

Microsoft Small Basic

プログラミング入門

Small Basic とプログラミング

コンピューター プログラミングとは、プログラミング言語を使用してコンピューター ソフトウェアを作成するプロセスです。私達が英語やスペイン語やフランス語を話したり理解するのと同じように、コンピューターも 特定の言語で書かれたプログラムを理解することができます。これらはプログラミング言語と呼ばれています。初期の頃はわずかなプログラミング言語しかなく、それらはとても簡単に学んだり理解することができました。しかしコンピューターやソフトウェアがさらに洗練され、プログラミング言語もさらに複雑な概念を取り入れながら急速に進化してきました。その結果、現代のプログラミング言語とそれらの概念は、初心者にとっては理解するのが難しいものとなっています。その事が、コンピュータープログラミングを学んだり、プログラミングをすることの妨げになってきました。

Small Basic は初心者にとって、非常に簡単で、楽しく親しみやすいプログラミング言語になっています。Small Basic は、プログラミングへの壁を取り除き、コンピューター プログラミングの素晴らしい世界へと導くことを目的としています。

Small Basic の環境

まずはじめに Small Basic の環境について簡単に紹介します。はじめて Small Basic を起動すると、以下の図のようなウィンドウが表示されます。

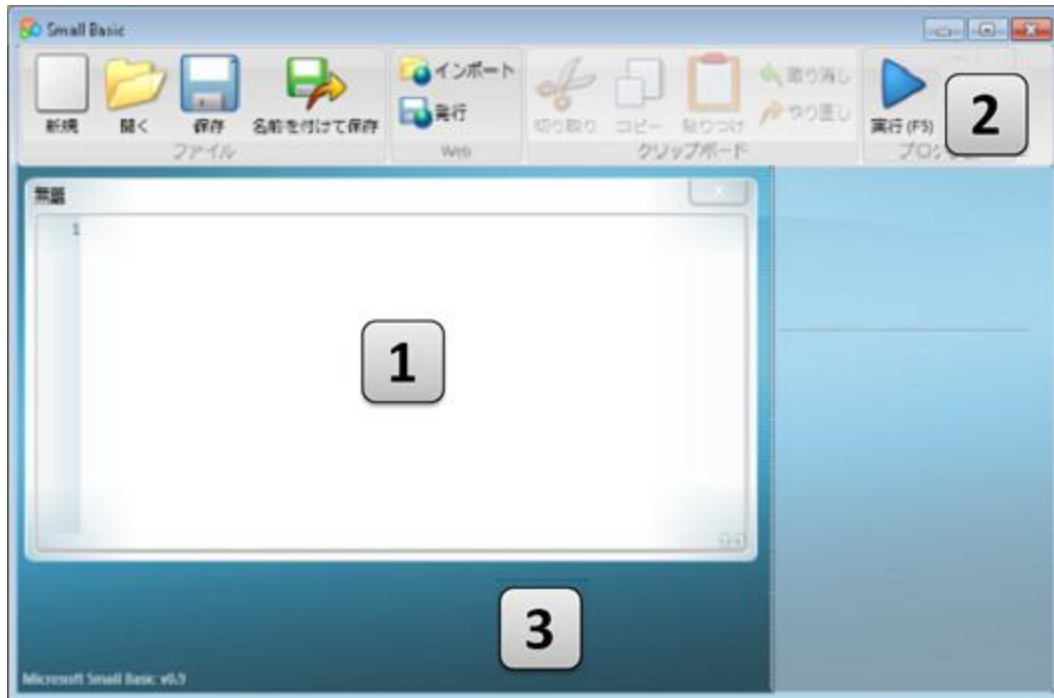


図 1 - Small Basic の環境

これが Small Basic プログラムを書いたり、実行したりする環境です。この環境には、いくつか個別の要素があり、それぞれ番号によって示されています。

番号 [1] に示される **エディター** は、私達が Small Basic のプログラムを書く場所です。サンプル プログラムを開いたり、以前に保存したプログラムを開いたりする場合にも、このエディター内に表示されます。プログラムを編集したり、後から使うために保存することもできます。

同時に複数のプログラムを開いたり、作業をしたりすることもできます。作業中の各プログラムは、それぞれ別のエディター内に表示されます。現在作業中のプログラムを含むエディターのことを、**アクティブ エディター**と呼びます。

番号 [2] に示される **ツールバー** は、アクティブ エディターや環境でコマンドを実行するために使われます。ツールバー内の色々なコマンドについては章を追って学習していきます。

番号 [3] に示される **Surface** は、すべてのエディター ウィンドウが表示される場所です。

はじめてのプログラム

Small Basic の環境について学んだので、さっそくプログラミングにとりかかりましょう。上記のように、エディターは私達がプログラムを書く場所です。それでは、以下の行をエディター内に書いてみましょう。

```
TextWindow.WriteLine("Hello World")
```

これは私達の最初の Small Basic プログラムです。正しく入力できたら、以下の図のようになっているはずです。

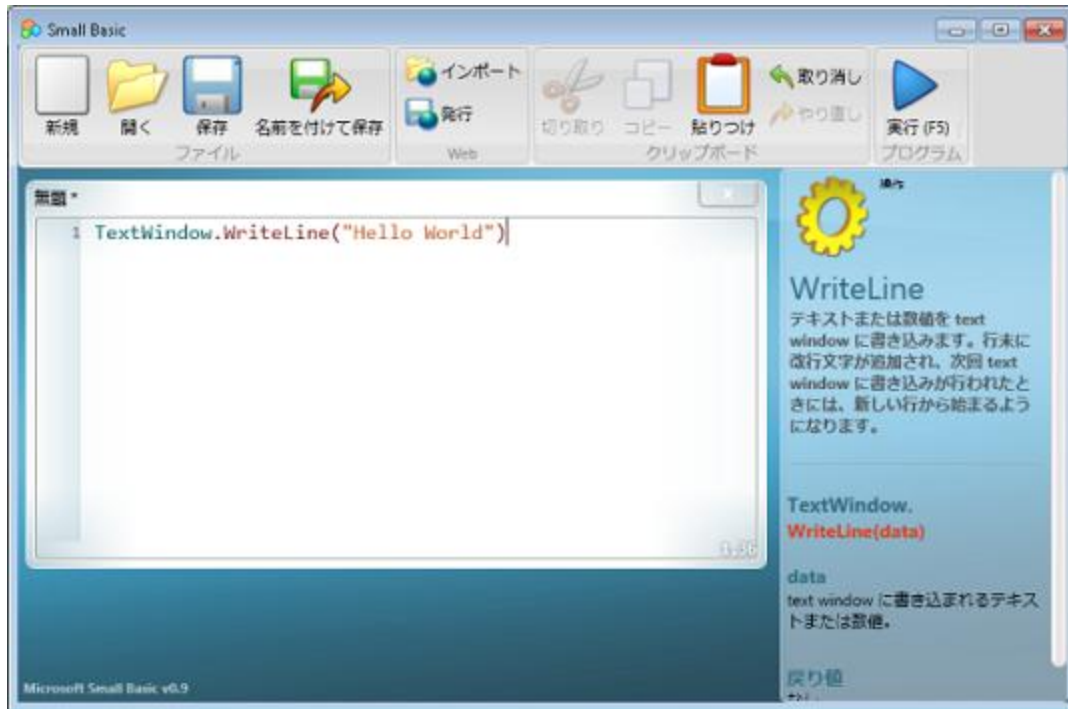


図 2 - はじめてのプログラム

新しいプログラムを入力したので、何が起きるか、実行して見てみましょう。プログラムは、ツールバー上の 実行 ボタンをクリックするか、またはキーボード上のショートカットキー F5 を押す事によって実行できます。すべてうまくいった場合、実行結果は以下のようになるはずです。

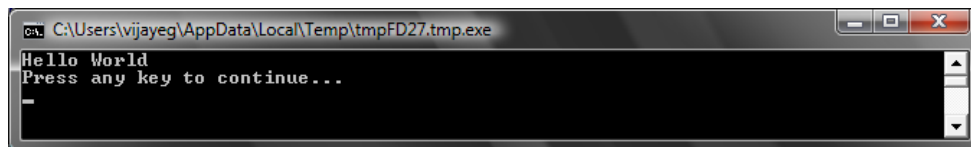


図 3 - はじめてのプログラムの出力結果

よくできました! あなたは、はじめての Small Basic プログラムを書いて実行しました。とても小さくて簡単なプログラムですが、本物のコンピュータープログラマーになる大きな第一歩です! 大きなプログラムを作成する前に、ここで、もう一つだけ、説明しておくことがあります。ここで起きたことを理解する必要があります - 私達がコンピューターに何をするように伝え、また、コンピューターはどのように何をするのか理解したのか? 次の章では、さらに理解を深めるために、私達がたった今書いたプログラムを分析してみることにします。

はじめてのプログラムを入力している時に、アイテムのリストがポップアップ表示されるのに気づいたかもしれません (図 4)。これは “intellisense” と呼ばれるもので、プログラミングをすばやく入力するのを助けるためのものです。上/下の矢印キーを押して、そのリスト内をたどることができます、そして使いたいものが見つかったら、Enter キーを押すことによって、選択されたアイテムをプログラム内に挿入できます。



図 4 - Intellisense

プログラムの保存

もし Small Basic を閉じて、先ほど入力したプログラムを後から編集したい場合には、プログラムを保存することができます。実際、プログラムを時々保存することは、予期しないシャットダウンや、電源が落ちたりした場合にも、情報を失わずに済むのでよい事です。現在のプログラムを保存したい場合には、ツールバー上の“保存”アイコンをクリックするか、またはショートカット“Ctrl+S” (Ctrl キーを押しながら S キーを押します) を押すことによって保存できます。

はじめてのプログラムを理解する

コンピューター プログラミングとは何でしょう？

プログラムはコンピューターに対する一連の命令です。これらの命令はコンピューターに何をするかを正確に伝え、コンピューターは常にこれらの命令に従います。人間と同じように、コンピューターも、コンピューターが理解できる言語で指定されている場合にのみ、命令に従うことができます。これらはプログラミング言語と呼ばれています。コンピューターが理解できる言語には、たくさんの言語があり、**Small Basic** はその一つです。

友達との間の会話を想像してみてください。あなたとあなたの友達は、言葉を使って、文章を作り、情報を相互にやりとりするでしょう。同様に、プログラミング言語にも、たくさんの言葉があり、文章を作ることもでき、コンピューターへ情報を伝えることができます。プログラムとは、基本的にプログラマーとコンピューターが理解できる一連の文章（数少ないものから、何千もの場合もあります）なのです。

Small Basic プログラム

典型的な Small Basic プログラムは多くのステートメントを含んでいます。プログラムの各行はステートメントであり、各ステートメントはコンピューターへの命令です。コンピューターに Small Basic プログラムを実行さ

せると、コンピューターは、プログラムの最初のステートメントを読み出します。コンピューターは私達が何を言おうとしているのかを理解し、命令を実行します。最初のステートメントの実行が終わると、コンピューターはプログラムに戻り、次のステートメントを読み出して実行します。コンピューターは同じ事をプログラムの最後に到達するまで繰り返します。最後に到達する時がプログラムが完了する時です。

コンピューターが理解できる言語にはたくさんの言語があります。Java、C++、Python、VB、等、これらはすべて強力な現代コンピューター言語で、簡単なプログラムから複雑なソフトウェアプログラムを開発するために使われています。

はじめてのプログラムに戻ってみましょう

私達を書いたはじめてのプログラムです:

```
TextWindow.WriteLine("Hello World")
```

これは1つのステートメントを含むとても簡単なプログラムです。そのステートメントはコンピューターに、テキストウインドウ内に一行の文 **Hello World** と書くように伝えています。

コンピューターの中では、文字通りに解釈されます:

```
Write Hello World
```

お気づきかもしれませんが、ステートメントは、文が単語に分割できるように、短いセグメントに分割できます。最初のステートメントには、3つの異なるセグメントがあります:

- a) TextWindow
- b) WriteLine
- c) "Hello World"

点、括弧および引用符はすべて、コンピューターが正しくその意味を理解できるよう、ステートメントの中で適切な位置にある必要があります。

あなたは、はじめてのプログラムを実行した時に表示された、黒いウインドウのことを覚えているかもしれません。その黒いウインドウは `TextWindow` と呼ばれ、またコンソールとも呼ばれています。そのウインドウは、プログラムの結果が表示される場所です。私達のプログラムでは `TextWindow` のことをオブジェクトと呼びます。そのようなプログラムの中で使えるオブジェクトはたくさんあります。それらのオブジェクトに対していくつかの異なる操作を行うことができます。既にプログラムの中で `WriteLine` ステートメントを使用しました。また、`WriteLine` ステートメントの後に、引用符で囲まれた `Hello World` があることにも既にお気づきかもしれません。このテキストは、`WriteLine` ステートメントへ入力値として渡され、その後ユーザーに対して表示されます。これはステートメントに対する入力と呼ばれています。いくつかのステートメントは複数の入力を取る一方で、入力をまったく必要としないステートメントもあります。

引用符、空白、括弧などの句読点はコンピュータープログラムではとても重要です。それらの位置や数によって、プログラムの意味や解釈が変わります。

二番目のプログラム

はじめてのプログラムを理解したところで、いくつか色を追加してもう少し見栄えのするプログラムを作ってみましょう。

```
TextWindow.ForegroundColor = "Yellow"  
TextWindow.WriteLine("Hello World")
```

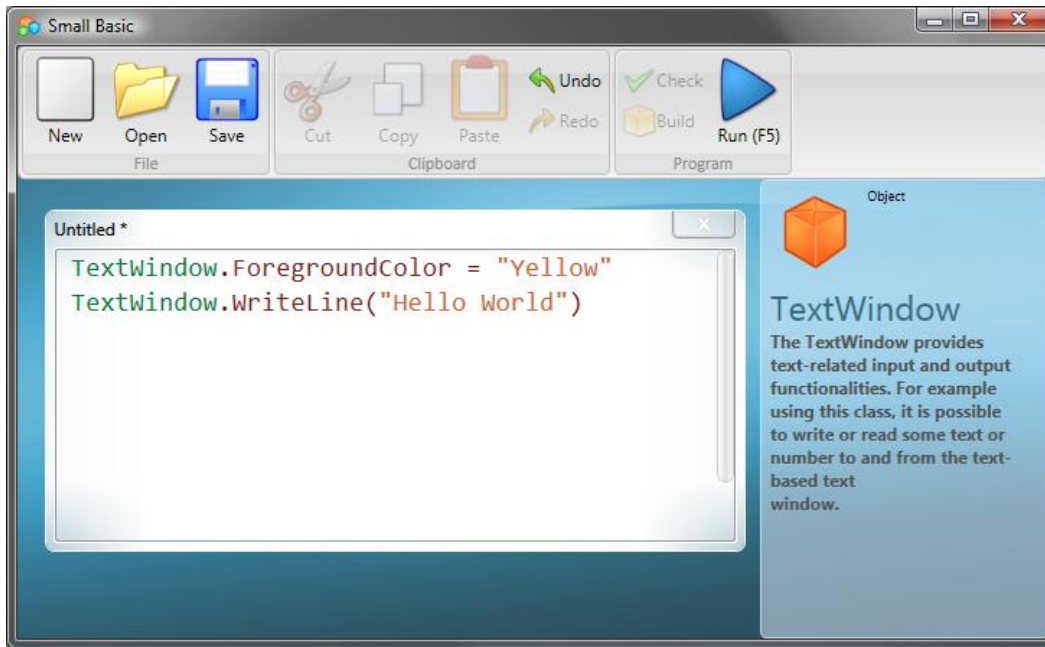


図 5 - 色の追加

上記のプログラムを実行すると、TextWindow 内に同じ“Hello World”という文を先に実行した時に表示された灰色ではなく、黄色で表示されることに気づくことでしょう。

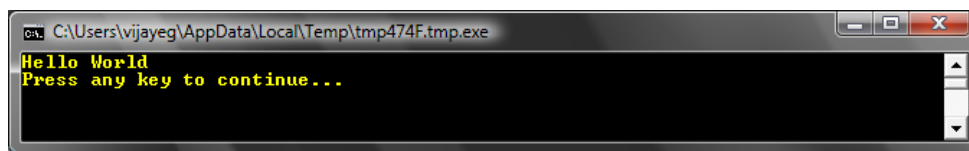


図 6 - 黄色で表示された Hello World

元のプログラムに追加した新しいステートメントに注目してください。“Yellow”を値として設定した *ForegroundColor* という新しい言葉が使われています。これは“Yellow”を *ForegroundColor* に割り当てたという意味です。さて、*ForegroundColor* と *WriteLine* 命令の違いですが、*ForegroundColor* は何の入力もなく、括弧も必要としません。そのかわりに、イコール(等式)の後ろに記号と言葉がついていました。*ForegroundColor* は *TextWindow* のプロパティとして定義されています。以下 *ForegroundColor* プロパティとして使える値の一覧です。“Yellow”をこれらの値の一つに置き換えてみて結果を見てみましょう - 引用符をつけるのを忘れないでください。

Black
Blue
Cyan
Gray
Green
Magenta
Red
White

Yellow

DarkBlue

DarkCyan

DarkGray

DarkGreen

DarkMagenta

DarkRed

DarkYellow

プログラム中での変数の使用

もし一般的な“Hello World?”と言うかわりに、ユーザー名と一緒に“Hello”と言えたらいいなあと思いませんか。そうするためには、まず最初にユーザーに彼/彼女の名前を聞き、その名前をどこかに保存して、それからユーザー名と共に“Hello”を表示することになります。どうやって表示するか、見てみましょう:

```
TextWindow.Write("Enter your Name: ")
name = TextWindow.Read()
TextWindow.WriteLine("Hello " + name)
```

このプログラムを入力して実行すると、以下のような出力結果が表示されます:

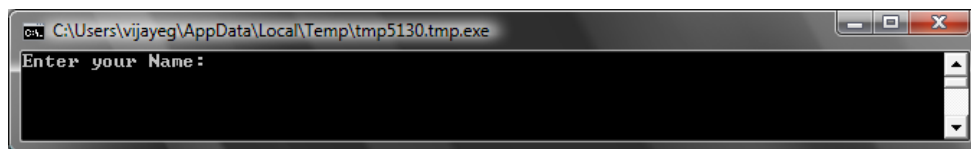


図 7 - ユーザー名の入力

あなたの名前を入力して ENTER を押すと、以下の出力結果が表示されます:

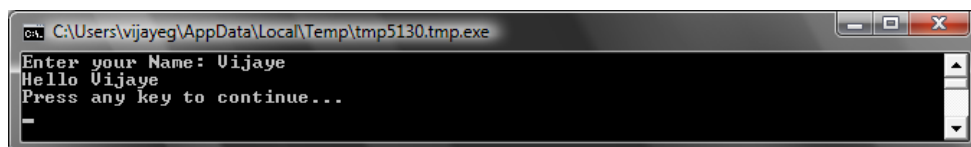


図 8 - フレンドリーな Hello

ここで、プログラムを再度実行すると、同じ質問が再度表示されます。別の名前を入力すると、コンピューターは入力された名前と Hello を表示します。

プログラムの分析

たった今実行したプログラムの中で、気にかかった行:

```
name = TextWindow.Read()
```

`Read()` は入力がないものの `WriteLine()` のように見えます。そのオペレーションは基本的に、コンピューターにユーザーが何か入力し、ENTER キーを押すまで待つように、と伝えます。ユーザーが ENTER キーを押すと、ユーザーが入力したものをプログラムへ返します。興味深い点は、ユーザーが何を入力したとしても、それらの入力は `name` という変数に保存されます。変数は、一時的に値を保存し、それらを後から使うことができる場所として定義されます。上記では、変数 `name` はユーザーの名前を保存するために使われています。

次の行も興味深いです:

```
TextWindow.WriteLine("Hello " + name)
```

ここが変数 `name` に保存された値を使う場所です。`name` 変数に保存された値を取り出し、“Hello” の後ろに追加して `TextWindow` に書き出します。

一旦変数が設定されると、何度でもその変数を使うことができます。例えば以下のように使えます:

`WriteLine` のように `Write` は `ConsoleWindow` 内での別な関数です。`Write` は `ConsoleWindow` に何かを表示しますが、後続のテキストを現在のテキストと同じ行に表示します。

```
TextWindow.Write("Enter your Name: ")
name = TextWindow.Read()
TextWindow.Write("Hello " + name + ". ")
TextWindow.WriteLine("How are you doing " + name + "?")
```

以下のような結果が表示されます:

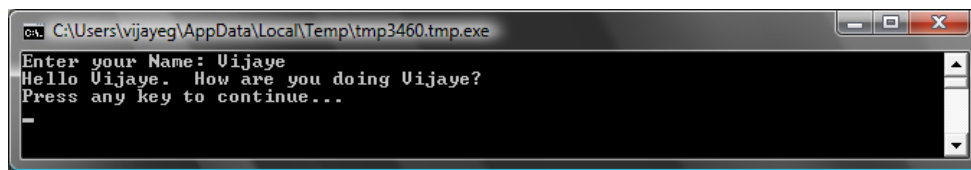


図 9 - 変数の再利用

変数のネーミングに関するルール

変数は変数に関連した名前を持ち、そのようにして変数を識別します。これらの変数のネーミングについては、簡単なルールと、とてもよいガイドラインがあります。それらは:

1. 名前は文字で始まり、**if**、**for**、**then** 等のキーワードではないこと。
2. 名前にはどんな文字や数字の組み合わせでもよく、アンダースコアも使えます。
3. 意味のある変数名にすると使いやすいです - 変数の長さに制限はないので、変数の目的がわかる名前をつけましょう。

数字を使ってみましょう

変数にユーザーの名前を保存する方法を見ました。次に学ぶいくつかのプログラムでは、どのように複数の数字を変数内に保存するかを見て行きます。それでは、とても簡単なプログラムから始めてみましょう:

```
number1 = 10
number2 = 20
number3 = number1 + number2
TextWindow.WriteLine(number3)
```

このプログラムを実行すると、以下のような結果が表示されます:

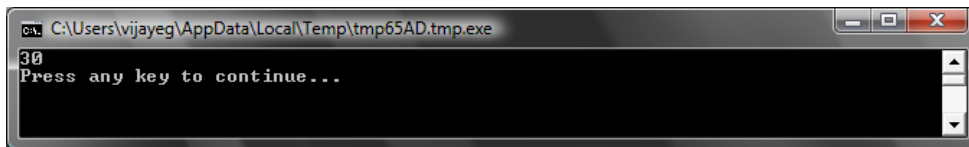


図 10 - 二つの数字を追加

プログラムの最初の行で、変数 **number1** に 10 という値を割り当てています。そして二行目では、変数 **number2** に 20 という数字を割り当てています。三行目では、**number1** と **number2** を足して、その結果を **number3** に割り当てています。この例では **number3** の値は 30 になります。そして、その値が **TextWindow** に出力されています。

ここで、プログラムを少しだけ変えて結果を見てみましょう:

数字のまわりに引用符がないことに注意してください。数字には引用符は必要ありません。文字を使う場合にのみ引用符が必要です。

```
number1 = 10
number2 = 20
number3 = number1 * number2
TextWindow.WriteLine(number3)
```

上のプログラムでは `number1` に `number2` をかけて(乗算) 結果を `number3` に保存します。そしてその結果は下のようになります:

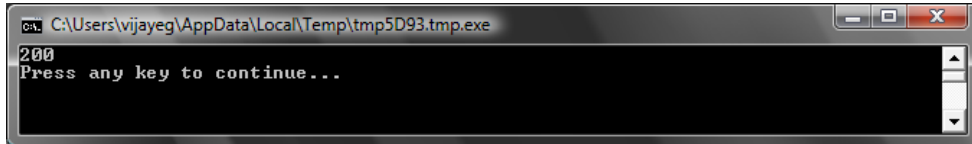


図 11 - 二つの数字の乗算

同様に、引き算や割り算もできます。引き算の例:

```
number3 = number1 - number2
```

割り算の記号は '/' です。割り算のプログラムの例:

```
number3 = number1 / number2
```

割り算の結果:

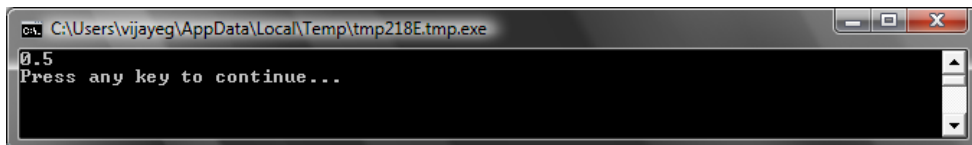


図 12 - 二つの数字の除算

簡単な温度コンバーター

次のプログラムでは、温度を華氏から摂氏に変換するための式 $C = (F - 32) \times \frac{5}{9}$ を使います。

まずはじめに、ユーザーから温度を華氏で入力してもらい、それを変数に保存します。`TextWindow.ReadNumber` という特別な関数を使ってユーザーが入力した数字を読み取ります。

```
TextWindow.Write("Enter temperature in Fahrenheit: ")
fahr = TextWindow.ReadNumber()
```

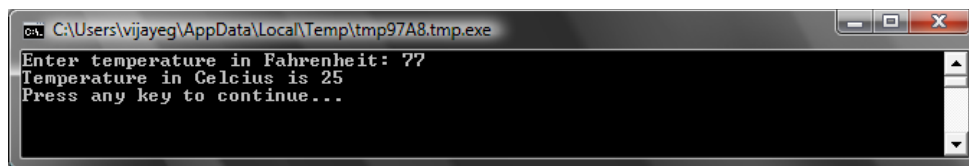
変数に華氏温度を保存したら、次のように摂氏に変換することができます:

```
celsius = 5 * (fahr - 32) / 9
```

括弧はコンピューターに `fahr - 32` の部分を先に計算して、それから残りの処理をするように指示しています。あとは結果をユーザーに表示するだけです。それらをすべて一緒にしてみると、このようなプログラムになります:

```
TextWindow.Write("Enter temperature in Fahrenheit: ")
fahr = TextWindow.ReadNumber()
celsius = 5 * (fahr - 32) / 9
TextWindow.WriteLine("Temperature in Celcius is " + celsius)
```

そしてプログラムの結果は以下のようになります:



```
C:\Users\vijayeg\AppData\Local\Temp\tmp97A8.tmp.exe
Enter temperature in Fahrenheit: 77
Temperature in Celcius is 25
Press any key to continue...
```

図 13 - 温度変換

最初のプログラムに戻ってみましょう、一般的な *Hello World* と言うだけでなく、1 日の時間帯によって、*Good Morning World*、や *Good Evening World* と言えたらカッコいいと思いませんか？ 次のプログラムでは、コンピューターが、午後 12 時前だったら *Good Morning World*、もし午後 12 時以降だったら *Good Evening* と言うように作成します。

```
If (Clock.Hour < 12) Then
  TextWindow.WriteLine("Good Morning World")
EndIf
If (Clock.Hour >= 12) Then
  TextWindow.WriteLine("Good Evening World")
EndIf
```

プログラムを実行する時間によって、次の出力結果のどちらかが表示されます：



図 14 – Good Morning World

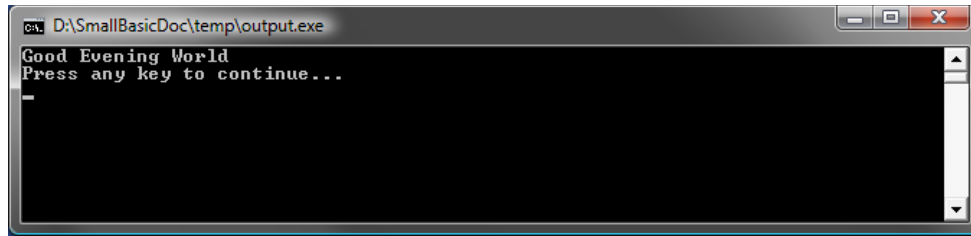


図 15 - Good Evening World

プログラムの最初の三行を分析してみましょう。既にお気づきのように、この行はコンピューターにもし `Clock.Hour` が 12 よりも小さい値だったら “Good Morning World.” と表示するように指示しています。 `If`、`Then` および `EndIf` はプログラムの実行時にコンピューターによって理解される特別な言葉です。`If` の後ろにはいつも条件がきます、この例では `(Clock.Hour < 12)` が条件になります。括弧はコンピューターがあなたの意図を理解するために必要だということを覚えておいてください。条件の後には `then` があり、それが実際に実行するステートメントです。そしてステートメントの後には `EndIf` が来ます。これはコンピューターに条件による実行が終了したことを知らせます。

Small Basic では `Clock` オブジェクトを使って現在の日にちと時間にアクセスすることができます。そのオブジェクトにはたくさんのプロパティがあり、現在の日にち、月、年、時間、分、秒の情報を個別に取得することができます。

`then` と `EndIf` の間には、複数のステートメントを指定することができ、コンピューターは条件が有効であればそれらすべてのステートメントを実行します。例えば、このような例のプログラムを書くことができます：

```
If (Clock.Hour < 12) Then
  TextWindow.Write("Good Morning. ")
  TextWindow.WriteLine("How was breakfast?")
EndIf
```

Else

この章の最初にあったプログラムの中で、二番目の条件はなんだか冗長に思えたかもしれません。`Clock.Hour` の値は 12 よりも大きいのか小さいのどちらかです。二番目のチェックは実は必要ありませんでした。このような時には、二つの `if..then..endif` 文を新しい言葉、`else` を使って一つに短くすることができます。

もし `else` を使ってこのプログラムを書き直すと、以下のようになります：

```
If (Clock.Hour < 12) Then
  TextWindow.WriteLine("Good Morning World")
Else
  TextWindow.WriteLine("Good Evening World")
EndIf
```


そしてこのプログラムは最初にしたプログラムとまったく同じように動き、このことはコンピュータープログラミングでのとても重要なレッスンを私達に教えています:

プログラミングでは、一般的に同じ事をするのにも多くのやり方があります。ある方法が他の方法よりも適していることが時々あります。もっとプログラムをたくさん書いて、経験を積んでいくと、これらの異なる技術やそれらの利点や不利な点に気づくようになります。

インデントの設定

すべての例の中で、*If*、*Else* および *EndIf* の間のステートメントはインデント(字下げ)になってます。このインデントは必要ありません。コンピューターはそれらのインデント無しでもプログラムをきちんと理解します。しかし、それらのインデントは私達がプログラムを読んだり理解する助けになっています。というわけで、そのようなブロックの間にインデントを設定することは、良い事とされています。

偶数または奇数

さて、*If..Then..Else..EndIf* ステートメントのコツを覚えたので、与えられた数字が偶数か奇数かを表示するプログラムを書いてみましょう。

```
TextWindow.Write("Enter a number: ")
num = TextWindow.ReadNumber()
remainder = Math.Remainder(num, 2)
If (remainder = 0) Then
    TextWindow.WriteLine("The number is Even")
Else
    TextWindow.WriteLine("The number is Odd")
EndIf
```

このプログラムを実行すると、以下のような出力結果が表示されます:



```
D:\SmallBasicDoc\temp\output.exe
Enter a number: 32485
The number is Odd
Press any key to continue...
```

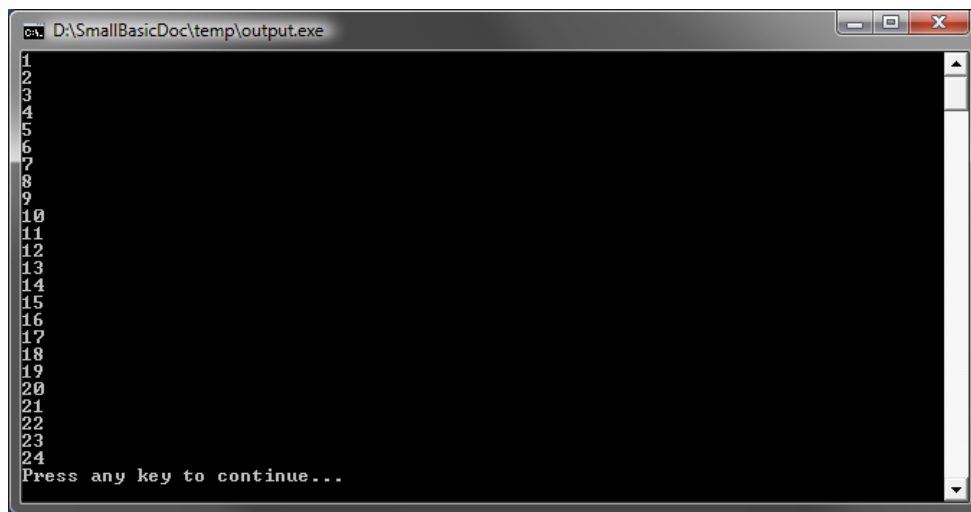
図 16 - 偶数または奇数

このプログラムでは、役に立つ新しい命令 **Math.Remainder** が使われています。そして、既にお気づきかもしれませんが、**Math.Remainder** は最初の数字を二番目の数字で割って、その余りを返します。

分岐

第二章で、コンピューターはプログラムを上から下の順に、一度に一文ずつ処理するというのを思い出してください。しかしながら、コンピューターに順番通りではなく、他の命令文にジャンプさせることができる特別なステートメントがあります。次のプログラムを見てみましょう。

```
i = 1
start:
  TextWindow.WriteLine(i)
  i = i + 1
  If (i < 25) Then
    Goto start
  EndIf
```



```
cmd, D:\SmallBasicDoc\temp\output.exe
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
Press any key to continue...
```

図 17 - Goto の使用

上のプログラムで、私達は変数 *i* に 1 という値を割り当てました。そして、コロン (:) で終わる新しいステートメントを追加しました。

```
start:
```

これらはラベルと呼ばれています。ラベルはコンピューターが理解するブックマークのようなものです。ブックマークは、すべてにユニークな名前が付けられている限り、どんな名前をつけることもでき、好きなだけ追加することもできます。

もう一つの興味深いステートメントの例:

```
i = i + 1
```

これはコンピューターに変数 i に 1 を足して、その値を変数 i に割り当てています。もし変数 i の値がこのステートメントの前までは 1 だった場合、このステートメントを実行した後は 2 になります。

そして最後は、

```
If (i < 25) Then
  Goto start
EndIf
```

この部分は、コンピューターに、もし変数 i の値が 25 より小さい場合には、ブックマーク **start** からステートメントを実行開始するように伝えています。

終わりのない実行

Goto ステートメントを使うと、コンピューターに何かを何度でも繰り返させることができます。例えば、偶数または奇数プログラムを以下のように変更すると、プログラムは永遠に走り続けます。ウィンドウの右上端にある、閉じる (X) ボタンをクリックすることによって、プログラムを終了できます。

```
begin:
TextWindow.Write("Enter a number: ")
num = TextWindow.ReadNumber()
remainder = Math.Remainder(num, 2)
If (remainder = 0) Then
  TextWindow.WriteLine("The number is Even")
Else
  TextWindow.WriteLine("The number is Odd")
EndIf
Goto begin
```



The screenshot shows a command prompt window titled "cmd, D:\SmallBasicDoc\temp\output.exe". The output text is as follows:

```
The number is Odd
Enter a number: 456
The number is Even
Enter a number: 2222
The number is Even
Enter a number: -34
The number is Even
Enter a number: -859
The number is Odd
Enter a number: 3302090
The number is Even
Enter a number:
```

図 18 - 偶数または奇数プログラムが走り続けているところ

For ループ

以下のコードは、前の章で作成したプログラムです。

```
i = 1
start:
  TextWindow.WriteLine(i)
  i = i + 1
  If (i < 25) Then
    Goto start
  EndIf
```

このプログラムでは、1 から 24 の数字が順番に出力されます。プログラミングでは、変数をインクリメント(増加)する処理はよく使われるので、プログラミング言語では簡単に使うことができます。上記のプログラムは、以下のプログラムと同じように動作します。

```
For i = 1 To 24
  TextWindow.WriteLine(i)
EndFor
```

出力結果は、以下のとおりです。

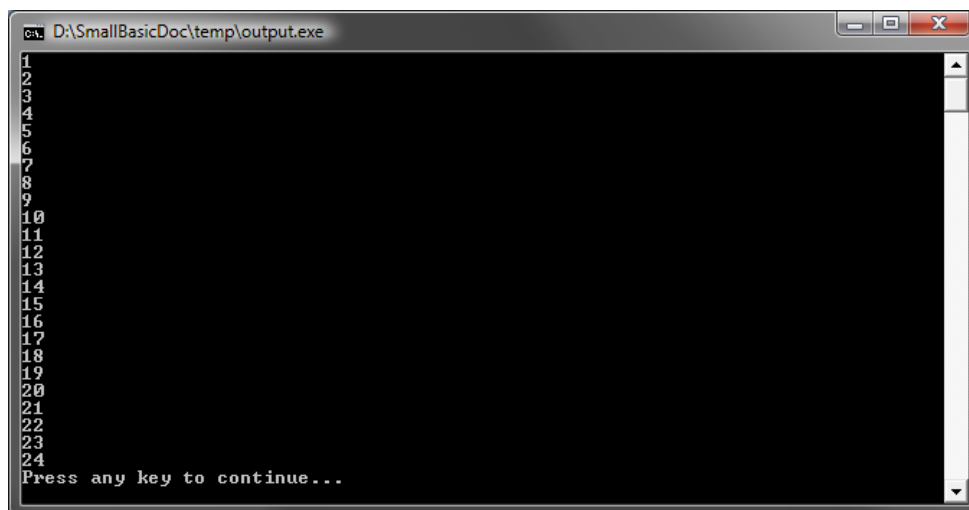


図 19 - For ループの使用

7 行のプログラムを 4 行のプログラムに減らしても、これらのプログラムの動作は同じです。前の章で説明したように、同じ処理を実行する場合でも複数の方法があります。

For..EndFor は、プログラミング用語ではループと呼ばれます。変数に初期値と終了値を入れて、その変数をインクリメントできます。変数がインクリメントされるたびに、**For..EndFor** 内のステートメントが実行されます。

たとえば、1 から 24 の間の奇数をすべて出力する場合など、増分値を 1 ではなく 2 に指定して、ループを使うこともできます。

```
For i = 1 To 24 Step 2
  TextWindow.WriteLine(i)
EndFor
```



図 20 - 奇数のみ出力

For ステートメントの **Step 2** は、*i* の値を 2 つずつインクリメントします。**Step** を使って、インクリメントする値を指定できます。また、次の例のように、**Step** に負の値を指定して、減算することもできます。

```
For i = 10 To 1 Step -1
    TextWindow.WriteLine(i)
EndFor
```



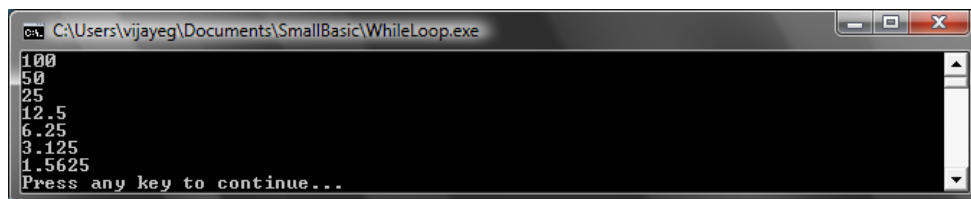
```
10
9
8
7
6
5
4
3
2
1
Press any key to continue...
```

図 21 - 減算

While ループ

While ループは、別のループの方法で、繰り返し回数が決まっていない場合に使うと便利です。For ループは、指定した回数の繰り返しですが、While ループは指定した条件式が正になるまで実行されます。以下の例では、*number* の値が 1 以下になるまで、*number* の値が二等分されます。

```
number = 100
While (number > 1)
    TextWindow.WriteLine(number)
    number = number / 2
EndWhile
```



```
100
50
25
12.5
6.25
3.125
1.5625
Press any key to continue...
```

図 22 - 2等分ループ

このプログラムでは、*number* の初期値は 100 です。*number* の値が 1 を超えている間、While ループが実行されます。ループが実行されている間、*number* の値が出力され、2 で除算されます。繰り返し 2 で除算された *number* の値が、このプログラムの出力結果です。

このループが何回実行されるか分からないので、For ループを使ってこのプログラムを作るのは非常に大変です。While ループでは、条件を指定することで、簡単にループを続行または終了させることができます。

While ループは、If..Then ステートメントに書き換えることもできます。たとえば、以下のプログラムを実行しても、実行結果は同じです。

```
number = 100
startLabel:
TextWindow.WriteLine(number)
number = number / 2
If (number > 1) Then
    Goto startLabel
EndIf
```

実際に、コンピューター内部では、While ループは If..Then と Goto ステートメントに書き換えられます。

グラフィックスを始める

これまでのサンプルでは、TextWindow を使って Small Basic 言語の基本を説明してきました。Small Basic は強力なグラフィックス機能を持っていますので、この章ではそれらを使っていきます。

GraphicsWindow の紹介

これまで TextWindow を使ってテキストや数字を取り扱ってきたように、Small Basic は **GraphicsWindow** を使って図形を描画する機能を提供します。では早速 GraphicsWindow を使ってみましょう。

```
GraphicsWindow.Show()
```

上記のプログラムを実行させると、これまでのような背景が黒のテキストウィンドウではなく、以下のような背景が白のウィンドウが表示されます。ここではまだこのウィンドウの上で何もしていませんが、これがこの章での土台となるウィンドウになります。右上の“X”ボタンをクリックすることで、このウィンドウを閉じることができます。

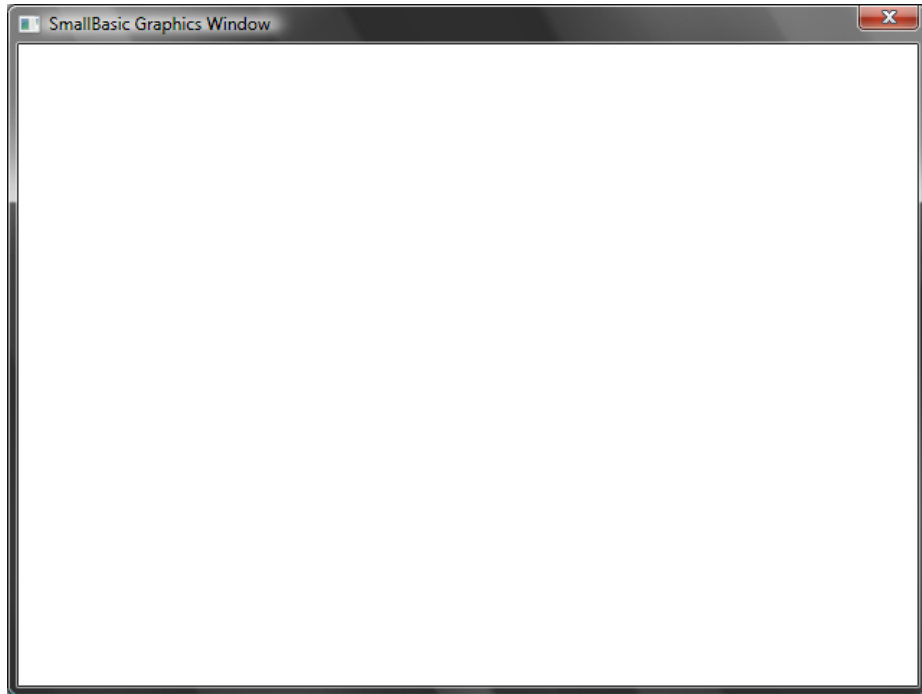


図 23 - 空の Graphics Window

Graphics Window を設定する

Graphics window は表示方法を希望どおりにカスタマイズすることができます。タイトルの変更や背景、サイズの変更などができます。少し Graphics window をカスタマイズして、その動作を理解しましょう。

```
GraphicsWindow.BackgroundColor = "SteelBlue"  
GraphicsWindow.Title = "My Graphics Window"  
GraphicsWindow.Width = 320  
GraphicsWindow.Height = 200  
GraphicsWindow.Show()
```

以下が、カスタマイズした Graphics Window の見た目です。背景色を Appendix B に掲載されているたくさんの値の中から選択して設定することができます。これらのプロパティを触って、どのようにウインドウの見た目を変更することができるのかを見てください。

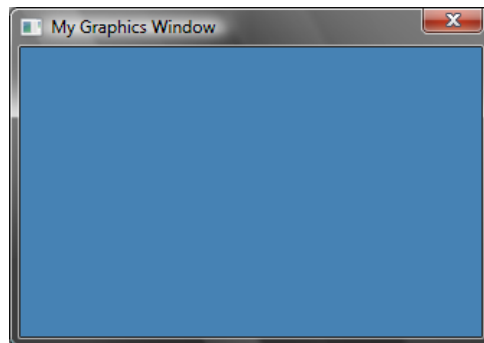


図 24 - カスタマイズした Graphics Window

直線を描く

Graphics Window の準備が整ったところで、その上に図形を描いたり、テキストや絵なども描いたりすることができます。ここでは何か簡単な図形を描くことから始めましょう。以下が、いくつかの直線を Graphics Window 上に描くためのプログラムです。

```
GraphicsWindow.Width = 200
GraphicsWindow.Height = 200
GraphicsWindow.DrawLine(10, 10, 100, 100)
GraphicsWindow.DrawLine(10, 100, 100, 10)
```

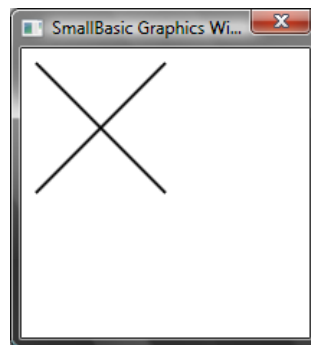


図 25 - 十字型の直線

プログラム中の最初の 2 行はウインドウの準備をするためのものであり、次の 2 行は十字型の直線を描くためのものです。DrawLine の中で指定されている最初の 2 つの数値は、直線の開始点の X 座標と Y 座標の値であり、残りの 2 つの数値は、直線の終点の X 座標

色の名前を使用するかわりに、Web の色の表記 (#RRGGBB) を使用することもできます。例えば、#FF0000 は赤を示しますし、#FFFF00 は黄色を示します。

と Y 座標の値です。コンピューター グラフィックスの世界において特徴的なのは、座標 (0,0) はウインドウの左上の端になることです。実際、ウインドウに表示される座標軸は、第四象限として考えられることになります。

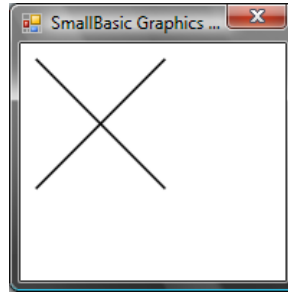


図 26 - 座標軸

直線を描くプログラムに戻りましょう。Small Basic は直線のプロパティ、例えば直線の色や太さなどを変更することができます。では最初に、直線の色を変更してみましょう。

```
GraphicsWindow.Width = 200  
GraphicsWindow.Height = 200  
GraphicsWindow.PenColor = "Green"  
GraphicsWindow.DrawLine(10, 10, 100, 100)  
GraphicsWindow.PenColor = "Gold"  
GraphicsWindow.DrawLine(10, 100, 100, 10)
```

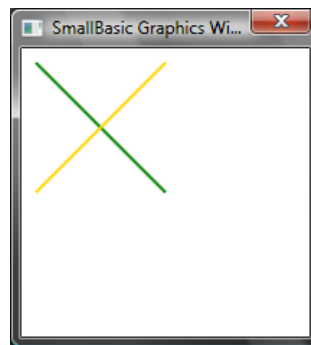


図 27 - 線の色を変更

今度はサイズも変更してみましょう。以下のプログラム中で、直線の幅を既定値の1から変更して10にしています。

```
GraphicsWindow.Width = 200  
GraphicsWindow.Height = 200  
GraphicsWindow.PenWidth = 10  
GraphicsWindow.PenColor = "Green"  
GraphicsWindow.DrawLine(10, 10, 100, 100)  
GraphicsWindow.PenColor = "Gold"
```

```
GraphicsWindow.DrawLine(10, 100, 100, 10)
```

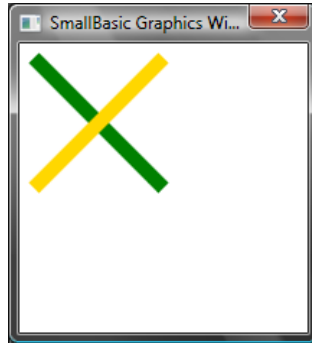


図 28 - 太くてカラフルな直線

PenWidth および PenColor は、直線を描くためのペンを変更します。これらの変更は直線の描写時に影響するだけでなく、このペンの変更を加えた後で描かれるあらゆる図形に影響します。

前の章で学んだ繰り返し命令を使用することで、複数の直線とその太さを変更しながら描くようなプログラムを簡単に書くことができます。

```
GraphicsWindow.BackgroundColor = "Black"  
GraphicsWindow.Width = 200  
GraphicsWindow.Height = 160  
GraphicsWindow.PenColor = "Blue"  
For i = 1 To 10  
  GraphicsWindow.PenWidth = i  
  GraphicsWindow.DrawLine(20, i * 15, 180, i * 15)  
endfor
```

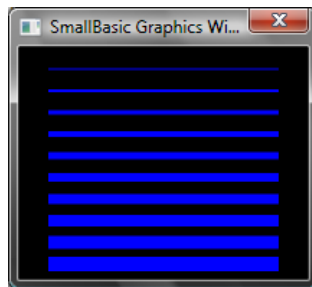


図 29 - 複数の太さのペン

このプログラムの中で特徴的なのは、PenWidth を繰り返し処理の中で毎回大きくして、それを使用して新しい直線を古い直線の下に描いていることです。

図形を描いたり塗りつぶしたりする

図形を描こうとすると、通常2つの操作が各図形に存在します。それは描く操作と塗る操作です。描く操作は図形の外枠をペンを使って描き、塗る操作は図形をブラシを使って塗りつぶします。例えば以下のプログラムでは、2つの長方形を描いています。ひとつは赤のペンを使用して描き、もうひとつは緑のブラシを使って塗っています。

```
GraphicsWindow.Width = 400
GraphicsWindow.Height = 300
GraphicsWindow.PenColor = "Red"
GraphicsWindow.DrawRectangle(20, 20, 300, 60)
GraphicsWindow.BrushColor = "Green"
GraphicsWindow.FillRectangle(60, 100, 300, 60)
```

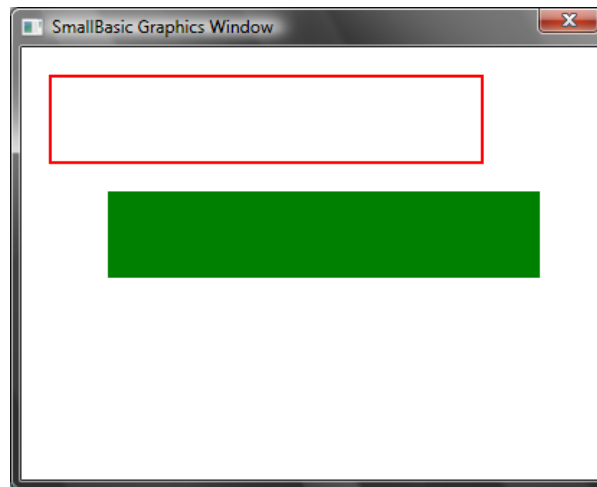


図 30 描画と塗りつぶし

長方形を描いたり塗ったりするために、4つの数値を指定する必要があります。最初の2つの数値は、左上端の X 座標と Y 座標を指定するためのものです。3番目の数値は長方形の横幅を指定するためのもので、4番目の数値はその高さを指定するためのものです。これらと同様の指定方法で、以下のプログラムのように楕円を描いたり塗ったりする場合にも使用できます。

```
GraphicsWindow.Width = 400
GraphicsWindow.Height = 300
GraphicsWindow.PenColor = "Red"
GraphicsWindow.DrawEllipse(20, 20, 300, 60)
GraphicsWindow.BrushColor = "Green"
GraphicsWindow.FillEllipse(60, 100, 300, 60)
```

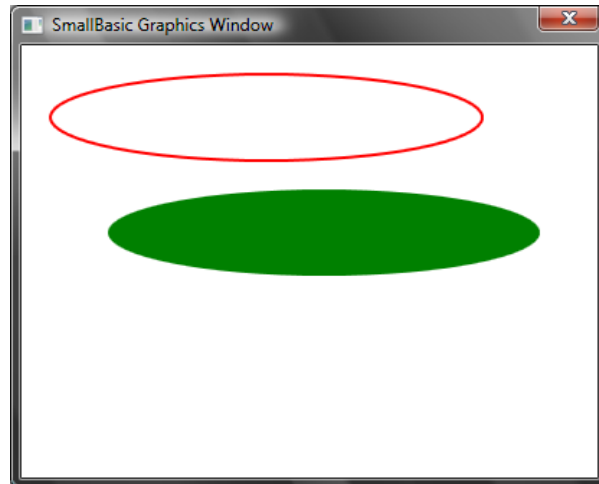


図 31 - 楕円の描画と塗りつぶし

楕円は実際には円の一般形です。もしも正円を描きたい場合には、横幅と高さを同じ値に指定して楕円を描くことで実現できます。

```
GraphicsWindow.Width = 400  
GraphicsWindow.Height = 300  
GraphicsWindow.PenColor = "Red"  
GraphicsWindow.DrawEllipse(20, 20, 100, 100)  
GraphicsWindow.BrushColor = "Green"  
GraphicsWindow.FillEllipse(100, 100, 100, 100)
```

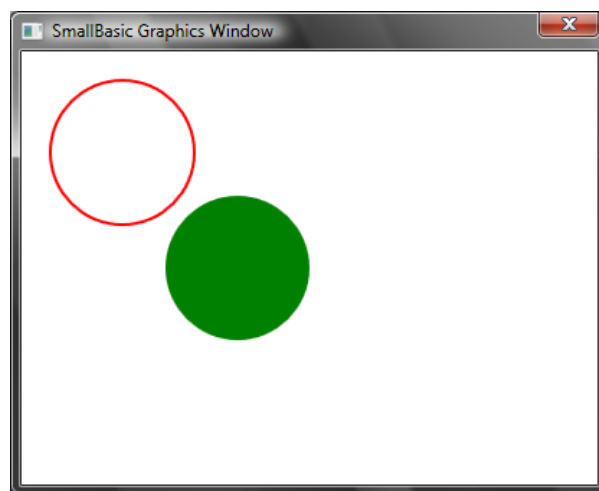


図 32 - 円

図形を楽しむ

これまでに学んだいろいろなことを使いながら、この章を楽しんでいきましょう。この章は、これまでに学んだことを組み合わせて見栄えのいいプログラムを作るための興味深い方法を示すサンプルから構成されています。

矩形描画

ここではループを使用して、複数の矩形をサイズを増加させながら描いてみましょう。

```
GraphicsWindow.BackgroundColor = "Black"  
GraphicsWindow.PenColor = "LightBlue"  
GraphicsWindow.Width = 200  
GraphicsWindow.Height = 200  
  
For i = 1 To 100 Step 5  
    GraphicsWindow.DrawRectangle(100 - i, 100 - i, i * 2, i * 2)  
EndFor
```

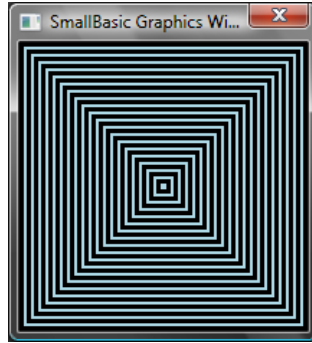


図 33 - 矩形描画

円形描画

前のプログラムを変形させて、四角の代わりに円を描いてみましょう。

```
GraphicsWindow.BackgroundColor = "Black"  
GraphicsWindow.PenColor = "LightGreen"  
GraphicsWindow.Width = 200  
GraphicsWindow.Height = 200  
  
For i = 1 To 100 Step 5  
    GraphicsWindow.DrawEllipse(100 - i, 100 - i, i * 2, i * 2)  
EndFor
```

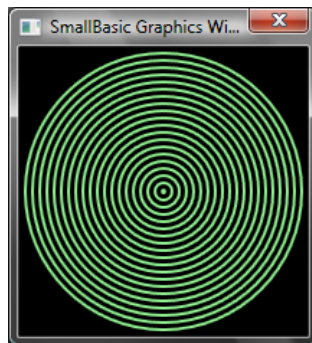


図 34 - 円形描画

ランダム化

このプログラムはランダムな色をブラシにセットするために `GraphicsWindow.GetRandomColor` を使用し、そして円の X 座標・Y 座標をセットするために `Math.GetRandomNumber` を使っています。これらのオペレーションは、毎

回プログラムを実行するために異なった結果になるような興味深いプログラムにするために、組み合わせて使用することができます。

```
GraphicsWindow.BackgroundColor = "Black"  
For i = 1 To 1000  
  GraphicsWindow.BrushColor = GraphicsWindow.GetRandomColor()  
  x = Math.GetRandomNumber(640)  
  y = Math.GetRandomNumber(480)  
  GraphicsWindow.FillEllipse(x, y, 10, 10)  
EndFor
```

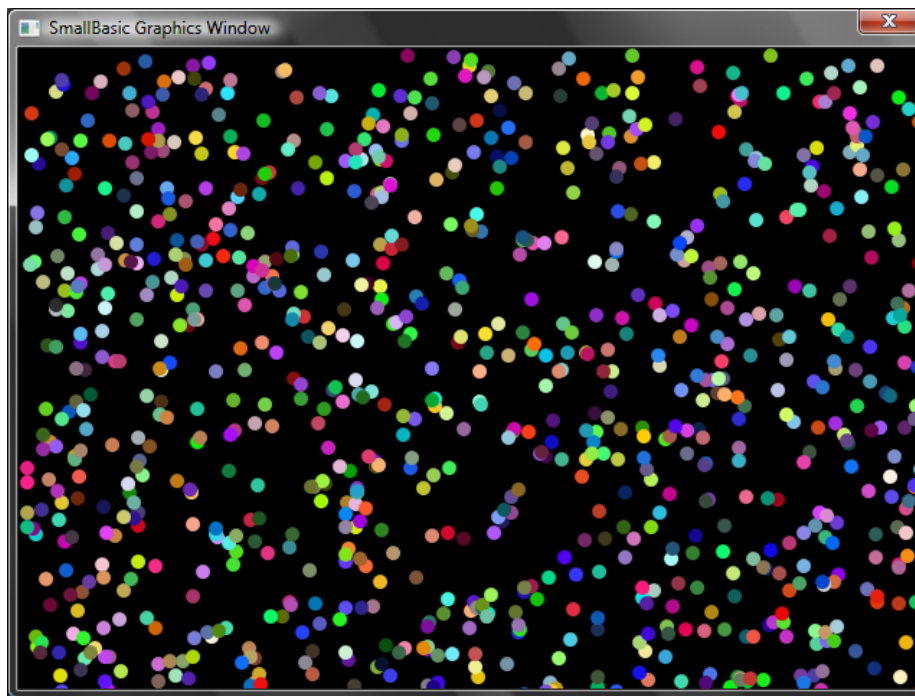


図 35 - ランダム化

フラクタル描画

以下のプログラムはシンプルなフラクタル三角形を乱数を使用して描画します。フラクタルとは、細分化することができるジオメトリ形で、細分化された個々のパーツは正確に元の図形と相似形をなしています。このプログラムの場合は、数百もの三角形のそれぞれが、その親となる三角形の相似形となるような図形を描画しています。そしてこのプログラムの実行には数秒の時間がかかるため、ただの点がゆっくりと三角形を形作るのを見ることができます。このプログラムのロジックを説明するのは少々困難なので、それはあなたのためのプログラムを解読する練習としておきたいと思います。

```
GraphicsWindow.BackgroundColor = "Black"  
x = 100
```

```
y = 100
```

```
For i = 1 To 100000
```

```
  r = Math.GetRandomNumber(3)
```

```
  ux = 150
```

```
  uy = 30
```

```
  If (r = 1) then
```

```
    ux = 30
```

```
    uy = 1000
```

```
  EndIf
```

```
  If (r = 2) Then
```

```
    ux = 1000
```

```
    uy = 1000
```

```
  EndIf
```

```
  x = (x + ux) / 2
```

```
  y = (y + uy) / 2
```

```
  GraphicsWindow.SetPixel(x, y, "LightGreen")
```

```
EndFor
```

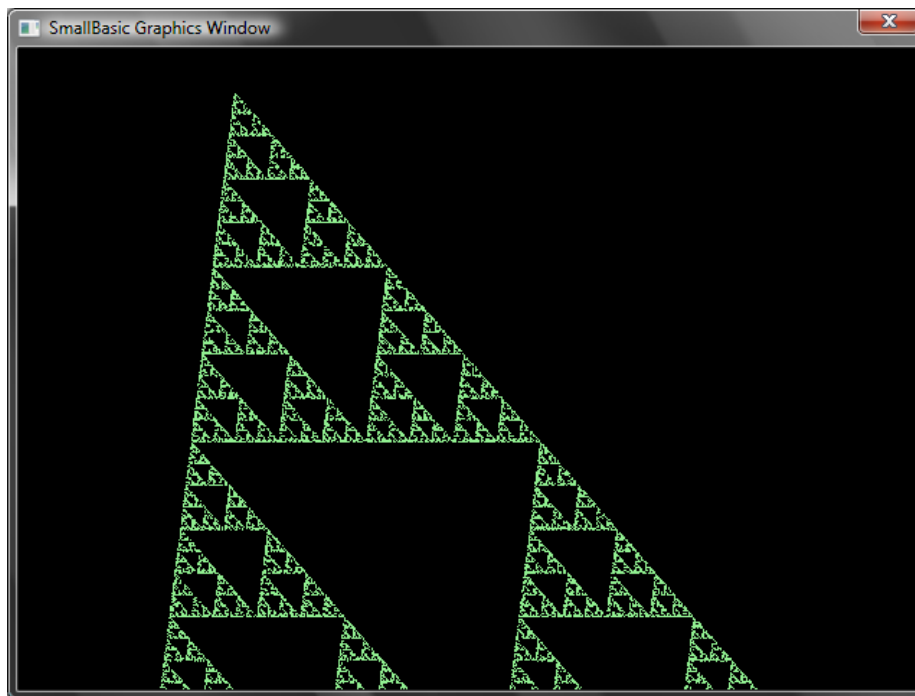


図 36 -フラクタル三角形描画

もしもプログラム中で描画される点が少しずつ三角形を形作るところをよく見たいのであれば、**Program.Delay** を使って遅延処理をループの中に入れることができます。

```
GraphicsWindow.BackgroundColor = "Black"
x = 100
y = 100

For i = 1 To 100000
    r = Math.GetRandomNumber(3)
    ux = 150
    uy = 30
    If (r = 1) then
        ux = 30
        uy = 1000
    EndIf

    If (r = 2) Then
        ux = 1000
        uy = 1000
    EndIf

    x = (x + ux) / 2
    y = (y + uy) / 2

    GraphicsWindow.SetPixel(x, y, "LightGreen")
    Program.Delay(2)
EndFor
```

遅延処理を増やしていけば、プログラムはより遅くなります。数値を変えて実験して、最もいいと感じるようにしましょう。

その他の変更可能な点として、以下の行を

```
GraphicsWindow.SetPixel(x, y, "LightGreen")
```

以下のように置き換えることができます。

```
color = GraphicsWindow.GetRandomColor()
GraphicsWindow.SetPixel(x, y, color)
```

この変更により、このプログラムはランダムな色を使って三角形を描くようになります。

タートルグラフィックス

Logo 言語について

1970 年代に、Logo と呼ばれるたいへんシンプルながらパワフルなプログラミング言語が存在し、若干の研究者たちの間で使用されていました。そして後にタートルグラフィックスと呼ばれるものが Logo 言語に追加され、“タートル”が画面上に表示されるようになり、Move Forward (前進), Turn Right (右旋回), Turn Left (左旋回)などのコマンドに反応するようになりました。

このタートルを使うことで、ユーザーは興味深い形状を画面の上にかくことができるようになりました。これによって、この言語はすべての年代の人達に対して利用が可能でかつアピールできるものになり、そして 1980 年代には広く人気のあるものになっていきました。

Small Basic には**タートル** オブジェクトと、多数の Small Basic のプログラムから呼び出すことのできるコマンドとが付属しています。この章ではそのタートルを使って、画面上にグラフィックスを表示させましょう。

タートル

最初に、タートルを画面上に表示させる必要があります。これはシンプルな一行のプログラムで実現できます。

```
Turtle.Show()
```

このプログラムを実行させると、前の章で見たような白いウィンドウが表示され、今回は画面の中央にタートルが表示されます。このタートルが私たちの指示に従って様々な図形を描画します。

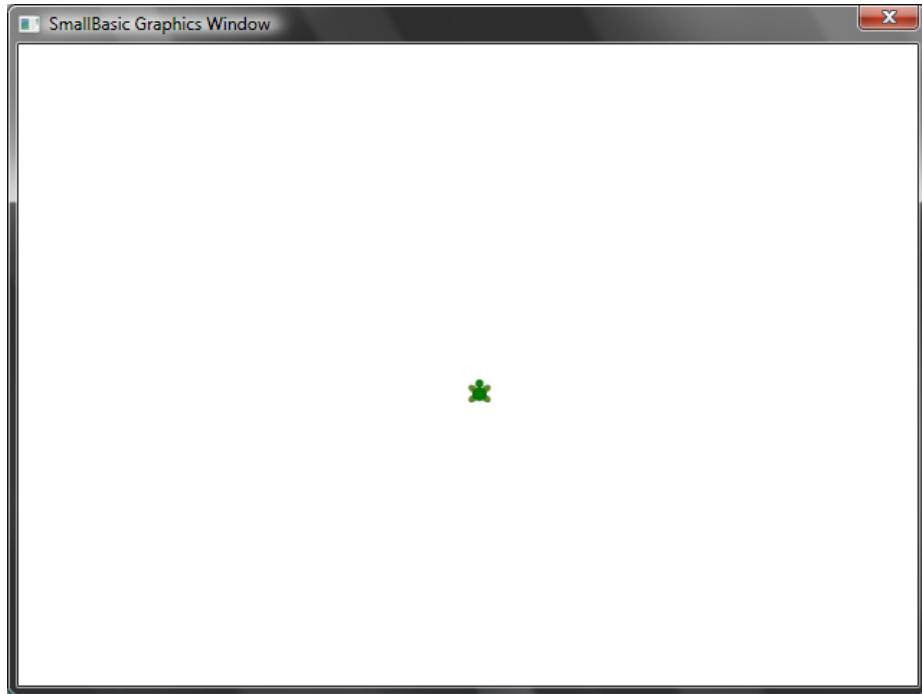


図 37 - タートルが表示される

動作と描画

タートルが理解できる命令の一つに **Move** があります。この操作は数値を入力として使用します。この数値はタートルがどのくらいの長さの移動をするかを表します。例えば以下のサンプルでは、タートルに対して 100 ピクセル分動くことを指示しています。

```
Turtle.Move(100)
```

このプログラムを実行すると、タートルがゆっくりと 100 ピクセル分上に移動するのが見られます。それと同時に、直線がタートルの後ろに描かれていることがわかります。タートルが動作を終了したとき、以下のような結果になります。

タートルに対して操作をするとき、Show()を呼び出す必要はありません。タートルは命令を実行するたびに自動的に結果表示を行います。

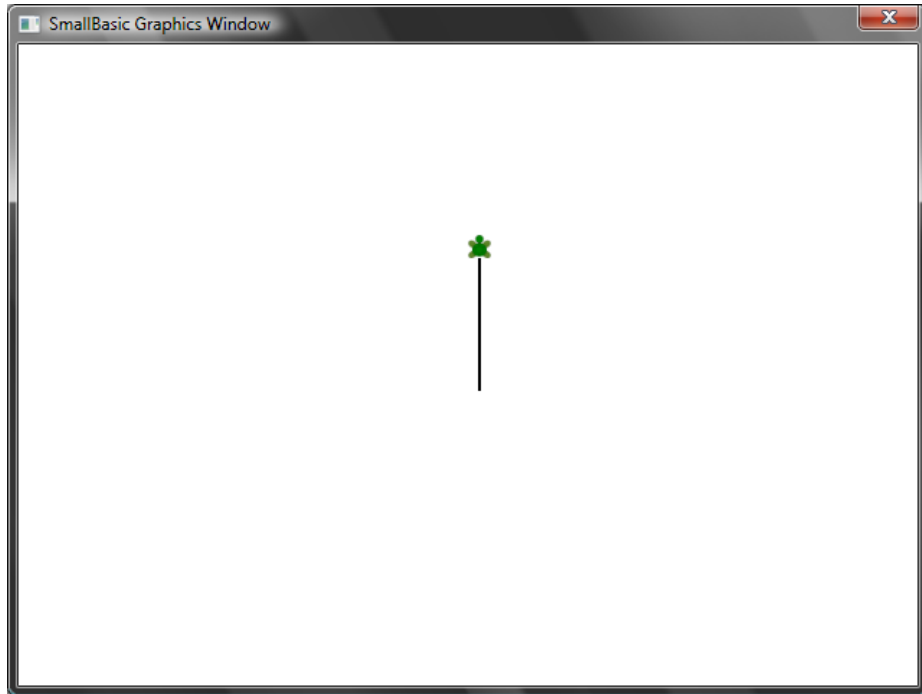


図 38 - 100ピクセル移動する

四角形を描く

四角形は4つの角と、2つの平行線および2つの垂直な線から成り立っています。四角形を描くためには、タートルにまず直線を描かせて、右に向きをかえて、また直線を描かせて、という処理を4回繰り返して終了する必要があります。これをプログラムに置き換えると、以下のようになります。

```
Turtle.Move(100)
Turtle.TurnRight()
Turtle.Move(100)
Turtle.TurnRight()
Turtle.Move(100)
Turtle.TurnRight()
Turtle.Move(100)
Turtle.TurnRight()
```

このプログラムを実行すると、タートルは四角形を描き始め、1本の直線を一回に描いて、最終的に以下のような結果になるはずですが、

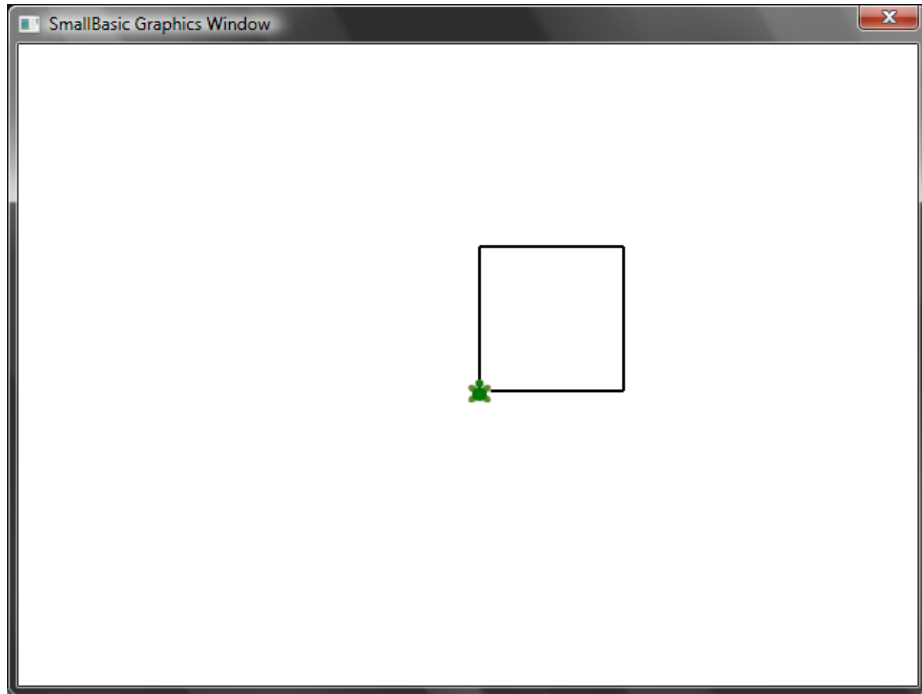


図 39 - タートルが四角形を描写する

この処理で気がつく興味深いところは、まったく同じ2つの命令を正確に4回繰り返していることです。そして我々はすでにこのようなループ処理の命令を学んでいます。そこで、このプログラムを `For..EndFor` を使ったループ処理を使って改良すると、大変シンプルなプログラムに落ち着きます。

```
For i = 1 To 4
  Turtle.Move(100)
  Turtle.TurnRight()
EndFor
```

色の変更

タートルは前の章で見たものと全く同じ `GraphicsWindow` の上で描写処理をします。つまり、前の章で学んだすべての操作は、ここでも有効であるということです。例えば、以下のプログラムは各辺を異なる色で四角形を描きます。

```
For i = 1 To 4
  GraphicsWindow.PenColor = GraphicsWindow.GetRandomColor()
  Turtle.Move(100)
  Turtle.TurnRight()
EndFor
```

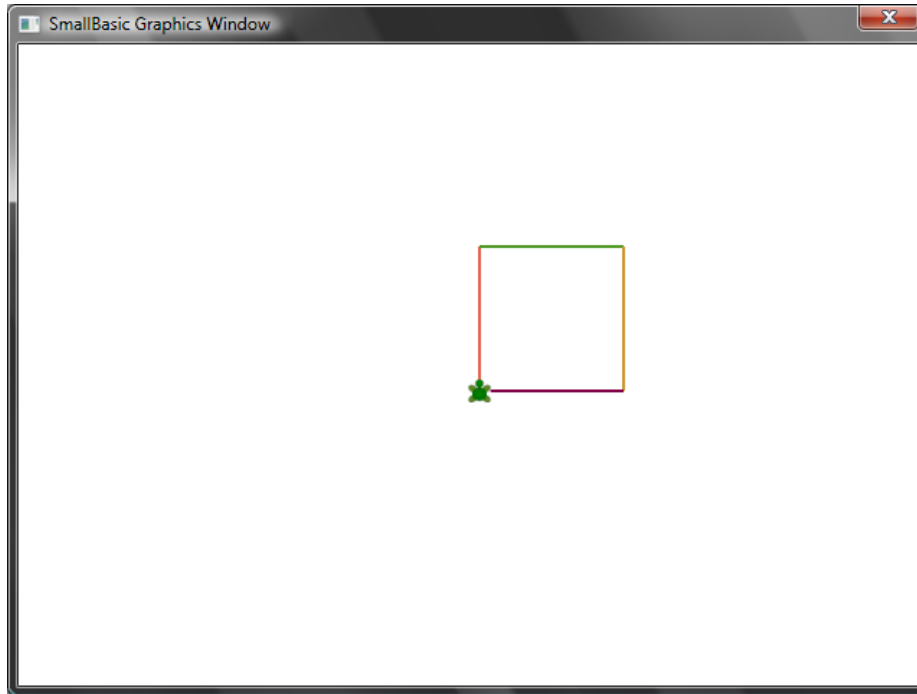


図 40 - 色を変更する

より複雑な図形を描く

タートルは、**TurnRight** 命令と **TurnLeft** 命令に加えて、**Turn** 命令を持っています。この命令は、回転角度を指定する入力値をとります。この命令を使用することにより、あらゆる角数の多角形を描くことができます。以下のプログラムは六角形を描きます。

```
For i = 1 To 6
  Turtle.Move(100)
  Turtle.Turn(60)
EndFor
```

本当にこのプログラムが六角形を描くかどうか見てみましょう。六角形の辺と辺の間の角度は 60 度であることから、**Turn(60)** を使います。このような、すべての角が同じ角度であるような多角形においては、それぞれの角の角度は 360 をその多角形の角数で割ることで簡単に算出できます。この情報を元に、その値を使って、より一般的な多角形を描くためのプログラムを書くことができます。

```
sides = 12
length = 400 / sides
angle = 360 / sides
For i = 1 To sides
  Turtle.Move(length)
  Turtle.Turn(angle)
```


EndFor

このプログラムを使用することで、あらゆる角数の多角形を、単に **sides** の値を変更することで描くことができます。値として 4 を入れれば四角形を描くでしょう。非常に大きな値、例えば 50 を入れれば結果として円と区別つかないようなものになるでしょう。

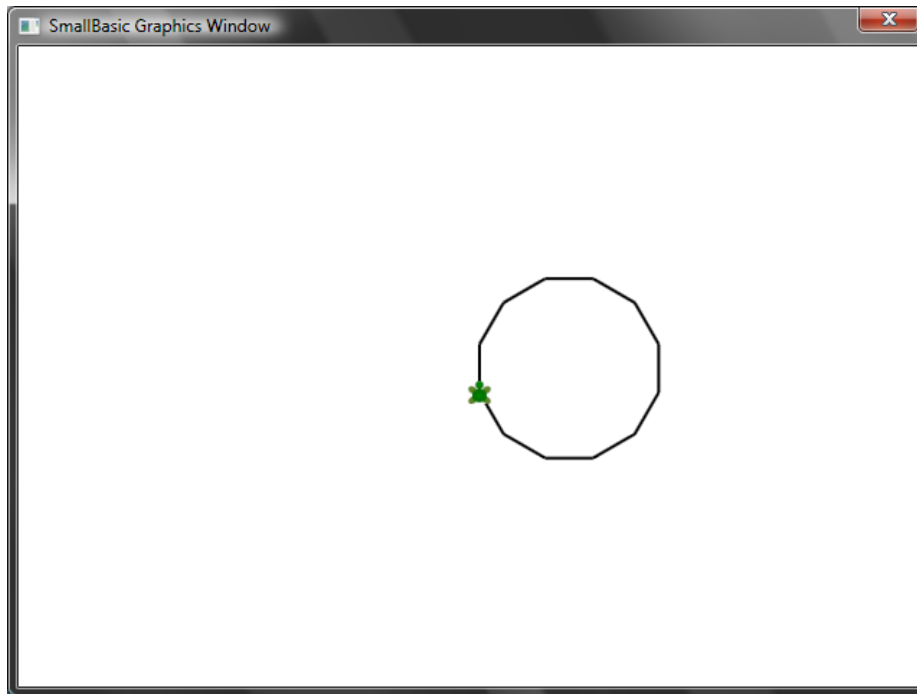


図 41 - 12 角形を描く

これまでに学んだ技術を使用して、タートルに複数の円を少しずつずらして描かせることで、興味深い出力を得ることができます。

```
sides = 50
length = 400 / sides
angle = 360 / sides
Turtle.Speed = 9
For j = 1 To 20
  For i = 1 To sides
    Turtle.Move(length)
    Turtle.Turn(angle)
  EndFor
  Turtle.Turn(18)
EndFor
```

このプログラムは2つの For..EndFor ループ処理を使用して、ひとつのループ処理がもう一つのループ処理を包括している形になっています。内側のループ処理 ($i = 1$ to sides) は多角形を描写するのと同様な処理をし、円を描く役割を持っています。外側のループ処理 ($j = 1$ to 20) は、タートルを一つの円を描くごとに少しずつずらしていき役割を持っています。ここではタートルに 20 個の円を描かせています。これを合わせて実行することで、このプログラムは非常に興味深い、以下のようなパターンを表示します。

上のプログラムでは、タートルを速く動かすために Speed を 9 に設定しています。この値は 1 から 10 までの任意の値に設定することができます。

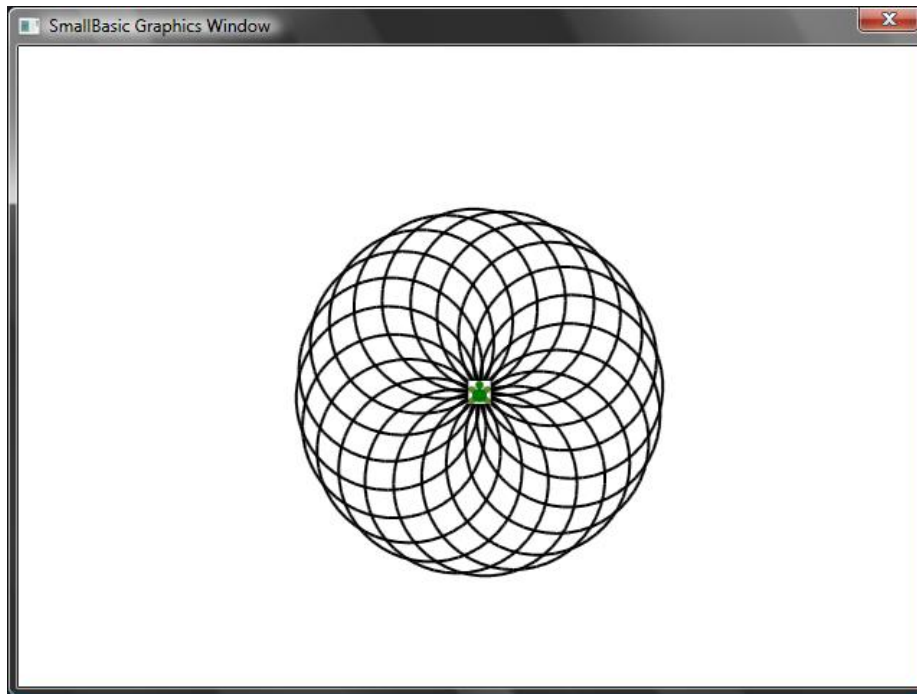


図 42 -円の中を描く

様々な移動をする

PenUp 命令を呼び出すことで、タートルに何も描かせないようにすることができます。この機能を使うことで、タートルを画面上の任意の位置へ、何も描くことなく移動することができます。また、PenDown 命令を呼び出すことで、タートルによる描画を再開することができます。

この機能を使用することで、例えば点線のような、興味深い効果を作ることができます。以下のプログラムはその機能を使って点線の多角形を描くものです。

```
sides = 6
length = 400 / sides
angle = 360 / sides
For i = 1 To sides
  For j = 1 To 6
    Turtle.Move(length / 12)
    Turtle.PenUp()
    Turtle.Move(length / 12)
    Turtle.PenDown()
  EndFor
  Turtle.Turn(angle)
EndFor
```

このプログラムも 2 重ループ処理を使用しています。内側のループ処理は点を使った一本の線を描くのに使用し、外側のループ処理は何本の線を描くのかを示しています。この例では、6 を **side** の値として使用しているので、以下のように点線による六角形を描いています。

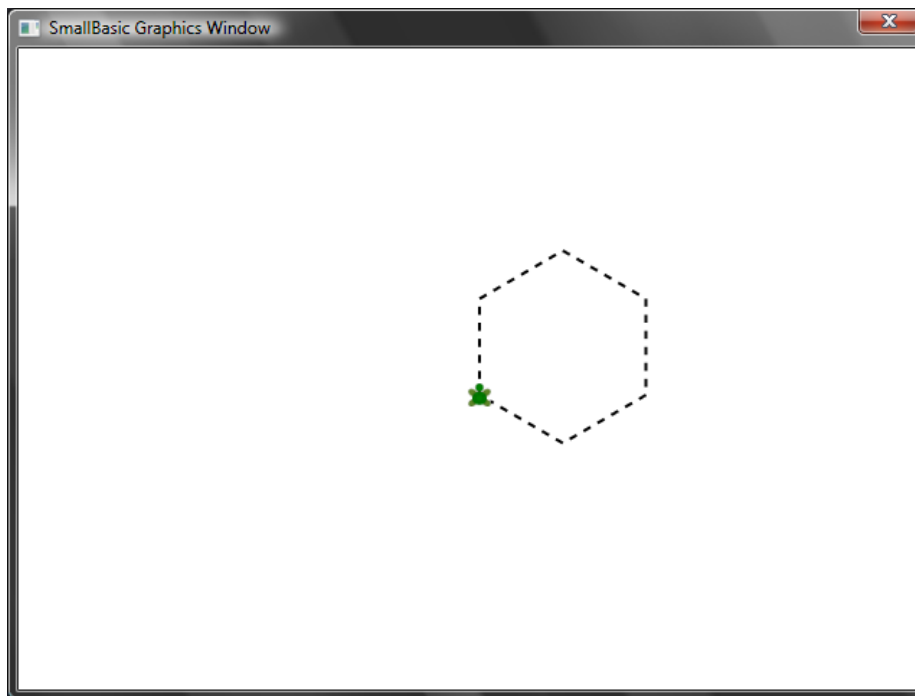


図 43 - PenUp と PenDown を使用する

サブルーチン

プログラムを作成していると、同じステップを繰り返し実行する必要がある場合があります。この場合は、何度も同じステートメントを書く代わりに、サブルーチンを使うと便利です。

サブルーチンは、大きなプログラムのコードの一部で、特定の処理を実行するときに使います。プログラムのどこからでも、呼び出すことができます。サブルーチンは、**Sub** キーワードから始まるサブルーチン名で識別され、**EndSub** キーワードで終了します。たとえば、次のスニペットは、*PrintTime* という名前のサブルーチンです。このサブルーチンでは、現在の時間が `TextWindow` に出力されます。

```
Sub PrintTime
    TextWindow.WriteLine(Clock.Time)
EndSub
```

次のプログラムでは、サブルーチンが複数箇所で呼び出されます。

```
PrintTime()
TextWindow.Write("Enter your name: ")
name = TextWindow.Read()
TextWindow.Write(name + ", the time now is: ")
PrintTime()
Sub PrintTime
    TextWindow.WriteLine(Clock.Time)
EndSub
```

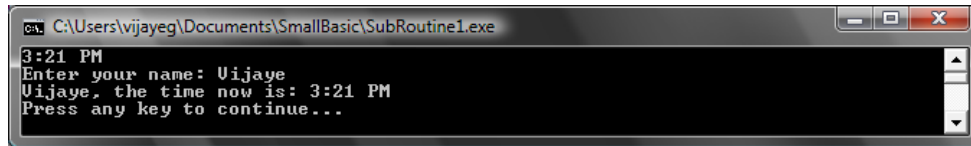


図 44 - サブルーチンの呼び出し

サブルーチンを実行するには、`SubroutineName()` を呼び出します。区切り記号 “()” は必須です。

サブルーチンの利点

サブルーチンを使うことによって、コードの量を減らすことができます。`PrintTime` サブルーチンを一度書くと、プログラムのどこからでも呼び出すことができ、現在の時間を出力できます。

また、サブルーチンを使って、複雑な問題を単純な問題に分解することができます。たとえば、複雑な数式を解く場合、単純な式に分割し、複数のサブルーチンを作って単純な式を解きます。それらの解を統合し、複雑な数式を解きます。

また、サブルーチンは、プログラムを読みやすくするのに役立ちます。よく使われるコード部分に、適切な名前を持つサブルーチンを使えば、読みやすくして、理解しやすいプログラムになります。ほかの人が書いたプログラムを理解したり、自分のプログラムを見直すときに、非常に便利です。

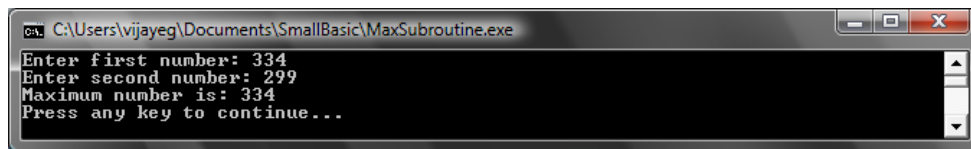
SmallBasic のサブルーチンは、同じプログラム内でのみ、呼び出すことができます。ほかのプログラムからサブルーチンを呼び出すことはできません。

変数の使用

プログラムの中の変数は、サブルーチンからアクセスして使用できます。たとえば、次のプログラムでは、入力された 2 つの数のうちの大きい方が出力されます。変数 `max` は、サブルーチンの中と外の両方で、使われます。

```
TextWindow.Write("Enter first number: ")
num1 = TextWindow.ReadNumber()
TextWindow.Write("Enter second number: ")
num2 = TextWindow.ReadNumber()
FindMax()
TextWindow.WriteLine("Maximum number is: " + max)
Sub FindMax
  If (num1 > num2) Then
    max = num1
  Else
    max = num2
  EndIf
EndSub
```

このプログラムの出力結果は、以下のとおりです。



```
C:\Users\vijayeg\Documents\SmallBasic\MaxSubroutine.exe
Enter first number: 334
Enter second number: 299
Maximum number is: 334
Press any key to continue...
```

図 45 - サブルーチンからの変数の使用

次の例でも、サブルーチンの使い方について解説します。このグラフィック プログラムでは、変数 x と y に格納される多数の点が計算されます。次に、 x と y を中心点に使う円を描くサブルーチン `DrawCircleUsingCenter` が呼び出されます。

```
GraphicsWindow.BackgroundColor = "Black"
GraphicsWindow.PenColor = "LightBlue"
GraphicsWindow.Width = 480
For i = 0 To 6.4 Step 0.17
    x = Math.Sin(i) * 100 + 200
    y = Math.Cos(i) * 100 + 200

    DrawCircleUsingCenter()
EndFor
Sub DrawCircleUsingCenter
    startX = x - 40
    startY = y - 40

    GraphicsWindow.DrawEllipse(startX, startY, 120, 120)
EndSub
```

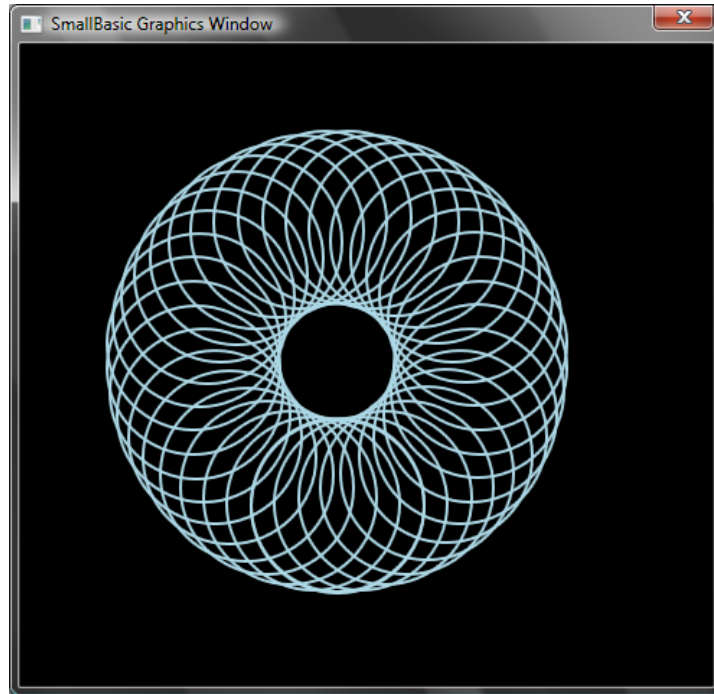


図 46 - グラフィックの例

ループ内でのサブルーチンの呼び出し

ループ内でサブルーチンを呼び出すこともできます。その間、同じステートメントが繰り返し実行されますが、使われる変数の値は、ループが実行されるたびに異なります。次の例は、値が素数であるかどうかを判別する、*PrimeCheck* という名前のサブルーチンです。このサブルーチンを使うと、ユーザーが入力した値が素数であるかどうか、判別するプログラムを作ることができます。

```
TextWindow.Write("Enter a number: ")
i = TextWindow.ReadNumber()
isPrime = "True"
PrimeCheck()
If (isPrime = "True") Then
    TextWindow.WriteLine(i + " is a prime number")
Else
    TextWindow.WriteLine(i + " is not a prime number")
EndIf
Sub PrimeCheck
    For j = 2 To Math.SquareRoot(i)
        If (Math.Remainder(i, j) = 0) Then
            isPrime = "False"
            Goto EndLoop
        EndIf
    Endfor
EndSub
```

```
EndLoop:  
EndSub
```

PrimeCheck サブルーチンでは、 i の値が除算されます。 i の値を除算して剰余がない場合、 i の値は素数ではありません。このとき、 $isPrime$ の値が“False”になり、サブルーチンが終了します。割り切れない数の場合は、 $isPrime$ の値は“True”のままです。

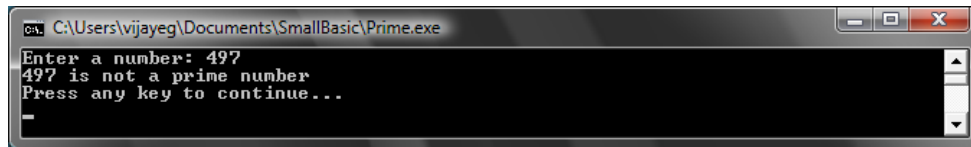
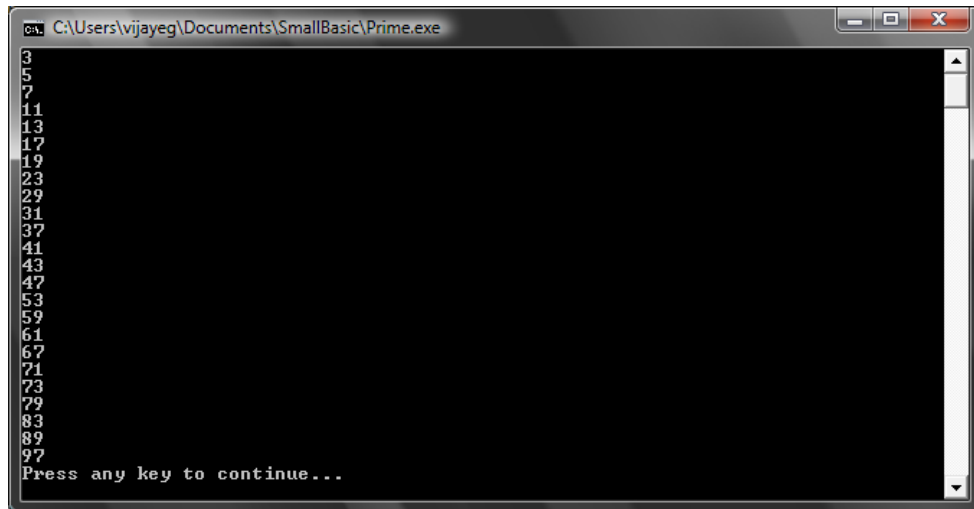


図 47 - 素数チェック

次の例では、PrimeCheck サブルーチンを使って、100 以下の素数をすべて表示します。ループ内で PrimeCheck サブルーチンが呼び出されるので、ループが実行されるたびに、PrimeCheck サブルーチンでは異なる値が計算されます。

```
For i = 3 To 100  
  isPrime = "True"  
  PrimeCheck()  
  If (isPrime = "True") Then  
    TextWindow.WriteLine(i)  
  EndIf  
EndFor  
Sub PrimeCheck  
  For j = 2 To Math.SquareRoot(i)  
    If (Math.Remainder(i, j) = 0) Then  
      isPrime = "False"  
      Goto EndLoop  
    EndIf  
  Endfor  
EndLoop:  
EndSub
```

このプログラムでは、ループが実行されるたびに i の値が変わります。ループ内で PrimeCheck サブルーチンが呼び出されると、PrimeCheck サブルーチンによって、変数 i が素数かどうか計算されます。この結果は、変数 $isPrime$ に格納され、サブルーチンの外のループからアクセスされます。 i の値が素数であれば、その値が出力されます。ループは、3 から開始されて 100 まで繰り返されるので、3 から 100 の間の素数がすべて表示されます。このプログラムの出力結果は、以下のとおりです。



```
C:\Users\vijayeg\Documents\SmallBasic\Prime.exe
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
Press any key to continue...
```

图 48 - 素数

第 10 章 配列

そろそろ変数の使い方にも慣れてきたことと思います。この章まで進めたということは、楽しんで学んでいるということですよね？

ここで変数を使って書いた最初のプログラムのおさらいをしてみましょう。

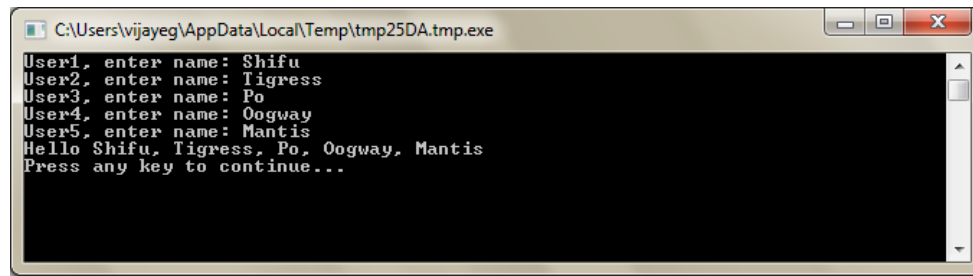
```
TextWindow.Write("Enter your Name: ")
name = TextWindow.Read()
TextWindow.WriteLine("Hello " + name)
```

このプログラムでは、ユーザーが入力した名前を **name** という変数に保存しました。そして、ユーザーの名前と "Hello" を表示します。さて、ここで複数のユーザー、たとえば 5 人のユーザーがいるとします。どうやって 5 人全部の名前を保存しますか？ ひとつの方法は以下のようになります：

```
TextWindow.Write("User1, enter name: ")
name1 = TextWindow.Read()
TextWindow.Write("User2, enter name: ")
name2 = TextWindow.Read()
TextWindow.Write("User3, enter name: ")
name3 = TextWindow.Read()
TextWindow.Write("User4, enter name: ")
name4 = TextWindow.Read()
TextWindow.Write("User5, enter name: ")
name5 = TextWindow.Read()
TextWindow.WriteLine("Hello ")
```

```
TextWindow.Write(name1 + ", ")
TextWindow.Write(name2 + ", ")
TextWindow.Write(name3 + ", ")
TextWindow.Write(name4 + ", ")
TextWindow.WriteLine(name5)
```

これを実行すると、次の結果が表示されます:



```
C:\Users\vijayeg\AppData\Local\Temp\tmp25DA.tmp.exe
User1, enter name: Shifu
User2, enter name: Tigress
User3, enter name: Po
User4, enter name: Oogway
User5, enter name: Mantis
Hello Shifu, Tigress, Po, Oogway, Mantis
Press any key to continue...
```

図 49 - 配列を使っていない例

そのような簡単なプログラムを書くにはもっとよい方法があるはずです。特にコンピュータはタスクを繰り返すことが得意なので、各ユーザーのために同じコードを何回も書く必要はないかも知れませんね。ここでのコツは、ひとつの変数を使って複数のユーザーの名前を保存し取り出すことです。同じ変数を使うことができれば、以前の章で学んだ **For** ループを使うことができます。ここで配列の出番です。

配列とは?

配列とは、一度に複数の値を持つことができる特別な種類の変数です。つまり、5人のユーザー名を保存するために `name1`、`name2`、`name3`、`name4` および `name5` を使わずに、5人すべてのユーザー名を `name` だけを使って保存できます。”インデックス”と呼ばれるものを使用して、複数の値を保存します。例えば、`name[1]`、`name[2]`、`name[3]`、`name[4]` および `name[5]` はそれぞれすべて値を保存できます。数字の 1、2、3、4 および 5 は配列のインデックスと呼ばれます。

`name[1]`、`name[2]`、`name[3]`、`name[4]` および `name[5]` はすべて異なる変数のように見えますが、実際はすべてひとつの変数なのです。そのどこが利点なのかと不思議に思うかもしれませんが、配列に値を保存することの最大の利点は、別の変数を使用してインデックスを指定できるところで、ループ内の配列に簡単にアクセスできるようになります。

さて、ここで今学んだ知識を、配列を使った前のプログラムに活かすとどうなるか見てみましょう。

```
For i = 1 To 5
    TextWindow.Write("User" + i + ", enter name: ")
    name[i] = TextWindow.Read()
EndFor
TextWindow.Write("Hello ")
For i = 1 To 5
```

```
TextWindow.Write(name[i] + ", ")
EndFor
TextWindow.WriteLine("")
```

かなり読みやすくなったと思いませんか? 太字の行に注目してください。最初の行は配列内の値を保存し、2 番目の行は配列からの値を読み出します。name[1] に保存する値は、name[2] に保存する値の影響を受けません。そのため、ほとんどの場合、name[1] と name[2] を同じ ID を持つ 2 つの異なる変数として扱うことができます。



```
C:\Users\vijayeg\AppData\Local\Temp\tmp8426.tmp.exe
User1, enter name: Shifu
User2, enter name: Tigress
User3, enter name: Po
User4, enter name: Oogway
User5, enter name: Mantis
Hello Shifu, Tigress, Po, Oogway, Mantis,
Press any key to continue...
```

図 50 - 配列を使った例

上記のプログラムは、配列なしのプログラムとほとんど同じ結果 (Mantis の後のカンマのみが異なります) をもたします。出力のループを以下のように書き直すと、その違いはなくなります:

```
TextWindow.Write("Hello ")
For i = 1 To 5
  TextWindow.Write(name[i])
  If i < 5 Then
    TextWindow.Write(", ")
  EndIf
EndFor
TextWindow.WriteLine("")
```

配列のインデックス

前述のプログラムでは、数字をインデックスとして使用して保存し、配列からの値を取り出しました。インデックスは数字のみに制限されず、実際には、テキストもインデックスに使えることがわかりました。例えば、次のプログラムは、ユーザーについてのいろいろな情報を聞いて保存してから、ユーザーが求める情報を出力します。

```
TextWindow.Write("Enter name: ")
user["name"] = TextWindow.Read()
TextWindow.Write("Enter age: ")
user["age"] = TextWindow.Read()
```

```
TextWindow.Write("Enter city: ")
user[" city"] = TextWindow.Read()
TextWindow.Write("Enter zip: ")
user[" zip"] = TextWindow.Read()
TextWindow.Write("What info do you want? ")
index = TextWindow.Read()
TextWindow.WriteLine(index + " = " + user[index])
```

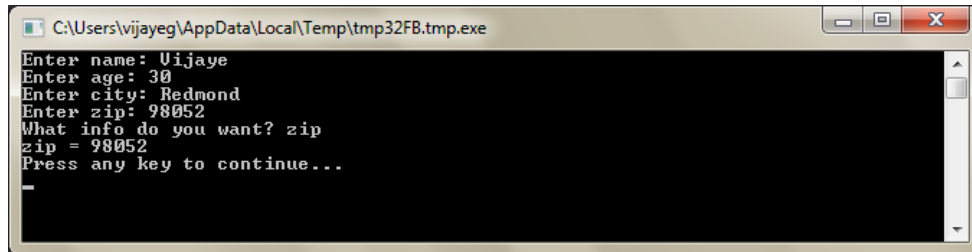


図 51 - 数字でないインデックスの使用

複数の次元

例えば、あなたの友人すべての名前と電話番号を保存して、必要な時に電話帳を使った場合と同じように電話番号を探し出したいとします。そんなプログラムをどうやって書きますか？

この場合、2つのセットのインデックス（配列の次元とも言われます）が必要です。各友人をそのニックネームで識別するとします。ニックネームは配列の最初のインデックスになります。最初のインデックスを友人の変数のために使用した後は、2番目のインデックス `name` と `phone number` が友人の本当の名前と電話番号を得るために使われます。

このデータを保存する方法は次のようになります：

```
friends["Rob"]["Name"] = "Robert"
friends["Rob"]["Phone"] = "555-6789"
friends["VJ"]["Name"] = "Vijaye"
friends["VJ"]["Phone"] = "555-4567"
friends["Ash"]["Name"] = "Ashley"
friends["Ash"]["Phone"] = "555-2345"
```

同一の配列 `friends` に2つのインデックスがあるので、この配列は2次元の配列と呼ばれます。

いったんこのプログラムをセットアップすると、友人のニックネームを入力すると、その友人の情報を出力できます。以下がそれを行うプログラムです：

```
friends["Rob"]["Name"] = "Robert"
```

```

friends["Rob"]["Phone"] = "555-6789"
friends["VJ"]["Name"] = "Vijaye"
friends["VJ"]["Phone"] = "555-4567"
friends["Ash"]["Name"] = "Ashley"
friends["Ash"]["Phone"] = "555-2345"
TextWindow.Write("Enter the nickname: ")
nickname = TextWindow.Read()
TextWindow.WriteLine("Name: " + friends[nickname]["Name"])
TextWindow.WriteLine("Phone: " + friends[nickname]["Phone"])

```



図 52 - 簡単な電話帳

グリッドを表すための配列の使用

複数の次元の配列では、グリッド/テーブルがよく使われます。グリッドには行と列があり、2次元の配列にぴったり合います。ボックスをグリッドに配置する簡単なプログラムが以下に示されています：

```

rows = 8
columns = 8
size = 40
For r = 1 To rows
  For c = 1 To columns
    GraphicsWindow.BrushColor = GraphicsWindow.GetRandomColor()
    boxes[r][c] = Shapes.AddRectangle(size, size)
    Shapes.Move(boxes[r][c], c * size, r * size)
  EndFor
EndFor

```

このプログラムは、ボックスを追加して 8x8 のグリッドの形に配置します。このプログラムは、これらのボックスを配置した後で、配列にボックスを保存します。こうすることにより、これらのボックスを保存して、必要な時に再使用することが簡単になります。

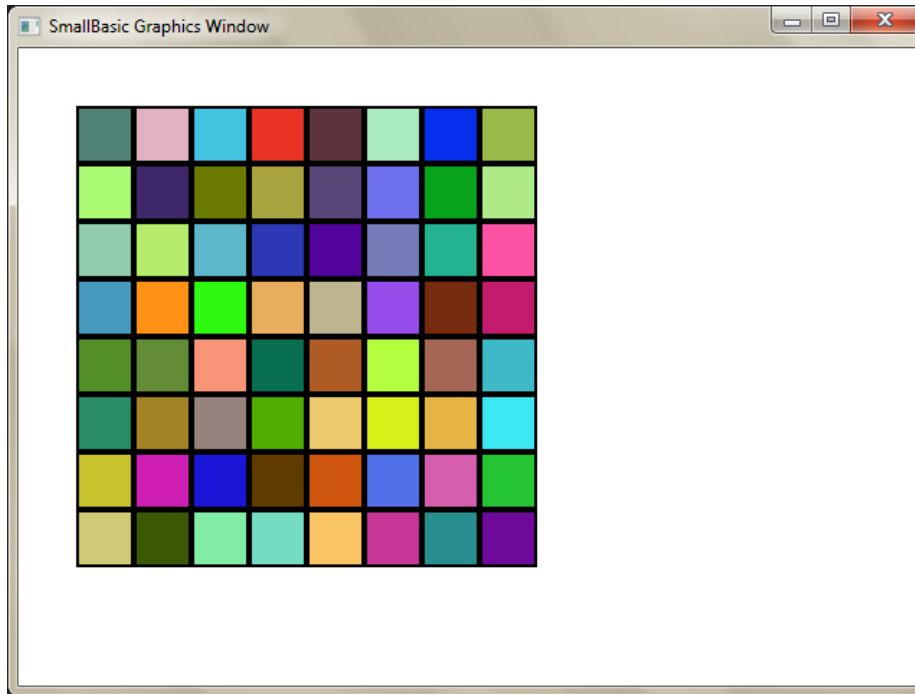


図 53 - グリッドにボックスを配置

例えば、以前のプログラムの最後に次のコードを追加すると、ボックスが左上の角に動いていく様子が見えます。

```
For r = 1 To rows
  For c = 1 To columns
    Shapes.Animate(boxes[r][c], 0, 0, 1000)
    Program.Delay(300)
  EndFor
EndFor
```

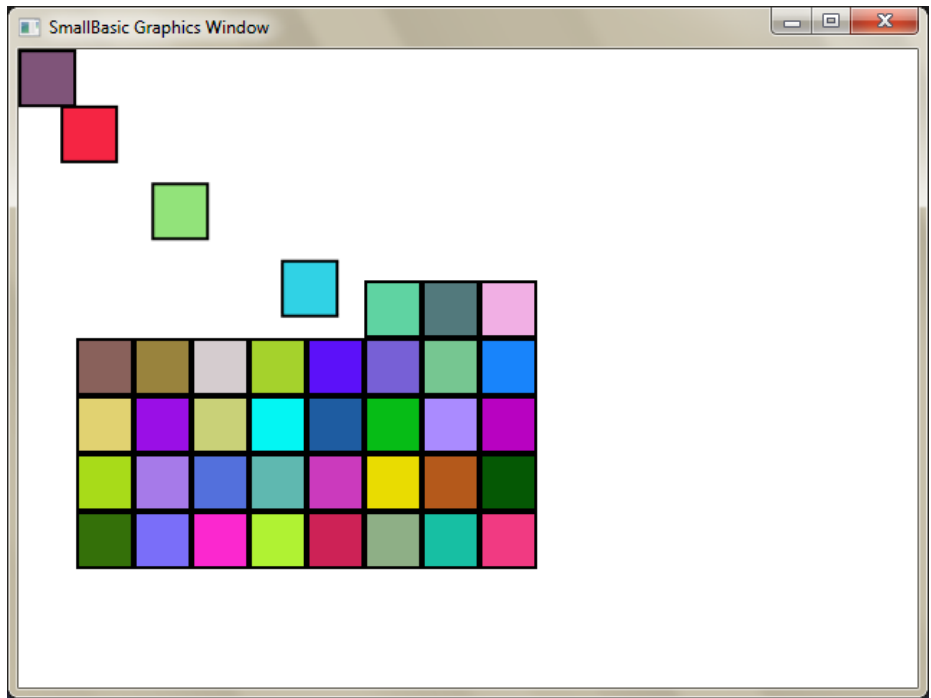


図 54 - グリッド内のボックスの軌跡

イベントと対話

最初の 2 章でプロパティとオペレーションを持つオブジェクトについて学びました。プロパティとオペレーションに加え、いくつかのオブジェクトはイベントと呼ばれる機能を持っています。イベントは信号のようなもので、例えば、マウスを動かしたりクリックしたりするような、ユーザーのアクションに対する応答です。ある意味でイベントは演算？処理の反対とも言えます。演算？処理の場合は、プログラマーであるあなたはコンピューターに何かの処理をさせるためにそれを呼び出しますが、イベントの場合には、何か興味深い事が起きた場合にコンピューターがあなたにその事を伝えます。

イベントはどのくらい便利なのでしょう？

イベントは対話型プログラムの中心です。もしユーザーがプログラムと対話できるようにしたい場合には、イベントを使います。たとえば、三目並べ(Tic-Tac-Toe) ゲームを作るとしましょう。ユーザーが、自分で選択をしてプレイできるようにしたいと思います。そこがイベントを使うところです - イベントを使って、プログラム内でユーザーからの入力を受け取ります。少し難しく感じるかもしれませんが、心配ご無用、次の簡単な例は、イベントとは何か、どのように使うのかを理解する手助けとなるでしょう。

以下は、一つのステートメントと一つのサブルーチンがあるだけの非常に簡単なプログラムです。サブルーチンは *ShowMessage* オペレーションを使い GraphicsWindow オブジェクト内にメッセージボックスを表示します。

```
GraphicsWindow.MouseDown = OnMouseDown
Sub OnMouseDown
    GraphicsWindow.ShowMessage("You Clicked.", "Hello")
EndSub
```

上のプログラムの中で興味深い部分は、GraphicsWindow オブジェクトのサブルーチンの名前を **MouseDown** に割り当てている行です。MouseDown はまるでプロパティのようですね - ただしプロパティに値を割り当てるかわりに、*OnMouseDown* をサブルーチンを割り当てています。そこがイベントの特別な所です - イベントが発生すると

、サブルーチンが自動的に呼び出されます。この例では、ユーザーがマウスをクリックするたびに、サブルーチン *OnMouseDown* が GraphicsWindow オブジェクト上で呼び出されます。早速プログラムを実行してみましょう。GraphicsWindow をマウスでクリックするたびに、以下のようなメッセージボックスが表示されます。



図 55 -イベントへの応答

このような種類のイベント処理はとても強力で、創造的かつおもしろいプログラムを作成することができます。このような形式で書かれたプログラムの事を、イベント駆動型プログラムと呼ばれています。

OnMouseDown サブルーチンをメッセージ ボックスを表示する以外の処理に変えることもできます。例えば、下のプログラムのようにすると、ユーザーがマウスをクリックした場所に大きな青い点を描きます。

```
GraphicsWindow.BrushColor = "Blue"  
GraphicsWindow.MouseDown = OnMouseDown  
Sub OnMouseDown  
    x = GraphicsWindow.MouseX - 10  
    y = GraphicsWindow.MouseY - 10  
    GraphicsWindow.FillEllipse(x, y, 20, 20)  
EndSub
```

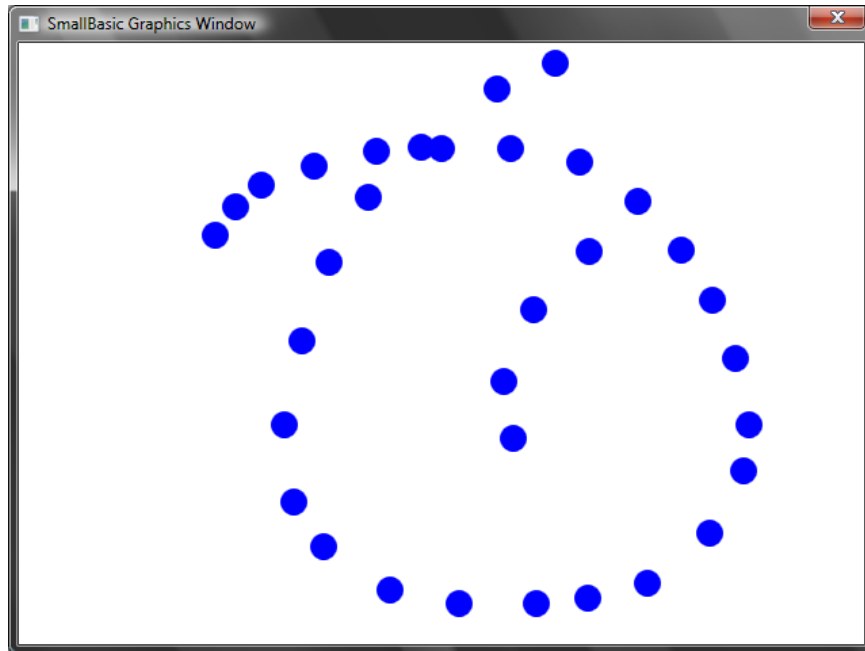


図 56 - マウス ダウン イベント処理

上記のプログラム内で、*MouseX*と *MouseY*を使ってマウスの座標を取得していることに気づいたと思います。そして、取得したマウスの座標を中心とした円を描くのに使っています。

複数のイベント処理

複数のイベント処理には特に数の制限はありません。一つのサブルーチンで複数のイベントを処理することもできます。が、しかし、一つのイベントは一度しか処理できません。もし同じイベントに対して二つのサブルーチンを割り当てると、二つ目のサブルーチンが処理されます。

実際にどうなるかを見るために、先ほどの例にキーが押された時の処理を持つサブルーチンを追加してみましょう。ついでに、マウスをクリックした時に円を描くブラシの色を変えてみましょう、こうするとマウスをクリックすると違う色の点が描画されます。

```
GraphicsWindow.BrushColor = "Blue"  
GraphicsWindow.MouseDown = OnMouseDown  
GraphicsWindow.KeyDown = OnKeyDown  
Sub OnKeyDown  
    GraphicsWindow.BrushColor = GraphicsWindow.GetRandomColor()  
EndSub  
Sub OnMouseDown  
    x = GraphicsWindow.MouseX - 10  
    y = GraphicsWindow.MouseY - 10  
    GraphicsWindow.FillEllipse(x, y, 20, 20)  
EndSub
```

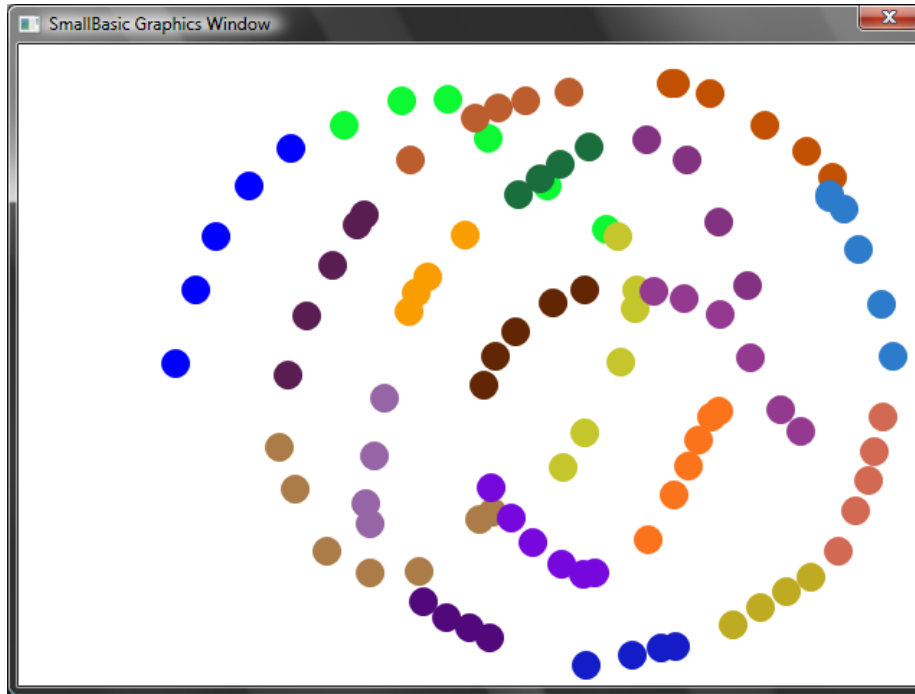


図 57 - 複数のイベント処理

このプログラムを実行して、マウスをウィンドウ上でクリックすると、青い点が描画されます。ここで、任意のキーを押してからマウスをクリックしてみると、違う色の点が描画されます。何が起きているのかというと、キーを押した時に実行されるサブルーチン *OnKeyDown* によって、ブラシの色がランダムに変わっているのです。キーを押した後にマウスをクリックすると、新しくランダムに設定された色の円が描かれるというわけです。

ペイントプログラム

イベントやサブルーチンを駆使して、ユーザーがウィンドウ上に描画できるプログラムを書くことができます。しかも、問題を細かく見て解決していくことで、驚くほど簡単にそういうプログラムを書けるのです。最初のステップとして、ユーザーがグラフィック ウィンドウ上でマウスを動かすことができ、またマウスの軌跡が残るようなプログラムを書いてみましょう。

```
GraphicsWindow.MouseMove = OnMouseMove
Sub OnMouseMove
  x = GraphicsWindow.MouseX
  y = GraphicsWindow.MouseY
  GraphicsWindow.DrawLine(prevX, prevY, x, y)
  prevX = x
  prevY = y
EndSub
```

このプログラムを実行すると、最初の行は常にウィンドウの左上隅の (0,0) の位置から始まってしまいます。この問題は *MouseDown* イベント処理で *prevX* および *prevY* の値を得ることによって解決することができます。

また、必要なのはユーザーがマウスボタンを押した時の軌跡のみです。その他の時には線を描く必要はありません。このような動作をするためには、**Mouse** オブジェクトの *IsLeftButtonDown* プロパティを使用します。このプロパティは左のボタンが押されているかどうかを知らせます。もしこのプロパティの値が `true` だったら線を描き、そうでない場合には線は描きません。

```
GraphicsWindow.MouseMove = OnMouseMove
GraphicsWindow.MouseDown = OnMouseDown
Sub OnMouseDown
    prevX = GraphicsWindow.MouseX
    prevY = GraphicsWindow.MouseY
EndSub
Sub OnMouseMove
    x = GraphicsWindow.MouseX
    y = GraphicsWindow.MouseY
    If (Mouse.IsLeftButtonDown) Then
        GraphicsWindow.DrawLine(prevX, prevY, x, y)
    EndIf
    prevX = x
    prevY = y
EndSub
```

タートル フラクタル

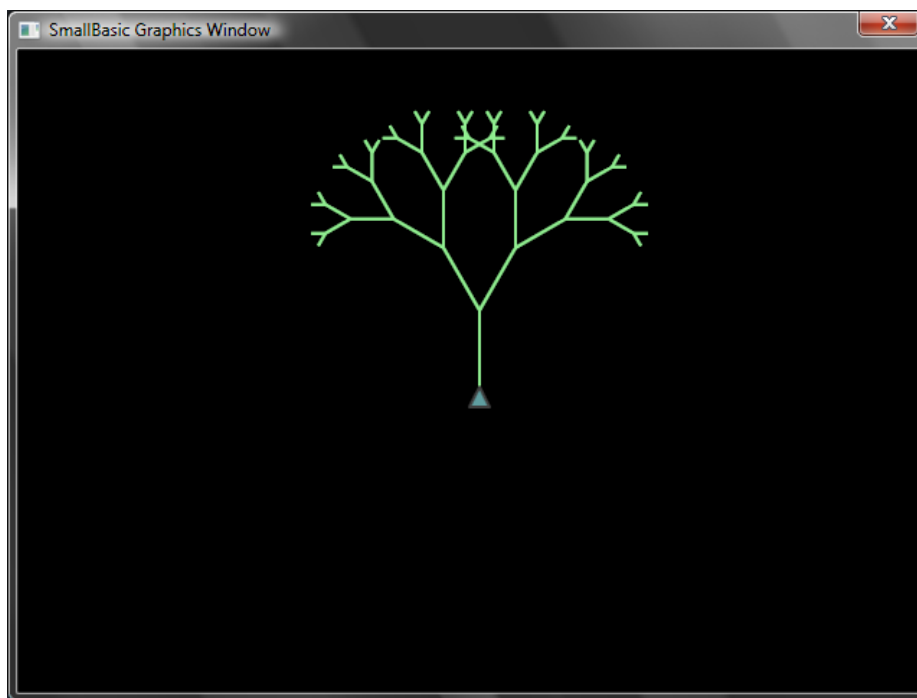


図 58 - タートルがツリー フラクタルを描いているところ

```
angle = 30  
delta = 10  
distance = 60  
Turtle.Speed = 9
```

```
GraphicsWindow.BackgroundColor = "Black"  
GraphicsWindow.PenColor = "LightGreen"  
DrawTree()
```

```
Sub DrawTree
```

```
  If (distance > 0) Then
```

```
    Turtle.Move(distance)
```

```
    Turtle.Turn(angle)
```

```
    Stack.PushValue("distance", distance)
```

```
    distance = distance - delta
```

```
    DrawTree()
```

```
    Turtle.Turn(-angle * 2)
```

```
    DrawTree()
```

```
    Turtle.Turn(angle)
```

```
    distance = Stack.PopValue("distance")
```

```
  Turtle.Move(-distance)
```

```
EndIf
```

```
EndSub
```

Flickr の写真

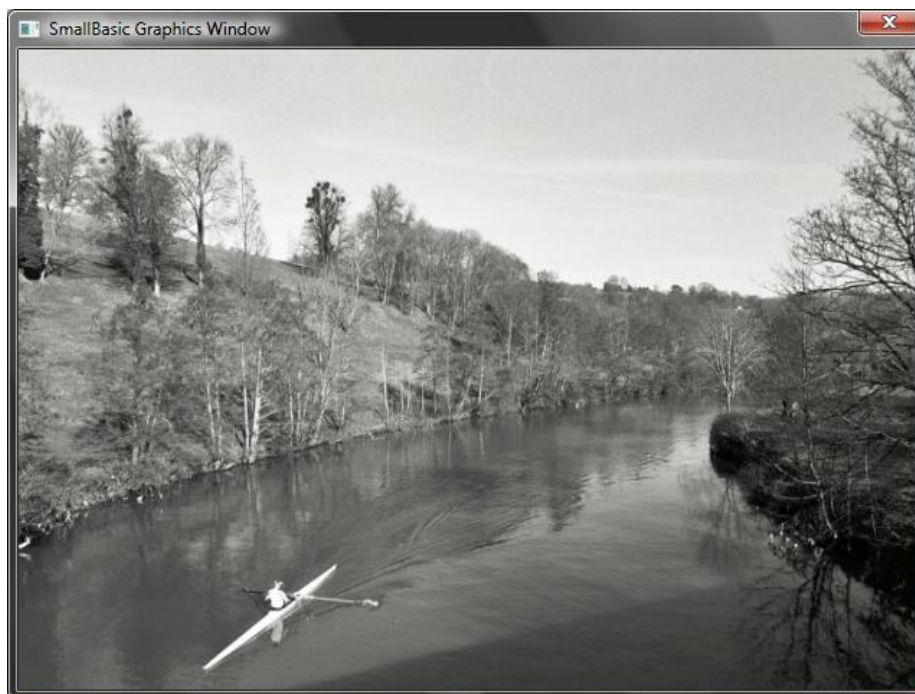


図 59 - Flickr から写真を読み込みます

```
GraphicsWindow.BackgroundColor = "Black"  
GraphicsWindow.MouseDown = OnMouseDown  
  
Sub OnMouseDown  
    pic = Flickr.GetRandomPicture(" mountains, river")  
    GraphicsWindow.DrawResizedImage(pic, 0, 0, 640, 480)  
EndSub
```

ダイナミック デスクトップ壁紙

```
For i = 1 To 10  
    pic = Flickr.GetRandomPicture(" mountains")  
    Desktop.SetWallPaper(pic)  
    Program.Delay(10000)  
EndFor
```

パドル ゲーム

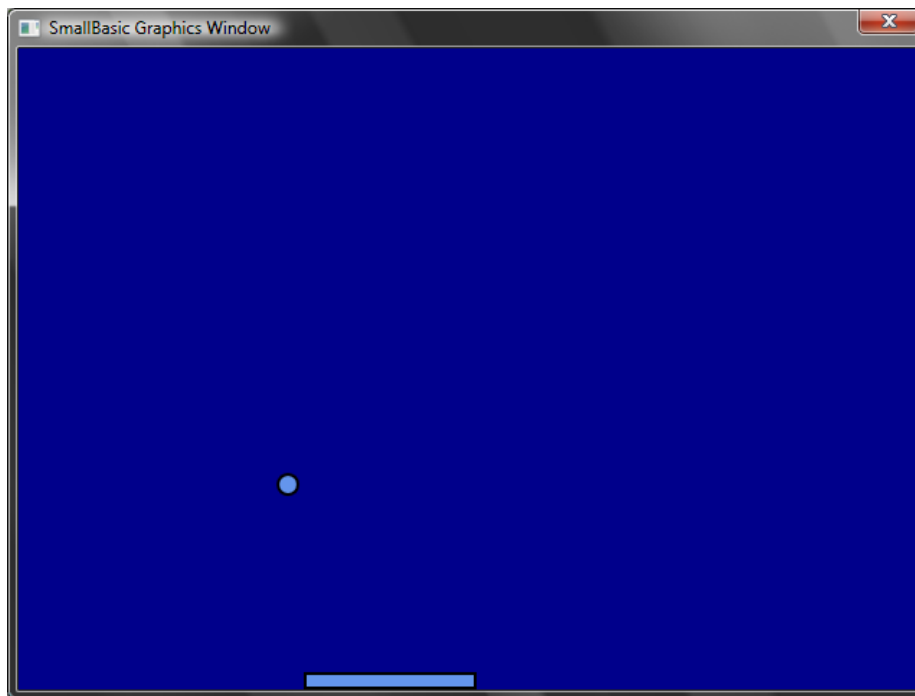


図 60 - パドル ゲーム

```
GraphicsWindow.BackgroundColor = "DarkBlue"  
paddle = Shapes.AddRectangle(120, 12)
```



```

ball = Shapes.AddEllipse(16, 16)
GraphicsWindow.MouseMove = OnMouseMove

x = 0
y = 0
deltaX = 1
deltaY = 1

RunLoop:
  x = x + deltaX
  y = y + deltaY

  gw = GraphicsWindow.Width
  gh = GraphicsWindow.Height
  If (x >= gw - 16 or x <= 0) Then
    deltaX = -deltaX
  EndIf
  If (y <= 0) Then
    deltaY = -deltaY
  EndIf

  padX = Shapes.GetLeft (paddle)
  If (y = gh - 28 and x >= padX and x <= padX + 120) Then
    deltaY = -deltaY
  EndIf

  Shapes.Move(ball, x, y)
  Program.Delay(5)

  If (y < gh) Then
    Goto RunLoop
  EndIf

GraphicsWindow.ShowMessage("You Lose", "Paddle")

Sub OnMouseMove
  paddleX = GraphicsWindow.MouseX
  Shapes.Move(paddle, paddleX - 60, GraphicsWindow.Height - 12)
EndSub

```

これは基本色調 (hue) 別に分類された Small Basic でサポートされている色のリストです。

Red Colors

IndianRed	#CD5C5C
LightCoral	#F08080
Salmon	#FA8072
DarkSalmon	#E9967A
LightSalmon	#FFA07A
Crimson	#DC143C
Red	#FF0000
FireBrick	#B22222
DarkRed	#8B0000

Pink Colors

Pink	#FFC0CB
LightPink	#FFB6C1
HotPink	#FF69B4
DeepPink	#FF1493
MediumVioletRed	#C71585
PaleVioletRed	#DB7093

Orange Colors

LightSalmon	#FFA07A
Coral	#FF7F50
Tomato	#FF6347
OrangeRed	#FF4500
DarkOrange	#FF8C00
Orange	#FFA500

Yellow Colors

Gold	#FFD700
Yellow	#FFFF00
LightYellow	#FFFFE0
LemonChiffon	#FFFACD
LightGoldenrodYellow	#FAFAD2
PapayaWhip	#FFEFD5
Moccasin	#FFE4B5
PeachPuff	#FFDAB9

PaleGoldenrod	#EEE8AA
Khaki	#F0E68C
DarkKhaki	#BDB76B

Purple Colors

Lavender	#E6E6FA
Thistle	#D8BFD8
Plum	#DDA0DD
Violet	#EE82EE
Orchid	#DA70D6
Fuchsia	#FF00FF
Magenta	#FF00FF
MediumOrchid	#BA55D3
MediumPurple	#9370DB
BlueViolet	#8A2BE2
DarkViolet	#9400D3
DarkOrchid	#9932CC
DarkMagenta	#8B008B
Purple	#800080
Indigo	#4B0082
SlateBlue	#6A5ACD
DarkSlateBlue	#483D8B
MediumSlateBlue	#7B68EE

Green Colors

GreenYellow	#ADFF2F
Chartreuse	#7FFF00
LawnGreen	#7CFC00
Lime	#00FF00
LimeGreen	#32CD32
PaleGreen	#98FB98
LightGreen	#90EE90

MediumSpringGreen	#00FA9A
SpringGreen	#00FF7F
MediumSeaGreen	#3CB371
SeaGreen	#2E8B57
ForestGreen	#228B22
Green	#008000
DarkGreen	#006400
YellowGreen	#9ACD32
OliveDrab	#6B8E23
Olive	#808000
DarkOliveGreen	#556B2F
MediumAquamarine	#66CDAA
DarkSeaGreen	#8FBC8F
LightSeaGreen	#20B2AA
DarkCyan	#008B8B
Teal	#008080

Blue Colors

Aqua	#00FFFF
Cyan	#00FFFF
LightCyan	#E0FFFF
PaleTurquoise	#AFEEEE
Aquamarine	#7FFFD4
Turquoise	#40E0D0
MediumTurquoise	#48D1CC
DarkTurquoise	#00CED1
CadetBlue	#5F9EA0
SteelBlue	#4682B4
LightSteelBlue	#B0C4DE
PowderBlue	#B0E0E6
LightBlue	#ADD8E6
SkyBlue	#87CEEB

LightSkyBlue	#87CEFA
DeepSkyBlue	#00BFFF
DodgerBlue	#1E90FF
CornflowerBlue	#6495ED
MediumSlateBlue	#7B68EE
RoyalBlue	#4169E1
Blue	#0000FF
MediumBlue	#0000CD
DarkBlue	#00008B
Navy	#000080
MidnightBlue	#191970

Brown Colors

Cornsilk	#FFF8DC
BlanchedAlmond	#FFEBCD
Bisque	#FFE4C4
NavajoWhite	#FFDEAD
Wheat	#F5DEB3
BurlyWood	#DEB887
Tan	#D2B48C
RosyBrown	#BC8F8F
SandyBrown	#F4A460
Goldenrod	#DAA520
DarkGoldenrod	#B8860B
Peru	#CD853F
Chocolate	#D2691E
SaddleBrown	#8B4513
Sienna	#A0522D
Brown	#A52A2A
Maroon	#800000

White Colors

White	#FFFFFF
Snow	#FFFAFA
Honeydew	#F0FFF0
MintCream	#F5FFFA
Azure	#F0FFFF
AliceBlue	#F0F8FF
GhostWhite	#F8F8FF
WhiteSmoke	#F5F5F5
Seashell	#FFF5EE
Beige	#F5F5DC
OldLace	#FDF5E6
FloralWhite	#FFFAF0
Ivory	#FFFFF0
AntiqueWhite	#FAEBD7
Linen	#FAF0E6
LavenderBlush	#FFF0F5
MistyRose	#FFE4E1

Gray Colors

Gainsboro	#DCDCDC
LightGray	#D3D3D3
Silver	#C0C0C0
DarkGray	#A9A9A9
Gray	#808080
DimGray	#696969
LightSlateGray	#778899
SlateGray	#708090
DarkSlateGray	#2F4F4F
Black	#000000