عنوان: **CoffeeScript #1**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۳/۲۷ ۹:۲۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

مقدمه CoffeeScript یک زبان برنامه نویسی برای تولید کدهای جاوااسکریپت است که Syntax آن الهام گرفته از Ruby و Python است و بسیاری از ویژگی‌هایش، از این دو زبان پیاده سازی شده است.
سوالی که ممکن است برای هر کسی پیش بیاید این است که چرا باید از CoffeeScript استفاده کرد و یا چرا نوشتن CoffeeScript بهتر از نوشتن مستقیم جاوااسکریپت است؟

از جمله دلایلی که می‌شود عنوان کرد:
حجم کد کمتری نوشته می‌شود (تجربه شخصی من: تقریبا کدنویسی شما به یک سوم تا نصف تبدیل می‌شود)، بسیار مختصر است و پیاده سازی aliases prototype و classes به سادگی و با حداقل کدنویسی انجام می‌گیرد.

CoffeeScript زیرمجموعه‌ای از جاوااسکریپت نیست، اگرچه از کتابخانه‌های خارجی جاوااسکریپت می‌توان در کدهای CoffeeScript استفاده کرد، اما برای اینکار باید کدهای مورد نیاز را به CoffeeScript تبدیل کرد تا از خطای زمان کامپایل جلوگیری شود.

پیش نیاز نوشتن کد به زبان CoffeeScript، شناخت جاوااسکریپت است تا بتوان خطاهای زمان اجرا را اصلاح کرد.

CoffeeScript محدودیتی در مرورگر ندارد و می‌توان در برنامه‌های جاوااسکریپتی تحت سرور مانند Node.js با کیفیت بالا نیز از آن استفاده کرد.
زمانی را که برای یادگیری CoffeeScript صرف می‌کنید در زمان نوشتن پروژه، نتیجه‌ی آن‌را متوجه خواهید شد.

راه اندازی اولیه
یکی از ساده‌ترین راه‌های نوشتن CoffeeScript استفاده از نسخه‌ی مرورگر این زبان است و برای اینکار باید وارد سایت CoffeeScript.Org شده و بر روی تب *Try CoffeeScript* کلیک کنید. این سایت از نسخه‌ی مرورگر CoffeeScript Compiler استفاده می‌کند و هر کدی CoffeeScript ایی که در پنل سمت چپ سایت بنویسید، تبدیل به جاوااسکریپت می‌شود و در پنل راست سایت، نمایش داده می‌شود.

همچنین می‌توانید با استفاده از پروژه‌ی js2coffee کدهای جاوااسکریپت را به کدهای CoffeeScript تبدیل کنید.

در صورتیکه بخواهید از نسخه‌ی درون مرورگری CoffeeScript Compiler استفاده کنید، باید یک تگ اسکریپت لینک به این اسکریپت و با اضافه کردن تگ اسکریپت با type coffeescript این کار را انجام دهید. برای نمونه:

```
<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>
```

بدیهی است که استفاده از چنین روشی برای تحویل پروژه به مشتری صحیح نیست چرا که به خاطر تفسیر کدهای CoffeeScript در زمان اجرا، سرعت اجرایی پایین خواهد بود. به جای این روش CoffeeScript پیشنهاد می‌کند که از compiler Node.js و تبدیل آن به فایل‌های coffeescript pre-process استفاده کنید.
برای نصب باید آخرین نسخه‌ی Node.js و (Node Package Manager) npm را نصب کرده باشید. برای نصب CoffeeScript با استفاده از npm از دستور زیر استفاده کنید.

```
npm install -g coffee-script
```

پس از نصب می‌توانید با استفاده از دستور coffee فایل‌های CoffeeScript خود را (بدون پارامتر) اجرا کنید و در صورتیکه بخواهید خروجی جاوااسکریپت داشته باشید، از پارامتر *compile*-- استفاده کنید.

```
coffee --compile my-script.coffee
```

در صورتیکه پارامتر *output*-- تعریف نشود CoffeeScript فایل خروجی را هم نام با فایل اصلی قرار می‌دهد که در مثال بالا فایل خروجی می‌شود *my-script.js* . در صورتیکه فایلی از قبل موجود باشد، بازنویسی انجام می‌شود.

نظرات خوانندگان

نویسنده: محسن معتمدی
تاریخ: ۱۳۹۴/۰۴/۰۳ ۱۴:۵

با سلام
چند سوال داشتم
-1 ایا این امکان فراهم هست که در این زبان از jquery استفاده بشه؟استفاده از پلتفرم angularjs چطور؟ چگونه؟
-2 آیا میشود کاری کرد تا کامپایلر این زبان را به وسیله dotnet فراخوانی و استفاده کرد؟

نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۴/۰۳ ۱۴:۵۱

سلام و خدا قوت

در مورد سوال اول باید بگم که بله، هر کدی که در قالب جاوااسکریپت باشه رو می‌شه با CoffeeScript نوشت. مثلا برای استفاده از jQuery به این شکل باید عمل کنید.

```
$ ".className"
  .fadeOut "slow"
```

یا اگه بخواید به صورت یک خطی بنویسید

```
$(".className").fadeOut "slow"
```

در مورد سوال دومتون باید بگم که با استفاده از افزونه Web Essentials این کارو انجام بدید که هر وقت فایل CoffeeScript رو ذخیره کنید به صورت خودکار کار کامپایل اون انجام می‌شه.

عنوان: **CoffeeScript #2**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۳/۲۷ ۱۹:۵۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

Syntax

برای کار با CoffeeScript، ابتدا باید با ساختار Syntax آن آشنا شد. CoffeeScript در بسیاری از موارد با جاوااسکریپت یکسان است در حالیکه در قسمت قبل گفته شد که CoffeeScript زیر مجموعه‌ای جاوااسکریپت نیست؛ بنابراین برخی از کلمات کلیدی مانند *function* و *var* در آن مجاز نیست و سبب بروز خطا در زمان کامپایل می‌شوند. وقتی شما شروع به نوشتن فایل CoffeeScript می‌کنید، باید تمام کدهایی را که می‌نویسید، با Syntax کامل CoffeeScript بنویسید و نمی‌توانید قسمتی را با جاوااسکریپت و قسمتی را با CoffeeScript بنویسید.

برای نوشتن توضیحات در فایل CoffeeScript باید از علامت # استفاده کنید که این قسمت را از زبان Ruby گرفته است.

```
# A comment
```

در صورتیکه نیاز به نوشتن توضیحات را در چندین خط داشته باشید نیز این امکان دیده شده است:

```
###
  A multiline comment
###
```

نکته: تفاوتی که در توضیح یک خطی و چند خطی وجود دارد این است که توضیحات چند خطی پس از کامپایل، در فایل جاوااسکریپت خروجی نوشته می‌شوند، ولی توضیحات یک خطی در فایل خروجی تولید می‌شود.

در زبان CoffeeScript فاصله (space) بسیار مهم است؛ چرا که زبان Python براساس میزان تو رفتگی کدها، بدنه‌ی شرط‌ها و حلقه‌ها را تشخیص می‌دهد و CoffeeScript نیز از این ویژگی استفاده می‌کند. هرگاه بخواهید از {} استفاده کنید فقط کافی است از کلید Tab استفاده کنید تا پس از کامپایل به صورت {} تبدیل شود.

Variables & Scope

CoffeeScript یکی از باگهایی را که در نوشتن جاوااسکریپت وجود دارد (متغیرهای سراسری) حل کرده است. در جاوااسکریپت درصورتیکه هنگام تعریف متغیری از کلمه‌ی کلیدی *var* در پشت اسم متغیر استفاده نشود، به صورت سراسری تعریف می‌شود. CoffeeScript به سادگی متغیرهای سراسری را حذف می‌کند. در پشت صحنه‌ی این حذف، اسکریپت نوشته شده را درون یک تابع بدون نام قرار می‌دهد و با این کار تمامی متغیرها در ناحیه‌ی محلی قرار می‌گیرند و سپس قبل از نام هر متغیری، کلمه‌ی کلیدی *var* را قرار می‌دهد. برای مثال:

```
myVariable = "vahid"
```

که نتیجه کامپایل آن می‌شود:

```
var myVariable;
myVariable = "vahid";
```

همان طور که مشاهده می‌کنید، متغیر تعریف شده به صورت محلی تعریف شده و با این روش تعریف متغیر سراسری را به صورت اشتباهی، غیرممکن می‌کند. این روش استفاده شده در CoffeeScript جلوی بسیاری از اشتباهات معمول توسعه دهندگان وب را می‌گیرد.
با این حال گاهی اوقات نیاز است که متغیر سراسری تعریف کنید. برای اینکار باید از شیء سراسری موجود در مرورگر (window)

یا از روش زیر استفاده کنید:

```
exports = this
exports.MyVariable = "vahid"
```

Functions

CoffeeScript برای راحتی در نوشتن توابع، کلمه کلیدی function را حذف کرده و به جای آن از <- استفاده می‌کند. توابع در CoffeeScript می‌توانند در یک خط یا به صورت تورفته در چندین خط نوشته شده باشند. آخرین عبارتی که در یک تابع نوشته می‌شود به صورت ضمنی بازگشت داده می‌شود. در صورتیکه نیاز به بازگرداندن مقداری در تابع ندارید، از کلمه‌ی *return* به تنهایی استفاده کنید.

```
func = -> "vahid"
```

نتیجه‌ی کامپایل آن می‌شود:

```
var func;
func = function() {
  return "vahid";
};
```

همان طور که در بالا گفته شده، در صورتیکه بخواهید تابعی با چندین خط دستور داشته باشید، باید ساختار تو رفتگی را حفظ کرد. برای مثال:

```
func = ->
  # An extra line
  "vahid"
```

نتیجه کامپایل کد بالا نیز همانند کد قبلی می‌باشد.

Function arguments

برای تعریف آرگومان در توابع باید قبل از <- از () استفاده کرد و آرگومان هایی را که نیاز است، در داخل آن تعریف کرد. برای مثال:

```
func = (a, b) -> a * b
```

نتیجه‌ی کامپایل آن می‌شود:

```
var func;
func = function(a, b) {
  return a * b;
};
```

CoffeeScript از مقدار پیش فرض برای آرگومان‌های توابع نیز پشتیبانی می‌کند:

```
func = (a = 1, b = 2) -> a * b
```

همچنین در صورتیکه تعداد آرگومان‌های یک تابع برای شما مشخص نبود، می‌توانید از " ... " استفاده کنید. مثلا وقتی می‌خواهید جمع n عدد را بدست آورید که n عدد به صورت آرگومان به تابع ارسال می‌شوند:

```
sum = (nums...) ->
  result = 0
```

```
  nums.forEach (n) -> result += n
  result
```

در مثال فوق آرگومان nums آرایه‌ای از تمام آرگومان‌های ارسال شده به تابع است و نتیجه‌ی کامپایل آن می‌شود:

```
var sum,
  slice = [].slice;

sum = function() {
  var nums, result;
  nums = 1 <= arguments.length ? slice.call(arguments, 0) : [];
  result = 0;
  nums.forEach(function(n) {
    return result += n;
  });
  return result;
};
```

فراخوانی توابع

برای فراخوانی توابع می‌توانید به مانند جاوااسکریپت از با پرانتز () یا ()apply و یا ()call صدا زده شوند. اگرچه مانند Ruby، کامپایلر CoffeeScript می‌توانند به صورت اتوماتیک توابعی با حداقل یک آرگومان را فراخوانی کند.

```
a = "Vahid!"

alert a
# برابر است با
alert(a)

alert inspect a
# برابر است با
alert(inspect(a))
```

اگرچه استفاده از پرانتز اختیاری است اما توصیه می‌شود در مواقعی که آرگومان‌های ارسالی بیش از یک مورد باشد توصیه می‌شود از پرانتز استفاده کنید.
در صورتی که تابعی بدون آرگومان باشد، برای فراخوانی آن بدون نوشتن پرانتز بعد از نام تابع، CoffeeScript نمی‌تواند تشخیص دهند که این یک تابع است و مانند یک متغیر با آن برخورد می‌کند. دراین رابطه، رفتار CoffeeScript بسیار شبیه به Python می‌باشد.

عنوان: **CoffeeScript #3**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۳/۲۸ ۲۱:۴۰
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

Syntax

Object & Array

برای تعریف Object در CoffeeScript می‌توان دقیقا مانند جاوااسکریپت عمل کرد؛ با یک جفت براکت و ساختار کلید / مقدار. البته همانند تابع، نوشتن براکت اختیاری است. در واقع، شما می‌توانید از تورفتگی و هر کلید/مقدار، در خط جدید به جای کاما استفاده کنید:

```
object1 = {one: 1, two: 2}

# Without braces
object2 = one: 1, two: 2

# Using new lines instead of commas
object3 =
  one: 1
  two: 2

User.create(name: "Vahid Mohammad Taheri")
```

به همین ترتیب، برای تعریف آرایه‌ها می‌توانید از کاما به عنوان جدا کننده و یا هر مقدار آرایه را در یک خط جدید وارد کنید؛ هر چند براکت [] هنوز هم مورد نیاز است.

```
array1 = [1, 2, 3]

array2 = [
  1
  2
  3
]

array3 = [1,2,3,]
```

Flow control

طبق قاعده‌ای که برای نوشتن پرانتز در قبل گفته شد (پرانتز اختیاری است)، در دستورات if و else نیز چنین است:

```
if true == true
  "We're ok"

if true != true then "Vahid"

# برابر است با:
#  (1 > 0) ? "Yes" : "No!"
if 1 > 0 then "Yes" else "No!"
```

همانطوری که در مثال بالا مشاهده می‌کنید، در صورتی که از if در یک خط استفاده شود باید پس از شرط، کلمه کلیدی then را بنویسید.
CoffeeScript از اپراتورهای شرطی (:?) پشتیبانی **نمی کند** و به جای آن از if / else استفاده کنید.
CoffeeScript نیز همانند Ruby امکان نوشتن بدنه شرط را به صورت پسوندی ایجاد کرده است.

```
alert "It's cold!" if 1 < 5
```

به جای استفاده از علامت ! برای منفی سازی شرط، می‌توانید از کلمه‌ی کلیدی *not* استفاده کنید که سبب خوانایی بیشتر کد نوشته شده می‌شود:

```
if not true then "Vahid"
```

CoffeeScript امکان نوشتن خلاصه‌تر if not را نیز ایجاد کرده است؛ برای این کار از کلمه‌ی کلیدی *unless* استفاده کنید. معادل مثال بالا:

```
unless true
  "Vahid"
```

همانند not که برای خوانایی بالاتر کد به کار می‌رود، CoffeeScript کلمه کلیدی is را مطرح کرده‌است که پس از کامپایل به === ترجمه می‌شود.

```
if true is 1
  "OK!"
```

برای نوشتن ==! نیز می‌توان از is not استفاده کرد، که شکل خلاصه‌تر آن *isnt* است.

```
if true isnt true
  alert "OK!"
```

همانطوری که در بالا گفته شد، CoffeeScript عملگر == را به === و =! به ==! تبدیل می‌کند. دلیلی که CoffeeScript این عمل را انجام می‌دهد این است که جاوااسکریپت عمل مقایسه را بر روی نوع و سپس مقدار آن انجام می‌دهد و سبب پیشگیری از باگ در کد نوشته شده می‌شود.

الحاق رشته ها CoffeeScript امکان الحاق رشته‌ها را با استفاده از روش الحاق رشته‌ها در Ruby فراهم کرده است. برای انجام این عمل از #{} در داخل " " استفاده کنید که در داخل براکت می‌توانید از دستورات مختلف استفاده کنید. برای مثال:

```
favorite_color = "Blue. No, yel..."
question = "Sam: What... is your favorite color?
            Ben: #{favorite_color}
            Sam: Wrong!
            "
```

نتیجه‌ی کامپایل کد بالا می‌شود:

```
var favorite_color, question;
favorite_color = "Blue. No, yel...";
question = "Sam: What... is your favorite color?           Ben: " + favorite_color + "           Sam:
Wrong!";
```

عنوان: **CoffeeScript #4**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۳/۳۱ ۱۴:۳۰
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

Syntax

Loops

```
for name in ["Vahid", "Hamid", "Saeid"]
  alert "Hi #{name}"
```

نتیجه‌ی کامپایل کد بالا می‌شود:

```
var i, len, name, ref;

ref = ["Vahid", "Hamid", "Saeid"];
for (i = 0, len = ref.length; i < len; i++) {
  name = ref[i];
  alert("Hi " + name);
}
```

درصورتیکه نیاز به شمارنده‌ی حلقه داشته باشید، کافیست یک آرگومان اضافه را ارسال کنید. برای نمونه:

```
for name, i in ["Vahid", "Hamid", "Saeid"]
  alert "#{i} - Hi #{name}"
```

همچنین می‌توانید حلقه را به صورت یک خطی نیز بنویسید:

```
alert name for name in ["Vahid", "Hamid", "Saeid"]
```

همچنین مانند Python نیز می‌توانید از فیلتر کردن در حلقه، استفاده کنید.

```
names = ["Vahid", "Hamid", "Saeid"]
alert name for name in names when name[0] is "V"
```

و نتیجه کامپایل کد بالا می‌شود:

```
var i, len, name, names;

names = ["Vahid", "Hamid", "Saeid"];

for (i = 0, len = names.length; i < len; i++) {
  name = names[i];
  if (name[0] === "V") {
    alert(name);
  }
}
```

شما می‌توانید حلقه را برای یک object نیز استفاده کنید. به جای استفاده از کلمه‌ی کلیدی *in* ، از کلمه کلیدی *of* استفاده کنید.

```
names = "Vahid": "Mohammad Taheri", "Ali": "Ahmadi"
alert("#{first} #{last}") for first, last of names
```

پس از کامپایل نتیجه می‌شود:

```
var first, last, names;

names = {
  "Vahid": "Mohammad Taheri",
  "Ali": "Ahmadi"
};

for (first in names) {
  last = names[first];
  alert(first + " " + last);
}
```

حلقه while در CoffeeScript به مانند جاوااسکریپت عمل می‌کند؛ ولی مزیتی نیز به آن اضافه شده است که آرایه‌ای از نتایج را بر می‌گرداند. به عنوان مثال مانند تابع Array.prototype.map() .

```
num = 6
minstrel = while num -= 1
  num + " Hi"
```

نتیجه‌ی کامپایل آن می‌شود:

```
var minstrel, num;
num = 6;
minstrel = (function() {
  var _results;
  _results = [];
  while (num -= 1) {
    _results.push(num + " Hi");
  }
  return _results;
})();
```

CoffeeScript **Arrays** با الهام گرفتن از Ruby، به وسیله تعیین محدوده، آرایه را ایجاد می‌کند. محدوده آرایه به وسیله دو عدد تعیین می‌شوند که با .. یا ... از هم جدا می‌شوند.

```
range = [1..5]
```

نتیجه‌ی کامپایل می‌شود:

```
var range;
range = [1, 2, 3, 4, 5];
```

در صورتی که محدوده‌ی آرایه **بلافاصله** بعد از یک متغیر بیاید CoffeeScript، کد نوشته شده را به تابع slice() تبدیل می‌کند.

```
firstTwo = ["one", "two", "three"][0..1]
```

نتیجه کامپایل می‌شود:

```
var firstTwo;
firstTwo = ["one", "two", "three"].slice(0, 2);
```

در مثال بالا محدوده تعیین شده سبب می‌شود که یک آرایه جدید با دو عنصر "one" و "two" ایجاد شود. همچنین می‌توانید برای جایگزینی مقادیر جدید، در یک آرایه از قبل تعریف شده نیز از روش زیر استفاده کنید.

```
numbers = [0..9]
numbers[3..5] = [-3, -4, -5]
```

نکته: در صورتیکه متغیری قبل از تعریف محدوده آرایه قرار گیرد، اگر رشته باشد، نتیجه‌ی خروجی، آرایه‌ای از کاراکترهای آن می‌شود.

```
my = "my string"[0..2]
```

چک کردن وجود یک مقدار در آرایه، یکی از مشکلاتی است که در جاوااسکریپت وجود دارد (عدم پشتیبانی از ()indexOf در IE کمتر از 9). CoffeeScript با استفاده از کلمه‌ی کلیدی *in* این مشکل را برطرف کرده است.

```
words = ["Vahid", "Hamid", "Saeid", "Ali"]
alert "Stop" if "Hamid" in words
```

نکات مهم

در صورت تعریف محدوده آرایه به صورت numbers[3..] (که آرایه numbers از قبل تعریف شده باشد)، خروجی، آرایه‌ای از مقادیر موجود در numbers را از خانه شماره 4 تا انتهای آن برمی گرداند.
در صورت تعریف محدوده آرایه به صورت numbers[-3..] (که آرایه numbers از قبل تعریف شده باشد)، خروجی، آرایه‌ای از مقادیر موجود در numbers را از خانه انتهایی به میزان 3 خانه به سمت ابتدای آرایه برمیگرداند.
در صورت عدم تعریف محدوده آرایه و فقط استفاده از [..] یا [...] (یک شکل عمل می‌کنند)، کل مقادیر آرایه اصلی (که از قبل تعریف شده باشد)، برگردانده می‌شود.

تفاوت .. و ... در حالتی که دو عدد برای محدوده تعریف شود، در این است که ... آرایه به صورت عدد انتهایی - 1 تعریف می‌شود. مثلا [0...3] یعنی خانه‌های آرایه از 0 تا 2 را به عنوان خروجی برگردان.

CoffeeScript **Aliases** شامل یک سری نام‌های مستعار است که برای خلاصه نویسی بیشتر بسیار مفید هستند. یکی از آن نام ها، @ است که به جای نوشتن *this* به کار می‌رود.

```
@name = "Vahid"
```

نتیجه کامپایل آن می‌شود:

```
this.name = "Vahid";
```

یکی دیگر از این نام ها، :: می‌باشد که به جای نوشتن prototype به کار می‌رود.

```
User::first = -> @records[0]
```

نتیجه کامپایل آن می‌شود:

```
User.prototype.first = function() {
  return this.records[0];
};
```

یکی از عمومی‌ترین شرط هایی که در جاوااسکریپت استفاده می‌شود، شرط چک کردن not null است. CoffeeScript این کار را با استفاده از ? انجام می‌دهد و در صورتی که متغیر برابر با null یا undefined نباشد، مقدار true را برمی گرداند. این ویژگی همانند ?nil در Ruby است.

```
alert "OK" if name?
```

نتیجه‌ی کامپایل آن می‌شود:

```
if (typeof name !== "undefined" && name !== null) {
  alert("OK");
}
```

از ? به جای || نیز می‌توانید استفاده کنید.

```
name = myName ? "-"
```

نتیجه‌ی کامپایل آن می‌شود:

```
var name;

name = typeof myName !== "undefined" && myName !== null ? myName : "-";
```

در صورتیکه بخواهید به یک property از یک شیء دسترسی داشته باشید و بخواهید null نبودن آن را قبل از دسترسی به آن چک کنید، می‌توانید از ? استفاده کنید.

```
user.getAddress()?.getStreetName()
```

نتیجه‌ی کامپایل آن می‌شود:

```
var ref;

if ((ref = user.getAddress()) != null) {
  ref.getStreetName();
}
```

همچنین در صورتیکه بخواهید چک کنید یک property در واقع یک تابع است یا نه (مثلا برای مواقعی که می‌خواهید callback بسازید) و سپس آن را فراخوانی کنید، نیز از ? می‌توانید استفاده کنید.

```
user.getAddress().getStreetName?()
```

و نتیجه‌ی کامپایل آن می‌شود:

```
var base;

if (typeof (base = user.getAddress()).getStreetName === "function") {
  base.getStreetName();
}
```

عنوان: **CoffeeScript #5**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۳/۳۱ ۱۶:۵۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

Classes

کلاس نه تنها در جاوااسکریپت بلکه در سایر زبان‌ها از جمله CoffeeScript نیز، بسیار مفید است.

در پشت صحنه، CoffeeScript برای ایجاد کلاس از prototype استفاده می‌کند. برای ساختن کلاس در CoffeeScript از کلمه کلیدی *class* باید استفاده کنید.

```
class Animal
```

نتیجه‌ی کامپایل مثال بالا می‌شود:

```
var Animal;
Animal = (function() {
  function Animal() {}
  return Animal;
})();
```

در مثال بالا، Animal نام کلاس و همچنین نامی است که برای ایجاد یک نمونه از آن می‌توانید استفاده کنید. در پشت صحنه CoffeeScript با استفاده از سازنده توابع این کار را انجام می‌دهد. یعنی شما می‌توانید با استفاده از کلمه کلیدی *new* یک نمونه از کلاس نوشته شده را بسازید.

```
animal = new Animal
```

تعریف سازنده برای کلاس بسیار ساده است. فقط کافی است از کلمه کلیدی *constructor* به عنوان یک تابع در کلاس تعریف شده استفاده کنید. این تابع شبیه به initialize در Ruby و __init__ در Python است.

```
class Animal
  constructor: (name) ->
    @name = name
```

نتیجه کامپایل کد بالا می‌شود:

```
var Animal;
Animal = (function() {
  function Animal(name) {
    this.name = name;
  }
  return Animal;
})();
```

همچنین CoffeeScript امکان خلاصه نویسی را در سازنده کلاس نیز ایجاد کرده است. برای اینکار با اضافه کردن @ به آرگومان‌های تابع سازنده به صورت پیشوند این کار انجام می‌شود. مثال زیر معادل مثال قبل است که به صورت دستی مقدار دهی انجام شده است.

```
class Animal
  constructor: (@name) ->
```

و برای استفاده از این کلاس

```
animal = new Animal "Cat"
alert "Animal is a #{animal.name}"
```

Instance properties اضافه کردن property به یک کلاس بسیار ساده و راحت است، syntax این کار دقیقا مانند اضافه کردن property به یک object است. فقط نکته ای که باید رعایت شود میزان تو رفتگی property نوشته شده است که به طور صحیح در داخل بدنه کلاس قرار بگیرد.

```
class Animal
  price: 5

  sell: (customer) ->

animal = new Animal
animal.sell(new Customer)
```

نتیجه کامپایل کد بالا می‌شود:

```
var Animal, animal;

Animal = (function() {
  function Animal() {}

  Animal.prototype.price = 5;

  Animal.prototype.sell = function(customer) {};

  return Animal;

})();

animal = new Animal;
animal.sell(new Customer);
```

Static properties

برای تعریف property به صورت static باید کلمه کلیدی this را به ابتدای آن اضافه کنید.

```
class Animal
  this.find = (name) ->

Animal.find("Dog")
```

در قسمت‌های قبل گفته شد، که به جای this می‌توان از @ استفاده کرد، در اینجا نیز می‌توان چنین کاری را انجام داد.

```
class Animal
  @find: (name) ->

Animal.find("Dog")
```

نتیجه‌ی کامپایل آن می‌شود:

```
var Animal;

Animal = (function() {
  function Animal() {}

  Animal.find = function(name) {};

  return Animal;

})();
```

CoffeeScript #5

```
Animal.find("Dog");
```

عنوان: **CoffeeScript #6**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۴/۰۲ ۲۰:۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

Classes

Inheritance & Super

شما می‌توانید به راحتی از کلاس‌های دیگری که نوشته‌اید، با استفاده از کلمه‌ی کلیدی extends، ارث بری کنید:

```
class Animal
  constructor: (@name) ->

  alive: ->
    true

class Parrot extends Animal
  constructor: ->
    super("Parrot")

  dead: ->
    not @alive()
```

در مثال بالا، Parrot (طوطی) از کلاس Animal ارث بری شده، که تمام خصوصیات آن را مانند ()alive، ارث برده است.
همانطوری که در مثال بالا مشاهده می‌کنید، در کلاس Parrot در تابع constructor، تابع super فراخوانی شده است. با استفاده از کلمه‌ی کلیدی super می‌توان تابع سازنده‌ی کلاس پدر را فراخوانی کرد. نتیجه‌ی کامپایل super در مثال بالا به این صورت می‌شود:

```
Parrot.__super__.constructor.call(this, "Parrot");
```

تابع super در CoffeeScript دقیقا مانند Ruby و Python عمل می‌کند.
در صورتیکه تابع constructor را در کلاس فرزند ننوشته باشید، به طور پیش فرض CoffeeScript سازنده کلاس پدر را فراخوانی می‌کند.

CoffeeScript با استفاده از prototypal inheritance، به صورت خودکار تمامی خصوصیات کلاس پدر، به فرزندان انتقال پیدا می‌کند. این ویژگی سبب داشتن کلاس‌های پویا می‌شود. برای درک بهتر این موضوع، فرض کنید که خصوصیتی را به کلاس پدر بعد از ارث بری کلاس فرزند اضافه می‌کنید. خصوصیت اضافه شده به تمامی فرزندان کلاس پدر به صورت خودکار اضافه می‌شود.

```
class Animal
  constructor: (@name) ->

class Parrot extends Animal

Animal::rip = true

parrot = new Parrot("Macaw")
alert("This parrot is no more") if parrot.rip
```

Mixins **Mixins** توسط CoffeeScript پشتیبانی نمی‌شود و برای همین نیاز است که این قابلیت را برای خودمان پیاده سازی کنیم، به مثال زیر توجه کنید.

```
extend = (obj, mixin) ->
  obj[name] = method for name, method of mixin
  obj
```

```
include = (klass, mixin) ->
  extend klass.prototype, mixin

# Usage
include Parrot,
  isDeceased: true

alert (new Parrot).isDeceased
```

نتیجه کامپایل آن می‌شود:

```
var extend, include;
extend = function(obj, mixin) {
  var method, name;
  for (name in mixin) {
    method = mixin[name];
    obj[name] = method;
  }
  return obj;
};
include = function(klass, mixin) {
  return extend(klass.prototype, mixin);
};
include(Parrot, {
  isDeceased: true
});
alert((new Parrot).isDeceased);
```

Mixins یک الگوی عالی برای به اشتراک گذاشتن خصوصیت‌های مشترک، در بین کلاس‌هایی است که امکان ارث بری در آنها وجود ندارد. مهمترین مزیت استفاده از Mixins این است که می‌توان چندین خصوصیت را به یک کلاس اضافه کرد؛ در حالیکه برای ارث بری فقط از یک کلاس می‌توان ارث بری داشت.

Extending classes

Mixins خیلی مرتب و خوب است اما خیلی شیء گرا نیست؛ در عوض امکان ادغام را در کلاس‌های CoffeeScript ایجاد می‌کند. برای اینکه اصول شیء گرایی را بخواهیم رعایت کنیم و ویژگی ادغام را نیز داشته باشیم، کلاسی با نام Module را پیاده سازی می‌کنیم و تمامی کلاس‌هایی را که می‌خواهیم ویژگی ادغام را داشته باشند، از آن ارث بری می‌کنیم.

```
moduleKeywords = ['extended', 'included']

class Module
  @extend: (obj) ->
    for key, value of obj when key not in moduleKeywords
      @[key] = value

    obj.extended?.apply(@)
    this

  @include: (obj) ->
    for key, value of obj when key not in moduleKeywords
      # Assign properties to the prototype
      @::[key] = value

    obj.included?.apply(@)
    this
```

برای استفاده از کلاس Module به مثال زیر توجه کنید:

```
classProperties =
  find: (id) ->
  create: (attrs) ->

instanceProperties =
  save: ->

class User extends Module
```

```
  @extend classProperties
  @include instanceProperties

# Usage:
user = User.find(1)

user = new User
user.save()
```

همانطور که مشاهده می‌کنید دو خصوصیت ثابت (static property)، را به کلاس User اضافه کردیم (find, create) و خصوصیت save.

همچنین برای خلاصه نویسی بیشتر می‌توان از این الگو استفاده کرد (ساده و زیبا).

```
ORM =
  find: (id) ->
  create: (attrs) ->
  extended: ->
    @include
      save: ->

class User extends Module
  @extend ORM
```

عنوان: **CoffeeScript #7**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۴/۰۴ ۱۹:۴۰
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

اصطلاحات عمومی CoffeeScript

هر زبانی دارای مجموعه‌ای از اصطلاحات و روش هاست. CoffeeScript نیز از این قاعده مستثنی نیست. در این قسمت می‌خواهیم مقایسه‌ای بین جاوااسکریپت و CoffeeScript انجام دهیم تا به وسیله‌ی این مقایسه، مفهوم عملی این زبان را درک کنید.

Each

در جاوااسکریپت وقتی می‌خواهیم بر روی آرایه‌ای با بیش از یک خانه، کاری را چندین بار انجام دهیم، می‌توانیم از تابع forEach() یا از همان قالب حلقه‌ی for در زبان C استفاده کنیم:

```
for (var i=0; i < array.length; i++)
  myFunction(array[i]);

array.forEach(function(item, i){
  myFunction(item)
});
```

اگرچه تابع forEach() مختصر و خواناتر است ولی یک مشکل دارد؛ به دلیل فراخوانی تابع callback در هر بار اجرای حلقه، بسیار کندتر از حلقه for اجرا می‌شود.
حال به نحوه‌ی کارکرد CoffeeScript دقت کنید.

```
myFunction(item) for item in array
```

که پس از کامپایل می‌شود:

```
var i, item, len;

for (i = 0, len = array.length; i < len; i++) {
  item = array[i];
  myFunction(item);
}
```

همانطوری که مشاهده می‌کنید، از نظر syntax بسیار ساده و با خوانایی بالا است و مطمئن هستم شما هم با من موافق هستید و نکته‌ی مهمی که وجود دارد، کامپایل حلقه‌ی با ظاهر forEach به حلقه‌ی for، توسط CoffeeScript و حفظ سرعت اجرای آن است.

Map

همانند تابع forEach که در استاندارد ES5 قرار داشت، تابع دیگری به نام map() وجود دارد که از نظر syntax بسیار خلاصه‌تر از حلقه‌ی for می‌باشد. ولی متاسفانه همانند تابع forEach، این تابع نیز به دلیل فراخوانی تابع، بسیار کندتر از for اجرا می‌شود.

```
var result = []
for (var i=0; i < array.length; i++)
  result.push(array[i].name)

var result = array.map(function(item, i){
  return item.name;
});
```

همانطور که مشاهده می‌کنید در اینجا طریقه‌ی استفاده از تابع map و پیاده سازی آن بدون استفاده از تابع map نشان داده شده

است. حال به مثال زیر توجه کنید:

```
result = (item.name for item in array)
```

با استفاده از ساختار حلقه‌ها که در قسمت 4 گفتیم و تنها با قراردادن () در اطراف آن می‌توان تابع map را به راحتی پیاده سازی کرد.
نتیجه‌ی کامپایل مثال بالا می‌شود:

```
var item, result;

result = (function() {
  var i, len, results;
  results = [];
  for (i = 0, len = array.length; i < len; i++) {
    item = array[i];
    results.push(item.name);
  }
  return results;
})();
```

Select

یکی دیگر از توابع ES5، تابع filter() است که برای کاهش خانه‌های آرایه استفاده می‌شود.

```
var result = []
for (var i=0; i < array.length; i++)
  if (array[i].name == "test")
    result.push(array[i])

result = array.filter(function(item, i){
  return item.name == "test"
});
```

CoffeeScript با استفاده از کلمه‌ی کلیدی when، عمل فیلتر کردن آیتم‌هایی را که نمی‌خواهیم در آرایه باشند، انجام می‌دهد و در پشت صحنه، با استفاده از یک حلقه‌ی for این عمل را انجام می‌دهد.

```
result = (item for item in array when item.name is "test")
```

در اینجا نیز همانند تابع map برای جلوگیری از تداخل متغیرها از یک تابع بی‌نام استفاده می‌کند.

```
var item, result;

result = (function() {
  var i, len, results;
  results = [];
  for (i = 0, len = array.length; i < len; i++) {
    item = array[i];
    if (item.name === "test") {
      results.push(item);
    }
  }
  return results;
})();
```

نکته‌ی مهم: در صورت فراموشی اضافه کردن () در اطراف حلقه‌ی نوشته شده، نتیجه‌ی صحیحی تولید نخواهد شد و نتها آخرین عضو از خروجی را باز می‌گرداند.

```
var i, item, len, result;

for (i = 0, len = array.length; i < len; i++) {
  item = array[i];
  if (item.name === "test") {
```

```
    result = item;
  }
}
```

قوه‌ی درک CoffeeScript بسیار بالا و انعطاف پذیر است، به مثال زیر توجه کنید:

```
passed = []
failed = []
(if score > 60 then passed else failed).push score for score in [49, 58, 76, 82, 88, 90]

# Or
passed = (score for score in scores when score > 60)
```

و یا در صورتیکه طول خط نوشته شده زیاد باشد می‌توانید به صورت چند خطی آن را بنویسید:

```
passed = []
failed = []
for score in [49, 58, 76, 82, 88, 90]
  (if score > 60 then passed else failed).push score
```

و نتیجه‌ی کامپایل مثال آخر می‌شود:

```
var failed, i, len, passed, ref, score;

passed = [];

failed = [];

ref = [49, 58, 76, 82, 88, 90];
for (i = 0, len = ref.length; i < len; i++) {
  score = ref[i];
  (score > 60 ? passed : failed).push(score);
}
```

عنوان: CoffeeScript #8
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۴/۲۲ ۱۴:۴۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

اصطلاحات عمومی CoffeeScript

Includes

برای چک کردن وجود یک مقدار در یک آرایه به طور معمول از indexOf استفاده می‌شود؛ در حالی که تمامی نسخه‌های IE به طور کامل از آن پشتیبانی نمی‌کنند.

```
var included = (array.indexOf("test") != -1)
```

CoffeeScript برای حل این مشکل، کلمه‌ی کلیدی in را ارائه کرده است:

```
included = "test" in array
```

متاسفانه برای چک کردن یک کلمه در یک متن می‌بایست از indexOf استفاده کرد و از کلمه‌ی کلیدی in نمی‌توان استفاده کرد. همچنین در صورتیکه بخواهید نبود چیزی را چک کنید نیز باید از indexOf استفاده کنید.

```
included = "a long test string".indexOf("test") isnt -1
```

و روش بهتر بجای مقایسه با مقدار 1-، استفاده از هک‌های اپراتور بیتی است:

```
string   = "a long test string"
included = !~ string.indexOf "test"
```

تکرار Propertyها

در صورتی که به خصوصیات یک شیء چندین بار نیاز داشته باشید، در جاوااسکریپت باید از کلمه‌ی کلیدی in استفاده کنید:

```
var object = {one: 1, two: 2}
for(var key in object) alert(key + " = " + object[key])
```

در حالیکه برای پیاده سازی توسط CoffeeScript باید از کلمه‌ی کلیدی of استفاده کرد:

```
object = {one: 1, two: 2}
alert("#{key} = #{value}") for key, value of object
```

نتیجه‌ی کامپایل آن می‌شود:

```
var key, object, value;

object = {
  one: 1,
  two: 2
};

for (key in object) {
  value = object[key];
  alert(key + " = " + value);
}
```

همانطور که در مثال بالا مشاهده می‌کنید، شما می‌توانید برای دسترسی به کلید و مقدار خصوصیات موجود در شیء، متغیری را برای هر کدام تعریف کنید که ما در اینجا از key و value استفاده کرده‌ایم.

Min/Max

درست است که این تکنیک مخصوص CoffeeScript نیست، اما اشاره به آن می‌تواند مفید باشد. تابع Math.max و Max.min می‌توانند چندین آرگومان یا یک آرایه را به عنوان ورودی گرفته و بر روی آن محاسبات خود را انجام داده و خروجی را نشان دهند:

```
Math.max [14, 35, -7, 46, 98]... # 98
Math.min [14, 35, -7, 46, 98]... # -7
```

نتیجه‌ی آن پس از کامپایل می‌شود:

```
Math.max.apply(Math, [14, 35, -7, 46, 98]);

Math.min.apply(Math, [14, 35, -7, 46, 98]);
```

نکته: در صورتیکه آرگومان‌ها یا تعداد خانه‌های آرایه ارسالی زیاد باشند، چون مرورگرها محدودیتی را در تعداد پارامترهای ارسالی به یک تابع دارند، خروجی تولید نخواهد شد.

عنوان: **CoffeeScript #9**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۴/۲۲ ۱۹:۵۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

اصطلاحات عمومی CoffeeScript

Multiple arguments

همانطوری که در قسمت قبل در تابع Math.max مشاهده کردید، با استفاده از ... آرایه را به عنوان آرگومان چندگانه به تابع max ارسال کردیم. در پشت صحنه CoffeeScript برای اطمینان از ارسال کامل آرایه به تابع max، برای فراخوانی از تابع apply() استفاده می‌کند. ما نیز می‌توانیم از این ویژگی در جای دیگری استفاده کنیم.

```
Log =
  log: ->
    console?.log(arguments...)
```

نتیجه‌ی کامپایل آن می‌شود:

```
var Log;

Log = {
  log: function() {
    return typeof console !== "undefined" && console !== null ? console.log.apply(console, arguments) :
void 0;
  }
};
```

و یا می‌توان قبل از ارسال آرگومان‌ها در آنها تغییر ایجاد کرد.

```
Log =
  logPrefix: "(App)"

  log: (args...) ->
    args.unshift(@logPrefix) if @logPrefix
    console?.log(args...)
```

نتیجه‌ی کامپایل آن می‌شود:

```
var Log,
  slice = [].slice;

Log = {
  logPrefix: "(App)",
  log: function() {
    var args;
    args = 1 <= arguments.length ? slice.call(arguments, 0) : [];
    if (this.logPrefix) {
      args.unshift(this.logPrefix);
    }
    return typeof console !== "undefined" && console !== null ? console.log.apply(console, args) : void
0;
  }
};
```

همانطور که مشاهده می‌کنید آرگومان‌های ارسالی به تابع log پس از چک شدن متغیر logPrefix و در صورت داشتن مقدار توسط تابع unshift به ابتدای آنها اضافه می‌شود.

And/Or

طبق ساختار syntax ایی که در قسمت‌های قبل با آن آشنا شدیم، **or** به جای **||** و **and** به جای **&&** استفاده شده و سبب خوانایی

بیشتر کد نوشته می‌شوند؛ در صورتیکه هر دو روش نتایج یکسانی را تولید می‌کنند.

همچنین به جای استفاده از **==** از **is** و برای **=!** از **isnt** استفاده می‌شود.

```
string = "migrating coconuts"
string == string # true
string is string # true
```

یکی از ویژگی‌های فوق العاده خوب که به CoffeeScript افزوده شده 'or equals' است که با الگو گرفتن از Ruby پیاده سازی شده است.

```
hash or= {}
```

نتیجه‌ی کامپایل آن می‌شود:

```
hash || (hash = {});
```

در اینجا در صورتیکه ارزیابی hash برابر false شود، مقدار آن برابر یک شیء خالی می‌شود. **نکته‌ی مهمی** که وجود دارد در صورتیکه hash مقداری برابر 0 ، "" و یا null داشته باشد، ارزیابی آن برابر false می‌شود. در صورتی که چنین قصدی ندارید باید از عملگرهای وجودی CoffeeScript استفاده کنید که تنها در حالیکه hash برابر null و یا undefined باشد، فعال می‌شوند.

```
hash ?= {}
```

نتیجه‌ی کامپایل آن می‌شود:

```
if (typeof hash !== "undefined" && hash !== null) {
  hash;
} else {
  hash = {};
};
```

عنوان: CoffeeScript #10
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۴/۲۲ ۲۰:۱۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

اصطلاحات عمومی CoffeeScript

Destructuring Assignments

با استفاده از Destructuring assignments می‌توانید خصوصیات را از آرایه‌ها یا اشیاء، با هر میزان عمقی استخراج کنید.

```
someObject = { a: 'value for a', b: 'value for b' }
{ a, b } = someObject
console.log "a is '#{a}', b is '#{b}'"
```

نتیجه‌ی کامپایل آن می‌شود:

```
var a, b, someObject;

someObject = {
  a: 'value for a',
  b: 'value for b'
};

a = someObject.a, b = someObject.b;

console.log("a is '" + a + "', b is '" + b + "'");
```

این موضوع به خصوص در برنامه‌های کاربردی، وقتی نیاز به ماژول‌های دیگر است، مفید خواهد بود.

```
{join, resolve} = require('path')

join('/Users', 'Vahid')
```

External libraries

استفاده از کتابخانه‌های خارجی دقیقا مانند فراخوانی توابع CoffeeScript است. در پایان نوشتن کدهای CoffeeScript، همه به جاوااسکریپت تبدیل می‌شوند:

```
# Use local alias
$ = jQuery

$ ->
  # DOMContentLoaded
  $(".el").click ->
    alert("Clicked!")
```

نتیجه‌ی کامپایل آن می‌شود:

```
var $;
$ = jQuery;
$(function() {
  return $(".el").click(function() {
    return alert("Clicked!");
  });
});
```

از آنجاییکه خروجی همه کدهای CoffeeScript در داخل یک تابع بدون نام قرار می‌گیرد، می‌توانیم از یک متغیر محلی به نام $ به عنوان نام مستعار jQuery استفاده کنیم. این کار به شما کمک می‌کند تا حتی وقتی که حالت jQuery.noConflict نیز فراخوانی

شده باشد، $ مجدد تعریف شده و اسکریپت ما به طور کامل اجرا شود.

Private variables

کلمه‌ی کلید **do** در CoffeeScript به ما اجازه می‌دهد تا توابع را مستقیما اجرا کنیم و این روش یک راه خوب برای کپسوله سازی و حفاظت از متغیرهاست. در مثال زیر متغیر classToType را در context یک تابع بدون نام که به وسیله‌ی do فراخوانی می‌شود، تعریف کرده‌ایم. تابع بدون نام دوم، مقدار نهایی از type است را برمی گرداند. از آنجایی که classToType در context تعریف شده است و هیچ ارجایی به آن نگهداری نمی‌شود، پس امکان دسترسی به آن خارج از این scope وجود ندارد.

```
# Execute function immediately
type = do ->
  classToType = {}
  for name in "Boolean Number String Function Array Date RegExp Undefined Null".split(" ")
    classToType["[object " + name + "]"] = name.toLowerCase()

  # Return a function
  (obj) ->
    strType = Object::toString.call(obj)
    classToType[strType] or "object"
```

نتیجه‌ی کامپایل آن می‌شود:

```
var type;

type = (function() {
  var classToType, i, len, name, ref;
  classToType = {};
  ref = "Boolean Number String Function Array Date RegExp Undefined Null".split(" ");
  for (i = 0, len = ref.length; i < len; i++) {
    name = ref[i];
    classToType["[object " + name + "]"] = name.toLowerCase();
  }
  return function(obj) {
    var strType;
    strType = Object.prototype.toString.call(obj);
    return classToType[strType] || "object";
  };
})();
```

به بیان دیگر classToType به طور کامل private است و امکان دسترسی به آن از طریق تابع بدون نام اجرا کننده وجود ندارد. این الگو راه بسیار خوب و مناسبی برای کپسوله سازی scope و مخفی سازی متغیرها است.

عنوان: **CoffeeScript #11**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۵/۰۳ ۱۰:۴۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

کامپایل خودکار CoffeeScript

همانطور که گفته شده CoffeeScript یک لایه میان شما و جاوااسکریپت است و هر زمان که فایل CoffeeScript تغییر کرد، باید به صورت دستی آن را کامپایل کرد. خوشبختانه CoffeeScript روش‌های دیگری را برای کامپایل کردن دارد که به وسیله آن می‌توان چرخه‌ی توسعه را بسیار ساده‌تر نمود.

در قسمت اول گفته شد، برای کامپایل فایل CoffeeScript با استفاده از *coffee* به صورت زیر عمل می‌کردیم:

```
coffee --compile --output lib src
```

همانطور که در مثال بالا مشاهده می‌کنید، تمامی فایل‌های *coffee*. در داخل پوشه *src* را کامپایل می‌کنید و فایل‌های جاوااسکریپت تولید شده را در پوشه lib ذخیره می‌کنید.
حال به کامپایل خودکار CoffeeScript توجه کنید.

Cake

Cake یک سیستم فوق العاده ساده برای کامپایل خودکار است که مانند Make و Rake عمل می‌کند. این کتابخانه همراه پکیج npm *coffee-script* نصب می‌شود و برای استفاده با فراخوانی *cake* اجرا می‌شود.

برای ایجاد فایل tasks در cake که Cakefile نامیده می‌شود، می‌توان از خود CoffeeScript استفاده کرد. برای اجرای cake با استفاده از دستور cake [options] [task] می‌توان عمل کرد. برای اطلاع از لیست امکانات cake کافی است دستور cake را به تنهایی اجرا کنید.

وظایف را می‌توان با استفاده از تابع task، با ارسال نام و توضیحات (اختیاری) و تابع callback، تعریف کرد. به مثال زیر توجه کنید:

```
fs = require 'fs'

{print} = require 'sys'
{spawn} = require 'child_process'

build = (callback) ->
  coffee = spawn 'coffee', ['-c', '-o', 'lib', 'src']
  coffee.stderr.on 'data', (data) ->
    process.stderr.write data.toString()
  coffee.stdout.on 'data', (data) ->
    print data.toString()
  coffee.on 'exit', (code) ->
    callback?() if code is 0

task 'build', 'Build lib/ from src/', ->
  build()
```

همانطور که در مثال بالا مشاهده می‌کنید، تابع task را با نام **build** تعریف کردیم و با استفاده از دستور cake build می‌توان آن را اجرا نمود. پس از اجرا همانند مثال قبل تمامی فایل‌های CoffeeScript در پوشه‌ی src به فایل‌های جاوااسکریپت در پوشه lib تبدیل می‌شوند.
همان طور که مشاهده می‌کنید پس از تغییر در فایل CoffeeScript باید به صورت دستی cake build را فراخوانی کنیم که این دور از حالت ایده آل است.
خوشبختانه دستور coffee پارامتر دیگری به نام watch-- دارد که به وسیله آن می‌توان تمامی تغییرات یک پوشه را زیر نظر گرفت

و در صورت نیاز دوباره کامپایل انجام شود. به مثال زیر توجه کنید:

```
task 'watch', 'Watch src/ for changes', ->
    coffee = spawn 'coffee', ['-w', '-c', '-o', 'lib', 'src']
    coffee.stderr.on 'data', (data) ->
      process.stderr.write data.toString()
    coffee.stdout.on 'data', (data) ->
      print data.toString()
```

در صورتی که task ایی وابسته به task دیگری باشد، می‌توانید برای اجرای taskهای دیگر از دستور *invoke(name)* استفاده کنید. برای مثال یک task را به فایل Cakefile اضافه می‌کنیم که در آن ابتدا فایل index.html را باز کرده و سپس شروع به زیر نظر گرفتن پوشه src می‌کنیم.

```
task 'open', 'Open index.html', ->
  # First open, then watch
  spawn 'open', 'index.html'
  invoke 'watch'
```

همچنین می‌توانید با استفاده از تابع ()option، options را برای taskها تعریف کنید.

```
option '-o', '--output [DIR]', 'output dir'

task 'build', 'Build lib/ from src/', ->
  # Now we have access to a `options` object
  coffee = spawn 'coffee', ['-c', '-o', options.output or 'lib', 'src']
  coffee.stderr.on 'data', (data) ->
    process.stderr.write data.toString()
  coffee.stdout.on 'data', (data) ->
    print data.toString()
```

Cake یک روش عالی برای انجام وظایف معمول به صورت خودکار است، مانند کامپایل فایل‌های CoffeeScript است. همچنین برای آشنایی بیشتر می‌توانید به سورس cake نگاهی کنید.

عنوان: CoffeeScript #12
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۵/۰۸ ۱۲:۳۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

بخش‌های بد

جاوااسکریپت یک زبان پیچیده است که شما برای کار با آن، نیاز است قسمت‌هایی را که **باید** از آن‌ها دوری کنید و قسمت‌های مهمی را که **باید** استفاده کنید، بشناسید. همانطور که Sun Tzu گفته "دشمن خود را بشناس"، ما نیز در این قسمت می‌خواهیم برای شناخت بیشتر قسمت‌های تاریک و روشن جاوااسکریپت به آن بپردازیم.

همانطور که در قسمت‌های قبل گفته شد، CoffeeScript تنها به یک syntax محدود نمی‌شود و توانایی برطرف کردن برخی از مشکلات جاوااسکریپت را نیز دارد. با این حال، با توجه به این واقعیت که کدهای CoffeeScript به صورت مستقیم به جاوااسکریپت تبدیل می‌شوند و نمی‌توانند تمامی مشکلاتی را که در جاوااسکریپت وجود دارند، حل کنند، پس برخی از مسائل وجود دارند که شما باید از آنها آگاهی داشته باشید.

اول از قسمت‌هایی که توسط CoffeeScript حل شده‌اند شروع می‌کنیم.

A JavaScript Subset

with یک دستور بسیار زمانبر است و **مضر** شناخته شده است و نباید از آن استفاده کنید. with با ایجاد یک ساختار خلاصه نویسی، برای جستجو بر روی خصوصیات اشیاء در نظر گرفته شده بود. برای نمونه به جای نوشتن:

```
dataObj.users.vahid.email = "info@vmt.ir";
```

می‌توانید به این صورت این کار را انجام دهید:

```
with(dataObj.users.vahid) {
  email = "info@vmt.ir";
}
```

مفسر جاوااسکریپت دقیقا نمی‌داند که شما می‌خواهید چه کاری را با with انجام دهید، و به شیء مشخص شده فشار می‌آورد تا اول اسم همه مراجعه شده‌ها را جستجو کند. این عمل واقعا به عملکرد و کارآیی لطمه می‌زند. یعنی مترجم، تمام انواع بهینه سازی‌های JIT را خاموش می‌کند. همچنین پیشنهادهایی مبنی بر حذف کامل آن از نسخه‌های بعدی جاوااسکریپت نیز مطرح شده است.
همه چیز برای عدم استفاده از with در نظر گرفته شده است. CoffeeScript یک قدم جلوتر از همه برداشته و with را از syntax خود حذف کرده است. به عبارت دیگر در صورتیکه شما از آن استفاده کنید، کامپایلر CoffeeScript خطا صادر می‌کند.

Global variables

به طور پیش فرض تمامی برنامه‌های جاوااسکریپت در دامنه global اجرا می‌شوند و تمامی متغیرهایی که ساخته می‌شوند به طور پیش فرض در ناحیه‌ی global قرار می‌گیرند. اگر شما بخواهید متغیری را در ناحیه‌ی local ایجاد کنید، باید از کلمه کلیدی *var* استفاده کنید.

```
usersCount = 1;         // Global
var groupsCount = 2;    // Global

(function(){
  pagesCount = 3;       // Global
  var postsCount = 4;   // Local
})()
```

اکثر اوقات شما می‌خواهید متغیر local ایی را ایجاد کنید و نه global. توسعه دهندگان باید همیشه به یاد داشته باشند که قبل از مقداردهی اولیه‌ی هر متغیری، کلمه‌ی کلیدی var را قرار دهند یا با انواع و اقسام مشکلات، هنگامی که متغیرها به طور تصادفی با یکدیگر برخورد و یا بازنویسی بر روی یکدیگر انجام می‌دهند، روبرو شوند.
خوشبختانه CoffeeScript به کمک شما می‌آید و به طور کامل انتساب متغیرهای global را به طور ضمنی از بین می‌برد. به عبارت دیگر کلمه کلیدی var در CoffeeScript رزرو شده است و در صورت استفاده خطا صادر می‌شود.
به صورت پیش فرض به طور ضمنی متغیرها local ایجاد می‌شوند و خیلی سخت می‌شود متغیر global ایی را بدون انتساب آن به عنوان خصوصیتی از شیء window ایجاد کرد.

```
outerScope = true
do ->
  innerScope = true
```

نتیجه‌ی کامپایل آن می‌شود:

```
var outerScope;
outerScope = true;
(function() {
  var innerScope;
  return innerScope = true;
})();
```

همانطور که مشاهده می‌کنید CoffeeScript مقداردهی اولیه متغیر را (با استفاده از var) به صورت خودکار در context ایی که برای اولین بار استفاده شده است انجام می‌دهد. باید مواظب باشید تا از نام متغیر خارجی مجددا استفاده نکنید که این اتفاق ممکن است در کلاس یا تابع با عمق زیاد ایجاد شود. برای مثال، در اینجا به صورت تصادفی متغیر package در یک تابع کلاس بازنویسی شده است:

```
package = require('./package')

class Test
  build: ->
    # Overwrites outer variable!
    package = @testPackage.compile()

  testPackage: ->
    package.create()
```

برای ایجاد متغیرهای global باید از انتساب آنها به عنوان خصوصیتی از شیء window استفاده کرد.

```
class window.Asset
    constructor: ->
```

با تضمین متغیرهای global به صورت صریح و روشن به جای به طور ضمنی بودن آنها، CoffeeScript یکی از منابع اصلی ایجاد مشکلات در جاوااسکریپت را حذف کرده‌است.

Semicolons

جاوااسکریپت اجباری برای نوشتن " **;** " ندارد، بنابراین ممکن است یک سری از دستورات از قلم بیافتند. با این حال در پشت صحنه‌ی کامپایلر جاوااسکریپت به " **;** " احتیاج دارد. به طوری که parser جاوااسکریپت به صورت خودکار هر زمانی که نتواند ارزیابی از دستورات داشته باشد، یک بار دیگر با " **;** " این کار را انجام می‌دهد و درصورت موفقیت، پیام خطایی مبنی بر نبود " **;** " را صادر می‌کند.
متاسفانه این یک ایده **بد** است. چرا که ممکن است تغییر رفتاری در کد نوشته شده به وجود آید. به مثال زیر توجه کنید. به نظر کد نوشته شده صحیح است؛ درسته؟

```
function() {}
```

```
(window.options || {}).property
```

اشتباه است، حداقل با توجه به parser، یک خطای syntax صادر می‌شود. در مورد دوم نیز parser، " ; " اضافه نمی‌کند و کد نوشته شده به کد یک خطی تبدیل می‌شوند.

```
function() {}(window.options || {}).property
```

حالا شما می‌توانید این موضوع را ببینید که چرا parser خطا داده‌است. وقتی شما در حال نوشتن کد جاوااسکریپتی هستید، باید بعد از هر دستور از " ; " استفاده کنید. خوشبختانه در تمام زمانیکه درحال نوشتن کد CoffeeScript هستید، نیازی به نوشتن " ; " ندارید. در زمانیکه کد CoffeeScript نوشته شده کامپایل می‌شود، به صورت خودکار " ; " را در جای مناسبی قرار می‌دهد.

عنوان: CoffeeScript #13
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۶/۱۷ ۲۳:۳۰
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

بخش‌های بد

در ادامه‌ی قسمت قبل ، به مواردی که توسط CoffeeScript اصلاح شده‌اند، می‌پردازیم.

Reserved words

کلمات کلیدی خاصی در جاوااسکریپت وجود دارد مانند **enum** ، **class** و **const** که برای نسخه‌های بعدی جاوااسکریپت در آینده رزرو شده‌اند. استفاده از این کلمات در برنامه‌های جاوااسکریپت می‌تواند نتایج غیرقابل پیش بینی داشته باشد. برخی از مرورگرهای به خوبی از عهده‌ی این کار برمی‌آیند و بعضی دیگر به طور کامل جلوی استفاده از این‌ها را گرفته‌اند. CoffeeScript بعد از تشخیص استفاده از یک کلمه‌ی کلیدی، با یک راه کار خاص، از این موضوع می‌گریزد.

به عنوان مثال، فرض کنید می‌خواهیم از کلمه کلیدی class به عنوان یک خصوصیت در یک شیء استفاده کنیم:

```
myObj = {
  delete: "I am a keyword!"
}
myObj.class = ->
```

پس از کامپایل، پارسر CoffeeScript متوجه استفاده شما از کلمه کلیدی رزرو شده می‌شود و آنها را در بین "" قرار می‌دهد.

```
var myObj;
myObj = {
  "delete": "I am a keyword!"
};
myObj["class"] = function() {};
```

Equality comparisons

مقایسه برابری ضعف دیگری است که در جاوااسکریپت باعث ایجاد رفتاری گیج کننده و اغلب باعث ایجاد اشکالاتی در کد نوشته شده می‌شود. به مثال زیر توجه کنید:

```
""           ==   "0"// false
0            ==   ""// true
0            ==   "0"// true
false        ==   "false"// false
false        ==   "0"// true
false        ==   undefined// false
false        ==   null// false
null         ==   undefined// true
" \t\r\n"    ==   0// true
```

مطمئنم که شما هم با من موافقید که همه‌ی مقایسه‌های بالا بسیار مبهم هستند و استفاده از آن‌های می‌توانند منجر به نتایج غیر منتظره شوند و همچنین مشکلاتی را پیش بیاورند.
راه حل این کار استفاده از عملگر برابری سختگیرانه است، که از 3 مساوی تشکیل شده است: **===** عملگر برابر سخت گیرانه دقیقا مانند عملگر برابری عادی عمل می‌کند و تنها نوع داده‌ها را بررسی می‌کند که با هم برابر باشند.
توصیه می‌شود که همیشه از عملگر برابری سختگیرانه استفاده کنید و هرجا لازم بود قبل مقایسه عمل تبدیل نوع داده‌ها را انجام دهید.
CoffeeScript این مشکل را به صورت کامل حل کرده است؛ یعنی هر جایی که عمل مقایسه == انجام شود به === تبدیل می‌شود. شما باید به صورت صریح نوع داده‌ها را قبل از مقایسه تبدیل کرده باشید.

نکته: در مقایسه‌ها رشته خالی ""، undefined، null و عدد 0 همگی false برمی گردانند.

```
alert "Empty Array"  unless [].length
alert "Empty String" unless ""
alert "Number 0"      unless 0
```

که پس از کامپایل می‌شود:

```
if (![].length) {
  alert("Empty Array");
}

if (!"") {
  alert("Empty String");
}

if (!0) {
  alert("Number 0");
}
```

در صورتیکه می‌خواهید به صورت صریح null و یا undefined را بررسی کنید، می‌توانید از عملگر ? CoffeeScript استفاده کنید:

```
alert "This is not called" unless ""?
```

پس از کامپایل می‌شود:

```
if ("" == null) {
  alert("This is not called");
}
```

با اجرای مثال بالا alert اجرای نمی‌شود چون رشته خالی با null برابر نیست.

Function definition

خیلی جالب است که در جاوااسکریپت می‌توانید تابعی را بعد از اینکه فراخوانی کردید، تعریف کنید. به عنوان مثال، کد زیر به صورت کامل اجرا می‌شود:

```
wem();
function wem() {alert("hi");}
```

این به دلیل دامنه (scope) تابع است. تمام توابع قبل از اجرای برنامه، به بالا برده می‌شوند و در همه جا در دامنه‌ای که در آن تعریف شده‌اند، قابل دسترسی می‌باشند؛ حتی اگر قبل از تعریف واقعی در منبع، فراخوانی شده باشد. مشکل اینجاست که عمل بالابردن توابع در مرورگرها با یکدیگر متفاوت است. به مثال زیر توجه کنید:

```
if (true) {
  function declaration() {
    return "first";
  }
} else {
  function declaration() {
    return "second";
  }
}
declaration();
```

در بعضی از مرورگرها مانند Firefox ، تابع declaration() مقدار " **first** " را برگشت خواهد داد و در دیگر مرورگرها مانند Chrome، مقدار " **second** " برگشت داده خواهد شد. در حالیکه به نظر می‌رسد که قسمت else هیچگاه اجرا نخواهد شد.

در صورتیکه علاقمند به کسب اطلاعات بیشتری درباره‌ی نحوه تعریف توابع، هستید باید راهنمای آقای Juriy Zaytsev را مطالعه کنید. به صورت خلاصه، رفتار نسبتا مبهم مرورگرها می‌تواند منجر به ایجاد مشکلاتی در مسیر نوشتن یک پروژه شوند.
همه چیز در CoffeeScript در نظر گرفته شده است و بهترین روش برای حل این مشکل، حذف کلمه function و به جای آن استفاده از عبارت (expression) تابع است.

Number property lookups نقصی که در پارسر جاوااسکریپت در مواجه با نماد نقطه (*dot notation*) بر روی اعداد وجود دارد، تفسیر آن به ممیز شناور، بجای مراجعه به ویژگی‌های آن است. برای مثال کد جاوااسکریپت زیر باعث ایجاد خطای نحوی می‌شود:

```
5.toString();
```

پارسر جاوااسکریپت بعد از نقطه به دنبال یک عدد دیگر می‌گردد و با برخورد با ()toString، باعث ایجاد یک Unexpected token می‌شود. راه حل این مشکل، استفاده از **پرانتز** یا اضافه کردن یک نقطه دیگر است.

```
(5).toString();
5..toString();
```

خوشبختانه پارسر CoffeeScript به اندازه‌ی کافی هوشمندانه با این مسئله برخورد می‌کند و هر زمانی که شما دسترسی به ویژگی‌های اعداد را داشته باشید، به صورت خودکار با اضافه کردن دوتا نقطه (همانند مثال بالا) جلوی ایجاد خطا را می‌گیرد.

عنوان: **CoffeeScript #14**
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۷/۱۷ ۱۰:۲۵
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

قسمت‌های اصلاح نشده CoffeeScript در حال رفع برخی از معایب طراحی جاوااسکریپت است و این راه، بس طولانی است.
همانطور که قبلا گفته شد، CoffeeScript به شدت به تجزیه و تحلیل استاتیک در زمان طراحی محدود شده است و هیچ بررسی در زمان اجرایی را برای بهبود کارآیی آن انجام نمی‌دهد.
CoffeeScript از یک کامپایلر مستقیم منبع به منبع استفاده می‌کند. با این دیدگاه که هر دستور در CoffeeScript در نتیجه به یک دستور معادل در جاوااسکریپت تبدیل می‌شود.
CoffeeScript برای همه‌ی کلمات کلیدی جاوااسکریپت، کلمه‌ی معادلی ایجاد نمی‌کند، مانند typeof؛ و همچنین برخی از معایب طراحی جاوااسکریپت، به CoffeeScript نیز اعمال می‌شود.
در دو قسمت قبل + و + بر روی معایب طراحی در جاوااسکریپت که توسط CoffeeScript اصلاح شده بود، توضیح دادیم. حال می‌خواهیم درباره برخی از معایب جاوااسکریپت که CoffeeScript تا به حال نتوانسته است آنها را اصلاح کند صحبت کنیم.

استفاده از eval

در حالیکه CoffeeScript برخی از نقاط ضعف جاوااسکریپت را اصلاح کرده است، اما همچنان معایب دیگری نیز وجود دارند، که شما تنها باید از این نقاط ضعف آگاه باشید. یکی از این موارد، تابع **eval** است. برای استفاده از آن، باید با اشکالاتی که در حین کار با آن مواجه می‌شوید، آگاهی کامل داشته باشید و در صورت امکان از استفاده از آن اجتناب کنید.

تابع **eval** یک رشته از کد جاوااسکریپت را در حوزه‌ی محلی اجرا می‌کند و توابعی مانند **setTimeout** و **setInterval** نیز می‌توانند در آرگومان اولشان یک رشته از کد جاوااسکریپت را دریافت و ارزیابی کنند.

با این حال، مانند with، eval نیز ردیابی کامپایلر را از کار می‌اندازد و این امر تاثیر بسیار زیادی بر روی کارآیی آن دارد. کامپایلر هیچ ایده ای درباره کدی که درون eval قرار داده شده است، ندارد تا زمانی که آن را اجرا کند. به همین دلیل نمی‌تواند هیچ عمل بهینه سازی را بر روی انجام دهد. یکی دیگر از نگرانی‌های استفاده‌ی از eval، **امنیت** است. در صورتیکه شما ورودی را به eval ارسال کنید، eval باعث می‌شود که کد شما به راحتی در معرض حملات تزریق کد قرار می‌گیرد. در 99% از مواقع، وقتی شما می‌خواهید از eval استفاده کنید، راه‌های بهتر و امن‌تری وجود دارند (مانند استفاده از براکت).

```
# Don't do this
model = eval(modelName)

# Use square brackets instead
model = window[modelName]
```

استفاده از typeof

اپراتور typeof احتمالا بزرگترین نقص طراحی جاوااسکریپت است؛ تنها به این دلیل که اساسا به طور کامل شکست خورده است. در واقع از آن فقط یک استفاده می‌شود تا تشخیص داده شود که یک مقدار undefined است یا نه.

```
typeof undefinedVar is "undefined"
```

برای چک کردن type همه types، متاسفانه **typeof** نمی تواند به درستی این کار را انجام دهد و مقدار بازگشتی آن وابسته به مرورگر و چگونگی نمونه سازی آن نمونه است. در این رابطه CoffeeScript هیچ کمکی به شما نمی‌تواند بکند، چرا که قبلا نیز گفته شد، CoffeeScript یک زبان با تجزیه و تحلیل استاتیک است و هیچ بررسی در زمان اجرایی بر روی نوع آن ندارد.
در اینجا لیستی از مشکلات، هنگام استفاده از typeof را مشاهده می‌کنید:

```
Value               Class      Type
```

```
-----------------------------------------------
"foo"               String     string
new String("foo")   String     object
1.2                 Number     number
new Number(1.2)     Number     object
true                Boolean    boolean
new Boolean(true)   Boolean    object
new Date()          Date       object
new Error()         Error      object
[1,2,3]             Array      object
new Array(1, 2, 3)  Array      object
new Function("")    Function   function
/abc/g              RegExp     object
new RegExp("meow")  RegExp     object
{}                  Object     object
new Object()        Object     object
```

همانطور که مشاهده می‌کنید تعریف یک رشته در داخل **""** و یا با کلاس **String** ، در نتیجه‌ی استفاده از typeof تاثیر گذار است. به طور منطقی typeof باید "string" را به عنوان خروجی در هر دو حالت نشان دهد، اما برای دومی به صورت "object" باز می‌گرداند.

سوالی که اینجا مطرح می‌شود این است که ما چطور می‌توانیم یک نوع را در جاوااسکریپت چک کنیم؟

خوب، خوشبختانه Object.prototype.toString() ما را نجات داده است. اگر ما این تابع را بر روی یک شیء خاص فراخوانی کنیم، مقدار صحیح را بر می‌گرداند.

در اینجا مثالی از نحوه‌ی پیاده سازی jQuery.type را مشاهده می‌کنید:

```
type = do ->
  classToType = {}
  for name in "Boolean Number String Function Array Date RegExp Undefined Null".split(" ")
    classToType["[object " + name + "]"] = name.toLowerCase()

  (obj) ->
    strType = Object::toString.call(obj)
    classToType[strType] or "object"

# Returns the sort of types we'd expect:
type("")          # "string"
type(new String) # "string"
type([])          # "array"
type(/\d/)        # "regexp"
type(new Date)    # "date"
type(true)        # "boolean"
type(null)        # "null"
type({})          # "object"
```

در صورتیکه بخواهید تشخیص دهید یک متغیر تعریف شده است یا نه، باید از typeof استفاده کنید؛ در غیر این صورت پیام خطای ReferenceError را دریافت خواهید کرد.

```
if typeof aVar isnt "undefined"
  objectType = type(aVar)
```

و یا به طور خلاصه‌تر با استفاده از اپراتور وجودی:

```
objectType = type(aVar?)
```

راه دیگری برای چک کردن نوع، استفاده از اپراتور وجودی CoffeeScript است. برای مثال: می‌خواهیم یک مقدار را در یک آرایه اضافه کنیم. می‌توان گفت تا زمانیکه تابع push پیاده سازی شده باشد ما باید با آن مانند یک آرایه رفتار کنیم.

```
anArray?.push? aValue
```

اگر anArray یک شیء به غیر از آرایه باشد، اپراتور وجودی تضمین خواهد کرد که هیچگاه تابع push فراخوانی نخواهد شد.

استفاده از instanceof

کلمه‌ی کلیدی **instanceof** نیز تقریبا همانند **typeof** شکست خورده است. در حالت ایده آل، instanceof، سازنده‌ی دو شیء را با هم مقایسه می‌کند، در صورتیکه یک شیء نمونه‌ای از شیء دیگر باشد، یک مقدار boolean را باز می‌گرداند. در واقع instanceof موقعی کار مقایسه را انجام می‌دهد که اشیاء، سفارشی سازی شده باشند. وقتی عمل مقایسه می‌خواهد بر روی این نوع اشیاء سفارشی سازی شده، انجام شود، استفاده از typeof بی‌فایده است.

```
new String("foo") instanceof String # true
"foo" instanceof String             # false
```

علاوه بر این، instanceof همچنین بر روی اشیاء در فریم‌های مختلف مرورگر عمل مقایسه را نمی‌تواند انجام دهد. در واقع instanceof فقط نتیجه‌ی صحیح مقایسه را در اشیاء سفارشی سازی شده برمی‌گرداند؛ مانند کلاس‌های CoffeeScript.

```
class Parent
class Child extends Parent

child = new Child
child instanceof Child  # true
child instanceof Parent # true
```

مواقعی از instanceof استفاده کنید که مطمئن هستید بر روی اشیای ساخته شده توسط شما بکار گرفته می‌شود و یا هرگز از آن استفاده نکنید.

استفاده از delete از کلمه کلیدی **delete** برای حذف خصوصیات موجود در اشیاء به صورت کاملا مطمئن، می‌توان استفاده کرد.

```
anObject = {one: 1, two: 2}
delete anObject.one
anObject.hasOwnProperty("one") # false
```

هر نوع استفاده دیگر، از قبیل حذف متغیرها و یا توابع کار نخواهد کرد.

```
aVar = 1
delete aVar
typeof Var # "integer"
```

در صورتیکه می‌خواهید یک اشاره گر به یک متغیر را حذف کنید فقط کافیست مقدار **null** را به آن انتساب دهید.

```
aVar = 1
aVar = null
```

عنوان: CoffeeScript #15
نویسنده: وحید محمّدطاهری
تاریخ: ۱۳۹۴/۰۷/۲۱ ۱۲:۵۰
آدرس: www.dotnettips.info
گروه‌ها: JavaScript, CoffeeScript

قسمت‌های اصلاح نشده در ادامه‌ی مطالب قسمت قبل ، به برخی دیگر از معایب طراحی در جاوااسکریپت که در CoffeeScript نیز اصلاح نشده‌اند می‌پردازیم.

استفاده از parseInt

تابع parseInt() در جاوااسکریپت، در صورتیکه یک مقدار رشته‌ای را به آن ارسال کنید و پایه‌ی مناسب آن را تعیین نکنید، نتایج غیره منتظره‌ای (*unexpected*) را باز می‌گرداند . برای مثال:

```
# Returns 8, not 10!
parseInt('010') is 8
```

البته ممکن است شما این کد را در مرورگر خود تست کنید و مقدار 10 را باز گرداند؛ اما این برای همه‌ی مرورگرها یکسان نیست. برای اطمینان از مقدار بازگشتی صحیح، همیشه پایه‌ی آن را تعیین کنید.

```
# Use base 10 for the correct result
parseInt('010', 10) is 10
```

دقت کنید این چیزی نیست که CoffeeScript بتواند برای شما انجام دهد؛ شما فقط یادتان باشد که همیشه پایه‌ی صحیح را در موقع استفاده‌ی از ()parseInt تعریف کنید.

Strict mode

Strict mode یکی از قابلیت‌های ECMAScript 5 است که به شما اجازه می‌دهد تا یک برنامه یا تابع جاوااسکریپت را در محیطی محدود اجرا کنید. این محدودیت موجب نمایش بیشتر خطاها و هشدارها نسبت به حالت نرمال می‌شود و به توسعه دهندگان این امکان را می‌دهد تا از نوشتن کدهای غیر قابل بهینه سازی برای اشتباهات رایج جلوگیری کنند.
به عبارت دیگر Strict mode باعث کاهش اشکالات، افزایش امنیت، بهبود عملکرد و حذف برخی از سختی‌های استفاده از ویژگی‌های زبان می‌شود.
در حال حاضر Strict mode، در مرورگرهای زیر پشتیبانی می‌شود:

Chrome >= 13.0
Safari >= 5.0
Opera >= 12.0
Firefox >= 4.0
IE >= 10.0

با این حال، Strict mode به طور کامل با مرورگرهای قدیمی سازگار است.

تغییرات Strict mode

بیشتر تغییرات Strict mode مربوط به syntax جاوااسکریپت بوده است:

خطا در پروپرتی‌ها و نام آرگومان‌های تابع تکراری
خطا در عدم استفاده‌ی صحیح از *delete*
خطا در زمان دسترسی به arguments.caller و arguments.callee (به دلایل عملکرد)

استفاده از عمگر *with* سبب بروز خطای نحوی می‌شود
متغیرهای خاص مانند *undefined* که قابل نوشتن نیستند
معرفی کلمات کلیدی رزرو شده مانند *static* ، *public* ، *protected* ، *private* ، *package* ، *let* ، *interface* ، *implements* و *yield* .

با این حال، برخی از رفتارهای زمان اجرای Strict mode نیز تغییر کرده است:
متغییرهای سراسری به صورت صریح و روشن هستند (کلمه کلیدی var نیاز است). مقدار سراسری *this* نیز به صورت undefined است.
eval نمی‌تواند متغیر جدیدی را در حوزه‌ی محلی خود تعریف کند.
بدنه‌ی هر تابع باید قبل از استفاده تعریف شده باشد (قبلا گفتم که در جاوااسکریپت شما می‌توانید قبل از تعریف تابع آن را فراخوانی کنید).
آرگومان‌ها تغییر ناپذیر هستند.

CoffeeScript در حال حاضر بسیاری از الزامات Strict mode را پیاده سازی کرده‌است مانند: همیشه از کلمه کلیدی var برای تعریف متغیر استفاده می‌کند؛ اما فعال کردن Strict mode در برنامه‌های CoffeeScript نیز بسیار مفید خواهد بود. در واقع CoffeeScript بر روی انطباق برنامه‌ها با Strict mode در زمان کامپایل را، در برنامه‌های آینده خود دارد.

استفاده از Strict mode
برای فعال کردن بررسی محدودیت، کد و توابع خود را با این رشته شروع کنید:

```
->
  "use strict"

  # ... your code ...
```

فقط با استفاده از رشته "use strict". به مثال زیر توجه کنید:

```
do ->
  "use strict"
  console.log(arguments.callee)
```

اجرای قطعه کد بالا درحالت strict mode، سبب بروز خطای syntax می‌شود؛ در حالیکه در حالت معمول این کد به خوبی اجرا می‌شود.
Strict mode دسترسی به arguments.callee و arguments.caller، که تاثیر بدی را بر روی عملکرد کد شما دارند، حذف می‌کند و استفاده‌ی از آنها سبب بروز خطا می‌شود.

در مثال زیر در حالت strict mode سبب بروز خطای *TypeError* می‌شود، اما در حالت نرمال به خوبی اجرا شده و یک متغیر سراسری را ایجاد می‌کند.

```
do ->
  "use strict"
  class @Spine
```

دلیل این رفتار این است که در Strict mode متغیر *this* به صورت *undefined* است؛ در حالیکه در حالت نرمال، *this* به شیء *window* اشاره می‌کند. راه حل این مشکل تعریف متغیرهای سراسری به صورت صریح به شیء window است.

```
do ->
  "use strict"
  class window.Spine
```

در حالیکه توصیه می‌شود که همیشه Strict mode فعال باشد، اما Strict mode هیچ یک از ویژگی‌های جدید جاوااسکرییت را که

هنوز آماده نیست، فعال نمی‌کند و در واقع به علت بررسی بیشتر کدهای شما در زمان اجرا، باعث کاهش سرعت می‌شود.
شما می‌توانید در زمان توسعه برنامه جاوااسکریپت خود Strict mode را فعال کنید و در زمان انتشار، بدون Strict mode برنامه‌ی خود را منتشر کنید.

JavaScript Lint

JavaScript Lint یک ابزار بررسی کیفیت کدهای جاوااسکریپت است و اجرای برنامه‌ی شما از طریق این راه عالی باعث بهبود کیفیت و بهترین شیوه‌ی کد نویسی می‌شود. این پروژه براساس ابزار JSLint است. شما می‌توانید چک لیست سایت JSLint را که شامل موضوعاتی است که باید آن‌ها در نظر داشته باشید، مانند متغیرهای سراسری، فراموش کردن نوشتن سمی کالن، کیفیت ضعیف عمل مقایسه را نام برد.

خبر خوب این است که CoffeeScript تمام موارد گفته شده‌ی در چک لیست را انجام می‌دهد. بنابراین کد تولیدی CoffeeScript با JavaScript Lint کاملا سازگار است. در واقع ابزار *coffee* از option، lint پشتیبانی می‌کند.

```
coffee --lint index.coffee
  index.coffee: 0 error(s), 0 warning(s)
```