# Speech Enabled Calculator For Windows Phone 8

This article explains how to create an voice controlled calculator app for Windows Phone using the Voice Command and Speech APIs.

Note: This article was a winner in the Windows Phone 8 Wiki Competition 2012Q4.

## Introduction

Windows Phone 8 introduces new Speech APIs which includes

- Voice Commands API.
- Speech To Text or Speech Recognition API.
- Text To Speech API.

Which is one of the great features Windows Phone 8 provides for creating awesome applications. Check out What's new in Windows Phone 8 and Speech APIs for Windows Phone 8 for an overview of the APIs.

In the following sections you will learn to:

- Create a basic calculator application which can perform simple mathematical operations by taking input from layout button.
- Extend our app to launch using Voice Commands.
- Once the app is launched, take speech input from user using Voice Recognition from within the app and perform operations.
- Speak out the results using Text To Speech.

Tip: We can also use Voice Commands to take speech input for performing operations. But for that part, its better to use Speech Recognition from within the app due to few limitations in specifying speech rules in Voice Commands as discussed later in the article. Remember, Voice Commands are for launching your app and to execute small actions.

## Project Setup

1. Make sure you have downloaded and installed the Windows Phone SDK 8.0.
2. Launch Visual Studio from the Windows Start screen.
3. Create a new project by selecting the **FILE | New Project** menu command, the New Project window appears.
4. Expand the installed Visual C# templates, and then select the Windows Phone templates.
5. In the list of Visual C# templates, in the Windows Phone group, select the **Windows Phone App** template.
6. At the bottom of the New Project window, type your project's Name, "Talking Calculator" in this case.
7. Click **OK**. The Windows Phone platform selection dialog box appears.
8. For the Target Windows Phone OS Version, select **Windows Phone OS 8.0**.
9. Click **OK**. The new project is created with basic files and layout, and opens in Visual Studio. The designer displays **MainPage.xaml**, which contains the user interface for the app.

### Required Capabilities

The application requires some capabilities for using the Speech and Voice Commands APIs. Enable them in **WPAppManifest.xml** file under the Capabilities tab. Capabilities needed are:

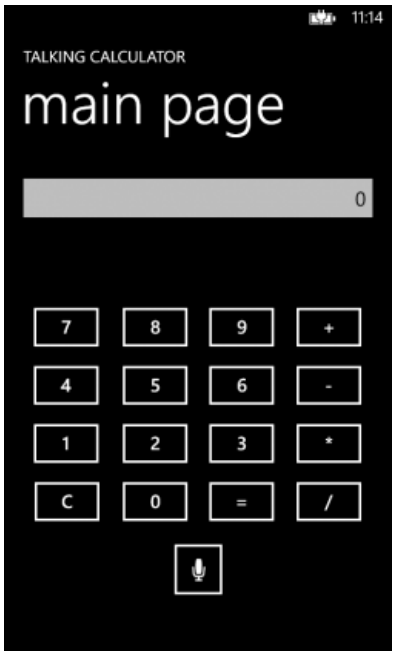
- `ID_CAP_NETWORKING`
- `ID_CAP_SPEECH_RECOGNITION`
- `ID_CAP_MICROPHONE`

## User Interface

In the solution explorer, double click on **MainPage.xaml** which contains a default layout of the app. This is the file you need to modify to create your applications layout. Our applications UI resembles layout of a basic calculator with buttons from 0-9 for


(C) Copyright Nokia 2014. All rights reserved.

entering operands, "+", "-", "*" and "/" buttons for entering operands, "=" for performing operations, "C" button for clearing up stack , a "mic" button to start taking input using speech and a text box to display operands and results.




There are always numerous ways for creating a layout. In this implementation we took a "*Single Row Grid*" and placed all our controls(Buttons, Text box) into it then adjusted their size and margins to make it look like a calculator layout.

Alternatively we could have made a "*Grid of 4 rows and4 Columns (16 cells)*" and placed one button per Cell with Text Box above the Grid and mic button below the grid to get the same layout.

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
            <Button Content="7" HorizontalAlignment="Left" Margin="13,170,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>
            <Button Content="8" HorizontalAlignment="Left" Margin="123,170,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>
            <Button Content="9" HorizontalAlignment="Left" Margin="230,170,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>
            <Button Content="4" HorizontalAlignment="Left" Margin="13,242,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>
            <Button Content="5" HorizontalAlignment="Left" Margin="123,242,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>
            <Button Content="6" HorizontalAlignment="Left" Margin="230,242,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>
            <Button Content="1" HorizontalAlignment="Left" Margin="13,314,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>
            <Button Content="2" HorizontalAlignment="Left" Margin="123,314,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>
            <Button Content="3" HorizontalAlignment="Left" Margin="230,314,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>
            <Button Content="0" HorizontalAlignment="Left" Margin="123,386,0,0"
VerticalAlignment="Top" Click="NumberPressed" Width="104"/>

            <Button Content="-" HorizontalAlignment="Left" Margin="338,242,0,0"
VerticalAlignment="Top" Click="OperatorPressed" Width="104"/>
            <Button Content="+" HorizontalAlignment="Left" Margin="338,170,0,0"
VerticalAlignment="Top" Click="OperatorPressed" Width="104"/>
            <Button Content="*" HorizontalAlignment="Left" Margin="338,314,0,0"
VerticalAlignment="Top" Click="OperatorPressed" Width="104"/>
            <Button Content="/" HorizontalAlignment="Left" Margin="338,386,0,0"
VerticalAlignment="Top" Click="OperatorPressed" Width="104"/>
```


(C) Copyright Nokia 2014. All rights reserved.

```
            <Button Content="=" HorizontalAlignment="Left" Margin="230,386,0,0"
VerticalAlignment="Top" Click="EqualPressed" Width="104"/>

            <Button Content="C" HorizontalAlignment="Left" Margin="14,386,0,0"
VerticalAlignment="Top" Click="ClearPressed" Width="104"/>
            <Button Content="" HorizontalAlignment="Left" Margin="187,463,0,0"
VerticalAlignment="Top" Width="83" Height="86" Click="RecordClicked">
                <Button.Background>
                    <ImageBrush Stretch="Fill" ImageSource="/images/microphone.png"/>
                </Button.Background>
            </Button>
            <TextBox Name="resultTextBox" HorizontalAlignment="Left" Height="72"
Margin="0,10,0,0" TextWrapping="Wrap" VerticalAlignment="Top" Width="456"
TextAlignment="Right" Text="0"/>
 </Grid>
```

### Observations from the layout code

- All the number buttons( 0 - 9 ) are associated with click handler `NumberPressed()`.
- All the operand buttons( +, -, *, / ) are associated with click handler `OperatorPressed()`.
- The "=" button is associated with click handler `EqualPressed()`.
- The "C" button is associated with click handler `ClearPressed()`.
- The "mic" button is associated with click handler `RecordClicked()`.

## Basic Calculator Code

In the solution explorer find **MainPage.xaml.cs** file, and open it by double clicking. This file contains the c# code for your application. Hence unless we create a new class, all our implementation code will be written in this file only.

Implementation of calculator functions are based on stack.

- When user inputs a number **push** it onto stack.
- When user inputs an operator **push** it onto stack.
- When user inputs = **pop** top three items from the stack and perform operation.

Following is the implementation for push, Pop and PerformOperation methods for the stack.

```
//List of strings to hold items
private List<string> myStack = new List<String>();

//Boolean to check if user is in middle of typing a number
private bool isInMiddleOfTyping = false;

//Push Method
private void Push(string myValue)
{
        //Make null check before pushing operand
 if (myValue != null)
 {
  myStack.Add(myValue);
 }
}

//Pop method
private String Pop()
{
 string value = null;
```


(C) Copyright Nokia 2014. All rights reserved.

```
        //Make sure stack is not empty before popping an item
 if (myStack.Count > 0)
 {
  value = myStack.ElementAt(0);
  myStack.RemoveAt(0);
 }
 return value;
}

//Perform operation method
private string PerformOperation()
{
 float result = 0;
 int firstOperand = 0;
 int secondOperand = 0;
 string myOperator = null;
 string topOfStack = pop();
 if (topOfStack != null && int.TryParse(topOfStack, out firstOperand))
 {
  topOfStack = pop();
  if (topOfStack != null)
  {
   myOperator = topOfStack;
   topOfStack = pop();
   if (topOfStack != null && int.TryParse(topOfStack, out secondOperand))
   {
    if (myOperator.Equals("+"))
    {
     result = firstOperand + secondOperand;
    }
    else if (myOperator.Equals("-"))
    {
     result = firstOperand - secondOperand;
    }
    else if (myOperator.Equals("*"))
    {
     result = firstOperand * secondOperand;
    }
    else if (myOperator.Equals("/"))
    {
     if (secondOperand == 0)
      result = 0;
     else
      result = (float)firstOperand / (float)secondOperand;
    }
    else
    {
     ClearStack();
    }
   }
   else
   {
    ClearStack();
   }
  }
  else
```


(C) Copyright Nokia 2014. All rights reserved.

```
  {
   ClearStack();
  }
 }
 else
 {
  ClearStack();
 }
 return result.ToString();
}

//ClearStack Method
private void ClearStack()
{
 myStack.Clear();
 resultTextBox.Text = "0";
 isInMiddleOfTyping = false;
}
```

We have four click handlers

- For number buttons ( 0 - 9 )

```
private void NumberPressed(object sender, RoutedEventArgs e)
{
        //Cast the sender to button type
 Button myButton = (Button)sender;
       //get number displayed on the button
 String pressedValue = myButton.Content.ToString();

        //if user was in middle of typing the number keep appending to the typed number
else replace the default "0" with pressed number on display text box
 if (isInMiddleOfTyping)
 {
  resultTextBox.Text = resultTextBox.Text + pressedValue;
 }
 else
 {
  resultTextBox.Text = pressedValue;
 }
 isInMiddleOfTyping = true;
}
```

- For operation buttons ( +, -, *, / )

```
private void OperatorPressed(object sender, RoutedEventArgs e)
{
 push(resultTextBox.Text);
 resultTextBox.Text = "0";
 if (myStack.Count == 3)
 {
  String intermediateResult = PerformOperation();
  if (intermediateResult != null)
   push(intermediateResult);
 }
 Button myButton = (Button)sender;
```


(C) Copyright Nokia 2014. All rights reserved.

```
 push(myButton.Content.ToString());
 isInMiddleOfTyping = false;
}
```

- For "=" button which implies "Perform Operation"

```
private void EqualPressed(object sender, RoutedEventArgs e)
{
        //push second operand onto the stack
 Push(resultTextBox.Text);

        //perform operation over top three items of stack
 resultTextBox.Text = performOperation();
}
```

- For "C" button.

```
private void ClearPressed(object sender, RoutedEventArgs e)
{
 ClearStack();
}
```

## Adding Voice Commands Feature

Now that we have implemented all basic calculator functions, let us add speech functionality to our application. We will start by adding *Voice Command* feature so that our application can be launched using *Voice Commands*.
For that we need to:

- Create a *Voice Command Definition(VCD)* file.
- Register the VCD file with Windows Phone OS.
- Handle Invoked Voice Commands.

Let us start with VCD file.

### Voice Commands Overview

Note: Please see Voice commands for Windows Phone 8 for a detailed description. This section is a very brief overview.

A VCD (Voice Command Definition) file defines the rules for voice commands that OS can recognize and wake the corresponding application to act upon. Each command consists of three parts:

1. App name
2. Command
3. Phrase list

For example, to use the *Media Player* app to *play* a file named *MySong* you might say:

```
Media Player play MySong
```

### Creating a VCD file

A VCD file can be created by following below steps (see Voice commands for Windows Phone 8 for more details):

1. Right click the windows phone project in visual studio.
2. Click on Add New Item.


(C) Copyright Nokia 2014. All rights reserved.

3. Select "Voice Command Definition" from list.
4. Give it a name.
5. Save it.

Our VCD file looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<VoiceCommands xmlns="http://schemas.microsoft.com/voicecommands/1.0">
  <CommandSet xml:lang="en-US">
    <CommandPrefix> Talking Calculator </CommandPrefix>
    <Example> Start </Example>

    <Command Name="startCalculator">
      <Example> run </Example>
      <ListenFor> start </ListenFor>
      <ListenFor> run </ListenFor>
      <Feedback> Starting Calculator... </Feedback>
      <Navigate />
    </Command>
  </CommandSet>
</VoiceCommands>
```

Looking at the VCD file

| "**Field**" | "**Value**" | "**Description**" |
|---|---|---|
| Command Prefix | Talking Calculator | Optional child element of the VoiceCommands element. If present, must be the first child element of the CommandSet element. Specifies a user-friendly name for an app that a user can speak when giving a voice command. This is useful for apps with names that are long or are difficult to pronounce. |
| Command Name | startCalculator | This field is particularly helpful to find out which Voice Command started up the application as we will see soon. |
| Voice commands | start or run | These are the commands that user can say in combination with Command Prefix to start our application. |
| Feedback | Starting Calculator... | This element specifies the text that will be displayed and read back to the user when the command is recognized. |

Tip: Which means the application can be launched using commands like "**Talking Calculator start**", "**Talking Calculator run**" etc.

## Registering the VCD File

We'll have to register the VCD file so that Windows Phone can recognize the voice commands for the app. This is done preferably on app start-up. Best place to register a VCD file is in the class constructor.

```
private async void RegisterVoiceCommandsFile()
{
 await VoiceCommandService.InstallCommandSetsFromFileAsync(new Uri("file://" +
Windows.ApplicationModel.Package.Current.InstalledLocation.Path +
@"/VoiceCommandDefinition1.xml", UriKind.RelativeOrAbsolute));
}
```

Tip: include below lines in the using block (on top of the .cs file) to make use of Speech and Voice Command APIs


(C) Copyright Nokia 2014. All rights reserved.

```
using Windows.Phone.Speech.VoiceCommands;
using Windows.Phone.Speech.Synthesis;
```

## Handling Invoked Voice Commands

Now, we'll have to handle the event when a voice command invokes the app. For that we need to override `OnNavigatedTo` handler.

```
protected override void OnNavigatedTo(System.Windows.Navigation.NavigationEventArgs e)
{
 base.OnNavigatedTo(e);
       //implementation followed
}
```

First we need to check check if the navigation to this page is due to a Voice Command. We do this by checking if a parameter named `voiceCommandName` exists in the `QueryString`. This corresponds to the Command Name parameter we specified in the VCD file.

```
if (NavigationContext.QueryString.ContainsKey("voiceCommandName"))
{
        //Write code to take actions depending on voice command
}
```

if there exists such parameter, it contains the name of the Voice Command which caused navigation to the page. So we can write cases for different Voice Commands we had registered for

```
if (NavigationContext.QueryString.ContainsKey("voiceCommandName"))
{
 // If so, get the name of the voice command.
 string voiceCommandName = NavigationContext.QueryString["voiceCommandName"];

 // Define app actions for each voice command name.
 switch (voiceCommandName)
 {
  case "Some case 1":
   //Do required operation
   break;
  case "Some case 2":
   //Do required operation
   break;
  default:
   break;
 }
}
```

Finally extract the parameters and perform required operations. This can be done by querying the parameters on `NavigationContext` which holds all the parameters regarding navigation.

```
string voiceCommandName = NavigationContext.QueryString["voiceCommandName"];

// Define app actions for each voice command name.
switch (voiceCommandName)
```


(C) Copyright Nokia 2014. All rights reserved.

```
{
 case "startCalculator":
  MessageBox.Show(@"Hit the voice command key and say, one plus one",
      "Voice Commands recognized.",
      MessageBoxButton.OK);
  break;
 default:
  break;
}
```

### Localizing Voice Commands

For supporting multiple languges you can create multiple CommandSets, each with a different `xml:lang` attribute to allow your app to be used in different markets. The value of the `xml:lang` attribute must be unique in the VoiceCommand document. Refer Language Codes for list of supported language codes. So for supporting french language in our application we will add below CommandSet to VCD file

```
<CommandSet xml:lang="fr-FR">
 <CommandPrefix> parler calculatrice </CommandPrefix>
 <Example> commencer </Example>
 <Command Name="startCalculator">
         <Example> courir </Example>
         <ListenFor> commencer </ListenFor>
         <ListenFor> courir </ListenFor>
         <Feedback> à partir de la calculatrice... </Feedback>
         <Navigate />
        </Command>
</CommandSet>
```

Note: If multiple CommandSets are specified in the VCD file then the CommandSet whose language matches with language selected on phone is used. This language is set by the user on the phone's Settings > System > Speech > Speech Language screen.

## Speech Recognition

### Why Use Speech Recognition

We can use Voice Commands instead of Speech Recognition for taking speech input by following below steps.
Create a `PhraseList` for numbers and operators. Note that we will need different `PhraseList` for number1 and number2 to get correct values in code after Voice Command is recognized.

```
        <PhraseList Label="number1">
            <Item> 1 </Item>
            <Item> 2 </Item>
            <Item> 3 </Item>
            <Item> 4 </Item>
            <Item> 5 </Item>
            <Item> 6 </Item>
            <Item> 7 </Item>
            <Item> 8 </Item>
            <Item> 9 </Item>
            <Item> 0 </Item>
        </PhraseList>
        <PhraseList Label="number2">
```


(C) Copyright Nokia 2014. All rights reserved.

```
            <Item> 1 </Item>
            <Item> 2 </Item>
            <Item> 3 </Item>
            <Item> 4 </Item>
            <Item> 5 </Item>
            <Item> 6 </Item>
            <Item> 7 </Item>
            <Item> 8 </Item>
            <Item> 9 </Item>
            <Item> 0 </Item>
        </PhraseList>
        <PhraseList Label="operator">
            <Item> plus </Item>
            <Item> add </Item>
            <Item> minus </Item>
            <Item> subtract </Item>
            <Item> multiply </Item>
            <Item> multiply by </Item>
            <Item> into </Item>
            <Item> divide </Item>
            <Item> divide by </Item>
        </PhraseList>
```

Create `ListenFor` tag in *number1 operator number2* format

```
<ListenFor> {number1} {operator} {number2} </ListenFor>
```

It goes good until we are working with single digit numbers, but it becomes very tidy to manage multiple digit numbers. We will need to specify all the numbers we want to recognize in the phrase list, which is completely in-feasible. So, it is better to do that part using Speech Recognition API, where in we can use an SRGS file to specify complex speech rules.

## Implementing Speech Recognition Feature

Now that our application is been launched by voice commands, users can speak to give input or to accomplish tasks by using speech recognition. Speech recognition conceptually seems very similar to the voice command feature, but is developed in a different way, using a different API. The key is that speech recognition occurs when you are in the app, and voice commands occur from outside of the app.
Speech recognition can be used with the predefined dictation grammar included with Windows Phone 8. The dictation grammar will recognize most words and short phrases in a language, and is activated by default when a speech recognizer object is instantiated. This can be achieved using below code.

```
   // Create an instance of SpeechRecognizerUI.
   SpeechRecognizerUI recoWithUI; = new SpeechRecognizerUI();

   // Start recognition (load the dictation grammar by default).
   SpeechRecognitionUIResult recoResult = await recoWithUI.RecognizeWithUIAsync();

   // Do something with the recognition result.
   MessageBox.Show(string.Format("You said {0}.", recoResult.RecognitionResult.Text));
```

In case if your application acts upon very small set of keywords, you can use list of those keywords as grammer for your SpeechRecognizer using below code

```
   // Create an instance of SpeechRecognizerUI.
```


(C) Copyright Nokia 2014. All rights reserved.

```
SpeechRecognizerUI recoWithUI; = new SpeechRecognizerUI();

// Create a string array of numbers.
string[] numbers = { "one", "two", "three", "four", "five",
                     "six", "seven", "eight", "nine" };

// Create a list grammar from the string array and add it to the grammar set.
recoWithUI.Recognizer.Grammars.AddGrammarFromList("Numbers", numbers);
```

But in our application speech recognition rules are bit complex. So we will use an external SRGS file containing rules for identifying spoken mathematical commands. Following section covers up SRGS File.

## Speech Recognition Grammar Specification(SRGS) File

The SRGS Version 1.0 is the industry-standard markup language for creating XML format grammars for speech recognition. SRGS grammars provide a full set of features to help you architect complex voice interaction with your apps. For example, with SRGS grammars you can:

- Specify the order in which words and phrases must be spoken to be recognized.
- Combine words from multiple lists and phrases to be recognized.
- Link to other grammars.
- Assign a weight to an alternative word or phrase to increase or decrease the likelihood that it will be used to match speech input.
- Include optional words or phrases.
- Use special rules that help filter out unspecified or unanticipated input, such as random speech that doesn't match the grammar, or background noise.
- Use semantics to define what speech recognition means to your app.
- Specify pronunciations, either inline in a grammar or via a link to a lexicon.

For a detailed description about SRGS grammar files follow SRGS Grammar.

## Creating a SRGS File

Now we need to create a SRGS file which defines rules to recognize spoken commands in format "**Number Operator Number**".

Note: To create an SRGS file follow same instructions as in creating a VCD File. Just select **SRGS Grammar** instead of **Voice Command Definition**.

First let us create rules for recognizing numbers.

```
<rule id = "Operand">
<one-of>
 <item>1</item>
 <item>2</item>
 <item>3</item>
 <item>4</item>
 <item>5</item>
 <item>6</item>
 <item>7</item>
 <item>8</item>
 <item>9</item>
 <item>0</item>
</one-of>
</rule>
```

Same way define rules for recognizing operators.

```
<rule id = "Operator">
```


(C) Copyright Nokia 2014. All rights reserved.

```
<one-of>
 <item>plus</item>
 <item>add</item>
 <item>minus</item>
 <item>subtract</item>
 <item>multiply</item>
 <item>multiply by</item>
        <item>cross</item>
 <item>into</item>
 <item>divide</item>
 <item>divide by</item>
 <item>divided by</item>
 <item>slash</item>
</one-of>
</rule>
```

At this point we can identify numbers and operators separately let us create a rule to identify pattern "**Number Operator Number**" using below code.

```
<rule id="command" scope="public">
 <example>4 plus 2</example>
 <example>4 divided by 2</example>
 <item repeat="0-1"> Calculate </item>
 <item repeat="1-10"><ruleref uri="#Operand" /></item>
 <tag> out.FirstOperand=rules.latest(); </tag>
 <ruleref uri="#Operator" />
 <tag> out.Operator=rules.latest(); </tag>
 <item repeat="1-10"><ruleref uri="#Operand" /></item>
 <tag> out.SecondOperand=rules.latest(); </tag>
</rule>

<rule id = "Operator">
<one-of>
 <item>plus</item>
 <item>add</item>
 <item>minus</item>
 <item>subtract</item>
 <item>multiply</item>
 <item>multiply by</item>
        <item>cross</item>
 <item>into</item>
 <item>divide</item>
 <item>divide by</item>
 <item>divided by</item>
 <item>slash</item>
</one-of>
</rule>

<rule id = "Operand">
<one-of>
 <item>1</item>
 <item>2</item>
 <item>3</item>
 <item>4</item>
 <item>5</item>
 <item>6</item>
```


(C) Copyright Nokia 2014. All rights reserved.

```
 <item>7</item>
 <item>8</item>
 <item>9</item>
 <item>0</item>
</one-of>
</rule>
```

Things to note.

- Observe how rules are used inside a `rule` using `ruleref` tag providing id of external rule.
- `repeat` attribute in item tag specifies number of times a contained expansion can be repeated. The value of this attribute can be any positive integer, or any range of positive integers, but cannot be zero. The value of this attribute applies to the entire content of the item element. In our case it can be repeated in the range of 1 to 9. Which means we can input at the max 9 digit numbers. Which is far more than enough for a sample application.
- `one-of` matches any one of the nested items.
- A `tag` element contains semantic information, which returns additional information when an element or series of elements is recognized. A `tag` element does not affect the legal word patterns defined by the grammars or the process of recognizing speech or other input given a grammar.
- `example` tag also does not affect the process of recognizing speech. It is for displaying examples to the user in speech recognition UI.

At this point SRGS file is ready, now we need to instantiate an object of `SpeechRecognizerUI` and tell it to use our SRGS file for speech recognition. It can be done using below code.

```
// Initialize the SpeechRecognizerUI object.
SpeechRecognizerUI recoWithUI = new SpeechRecognizerUI();

// Initialize a URI with a path to the SRGS-compliant XML file.
Uri calculatorRulesUri = new Uri("file://" +
Windows.ApplicationModel.Package.Current.InstalledLocation.Path +
@"/CalculatorRules.xml", UriKind.RelativeOrAbsolute);

// Add the grammar to the grammar set.
recoWithUI.Recognizer.Grammars.AddGrammarFromUri("calculatorRulesUri",
calculatorRulesUri);

// Load the grammar set and start recognition.
SpeechRecognitionUIResult recoResult = await recoWithUI.RecognizeWithUIAsync();
```

Here **CalculatorRules.xml** is name of our SRGS file. Once above lines of code executes, a speech recognition dialog will appear for user to give input commands using speech. Once a command a recognized it must be processed and corresponding actions should be taken. `SpeechRecognitionUIResult` class serves the purpose.

```
// Load the grammar set and start recognition.
SpeechRecognitionUIResult recoResult = await recoWithUI.RecognizeWithUIAsync();

// If speech recognization succeded
if (recoResult.ResultStatus == SpeechRecognitionUIStatus.Succeeded)
{
 //get and parse the semantic information retrieved from the recognition result.
 String myStr = recoResult.RecognitionResult.Text;

 //remove spaces from the recognized string
 ParseRecognizedResult(myStr.Replace(" ", ""));
```


(C) Copyright Nokia 2014. All rights reserved.

```
}
```

## Parsing Matched Commands

Once a command is recognized it will be in raw string format, kind of a English sentence. We need to parse it to extract our operands and operator for performing required operation. Considering the fact our commands will always be in the **Number Operator Number** format, we can make use of Regular Expressions to separate out the parameters using below code.

```
private void ParseRecognizedResult(String result)
{
 Regex myRegex = new Regex("(?<number1>[0-9]{1,9})(?<operator>[^0-9]+)(?<number2>[0-
9]{1,9})");
 Match m = myRegex.Match(result);
 if (m.Success)
 {
  PushNumbersOnStackAndPerformOperation(m.Groups["number1"].Value,
m.Groups["operator"].Value, m.Groups["number2"].Value);
 }
 else
 {
  SpeakString("Sorry, unable to perform operation.");
 }
}
```

If `RegExp` succeds in finding a match, it will put **first operand** in **number1** group, **second operand** in **number2** group and **operator** in **operator** group of `Match` object followed by performing operation. Else it will speak out an error message.

Tip: Visit Regular Expressions for details about using regular expressions in Windows Phone 8..

For implemenataion for `PushNumbersOnStackAndPerformOperation(String, String, String)`, firstly operator is filtered out to be a genuine operator symbol before pushing onto stack as it will be in string format like "plus", "minus" etc in speech recognition results, followed by pushing operator and operands on the stack followed by calling `PerformOperation()`. to perform the required operation ans speak out he results.

```
private void PushNumbersOnStackAndPerformOperation(String number1, String myOperator,
String number2)
{
 string rawOperator = myOperator;
 string filteredOperator = getOperator(rawOperator);

 if (number1 != null && filteredOperator != null && number2 != null)
 {
  push(number1);
  push(filteredOperator);
  push(number2);

                String result = PerformOperation();
                resultTextBox.Text = result;
                //Perform operation on recently pushed variables
                SpeakString(number1 + " " + rawOperator + " " + number2 + " is, " +
result);
 }
 else
 {
  SpeakString("sorry, unable to perform operation.");
```


(C) Copyright Nokia 2014. All rights reserved.

```
 }
}
```

Before utilizing the values fetched from the `NavigationContext` those must be filtered or we can say verified to be proper values. Since operands will always be digits(since we specified only digits in SRGS file) there is no need to filter them. So we create method `FilterOperator` for filtering out Operators.

```
private String FilterOperator(String op)
{
 String matchedOperator = null;
 if (op != null && op.Length > 0)
 {
  if (op.Equals("plus", StringComparison.OrdinalIgnoreCase) || op.Equals("add",
StringComparison.OrdinalIgnoreCase))
  {
   matchedOperator = "+";
  }
  else if (op.Equals("minus", StringComparison.OrdinalIgnoreCase) ||
op.Equals("subtract", StringComparison.OrdinalIgnoreCase))
  {
   matchedOperator = "-";
  }
  else if (op.Equals("multiply", StringComparison.OrdinalIgnoreCase) ||
op.Equals("multiply by", StringComparison.OrdinalIgnoreCase) || op.Equals("into",
StringComparison.OrdinalIgnoreCase || op.Equals("cross",
StringComparison.OrdinalIgnoreCase))
  {
   matchedOperator = "*";
  }
  else if (op.Equals("divide", StringComparison.OrdinalIgnoreCase) || op.Equals("divided
by", StringComparison.OrdinalIgnoreCase) || op.Equals("divide by",
StringComparison.OrdinalIgnoreCase || op.Equals("slash",
StringComparison.OrdinalIgnoreCase))
  {
   matchedOperator = "/";
  }
 }
 return matchedOperator;
}
```

## Making our Calculator Speak the Result

After we perform the operations fired by recognizing voice commands we have to speak out the calculated results to the user and this is done by using `SpeechSynthesizer` class.

```
private async void SpeakString(string myString)
{
        //Create an instance of SpeechSynthesizer class.
        var text2speech = new SpeechSynthesizer();

        //You can set voice of your choice depending on language, gender
        //text2speech.SetVoice(voice_of_your_choice);
```

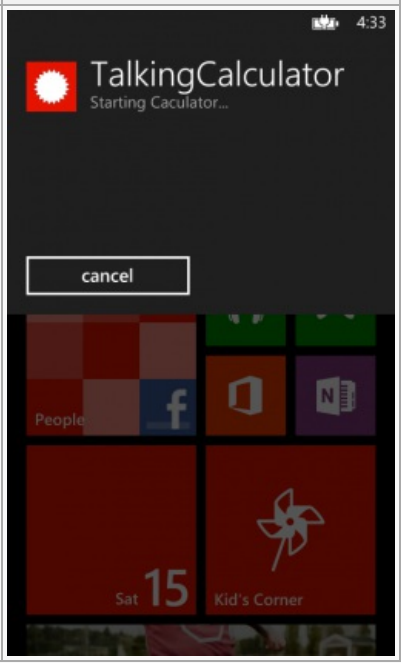

(C) Copyright Nokia 2014. All rights reserved.

```
        //Invoke method SpeakTextAsync passing it the string to be spoken out.
        await text2speech.SpeakTextAsync(myString);
}
```

## Running the Application

Note: Make sure 'en-US' language pack is installed on the phone before running the application. Language pack is necessary for speech recognition to work.

- Build the solution by selecting the **BUILD | Build Solution** menu command.
- If any errors occur, they're listed in the Error List window. If there are errors, review the steps above, correct any errors, and then build the solution again.
- On the standard toolbar, make sure the deployment target for the app is set to one of the values for the Windows Phone 8 Emulator.
- Run the app by pressing F5 or by selecting the **DEBUG | Start Debugging** menu command or pressing the green arrow on toolbar. This opens the emulator window and launches the app. If this is the first time you are starting the emulator, it might take a few seconds for it to start and launch your app.
- Once the application is launched Hit the home key to go to the start screen. Then press and hold the Windows button which will display Voice Command listening screen. Now say **Talking Calculator, start**, which will displaying Starting Caclulator UI and launch the calculator application.

| Voice Command listening UI | Starting Caclulator UI |
|---|---|
| |   |

- Now press "mic button" on the application layout and say a mathematical command like "**one plus two**", "**two four minus four**" etc. If it recognizes the command it will show Heard You Say screen.

| Heard You Say |
|---|


(C) Copyright Nokia 2014. All rights reserved.

- In case there are ambiguities regarding spoken command, "did you say" screen is displayed with list of commands to select from.

| Did You Say |
| --- |
| |



Note: numbers should be spoken on digit by digit basis i.e **123** should be pronounced as **one two three** and *not* **one hundred and twenty three**.

## TODO

If you have successfully created and ran Talking Calculator application, Congratulations! You have done a great job.:) Now as enhancement and maybe kind of homework you can try implementing following things :

1. Current logic is not robust enough to check if user is entering values in correct order and may crash if wrong values are provided. Add the checks and make sure our application handle all kind of scenarios.
2. Recreate the calculator layout by taking "*Grid of 4 rows and4 Columns (16 cells)*" , placing one button per Cell with Text Box above the Grid and mic button below the Grid.


(C) Copyright Nokia 2014. All rights reserved.

## Summary

We learnt following things from the article:

- Creating simple layouts for Windows Phone.
- Implementing basic calculator functions using stack.
- Using Voice Commands API.
- Using Speech Recognition within app.
- Using Text To Speech API.
- Usage of regular expressions.

## Source Code

Follow File:Talking Calculator Source.zip for source code of the application.

## Version Hint

**Windows Phone:** [[Category:Windows Phone]]
[[Category:Windows Phone 7.5]]
[[Category:Windows Phone 8]]

**Nokia Asha:** [[Category:Nokia Asha]]
[[Category:Nokia Asha Platform 1.0]]

**Series 40:** [[Category:Series 40]]
[[Category:Series 40 1st Edition]] [[Category:Series 40 2nd Edition]]
[[Category:Series 40 3rd Edition (initial release)]] [[Category:Series 40 3rd Edition FP1]] [[Category:Series 40 3rd Edition FP2]]
[[Category:Series 40 5th Edition (initial release)]] [[Category:Series 40 5th Edition FP1]]
[[Category:Series 40 6th Edition (initial release)]] [[Category:Series 40 6th Edition FP1]] [[Category:Series 40 Developer Platform 1.0]] [[Category:Series 40 Developer Platform 1.1]] [[Category:Series 40 Developer Platform 2.0]]

**Symbian**: [[Category:Symbian]]
[[Category:S60 1st Edition]] [[Category:S60 2nd Edition (initial release)]] [[Category:S60 2nd Edition FP1]] [[Category:S60 2nd Edition FP2]] [[Category:S60 2nd Edition FP3]]
[[Category:S60 3rd Edition (initial release)]] [[Category:S60 3rd Edition FP1]] [[Category:S60 3rd Edition FP2]]
[[Category:S60 5th Edition]]
[[Category:Symbian^3]] [[Category:Symbian Anna]] [[Category:Nokia Belle]]


(C) Copyright Nokia 2014. All rights reserved.