عنوان: **Closure در JavaScript**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۴/۰۴ ۷:۱۸
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, JavaScript, Closure

در قسمت قبلی درباره علت نیاز به الگوهای طراحی در JavaScript و Function Spaghetti code صحبت شد. در این قسمت Closure در JavaScript مورد بررسی قرار می‌گیرد.
در JavaScript می‌توان توابع تو در تو نوشت (nested functions) ، زمانی که یک تابع درون تابع دیگر تعریف می‌شود تابع درونی به تمام متغیرها و توابع تابع بیرونی (Parent) دسترسی دارد.
Douglas Crockford
برای تعریف Closure می‌گوید :

```
an inner function always has access to the vars and parameters of its outer function, even after the
outer



function has returned
```

یک تابع درونی (nested) همیشه به متغیرها و پارامترها تابع بیرونی دسترسی دارد ، حتی اگر تابع بیرونی مقدار برگردانده باشد.

تابع زیر را در نظر بگیرید :

```
// The getDate() function returns a nested function which refers the 'date' variable defined
// by outer function getDate()
function getDate() {
   var date = new Date();    // This variable stays around even after function returns

   // nested function
   return function () {
      return date.getMilliseconds();
   }
}
```

اکنون اگر به صورت زیر تابع getDate فراخوانی شود مشاهده می‌شود که تابع درونی (با کامنت nested function مشخص شده است.) به شیء date دسترسی دارد.

```
// Once getDate() is executed the variable date should be out of scope and it is, but since
// the inner function
// referenes date, this value is available to the inner function.
var dt = getDate();

alert(dt());
alert(dt());
```

خروجی هر 2 alert یک مقدار خواهد بود.
اگر از فردی که به تازگی رو به JavaScript آورده است خواسته شود تابعی بنویسد که میلی ثانیه‌ی زمان جاری را برگداند احتمالا همچین کدی تحویل می‌دهد :

```
function myNonClosure() {
            var date = new Date();
            return date.getMilliseconds();
        }
```

در کد بالا پس از اجرای myNonClosure متغیر date از بین خواهد رفت ، این مسئله در دنیای JavaScript طبیعی هست.
این مثال را در نظر بگیرید :

```
var MyDate = function () {
    var date = new Date();
    var getMilliSeconds = function () {
        return date.getMilliseconds();
    }
}

var dt = new MyDate();

alert(dt.getMilliSeconds());  // This will throw error as getMilliSeconds is not accessible.
```

در صورت اجرای مثال بالا خطایی با این مضمون دریافت خواهد شد که getMilliSeconds دستیابی پذیر نیست. (کپسوله شده)
برای اینکه آن را دستیابی پذیر کنیم کد را به این صورت تغییر می‌دهیم :

```
// This is closure
var MyDate = function () {
    var date = new Date();   // variable stays around even after function returns
    var getMilliSeconds = function () {
        return date.getMilliseconds();
    };
    return {
        getMs : getMilliSeconds
    }
}
```

آنچه در تابع بالا انجام شده کپسوله سازی همه‌ی منطق کار (منطق کار در اینجا برگرداندن میلی ثانیه زمان جاری می‌باشد) در یک فضای نام به نام MyDate می‌باشد. همچنین فقط متدهای عمومی در اختیار استفاده کننده این تابع قرار داده شده است. برای استفاده می‌توان بدین صورت عمل کرد :

```
var dt = new MyDate();
alert(dt.getMs());  // This should work.
```

در کد بالا برای توابع و متغیرهای درونی یک container ایجاد کردیم که باعث جلوگیری از تداخل در نام متغیرها با دیگر کدها خواهد شد . (برای مشاهده‌ی تداخل‌ها به قسمت قبلی توجه کنید.)
اگر بخواهیم Closure را تشبیه کنیم ، Closure شبیه به کلاس‌ها در C# یا Java هست.
Closure یک حوزه (scope) برای متغیرها و توابع درونی خودش ایجاد می‌کند.
jQuery بهترین مثال کاربردی برای Closure می‌باشد :

```
(function($) {

    // $() is available here

})(jQuery);
```

در ادامه این مفاهیم بیشتر توضیح داده می‌شودند ، اکنون می‌خواهیم مشکلی که در قسمت قبلی مطرح کردیم به کمک Closure حل کنیم :

در آن مثال گفته شد که اگر :

```
// file1.js
function saveState(obj) {
    // write code here to saveState of some object
    alert('file1 saveState');
}
// file2.js (remote team or some third party scripts)
function saveState(obj, obj2) {
```

```
     // further code...
    alert('file2 saveState");
}
```

اگر تابعی به نام saveState در 2 فایل مختلف داشته باشیم و این 2 فایل را بدین صورت در برنامه آدرس دهیم :

```
<script src="file1.js" type="text/javascript"></script>
<script src="file2.js" type="text/javascript"></script>
```

تابع saveState در فایل دوم تابع saveState فایل اول را override می‌کند. یک از توابع بالا را به صورت زیر باز نویسی می‌کنیم و منطق کار را کپسوله می‌کنیم :

```
function App() {
    var save = function (o) {
        // write code to save state here..
        // you have acces to 'o' here...
        alert(o);
    };

    return {
        saveState: save
    };
}
```

بدون نگرانی تداخل saveState با بقیه saveStateها در هر پلاگین یا فایل دیگری می‌توان از saveState اینگونه استفاده کرد :

```
var app = new App();

app.saveState({ name: "rajesh"});
```

برای اطلاعات بیشتر در مورد Closure ها این لینک را بررسی کنید.

نظرات خوانندگان

نویسنده: MBE
تاریخ: ۱۳۹۱/۰۴/۰۴ ۲۰:۳۱

ممنون عالی بود . اما این بحث واقعا جای کار داره . من که منتظر مقالات بیشتری در مورد Closure هستم .

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۴/۰۴ ۲۱:۳۱

خواهش می‌کنم. در ادامه درباره الگوهای Prototype و Module مطلب می‌نویسم و سعی می کنم با یک مثال بحث را بیشتر باز کنم.

نویسنده: صالح
تاریخ: ۱۳۹۱/۰۴/۰۴ ۲۳:۴۸

نکته بسیار خوب و کاربردی در جاوا اسکریپت بود. ممون

نویسنده: پوران
تاریخ: ۱۳۹۲/۱۱/۱۹ ۱۵:۵۵

واقعا مرسی ... عالی بود ...