ELM PROGRAMMING

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Elm is a pure functional programming language that compiles to JavaScript. It simplifies the language as well as an application framework. Elm is designed specifically for web frontend with the unique feature of no Runtime exceptions.

This tutorial adopts a simple and practical approach to describe the concepts of Elm programming.

Audience

This tutorial has been prepared for beginners to help them understand the basic and advanced concepts of Elm.

Prerequisites

An understanding of basic programming concepts is necessary for this course.

Copyright & Disclaimer

Table of Contents

1. Elm — Introduction

Elm is a functional programming language. It was Designed by Evan Czaplicki in 2012. Elm is specifically used for designing front end of web applications.

Elm compiles to JavaScript and runs in the browser. It is fast, testable, maintainable, and comes with no Runtime exceptions.

Some practical applications of the Elm programming platform include –

- Games
- Graphics
- Single Page Applications

Why Elm

Elm eliminates most of the common problems faced by frontend developers. This includes –

No Runtime Exceptions

Elm is a statically typed language. All possible errors are validated and corrected at compile-time. This makes it possible to have no runtime exceptions.

Developer Friendly Error Messages

Unlike other programming languages, Elm's compiler is designed to provide very specific and developer-friendly error messages at compile time. The error messages also include hints such as links to recommended design documentations.

Easy to Test

Each Elm function can be tested in isolation of all others. This makes programs written in Elm easily testable.

Automatic Semantic Versioning

Elm enforces automatic semantic versioning of packages. This ensures that a patch change does not crash an already running application.

Reusable Code

Elm functions are inherently easy to reuse compared to functions in JavaScript, Python, or TypeScript.

2. Elm — Environment Setup

This chapter discusses steps to install Elm on Windows, Mac and Linux platforms.

Local Environment Setup

Consider the steps shown below to install Elm in your local environment.

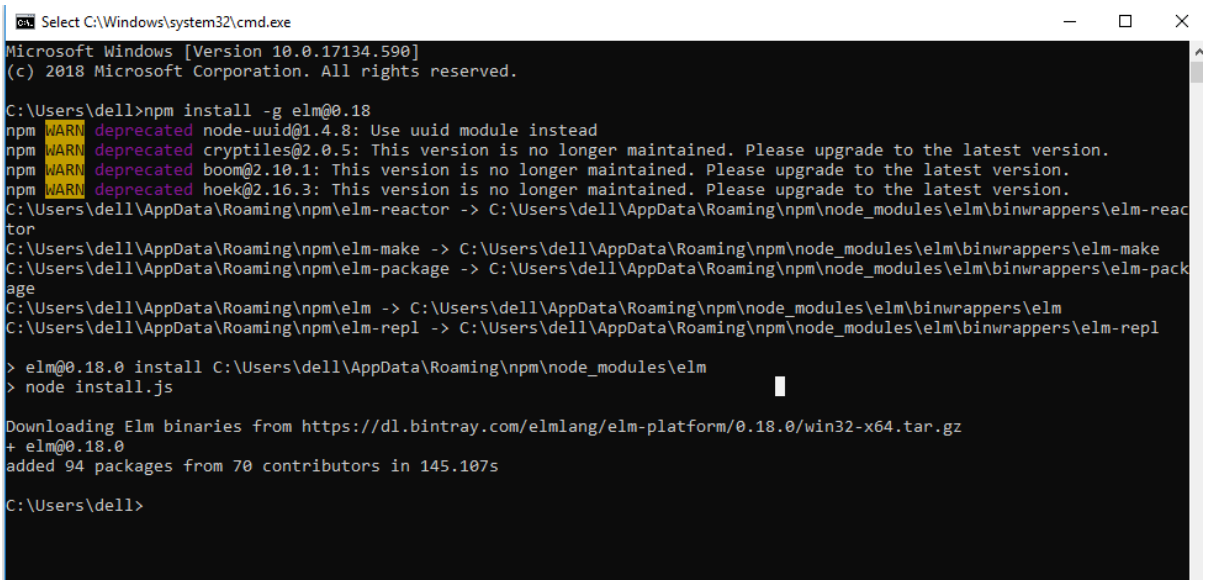
Step 1: Install node

Since elm is compiled to JavaScript, the target machine should have **node** installed. Refer to TutorialsPoint NodeJS course for steps to setup **node** and **npm** Node setup.

Step 2: Install elm

Execute the following command on the terminal to install elm. Note that the stable version of elm was 0.18 at the time of writing this course.

```
npm install -g elm@0.18
```



After installation, execute the following command to verify the version of Elm.

```
C:\Users\dell>elm --version
0.18.0
```

Step 3: Install the Editor

The development environment used here is Visual Studio Code (Windows platform).

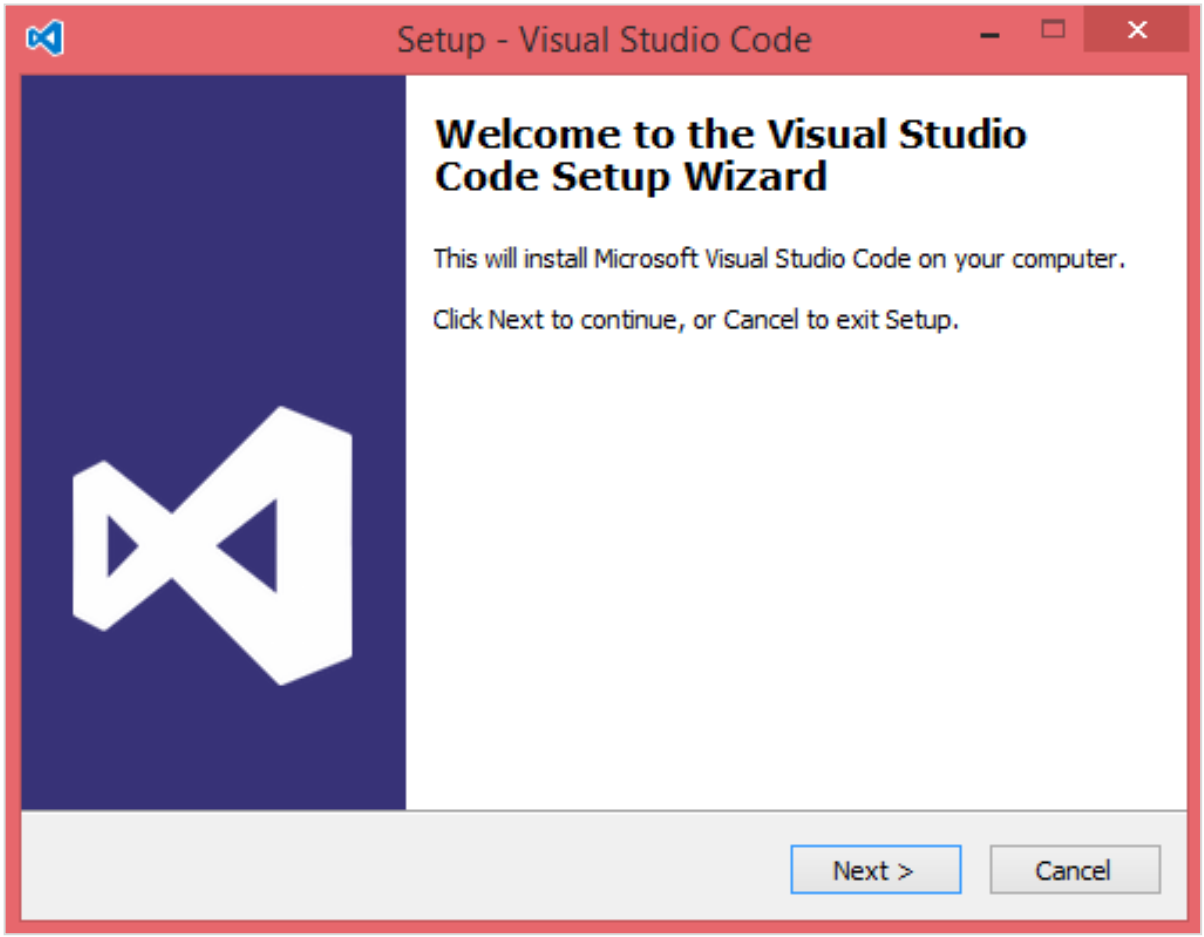
Visual Studio Code is an open source IDE from Visual Studio. It is available for Mac OS X, Linux and Windows platforms. VSCode is available at Visual Studio Code

Installation on Windows

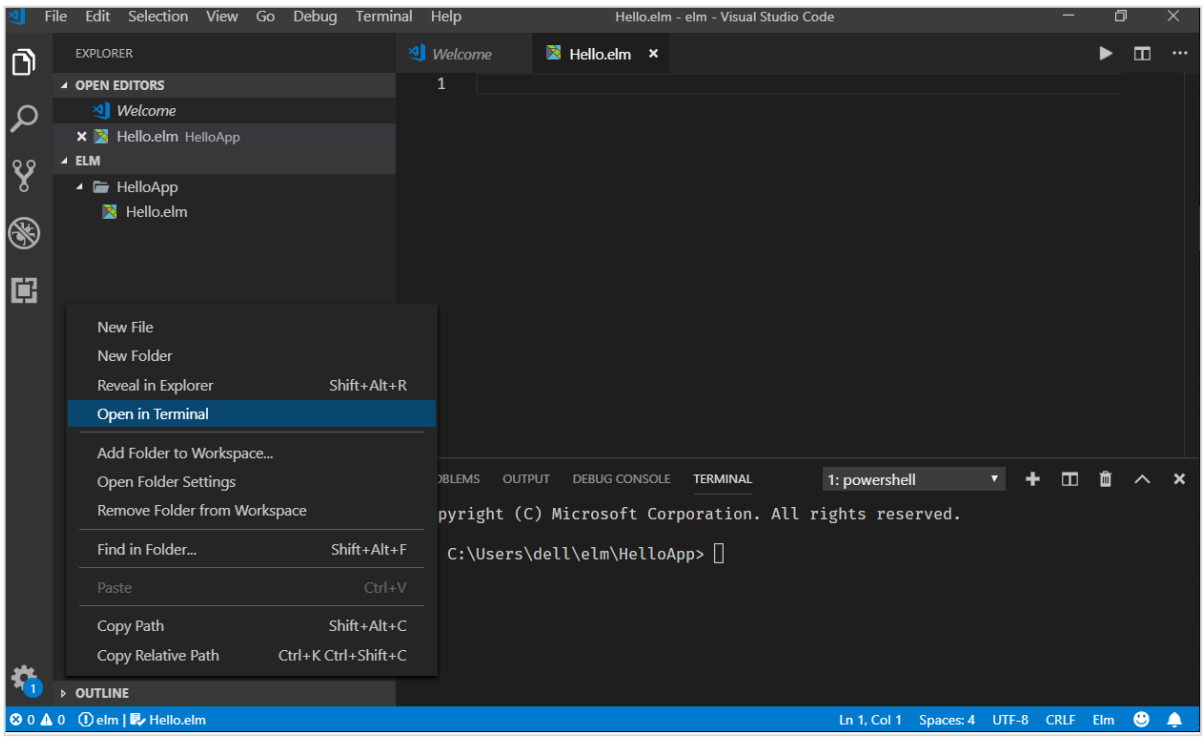
In this section, we will discuss the steps to install Elm on Windows.

Download Visual Studio Code for Windows.

Double-click on VSCodeSetup.exe to launch the setup process. This will only take a minute.



You may directly traverse to the file's path by right clicking on File → Open in command prompt. Similarly, the **Reveal** in Explorer option shows the file in the File Explorer.



Installation on Mac OS X

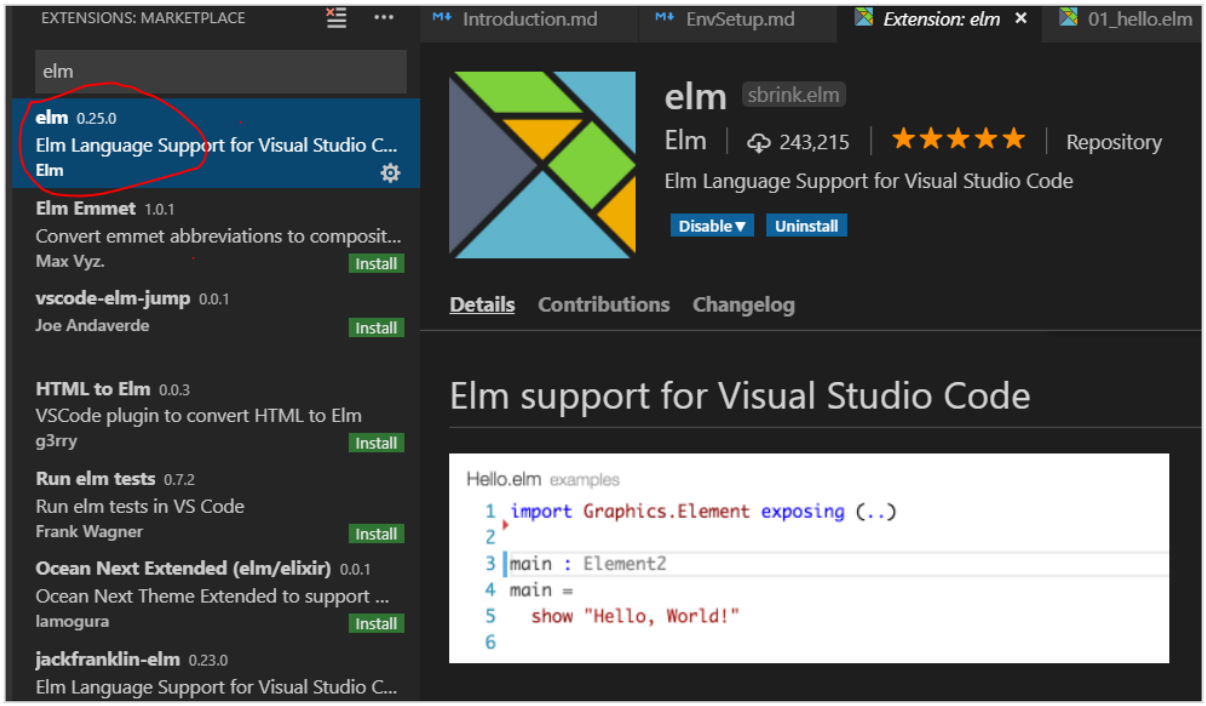
Visual Studio Code's Mac OS X specific installation guide can be found at VSCode Installation-MAC.

Installation on Linux

Visual Studio Code's Linux specific installation guide can be found at VSCode Installation-Linux

Step 4: Install the elm Extension

Install the elm extension in VSCode as shown below.



Elm REPL

REPL stands for Read Eval Print Loop. It represents a computer environment like a Windows console or Unix/Linux shell where a command is entered and the system responds with an output in an interactive mode.

Elm comes bundled with a REPL environment. It performs the following tasks –

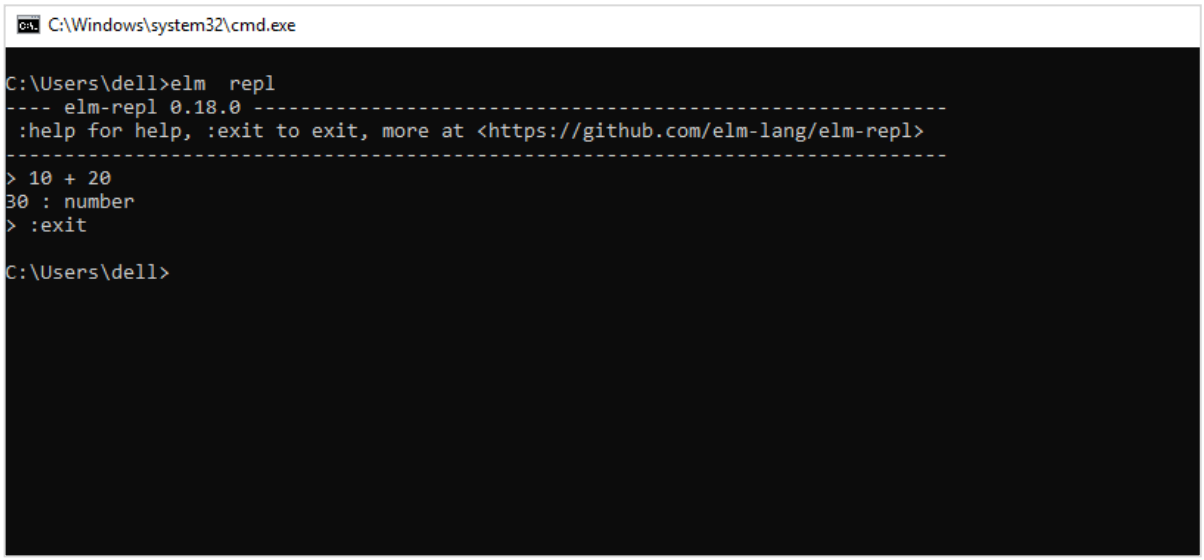
- Read – Reads user's input, parses the input into elm data-structure, and stores in memory.
- Eval – Takes and evaluates the data structure.
- Print – Prints the result.
- Loop – Loops the above command until the user exits. Use the command ***:exit*** to exit REPL and return to the terminal.

A simple example to add two numbers in REPL is shown below:

Open the VSCode terminal and type the command elm REPL.

The REPL terminal waits for the user to enter some input. Enter the following expression 10 + 20. The REPL environment processes the input as given below:

- Reads numbers 10 and 20 from user.
- Evaluates using the + operator.
- Prints result as 30.
- Loops for next user input. Here we exit from loop.

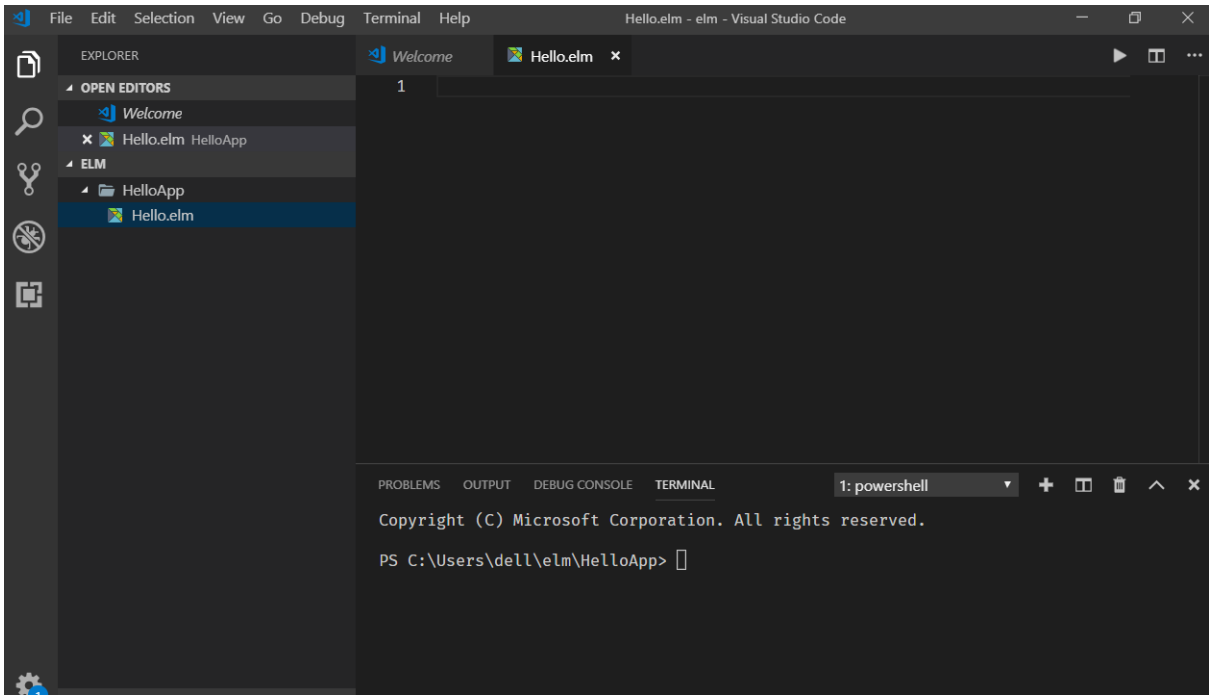


3. Elm — Basic Syntax

This chapter discusses how to write a simple program in elm.

Step 1: Create a directory HelloApp in VSCode.

Now, create a file - **Hello.elm** in this directory.



The above diagram shows project folder **HelloApp** and terminal opened in VSCode.

Step 2: Install the necessary elm packages

The package manager in elm is *elm-package*. Install the *elm-lang/html* package. This package will help us display output of elm code in the browser.

Traverse to the **HelloApp** project folder by right clicking on File → Open in command prompt in VSCode.

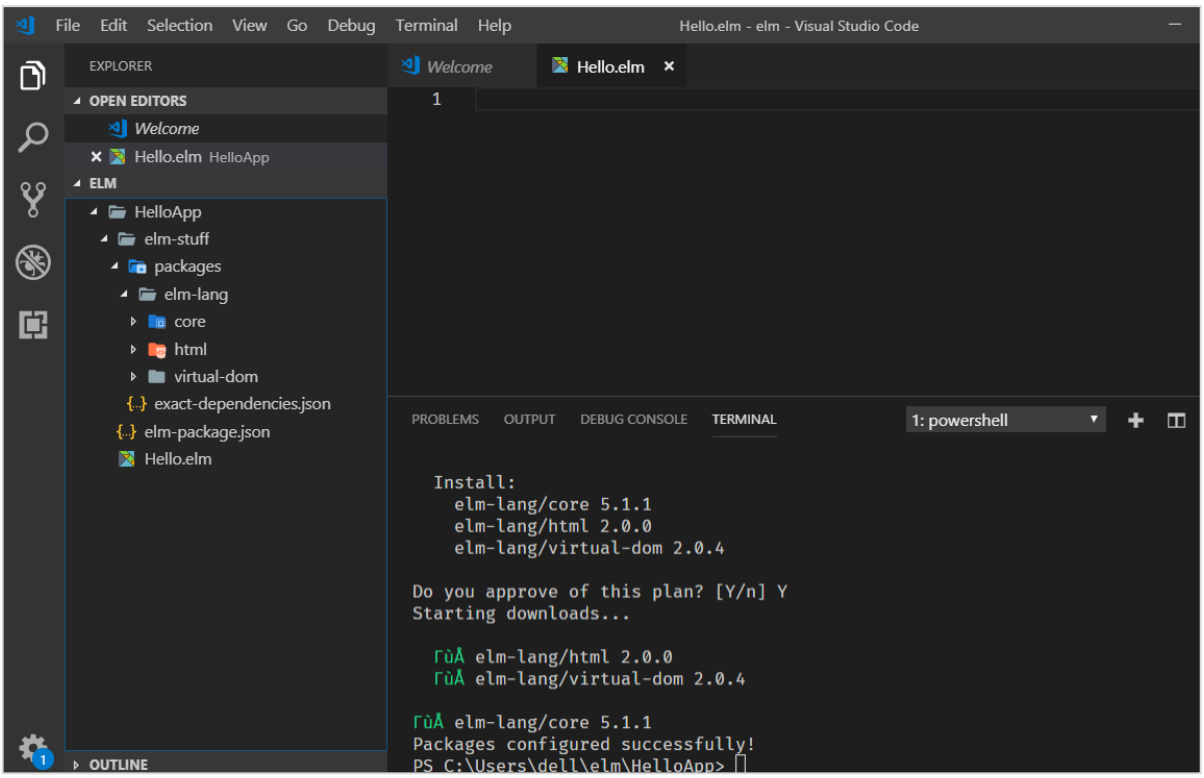
Execute the following command in the terminal window:

```
C:\Users\dell\Elm\HelloApp> elm-package install elm-lang/html
```

The following files/folders are added to the project directory on installing the package.

- elm-package.json (file), stores project meta data
- elm-stuff (folder), stores external packages

The following message will appear once the package is installed successfully.



Step 3: Add the following code to the Hello.elm file

```
-- importing Html module and the function text
import Html exposing (text)


-- create main method
main =
-- invoke text function
 text "Hello Elm from TutorialsPoint"
```

The above program will display a string message ***Hello Elm from TutorialsPoint*** in the browser.

For this, we need to import the function ***text*** within the ***Html*** module. The *text* function is used to print any string value in the browser. The *main* method is the entry point to a program. The *main* method invokes the *text* function and passes a string value to it.

Step 4: Compile the project

Execute the following command in VSCode terminal window.

```
elm make Hello.elm
```

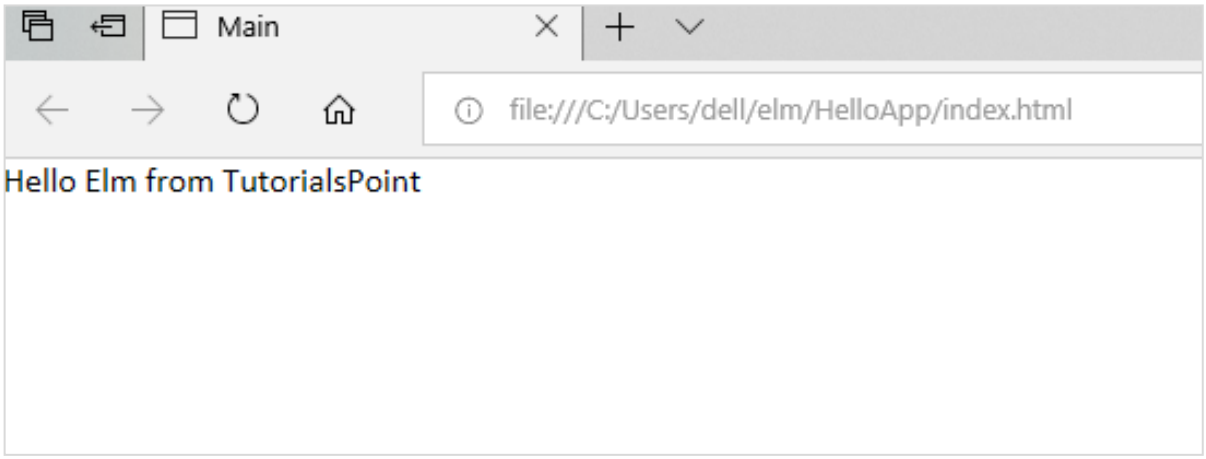
The output of the above command is as shown below-

```
//update path to the proj folder in the command elm make
C:\Users\dell\elm\HelloApp>elm make Hello.elm
Success! Compiled 38 modules.
Successfully generated index.html
```

The above command will generate an **index.html** file. The elm compiler converts *.elm* file to JavaScript and embeds it in the **index.html** file.

Step 5: Open the index.html in the browser

Open the *index.html* file in any browser. The output will be as shown below:



Comments in Elm

Comments are a way to improve the readability of a program. Comments can be used to include additional information about a program like author of the code, hints about a function construct, etc. Comments are ignored by the compiler.

Elm supports the following types of comments –

- Single-line comments (--): Any text between a -- and the end of a line is treated as a comment.
- Multi-line comments ({- -}): These comments may span multiple lines.

Illustration

```
-- this is single line comment


{- This is a
   Multi-line comment
-}
```

Lines and Indentation

Elm provides no braces to indicate blocks of code for function definitions or flow control. Blocks of code are denoted by line indentation, which is rigidly enforced. All statements within a block must be indented the same amount. For example –

```
module ModuleIf exposing (..)
x = 0

function1=
 if x > 5 then
    "x is greater"
 else
    "x is small"
```

However, the following block generates an error –

```
-- Create file ModuleIf.elm
module ModuleIf exposing (..)
x = 0

function1=
 if x > 5 then
    "x is greater"
else         --Error:else indentation not at same level of if statement
    "x is small"
```

Thus, in Elm all the continuous lines indented with same number of spaces would form a block.

```
C:\Users\admin>elm repl
---- elm-repl 0.18.0 -----------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
-------------------------------------------------------------------------------
-

> import ModuleIf exposing(..) -- importing module from ModuleIf.elm file
>function1  -- executing function from module
-- SYNTAX PROBLEM -------------------------------------------------
```

```
I need whitespace, but got stuck on what looks like a new declaration. You are
either missing some stuff in the declaration above or just need to add some
spaces here:

7| else
   ^
I am looking for one of the following things:

    whitespace
```

4. Elm — Data Types

The Type System represents the different types of values supported by the language. The Type System checks validity of the supplied values, before they are stored or manipulated by the program. This ensures that the code behaves as expected. The Type System further allows for richer code hinting and automated documentation too.

Elm is a statically typed language. Elm has types that are similar to those from other languages.

Number

The *number* data type represents numeric values. The Elm type system supports the following numeric types -

Sr. No.	Type	Example
1	number: Stores any number	7 is number type
2	Float: Stores fractional values	7/2 gives 3.5 result as Float
3	Int: Stores non-fractional values	7//2 gives 3 result as Int

The type *number* accommodates both fractional and non-fractional values. Open the elm REPL and try the examples given below -

```
C:\Users\admin>elm repl
---- elm-repl 0.18.0 -----------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
-------------------------------------------------------------------------------
-
> 7
7 : number
> 7/2
3.5 : Float
> 7//2
3 : Int
>
```

String and Char

The *String* data type is used to represent a sequence of characters. The *Char* data type is used to represent a single character. *String* values are defined within a double quote " and *Char* values are enclosed within a single quote '.

Sr. No.	Type	Example
1	String: Stores a sequence of characters	"TutorialsPoint"
2	Char: Stores a single character value	'T'

Open the elm REPL and try the examples given below –

```
C:\Users\admin>elm repl
---- elm-repl 0.18.0 ---------------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
-
> "TutorialsPoint"
"TutorialsPoint" : String
> 'T'
'T' : Char
```

Bool

The Bool data type in Elm supports only two values — True and False. The keyword *Bool* is used to represent a Boolean value.

Sr. No.	Type	Example
1	Bool : Stores values True or False	1==1 returns True

Open the elm REPL and try the examples given below –

```
C:\Users\dell\elm>elm repl
---- elm-repl 0.18.0 ---------------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
```

```
------------------------------------------------------------------------------
-
> True
True : Bool
> False
False : Bool
> 1==1
True : Bool
> 1==2
False : Bool
> 1 /= 2   -- not equal
True : Bool
> not True
False : Bool
> not False
True : Bool
```

Custom Types

Elm supports creating user defined types. For example, consider a payment application. The application needs to store different modes of payment – credit card, debit card and net banking. This can be achieved by defining a custom type and restricting its value to the three acceptable modes of payments.

```
The following example shows how to make a custom type.
> type PaymentMode = CreditCard|NetBanking|DebitCard
> payment1 = CreditCard
CreditCard : Repl.PaymentMode
> payment2 = DebitCard
DebitCard : Repl.PaymentMode
> payment3 = UPI
-- NAMING ERROR ---------------------------------------------- repl-temp-
000.elm

Cannot find variable `UPI`

7| payment3 = UPI
```

In the above example, we created a *PaymentMode* custom type. Variables payment1 and payment2 are assigned to PaymentMode values. If the value assigned to the variable does

not match any of the values defined by the *PaymentMode* type, the application will throw a syntax error.

Structured Data types

Structured data types can be used to store multiple values in a structured format. Elm supports the following structured data types:

- Tuple
- List
- Record

These will be discussed in detail in the upcoming chapters.

5. Elm — Variables

A variable, by definition, is "a named space in the memory" that stores values. In other words, it acts as a container for values in a program. A variable helps programs to store and manipulate values.

Variables in Elm are associated with a specific data type. The data type determines the size and layout of the variable's memory, the range of values that can be stored within that memory and the set of operations that can be performed on the variable.

Variable Naming-Rules

In this section, we will learn about the Variable Naming-Rules.

- Variable names can be composed of letters, digits, and the underscore character.
- Variable names cannot begin with a digit. It must begin with either a letter or an underscore.
- Upper and lowercase letters are distinct because Elm is case-sensitive.

Variable Declaration in Elm

The type syntax for declaring a variable in Elm is given below:

Syntax 1

```
variable_name:data_type=value
```

The **" : "** syntax (known as type annotation) is used to associate the variable with a data type.

Syntax 2

```
variable_name=value-- no type specified
```

The data type is optional while declaring a variable in Elm. In this case, the data type of the variable is inferred from the value assigned to it.

Illustration

This example uses VSCode editor to write an elm program and execute it using the elm repl.

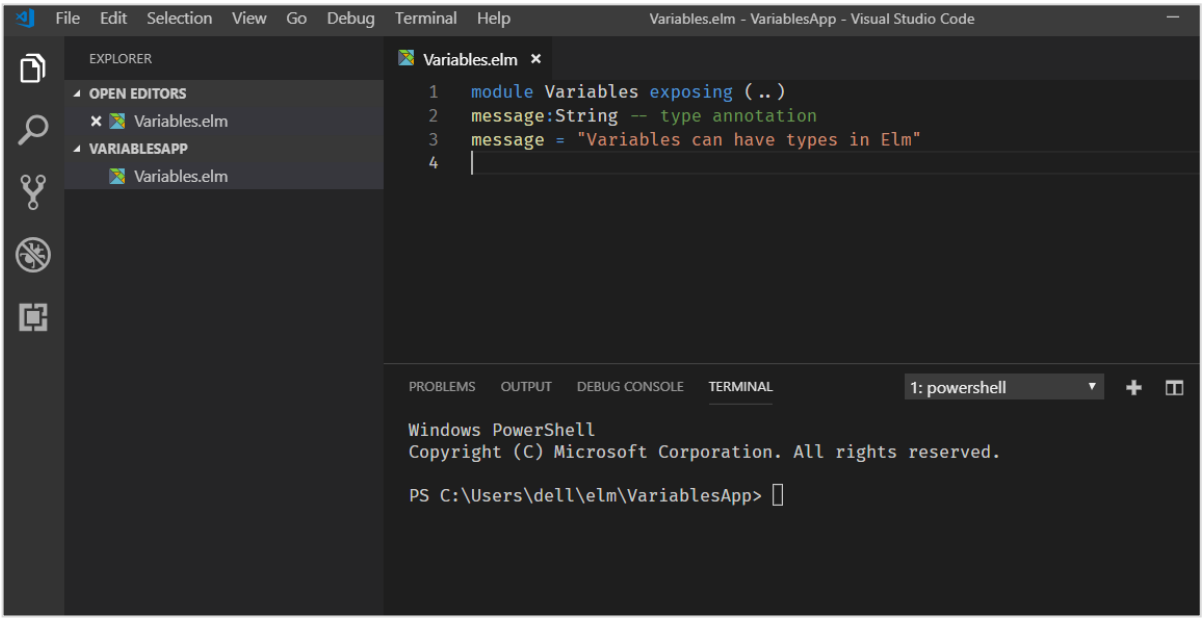
Step 1: Create a project folder - VariablesApp. Create a Variables.elm file in the project folder.

Add the following content to the file.

```
   module Variables exposing (..) //Define a module and expose all contents in
the module

   message:String -- type annotation

   message = "Variables can have types in Elm"
```

The program defines a module *Variables.* The name of a module must be the same as that of the elm program file. The *(..)* syntax is used to expose all components in the module.

The program declares a variable *message* of the type *String*.



Step 2: Execute the program.

- Type the following command in the VSCode terminal to open the elm REPL.

```
elm repl
```

- Execute the following elm statement in the REPL terminal.

```
> import Variables exposing (..) --imports all components from the Variables
module

> message                        --Reads value in the message varaible and
prints it to the REPL
```

```
"Variables can have types in Elm":String
>
```

Illustration

Use Elm REPL to try the following example.

```
C:\Users\dell\elm>elm repl
---- elm-repl 0.18.0 -----------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
> company = "TutorialsPoint"
"TutorialsPoint" : String
> location = "Hyderabad"
"Hyderabad" : String
> rating = 4.5
4.5 : Float
```

Here, the variables *company* and *location* are String variables and *rating* is a Float variable.

The elm REPL does not support type annotation for variables. The following example throws an error if the data type is included while declaring a variable.

```
C:\Users\dell\elm>elm repl
---- elm-repl 0.18.0 -----------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
-
> message:String
-- SYNTAX PROBLEM ------------------------------------------ repl-temp-
000.elm


A single colon is for type annotations. Maybe you want :: instead? Or maybe you
are defining a type annotation, but there is whitespace before it?


3|   message:String
             ^
Maybe <http://elm-lang.org/docs/syntax> can help you figure it out.
```

To insert a line break while using the elm REPL, use the \ syntax as shown below:

```
C:\Users\dell\elm>elm repl
---- elm-repl 0.18.0 -----------------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
-
> company \    -- firstLine
|   = "TutorialsPoint"  -- secondLine
"TutorialsPoint" : String
```

6. Elm — Operators

An operator defines some function that will be performed on the data. The values on which the operators work are called operands. Consider the following expression –

7 + 5 = 12

Here, the values 7, 5, and 12 are operands, while + and = are operators.

The major operators in Elm can be classified as:

- Arithmetic
- Relational
- Logical

Arithmetic Operators

Assume the values in variables a and b are 7 and 2 respectively.

Sr. No.	Operator	Description	Example
1	+(Addition)	returns the sum of the operands	a+b is 9
2	-(Subtraction)	returns the difference of the values	a-b is 5
3	* (Multiplication)	returns the product of the values	a*b is 14
4	/ (Float Division)	performs division operation and returns a float quotient	a / b is 3.5
5	//(Integer Division)	performs division operation and returns a integer quotient	a // b is 3
6	% (Modulus)	performs division operation and returns the remainder	a % b is 1

Illustration

Try the following example in REPL –

```
 > a = 7
7 : number
> b = 2
2 : number
> a+b
9 : number
> a-b
5 : number
> a*b
14 : number
> a/b
3.5 : Float
> a//b
3 : Int
> a % b
1 : Int
```

Relational Operators

Relational Operators test or define the kind of relationship between two entities. These operators are used to compare two or more values. Relational operators return a Boolean value, i.e. true or false.

Assume the value of *a* is 10 and *b* is 20.

Sr. No.	Operator	Description	Example
1	>	Greater than	(a > b) is False
2	<	Lesser than	(a < b) is True
3	>=	Greater than or equal to	(a >= b) is False
4	<=	Lesser than or equal to	(a <= b) is True
5	==	Equality	(a == b) is false
6	!=	Not equal	(a != b) is True

Illustration

Open the elm REPL and execute the following operations –

```
> a = 10
10 : number
> b = 20
20 : number
> a>b
False : Bool
> a<b
True : Bool
> a>=b
False : Bool
> a<=b
True : Bool
> a==b
False : Bool
> a/=b
True : Bool
```

Comparable Types

Comparison operators like >= or < work with comparable types. These are defined as numbers, characters, strings, and lists, tuples. The comparable types on both sides of the operator must be the same.

Sr. No.	Comparable Type	Example
1	number	7>2 gives True
2	character	'a' =='b' gives False
3	string	"hello" =="hello" gives True
4	tuple	(1,"One")==(1,"One") gives True
5	list	[1,2]==[1,2] gives True

Open the elm REPL and try the examples shown below –

```
C:\Users\admin>elm repl
---- elm-repl 0.18.0 -----------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
-
> 7>2
True : Bool
> 7.0>2
True : Bool
> 7.0<2.0
False : Bool
> 'a' > 'b'
False : Bool
> 'a' < 'b'
True : Bool
> "a" < "b"
True : Bool
> (1,2) > (2,3)
False : Bool
> ['1','3'] < ['2','1']
True : Bool
>
```

Logical Operators

Logical Operators are used to combine two or more conditions. Logical operators too return a Boolean value.

Sr. No.	Operator	Description	Example
1	&&	The operator returns true only if all the expressions specified return true	(10>5) && (20>5) returns True
2	||	The operator returns true if at least one of the expressions specified return true	(10 < 5) || (20 >5) returns True

Sr. No.	Operator	Description	Example
3	not	The operator returns the inverse of the expression's result. For E.g.: !(>5) returns false.	not (10 < 5) returns True
4	xor	The operator returns true only if exactly one input returns true. The operator returns false if both the expressions return true.	xor (10 > 5) (20 > 5) returns false

Illustration

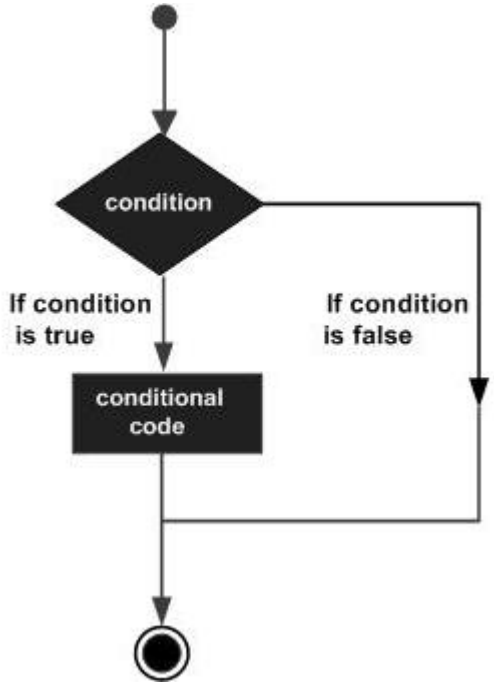
Open the elm REPL and try examples shown below

```
C:\Users\admin>elm repl
---- elm-repl 0.18.0 -----------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
-
> 10 > 5 || 20>5
True : Bool
> 10 >5 && 20>5
True : Bool
> 10 /= 20
True : Bool
> not True
False : Bool
> not (10>20)
True : Bool
> xor True True
False : Bool
> xor (10>20) (20>10)
True : Bool
```

7. Elm — Decision Making

Decision-making structures requires that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Shown below is the general form of a typical decision-making structure found in most of the programming languages –



A decision-making construct evaluates a condition before the instructions are executed. Decision-making constructs in Elm are classified as follows –

Sr. No.	Statement	Description
1	if...then...else statement	The if statement consists of a Boolean expression followed by then which is executed if the expression returns true and else which is executed if the expression returns false
2	nested if statement	You can use one if...then...else inside another if.
3	case statement	Tests the value of a variable against a list of values.

if...then...else Statement

The **if...then** construct evaluates a condition before a block of code is executed. If the Boolean expression evaluates to true, then the block of code inside the then statement will be executed. If the Boolean expression evaluates to false, then the block of code inside the else statement will be executed.

Unlike other programming languages, in Elm we must provide the else branch. Otherwise, Elm will throw an error.

Syntax

```
if boolean_expression then statement1_ifTrue else statement2_ifFalse
```

Illustration

Try the following example in the REPL terminal.

```
> if 10>5 then "10 is bigger" else "10 is small"
"10 is bigger" : String
```

Nested If

The nested if statement is useful for testing multiple conditions. The syntax of a nested if statement is given below:

```
if boolean_expression1 then statement1_ifTrue else if boolean_expression2 then
statement2_ifTrue else statement3_ifFalse
```

Illustration

Try the following example in the Elm REPL-

```
> score=80
80 : number
> if score>=80 then "Outstanding" else if score >= 70 then "good" else
"average"
"Outstanding" : String
```

Case statement

The case statement can be used to simplify the **if then else** statement. The syntax of a case statement is as given below:

```
case variable_name of
  constant1 -> Return_some_value
  constant2 -> Return_some_value
```

```
   _ -> Return_some_value if none of the above values match
```

The case statement checks if the value of a variable matches a predefined set of constants and returns the corresponding value. Note that value returned by each case must be of the same type. If the variables value does not match any of the given constants, the control is passed to * default * (denoted by //_) and the corresponding value is returned.

Illustration

Try the following example in the Elm REPL –

```
> n = 10
10 : number
> case n of \
|   0 -> "n is Zero" \
|   _ -> "n is not Zero"
"n is not Zero" : String
```

The above code snippet checks if the value of *n* is zero. The control is passed to *default*, which returns the string "n is not Zero".

8. Elm — Loop

Elm is a functional programming language. Elm uses the concept of recursion as an alternative to traditional looping constructs.

This chapter discusses the concept of recursion.

Recursion

Some computer programming languages allow a module or function to call itself. This technique is known as recursion.

Illustration

In this program, we will see how to use recursion to display hello five times.

Step 1: Create a file Loop.elm

Create a module Loop and define a function **sayHello**. The function sayHello takes an integer value as input and returns a string value.

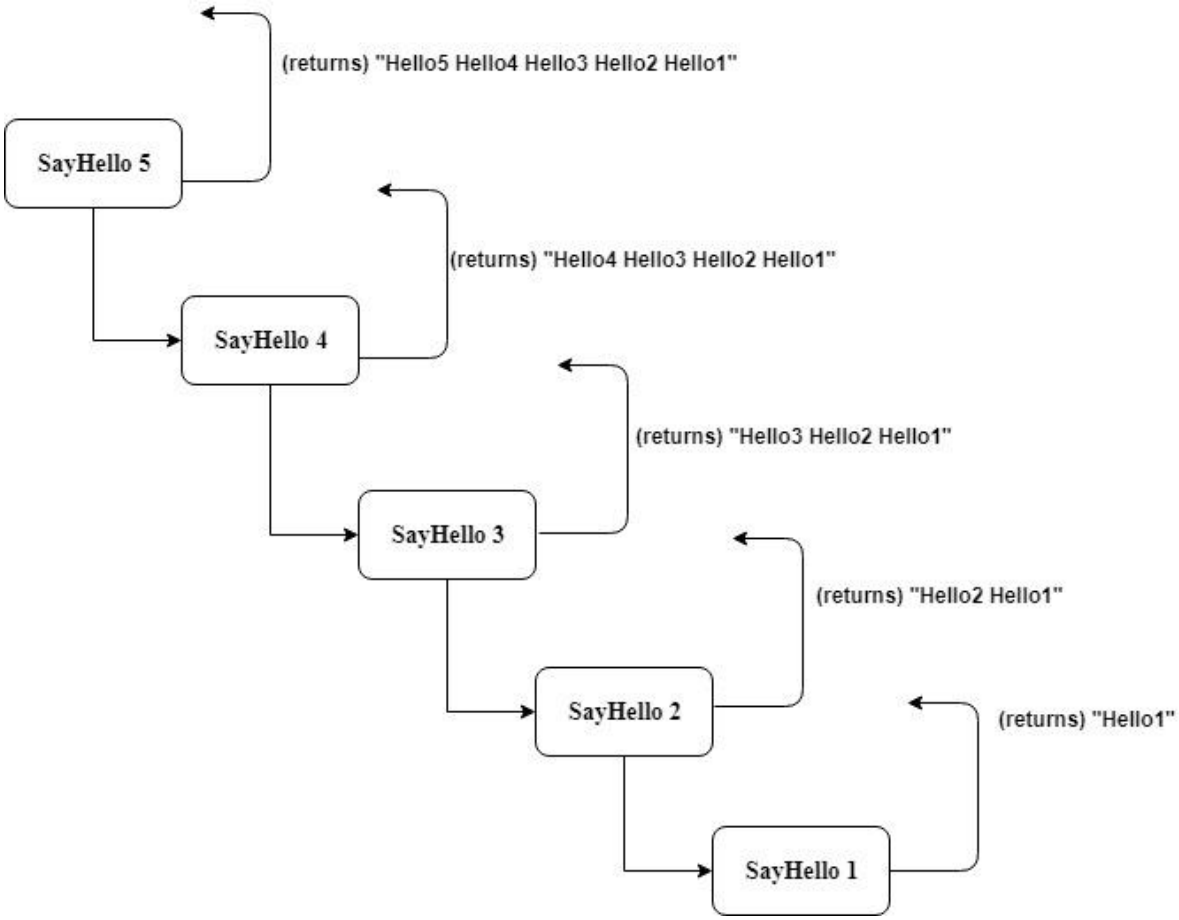
```
module Loop exposing(..)
//function signature
sayHello:Int ->String
//function implementation
sayHello n =
  case n of
   1 ->   "Hello:1 "
   _ -> "Hello:" ++ toString (n) ++ " " ++  sayHello(n-1)
```

The function sayHello checks if parameter passed is 1. If the parameter is 1, then function will return, otherwise it will create a string Hello and call the same function.

Step 2: Invoke sayHello from REPL

Open the elm REPL from current project folder (location of Loop.elm file).

```
//import the module Loop
> import Loop exposing(..)
//invoke the sayHello function with parameter value as 5
> sayHello 5
"Hello:5 Hello:4 Hello:3 Hello:2 Hello:1 Hello:0 " : String
>
```



Illustration

The following example prints the sum of n numbers using recursion.

```
>  sumOfNos n =\
|  if n==0 then 0 \
|   else (n) + sumOfNos (n-1)
<function> : number -> number1
```

In the elm REPL, we created a function *sumOfNos* that takes an input number and sums all numbers from 0 to that number.

For example, if we pass input as 5, it will sum up *1+2+3+4+5* which is *15*.

```
> sumOfNos 5
15 : number
```

The output of the program is shown above.

9. Elm — Functions

Functions are the building blocks of an Elm program. A function is a set of statements to perform a specific task.

Functions organize the program into logical blocks of code. Once defined, functions may be called to access code. This makes the code reusable. Moreover, functions make it easy to read and maintain the program's code.

Steps to using a function

There are three steps to using a function:

Function Declaration

A function declaration tells the compiler about a function's name, return type, and parameters. The syntax for declaring a function is given below:

```
fn_name:data_type_of_the_parameters ->return_type
```

The function declaration specifies the following:

- Name of the function.
- Data type of the parameters. This is optional as a function may or may not have parameters.
- Data type of the value, which the function will return. Functions in Elm must always return a value as Elm is a functional programming language. Unlike functions in other programing languages, Elm functions do not use the *return* keyword to return a value.

Function Definition or Function Implementation

A function definition provides the actual body of the function. A function definition specifies how a specific task would be done. The syntax for defining a function is as given below:

```
fn_name parameter1 parameter2 =
  statements
```

Invoking or Calling a Function

A function must be called so as to execute it. The syntax for calling a function is given below:

```
fn_name parameter1 parameter2
```

Illustration

The following code defines a function greet. The function returns a string "Hello".

```
> greet = \
|  if True then \
|    "Hello" \
|   else \
|     "GoodBye"
"Hello" : String
> greet
"Hello" : String
```

Parameterized Functions

Parameters are a mechanism to pass values to a function. The values of the parameters are passed to the function at the time of function invocation.

Illustration 1

The following example defines a function *fn_add*. The function accepts two numbers as parameters and returns their sum. Try the following in elm REPL –

```
> fn_add x y = x+y
<function> : number -> number -> number
> fn_add 10 20
30 : number
```

Illustration 2

The following example defines a function sayHello. The sayHello function accepts and returns a String value as parameter and returns a String.

```
> sayHello name = "Hello "++ name
<function> : String -> String
> sayHello "Tutorialspoint"
"Hello Tutorialspoint" : String
>
```

Pipe Operator

To understand pipe operator |>, let us consider an example where we have a list of different strings *["a","b","c"]* . Now we need a single string, which is separated by -.

The following example shows how to do that with *String.join*

```
> String.join "-" ["a","b","c","d","e","f"]
"a-b-c-d-e-f" : String
```

The same action can be performed by using a pipe operator |>. The pipe operator can be used to chain multiple function calls.

```
> ["a","b","c","d","e","f"] |> String.join "-"
"a-b-c-d-e-f" : String
> ["a","b","c","d","e","f"] |> List.reverse |> String.join "-"
"f-e-d-c-b-a" : String
```

In the first example, we are chaining the list to join method. In the second case, the same list is piped to reverse function and thereafter piped to join. So, the list is displayed in reversed and joined.

10. Elm — String

A sequence of Unicode characters is called a String. In Elm, strings are enclosed in "" *double quotes*. A *String* is a chunk of text as shown below.

```
> "TutorialsPoint"
"TutorialsPoint" : String
> location = "Hyderabad"  --variable
"Hyderabad" : String
> location
"Hyderabad" : String
>
```

String Functions

Some common functions that can be used to query or manipulate string values are given below. Use REPL to try the examples given below.

Sr. No.	Method	Description
1	isEmpty : String -> Bool	checks string is empty
2	reverse : String -> String	reverses a input string
3	length : String -> Int	returns an integer length
4	append :String -> String -> String	appends two string and returns a new string
5	concat : List String -> String	appends a list of strings and returns a new string
6	split : String -> String -> List String	splits an input string using a given separator, returns a string list
7	slice : Int -> Int -> String -> String	returns a substring given a start , end index and input string
8	contains : String -> String -> Bool	returns true if second string contains the first one

Sr. No.	Method	Description
9	toInt : String -> Result.Result String Int	parses a String to Integer
10	toInt : String -> Result.Result String Int	parses a String to Integer
11	toFloat : String -> Result.Result String Float	parses a String to float
12	fromChar : Char -> String	creates a string from a given character.
13	toList : String -> List Char	converts string to list of characters
14	fromList : List Char -> String	converts a list of characters into a String
15	toUpper : String -> String	converts input string to upper case
16	trim : String -> String	gets rid of whitespace on both sides of a string.
17	filter : (Char -> Bool) -> String -> String	filters set of characters from input string
18	map : (Char -> Char) -> String -> String	transforms every character in an input string

isEmpty

This function can be used to determine if a string is empty. This function returns True if the supplied String is empty.

Syntax

```
String.isEmpty String_value
```

To check the signature of function, type the following in elm REPL –

```
> String.isEmpty
<function> : String -> Bool
```

Signature of the function shows Bool as return type and input type as String –

Illustration

```
> String.isEmpty ""
True : Bool
> String.isEmpty "Tutorialspoint"
False : Bool
> location = "Hyderabad"
"Hyderabad" : String
> String.isEmpty location
False : Bool
```

reverse

This function reverses a string.

Syntax

```
String.reverse String_value
```

To check the signature of function, type the following in elm REPL –

```
> String.reverse
<function> : String -> String
```

Signature of the function shows String as return type and input type as String –

Illustration

```
> String.reverse "TutorialsPoint"
"tnioPslairotuT" : String
```

length

This function returns the length of a string.

Syntax

```
String.length String_value
```

To check the signature of function, type the following in elm REPL –

```
> String.length
<function> : String -> Int
```

Signature of the function shows Int as return type and input type as String.

Illustration

```
> String.length "Mohtashim"
9 : Int
```

append

This function returns a new string by appending two strings.

Syntax

```
String.append String_value1 String_value2
```

To check the signature of function, type the following in elm REPL –

```
> String.append
<function> : String -> String -> String
```

Signature of shows two String input parameters and one String output parameter

Illustration

```
> String.append "Tutorials" "Point"
TutorialsPoint : String
```

concat

This function returns a new string by concatenating many strings into one.

Syntax

```
String.concat [String1,String2,String3]
```

To check the signature of function, type the following in elm REPL –

```
> String.concat
<function> : List String -> String
```

Signature of shows a List of String input parameter and String return type

Illustration

```
> String.concat ["Hello","Tutorials","Point"]
HelloTutorialsPoint : String
```

split

This function splits a string using a given separator.

Syntax

```
String.split string_seperator String_value
```

To check the signature of function, type the following in elm REPL –

```
> String.split
<function> : String -> String -> List String
```

Signature of shows two input String parameters and output as a list of string type.

Illustration

```
> String.split "," "Hello,Tutorials,Point"
["Hello","Tutorials","Point"] : List String
```

slice

This function returns a substring given a start and end index. Negative indexes are taken starting from the end of the list. The value of the index starts from zero.

Syntax

```
String.slice start_index end_index String_value
```

To check the signature of function, type the following in elm REPL –

```
> String.slice
<function> : Int -> Int -> String -> String
```

Signature of shows three input parameter and one return type

Illustration

```
> String.slice 0 13 "TutorialsPoint"
"TutorialsPoin" : String
```

contains

This function returns a True if the second string contains the first one.

Syntax

```
String.contains string1 string2
```

To check the signature of function, type the following in elm REPL –

```
> String.contains
<function> : String -> String -> Bool
```

Signature of shows bool return type and two input parameters

Illustration

```
> String.contains "Point" "TutorialsPoint"
True : Bool
```

toInt

This function converts a string into an int.

Syntax

```
String.toInt string_value
```

To check the signature of function, type the following in elm REPL –

```
> String.toInt
<function> : String -> Result.Result String Int
```

Since toInt can return error, the return type is Result, which is String or Int.

Illustration

```
> String.toInt "20"
Ok 20 : Result.Result String Int
> String.toInt "abc"
Err "could not convert string 'abc' to an Int" : Result.Result String Int
```

toFloat

This function converts a string into a float.

Syntax

```
String.toFloat string_value
```

To check the signature of function, type the following in elm REPL –

```
> String.toFloat
<function> : String -> Result.Result String Float
```

Since toFloat can return error, the return type is Result, which is String or Float.

Illustration

```
> String.toFloat "20.50"
Ok 20.5 : Result.Result String Float
> String.toFloat "abc"
Err "could not convert string 'abc' to a Float" : Result.Result String Float
```

fromChar

This function creates a string from a given character.

Syntax

```
String.fromChar character_value
```

To check the signature of function type following in elm REPL –

```
> String.fromChar
<function> : Char -> String
```

The signature shows String as return type and input as Char type

Illustration

```
> String.fromChar 'c'
"c" : String
```

toList

This function converts a string to a list of characters.

Syntax

```
String.toList string_value
```

To check the signature of function, type the following in elm REPL –

```
> String.toList
<function> : String -> List Char
```

The signatures shows function returns a list of characters and takes input a string.

Illustration

```
> String.toList "tutorialspoint"
['t','u','t','o','r','i','a','l','s','p','o','i','n','t'] : List Char
```

fromList

This function converts a list of characters into a String.

Syntax

```
String.fromList list_of_characters
```

To check the signature of function, type the following in elm REPL –

```
> String.fromList
<function> : List Char -> String
```

Illustration

```
> String.fromList ['h','e','l','l','o']
"hello" : String
```

toUpper

This function converts a string to all upper case.

Syntax

```
String.toUpper String_value
```

To check the signature of function, type the following in elm REPL –

```
> String.toUpper
<function> : String -> String
```

Illustration

```
> String.toUpper "hello"
"HELLO" : String
```

toLower

This function converts a string to all lower case.

Syntax

```
String.toLower String_value
```

To check the signature of function, type the following in elm REPL –

```
> String.toLower
<function> : String -> String
```

Illustration

```
> String.toLower "AbCd"
"abcd" : String
```

trim

This function gets rid of whitespace on both sides of a string.

Syntax

```
 String.trim String_value
```

To check the signature of function, type the following in elm REPL –

```
> String.trim
<function> : String -> String
```

Illustration

```
> String.trim "tutorialspoint   "
"tutorialspoint" : String
```

filter

This function filters a set of characters from input String. Keep only the characters that pass the test.

Syntax

```
 String.filter test_function string_value
```

To check the signature of function, type the following in elm REPL –

```
> String.filter
<function> : (Char -> Bool) -> String -> String
```

The signature shows filter takes two input parameters and returns a String. The first parameter is a function, which has input Char and returns Bool.

Illustration

In the example, we are passing *Char.isUpper* as parameter to filter method; it returns all upper-case characters as shown below.

```
> import Char
> String.filter Char.isUpper "abcDEF"
```

```
"DEF" : String
```

map

This function takes a String and transforms every character in a string.

Syntax

```
 String.filter mapping_function string_value
```

To check the signature of function, type the following in elm REPL –

```
> String.map
<function> : (Char -> Char) -> String -> String
```

Illustration

The following example replaces the character o with @ –

```
> String.map (\c -> if c == 'o' then '@' else c)  "TutorialsPoint"
"Tut@rialsP@int" : String
```

11. Elm — List

The List, Tuples and Record data structures can be used to store a collection of values. This chapter discusses how to use List in Elm.

A List is a collection of homogeneous values. The values in a list must all be of the same data type.

Consider the following limitations while using variables to store values:

- Variables are scalar in nature. In other words, at the time of declaration a variable can hold only one value. This means that to store **n** values in a program, **n** variable declarations will be needed. Hence, the use of variables is not feasible when one needs to store a larger collection of values.
- Variables in a program are allocated memory in random order, thereby making it difficult to retrieve/read the values in the order of their declaration.

Syntax

```
List_name = [value1,value2,value3.....valuen]
```

Illustration

The following example shows how to use a List in Elm. Try this example in elm REPL –

```
> myList1 = [10,20,30]
[10,20,30] : List number
> myList2 = ["hello","world"]
["hello","world"] : List String
```

If we try adding values of different types into a list, the compiler will throw a type mismatch error. This is shown below.

```
> myList = [1,"hello"]
-- TYPE MISMATCH ---------------------------------------- repl-temp-
000.elm

The 1st and 2nd entries in this list are different types of values.

4|          [1,"hello"]
               ^^^^^^^
The 1st entry has this type:
```

```
    number

But the 2nd is:

    String
```

List operations

Following table shows the common operations on a List:

Sr. No.	Method	Description
1	isEmpty : List a -> Bool	checks if list is empty
2	reverse : List a -> Bool	reverses input list
3	length : List a -> Int	returns size of the list
4	maximum : List comparable -> Maybe.Maybe comparable	returns maximum value
5	minimum : List comparable -> Maybe.Maybe comparable	returns minimum value
6	sum : List number -> number	returns sum of all elements in list
7	product : List number -> number	checks if list is empty
8	sort : List comparable -> List comparable	sorts list in ascending order
9	concat : List (List a) -> List a	merges a bunch of list into one
10	append : List a -> List a -> List a	merges two lists together
11	range : Int -> Int -> List Int	returns a list of numbers from start to end

Sr. No.	Method	Description
12	filter : (a -> Bool) -> List a -> List a	filters list of values from input list
13	head : List a -> Maybe.Maybe a	returns the first element from list
14	tail : : List a -> Maybe.Maybe (List a)	returns all elements except the head

isEmpty

This function returns true if a list is empty.

Syntax

```
List.isEmpty list_name
```

To check the signature of function, type the following in elm REPL –

```
> List.isEmpty
<function> : List a -> Bool
```

Illustration

```
> List.isEmpty
<function> : List a -> Bool

> List.isEmpty [10,20,30]
False : Bool
```

reverse

This function reverses the list.

Syntax

```
List.reverse list_name
```

To check the signature of function, type the following in elm REPL –

```
> List.reverse
<function> : List a -> List a
```

Illustration

```
> List.reverse [10,20,30]
[30,20,10] : List number
```

length

This function returns the length of a list.

Syntax

```
List.length list_name
```

To check the signature of function, type the following in elm REPL –

```
>List.length
<function> : List a -> Int
```

Illustration

```
> List.length [10,20,30]
3 : Int
```

maximum

This function returns the maximum element in a non-empty list.

Syntax

```
List.maximum list_name
```

To check the signature of function, type the following in elm REPL –

```
> List.maximum
<function> : List comparable -> Maybe.Maybe comparable
```

Illustration

```
> List.maximum [10,20,30]
Just 30 : Maybe.Maybe number
> List.maximum []
```

```
Nothing : Maybe.Maybe comparable
```

minimum

This function returns the minimum element in a non-empty list.

Syntax

```
List.minimum list_name
```

To check the signature of function, type the following in elm REPL –

```
> List.minimum
<function> : List comparable -> Maybe.Maybe comparable
```

Illustration

```
> List.minimum [10,20,30]
Just 10 : Maybe.Maybe number
```

sum

This function returns the sum of all elements in a list.

Syntax

```
List.sum list_name
```

To check the signature of function, type the following in elm REPL –

```
> List.sum
<function> : List number -> number
```

Illustration

```
> List.sum [10,20,30]
60 : number
```

product

This function returns the product of all elements in a list.

Syntax

```
List.product list_name
```

To check the signature of function, type the following in elm REPL –

```
<function> : List number -> number
```

Illustration

```
 List.product [10,20,30]
6000 : number
```

sort

This function sorts values from lowest to highest in a list.

Syntax

```
List.sort list_name
```

To check the signature of function, type the following in elm REPL –

```
> List.sort
<function> : List comparable -> List comparable
```

Illustration

```
> List.sort [10,20,30]
[10,20,30] : List number
```

concat

This function concatenates a bunch of lists into a single list.

Syntax

```
List.concat [ [list_name1],[list_name2],[list_name3],.....[list_nameN] ]
```

To check the signature of function, type the following in elm REPL

```
> List.concat
<function> : List (List a) -> List a
```

Illustration

```
> List.concat [[10,20], [30,40],[50,60]]
```

```
[10,20,30,40,50,60] : List number
```

append

This function puts two lists together.

Syntax

```
List.append  [list_name1] [list_name2]
```

To check the signature of function, type the following in elm REPL –

```
>List.append
<function> : List a -> List a -> List a
```

Illustration

```
> List.append  [10,20] [30,40]
[10,20,30,40] : List number
```

The ++ operator can also be used to append a list to another. This is shown in the example below:

```
> [10.1,20.2] ++ [30.3,40.4]
[10.1,20.2,30.3,40.4] : List Float
```

range

This function creates a list of numbers, every element increasing by one. The lowest and the highest number that should be in the list is passed to the function.

Syntax

```
List.range start_range end_range
```

To check the signature of function, type the following in elm REPL –

```
> List.range
<function> : Int -> Int -> List Int
```

Illustration

```
> List.range 1 10
[1,2,3,4,5,6,7,8,9,10] : List Int
```

filter

This function filters a set of values from input list. Keep only the values that pass the test.

Syntax

```
List.filter test_function input_list
```

To check the signature of function, type the following in elm REPL –

```
> List.filter
<function> : (a -> Bool) -> List a -> List a
```

Illustration

Following example filters all even numbers from an input list

```
> List.filter (\n -> n%2==0) [10,20,30,55]
[10,20,30] : List Int
```

head

This function returns the first element from input list.

Syntax

```
List.head  input_list
```

To check the signature of function, type the following in elm REPL –

```
> List.head
<function> : List a -> Maybe.Maybe a
```

Illustration

```
> List.head [10,20,30,40]
Just 10 : Maybe.Maybe number
> List.head []
Nothing : Maybe.Maybe a
```

tail

This function returns all elements after first in the list.

Syntax

```
List.tail  input_list
```

To check the signature of function, type the following in elm REPL

```
> List.tail
<function> : List a -> Maybe.Maybe (List a)
```

Illustration

```
> List.tail [10,20,30,40,50]
Just [20,30,40,50] : Maybe.Maybe (List number)
> List.tail [10]
Just [] : Maybe.Maybe (List number)
> List.tail []
Nothing : Maybe.Maybe (List a)
```

Using the Cons Operator

The cons operator (::) adds an element to the front of a list.

Illustration

```
> 10::[20,30,40,50]
[10,20,30,40,50] : List number
```

The new element to be added and the data-type of the values in the list must match. The compiler throws an error if the data types do not match.

```
> [1,2,3,4]::[5,6,7,8]
-- TYPE MISMATCH ---------------------------------------- repl-temp-
000.elm


The right side of (::) is causing a type mismatch.


3|   [1,2,3,4]::[5,6,7,8]
                ^^^^^^^^^
(::) is expecting the right side to be a:


    List (List number)


But the right side is:
```

```
    List number


Hint: With operators like (::) I always check the left side first. If it seems
fine, I assume it is correct and check the right side. So the problem may be in
how the left and right arguments interact.
```

Lists are immutable

Let us check if lists are immutable in Elm. The first list *myList* when concatenated with value 1 creates a new list and is returned to *myListCopy*. Therefore, if we display initial list, its values will not be changed.

```
> myList = [10,20,30]
[10,20,30] : List number
> myListCopy = 1::myList
[1,10,20,30] : List number
> myList
[10,20,30] : List number
>myList == myListCopy
False : Bool
```

12. Elm — Tuples

At times, there might be a need to store a collection of values of varied types. Elm gives us a data structure called tuple that serves this purpose.

A tuple represents a heterogeneous collection of values. In other words, tuples enable storing multiple fields of different types. A tuple stores fixed number of values. Tuples are useful when you want to return multiple values of different types from a function. These data structures are immutable like other types in elm.

Syntax

```
(data1,data2)
```

A simple example is shown below –

```
> ("TuotrialsPoint",5,True,"Hyderabad")
("TuotrialsPoint",5,True,"Hyderabad") : ( String, number, Bool, String )
```

In our subsequent sections, we will learn about the different tuple operations.

first

This operation extracts the first value from a tuple.

Syntax

```
Tuple.first tuple_name
> Tuple.first
<function> : ( a1, a2 ) -> a1
```

Illustration

```
> Tuple.first (10,"hello")
10 : number
```

second

The **second** tuple operation extracts the second value from a tuple.

Syntax

```
Tuple.second tuple_name
> Tuple.second
```

```
<function> : ( a1, a2 ) -> a2
```

Illustration

```
 > Tuple.second (10,"hello")
"hello" : String
```

List of tuples

A List can store Tuples. If tuples are used inside a list, make sure they all are of the same data type and have the same number of parameters.

Illustration

```
> [("hello",20),("world",30)]
[("hello",20),("world",30)] : List ( String, number )
```

Tuple with function

A function can return tuples. In addition, tuples can be passed as parameters to functions.

Illustration 1

The following example defines a function *fn_checkEven*. This function accepts an integer value as parameter and returns a tuple.

```
> fn_checkEven no = \
    if no%2 == 0 then \
      (True,"The no is Even")\
    else \
      (False,"No is not even")
<function> : Int -> ( Bool, String )
> fn_checkEven 10
(True,"The no is Even") : ( Bool, String )
> fn_checkEven 11
(False,"No is not even") : ( Bool, String )
>
```

Illustration 2

The following passes a tuple as a parameter to a function.

```
> fn_add (a,b) = \
|   a+b
```

```
<function> : ( number, number ) -> number
> fn_add (10,20)
30 : number
```

The function *fn_add* takes a tuple with 2 numeric values and returns their sum.

Destructuring

Destructuring involves breaking a tuple into individual values. To access individual values in a tuple with three or more elements, we use destructuring. Here, we assign each value in a tuple to different variables. Using _ one can define placeholders for values that will be ignored or skipped.

Illustration

```
 > (first,_,_) = (10,20,30)
10 : number
> first
10 : number
```

Illustration

In this example, we wil use *let..in* block syntax to destructure. The let block contains the variables and the in block contains expressions that should be evaluated and value that should be returned.

```
> t1 = (10,20,30)
(10,20,30) : ( number, number1, number2 )
> let \
   (a,b,c) = t1 \
  in\
   a + b +c
60 : number
```

We are declaring variables *a b c* in *let* clause and accessing them using *in* clause.

13. Elm — Records

The record data structure in Elm can be used to represent data as key-value pairs. A record can be used to organize related data to enable easy access and updating data. Elm records are similar to objects in JavaScript. Data elements in a record are known as fields.

Defining a Record

Use the following syntax to define a record –

Syntax

```
record_name= {fieldname1=value1, fieldname2=value2....fieldnameN=valueN}
```

A record can store data of multiple types. The field names in a record must conform to the general rules for naming an Elm identifier.

Accessing record values

Use the following syntax to access individual fields in a record.

Syntax

```
record_name.fieldname
```

OR

```
.fieldname record_name
```

Illustration

Try the following in the Elm REPL-

```
> company = {name="TutorialsPoint",rating=4.5}
{ name = "TutorialsPoint", rating = 4.5 } : { name : String, rating : Float }
> company.name
"TutorialsPoint" : String
> .rating company
4.5 : Float
```

Using Record with List

A record can be stored inside a list. All field values of the record should be of the same type.

Syntax

```
list_name =[ {field_name1=value1},{field_name1=value2}]
```

OR

```
list_name=[record_name1, record_name2, record_name3....record_nameN]
```

Illustration

Try the following in Elm REPL –

```
> [{name="Mohtashim"},{name="kannan"}]
[{ name = "Mohtashim" },{ name = "kannan" }] : List { name : String }
> record1={name="FirstRecord"}
{ name = "FirstRecord" } : { name : String }
> record2={name="SecondRecord"}
{ name = "SecondRecord" } : { name : String }
> recordList=[record1,record2]
[{ name = "FirstRecord" },{ name = "SecondRecord" }] : List { name : String }
```

Update a Record

Records are immutable in Elm. When a record is updated, a new record with updated values is returned. The field can hold value of a different type when updating a record.

Syntax

```
{record_name | field_name1= new_value1, field_name2= new_value2,field_name3=
new_value3....field_nameN= new_valueN}
```

Illustration

Try the following in Elm REPL –

```
> record1={name="FirstRecord"}
{ name = "FirstRecord" } : { name : String }
> record1_updated = {record1 | name="FirstRecordUpdate"}
{ name = "FirstRecordUpdate" } : { name : String }
> record1
{ name = "FirstRecord" } : { name : String }
> record1 == record1_updated
False : Bool
```

Illustration

The following example updates multiple fields of a record. Try the following in Elm REPL –

```
> record3 = {a=1,b=2,c=3,d=4,e=5}
{ a = 1, b = 2, c = 3, d = 4, e = 5 }
    : { a : number, b : number1, c : number2, d : number3, e : number4 }
> record4 = {record3 | d=400 ,e=500}
{ a = 1, b = 2, c = 3, d = 400, e = 500 }
    : { a : number2, b : number3, c : number4, d : number, e : number1 }
>
```

Types alias

Type alias defines a schema for a record. In other words, a type alias defines which fields can the record store and the type of value these fields can store. Therefore, programmer will not make mistake of missing any specific attribute while assigning values.

Syntax

```
type alias
alias_name={field_name1:data_type,field_name2:data_type,....field_nameN:data_ty
pe}
```

Illustration

Execute the following in Elm REPL –

```
> type alias Developer = { name:String,location:String,age:Int}
> dev1 = Developer "kannan" "Mumbai" 20
{ name = "kannan", location = "Mumbai", age = 20 } : Repl.Developer
> dev2 = Developer "mohtashim" "hyderabad" 20
{ name = "mohtashim", location = "hyderabad", age = 20 } : Repl.Developer
```

Now if you forget to type location and age, the statement returns a function, which has input parameters for location and age fields.

```
> dev3 = Developer "Bhagavati"
<function> : String -> Int -> Repl.Developer
We can invoke the function as shown below and pass to it the values for
location and age fields.
 > dev3 "Pune" 25
{ name = "Bhagavati", location = "Pune", age = 25 } : Repl.Developer
```

14. Elm — Error Handling

An error is any unexpected condition in a program. Errors can occur at either compile-time or runtime. Compile time errors occur during the compilation of a program (For example, error in the program's syntax) while runtime errors occur during the program's execution. Unlike other programming languages, Elm does not throw runtime errors.

Consider an application that accepts the age of a user. The application should throw an error if the age is zero or negative. In this case, the Elm application can use the concept of error handling to explicitly raise an error at runtime if the user enters zero or a negative value as age. Error handling specifies the course of action if anything unexpected happens during the program's execution.

Elm programming language handles errors in the following ways –

- MayBe
- Result

MayBe

Consider the search feature in an application. The search function returns related data if the search keyword is found else does not return anything. This use case can be implemented in Elm using the MayBe type.

Syntax

```
variable_name:MayBe data_type
```

A variable of type MayBe can contain either of the following values –

- Just some_Value: This is used if there is valid data.
- Nothing: This is used if the value is absent or unknown. Nothing is equivalent to null in other programming languages.

Illustration

The following example shows how to use MayBe type with variables and function.

Step 1: Create a **MayBeDemo.elm** file and add the following code to it –

```
--  MayBeDemo.elm
module MayBeDemo exposing(..)
import Maybe

--declaring  a MayBe variable and assigning value to it
userName : Maybe String
```

```
userName = Just "Mohtashim"


--declaring  a MayBe variable and assigning value to it
userAge :Maybe Int
userAge = Just 20


--declaring  a MayBe variable and assigning value to it
userSalary:Maybe Float
userSalary = Nothing


--declaring  a custom type
type Country = India | China | SriLanka


--defining a function that takes a String parameter as input and returns a
value of type MayBe


getCountryFromString : String -> Maybe Country
getCountryFromString p =
case p of
  "India"
    -> Just India
  "China"
    -> Just China
  "SriLanka"
    -> Just SriLanka
  _
    ->  Nothing
```

Step 2: Import the module in elm repl and execute as given below –

```
 E:\ElmWorks\ErroApp> elm repl
---- elm-repl 0.18.0 -----------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
-
> import MayBeDemo exposing(..)
> userName
```

```
Just "Mohtashim" : Maybe.Maybe String
> userAge
Just 20 : Maybe.Maybe Int
> userSalary
Nothing : Maybe.Maybe Float
> getCountryFromString "India"
Just India : Maybe.Maybe MayBeDemo.Country
> getCountryFromString "india"
Nothing : Maybe.Maybe MayBeDemo.Country
```

The function checks if the value passed to the function is India or China or SriLanka. If the parameter's value does not match any of these, it returns nothing.

Result

Consider an example, where the application needs to validate some condition and raise an error if the condition is not satisfied. The Result type can be used to achieve this. The Result type should be used if the application wants to explicitly raise an error and return details about what went wrong.

Syntax

The Result type declaration takes two parameters – the data type of the error (usually String) and the data type of the result to be returned if everything goes fine.

```
type Result error_type data_value_type
= Ok data_value
| Err error_message
```

The Result type returns either of the following values-

- Ok some_value: Represents result to be returned
- Err: Represents the error message to be returned if the expected conditions are not satisfied.

Illustration 1

Try the following example in the Elm REPL –

```
> String.toInt
<function> : String -> Result.Result String Int
-- successful result
> String.toInt "10"
Ok 10 : Result.Result String Int
-- unsuccessful result , Error
> String.toInt "a"
Err "could not convert string 'a' to an Int" : Result.Result String Int
```

The String.toInt function returns Integer value if the parameter passed is valid. If the parameter is not a number, the function returns an error.

Illustration 2

The following example accepts age as a parameter. The function returns the age if it is between 0 and 135 else it returns an appropriate error message.

Step 1: Create a ResultDemo.elm file and add the following code to it.

```
--ResultDemo.elm
module ResultDemo exposing(..)

userId : Result String Int
userId = Ok 10

emailId : Result String Int
emailId = Err "Not valid emailId"

isReasonableAge : String -> Result String Int
isReasonableAge input =
  case String.toInt input of
    Err r ->
      Err "That is not a age!"

    Ok age ->
      if age < 0 then
        Err "Please try again ,age can't be negative"
```

```
      else if age > 135 then
        Err "Please try agian,age can't be this big.."

      else
        Ok age
```

Step 2: Import the module in elm package and execute as given below –

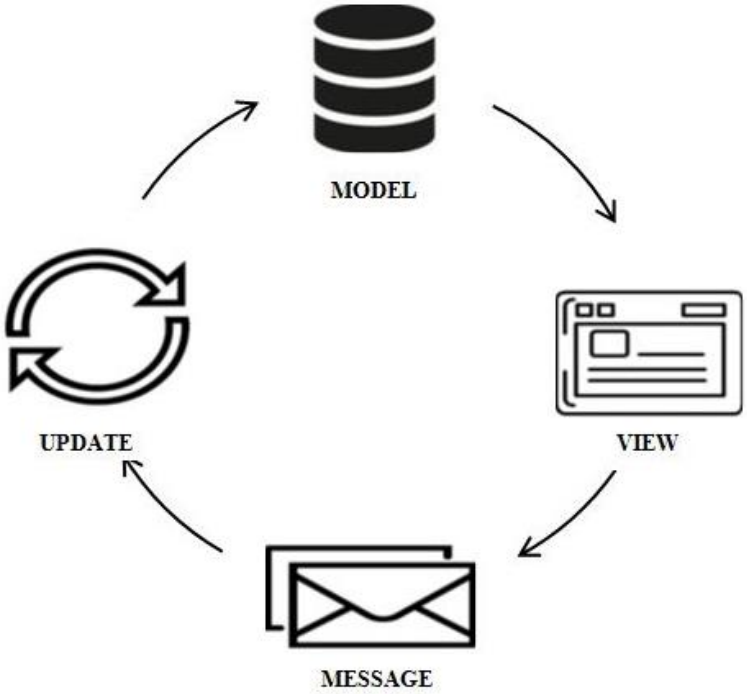
```
E:\ElmWorks\ElmRepo\15_ErrorHandling\15_Code> elm repl
---- elm-repl 0.18.0 -----------------------------------------------------
-
 :help for help, :exit to exit, more at <https://github.com/elm-lang/elm-repl>
--------------------------------------------------------------------------------
-
> import ResultDemo exposing (..)

> userId
Ok 10 : Result.Result String Int
> emailId
Err "Not valid emailId" : Result.Result String Int

> isReasonableAge "10"
Ok 10 : Result.Result String Int

> isReasonableAge "abc"
Err "That is not a age!" : Result.Result String Int
```

15. Elm — Architecture

In this chapter, we will discuss the standard way to create applications in Elm platform. Elm uses an architectural pattern similar to Model-View-Controller pattern.

Following are the four main parts of Elm Architecture.

- Model
- View
- Message
- Update



How does the Elm architecture work?

The **model** contains the application state. For example, if an application displays a list of customers then the state will contain each customer data. To display the state in a presentable way, a **view**/html has to be generated. Once the user interacts with view by pressing a button or typing data in a form, view generates signals called **messages**. Messages are passed to the **update** method, which evaluates the messages and takes proper action. Therefore, the update method will generate a new model.

The new model generates a new view. The view will lead to new interactions from user to signal messages, that goes to update function. Further, the function creates a new model. So, the cycle repeats as shown in the above diagram.

Model

Model deals with the application's state. The syntax for defining a Model is given below –

```
-- Model syntax

type alias Model = {
   property1:datatype,
   proptery2:datatype
   ...
 }
```

To create a model, we need to first create a template with all property required in it. Each property specifies the state of the application.

View

View is a visual representation of the application state. The View knows how to take data and generate web page out of it. When a user interacts with the View, the user can manipulate the state by generating messages. The syntax for defining a View is given below –

```
--View Syntax
view model =some_implementation
```

Message

Message is a request from the user to alter the application state. Messages are passed as parameter to the update function.

```
--Message Syntax
type Message = Message1 |Message2 ...
```

The syntax shows a type Message. The elm application will edit the state based on messages passed to it. These decisions are made in the update method.

Update

The update function interprets the messages, which are passed as parameter to it, and updates the model.

```
--Update Syntax
update Message_type model =
  some_implementation
```

The update function takes **Message** and Model as parameters.

16. Elm — Package Manager

A package manager is a command-line tool that automates the process of installing, upgrading, configuring, and removing packages in your application.

Just like JavaScript has a package manager called *npm*, elm has a package manager called *elm-package*.

The package manager performs the following three tasks:

- Installs all dependencies that an elm application need
- Publishes custom packages
- Determines the version of your package when you are ready to publish and update.

Elm Package Manager Commands

The following table lists down the various Elm package manager commands:

Sr. No.	Command	Syntax	Description
1	install	elm-package install	Installs packages to use locally
2	publish	elm-package publish	Publishes your package to the central catalog
3	bump	elm-package bump	Bumps version numbers based on API changes
4	diff	elm-package diff	Gets differences between two APIs

In order to publish your package, you need to host source code on GitHub and have the version properly labeled with a git tag. Following illustration shows how to use elm-package manager to pull an external dependency.

Illustration - Installing svg package

In this example, we will see how to integrate Scalable Vector Graphics(SVG) into an elm application.

Step 1: Create a folder elmSvgApp

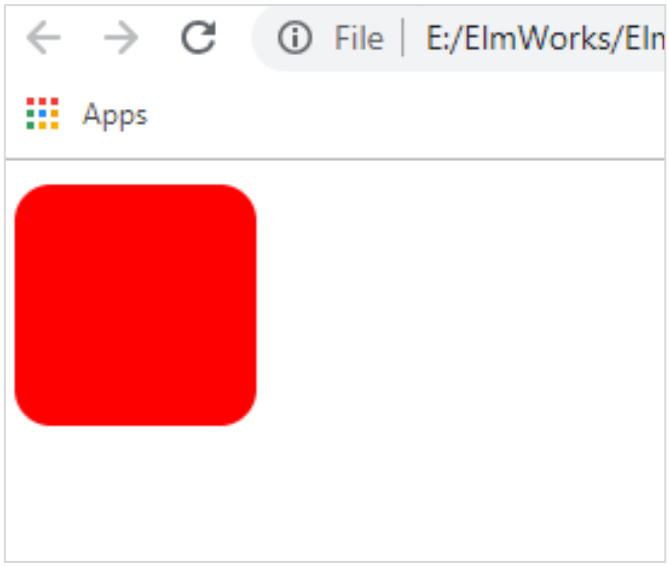
Step 2: Install svg package using the following command:

```
elm-package install elm-lang/svg
```

Step 3: Create a SvgDemo.elm file and type the content given below. We import Svg module to draw a rectangle of *100x100* dimension and fill the colour red.

```
import Svg exposing (..)
import Svg.Attributes exposing (..)


main =
  svg
    [ width "120"
    , height "120"
    , viewBox "0 0 120 120"
    ]
    [ rect
        [ x "10"
        , y "10"
        , width "100"
        , height "100"
        , rx "15"
        , ry "15"
        ,fill "red"
        ]
        []
    ]
```

Step 4: Now build the project using *elm make .\SvgDemo.*elm. This will generate an index.html as shown below:



17. Elm — Messages

Message is a component in the Elm architecture. These components are generated by the View in response to the user's interaction with the application's interface. Messages represent user requests to alter the application's state.

Syntax

```
--Message Syntax
type Message = some_message1 |some_message2 ...|some_messageN
```

Illustration

The following example is a simple counter application. The application increments and decrements the value of a variable by 1 when the user clicks on the Add and Subtract buttons respectively.

The application will have 4 components. The components are described below -

Message

The messages for this example will be –

```
type Message =  Add | Subtract
```

Model

The model represents the state of the application. In the counter application the model definition is given below; the initial state of counter will be zero.

```
model = 0
```

View

The view represents the visual elements of the application. The view contains two buttons **(+)** and **(-)** . The messages *Add* and *Subtract* are generated by the View when the user clicks on the + and - buttons respectively. The modified value of the model is then displayed by the View.

```
view model =
-- invoke text function
h1[]
 [   div[] [text "CounterApp from TutorialsPoint" ]
     ,button[onClick Subtract] [text "-"]
     ,div[][text (toString  model)]
```

```
    ,button[onClick Add] [text "+"]
 ]
```

Update

This component contains code that should be executed for each message generated by the view. This is shown in the example below:

```
   update msg model =
 case msg of
   Add -> model+1
   Subtract -> model-1
```

Putting it all together

Step 1: Create a folder MessagesApp and file *MessagesDemo.elm*

Step 2: Add the following code in elm file:

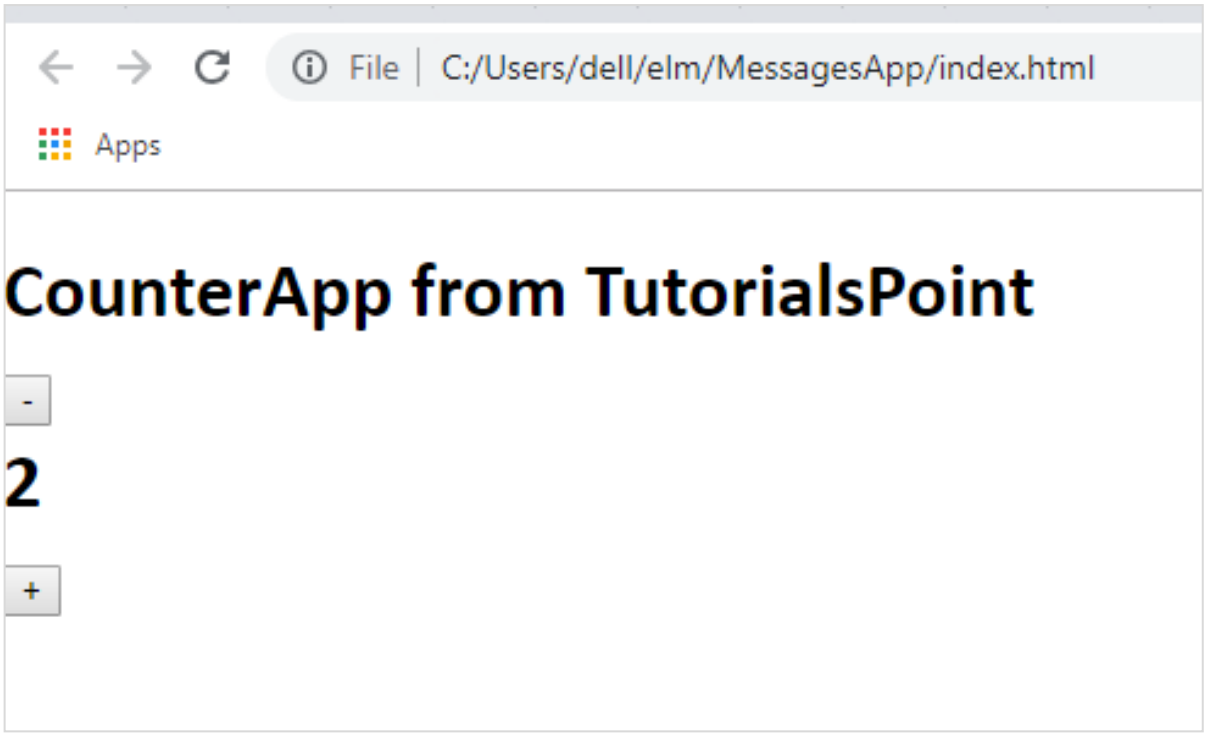
```
import Html exposing (..)
import Html.Events exposing(onClick)

model = 0     -- Defining the Model

--Defining the View

view model =
 h1[]
  [   div[] [text "CounterApp from TutorialsPoint" ]
      ,button[onClick Subtract] [text "-"]
      ,div[][text (toString  model)]
      ,button[onClick Add] [text "+"]
  ]

--Defining the Messages

type Message =  Add | Subtract

--Defining Update

update msg model =
```

```
 case msg of
   Add -> model+1
   Subtract -> model-1

-- Define the  main method
main =
   beginnerProgram
     {
         model=model
         ,view=view
         ,update=update
     }
```

Step 3: Execute the **elm make command** in terminal. The **elm make command** compiles the code and generates an HTML file from the .elm file created above.

```
C:\Users\dell\elm\MessagesApp> elm make .\MessageDemo.elm
Some new packages are needed. Here is the upgrade plan.


  Install:
    elm-lang/core 5.1.1
    elm-lang/html 2.0.0
    elm-lang/virtual-dom 2.0.4

Do you approve of this plan? [Y/n] y
Starting downloads...

  ΓùÅ elm-lang/html 2.0.0
  ΓùÅ elm-lang/virtual-dom 2.0.4

ΓùÅ elm-lang/core 5.1.1
Packages configured successfully!
Success! Compiled 38 modules.
Successfully generated index.html
```

Step 4: Open the **index.html** and verify the working as shown below:



18. Elm — Commands

In the previous chapters, we discussed the various components of Elm architecture and their functions. The user and the application communicate with one another using Messages.

Consider an example, where the application needs to communicate with other components like an external server, APIs, microservice, etc. to serve the user request. This can be achieved by using Commands in Elm. Messages and commands are not synonymous. Messages represent the communication between an end user and the application while commands represent how an Elm application communicates with other entities. A command is triggered in response to a message.

The following figure shows the workflow of a complex Elm application –

The user interacts with the view. The view generates an appropriate message based on the user's action. The update component receives this message and triggers a command.

Syntax

The syntax for defining a command is as given below:

```
type Cmd msg
```

The message generated by the view is passed to the command.

Illustration

The following example makes a request to an API and displays the result from the API.

The application accepts a number from the user, passes it to the Numbers API. This API returns facts related to the number.

The various components of the application are as follows:

Http Module

The Http Module of Elm is used to create and send HTTP requests. This module is not a part of the core module. We will use the elm package manager to install this package.

API

In this example, the application will communicate with the Numbers API – "http://numbersapi.com/".

View

The application's view contains a textbox and a button.

```
view : Model -> Html Msg
view model =
  div []
    [ h2 [] [text model.heading]
    ,input [onInput Input, value model.input] []
    , button [ onClick ShowFacts ] [ text "show facts" ]
    , br [] []
    , h3 [][text model.factText]
    ]
```

Model

The Model represents the value entered by the user and the result that will be returned by the API.

```
type alias Model =
  { heading : String
  , factText : String
  , input :String
  }
```

Message

The application has the following three messages:

- ShowFacts
- Input
- NewFactArrived

Upon clicking the *Show Facts* button, *ShowFacts* message is passed to the update method. When the user types some value in the textbox, the *Input* message is passed to update

method. Finally, when the Http server response is received, the *NewFactArrived* message will be passed to update.

```
type Msg
  = ShowFacts
  |Input String
  | NewFactArrived (Result Http.Error String)
```

Update

The update method returns a tuple, which contains the model and command objects. When the user clicks on the *Show Facts* button, the Message is passed to the update which then calls the NumbersAPI.

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Input newInput ->
    (Model "NumbersApi typing.." "" newInput ,Cmd.none)
    ShowFacts ->
      (model, getRadmonNumberFromAPI model.input)

    NewFactArrived (Ok newFact) ->
      (Model "DataArrived" newFact "", Cmd.none)

    NewFactArrived (Err _) ->
      (model, Cmd.none)
```

Helper Function

The helper function *getRandomNumberFromAPI* invokes the NumbersAPI and passes to it the number entered by the user. The result returned by the API is used to update the model.

```
getRadmonNumberFromAPI : String->Cmd Msg
getRadmonNumberFromAPI newNo =
  let
    url =
      "http://numbersapi.com/"++newNo
  in
    Http.send NewFactArrived (Http.getString url)
```

Sr. No.	Method	Signature	Description
1	Http.getString	getString : String -> Request String	Create a GET request and interpret the response body as a String.
2	Http.send	send:(Result Error a -> msg) -> Request a -> Cmd msg	Send a Http request.

main

This is the entry point of the Elm project.

```
main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

Putting it all together

Step 1: Create folder CommandApp and file CommandDemo.elm.

Step 2: Install http module using command *elm package install elm-lang/http*.

Step 3: Type the contents for CommandDemo.elm as shown below -

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
import Http



main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

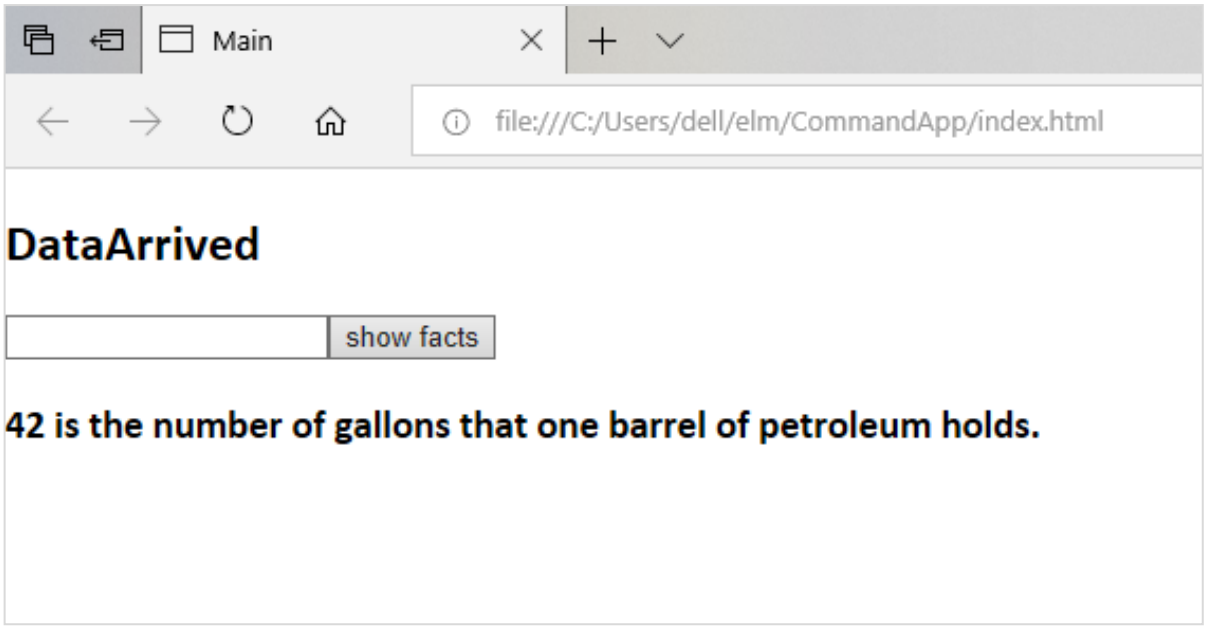
```
-- MODEL
type alias Model =
  { heading : String
  , factText : String
  , input :String
  }

init :  (Model, Cmd Msg)
init  =
  ( Model "NumbersAPI" "NoFacts" "42"-- set model two fields
  , Cmd.none  -- not to invoke api initially
  )


-- UPDATE

type Msg
  = ShowFacts
  |Input String
  | NewFactArrived (Result Http.Error String)


update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
  case msg of
    Input newInput ->
    (Model "NumbersApi typing.." "" newInput ,Cmd.none)
    ShowFacts ->
      (model, getRadmonNumberFromAPI model.input)

    NewFactArrived (Ok newFact) ->
      (Model "DataArrived" newFact "", Cmd.none)

    NewFactArrived (Err _) ->
      (model, Cmd.none)
```

```
-- VIEW


view : Model -> Html Msg
view model =
  div []
    [ h2 [] [text model.heading]
    ,input [onInput Input, value model.input] []
    , button [ onClick ShowFacts ] [ text "show facts" ]
    , br [] []
    , h3 [][text model.factText]
    ]



-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions model =
  Sub.none



-- HTTP


getRadmonNumberFromAPI : String->Cmd Msg
getRadmonNumberFromAPI newNo =
  let
    url =
      "http://numbersapi.com/"++newNo
  in
    Http.send NewFactArrived (Http.getString url)
```

Step 4: Fire the command.

```
C:\Users\dell\elm\CommandApp> elm make .\CommandDemo.elm
```

This will generate the html file as shown below.



19. Elm — Subscriptions

In the previous chapter, we discussed that a View interacts with other components using Commands. Similarly, a component (E.g. WebSocket) can talk to a View using Subscriptions. Subscriptions are a way that an Elm application can receive external inputs like keyboard events, timer events and WebSocket events.

The following figure explains the role of Subscriptions in an Elm application. The user interacts with an Elm application via messages. The application given uses WebSocket and it has two modes of operations –

- Send client-side data to socket server via Command
- Receive data anytime from the socket server via Subscription

Syntax

The syntax for defining a subscription is given below –

```
type Sub msg
```

Illustration

Let us understand subscriptions using a simple example.

In the example given below, the application sends a message to the server. The server is an echo server, which responds to the client with the same message. All the incoming messages are later displayed in a list. We will use WebSocket (wss protocol) to be able to continuously listen for messages from the server. The WebSocket will send user input to the server using Commands while it will use Subscription to receive messages from the server.

The various components of the application are given below –

Echo server

The echo server can be accessed using the *wss* protocol. The echo server sends back user input to the application. The code for defining an echo server is given below:

```
echoServer : String
echoServer =
  "wss://echo.websocket.org"
```

Model

The Model represents user input and a list of incoming messages from the socket server. The code for defining the Model is as given below –

```
type alias Model =
  { input : String
  , messages : List String
  }
```

Messages

The message type will contain *Input* for taking text input from user. The Send message will be generated when user clicks the button to send message to WebSocket server. The *NewMessage* is used when message arrives from echo server.

```
type Msg
  = Input String
  | Send
  | NewMessage String
```

View

The application's view contains a textbox and a submit button to send user input to the server. The response from the server is displayed on the View using a *div* tag.

```
view : Model -> Html Msg
view model =
  div []
    [ input [onInput Input, value model.input] []
    , button [onClick Send] [text "Send"]
    , div [] (List.map viewMessage (List.reverse model.messages))
    ]

viewMessage : String -> Html msg
viewMessage msg =
  div [] [ text msg ]
```

Update

The update function takes the message and the model components. It updates the model based on the message type.

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg {input, messages} =
  case msg of
    Input newInput ->
      (Model newInput messages, Cmd.none)

    Send ->
      (Model "" messages, WebSocket.send echoServer input)

    NewMessage str ->
      (Model input (str :: messages), Cmd.none)
```

Sr. No.	Method	Signature	Description
1	WebSocket.listen	listen : String -> (String -> msg) -> Sub msg	Subscribes to any incoming messages on a websocket.
2	WebSocket.send	send : String -> String -> Cmd msg	Sends a wss request to a server address. It is important that you are also subscribed to this address with listen. If you are not, the web socket will be created to send one message and then closed.

Subscription

The subscription function takes in the model object. To receive the messages from WebSocket server, we call *WebSocket.listen* passing in the message as *NewMessage*. When a new message comes from the server, the update method is called.

```
subscriptions : Model -> Sub Msg
subscriptions model =
WebSocket.listen echoServer NewMessage
```

main

The main function is the entry point to the elm application as shown below.

```
main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }
```

Putting it all together

Step 1: Create a directory,SubscriptionApp and add a file,SubscriptionDemo.elm to it.

Step 2: Add the following contents to SubscriptionDemo.elm file:

```
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
import WebSocket



main =
  Html.program
    { init = init
    , view = view
    , update = update
    , subscriptions = subscriptions
    }



echoServer : String
echoServer =
  "wss://echo.websocket.org"


-- MODEL



type alias Model =
```

```
  { input : String
  , messages : List String
  }

init : (Model, Cmd Msg)
init =
  (Model "" [], Cmd.none)

-- UPDATE

type Msg
  = Input String
  | Send
  | NewMessage String


update : Msg -> Model -> (Model, Cmd Msg)
update msg {input, messages} =
  case msg of
    Input newInput ->
      (Model newInput messages, Cmd.none)

    Send ->
      (Model "" messages, WebSocket.send echoServer input)

    NewMessage str ->
      (Model input (str :: messages), Cmd.none)

-- SUBSCRIPTIONS


subscriptions : Model -> Sub Msg
subscriptions model =
 WebSocket.listen echoServer NewMessage

-- VIEW
```

```
view : Model -> Html Msg
view model =
  div []
    [ input [onInput Input, value model.input] []
    , button [onClick Send] [text "Send"]
    , div [] (List.map viewMessage (List.reverse model.messages))
    ]

viewMessage : String -> Html msg
viewMessage msg =
  div [] [ text msg ]
```

Step 3: Install the websockets package using elm package manager.

```
C:\Users\dell\elm\SubscriptionApp> elm-package install elm-lang/websocket
```

Step 4: Build and generate index.html file as shown below.

```
C:\Users\dell\elm\SubscriptionApp> elm make .\SubscriptionDemo.elm
```

Step 5: Upon execution, the following output will be generated: