CoffeeScript

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

CoffeeScript is a lightweight language which transcompiles into JavaScript. It provides better syntax avoiding the quirky parts of JavaScript, still retaining the flexibility and beauty of the language.

Audience

This tutorial has been prepared for beginners to help them understand the basic functionality of CoffeeScript to build dynamic webpages and web applications.

Prerequisites

For this tutorial, it is assumed that the readers have a prior knowledge of HTML coding and JavaScript. It would help if the reader has some prior exposure to object-oriented programming concepts and a general idea on creating online applications.

Execute CoffeeScript Online

For most of the examples given in this tutorial, you will find **Try it** option, so just make use of this option to transcompile your CoffeeScript programs to JavaScript programs on the spot and enjoy your learning.

Try the following example using the **Try it** option available at the top right corner of the below sample code box –

```
console.log "Hello Welcome to Tutorials point"
```

Copyright & Disclaimer

Table of Contents

1. CoffeeScript – Overview

At present, JavaScript is the fastest mainstream dynamic language available, and it is known as the *lingua franca* of the web. It is developed by Brendan Eich in the year of 1995 in 10 days.

Because of its effective features, JavaScript became popular and went global quickly. It was there in lab for a very less time, which was not enough to polish the language. May be for this reason, inspite of its good parts, JavaScript has a bunch of design errors and it bagged a bad reputation of being a quirky language.

What is CoffeeScript ?

CoffeeScript is a lightweight language based on Ruby and Python which **transcompiles** (compiles from one source language to another) into JavaScript. It provides better syntax avoiding the quirky parts of JavaScript, still retaining the flexibility and beauty of the language.

Advantages of CoffeeScript

Following are the advantages of CoffeeScript –

- **Easily understandable** – CoffeeScript is a shorthand form of JavaScript, its syntax is pretty simple compared to JavaScript. Using CoffeeScript, we can write clean, clear, and easily understandable codes.
- **Write less do more** – For a huge code in JavaScript, we need comparatively very less number of lines of CoffeeScript.
- **Reliable** – CoffeeScript is a safe and reliable programming language to write dynamic programs.
- **Readable and maintainable** – CoffeeScript provides aliases for most of the operators which makes the code readable. It is also easy to maintain the programs written in CoffeeScript.
- **Class-based inheritance** – JavaScript does not have classes. Instead of them, it provides powerful but confusing prototypes. Unlike JavaScript, we can create classes and inherit them in CoffeeScript. In addition to this, it also provides instance and static properties as well as **mixins**. It uses JavaScript's native prototype to create classes.
- **No var keyword** – There is no need to use the **var** keyword to create a variable in CoffeeScript, thus we can avoid the accidental or unwanted scope deceleration.
- **Avoids problematic symbols** – There is no need to use the problematic semicolons and parenthesis in CoffeeScript. Instead of curly braces, we can use whitespaces to differentiate the block codes like functions, loops, etc.
- **Extensive library support** – In CoffeeScript, we can use the libraries of JavaScript and vice versa. Therefore, we have access to a rich set of libraries while working with CoffeeScript.

History of CoffeeScript

- CoffeeScript is developed by Jeremy Ashkenas. It was first committed in Git On December 13, 2009.

- Originally the compiler of the CoffeeScript was written in Ruby language.
- In March 2010, the CoffeeScript compiler was replaced; this time instead of Ruby, they used CoffeeScript itself.
- And in the same year, CoffeeScript 1.0 was released and at the time of release, it was one of the most wanted projects of the Git hub.

Limitations of CoffeeScript

Sensitive to whitespaces – CoffeeScript is very sensitive to whitespaces, so programmers need to be very careful while providing indentations. If we do not maintain proper indentation, the entire code may go wrong.

TutorialsPoint's CoffeeScript IDE

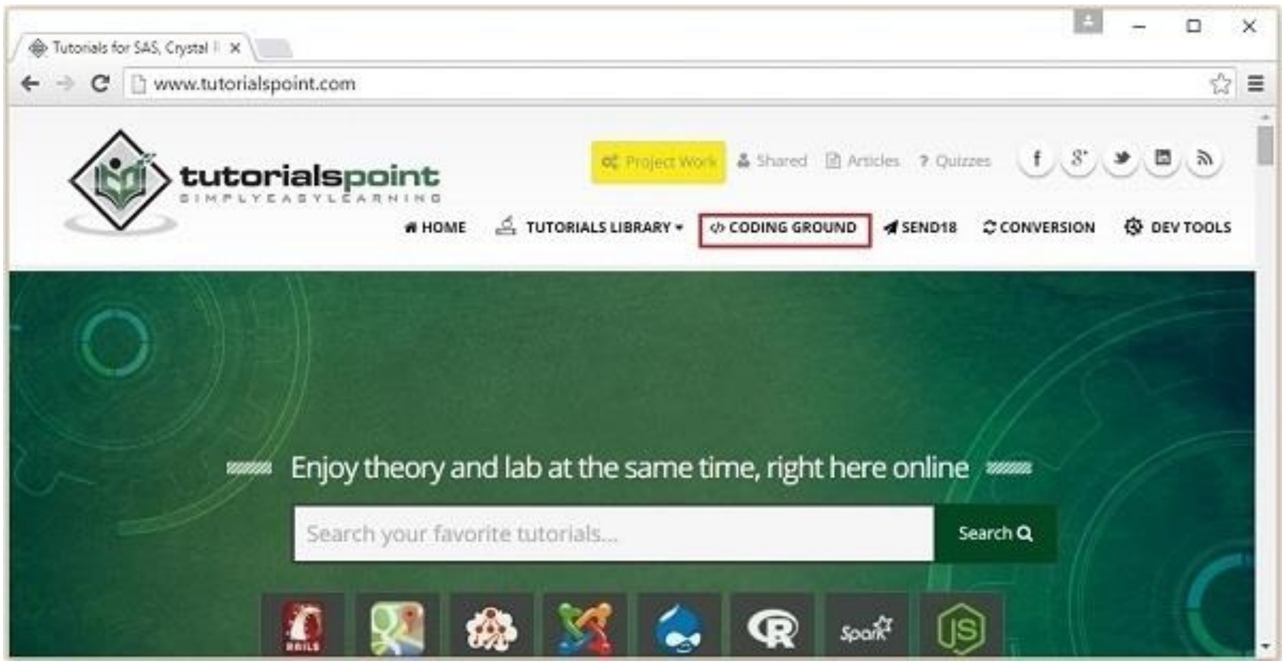
You can compile CoffeeScript files using Tutorials Point's CoffeeScript compiler provided in our Coding Ground section (http://www.tutorialspoint.com/codingground.htm). Follow the steps given below to use our CoffeeScript compiler.

Step 1

Click on the following link www.tutorialspoint.com. You will be directed to the homepage of our website.

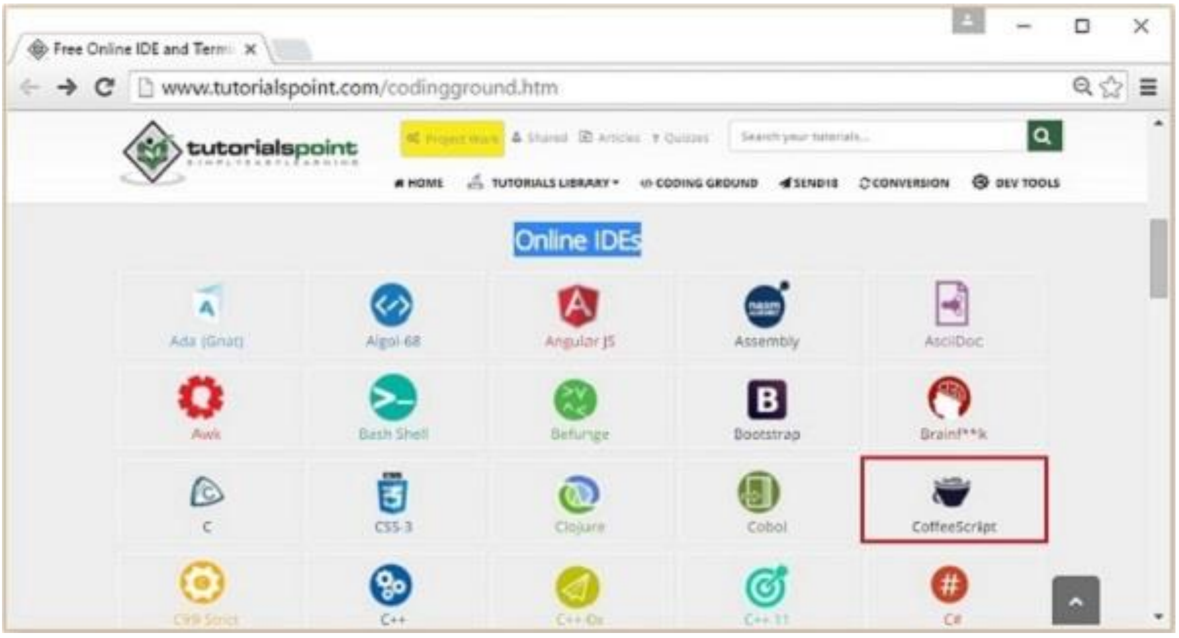
Step 2

Click on the button named **CODING GROUND** that is located at the top right corner of the homepage as highlighted in the snapshot given below.



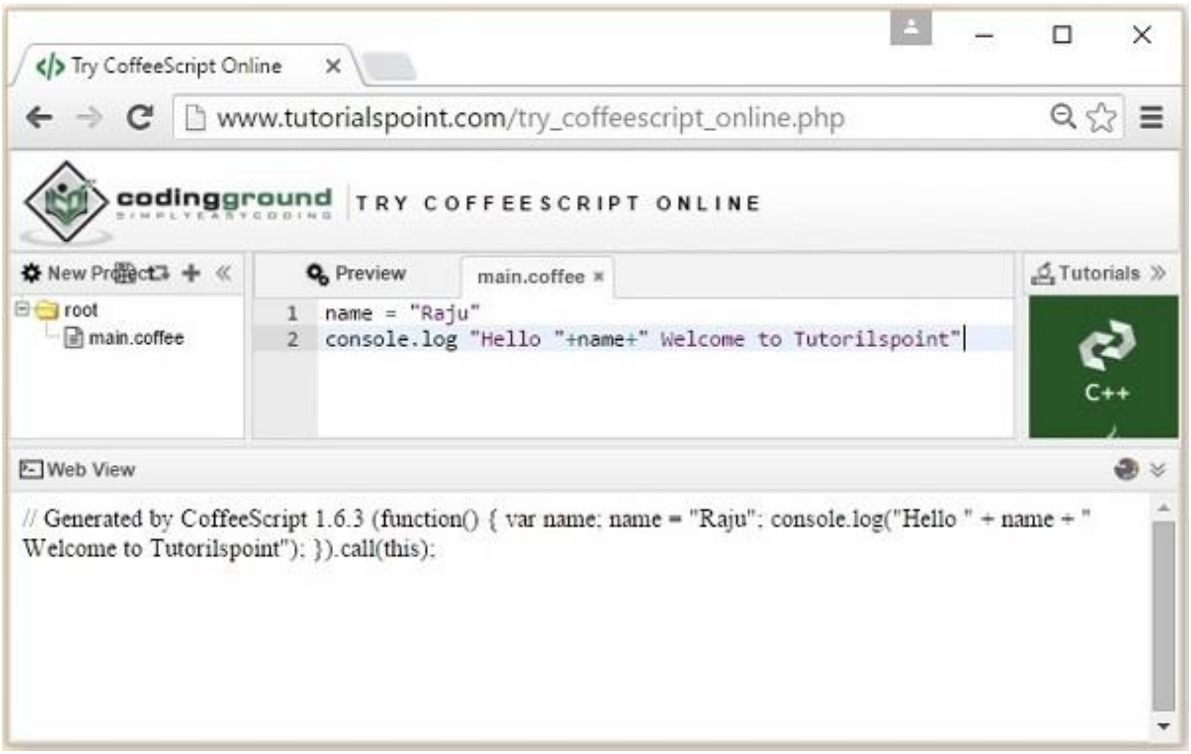
Step 3

This will lead to our **CODING GROUND** section which provides online terminals and IDEs for about 135 programming languages. Open CoffeeScript IDE in the Online IDEs section which is shown in the following snapshot.



Step 4

If you paste your CoffeeScript code in **main.coffee** (You can change the file name) and click the **Preview** button, then you can see the compiled JavaScript in the console as shown in the following snapshot.



```
name = "Raju"
console.log "Hello "+name+" Welcome to Tutorilspoint"
```

```
// Generated by CoffeeScript 1.6.3 (function() { var name; name = "Raju"; console.log("Hello " + name + " Welcome to Tutorilspoint"); }).call(this);
```

2. CoffeeScript – Environment

The Compiler of the latest versions of CoffeeScript is written in CoffeeScript itself. To run CoffeeScript files in your system without a browser, you need a JavaScript runtime.

Node.js

Node.js is a JavaScript framework which is used to develop network server applications. It also acts as a bridge between JavaScript and the Operating System.

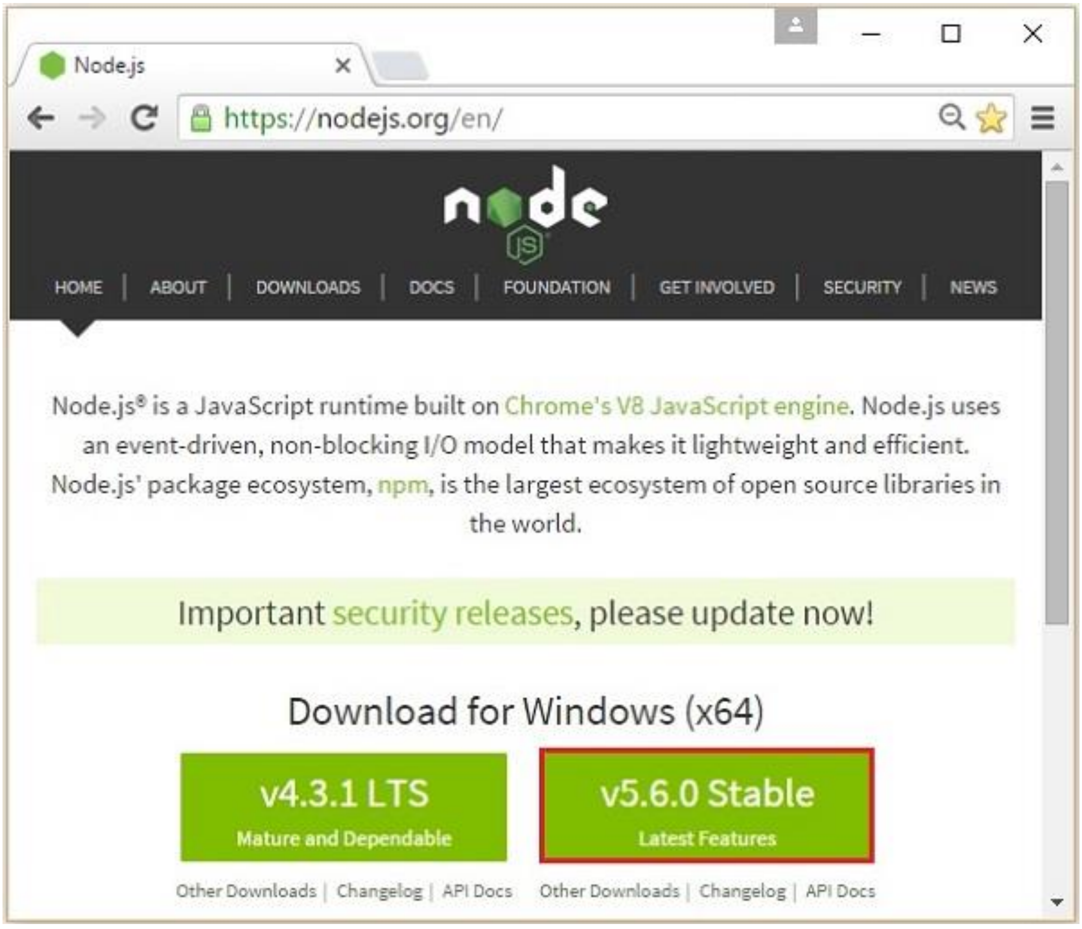
The command-line version of CoffeeScript is distributed as a Node.js package. Therefore, to install CoffeeScript (command-line) in your system, you first need install node.js.

Installing Node.js

Here are the steps to download and install Node.js in your system.

Step 1

Visit the nodejs homepage and download its stable version for windows by clicking on the button hilighted in the snapshot given below.


Node.js
https://nodejs.org/en/
node
HOME | ABOUT | DOWNLOADS | DOCS | FOUNDATION | GET INVOLVED | SECURITY | NEWS

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

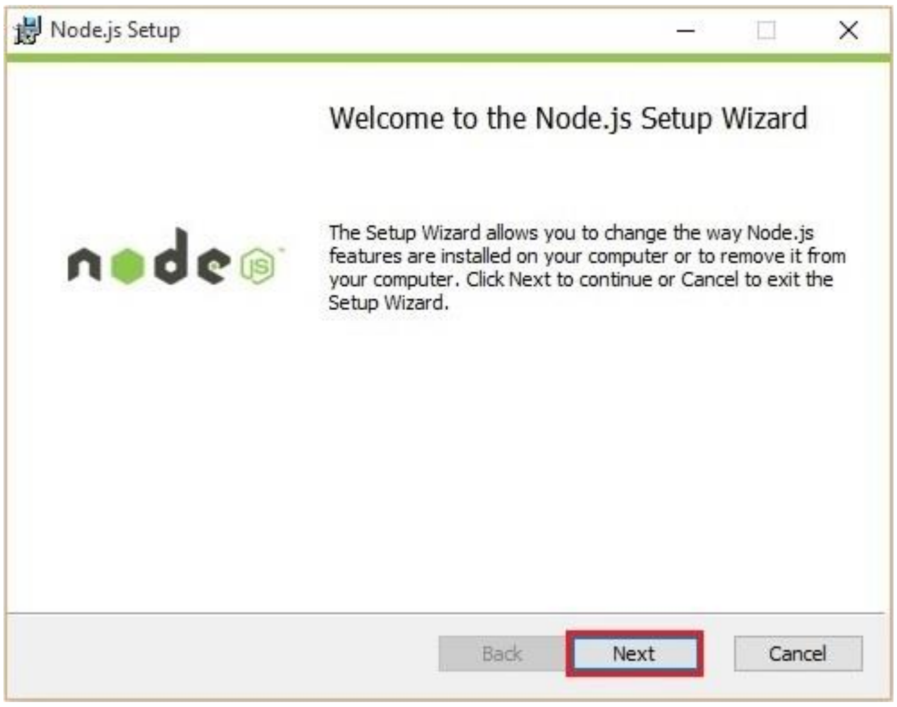
Important security releases, please update now!

Download for Windows (x64)

v4.3.1 LTS
Mature and Dependable
v5.6.0 Stable
Latest Features
Other Downloads | Changelog | API Docs
Other Downloads | Changelog | API Docs

Step 2

On clicking, a *.msc* file named **node-v5.50-x64** will be downloaded into your system, run the downloaded file to start the Node.js set-up. Here is the snapshot of the Welcome page of No.js set-up wizard.

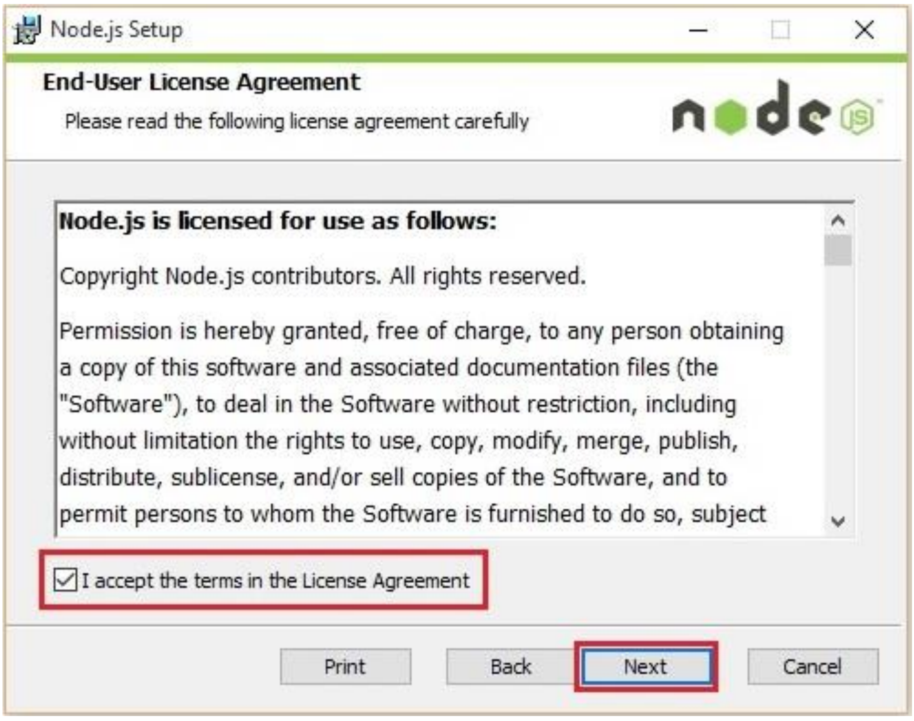


Welcome to the Node.js Setup Wizard

The Setup Wizard allows you to change the way Node.js features are installed on your computer or to remove it from your computer. Click Next to continue or Cancel to exit the Setup Wizard.

Step 3

Click on the Next button in the Welcome page of the Node.js set-up wizard which will lead you to the End-user License Agreement page. Accept the license agreement and click on the Next button as shown below.



End-User License Agreement

Please read the following license agreement carefully

Node.js is licensed for use as follows:

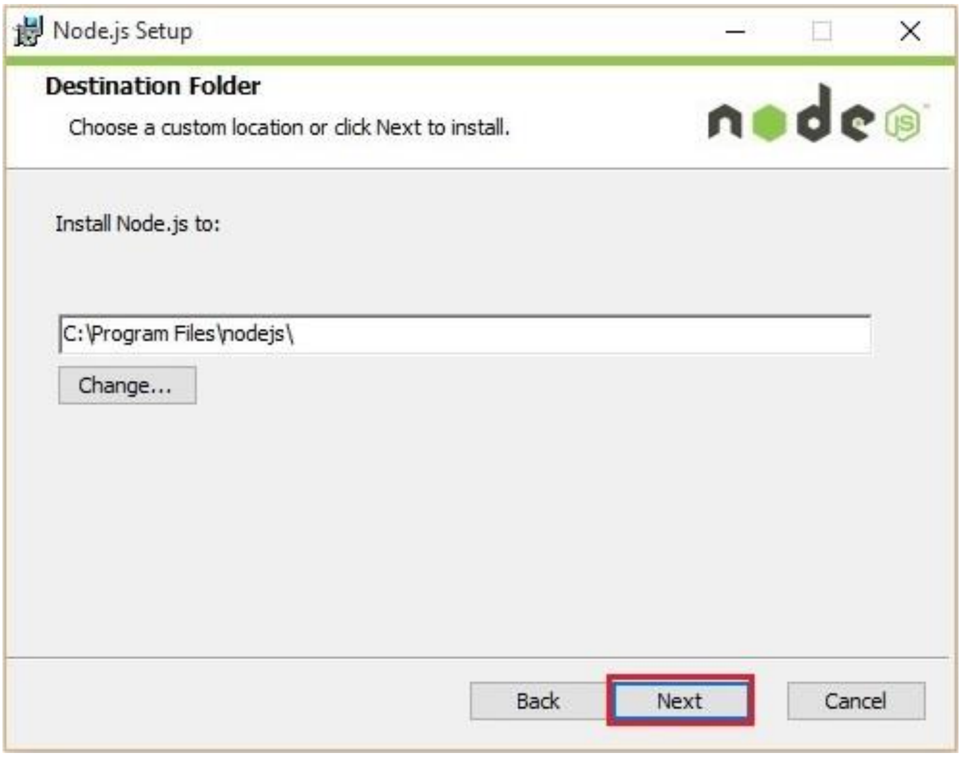
Copyright Node.js contributors. All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject

☑ I accept the terms in the License Agreement

Step 4

On the next page, you need to set the destination folder to the path where you want to install Node.js. Change the path to the required folder and click on the Next button.



Node.js Setup

Destination Folder

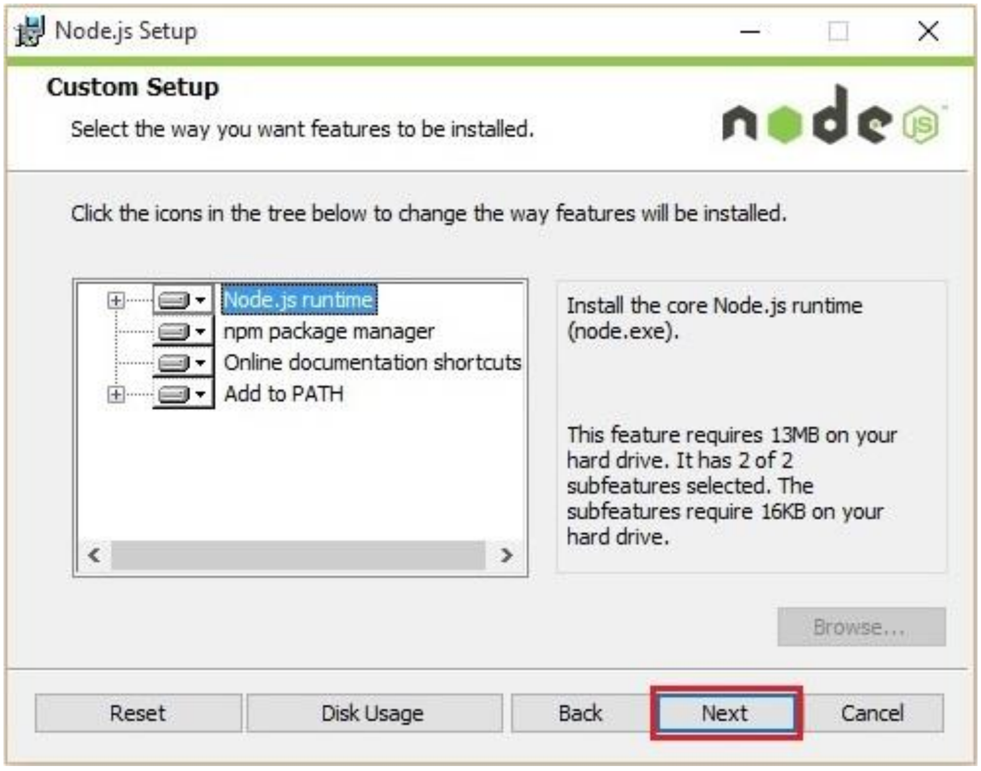
Choose a custom location or click Next to install.

Install Node.js to:

C:\Program Files\nodejs\

Step 5

In the **Custom setup** page, select the Node.js runtime to install node.exe file and click Next.



Node.js Setup

Custom Setup

Select the way you want features to be installed.

Click the icons in the tree below to change the way features will be installed.

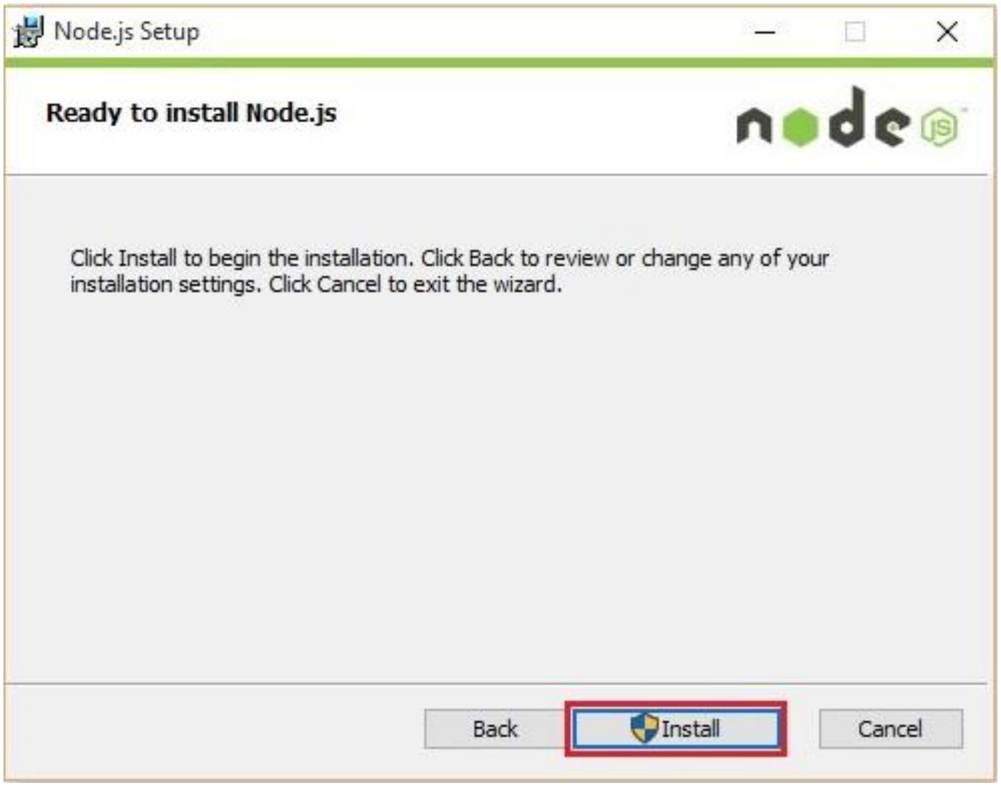
- Node.js runtime
- npm package manager
- Online documentation shortcuts
- Add to PATH

Install the core Node.js runtime (node.exe).

This feature requires 13MB on your hard drive. It has 2 of 2 subfeatures selected. The subfeatures require 16KB on your hard drive.

Step 6

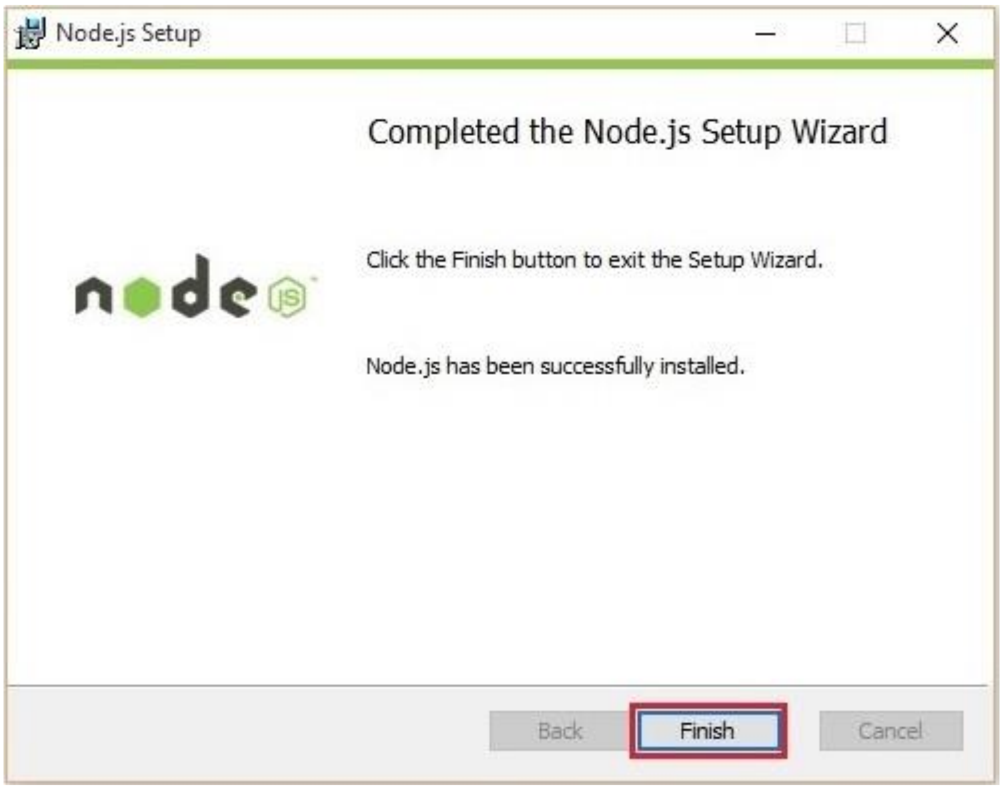
Finally, click on the Install button which will start the Node.js installation.



Ready to install Node.js

Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.

Click on the Finish button of the Node.js set-up wizard as shown below to complete the Node.js installation.



Completed the Node.js Setup Wizard

Click the Finish button to exit the Setup Wizard.

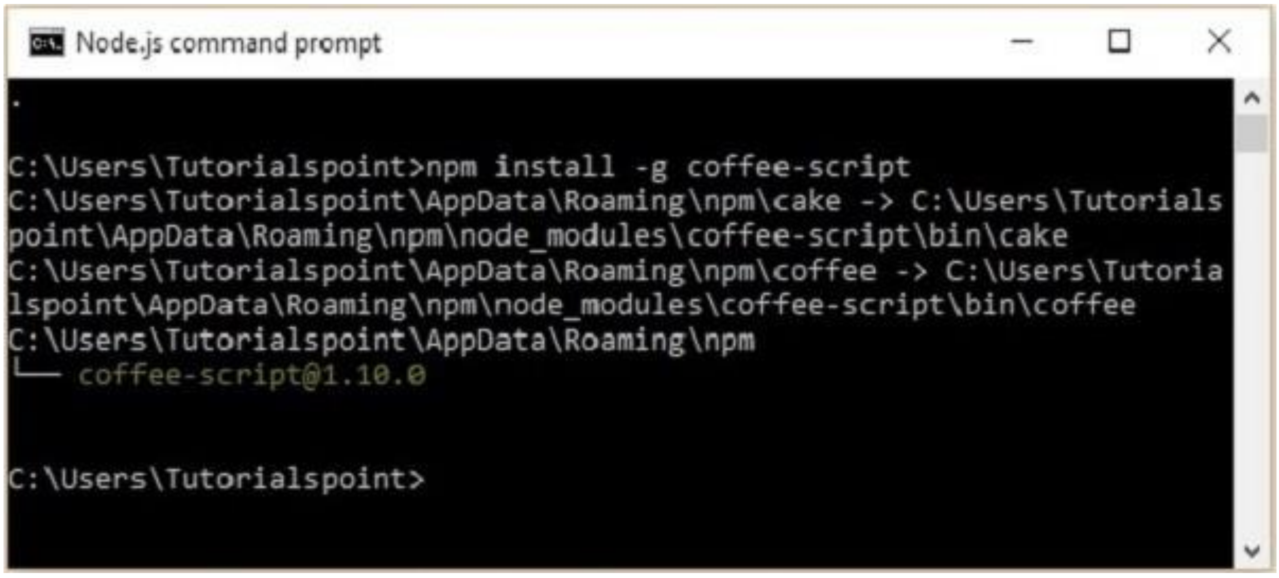
Node.js has been successfully installed.

Installing CoffeeScript

Node.js provides you a command prompt (**Node.js command prompt**). You can install CoffeeScript globally by entering the following command in it.

```
c:\> npm install -g coffeescript
```

On executing the the above command, CoffeeScript will be installed in your system by producing the following output



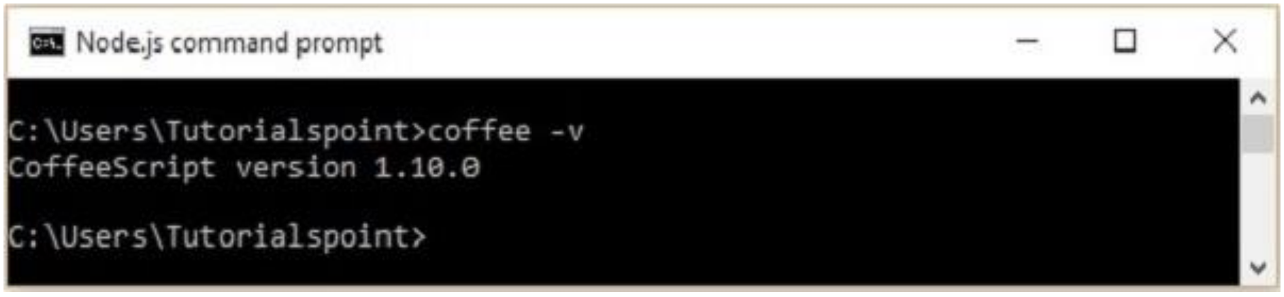
```
.

C:\Users\Tutorialspoint>npm install -g coffee-script
C:\Users\Tutorialspoint\AppData\Roaming\npm\cake -> C:\Users\Tutorials
point\AppData\Roaming\npm\node_modules\coffee-script\bin\cake
C:\Users\Tutorialspoint\AppData\Roaming\npm\coffee -> C:\Users\Tutoria
lspoint\AppData\Roaming\npm\node_modules\coffee-script\bin\coffee
C:\Users\Tutorialspoint\AppData\Roaming\npm
└── coffee-script@1.10.0


C:\Users\Tutorialspoint>
```

Verification

You can verify the installation of the CoffeeScript by typing the following command.

```
c:\> coffee -v
```

On successful installation, this command gives you the version of CoffeeScript as shown below.



```
C:\Users\Tutorialspoint>coffee -v
CoffeeScript version 1.10.0

C:\Users\Tutorialspoint>
```

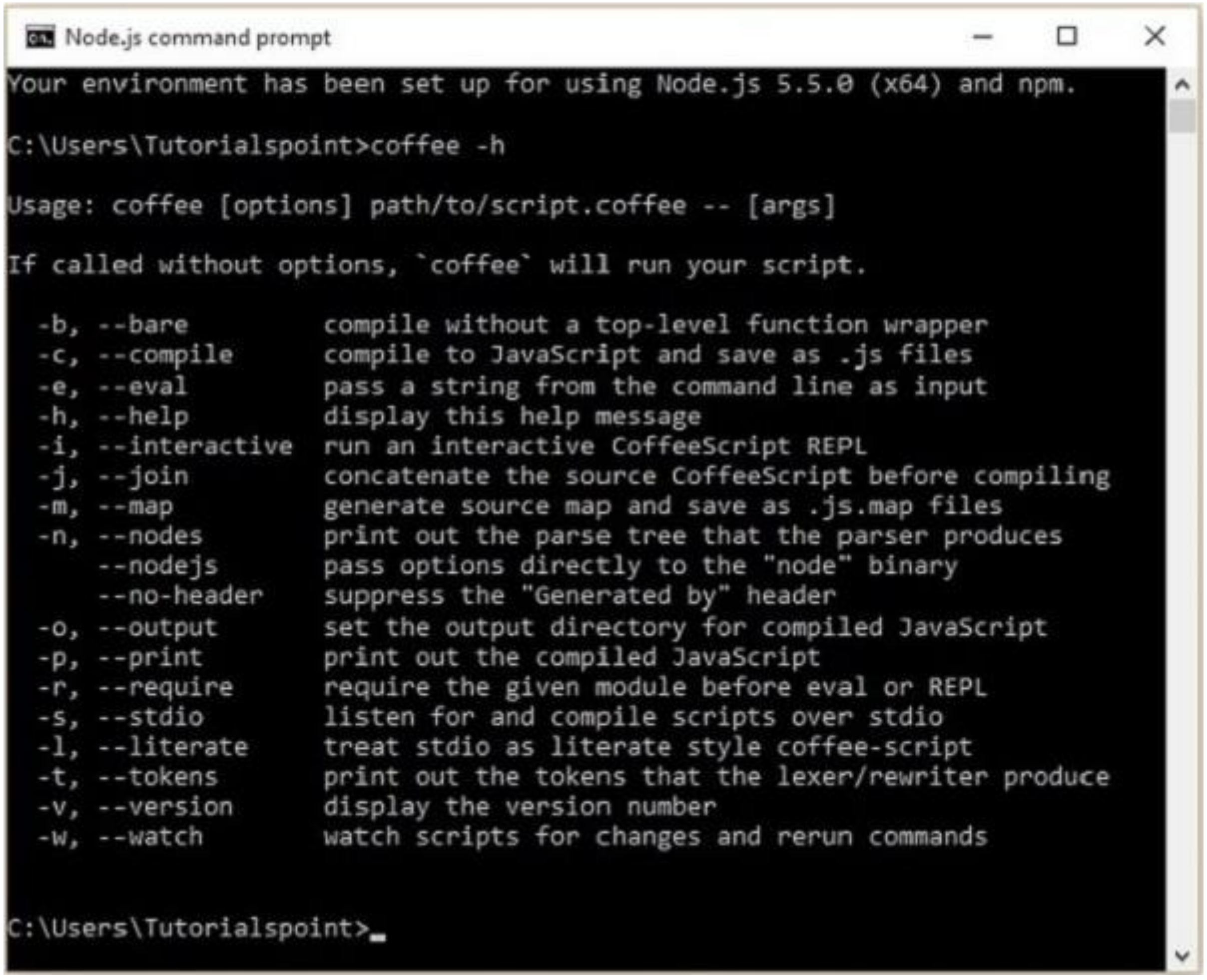
3. CoffeeScript – Command-line Utility

On installing CoffeeScript on Node.js, we can access the **coffee-command line utility**. In here, the **coffee** command is the key command. Using various options of this command, we can compile and execute the CoffeeScript files.

You can see the list of options of the **coffee** command using its **-h** or **--help** option. Open the **Node.js command prompt** and execute the following command in it.

```
c:\>coffee -help
```

This command gives you the list of various options of the **coffee**, along with the description of the operation performed by each of them as shown below.



```
Your environment has been set up for using Node.js 5.5.0 (x64) and npm.

C:\Users\Tutorialspoint>coffee -h

Usage: coffee [options] path/to/script.coffee -- [args]

If called without options, `coffee` will run your script.

  -b, --bare          compile without a top-level function wrapper
  -c, --compile       compile to JavaScript and save as .js files
  -e, --eval          pass a string from the command line as input
  -h, --help          display this help message
  -i, --interactive   run an interactive CoffeeScript REPL
  -j, --join          concatenate the source CoffeeScript before compiling
  -m, --map           generate source map and save as .js.map files
  -n, --nodes         print out the parse tree that the parser produces
      --nodejs        pass options directly to the "node" binary
      --no-header     suppress the "Generated by" header
  -o, --output        set the output directory for compiled JavaScript
  -p, --print         print out the compiled JavaScript
  -r, --require       require the given module before eval or REPL
  -s, --stdio         listen for and compile scripts over stdio
  -l, --literate      treat stdio as literate style coffee-script
  -t, --tokens        print out the tokens that the lexer/rewriter produce
  -v, --version       display the version number
  -w, --watch         watch scripts for changes and rerun commands


C:\Users\Tutorialspoint>_
```

Compiling the CoffeeScript Code

The CoffeeScript files are saved with the extension **.coffee**. You can compile these files using the **-c or --compile** option of the coffee command as shown below.

```
c:\>coffee -c filename.coffee
```

Example

Suppose there is a file in your system with the following CoffeeScript code which prints a message on the console.

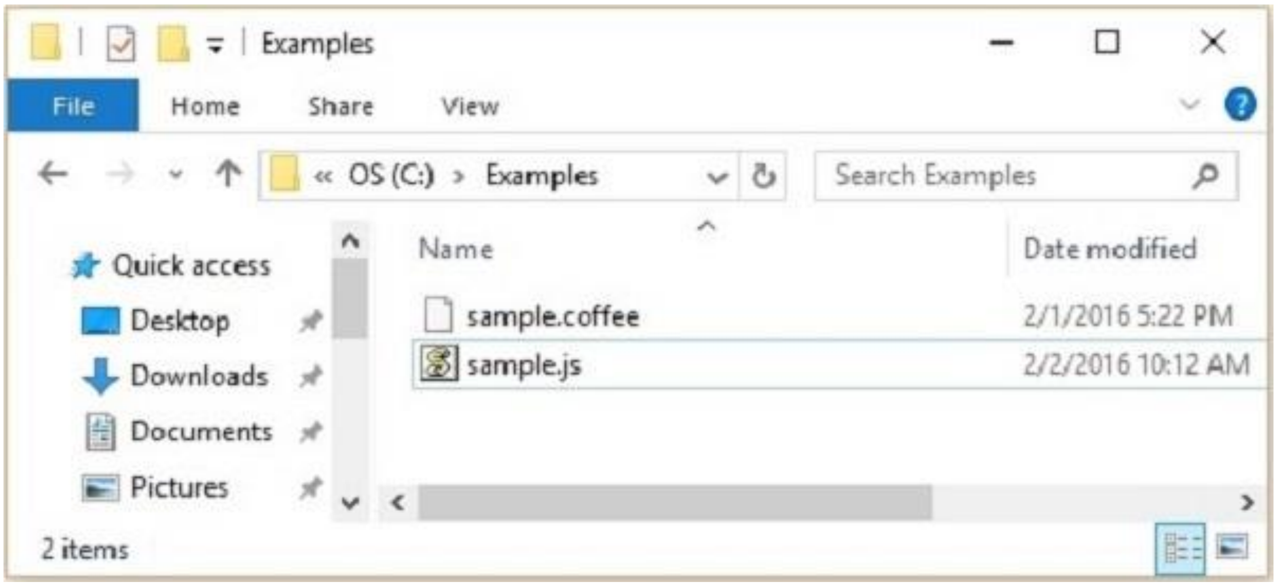
```
name = "Raju"
console.log "Hello"+name+" Welcome to Tutorilspoint"
```

Note – The **console.log()** function prints the given string on the consloe.

To compile the above code, save it in a file with the name **sample.coffee**. Open the Node.js command prompt. Browse through the path where you have saved the file and compile it using the **-c** option of the coffee command of the **coffee command-line utility** as shown below.

```
c:\> coffee -c sample.coffee
```

On executing the above command, the CoffeeScript compiler compiles the given file (sample.coffee) and saves it in the current location with a name sample.js as shown below.



If you open the sample.js file, you can observe the generated JavaScript as shown below.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var name;
  name = "Raju";
  console.log("Hello " + name + " Welcome to Tutorilspoint");

}).call(this);
```

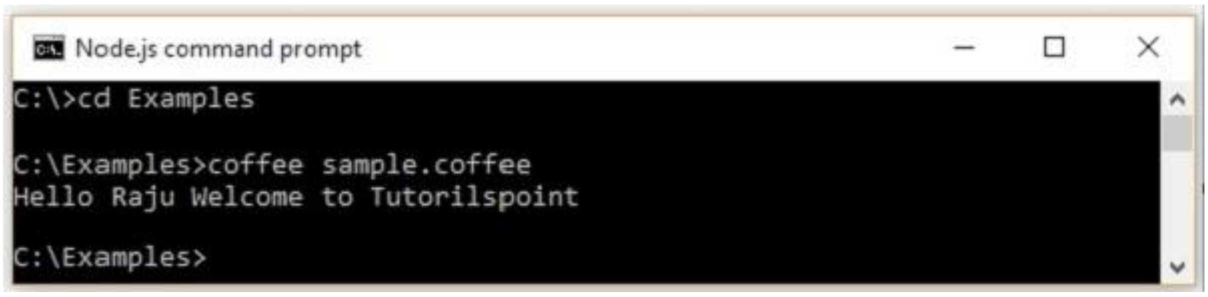
Executing the CoffeeScript code

You can execute a CoffeeScript file by simply passing the file name to the coffee command in the Node.js command prompt as follows.

```
c:\> coffee sample.coffee
```

Example

For example, let us execute the sample.coffee file. For this, open the Node.js command prompt. Browse through the path where you have saved the file and execute the file by directly passing its name to the coffee command as shown below.



```
C:\>cd Examples

C:\Examples>coffee sample.coffee
Hello Raju Welcome to Tutorilspoint

C:\Examples>
```

Watch and Compile

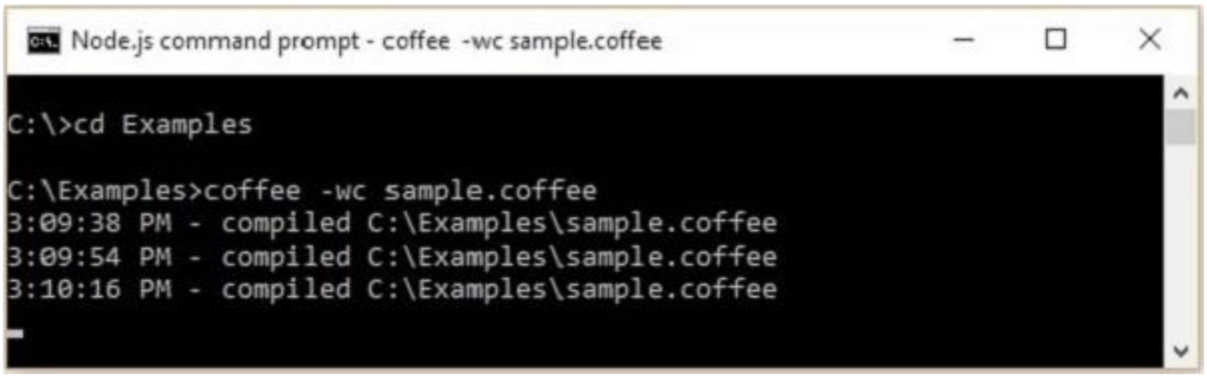
In some scenarios, there is a chance that we do a lot of changes to our scripts. Using the ***–w*** option of the coffee command, you watch your scripts for changes.

You can watch and compile a file simultaneously using the ***-wc*** option as shown below. When we use this option, the file will be recompiled each time you make changes in your script.

```
c:\>coffee -wc file_name
```

Example

Suppose we have compiled a file named **sample.coffee** using the **-wc** option and we modified the script thrice. Each time we change the script, the **.coffee** file is recompiled leaving the Node.js command prompt as shown below.



```
C:\>cd Examples

C:\Examples>coffee -wc sample.coffee
3:09:38 PM - compiled C:\Examples\sample.coffee
3:09:54 PM - compiled C:\Examples\sample.coffee
3:10:16 PM - compiled C:\Examples\sample.coffee
```

Setting the Output Directory

Using the ***-o*** option, we can set the output directory to place the compiled JavaScript files as shown below.

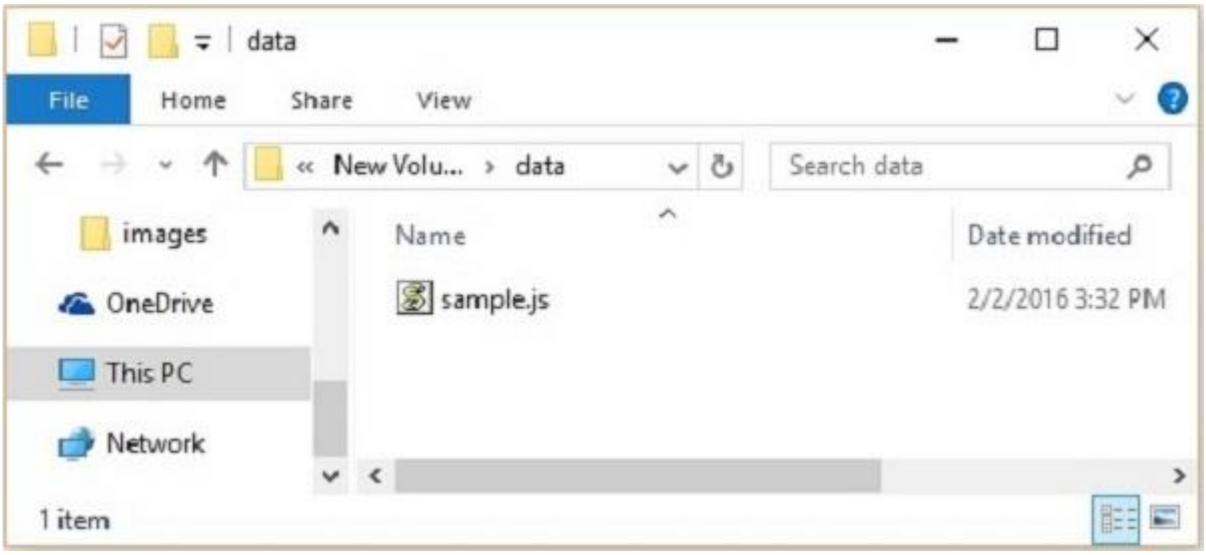
```
c:\>coffee -o "Required path where we want our .js files" file_name
```

Example

Let us save the JavaScript code of the sample.coffee file in a folder named **data** in the E drive using the ***-o*** option by executing the following command in the command prompt.

```
c:\>coffee -o E://data sample.coffee
```

Following is the snapshot of the given folder after executing the above command. Here you can observe the JavaScript file of the sample.coffee



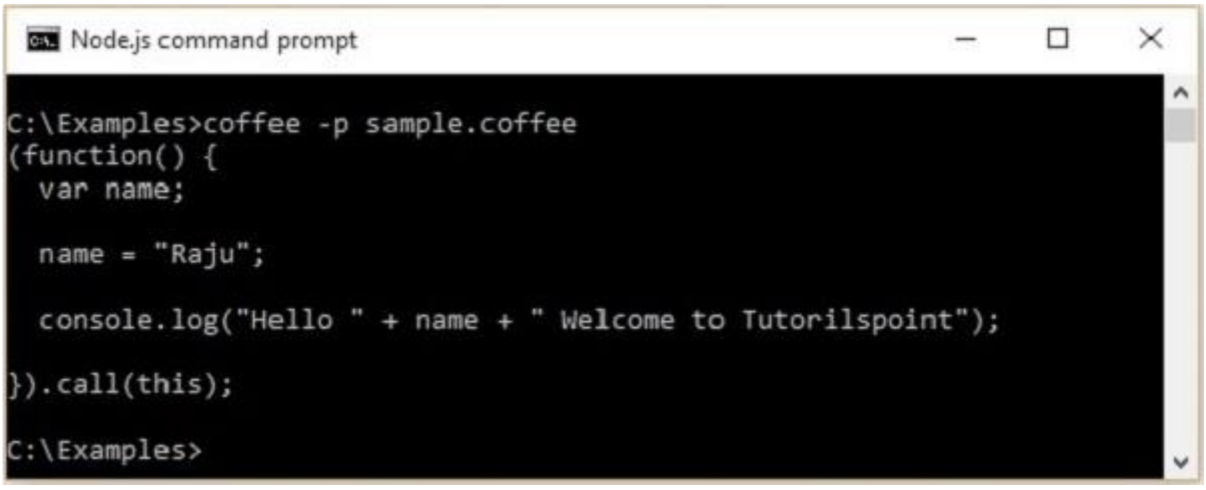
Print the Compiled JavaScript

If we want to print the compiled javascript on the console itself, we have to use the ***-p*** option of the coffee command as shown below.

```
c:\>coffee -p file_name
```

Example

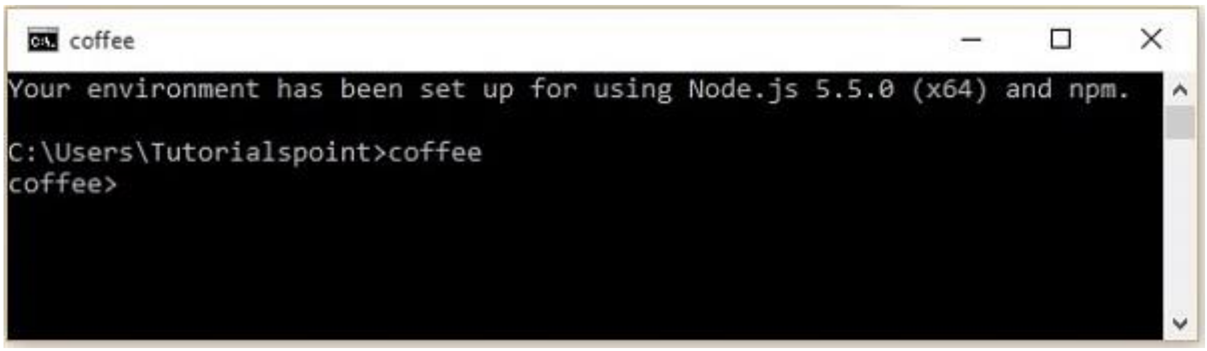
For example, you can print the compiled JavaScript code of the *sample.coffee* file on the console using the *-p* option as shown below.



```
C:\Examples>coffee -p sample.coffee
(function() {
  var name;

  name = "Raju";

  console.log("Hello " + name + " Welcome to Tutorilspoint");

}).call(this);

C:\Examples>
```

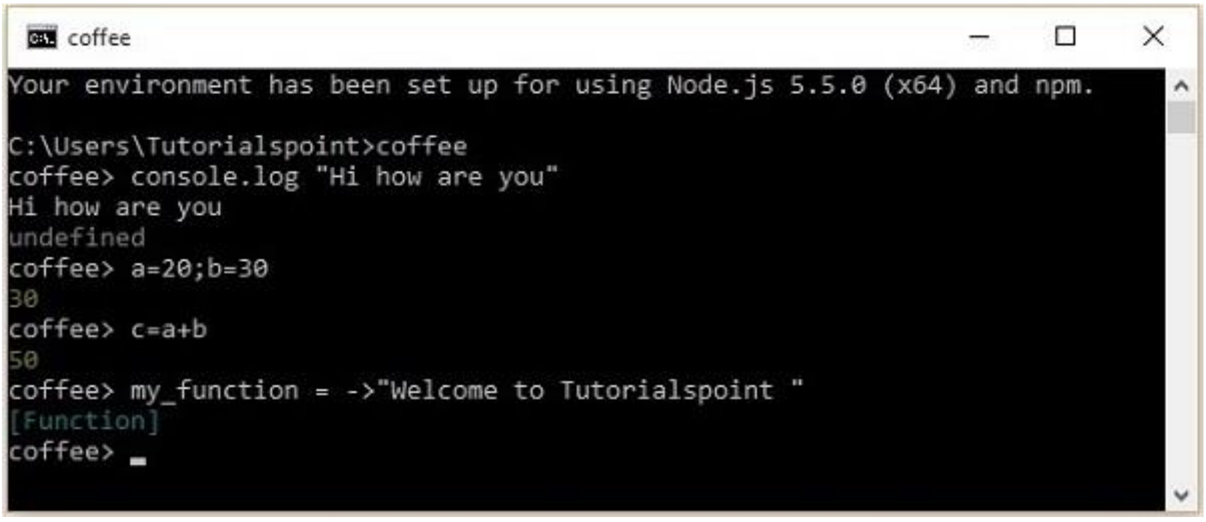
The REPL (Read Evaluate Print Loop)

CoffeeScript provides you an REPL-interactive shell. This shell is used to evaluate the CoffeeScript expressions. You can type any CoffeeScript code in this shell and get the result immediately. You can open REPL by executing the **coffee** command without any options as shown below.



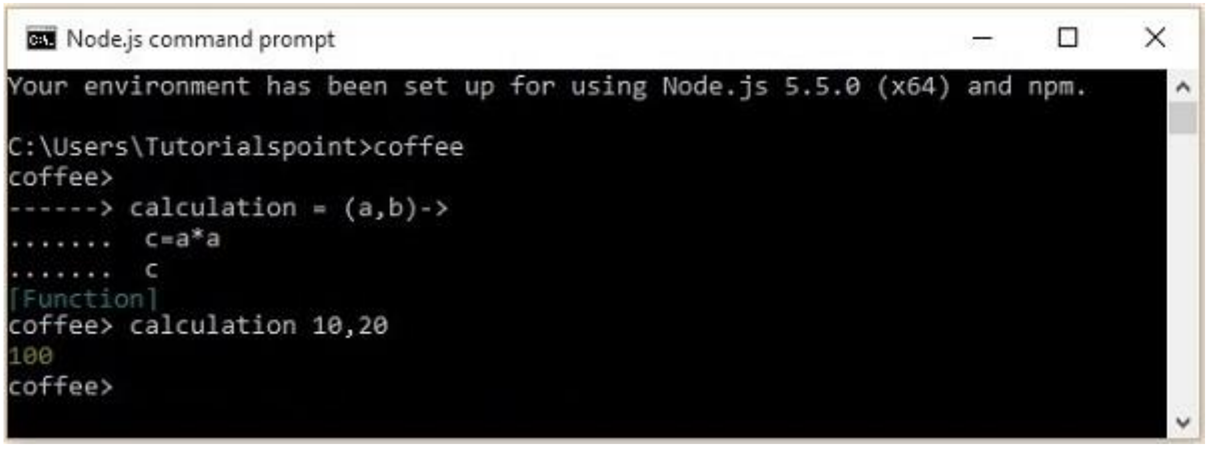
```
Your environment has been set up for using Node.js 5.5.0 (x64) and npm.

C:\Users\Tutorialspoint>coffee
coffee>
```

Using this shell, we can assign values to variables, create functions, and evaluate results. As shown in the following screenshot, if we call functions in REPL, it prints the value of the function. If we give an expression to it, it evaluates and prints the result of the expression. And if we simply type the statements in it, it prints the value of the last statement.



```
Your environment has been set up for using Node.js 5.5.0 (x64) and npm.

C:\Users\Tutorialspoint>coffee
coffee> console.log "Hi how are you"
Hi how are you
undefined
coffee> a=20;b=30
30
coffee> c=a+b
50
coffee> my_function = ->"Welcome to Tutorialspoint "
[Function]
coffee>
```

In RPEL, you can access multiple line mode by pressing *ctrl+v* where you can evaluate the code with multiple lines (like functions) and you can get back to REPL mode from it by pressing *ctrl+v* again. Here is an example usage of the multi line mode.



```
Your environment has been set up for using Node.js 5.5.0 (x64) and npm.

C:\Users\Tutorialspoint>coffee
coffee>
------> calculation = (a,b)->
.......   c=a*a
.......   c
[Function]
coffee> calculation 10,20
100
coffee>
```

Running CoffeeScript through Browser

We can run CoffeeScript using the <script> tag of the HTML just like JavaScript as shown below.

```
<script src="http://jashkenas.github.com/coffee-script/extras/coffee-script.js"
type="text/javascript" charset="utf-8"></script>

<script type="text/coffeescript">

  # Some CoffeeScript

</script>
```

But for this, we have to import the library in each application and the CoffeeScript code will be interpreted line by line before the output is shown. This will slow down your applications, therefore this approach is not recommended.

Therefore, to use CoffeeScript in your applications, you need to pre-compile them using the Coffee command-line utility and then you can use the generated JavaScript in your applications.

4. CoffeeScript – Syntax

In the previous chapter, we have seen how to install CoffeeScript. In this chapter, let us check out the syntax of CoffeeScript.

The syntax of CoffeeScript is more graceful when compared to the syntax of JavaScript. It avoids the troublesome features like curly braces, semicolons, and variable decelerations.

CoffeeScript Statements

Unlike many other programming languages like C, C++, and Java, the statements in CoffeeScript do not end with semicolons (**;**). Instead of that, every new line is considered as a separate statement by the CoffeeScript compiler.

Example

Here is an example of a CoffeeScript statement.

```
name = "Javed"

age = 26
```

In the same way, we can write two statements in a single line by separating them using semicolon as shown below.

```
name = "Javed";age = 26
```

CoffeeScript Variables (No var Keyword)

In JavaScript, we declare a variable using the **var** keyword before creating it, as shown below.

```
var name = "Javed"

var age = 20
```

While creating variables in CoffeeScript, there is no need to declare them using the **var** keyword. We can directly create a variable just by assigning a value to it as shown below.

```
name = "Javed"

age = 20
```

No Parentheses

In general, we use parenthesis while declaring the function, calling it, and also to separate the code blocks to avoid ambiguity. In CoffeeScript, there is no need to use parentheses, and while creating functions, we use an arrow mark (**->**) instead of parentheses as shown below.

```
myfunction = -> alert "Hello"
```

Still, we have to use parentheses in certain scenarios. While calling functions without parameters, we will use parentheses. For example, if we have a function named my_function in CoffeeScript, then we have to call it as shown below.

```
my_function()
```

In the same way, we can also separate the ambiguous code using parentheses. If you observe the following example, without braces, the result is 2233 and with braces, it will be 45.

```
alert "The result is  "+(22+23)
```

No Curly Braces

In JavaScript, for the block codes such as functions, loops, and conditional statements, we use curly braces. In CoffeeScript, there is no need to use curly braces. Instead, we have to maintain proper indentations (white spaces) within the body. This is the feature which is inspired from the Python language.

Following is an example of a function in CoffeeScript. Here you can observe that instead of curly braces, we have used three whitespaces as indentation to separate the body of the function.

```
myfunction = ->

   name = "John"

   alert "Hello"+name
```

CoffeeScript Comments

In any programming language, we use comments to write description about the code we have written. These comments are not considered as the part of the programs. The comments in CoffeeScript are similar to the comments of Ruby language. CoffeeScript provides two types of comments as follows –

Single-line Comments

Whenever we want to comment a single line in CoffeeScript, we just need to place a hash tag before it as shown below.

```
# This is the single line comment in CoffeeScript
```

Every single line that follows a hash tag (**#**) is considered as a comment by the CoffeeScript compiler and it compiles the rest of the code in the given file except the comments.

Multiline Comments

Whenever we want to comment more than one line in CoffeeScript (multiple lines), we can do that by wrapping those lines within a pair of triple hash tags as shown below.

```
###

These are the multi line comments in CoffeeScript

We can write as many number of lines as we want
```

```
within the pair of triple hash tags.
###
```

CoffeeScript Reserved keywords

A list of all the reserved words in CoffeeScript are given in the following table. They cannot be used as CoffeeScript variables, functions, methods, loop labels, or any object names.

case	else	Instanceof	undefined
default	interface	return	then
function	package	throw	unless
var	private	break	until
void	protected	continue	loop
with	public	debugger	of
const	static	if	by
let	yield	else	when
enum	true	switch	and
export	false	for	or
import	null	while	is
native	this	do	isnt
__hasProp	new	try	not
__extends	delete	catch	yes
__slice	typeof	finally	no
__bind	in	class	on
__indexOf	arguments	extends	off
implements	eval	super	

5. CoffeeScript – Data types

CoffeeScript Data Types

One of the most fundamental characteristics of a programming language is the set of data types it supports. These are the type of values that can be represented and manipulated in a programming language.

As CoffeeScript compiles line by line to JavaScript, the data types provided by CoffeeScript are same as JavaScript. Except for the fact that CoffeeScript adds some additional essence.

CoffeeScript provides the following data types to work with –

- **Strings** – The String data type represents a group of characters in general and we represent a string value with in-between double quotes (" ")
  Example: "Raj", "Rahman"

- **Number** – The number data type represents the numerical values.
  Example: 12, 212, etc.

- **Boolean**– Boolean data type represents one bit of information. There are only two possible values: true and false.

- **Arrays**– The Array object lets you store multiple values in a single variable. It stores a fixed-size sequential collection of elements of the same type
  Example: student = ["Rahman","Ramu","Ravi","Robert"]

  Objects– The Objects in CoffeeScript are similar to those in JavaScript these are collection of the properties. Where a property includes a key and a value separated by a semi colon (:). In short, CoffeeScript objects are a collection of key-value pairs. For example,
  student = {name: "Mohammed", age: 24, phone: 9848022338 }

- **Null** – A variable that is defined and does not hold any value is considered and null. This is similar to the null value in JavaScript.

- **Undefined** – A variable which hasn't had any value assigned to it is considered as undefined variable. If you use such variables in your code, then you will get an undefined error.

We will cover the data types Arrays and Objects in detail in separate chapters.

6. CoffeeScript – Variables

Variables are nothing but named containers. You can place data into these containers and then refer to the data using the name of its container.

CoffeeScript Variables

In JavaScript, before using a variable, we need to declare and initialize it (assign value). Unlike JavaScript, while creating a variable in CoffeeScript, there is no need to declare it using the **var** keyword. We simply create a variable just by assigning a value to a literal as shown below.

```
name = variable name
```

Example

In the following CoffeeScript code, we have defined two variables **name** and **age**, of string and number data types respectively. Save it in a file with the name **variable_example.coffee**

```
name = "Javed"

age = 25
```

Compiling the code

Compile the above CoffeeScript code by executing the following command in the command prompt.

```
c:\> compile -c variable_example.coffee
```

On compiling, a JavaScript file named **variable_example.js** will be generated with the following content. Here you can observe that the compiler declared the variables (age and name) using the **var** keyword on behalf of us.

```
(function() {

  var age, name;

  name = "Javed";

  age = 25;

}).call(this);
```

Variable Scope

The scope of a variable is the region of your program in which it is defined. JavaScript and CoffeeScript variables have only two scopes.

- **Global Variables** – A global variable has global scope which means it can be used anywhere in your JavaScript code.

- **Local Variables** – A local variable will be visible only within a function where it is defined. Function parameters are always local to that function.

The Problem with Variables in JavaScript

In JavaScript, whenever we define a variable without using the **var** keyword, it is created with global scope. This causes a lot of problems. Consider the following example –

```
<script type = "text/javascript">
   var i = 10;
   document.write("The value of global variable i is "+ i);
   document.write("<br>");
   test();
   function test() {
      i = 20;
      document.write("The value of local variable i is "+i);
      document.write("<br>");
   }
   document.write("The value of global variable i is "+i);
</script>
```

On executing, the above JavaScript gives you the following output –

```
The value of global variable i is 10
The value of local variable i is 20
The value of global variable i is 20
```

In the above example, we have created a variable named **i** in the global space and assigned the value 10 to it. And within the function, on an attempt to create a local variable with the same name, we have declared as *i=20;* without var keyword. Since we missed the **var** keyword, the value of global variable **i** is reassigned to 20.

For this reason, it is recommended to declare variables using the **var** keyword.

Variable Scope in CoffeeScript

Whenever we compile a CoffeeScript file, the CoffeeScript compiler creates an anonymous function, and within that function, it transpiles the CoffeeScript code in to JavaScript line by line. (If we want, we can remove the top level function wrapper using the **-b** or **--bare** option of the compile command) Every variable that we create is declared using the **var** keyword within the anonymous function and thus, by default, every variable is local in CoffeeScript.

```
(function() {
  var age, name;
  name = "javed";
  age = 20;
}).call(this);
```

Anyway, if we want, we can declare a variable with global namespace. We can do it explicitly as shown below.

```
obj = this
obj.age = 30
```

CoffeeScript Variable Names (Literals)

While naming your variables in CoffeeScript, keep the following rules in mind.

- You should not use any of the CoffeeScript reserved keywords as a variable name. These keywords are mentioned in the next section. For example, break or Boolean variable names are not valid.

- CoffeeScript variable names should not start with a numeral (0-9). They must begin with a letter or an underscore character. For example, 123test is an invalid variable name but _123test is a valid one.

- CoffeeScript variable names are case-sensitive. For example, **Name** and **name** are two different variables.

7. CoffeeScript – Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. Let us take a simple expression **4 + 5 is equal to 9**. Here 4 and 5 are called **operands** and '+' is called the **operator**.

The operators provided by CoffeeScript are same as in JavaScript except a few differences. There are some problematic operators in JavaScript. CoffeeScript either removed them or modified their functionality and it also introduced some new operators.

CoffeeScript Aliases

In addition to operators, CoffeeScript also provides aliases. CoffeeScript provides aliases to various operators and symbols in order to make your CoffeeScript code readable and more user friendly.

Following is the list of operators supported by CoffeeScript.

- Arithmetic Operators
- Comparison Operators
- Logical (or Relational) Operators
- Assignment Operators

Let us have a look at all the operators and aliases of CoffeeScript one by one.

Arithmetic Operators

CoffeeScript supports the following arithmetic operators. Assume variable **A** holds **10** and variable **B** holds **20**, then –

Show Examples

Sr.No	Operator and Description	Example
1	**+ (Addition)** Adds two operands	A + B = 30
2	**- (Subtraction)** Subtracts the second operand from the first	A - B = -10
3	*** (Multiplication)** Multiply both operands	A * B = 200
4	**/ (Division)** Divide the numerator by the denominator	B / A = 2
5	**% (Modulus)** Outputs the remainder of an integer division	B % A = 0

6	**++ (Increment)** Increases an integer value by one	A++ = 11
7	**-- (Decrement)** Decreases an integer value by one	A-- = 9

Example

The following example shows how to use arithmetic operators in CoffeeScript. Save this code in a file with name **airthmatic_example.coffee**

```
a = 33
b = 10
c = "test"
console.log "The value of a + b = is"
result = a + b
console.log result

result = a - b
console.log "The value of a - b = is "
console.log result

console.log "The value of a / b = is"
result = a / b
console.log result

console.log "The value of a % b = is"
result = a % b
console.log result

console.log "The value of a + b + c = is"
result = a + b + c
console.log result

a = ++a
console.log "The value of ++a = is"
result = ++a
console.log result
```

```
b = --b
console.log "The value of --b = is"
result = --b
console.log result
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c airthmatic_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var a, b, c, result;
  a = 33;
  b = 10;
  c = "test";

  console.log("The value of a + b = is");
  result = a + b;
  console.log(result);

  result = a - b;
  console.log("The value of a - b = is ");
  console.log(result);

  console.log("The value of a / b = is");
  result = a / b;
  console.log(result);

  console.log("The value of a % b = is");
  result = a % b;
  console.log(result);

  console.log("The value of a + b + c = is");
  result = a + b + c;
  console.log(result);

  a = ++a;
  console.log("The value of ++a = is");
  result = ++a;
  console.log(result);

  b = --b;
  console.log("The value of --b = is");
  result = --b;
  console.log(result);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee airthmatic_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The value of a + b = is
43
The value of a - b = is
23
The value of a / b = is
3.3
The value of a % b = is
3
The value of a + b + c = is
43test
The value of ++a = is
35
The value of --b = is
8
```

Comparison Operators

JavaScript supports the following comparison operators. Assume variable **A** holds **10** and variable **B** holds **20**, then –

Show Examples

Sr.No	Operator and Description	Example
1	= = (Equal) Checks if the value of two operands are equal or not, if yes, then the condition becomes true.	(A == B) is not true.
2	!= (Not Equal) Checks if the value of two operands are equal or not, if the values are not equal, then the condition becomes true.	(A != B) is true.
3	> (Greater than) Checks if the value of the left operand is greater than the value of the right operand, if yes, then the condition becomes true.	(A > B) is not true.
4	< (Less than) Checks if the value of the left operand is less than the value of the right operand, if yes, then the condition becomes true.	(A < B) is true.
5	>= (Greater than or Equal to) Checks if the value of the left operand is greater than or equal to the value of the right operand, if yes, then the condition becomes true.	(A >= B) is not true.
6	<= (Less than or Equal to) Checks if the value of the left operand is less than or equal to the value of the right operand, if yes, then the condition becomes true.	(A <= B) is true.

Example

The following code shows how to use comparison operators in CoffeeScript. Save this code in a file with name comparison_example.coffee

```
a = 10
b = 20
console.log "The result of (a == b) is "
result = a == b
console.log result

console.log "The result of (a < b) is "
result = a < b
console.log result

console.log "The result of (a > b) is "
result = a > b
```

```
console.log result

console.log "The result of (a != b) is "
result = a != b
console.log result

console.log "The result of (a >= b) is "
result = a <= b
console.log result

console.log "The result of (a <= b) is "
result = a >= b
console.log result
```

Open the **command prompt** and compile the comparison_example.coffee file as shown below.

```
c:/> coffee -c comparison_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var a, b, result;
  a = 10;
  b = 20;

  console.log("The result of (a == b) is ");
  result = a === b;
  console.log(result);

  console.log("The result of (a < b) is ");
  result = a < b;
  console.log(result);

  console.log("The result of (a > b) is ");
  result = a > b;
  console.log(result);

  console.log("The result of (a != b) is ");
  result = a !== b;
  console.log(result);

  console.log("The result of (a >= b) is ");
  result = a <= b;
  console.log(result);

  console.log("The result of (a <= b) is ");
  result = a >= b;
  console.log(result);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:/> coffee comparison_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The result of (a == b) is
false
The result of (a < b) is
true
The result of (a > b) is
false
The result of (a != b) is
true
The result of (a >= b) is
true
The result of (a <= b) is
false
```

Following table shows the aliases for few of the Comparison operators. Suppose **A** holds **20**and variable **B** holds **20**.

Show Examples

Operator	Alias	Example
= = (Equal)	**is**	**A is B** gives you true.

!= = (Not Equal)	**isnt**	**A isnt B** gives you false.

Example

The following code shows how to use aliases for comparison operators in CoffeeScript. Save this code in a file with name comparison_aliases.coffee

```
a = 10
b = 20
console.log "The result of (a is b) is "
result = a is b
console.log result

console.log "The result of (a isnt b) is "
result = a isnt b
console.log result
```

Open the **command prompt** and compile the comparison_example.coffee file as shown below.

```
c:/> coffee -c comparison_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var a, b, result;

  a = 10;

  b = 20;

  console.log("The result of (a is b) is ");

  result = a === b;

  console.log(result);

  console.log("The result of (a isnt b) is ");

  result = a !== b;
  console.log(result);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:/> coffee comparison_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The result of (a is b) is
false
The result of (a isnt b) is
true
```

Logical Operators

CoffeeScript supports the following logical operators. Assume variable **A** holds **10** and variable **B** holds **20**, then –

Show Examples

Sr.No	Operator and Description	Example
1	**&& (Logical AND)** If both the operands are non-zero, then the condition becomes true.	(A && B) is true.
2	**|| (Logical OR)** If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
3	**! (Logical NOT)** Reverses the logical state of its operand. If a condition is true, then the Logical NOT operator will make it false.	! (A && B) is false.

Example

Following is the example demonstrating the use of logical operators in coffeeScript. Save this code in a file with name logical_example.coffee.

```
a = true
b = false
```

```
console.log "The result of (a && b) is "
result = a and b
console.log result

console.log "The result of (a || b) is "
result = a or b
console.log result

console.log "The result of !(a && b) is "
result = !(a and b)
console.log result
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c logical_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var a, b, result;
  a = true;
  b = false;

  console.log("The result of (a && b) is ");
  result = a && b;
  console.log(result);

  console.log("The result of (a || b) is ");
  result = a || b;
  console.log(result);

  console.log("The result of !(a && b) is ");
  result = !(a && b);
  console.log(result);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee logical_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The result of (a && b) is
false
The result of (a || b) is
true
The result of !(a && b) is
true
```

The following table shows the aliases for some of the logical operators. Suppose **X** holds **true** and variable **Y** holds **false**.

Show Examples

Operator	Alias	Example
&& (Logical AND)	**and**	**X and Y** gives you false
|| (Logical OR)	**or**	**X or Y** gives you true
! (not x)	**not**	**not X** gives you false

Example

The following example demonstrates the use aliases for logical operators in CoffeeScript. Save this code in a file with name logical_aliases.coffee.

```
a = true
b = false

console.log "The result of (a and b) is "
result = a and b
console.log result

console.log "The result of (a or b) is "
result = a or b
console.log result

console.log "The result of not(a and b) is "
```

```
result = not(a and b)
console.log result
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c logical_aliases.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var a, b, result;
  a = true;
  b = false;

  console.log("The result of (a and b) is ");
  result = a && b;
  console.log(result);

  console.log("The result of (a or b) is ");
  result = a || b;
  console.log(result);

  console.log("The result of not(a and b) is ");
  result = !(a && b);
  console.log(result);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee logical_aliases.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The result of (a and b) is
false
The result of (a or b) is
true
The result of not(a and b) is
true
```

Bitwise Operators

CoffeeScript supports the following bitwise operators. Assume variable **A** holds **2** and variable **B** holds **3**, then –

Show Examples

Sr.No	Operator and Description	Example
1	**& (Bitwise AND)** It performs a Boolean AND operation on each bit of its integer arguments.	**(A & B) is 2.**
2	**| (BitWise OR)** It performs a Boolean OR operation on each bit of its integer arguments.	(A | B) is 3.
3	**^ (Bitwise XOR)** It performs a Boolean exclusive OR operation on each bit of its integer arguments. Exclusive OR means that either operand one is true or operand two is true, but not both.	(A ^ B) is 1.
4	**~ (Bitwise Not)** It is a unary operator and operates by reversing all the bits in the operand.	(~B) is -4.
5	**<< (Left Shift)** It moves all the bits in its first operand to the left by the number of places specified in the second operand. New bits are filled with zeros. Shifting a value left by one position is equivalent to multiplying it by 2, shifting two positions is equivalent to multiplying by 4, and so on.	(A << 1) is 4.
6	**>> (Right Shift)** Binary Right Shift Operator. The left operand’s value is moved right by the number of bits specified by the right operand.	(A >> 1) is 1.
7	**>>> (Right shift with Zero)** This operator is just like the >> operator, except that the bits shifted in on the left are always zero.	(A >>> 1) is 1.

Example

The following example demonstrates the usage of bitwise operators in CoffeeScript. Save this code in a file with name **bitwise_example.coffee**

```
a = 2 # Bit presentation 10
b = 3 # Bit presentation 11


console.log "The result of (a & b) is "
```

```
result = a & b
console.log result

console.log "The result of (a | b) is "
result = a | b
console.log result

console.log "The result of (a ^ b) is "
result = a ^ b
console.log result

console.log "The result of (~b) is "
result = ~b
console.log result

console.log "The result of (a << b) is "
result = a << b
console.log result

console.log "The result of (a >> b) is "
result = a >> b
console.log result
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:/> coffee -c bitwise_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var a, b, result;
  a = 2;
  b = 3;

  console.log("The result of (a & b) is ");
  result = a & b;
  console.log(result);
  console.log("The result of (a | b) is ");
  result = a | b;
  console.log(result);

  console.log("The result of (a ^ b) is ");
  result = a ^ b;
  console.log(result);

  console.log("The result of (~b) is ");
  result = ~b;
  console.log(result);

  console.log("The result of (a << b) is ");
  result = a << b;
  console.log(result);

  console.log("The result of (a >> b) is ");
  result = a >> b;
  console.log(result);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:/> coffee bitwise_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The result of (a & b) is
2
The result of (a | b) is
3
The result of (a ^ b) is
1
The result of (~b) is
-4
The result of (a << b) is
16
The result of (a >> b) is
```

```
0
```

Assignment Operators

CoffeeScript supports the following assignment operators –

Show Examples

Sr.No	Operator and Description	Example
1	**= (Simple Assignment)** Assigns values from the right side operand to the left side operand	C = A + B will assign the value of A + B into C
2	**+= (Add and Assignment)** It adds the right operand to the left operand and assigns the result to the left operand.	C += A is equivalent to C = C + A
3	**−= (Subtract and Assignment)** It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
4	***= (Multiply and Assignment)** It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
5	**/= (Divide and Assignment)** It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
6	**%= (Modules and Assignment)** It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A

Note – Same logic applies to Bitwise operators so they will become like <<=, >>=, >>=, &=, |= and ^=.

Example

The following example demonstrates the usage of assignment operators in CoffeeScript. Save this code in a file with name **assignment _example.coffee**

```
a = 33
b = 10


console.log "The value of a after the operation (a = b) is "
result = a = b
```

```
console.log result

console.log "The value of a after the operation (a += b) is "
result = a += b
console.log result

console.log "The value of a after the operation (a -= b) is "
result = a -= b
console.log result

console.log "The value of a after the operation (a *= b) is "
result = a *= b
console.log result

console.log "The value of a after the operation (a /= b) is "
result = a /= b
console.log result

console.log "The value of a after the operation (a %= b) is "
result = a %= b
console.log result
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:/> coffee -c assignment _example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var a, b, result;
  a = 33;
  b = 10;

  console.log("The value of a after the operation (a = b) is ");
  result = a = b;
  console.log(result);

  console.log("The value of a after the operation (a += b) is ");
  result = a += b;
  console.log(result);

  console.log("The value of a after the operation (a -= b) is ");
  result = a -= b;
  console.log(result);

  console.log("The value of a after the operation (a *= b) is ");
  result = a *= b;
  console.log(result);

  console.log("The value of a after the operation (a /= b) is ");
  result = a /= b;
  console.log(result);

  console.log("The value of a after the operation (a %= b) is ");
  result = a %= b;
  console.log(result);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:/> coffee assignment _example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The value of a after the operation (a = b) is
10
The value of a after the operation (a += b) is
20
The value of a after the operation (a -= b) is
10
The value of a after the operation (a *= b) is
100
The value of a after the operation (a /= b) is
10
The value of a after the operation (a %= b) is
```

```
0
```

Equality Operator in CoffeeScript

While working with JavaScript, you will encounter two types of equality operators **==** and**===**.

The **==** operator in JavaScript is **type coercive**, i.e., if the types of the two operands in an operation are different, then the data type of one of the operator is converted into other and then both are compared.

CoffeeScript avoids this undesirable coercion, it compiles the **==** operator in to the strict comparison operator of JavaScript **===**.

If we compare two operands using **===**, then it returns **true**, only if both the value and datatypes of them are equal, else it returns **false**.

Example

Consider the following example. Here we have two variables **a** and **b**. **a** holds the value 21 of integer type and **b** holds the same value, but it is of **string** type. In CoffeeScript, when we compare **a** and **b**, the result will be **false**. (Since the **==** operator of CoffeeScript is converted to === operator of JavaScript)

```
a=21
b="21"
result = 21=='21'
console.log result
```

On compiling, the above CoffeeScript produces the following JavaScript

```
// Generated by CoffeeScript 1.10.0
(function() {
  var a, b, result;
  a = 21;
  b = "21";

  result = a === b;
  console.log(result);
}).call(this);
```

On executing, it produces the following output.

```
false
```

The existential Operator

CoffeeScript provides a new operator known as existential operator to verify the existence of a variable. It is denoted by **?**. Unless a variable is null or undefined, the existential operator returns true.

Example

Given below is an example of the existential operator. Here we have three variables, namely **name**, **age**, and **subject** and we are verifying the existence of the variables name and phone using existential operator.

```
name="Ramu"
age=24
subject="Engineering"
verify_name = name?
verify_phone = phone?
console.log verify_name
console.log verify_phone
```

On compiling, this will generate the following JavaScript code.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var age, name, subject, verify_name, verify_phone;
  name = "Ramu";
  age = 24;

  subject = "Engineering";
  verify_name = name != null;
  verify_phone = typeof phone !== "undefined" && phone !== null;
  console.log(verify_name);
  console.log(verify_phone);

}).call(this);
```

If we execute the above CoffeeScript file, it produces the following output.

```
true
false
```

Note – We have an accessor variant of the existential operator **?.** We can use it instead of the **.** operator to find out the null references.

Chained Comparisons

As in Python, we can use a chain of comparison operators in a single expression in CoffeeScript.

Example

Following is an example of using chained comparison.

```
score = 70
passed = 100 > score > 40

console.log passed
```

On compiling, the example CoffeeScript gives you the following JavaScript code.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var passed, score;

  score = 70;

  passed = (100 > score && score > 40);

  console.log(passed);

}).call(this);
```

If you execute the above CoffeeScript code, it produces the following output.

```
true
```

Note – CoffeeScript removes the ternary operator; instead of it, we can use the **inline if** statement.

CoffeeScript Aliases

In general, CoffeeScript provides aliases to various operators and symbols in order to make your CoffeeScript code readable and more user friendly. Following are the aliases provided by CoffeeScript.

Name	Operator / symbol	Aliases
"equals to" operator	==	is
"not equals to" operator	!==	isnt
"not" operator	!	not

"and" operator	&&	and
"or" operator	||	or
Boolean value true	true	true, yes, on
Boolean value false	false	off, no
current object	this	@
new line (or) semi colon	/n or ;	then
Inverse of if	! if	unless
To test for array presence		in
To test for object presence		of
Exponentiation		a**b
Integer division		a//b
dividend dependent modulo		a%%b

Example

The following example shows how to use aliases in CoffeeScript –

```
a=21; b=21

x = true; y = false

console.log a is b

console.log a isnt b

console.log x and y

console.log x or y

console.log yes or no

console.log on or off

console.log a**b
console.log a//b

console.log a%%b
```

On compiling the above example, it gives you the following JavaScript code.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var a, b, x, y,
    modulo = function(a, b) { return (+a % (b = +b) + b) % b; };

  a = 21;

  b = 21;

  x = true;

  y = false;

  console.log(a === b);

  console.log(a !== b);

  console.log(x && y);

  console.log(x || y);

  console.log(true || false);

  console.log(true || false);

  console.log(Math.pow(a, b));

  console.log(Math.floor(a / b));

  console.log(modulo(a, b));

}).call(this);
```

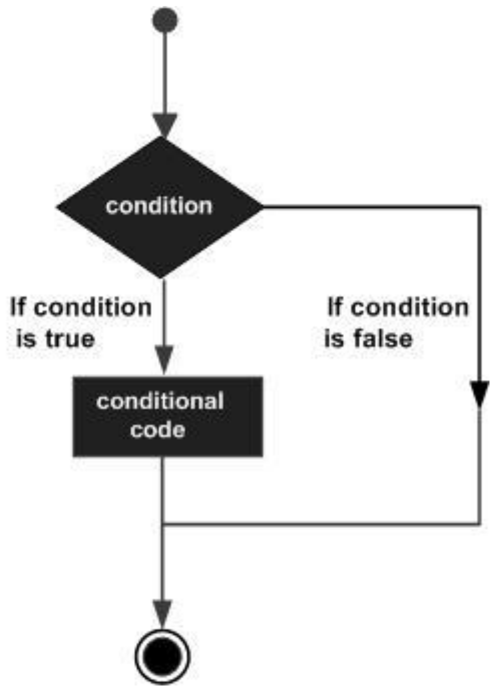
If you execute the above CoffeeScript file, it produces the following output –

```
true
false
false
true
true
true
5.842587018385982e+27
1
0
```

8. CoffeeScript – Conditionals

While programming, we encounter some scenarios where we have to choose a path from a given set of paths. In such situations, we need conditional statements. Conditional statements help us take decisions and perform right actions.

Following is the general form of a typical decision-making structure found in most of the programming languages.



JavaScript supports the **if** statement (including its variants) and **switch** statement. In addition to the conditionals available in JavaScript, CoffeeScript includes the **unless** statement, the negation of if, and even more.

Following are the conditional statements provided by CoffeeScript.

S.N.	Statement & Description
1	**if statement** An **if statement** consists of a Boolean expression followed by one or more statements. These statements execute when the given Boolean expression is true.
2	**if...else statement** An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is false.
3	**unless statement** An **unless** statement is similar to **if** with a Boolean expression followed by one or more statements except. These statements execute when a given Boolean expression is false.
4	**unless...else statement**

	An **unless statement** can be followed by an optional **else statement**, which executes when a Boolean expression is true.
5	**switch statement** A **switch** statement allows a variable to be tested for equality against a list of values.

if Statement

The **if** statement is the fundamental control statement that allows us to make decisions and execute statements conditionally.

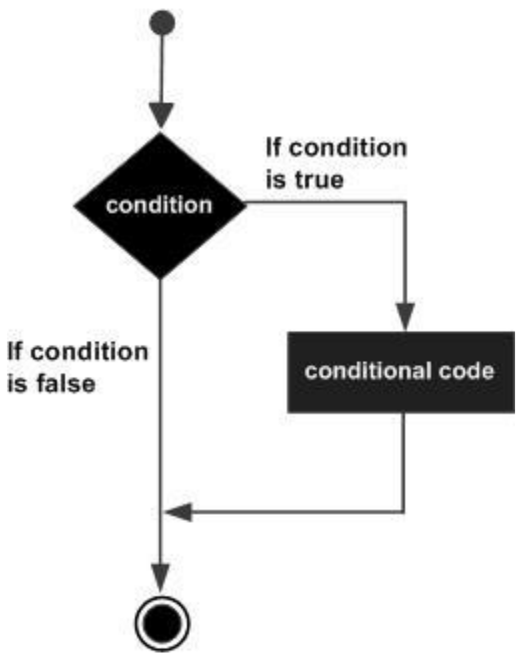
The **if** statement in CoffeeScript is similar to that we have in JavaScript. The difference is that while writing an **if** statement in CoffeeScript, there is no need to use parentheses to specify the Boolean condition. Also, instead of curly braces, we separate the body of the conditional statement by using proper indentations.

Syntax

Given below is the syntax of the **if** statement in CoffeeScript. It contains a keyword **if**, soon after the **if** keyword, we have to specify a Boolean expression which is followed by a block of statements. If the given expression is **true,** then the code in the **if** block is executed.

```
if expression
   Statement(s) to be executed if expression is true
```

Flow Diagram



Example

The following example demonstrates how to use the **if** statement in CoffeeScript. Save this code in a file with the name **if_example.coffee**

```
name = "Ramu"
score = 60
if score>=40
  console.log "Congratulations you have passed the examination"
```

Open the command prompt and compile the .coffee file as shown below.

```
c:\> coffee -c if_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var name, score;

  name = "Ramu";

  score = 60;

  if (score >= 40) {
    console.log("Congratulations you have passed the examination");
  }

}).call(this);
```

Now, open the command prompt again and run the CoffeeScript file as shown below.

```
c:\> coffee if_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Congratulations you have passed the examination
```

if…else Statement

The **if** statement executes the given block of code if the specified Boolean expression is true. What if the Boolean expression is false?

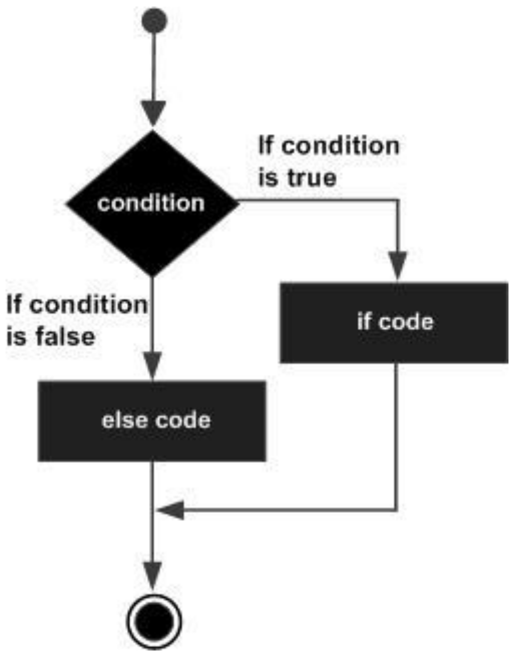
The **'if...else'** statement is the next form of control statement that allows CoffeeScript to execute statements in a more controlled way. It will have an **else** block which executes when the Boolean expression is **false**.

Syntax

Given below is the syntax of the **if-else** statement in CoffeeScript. If the given expression is true, then the statements in the **if** block are executed and if it is false the statements in the **else** block are executed.

```
if expression
   Statement(s) to be executed if the expression is true
else
   Statement(s) to be executed if the expression is false
```

Flow Diagram



Example

The following example demonstrates how to use the **if-else** statement in CoffeeScript. Save this code in a file with name **if_else_example.coffee**

```
name = "Ramu"
score = 30
if score>=40
  console.log "Congratulations have passed the examination"
else
  console.log "Sorry try again"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c if_else_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var name, score;

  name = "Ramu";

  score = 30;

  if (score >= 40) {
    console.log("Congratulations have passed the examination");
  } else {
    console.log("Sorry try again");
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as –

```
c:\> coffee if_else_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Sorry try again
```

unless Statement

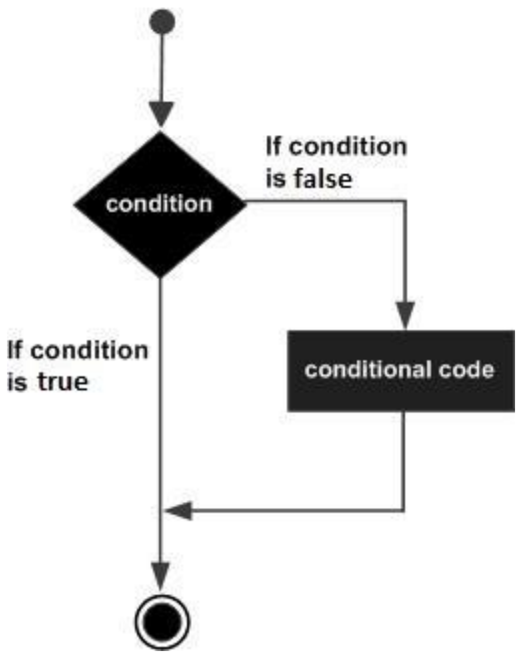
The **unless** statement is an additional feature provided by CoffeeScript. It is exactly opposite to the **if** statement. The code in the **unless** block will be executed if the given condition is **false.**

Syntax

Given below is the syntax of the **unless** statement in CoffeeScript. Just like **if** statement, it contains an **unless** keyword, a Boolean expression, and a block of code which will be executed if the given expression is **false**.

```
unless expression
   Statement(s) to be executed if the expression is false
```

Flow Diagram



Example

The following example demonstrates the usage of **unless** statement in CoffeeScript. Save this code in a file with the name **unless_example.coffee**

```
name = "Ramu"
score = 30
unless score>=40
  console.log "Sorry try again"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c unless_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var name, score;

  name = "Ramu";

  score = 30;

  if (!(score >= 40)) {
    console.log("Sorry try again");
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as –

```
c:\> coffee unless_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Sorry try again
```

unless…else Statement

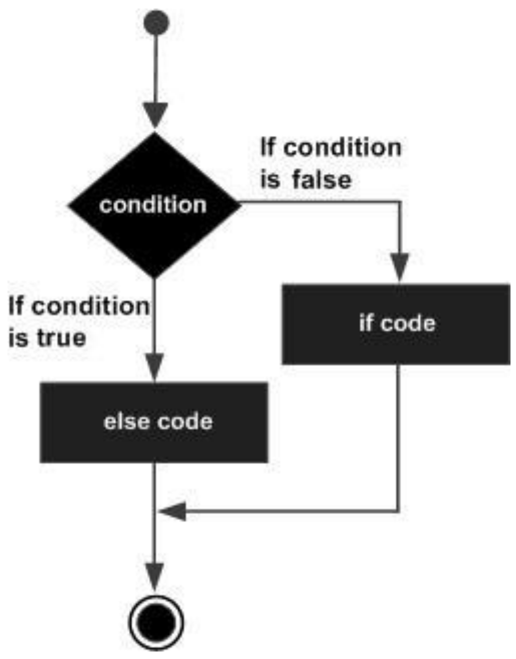
Just like the **if else** statement, we also have an **unless else** statement in CoffeeScript. It contains a Boolean expression, an **unless** block, and an **else** block. If the given expression is **false**, the **unless** block is executed and if it is true, the **else** block is executed.

Syntax

Given below is the syntax of the **unless else** statement in CoffeeScript.

```
unless expression
   Statement(s) to be executed if the expression is false
else
   Statement(s) to be executed if the expression is true
```

Flow Diagram



Example

The following example demonstrates the usage of **unless-else** statement in CoffeeScript. Save this code in a file with the name **unless_else_example.coffee**

```
name = "Ramu"
score = 60
unless score>=40
  console.log "Sorry try again"
else
  console.log "Congratulations you have passed the exam"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c unless_else_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var name, score;

  name = "Ramu";

  score = 60;

  if (!(score >= 40)) {
    console.log("Sorry try again");
  } else {
    console.log("Congratulations you have passed the exam");
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee unless_else_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Congratulations you have passed the exam
```

switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a **case**, and the variable being switched on is checked for each switch case. Here is the syntax of **switch** in JavaScript.

```
switch (expression){
   case condition 1: statement(s)
   break;

   case condition 2: statement(s)
   break;

   case condition n: statement(s)
   break;

   default: statement(s)
}
```

In JavaScript, after each switch case, we have to use the **break** statement. If we accidentally forget the **break** statement, then there is a chance of falling from one switch case to other.

Switch Statement in CoffeeScript

CoffeeScript resolves this problem by using the combination of **switch-when-else** clauses. Here we have an optional switch expression followed by case statements.

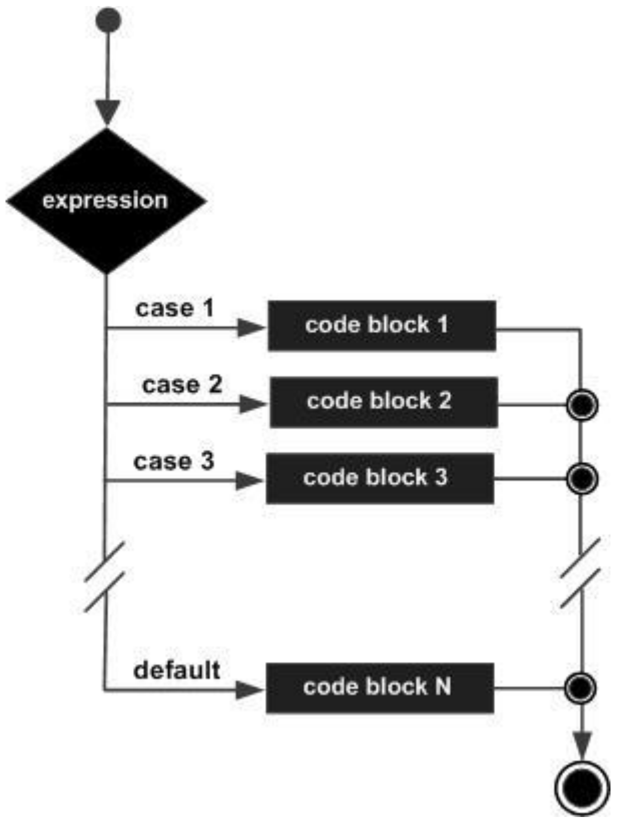
Each case statement have two clauses **when** and **then**. The **when** is followed by condition and **then** is followed by the set of statements that are to be executed if that particular condition is met. And finally, we have the optional **else** clause which holds the action for the default condition.

Syntax

Given below is the syntax of the **switch** statement in CoffeeScript. We specify the expression without parentheses and we separate the case statements by maintaining proper indentations.

```
switch expression
   when condition1 then statements
   when condition2 then statements
   when condition3 then statements
   else statements
```

Flow Diagram



Example

The following example demonstrates the usage of switch statement in CoffeeScript. Save this code in a file with name **switch_example.coffee**

```
name="Ramu"
score=75
message = switch
   when score>=75 then "Congrats your grade is A"
   when score>=60 then "Your grade is B"
   when score>=50 then "Your grade is C"
   when score>=35 then "Your grade is D"
   else "Your grade is F and you are failed in the exam"
console.log message
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c switch_exmple.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var message, name, score;

  name = "Ramu";

  score = 75;

  message = (function() {
    switch (false) {
      case !(score >= 75):
        return "Congrats your grade is A";
      case !(score >= 60):
        return "Your grade is B";
      case !(score >= 50):
        return "Your grade is C";
      case !(score >= 35):
        return "Your grade is D";
      default:
        return "Your grade is F and you are failed in the exam";
    }
  })();

  console.log(message);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as –

```
c:\> coffee switch_exmple.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Congrats your grade is A
```

Multiple values for when clause

We can also specify multiple values for a single when clause by separating them using commas (,) in the switch cases.

Example

The following example shows how to write a CoffeeScript switch statement by specifying multiple values for the **when** clause. Save this code in a file with name **switch_multiple_example.coffee**

```
name="Ramu"
score=75
message = switch name
   when "Ramu","Mohammed" then "You have passed the examination with grade A"
   when "John","Julia" then "You have passed the examination with grade is B"
   when "Rajan" then "Sorry you failed in the examination"
   else "No result"
console.log message
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c switch_multiple_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var message, name, score;

  name = "Ramu";

  score = 75;

  message = (function() {
    switch (name) {
      case "Ramu":
```

```
      case "Mohammed":
        return "You have passed the examination with grade A";
      case "John":
      case "Julia":
        return "You have passed the examination with grade is B";
      case "Rajan":
        return "Sorry you failed in the examination";
      default:
        return "No result";
    }
  })();

  console.log(message);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee switch_multiple_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
You have passed the examination with grade A
```

The then Keyword in CoffeeScript

The **if** and **unless** statements are block statements that are written in multiple lines. CoffeeScript provides the **then** keyword using which we can write the **if** and the **unless** statements in a single line.

Following are the statements in CoffeeScript that are written using **then** keyword.

S.N.	Statement & Description
1	**if-then statement** Using the if-then statement we can write the **if** statement of CoffeeScript in a single line. It consists of a Boolean expression followed by then keyword, which is followed by one or more statements. These statements execute when the given Boolean expression is true.
2	**if-then...else statement** The if-then statement can be followed by an optional **else** statement, which executes when the Boolean expression is false. Using if-then...else statement, we can write the if...else statement in a single line.

3	**unless-then statement** Using the unless-then statement, we can write the **unless** statement of CoffeeScript in a single line. It consists of a Boolean expression followed by **then** keyword, which is followed by one or more statements. These statements execute when the given Boolean expression is false.
4	unless...then else statement The unless-then statement can be followed by an optional **else** statement, which executes when the Boolean expression is true. Using if-then...else statement, we can write the if...else statement in a single line.

if…then Statement

Using the **if-then** statement, we can write the **if** statement of CoffeeScript in a single line. It consists of a Boolean expression followed by then keyword, which is followed by one or more statements. These statements execute when the given Boolean expression is true.

Syntax

Following is the syntax of the **if-then** statement in CoffeeScript.

```
if expression then Statement(s) to be executed if expression is true
```

Example

Given below is the example of the **if-then** statement of CoffeeScript. Save this code in a file with name **if_then_exmple.coffee**

```
name = "Ramu"

score = 60

if score>40 then console.log "Congratulations you have passed the examination"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c if_then_exmple.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0

(function() {

  var name, score;


  name = "Ramu";


  score = 60;

  if (score > 40) {
    console.log("Congratulations you have passed the examination");
  }

}).call(this);
```

Now, open the **Node.js command** prompt again and run the CoffeeScript file as –

```
c:\> coffee if_then_exmple.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Congratulations you have passed the exam
```

if…then…else Statement

Using the **if-then...else** statement, we can write the **if** statement of CoffeeScript in a single line. It consists of a Boolean expression followed by **then** keyword, which is followed by one or more statements. These statements execute when the given Boolean expression is true.

Syntax

Following is the syntax of the **if-then** statement in CoffeeScript.

```
if expression then Statement(s) to be executed if expression is true
```

Example

Given below is the example of the **if-then** statement of CoffeeScript. Save this code in a file with name **if_then_exmple.coffee**

```
name = "Ramu"
score = 30
if score>=40 then console.log "Congratulations you have passed the examination" else
console.log "Sorry try again"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c if_then_exmple.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var name, score;

  name = "Ramu";
  score = 30;

  if (score >= 40) {
    console.log("Congratulations you have passed the examination");
  } else {
    console.log("Sorry try again");
  }

}).call(this);
```

Now, open the **Node.js command** prompt again and run the CoffeeScript file as –

```
c:\> coffee if_then_exmple.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Sorry try again
```

unless…then Statement

Using the **unless..then** statement, we can write the unless statement of CoffeeScript in a single line. It consists of a Boolean expression followed by **then** keyword, which is followed by one or more statements. These statements execute when the given Boolean expression is false.

Syntax

Following the syntax of the **unless-then** statement in CoffeeScript.

```
unless expression then Statement(s) to be executed if expression is false
```

Example

Given below is the example of the **unless-then** statement of CoffeeScript. Save the following example in a file with name **unless_then_exmple.coffee**

```
name = "Ramu"
score = 30
unless score>=40 then console.log "Sorry try again"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c unless_then_exmple.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var name, score;

  name = "Ramu";

  score = 30;

  if (!(score >= 40)) {
    console.log("Sorry try again");
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as –

```
c:\> coffee unless_then_exmple.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Sorry try again
```

unless-then…else Statement

Using the **unless-then** statement, we can write the **unless** statement of CoffeeScript in a single line. It consists of a Boolean expression followed by then keyword, which is followed by one or more statements. These statements execute when the given Boolean expression is false.

Syntax

Following the syntax of the **unless-then** statement in CoffeeScript.

```
unless expression then Statement(s) to be executed if expression is false
```

Example

Given below is the example of the **unless-then** statement of CoffeeScript. Save the following example in a file with name **unless_then_exmple.coffee**

```
name = "Ramu"
```

```
score = 60
unless score>=40 then console.log "Sorry try again" else console.log "congratulations you have passed the examination."
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c unless_then_exmple.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var name, score;

  name = "Ramu";

  score = 60;

  if (!(score >= 40)) {
    console.log("Sorry try again");
  } else {
    console.log("congratulations you have passed the examination.");
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as –

```
c:\> coffee unless_then_exmple.coffee
```

On executing, the CoffeeScript file produces the following output.

```
congratulations you have passed the examination.
```

postfix if and postfix unless Statements

In CoffeeScript, you can also write the **if** and **unless** statements having a code block first followed by **if** or **unless** keyword as shown below. This is the postfix form of those statements. It comes handy while writing programs in CoffeeScript.

```
#Postfix if
Statements to be executed if expression
```

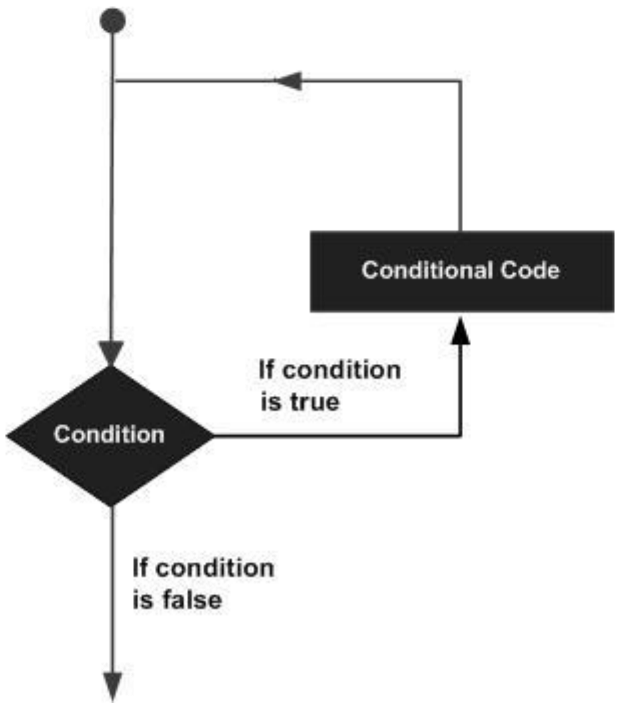
```
#Postfix unless
Statements to be executed unless expression
```

9. CoffeeScript – Loops

While coding, you may encounter a situation where you need to execute a block of code over and over again. In such situations, you can use loop statements.

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages



JavaScript provides **while**, **for** and **for..in** loops. The loops in CoffeeScript are similar to those in JavaScript.

while loop and its variants are the only loop constructs in CoffeeScript. Instead of the commonly used **for** loop, CoffeeScript provides you **Comprehensions** which are discussed in detail in later chapters.

The while loop in CoffeeScript

The **while** loop is the only low-level loop that CoffeeScript provides. It contains a Boolean expression and a block of statements. The **while** loop executes the specified block of statements repeatedly as long as the given Boolean expression is true. Once the expression becomes false, the loop terminates.

Syntax

Following is the syntax of the **while** loop in CoffeeScript. Here, there is no need of the parenthesis to specify the Boolean expression and we have to indent the body of the loop using (consistent number of) whitespaces instead of wrapping it with curly braces.

```
while expression
   statements to be executed
```

Example

The following example demonstrates the usage of while loop in CoffeeScript. Save this code in a file with name **while_loop_example.coffee**

```
console.log "Starting Loop "
count = 0
while count < 10
   console.log "Current Count : " + count
   count++;

console.log "Set the variable to different value and then try"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c while_loop_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var count;

  console.log("Starting Loop ");

  count = 0;

  while (count < 10) {
    console.log("Current Count : " + count);
    count++;
  }

  console.log("Set the variable to different value and then try");

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee while_loop_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Starting Loop
Current Count : 0
Current Count : 1
Current Count : 2
Current Count : 3
Current Count : 4
Current Count : 5
Current Count : 6
Current Count : 7
Current Count : 8
Current Count : 9
Set the variable to different value and then try
```

Variants of while

The While loop in CoffeeScript have two variants namely the **until variant** and the **loop variant.**

S.N.	Loop Type & Description
1	**until variant of while** The **until** variant of the **while** loop contains a Boolean expression and a block of code. The code block of this loop is executed as long as the given Boolean expression is false.
2	**loop variant of while** The **loop** variant is equivalent to the **while** loop with true value (**while true**). The statements in this loop will be executed repeatedly until we exit the loop using the **Break** statement.

The until Variant of while

The **until** alternative provided by the CoffeeScript is exactly opposite to the **while** loop. It contains a Boolean expression and a block of code. The code block of the **until** loop is executed as long as the given Boolean expression is false.

Syntax

Given below is the syntax of the until loop in CoffeeScript.

```
until expression
   statements to be executed if the given condition Is false
```

Example

The following example demonstrates the usage of until loop in CoffeeScript. Save this code in a file with name **until_loop_example.coffee**

```
console.log "Starting Loop "
count = 0
until count > 10
   console.log "Current Count : " + count
   count++;

console.log "Set the variable to different value and then try"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c until_loop_example.coffee
```

On compiling, it gives you the following JavaScript. Here you can observe that the **until** loop is converted into **while** not in the resultant JavaScript code.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var count;

  console.log("Starting Loop ");

  count = 0;

  while (!(count > 10)) {
    console.log("Current Count : " + count);
    count++;
  }

  console.log("Set the variable to different value and then try");

}).call(this);
```

Now, open the **command prompt** again and run the Coffee Script file as shown below.

```
c:\> coffee until_loop_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Starting Loop
Current Count : 0
Current Count : 1
Current Count : 2
Current Count : 3
Current Count : 4
Current Count : 5
Current Count : 6
Current Count : 7
Current Count : 8
Current Count : 9
Set the variable to different value and then try
```

The loop Variant of while

The **loop** variant is equivalent to the while loop with true value (**while true**). The statements in this loop will be executed repeatedly until we exit the loop using the **break** statement

Syntax

Given below is the syntax of the loop alternative of the while loop in CoffeeScript.

```
loop
   statements to be executed repeatedly
   condition to exit the loop
```

Example

The following example demonstrates the usage of until loop in CoffeeScript. Here we have used the Math function **random()** to generate random numbers, and if the number generated is 3, we are exiting the loop using **break** statement. Save this code in a file with name **until_loop_example.coffee**

```
loop
   num = Math.random()*8|0
   console.log num
   if num == 5 then break
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c loop_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var num;

  while (true) {
    num = Math.random() * 8 | 0;
    console.log(num);
    if (num === 5) {
      break;
    }
  }
}).call(this);
```

Now, open the **command prompt** again and run the Coffee Script file as shown below.

```
c:\> coffee loop_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
2
0
2
3
7
4
6
2
0
1
4
6
5
```

10. CoffeeScript – Comprehensions

In the previous chapter, we have learnt various loops provided by CoffeeScript, **while** and its variants. In addition to those, CoffeeScript provides additional loop structures known as **comprehensions**.

These comprehensions replace the **for** loop in other programming languages, if we add the optional guard clauses and the value of the current array index explicitly. Using comprehensions, we can iterate arrays as well as objects and the comprehensions that iterate arrays are expressions, and we can return them in a function or assign to a variable.

S.N.	Statement & Description
1	**for..in comprehensions** The for..in comprehension is the basic form of comprehension in CoffeeScript using this we can iterate the elements of a list or array.
2	**for..of comprehensions** Just like Arrays CoffeeScriptScript provides a containers to store key-value pairs known as objects. We can iterate objects using the for..of comprehensions provided by CoffeeScript.
3	**list comprehensions** The list comprehensions in CoffeeScript are used to map an array of objects to another array.

for…in Comprehensions

The **for..in** comprehension is the basic form of comprehension in CoffeeScript. Using this, we can iterate the elements of a list or array.

Syntax

Suppose we have an array of elements in CoffeeScript as *['element1', 'element2', 'element3']* then you can iterate these elements using the **for-in** comprehension as shown below.

```
for element in ['element1', 'element2', 'element3']
   console.log element
```

Example

The following example demonstrates the usage of **for…in** comprehension in CoffeeScript. Save this code in a file with name **for_in_comprehension.coffee**

```
for student in ['Ram', 'Mohammed', 'John']
   console.log student
```

Open the **Node.js command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c for_in_comprehension.coffee
```

On compiling, it gives you the following JavaScript. Here you can observe that the comprehension is converted into the **for** loop.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var i, len, ref, student;

  ref = ['Ram', 'Mohammed', 'John'];
  for (i = 0, len = ref.length; i < len; i++) {
    student = ref[i];
    console.log(student);
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee for_in_comprehension.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Ram
Mohammed
John
```

for…of Comprehensions

Just like Arrays, CoffeeScript provides **containers** to store key-value pairs known as **objects**. We can iterate objects using the **for..of** comprehensions provided by CoffeeScript.

Syntax

Suppose we have an object in CoffeeScript as *{ key1: value, key2: value, key3: value}* then you can iterate these elements using the **for..of** comprehension as shown below.

```
for key,value of { key1: value, key2: value, key3: value}
   console.log key+"::"+value
```

Example

The following example demonstrates the usage of the **for..of** comprehension provided by CoffeeScript. Save this code in a file with name **for_of_example.coffee**

```
for key,value of { name: "Mohammed", age: 24, phone: 9848022338}
   console.log key+"::"+value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c for_of_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var key, ref, value;

  ref = {
    name: "Mohammed",
    age: 24,
    phone: 9848022338
  };
  for (key in ref) {
    value = ref[key];
    console.log(key + "::" + value);
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee for_of_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
name::Mohammed
age::24
phone::9848022338
```

Note – We will discuss arrays, objects, and ranges in detail in individual chapters later in this tutorial.

list Comprehensions

In CoffeeScript, we can also store a group of objects in an array. The **list** comprehensions are used to map an array of objects to another array.

Syntax

Suppose we have an array of objects in CoffeeScript as *[{key1: "value", key2: value}, {key1: "value", key2: value}]* then you can iterate these elements using the **list** comprehension as shown below.

```
for key,value of  [ {key1: "value", key2: value}, {key1: "value", key2: value} ]
   console.log key+"::"+value
```

Example

The following example demonstrates the usage of the **list** comprehension provided by CoffeeScript. Save this code in a file with name**list_comprehensions.coffee**

```
students =[
    name: "Mohammed"
    age: 24
    phone: 9848022338
  ,
    name: "Ram"
    age: 25
    phone: 9800000000
  ,
    name: "Ram"
    age: 25
    phone: 9800000000
 ]

names = (student.name for student in students)
console.log names
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c list_comprehensions.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var names, student, students;

  students = [
    {
      name: "Mohammed",
      age: 24,
      phone: 9848022338
    }, {
      name: "Ram",
      age: 25,
      phone: 9800000000
    }, {
      name: "Ram",
      age: 25,
      phone: 9800000000
    }
  ];

  names = (function() {
    var i, len, results;
    results = [];
    for (i = 0, len = students.length; i < len; i++) {
      student = students[i];
      results.push(student.name);
    }
    return results;
  })();

  console.log(names);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> list_comprehensions.coffee
```

On executing, the CoffeeScript file produces the following output.

```
[ 'Mohammed', 'Ram', 'Ram' ]
```

Index of comprehensions

The list/array of elements have an index which can be used in comprehensions. You can use it in comprehensions using a variable as shown below.

```
for student,i in [element1, element2, element3]
```

Example

The following example demonstrates the usage of index of the **for…in** comprehension in CoffeeScript. Save this code in a file with name **for_in_index.coffee**

```
for student,i in ['Ram', 'Mohammed', 'John']
   console.log "The name of the student with id "+i+" is: "+student
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c for_in_index.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var i, j, len, ref, student;

  ref = ['Ram', 'Mohammed', 'John'];
  for (i = j = 0, len = ref.length; j < len; i = ++j) {
    student = ref[i];
    console.log("The name of the student with id " + i + " is: " + student);
  }
}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee for_in_index.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The name of the student with id 0 is: Ram
The name of the student with id 1 is: Mohammed
The name of the student with id 2 is: John
```

Postfix form of comprehensions

Just like postfix **if** and **unless**, CoffeeScript provides the postfix form of the Comprehensions which comes handy while writing the code. Using this, we can write the **for..in** comprehension in a single line as shown below.

```
#Postfix for..in comprehension
console.log student for student in ['Ram', 'Mohammed', 'John'


#postfix for..of comprehension
console.log key+"::"+value for key,value of { name: "Mohammed", age: 24, phone: 9848022338}
```

Example

```
#Postfix for..in comprehension
console.log student for student in ['Ram', 'Mohammed', 'John'


#postfix for..of comprehension
console.log key+"::"+value for key,value of { name: "Mohammed", age: 24, phone: 9848022338}
```

Postfix for..in comprehension

The following example demonstrates the usage of postfix form of the for..in comprehension provided by CoffeeScript. Save this code in a file with name**for_in_postfix.coffee**

```
console.log student for student in ['Ram', 'Mohammed', 'John']
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c for_in_postfix.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var i, len, ref, student;


  ref = ['Ram', 'Mohammed', 'John'];
  for (i = 0, len = ref.length; i < len; i++) {
    student = ref[i];
    console.log(student);
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee for_in_postfix.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Ram
Mohammed
John
```

Postfix for..of comprehension

The following example demonstrates the usage of postfix form of the **for..of** comprehension provided by CoffeeScript. Save this code in a file with name**for_of_postfix.coffee**

```
console.log key+"::"+value for key,value of { name: "Mohammed", age: 24, phone: 9848022338}
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c for_of_postfix.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var key, ref, value;

  ref = {
    name: "Mohammed",
    age: 24,
    phone: 9848022338
  };
  for (key in ref) {
    value = ref[key];
    console.log(key + "::" + value);
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee for_of_postfix.coffee
```

On executing, the CoffeeScript file produces the following output.

```
name::Mohammed
age::24
phone::9848022338
```

Assigning to a variable

The comprehension we use to iterate arrays can be assigned to a variable and also returned by a function.

Example

Consider the example given below. Here you can observe that we have retrieved the elements of an array using **for..in** comprehension and assigned this to a variable named **names**. And we also have a function which returns a comprehension explicitly using the**return** keyword. Save this code in a file with name **example.coffee**

```
my_function =->
   student = ['Ram', 'Mohammed', 'John']

   #Assigning comprehension to a variable
   names = (x for x in student )
   console.log "The contents of the variable names are ::"+names

   #Returning the comprehension
   return x for x in student
console.log "The value returned by the function is "+my_function()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var my_function;

  my_function = function() {
    var i, len, names, student, x;
    student = ['Ram', 'Mohammed', 'John'];
    names = (function() {
      var i, len, results;
      results = [];
      for (i = 0, len = student.length; i < len; i++) {
        x = student[i];
        results.push(x);
      }
      return results;
    })();
    console.log("The contents of the variable names are ::" + names);
    for (i = 0, len = student.length; i < len; i++) {
      x = student[i];
      return x;
    }
  };

  console.log("The value returned by the function is " + my_function());

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The contents of the variable names are ::Ram,Mohammed,John
The value returned by the function is Ram
```

The by keyword

CoffeeScript provides ranges to define a list of elements. For example, the range [1..10] is equivalent to [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] where, every element is incremented by 1. We can also change this increment using the **by** keyword of comprehensions.

Example

The following example demonstrates the usage of the **by** keyword of the **for..in** comprehension provided by CoffeeScript. Save this code in a file with name**by_keyword_example.coffee**

```
array = (num for num in [1..10] by 2)
console.log array
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c by_keyword_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var array, num;

  array = (function() {
    var i, results;
    results = [];
    for (num = i = 1; i <= 10; num = i += 2) {
      results.push(num);
    }
    return results;
  })();

  console.log(array);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee by_keyword_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
[ 1, 3, 5, 7, 9 ]
```

11. CoffeeScript – Functions

A function is a block of reusable code that can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmers in writing modular codes.

Functions allow a programmer to divide a big program into a number of small and manageable functions.

In general, using JavaScript, we can define two types of functions – **named functions**, the regular functions with function name body and, **Function expressions**. Using function expressions, we can assign functions to variables.

```
//named function
function sayHello(){
   return("Hello there");
 }

//function expressions
var message = function sayHello(){
   return("Hello there");
 }
```

Functions in CoffeeScript

The syntax of function in CoffeeScript is simpler as compared to JavaScript. In CoffeeScript, we define only function expressions.

The **function** keyword is eliminated in CoffeeScript. To define a function here, we have to use a thin arrow (**->**).

Behind the scenes, the CoffeeScript compiler converts the arrow in to the function definition in JavaScript. (**function() {});**)

It is not mandatory to use the **return** keyword in CoffeeScript. Every function in CoffeeScript returns the last statement in the function automatically.

- If we want to return to the calling function or return a value before we reach the end of the function, then we can use the **return** keyword.

- In addition to in-line functions (functions that are in single line), we can also define multi-line functions in CoffeeScript. Since the curly braces are eliminated, we can do it by maintaining proper indentations.

Defining a Function

Following is the syntax of defining a function in CoffeeScript.

```
function_name = -> function_body
```

Example

Given below is an example of a function in CoffeeScript. In here, we have created a function named **greet**. This function automatically returns the statement in it. Save it in a file with the name **function_example.coffee**

```
greet = -> "This is an example of a function"
```

Compile it by executing the following command in the command prompt.

```
c:\>coffee -c function_example.coffee
```

On compiling, it generates the following JavaScript code. Here you can observe that the CoffeeScript compiler automatically returned the string value in the function named **greet()**.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var greet;

  greet = greet() {
    return "This is an example of a function";
  };

}).call(this);
```

Multi-line Functions

We can also define a function with multiple lines by maintaining indentations instead of curly braces. But we have to be consistent with the indentation we follow for a line throughout a function.

```
greet =  ->
  console.log "How hello how are you"
```

On compiling, the above CoffeeScript gives you the following JavaScript code. The CoffeeScript compiler grabs the body of the function that we have separated using indentations and placed within the curly braces.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var greet;
  greet = function() {
    return console.log("Sum of the two numbers is " + c);
  };

}).call(this);
```

On executing, the above CoffeeScript generates the following output.

Functions with Arguments

We can also specify arguments in a function using parenthesis as shown below.

```
add =(a,b) ->
  c=a+b
  console.log "Sum of the two numbers is: "+c
```

On compiling the above CoffeeScript file, it will generate the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var my_function;

  my_function = function() {
    return "This is an example of a function";
  };

}).call(this);
```

Invoking a Function

After defining a function, we need to invoke that function. You can simply invoke a function by placing parenthesis after its name as shown in the following example.

```
add = ->
  a=20;b=30
  c=a+b
  console.log "Sum of the two numbers is: "+c
add()
```

On compiling, the above example gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var add;

  add = function() {
    var a, b, c;
    a = 20;
    b = 30;
    c = a + b;
    return console.log("Sum of the two numbers is: " + c);
  };
  add();
}).call(this);
```

On executing the above CoffeeScript code, it generates the following output.

```
Sum of the two numbers is: 50
```

Invoking Functions with Arguments

In the same way, we can invoke a function with arguments by passing them to it as shown below.

```
my_function argument_1,argument_2


or


my_function (argument_1,argument_2)
```

Note – While invoking a function by passing arguments to it, the usage of parenthesis is optional.

In the following example, we have created a function named **add()** that accepts two parameters and we have invoked it.

```
add =(a,b) ->
  c=a+b
  console.log "Sum of the two numbers is: "+c
add 10,20
```

On compiling, the above example gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var add;

  add = function(a, b) {
    var c;
    c = a + b;
    return console.log("Sum of the two numbers is: " + c);
  };

  add(10, 20);

}).call(this);
```

On executing, the above CoffeeScript code it generates the following output.

```
Sum of the two numbers is: 30
```

Default Arguments

CoffeeScript supports default arguments too. We can assign default values to the arguments of a function, as shown in the following example.

```
add =(a = 1, b = 2) ->
  c=a+b
  console.log "Sum of the two numbers is: "+c
add 10,20

#Calling the function with default arguments
add()
```

On compiling, the above CoffeeScript generates the following JavaScript file.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var add;

  add = function(a, b) {
    var c;
    if (a == null) {
      a = 1;
    }
    if (b == null) {
      b = 2;
    }
    c = a + b;
    return console.log("Sum of the two numbers is: " + c);
  };

  add(10, 20);
  add()

}).call(this);
```

On executing the above CoffeeScript code, it generates the following output.

```
Sum of the two numbers is: 30
Sum of the two numbers is: 3
```

CoffeeScript – Object Oriented

12. CoffeeScript – Strings

The String object lets you work with a series of characters. As in most of the programming languages, the Strings in CoffeeScript are declared using quotes as –

```
my_string = "Hello how are you"
```

On compiling, it will generate the following JavaScript code.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var my_string;

  my_string = "Hello how are you";

  console.log(my_string);

}).call(this);
```

String Concatenation

We can concatenate two strings using the **"+"** symbol as shown below.

```
new_string = "Hello how are you "+" Welcome to Tutorials point"
console.log new_String
```

On compiling, it will generate the following JavaScript code.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var new_string;

  new_string = "Hello how are you " + " Welcome to Tutorials point";

  console.log(new_String);

}).call(this);
```

If you execute the above example, you can observe the concatenated String as shown below.

```
Hello how are youWelcome to Tutorials point
```

String Interpolation

CoffeeScript also provides a feature known as **String interpolation** to include variables in stings. This feature of CoffeeScript was inspired from Ruby language.

String interpolation was done using the double quotes **""**, a hash tag **#** and a pair of curly braces **{ }**. The String is declared in curly braces and the variable that is to be interpolated is wrapped within the curly braces which are prefixed by a hash tag as shown below.

```
name = "Raju"
age = 26
message ="Hello #{name} your age is #{age}"
console.log message
```

On compiling the above example, it generates the following JavaScript. Here you can observe the String interpolation is converted into normal concatenation using the **+** symbol.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var age, message, name;

  name = "Raju";

  age = 26;

  message = "Hello " + name + " your age is " + age;

  console.log(message);

}).call(this);
```

If you execute the above CoffeeScript code, it gives you the following output.

```
Hello Raju your age is 26
```

The variable that is passed as **#{variable}** is interpolated only if the string is enclosed between double quotes **" "**. Using single quotes **' '** instead of double quotes produces the line as it is without interpolation. Consider the following example.

```
name = "Raju"
age = 26
```

```
message ='Hello #{name} your age is #{age}'
console.log message
```

If we use single quotes instead of double quotes in interpolation, you will get the following output.

```
Hello #{name} your age is #{age}
```

CoffeeScript allows multiple lines in Strings without concatenating them as shown below.

```
my_string = "hello how are you
Welcome to tutorials point
Have a nice day."
console.log my_string
```

It generates the following output.

```
hello how are you Welcome to tutorials point Have a nice day.
```

JavaScript String Object

The String object of JavaScript lets you work with a series of characters. This object provides you a lot of methods to perform various operations on Stings.

Since we can use JavaScript libraries in our CoffeeScript code, we can use all those methods in our CoffeeScript programs.

String Methods

Following is the list of methods of the String object of JavaScript. Click on the name of these methods to get an example demonstrating their usage in CoffeeScript.

Method	Description
charAt()	Returns the character at the specified index.
charCodeAt()	Returns a number indicating the Unicode value of the character at the given index.
concat()	Combines the text of two strings and returns a new string.
indexOf()	Returns the index within the calling String object of the first occurrence of the specified value, or -1 if not found.
lastIndexOf()	Returns the index within the calling String object of the last occurrence of the specified value, or -1 if not found.
localeCompare()	Returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.
match()	Used to match a regular expression against a string.

search()	Executes the search for a match between a regular expression and a specified string.
slice()	Extracts a section of a string and returns a new string.
split()	Splits a String object into an array of strings by separating the string into substrings.
substr()	Returns the characters in a string beginning at the specified location through the specified number of characters.
substring()	Returns the characters in a string between two indexes into the string.
toLocaleLowerCase()	The characters within a string are converted to lower case while respecting the current locale.
toLocaleUpperCase()	The characters within a string are converted to upper case while respecting the current locale.
toLowerCase()	Returns the calling string value converted to lower case.
toUpperCase()	Returns the calling string value converted to uppercase.

charAt method

Description

The **charAt()** method of JavaScript returns the character of the current string that exists in the specified index.

Characters in a string are indexed from left to right. The index of the first character is 0, and the index of the last character is one less than the length of the string. (stringName_length – 1)

Syntax

Given below is the syntax of charAt() method of JavaScript. We can use the same method from the CoffeeScript code.

```
string.charAt(index);
```

It accepts an integer value representing the index of the String and returns the character at the specified index.

Example

The following example demonstrates the usage of **charAt()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_charat.coffee**

```
str = "This is string"


console.log "The character at the index (0) is:" + str.charAt 0

console.log "The character at the index (1) is:" + str.charAt 1

console.log "The character at the index (2) is:" + str.charAt 2
console.log "The character at the index (3) is:" + str.charAt 3
console.log "The character at the index (4) is:" + str.charAt 4
console.log "The character at the index (5) is:" + str.charAt 5
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c string_charat.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var str;

  str = "This is string";

  console.log("The character at the index (0) is:" + str.charAt(0));

  console.log("The character at the index (1) is:" + str.charAt(1));

  console.log("The character at the index (2) is:" + str.charAt(2));

  console.log("The character at the index (3) is:" + str.charAt(3));

  console.log("The character at the index (4) is:" + str.charAt(4));

  console.log("The character at the index (5) is:" + str.charAt(5));

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_charat.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The character at the index (0) is:T
The character at the index (1) is:h
The character at the index (2) is:i
The character at the index (3) is:s
The character at the index (4) is:
```

```
The character at the index (5) is:i
```

charCodeAt Method

Description

This method returns a number indicating the Unicode value of the character at the given index.

Unicode code points range from 0 to 1,114,111. The first 128 Unicode code points are a direct match of the ASCII character encoding. **charCodeAt()** always returns a value that is less than 65,536.

Syntax

Given below is the syntax of charCodeAt() method of JavaScript. We can use the same method from the CoffeeScript code.

```
string. charCodeAt(index)
```

It accepts an integer value representing the index of the String and returns the Unicode value of the character existing at the specified index of the String. It returns **NaN** if the given index is not between 0 and 1 less than the length of the string.

Example

The following example demonstrates the usage of **charCodeAt()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_charcodeat.coffee**

```
str = "This is string"


console.log "The Unicode of the character at the index (0) is:" + str.charCodeAt 0
console.log "The Unicode of the character at the index (1) is:" + str.charCodeAt 1
console.log "The Unicode of the character at the index (2) is:" + str.charCodeAt 2
console.log "The Unicode of the character at the index (3) is:" + str.charCodeAt 3
console.log "The Unicode of the character at the index (4) is:" + str.charCodeAt 4
console.log "The Unicode of the character at the index (5) is:" + str.charCodeAt 5
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c string_charcodeat.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var str;
  str = "This is string";

  console.log("The Unicode of the character at the index (0) is:" + str.charCodeAt(0));

  console.log("The Unicode of the character at the index (1) is:" + str.charCodeAt(1));

  console.log("The Unicode of the character at the index (2) is:" + str.charCodeAt(2));

  console.log("The Unicode of the character at the index (3) is:" + str.charCodeAt(3));

  console.log("The Unicode of the character at the index (4) is:" + str.charCodeAt(4));

  console.log("The Unicode of the character at the index (5) is:" + str.charCodeAt(5));

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_charcodeat.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The Unicode of the character at the index (0) is:84
The Unicode of the character at the index (1) is:104
The Unicode of the character at the index (2) is:105
The Unicode of the character at the index (3) is:115
The Unicode of the character at the index (4) is:32
The Unicode of the character at the index (5) is:105
```

concat() Method

Description

This method adds two or more strings and returns the resultant string.

Syntax

Given below is the syntax of **concat()** method of JavaScript. We can use the same method from the CoffeeScript code.

```
string.concat(string2, string3[, ..., stringN]);
```

Example

The following example demonstrates the usage of **concat()** method of JavaScript in CoffeeScript code. Save this code in a file with the name **string_concat.coffee**

```
str1 = "Hello how are you. "
str2 = "Welcome to Tutorials Point."
str3 = str1.concat str2

console.log "Concatenated String :" + str3
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c string_concat.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var str1, str2, str3;

  str1 = new String("Hello how are you. ");

  str2 = new String("Welcome to Tutorials Point.");

  str3 = str1.concat(str2);

  console.log("Concatenated String :" + str3);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_concat.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Concatenated String :Hello how are you. Welcome to Tutorials Point.
```

indexOf() Method

Description

This method accepts a sub string and returns the index of its **first** occurrence within the calling String object. It also accepts an optional parameter **fromIndex** which will be the starting point of the search. This method returns −1 if the value is not found.

Syntax

Given below is the syntax of **indexOf()** method of JavaScript. We can use the same method from the CoffeeScript code.

```
string.indexOf(searchValue[, fromIndex])
```

Example

The following example demonstrates the usage of **indexOf()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_indexof.coffee**

```
str1 = "This is string one"
index = str1.indexOf "string"
console.log "indexOf the given string string is :" + index


index = str1.indexOf "one"
console.log "indexOf the given string one is :" + index
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c string_indexof.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var index, str1;


  str1 = "This is string one";


  index = str1.indexOf("string");


  console.log("indexOf the given string string is :" + index);


  index = str1.indexOf("one");

  console.log("indexOf the given string one is :" + index);


}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_indexof.coffee
```

On executing, the CoffeeScript file produces the following output.

```
indexOf the given string string is :8
indexOf the given string one is :15
```

lastIndexOf() Method

Description

This method accepts a sub string and returns the index of its **last** occurrence within the calling String object. It also accepts an optional parameter **fromIndex** to start the search from, it returns -1 if the value is not found.

Syntax

Given below is the syntax of **lastIndexOf()** method of JavaScript. We can use the same method from the CoffeeScript code.

```
string.lastIndexOf(searchValue[, fromIndex])
```

Example

The following example demonstrates the usage of **lastIndexOf()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_lastindexof.coffee**

```
str1 = "A sentence does not end with because because, because is a conjunction."
index = str1.lastIndexOf  "because"
console.log "lastIndexOf the given string because is :" + index


index = str1.lastIndexOf "a"
console.log "lastIndexOf the letter a is :"+ index
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c string_last_indexof.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var index, str1;

  str1 = "A sentence does not end with because, because because is a conjunction.";

  index = str1.lastIndexOf("because");

  console.log("lastIndexOf the given string because is :" + index);

  index = str1.lastIndexOf("a");

  console.log("lastIndexOf the letter a is :" + index);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_last_indexof.coffee
```

On executing, the CoffeeScript file produces the following output.

```
lastIndexOf the given string because is :46
lastIndexOf the letter a is :57
```

localeCompare() Method

Description

This method accepts a string and compares it with the calling String object. If both are equal, it returns 0; else it returns -1 or 1. And if the string passed as parameter comes first in the sorted order according to local browser language, it returns 1; and if the calling string comes first in the sorted order, -1 is returned.

Syntax

Given below is the syntax of **localeCompare()** method of JavaScript. We can use the same method from the CoffeeScript code.

```
string.localeCompare( param )
```

Example

The following example demonstrates the usage of **localeCompare()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_localecompare.coffee**

```
str1 = "This is beautiful string"
str2 = "This is beautiful string"
str3 = "abcd"
str4 = "xyz"
console.log "The value of str1:: "+str1
console.log "The value of str2:: "+str2
console.log "The value of str3:: "+str3
console.log "comparing the strings str1 and str2 ::"

index = str1.localeCompare str2
switch index
   when 0 then console.log "Both strings are equal"
   when 1 then console.log "Both strings are not equal and the string passed as 
parameter will be first in the sorted order."
   when -1 then console.log "Both strings are not equal and the calling string object 
will be first in the sorted order."



console.log "comparing the strings str1 and str3 ::"
index = str1.localeCompare str3
switch index
   when 0 then console.log "Both strings are equal"
   when 1 then console.log "Both strings are not equal and the string passed as 
parameter will be first in the sorted order."
   when -1 then console.log "Both strings are not equal and the calling string object 
will be first in the sorted order."

console.log "comparing the strings str1 and str4 ::"
index = str1.localeCompare str4
index = str1.localeCompare str3
switch index
   when 0 then console.log "Both strings are equal"
   when 1 then console.log "Both strings are not equal and the string passed as 
parameter will be first in the sorted order."
   when -1 then console.log "Both strings are not equal and the calling string object 
will be first in the sorted order."
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c string_localecompare.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var index, str1, str2, str3, str4;

  str1 = "This is beautiful string";

  str2 = "This is beautiful string";

  str3 = "abcd";

  str4 = "xyz";

  console.log("The value of str1:: " + str1);

  console.log("The value of str2:: " + str2);

  console.log("The value of str3:: " + str3);

  console.log("comparing the strings str1 and str2 ::");

  index = str1.localeCompare(str2);

  switch (index) {
    case 0:
      console.log("Both strings are equal");
      break;
    case 1:
      console.log("Both strings are not equal and the string passed as parameter will
be first in the sorted order.");
      break;
    case -1:
      console.log("Both strings are not equal and the calling string object will be
first in the sorted order.");
```

```
  }

  console.log("comparing the strings str1 and str3 ::");

  index = str1.localeCompare(str3);

  switch (index) {
    case 0:
      console.log("Both strings are equal");
      break;
    case 1:
      console.log("Both strings are not equal and the string passed as parameter will
be first in the sorted order.");
      break;
    case -1:
      console.log("Both strings are not equal and the calling string object will be
first in the sorted order.");
  }

  console.log("comparing the strings str1 and str4 ::");

  index = str1.localeCompare(str4);

  index = str1.localeCompare(str3);

  switch (index) {
    case 0:
      console.log("Both strings are equal");
      break;
    case 1:
      console.log("Both strings are not equal and the string passed as parameter will
be first in the sorted order.");
      break;
    case -1:
      console.log("Both strings are not equal and the calling string object will be
first in the sorted order.");
  }
```

```
}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_localecompare.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The value of str1:: This is beautiful string
The value of str2:: This is beautiful string
The value of str3:: abcd
comparing the strings str1 and str2 ::
Both strings are equal
comparing the strings str1 and str3 ::
Both strings are not equal and the string passed as parameter will be first in the
sorted order.
comparing the strings str1 and str4 ::
Both strings are not equal and the string passed as parameter will be first in the
sorted o
```

match() Method

Description

This method is used to retrieve the matches when matching a string against a regular expression. It works similar to **regexp.exec(string)** without the **g** flag and it returns an array with all matches with the **g** flag.

Syntax

Given below is the syntax of **match()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
string.match( param )
```

Example

The following example demonstrates the usage of **match()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_localecompare.coffee**

```
str = "For more information, see Chapter 3.4.5.1";
re = /(chapter \d+(\.\d)*)/i;
found = str.match re

console.log found
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c coffee string_match.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var found, re, str;

  str = "For more information, see Chapter 3.4.5.1";

  re = /(chapter \d+(\.\d)*)/i;

  found = str.match(re);

  console.log(found);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_match.coffee
```

On executing, the CoffeeScript file produces the following output.

```
[ 'Chapter 3.4.5.1',
  'Chapter 3.4.5.1',
  '.1',
  index: 26,
  input: 'For more information, see Chapter 3.4.5.1' ]
```

search() Method

Description

This method accepts a regular expression in the form of object and searches the calling string for the given regular expression. If a match occurs, it returns the index of the regular expression inside the string and if it doesn't, it returns the value **-1**.

Syntax

Given below is the syntax of **search()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
string.search(regexp)
```

Example

The following example demonstrates the usage of **search()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_search.coffee**

```
regex = /apples/gi
string = "Apples are round, and apples are juicy."

if string.search(regex) == -1
  console.log "Does not contain Apples"
else
  console.log "Contains Apples"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c coffee string_search.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var regex, string;

  regex = /apples/gi;

  string = "Apples are round, and apples are juicy.";

  if (string.search(regex) === -1) {
    console.log("Does not contain Apples");
  } else {
    console.log("Contains Apples");
  }
}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_search.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Contains Apples
```

slice() Method

Description

This method accepts **begin** and **end index** values, and returns the portion of the calling string object that exists between the given index values. If we doesn't pass the end index value it takes the end of the string as the end index value.

Note – We can also slice a string using ranges.

Syntax

Given below is the syntax of **slice()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
string.slice( beginslice [, endSlice] )
```

Example

The following example demonstrates the usage of **slice()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_slice.coffee**

```
my_string = "Apples are round, and apples are juicy."
result = my_string.slice 3, -2


console.log "The required slice of the string is :: "+result
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c coffee string_slice.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var my_string, result;


  my_string = "Apples are round, and apples are juicy.";
  result = my_string.slice(3, -2);

  console.log("The required slice of the string is :: " + result);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_slice.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The required slice of the string is :: les are round, and apples are juic
```

split() Method

Description

This method is used to split a string in to small parts. It accepts a special character and an integer. The character acts as a separator and indicates where to split the string and the integer indicates into how many parts the string is to be divided. If we do not pass a separator, the whole string is returned.

Syntax

Given below is the syntax of **split()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
string.split([separator][, limit])
```

Example

The following example demonstrates the usage of **split()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_split.coffee**

```
my_string = "Apples are round, and apples are juicy."
result = my_string.split " ", 3

console.log "The two resultant strings of the split operation are :: "
console.log my_string
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c coffee string_split.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var my_string, result;

  my_string = "Apples are round, and apples are juicy.";

  result = my_string.split(" ", 3);

  console.log("The two resultant strings of the split operation are :: ");

  console.log(my_string);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_split.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The two resultant strings of the split operation are ::
Apples are round, and apples are juicy.
```

substr() Method

Description

This method is used to return a required substring of a string. It accepts an integer value indicating the starting value of the substring and the length of the string and returns the required substring. If the start value is negative, then **substr()** method uses it as a character index from the end of the string.

Syntax

Given below is the syntax of **substr()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
string.substr(start[, length])
```

Example

The following example demonstrates the usage of **substr()** method of JavaScript in CoffeeScript code. Save this code in a file with name **string_substr.coffee**

```
str = "Apples are round, and apples are juicy.";
```

```
console.log "The sub string having start and length as (1,2) is : " + str.substr 1,2
console.log "The sub string having start and length as (-2,2) is : " + str.substr -
2,2
console.log "The sub string having start and length as (1) is : " + str.substr 1
console.log "The sub string having start and length as (-20, 2) is : " + str.substr -
20,2
console.log "The sub string having start and length as (20, 2) is : " + str.substr
20,2;
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c coffee string_substr.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var str;

  str = "Apples are round, and apples are juicy.";

  console.log("The sub string having start and length as (1,2) is : " + str.substr(1,
2));

  console.log("The sub string having start and length as (-2,2) is : " + str.substr(-
2, 2));

  console.log("The sub string having start and length as (1) is : " + str.substr(1));

  console.log("The sub string having start and length as (-20, 2) is : " +
str.substr(-20, 2));

  console.log("The sub string having start and length as (20, 2) is : " +
str.substr(20, 2));

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee string_substr.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The sub string having start and length as (1,2) is : pp
The sub string having start and length as (-2,2) is : y.
The sub string having start and length as (1) is : pples are round, and apples are
juicy.
The sub string having start and length as (-20, 2) is : nd
The sub string having start and length as (20, 2) is : d
```

toLocaleLowerCase() method

Description

This method is used to convert the characters within a string to lower case while respecting the current locale. For most languages, it returns the same output as toLowerCase.

Syntax

Given below is the syntax of **toLocaleLowerCase( )** method of JavaScript. We can use the same method in the CoffeeScript code.

```
string.toLocaleLowerCase( )
```

Example

The following example demonstrates the usage of **toLocaleLowerCase( )** method of JavaScript in CoffeeScript code. Save this code in a file with name**toLocaleLowerCase.coffee**

```
str = "APPLES ARE ROUND AND ORANGES ARE JUCY.";
console.log str.toLocaleLowerCase( )
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c coffee toLocaleLowerCase.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var str;

  str = "APPLES ARE ROUND AND ORANGES ARE JUICY.";

  console.log(str.toLocaleLowerCase());

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee toLocaleLowerCase.coffee
```

On executing, the CoffeeScript file produces the following output.

```
apples are round and oranges are juicy.
```

toLocaleUpperCase() method

Description

This method is used to convert the characters within a string to uppercase while respecting the current locale. For most languages, it returns the same output as toUpperCase.

Syntax

Given below is the syntax of **toLocaleUpperCase( )** method of JavaScript. We can use the same method in the CoffeeScript code.

```
string.toLocaleUpperCase( )
```

Example

The following example demonstrates the usage of **toLocaleUpperCase( )** method of JavaScript in CoffeeScript code. Save this code in a file with name**toLocaleUpperCase.coffee**

```
str = "Apples are round, and Apples are Juicy.";
console.log str.toLocaleUpperCase( )
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c coffee toLocaleUpperCase.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var str;

  str = "Apples are round, and Apples are Juicy.";

  console.log(str.toLocaleUpperCase());

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee toLocaleUpperCase.coffee
```

On executing, the CoffeeScript file produces the following output.

```
APPLES ARE ROUND, AND APPLES ARE JUICY.
```

toLowerCase() method

Description

This method returns the calling string value converted to lowercase.

Syntax

Given below is the syntax of **toLowerCase( )** method of JavaScript. We can use the same method in the CoffeeScript code.

```
string.toLowerCase( )
```

Example

The following example demonstrates the usage of **toLowerCase( )** method of JavaScript in CoffeeScript code. Save this code in a file with name **tolowercase.coffee**

```
// Generated by CoffeeScript 1.10.0
(function() {
  var str;

  str = "Apples are round, and Apples are Juicy.";

  console.log(str.toLowerCase());

}).call(this);
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c coffee tolowercase.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var str;
  str = "Apples are round, and Apples are Juicy.";

  console.log(str.toLowerCase());

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee tolowercase.coffee
```

On executing, the CoffeeScript file produces the following output.

```
apples are round, and apples are juicy.
```

toUpperCase() method

Description

This method returns the calling string value converted to Uppercase.

Syntax

Given below is the syntax of **toUpperCase( )** method of JavaScript. We can use the same method in the CoffeeScript code.

```
string.toUpperCase( )
```

Example

The following example demonstrates the usage of **toUpperCase( )** method of JavaScript in CoffeeScript code. Save this code in a file with name **touppercase.coffee**

```
str = "Apples are round, and Apples are Juicy."
console.log str.toUpperCase( )
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c coffee touppercase.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var str;
  str = "Apples are round, and Apples are Juicy.";

  console.log(str.toUpperCase());

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee touppercase.coffee
```

On executing, the CoffeeScript file produces the following output.

```
APPLES ARE ROUND, AND APPLES ARE JUICY.
```

13. CoffeeScript – Arrays

The Array object lets you store multiple values in a single variable. It stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Syntax

To create an array, we have to instantiate it using the **new** operator as shown below.

```
array = new (element1, element2,....elementN)
```

The Array() constructor accepts the list of string or integer types. We can also specify the length of the array by passing a single integer to its constructor.

We can also define an array by simply providing the list of its elements in the square braces (**[ ]**) as shown below.

```
array = [element1, element2, ......elementN]
```

Example

Following is an example of defining an array in CoffeeScript. Save this code in a file with name **array_example.coffee**

```
student = ["Rahman","Ramu","Ravi","Robert"]
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c array_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var student;

  student = ["Rahman", "Ramu", "Ravi", "Robert"];

}).call(this);
```

New line instead of comma

We can also remove the comma (**,**) between the elements of an array by creating each element in a new line by maintaining proper indentation as shown below.

```
student = [
  "Rahman"
  "Ramu"
  "Ravi"
  "Robert"
  ]
```

Comprehensions over arrays

We can retrieve the values of an array using comprehensions.

Example

The following example demonstrates the retrieval of elements of an array using comprehensions. Save this code in a file with name **array_comprehensions.coffee**

```
students = [ "Rahman", "Ramu", "Ravi", "Robert" ]
console.log student for student in students
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c array_comprehensions.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var i, len, student, students;

  students = ["Rahman", "Ramu", "Ravi", "Robert"];

  for (i = 0, len = students.length; i – len; i++) {
    student = students[i];
    console.log(student);
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee array_comprehensions.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Rahman
Ramu
Ravi
Robert
```

Unlike the Arrays in other programming languages the arrays in CoffeeScript can have multiple types of data i.e. both string and numericals.

Example

Here is an example of a CoffeeScript array holding multiple types of data.

```
students = [ "Rahman", "Ramu", "Ravi", "Robert",21 ]
```

14. CoffeeScript – Objects

Objects in CoffeeScript are similar to those in JavaScript. These are a collection of the properties, where a property includes a key and a value separated by a semi colon (**:**). In short, CoffeeScript objects are a collection of key-value pairs. The objects are defined using curly braces, an empty object is represented as **{}.**

Syntax

Given below is the syntax of an object in CoffeeScript. In here, we place the key-value pairs of the objects within the curly braces and they are separated using comma (**,**).

```
object ={key1: value, key2: value,......keyN: value}
```

Example

Following is an example of defining an object in CoffeeScript. Save this code in a file with name **objects_example.coffee**

```
student = {name: "Mohammed", age: 24, phone: 9848022338 }
```

Open the **command prompt** and compile the .coffee file as shown below.

```
> coffee -c objects_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var student;

  student = {
    name: "Mohammed",
    age: 24,
    phone: 9848022338
  };

}).call(this);
```

Just as in arrays, we can remove the commas by specifying the key-value pairs in new lines as shown below.

```
student = {
  name: "Mohammed"
age: 24
phone: 9848022338
}
```

Indentations instead of curly braces

Just like other block statements in CoffeeScript, we can use indentations instead of curly braces **{}** as shown in the following example.

Example

We can rewrite the above example without curly braces as shown below.

```
student =
  name: "Mohammed"
  age: 24
  phone: 9848022338
```

Nested objects

In CoffeeScript, we can write objects within objects.

Example

The following example demonstrates the nested objects in CoffeeScript. Save this code in a file with name **nested_objects.coffee**

```
contact =
  personal:
    email: "personal@gmail.com"
    phone:  9848022338
  professional:
    email: "professional@gmail.com"
    phone:  9848033228
```

Open the **command prompt** and compile the .coffee file as shown below.

```
> coffee -c nested_objects.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var contact;
  contact = {
    personal: {
      email: "personal@gmail.com",
      phone: 9848022338
    },
    professional: {
      email: "professional@gmail.com",
      phone: 9848033228
    }
  };

}).call(this);
```

Comprehensions over objects

To iterate over the contents of an object, we can use comprehensions. Iterating the contents of an object is same as iterating the the contents of an array. In objects, since we have to retrive two elements keys and values we will use two variables.

Example

The following is an example showing how to iterate the contents of an object using comprehensions. Save this code in a file with name **object_comprehensions.coffee**

```
student =
  name: "Mohammed"
  age: 24
  phone: 9848022338

console.log key+"::"+value for key,value of student
```

Open the **command prompt** and compile the .coffee file as shown below.

```
> coffee -c object_comprehensions.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var key, student, value;
  student = {
    name: "Mohammed",
    age: 24,
    phone: 9848022338
  };

  for (key in student) {
    value = student[key];
    console.log(key(+"::" + value));
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
> coffee object_comprehensions.coffee
```

On executing, the CoffeeScript file produces the following output.

```
name::Mohammed
age::24
phone::9848022338
```

Arrays of Objects

In CoffeeScript, an array can also contain objects in as shown below.

```
  a = [
     object1_key1: value
     object1_key2: value
     object1_key3: value
  ,
     object2_key1: value
     object2_key2: value
     object2_key3: value
]
```

The following example shows how to define an array of objects. We can just list the key value pairs of the objects we want in an array by separating them using commas **(,)**.

```
students =[
    name: "Mohammed"
    age: 24
    phone: 9848022338
  ,
    name: "Ram"
    age: 25
    phone: 9800000000
  ,
    name: "Ram"
    age: 25
    phone: 9800000000
 ]
console.log student for student in students
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c array_of_objects.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var i, len, student, students;

  students = [
    {
      name: "Mohammed",
      age: 24,
      phone: 9848022338
    }, {
      name: "Ram",
      age: 25,
      phone: 9800000000
    }, {
      name: "Ram",
      age: 25,
      phone: 9800000000
    }
  ];

  for (i = 0, len = students.length; i < len; i++) {
    student = students[i];
    console.log(student);
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee array_of_objects.coffee
```

On executing, the CoffeeScript file produces the following output.

```
{ name: 'Mohammed', age: 24, phone: 9848022338 }
{ name: 'Ram', age: 25, phone: 9800000000 }
{ name: 'Ram', age: 25, phone: 9800000000 }
```

Reserved Keywords

JavaScript does not allow reserved keywords as property names of an object, if we want use them, we have to wrap them using double quotes **" "**

Example

Consider the following example. Here we have created a property with name **class**, which is a reserved keyword. Save this code in a file with name **reserved_keywords.coffee**

```
student ={
  name: "Mohammed"
  age: 24
  phone: 9848022338
  class: "X"
  }
console.log key+"::"+value for key,value of student
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c reserved_keywords.coffee
```

On compiling, it gives you the following JavaScript. Here you can observe that the CoffeeScript compiler wrapped the keyword class with double quotes on behalf of us.

```
// Generated by CoffeeScript 1.10.0
```

```
(function() {
  var key, student, value;

  student = {
    name: "Mohammed",
    age: 24,
    phone: 9848022338,
    "class": "X"
  };

  for (key in student) {
    value = student[key];
    console.log(key + "::" + value);
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee array_of_objects.coffee
```

On executing, the CoffeeScript file produces the following output.

```
name::Mohammed
age::24
phone::9848022338
class::X
```

15. CoffeeScript – Ranges

In the previous chapter, we have seen Arrays in CoffeeScript, while programming we will face some scenarios where we have to store a sequence of numerical values in an array as shown below.

```
numbers =[1,2,3,4,5,6,7,8,9,10]
```

CoffeeScript provides a shorter way of expressing the arrays containing a sequence of numerical values, known as **ranges**. This feature of CoffeeScript is inspired from Ruby.

Syntax

Ranges are created by two numerical values, the first and last positions in the range, separated by .. or .... With two dots (1..4), the range is inclusive (1, 2, 3, 4); with three dots (1...4), the range excludes the end (1, 2, 3).

Given below is the syntax of ranges in CoffeeScript. We will define the values in a range between square braces **[ ]** just like arrays. In ranges, while storing a sequence of numerical values, instead of providing the values of the whole sequence, we can just specify its **begin** and **end** values separated by two dots (**..**) as shown below.

```
range =[Begin..End]
```

Example

Here is an example of ranges in CoffeeScript. Save this in a file with name ranges_example.coffee.

```
numbers =[0..9]
console.log "The contents of the range are: "+ numbers
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c ranges_example.coffee
```

On compiling, it gives you the following JavaScript. Here you can observe that the range is converted in to complete CoffeeScript array.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var numbers;

  numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

  console.log("The contents of the range are:: " + numbers);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee ranges_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The contents of the range are:: 0,1,2,3,4,5,6,7,8,9
```

Excluding the end Value

The ranges are compiled into complete arrays containing all numbers. If we want to exclude the **end** value, then we have to separate the **start** and **end** elements of the range using three dots (**...**) as shown below.

```
range =[Begin...End]
```

Example

We can rewrite the above example by excluding the **end** value as shown below. Save the following contents in a file with name **range_excluding_end.coffee**

```
numbers =[0...9]
console.log "The contents of the range are:: "+ numbers
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c ranges_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var numbers;

  numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8];

  console.log("The contents of the range are:: " + numbers);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee ranges_example.coffee
```

On executing, the CoffeeScript file produces the following output. In here, you can observe that the end value **9** is excluded.

```
The contents of the range are:: 0,1,2,3,4,5,6,7,8
```

Using Ranges with Variables

We can also define a range by assigning the start and end values to variables.

Example

Consider the following example. Here we have defined a range using variables. Save this code in a file with name **range_variables.coffee**

```
start=0

end=9

numbers =[start..end]

console.log "The contents of the range are: "+ numbers
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c range_variables.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var end, i, numbers, results, start;


  start = 0;


  end = 9;


  numbers = (function() {
    results = [];
    for (var i = start; start <= end ? i <= end : i >= end; start <= end ? i++ : i-
-){ results.push(i); }
    return results;
  }).apply(this);


  console.log("The contents of the range are:: " + numbers);


}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee range_variables.coffee
```

On executing, the CoffeeScript file produces the following output. In here, you can observe that the end value **9** is excluded.

```
The contents of the range are:: 0,1,2,3,4,5,6,7,8,9
```

Ranges with Arrays

We can slice arrays by using them with ranges. Whenever we specify ranges immediately after arrays (variables), then the CoffeeScript compiler converts it in to a **slice()** method call of JavaScript.

Assume that we have an array having numerical values, say 0 to 9, then we can retrieve the first 4 elements of it as shown below.

```
num  = [1, 2, 3, 4, 5, 6, 7, 8, 9]

data = num[0..5]
```

Negative values represent the elements from the end, for example, -1 indicates 9. If we specify a negative number 3 followed by two dots, the last three elements of the array will be extracted.

```
data = num[-3..]
```

If we specify only two dots in the range of an array as **num[..]**, then the complete array will be extracted. We can also replace an array segment with other elements using ranges as shown below.

```
num[2..6] = [13,14,15,16,17]
```

Example

The following example demonstrates the use of ranges with arrays. Save this code in a file with name **range_arrays.coffee**

```
#slicing an array using ranges

num  = [1, 2, 3, 4, 5, 6, 7, 8, 9]

data = num[0..5]

console.log "The first four elements of the array : "+data



#Using negative values

data = num[-3..]

console.log "The last 3 elements of the array : "+data
```

```
#Extracting the whole array
console.log "Total elements of the array : "+num[..]



#Replacing the elements of an array
num[2..6] = [13,14,15,16,17]
console.log "New array : "+num
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c range_arrays.coffee
```

On compiling, it gives you the following JavaScript. Here you can observe that all the ranges are converted in to the slice() method calls of JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var data, num, ref;

  num = [1, 2, 3, 4, 5, 6, 7, 8, 9];

  data = num.slice(0, 6);

  console.log("The first four elements of the array : " + data);

  data = num.slice(-3);

  console.log("The last 3 elements of the array : " + data);

  console.log("Total elements of the array : " + num.slice(0));

  [].splice.apply(num, [2, 5].concat(ref = [13, 14, 15, 16, 17])), ref;

  console.log("New array : " + num);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee range_arrays.coffee
```

On executing, the CoffeeScript file produces the following output. In here, you can observe that the end value **9** is excluded.

```
The first four elements of the array : 1,2,3,4,5,6
The last 3 elements of the array : 7,8,9
Total elements of the array : 1,2,3,4,5,6,7,8,9
New array : 1,2,13,14,15,16,17,8,9
```

Ranges with Strings

We can also use ranges with Strings. If we specify ranges after Strings, then CoffeeScript slices them and returns a new subset of characters.

Example

The following example demonstrates the use of ranges with Strings. Here we have created a string and extracted a substring from it using ranges. Save this code in a file with name **ranges_with_strings.coffee**

```
my_string = "Welcome to tutorialspoint"
new_string = my_string[0..10]
console.log new_string
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c ranges_with_strings.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var my_string, new_string;

  my_string = "Welcome to tutorialspoint";

  new_string = my_string.slice(0, 6);

  console.log(new_string);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee ranges_with_strings.coffee
```

On executing, the CoffeeScript file produces the following output. In here, you can observe that the end value **9** is excluded.

```
Welcome to
```

Comprehensions over Ranges

As objects and arrays, we can also iterate the elements of a range using comprehensions.

Example

Following is an example of using comprehensions over ranges. Here we have created a range and retrieved the elements in it using comprehensions. Save this code in a file with the name **comprehensions_over_ranges.coffee**

```
numbers =[0..9]
console.log "The elements of the range are: "
console.log num for num in numbers
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c comprehensions_over_ranges.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var i, len, num, numbers;

  numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

  console.log("The elements of the range are: ");

  for (i = 0, len = numbers.length; i < len; i++) {
    num = numbers[i];
    console.log(num);
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee comprehensions_over_ranges.coffee
```

On executing, the CoffeeScript file produces the following output. In here, you can observe that the end value **9** is excluded.

```
The elements of the range are:
0
1
2
3
4
5
6
7
8
9
```

In the same way We can also change this increment using the by keyword of comprehensions.

```
array = (num for num in [1..10] by 2)
console.log array
```

16. CoffeeScript – Splat

In the previous chapters, we have seen how to define a function and invoke a function and pass arguments to it. In general, we can pass a fixed number of arguments to a function. While programming, we may face situations where we need to pass variable arguments to these functions. In JavaScript, we use objects to accept variable number of arguments to a function.

CoffeeScript provides a feature called **splats** to pass multiple arguments to functions. We use splats in functions by placing three dots after the argument name and, it is denoted by**...**

Syntax

Given below is the syntax of accepting multiple arguments within a function using splats.

```
my_function = (arguments...)->
   ............
   ............
   ............
```

Example

Following is an example of accepting multiple arguments within a function, using splats. Here we have defined a function named **indian_team()** using splats. We are calling this function thrice and we are passing 4 players, 6 players, and full squad simultaneously, each time we call it. Since we have used splats in the function definition, it accepts variable number of arguments each time we call it. Save this code in a file with name**splats_definition.coffee**

```
indian_team = (first, second, others...) ->
  Captain = first
  WiseCaptain = second
  team  = others
  console.log "Captain: " +Captain
  console.log "Wise captain: " +WiseCaptain
  console.log "Other team members: " +team

#Passing 4 arguments
console.log "############## Four Players ############"
indian_team "Mahendra Singh Dhoni", "Virat Kohli", "Shikhar Dhawan", "Rohit Sharma"

#Passing 6 arguments
console.log "############## Six Players ############"
indian_team "Mahendra Singh Dhoni", "Virat Kohli", "Shikhar Dhawan", "Rohit Sharma",
"Gurkeerat Singh Mann", "Rishi Dhawan"
```

```
#Passing full squad

console.log "############## Full squad #############"

indian_team "Mahendra Singh Dhoni", "Virat Kohli", "Shikhar Dhawan", "Rohit Sharma", "Gurkeerat Singh Mann", "Rishi Dhawan", "Ravindra Jadeja", "Axar Patel", "Jasprit Bumrah", "Umesh Yadav", "Harbhajan Singh", "Ashish Nehra", "Hardik Pandya", "Suresh Raina", "Yuvraj Singh", "Ajinkya Rahane"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c splats_definition.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0

(function() {

  var indian_team,

    slice = [].slice;


  indian_team = function() {

    var Captain, WiseCaptain, first, others, second, team;

    first = arguments[0], second = arguments[1], others = 3 <= arguments.length ? slice.call(arguments, 2) : [];

    Captain = first;

    WiseCaptain = second;

    team = others;

    console.log("Captain: " + Captain);

    console.log("Wise captain: " + WiseCaptain);

    return console.log("Other team members: " + team);

  };


  console.log("############## Four Players ############");


  indian_team("Mahendra Singh Dhoni", "Virat Kohli", "Shikhar Dhawan", "Rohit Sharma");


  console.log("############## Six Players ############");


  indian_team("Mahendra Singh Dhoni", "Virat Kohli", "Shikhar Dhawan", "Rohit Sharma", "Gurkeerat Singh Mann", "Rishi Dhawan");
  console.log("############## Full squad #############");


  indian_team("Mahendra Singh Dhoni", "Virat Kohli", "Shikhar Dhawan", "Rohit
Sharma", "Gurkeerat Singh Mann", "Rishi Dhawan", "Ravindra Jadeja", "Axar Patel",
"Jasprit Bumrah", "Umesh Yadav", "Harbhajan Singh", "Ashish Nehra", "Hardik Pandya",
"Suresh Raina", "Yuvraj Singh", "Ajinkya Rahane");


}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee splats_definition.coffee
```

On executing, the CoffeeScript file produces the following output.

```
############## Four Players ############
Captain: Mahendra Singh Dhoni
Wise captain: Virat Kohli
Other team members: Shikhar Dhawan,Rohit Sharma
############## Six Players ############
Captain: Mahendra Singh Dhoni
Wise captain: Virat Kohli
Other team members: Shikhar Dhawan,Rohit Sharma,Gurkeerat Singh Mann,Rishi Dhawan
############## Full squad #############
Captain: Mahendra Singh Dhoni
Wise captain: Virat Kohli
Other team members: Shikhar Dhawan,Rohit Sharma,Gurkeerat Singh Mann,Rishi
Dhawan,Ravindra Jadeja,Axar Patel,Jasprit Bumrah,Umesh Yadav,Harbhajan Singh,Ashish
Nehra,Hardik Pandya,Suresh Raina,Yuvraj Singh,Ajinkya Rahane
```

Calling Functions using Splats

We can also call a function using splats. For that, we have to create an array holding the elements we need to pass to the function, and we have to call the function by passing the array suffixed by three dots as shown below.

```
my_function values...
```

Example

Following is an example of calling a function using splats. Save this code in a file with name**splats_call.coffee**

```
indian_team = (first, second, others...) ->
  Captain = first
  WiseCaptain = second
  team  = others
  console.log "Captain: " +Captain
  console.log "Wise captain: " +WiseCaptain
  console.log "Other team members: " +team

squad = [
   "Mahendra Singh Dhoni"
   "Virat Kohli"
   "Shikhar Dhawan"
   "Rohit Sharma"
   "Gurkeerat Singh Mann"
   "Rishi Dhawan"
   "R Ashwin"
   "Ravindra Jadeja"
   "Axar Patel"
   "Jasprit Bumrah"
   "Umesh Yadav"
   "Harbhajan Singh"
   "Ashish Nehra"
   "Hardik Pandya"
   "Suresh Raina"
   "Yuvraj Singh"
   "Ajinkya Rahane"
 ]

indian_team squad...
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c splats_call.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var indian_team, squad,
    slice = [].slice;

  indian_team = function() {
    var Captain, WiseCaptain, first, others, second, team;
    first = arguments[0], second = arguments[1], others = 3 <= arguments.length ?
slice.call(arguments, 2) : [];
    Captain = first;
    WiseCaptain = second;
    team = others;
    console.log("Captain: " + Captain);
    console.log("Wise captain: " + WiseCaptain);
    return console.log("Other team members: " + team);
  };

  squad = ["Mahendra Singh Dhoni", "Virat Kohli", "Shikhar Dhawan", "Rohit Sharma",
"Gurkeerat Singh Mann", "Rishi Dhawan", "R Ashwin", "Ravindra Jadeja", "Axar Patel",
"Jasprit Bumrah", "Umesh Yadav", "Harbhajan Singh", "Ashish Nehra", "Hardik Pandya",
"Suresh Raina", "Yuvraj Singh", "Ajinkya Rahane"];

  indian_team.apply(null, squad);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee splats_call.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Captain: Mahendra Singh Dhoni
Wise captain: Virat Kohli
Other team members: Shikhar Dhawan,Rohit Sharma,Gurkeerat Singh Mann,Rishi Dhawan,R
Ashwin,Ravindra Jadeja,Axar Patel,Jasprit Bumrah,Umesh Yadav,Harbhajan Singh,Ashish
Nehra,Hardik Pandya,Suresh Raina,Yuvraj Singh,Ajinkya Rahane
```

Splats with a Tailing Argument

We can also pass tailing arguments to splats. In the example given below, we have passed a tailing argument named **last** after the splat. Save this example in a file with the name **tailing_arguments.coffee**

```
indian_team = (first, second, others..., last) ->
  Captain = first
  WiseCaptain = second
  team  = others
  Wicketkeeper =last
  console.log "Captain: " +Captain
  console.log "Wise captain: " +WiseCaptain
  console.log "Wicket keeper is:"+last
  console.log "Other team members: " +team

squad = [
   "Mahendra Singh Dhoni"
   "Virat Kohli"
   "Shikhar Dhawan"
   "Rohit Sharma"
   "Gurkeerat Singh Mann"
   "Rishi Dhawan"
   "R Ashwin"
   "Ravindra Jadeja"
   "Axar Patel"
   "Jasprit Bumrah"
   "Umesh Yadav"
   "Harbhajan Singh"
   "Ashish Nehra"
   "Hardik Pandya"
   "Suresh Raina"
   "Yuvraj Singh"
   "Ajinkya Rahane"
 ]

indian_team squad...
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c tailing_arguments.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var indian_team, squad,
    slice = [].slice;

  indian_team = function() {
    var Captain, Wicketkeeper, WiseCaptain, first, i, last, others, second, team;
    first = arguments[0], second = arguments[1], others = 4 <= arguments.length ?
slice.call(arguments, 2, i = arguments.length - 1) : (i = 2, []), last =
arguments[i++];
    Captain = first;
    WiseCaptain = second;
    team = others;
    Wicketkeeper = last;
    console.log("Captain: " + Captain);
    console.log("Wise captain: " + WiseCaptain);
    console.log("Wicket keeper is:" + last);
    return console.log("Other team members: " + team);
  };

  squad = ["Mahendra Singh Dhoni", "Virat Kohli", "Shikhar Dhawan", "Rohit Sharma",
"Gurkeerat Singh Mann", "Rishi Dhawan", "R Ashwin", "Ravindra Jadeja", "Axar Patel",
"Jasprit Bumrah", "Umesh Yadav", "Harbhajan Singh", "Ashish Nehra", "Hardik Pandya",
"Suresh Raina", "Yuvraj Singh", "Ajinkya Rahane"];

  indian_team.apply(null, squad);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee tailing_arguments.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Captain: Mahendra Singh Dhoni
```

```
Wise captain: Virat Kohli

Wicket keeper is:Ajinkya Rahane

Other team members: Shikhar Dhawan,Rohit Sharma,Gurkeerat Singh Mann,Rishi Dhawan,R
Ashwin,Ravindra Jadeja,Axar Patel,Jasprit Bumrah,Umesh Yadav,Harbhajan Singh,Ashish
Nehra,Hardik Pandya,Suresh Raina,Yuvraj Singh
```

Comprehensions with Splats

Within the function, we can also iterate the elements of a splat using comprehensions as shown in the following example. Save this code in a file with the name **splats_comprehensions.coffee**

```
indian_team = (first, second, others...) ->
  Captain = first
  WiseCaptain = second
  team  = others
  console.log "Captain: " +Captain
  console.log "Wise captain: " +WiseCaptain
  console.log "Other team members:: "
  console.log member for member in others

squad = [
   "Mahendra Singh Dhoni"
   "Virat Kohli"
   "Shikhar Dhawan"
   "Rohit Sharma"
   "Gurkeerat Singh Mann"
   "Rishi Dhawan"
   "R Ashwin"
   "Ravindra Jadeja"
   "Axar Patel"
   "Jasprit Bumrah"
   "Umesh Yadav"
   "Harbhajan Singh"
   "Ashish Nehra"
   "Hardik Pandya"
   "Suresh Raina"
   "Yuvraj Singh"
   "Ajinkya Rahane"
```

```
 ]


indian_team squad...
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c splats_comprehensions.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var indian_team, squad,
    slice = [].slice;


  indian_team = function() {
    var Captain, WiseCaptain, first, i, len, member, others, results, second, team;
    first = arguments[0], second = arguments[1], others = 3 <= arguments.length ?
slice.call(arguments, 2) : [];
    Captain = first;
    WiseCaptain = second;
    team = others;
    console.log("Captain: " + Captain);
    console.log("Wise captain: " + WiseCaptain);
    console.log("Other team members:: ");
    results = [];
    for (i = 0, len = others.length; i < len; i++) {
      member = others[i];
      results.push(console.log(member));
    }
    return results;
  };


  squad = ["Mahendra Singh Dhoni", "Virat Kohli", "Shikhar Dhawan", "Rohit Sharma",
"Gurkeerat Singh Mann", "Rishi Dhawan", "R Ashwin", "Ravindra Jadeja", "Axar Patel",
"Jasprit Bumrah", "Umesh Yadav", "Harbhajan Singh", "Ashish Nehra", "Hardik Pandya",
"Suresh Raina", "Yuvraj Singh", "Ajinkya Rahane"];


  indian_team.apply(null, squad);
}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee splats_comprehensions.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Captain: Mahendra Singh Dhoni
Wise captain: Virat Kohli
Other team members::
Shikhar Dhawan
Rohit Sharma
Gurkeerat Singh Mann
Rishi Dhawan
R Ashwin
Ravindra Jadeja
Axar Patel
Jasprit Bumrah
Umesh Yadav
Harbhajan Singh
Ashish Nehra
Hardik Pandya
Suresh Raina
Yuvraj Singh
Ajinkya Rahane
```

17. CoffeeScript – Date

The Date object is a data-type built into the JavaScript language. Date objects are created as **new Date( )**.

Once a Date object is created, a number of methods allow you to operate on it. Most methods simply allow you to get and set the year, month, day, hour, minute, second, and millisecond fields of the object, using either local time or UTC (universal, or GMT) time.

The ECMAScript standard requires the Date object to be able to represent any date and time, to millisecond precision, within 100 million days before or after 1/1/1970. This is a range of plus or minus 273,785 years, so JavaScript can represent date and time till the year 275755.

Similar to other JavaScript objects we can also use the date object of our CoffeeScript code.

Date Methods

Following is the list of methods of the **Date** object of JavaScript. Click on the name of these methods to get an example demonstrating their usage in CoffeeScript.

Method	Description
getDate()	Returns the day of the month for the specified date according to local time.
getDay()	Returns the day of the week for the specified date according to local time.
getFullYear()	Returns the year of the specified date according to local time.
getHours()	Returns the hour in the specified date according to local time.
getMilliseconds()	Returns the milliseconds in the specified date according to local time.
getMinutes()	Returns the minutes in the specified date according to local time.
getMonth()	Returns the month in the specified date according to local time.
getSeconds()	Returns the seconds in the specified date according to local time.
getTime()	Returns the numeric value of the specified date as the number of milliseconds since January 1, 1970, 00:00:00 UTC.
getTimezoneOffset()	Returns the time-zone offset in minutes for the current locale.
getUTCDate()	Returns the day (date) of the month in the specified date according to universal time.
getUTCDay()	Returns the day of the week in the specified date according to universal time.
getUTCFullYear()	Returns the year in the specified date according to universal time.
getUTCHours()	Returns the hours in the specified date according to universal time.
getUTCMilliseconds()	Returns the milliseconds in the specified date according to universal time.

getUTCMinutes()	Returns the minutes in the specified date according to universal time.
getUTCMonth()	Returns the month in the specified date according to universal time.
getUTCSeconds()	Returns the seconds in the specified date according to universal time.
getYear()	Deprecated - Returns the year in the specified date according to local time. Use getFullYear instead.
setDate()	Sets the day of the month for a specified date according to local time.
setFullYear()	Sets the full year for a specified date according to local time.
setHours()	Sets the hours for a specified date according to local time.
setMilliseconds()	Sets the milliseconds for a specified date according to local time.
setMinutes()	Sets the minutes for a specified date according to local time.
setMonth()	Sets the month for a specified date according to local time.
setSeconds()	Sets the seconds for a specified date according to local time.
setTime()	Sets the Date object to the time represented by a number of milliseconds since January 1, 1970, 00:00:00 UTC.
setUTCDate()	Sets the day of the month for a specified date according to universal time.
setUTCFullYear()	Sets the full year for a specified date according to universal time.
setUTCHours()	Sets the hour for a specified date according to universal time.
setUTCMilliseconds()	Sets the milliseconds for a specified date according to universal time.
setUTCMinutes()	Sets the minutes for a specified date according to universal time.
setUTCMonth()	Sets the month for a specified date according to universal time.
setUTCSeconds()	Sets the seconds for a specified date according to universal time.
setYear()	Deprecated - Sets the year for a specified date according to local time. Use setFullYear instead.
toDateString()	Returns the "date" portion of the Date as a human-readable string.
toLocaleDateString()	Returns the "date" portion of the Date as a string, using the current locale's conventions.
toLocaleString()	Converts a date to a string, using the current locale's conventions.
toLocaleTimeString()	Returns the "time" portion of the Date as a string, using the current locale's conventions.
toTimeString()	Returns the "time" portion of the Date as a human-readable string.
toUTCString()	Converts a date to a string, using the universal time convention.

getDate() Method

Description

The **getDate()** method returns the day of the month for the specified date according to local time. The value returned by getDate() is an integer between 1 and 31.

Syntax

Given below is the syntax of **getDate()** method.

```
Date.getDate()
```

Example

The following example demonstrates the usage of the **getDate()** method in CoffeeScript. Save this code in a file with name **date_getdate.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
console.log "The day of the current date object is : " + dt.getDate()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getdate.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("December 25, 1995 23:15:00");

  console.log("The day of the current date object is : " + dt.getDate());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getdate.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The date in the current date object is : 25
```

getDay() Method

Description

The **getDay()** method returns the day of the week for the specified date according to local time. The value returned by getDay() is an integer corresponding to the day of the week: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so on.

Syntax

Given below is the syntax of **getDay()** method.

```
Date.getDay()
```

Example

The following example demonstrates the usage of the **getDay()** method in CoffeeScript. Save this code in a file with name **date_getday.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
console.log "The day of the current date object is : " + dt.getDate()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getdate.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("December 25, 1995 23:15:00");

  console.log("The day of the current date object is : " + dt.getDate());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getdate.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The date in the current date object is : 25
```

getFullYear() Method

Description

The **getFullYear()** method returns the year of the specified date according to local time. The value returned by **getFullYear()** is an absolute number. For dates between the years 1000 and 9999, getFullYear() returns a four-digit number, for example, 2008.

Syntax

Given below is the syntax of **getFullYear()** method.

```
getFullYear()
```

Example

The following example demonstrates the usage of the **getFullYear()** method in CoffeeScript. Save this code in a file with name **date_getFullYear.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
console.log "The year the current date object is : " + dt.getFullYear()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getFullYear.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  console.log("The year the current date object is : " + dt.getFullYear());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getfullyear.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The year the current date object is : 2016
```

getHours() Method

Description

The **getHours()** method returns the hour in the specified date according to local time. The value returned by getHours() is an integer between 0 and 23.

Syntax

Given below is the syntax of **getHours()** method.

```
getHours()
```

Example

The following example demonstrates the usage of the **getHours()** method in CoffeeScript. Save this code in a file with name **date_gethours.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
console.log "The hour value in the current date object is : " + dt.getHours()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_gethours.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  console.log("The hour value in the current date object is : " + dt.getHours());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_gethours.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The hour value in the current date object is : 23
```

getMilliseconds() Method

Description

The **getMilliseconds ()** method returns the milliseconds in the specified date according to local time. The value returned by getMilliseconds() is a number between 0 and 999.

Syntax

Given below is the syntax of **getMilliseconds()** method.

```
Date. getMilliseconds()
```

Example

The following example demonstrates the usage of the **getMilliseconds()** method in CoffeeScript. Save this code in a file with name **date_getmilliseconds.coffee**.

```
dt = new Date "February 19, 2016 23:15:25:22"
console.log "The value of the milliseconds in the specified date is : " +
dt.getMilliseconds()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getmilliseconds.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;


  dt = new Date("February 19, 2016 23:15:25:22");


  console.log("The value of the milliseconds in the specified date is : " +
dt.getMilliseconds());


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getmilliseconds.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The value of the milliseconds in the specified date is : 22
```

getMinutes() Method

Description

The **getMinutes()** method returns the minutes in the specified date according to local time. The value returned by getMinutes() is an integer between 0 and 59.

Syntax

Given below is the syntax of **getMinutes()** method.

```
Date. getMinutes()
```

Example

The following example demonstrates the usage of the **getMinutes()** method in CoffeeScript. Save this code in a file with name **date_minutes.coffee.**

```
dt = new Date "February 19, 2016 23:15:25:22"
console.log "The value of the milliseconds in the specified date is : " +
dt.getMinutes()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getminutes.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;


  dt = new Date("February 19, 2016 23:15:25:22");


  console.log("The value of the minutes in the specified date is : " +
dt.getMinutes());


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getminutes.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The value of the minutes in the specified date is : 15
```

getMonth() Method

Description

The **getMonth()** method returns the month in the specified date according to local time. The value returned by getMonth() is an integer between 0 and 11. 0 corresponds to January, 1 to February, and so on.

Syntax

Given below is the syntax of **getMonth()** method.

```
Date. getMonth()
```

Example

The following example demonstrates the usage of the **getMonth()** method in CoffeeScript. Save this code in a file with name **date_getmonth.coffee**.

```
dt = new Date "February 19, 2016 23:15:25:22"
console.log "The month in the specified date is : " + dt.getMonth()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_ getmonth.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:25:22");

  console.log("The month in the specified date is : " + dt.getMonth());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_ getmonth.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The month in the specified date is : 1
```

getUTCSeconds() Method

Description

The **getSeconds()** method returns the seconds in the specified date according to local time. The value returned by getSeconds() is an integer between 0 and 59.

Syntax

Given below is the syntax of **getSeconds()** method.

```
Date.getSeconds()
```

Example

The following example demonstrates the usage of the **getSeconds()** method in CoffeeScript. Save this code in a file with name **date_getseconds.coffee**.

```
dt = new Date "February 19, 2016 23:15:25:22"
console.log "The month in the specified date is : " + dt.getSeconds()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_ getseconds.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:25:22");

  console.log("The month in the specified date is : " + dt.getSeconds());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_ getseconds.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The month in the specified date is : 25
```

getTime() Method

Description

The **getTime()** method returns the numeric value corresponding to the time for the specified date according to universal time. The value returned by the getTime method is the number of milliseconds since 1 January 1970 00:00:00.

Syntax

Given below is the syntax of **getTime()** method.

```
Date.getTime()
```

Example

The following example demonstrates the usage of the **getTime()** method in CoffeeScript. Save this code in a file with name **date_gettime.coffee**.

```
dt = new Date "February 19, 2016 23:15:25:22"
console.log "The time in the specified date is : " + dt.getTime()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_ gettime.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:25:22");

  console.log("The time in the specified date is : " + dt.getTime());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_ gettime.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The time in the specified date is : 1455903925022
```

getTimezoneOffset() Method

Description

The **getTimezoneOffset()** method returns the time-zone offset in minutes for the current locale. The time-zone offset is the minutes in difference, the Greenwich Mean Time (GMT) is relative to your local time.

For example, if your time zone is GMT+10, -600 will be returned. Daylight savings time prevents this value from being a constant.

Syntax

Given below is the syntax of **getTimezoneOffset()** method.

```
Date. getTimezoneOffset()
```

Example

The following example demonstrates the usage of the **getTimezoneOffset()** method in CoffeeScript. Save this code in a file with name **date_gettimezoneoffset.coffee**.

```
dt = new Date "February 19, 2016 23:15:25:22"

console.log "The time zone offset value of the specified date is : " +
dt.getTimezoneOffset()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_ gettimezoneoffset.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;


  dt = new Date("February 19, 2016 23:15:25:22");


  console.log("The time zone offset value of the specified date is : " +
dt.getTimezoneOffset());


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_ gettimezoneoffset.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The time zone offset value of the specified date is : -330
```

getUTCDate() Method

Description

The **getUTCDate()** method returns the day of the month in the specified date according to universal time. The value returned by getUTCDateis an integer between 1 and 31.

Syntax

Given below is the syntax of **getUTCDate()** method.

```
Date. getUTCDate()
```

Example

The following example demonstrates the usage of the **getUTCDate()** method in CoffeeScript. Save this code in a file with name **date_getutcdate.coffee**.

```
dt = new Date "February 19, 2016 23:15:25:22"
console.log "The UTC date value of the specified date is : " + dt.getUTCDate()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getutcdate.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:25:22");

  console.log("The UTC date value of the specified date is : " + dt.getUTCDate());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getutcdate.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The UTC date value of the specified date is : 19
```

getUTCDay() method

Description

The **getUTCDay()** method returns the day of the week in the specified date according to universal time. The value returned by getUTCDay() is an integer corresponding to the day of the week: 0 for Sunday, 1 for Monday, 2 for Tuesday, and so on.

Syntax

Given below is the syntax of **getUTCDay()** method.

```
Date. getUTCDay()
```

Example

The following example demonstrates the usage of the **getUTCDay()** method in CoffeeScript. Save this code in a file with name **date_getutcday.coffee**.

```
dt = new Date "February 19, 2016 23:15:25:22"
console.log "The UTC day in the specified date is : " + dt.getUTCDay()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getutcday.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:25:22");

  console.log("The UTC day in the specified date is : " + dt.getUTCDay());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getutcday.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The UTC day in the specified date is : 5
```

getUTCFullYear() Method

Description

The **getUTCFullYear()** method returns the year in the specified date according to universal time. The value returned by getUTCFullYear() is an absolute number that is compliant with year-2000, for example, 2008.

Syntax

Given below is the syntax of **getUTCFullYear()** method.

```
Date.getUTCFullYear()
```

Example

The following example demonstrates the usage of the **getUTCFullYear()** method in CoffeeScript. Save this code in a file with name **date_getutcfullyear.coffee**.

```
dt = new Date "February 19, 2016 23:15:25:22"
console.log "The UTC full year in the specified date is : " + dt.getFullYear()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getutcfullyear.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:25:22");

  console.log("The UTC full year in the specified date is : " + dt.getFullYear());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getutcfullyear.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The UTC full year in the specified date is : 2016
```

getUTCHours() Method

Description

The **getUTCHours()** method returns the hours in the specified date according to universal time. The value returned by getUTCHours() is an integer between 0 and 23.

Syntax

Given below is the syntax of **getUTCHours()** method.

```
Date.getUTCHours()
```

Example

The following example demonstrates the usage of the **getUTCHours()** method in CoffeeScript. Save this code in a file with name **date_getutchours.coffee**.

```
dt = new Date()
console.log "The UTC hours in the specified date is : " + dt.getUTCHours()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getutchours.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date();

  console.log("The UTC hours in the specified date is : " + dt.getUTCHours());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getutchours.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The UTC hours in the specified date is : 17
```

getUTCMilliseconds() Method

Description

The **getUTCMilliseconds()** method returns the milliseconds in the specified date according to universal time. The value returned by getUTCMilliseconds() is an integer between 0 and 999.

Syntax

Given below is the syntax of **getUTCMilliseconds()** method.

```
Date.getUTCMilliseconds()
```

Example

The following example demonstrates the usage of the **getUTCMilliseconds()** method in CoffeeScript. Save this code in a file with name **date_getutcmilliseconds.coffee**.

```
dt = new Date()

console.log "The UTC milliseconds in the specified date is : " +
dt.getUTCMilliseconds()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getutcmilliseconds.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0

(function() {

  var dt;


  dt = new Date();


  console.log("The UTC milliseconds in the specified date is : " +
dt.getUTCMilliseconds());


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getutcmilliseconds.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The UTC milliseconds in the specified date is : 22
```

getUTCMinutes() Method

Description

The **getUTCMinutes()** method returns the minutes in the specified date according to universal time. The value returned by getUTCMinutes() is an integer between 0 and 59.

Syntax

Given below is the syntax of **getUTCMinutes()** method.

```
Date.getUTCMinutes()
```

Example

The following example demonstrates the usage of the **getUTCMinutes()** method in CoffeeScript. Save this code in a file with name **date_getutcminutes.coffee**.

```
dt = new Date()
console.log "The UTC minutes in the specified date is : " + dt.getUTCMinutes()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getutcminutes.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date();

  console.log("The UTC minutes in the specified date is : " + dt.getUTCMinutes());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getutcminutes.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The UTC minutes in the specified date is : 34
```

getUTCMonth() Method

Description

The **getUTCMonth()** method returns the month in the specified date according to universal time. The value returned by getUTCMonth() is an integer between 0 and 11 corresponding to the month. 0 for January, 1 for February, 2 for March, and so on.

Syntax

Given below is the syntax of **getUTCMonth()** method.

```
Date.getUTCMonth()
```

Example

The following example demonstrates the usage of the **getUTCMonth()** method in CoffeeScript. Save this code in a file with name **date_getutcmonth.coffee**.

```
dt = new Date()
console.log "The UTC minutes in the specified date is : " + dt.getUTCMonth()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getutcmonth.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date();

  console.log("The UTC minutes in the specified date is : " + dt.getUTCMonth());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getutcmonth.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The UTC minutes in the specified date is : 1
```

getUTCSeconds() Method

Description

The **getUTCSeconds()** method returns the seconds in the specified date according to universal time. The value returned by getUTCSeconds() is an integer between 0 and 59.

Syntax

Given below is the syntax of **getUTCSeconds()** method.

```
Date.getUTCSeconds()
```

Example

The following example demonstrates the usage of the **getUTCSeconds()** method in CoffeeScript. Save this code in a file with name **date_getutcseconds.coffee**.

```
dt = new Date()
console.log "The UTC minutes in the specified date is : " + dt.getUTCSeconds()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getutcseconds.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date();

  console.log("The UTC minutes in the specified date is : " + dt.getUTCSeconds());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getutcseconds.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The UTC minutes in the specified date is : 12
```

getYear() Method

Description

The **getYear()** method returns the year for a specified date according to universal time.

Syntax

Given below is the syntax of **getYear()** method.

```
Date.getYear()
```

Example

The following example demonstrates the usage of the **getYear()** method in CoffeeScript. Save this code in a file with name **date_getyear.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
year = dt.getYear()
console.log year
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_getyear.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  year = dt.getYear();

  console.log(year);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_getyear.coffee
```

On executing, the CoffeeScript file produces the following output.

```
2016
```

setDate() Method

Description

The **setDate()** method sets the day of the month for a specified date according to local time.

Syntax

Given below is the syntax of **setDate()** method.

```
Date.setDate dayValue
```

Example

The following example demonstrates the usage of the **setDate()** method in CoffeeScript. Save this code in a file with name **date_setdate.coffee**.

```
dt = new Date "Aug 28, 2008 23:30:00"
dt.setDate 24
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setdate.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("Aug 28, 2008 23:30:00");

  dt.setDate(24);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setdate.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Sun Aug 24 2008 23:30:00 GMT+0530 (India Standard Time)
```

setFullYear() Method

Description

The **setFullYear()** method sets the full year for a specified date according to local time.

Syntax

Given below is the syntax of **setFullYear()** method.

```
Date.setFullYear(yearValue[, monthValue[, dayValue]])
```

Example

The following example demonstrates the usage of the **setFullYear()** method in CoffeeScript. Save this code in a file with name **date_setfullyear.coffee**.

```
dt = new Date "Aug 28, 2008 23:30:00"
dt.setFullYear 2014
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setfullyear.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("Aug 28, 2008 23:30:00");

  dt.setFullYear(2014);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setfullyear.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Thu Aug 28 2014 23:30:00 GMT+0530 (India Standard Time)
```

setHours() Method

Description

The **setHours()** method sets the hour for a specified date according to local time.

Syntax

Given below is the syntax of **setHours()** method.

```
Date.setHours(hoursValue[, minutesValue[, secondsValue[, msValue]]])
```

Parameter Detail

- **hoursValue –** An integer between 0 and 23, representing the hour.
- **minutesValue –** An integer between 0 and 59, representing the minutes.
- **secondsValue –** An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.
- **msValue –**A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the minutesValue, secondsValue, and msValue parameters, the values returned from the getUTCMinutes, getUTCSeconds, and getMilliseconds methods are used.

Example

The following example demonstrates the usage of the **setUTCHours()** method in CoffeeScript. Save this code in a file with name **date_setutchours.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setHours 15
console.log dt
```

Open the **Node.js command prompt** and compile the .coffee file as shown below.

```
> coffee -c date_sethours.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setHours(15);
  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_sethours.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Fri Feb 19 2016 15:15:00 GMT+0530 (India Standard Time)
```

setMilliseconds() Method

Description

The **setMilliseconds()** method sets the milliseconds for a specified date according to local time.

Syntax

Given below is the syntax of **setMilliseconds()** method.

```
Date.setMilliseconds(millisecondsValue)
```

Example

The following example demonstrates the usage of the **setMilliseconds()** method in CoffeeScript. Save this code in a file with name **date_setmilliseconds.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setFullYear 1010
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setmilliseconds.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");
  dt.setFullYear(1010);


  console.log(dt);


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setmilliseconds.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Mon Feb 19 1010 23:15:00 GMT+0530 (India Standard Time)
```

setMinutes() Method

Description

The **setMinutes()** method sets the minutes for a specified date according to local time.

Syntax

Given below is the syntax of **setMinutes()** method.

```
Date.setMinutes(minutesValue[, secondsValue[, msValue]])
```

Parameter Detail

- **secondsValue –** An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.

- **msValue –**A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the secondsValue and msValue parameters, the values returned from getSeconds and getMilliseconds methods are used.

Example

The following example demonstrates the usage of the **setMinutes()** method in CoffeeScript. Save this code in a file with name **date_setminutes.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setMinutes 45
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setminutes.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;
  dt = new Date("February 19, 2016 23:15:00");

  dt.setMinutes(45);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setminutes.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Fri Feb 19 2016 23:45:00 GMT+0530 (India Standard Time)
```

setMonth() Method

Description

The **setMonth()** method sets the month for a specified date according to local time.

Syntax

Given below is the syntax of **setMonth()** method.

```
Date.setMonth(monthValue[, dayValue])
```

Parameter Detail

- **monthValue –** An integer between 0 and 11 (representing the months January through December).

- **dayValue –** An integer from 1 to 31, representing the day of the month.

- **msValue –**A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the dayValue parameter, the value returned from the getDate method is used. If a parameter you specify is outside of the expected range, setMonth attempts to update the date information in the Date object accordingly. For example, if you use 15 for monthValue, the year will be incremented by 1 (year + 1), and 3 will be used for month.

Example

The following example demonstrates the usage of the **setMonth()** method in CoffeeScript. Save this code in a file with name **date_setmonth.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setMonth 5
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setmonth.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setMonth(5);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setmonth.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Sun Jun 19 2016 23:15:00 GMT+0530 (India Standard Time)
```

setSeconds() Method

Description

The **setSeconds()** method sets the seconds for a specified date according to local time

Syntax

Given below is the syntax of **setSeconds()** method.

```
Date.setSeconds(secondsValue[, msValue])
```

Parameter Detail

- **secondsValue –** An integer between 0 and 59.
- **msValue –** A number between 0 and 999, representing the milliseconds.

If you do not specify the msValue parameter, the value returned from the getMilliseconds method is used. If a parameter you specify is outside of the expected range, setSeconds attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes stored in the Date object will be incremented by 1, and 40 will be used for seconds.

Example

The following example demonstrates the usage of the **setSeconds()** method in CoffeeScript. Save this code in a file with name **date_setseconds.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setSeconds 25
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setseconds.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setSeconds(25);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setseconds.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Fri Feb 19 2016 23:15:25 GMT+0530 (India Standard Time)
```

setTime() Method

Description

The **setTime()** method sets the Date object to the time represented by a number of milliseconds since January 1, 1970, 00:00:00 UTC.

Syntax

Given below is the syntax of **setTime()** method.

```
Date.setTime(timeValue)
```

Parameter Detail

- **timeValue –** An integer representing the number of milliseconds since 1 January 1970, 00:00:00 UTC.

Example

The following example demonstrates the usage of the **setTime()** method in CoffeeScript. Save this code in a file with name **date_settime.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setTime 5000000
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_settime.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");
  dt.setTime(5000000);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_settime.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Thu Jan 01 1970 06:53:20 GMT+0530 (India Standard Time)
```

setUTCDate() Method

Description

The **setUTCDate()** method sets the day of the month for a specified date according to universal time.

Syntax

Given below is the syntax of **setUTCDate()** method.

```
Date.setUTCDate(dayValue)
```

Parameter Detail

- **dayValue –** An integer from 1 to 31, representing the day of the month.

Example

The following example demonstrates the usage of the **setUTCDate()** method in CoffeeScript. Save this code in a file with name **date_setutcdate.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setUTCDate 20
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setutcdate.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");
  dt.setUTCDate(20);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setutcdate.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Sat Feb 20 2016 23:15:00 GMT+0530 (India Standard Time)
```

setUTCFullYear() Method

Description

The **setUTCFullYear()** method sets the full year for a specified date according to universal time.

Syntax

Given below is the syntax of **setUTCFullYear()** method.

```
Date.setUTCFullYear(yearValue[, monthValue[, dayValue]])
```

Parameter Detail

- **yearValue –** An integer specifying the numeric value of the year, for example, 2008.
- **monthValue –** An integer between 0 and 11 representing the months January through December.
- **dayValue –**An integer between 1 and 31 representing the day of the month. If you specify the dayValue parameter, you must also specify the monthValue.

If you do not specify the monthValue and dayValue parameters, the values returned from the getMonth and getDate methods are used. If a parameter you specify is outside of the expected range, setUTCFullYear attempts to update the other parameters and the date information in the Date object accordingly. For example, if you specify 15 for monthValue, the year is incremented by 1 (year + 1), and 3 is used for the month.

Example

The following example demonstrates the usage of the **setUTCFullYear()** method in CoffeeScript. Save this code in a file with name **date_setutcfullYear.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
```

```
dt.setUTCFullYear 20
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setutcfullYear.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setUTCFullYear(20);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setutcfullYear.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Wed Feb 19 20 23:15:00 GMT+0530 (India Standard Time)
```

setUTCHours() Method

Description

The **setUTCHours()** method sets the hour for a specified date according to local time.

Syntax

Given below is the syntax of **setUTCHours()** method.

```
Date.setUTCHours(hoursValue[, minutesValue[, secondsValue[, msValue]]])
```

Parameter Detail

- **hoursValue –** An integer between 0 and 23, representing the hour.
- **minutesValue –** An integer between 0 and 59, representing the minutes.

- **secondsValue –** An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.
- **msValue –**A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the minutesValue, secondsValue, and msValue parameters, the values returned from the getUTCMinutes, getUTCSeconds, and getUTCMilliseconds methods are used.

If a parameter you specify is outside the expected range, setUTCHours attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes will be incremented by 1 (min + 1), and 40 will be used for seconds.

Example

The following example demonstrates the usage of the **setUTCHours()** method in CoffeeScript. Save this code in a file with name **date_setutchours.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setUTCHours 15
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setutchours.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setUTCHours(15);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setutchours.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Fri Feb 19 2016 21:15:00 GMT+0530 (India Standard Time)
```

setUTCMilliseconds() Method

Description

The **setUTCMilliseconds()** method sets the milliseconds for a specified date according to universal time.

Syntax

Given below is the syntax of **setMinutes()** method.

```
Date.setUTCMilliseconds(millisecondsValue)
```

Parameter Detail

- **millisecondsValue –** A number between 0 and 999, representing the milliseconds.

If a parameter you specify is outside the expected range, setUTCMilliseconds attempts to update the date information in the Date object accordingly. For example, if you use 1100 for millisecondsValue, the seconds stored in the Date object will be incremented by 1, and 100 will be used for milliseconds.

Example

The following example demonstrates the usage of the **setUTCMilliseconds()** method in CoffeeScript. Save this code in a file with name **date_setutcmilliseconds.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setUTCMilliseconds 1100
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setutcmilliseconds.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setUTCMilliseconds(1100);

  console.log(dt);


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setutcmilliseconds.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Fri Feb 19 2016 23:15:01 GMT+0530 (India Standard Time)
```

setUTCMinutes() Method

Description

The **setUTCMinutes()** method sets the minutes for a specified date according to universal time.

Syntax

Given below is the syntax of **setUTCMinutes()** method.

```
Date.setUTCMinutes(minutesValue[, secondsValue[, msValue]])
```

Parameter Detail

- **minutesValue –** An integer between 0 and 59, representing the minutes.
- **secondsValue –** An integer between 0 and 59, representing the seconds. If you specify the secondsValue parameter, you must also specify the minutesValue.
- **msValue –** A number between 0 and 999, representing the milliseconds. If you specify the msValue parameter, you must also specify the minutesValue and secondsValue.

If you do not specify the secondsValue and msValue parameters, the values returned from getUTCSeconds and getUTCMilliseconds methods are used.

If a parameter you specify is outside of the expected range, setUTCMinutes attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes (minutesValue) will be incremented by 1 (minutesValue + 1), and 40 will be used for seconds.

Example

The following example demonstrates the usage of the **setUTCMinutes()** method in CoffeeScript. Save this code in a file with name **date_setutcminutes.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"

dt.setUTCMinutes 65

console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setutcminutes.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setUTCMilliseconds(1100);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setutcminutes.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Fri Feb 19 2016 23:35:00 GMT+0530 (India Standard Time)
```

setUTCMonth() Method

Description

The **setUTCMonth()** method sets the month for a specified date according to universal time.

Syntax

Given below is the syntax of **setUTCMonth()** method.

```
Date.setUTCMonth( monthvalue )
```

Parameter Detail

- **monthValue –** An integer between 0 and 11, representing the month.

Example

The following example demonstrates the usage of the **setUTCMonth()** method in CoffeeScript. Save this code in a file with name **date_setutcmonth.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setUTCMonth 2
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setutcmonth.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setUTCMonth(2);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setutcmonth.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Sat Mar 19 2016 23:15:00 GMT+0530 (India Standard Time)
```

setUTCSeconds() Method

Description

The **setUTCSeconds()** method sets the seconds for a specified date according to universal time.

Syntax

Given below is the syntax of **setUTCSeconds()** method.

```
Date.setUTCSeconds(secondsValue[, msValue])
```

Parameter Detail

- **secondsValue –** An integer between 0 and 59, representing the seconds.
- **msValue –** A number between 0 and 999, representing the milliseconds.

If you do not specify the msValue parameter, the value returned from the getUTCMilliseconds methods is used.

If a parameter you specify is outside the expected range, setUTCSeconds attempts to update the date information in the Date object accordingly. For example, if you use 100 for secondsValue, the minutes stored in the Date object will be incremented by 1, and 40 will be used for seconds.

Example

The following example demonstrates the usage of the **setUTCSeconds()** method in CoffeeScript. Save this code in a file with name **date_setutcseconds.coffee.**

```
dt = new Date "February 19, 2016 23:15:00"
dt.setUTCSeconds 2
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setutcseconds.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setUTCSeconds(2);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setutcseconds.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Fri Feb 19 2016 23:15:02 GMT+0530 (India Standard Time)
```

setYear() Method

Description

The **setYear()** method sets the year for a specified date according to universal time.

Syntax

Given below is the syntax of **setYear()** method.

```
Date.setYear(yearValue)
```

Parameter Detail

- **yearValue –** An integer value.

Example

The following example demonstrates the usage of the **setYear()** method in CoffeeScript. Save this code in a file with name **date_setyear.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.setYear 2000
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_setyear.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.setYear(2000);

  console.log(dt);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_setyear.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Sat Feb 19 2000 23:15:00 GMT+0530 (India Standard Time)
```

toDateString() Method

Description

The **toDateString()** method returns the date portion of a Date object in human readable form.

Syntax

Given below is the syntax of **toDateString()** method.

```
Date.toDateString()
```

Return Value

Returns the date portion of a Date object in human readable form.

Example

The following example demonstrates the usage of the **toDateString()** method in CoffeeScript. Save this code in a file with name **date_todatestring.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.toDateString()
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_todatestring.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date("February 19, 2016 23:15:00");

  dt.toDateString();

  console.log(dt);
}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_todatestring.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Fri Feb 19 2016 23:15:00 GMT+0530 (India Standard Time)
```

toLocaleDateString() Method

Description

The **toLocaleDateString()** method converts a date to a string, returning the "date" portion using the operating system's locale's conventions.

Syntax

Given below is the syntax of **toLocaleDateString()** method.

```
Date.toLocaleDateString()
```

Return Value

Returns the "date" portion using the operating system's locale's conventions.

Example

The following example demonstrates the usage of the **toLocaleDateString ()** method in CoffeeScript. Save this code in a file with name **date_tolocaledatestring.coffee**.

```
dt = new Date "February 19, 2016 23:15:00"
dt.toLocaleDateString()
console.log dt
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_tolocaledatestring.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;


  dt = new Date("February 19, 2016 23:15:00");
  dt.toLocaleDateString();


  console.log(dt);


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_tolocaledatestring.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Fri Feb 19 2016 23:15:00 GMT+0530 (India Standard Time)
```

toLocaleString() Method

Description

The **toLocaleString()** method converts a date to a string, using the operating system's local conventions.

The toLocaleString method relies on the underlying operating system in formatting dates. It converts the date to a string using the formatting convention of the operating system where the script is running. For example, in the United States, the month appears before the date (04/15/98), whereas in Germany the date appears before the month (15.04.98).

Syntax

Given below is the syntax of **toLocaleString()** method.

```
Date.toLocaleString()
```

Return Value

Returns the formatted date in a string format.

Example

The following example demonstrates the usage of the **toLocaleString()** method in CoffeeScript. Save this code in a file with name **date_tolocalestring.coffee**.

```
dt = new Date(1993, 6, 28, 14, 39, 7);
console.log dt.toLocaleString()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_tolocalestring.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date(1993, 6, 28, 14, 39, 7);

  console.log(dt.toLocaleString());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_tolocalestring.coffee
```

On executing, the CoffeeScript file produces the following output.

```
7/28/1993, 2:39:07 PM
```

toLocaleTimeString() Method

Description

The **toLocaleTimeString()** method converts a date to a string, returning the "date" portion using the current locale's conventions.

The toLocaleTimeString method relies on the underlying operating system in formatting dates. It converts the date to a string using the formatting convention of the operating system where the script is running. For example, in the United States, the month appears before the date (04/15/98), whereas in Germany, the date appears before the month (15.04.98).

Syntax

Given below is the syntax of **toLocaleTimeString()** method.

```
Date.toLocaleTimeString()
```

Return Value

Returns the formatted date in a string format.

Example

The following example demonstrates the usage of the **toLocaleTimeString()** method in CoffeeScript. Save this code in a file with name **date_tolocaletimestring.coffee**.

```
dt = new Date(1993, 6, 28, 14, 39, 7);
```

```
console.log dt.toLocaleTimeString()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_tolocaletimestring.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date(1993, 6, 28, 14, 39, 7);

  console.log(dt.toLocaleTimeString());

}).call(this);
```

Now, open the **Node.js command prompt** again, and run the CoffeeScript file as shown below.

```
> coffee date_tolocaletimestring.coffee
```

On executing, the CoffeeScript file produces the following output.

```
2:39:07 PM
```

toTimeString() Method

Description

The **toTimeStrin()** method returns the time portion of a Date object in human readable form.

Syntax

Given below is the syntax of **toTimeString()** method.

```
Date.toTimeString()
```

Return Value

Returns the time portion of a Date object in human readable form.

Example

The following example demonstrates the usage of the **toTimeString()** method in CoffeeScript. Save this code in a file with name **date_totimestring .coffee**.

```
dt = new Date(1993, 6, 28, 14, 39, 7);
console.log dt.toTimeString()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_totimestring .coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date(1993, 6, 28, 14, 39, 7);

  console.log(dt.toTimeString());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_totimestring.coffee
```

On executing, the CoffeeScript file produces the following output.

```
14:39:07 GMT+0530 (India Standard Time)
```

toTimeString Method

Description

The **toUTCString()** Method returns the time portion of a Date object in human readable form.

Syntax

Given below is the syntax of **toUTCString()** method.

```
Date.toUTCString()
```

Return Value

Returns converted date to a string, using the universal time convention.

Example

The following example demonstrates the usage of the **toUTCString()** method in CoffeeScript. Save this code in a file with name **date_toutcstring.coffee**.

```
dt = new Date(1993, 6, 28, 14, 39, 7);
console.log dt.toUTCString()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c date_toutcstring.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var dt;

  dt = new Date(1993, 6, 28, 14, 39, 7);

  console.log(dt.toUTCString());

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee date_toutcstring.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Wed, 28 Jul 1993 09:09:07 GMT
```

18. CoffeeScript – Math

The **Math** object of JavaScript provides you properties and methods for mathematical constants and functions. Unlike other global objects, **Math** is not a constructor. All the properties and methods of **Math** are static and can be called by using Math as an object without creating it.

Thus, you refer to the constant **pi** as **Math.PI** and you call the sine function as **Math.sin(x)**, where **x** is the method's argument. We can use the JavaScript's Math object in our CoffeeScript code to perform math operations.

Mathematical constants

If we want to use any common mathematical constants like **pi** or **e,** we can use them using the JavaScript's **Math** object.

Following is the list of the Math constants provided by the Math object of JavaScript

Property	Description
E	Euler's constant and the base of natural logarithms, approximately 2.718.
LN2	Natural logarithm of 2, approximately 0.693.
LN10	Natural logarithm of 10, approximately 2.302.
LOG2E	Base 2 logarithm of E, approximately 1.442.
LOG10E	Base 10 logarithm of E, approximately 0.434.
PI	Ratio of the circumference of a circle to its diameter, approximately 3.14159.
SQRT1_2	Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.
SQRT2	Square root of 2, approximately 1.414.

Example

The following example demonstrates the usage of the mathematical constants provided by JavaScript in CoffeeScript. Save this code in a file with name **math_example.coffee**

```
e_value = Math.E
console.log "The value of the constant E is: " + e_value


LN2_value = Math.LN2
console.log "The value of the constant LN2 is: " + LN2_value


LN10_value = Math.LN10
console.log "The value of the constant LN10 is: " + LN10_value


LOG2E_value = Math.LOG2E
```

```
console.log "The value of the constant LOG2E is: " + LOG2E_value

LOG10E_value = Math.LOG10E
console.log "The value of the constant LOG10E is: " + LOG10E_value

PI_value = Math.PI
console.log "The value of the constant PI is: " + PI_value

SQRT1_2_value = Math.SQRT1_2
console.log "The value of the constant SQRT1_2 is: " + SQRT1_2_value

SQRT2_value = Math.SQRT2
console.log "The value of the constant SQRT2 is: " + SQRT2_value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var LN10_value, LN2_value, LOG10E_value, LOG2E_value, PI_value, SQRT1_2_value,
SQRT2_value, e_value;

  e_value = Math.E;

  console.log("The value of the constant E is: " + e_value);

  LN2_value = Math.LN2;

  console.log("The value of the constant LN2 is: " + LN2_value);

  LN10_value = Math.LN10;

  console.log("The value of the constant LN10 is: " + LN10_value);

  LOG2E_value = Math.LOG2E;
  console.log("The value of the constant LOG2E is: " + LOG2E_value);

  LOG10E_value = Math.LOG10E;

  console.log("The value of the constant LOG10E is: " + LOG10E_value);

  PI_value = Math.PI;

  console.log("The value of the constant PI is: " + PI_value);

  SQRT1_2_value = Math.SQRT1_2;

  console.log("The value of the constant SQRT1_2 is: " + SQRT1_2_value);

  SQRT2_value = Math.SQRT2;

  console.log("The value of the constant SQRT2 is: " + SQRT2_value);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee math_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The value of the constant E is: 2.718281828459045
The value of the constant LN2 is: 0.6931471805599453
The value of the constant LN10 is: 2.302585092994046
The value of the constant LOG2E is: 1.4426950408889634
The value of the constant LOG10E is: 0.4342944819032518
The value of the constant PI is: 3.141592653589793
The value of the constant SQRT1_2 is: 0.7071067811865476
The value of the constant SQRT2 is: 1.4142135623730951
```

Math Methods

In addition to properties, the Math object also provides methods. Following is the list of methods of the **Math** object of JavaScript. Click on the name of these methods to get an example demonstrating their usage in CoffeeScript

Method	Description
abs()	Returns the absolute value of a number.
acos()	Returns the arccosine (in radians) of a number.
asin()	Returns the arcsine (in radians) of a number.
atan()	Returns the arctangent (in radians) of a number.
atan2()	Returns the arctangent of the quotient of its arguments.
ceil()	Returns the smallest integer greater than or equal to a number.
cos()	Returns the cosine of a number.
exp()	Returns E^N, where N is the argument, and E is Euler's constant, the base of the natural logarithm.
floor()	Returns the largest integer less than or equal to a number.
log()	Returns the natural logarithm (base E) of a number.
max()	Returns the largest of zero or more numbers.
min()	Returns the smallest of zero or more numbers.
pow()	Returns base to the exponent power, that is, base exponent.
random()	Returns a pseudo-random number between 0 and 1.
round()	Returns the value of a number rounded to the nearest integer.
sin()	Returns the sine of a number.
sqrt()	Returns the square root of a number.
tan()	Returns the tangent of a number.

abs() Method

Description

This method accepts an integer and returns the absolute value of the given integer.

Syntax

Following is the syntax of this method.

```
Math.abs( x )
```

Example

The following example demonstrates the usage of the **abs()** method in CoffeeScript. Save this code in a file with name **math_abs.coffee**.

```
value = Math.abs(-1);
```

```
console.log "The absolute value of -1 is : " + value

value = Math.abs(null);
console.log "The absolute value of null is : " + value

value = Math.abs(20);
console.log "The absolute value of 20 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_abs.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.abs(-1);

  console.log("The absolute value of -1 is : " + value);

  value = Math.abs(null);

  console.log("The absolute value of null is : " + value);

  value = Math.abs(20);

  console.log("The absolute value of 20 is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_abs.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The absolute value of -1 is : 1
The absolute value of null is : 0
The absolute value of 20 is : 20
```

acos() Method

Description

The **acos()** method accepts an number and returns its arc-cosine in radians. It returns a numeric value between 0 and pi radians for x between -1 and 1. If the value of number is outside this range, it returns NaN.

Syntax

Given below is the syntax of **acos()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.acos( x )
```

Example

The following example demonstrates the usage of the **acos()** method in CoffeeScript. Save this code in a file with name **math_acos.coffee**.

```
value = Math.acos -1
console.log "The arc cosine value of -1 is : " + value


value = Math.acos null
console.log "The arc cosine value of null is : " + value


value = Math.acos 20
console.log "The arc cosine value of 20 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_acos.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;


  value = Math.acos(-1);


  console.log("The arc cosine value of -1 is : " + value);


  value = Math.acos(null);

  console.log("The arc cosine value of null is : " + value);

  value = Math.acos(20);

  console.log("The arc cosine value of 20 is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_acos.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The arc cosine value of -1 is : 3.141592653589793
The arc console value of null is : 1.5707963267948966
The arc cosine value of 20 is : NaN
```

asin() Method

Description

The **asin()** method accepts an integer value and returns its arc-sine in radians. The asin method returns a numerical value between -pi/2 and pi/2 radians for x between -1 and 1. If the value of number is outside this range, it returns NaN.

Syntax

Given below is the syntax of **asin()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.asin( x )
```

Example

The following example demonstrates the usage of the **asin()** method in CoffeeScript. Save this code in a file with name **math_asin.coffee.**

```
value = Math.asin -1
console.log "The arc sine value of -1 is : " + value

value = Math.asin null
console.log "The arc sine value of null is : " + value

value = Math.asin 20
```

```
console.log "The arc sine value of 20 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_asin.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.asin(-1);

  console.log("The arc sine value of -1 is : " + value);

  value = Math.asin(null);

  console.log("The arc sine value of null is : " + value);

  value = Math.asin(20);

  console.log("The arc sine value of 20 is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_asin.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The arc sine value of -1 is : -1.5707963267948966
The arc sine value of null is : 0
The arc sine value of 20 is : NaN
```

atan() Method

Description

The **atan()** method accepts a number and returns its arctangent value in radians. This method returns a numeric value between -pi/2 and pi/2 radians.

Syntax

Given below is the syntax of **atan()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.atan( x )
```

Example

The following example demonstrates the usage of the **atan()** method in CoffeeScript. Save this code in a file with name **math_atan.coffee**.

```
value = Math.atan -1
console.log "The arc tangent value of -1 is : " + value


value = Math.atan null
console.log "The arc tangent value of null is : " + value


value = Math.atan 20
console.log "The arc tangent value of 20 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_atan.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;


  value = Math.atan(-1);


  console.log("The arc tangent value of -1 is : " + value);


  value = Math.atan(null);


  console.log("The arc tangent value of null is : " + value);


  value = Math.atan(20);


  console.log("The arc tangent value of 20 is : " + value);
}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_atan.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The arc tangent value of -1 is : -0.7853981633974483

The arc tangent value of null is : 0

The arc tangent value of 20 is : 1.5208379310729538
```

atan2() Method

Description

The **atan2()** method accepts two numbers and returns the arctangent value in radians. This method returns a numeric value between -pi and pi representing the angle theta of an (x, y) point.

Syntax

Given below is the syntax of **atan2()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.atan( x )
```

Example

The following example demonstrates the usage of the **atan2()** method in CoffeeScript. Save this code in a file with name **math_atan2.coffee**.

```
value = Math.atan2 90,15

console.log "The arc tangent value of 90,15 is : " + value


value = Math.atan2 15,90

console.log "The arc tangent value of 15,90 is : " + value


value = Math.atan2 0,-0

console.log "The arc tangent value of 0,-0 is : " + value


value = Math.atan2 +Infinity, -Infinity

console.log "The arc tangent value of +Infinity, -Infinity is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_atan2.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.atan(-1);

  console.log("The arc tangent value of -1 is : " + value);

  value = Math.atan(null);

  console.log("The arc tangent value of null is : " + value);

  value = Math.atan(20);

  console.log("The arc tangent value of 20 is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_atan2.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The arc tangent value of 90,15 is : 1.4056476493802699
The arc tangent value of 15,90 is : 0.16514867741462683
The arc tangent value of 0,-0 is : 3.141592653589793
The arc tangent value of +Infinity, -Infinity is : 2.356194490192345
```

ceil() Method

Description

The **ceil()** method accepts a number and returns its ceil value.

Syntax

Given below is the syntax of **ceil()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.ceil( x )
```

Example

The following example demonstrates the usage of the **ceil()** method in CoffeeScript. Save this code in a file with name **math_ceil.coffee**.

```
value = Math.ceil 90.15
console.log "The ceil value of 90.15 is : " + value


value = Math.ceil 15.90
console.log "The ceil value of 15.90 is : " + value


value = Math.ceil -90.15
console.log "The ceil value of -90.15 is : " + value


value = Math.ceil -15.90
console.log "The ceil value of -15.90 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_ceil.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;


  value = Math.ceil(90.15);


  console.log("The ceil value of 90.15 is : " + value);


  value = Math.ceil(15.90);


  console.log("The ceil value of 15.90 is : " + value);


  value = Math.ceil(-90.15);

  console.log("The ceil value of -90.15 is : " + value);


  value = Math.ceil(-15.90);


  console.log("The ceil value of -15.90 is : " + value);


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_ceil.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The ceil value of 90.15 is : 91
The ceil value of 15.90 is : 16
The ceil value of -90.15 is : -90
The ceil value of -15.90 is : -15
```

cos() Method

Description

The **cos()** method accepts a number and returns its cosine value which is between -1 and 1.

Syntax

Given below is the syntax of **cos()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.cos( x )
```

Example

The following example demonstrates the usage of the **cos()** method in CoffeeScript. Save this code in a file with name **math_cos.coffee**.

```
value = Math.cos 90
console.log "The cosine value of 90 is : " + value


value = Math.cos -1
console.log "The cosine value of -1 is : " + value


value = Math.cos 2*Math.PI
```

```
console.log "The cosine value of 2*Math.PI is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_cos.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.cos(90);

  console.log("The cosine value of 90 is : " + value);

  value = Math.cos(-1);

  console.log("The cosine value of -1 is : " + value);

  value = Math.cos(2 * Math.PI);

  console.log("The cosine value of 2*Math.PI is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_cos.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The cosine value of -1 is : -0.4480736161291702
The cosine value of null is : 0.5403023058681398
The cosine value of 20 is : 1
```

exp() Method

Description

The **exp()** method accepts a number and returns its E^x value. Where x is the argument, and E is Euler's constant the base of the natural logarithms

Syntax

Given below is the syntax of **exp()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.exp ( x )
```

Example

The following example demonstrates the usage of the **exp()** method in CoffeeScript. Save this code in a file with name **math_exp.coffee**.

```
value = Math.exp 1
console.log "The exponential value of 1 is : " + value

value = Math.exp 52
console.log "The exponential value of 52 is : " + value

value = Math.exp 0.78
console.log "The absolute value of 0.78 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_exp.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;



  value = Math.exp(1);

  console.log("The exponential value of 1 is : " + value);

  value = Math.exp(52);

  console.log("The exponential value of 52 is : " + value);

  value = Math.exp(0.78);

  console.log("The absolute value of 0.78 is : " + value);
}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_exp.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The exponential value of 1 is : 2.718281828459045
The exponential value of 52 is : 3.831008000716576e+22
The absolute value of 0.78 is : 2.1814722654982015
```

floor() Method

Description

The **floor()** method accepts a number and returns its floor value.

Syntax

Given below is the syntax of **floor()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.floor ( x )
```

Example

The following example demonstrates the usage of the **floor()** method in CoffeeScript. Save this code in a file with name **math_floor.coffee**.

```
value = Math.floor 10.3
console.log "The floor value of 10.3 is : " + value

value = Math.floor 30.9
console.log "The floor value of 30.9 is : " + value

value = Math.floor -2.2
console.log "The floor value of -2.2 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_floor.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.floor(10.3);

  console.log("The floor value of 10.3 is : " + value);

  value = Math.floor(30.9);

  console.log("The floor value of 30.9 is : " + value);

  value = Math.floor(-2.2);

  console.log("The floor value of -2.2 is : " + value);


}).call(this);
```

Now, open the **Node.js command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_floor.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The floor value of 10.3 is : 10
The floor value of 30.9 is : 30
The floor value of -2.2 is : -3
```

log() Method

Description

The **log()** method accepts a number and returns its the natural logarithm (base E) of a number. If the value of number is negative, the return value is always NaN.

Syntax

Given below is the syntax of **log()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.log ( x )
```

Example

The following example demonstrates the usage of the **log()** method in CoffeeScript. Save this code in a file with name **math_log.coffee**.

```
value = Math.log 10
console.log "The log value of 10 is : " + value

value = Math.log 0
console.log "The log value of 0 is : " + value

value = Math.log 100
console.log "The log value of 100 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_log.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.log(10);

  console.log("The log value of 10 is : " + value);

  value = Math.log(0);

  console.log("The log value of 0 is : " + value);

  value = Math.log(100);

  console.log("The log value of 100 is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_log.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The log value of 10 is : 2.302585092994046
The log value of 0 is : -Infinity
The log value of 100 is : 4.605170185988092
```

max() Method

Description

The **max()** method accepts a set of numbers and returns the maximum value among the given numbers. On calling this method without passing arguments it returns -Infinity.

Syntax

Given below is the syntax of **max()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.max ( x )
```

Example

The following example demonstrates the usage of the **max()** method in CoffeeScript. Save this code in a file with name **math_max.coffee**.

```
value = Math.max 10, 20, -1, 100
console.log "The max value among (10, 20, -1, 100) is : " + value

value = Math.max -1, -3, -40
console.log "The max value among (-1, -3, -40) is : " + value

value = Math.max 0, -1
console.log "The max value among (0, -1) is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_max.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.max(10, 20, -1, 100);
  console.log("The max value among (10, 20, -1, 100) is : " + value);

  value = Math.max(-1, -3, -40);

  console.log("The max value among (-1, -3, -40) is : " + value);

  value = Math.max(0, -1);

  console.log("The max value among (0, -1) is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_max.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The max value among (10, 20, -1, 100) is : 100
The max value among (-1, -3, -40) is : -1
The max value among (0, -1) is : 0
```

min() Method

Description

The **min()** method accepts a set of numbers and returns the minimum value among the given numbers. On calling this method without passing arguments it returns +Infinity.

Syntax

Given below is the syntax of **min()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.min ( x )
```

Example

The following example demonstrates the usage of the **min()** method in CoffeeScript. Save this code in a file with name **math_min.coffee**.

```
value = Math.min 10, 20, -1, 100
console.log "The min value among (10, 20, -1, 100) is : " + value
```

```

value = Math.min -1, -3, -40
console.log "The min value among (-1, -3, -40) is : " + value

value = Math.min 0, -1
console.log "The min value among (0, -1) is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_min.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.min(10, 20, -1, 100);

  console.log("The min value among (10, 20, -1, 100) is : " + value);

  value = Math.min(-1, -3, -40);

  console.log("The min value among (-1, -3, -40) is : " + value);

  value = Math.min(0, -1);

  console.log("The min value among (0, -1) is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_max.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The min value among (10, 20, -1, 100) is : -1
The min value among (-1, -3, -40) is : -40
The min value among (0, -1) is : -1
```

pow() Method

Description

The **pow()** method accepts two numbers, a base and an exponent and this method returns the base to the exponent power, that is, base^{exponent}.

Syntax

Given below is the syntax of **pow()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.pow ( x )
```

Example

The following example demonstrates the usage of the **pow()** method in CoffeeScript. Save this code in a file with name **math_pow.coffee.**

```
value = Math.pow 7, 2
console.log "The value of pow(7,2) is : " + value

value = Math.pow 3,9
console.log "The value of pow(3,9) is : " + value

value = Math.pow 12,8
console.log "The value of pow(12,8) is : " + value

value = Math.pow 125,0
console.log "The value of pow(125,0) is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_pow.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.pow(7, 2);

  console.log("The value of pow(7,2) is : " + value);
  value = Math.pow(3, 9);

  console.log("The value of pow(3,9) is : " + value);

  value = Math.pow(12, 8);

  console.log("The value of pow(12,8) is : " + value);

  value = Math.pow(125, 0);

  console.log("The value of pow(125,0) is : " + value);


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_pow.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The value of pow(7,2) is : 49
The value of pow(3,9) is : 19683
The value of pow(12,8) is : 429981696
The value of pow(125,0) is : 1
```

random() Method

Description

The **random()** method returns a random number between 0 (inclusive) and 1 (exclusive).

Syntax

Given below is the syntax of **random()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.random ( x )
```

Example

The following example demonstrates the usage of the **random()** method in CoffeeScript. Save this code in a file with name **math_random.coffee**.

```
value = Math.random()
console.log "The first number is : " + value


value = Math.random()
console.log "The second number is : " + value


value = Math.random()
console.log "The third number is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_random.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.random();

  console.log("The first number is : " + value);

  value = Math.random();

  console.log("The second number is : " + value);

  value = Math.random();

  console.log("The third number is : " + value);


}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_tan.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The first number is : 0.44820353598333895

The second number is : 0.10985115729272366

The third number is : 0.6831563576124609
```

round() Method

Description

The **round()** method accepts a number and returns the value of a number rounded to the nearest integer

Syntax

Given below is the syntax of **round()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.round ( x )
```

Example

The following example demonstrates the usage of the **round()** method in CoffeeScript. Save this code in a file with name **math_round.coffee**.

```
value = Math.round 0.5

console.log "The nearest integer to 0.5 is : " + value


value = Math.round 20.7

console.log "The nearest integer to 20.7 is : " + value


value = Math.round -20.3

console.log "The nearest integer to -20.3 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_round.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0

(function() {

  var value;
  value = Math.round(0.5);

  console.log("The nearest integer to 0.5 is : " + value);

  value = Math.round(20.7);

  console.log("The nearest integer to 20.7 is : " + value);

  value = Math.round(-20.3);

  console.log("The nearest integer to -20.3 is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_round.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The nearest integer to 0.5 is : 1
The nearest integer to 20.7 is : 21
The nearest integer to -20.3 is : -20
```

sin() Method

Description

The **sin()** method accepts a number and returns its sine value which is between -1 and 1.

Syntax

Given below is the syntax of **sin()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.sin( x )
```

Example

The following example demonstrates the usage of the **sin()** method in CoffeeScript. Save this code in a file with name **sin.coffee**.

```
value = Math.sin 90
console.log "The sine value of 90 is : " + value

value = Math.sin 0.5
console.log "The sine value of 0.5 is : " + value

value = Math.sin 2*Math.PI/2
console.log "The sine value of 2*Math.PI/2 is : " + value
```

Open the **Node.js command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_sin.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.sin(90);

  console.log("The sine value of 90 is : " + value);

  value = Math.sin(0.5);

  console.log("The sine value of 0.5 is : " + value);

  value = Math.sin(2 * Math.PI / 2);

  console.log("The sine value of 2*Math.PI/2 is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_sin.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The sine value of 90 is : 0.8939966636005579
The sine value of 0.5 is : 0.479425538604203
The sine value of 2*Math.PI/2 is : 1.2246467991473532e-16
```

sqrt() Method

Description

The **sqrt()** method accepts a number and returns its square root value. If the value of a number is negative, sqrt returns NaN.

Syntax

Given below is the syntax of **sqrt()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.sqrt ( x )
```

Example

The following example demonstrates the usage of the **sqrt()** method in CoffeeScript. Save this code in a file with name **math_sqrt.coffee**.

```
value = Math.sqrt 0.5
console.log "The square root of 0.5 is : " + value


value = Math.sqrt 81
console.log "The square root of 81 is : " + value



value = Math.sqrt 13
console.log "The square root of 13 is : " + value


value = Math.sqrt -4
console.log "The square root of -4 is : " + value
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_sqrt.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;
  value = Math.sqrt(0.5);

  console.log("The square root of 0.5 is : " + value);

  value = Math.sqrt(81);

  console.log("The square root of 81 is : " + value);

  value = Math.sqrt(13);

  console.log("The square root of 13 is : " + value);

  value = Math.sqrt(-4);

  console.log("The square root of -4 is : " + value);

}).call(this);
```

Now, open the **command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_sqrt.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The square root of 0.5 is : 0.7071067811865476
The square root of 81 is : 9
The square root of 13 is : 3.605551275463989
The square root of -4 is : NaN
```

tan() Method

Description

The **tan()** method accepts a number and returns its tangent value.

Syntax

Given below is the syntax of **tan()** method of JavaScript. We can use the same method in the CoffeeScript code.

```
Math.tan ( x )
```

Example

The following example demonstrates the usage of the **tan()** method in CoffeeScript. Save this code in a file with name **math_tan.coffee**.

```
value = Math.tan -30
console.log "The tangent value of -30 is : " + value

value = Math.tan 90
console.log "The tangent value of 90 is : " + value

value = Math.tan(2*Math.PI/180)
console.log "The tangent value of 2*Math.PI/180 is : " + value
```

Open the **Node.js command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c math_tan.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var value;

  value = Math.tan(-30);

  console.log("The tangent value of -30 is : " + value);

  value = Math.tan(90);

  console.log("The tangent value of 90 is : " + value);

  value = Math.tan(2 * Math.PI / 180);

  console.log("The tangent value of 2*Math.PI/180 is : " + value);

}).call(this);
```

Now, open the **Node.js command prompt** again, and run the CoffeeScript file as shown below.

```
c:\> coffee math_tan.coffee
```

On executing, the CoffeeScript file produces the following output.

```
The tangent value of -30 is : 6.405331196646276
The tangent value of 90 is : -1.995200412208242
The tangent value of 2*Math.PI/180 is : 0.03492076949174773
```

19. CoffeeScript – Exception Handling

An exception (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs, the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore these exceptions are to be handled.

An exception can occur for many different reasons. Here are some scenarios where an exception occurs.

- A user has entered invalid data.
- A file that needs to be opened cannot be found.

Exceptions in CoffeeScript

CoffeeScripts supports exception/error handling using the **try catch and finally** blocks. The functionalities of these blocks are same as in JavaScript, the **try** block holds the exceptional statements, the **catch** block has the action to be performed when an exception occurs, and the **finally** block is used to execute the statements unconditionally.

Following are the syntaxes of **try catch** and **finally** blocks in CoffeeScript.

```
try
  // Code to run

catch ( e )
  // Code to run if an exception occurs

finally
  // Code that is always executed regardless of
  // an exception occurring
```

The **try** block must be followed by either exactly one **catch** block or one **finally** block (or one of both). When an exception occurs in the **try** block, the exception is placed in **e** and the **catch** block is executed. The optional **finally** block executes unconditionally after try/catch.

Example

The following example demonstrates the Exception handling using try and catch blocks in CoffeeScript. In here, we are trying to use an undefined symbol in CoffeeScript operation and we handled the error occurred using the **try** and **catch** blocks. Save this code in a file with the name **Exception_handling.coffee**

```
try
  x = y+20
  console.log "The value of x is :" +x
catch e
  console.log "exception/error occurred"
  console.log "The STACKTRACE for the exception/error occurred is ::"
  console.log e.stack
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c Exception_handling.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var e, error, x;

  try {
    x = y + 20;
    console.log("The value of x is :" + x);
  } catch (error) {
    e = error;
    console.log("exception/error occurred");
    console.log("The STACKTRACE for the exception/error occurred is ::");
    console.log(e.stack);
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee Exception_handling.coffee
```

On executing, the CoffeeScript file produces the following output.

```
exception/error occurred
The STACKTRACE for the exception/error occurred is ::
ReferenceError: y is not defined
  at Object.<anonymous>
(C:\Examples\strings_exceptions\Exception_handling.coffee:3:7)
```

```
  at Object.<anonymous>
(C:\Examples\strings_exceptions\Exception_handling.coffee:2:1)

  at Module._compile (module.js:413:34)

  at Object.exports.run
(C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-script\lib\coffee-
script\coffee-script.js:134:23)

  at compileScript (C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-
script\lib\coffee-script\command.js:224:29)

  at compilePath (C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-
script\lib\coffee-script\command.js:174:14)

  at Object.exports.run
(C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-script\lib\coffee-
script\command.js:98:20)

  at Object.<anonymous>
(C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-
script\bin\coffee:7:41)

  at Module._compile (module.js:413:34)

  at Object.Module._extensions..js (module.js:422:10)

  at Module.load (module.js:357:32)

  at Function.Module._load (module.js:314:12)

  at Function.Module.runMain (module.js:447:10)

  at startup (node.js:139:18)

  at node.js:999:3
```

The finally block

We can also rewrite the above example using **finally** block. If we do so, the contents of this block are executed unconditionally after **try** and **catch**. Save this code in a file with the name **Exception_handling_finally.coffee**

```
try

  x = y+20

  console.log "The value of x is :" +x

catch e

  console.log "exception/error occurred"

  console.log "The STACKTRACE for the exception/error occurred is ::"

  console.log e.stack


finally

  console.log "This is the statement of finally block"
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c Exception_handling_finally.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var e, error, x;

  try {
    x = y + 20;
    console.log("The value of x is :" + x);
  } catch (error) {
    e = error;
    console.log("exception/error occurred");
    console.log("The STACKTRACE for the exception/error occurred is ::");
    console.log(e.stack);
  } finally {
    console.log("This is the statement of finally block");
  }

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee Exception_handling_finally.coffee
```

On executing, the CoffeeScript file produces the following output.

```
exception/error occurred
The STACKTRACE for the exception/error occurred is ::
ReferenceError: y is not defined
  at Object.<anonymous>
(C:\Examples\strings_exceptions\Exception_handling.coffee:3:7)
  at Object.<anonymous>
(C:\Examples\strings_exceptions\Exception_handling.coffee:2:1)
  at Module._compile (module.js:413:34)
  at Object.exports.run
(C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-script\lib\coffee-
script\coffee-script.js:134:23)
  at compileScript (C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-
script\lib\coffee-script\command.js:224:29)
```

```
  at compilePath (C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-script\lib\coffee-script\command.js:174:14)

  at Object.exports.run
(C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-script\lib\coffee-script\command.js:98:20)

  at Object.<anonymous>
(C:\Users\Tutorialspoint\AppData\Roaming\npm\node_modules\coffee-script\bin\coffee:7:41)

  at Module._compile (module.js:413:34)

  at Object.Module._extensions..js (module.js:422:10)

  at Module.load (module.js:357:32)

  at Function.Module._load (module.js:314:12)

  at Function.Module.runMain (module.js:447:10)

  at startup (node.js:139:18)

  at node.js:999:3


This is the statement of finally block
```

The throw Statement

CoffeeScript also supports the **throw** statement. You can use throw statement to raise your built-in exceptions or your customized exceptions. Later these exceptions can be captured and you can take an appropriate action.

Example

The following example demonstrates the usage of the **throw** statement in CoffeeScript. Save this code in a file with name **throw_example.coffee**

```
myFunc = ->
  a = 100
  b = 0
  try
    if b == 0
      throw ("Divided by zero error.")
    else
      c = a / b
  catch e
    console.log "Error: " + e

myFunc()
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c throw_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var myFunc;

  myFunc = function() {
    var a, b, c, e, error;
    a = 100;
    b = 0;
    try {
      if (b === 0) {
        throw "Divided by zero error.";
      } else {
        return c = a / b;
      }
    } catch (error) {
      e = error;
      return console.log("Error: " + e);
    }
  };

  myFunc();

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee throw_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
Divided by zero error.
```

20. CoffeeScript – Regular Expressions

A regular expression is an object that describes a pattern of characters JavaScript supports. In JavaScript, RegExp class represents regular expressions, and both String and RegExp define methods that use regular expressions to perform powerful pattern-matching and search-and-replace functions on text.

Regular Expressions in CoffeeScript

The regular expressions in CoffeeScript are same as JavaScript. Visit the following link to see the regular expressions in JavaScript – javascript_regular_expressions

Syntax

A regular expression in CoffeeScript is defined by placing the RegExp pattern between the forward slashes as shown below.

```
pattern =/pattern/
```

Example

Following is an example of regular expressions in CoffeeScript. In here, we have created an expression that finds out the data that is in bold (data between <b> and </b> tags). Save this code in a file with name **regex_example.coffee**

```
input_data ="hello how are you welcome to <b>Tutorials Point.</b>"
regex = /<b>(.*)<\/b>/
result = regex.exec(input_data)
console.log result
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c regex_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var input_data, regex, result;

  input_data = "hello how are you welcome to <b>Tutorials Point.</b>";

  regex = /<b>(.*)<\/b>/;

  result = regex.exec(input_data);


  console.log(result);


}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee regex_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
[ '<b>Tutorials Point.</b>',
  'Tutorials Point.',
  index: 29,
  input: 'hello how are you welcome to <b> Tutorials Point.</b>' ]
```

heregex

The complex regular expressions we write using the syntax provided by JavaScript are unreadable, therefore to make Regular expressions more readable, CoffeeScript provides an extended syntax for regular expressions known as **heregex**. Using this syntax, we can break the normal regular expressions using whitespaces and we can also use comments in these extended regular expressions, thus making them more user friendly.

Example

The following example demonstrates the usage of the advanced regular expressions in CoffeeScript **heregex**. In here, we are rewriting the above example using the advanced regular expressions. Save this code in a file with name **heregex_example.coffee**

```
input_data ="hello how are you welcome to Tutorials Point."
heregex = ///
<b>  #bold opening tag
(.*) #the tag value
</b>  #bold closing tag
///
result = heregex.exec(input_data)
console.log result
```

Open the **command prompt** and compile the .coffee file as shown below.

```
c:\> coffee -c heregex_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var heregex, input_data, result;

  input_data = "hello how are you welcome to <b> Tutorials Point.</b>";

  heregex = /<b>(.*) <\/b>/;

  result = heregex.exec(input_data);

  console.log(result);

}).call(this);
```

Now, open the **command prompt** again and run the CoffeeScript file as shown below.

```
c:\> coffee heregex_example.coffee
```

On executing, the CoffeeScript file produces the following output.

```
[ '<b>Tutorials Point.</b>',
  'Tutorials Point.',
  index: 29,
  input: 'hello how are you welcome to <b>Tutorials Point.</b>' ]
```

21. CoffeeScript – Classes and Inheritance

JavaScript does not provide the **class** keyword. We can achieve inheritance in JavaScript using objects and their prototypes. Every object have their own prototype and they inherit functions and properties from their prototypes. Since the prototype is also an object, it also has its own prototype.

Though the prototypal inheritance is far more powerful than classic inheritance, it is difficult and confusing for novice users.

Classes in CoffeeScript

Addressing to this problem, CoffeeScript provides a basic structure known as **class** which is built using the JavaScript's prototypes. You can define a class in CoffeeScript using the class keyword as shown below.

```
class Class_Name
```

Example

Consider the following example, here we have created a class named **Student** using the keyword **class.**

```
class Student
```

If you compile the above code, it will generate the following JavaScript.

```
var Student;


Student = (function() {
  function Student() {}


  return Student;


})();
```

Instantiating a class

We can instantiate a class using the new operator just like other object oriented programming languages as shown below.

```
new Class_Name
```

Example

You can instantiate the above created (Student) class using the **new** operator as shown below.

```
class Student
new  Student
```

If you compile the above code, it will generate the following JavaScript.

```
var Student;

Student = (function() {
  function Student() {}

  return Student;

})();

new Student;
```

Defining a Constructor

A constructor is a function that is invoked when we instantiate a class, its main purpose is to initialize the instance variables. In CoffeeScript, you can define a constructor just by creating a function with name the **constructor** as shown below.

```
class Student
  constructor: (name)->
  @name = name
```

In here, we have defined a constructor and assigned the local variable name to the instance variable. The **@** operator is an alias to the **this** keyword, it is used to point the instance variables of a class.

If we place **@** before an argument of the constructor, then it will be set as an instance variable automatically. Therefore, the above code can be written simply as shown below –

```
class Student
  constructor: (@name)->
```

Example

Here is an example of a constructor in CoffeeScript. Save it in a file with the name **constructor_example.coffee**

```
#Defining a class
class Student
  constructor: (@name)->

#instantiating a class by passing a string to constructor
student = new Student("Mohammed");
console.log "the name of the student is :"+student.name
```

Compiling the code

Open command prompt and compile the above example as shown below.

```
c:\>coffee -c constructor_example.coffee
```

On executing the above command it will produce the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var Student, student;

  Student = (function() {
    function Student(name) {
      this.name = name;
    }

    return Student;

  })();

  student = new Student("Mohammed");

  console.log("The name of the student is :"+student.name);

}).call(this);
```

Executing the Code

Run the above example by executing the following command on the command prompt.

```
coffee constructor_example.coffee
```

On running, the above example gives you the following output.

```
The name of the student is :Mohammed
```

Instance Properties

Same as in objects, we can also have properties within a class. And these are known as **instance properties**.

Example

Consider the following example. In here, we have created variables (name, age) and a function (message()) within the class and accessed them using its object. Save this example in a file named **instance_properties_example.coffee**

```
#Defining a class
class Student
  name="Ravi"
  age=24
  message: ->
    "Hello "+name+" how are you"

#instantiating a class by passing a string to constructor
student = new Student();
console.log student.message()
```

On compiling, the above code generates the following output.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var Student, student;

  Student = (function() {
    var age, name;

    function Student() {}

    name = "Ravi";

    age = 24;

    Student.prototype.message = function() {
      return "Hello " + name + " how are you";
    };

    return Student;

  })();

  student = new Student();

  console.log(student.message());

}).call(this);
```

Static Properties

We can define static properties in the class. The scope of the static properties is restricted within the class and we create static functions using the **this keyword** or its alias @ symbol and we have to access these properties using the class name as *Class_Name.property*.

Example

In the following example, we have created a static function named message. and accessed it. Save it in a file with the name **static_properties_example.coffee.**

```
#Defining a class
class Student
  @message:(name) ->
    "Hello "+name+" how are you"
console.log Student.message("Raju")
```

Open the command prompt and compile the above CoffeeScript file using the following command.

```
c:\>coffee -c  static_properties_example.coffee
```

On compiling, it gives you the following JavaScript.

```
// Generated by CoffeeScript 1.10.0
(function() {
  var Student;

  Student = (function() {
    function Student() {}
    Student.message = function(name) {
      return "Hello " + name + " how are you";
    };

    return Student;

  })();

  console.log(Student.message("Raju"));

}).call(this);
```

Execute the above coffeeScript in command prompt as shown below.

```
c:\>coffee static_properties_example.coffee
```

On executing, the above example gives you the following output.

```
Hello Raju how are you
```

Inheritance

In CoffeeScript, we can inherit the properties of one class in another class using **extends** keyword.

Example

Following is an Example of inheritance in CoffeeScript. In here, we have two classes namely **Add** and **My_class.** We inherited the properties of class named Add in the class My_class, and accessed them using the **extends** keyword.

```
#Defining a class
class Add
   a=20;b=30

   addition:->
     console.log "Sum of the two numbers is :"+(a+b)

class My_class extends Add

my_class = new My_class()
```

```
my_class.addition()
```

CoffeeScript uses prototypal inheritance behind the scenes. In CoffeeScript, whenever we create instances, the parent class's constructor is invoked until we override it.

We can invoke the constructor of the parent class from the subclass, using the **super()** keyword as shown in the example given below.

```
#Defining a class
class Add
   constructor:(@a,@b) ->

   addition:=>
     console.log "Sum of the two numbers is :"+(@a+@b)

class Mul extends Add
   constructor:(@a,@b) ->
     super(@a,@b)

   multiplication:->
     console.log "Product of the two numbers is :"+(@a*@b)

mul = new Mul(10,20)
mul.addition()
mul.multiplication()
```

Dynamic Classes

CoffeeScript uses prototypal inheritance to automatically inherit all of a class's instance properties. This ensures that classes are dynamic; even if you add properties to a parent class after a child has been created, the property will still be propagated to all of its inherited children.

```
class Animal
  constructor: (@name) ->

class Parrot extends Animal

Animal::rip = true

parrot = new Parrot("Macaw")
alert("This parrot is no more") if parrot.rip
```

The Fat Arrow (=>)

CoffeeScript can lock the value of this to a particular context using a fat arrow function: =>. This ensures that whatever context a function is called under, it'll always execute inside the context it was created in.

22. CoffeeScript – Ajax

AJAX is a web development technique for creating interactive web applications.

- AJAX stands for **A**synchronous **Ja**vaScript and **X**ML. It is a new technique for creating better, faster, and more interactive web applications with the help of XML, HTML, CSS, and Java Script.

- Ajax uses XHTML for content, CSS for presentation, along with Document Object Model and JavaScript for dynamic content display.

- Conventional web applications transmit information to and from the server using synchronous requests. It means you fill out a form, hit submit, and get directed to a new page with new information from the server.

- With AJAX, when you hit submit, JavaScript will make a request to the server, interpret the results, and update the current screen. In the purest sense, the user would never know that anything was even transmitted to the server.

- XML is commonly used as the format for receiving server data, although any format, including plain text, can be used.

- AJAX is a web browser technology independent of web server software.

- A user can continue to use the application while the client program requests information from the server in the background.

In general, we use jQuery to work with Ajax. Following is an example of Ajax and jQuery

```
<html>


   <head>
      <title>The jQuery Example</title>
      <script type = "text/javascript"
         src =
"http://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>


      <script type = "text/javascript" language = "javascript">
         $(document).ready(function() {
            $("#driver").click(function(event){
               $('#stage').load('/jquery/result.html');
            });
         });
      </script>
   </head>
   <body>

      <p>Click on the button to load /jquery/result.html file –</p>

      <div id = "stage" style = "background-color:cc0;">
         STAGE
      </div>

      <input type = "button" id = "driver" value = "Load Data" />

   </body>

</html>
```

Here **load()** initiates an Ajax request to the specified URL **/coffeescript/result.html** file. After loading this file, all the content would be populated inside <div> tagged with ID *stage*. Assuming that our /jquery/result.html file has just one HTML line –

```
<h1>THIS IS RESULT...</h1>
```

When you click the given button, then result.html file gets loaded.

CoffeeScript with Ajax

We can rewrite the above example using CoffeeScript as shown below.

```
<html>

   <head>
      <title>The jQuery Example</title>
      <script type = "text/javascript"
         src =
"http://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
      <script src="http://coffeescript.org/extras/coffee-script.js"></script>

      <script type="text/coffeescript">
        $(document).ready ->
          $('#driver').click (event) ->
            $('#stage').load '/jquery/result.html'
```

```
             return
           return
       </script>
    </head>

    <body>

       <p>Click on the button to load /jquery/result.html file -</p>

       <div id = "stage" style = "background-color:cc0;">
          STAGE
       </div>

       <input type = "button" id = "driver" value = "Load Data" />

    </body>

</html>
```

23. CoffeeScript – jQuery

jQuery is a fast and concise library/framework built using JavaScript created by John Resig in 2006 with a nice motto – Write less, do more.

jQuery simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. Visit our jQuery tutorial to know about jQuery.

We can also use CoffeeScript to work with **jQuery**. This chapter teaches you how to use CoffeeScript to work with jQuery.

Using CoffeeScript with jQuery

Though jQuery solves the browser issues, using it with JavaScript which have some bad parts is a bit problematic. Using CoffeeScript instead of JavaScript is a better idea.

Keep the following points in mind while converting the to be while using jQuery with CoffeeScript.

The **$** symbol indicates the jQuery code in our application. Use this to separate the jQuery code from the scripting language as shown below.

```
$(document).ready
```

There is no need of using braces in in CoffeeScript except while calling the functions with parameters and dealing with the ambiguous code and we have to replace the function definition **function()** with an arrow mark as shown below.

```
$(document).ready ->
```

Remove the unnecessary return statements, since CoffeeScript implicitly returns the tailing statements of a function.

Example

Following is an JavaScript code where <div> elements are being inserted just before the clicked element –

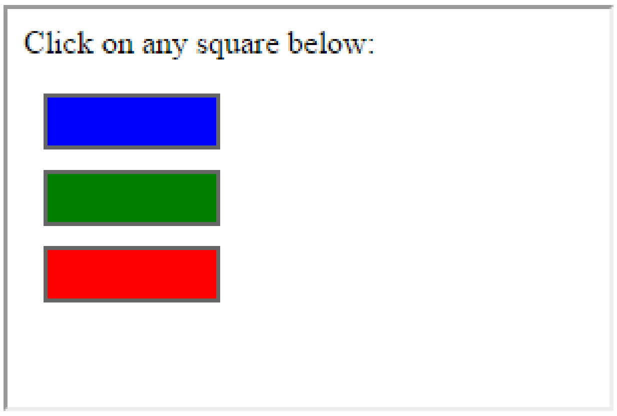
```
<html>


   <head>
      <title>The jQuery Example</title>
      <script type = "text/javascript"
         src =
"http://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>


      <script type = "text/javascript" language = "javascript">
         $(document).ready(function() {
            $("div").click(function () {
               $(this).before('<div class="div"></div>' );
            });
         });
      </script>

      <style>
         .div{ margin:10px;padding:12px; border:2px solid #666; width:60px;}
      </style>
   </head>

   <body>

      <p>Click on any square below:</p>
      <span id = "result"> </span>

      <div class = "div" style = "background-color:blue;"></div>
      <div class = "div" style = "background-color:green;"></div>
      <div class = "div" style = "background-color:red;"></div>

   </body>

</html>
```

Now, we can convert the above code into CoffeeScript code as shown below

```
  <html>

    <head>
       <title>The jQuery Example</title>
       <script type = "text/javascript" src =
 "http://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
       <script src="http://coffeescript.org/extras/coffee-script.js"></script>

       <script type="text/coffeescript">
         $(document).ready ->
           $('div').click ->
             $(this).before '<div class="div"></div>'
             return
```

```
          return
      </script>

      <style>
         .div{ margin:10px;padding:12px; border:2px solid #666; width:60px;}
      </style>
   </head>

   <body>

      <p>Click on any square below:</p>
      <span id = "result"> </span>

      <div class = "div" style = "background-color:blue;"></div>
      <div class = "div" style = "background-color:green;"></div>
      <div class = "div" style = "background-color:red;"></div>

   </body>

</html>
```

On executing, this gives you the following output.


Click on any square below:

What is Callback?

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task. Node makes heavy use of callbacks. All APIs of Node are written is such a way that they supports callbacks.

For example, a function to read a file may start reading file and return the control to execution environment immidiately so that next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process high number of request without waiting for any function to return result.

Blocking Code Example

Create a text file named input.txt having following content

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Create a js file named main.js which has the following code:

```
var fs = require("fs");

var data = fs.readFileSync('input.txt');

console.log(data.toString());
console.log("Program Ended");
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
Program Ended
```

Non-Blocking Code Example

Create a text file named input.txt having following content

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Update main.js file to have following code:

```
var fs = require("fs");

fs.readFile('input.txt', function (err, data) {
    if (err) return console.error(err);
    console.log(data.toString());
});
```

```
console.log("Program Ended");
```

Now run the main.js to see the result:

```
$ node main.js
```

Verify the Output.

```
Program Ended

Tutorials Point is giving self learning content

to teach the world in simple and easy way!!!!!
```

These two examples explain the concept of **blocking and non-blocking calls**. The first example shows that the program blocks until it reads the file and then only, it proceeds to end the program, whereas in the second example, the program does not wait for file reading but it just proceeded to print "Program Ended".

Thus, a blocking program executes very much in sequence. From programming point of view, its easier to implement the logic but non-blocking programs do not execute in sequence. In case a program needs to use any data to be processed, it should be kept within the same block to make it sequential execution.

24. CoffeeScript – MangoDB

MongoDB is a cross-platform, document oriented database that provides, high performance, high availability, and easy scalability. MongoDB works on concept of collection and document. For more information read our MongoDB Tutorial.

In this chapter you will learn how to communicate with MongoDB database using CoffeeScript.

Installation

The MongoDB database can be integrated with CoffeeScript using **Node.js 2.0** driver of MongoDB. First of all you need to install MongoDB in your system, by referring theenvironment chapter of our MangoDB tutorial.

After installing MangoDB successfully browse through its **bin** folder (if you haven't set the path) and start the MangoDB service as shown below.

```
C:\Program Files\MongoDB\Server\3.2\bin> mongod
```

Finally install MangoDB driver and it's dependencies by executing the following NPM command in the command prompt.

```
npm install mongodb --save
```

Connecting to MangoDB

In order to connect to MongoDB, first of all create MongoClient using this, invoke the**connect()** function. This function accepts url, and a callback function as parameters.

Following CoffeeScript code shows how to connect to MongoDB server. If the MongoDB server is running in your system this program establishes a connection to the server.

```
#Requiring the Mongodb package
mongo = require 'mongodb'

#Creating a MongoClient object
MongoClient = mongo.MongoClient

#Preparing the URL
url = 'mongodb://localhost:27017/testdb'

#Connecting to the server
MongoClient.connect url, (err, db) ->
  if err
```

```
    console.log 'Unable to connect . Error:', err
  else
    console.log 'Connection established to', url
    #Close connection
    db.close()
  return
```

Save the above code in a file with name **connect_db.coffee** and execute it as shown below. If database is successfully created then it will give following message

```
c:\> coffee connect_db.coffee
coffee connect_db.collection
Connection established to mongodb://localhost:27017/testdb
```

Creating a Collection

A collection in MongoDB holds the documents we store in it. You can create a collection by using the **collection()** function. This function accepts a string argument that represents the name of the collection we want to create.

Following CoffeeScript code shows how to create a collection in MongoDB. In case of any errors, they will be displayed on the console.

```
#Requiring the Mongodb package
mongo = require 'mongodb'

#Creating a MongoClient object
MongoClient = mongo.MongoClient

#Preparing the URL
url = 'mongodb://localhost:27017/testdb'

#Connecting to the server
MongoClient.connect url, (err, db) ->
  if err
    console.log 'Unable to connect . Error:', err
  else
    console.log 'Connection established to', url


    #Create collection
    col = db.collection('My_collection')
    console.log "Collection created successfully."

    #Close connection
    db.close()
  return
```

Save the above code in a file with name **create_collection.coffee** and execute it as shown below. If the collection is created successfully then it will give following message

```
c:/>
coffee create_collection.coffee
Connection established to mongodb://localhost:27017/testdb
Collection created successfully.
```

Inserting Documents

You can inset documents in to a collection in MangoDB you need to invoke a function named **insert()** by passing the list of documents that are needed to be inserted, as parameters.

Following CoffeeScript code shows how to insert documents in to a collection named**My_collection.** In case of any errors, they will be displayed on the console.

```
#Sample JSON Documents
doc1 = {name: 'Ram', age: 26, city: 'Hyderabad'}
doc2 = {name: 'Rahim', age: 27, city: 'Banglore'}
doc3 = {name: 'Robert', age: 28, city: 'Mumbai'}

#Requiring the Mongodb package
mongo = require 'mongodb'

#Creating a MongoClient object
MongoClient = mongo.MongoClient

#Preparing the URL
url = 'mongodb://localhost:27017/testdb'

#Connecting to the server
MongoClient.connect url, (err, db) ->
  if err
```

```
    console.log 'Unable to connect . Error:', err
  else
    console.log 'Connection established to', url
  #Creating collection
  col = db.collection('My_collection')

  #Inserting documents
  col.insert [doc1,doc2,doc3], (err, result) ->
    if err
      console.log err
    else
      console.log "Documents inserted successfully"
    #Close connection
    db.close()
    return
  return
```

Save the above code in a file with name **insert_documents.coffee** and execute it as shown below. If the documents are inserted successfully then it gives following message

```
c:/>
coffee insert_documents.coffee
Connection established to mongodb://localhost:27017/testdb
Documents inserted successfully
```

Reading Documents

You can retrieve the documents that are stored in MongoDB using a function named **find().** The following CoffeeScript code shows how to retrieve the records that are stored in MangoDB.

```
#Requiring the Mongodb package
mongo = require 'mongodb'

#Creating a MongoClient object
MongoClient = mongo.MongoClient

#Preparing the URL
url = 'mongodb://localhost:27017/testdb'

```

```
#Connecting to the server
MongoClient.connect url, (err, db) ->
  if err
    console.log 'Unable to connect . Error:', err
  else
    console.log 'Connection established to', url
     #Creating collection object
    col = db.collection('My_collection')
    #Inserting Documents
    col.find({name: 'Ram'}).toArray (err, result)->
      if err
        console.log err
      else
      console.log 'Found:', result
      #Closing connection
      db.close()
      return
  return
```

Save the above code in a file with name **read_documents.coffee** and execute it as shown below. This programs retrieves the required document in the specified collection and displays it as shown below.

```
C:\> coffee read_documents.coffee
Connection established to mongodb://localhost:27017/testdb
Found: [ { _id: 56e269c10478809c3009ad1e,
    name: 'Ram',
    age: 26,
    city: 'Hyderabad' } ]
```

You can also read all the documents existing in a particular collection by executing the**find()** function with out passing any arguments to it as shown below.

```
#Requiring the Mongodb package
mongo = require 'mongodb'

#Creating a MongoClient object
MongoClient = mongo.MongoClient

#Preparing the URL
```

```
url = 'mongodb://localhost:27017/testdb'


#Connecting to the server
MongoClient.connect url, (err, db) ->
  if err
    console.log 'Unable to connect . Error:', err
  else
    console.log 'Connection established to', url
     #Creating collection object
    col = db.collection('My_collection')
    #Reading all Documents
    col.find().toArray (err, result)->
      if err
        console.log err
      else
      console.log 'Found:', result
      #Closing connection
      db.close()
      return
  return
```

Save the above code in a file with name **read_all_documents.coffee** and execute it as shown below. this programs retrieves all the documents in the specified collection and displays it as shown below.

```
C:\> coffee read_all_documents.coffee
Connection established to mongodb://localhost:27017/testdb
Found: [ { _id: 56e2c5e27e0bad741a68c03e,
    name: 'Ram',
    age: 26,
    city: 'Hyderabad' },
  { _id: 56e2c5e27e0bad741a68c03f,
    name: 'Rahim',
    age: 27,
    city: 'Banglore' },
  { _id: 56e2c5e27e0bad741a68c040,
    name: 'Robert',
    age: 28,
    city: 'Mumbai' } ]
```

Updating Documents

You can update the documents that are stored in MongoDB using a function named**update().** Following CoffeeScript code shows how to update the records that are stored in MangoDB.

```
#Get mongo client object
MongoClient = require('mongodb').MongoClient
#Connecting to mongodb
url = 'mongodb://localhost:27017/testdb'
MongoClient.connect url, (err, db) ->
  if err
    console.log 'Unable to connect . Error:', err
  else
    console.log 'Connection established to', url
     #Creating collection
    col = db.collection('My_collection')
    #Reading Data
    col.update {name:'Ram'},{$set:{city:'Delhi'}},(err, result)->
      if err
        console.log err
      else
      console.log "Document updated"

      #Closing connection
      db.close()
       return
  return
```

This program updates the city of the employee named Ram from Hyderabad to Delhi.

Save the above code in a file with name **update_documents.coffee** and execute it as shown below. this programs retrieves the documents in the specified collection and displays it as shown below.

```
C:\> coffee update_documents.coffee
Connection established to mongodb://localhost:27017/testdb
Document updated
```

After updating, if you execute the **read_documents.coffee** program, then you will observe that the city name of the person named Ram is updated from **Hyderabad** to **Delhi**.

```
C:\> coffee Read_all_documents.coffee
Connection established to mongodb://localhost:27017/testdb
Found: [ { _id: 56e2c5e27e0bad741a68c03e,
    name: 'Ram',
    age: 26,
    city: 'Delhi' },
  { _id: 56e2c5e27e0bad741a68c03f,
    name: 'Rahim',
    age: 27,
    city: 'Banglore' },
  { _id: 56e2c5e27e0bad741a68c040,
    name: 'Robert',
    age: 28,
    city: 'Mumbai' } ]
```

Deleting Documents

You can delete all the documents from the collection using the **remove()** function. Following CoffeeScript code shows how to delete all the records that are stored in MangoDB.

```
#Get mongo client object
MongoClient = require('mongodb').MongoClient
#Connecting to mongodb
url = 'mongodb://localhost:27017/testdb'
MongoClient.connect url, (err, db) ->
  if err
    console.log 'Unable to connect . Error:', err
  else
    console.log 'Connection established to', url

     #Creating collection
    col = db.collection('My_collection')
    #Deleting Data
    col.remove()
    console.log "Document deleted"
    #Closing connection
    db.close()
  return
```

Save the above code in a file with name **delete_documents.coffee** and execute it as shown below. this programs removes all the documents in the specified collection displaying the following messages.

```
C:\> coffee delete_documents.coffee
Connection established to mongodb://localhost:27017/testdb
Document deleted
```

After deleting, if you execute the **read_documents.coffee** program, then you will get an empty collection as shown below.

```
C:\> coffee Read_all_documents.coffee
Connection established to mongodb://localhost:27017/testdb
Found: [ ]
```

25. CoffeeScript – SQLite

SQLite is a lightweight, schema-based relational database engine. It is a popular choice as embedded database software for local storage in web browsers.

Unlike many other database management systems, SQLite is not a client–server database engine. For more information read our SQLite Tutorial

In this chapter you will learn how to communicate with SQLite database using CoffeeScript.

Installation

The SQLite3 database can be integrated with CoffeeScript using **node-sqlite3** module. This module works with Node.js v0.10.x, v0.12.x, v4.x, and v5.x. This module caters various functions to communicate with SQLite3 using CoffeeScript, in addition to this it also provides an Straightforward query and parameter binding interface, and an Query serialization API.

You can install the node-sqlite3 module using npm as shown below.

```
npm install sqlite3
```

To use sqlite3 module, you must first create a connection object that represents the database and this object will help you in executing all the SQL statements.

Connecting to Database

In order to connect to SQLite database first of all create an its package by invoking the **require()** function of the **node-sqlite3** module and pass the string **sqlite3** as a parameter to it. Then connect to a database by passing the name of the database to **sqlite3.Database()** construct.

Following CoffeeScript code shows how to connect to an existing database. If database does not exist, then it will be created with the given name **test.db**, opened and finally the database object will be returned.

```
#Creating sqlite3 package
sqlite3 = require('sqlite3')


#Creating a Database instance
db = new (sqlite3.Database)('test.db')
console.log "Database opened successfully."
```

We can also supply **:memory:** to create an anonymous in-memory database and, an empty string to create anonymous disk-based database, instead of *test.db*. Save the above code in a file with name **create_db.coffee** and execute it as shown below. If the database is successfully created, then it will produce the following message –

```
c:\> coffee create_db.coffee
Successfully connected
```

Creating a Table

You can create a table in SQLite database through CoffeeScript using the **run()** function. Pass the query to create a table to this function in String format.

The following CoffeeScript program will be used to create a table in previously **test.db** database –

```
#Creating sqlite3 package
sqlite3 = require('sqlite3')

#Creating a Database instance
db = new (sqlite3.Database)('test.db')
console.log "Successfully connected"

db.serialize ->
  db.run 'CREATE TABLE STUDENT (name TEXT, age INTEGER, city TEXT)'
  console.log "Table created successfully"
  return
db.close()
```

The **serialize()** function sets the database in serialized mode. In this mode when ever a callback encounters, it will be called immediately. The quires in that callback are executes serially. Soon the function returns The database will be set to normal mode again. After completing the transaction we need to close the connection using **close()** function.

Save the above code in a file with name **create_table.coffee** and execute it as shown below. This will create a table named **STUDENT** in the database *test.db* displaying the following messages.

```
C:\> coffee create_table.coffee
Successfully connected
Table created successfully
```

Inserting / Creating Data

You can insert data into SQLite database through CoffeeScript code by executing the insert statement. To do so we can use the **prepare()** function which prepares SQL statements.

It also accepts query with bind variables (**?**), values to these variables can be attached using **run()** function. You can insert multiple records using prepared statement, and after inserting all the records, you need to finalize the prepared statement using **finalize()** function.

The following CoffeeScript program shows how to insert records in the table named STUDENT created in previous example.

```
#Creating sqlite3 package
sqlite3 = require('sqlite3').verbose()

#Creating a Database instance
db = new (sqlite3.Database)('test.db')
console.log "Successfully connected"

db.serialize ->
  stmt = db.prepare('INSERT INTO STUDENT VALUES (?,?,?)')
  stmt.run 'Ram',24,'Hyderabad'
  stmt.run 'Robert',25,'Mumbai'
  stmt.run 'Rahim',26,'Bangalore'
  stmt.finalize()
  console.log "Data inserted successfully"
  return
db.close()
```

Save the above code in a file with name **insert_data.coffee** and execute it as shown below. This will populate the table named STUDENT displaying the following messages.

```
C:\> coffee insert_data.coffee
Successfully connected
Data inserted successfully
```

Reading / Retrieving Data

You can get the data from an SQLite table using the **each()** function. This function accepts an optional callback function which will be called on each row.

The following CoffeeScript program shows how we can fetch and display records from the table named STUDENT created in the previous example

```
#Creating sqlite3 package
sqlite3 = require('sqlite3').verbose()

#Creating a Database instance
db = new (sqlite3.Database)('test.db')
console.log "Successfully connected"
```

```
db.serialize ->
  console.log "The contents of the table STUDENT are ::"
  db.each 'SELECT rowid AS id, name,age,city FROM STUDENT', (err, row) ->
    console.log row.id + ': ' +row.name+', '+ row.age+', '+ row.city
    return
  return
db.close()
```

Save the above code in a file with name **retrive_data.coffee** and execute it as shown below. This retrieves all the records in the table named STUDENT and displays on the console as follows.

```
C:\> coffee retrive_data.coffee
Successfully connected
The contents of the table STUDENT are ::
1: Ram, 24, Hyderabad
2: Robert, 25, Mumbai
3: Rahim, 26, Bangalore
```

Updating Data

The following CoffeeScript code shows how we can use UPDATE statement to update any record and then fetch and display updated records in the table named STUDENT

```
#Creating sqlite3 package
sqlite3 = require('sqlite3').verbose()

#Creating a Database instance
db = new (sqlite3.Database)('test.db')
console.log "Successfully connected"

db.serialize ->
  #Updating data
  stmt = db.prepare('UPDATE STUDENT SET city = ? where name = ?')
  stmt.run 'Delhi','Ram'
  console.log "Table updated"
  stmt.finalize()



  #Retrieving data after update operation
  console.log "The contents of the table STUDENT after update operation are ::"
  db.each 'SELECT rowid AS id, name, city FROM STUDENT', (err, row) ->
    console.log row.id + ': ' +row.name+', '+ row.city
    return
  return
db.close()
```

Save the above code in a file with name **update_data.coffee** and execute it as shown below. This updates the city of the student named ram and displays all the records in the table after update operation as follows.

```
C:\> coffee update_data.coffee
Successfully connected
Table updated
The contents of the table STUDENT after update operation are ::
1: Ram, Delhi
2: Robert, Mumbai
3: Rahim, Bangalore
```

Deleting Data

The following CoffeeScript code shows how we can use DELETE statement to delete any record and then fetch and display remaining records from the table named STUDENT.

```
#Creating sqlite3 package
sqlite3 = require('sqlite3').verbose()

#Creating a Database instance
db = new (sqlite3.Database)('test.db')
console.log "Successfully connected"

db.serialize ->
  #Deleting data
  stmt = db.prepare('DELETE FROM STUDENT WHERE name = ?')
  stmt.run 'Ram'
  console.log "Record deleted"

  stmt.finalize()

  #Retrieving data after delete operation
  console.log "The contents of the table STUDENT after delete operation are ::"
  db.each 'SELECT rowid AS id, name, city FROM STUDENT', (err, row) ->
    console.log row.id + ': ' +row.name+', '+ row.city
    return
  return
db.close()
```

Save the above code in a file with name **delete_data.coffee** and execute it as shown below. It deletes the record of the student named ram and displays all the remaining in the table after delete operation as follows.

```
Successfully connected
Record deleted
The contents of the table STUDENT after delete operation are ::
2: Robert, Mumbai
3: Rahim, Bangalore
```