There's a lot of hype and fuss about CoffeeScript, a new programming language built on top of JavaScript. CoffeeScript offers a clean syntax that should appeal to those who like Python or Ruby. It also provides many functional programming features inspired by languages such as Haskell and Lisp. CoffeeScript compiles into efficient JavaScript, and in addition to running the JavaScript in a web browser, you can use it with technologies such as Node.js for server applications. In this first article of a four-part series, get started with CoffeeScript and learn about the perks for developers. You will set up the CoffeeScript compiler and use it to create code that's ready to run in a browser or server.

Introduction

The CoffeeScript programming language is built on top of JavaScript, and it compiles into efficient JavaScript that you can run in a web browser or use with technologies such as Node.js for server applications. The compilation is usually straightforward, and the JavaScript that is produced is consistent with many best practices. In this article, learn about the features of the CoffeeScript programming language. After you install CoffeeScript and run the compiler, you'll walk through a simple example using CoffeeScript in a web page.

Download the source code used in this article.

The appeal of CoffeeScript

Frequently used abbreviations

- NPM: Node Package Manager
- REPL: Read-Evaluate-Print-Loop

It's easy to argue that JavaScript is the most important programming language these days. It is the language of browsers, and is increasingly found in desktop and mobile applications. With the growing popularity of Node.js, JavaScript has become a viable option for server and system applications. Some developers vehemently resist JavaScript largely because of its inconsistent syntax and quirky implementations. However, the quirky implementations have subsided as more standardization has come to JavaScript virtual machines. The inconsistent syntax may be addressed somewhat by the next evolution of JavaScript: the ECMAScript.next, an emerging standard that has been heavily influenced by CoffeeScript. But, until a new standard is agreed upon and implemented by the popular virtual machines, there's a lot of room for improvement of JavaScript's syntax.
If you're writing for a JavaScript runtime, CoffeeScript is an enticing option. From a syntax perspective, JavaScript is quite the hodge-podge. It has many features of functional programming languages and is heavily influenced by Scheme, in particular. However, Scheme is a very simple syntax built on s-expressions. JavaScript shares many of the concepts in Scheme, but not its syntax. Instead, JavaScript has a C-like syntax. The result is a language with functional concepts, but a verbose syntax that has no natural constructs for expressing these concepts. For example, JavaScript allows for higher-order functions, such as functions whose input parameters include other functions. This is both useful and powerful, and is a feature missing from many languages. JavaScript can have not-so-elegant syntax, though, as shown in Listing 1.

**Listing 1. Ugly JavaScript**

```javascript
pmb.requestPaymentInfo('type', function(info){
  $('result').innerHTML = info.name;
});
```

There is a lot of boilerplate in the example—parentheses, commas, curly braces, semicolons, and language keywords that are not really necessary.

JavaScript's primary use is as a client-side web application language. Desktop and mobile application frameworks such as Cocoa, Windows® Forms, and Android are all object-oriented. The object-oriented paradigm is not perfect for everything, but it's a nice fit for applications with a graphical user interface. JavaScript is also an object-oriented language complete with inheritance, but it is prototypal—not a class-based language like the languages used by most application frameworks. Thus, application programming in JavaScript can be very cumbersome.

CoffeeScript addresses the JavaScript pain points. CoffeeScript:

- Provides a simpler syntax that reduces boilerplate, such as parentheses and commas
- Uses whitespace as a way to organize blocks of code
- Provides simple syntax for expressing functions
- Supplies class-based inheritance (which is optional, but can be very useful when doing application development)

You might speculate that CoffeeScript, with an abstracted syntax, could have some disadvantages compared to JavaScript. For example, is CoffeeScript much slower than JavaScript, or does it require a heavy runtime library? In actuality, CoffeeScript compiles into clean, efficient JavaScript. You can always see exactly what it is compiling to, so you can be confident there is nothing excessive being introduced. And, because CoffeeScript compiles into fully functional JavaScript, there's no need for any type of runtime library. CoffeeScript gives you a syntax that lets you fully exploit the power of JavaScript, with minimal runtime overhead.

**Prerequisites**

As mentioned, you can use CoffeeScript to write server and system applications that run on top of Node.js. The relationship between CoffeeScript and Node.js runs deeper, though. To install CoffeeScript you need to first install Node.js because:
• CoffeeScript is distributed as a Node.js package using node's package manager, NPM.
• CoffeeScript must be compiled. Its compiler is actually written in CoffeeScript and, thus, requires a JavaScript runtime to do its compiling. The V8 JavaScript virtual machine at the heart of Node.js is perfect for the task.

To follow along with the example in this article, you need to install Node.js.

Installation

Have you ever wished you could run JavaScript from the command line? I haven't, but maybe CoffeeScript will change that. With Node.js, you can run JavaScript from the command line or as part of an executable script. This key feature of Node.js allows CoffeeScript code to be executed on the command line, providing the runtime needed for the CoffeeScript compiler (which is written in CoffeeScript).

The very first step is to install Node.js. You have several options for installation; you could compile the source code or run one of the installers that are available for various systems. Run `node -v` from the command line to confirm that Node.js is installed and on your path.

With Node.js you get an extra bonus: the node package manager (NPM). After you've run `npm -v` from the command line to confirm that NPM is installed and on your path, you can use NPM to install CoffeeScript as follows.

1. Run `npm install --global coffee-script` from the command line.
   The `--global` flag makes CoffeeScript available system-wide and not just for a specific project.
2. The `npm` command should output something like `/usr/bin/coffee -> /usr/lib/node_modules/coffee-script/bin/coffee`
   NPM creates a shortcut in `/usr/bin`, so now the coffee executable is on the correct path. This is the CoffeeScript compiler and interpreter.
3. To verify that the coffee executable is on the path, run `coffee -v` from the command line.

There's one last step to ensure that the CoffeeScript environment is set up properly. To make CoffeeScript available to any Node.js processes that you start, you need to add it to what Node.js calls its NODE_PATH. Node.js will search NODE_PATH for modules (libraries) when it encounters unrecognized functions.

For the example in this article, you'll use Node.js mainly as a runtime for the CoffeeScript executables. The easiest approach is to simply add all of the NPM modules to NODE_PATH. To find where the NPM modules are located, enter `npm ls -g`. You need to add an environment variable pointing NODE_PATH to this location. For example, if `npm ls -g` prints `/usr/lib`, the modules are located in `/usr/lib/node_modules`. To set a NODE_PATH environment variable, run `export NODE_PATH=/usr/lib/node_modules`.

You could streamline things even more by putting the previous command in your startup script (~/.bash_profile, for example). To verify the changes, start a Node.js shell by executing Node and then type `require('coffee-script')`. The Node.js shell should load the CoffeeScript library. If it
works, your CoffeeScript environment is good to go. You can now start exploring CoffeeScript by starting with the compiler.

**The compiler**

Running the CoffeeScript compiler is as easy as entering `coffee -c`, which launches the CoffeeScript read-evaluate-print-loop (REPL). To execute the compiler, you need to pass it a CoffeeScript file that you want to compile. Create a file called `cup0.coffee` and paste the contents of Listing 2 into the file.

### Listing 2. Cup 0

```coffee
for i in [0..5]
  console.log "Hello #{i}"
```

You can probably guess what the two lines of code in Listing 2 will do. Listing 3 shows the output of running `coffee cup0.coffee`.

### Listing 3. Running your first CoffeeScript

```
$ coffee cup0.coffee
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
```

To get a better idea of what is happening, try running the compiler. Enter `coffee -c cup0.coffee`, which creates a file named `cup0.js`. Listing 4 shows the contents of `cup0.js`.

### Listing 4. Cup 0 JavaScript

```javascript
(function() {
  var i;
  for (i = 0; i <= 5; i++) {
    console.log("Hello " + i);
  }
}).call(this);
```

A benefit of CoffeeScript is that, even though it provides a syntax that's more elegant than JavaScript, it compiles into very simple, logical JavaScript. You might wonder why all of the code is wrapped in a function. It's because JavaScript only supports function-level scoping. By wrapping everything in a function, you make sure that the variable is only scoped to that function and does not become global (or change an existing global variable).

Open a new file called `cup1.coffee` and enter the more complicated code in Listing 5.
Listing 5. Cup 1

```coffee
stdin = process.openStdin()
stdin.setEncoding 'utf8'
stdin.on 'data', (input) ->
  name = input.trim()
  process.exit() if name == 'exit'
  console.log "Hello #{name}"
  console.log "Enter another name or 'exit' to exit"
console.log 'Enter your name'
```

The program in Listing 5 prompts the user to enter their name and then greets them appropriately. JavaScript does not have any built-in libraries for reading from standard input, but Node.js does. This is another example of taking advantage of the CoffeeScript/Node.js symbiosis. In languages such as C, reading from standard input is a blocking call. No code can execute until the reading from standard input finishes. If you're familiar with Node.js, you know you cannot do things that way; Node.js does not allow for blocking I/O. Instead, you must register a callback with `stdin.on`.

Run `coffee -c cup1.coffee` to see, as in Listing 6, the JavaScript that the CoffeeScript compiler produces.

Listing 6. Cup 1 JavaScript

```javascript
(function() {
  var stdin;
  stdin = process.openStdin();
  stdin.setEncoding('utf8');
  stdin.on('data', function(input) {
    var name;
    name = input.trim();
    if (name === 'exit') {
      process.exit();
    }
    console.log("Hello " + name);
    return console.log("Enter another name or 'exit' to exit");
  });
  console.log('Enter your name');
}).call(this);
```

The `stdin.on` function uses a typical event binding format. You specify the kind of event (`'data'`) you want to listen for, then give it a callback function to be executed when the event is fired. In the compiled JavaScript, you see typical verbose JavaScript for creating an inline function and passing it to another function. Compare this to the equivalent CoffeeScript. Does it make you miss all of those parentheses, curly braces, semicolons, and keywords?

Now that you know how to compile CoffeeScript programs, the next section looks at one of the most useful features for learning CoffeeScript: the REPL.

**REPL**

REPL is a standard tool found in many programming languages, particularly of the functional variety. REPL is the equivalent of Ruby's IRB. Simply entering `coffee` will launch the CoffeeScript REPL. For example, experiment with this CoffeeScript feature and solve simple problems, as shown in Listing 7.
Listing 7. Using the REPL

```
coffee> nums = [1..10]
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
coffee> isOdd = (n) -> n % 2 == 1
[Function]
coffee> odds = nums.filter isOdd
[ 1, 3, 5, 7, 9 ]
coffee> odds.reduce (a,b) -> a + b
25
```

Each time you enter an expression into the REPL, it will evaluate the expression, print the result, and then wait for the next expression. The example defines a variable called `nums` with a range from 1 to 10. The REPL prints out the value of the variable that was just defined. This feature can be immediately useful. Perhaps you don't remember if the range you defined is inclusive (includes the last number—in this case, 10)—or exclusive. The REPL shows you that 10 is included, so it is an inclusive range. If you wanted an exclusive range, just use `nums = [1...10]`.

The `isOdd` function is defined next. CoffeeScript has very concise syntax for functional declarations, which is a nice feature given the functional nature of JavaScript. In the example, the REPL simply shows `[Function]` to let you know that the `isOdd` variable is equal to a function. A new variable, `odds`, is then declared. You get the value of `odds` by invoking the filter function on `nums` and passing in `isOdd`. This produces a new array whose elements must produce true when passed to `isOdd`. The `reduce` function is invoked on `odds`. Passing in a function adds each value of the array to the previous sum, which sums up the values in the array, and the REPL displays the sum of 25.

The next section covers a topic near and dear to any JavaScript developer: scripting in the browser.

Simple web example

You've seen how you can write CoffeeScript files and compile them into JavaScript, which, of course, can then be used in a web application. For development purposes, there is an easier way. The CoffeeScript compiler can run inside the browser, allowing you to use CoffeeScript directly in a web page. However, if you're building a high performance web application, this is not the advised way to use CoffeeScript. The CoffeeScript compiler is a large file; having to compile your CoffeeScript on the fly definitely slows things down. CoffeeScript does provide an easy way to develop your application with an obvious "path to production."

CoffeeScript is not a JavaScript toolkit or framework. It is a programming language, and, thus, does not include a lot of DOM-related convenience functions. However, you can use CoffeeScript with your favorite toolkit. The most common combination is to use it with jQuery. Listing 8 shows a simple web page that uses jQuery and CoffeeScript.

Listing 8. CoffeeScript in a web page

```html
<html>
<head>
</head>
<body>
</body>
</html>
```
The web page uses the Google Ajax library to load jQuery. It loads the CoffeeScript compiler library from CoffeeScript creator Jeremy Ashkenas' Github repository (see Related topics). The code then includes a script block. Instead of it being of type text/javascript, the script block is of type text/coffeescript, which is how the CoffeeScript compiler knows to compile the script contents. The script then creates a function called gcd that calculates the greatest common divisor of two integers. jQuery is used to create a click handler for the button on the page. In this handler you get the values of the two text inputs and pass those to the gcd function. The result is written back to the web page. Functions such as $(), val(), and html() are jQuery functions, but they can be easily used with CoffeeScript and take advantage of CoffeeScript's clean syntax.

**Conclusion**

In this article, you got a quick tour of CoffeeScript. With your development environment up and running, you can now explore CoffeeScript using the REPL. You learned to use the compiler to see what kind of JavaScript it produces, and also learned how to write CoffeeScript and run it directly in your web pages. The examples provided a taste of CoffeeScript's syntax, though often without a lot of explanation.

Part 2 in this series will dive deeper into the details of CoffeeScript key concepts.
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>cs1.zip</td>
<td>1KB</td>
</tr>
</tbody>
</table>
Related topics

- "Just what is Node.js?" (developerWorks, May 2011) explains how Node is a server-side JavaScript interpreter that changes the notion of how a server should work.
- Node.js: Use this launching point to learn more about the application.
- "Use Node.js as a full cloud environment development stack" (developerWorks, April 2011) details how Node.js can be combined with cloud technologies.
- "High-performance web development with Google Web Toolkit and Eclipse" (developerWorks, October 2009): Compiling into JavaScript is not a new idea, so if you are a fan of the Java programming language you should read this article.
- "All aboard! An introduction to Rails 3" (developerWorks, March 2010) explains how CoffeeScript is now part of Ruby on Rails. Check out the other new features in Rails in this article.
- CoffeeScript on Github: Start here to learn about CoffeeScript.
- "Create Ajax applications for the mobile web" (developerWorks, March 2010) gives you more information about using Ajax in mobile web applications.
- developerWorks on Twitter: Join today to follow developerWorks tweets.
- Node.js: Download Node.js and start easily building scalable network programs.
- IBM product evaluation versions: Download or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.