中に渡されてきた値を受け取る変数を宣言します。また変数の先頭には「ByVal」を付けます。ここで「ByVal」とは値渡しという意味です。こちらは後で説明します。また値を受け取るための変数を引数とも言います。

先ほどのサンプルでは次のように記述していました。

```
Sub warikireCheck2(ByVal num As Integer)
```

これは Integer 型の変数「num」にプロシージャ呼び出しの際に一緒に渡されてきた値を格納する、と言う事です。つまり変数「num」には 10 という値が格納されます。この変数「num」はプロシージャ内で通常の変数と同じように使うことが出来ます。そこで、この渡されてきた値が 2 で割り切れるかどうかをその後のプログラムで行っています。

```
Sub warikireCheck2(ByVal num As Integer)
    If num Mod 2 = 0 Then
        Range("A1").Value = "割り切れます"
    Else
        Range("A1").Value = "割り切れません"
    End If
End Sub
```

このように呼び出し元から呼び出し先へ値を渡すことで、色々な用途に使うことが出来るプロシージャを作成する事が出来ます。

呼び出す時に渡す事ができる値は数値だけではなく、文字列やオブジェクトなどどのようなデータ型の値も渡す事ができます。次のサンプルは文字列を別のプロシージャに引数として渡しています。

```
Sub TEST26()
    Dim str As String
    str = "こんにちは"
    Call setCellValue(str)
End Sub

Sub setCellValue(ByVal str As String)
    str = str & "お元気ですか"
    Range("A1").Value = str
End Sub
```

注意点としては、呼び出し側で指定した値のデータ型と、受け取る側の引数のデータ型は同じでなければなりません。呼び出し側で Integer 型の値を記述したのに、受け取る側で String 型として受け取るようなことはできません。

またプロシージャを呼び出す時に引数として変数を指定していますが、変数に格納された値が実際にはプロシージャに渡されます。値だけが渡されますので、呼び出されたプロシージャ側で値を変更したりしても呼び出し元の変数「str」の値には呼び出す前と後で何の影響もありません。

つまり呼び出された「setCellValue」プロシージャ内の変数「str」は"こんにちはお元気ですか"という値にプロシージャ内で変更されていますが、呼び出し元の「テスト」プロシージャ内の変数「str」は相変わらず「こんにちは」が格納されていることになります。

このように引数に変数を指定した場合に、実際には変数に含まれている値だけが呼び出し側へ渡される場合を「値渡し」と言います。

```
Sub TEST27()
    Dim str As String
    str = "こんにちは"
    Call setCellValue(str)
    Range("A2").Value = str
End Sub

Sub setCellValue(ByVal str As String)
    str = str & "お元気ですか"
    Range("A1").Value = str
End Sub
```

複数の引数を渡す

他のプロシージャを呼び出す場合に引数として値を渡す事ができることは前のページで確認しましたが、引数は 1 つだけではなく複数の引数を渡す事ができます。

複数の値を引数としてプロシージャに渡す場合には、引数の箇所にカンマ(,)で区切って並べて記述します。

```
Call プロシージャ名(値 1, 値 2, ...)
```

※「Call」を使わない場合は次のようになります。

```
プロシージャ名 値 1, 値 2, ...
```

値を受け取る側でも同じように渡されてくるはずの引数の数だけ変数を宣言しておきます。変数と変数の間はカンマ(,)で区切ります。

```
Sub プロシージャ名(ByRef 変数名 1 As データ型 1, ByRef 変数名 2 As データ型 2, ...)
```

引数は全て同じデータ型である必要はありませんので、色々なデータ型の値を同時に別のプロシージャに渡すことが可能です。

```
Sub TEST28()
    Dim str As String
    str = "こんにちは"
    Call setCellValue2(str, 2)
End Sub

Sub setCellValue2(ByVal str As String, ByVal count As Integer)
    Dim dispStr As String
    Dim i As Integer
    For i = 1 To count
        dispStr = dispStr & str
    Next i
    Range("A1").Value = dispStr
End Sub
```

今回のサンプルでは最初に引数に表示したい文字列を、2 番目の引数に繰り返す回数を指定しています。

参照渡しで引数を渡す

今までの方法ですと、呼び出すときに値を渡す事はできますが、逆に呼び出した方に値を返す事ができません。呼び出し元に値を返す事が出来れば、複雑な計算を行うようなプロシージャを用意しておき、結果を返してくれるようなプロシージャを作成する事が出来ます。

値を呼び出し元に返す方法は二通りあり、参照私を使う方法と戻り値を使う方法です。ここでは参照渡しを見ていきます。
まず定義を見ておきます。プロシージャを呼び出す方は変更がありませんが、呼び出される方のプロシージャにて、引数の定義の仕方が異なります。

```
Sub プロシージャ名(ByRef 変数名 As データ型)
```

値渡しの場合は「ByVal」を変数定義の前に使っていましたが、参照渡しの場合は「ByRef」を変数定義の前に付けます。
今まで使っていた値渡しによる方法では、プロシージャを呼び出すときに変数を使って引数を指定した場合、変数にその時点で含まれている値だけがプロシージャに渡されていました。その為、呼び出し先のプロシージャで値を変更しても呼び出し元の方の変数には影響がありませんでした。

```
Sub TEST29()
    Dim str As String
    str = "こんにちは"
    'この時点では変数 str には"こんにちは"が
    '格納されています
    Call setCellValue(str)
    '呼び出しから帰った時点でも変数 str には
    '"こんにちは"が格納されたままです
End Sub

Sub createString(ByVal str As String)
    str = "こんばんは"
    '呼び出されたプロシージャで変数 str には
    '"こんばんは"が格納されます
End Sub
```

参照渡しを使うと、プロシージャを呼び出したときに変数を使って引数を指定した場合、変数に格納されている値ではなく変数そのものがプロシージャに渡されます。その為、呼び出された側のプロシージャにて変数の値を変更すると、呼び出し元のプロシージャの変数に格納されている値も変更されます。

```
Sub TEST29()
    Dim str As String
    str = "こんにちは"
    'この時点では変数 str には"こんにちは"が
    '格納されています
    Call setCellValue(str)
    '呼び出しから帰った時点では変数 str には
    '"こんばんは"が格納されています
End Sub

Sub createString(ByRef str As String)
    str = "こんばんは"
    '呼び出されたプロシージャで変数 str には
    '"こんばんは"が格納されます
End Sub
```

このように参照渡しを使うことで、呼び出されたプロシージャで変更された状態を呼び出し元でも利用することが出来るようになります。

```
Sub TEST30()
    Dim str As String
    str = "伊藤さん"
    Call createString2(str)
    Range("A1").Value = str
End Sub

Sub createString2(ByRef str As String)
    str = str & "、こんにちは"
End Sub
```

Function プロシージャ

プロシージャには Sub プロシージャだけではなく、Function と言うプロシージャが用意されています。Function プロシージャの大きな特徴は呼び出し元に値を戻す事ができることです。その為、あたかも関数のように動作しますのでユーザー定義関数とも言います。
例えば複雑な計算をするプロシージャを用意し、実行した結果を受け取るような事が出来るわけです。

Function プロシージャの構文は次のようになっています。

```
Function プロシージャ名(引数 As データ型) As 戻り値のデータ型
    プロシージャ名 = 戻り値
End Function
```

プロシージャ内で様々な処理を行った後で、プロシージャ名に値を格納することで呼び出し元に値を返す事ができます。この返される値を戻り値と言います。例えば Integer 型の引数を 1 つ受け取り、戻り値として String 型の値を返すようなプロシージャは次のようになります。

```
Function hantei(ByVal tokuten As Integer) As String
    Dim kekka As String
    If tokuten >= 80 Then
        kekka = "合格"
    Else
        kekka = "不合格"
    End If
    hantei = kekka
End Function
```

次に呼び出し元の方です。Function プロシージャを呼び出すと同時に、呼び出した Function プロシージャから戻ってくる値を受け取るように記述しなければなりません。

```
Sub TEST()
    Dim kekka As String
    kekka = hantei(75)
End Sub
```

上記の場合、「hantei」プロシージャを呼び出すと、「hantei」プロシージャ内で処理が行われた結果、値が戻ってきます。その戻ってきた値が変数「kekka」に格納されるわけです。
このように Fuction プロシージャを使うことで、自分で関数を定義するように何か処理を行った結果を返す事ができるプロシージャを定義することが出来ます。

```
Sub TEST31()
    Dim kekka As String
    kekka = hantei(75)
    Range("A1").Value = kekka
    kekka = hantei(92)
    Range("A2").Value = kekka
End Sub

Function hantei(ByVal tokuten As Integer) As String
    Dim kekka As String
    If tokuten >= 80 Then
        kekka = "合格"
    Else
        kekka = "不合格"
    End If
    hantei = kekka
End Function
```

ダイアログ

メッセージを表示したり、ユーザーからの入力を行うためのダイアログボックスの使い方について見ていきます。

メッセージの表示

まずはメッセージを表示するためのダイアログについて見てみます。構文は下記のようになっています。

```
MsgBox 表示文字列, ボタンの種類, タイトル文字列
```

上記は単にメッセージを表示し、ボタンが 1 つだけある場合の構文です。

ダイアログに表示されているどのボタンを押したかなど、ダイアログからの結果を取得する場合には次のように記述します。

```
Dim ans As Integer
ans = MsgBox (表示文字列, ボタンの種類, タイトル文字列)
```

ダイアログでどのボタンが押されたかは Integer 型の値とをして取得する事が出来ます。

ダイアログに表示する文字列、どのようなボタンを表示するかの指定、ダイアログのウィンドウタイトル、の 3 つを指定して作成します。ボタンの種類とタイトル文字列は省略可能で省略した場合はデフォルトの値が使われます

```
Sub TEST32()
    MsgBox "ダイアログテスト", vbOKOnly, "タイトル"
End Sub
```

ボタンの種類

それではダイアログに表示するボタンの種類の指定方法について見ていきます。単にメッセージや注意を表示したいのであればダイアログを閉じるための「OK」ボタンが 1 つあればいいですが、ファイル削除の確認などの確認を取るためのものであれば「はい」や「いいえ」などのボタンが必要となります。

どのようなボタンがあるダイアログにするかは既に VBA で定義されています。

定義済み定数	実際の値	説明
vbOKOnly	0	OK
vbOKCancel	1	OK・キャンセル
vbAbortRetryIgnore	2	中止・再試行・無視
vbYesNoCancel	3	はい・いいえ・キャンセル
vbYesNo	4	はい・いいえ
vbRetryCancel	5	再試行・キャンセル

定義済み定数のどれかの値を「MsgBox」関数を呼び出す時の 2 番目の引数に指定します。

「vbOKOnly」以外を選択した場合はボタンが 2 つ以上ありますので、どのボタンが押されたかを判定する必要があります。帰ってくる値は Integer 型の値として取得でき、それぞれのボタンを押した場合に帰ってくる値の一覧は次の通りです。

定義済み定数	実際の値	説明
vbOk	1	OK
vbCancel	2	キャンセル
vbAbort	3	中止
vbRetry	4	再試行
vbIgnore	5	無視
vbYes	6	はい
vbNo	7	いいえ

例えば「vbOKCancel」ボタンを使ったダイアログを表示した場合、ボタンは「OK」か「キャンセル」ですので、実際に使う場合には次のようになります。

```
Sub TEST33()
    Dim ans As Integer
    ans = MsgBox("実行しますか？", vbOKCancel, "テスト")
    If ans = vbOK Then
        Range("A1").Value = "OK が押されました"
    Else
        Range("A1").Value = "キャンセルが押されました"
    End If
End Sub
```

では一通り試してみましょう。

vbOKCancel

```
Sub TEST34()
    Dim ans As Integer
    ans = MsgBox("実行しますか？", vbOKCancel, "テスト")
    If ans = vbOK Then
        Range("A1").Value = "OK が押されました"
    Else
        Range("A1").Value = "キャンセルが押されました"
    End If
End Sub
```

vbAbortRetryIgnore

```
Sub TEST35()
    Dim ans As Integer
    ans = MsgBox("失敗しました。再度実行しますか？", vbAbortRetryIgnore, "テスト")
End Sub
```

vbYesNoCancel

```
Sub TEST36()
    Dim ans As Integer
    ans = MsgBox("ファイルを削除します。宜しいですか？", vbYesNoCancel, "テスト")
End Sub
```

vbYesNo

```
Sub TEST37()
    Dim ans As Integer
    ans = MsgBox("アプリケーションを終了します。宜しいですか？", vbYesNo, "テスト")
End Sub
```

vbRetryCancel

```
Sub TEST38()
    Dim ans As Integer
    ans = MsgBox("ファイルの読み込みに失敗しました", vbRetryCancel, "テスト")
End Sub
```

メッセージを改行する

ダイアログに表示されるメッセージを任意の位置で改行する方法を確認します。

改行するには改行を表す値をメッセージの中に記述します。これらは VBA で定義されています。

定義済み定数	実際の値	説明
vbCr	Chr(13)	キャリッジリターン
vbLf	Chr(10)	ラインフィード
vbCrLf	Chr(13)+Chr(10)	キャリッジリターンとラインフィードの組み合わせ
vbNewLine	Chr(13) + Chr(10) または Chr(13)	プラットフォームで指定した改行文字。現在のプラットフォームで適切ないずれかを使用します。

この中の「vbNewLine」を使えばいいかと思います。

```
Sub TEST39()
    Dim msg As String
    msg = "こんにちは" & vbNewLine & "お元気ですか"
    MsgBox msg, vbOKOnly + vbInformation, "テスト"
End Sub
```

入力ボックス付きダイアログを表示

今までのダイアログでは事前に設定したボタンを押してもらうしかできませんでしたが、入力用のテキストボックスを表示し、ユーザーに値を入力してもらうダイアログを表示することも可能です。

構文は下記のようになっています。

```
Dim ans As String
str = InputBox(表示文字列, タイトル文字列, デフォルト値)
```

ダイアログに表示される文字列とタイトルは MsgBox と同じですが、入力用テキストボックスにデフォルトで表示される文字列を指定します。また InputBox を使った場合にはボタンは「OK」と「キャンセル」の 2 つが表示されます。

またユーザーがテキストボックスに入力した文字は、「OK」ボタンが押された時に String 型の値として取得できます。「キャンセル」ボタンやウィンドウの右上にある「×」ボタンでダイアログを閉じた場合には空の文字列("")が戻ってきます。

```
Sub TEST40()
    Dim ans As String
    ans = InputBox("お名前は？", "年齢確認", "")
    If ans <> "" Then
        Range("A1").Value = ans
    End If
```

```
End Sub
```

Range オブジェクトの取得

セルを表すオブジェクトは Range オブジェクトになります。ここでは Range オブジェクトを取得する方法について確認します。

セルの参照

セルに対する様々な操作を行う上で、対象となるセルをオブジェクトとして取得することがまず必要になります。

セルを表すオブジェクトは Range オブジェクトです。Range オブジェクトを取得する方法はいくつかありますが、まずは元になっている Worksheet オブジェクトなどのプロパティを使って Range オブジェクトを取得する方法です。使用するプロパティは「Range」プロパティです。

```
Dim range1 As Range
Set range1 = オブジェクト.Range("A1")
range1.Value = 10
```

「Range」プロパティの括弧の中にダブルクオーテーション("")で囲んだ中に Excel の A1 形式でセルの位置を記述することで、対象オブジェクトに含まれる Range オブジェクトを取得することができます。(オブジェクトは Worksheet の他に、既にある Range オブジェクトなどでも使えます)。

今までのサンプルで使っていたのはオブジェクトを省略し(省略すると現在アクティブのワークシートになります)、Range 型の変数を別途用意せずにまとめて記述していました。

```
Range("A1").Value = 10
```

普段はあまり気にする必要はありませんが、「Range("A1")」と書かれていた場合は、これが Range オブジェクトそのものではなく、ワークシートなどのオブジェクトの Range プロパティであり、このプロパティによって Range オブジェクトが取得出来ているという点だけ覚えておいてください。

```
Sub TEST41()
    Dim range1 As Range
    Set range1 = Range("A1")
    range1.Value = 10
    Range("C3").Value = "日本語"
End Sub
```

連続したセルの参照

Range オブジェクトは単一のセルだけではなく、連続した領域のセルをまとめて管理することができます。

連続した領域のセルを現す Range オブジェクトを取得するには、「Range」プロパティの引数を「"左上のセル:右下のセル"」の形式で指定します。

```
Dim range1 As Range
Set range1 = Range("A1:C3")
range1.Value = 10
```

上記の場合、左上がセル A1、右下がセル C3 の長方形の領域を表す Range オブジェクトを取得します。

まとめて次のように記述しても構いません。

```
Range("A1:C3").Value = 10
```

また別の記述の仕方として「"左上のセル", "右下のセル"」の形式でも指定できます。

```
Range("A1", "C3").Value = 10
```

この場合も左上がセル A1、右下がセル C3 の長方形の領域を表す Range オブジェクトを取得します。

```
Sub TEST42()
    Dim range1 As Range
    Set range1 = Range("A1:C3")
    range1.Value = 10
    Range("D5", "E6").Value = "日本語"
End Sub
```

離れたセルの参照

Range オブジェクトは、離れたセルをまとめて管理することができます。
離れたセルを現す Range オブジェクトを取得するには、「Range」プロパティの引数を「"セル 1, セル 2, セル 3, ..."」の形式で指定します。

```
Dim range1 As Range
Set range1 = Range("A1, C2, D5")
range1.Value = 10
```

上記の場合、セル A1、セル C2、セル D5 のそれぞれ単独のセルをまとめて表す Range オブジェクトを取得します。

まとめて次のように記述しても構いません。

```
Range("A1, C2, D5").Value = 10
```

またそれぞれのセルは単独のセルだけではなく、セル範囲を指定することも可能です。

```
Range("A1, C3:D5").Value = 10
```

この場合は、セル A1 とセル範囲 C3:D5 をまとめて表す Range オブジェクトを取得します。

```
Sub TEST43()
    Dim range1 As Range
    Set range1 = Range("A1, C3:D5")
    range1.Value = 10
End Sub
```

単一セルの参照

前のページではワークシートなどの Range プロパティを使って Range オブジェクトを取得する方法を見てみましたが、同じようにワークシートなどの Cells プロパティを使っても Range オブジェクトを取得できます。

```
オブジェクト.Cells(行番号, 列番号)
```

Range オブジェクトを取得したいセルの行番号、及び列番号を数値にて指定します。行番号及び列番号はそれぞれ「1」から開始されます。よって左上のセルは「Cells(1, 1)」となります。

オブジェクトには Worksheet オブジェクトや Range オブジェクトなどが指定できます。省略した場合は、現在のアクティブなワークシートになります。
使い方は次のようになります。

```
Dim range1 As Range
Set range1 = Cells(2, 3)
range1.Value = 10
```

Cells プロパティは単一のセルを取り扱う時に使います。Range プロパティでも同じことが出来るのですが、Cells プロパティの大きな特徴はセルの位置を数値で指定可能なことです。数値で指定できることによって、繰り返し処理などの中でセルの位置を動的に変更して処理などが行いやすくなります。

Range プロパティを使った場合と同じように、いったん Range オブジェクトを作成せずに次のように記述することも可能です。

```
Cells(2, 3).Value = 10
```

```
Sub TEST44()
    Dim i As Integer
    For i = 1 To 7
        Cells(i, 2).Value = i
    Next i
End Sub
```

Cells プロパティを使うと、数値でセルの位置が指定できるためプログラム中で動的にセルの位置を変更したい場合などに便利です。

値が含まれる最後のセルの取得

基準となるセルの位置から指定した方向へ移動させ値が含まれている最後のセルを取得する方法を見ていきます。例えば下の方向へセルを順次見ていき、空白のセルが現れる一つ前のセルを取得します。

取得するには基準の位置となる Range オブジェクトに対して「End」プロパティを使います。

```
Dim range1 As Range
Set range1 = Range("A1").End(xlDown)
```

「End」プロパティを参照すると Range オブジェクトを取得できます。「End」プロパティの引数にどちらの方

向へ調べるのかを指定します。

指定できる値は以下の通りです。

定数	方向
xlDown	下
xlToRight	右
xlToLeft	左
xlUp	上

例えば右方向にする指定は「xlToRight」を設定します。

セル C3 を基準として、下方向の最後のセルと右方向の最後のセルを取得してみます。

```
Sub TEST45()
    Dim range1 As Range
    Set range1 = Range("C3").End(xlDown)
    range1.Value = "下端のセル"
    Set range1 = Range("C3").End(xlToRight)
    range1.Value = "右端のセル"
End Sub
```

セル C6 がセル C3 から見て下方向の最後のセル、セル E3 がセル C3 からみて右方向の最後のセルとなります。

	A	B	C	D	E
1					
2		氏名	性別	年齢	出身地
3		山田	男性	25	鹿児島
4		伊藤	女性	35	福岡
5		高橋	男性	20	大分
6		加藤	女性	29	小倉
7					

	A	B	C	D	E
1					
2		氏名	性別	年齢	出身地
3		山田	男性	25	右端のセル
4		伊藤	女性	35	福岡
5		高橋	男性	20	大分
6		加藤	下端のセル	29	小倉
7					

指定したオフセットだけ移動したセルの取得

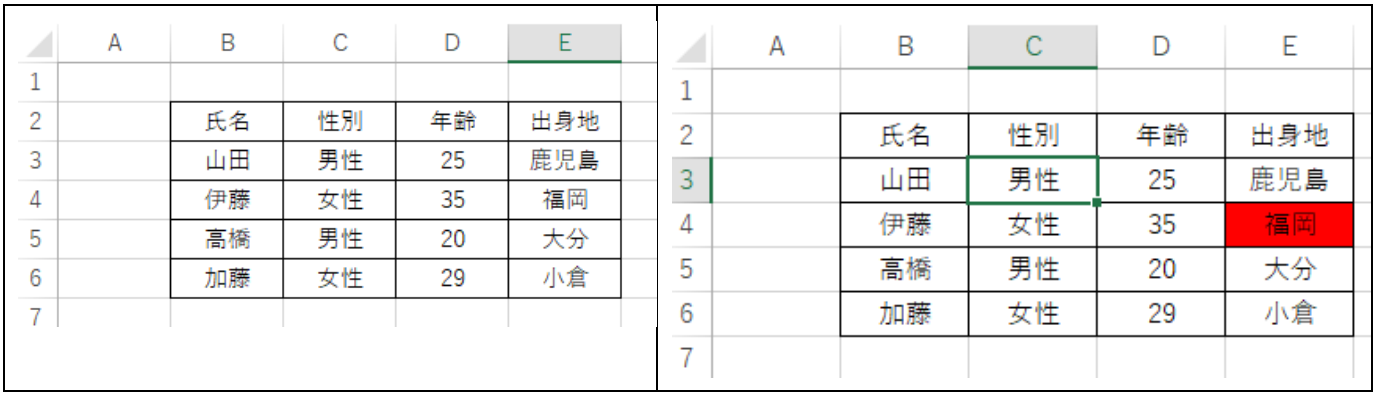
基準となるセルの位置から指定したオフセット分だけ移動したセルを取得する方法を見ていきます。行方向と列方向へそれぞれオフセットを指定します。

取得するには基準の位置となる Range オブジェクトに対して「Offset」プロパティを使います。

```
Dim range1 As Range
Set range1 = Range("A1").Offset(RowOffset:=3, ColumnOffset:=3)
```

「Offset」プロパティを参照すると Range オブジェクトを取得できます。「Offset」プロパティの引数には省略可能な 2 つの引数を指定します。「RowOffset」には行方向のオフセットを「ColumnOffset」には列方向のオフセットをそれぞれ正又は負又は 0 で指定します。デフォルトは 0 です。

```
Sub TEST46()
    Dim range1 As Range
    Set range1 = Range("C3").Offset(1,2)
    range1.Interior.ColorIndex = 3
End Sub
```



	A	B	C	D	E
1					
2		氏名	性別	年齢	出身地
3		山田	男性	25	鹿児島
4		伊藤	女性	35	福岡
5		高橋	男性	20	大分
6		加藤	女性	29	小倉
7					

	A	B	C	D	E
1					
2		氏名	性別	年齢	出身地
3		山田	男性	25	鹿児島
4		伊藤	女性	35	福岡
5		高橋	男性	20	大分
6		加藤	女性	29	小倉
7					

値と計算式の設定

セルに値や計算式を設定したり、既に設定されている値などを取得する方法を見ていきます。

値と計算式の設定

セルに値を設定する場合、Range オブジェクトを取得した後で「Value」プロパティを使ってセルに値を設定したり取得したりすることが出来ます。(今まで見てきた通りです)。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.Value = 10
```

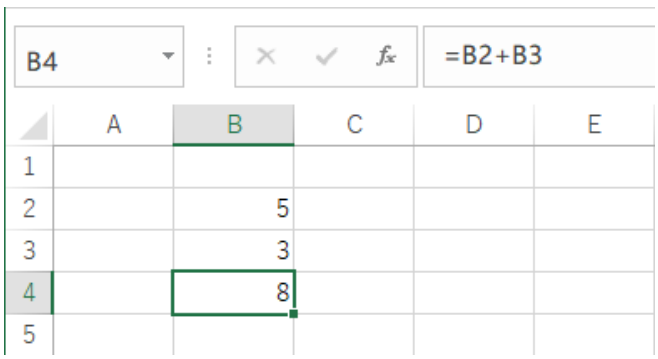
また Range オブジェクトを取り出す部分を省略して次のようにも記述可能です。

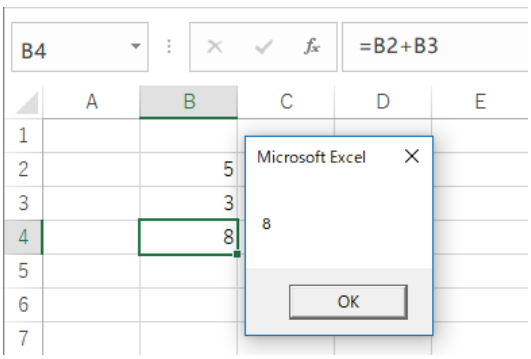
```
Range("A1").Value = 10
Cells(2, 2).Value = "月曜日"
```

注意すべき点としては「Value」プロパティを使う場合には、セルに数式などが設定されていた場合でも、数式ではなく値が取得されます。

セル("B4")には計算式である「=B2+B3」が含まれており、その結果として「8」という値が表示されています。ではセル("B4")の Range オブジェクトを取得し、Value プロパティで取得できる値を確認してみます。

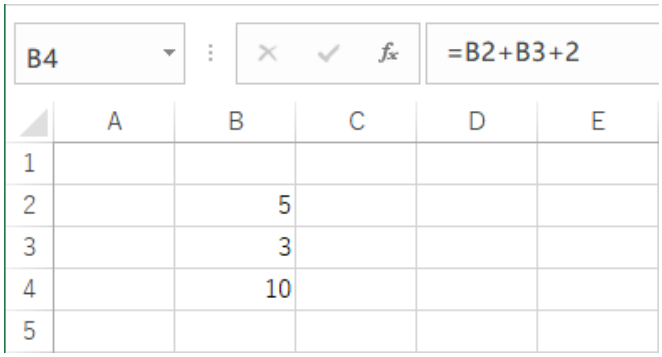
```
Sub TEST47()
    MsgBox Range("B4").Value
End Sub
```





このように「Value」プロパティを使った場合には「式」ではなく「値」が取得されます。
では逆に「Value」プロパティを使って「式」を設定してみます。

```
Sub TEST48()
    Range("B4").Value = "=B2+B3+2"
End Sub
```



このように「Value」プロパティにセットする場合は、「式」でもセットできるようです。
読むときは「値」になり、設定する時は「値」でも「式」でも可能なのですが、セルに設定されている「式」を読み取ったり「式」を設定するために「Formula」プロパティが別途用意されていますので、「Value」プロパティを使う時は読み込みも設定も「値」に統一しておいたほうがいいかもしれません。

式の設定と取得

セルに数式を設定したり、セルに設定されている数式を取得するには、Range オブジェクトの「Formula」プロパティを使います。

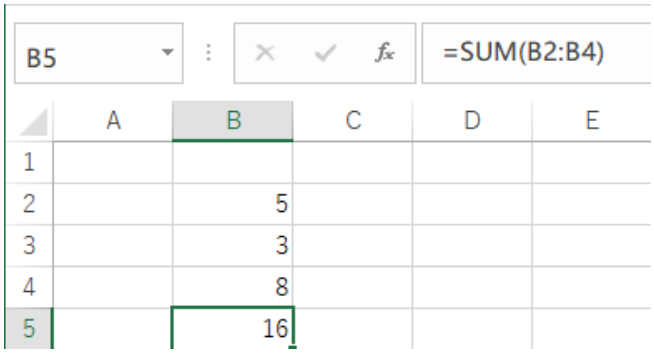
```
Dim range1 As Range
Set range1 = Range("B5")
range1.Formula = "=Sum(B2:B4)"
```

また Range オブジェクトを取り出す部分を省略して次のようにも記述可能です。

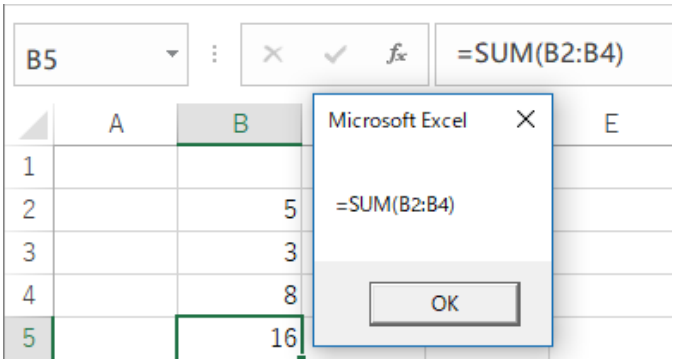
```
Range("B5").Formula = "=Sum(B2:B4)"
```

注意すべき点としては「Value」プロパティを使う場合には、セルに数式などが設定されていた場合でも、数式ではなく値が取得されます。

```
Sub TEST49()
    Dim range1 As Range
    Set range1 = Range("B5")
    'range1.Formula = "=Sum(B2:B4)"
    Cells(5, 2).Formula = "=Sum(B2:B4)"
End Sub
```




```
Sub TEST50()
    MsgBox Cells(5, 2).Formula
End Sub
```



このように「Formula」プロパティを使った場合には「値」ではなく「式」が取得されます。

セルの表示形式の設定

セルに表示される値の表示形式の設定方法について確認します。

NumberFormatLocal プロパティ

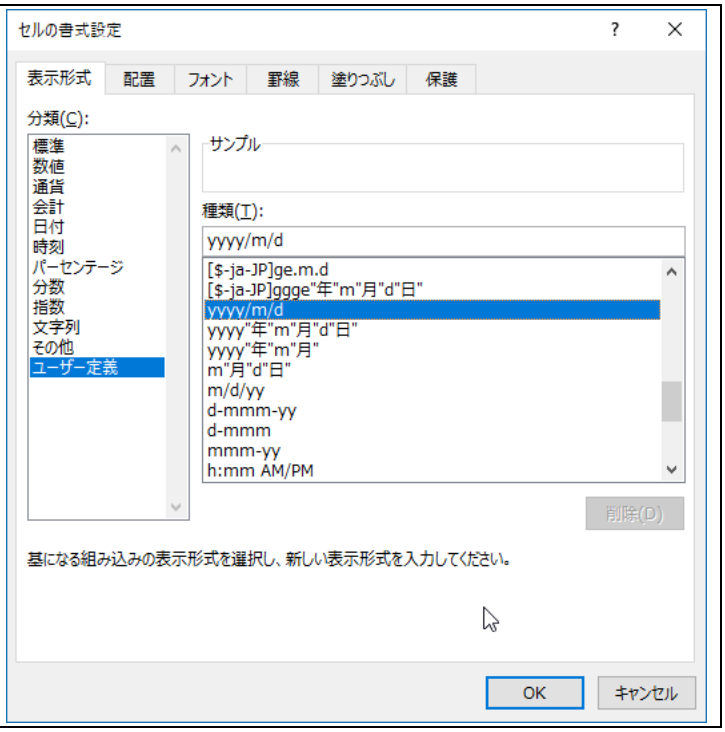
セルに含まれる値の表示形式は Range オブジェクトの「NumberFormatLocal」プロパティを使って管理しています。

```
Dim range1 As Range
Set range1 = Range("B2")
range1.NumberFormatLocal = "yyyy/mm/dd"
```

まとめて次のように記述しても構いません。

```
Range("B2").NumberFormatLocal = "yyyy/mm/dd"
```

「NumberFormatLocal」プロパティに設定できる値は Excel の「セルの書式設定」の「表示形式」で指定できるものと同じものになります。



大きく分ければ数値、日付、時刻の表示形式の設定となります。それぞれ詳しく見ていきます。

数値の書式

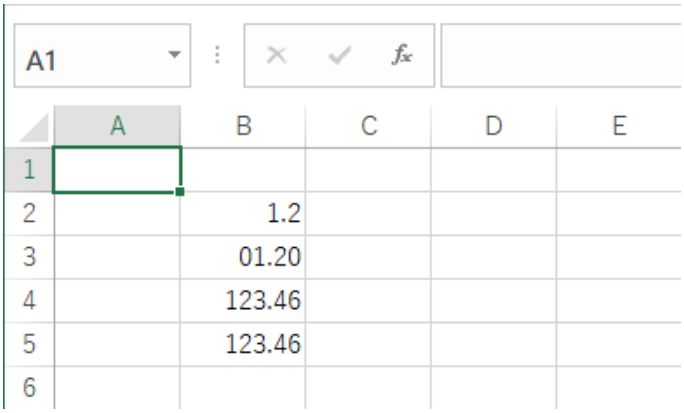
数値の書式の基本は「#」と「0」です。小数点は「.」、桁区切りは「,」を使います。

まず「#」と「0」の違いは次の例を見てください。

書式	対象	表示
##.##	1.2	1.2
00.00	1.2	01.20
##.##	123.456	123.46
00.00	123.456	123.46

「#」の場合は表示する数値が無い場合は無視されますが、「0」の場合は 0 が補って表示されます。またどちらの場合でも整数部分が書式に指定した文字数よりも多くてもそのまま表示されますが、小数点以下の部分は書式に指定した桁の位置で四捨五入されて表示されます。

```
Sub TEST51()
    Range("B2:B3").Value = 1.2
    Range("B4:B5").Value = 123.456
    Range("B2").NumberFormatLocal = "##.##"
    Range("B4").NumberFormatLocal = "##.##"
    Range("B3").NumberFormatLocal = "00.00"
    Range("B5").NumberFormatLocal = "00.00"
End Sub
```

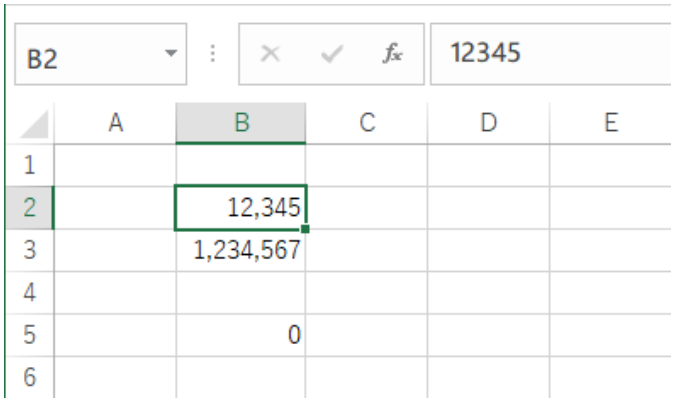


桁区切り

次に桁区切りです。「#,###」のように記述した場合、1000 単位で桁区切りが付きます。また「#」だけでの場合 0 の値が表示されないため通常は「#,##0」のように記述します。

書式	対象	表示
#,###	12345	12,345
#,###	1234567	1,234,567
#,###	0	表示なし
#,##0	0	0

```
Sub TEST52()
    Range("B2").Value = 12345
    Range("B2").NumberFormatLocal = "#,###"
    Range("B3").Value = 1234567
    Range("B3").NumberFormatLocal = "#,###"
    Range("B4").Value = 0
    Range("B4").NumberFormatLocal = "#,###"
    Range("B5").Value = 0
    Range("B5").NumberFormatLocal = "#,##0"
End Sub
```

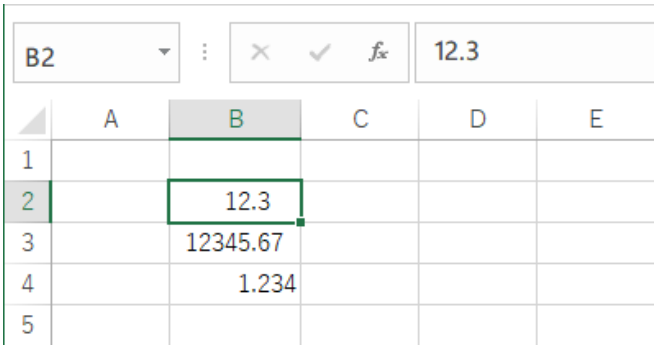


スペースでの位置調整

次に「0」ではなくスペースで桁をそろえて表示する方法です。「0」の代わりに「?」を使い「???.???」のように書式を指定します。少し分かりにくいですが小数点の位置が同じ位置になるように表示されます。

```
    12.3
12345.67
     1.234
```

```
Sub TEST53()
    Range("B2").Value = 12.3
    Range("B3").Value = 12345.67
    Range("B4").Value = 1.234
    Range("B2:B4").NumberFormatLocal = "?.???"
End Sub
```

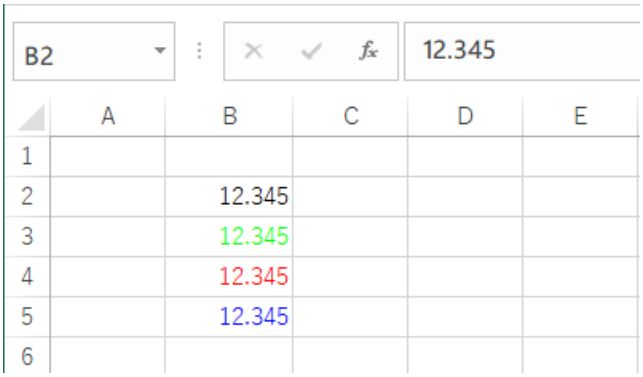


色の指定

書式として色を指定することも出来ます。指定できる色は 8 色で、書式の先頭に括弧[]の中に色を記述して指定します。8 色は次の通りです。

[黒] [青] [水] [緑] [紫] [赤] [白] [黄]

```
Sub TEST54()
    Range("B2:B5").Value = 12.345
    Range("B2").NumberFormatLocal = _
                                    "[黒]0.000"
    Range("B3").NumberFormatLocal = _
                                    "[緑]0.000"
    Range("B4").NumberFormatLocal = _
                                    "[赤]0.000"
    Range("B5").NumberFormatLocal = _
                                    "[青]0.000"
End Sub
```



通貨記号

数値の先頭に通貨記号などを付けて表示したい場合には例えば「¥#,##0」などのように普通の文字列を書式設定用の文字と並んで記述します。

```
Range("B2").NumberFormatLocal = "¥ #,##0"
Range("B2").NumberFormatLocal = "$ #,##0"
```

文字列は何でも指定可能ですが、書式として意味を持つ文字を使う場合(例えば#や Y など)にはダブルクオーテーションで囲んで記述します。

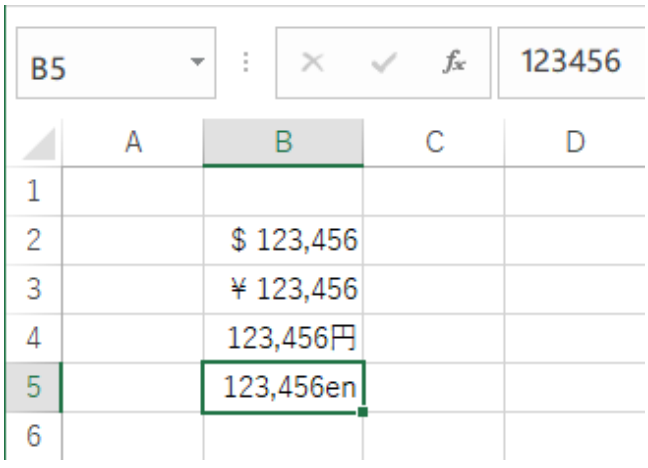
```
Range("B2").NumberFormatLocal = "#,##0 ""en"""
```

※ダブルクオーテーションで囲まれた中でダブルクオーテーションを記述する場合は、ダブルクオーテーションを 2 つ重ねて記述します。よって"en"は""en""になります。

また全角文字を記述する場合もダブルクオーテーションで囲んで記述します。(ダブルクオーテーションで囲まなくてもエラーにはなりませんが、正確には囲んでおいたほうがいいようです)。

```
Range("B2").NumberFormatLocal = "#,##0 ""円"""
```

```
Sub TEST55()
    Range("B2:B5").Value = 123456
    Range("B2").NumberFormatLocal = "$ #,##0"
    Range("B3").NumberFormatLocal = "¥ #,##0"
    Range("B4").NumberFormatLocal = "#,##0""円
"""
    Range("B5").NumberFormatLocal =
"#,##0""en"""
End Sub
```



正と負で書式を分ける

正の値と負の値の場合で書式を分けたい場合があります。よくあるのは負の値の場合は文字を赤くしたり、数値全体を括弧で囲んだり、負を表す「-」記号を△に変更したりといったことです。

正の値と負の値の場合で書式を分ける場合には、正の数用の書式と負の数用の書式を用意して、セミコロン(;)で結んで指定します。

```
オブジェクト.NumberFormatLocal = "(正の数用の書式);(負の数用の書式)"
```

例えば負の場合には赤字にする場合は次のようになります。

```
Range("B2").NumberFormatLocal = "#,##0;[赤]#,##0"
```

※このように負の値の書式を定義すると、負の値であっても先頭に「-」は付かなくなりますので注意して下さい。

他によくある書式としては次のようになります。

```
Range("B2").NumberFormatLocal = "#,##0;(#,##0)"
Range("B2").NumberFormatLocal = "#,##0;""△"" #,##0"
```

正と負とゼロで書式を分ける

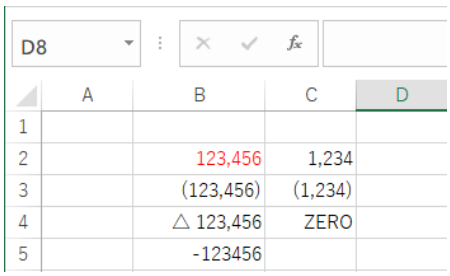
正と負だけではなく、ゼロの場合の書式も定義することができます。ゼロの書式も別途用意して、セミコロン(;)で区切って最後に追加します。

```
オブジェクト.NumberFormatLocal = "(正の数用の書式);(負の数用の書式);(ゼロの時の書式)"
```

ゼロの書式ですので設定できる書式にも限度がありますけど、色を変更したり、他の文字列を表示したりできます。

```
Range("B2").NumberFormatLocal = "#,##0;(#,##0);[青]0"
Range("B2").NumberFormatLocal = "#,##0;(#,##0);""ZERO"""
```

```
Sub TEST56()
    Range("B2:B5").Value = -123456
    Range("B2").NumberFormatLocal = "#,##0;[赤]#,##0"
    Range("B3").NumberFormatLocal = "#,##0;(#,##0)"
    Range("B4").NumberFormatLocal = "#,##0;""△"" #,##0"
    Range("C2:C4").NumberFormatLocal =
"#,##0;(#,##0);""ZERO"""
    Range("C2").Value = 1234
    Range("C3").Value = -1234
    Range("C4").Value = 0
End Sub
```



日付の書式

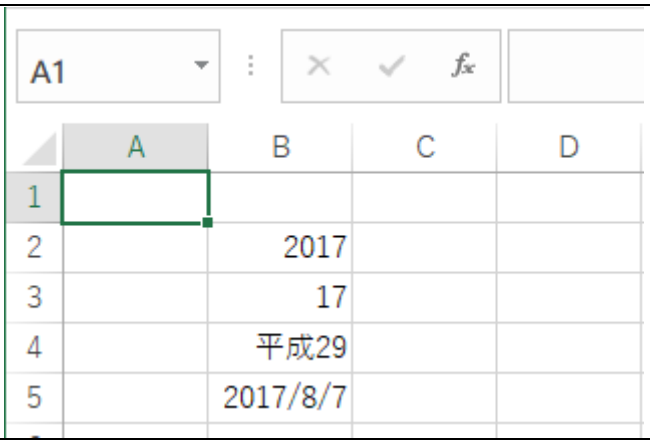
日付の書式は日や年などをどのように表示するかで書式用の文字を組み合わせて利用します。

年

年については西暦や和暦が使えます。

書式	パーツ	対象	表示
Yyyy	西暦	2017/08/07	2017
Yy	西暦	2017/08/07	17
G	和号	2017/08/07	H
gg	和号	2017/08/07	平
ggg	和号	2017/08/07	平成
E	和暦	1997/08/07	9
ee	和暦	1997/08/07	09

```
Sub TEST58()
    Range("B2:B5").Value = "2017/08/07"
    Range("B2").NumberFormatLocal = "yyyy"
    Range("B3").NumberFormatLocal = "yy"
    Range("B4").NumberFormatLocal = "gggee"
End Sub
```

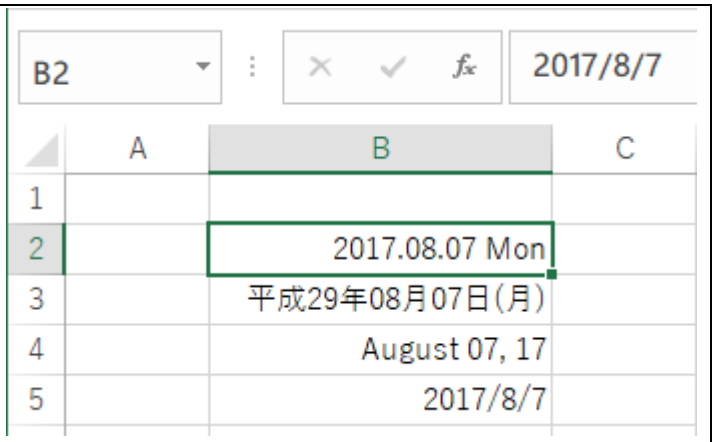


月と日と曜日

月は数値以外に英語表記が可能です。曜日は英語表記及び日本語表記が可能です。

書式	パーツ	対象	表示
m	月	2017/08/07	8
mm	月	2017/08/07	08
mmm	月	2017/08/07	Aug
mmmm	月	2017/08/07	August
d	日	2017/08/07	7
dd	日	2017/08/07	07
ddd	曜日(英語)	2017/08/07	Mon
dddd	曜日(英語)	2017/08/07	Monday
aaa	曜日(日本語)	2017/08/07	月
aaaa	曜日(日本語)	2017/08/07	月曜日

```
Sub TEST59()
    Range("B2:B5").Value = "2017/08/07"
    Range("B2").NumberFormatLocal _
        = "yyyy.mm.dd ddd"
    Range("B3").NumberFormatLocal _
        =  "gggee"" 年 ""mm"" 月 ""dd"" 日 ""(aaa)"
    Range("B4").NumberFormatLocal _
        = "mmmm dd, yy"
End Sub
```



日付の場合でも、好きな文字を書式の間に記述することが出来ます。

時刻の書式

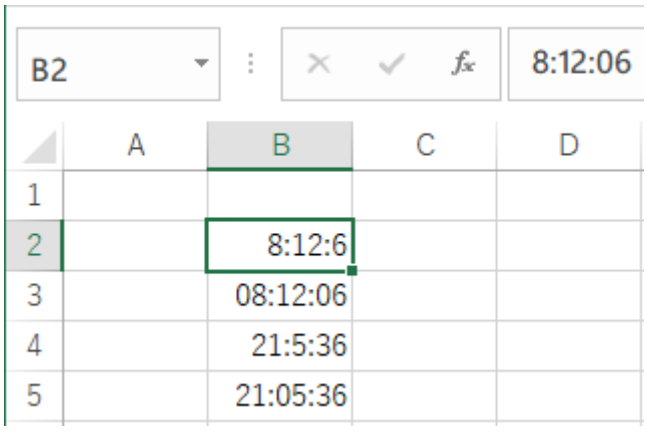
時刻の書式は時や分をどのように表示するかで書式用の文字を組み合わせて利用します。

時分秒については次の通りです。

書式	パーツ	対象	表示
h	時間	01:02:03	1
hh	時間	01:02:03	01
m	分	01:02:03	2
mm	分	01:02:03	02
s	秒	01:02:03	3
ss	秒	01:02:03	03

「m」や「mm」は月の書式文字と同じですが、「h」や「s」などと一緒に使うことで分を表す書式文字として使えます。

```
Sub TEST59()
    Range("B2:B3").Value = "08:12:06"
    Range("B2").NumberFormatLocal = "h:m:s"
    Range("B3").NumberFormatLocal =
"hh:mm:ss"
    Range("B4:B5").Value = "21:05:36"
    Range("B3").NumberFormatLocal = "h:m:s"
    Range("B5").NumberFormatLocal =
"hh:mm:ss"
End Sub
```



時刻の書式

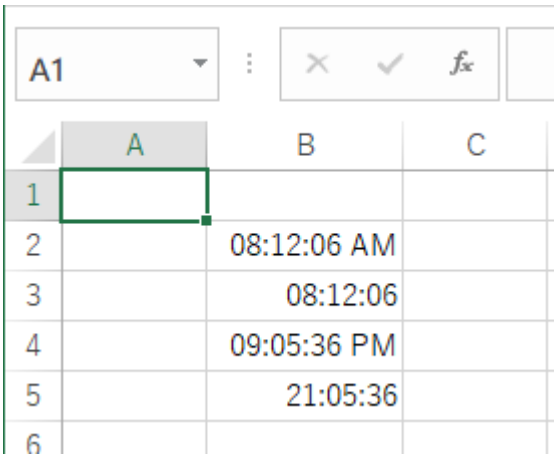
AM/PM

時刻は 24 時間表記となってますが、「AM/PM」を追加で記述すると 12 時間表記とすることもできます。例えば次のように記述します。

```
Range("B2").NumberFormatLocal = "hh:mm:ss AM/PM"
```

このように AM/PM を追加することで、時間は 12 時間表記に変わり、午前または午後に応じて AM または PM が表示されます。

```
Sub TEST60()
    Range("B2:B3").Value = "08:12:06"
    Range("B2").NumberFormatLocal _
            = "hh:mm:ss AM/PM"
    Range("B3").NumberFormatLocal _
            = "hh:mm:ss"
    Range("B4:B5").Value = "21:05:36"
    Range("B4").NumberFormatLocal _
            = "hh:mm:ss AM/PM"
    Range("B5").NumberFormatLocal _
            = "hh:mm:ss"
End Sub
```



時刻の場合でも、好きな文字を書式の間に記述することが出来ます。

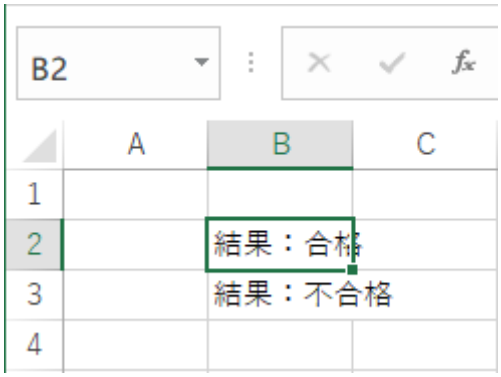
文字の書式

文字の場合は書式というよりも入力された値の前後に他の文字を付け加えるような利用方法の確認です。
セルに含まれている実際の値は「@」で表すことができます。よって何か前後に文字を表示する場合には次のようになります。

```
Range("B2").NumberFormatLocal = """結果：""@"
```

実際の値の前に「結果：」という文字列を表示するようにしています。(全角なのでダブルクオーテーションで囲み、さらにダブルクオーテーションをエスケープするために「"」を重ねて記述しています。

```
Sub TEST61()
    Range("B2:B3").NumberFormatLocal _
            = """結果：""@"
    Range("B2").Value = "合格"
    Range("B3").Value = "不合格"
End Sub
```



実際の値を変えることなく、必要な文字を前後に表示することが出来ます。

セルの文字配置の設定

セルの中で文字がどのように配置されるかについて設定する方法を確認します。

水平位置と垂直位置

まずセルの中の文字の水平位置と垂直位置についてです。Range オブジェクトの「HorizontalAlignment」プロパティで水平位置を、「VerticalAlignment」プロパティで垂直位置を設定します。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.HorizontalAlignment = xlCenter
range1.VerticalAlignment = xlTop
```

それぞれのプロパティに設定する値は既に Excel で定義されています。

まず水平位置を表す定数は次のものがあります。

定数	水平位置
xlGeneral	標準
xlLeft	左詰
xlCenter	中央揃え
xlRight	右詰
xlFill	繰り返し
xlJustify	両端揃え
xlCneterAcrossSelection	選択範囲内で中央
xlDistributed	均等割り付け

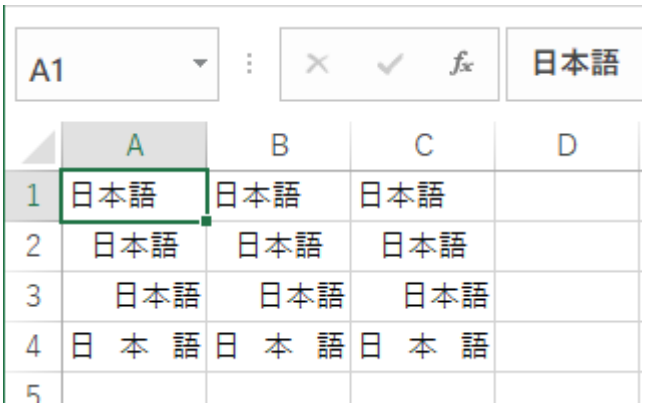
水平位置のデフォルトの値は「xlGeneral」です。

次に垂直方向を表す定数には次のものがあります。

定数	水平位置
xlTop	上詰
xlCenter	中央揃え
xlBottom	下詰
xlFill	繰り返し
xlJustify	両端揃え
xlDistributed	均等割り付け

垂直位置のデフォルトの値は「xlCenter」です。

```
Sub TEST62()
    Range("A1:C4").Value = "日本語"
    Range("A1:C1").HorizontalAlignment = xlLeft
    Range("A2:C2").HorizontalAlignment = xlCenter
    Range("A3:C3").HorizontalAlignment = xlRight
    Range("A4:C4").HorizontalAlignment _
        = xlDistributed
    Range("A1:A4").VerticalAlignment = xlTop
    Range("B1:B4").VerticalAlignment = xlCenter
    Range("C1:C4").VerticalAlignment = xlBottom
End Sub
```



インデントの設定

セルの水平位置の設定で「左詰め」「右詰め」「均等割り付け」を設定した場合に、インデントを設定することが出来ます。左詰めの場合は左側に、右詰めの場合は右側に、均等割り付けの場合は両端にインデントが設定されます。

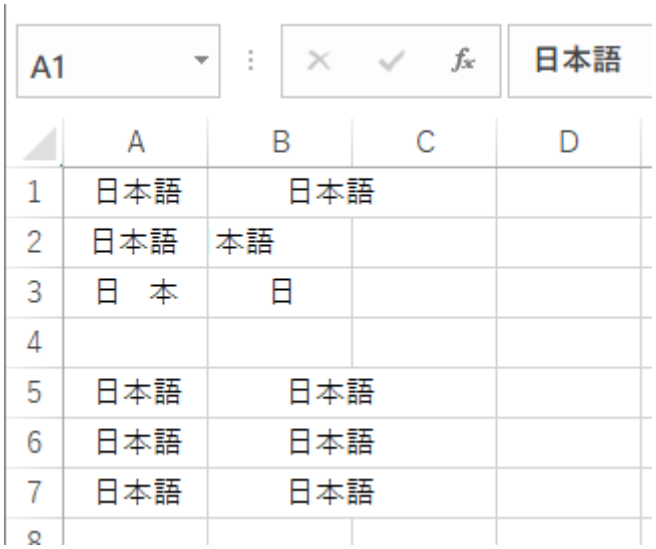
インデントを設定するには Range オブジェクトの「IndentLevel」プロパティで設定します。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.HorizontalAlignment = xlLeft
range1.IndentLevel = 1
```

設定する値はインデントする文字数です。1 文字と指定した場合には全角 1 文字に相当します。

注意する点としては、「IndentLevel」プロパティを設定すると、「HorizontalAlignment」プロパティが「xlLeft」に自動的に変更されてしまいます。その為、右詰めや均等割り付けでインデントを設定する場合には、まず「IndentLevel」プロパティを設定してからその後で「HorizontalAlignment」プロパティを設定して下さい。

```
Sub TEST63()
    Range("A1:B3").Value = "日本語"
    Range("A1:A3").IndentLevel = 1
    Range("B1:B3").IndentLevel = 3
    Range("A1:B1").HorizontalAlignment = xlLeft
    Range("A2:B2").HorizontalAlignment = xlRight
    Range("A3:B3").HorizontalAlignment _
        = xlDistributed
    Range("A5:B7").Value = "日本語"
    Range("A5:B5").HorizontalAlignment = xlLeft
    Range("A6:B6").HorizontalAlignment = xlRight
    Range("A7:B7").HorizontalAlignment _
        = xlDistributed
    Range("A5:A7").IndentLevel = 1
    Range("B5:B7").IndentLevel = 3
End Sub
```



上 3 つのセルと、下 3 つのセルは、設定している内容は同じなのですが、水平位置とインデントの設定する順番が異なります。インデントを設定すると水平位置の設定が全て「左詰め」になってしまうので、プロパティを設定する順番は注意して下さい。

均等割り付けで両端に空白を入れる

セルの水平位置の設定で「均等割り付け」を設定した場合に、両端に空白を入れることが出来ます。

そもそも均等割り付けは次のようなルールでセル内に文字を表示しています。例として日経新聞社という文字

を均等割り付けで表示する場合で考えてみましょう。

K○C○S○鹿○児○島○情○報○専○門○学○校

上記で○の部分は空白の領域です。全ての○は同じ大きさの空白の領域となります。セルの幅が大きくなれば、○の大きさを調整するわけです。

次に均等割り付けで両端に空白を入れる場合には次のようになります。

```
〇K〇C〇S〇鹿〇児〇島〇情〇報〇専〇門〇学〇校〇
```

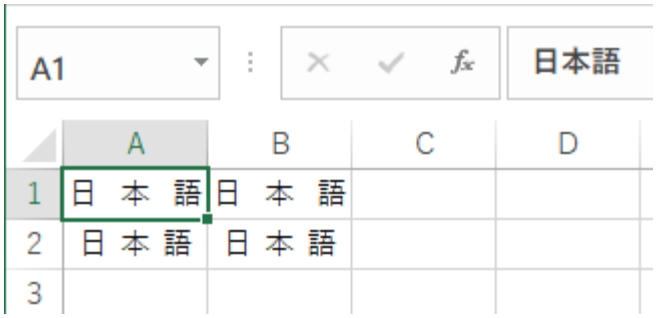
今度は最初と最後にも同じように空白が入るようになります。〇の大きさは全て同じですので、セルの大きさが同じならば両端に〇が増えた分だけ〇の部分の領域は小さくなりますが、両端に空白がありので読みやすくなります。

両端に空白を設定するには Range オブジェクトの「AddIndent」プロパティで設定します。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.HorizontalAlignment = xlDistributed
range1.AddIndent = True
```

設定する値は空白を設定するかどうかだけを指定しますので「True」か「False」を設定します。

```
Sub TEST64()
    Range("A1:B2").Value = "日本語"
    Range("A1:B2").HorizontalAlignment _
        = xlDistributed
    Range("A1:B1").AddIndent = False
    Range("A2:B2").AddIndent = True
End Sub
```



セルの大きさによって、均等割り付けの空白がどのように配分されるのか確認できると思います。

セルに収まらない場合の処理

1 つのセルに入りきらないほど多くの文字などがセルに含まれた場合、デフォルトではセルをはみ出して表示されますが、セル内で文字を折り返して全体を表示するようにしたり、文字のサイズを変更してセルに収まるようにすることが出来ます。

セルの幅でセルに含まれる値を折り返して表示させる場合には Range オブジェクトの「WrapText」プロパティで設定します。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.WrapText = True
```

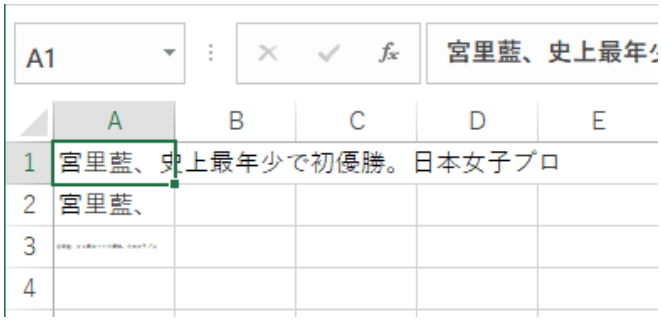
設定する値は「True」か「False」を設定します。True を設定した場合に折り返しが行われます。

また、表示される文字サイズを調整してセル内に全ての値が表示されるようにするには Range オブジェクトの「ShrinkToFit」プロパティで設定します。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.ShrinkToFit = True
```

設定する値は「True」か「False」を設定します。True を設定した場合に文字サイズの自動調整が行われます。

```
Sub TEST65()
    Range("A1:A3").Value _
    = "宮里藍、史上最年少で初優勝。日本女子プロ
"
    Range("A2").WrapText = True
    Range("A3").ShrinkToFit = True
End Sub
```



セルの結合と解除

ここではセルの結合の方法を見ていきます。セルを結合するには、結合するセル範囲を表す Range オブジェクトに対して「MergeCells」プロパティを True を設定します。

```
Dim range1 As Range
Set range1 = Range("A1:B2")
range1.MergeCells = True
```

セルを結合した場合、結合したセル範囲の左上にあるセルに含まれていた値だけが有効となります。その他のセルに含まれていた値は全て削除されます。

また結合されたセルに対して引き続き水平位置の指定などを行いたい場合には、結合したセル範囲の左上のセルに対して行います。

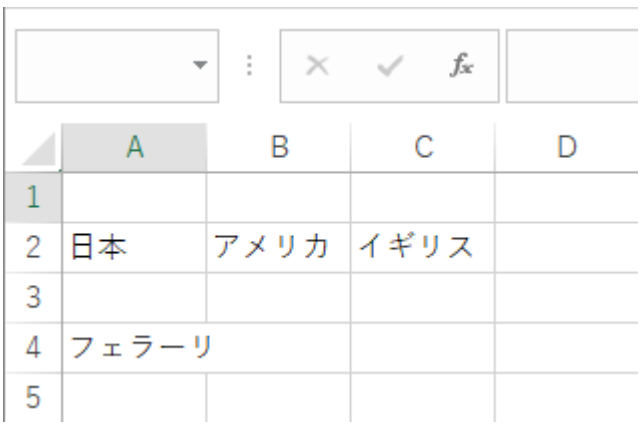
```
Dim range1 As Range
Set range1 = Range("A1:B2")
range1.MergeCells = True
Range("A1").HorizontalAlignment = xlCenter
Range("A1").Value = "新しい値"
```

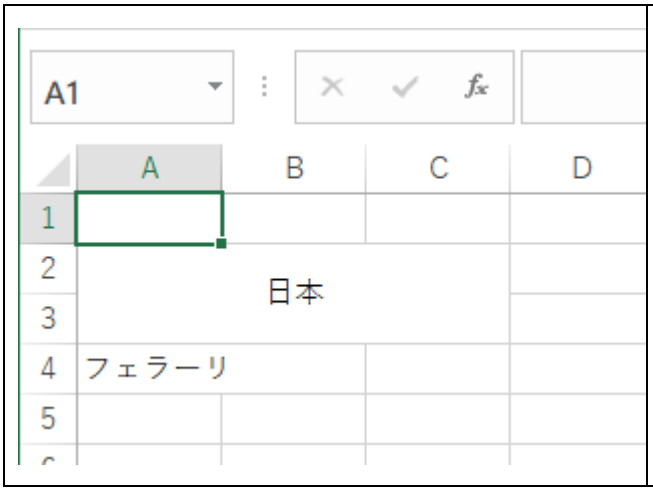
また結合されたセルを解除する場合には、同じプロパティに「False」を設定して下さい。

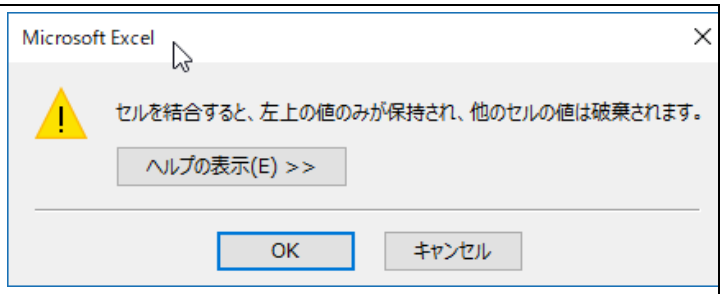
```
Range("A1:B2").MergeCells = True
Range("B1").MergeCells = False
```

セルを解除する場合には解除したい結合セルの中に含まれるどのセルの「MergeCells」プロパティを False にしても結構です。

```
Sub TEST66()
    Range("A2").value = "日本"
    Range("A4").value = "フェラーリ"
    Range("B2").value = "アメリカ"
    Range("C2").value = "イギリス"
    Range("A2:C3").MergeCells = True
    Range("A2").HorizontalAlignment = xlCenter
    Range("B4").MergeCells = False
End Sub
```







結合する場合に左上のセル以外のセルに値が含まれていた場合、セルの値が削除される警告表示が行われます。

縦書きと文字の角度

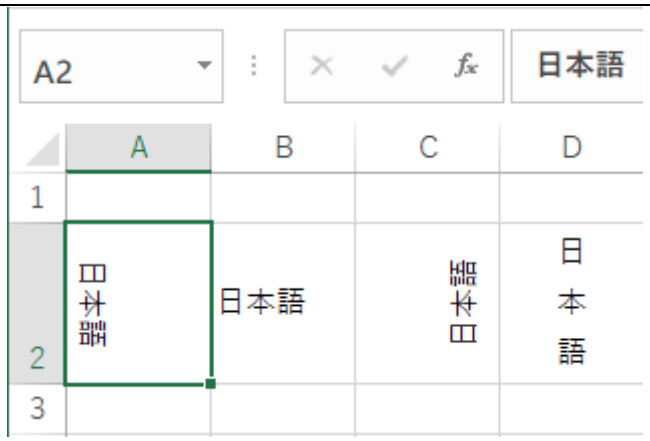
ここではセル内の文字の向きの設定方法です。向きには縦書きにする設定と、横書きのまま角度を設定する方法がありますが、どちらも Range オブジェクトの「Orientation」プロパティで設定します。

```
Dim range1 As Range
Set range1 = Range("A1:B2")
range1.Orientation = 45
```

縦書きにする場合には「Orientation」プロパティに「xlVertical」を設定します。横書きにする場合は「xlHorizontal」という定数も用意されていますが、代わりに角度を指定しても構いません。

角度は「-90」から「90」までの範囲で角度を指定します。「-90」の場合には文字の先頭が一番上になります。「90」の場合には文字の先頭が一番下になります。角度「0」がデフォルトの状態です。

```
Sub TEST67()
    Range("A2:D2").Value = "日本語"
    Range("A2").Orientation = -90
    Range("B2").Orientation = 0
    Range("C2").Orientation = 90
    Range("D2").Orientation = xlVertical
    Range("A2:D2").Select
    Selection.EntireRow.AutoFit
    Range("A1").Select
End Sub
```



フォントやサイズの設定

セルに表示されている値のフォントや文字の色などを設定する方法を確認します。

Font オブジェクト

書式に関する色々な情報を管理しているオブジェクトは Font オブジェクトです。Font オブジェクトは Range

オブジェクトの Font プロパティを使って取得することができます。

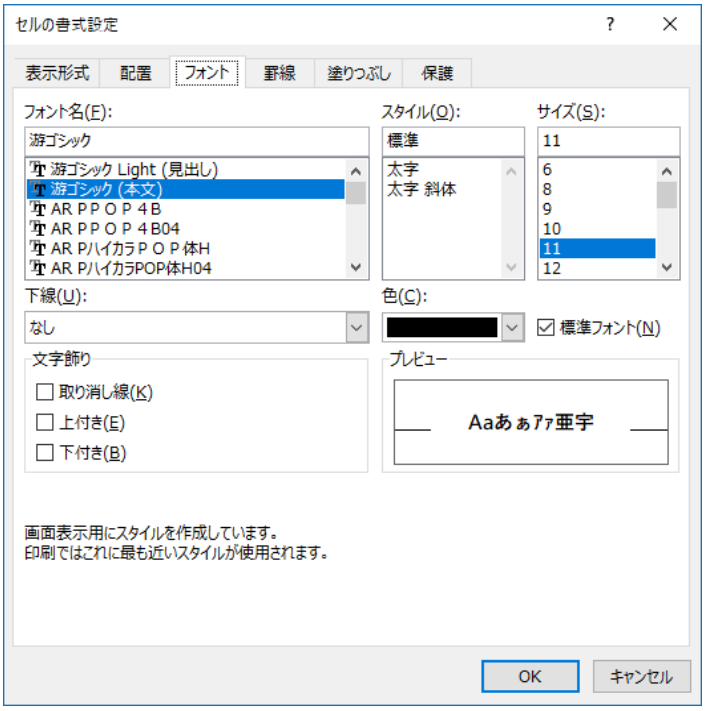
```
Dim font1 As Font
Set font1 = Range("A1").Font
font1.Size = 20
```

上記の例では Font オブジェクトを取り出した後で、今度は Font オブジェクトの Size プロパティの値を変更しています。

Font オブジェクトを別途取り出さずに、次のようにまとめて記述しても構いません。

```
Range("A1").Font.Size = 20
```

セルに関する情報を保持している Font オブジェクトを取り出したら、次に Font オブジェクトの各プロパティに値を設定することでセルに関する書式を変更することが可能です。

セルに設定可能な書式は、Excel の「セルの書式設定」で設定可能なものになります。	

フォント名とサイズの設定

まずはセルに表示されている値のフォントを変更してみます。変更するには Font オブジェクトの「Name」プロパティにフォント名を文字列として設定します。

```
Dim font1 As Font
Set font1 = Range("A1").Font
font1.Name = "ＭＳ　Ｐゴシック"
```

Font オブジェクトを別途取り出さずに、次のようにまとめて記述しても構いません。

```
Range("A1").Font.Name = "ＭＳ　Ｐゴシック"
```

設定可能なフォントは Excel の「セルの書式設定」で選択可能なフォント名が指定可能です。

次にフォントのサイズを変更してみます。変更するには Font オブジェクトの「Size」プロパティにフォントサイズを数値で設定します。

```
Dim font1 As Font
```

```
Set font1 = Range("A1").Font
font1.Size = 15
```

Font オブジェクトを別途取り出さずに、次のようにまとめて記述しても構いません。

```
Range("A1").Font.Size = 15
```

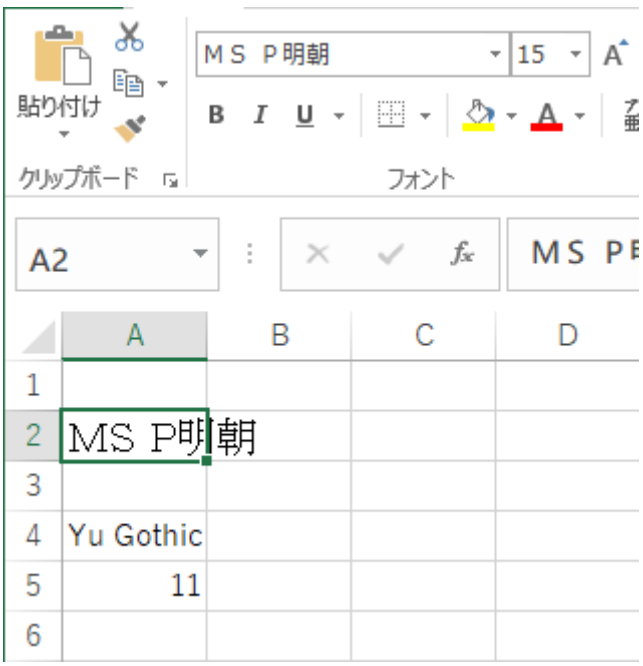
標準のフォント名とフォントサイズ

Excel で特にフォントを指定しない場合にデフォルトで使われるフォント名とフォントサイズを取得及び設定することが出来ます。どちらも Application オブジェクトのプロパティとして用意されています。(デフォルトのフォントはシートやブック単位ではなく、Excel 全体で同じ値を参照しています)。標準フォントは「StandardFont」プロパティで標準フォントサイズは「StandardFontSize」プロパティで取得や設定が可能です。

```
Application.StandardFont = "ＭＳ　Ｐゴシック"
Application.StandardFontSize = 12
```

注意点として、標準フォントや標準フォントサイズは変更が可能ですが、変更後 Excel を再移動するまでは変更が反映されませんので注意して下さい。

```
Sub TEST68()
    Range("A2").Value = "ＭＳ　Ｐ明朝"
    Range("A2").Font.name = "ＭＳ　Ｐ明朝"
    Range("A2").Font.Size = 15
    Range("A4").Value _
            = Application.StandardFont
    Range("A5").Value _
            = Application.StandardFontSize
End Sub
```



Bold と Italic の設定

次はスタイルの設定です。太字(Bold)と斜体(Italic)の設定です。太字を設定するには Font オブジェクトの「Bold」プロパティで、斜体を設定するには Font オブジェクトの「Italic」プロパティを使います。

```
Dim font1 As Font
Set font1 = Range("A1").Font
font1.Bold = True
font1.Italic = True
```

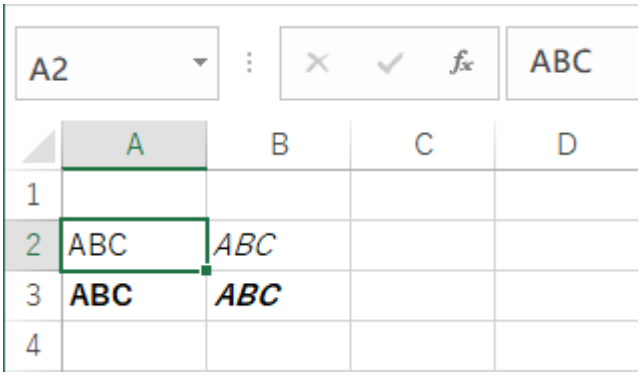
Font オブジェクトを別途取り出さずに、次のようにまとめて記述しても構いません。

```
Range("A1").Font.Bold = True
```

```
Range("A1").Font.Italic = True
```

「Bold」プロパティと「Italic」プロパティは排他的なプロパティではありませんので、同時に True を設定する事も可能です。

```
Sub TEST69()
    Range("A2:B3").Value = "ABC"
    Range("A3").Font.Bold = True
    Range("B2").Font.Italic = True
    Range("B3").Font.Bold = True
    Range("B3").Font.Italic = True
End Sub
```



下線の設定

次は下線の設定です。下線を設定するには Font オブジェクトの「Underline」プロパティを使います。

```
Dim font1 As Font
Set font1 = Range("A1").Font
font1.Underline = True
```

Font オブジェクトを別途取り出さずに、次のようにまとめて記述しても構いません。

```
Range("A1").Font.Underline = True
```

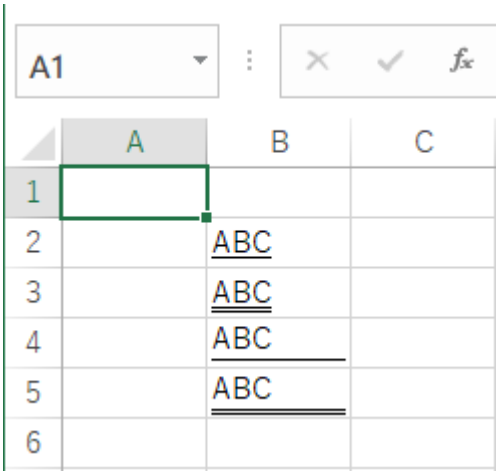
「Underline」プロパティは単に「True」を設定した場合には 1 本線の下線が引かれます。1 本線以外の形状の線を引く場合には次のどれかの値を設定して下さい。

定数	下線の種類
xlUnderlineStyleNone	無し
xlUnderlineStyleSingle	下線
xlUnderlineStyleDouble	二重下線
xlUnderlineStyleSingleAccounting	下線（会計）
xlUnderlineStyleDoubleAccounting	二重下線（会計）

二重下線を引く場合には次のように記述します。

```
Range("A1").Font.Underline = xlUnderlineStyleDouble
```

```
Sub TEST70()
    Range("B2:B5").Value = "ABC"
    Range("B2").Font.Underline _
        = xlUnderlineStyleSingle
    Range("B3").Font.Underline _
        = xlUnderlineStyleDouble
    Range("B4").Font.Underline _
        = xlUnderlineStyleSingleAccounting
    Range("B5").Font.Underline _
        = xlUnderlineStyleDoubleAccounting
End Sub
```



文字飾りの設定

次は文字飾りの設定です。文字飾りには取り消し線の描画と文字の上付き下付きの設定があります。
取り消し線を設定するには Font オブジェクトの「Strikethrough」プロパティを使います。

```
Dim font1 As Font
Set font1 = Range("A1").Font
font1.Strikethrough = True
```

Font オブジェクトを別途取り出さずに、次のようにまとめて記述しても構いません。

```
Range("A1").Font.Strikethrough = True
```

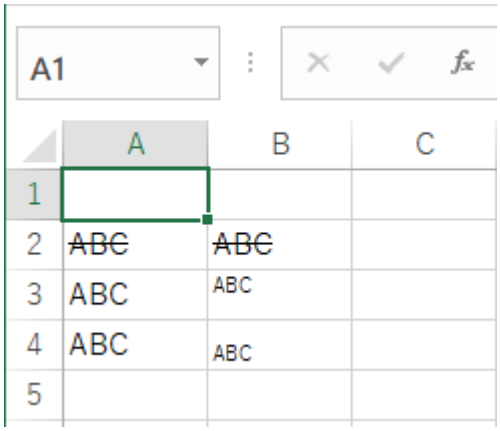
「Strikethrough」プロパティは「True」または「False」の値を設定します。「True」を設定した時に取り消し線が描画されます。

次に文字の上付きと下付きの設定です。それぞれ Font オブジェクトの「Superscript」プロパティと「Subscript」プロパティを使い「True」を設定すると有効になります。ただ、この二つのプロパティは同一の Font オブジェクトに対して両方同時に「True」の設定は出来ませんので注意して下さい。

記述例としては次のようになります。

```
Range("A1").Font.Superscript = True
Range("A2").Font.Subscript = True
```

```
Sub TEST71()
    Range("A2:B4").Value = "ABC"
    Range("A2:B2").Font.Strikethrough = True
    Range("B3").Font.Superscript = True
    Range("B4").Font.Subscript = True
End Sub
```



文字色の設定

次は文字色の設定です。使用できる色は既に用意されており全部で 57 種類の色が用意されています。それぞれの色には 0 から 56 のインデックス番号が対応しており、Font オブジェクトの「ColorIndex」プロパティにインデックス番号を指定することで文字色を指定できます。

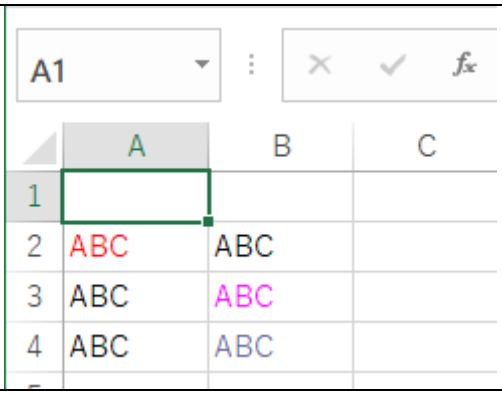
```
Dim font1 As Font
Set font1 = Range("A1").Font
font1.ColorIndex = 10
```

Font オブジェクトを別途取り出さずに、次のようにまとめて記述しても構いません。

```
Range("A1").Font.ColorIndex = 10
```

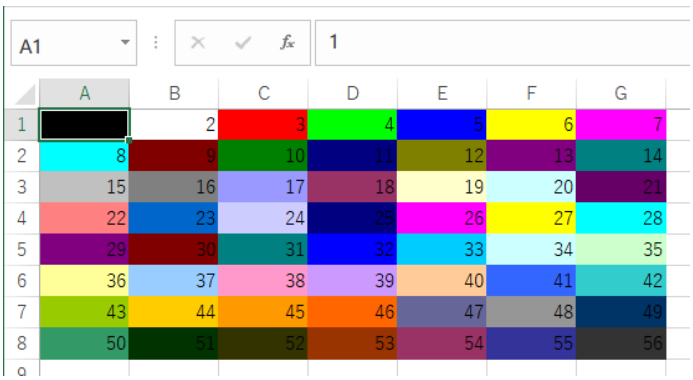
ここで指定できる 57 種類の色は、Excel で「セルの書式設定」で設定可能な色と同じです。

```
Sub TEST72()
    Range("A2:B4").Value = "ABC"
    Range("A2").Font.ColorIndex = 3
    Range("B3").Font.ColorIndex = 26
    Range("B4").Font.ColorIndex = 47
End Sub
```



また、色のインデックス番号と実際の色の一覧を確認しておきます。セルの背景色にも同じように色をインデックス番号で指定できますので、色々な色をセルの背景に設定してみましょう。

```
Sub TEST73()
    Dim i As Integer, j As Integer
    For i = 1 To 8
        For j = 1 To 7
          Cells(i, j).Interior.ColorIndex = (i - 1) * 7 + j
          Cells(i, j).Value = (i - 1) * 7 + j
        Next j
    Next i
End Sub
```



※インデックス番号「0」は色無しのとなりますが、文字色の場合は色無しは設定できません。

セルの罫線の設定

セルに表示されている値のフォントや文字の色などを設定する方法を確認します。

Border オブジェクト

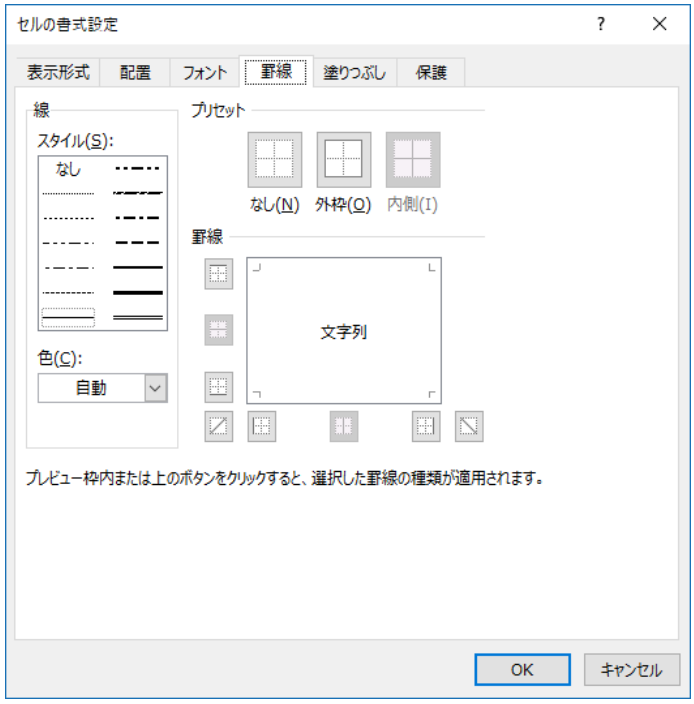
セルの罫線に関する情報を管理しているオブジェクトは Border オブジェクトです。セルに対する罫線を表示する場合には、Range オブジェクトの Borders プロパティを使って Border オブジェクトを取得します。

```
Dim border1 As Border
Set border1 = Range("A1").Borders(xlEdgeTop)
border1.LineStyle = xlContinuous
```

まとめて次のように記述しても構いません。

```
Range("A1").Borders(xlEdgeTop).LineStyle = xlContinuous
```

Border オブジェクトは 1 つ 1 つの罫線を表しています。単独のセルを対象とした罫線を考えた場合、罫線は上下左右の 4 つあるだけですが、複数のセル領域を対象と考えた場合、罫線は領域の上下左右以外に領域内の横線と縦線があります。また右あがりの罫線と右下がりの罫線があります。



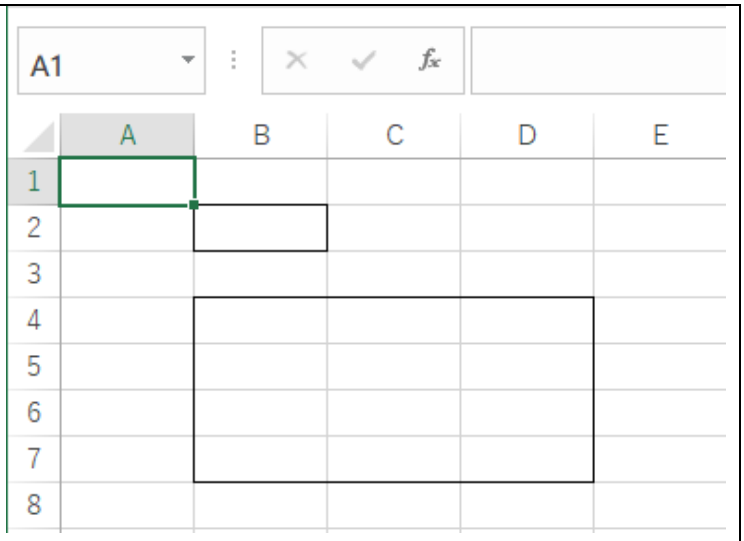
よって対象となるセルまたはセル範囲に対して罫線の位置は最大で 8 種類あります。Border オブジェクトを

取り出す場合には、「Borders」プロパティの引数にどの位置の罫線を対象とした Border オブジェクトを取り出すのかを指定します。

定数	罫線の位置
xlEdgeTop	上端
xlEdgeBottom	下端
xlEdgeLeft	左端
xlEdgeRight	右端
xlInsideHorizontal	内側横線
xlInsideVertical	内側縦線
xlDiagonalDown	右下がり斜線
xlDiagonalUp	右上がり斜線

指定した位置の罫線を表す Border オブジェクトを取り出したら、後は罫線の形状や太さ、色などを指定して線を引きます。

```
Sub TEST74()
    With Range("B2")
        .Borders(xlEdgeTop).LineStyle _
          = xlContinuous
        .Borders(xlEdgeBottom).LineStyle _
          = xlContinuous
        .Borders(xlEdgeLeft).LineStyle _
          = xlContinuous
        .Borders(xlEdgeRight).LineStyle _
          = xlContinuous
    End With
    With Range("B4:D7")
        .Borders(xlEdgeTop).LineStyle _
          = xlContinuous
        .Borders(xlEdgeBottom).LineStyle _
          = xlContinuous
        .Borders(xlEdgeLeft).LineStyle _
           = xlContinuous
        .Borders(xlEdgeRight).LineStyle _
          = xlContinuous
    End With
End Sub
```



このように Border オブジェクトを取り出す領域全体を 1 つのものと考えて、それに対してどの位置に罫線を引くのかを指定することになります。

罫線の種類の設定

罫線の線の種類は Border オブジェクトの「LineStyle」プロパティで管理されています。デフォルトでは罫線を表示しない「xlLineStyleNone」が設定されていますので罫線を表示したい場合には「LineStyle」プロパティに適切な値を設定します。

```
Dim border1 As Border
Set border1 = Range("A1").Borders(xlEdgeTop)
border1.LineStyle = xlContinuous
```

設定可能な罫線の種類は次のようになります。

定数	罫線の種類
xlContinuous	実線(細)
xlDash	破線
xlDashDot	一点鎖線
xlDashDotDot	二点鎖線
xlDot	点線
xlDouble	二重線
xlSlantDashDot	斜め斜線
xlLineStyleNone	無し

罫線は形状の他にも線の太さを設定可能です。Border オブジェクトの「Weight」プロパティで設定します。

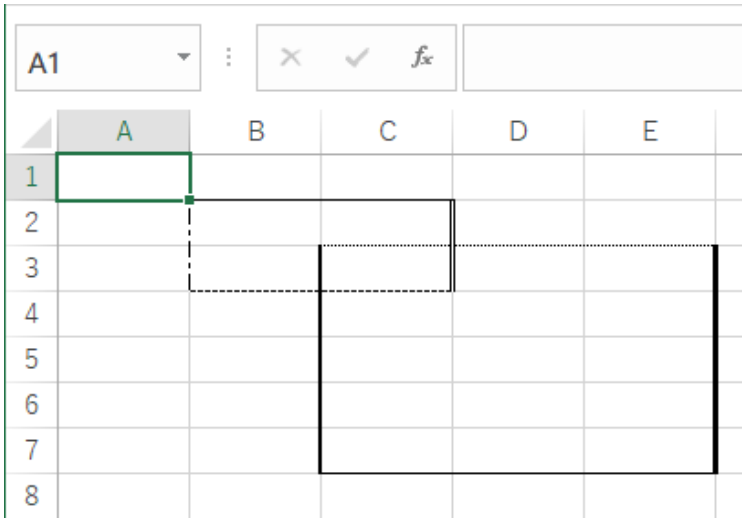
```
Dim border1 As Border
Set border1 = Range("A1").Borders(xlEdgeTop)
border1.LineStyle = xlContinuous
border1.Weight = xlMedium
```

設定可能な罫線の太さは次のようになります。

定数	罫線の太さ
xlHairline	極細
xlThin	細
xlMedium	中
xlThick	太

罫線の線種と太さを組み合わせることで多くの種類の罫線を描く事ができますが、設定が出来ない組み合わせもありますので注意して下さい。

```
Sub TEST75()
    With Range("B2:C3")
        .Borders(xlEdgeTop).LineStyle _
            = xlContinuous
        .Borders(xlEdgeBottom).LineStyle _
            = xlDash
        .Borders(xlEdgeLeft).LineStyle _
            = xlDashDot
        .Borders(xlEdgeRight).LineStyle _
            = xlDouble
    End With
    With Range("C3:E7")
        .Borders(xlEdgeTop).LineStyle _
            = xlContinuous
        .Borders(xlEdgeTop).Weight _
            = xlHairline
        .Borders(xlEdgeBottom).LineStyle _
            = xlContinuous
        .Borders(xlEdgeBottom).Weight _
            = xlThin
        .Borders(xlEdgeLeft).LineStyle _
            = xlContinuous
        .Borders(xlEdgeLeft).Weight _
            = xlMedium
        .Borders(xlEdgeRight).LineStyle _
            = xlContinuous
        .Borders(xlEdgeRight).Weight _
            = xlThick
    End With
End Sub
```



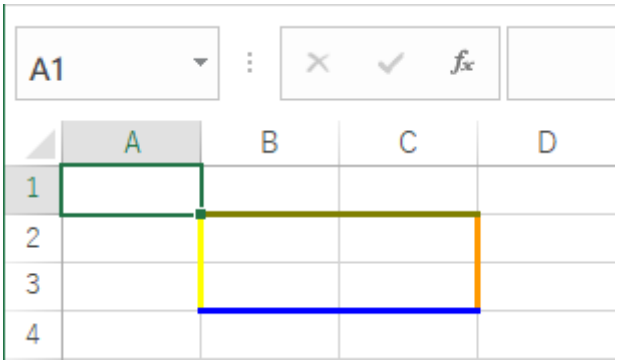
罫線の色の設定

罫線の色については Border オブジェクトの「ColorIndex」プロパティで設定します。ここで「ColorIndex」プロパティは文字色の設定で使った「ColorIndex」プロパティと同じで全部で 57 種類の色が用意されており、0 から 56 のインデックス番号で指定します。

記述の方法は次のようになります。

```
Dim border1 As Border
Set border1 = Range("A1").Borders(xlEdgeTop)
border1.ColorIndex = 12
```

```
Sub TEST76()
    With Range("B2:C3")
        .Borders(xlEdgeTop).LineStyle _
            = xlContinuous
        .Borders(xlEdgeTop).ColorIndex _
            = 12
        .Borders(xlEdgeTop).Weight = xlThick
        .Borders(xlEdgeBottom).LineStyle _
            = xlContinuous
        .Borders(xlEdgeBottom).ColorIndex _
            = 32
        .Borders(xlEdgeBottom).Weight = xlThick
        .Borders(xlEdgeLeft).LineStyle _
            = xlContinuous
        .Borders(xlEdgeLeft).ColorIndex _
            = 6
        .Borders(xlEdgeLeft).Weight = xlThick
        .Borders(xlEdgeRight).LineStyle _
            = xlContinuous
        .Borders(xlEdgeRight).ColorIndex _
            = 45
        .Borders(xlEdgeRight).Weight = xlThick
    End With
End Sub
```



Borders コレクション

Range オブジェクトの「Borders」プロパティで、プロパティに引数を指定すると指定した辺の位置にある Border オブジェクトを取得できますが、引数を指定せずに「Borders」プロパティの値を取得した時は Borders コレクションのオブジェクトを取得できます。

```
Dim border1 As Borders
Set border1 = Range("A1").Borders
```

Borders コレクションには上下左右の 4 つの辺が全て含まれています。このため、全ての辺で同じ罫線を表示したい場合には Border オブジェクトで 1 つ 1 つ設定するのではなく、Borders コレクションの各プロパティに設定すれば 4 つの辺にまとめて設定する事が出来ます。

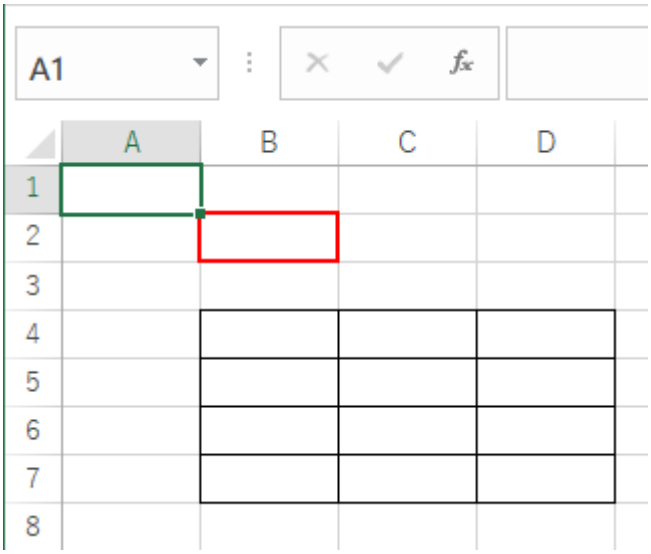
```
Dim border1 As Borders
Set border1 = Range("A1").Borders
border1.LineStyle = xlContinuous
border1.Weight = xlMedium
border1.ColorIndex = 10
```

単独のセルではなく、セル範囲から Borders コレクションを取得した場合には、領域内の全てのセルの上下左右の罫線が描画されます。

```
Range("A1:B4").Borders.LineStyle = xlContinuous
```

同じ罫線をまとめて描画したい場合には便利です。

```
Sub TEST77()
    Dim border1 As Borders
    Set border1 = Range("B2").Borders
    border1.LineStyle = xlContinuous
    border1.Weight = xlMedium
    border1.ColorIndex = 3
    Range("B4:D7").Borders.LineStyle _
      = xlContinuous
End Sub
```



セルの背景色の設定

セルの背景色や網かけの設定方法について確認します。

Interior オブジェクトと背景色

セルの背景色などセル自身の色などに関する情報を管理しているオブジェクトは Interior オブジェクトです。Range オブジェクトの「Interior」プロパティを使って Interior オブジェクトを取得します。

```
Dim interior1 As Interior
Set interior1 = Range("A1").Interior
interior1.ColorIndex = 3
```

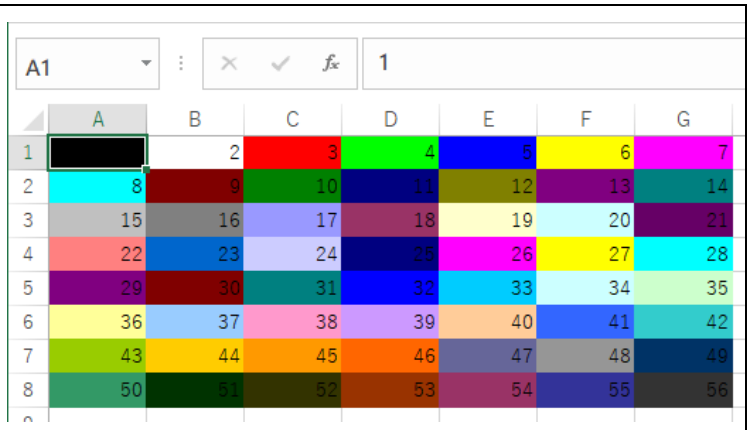
まとめて次のように記述しても構いません。

```
Range("A1").Interior.ColorIndex = 3
```

背景色の設定

Interior オブジェクトは主にセルの背景色の設定に使われます。背景色を設定するには Interior オブジェクトの「ColorIndex」プロパティに値を設定します。「ColorIndex」プロパティは文字の色や罫線の色で使ったものと同じです。全部で 57 種類の色が用意されており、0 から 56 のインデックス番号で指定します。(詳しくは『文字色の設定』を参照して下さい)。

```
Sub TEST78()
    Dim i As Integer, j As Integer
    For i = 1 To 8
        For j = 1 To 7
            Cells(i, j).Interior.ColorIndex _
                = (i - 1) * 7 + j
            Cells(i, j).Value = (i - 1) * 7 + j
        Next j
    Next i
End Sub
```



網かけの設定

セルの背景に網かけを行うことができます。網かけは網のパターンと網の色を指定して行います。背景色の上に網の色でパターンを描画することになります。

まず網のパターンは、Interior オブジェクトの「Pattern」プロパティで指定します。指定できる値は既に決められており、次の定数のいずれかを指定します。

定数	網かけパターン
xlSolid	塗りつぶし(網かけ無し)
xlGray75	75%灰色
xlGray50	50%灰色
xlGray25	25%灰色
xlGray16	12.5%灰色
xlGray8	6.25%灰色
xlHorizontal	横縞
xlVertical	縦縞
xlDown	右下がり縞
xlUp	右上がり縞
xlChecker	斜線格子
xlSemiGray75	極太斜線格子
xlLightHorizontal	横縞(広)

xlLightVertical	縦縞(広)
xlLightDown	右下がり縞(広)
xlLightUp	右上がり縞(広)
xlGrid	格子
xlCrissCross	斜線格子(薄)

実際の記述は次のようになります。

```
Dim interior1 As Interior
Set interior1 = Range("A1").Interior
interior1.Pattern = xlVertical
```

まとめて次のように記述しても構いません。

```
Range("A1").Interior.Pattern = xlVertical
```

次に網の色の指定です。Interior オブジェクトの「PatternColorIndex」プロパティで指定します。「PatternColorIndex」プロパティは文字の色や罫線の色で使ったものと同じです。全部で 57 種類の色が用意されており、0 から 56 のインデックス番号で指定します。

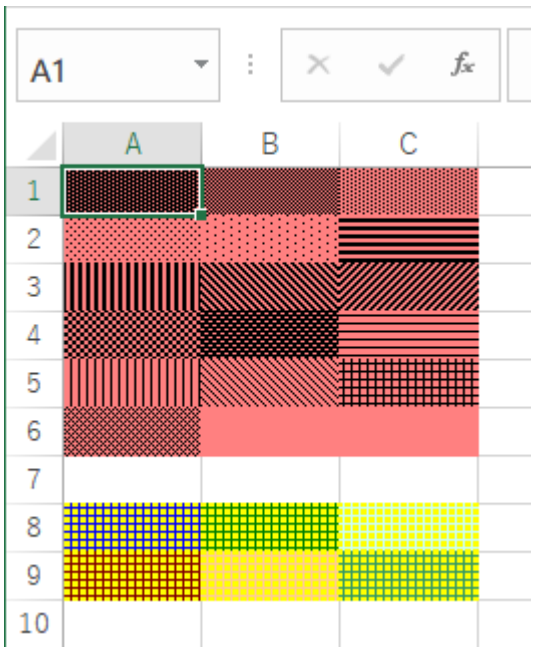
```
Dim interior1 As Interior
Set interior1 = Range("A1").Interior
interior1.PatternColorIndex = 23
```

まとめて次のように記述しても構いません。

```
Range("A1").Interior.PatternColorIndex = 23
```

```
Sub TEST79()
    Range("A1:C6").Interior.ColorIndex = 22

    Range("A1").Interior.Pattern = xlGray75
    Range("B1").Interior.Pattern = xlGray50
    Range("C1").Interior.Pattern = xlGray25
    Range("A2").Interior.Pattern = xlGray16
    Range("B2").Interior.Pattern = xlGray8
    Range("C2").Interior.Pattern = xlHorizontal
    Range("A3").Interior.Pattern = xlVertical
    Range("B3").Interior.Pattern = xlDown
    Range("C3").Interior.Pattern = xlUp
    Range("A4").Interior.Pattern = xlChecker
    Range("B4").Interior.Pattern = xlSemiGray75
    Range("C4").Interior.Pattern                =
```



```
xlLightHorizontal
    Range("A5").Interior.Pattern = xlLightVertical
    Range("B5").Interior.Pattern = xlLightDown
    Range("C5").Interior.Pattern = xlGrid
    Range("A6").Interior.Pattern = xlCrissCross
    Range("B6").Interior.Pattern = xlSolid

    Range("A8:C9").Interior.ColorIndex = 6
    Range("A8:C9").Interior.Pattern = xlGrid
    Range("A8").Interior.PatternColorIndex = 5
    Range("B8").Interior.PatternColorIndex = 10
    Range("C8").Interior.PatternColorIndex = 20
    Range("A9").Interior.PatternColorIndex = 30
    Range("B9").Interior.PatternColorIndex = 40
    Range("C9").Interior.PatternColorIndex = 50

End Sub
```

セルの選択

セルを選択したりアクティブにする方法を見ていきます。

セルを選択する

指定のワークシートの中のセルを選択します。選択したい Range オブジェクトに対して「Select」メソッドを使います。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.Select
```

ワークシートを指定しない場合には現在アクティブになっているワークシートが対象となります。

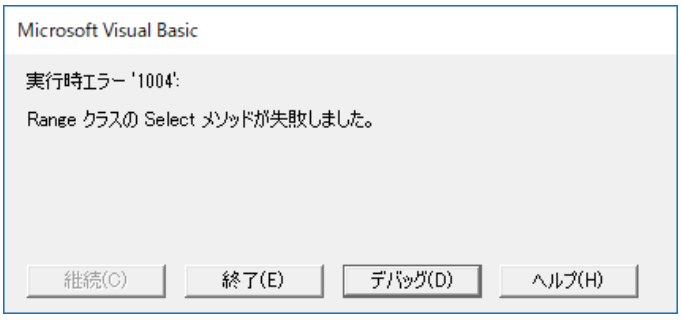
まとめて次のように記述しても構いません。

```
Range("A1").Select
```

ワークシートを指定してセルを選択する場合には、対象となるワークシートがアクティブになっている必要があります。

```
Worksheets(“Sheet2”).Range("A1").Select
```

アクティブでないワークシート内のセルを選択しようとすると次のエラーが表示されます。

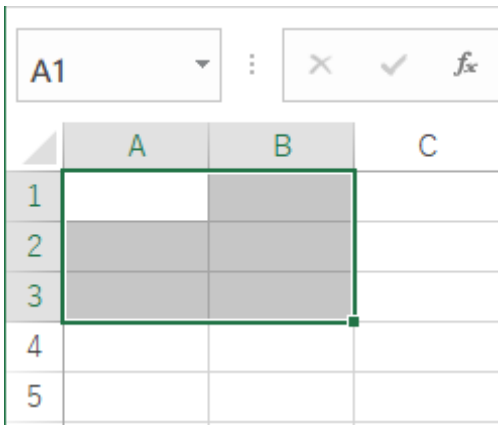


これは選択しようとしたセルが含まれるシートがアクティブになっていないためです。その為、シートを指定してセルを選択するかわりに、先に選択したいセルが含まれるシートをアクティブにした後で、ワークシートを指定せずにセルを選択するようにしましょう。

```
Worksheets(2).Activate
Range("A1").Select
```

またセルを選択すると、それ以前に選択されていたセルは選択が解除されてしまいます。セルを解除せずに追加でセルを選択する方法はないようですので、離れた位置にあるセルを同時に選択する場合には選択したいセルをまとめて Range オブジェクトを作成して下さい。

```
Sub TEST01()
    Dim range1 As Range
    Set range1 = Range("A1:B3")
    range1.Select
End Sub
```



選択されたセルを参照する

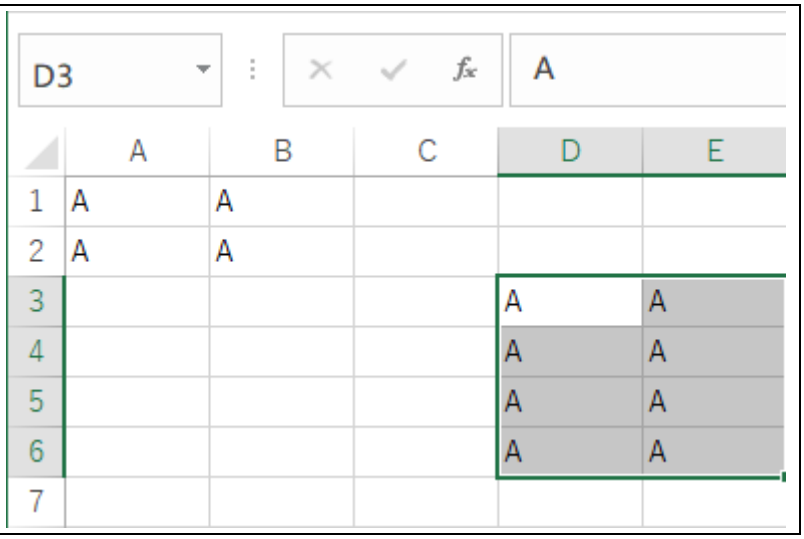
選択された状態のセルは、オブジェクトの「Selection」プロパティを使って取得することができます。対象のオブジェクトは Application オブジェクト又は Window オブジェクトです。オブジェクトが省略された場合はアクティブウィンドウが対象となります。

```
Dim range1 As Range
Range("A1:B2").Select
Set range1 = ActiveWindow.Selection
```

「Selection」プロパティを利用すると、選択範囲に対する処理などを記述する際に簡潔に記述をすることが可能になります。

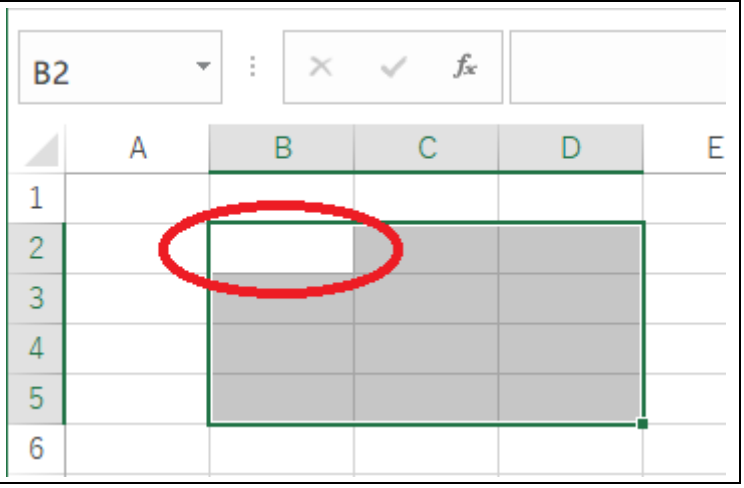
```
Sub TEST02()
    Range("A1:B2").Select
    call setValue
    Range("D3:E6").Select
    call setValue
End Sub

Sub setValue()
    Selection.Value = "A"
End Sub
```



セルをアクティブにする

指定のワークシートの中のセルをアクティブにします。アクティブなセルとは例えば次のようなセルです。

左記の赤丸で囲まれた部分が背景が白色になっていると思います。これがアクティブセルです。アクティブセルは選択されたセルの中に含まれており、1 つのセルだけが選択された状態であればアクティブセルも同じセルとなります。セルの領域が選択状態にある場合は、選択状態の中の 1 つのセルだけがアクティブとなります。	

セルをアクティブにするには、アクティブにしたい Range オブジェクトに対して「Activate」メソッドを使います。

```
Dim range1 As Range
Range("A1").Activate
```

ワークシートを指定しない場合には現在アクティブになっているワークシートが対象となります。

通常アクティブセルは単独のセルですが、セル領域に対して「Activate」メソッドを実行してもエラーとはなりません。その場合、セル領域の左上のセルがアクティブセルとなります。
また Range オブジェクトに対して「Select」メソッドを実行した場合、単独のセルを選択した場合はそのセルが、セル領域を選択した場合は左上のセルがアクティブセルとなります。

セルが選択された状態でアクティブセルを設定した場合

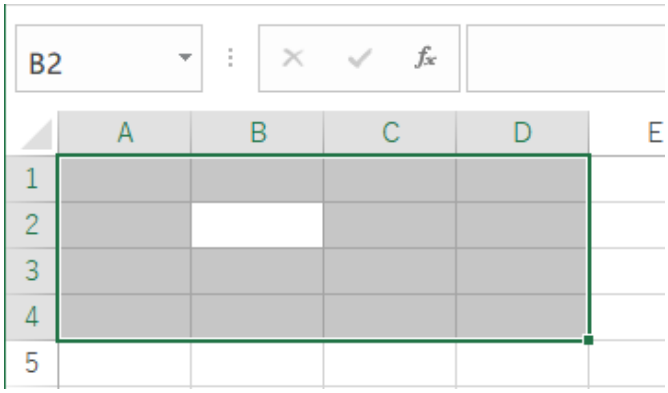
どこかのセルが選択されている状態の時にアクティブセルを設定した場合の挙動について確認します。
まずアクティブに設定するセルが選択された状態のセル領域のどこかであった場合には、セルの選択状態は変更されずにその領域内の指定したセルがアクティブセルとなります。
アクティブに設定するセルが選択された状態のセル領域の中ではなかった場合には、それまで選択されていたセル領域は選択が解除されて、アクティブセルに指定したセルだけがアクティブになります。
アクティブでないワークシートに含まれるセルをアクティブに設定する場合
「Select」メソッドを実行する時と同じように、ワークシートを指定してアクティブセルを設定する場合には、対象のワークシートがアクティブになっていなければなりません。

```
Worksheets(2).Range("A1").Activate
```

上記の場合で、指定したワークシートがアクティブでない場合にはエラーとなります。その為、ワークシートを指定する場合には先にワークシートをアクティブにしてから「Activate」メソッドを実行して下さい。

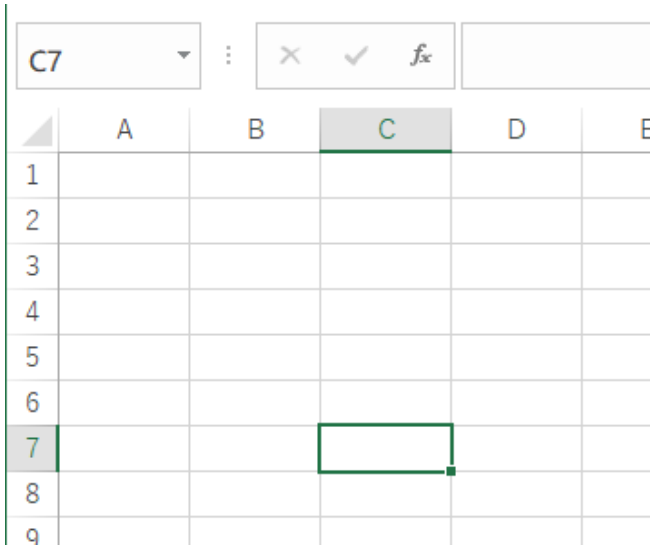
```
Worksheets(2).Activate
Range("A1").Activate
```

```
Sub TEST04()
    Range("A1:D4").Select
    Range("B2").Activate
End Sub
```



先にセル領域を選択してから、選択領域の中の 1 つをアクティブセルに設定しています。

```
Sub TEST05()
    Range("A1:D4").Select
    Range("C7").Activate
End Sub
```



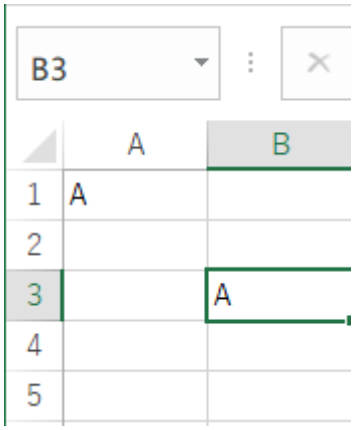
今度も先にセル領域を選択していますが、アクティブセルに設定したセルが選択領域内ではありませんので、選択されていた領域は選択が解除され、アクティブセルに設定したセルだけがアクティブに(同時に選択された状態に)なっています。

アクティブなセルを参照する

現在アクティブになっているセルは、オブジェクトの「ActiveCell」プロパティを使って取得することができます。対象のオブジェクトは Application オブジェクト又は Window オブジェクトです。オブジェクトが省略された場合はアクティブウィンドウが対象となります。

```
Dim range1 As Range
Range("A1").Activate
Set range1 = ActiveWindow.ActiveCell
```

```
Sub TEST06()
    Range("A1").Activate
    call setValue
    Range("B3").Activate
    call setValue
End Sub
Sub setValue()
    ActiveCell.Value = "A"
End Sub
```



セルの編集

セルに対して挿入や削除などを行う方法を確認していきます。

セルの削除

セルを削除する方法を確認します。削除したい Range オブジェクトに対して「Delete」メソッドを使います。セルが削除されると削除されたセルの位置に左方向又は上方向にセルを移動します。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.Delete Shift:=xlShiftToLeft
```

削除後にセルを移動させる方向を「Shift」引数を使って指定します。指定できる値は次のどちらかです。

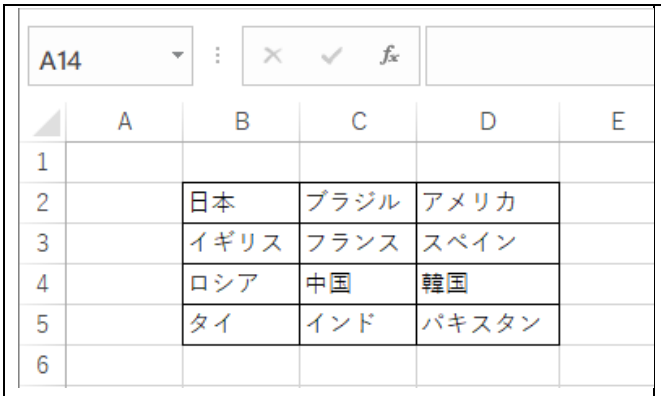
定数	移動方向
xlShiftToLeft	左方向へシフトする
xlShiftUp	上方向へシフトする

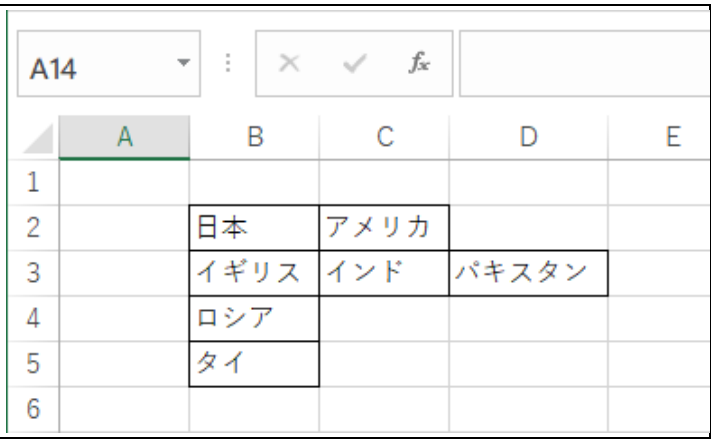
移動方向に関する引数は省略可能です。省略した場合にどちらの方向へセルが移動するかは Excel が判断します。

まとめて次のように記述しても構いません。

```
Range("A1:B3").Delete Shift:=xlShiftUp
```

```
Sub TEST01()
    Dim range1 As Range
    Set range1 = Range("C2")
    range1.Delete Shift:=xlShiftToLeft
    Range("C3:D4").Delete Shift:=xlShiftUp
End Sub
```





セルの挿入

セルを挿入する方法を確認します。挿入したい位置にある Range オブジェクトに対して「Insert」メソッドを使います。挿入される位置にあるセルは右方向又は下方向へ移動します

```
Dim range1 As Range
Set range1 = Range("A1")
range1.Insert Shift:=xlShiftToRight
```

挿入後にセルを移動させる方向を「Shift」引数を使って指定します。指定できる値は次のどちらかです。

定数	移動方向
xlShiftToRight	右方向へシフトする
xlShiftDown	下方向へシフトする

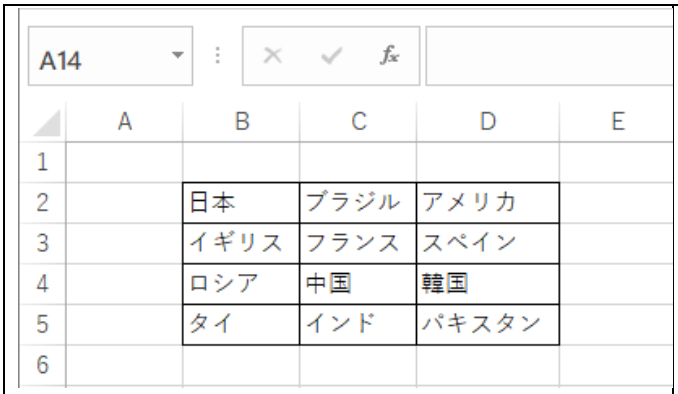
移動方向に関する引数は省略可能です。省略した場合にどちらの方向へセルが移動するかは Excel が判断します。

まとめて次のように記述しても構いません。

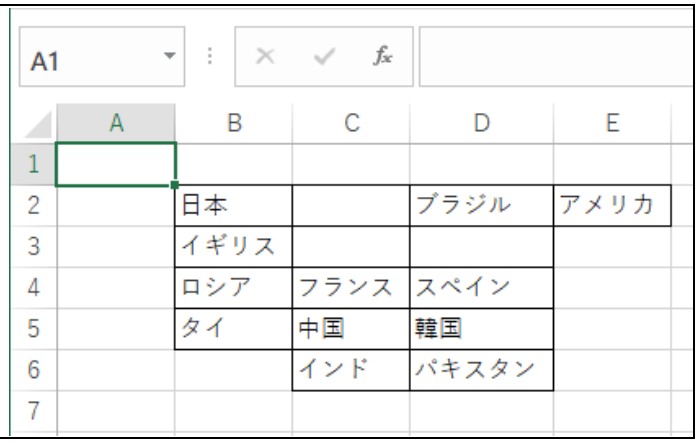
```
Range("A1:B3").Insert Shift:=xlShiftDown
```

```
Sub TEST02()
    Dim range1 As Range
    Set range1 = Range("C2")
    range1.Insert Shift:=xlShiftToRight
    Range("C3:D3").Insert Shift:=xlShiftDown
End Sub
```

セル C2 の位置にセルを挿入し右方向へ移動させ、次にセル範囲 C4:D4 の位置にセルを挿入し下方向へ移動させてみます。



	B	C	D
2	日本	ブラジル	アメリカ
3	イギリス	フランス	スペイン
4	ロシア	中国	韓国
5	タイ	インド	パキスタン



	B	C	D	E
2	日本		ブラジル	アメリカ
3	イギリス			
4	ロシア	フランス	スペイン	
5	タイ	中国	韓国	
6		インド	パキスタン	

セルの切り抜き

セルを切り取る方法を確認します。切り取ったセルをそのまま他の位置に貼り付ける事も出来ますし、クリップボードに保管した後で、「Paste」メソッドで貼り付ける事もできます。

セルを切り取るには、切り取りたい Range オブジェクトに対して「Cut」メソッドを使います。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.Cut Destination:=Range("B1")
```

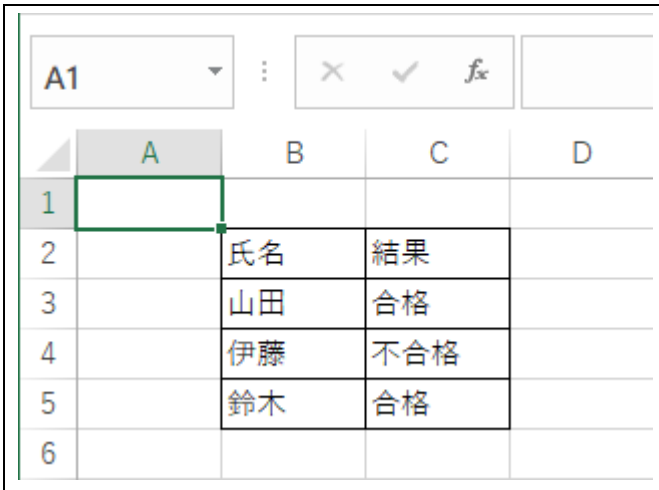
切り取ったセルを直接貼り付ける場合には「Destination」引数を使って貼り付け先の Range オブジェクトを指定します。切り取った Range オブジェクトがセル範囲であった場合でも、貼り付ける先の Range オブジェクトは単独のセルを指定します。このセルは貼り付けるセルの左上のセルになります。

まとめて次のように記述しても構いません。

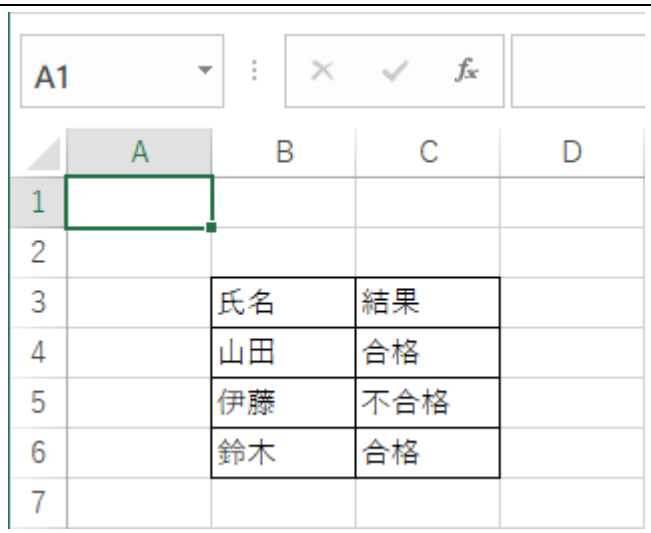
```
Range("A1:B3").Cut Destination:=Range("B1")
```

```
Sub TEST03()
    Dim range1 As Range
    Set range1 = Range("B2:C5")
    range1.Cut Destination:=Range("B3")
End Sub
```

表(セル範囲 B2:C5)をまとめて 1 つ下のセルへ移動させます。



	B	C
2	氏名	結果
3	山田	合格
4	伊藤	不合格
5	鈴木	合格



	B	C
3	氏名	結果
4	山田	合格
5	伊藤	不合格
6	鈴木	合格

セルのコピー

セルをコピーする方法を確認します。コピーしたセルをそのまま他の位置に貼り付ける事も出来ますし、クリップボードに保管した後で、「Paste」メソッドで貼り付ける事もできます。セルをコピーするには、切り取りたい Range オブジェクトに対して「Copy」メソッドを使います。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.Copy Destination:=Range("B1")
```

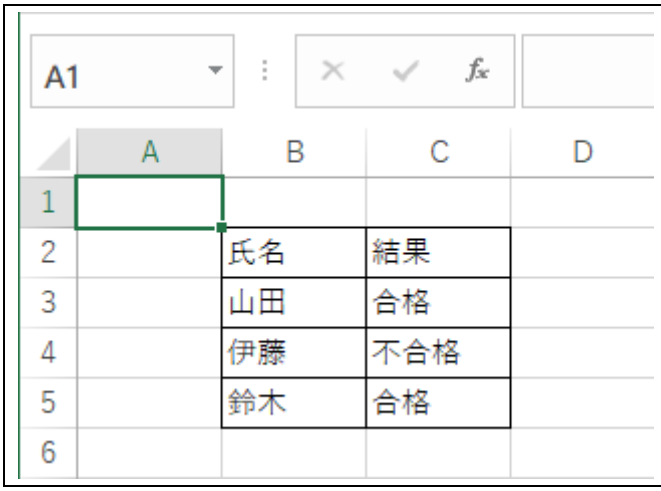
コピーしたセルを直接貼り付ける場合には「Destination」引数を使って貼り付け先の Range オブジェクトを指定します。コピーした Range オブジェクトがセル範囲であった場合でも、貼り付ける先の Range オブジェクトは単独のセルを指定します。このセルは貼り付けるセルの左上のセルになります。

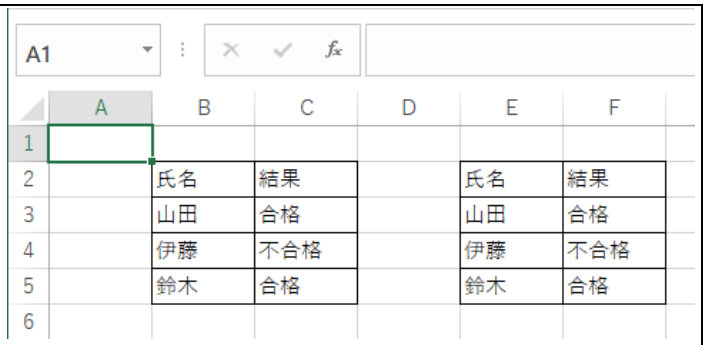
まとめて次のように記述しても構いません。

```
Range("A1:B3").Copy Destination:=Range("B1")
```

```
Sub TEST04()
    Dim range1 As Range
    Set range1 = Range("B2:C5")
    range1.Copy Destination:=Range("E2")
End Sub
```

表(セル範囲 B2:C5)をまとめてコピーし、セル E2 の位置へ貼り付けます。





セルの貼り付け

セルを切り取ったりコピーしたりした後でクリップボードに保存されたデータを貼り付ける方法を確認します。セルを貼り付けるには、Worksheet オブジェクトに対して「Paste」メソッドを使います。

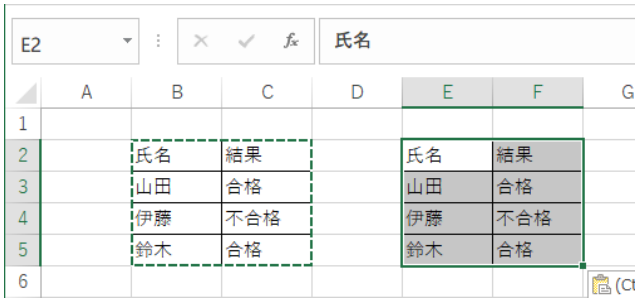
```
Range("A1").Copy
Range("B2").Select
ActiveSheet.Paste
```

シートの中で貼り付けられる場所は、現在シートの中で選択されているセルとなります。また「Destination」引数を使ってデータを貼り付けるセルを指定することも出来ます。

```
Range("A1").Copy
ActiveSheet.Paste Destination:=Range("E2")
```

コピーモード

「Cut」メソッドや「Copy」メソッドを使った場合に「Destination」プロパティをしなかった場合はクリップボードにデータが保存されただけの状態になります。この状態をコピーモードと呼びます。この状態ではコピーなどがされたセル領域が点滅した状態になっています。



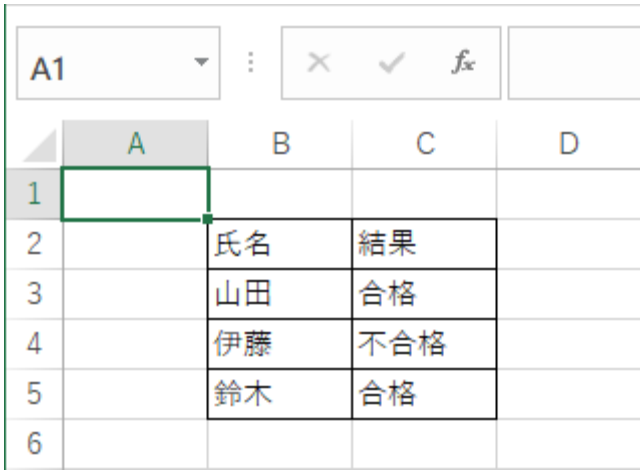
コピーモードにある場合に、他のセルに貼り付けを行うことができます。1 回貼り付けを行っただけではコピーモードは解除されません。

その為、コピーモードにある場合は続けて何度でも貼り付けを行うことが出来ます。
貼り付けが終わってコピーモードを解除する場合には、Application オブジェクトで用意されている「CutCopyMode」プロパティに「False」を設定します。

```
Range("A1").Copy
ActiveSheet.Paste Destination:=Range("B1")
ActiveSheet.Paste Destination:=Range("C1")
Application.CutCopyMode = False
```

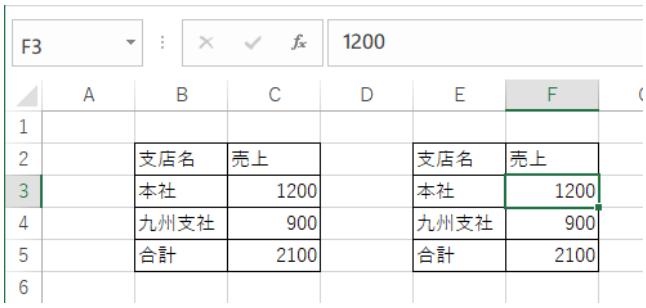
```
Sub TEST05()
    Dim range1 As Range
    Set range1 = Range("B2:C5")
    range1.Copy
    ActiveSheet.Paste Destination:=Range("E2")
    ActiveSheet.Paste Destination:=Range("B7")
    Application.CutCopyMode = False
End Sub
```

表(セル範囲 B2:C5)をまとめてコピーし、セル E2 とセル B7 の位置へ貼り付けます。

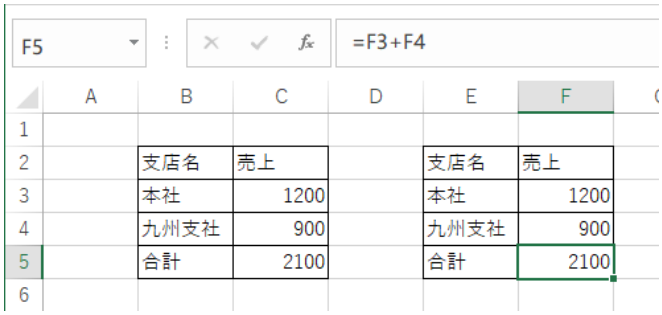


セルのリンク貼り付け

セルを単にコピーした貼り付けた場合には、コピー元の値がそのまま貼り付けられています。

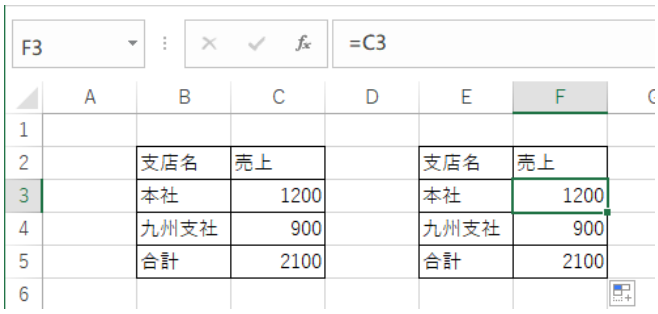


左記はセル範囲 B2:C5 をセル E2 に貼り付けたものです。例えばセル F3 の値はコピー元の値である「1200」と言う数値がそのままコピーされています。



また計算式などもセル F5 の値を見て頂くと分かる通り、コピー元で相対参照の形式で上 2 つのセルの合計を表す式が入力されていたので、コピーされたセル F5 の値も相対参照によって上 2 つのセルの合計が入るような計算式が入っています。

このように単にコピー元の値をコピーするのではなく、コピー元の値をそのまま参照するようにコピーすることが可能です。これをリンク貼り付けと言います。



左記はリンク貼り付けを行った例です。セル F3 の値を見てください。セル F3 にはコピー元のセルに入っていた値がコピーされるのではなく、コピー元のセルを参照するように設定されています。このため、コピー元のセル C3 の値を変更すると、自動的にセル F3 の値も変更されます。

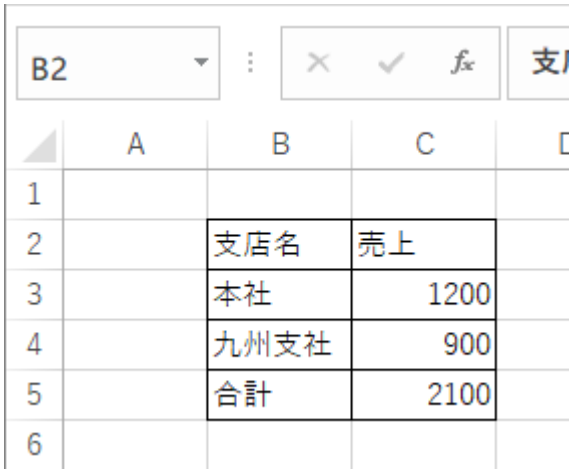
通常の貼り付けではなく、リンク貼り付けを行う場合には、「Paste」メソッドを実行する際に「Link」引数に「True」を設定します。

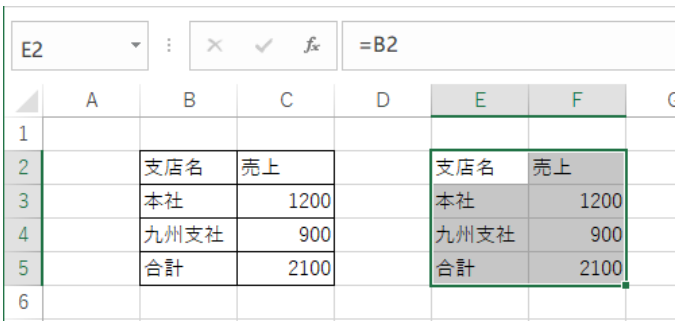
```
Range("A1").Copy
Range("B2").Select
ActiveSheet.Paste Link:=True
```

注意点としては「Link」引数を指定する場合には、「Destination」引数は同時に指定できないことです。その為、貼り付ける時点で選択されていたセルに貼り付けが行われます。
またリンク貼り付けを行う場合には、書式はコピーされないようですので注意して下さい。

```
Sub TEST06()
    Dim range1 As Range
    Set range1 = Range("B2:C5")
    range1.Copy
    Range("E2").Select
    ActiveSheet.Paste Link:=True
    Application.CutCopyMode = False
End Sub
```

表(セル範囲 B2:C5)をまとめてコピーし、セル E2 の位置へリンク貼り付けをします。

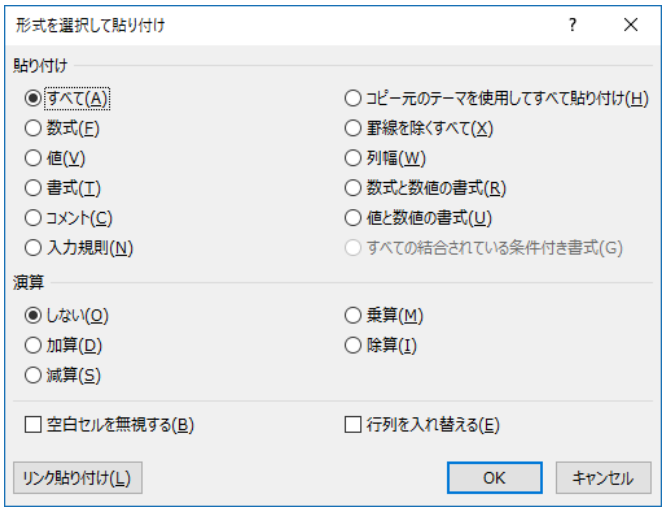




形式を選択して貼り付け

セルを貼り付ける際に、書式だけをコピーしたかったり値としてコピーしたかったりする場合があります。このような場合には形式を指定して貼り付けを行います。

Excel で貼り付けを行う場合にも「形式を選択して貼り付け」を行うと右記のようなウィンドウが表示されますが、これと同じことを VBA 上で行うことが出来ます。



形式を選択して貼り付けを行う場合には、貼り付けたい場所にある Range オブジェクトに対して「PasteSpecial」メソッドを使います。(Paste メソッドは Worksheet オブジェクトに対してでしたが、PasteSpecial メソッドは Range オブジェクトに対して行いますので注意して下さい)。

```
Range("A1").Copy
Range("B2").PasteSpecial Paste:=xlPasteAll, Operation:=xlPasteSpecialOperationNone
```

まず「Paste」引数に何を貼り付けるかを指定します。設定可能な値は以下の通りです。

定数	貼り付ける内容
xlPasteAll	すべて
xlPasteFormulas	数式
xlPasteValues	値
xlPasteFormats	書式
xlPasteComments	コメント
xlPasteValidation	入力規則
xlPasteAllExceptBorders	罫線を除くすべて
xlPasteColumnWidths	列幅
xlPasteFormulasAndNumberFormats	数式と数値の書式
xlPasteValuesAndNumberFormats	値と数値の書式

省略した場合は「xlPasteAll」がデフォルトの値となります。

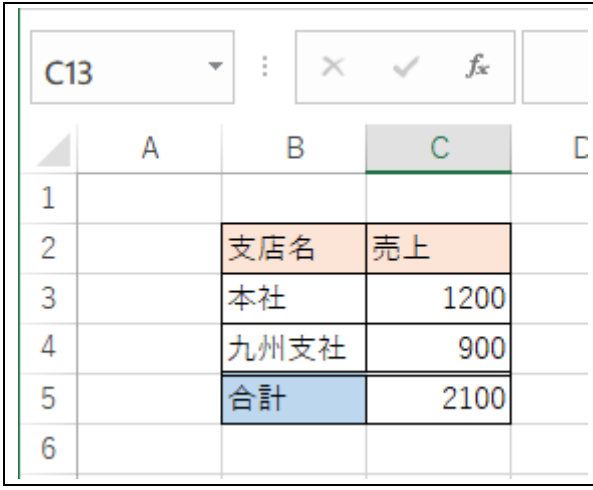
次に「Operation」引数に演算方法を指定します。設定可能な値は以下の通りです。

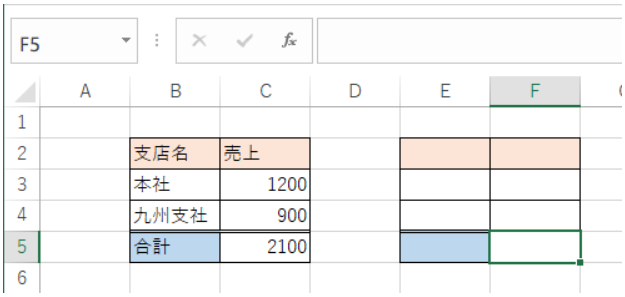
定数	貼り付ける内容
xlPasteSpecialOperationNone	演算をしない
xlPasteSpecialOperationAdd	加算
xlPasteSpecialOperationSubtract	減算
xlPasteSpecialOperationMultiply	乗算
xlPasteSpecialOperationDivide	除算

省略した場合は「xlPasteSpecialOperationNone」がデフォルトの値となります。
また「SkipBlanks」引数に「True」を設定すると、空白セルを無視して貼り付けが行われます。省略可能でデフォルトは「False」です。「Transpose」引数に「True」を指定すると行と列を入れ替えて貼り付けが行われます。省略可能でデフォルトは「False」です。
このように Excel で指定可能な値にそれぞれ対応しています。

```
Sub TEST07()
    Dim range1 As Range
    Set range1 = Range("B2:C5")
    range1.Copy
    Range("E2").PasteSpecial
Paste:=xlPasteFormats
    Application.CutCopyMode = False
End Sub
```

表(セル範囲 B2:C5)をまとめてコピーし、セル E2 の位置へ書式だけを貼り付けます。





※見やすいように、貼り付け後に他のセルをクリックして選択を解除しています。

数式と値のクリア

指定したセル領域の数式と値をクリアします。クリアしたい Range オブジェクトに対して「ClearContents」

メソッドを使います。

```
Dim range1 As Range
Set range1 = Range("A1:B3")
range1.ClearContents
```

まとめて次のように記述しても構いません。

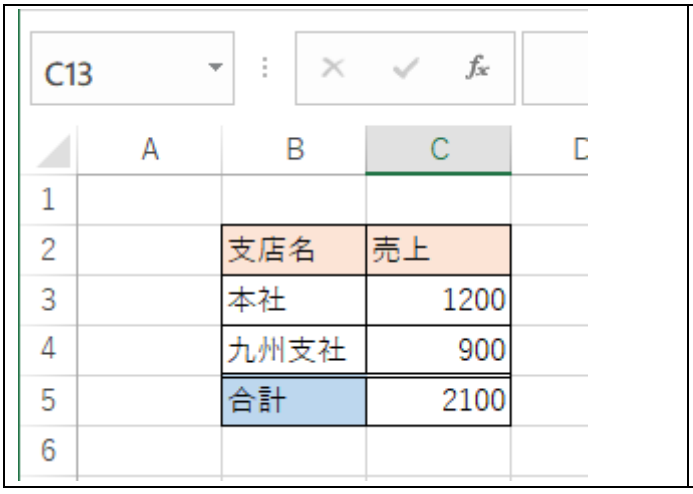
```
Range("A1:B3").ClearContents
```

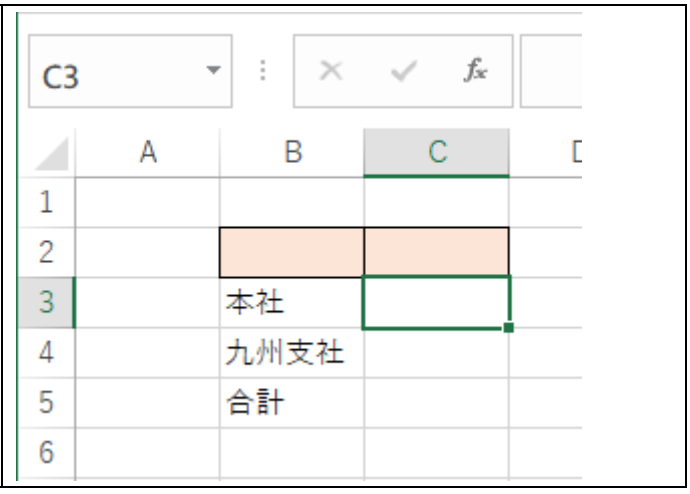
また同じようなメソッドとして、書式だけをクリアするには「ClearFormats」メソッドを使い、数式と値と書式を全てクリアするには「Clear」メソッドを使います。

```
Range("A1:B3").ClearFormats
Range("A1:B3").Clear
```

```
Sub TEST08()
    Range("B2:C2").ClearContents
    Range("B3:B5").ClearFormats
    Range("C3:C5").Clear
End Sub
```

セル領域 B2:C2 については数式と値をクリアし、セル領域 B3:B5 については書式をクリアし、セル領域 C3:C5 については全てを消去してみます





セルを結合する

セルの結合や解除の方法を確認していきます。

セルの結合

指定したセル範囲を結合します。セル範囲は長方形の領域である必要があります。結合されたセルには結合する領域の左上にあるセルの値が表示されます。それ以外のセルに含まれていた値は全て削除されます。

セルを結合するには、結合するセル範囲を表す Range オブジェクトの「MergeCells」プロパティに「True」を設定します

```
Dim range1 As Range
Set range1 = Range("A1:C3")
```

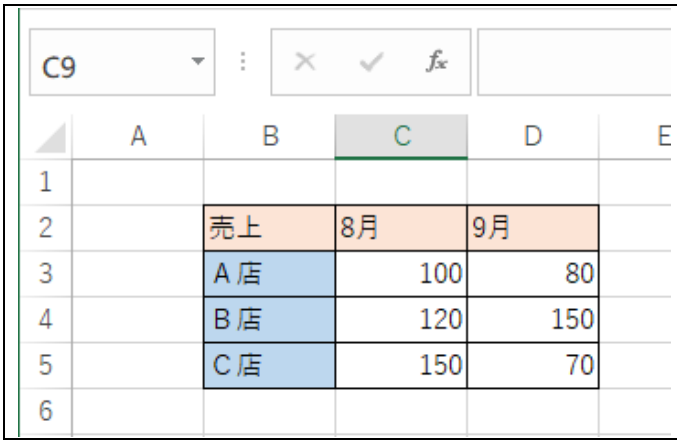
```
range1.MergeCells = True
```

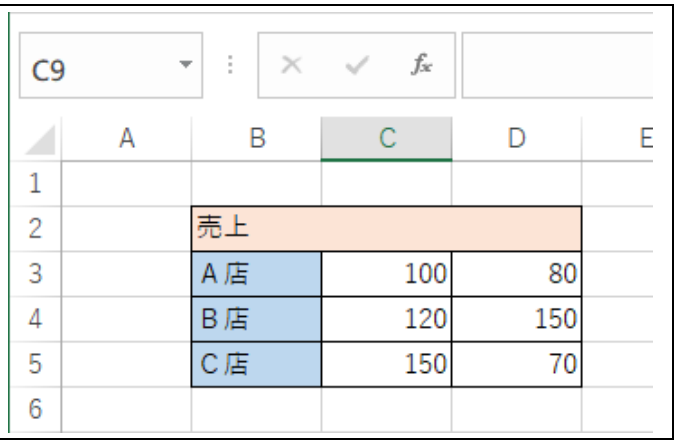
まとめて次のように記述しても構いません。

```
Range("A1:C3").MergeCells = True
```

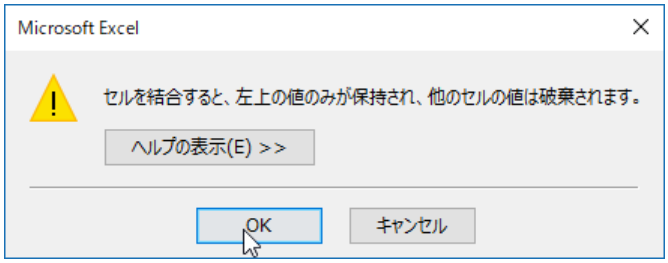
```
Sub TEST01()
    Dim range1 As Range
    Set range1 = Range("B2:D2")
    range1.MergeCells = True
End Sub
```

セル範囲 B2:D2 を結合してみます。





セルが結合されます。結合されたセルには、結合前の左上にあったセル B2 の値が表示されています。セルが結合される時に、左上以外のセルに値が入っている場合は右記のような警告ウィンドウが表示されます。(先ほどのサンプルでも表示されました)。



警告ウィンドウが表示されるといったんプログラムの流れが切れます。もし警告ウィンドウを表示したくない場合には Application オブジェクトの「DisplayAlerts」プロパティで設定を行います。

```
Application.DisplayAlerts = False
処理
Application.DisplayAlerts = True
```

セル結合の解除

結合されたセルを解除する方法を確認します。解除するには、結合するセルの元になった複数のセルの中のどれか 1 つのセルを表す Range オブジェクトの「MergeCells」プロパティに「False」を設定します。

```
Dim range1 As Range
Set range1 = Range("A1")
range1.MergeCells = False
```

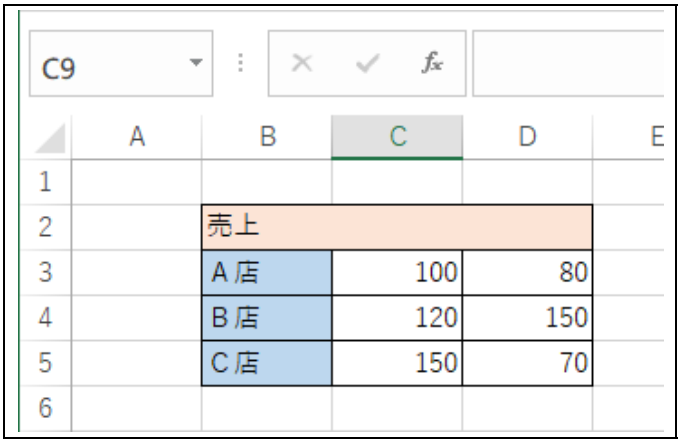
まとめて次のように記述しても構いません。

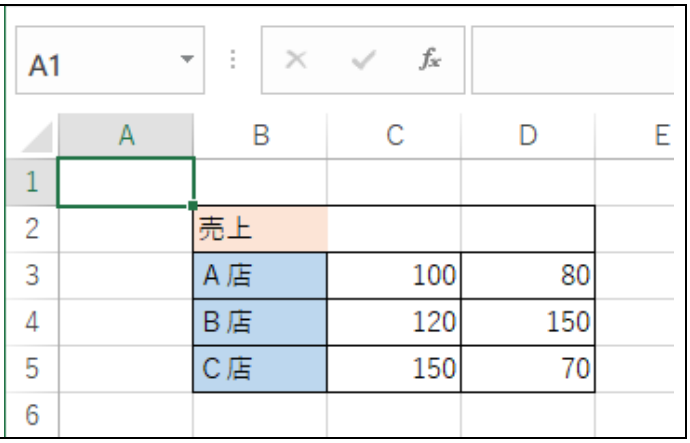
```
Range("A1").MergeCells = False
```

結合を解除するのに指定するセルは、結合されているセルに元々含まれていたセルであれば、どのセルであっても構いません。

```
Sub TEST02()
    Dim range1 As Range
    Set range1 = Range("C2")
    range1.MergeCells = False
End Sub
```

セル B2 の位置にあるセルの結合を解除します。





少し分かりにくいですが、セル B2 の位置にあったセルは結合が解除されて、それぞれセル B2、セル C2、セル D2 になっています。

結合されたセルの参照

結合セルに対して新しい値を設定したり書式を変更する場合に、結合セルを表す Range オブジェクトを取得する方法を確認します。

結合セルに対する Range オブジェクトを取得するには、結合される前のセル領域に含まれるどこか 1 つのセルを表す Range オブジェクトの「MergeArea」プロパティの値を取得します。

```
Dim range1 As Range
Set range1 = Range("A1").MergeArea
```

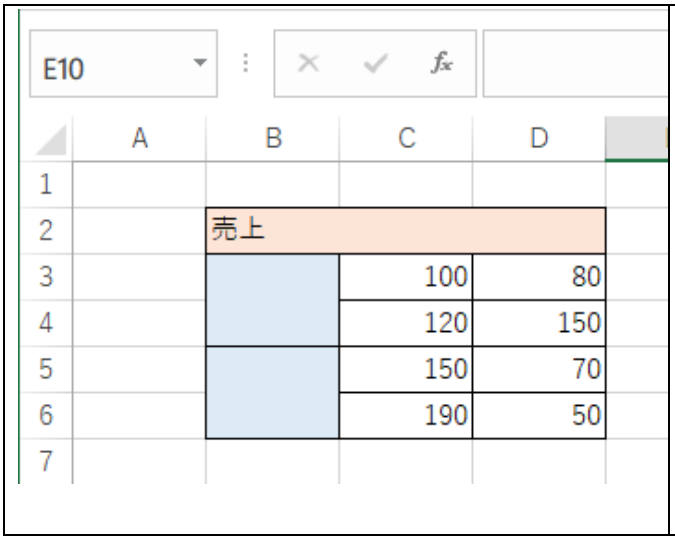
上記の場合は、セル A1 が含まれる結合セルを表す Range オブジェクトを取得しています。「MergeArea」プロパティは必ず単一のセルに対して取得しなくてはいけません。

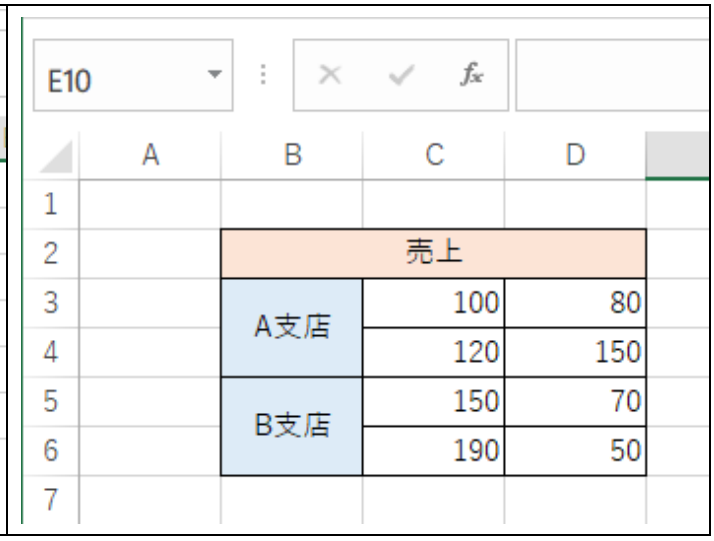
結合セルの Range オブジェクトを取得したら、あとは通常の Range オブジェクトに対する処理を行うことが出来ます。

```
Sub TEST03()
    Dim range1 As Range
    Set range1 = Range("C2").MergeArea
    range1.HorizontalAlignment = xlCenter
    Range("B3").MergeArea.Value = "A 支店"
```

結合されたセルに対して値の設定と文字位置の変更をしてみます。

```
    Range("B6").MergeArea.Value = "B 支店"
End Sub
```





その他の操作

セルに対するその他の操作について確認していきます。

ロックの解除

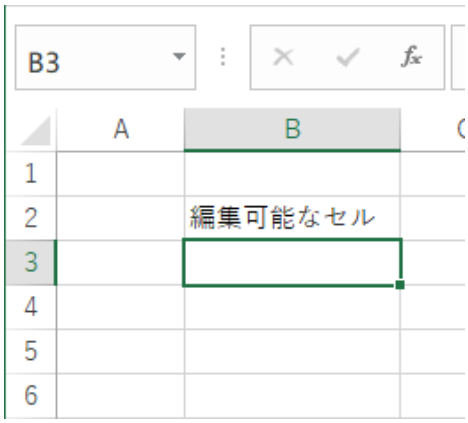
ワークシートを保護する際にロックされているセルが保護の対象となります。デフォルトでは全てのセルがロックされていますが、セルのロックを解除することでワークシートが保護された時にも編集が可能となります。セルのロックを解除するには、対象となるセル範囲を表す Range オブジェクトの「Locked」プロパティに「False」を設定します。

```
Dim range1 As Range
Set range1 = Range("A1:C3")
range1.Locked = False
```

まとめて次のように記述しても構いません。

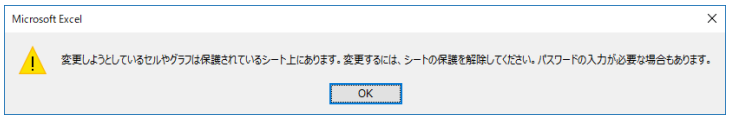
```
Range("A1:C3").Locked = False
```

```
Sub TEST04()
    Range("B2").Locked = False
    ActiveSheet.Protect
End Sub
```



ワークシートは保護されましたのでセルの編集は行えませんが、セル B2 に関してはロックを解除してあるため編集が可能です。

例えばセル B2 以外のセルを編集しようとすると右記のような警告ウィンドウが表示されます。



ワークシートとブックの操作

VBA を使ってワークシートやブックの作成などワークシートとブックの操作方法です。

ワークシートの参照

ブックの中に含まれるワークシートを選択したりアクティブにする方法について確認します。

Worsheet オブジェクトの取得

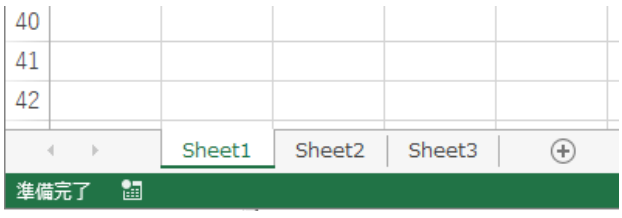
ワークシートを表すオブジェクトは Worksheet オブジェクトです。ワークシートは現在作業しているブックに含まれるワークシートの他、別のブックのワークシートでも取り出すことが出来ます。

Worksheet オブジェクトを取得するには、Application オブジェクト又は Workbook オブジェクトの「Worsheets」プロパティを使います

```
Dim ワークシート As Worksheet
Set ワークシート ＝ オブジェクト.Worksheets(インデックス番号)
```

Application オブジェクトの「Worksheets」プロパティを使った場合には現在作業しているブックに含まれるワークシートが対象となります。オブジェクトを省略した場合も同様です。Workbook オブジェクトは別のブックに含まれるワークシートを取得したい場合に使います。

「Worksheets」プロパティの引数にはインデックス番号を指定します。インデックス番号はブックに含まれるシートの中で左から順に 1,2,3,...の順となっています。



左記の場合で言えばインデックス番号は「Sheet1」が 1 番、「Sheet2」が 2 番、「Sheet3」が 3 番となります。ワークシートの位置が変わったりワークシートが削除されたりするとそれに合わせてインデックス番号は左から順に振りなおされますので、インデックス番号だけでワークシートを常に特定することは出来ません。

具体的な記述方法は次のようになります。

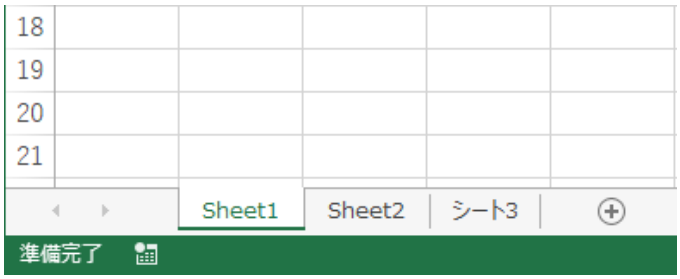
```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
```

次にインデックス番号の代わりにワークシート名を「Worksheets」プロパティの引数に指定することも出来ます。

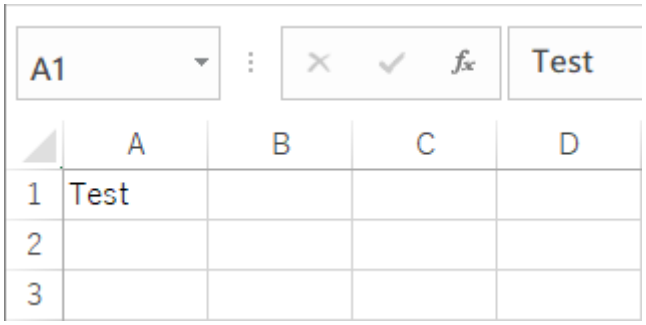
```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets("Sheet1")
```

ワークシート名を指定する場合には、ワークシートの並び順に関係無く常に特定のワークシートを対象とした Worksheet オブジェクトを取得できます。

```
Sub TEST01()
    Dim sheet1 As Worksheet
    Set sheet1 = Worksheets(2)
    sheet1.Range("A1").Value = "Test"
    Worksheets("Sheet3").name = "シート 3"
End Sub
```



Worksheet オブジェクトの「Name」プロパティを使って 3 番目のワークシートのシート名を変更しました。また「Sheet2」のワークシートをアクティブにして下さい。



このように元々もアクティブになっていなかったワークシートに含まれるセルに対しても、「ワークシート.Range オブジェクト」の形で指定することで操作することが可能となります。

シートをアクティブにする

指定したワークシートをアクティブにします。アクティブになったワークシートと言うのは、一番前面に表示されているワークシートのことです。一度にアクティブにできるワークシートは 1 つだけです。
ワークシートをアクティブにするには、アクティブにしたい Worksheet オブジェクトに対して「Activate」メソッドを使います。

```
Dim ワークシート As Worksheet
ワークシート.Activate
```

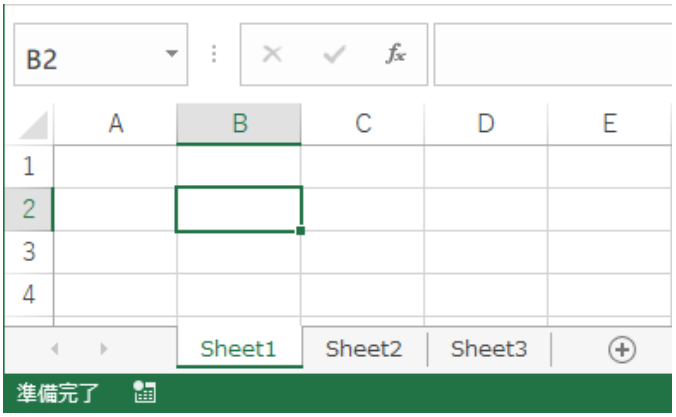
ワークシートはインデックス番号又はワークシート名を指定して取得した Worksheet オブジェクトなどに対してメソッドを実行して下さい。

具体的な記述方法は次のようになります。

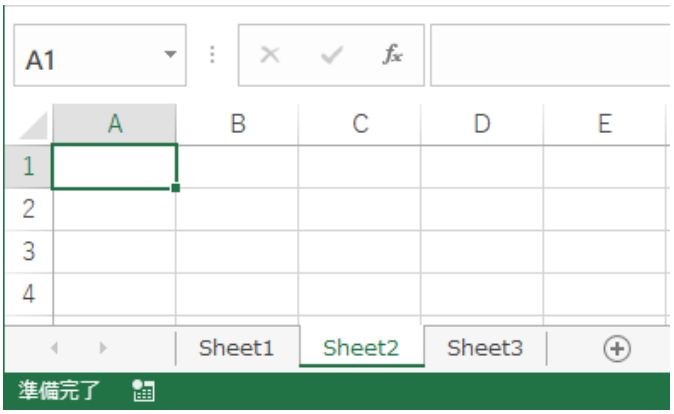
```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
```

```
sheet1.Activate
```

```
Sub TEST02()
    Dim sheet1 As Worksheet
    Set sheet1 = Worksheets(2)
    sheet1.Activate
End Sub
```



左から数えて2番目のシートがアクティブ化されて一番前面に表示されました。



アクティブシートのオブジェクトの取得

ワークシートのオブジェクトを取得するには、インデックス番号を指定するかワークシート名を指定してWorksheet オブジェクトを取得していましたが、取得する Worksheet オブジェクトを指定する方法として、現在アクティブになっているワークシートと指定することが出来ます。

アクティブなワークシートの Worksheet オブジェクトを取得するには、Application オブジェクト、Window オブジェクト、又は Workbook オブジェクトの「ActiveSheet」プロパティを使います。

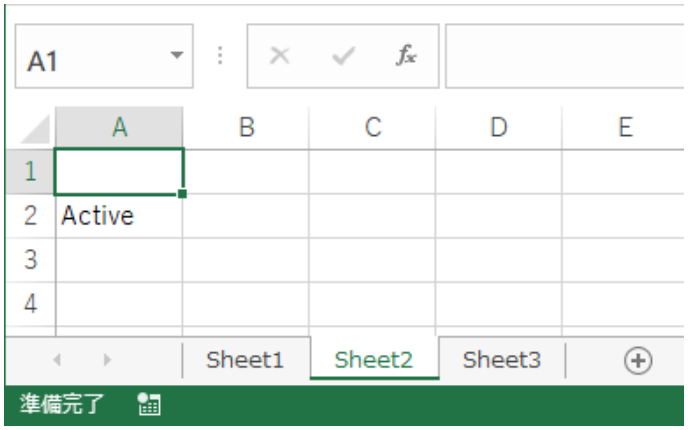
```
Dim ワークシート As Worksheet
Set ワークシート = オブジェクト.ActiveSheet
```

オブジェクトを省略した場合には、現在作業しているワークブックが対象となります。

具体的な記述方法は次のようになります。

```
Dim sheet1 As Worksheet
Set sheet1 = ActiveSheet
```

```
Sub TEST03()
    Dim sheet1 As Worksheet
    Worksheets(2).Activate
    Set sheet1 = ActiveSheet
    sheet1.Range("A2").Value = "Active"
End Sub
```



今回はまず左から数えて 2 番目のシートがアクティブにした後で、アクティブシートの中にあるセルに対して値を設定しました。

シートを選択する

指定したワークシートを選択します。複数のシートを同時に選択することも可能です。アクティブと選択の違いは、1 枚だけ選択した場合には選択されたシートはアクティブシートでもあります。複数のシートを選択した場合には、その選択されたシートの中で一番上に表示されていて作業可能になっているシートがアクティブシートとなります。

シートを選択するには、選択したい Worksheet オブジェクトに対して「Select」メソッドを使います。

```
Dim ワークシート As Worksheet
ワークシート.Select
```

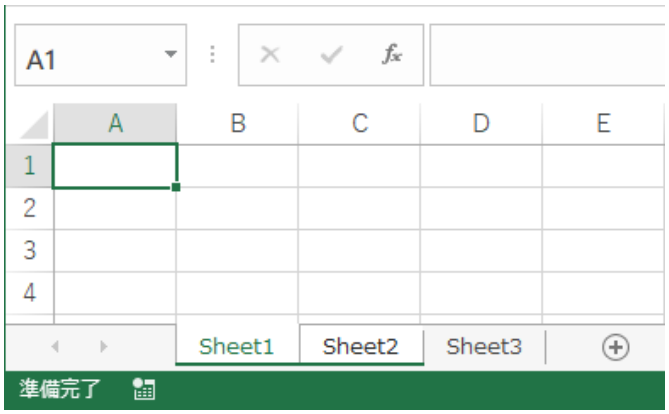
デフォルトではワークシートを選択するとそれ以前に選択されていたシートは非選択状態となります。同時に複数のシートを選択する場合には引数「Replace」に「False」を指定して下さい。

```
Dim ワークシート As Worksheet
ワークシート.Select Replace:=False
```

具体的な記述方法は次のようになります。

```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
sheet1.Select
Worksheets(2).Select Replace:=False
```

```
Sub TEST04()
    Dim sheet1 As Worksheet
    Set sheet1 = Worksheets(1)
    sheet1.Select
    Worksheets(2).Select Replace:=False
End Sub
```



左から 1 枚目と 2 枚目のシートを同時に選択しています。

複数のシートを一度に選択する

シートを選択する場合、Worksheet オブジェクトに対して「Select」メソッドを実行しますが、複数のワークシートに対してまとめて「Select」メソッドを実行することが出来ます。

Worksheet オブジェクトを取得する際に、Application オブジェクトの「Worksheets」プロパティなどを使いますが、今までのサンプルではインデックス番号かワークシート名を 1 つだけ記述していました。

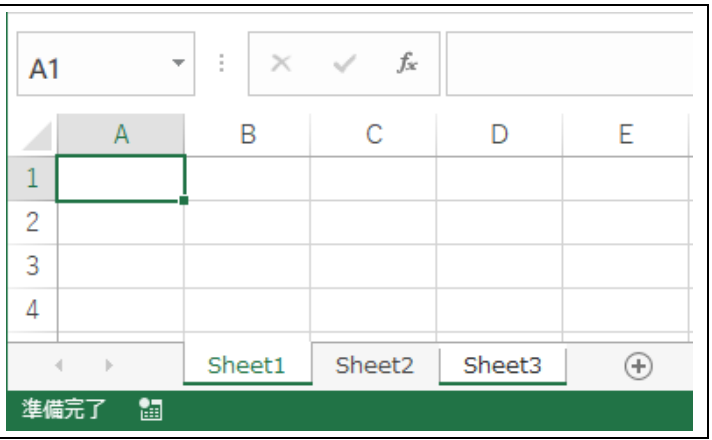
```
Worksheets(1).Select
Worksheets("Sheet1").Select
```

複数のワークシートを対象にする場合には Array 関数を使って複数のインデックス番号やシート名を同時に指定します。

```
Worksheets(Array(1, 2, 3)).Select
Worksheets(Array("Sheet1", "Sheet2", "Sheet3")).Select
```

このように Array 関数を使ってインデックス番号又はシート名を複数指定することで、複数のワークシートを対象とすることが出来ます。

```
Sub TEST05()
    Worksheets(Array(1, 3)).Select
End Sub
```



左から 1 枚目と 3 枚目のシートを一度に選択しています。

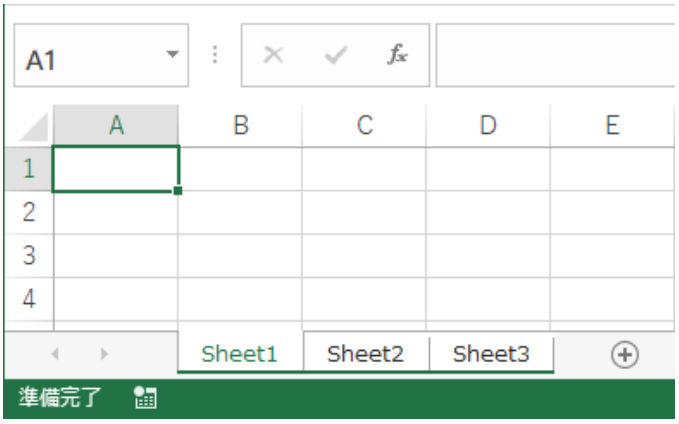
全てのシートを選択する

ブックに含まれる全てのシートをまとめて選択したい場合には、個々の Worksheet オブジェクトではなく Worksheets コレクションに対して「Select」メソッドを実行します。

```
Worksheets.Select
```

Worksheets コレクションは(対象となるブックに含まれる)全てのワークシートの集まりですので、ワークシート全てが選択されることになります。

```
Sub TEST06()
    Worksheets.Select
End Sub
```



全てのワークシートが一度に選択されました。

ワークシートの追加

ワークシートの追加や削除などの方法について確認します。

ワークシートの追加

ワークシートを追加します。追加する位置と追加する枚数を指定できます。

ワークシートへの追加は Worksheets コレクションに対して「Add」メソッドを使います。

```
Worksheets.Add After:=Worksheets(1)
```

追加する位置は既にあるシートの前又は後という形で指定します。「Before」引数を使う場合には指定したワークシートの前(左側)に追加されます。「After」引数を使う場合には指定したワークシートの後(右側)に追加されます。どちらも Worksheet オブジェクトを設定する値に使います。

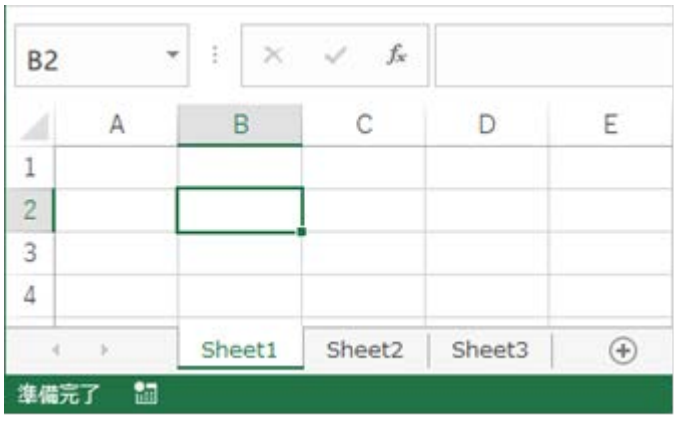
```
Worksheets.Add Before:=Worksheets("Sheet2")
```

「Before」も「After」も指定しない場合には、現在のアクティブシートの前に追加されます。

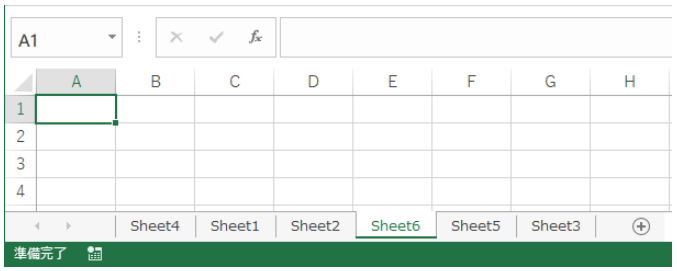
また追加する枚数も指定できます。デフォルトでは 1 枚ですが「Count」引数で追加する枚数を指定できます。

```
Worksheets.Add Before:=Worksheets("Sheet2"), Count:=2
```

右記のような Excel のファイルを用意します。
先頭のシートの前に 1 枚追加し、「Sheet2」シートの後ろに 2 枚追加してみます。



```
Sub TEST07()
    Worksheets.Add Before:=Worksheets(1)
    Worksheets.Add _
    After:=Worksheets("Sheet2"), count:=2
End Sub
```



まず 1 枚目が「Sheet1」シートの前に追加されます。そして「Sheet2」シートの後ろに 2 枚追加されます。2 枚追加される時ですが、「Sheet5」シートがまず追加されさらに「Sheet6」シートが(「Sheet2」シートの後ろに)追加されています。つまり 2 枚同時に追加されるのではなく、1 枚ずつ 2 回追加されるようです。

ワークシートの移動

ワークシートを移動します。移動する位置を指定できます。

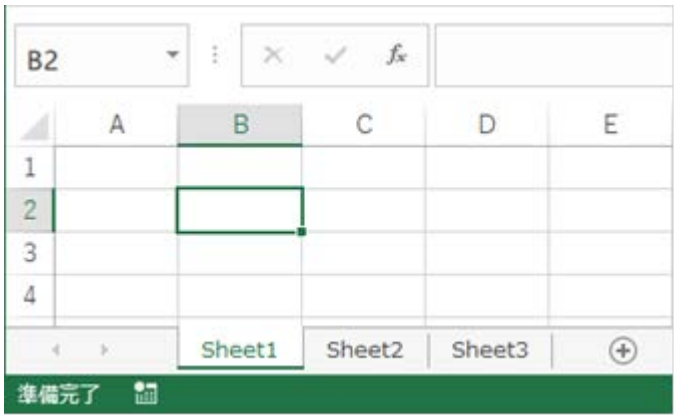
ワークシートへの移動は Worksheet オブジェクトに対して「Move」メソッドを使います。

```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
sheet1.Move Before:=Worksheets("Sheet2")
```

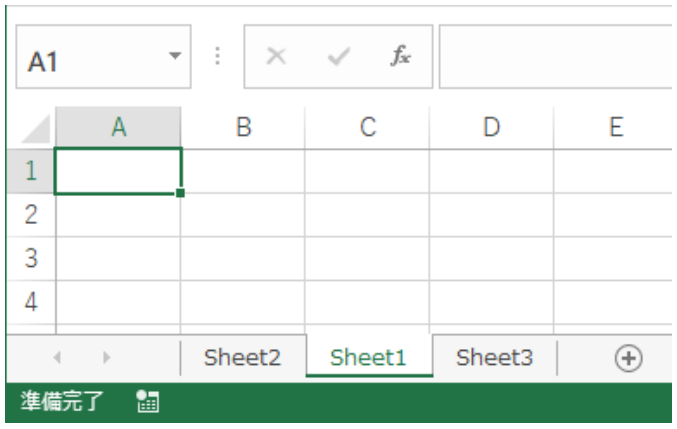
移動する位置は既にあるシートの前又は後という形で指定します。「Before」引数を使う場合には指定したワークシートの前(左側)に移動します。「After」引数を使う場合には指定したワークシートの後(右側)に移動します。どちらも Worksheet オブジェクトを設定する値に使います。

注意する点としては、移動先を指定する引数を省略した場合、新しいブックが作成されてそのブックに移動します。

右記のような Excel のファイルを用意します。
先頭のシートを「Sheet2」シートの後ろに移動させてみます。




```
Sub TEST08()
    Worksheets(1).Move _
        After:=Worksheets("Sheet2")
End Sub
```



ワークシートのコピー

ワークシートをコピーします。コピーしたセルを貼り付ける位置を指定できます。(コピーしてから貼り付けるというよりも、既にあるワークシートをコピーしたものを指定の位置に移動させるという感じです。その為コピーと貼り付けは Copy メソッドでまとめて行います)。

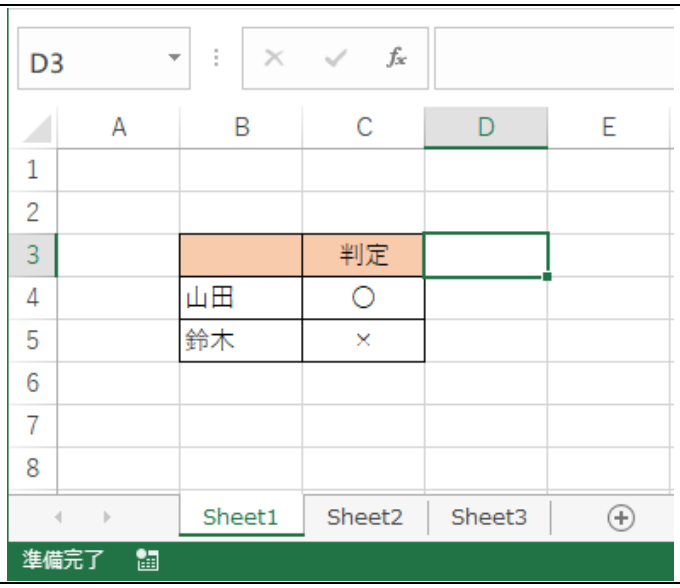
ワークシートへの移動は Worksheet オブジェクトに対して「Copy」メソッドを使います。

```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
sheet1.Copy Before:=Worksheets("Sheet2")
```

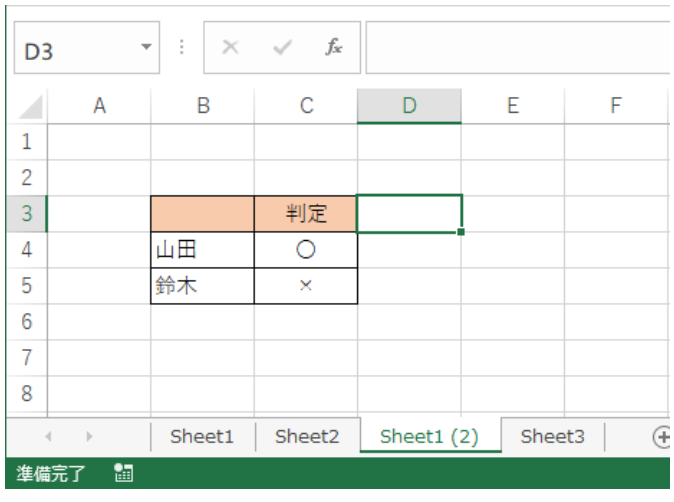
コピーしたものを貼り付ける位置は既にあるシートの前又は後という形で指定します。「Before」引数を使う場合には指定したワークシートの前(左側)にコピーしたワークシートが貼り付けられます。「After」引数を使う場合には指定したワークシートの後(右側)に貼り付けられます。どちらも Worksheet オブジェクトを設定する値に使います。

注意する点としては、貼り付け先の引数を省略した場合、新しいブックが作成されてそのブックに貼り付けが行われます。また同じ名前のワークシートが貼り付け先のブックに既にある場合には、名前の後ろに括弧つきの番号「(2)や(3)」がワークシートの名前の後ろに順に付いていきます。

右記のような Excel のファイルを用意します。
先頭のシートをコピーし、「Sheet2」シートの後ろに貼り付けます。




```
Sub TEST09()
    Worksheets(1).Copy _
        After:=Worksheets("Sheet2")
End Sub
```



ワークシートの削除

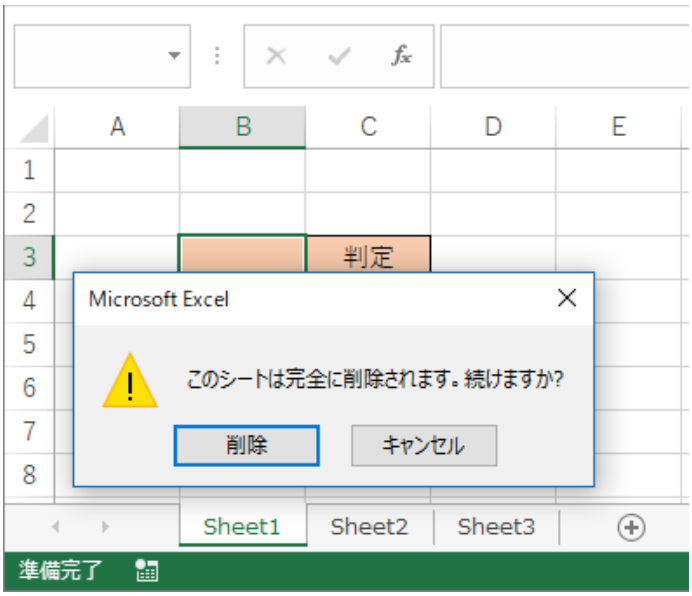
ワークシートを削除します。削除したいワークシートを表す Worksheet オブジェクトに対して「Delete」メソッドを使います。

```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
sheet1.Delete
```

まとめて次のように記述して頂いても構いません。

```
Worksheets(1).Delete
```

```
Sub TEST10()
    Worksheets(2).Delete
End Sub
```



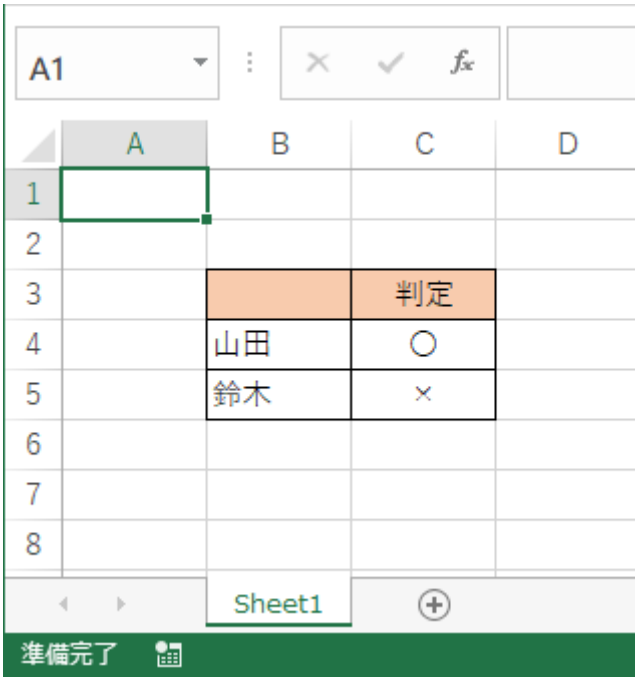
ワークシートを削除する場合には警告ウィンドウが表示されます。もし問題がなければ「削除」ボタンをクリックして下さい。

2 枚目にあったワークシートが削除されました。

警告ウィンドウを表示せずにワークシートを削除する

ワークシートの削除のように警告ウィンドウが表示されるといったんプログラムの流れが切れます。もし警告ウィンドウを表示したくない場合には Application オブジェクトの「DisplayAlerts」プロパティで設定を行います。

```
Sub TEST11()
    Application.DisplayAlerts = False
    Worksheets(2).Delete
    Application.DisplayAlerts = True
End Sub
```



今度は警告ウィンドウが表示されずにワークシートが削除されました。

ワークシートの操作

ワークシートの表示/非表示の操作や保護の設定などの方法について確認します。

表示/非表示の切り替え

ワークシートの表示を制御します。Worksheet オブジェクトの「Visible」プロパティに値を設定します。

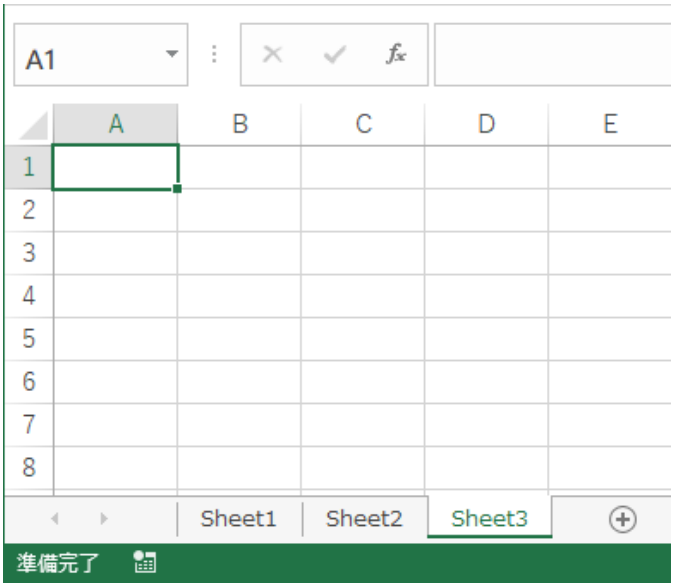
```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
sheet1.Visible = False
```

また「True」と「False」以外にも設定できる値として以下が用意されています。

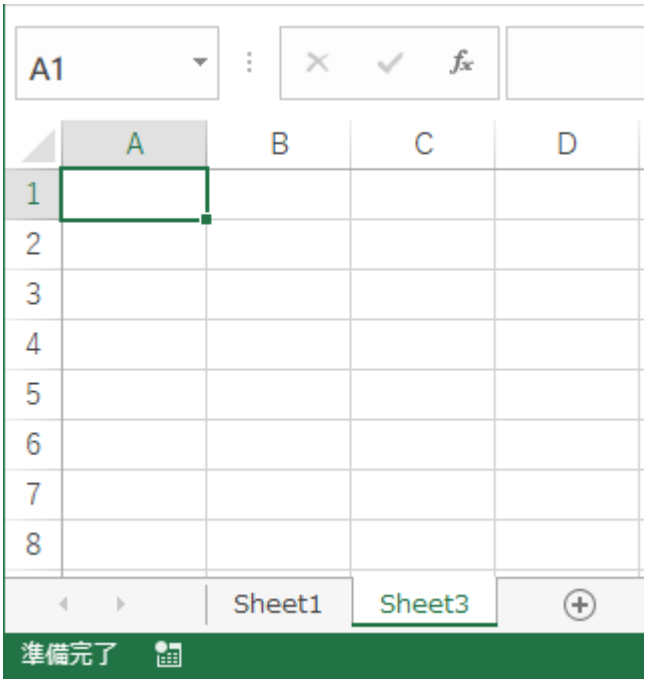
定数	シートの状態
xlSheetHidden	非表示(手動で再表示可能)
xlSheetVeryHidden	非表示(手動で再表示不可)
xlSheetVisible	表示

「xlSheetHidden」を指定した場合には「False」を指定した場合と同様ですし、「xlSheetVisible」を指定した場合には「True」を指定した場合と同様です。

右記のような Excel ファイルを用意します。
シート名「Sheet2」のシートを非表示にしてみます。

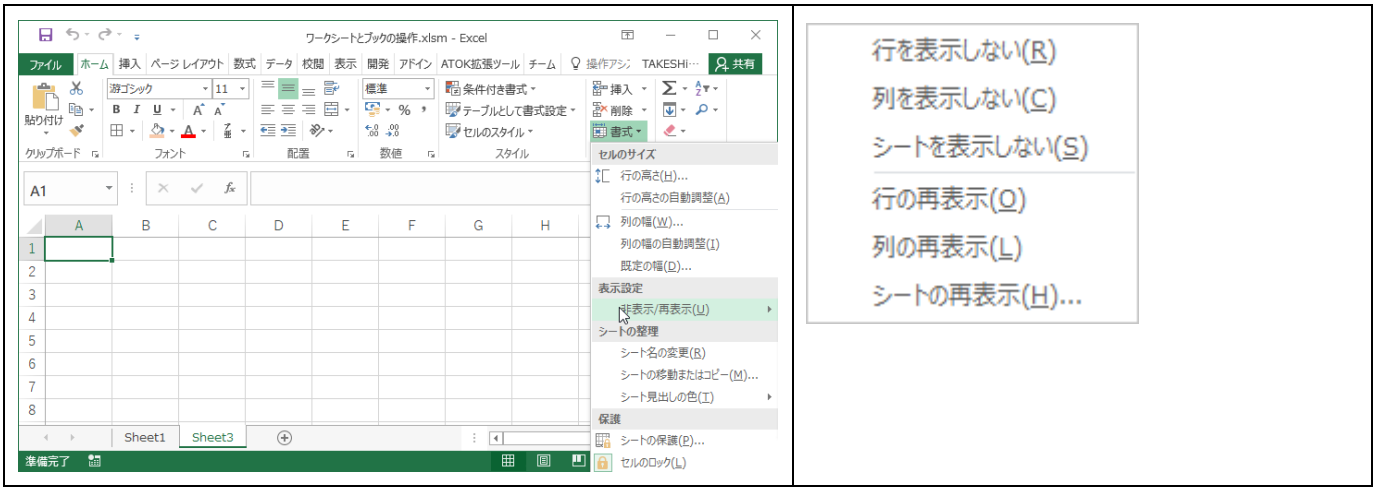


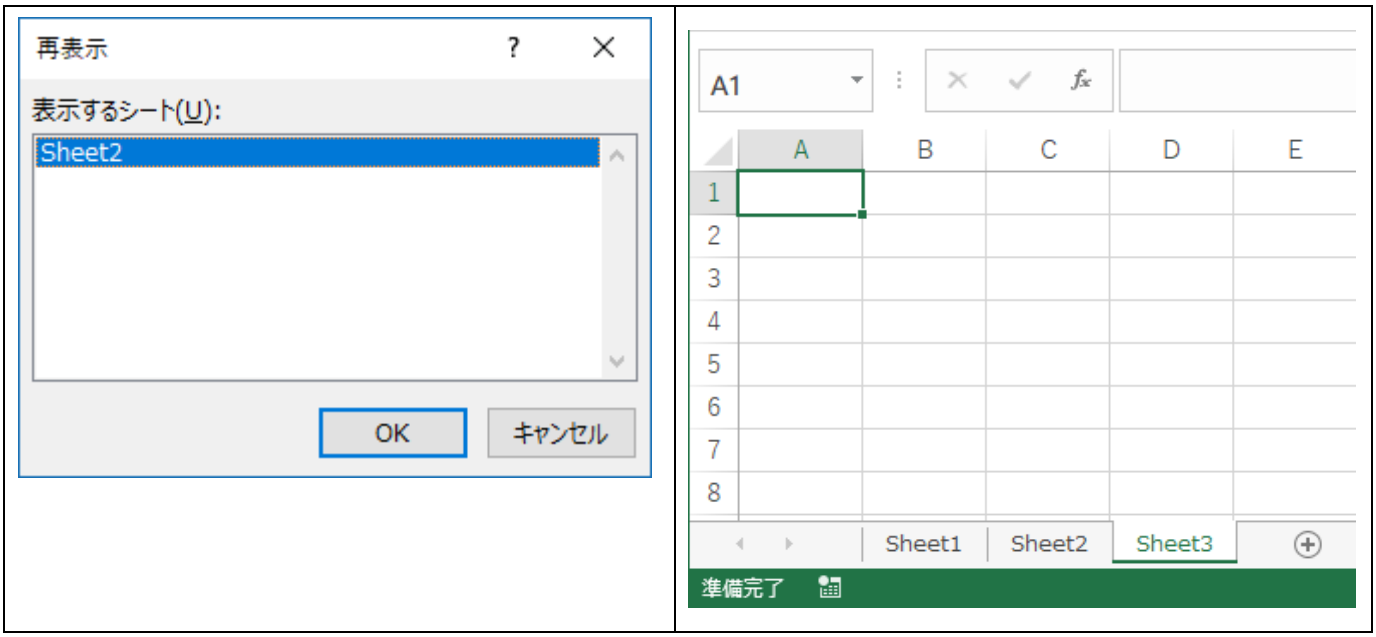

```
Sub TEST12()
    Worksheets("Sheet2").Visible = False
End Sub
```



「Sheet2」シートが非表示になりました。

今回の非表示は手動で再表示可能です。「書式」メニューの中の「シート」メニューをクリックし、さらに「再表示」をクリックします。





「Sheet2」を選択してから「OK」ボタンをクリックして下さい。

非表示なっていた「Sheet2」シートが再度表示されます。

シートの保護

ワークシートの保護を行います。Worksheet オブジェクトの「Protect」メソッドを使います。

```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
sheet1.Protect Password:="pass"
```

保護の解除の際にパスワード入力が必要とするには「Password」引数に保護解除のためのパスワードを設定します。

また保護の対象を細かく設定することが可能です。設定可能な項目と省略した場合のデフォルト値は次の通りです。

定数	対象	デフォルト値
DrawingObjects	描画オブジェクトを保護させるには、True を指定します	False
Contents	オブジェクトの内容を保護させるには、True を指定します。対象はワークシートの場合はロックされているセルです	True
Scenarios	シナリオを保護するには、True を指定します	True
UserInterfaceOnly	True を指定すると、画面上からの変更は保護されますが、マクロからの変更は保護されません。この引数を省略すると、マクロからも、画面上も変更することができなくなります。	False
AllowFormattingCells	True を指定すると、ユーザーは保護されたワークシートのセルを書式化することができます	False
AllowFormattingColumns	True を指定すると、ユーザーは保護されたワークシートの列を書式化することができます	False
AllowFormattingRows	True を指定すると、ユーザーは保護されたワークシートの行を書式化することができます	False
AllowInsertingColumns	True を指定すると、ユーザーは保護されたワークシートに列を挿入することができます	False
AllowInsertingRows	True を指定すると、ユーザーは保護されたワークシートに行を挿入することができます	False
AllowInsertingHyperlinks	True を指定すると、ユーザーは保護されたワークシートにハイパーリンクを挿入することができます	False
AllowDeletingColumns	True を指定すると、ユーザーは保護されたワークシートの列を削除することができ、削除される列のセルはすべてロック解除されます	False
AllowDeletingRows	True を指定すると、ユーザーは保護されたワークシートの行を削除することができ、削除される行のセルはすべてロック解除されます	False

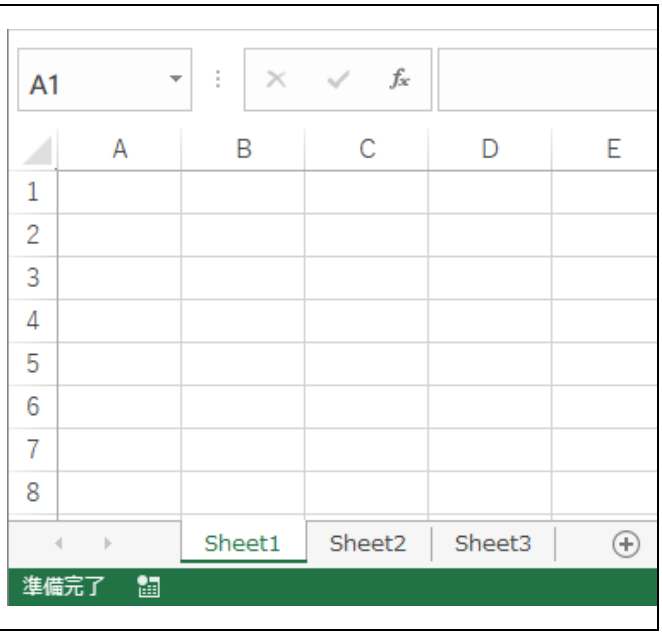
AllowSorting	True を指定すると、ユーザーは保護されたワークシートで並べ替えを行うことができます。並べ替え範囲内のセルは、ロックと保護が解除されている必要があります	False
AllowFiltering	True を指定すると、ユーザーは保護されたワークシートにフィルタを設定することができます。ユーザーは、フィルタ条件を変更できますが、オート フィルタの有効と無効を切り替えることはできません	False
AllowUsingPivotTables	True を指定すると、ユーザーは保護されたワークシートでピボットテーブル レポートを使用することができます	False

基本は全ての操作を保護する内容になっていますので、許可したい操作に関して「Allow....」引数に「True」を設定して許可して下さい。

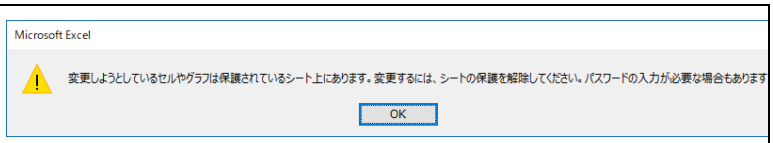
セルの保護については、ロックされたセルが対象となります。セルはデフォルトでは全てロックされているので、特に設定変更をしていなければ全てのセルが保護の対象となります。

「UserInterfaceOnly」引数を「True」に設定した場合、シートが保護されていてもマクロからは編集が可能となります。ただし、対象のセルが一度閉じられてから再度開いた場合には、マクロからも編集は不可となります。

```
Sub TEST13()
    Dim sheet1 As Worksheet
    Set sheet1 = Worksheets("Sheet1")
    sheet1.Protect Password:="pass", _
        AllowFormattingCells:=True
End Sub
```

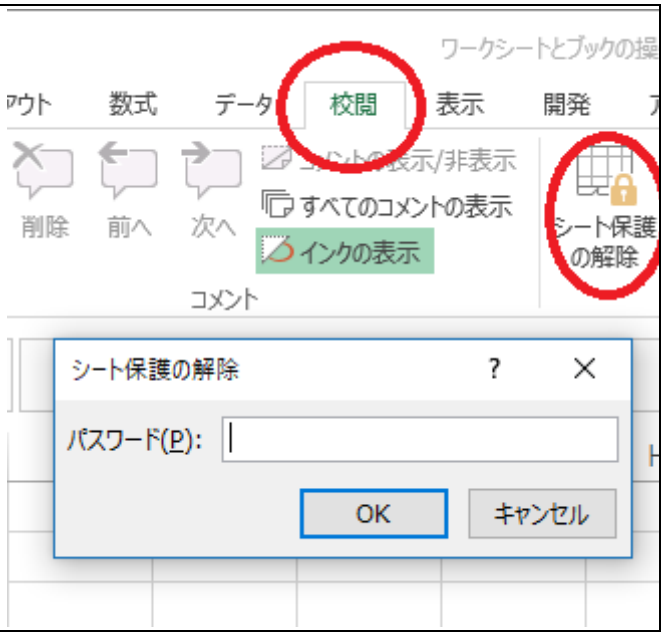


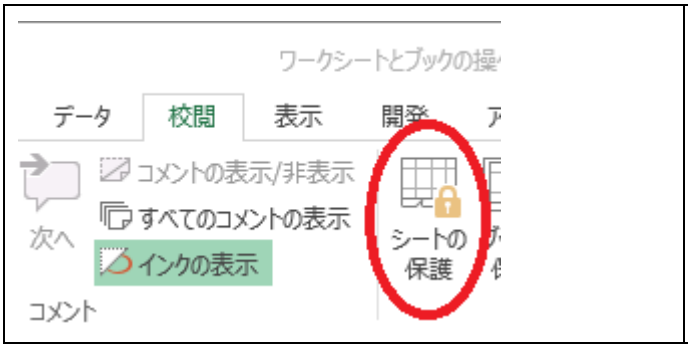
シートを保護しただけでは見た目上は変わりありませんがセルを編集しようとすると次のような警告ウィンドウが表示されます。

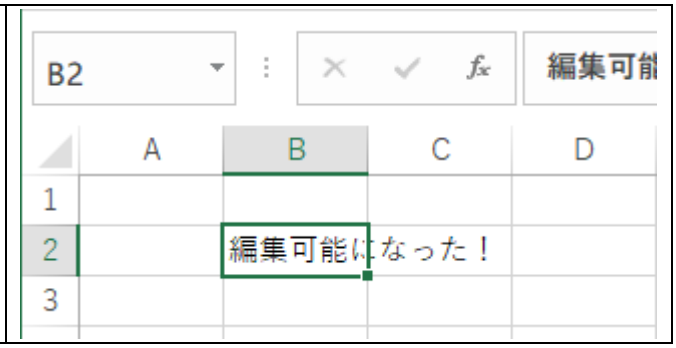


また今回は「AllowFormattingCells」引数に「True」を設定して、セルの書式設定だけは可能なように設定してあります。その為、「書式」メニューの中の「セル」メニューが有効になっています。

念のため Excel の画面上からシートの保護を解除してみましょう。まず「校閲」タブうをクリックし、さらに「シート保護の解除」をクリックします。
今回はパスワード付きでシートの保護を行っていますので、パスワード入力ウィンドウが表示されます。シートを保護する時に設定したパスワードを入力します。今回は「pass」です。入力したら「OK」ボタンをクリックします。







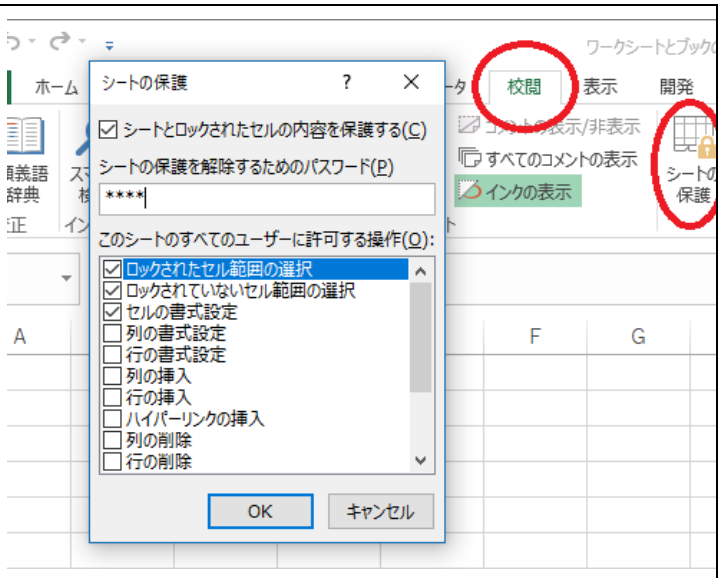
シートの保護を解除

ワークシートの保護を解除します。Worksheet オブジェクトの「Unprotect」メソッドを使います。

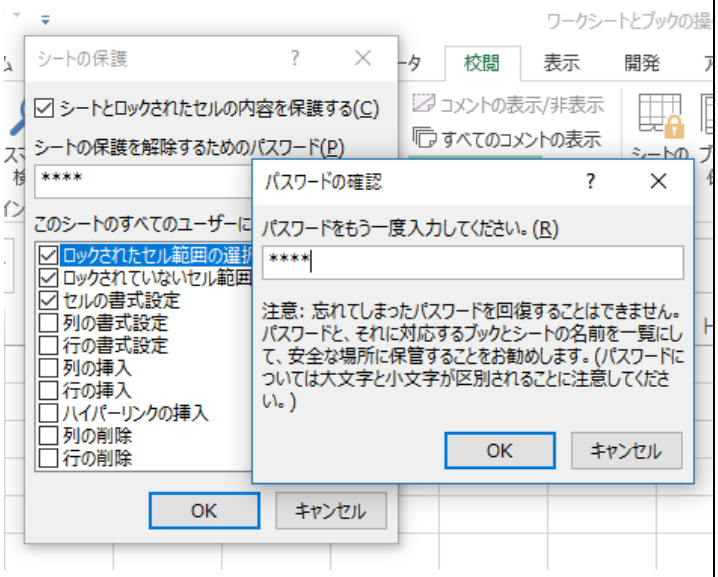
```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
sheet1.Unprotect Password:="pass"
```

保護の設定をする時にパスワードが設定されている場合には「Password」引数に保護解除のためのパスワードを設定します。

まず Excel ファイルのシートを保護しておきます。「校閲」タブをクリックし、さらに「シートの保護」をクリックします。今回はパスワードを設定しておきます。テキストボックス内に今回は「pass」と入力しておきます。

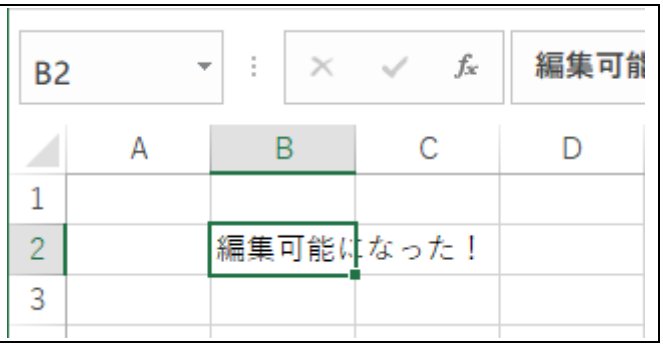


パスワードの再入力を求められますので、再度は「pass」と入力しておきます。



これでワークシートが保護された状態になりました。ではプログラム上から保護を解除してみます。

```
Sub TEST14()
    Dim sheet1 As Worksheet
    Set sheet1 = Worksheets("Sheet1")
    sheet1.Unprotect Password:="pass"
End Sub
```



見た目上は変わりありませんがセルの保護が解除されており、セルが編集できる状態となっています。

```
Sub TEST15()
    Dim sheet1 As Worksheet
    Set sheet1 = Worksheets("Sheet1")
    sheet1.Unprotect
End Sub
```

パスワード付きで保護されたシートを、パスワードを指定せずに保護解除しようとした場合、パスワードが必要なのに「Unprotect」メソッドの引数でパスワードを指定していないため、パスワード入力ウィンドウが表示されます。パスワードをここで入力すればシートの保護は解除できます。

シート名の変更

ワークシートの名前を変更します。Worksheet オブジェクトの「Name」プロパティで設定します。

```
Dim sheet1 As Worksheet
Set sheet1 = Worksheets(1)
sheet1.Name = "新しいシート名"
```

シート名は 31 文字以内で設定します。次の文字は使えませんので注意して下さい。コロン(:)、円(¥)、スラッシュ(/)、疑問符(?)、アスタリスク(*)、左角括弧(「)、右角括弧(」)。これらは半角全角に関わらず使用できません。また空白のシート名も設定できません。

```
Sub TEST16()
    Dim sheet1 As Worksheet
    Set sheet1 = Worksheets("Sheet1")
    sheet1.name = "新しいシート名"
End Sub
```

ブックの参照

ブックの中に含まれるワークシートを選択したりアクティブにする方法について確認します。

Workbook オブジェクトの取得

ブックを表すオブジェクトは Workbook オブジェクトです。ブックに対して何らかの作業を行うには、ブックは開いておかなければなりません。

開いている全てのブックは Workbooks コレクションに含まれています。個々のブックを表す Workbook オブジェクトは Application オブジェクトの「Workbooks」プロパティを使います。

```
Dim book1 As Workbook
Set book1 = Application.Workbooks(インデックス番号)
```

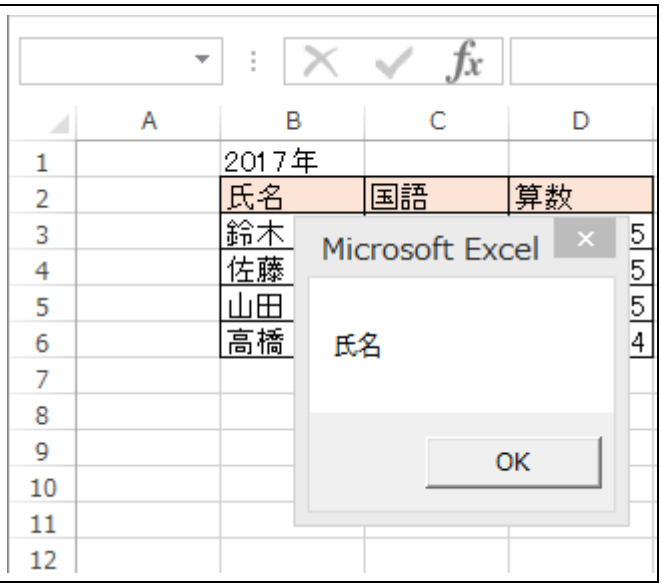
Application オブジェクトを省略した場合にも、デフォルトの値として Application オブジェクトが設定されているので記述してもしなくても構いません。

取得したいブックを指定するにはインデックス番号で指定するかブック名で指定します。インデックス番号はブックが開かれた順番に 1 から開始されるようです。(ただし、個人用マクロブックなども開いたブックとしてカウントされる場合もあるようですので、インデックス番号ではなくブック名を使ってブックを特定したほうがいいかもしれません)。

```
Dim book1 As Workbook
Set book1 = Workbooks("VBAsample.xls")
```

```
Sub TEST01()
    Dim book1 As Workbook
    Dim str As String
    'Workbooks.Open "C:¥Works¥2017 成績表.xlsm"
    Workbooks.Open "2017 成績表.xlsm"
    Set book1 = Workbooks("2017 成績表.xlsm")
    Str= _

book1.Worksheets("Sheet1").Range("B2").Value
    MsgBox str
End Sub
```



今回は、「2017 成績表.xlsm」というブックを開いた後でそのブックに関する Workbook オブジェクトを取得しています。そして Workbook オブジェクトに含まれるシートのさらにその中に含まれるセルの値を取得して表示しています。

ブックをアクティブにする

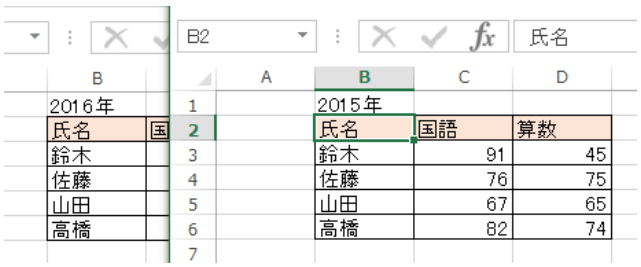
開かれているブックの中で指定のブックをアクティブにするには Workbook オブジェクトの「Activate」メソッドを使います。

```
Dim book1 As Workbook
Set book1 = Application.Workbooks(1)
book1.Activate
```

まとめて次のように記述しても構いません。

```
Workbooks(1).Activate
```

```
Sub TEST02()
    Workbooks.Open "2015 成績表.xlsm"
    Workbooks.Open "2016 成績表.xlsm"
    Workbooks("2015 成績表.xlsm").Activate
End Sub
```



今回は「2015 成績表.xlsm」と「2016 成績表.xlsm」という 2 つのブックを開いた後で、「2015 成績表.xlsm」ブックの方をアクティブに設定しています。

アクティブブックのオブジェクトを取得

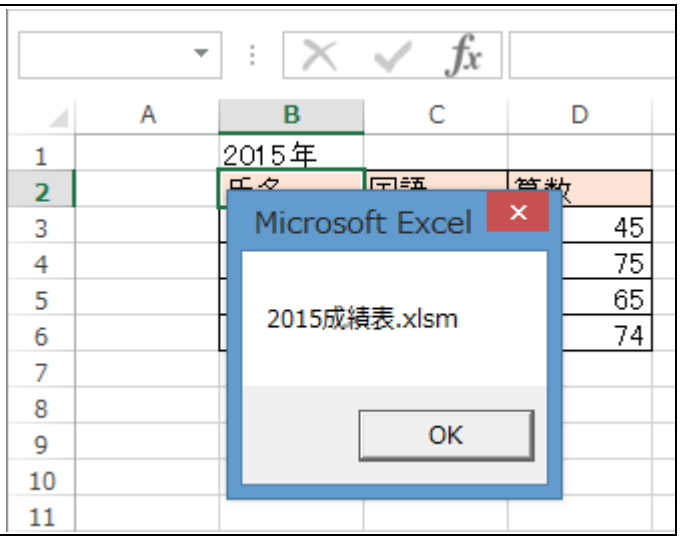
ブックのオブジェクトを取得するには、インデックス番号かブック名を指定して Workbook オブジェクトを取得していましたが、現在アクティブになっているブックの Workbook オブジェクトを取得するためのプロパティが用意されています。

現在アクティブになっている Workbook オブジェクトを取得するには、Application オブジェクトの「ActiveWorkbook」プロパティから取得します。

```
Dim book1 As Workbook
Workbooks(1).Activate
Set book1 = Application.ActiveWorkbook
```

Application オブジェクトを省略した場合でも、デフォルトの設定値が Application オブジェクトですので記述してもしなくても構いません。

```
Sub TEST03()
    Workbooks.Open "2015 成績表.xlsm"
    Workbooks.Open "2016 成績表.xlsm"
    Workbooks("2015 成績表.xlsm").Activate
    MsgBox ActiveWorkbook.name
End Sub
```



今回は「2015 成績表.xlsm」と「2016 成績表.xlsm」という 2 つのブックを開いた後で、「2015 成績表.xlsm」ブックの方をアクティブに設定しています。そしてアクティブブックのブック名を取得してメッセージボックスで表示しています。

ブックの作成と保存

ブックの新規作成や保存などの方法について確認します。

新規ブックの作成

新規にブックを作成します。新規作成の場合は、Workbooks コレクションに追加するという形になりますので、Workbooks コレクションの「Add」メソッドを使います。

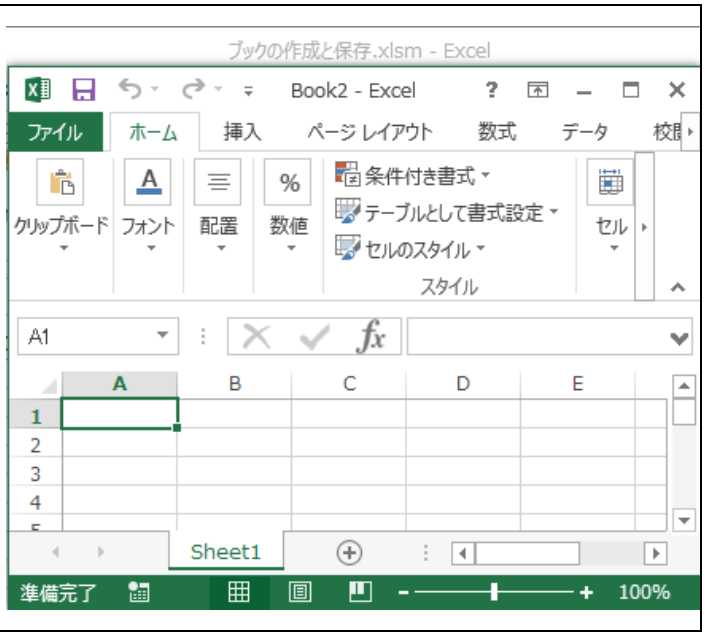
```
Workbooks.Add
```

引数として「Template」を指定可能です。この引数には作成するブックにデフォルトで含まれるシートの種類を指定します。

定数	シートの種類
xlWBATWorksheet	ワークシート
xlWBATChart	グラフシート
xlWBATExcel4MacroSheet	Excel4 マクロシート
xlWBATExcel4IntlMacroSheet	Excel4 インターナショナルマクロシート

省略した場合はワークシートがデフォルトで作成されてブックに格納されます。

```
Sub TEST01()
    Workbooks.Add
End Sub
```



新しいブックが作成されました。

ブックを開く

既に作成されているブックを開くには Workbooks コレクションの「Open」メソッドを使います。

```
Workbooks.Open Filename:="C:¥ブック名.xls"
```

「Filename」引数で開きたいブックを指定します。パスを指定しない場合にはカレントディレクトリにあるブックを検索します。

他、いくつかの引数が指定可能です。使いそうなものだけ抜粋して紹介します。

「UpdateLinks」引数にはファイル内のリンクの更新方法を指定します。リンクが設定されているファイルを開くとき、この引数を省略すると、リンク更新のダイアログ ボックスが表示されます。次のいずれかの値を指定します。

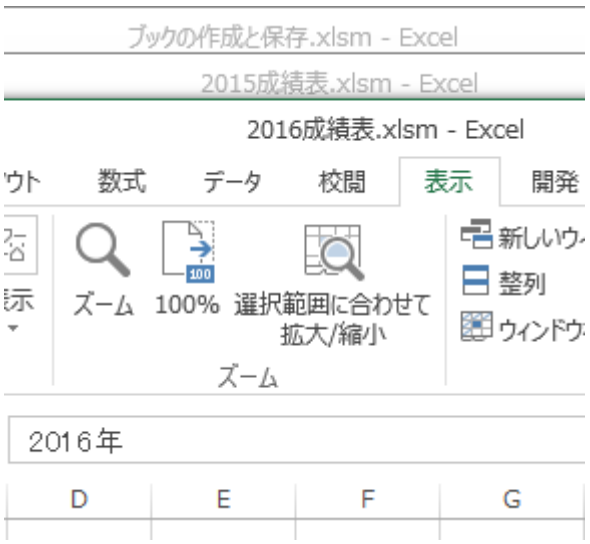
値	リンク方法
0	外部参照とリモート参照は共に更新されません。
1	外部参照は更新され、リモート参照は更新されません。
2	リモート参照は更新され、外部参照は更新されません。
3	外部参照とリモート参照は共に更新されます。

「ReadOnly」引数は、ブックを読み取り専用モードで開くには、True を指定します。

「Password」引数は、読み取りパスワードが設定されたブックを開くのに必要なパスワードを指定します。パスワードが必要なときにこの引数を省略すると、パスワードの入力を促すダイアログ ボックスが表示されます。

「WriteResPassword 」引数は、書き込みパスワードが設定されたブックを開くのに必要なパスワードを指定します。パスワードが必要なときにこの引数を省略すると、パスワードの入力を促すダイアログ ボックスが表示されます。

```
Sub TEST02()
    Workbooks.Open Filename:= _
            "2015 成績表.xlsm"
    Workbooks.Open Filename:= _
            "2016 成績表.xlsm"
End Sub
```



2 枚のブックを開く事が出来ました。

ファイル選択ダイアログの表示

ブックを開く時に「ファイルを開く」ダイアログを表示して、ユーザーに開くブックを選択してもらうことができます。「ファイルを開く」ダイアログを表示するには Application オブジェクトの「GetOpenFilename」メソッドを使います。

```
Dim fname As String
fname = Application.GetOpenFilename
```

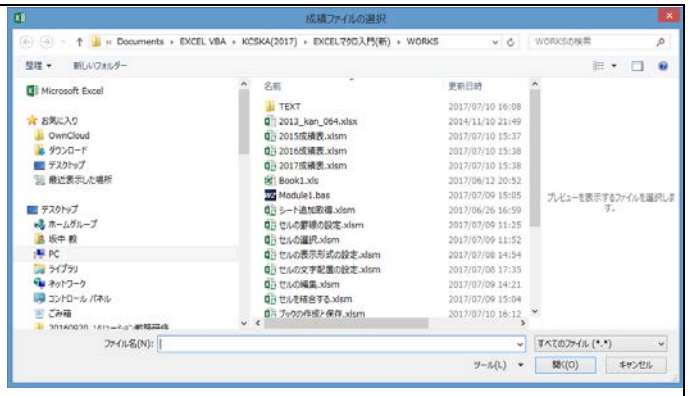
メソッドを実行した結果、選択されたファイル名を戻り値として返してきます。ただし、ダイアログで「キャンセル」ボタンをクリックした場合にはファイル名ではなく「False」が帰ってきます。このメソッドではファイル名が取得できるだけなので、ブックを開く場合には取得したファイル名を使って別途ブックを開く処理を行って下さい。

```
Dim fname As String
fname = Application.GetOpenFilename
If fname <> "False" Then
    Workbooks.Open FileName:=fname
End If
```

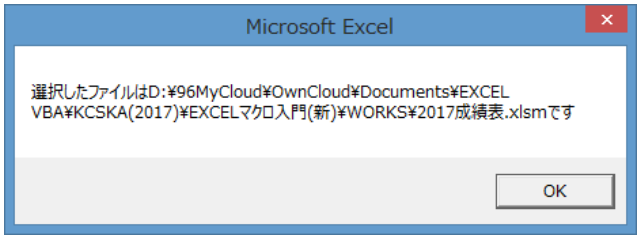
またデフォルトではダイアログのタイトルは「ファイルを開く」ですが、「Title」引数に値を設定することで、タイトルを任意の文字列に変更することができます。

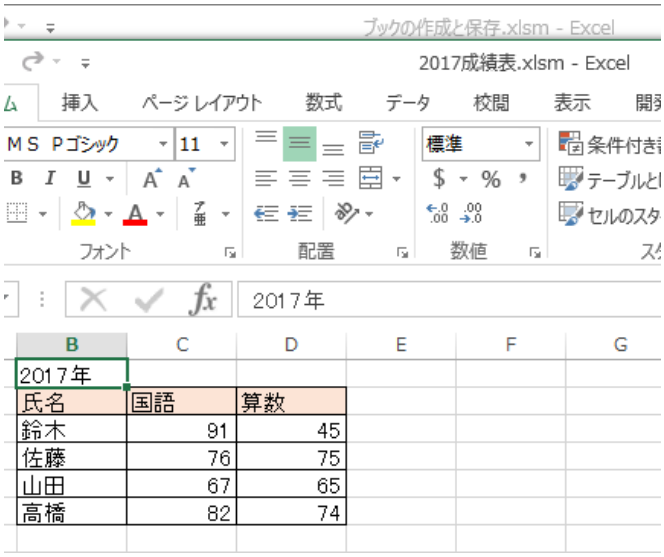
```
Dim fname As String
fname = Application.GetOpenFilename(Title:="売上ファイルの選択")
```

```
Sub TEST03()
   Dim fname As String
   fname = Application.GetOpenFilename( _
           Title:="成績ファイルの選択")
   If fname <> "False" Then
    MsgBox "選択したファイルは" & fname & "です
"
    Workbooks.Open filename:=fname
   End If
End Sub
```



ファイルを選択するためのダイアログが表示されます。「2017 成績表 xlsm」を選択してから「開く」ボタンをクリックします。





選択されたファイルが表示されます。ファイル名はフルパスで取得することが分かります。
選択されたファイル名のブックを開くことができました。

ファイル選択時のフィルタ設定

ファイル選択ダイアログを表示する時に、ダイアログ内に表示されるファイルをフィルタして表示することが出来ます。デフォルトでは全てのファイルが表示されるようになっていますが、拡張子が「.xls」のファイルだけを表示するようにしたり、「.bas」のファイルだけを表示したりするように設定できます。
フィルタを設定するには「GetOpenFilename」メソッドに「FileFilter」引数に設定します。

```
Dim fname As String
fname = Application.GetOpenFilename( _
      FileFilter:="Excel ファイル, *.xlsx")
```

フィルタは次の規則に従って記述します。

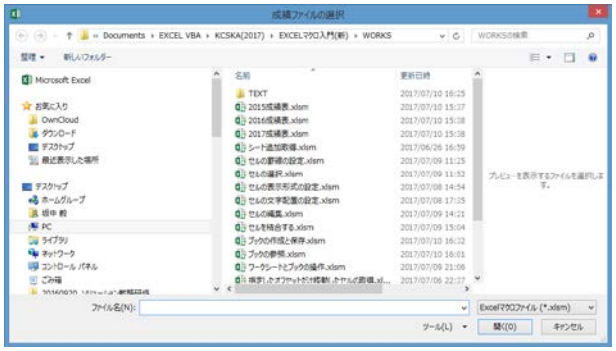
```
フィルタ文字列, ワイルドカード
```

フィルタ文字列は、表示される拡張子を持つファイルが何のファイルなのか分かりやすいように、任意の文字列を指定します。そしてファイル一覧として表示したいファイルを表すワイルドカードを指定します。例えば拡張子が「.xlsm」のファイルだけ表示したい場合には「*.xlsm」と記述します。

"Excel ファイル, *.xlsm"

上記のように設定した場合、ダイアログには次のように表示されます。

```
Sub TEST031()
    Dim fname As String
    fname = _
      Application.GetOpenFilename( _
        FileFilter:="Excelマクロファイル,*.xlsm", _
        Title:="成績ファイルの選択")
End Sub
```



ダイアログの下部の「ファイルの種類」の箇所に設定したフィルタの内容が表示されます。そして、表示されるファイルの一覧はワイルドカードで指定されたフィルタに適合するファイルだけが表示されています。

複数のフィルタを設定する

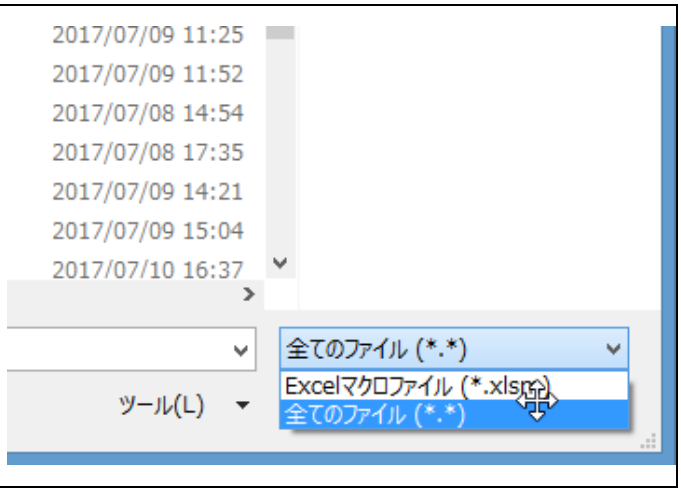
設定できるフィルタは 1 つだけではなく、複数のフィルタを設定し、ユーザーに選択してもらうことができます。複数のフィルタを設定するには、カンマ(,)で区切って並べて記述します。

フィルタ文字列 1, ワイルドカード 1, フィルタ文字列 2, ワイルドカード 2, ...

具体的には次のような記述となります。

"Excel ファイル, *.xls, 全てのファイル, *.*"

```
Sub TEST032()
    Dim fname As String
    fname = _
      Application.GetOpenFilename( _
        FileFilter:="Excel マクロファイル,*.xlsm,
全てのファイル,*.*", _
        Title:="成績ファイルの選択")
End Sub
```



このように記述した場合、利用者が「Excel マクロファイル」を選択すれば拡張子が「.xlsm」のファイルだけが表示され、「全てのファイル」を選択すれば全てのファイルが表示されるようになります。

1 つのフィルタに複数の拡張子を設定する

1 つのフィルタで複数の拡張子を対応させることもできます。例えば「Excel ファイル」には拡張子として「.xlsx」の他に「.bas」のファイルも表示させたい場合には、拡張子をセミコロン(;)で区切って指定します。

フィルタ文字列, ワイルドカード 1;ワイルドカード 2

具体的には次のように記述します。

"全ての Excel ファイル, *.xlsx;*.bas"

このフィルタを使用した場合には拡張子が「.xlsx」と「.bas」のファイルだけが表示されます。

最初に表示されるフィルタ

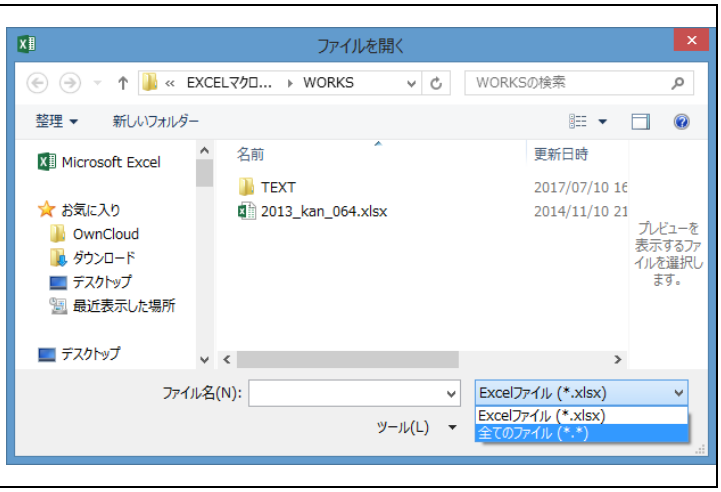
複数のフィルタを記述した場合、ファイル選択ダイアログが開いた時には一番目に記述されたフィルタがまず適用されて表示されています。このデフォルトで選択されるフィルタを指定することが可能です。

デフォルトで指定されるフィルタを指定するには「GetOpenFilename」メソッドに「FilterIndex」引数に設定します。

```
Dim fname As String
fname = Application.GetOpenFilename( _
    FileFilter:="Excel ファイル,*.xlsx,全てのファイル,*.*", _
    FilterIndex:=2)
```

指定する値はフィルタを記述した順に 1,2,...となります。よって 2 番目のフィルタをデフォルトで適用するには「FilterIndex」引数に「2」を指定します。

```
Sub TEST04()
    Dim fname As String
    fname = Application.GetOpenFilename( _
        FileFilter:="Excel ファイル,*.xlsx,全てのファイル,*.*", _
        FilterIndex:=1)
End Sub
```



上記のように設定したフィルタに該当するファイルだけが表示されています。(今回はファイル選択しても何も行いません)。

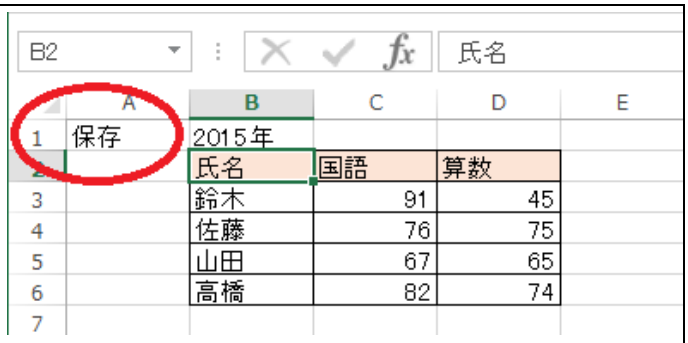
ブックを上書き保存

内容が変更されたブックを上書き保存します。Workbook オブジェクトに対して「Save」メソッドを使います。

```
Dim book1 As Workbook
Set book1 = Workbooks("VBAsample.xls")
book1.Save
```

Save メソッドが実行されると、対象となるブックが上書き保存されます。

```
Sub TEST05()
    Dim book1 As Workbook
    Workbooks.Open filename:="2015 成績表.xlsm"
    ActiveSheet.Range("A1").Value = "保存"
    Set book1 = ActiveWorkbook
    book1.Save
    book1.Close
End Sub
```



上記マクロを実行すると、「2015 成績表.xlsm」ブックを開いた後でセル A1 に文字をセットした後で上書き保

存します。いったんマクロ内で「2015 成績表.xlsm」ブックは閉じますので、改めて Excel を開いて見てみます。

ブックを名前を付けて保存

別の名前を付けてブックを保存します。Workbook オブジェクトに対して「SaveAs」メソッドを使います。

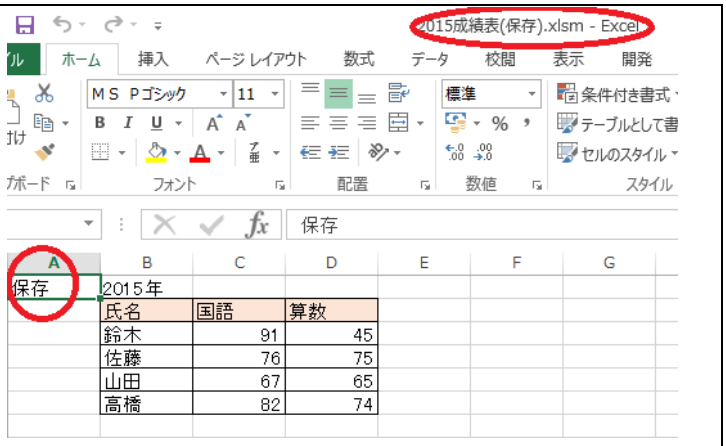
```
Dim book1 As Workbook
Set book1 = Workbooks("VBAsample.xls")
book1.SaveAs Filename:="C:¥excelsample¥OtherFile.xls"
```

保存する名前は「Filename」引数で指定します。パスを指定しない場合にはカレントディレクトリに保存されます。

保存しようとした名前と同じ名前のブックが開いていると、その時点でエラーとなります。また開いていなくても同じファイルが既に存在している場合には上書き保存するかどうかの確認ダイアログが表示されます。ダイアログで「いいえ」又は「キャンセル」を選択するとエラーとなります。

「SaveAs」メソッドには他にも多くの引数が用意されていますので次のページ以降で順次見ていきます。

```
Sub TEST06()
    Dim book1 As Workbook
    Workbooks.Open filename:="2015 成績表.xlsm"
    ActiveSheet.Range("A1").Value = "保存"
    Set book1 = ActiveWorkbook
    book1.SaveAs filename:="2015 成績表(保存).xlsm"
    book1.Close
End Sub
```



変更した内容が保存されていることが確認できます。(「2015 成績表(保存).xlsm」というブックが存在しない状態で試して下さい)。

読み取りパスワードを付けて保存

名前を付けて保存する際に、読み取りパスワードを設定します。読み取りパスワードを設定すると、ファイルを開こうとした時にパスワードの入力が求められます。パスワード入力に失敗した場合は、ファイルを開く事ができません。

設定するには「SaveAs」メソッドの引数に「Password」を設定します。

```
Dim book1 As Workbook
Set book1 = Workbooks("VBAsample.xls")
book1.SaveAs Filename:="C:¥excelsample¥OtherFile.xls", _
    Password:="pass"
```

パスワードには 15 文字以内の文字列を指定します。

一度パスワードを設定したファイルに対してパスワードを解除するには空の文字列を指定して保存します。

```
Dim book1 As Workbook
Set book1 = Workbooks("VBAsample.xls")
book1.SaveAs Filename:="C:¥excelsample¥OtherFile.xls", _
```

```
    Password:=""
```

```
Sub TEST07()
    Dim book1 As Workbook
    Workbooks.Open filename:="2015 成績表.xlsm"
    ActiveSheet.Range("A1").Value = "保存(パス付)"
    Set book1 = ActiveWorkbook
    book1.SaveAs filename:="2015 成績表(保存).xlsm", _
        Password:="pass"
    book1.Close
    Workbooks.Open filename:="2015 成績表(保存).xlsm"
End Sub
```

上記マクロを実行すると、「2015 成績表.xlsm」と言うブックを開いた後でセル A1 に文字をセットした後、「2015 成績表(保存).xlsm」という別の名前でブックを保存します。その際に読み取りパスワードが設定されています。改めて「2015 成績表(保存).xlsm」というブックを開こうとすると、パスワード要求が表示されます。
パスワードを入力できれば、ブックを開くことができます。
パスワードの入力に失敗したり、「キャンセル」ボタンをクリックするとエラーとなります。

書き込みパスワードを付けて保存

名前を付けて保存する際に、書き込みスワードを設定します。書き込みパスワードを設定すると、ファイルを開こうとした時にパスワードの入力が求められます。パスワード入力に失敗した場合は、ファイルを開くことは出来ますが読み取り専用となります。
設定するには「SaveAs」メソッドの引数に「WriteResPassword」を設定します。

```
Dim book1 As Workbook
Set book1 = Workbooks("VBAsample.xls")
book1.SaveAs Filename:="C:¥excelsample¥OtherFile.xls", _
    WriteResPassword:="pass"
```

一度パスワードを設定したファイルに対してパスワードを解除するには空の文字列を指定して保存します。

```
Dim book1 As Workbook
Set book1 = Workbooks("VBAsample.xls")
book1.SaveAs Filename:="C:¥excelsample¥OtherFile.xls", _
    WriteResPassword:=""
```

```
Sub TEST08()
    Dim book1 As Workbook
    Workbooks.Open filename:="2015 成績表.xlsm"
    ActiveSheet.Range("A1").Value = "保存(別名パス付)"
    Set book1 = ActiveWorkbook
    book1.SaveAs filename:="2015 成績表(保存).xlsm", _
```

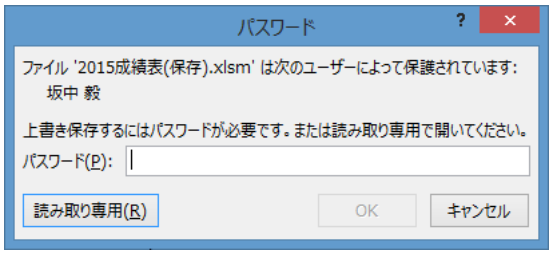
```
        WriteResPassword:="pass"
    book1.Close
    Workbooks.Open filename:="2015 成績表(保存).xlsm"
End Sub
```

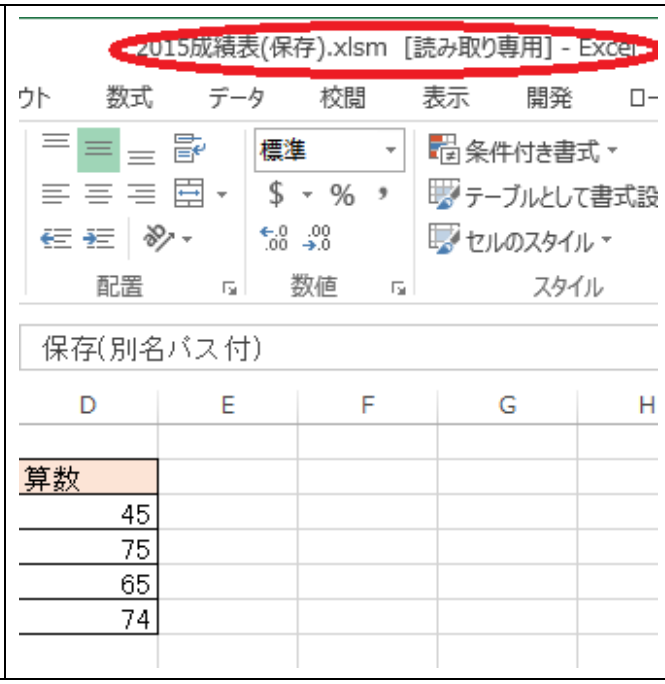
上記マクロを実行すると、「2015 成績表.xlsm」と言うブックを開いた後でセル A1 に文字をセットした後、「2015 成績表(保存).xlsm」という別の名前でブックを保存します。その際に読み取りパスワードが設定されています。改めて「2015 成績表(保存).xlsm」というブックを開こうとすると、パスワード要求が表示されます。

パスワードを入力できれば、ブックを開くことができます。

パスワードの入力に失敗したり、「キャンセル」ボタンをクリックするとエラーとなります。

パスワード入力時に「読み取り専用」ボタンをクリックすると、読み取り専用でブックが開きます。





また、読み取りパスワードと書き込みパスワードを両方設定した場合には、まず読み取りパスワードの入力ボックスが表示され、次に書き込みパスワードの入力ボックスが表示されます。

保存するフォーマットの指定

ファイルを保存する際に、保存するフォーマットを指定します。例えばテキスト形式や CSV 形式での保存を行う場合に使います。

設定するには「SaveAs」メソッドの引数に「FileFormat」を設定します。

```
Dim book1 As Workbook
Set book1 = Workbooks("VBAsample.xls")
book1.SaveAs Filename:="C:¥excelsample¥OtherFile.xls", _
    FileFormat:="xlCSV"
```

指定できるファイルフォーマットは以下の通りです。

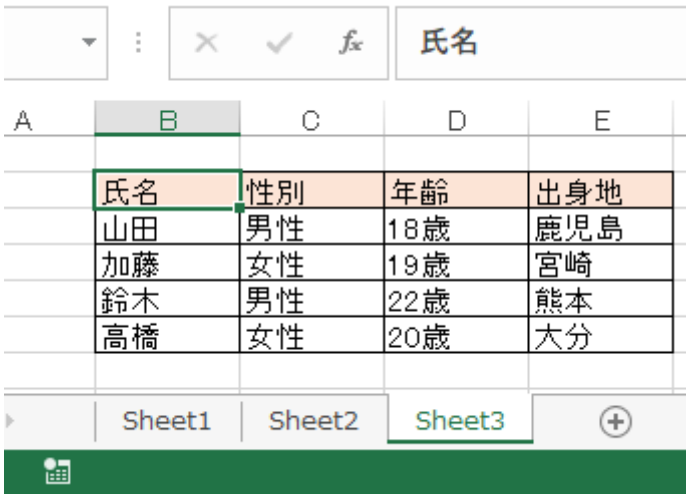
名前	拡張子	説明	値
xlAddIn	.xls	Excel 97-2003 アドイン	18

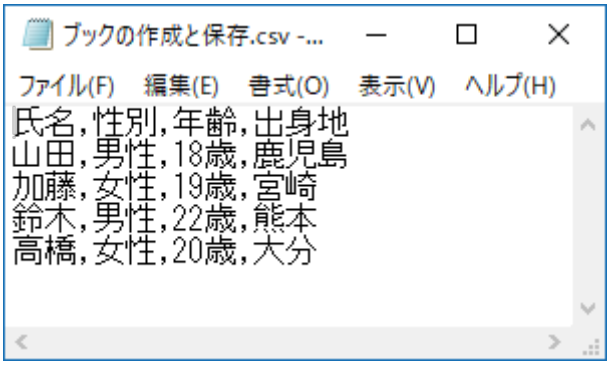
名前	拡張子	説明	値
xlAddIn8	.xls	Excel 97-2003 アドイン	18
xlCSV	.csv	CSV	6
xlCurrentPlatformText	.txt	テキストファイル	-4158
xlExcel8	.xls	Excel 97-2003 ブック （Excel2007 以降）	56
xlHtml	.htm	HTML 形式	44
xlOpenDocumentSpreadsheet	.ods	OpenDocument スプレッドシート	60
xlDIF	.dif	dif （Data Interchange format）	9
xlOpenXMLAddIn	.xlam	Excel アドイン	55
xlOpenXMLTemplate	.xltx	Excel テンプレート	53
xlOpenXMLTemplateMacroEnabled	.xltm	Excel マクロ有効テンプレート	51
xlOpenXMLWorkbook	**.xlsx**	Excel ブック	51
xlOpenXMLWorkbookMacroEnabled	.xlsm	Excel マクロ有効ブック	52
xlSYLK	.slk	SYLK（シンボリック リンク）形式 プリンタの問題を取り除くために使用 （破損した要素を除外できることがある）	2
xlTemplate	.xlt	Excel 97-2003 テンプレート	17
xlTemplate8	.xlt	Excel 97-2003 テンプレート	17
xlTextPrinter	.prn	PRN ファイル プリンタに渡すデータをファイル化したもの （印刷時の画面の「ファイルへ出力」と同じ）	36
xlUnicodeText	.txt	Unicode テキスト	42
xlWebArchive	.mht	Web ページのアーカイブファイル	45
xlWorkbookDefault	.xls or .xlsx （環境に依存）	通常のエクセル形式	51
xlWorkbookNormal	.xls	Excel 97-2003 ブック	-4143
xlXMLSpreadsheet	.xml	xml スプレッドシート	46

デフォルトは現在使用されている Excel のバージョンでのファイル形式が既定のファイル形式です。

```
Sub TEST09()
    Dim book1 As Workbook
    Set book1 = _
        Workbooks("ブックの作成と保存.xlsm")
    book1.SaveAs _
        filename:="ブックの作成と保存.csv", _
        FileFormat:=xlCSV
End Sub
```

左記のマクロを実行すると CSV 形式で保存された「ブックの作成と保存.csv」というファイルが作成されています。CSV 形式ですので Excel に含まれる値だけをテキストファイルとして保存しています。各値の区切りはカンマ区切りとなります。





ファイル指定ダイアログの表示

ファイルを保存する際に、ファイルの保存先を指定する「名前を付けて保存」ダイアログを表示する方法を見ていきます。「名前を付けて保存」ダイアログを表示するには Application オブジェクトの「GetSaveAsFilename」メソッドを使います。

```
Dim fname As String
fname = Application.GetSaveAsFilename
```

メソッドを実行した結果、指定したファイル名を戻り値として返してきます。ただし、ダイアログで「キャンセル」ボタンをクリックした場合にはファイル名ではなく「False」が帰ってきます。このメソッドではファイル名が取得できるだけなので、ブックを保存する場合には取得したファイル名を使って別途ブックを保存する処理を行って下さい。

```
Dim fname As String
fname = Application.GetSaveAsFilename
If fname <> "False" Then
    ActiveWorkbook.SaveAs Filename:=fname
End If
```

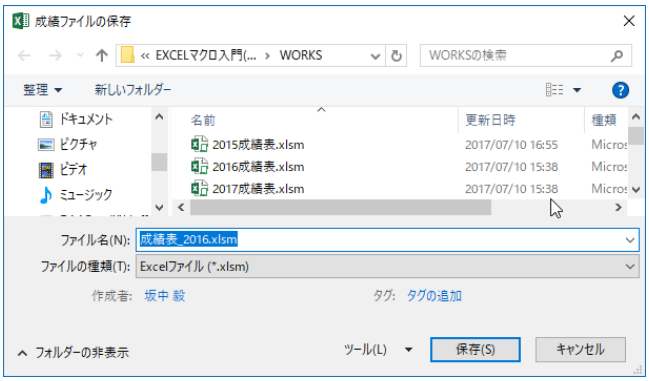
保存するファイル名としてデフォルトで表示するファイル名を指定出来ます。指定する場合には「InitialFilename」で指定します。省略した場合には保存しようとするブック名がデフォルトで表示されます。

```
Dim fname As String
fname = Application.GetSaveAsFilename( InitialFilename:="売上一覧表.xlsx")
```

またデフォルトではダイアログのタイトルは「名前を付けて保存」ですが、「Title」引数に値を設定することで、タイトルを任意の文字列に変更することができます。

```
Dim fname As String
fname = Application.GetSaveAsFilename( Title:="売上一覧の保存先")
```

```
Sub TEST10()
    Dim fname As String
    Workbooks.Open Filename:="2016 成績表.xlsm"
    ActiveSheet.Range("A1").Value = "保存"
    fname = Application.GetSaveAsFilename( _
        InitialFileName:="成績表_2016.xlsm", _
        Title:="成績ファイルの保存", _
        FileFilter:="Excel ファイル, *.xlsm")
    If fname <> "False" Then
        ActiveWorkbook.SaveAs Filename:=fname
    End If
End Sub
```



保存するファイル名を変更する場合は変更した後で「保存」ボタンをクリックします。指定した名前でブックが保存されていることを確認しておいて下さい。

ファイル指定時のフィルタ設定

ファイル名を指定するダイアログを表示する時に、ダイアログ内に表示されるファイルをフィルタして表示することが出来ます。デフォルトでは全てのファイルが表示されるようになっていますが、拡張子が「.xlsx」のファイルだけを表示するようにしたり、「.bas」のファイルだけを表示したりするように設定できます。

フィルタを設定するには「GetSaveAsFilename」メソッドに「FileFilter」引数に設定します。

```
Dim fname As String
fname = Application.GetSaveAsFilename(FileFilter:="Excel ファイル, *.xlsx")
```

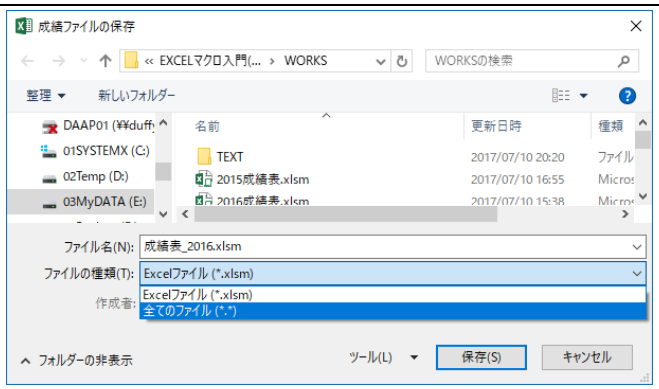
複数のフィルタを記述した場合、ファイル名指定ダイアログが開いた時には一番目に記述されたフィルタがまず適用されて表示されています。このデフォルトで選択されるフィルタを指定することが可能です。

デフォルトで指定されるフィルタを指定するには「GetSaveAsFilename」メソッドに「FilterIndex」引数に設定します。

```
Dim fname As String
fname = Application.GetSaveAsFilename(FileFilter:="Excel ファイル, *.xlsx",FilterIndex:=1)
```

指定する値はフィルタを記述した順に 1,2,...となります。よって 2 番目のフィルタをデフォルトで適用するには「FilterIndex」引数に「2」を指定します。

```
Sub TEST11()
    Dim fname As String
    Workbooks.Open Filename:="2016 成 績表.xlsm"
    ActiveSheet.Range("A1").Value = "保存"
    fname = Application.GetSaveAsFilename( _
        FileFilter:="Excel ファイル,*.xlsm,全てのファイル,*.*", _
        FilterIndex:=1, _
        InitialFileName:="成績表_2016.xlsm", _
        Title:="成績ファイルの保存")
    If fname <> "False" Then
        ActiveWorkbook.SaveAs Filename:=fname
    End If
End Sub
```



上記のように設定したフィルタに該当するファイルだけが表示されています。

最後に保存されてから変更されているか確認する

ブックが最後に保存されてから何か変更が行われているかどうかを確認する方法を見ていきます。Workbook オブジェクトの「Saved」プロパティで確認することが出来ます。

```
If ActiveWorkbook.Saved = True Then
    MsgBox "変更されていません"
Else
    MsgBox "変更された内容が保存されていません"
End if
```

「Saved」プロパティが「True」の場合には、ブックが最後に保存されてから何も変更が行われていない事になります。逆に「False」の場合には保存されていない変更分が存在していることになります。

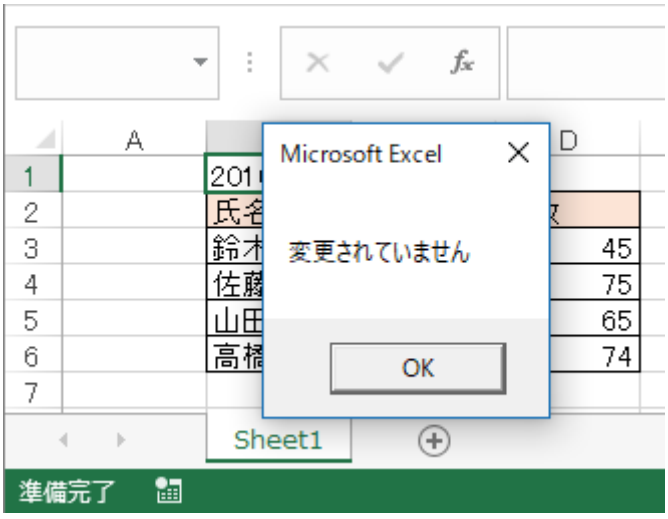
「Saved」プロパティは値を参照するだけではなく、値を設定することも可能です。

```
ActiveWorkbook.Saved = True
ActiveWorkbook.Close
```

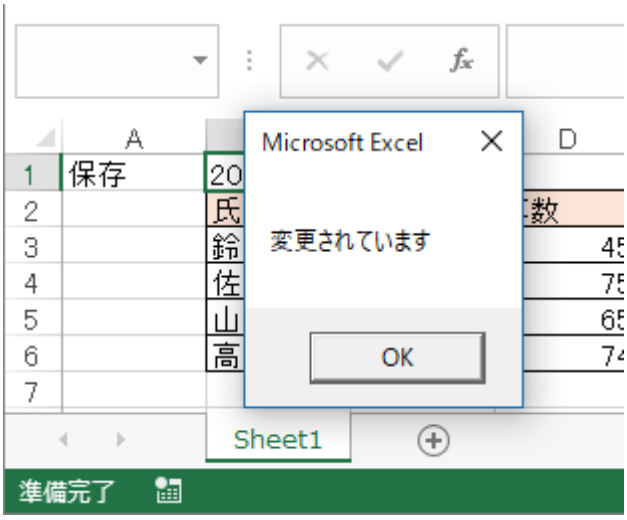
何か変更があったにも関わらず、単にブックを閉じようとすると「変更を保存しますか」という確認ダイアログが表示されますが、保存されていないにも関わらず「Saved」プロパティに「True」を設定すると最後に保存してから何も変更がなかったかのように扱われるため、確認ダイアログが表示されずにブックは閉じられるようになります。

注意する点は「Saved」プロパティを「True」に設定しても実際に保存がされるわけではありませんので注意して下さい。

```
Sub TEST12()
    Workbooks.Open filename:="2016 成績表.xlsm"
    If ActiveWorkbook.Saved = True Then
        MsgBox "変更されていません"
    Else
        MsgBox "変更されています"
    End If
    ActiveSheet.Range("A1").Value = "保存"
    If ActiveWorkbook.Saved = True Then
        MsgBox "変更されていません"
    Else
        MsgBox "変更されています"
    End If
End Sub
```



まずブックを開いた直後に変更がされたかどうかを確認しています。開いた直後ですので何も変更が行われていませんので上記のように表示されます。



次にセル A1 に値を設定しています。これで最後に保存されてから変更が加えられた状態となります。「Saved」プロパティを確認してみると上記のように変更されていることを認識していることが確認できます。

ブックを閉じる

開いているブックを閉じる方法を確認します。Workbook オブジェクトの「Close」メソッドを使います。

```
ActiveWorkbook.Close
```

閉じようとするブックが、最後に保存されてから変更が行われている場合にはファイルを保存するかどうか確認するための確認ダイアログが表示されます。

「SaveChanges」引数を使うことでブックに変更がある場合の処理を指定できます。「True」を設定する場合には上書き保存されます。また「FileName」引数も合わせて指定することで任意のファイル名で保存することが出来ます。「False」を指定した場合には変更を保存せずにブックを閉じます。(新規ブックの場合で「FileName」引数を指定しなかった場合には「名前を付けて保存」ダイアログが表示されて保存するファイル名を指定することになります。

```
ActiveWorkbook.Close SaveChanges:=True,FileName:="backup.xlsx"
```

「Close」メソッドは単一の Workbook オブジェクトに対してだけではなく、Workbooks コレクションに対しても実行できます。

```
Workbooks.Close
```

Workbooks コレクションに「Close」メソッドを使った場合には、全てのブックが閉じられます。この場合は引数を指定できません。よって変更されたブックがあった場合にはファイルを保存するかどうかの確認ダイアログが表示されます。

Sub TEST13() 　Workbooks.Open Filename:="2017 成績表.xlsm" 　ActiveSheet.Range("A1").Value = "保存" 　ActiveWorkbook.Close _ 　SaveChanges:=True, Filename:="2017backup.xlsm" End Sub	マクロを実行すると画面上は変化がありませんが、開いたブックには変更が加えられているので、閉じる際に「2017backup.xls」と言う名前で保存されたから閉じられます。(別の名前で保存していますので、元々のファイルには変更分は反映されません)。

ウィンドウの操作

ウィンドウを整列したり最大化/最小化などウィンドウに対する VBA による操作方法について確認します

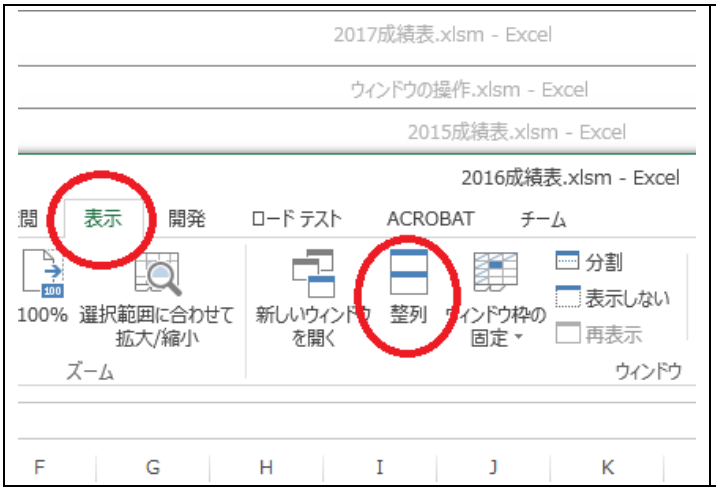
ウィンドウの参照

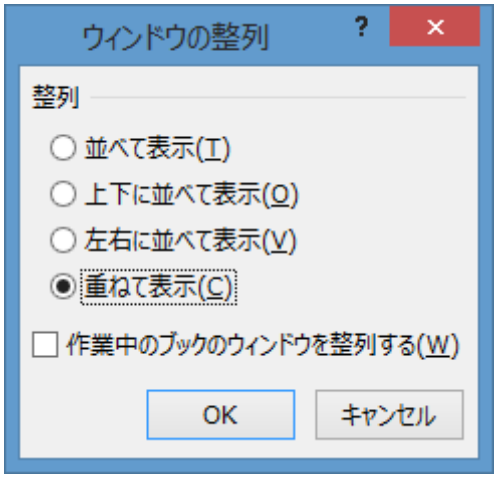
ウィンドウとは Excel 内に含まれる 1 つ 1 つのウィンドウのことです。Excel 内でブックを最大表示していると気が付きにくいですが、1 つ 1 つのブックはそれぞれ 1 つのウィンドウ内に表示されています。

ウィンドウに対する様々な処理をするにあたって、対象となるウィンドウオブジェクトを取得する方法を確認します。

Window オブジェクトの取得

まずウィンドウそのものを確認しておきます。Excel 上で複数のブックを開いた状態で、「表示」タブの「整列」をクリックして下さい。





「ウィンドウの整列」というウィンドウが表示されます。

「重ねて表示」を選択してから「OK」ボタンをクリックして下さい。

Excel 自体のウィンドウが重ねて最大表示されます。(図なし)

Window オブジェクトの取得

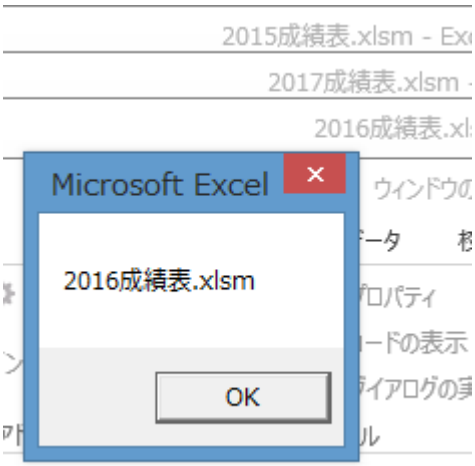
開いている全てのウィンドウは Windows コレクションに含まれています。1 つ 1 つのウィンドウを表す Window オブジェクトは Application オブジェクトの「Windows」プロパティで取得します。

```
Dim window1 As Window
Set window1 = Application.Windows(インデックス番号)
```

Application オブジェクトを省略した場合にも、デフォルトの値として Application オブジェクトが設定されているので記述してもしなくても構いません。

取得したいウィンドウを指定するにはインデックス番号で指定するかウィンドウ名で指定します。注意する点としてインデックス番号はアクティブなウィンドウが常に「1」になるためインデックス番号は常に変化します。その為、ウィンドウ名(ブック名)で指定したほうがいいでしょう。

```
Sub TEST01()
    Dim window1 As Window
    Set window1 = Application.Windows("2016 成績表.xlsm")
    MsgBox window1.Caption
End Sub
```



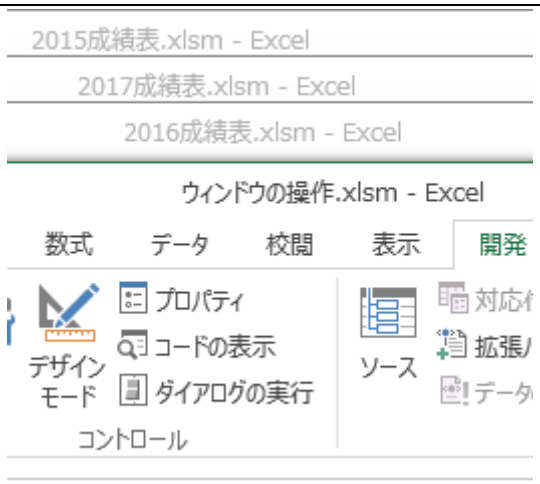
ウィンドウをアクティブにする

指定したウィンドウをアクティブにします。アクティブになったウィンドウと言うのは、一番前面に表示されているウィンドウのことです。一度にアクティブにできるウィンドウは 1 つだけです。

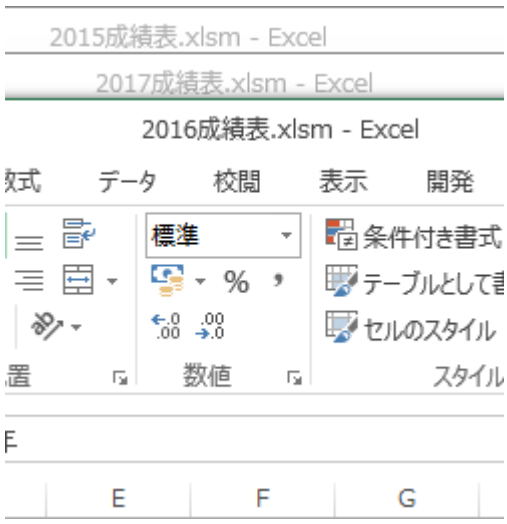
ウィンドウをアクティブにするには、アクティブにしたい Window オブジェクトに対して「Activate」メソッドを使います。

```
Dim window1 As Window
Set window1 = Application.Windows("VBAsample.xls")
window1.Activate
```

```
Sub TEST02()
    Dim window1 As Window
    Set window1 = Application.Windows("2016 成績表.xlsm")
    window1.Activate
End Sub
```



左図の様に指定したウィンドウ名がアクティブになった。



アクティブウィンドウのオブジェクトの取得

ウィンドウのオブジェクトを取得するには、インデックス番号を指定するかウィンドウ名を指定して Window オブジェクトを取得していましたが、それ以外にも現在アクティブになっているウィンドウを取得することが出来ます。

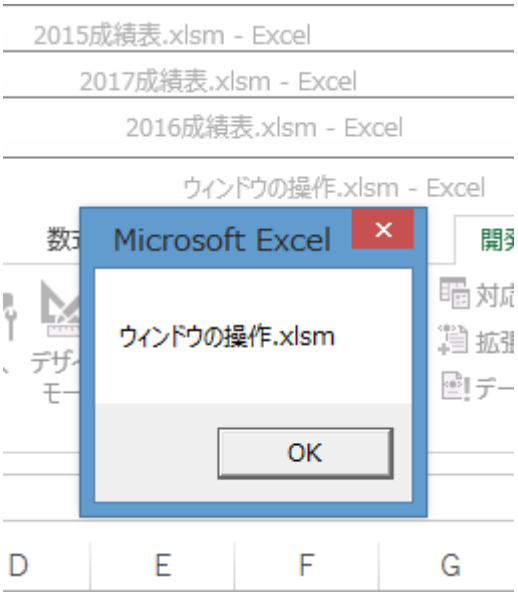
アクティブなウィンドウの Windows オブジェクトを取得するには、Application オブジェクトの「ActiveWindow」プロパティを使います。

```
Dim window1 As Window
Set window1 = Application.ActiveWindow
```

単に次のように記述しても構いません。

```
Dim window1 As Window
Set window1 = ActiveWindow
```

```
Sub TEST03()
    Dim window1 As Window
    Set window1 = ActiveWindow
    MsgBox window1.Caption
End Sub
```



ウィンドウの操作

ウィンドウの整列や拡大縮小など、ウィンドウに対する操作を行う方法を見ていきます。

ウィンドウの複製

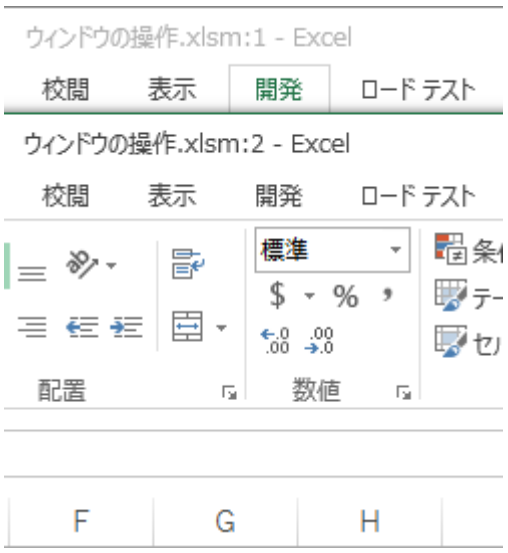
指定したウィンドウを複製します。複製されたウィンドウは同期しているため並べて表示し片方を見ながら片方で作業するといったことが可能になります。

ウィンドウを複製するには Window オブジェクトに対して「NewWindow」メソッドを使います。

```
Dim window1 As Window
Set window1 = Application.ActiveWindow
window1.NewWindow
```

複製されたウィンドウは自動的に開きます。

```
Sub TEST04()
    Dim window1 As Window
    Set window1 = ActiveWindow
    window1.NewWindow
    Windows.Arrange _
        ArrangeStyle:=xlArrangeStyleVertical, _
        ActiveWorkbook:=True
End Sub
```



Windows コレクションの「Arrange」メソッドを使ってウィンドウを整列させています。

開いている 2 つのウィンドウは同じブックを参照していますので、どちらかのブックに値を入力するともう一

方のブックにも同じように値が反映されます。複製されたウィンドウのタイトルは同じタイトルですが、タイトルの後ろに「:番号」が付与されて表示されます。番号は 1 から順に付けられます。

ウィンドウの整列

ウィンドウの整列を行います。既に開いているウィンドウを上下に並べたり左右に並べたりします。
ウィンドウを整列するには Windows コレクションに対して「Arrange」メソッドを使います。

```
Windows.Arrange ArrangeStyle:=xlArrangeStyleTiled
```

どのように整列するかは「ArrangeStyle」引数で指定します。指定可能な値は下記の 4 つです。

定数	整列方法
xlArrangeStyleCascade	重ねて表示
xlArrangeStyleTilede	並べて表示
xlArrangeStyleHorizontal	上下に並べて表示
xlArrangeStyleVertical	左右に並べて表示

設定を省略した場合は「xlArrangeStyleTiled」が使われます。

```
Sub TEST05()
    Windows.Arrange ArrangeStyle:=xlArrangeStyleCascade
    MsgBox "重ねて表示しました"
    Windows.Arrange ArrangeStyle:=xlArrangeStyleTiled
    MsgBox "並べて表示しました"
    Windows.Arrange ArrangeStyle:=xlArrangeStyleHorizontal
    MsgBox "上下に並べて表示しました"
    Windows.Arrange ArrangeStyle:=xlArrangeStyleVertical
    MsgBox "左右に並べて表示しました"
End Sub
```

3 つのブックを開いた状態から開始し、マクロを実行する整列を順次行っていきます。(図なし)

アクティブウィンドウの整列

ウィンドウの整列を行う時にアクティブウィンドウに対してだけ整列を行う事ができます。
アクティブウィンドウは通常 1 つだけですのでアクティブウィンドウだけ整列の対象としても、そのウィンドウが画面いっぱいに表示されるだけです。ただ、ウィンドウを複製している場合で、そのウィンドウがアクティブウィンドウだった場合、複製されたウィンドウなどが整列の対象となります。
アクティブウィンドウを整列するには Windows コレクションに対して「Arrange」メソッドを使う際に「ActiveWorkbook」引数を「True」に設定します。

```
Windows.Arrange ArrangeStyle:=xlArrangeStyleTiled, _
    ActiveWorkbook:=True
```

整列の仕方は「ArrangeStyle」引数で指定します。これは通常の場合と同じです。指定可能な値は下記の 4 つです。

定数	整列方法
xlArrangeStyleCascade	重ねて表示
xlArrangeStyleTilede	並べて表示
xlArrangeStyleHorizontal	上下に並べて表示
xlArrangeStyleVertical	左右に並べて表示

「ActiveWorkbook」引数が「True」の場合だけ、「SyncHorizontal」引数と「SyncVertical」引数の指定が可能です。「SyncHorizontal」引数を「True」に設定すると整列されたウィンドウが横方向のスクロールが同期するようになります。つまりどこかのウィンドウを横にスクロールすると、他のウィンドウも同じように横方向へスクロールします。「SyncVertical」引数に「True」を指定すると縦方向のスクロールが同期します。

```
Windows.Arrange ArrangeStyle:=xlArrangeStyleTiled, _
    ActiveWorkbook:=True, _
    SyncHorizontal:=True
```

```
Sub TEST06()
    Windows("ウィンドウの操作.xlsm").Activate
    ActiveWindow.NewWindow
    Windows.Arrange _
        ArrangeStyle:=xlArrangeStyleVertical, _
        ActiveWorkbook:=True, _
        SyncVertical:=True
End Sub
```

今回は元々3 つのブックが開いていた状態で、「ウィンドウの操作.xlsm」というブックをアクティブにしてから複製しています。そしてアクティブウィンドウだけを対象に左右に並べて表示させています。全部で 4 つのブックが開いているわけですが、整列の対象になっているのはアクティブウィンドウとその複製されたウィンドウだけです。今回は同時に縦方向のスクロールを同期させています。それぞれのウィンドウのタイトルに[垂直同期]と表示されています。(図なし)

タイトルの取得と変更

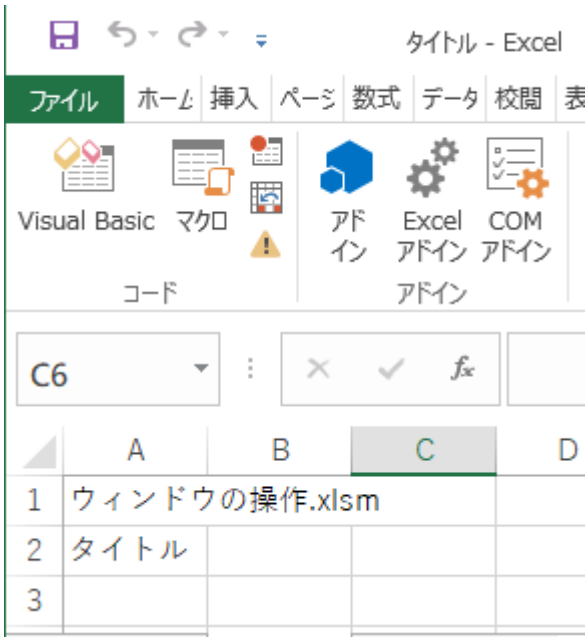
ウィンドウのタイトルにはデフォルトではウィンドウに含まれているワークブックのブック名が表示されています。このタイトルを任意の文字列に変更する事が可能です。

タイトルを変更するには、変更したい Window オブジェクトの「Caption」プロパティを使います。

```
ActiveWindow.Caption = "タイトル"
```

タイトルを変更してもブック名には変更はありません。またブックを保存してもタイトルの変更は保存されません。あくまで一時的にタイトルを変更するためだけのものです。

```
Sub TEST07()
    Dim window1 As Window
    Windows("ウィンドウの操作.xlsm").Activate
    Set window1 = ActiveWindow
    window1.ActiveSheet.Range("A1").Value _
        = window1.Caption
    window1.Caption = "タイトル"
    window1.ActiveSheet.Range("A2").Value _
        = window1.Caption
End Sub
```



表示倍率の変更

ウィンドウの表示倍率を変更してみます。表示倍率は通常の状態が 100%となっており、任意の倍率に変更して表示させることができます。

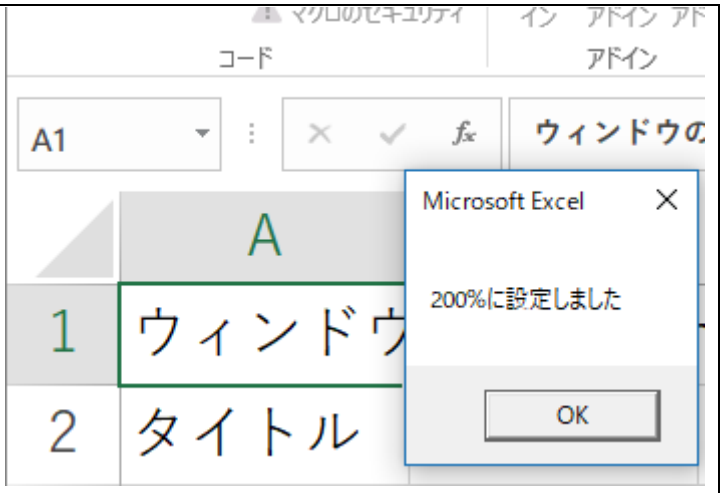
表示倍率を変更するには、変更したい Window オブジェクトの「Zoom」プロパティを使います。

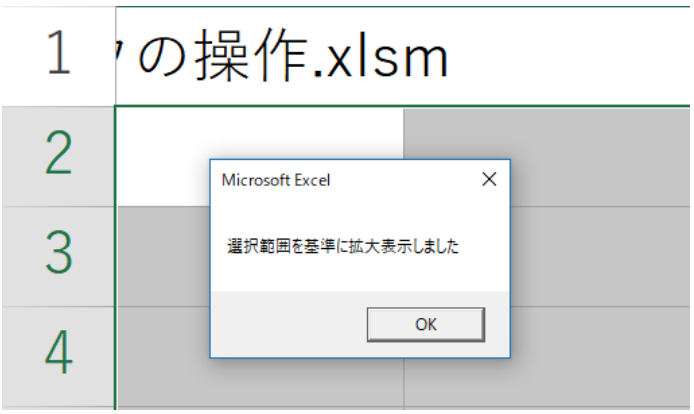
```
ActiveWindow.Zoom = 200
```

200%で表示したい場合には「Zoom」プロパティに「200」を設定します。また表示倍率を変更できるのはアクティブシートだけです。

また拡大縮小の倍率を指定する以外に、セルの一部を選択した状態で「Zoom」プロパティに「True」を設定すると、選択された領域がウィンドウいっぱいに表示されるように表示倍率を自動で設定してくれます。

```
Sub TEST08()
    Dim window1 As Window
    Windows("ウィンドウの操作.xlsm").Activate
    Set window1 = ActiveWindow
    ActiveWindow.Zoom = 200
    MsgBox "200%に設定しました"
    Range("B2:E6").Select
    ActiveWindow.Zoom = True
    MsgBox "選択範囲を基準に拡大表示しました"
End Sub
```



まず 200%に拡大しています。次にセル範囲 B2:E6 を選択して、選択したセルを基準に表示倍率を変更します。ちょうど選択した領域がウィンドウの幅または高さになるように表示倍率を自動的に設定します。	

左上の位置に表示されるセルを指定する

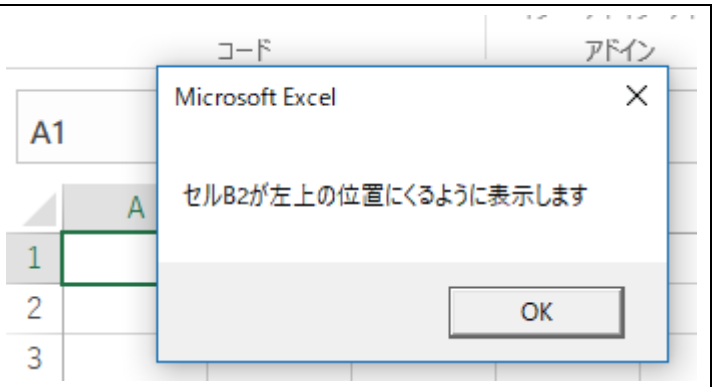
通常はウィンドウ内には(アクティブシートの)セル A1 が左上の位置に表示されています。この左上の位置に表示するセルの位置を指定することが出来ます。

行位置を指定するには Window オブジェクトの「ScrollRow」プロパティに設定します。列位置を指定するには Window オブジェクトの「ScrollColumn」プロパティに設定します。

```
ActiveWindow.ScrollRow = 2
ActiveWindow.ScrollColumn = 4
```

行及び列の位置は 1 から始まる数値で指定します。列の場合も「A,B,C...」ではなく「1,2,3...」で指定して下さい。

```
Sub TEST09()
    Dim window1 As Window
    Windows("ウィンドウの操作.xlsm").Activate
    Set window1 = ActiveWindow
    MsgBox "セル B2 が左上の位置にくるように表示します"
    window1.ScrollRow = 2
    window1.ScrollColumn = 2
End Sub
```



「OK」ボタンをクリックすると位置を移動します。

ウィンドウの最大化/最小化

ウィンドウの表示状態は最大/通常/最小の 3 つの状態があり変更することが可能です。

表示状態を指定するには Window オブジェクトの「WindowState」プロパティに設定します。

```
Dim window1 As Window
Set window1 = Windows(1)
window1.WindowState = xlNormal
```

設定できる値は次の 3 つです。

定数	表示状態
xlNormal	通常表示

xlMaximized	最大化表示
xlMinimized	最小化表示

注意する点としてはウィンドウサイズの変更が不可になっている場合はエラーとなります。

```
Sub TEST10()
    ActiveWindow.WindowState = xlMaximized
    MsgBox "最大化表示しました"
    ActiveWindow.WindowState = xlMinimized
    MsgBox "最小化表示しました"
    ActiveWindow.WindowState = xlNormal
    MsgBox "通常表示です"
End Sub
```

マクロを実行すると次のようまず最大化表示でアクティブシートを表示します。
次に最小化表示を行い、次に最小化表示を行います。最後に通常表示を行います。

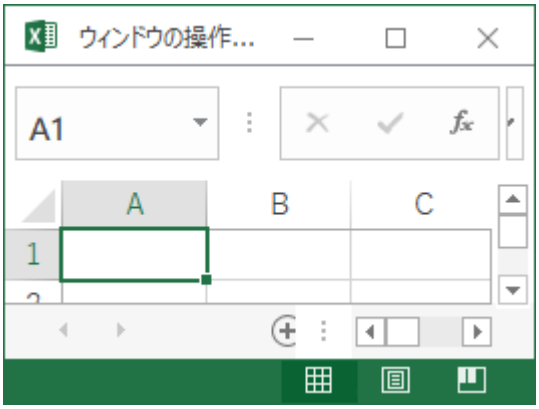
ウィンドウサイズの設定

ウィンドウの横幅及び高さの設定が可能です。設定するには Window オブジェクトに対して「Width」プロパティ及び「Height」プロパティを使います。

```
Dim window1 As Window
Set window1 = Windows("VBAsample.xls")
window1.Width = 300
window1.Height = 200
```

幅及び高さの単位はポイントです。(ピクセルでは無いので注意して下さい)。
注意する点としてはウィンドウが最大化されていたり最小化している時にサイズ設定を行おうとするとエラーとなりますので注意して下さい。

```
Sub TEST11()
    Dim window1 As Window
    Set window1 = ActiveWindow
    window1.Width = 200
    window1.Height = 150
End Sub
```

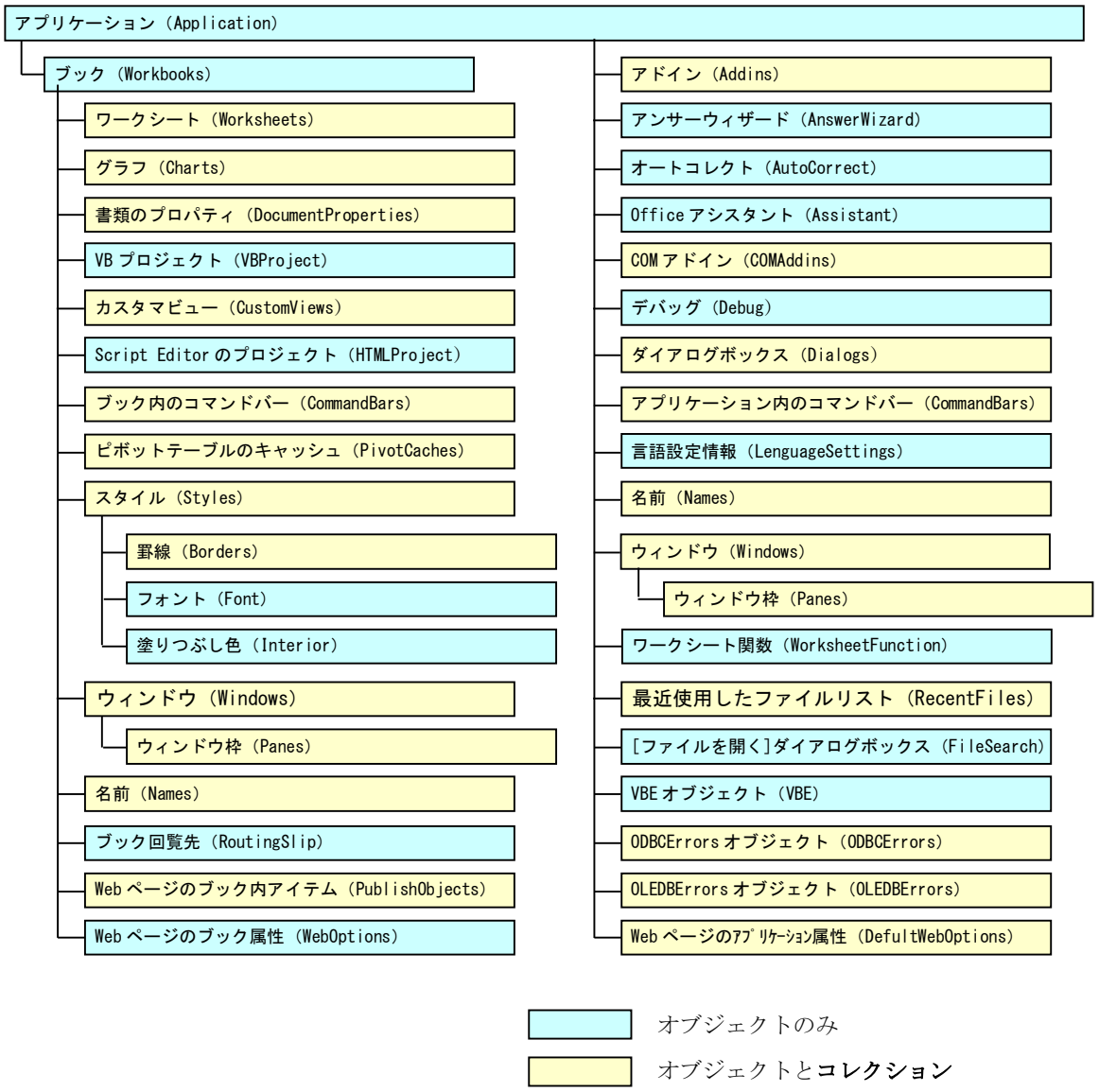


MDI と SDI について

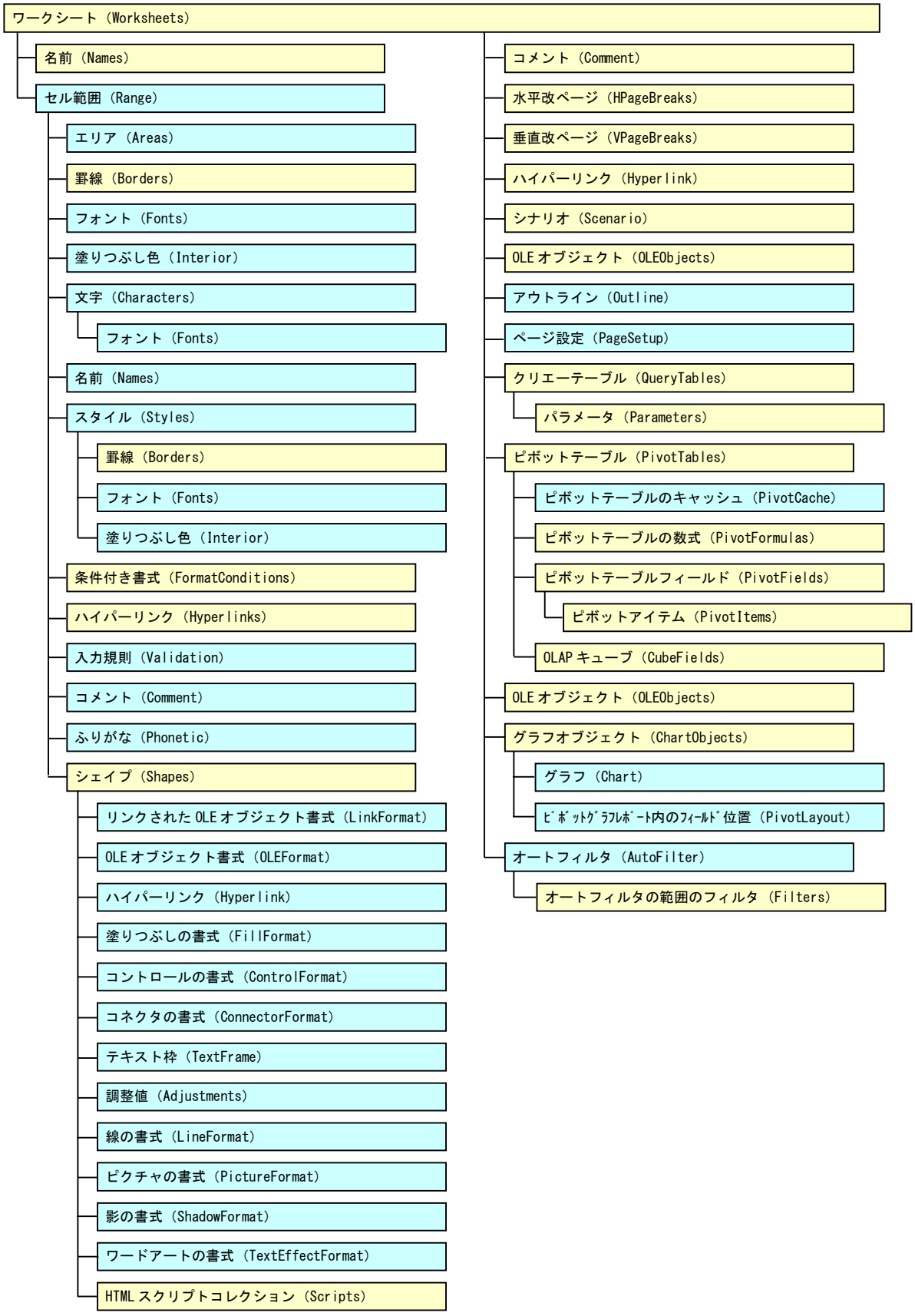
Excel 2010 までは、MDI（マルチ ドキュメント インターフェイス）が使用されていました。これは、Excel のアプリケーションウィンドウ内に、複数のブックを開き、メニューやツールバー等は、Excel ウィンドウにあるものを共有します。一方、Excel 2013 以降では、SDI（シングル ドキュメント インターフェイス）に変更され、ブックごとにウィンドウが独立するようになりました。

付録

Excel オブジェクトの階層構造



ワークシートオブジェクトの階層構造



改訂履歴	日　付	備　考
初版	2017 年 6 月 19 日	2 日間コース用に新規作成 http://www.officepro.jp/excelvba/を参考にしています。 (株式会社バズワードからの利用許諾・配布許可を受けています。)
1.01 版	2017 年 8 月 8 日	小訂正
1.02 版	2018 年 7 月 29 日	小訂正
1.03 版	2018 年 8 月 7 日	小訂正

メモ：