İleri Seviye Javascript

Yazan: Azer Koçulu

İndeks

Giriş

Bu dökümanda Javascript'in deneyselliğe dayanan doğasına uygun biçimde, web tabanlı program üretmenizi sağlayacak bilgileri vermeye çalıştım. Özellikle üstünde durmanız gereken konu olan fonksiyonlarda, kitaplarda yeralmayan ileri teknik ve pratikleri anlattım.

Dökümandan iyi verim almak için, Rainbow9'dan faydalanabilirsiniz. Rainbow9, geçen yıl hazırladığım, javascript deneyleri yapmayı ve web tabanlı programlar geliştirmeyi sağlayan bir programdır.İsterseniz Firefox'un Firebug eklentisinden de faydalanabilirsiniz.Rainbow9, Firebug'ın aksine bazı mobil tarayıcılar dahil tüm tarayıcılarda kurulmaya gerek olmadan çalışır.

Rainbow9: http://www.rainbow9.org Firebug: http://www.getfirebug.com

Dökümanı okuyup bitirdikten sonra daha deneysel ve iyi tekniklerle kod yazacağınızı, kod standartınızı geliştireceğinizi umut ediyorum.

Sabır ve gayretlerinizin karşılığını almanız dileğiyle,

Azer Koçulu <http://azer.kodfabrik.com>

Destekleyen, yardım eden, moral verenlere teşekkür ederim: Annem Nuray Koçulu, Can Çatalyürek, Hakan Bilgin, Muhammed Daud

---

Fonksiyonlar

C tabanlı dillerde her iş veya alt iş fonksiyonlar tarafından yürütülür.Fonksiyonların genel kullanım şekli, argüman tünelinden veri gönderip, bilgi çıkışı beklemektir.Genel yazılışı inceleyelim:

```
    function(argümanlar){ return argüman }
```

Gönderilen argümanlar herhangi bir türde olabilir, sayı, dizi fonksiyon...Buradaki tek dikkat unsuru argümanların her birine ayrı tanıtıcı belirlenmesidir.Argümanlar birden fazlaysa birbirlerinden virgülle ayrılmaları gerekir:

```
    (argüman1,argüman2,argüman3...)
```

ECMAScript'in kullandığımız sürümünde, argümanlar için PHP veya Python dillerinde olduğu gibi varsayılan değer atanamaz.Değer atanmayan argümanların tanıtıcıları yine tanımlanır ancak "undefined" türünde olurlar.Argümanlar sadece tanımlandıkları fonksiyonda çalıştırılabilirler.Global blokta tanımlanan değişkenlere fonksiyonlardan erişilebilir.

Bir fonksiyonda en son, fonksiyondan geriye değer döndürülmesini sağlayan "return" ifadesi çalıştırılır.Eğer bu ifadeden sonra fonksiyonun içeriği bitmediyse, ECMAScript bu içeriği görmezden gelir.

Fonksiyonlar çeşitli biçimlerde tanımlanabilir, ilk olarak tanıtıcı ve eşit işareti kullanarak klasik biçimde tanımlayalım:

```
    var islem = function(){}
```

Biraz daha pratik olan diğer yöntemdeyse pek çok programlama dilinden alışık olduğumuz biçimde tanımlıyoruz:

```
    function islem(){}
```

Örnek olarak argüman tünelinden aldığı iki sayının toplamı geriye döndüren bir fonksiyon tanımlayalım:

```
function topla(sayi1,sayi2){
    return sayi1+sayi2
}
```

Bu biçimlerde tanımlamalar genelde tercih edilse de, bazen programın akışına göre şekillenmesi gereken fonksiyon tanımlamaya ihtiyaç duyulur.Bu durumda tanıtıcıların program çalıştırılmadan önce belirlenmek zorunda olması, pek çok dilde rastladığımız bir kısıtlamadır.Fakat

ECMAScript'in Function() nesnesi sayesinde, kod bloğu tanıtıcı string tipinde belirtilerek, programın akışına göre şekillenen fonksiyonlar tanımlanabilmektedir:

```
    var topla = new Function("sayi1","sayi2","return sayi1+sayi2")
```

Fonksiyonların atama işlemleri diğer veri türleriyle aynı olsa da, fonksiyona ulaşıp çalıştırma işlemi ayrı biçimdedir.Bir fonksiyonu çalıştırmak için tanıtıcıyla beraber argüman tüneli kullanılır.Burada argüman tünelinin veri transferi yapıp yapmaması farketmez.Örnek olarak hazırladığımız topla fonksiyonunu kullanalım:

```
>>> var sonuc = topla(5,4)
    9
```

"sonuc" değişkeni integer türünde 9 değerini aldı.

Örnekteki topla() fonksiyonunda hem bilgi girişi hem de bilgi çıkışı yapılıyor, şimdi bilgi girişi veya çıkışı yapılmayan bir örneği inceleyelim:

```
>>> var durum = 'Negatif';
>>> var durumuAyarla = function(){
    durum = 'pozitif'
}
```

Bu örnekteyse programın global bloğunda "durum" adında bir değişken ve "durumuAyarla" adında, "durum" değişkeninin değerini string tipinde "pozitif" değeriyle değiştiren bir fonksiyon oluşturuluyor.Çalıştırıp deneyelim;

```
>>> durumuAyarla()
undefined
>>> durum
'pozitif'
```

Parantez Blokları

Fonksiyon tipindeki verilerin çalışmasını incelediğimiz örneklerde parantezlerin sağladığını görüyoruz.Parantezler işlevsel olarak pek çok konuda fayda sağlasa da, pek çok zaman yazılan kodun daha anlaşılır olması için blok oluşturmada kullanılır.Önce ne gibi işlevlerde kullandığımıza gözatalım.Birinci örnek matematiksel işlemlerde gruplama yaparken kullanılan parantezler üzerine:

```
>>> 1+2*3
7
>>> (1+2)*3
9
```

İkinci örnek, sayısal değerlerin ondalık kısmıyla methodların karışması problemine parantez çözümü:

```
>>> 1.toString()
missing ; before statement
[Break on this error] 1.toString()
javascript: with ... (line 1)
>>> 1.0.toString()
"1"
>>> (1).toString()
"1"
```

Gelelim üçüncü örneğe.Ecmascript'te her fonksiyon bloğu parantezler içine alınabilir.Ve elbette, parantez içine alınmış kod bloğuna çalıştırma komutu gönderilebilir.Bu yöntem daha sade ve temiz görünen kodlar yazmayı sağlar, çünkü fonksiyonu yazdıktan sonra tanıtıcıyı çağırmaya gerek duyulmaz.

Klasik yöntemle yazılan bir fonksiyonun çağırma işlemiyle, parantez bloğunun kodlarını karşılaştıralım:

```
>>> var merhaba = function(){
          return "Merhaba"
       }
>>> merhaba();
Merhaba
>>> (var merhaba = function(){
          alert("Merhaba");
       })();
"Merhaba"
```

İlk örnekte fonksiyonu tanımladık ve ardından tanıtıcısını kullanarak çalıştırdık, ikinci örnekteyse fonksiyonu parantez bloklarının içine alıp çalıştırdığımız için, tanıtıcıyla çağırma yaptığımız satıra gerek kalmadı.Tahmin edeceğiniz üzere, aslında parantez bloklarını kullanırken tanıtıcı kullanılmayabilir:

```
>>> (function(){
          alert("Merhaba");
       })();
"Merhaba"
```

Argümanlar

Fonksiyonlar, sadece faaliyet alanlarında erişilebilir durumda olan değerler alabilirler.Bu değerlerin tanıtıcısı, fonksiyon oluşturulurken argüman tünelinde yazılır.Fakat gönderilen argümanlar için tanıtıcı belirtmek zorunlu değildir, tanıtıcı olsun veya olmasın, fonksiyon içindeki "arguments" tanıtıcılı değişken argüman tünelindeki tüm verileri saklar.

Diziler, tek tanıtıcının altında veri parçalarını tanıtıcıya ihtiyaç kullanmadan saklamayı sağlayan veri türüdür, örnek olarak herbiri tek başına bir değer olan birkaç elmayı tutan sepeti düşünelim.Sepeti tutan el, programımızı çalıştıran browser olmalı :) Fonksiyonlardan bir sonraki konu başlığında dizileri daha detaylı inceleyeceğiz.

Az önce sözünü ettiğim "arguments" değeri, fonksiyonların argüman tünelinden gelen tüm değerleri sıralı olarak saklayan bir dizidir.Tanıtıcısı olsun veya olmasın, browserlar gönderilen tüm argümanlara erişmemizi sağlar.Bir örnek;

```
>>> var argumanlar = function(){
          return arguments;
      }
>>> argumanlar(3,5);
3,5
>>>
```

Örnekte "argumanlar" adında, tek işlevi kendisine gelen argümanları geri döndürmek olan bir fonksiyon oluşturduk ve bu fonksiyona gönderdiğimiz argümanların bize dizi tipinde geri döndüğünü gördük.Toplama örneğini "arguments" dizisinin elemanlarını seçip işlem yaptırarak yeniden ele alalım:

```
>>> var toplama = function(){
          return arguments[0]+arguments[1];
      }
>>> toplama(3,5);
8
```

Bu örnekte `arguments` dizisinin ilk iki elemanını, tanıtıcıları olan sıra numaralarıyla seçtik ve toplamlarını bulup geri döndürdük.Dizileri henüz iyice anlamadığınızı düşünüyorsanız endişelenmeyin, dizileri ayrı bir başlıkta detaylı biçimde inceleyeceğiz.

Fonksiyonların içinde kaç argüman tanımlandığını bulmak içinse, fonksiyon tanıtıcısının alt değişkeni olan .length değeri kullanılır.Örnek olarak, argüman tünelinde tanımlanan değişkenlerin sayısını gönderen bir fonksiyon yazalım:

```
>>> function test(arguman1,arguman2){
        return test.length;
    }
>>> test()
2
```

Fonksiyonlar için method ve alt değişken tanımlamak

ECMAScript'te, fonksiyonların her biri diğer dillerde rastladığımız "class"'lara benzer, alt değerler tanımlanabilir ve `new` operatörüyle klonlanabilir.

Alt değer tanımlamak için, fonksiyonun içindeki "this" değerine erişilir.Alt değerler fonksiyon bloğunun dışında da tanımlanabilir. Bir örnek;

```
>>> var islemci = function(){
        this.aciklama = "merhaba dünya!"
        this.toplama = function()
        {
           return arguments[0]+arguments[1];
        }
        this.cikarma = function(sonuc)
        {
            return arguments[0]-arguments[1];
        }
    }
>>> var islem1 = new islemci();
>>> islem1.toplama(3,5);
8
>>> islem2.cikarma(3,5);
-2
```

`islemci` adında, `toplama cikarma` methodlarını içeren bir fonksiyon yarattık ve ardındaki satırda, `islem1` tanıtıcısına yeni bir `islemci` objesini atadık.Bu kez, az önce yazdığımız `islemci` objesinde, `prototype`ı kullanarak yeni bir alt method yaratalım;

```
>>> islemci.prototype.carpma = function(){
          return arguments[0]*arguments[1];
    }
>>> var islem2 = new islemci();
>>> islem2.carpma(3,5);
15
```

`prototype` Klonlanabilir nesnelerin ortak etki alanına sahip olmasını sağlayan bir değerdir.Eğer `prototype`ı kullanmadan tanımlama yaparsak, alt methodu olduğumuz objenin etki alanına yani içerdiği değerlere erişemeyiz.Buna bir örnek verelim:

```
>>> islemci.prototype.aciklamayiGoster1 = function(){
        return this.aciklama;
    }
>>> islemci.aciklamayiGoster2 = function(){
        return this.aciklama;
    }
```

Görevleri `islemci` objesine ait `aciklama` değerini döndürmek olan etki alanları farklı, iki tane fonksiyon tanımladık.İlk tanımladığımız fonksiyonu deneyelim:

```
>>> islemci.aciklamayiGoster1()
"merhaba dünya!"
```

Gördüğünüz gibi, `islemci` nesnesine prototype köprüsünden erişti ve içerdiği değerleri aldı.Diğerini deneyelim:

```
>>> islemci.aciklamayiGoster2()
undefined
```

Tanımlanmamış değer döndürdü çünkü etki alanı kendisiydi.sadece kendi içindeki ve window altındaki değerlere ulaşabilirdi.Elbette bu problem çözümsüz değil, `call` ve `apply` methodları fonksiyonların etki alanını belirlememizi sağlar.

Dışarıdan Erişime Kapalı Özellikler

C Tabanlı programlama dillerindeki `class` yapılarında, methodların dışarından erişime açık olup olmadığı belirtilebilir.ECMAScript'in yeni sürümündeki `class` yapıları da public/private alt method ve değerleri destekler.

Şu an kullandığımız sürümde de public/private özellikler oluşturulabiliyor, bunun için fonksiyon bloklarının etki alanlarından faydalanılır;

```
>>> function test(){
        var _deger = "merhaba dünya!";
        this.degeriGetir = function(){
                return _deger;
        }
}
```

Yukaridaki örnekte test adında bir fonksiyon oluşturduk ve biri dışarıdan erişime kapalı, iki değer oluşturduk.

```
>>> test1 = new test();
>>> test1._deger
undefined
>>> test1.degeriGetir()
"merhaba dünya!"
```

Apply

`apply` Javascript'in isviçre çakısıdır.Bu özellik sayesinde, fonksiyonlar birbirlerinin yapılarının üstüne inşa edilebilir ve bunun saymakla bitmez faydalarından biri, modern object oriented dillerinin kalıtım özelliğini bize sağlamasıdır.Örneğin, kimlik bilgileri içeren "kişi" objesi üzerine, skor ve benzeri bilgileri içeren "oyuncu" objesi oluşturalım:

```
>>> var kisi = function(adsoyad,yas,dogumyeri){
          this.adsoyad = adsoyad;
          this.yas = yas;
          this.dogumyeri = dogumyeri;
      }
>>> var arda = new kisi("Arda Koçulu",17,"Kars");
>>> arda.dogumyeri
"Kars"
```

Kişi objesini inceleyelim; bir fonksiyon oluşturduk ve argüman tünelinden gelen üç veriyi bu fonksiyona alt değer olarak atadık.Ardındaki satırda "arda" değerini "new" operatörüyle yeni bir kişi olarak tanımladık.Kişi objesini oluşturduğumuza göre, oyuncu objesine de geçebiliriz:

```
>>> var oyuncu = function(adsoyad,yas,dogumyeri,takim,skor){
          this.takim = takim;
          this.skor = skor;
          kisi.apply(this,arguments);
       }
>>> var kemal = new oyuncu("kemal akın",15,"antalya","maviler",3);
>>> var ege = new oyuncu("ege akın",10,"izmir","kırmızılar",2);
>>> kemal.takim
maviler
>>> ege.dogumyeri
izmir
```

Oyuncu adında bir nesne oluşturduk, argüman tünelinden beklediğimiz değerleri tanıtıcılarına tanımladıktan sonra, `takim` ve `skor` alt değerlerini atadık.Ardından gelen satırda ise, `kisi` objesinin `apply` methodunu çalıştırdık ve "this","arguments" argümanlarını gönderdik. "this" argümanı, kişi objesinin hangi fonksiyon tarafından çağrıldığını göndermek için kullanılır.`arguments` Argümanıysa, `apply` methoduyla çağrılan fonksiyonun argüman tünelinden beklediği argümanları göndermemizi sağlar."arguments" değeri, oyuncu fonksiyonuna gelen değerleri içerdiğinden, "oyuncu"

fonksiyonuna gelen tüm argümanlar "kisi" fonksiyonuna gönderililir.

Call

`call` fonksiyonu tıpkı `apply` gibi, fonksiyonları kendi belirlediğimiz etki alanı ve argümanlarla çalıştırmamızı sağlar.Bir problemin çözümünü anlatarak örnek yapalım, javascript arguments dizisinin alt methodlarını kullanmamıza olanak tanımaz.Deneyelim;

```
>>> (function(){
alert(arguments.slice); // undefined değer döner
})("merhaba","dünya");
```

Parantez blokları içindeki fonksiyonumuza iki metinsel değer gönderdik.Fonksiyondaysa, gelen argüman listesinin `slice` methodunu kontrol ediyoruz ve tanımlanmamış olduğunu görüyoruz.

Şimdi bu problemin çözümünü inceleyelim:

```
>>> (function(){
alert(Array.prototype.slice.call(arguments,1) // dönen değer: ["dünya"]
})("merhaba","dünya");
```

`slice` Methodunun işlevi o kadar önemli değil, diziler konusunda inceliyoruz.Burada dikkat etmeniz gereken, prototipteki fonksiyon kaynağına ulaşmamız ve bu fonksiyonu `call` methodu ile çağırmamız.`call` nesnesine gönderdiğimiz argümanlardan birincisi etki alanını belirliyor, diğeri ise `slice` methodunun argüman tünelinden beklediği ilk değer. Gördüğünüz gibi, `call` methodu `apply`den farklı olarak argümanları topluca liste olarak almıyor, ilk argümanı etki alanı değeri kabul edip diğer bütün argümanları çalıştırdığı fonksiyona aktarıyor.

Call ile kalıtım

Kalıtım ilişkileri kurmak için `apply`'den daha çok tercih edilen methoddur.Mantık olarak pek farklılık yok, istenen objeye "scope" ve argümanlar gönderiliyor:

```
>>> var kisi = function(adsoyad,yas,dogumyeri){
          this.adsoyad = adsoyad;
          this.yas = yas;
          this.dogumyeri = dogumyeri;
      }
>>> var oyuncu = function(adsoyad,yas,dogumyeri,takim,skor){
          this.takim = takim;
          this.skor = skor;
          kisi.call(this,adsoyad,yas,dogumyeri);
       }
>>> var kemal = new oyuncu("kemal akın",15,"antalya","maviler",3);
>>> var ege = new oyuncu("ege akın",10,"izmir","kırmızılar",2);
>>> kemal.takim
maviler
>>> ege.dogumyeri
izmir
```

Örnekte gördüğünüz gibi, `apply` ile temel farklılık `call` methodunun argümanları serbest biçimde göndermemizi sağlaması.

Prototype

Prototype'ı bir binanın iskeletine benzetebiliriz, hatta bu benzetmeye tahmin ediyorum birkaç yüz dökümanda daha rastlayabilirsiniz çünkü `prototype` benzetmeden daha öte, fonksiyonların iskeletidir.

Prototype değerlerinin tipi objedir.Yani bir objeyi alıp, bir fonksiyonun iskeleti olarak kullanmak mümkündür.Tabii bunun tam tersi durum da sözkonusu.

Fonksiyonların klonlanarak objeye dönüştürülmesini anlatırken bir detayı atlamıştık; objenin özellikleri fonksiyonun tanımlandığı blokta olmak zorunda değil.Oluşturduğumuz fonksiyonun iskeletine ulaşabilir ve çeşitli eklemeler yapabiliriz.Şimdi adım adım bir fonksiyon yazalım ve konsolun döndürdüğü değerleri yorumlayalım:

```
>>> var kisi = function(adsoyad,yas,dogumyeri){
          this.adsoyad = adsoyad;
          this.yas = yas;
          this.dogumyeri = dogumyeri;
      }
>>> kisi.prototype
Object: There are no properties to show for this object.
```

Örnekte `kisi` adında bir fonksiyon oluşturup içine adsoyad,yas,dogumyeri özelliklerini yazdık.Ardındaki satırda `prototype`'ı incelediğimizde, boş bir objenin döndüğünü görüyoruz.Bu durumdan şunu çıkarmalıyız, iskeletin inşası prototype objesi dışında yapılamaz.Veya şöyle özetleyelim, fonksiyonun içinde iskelet oluşturamayız.

```
>>> kisi.prototype.meslek = null;
```

Yukaridaki örnekte `prototype` objesinin altında bir alan açarak, `kisi` fonksiyonuna meslek adında bir özellik ekledik.

```
>>> kisi.prototype
```

```
Object: (meslek = null)
>>> selcuk = new kisi("selçuk koçulu",48,"mardin");
>>> selcuk.meslek
null
```

Bu örnekteyse `selcuk` adında bir obje oluşturduk ve meslek özelliğini sorguladık.Dönen sonucun `null` -bu bizim belirlediğimiz bir varsayılan değerdi- olduğunu görüyoruz.

Buradaki problem, iskelete sonradan eklenen meslek özelliğini `constructor`'a nasıl dahil edeceğimiz.Bu iş için `constructor`'ın yeniden yazılması kaçınılmazdır;

```
>>> var kisi = function(adsoyad,yas,dogumyeri,meslek){
          this.adsoyad = adsoyad;
          this.yas = yas;
          this.dogumyeri = dogumyeri;
          this.meslek = meslek;
      }
>>> fikretkizilok = new kisi("fikret kızılok","60","İstanbul","Müzisyen");
>>> fikretkizilok.meslek
"müzisyen"
```

Bu örnekte dikkat etmemiz gereken, `kisi` fonksiyonunun eski iskeleti devralması.Peki bu durumda henüz tanımlanmamış fonksiyonların iskeleti oluşturulabilir mi? Evet, oluşturulabilir.

Bunu denemek için yeni bir fonksiyon yazalım:

```
>>> kargo.prototype.alan = null;
>>> kargo.prototype.gonderen = null;
>>> kargo.prototype.kargoicerigi = null;
>>> function kargo(alan,gonderen,kargoicerigi){
          this.alan = alan;
          this.gonderen = gonderen;
          this.kargoicerigi = kargoicerigi;
}
>>> kargo.prototype.ucret
0
```

Görüldüğü gibi, `undefined` değeri olan bir tanıtıcı altında iskelet oluşturduk ve ardından bu tanıtıcıya `fonksiyon` değeri verdik.Son satırda ise, tanıtıcı değer almadan oluşturduğumuz iskeletin çalışır durumda olduğunu onayladık.Biraz önce yazdığımız `kisi` nesnesinden de faydalanarak `kargo`'nun kullanımına bir örnek verelim:

```
>>> var azerkoculu = new kisi("azer koçulu",20,"kaş","web geliştirici");
>>> var billgates = new kisi("bill gates",53,"seattle","işadamı");
>>> var yeniKargo = new kargo(azerkoculu,billgates,"mozilla stickers");
```

Prototype ve Browser Nesneleri

Javascript'in disable olmadığı durumları saymazsak, browserlar her HTML belgesinde javascript komutlarının çalışmasını sağlayan bir zemin oluşturur.String,Number,Function,Array,Object nesneleri browser sözettiğimiz çekirdekte yer alan nesnelerden birkaçıdır.

Yazdığımız her sayı Number objesinden, her yazı String objesinden türer.Ve tahmin edeceğiniz gibi, bu değerlerin methodları, çekirdekteki nesnelere bağlıdır.String nesnesinin methodlarından birini deneyelim:

```
>>> String
String()
>>> String.prototype.replace
replace()
```

String nesnesinin prototype objesindeki replace fonksiyonuna erişip kontrol ettik.Şimdi de oluşturacağımız string değerinde `replace` methodunun kullanımını inceleyelim:

```
>>> "Merhaba Dünya".replace
replace()
```

ECMAScript standartını kullanan dillerden biri de Actionscript'tir.Bu dil de String,Number gibi objeler oluşturur fakat geliştiricinin bu nesneleri değiştirmesi kısıtlanmıştır. Neyse ki javascript için böyle bir durum söz konusu değil, browser nesnelerini dilediğimiz gibi değiştirebilmekteyiz.Örnek olarak Number objesine bir method ekleyelim:

```
>>> Number.prototype.topla = function(sayiDegeri){
          return this+SayiDegeri
}
```

Number nesnesinin iskeletine "topla" adında bir fonksiyon ekledik.Böylece, tüm sayılar için pratik bir toplama methodu yaratmış olduk, deneyelim:

```
>>> (4).topla(8)
12
>>> 4.0.topla(8)
```

```
12
```

Aklınıza şu soru gelmiş olmalı: Peki ondalık hanelerin nokta işaretleriyle methodlarınki karışmaz mı? Evet karışır.Bunun için sayılar parantez içine alınır, veya ondalık hanesinden sonra method yazılır.

Örnekte gördüğünüz gibi, Number sınıfından türeyen sayı değerlerinin altında topla methodu kendiliğinden gelmektedir.

Küçük bir ipucu: `prototype` Obje tipindedir yani Object sınıfından türemiştir.Şimdi browser'ımızı heyecanlandıracak birkaç deneme yapalım:

```
>>> Object.prototype.deneme = "foo bar"
>>> "".deneme
"foo bar"
```

Örnekte, Object sınıfının iskeletine bir ekleme yaptık ve böylece Object sıfından türemiş olan tüm iskeletlere değişikliği uyguladık.Yani, deneme fonksiyonu tüm iskeletlere eklendi.Bir sonraki satırda oluşturduğumuz string değeri kontrol ettik ve bu durumu doğruladık.

Protype'lar hakkında genel bilgi sahibi oldunuz.Obje türlerini inceledikten sonra prototype'lara tekrar dönüp deney yapmanızda fayda var.

Caller

Fonksiyonların nasıl çağrıldığını anlamak için, alt özellikleri olan "caller" kontrol edilir, eğer herhangi bir fonksiyon içinde çağrılmadıysa, "null" değerini döndürür.Bu özellik "fonksiyonAdı.caller" biçiminde kullanılır:

```
>>> var merhaba = function(){
          return merhaba.caller;
      }
>>> merhaba()
null
>>> (function(){ return merhaba() })()
(function(){ return merhaba() })()
```

Örnekte, "merhaba" adında, kendisini çağıran fonksiyonu döndüren bir fonksiyon oluşturduk.Global blokta çağırdığımız satırda "null" değer dönerken, diğerinde fonksiyonun kendisi döndü.

Örnek olarak, yazdığımız fonksiyonun başka bir fonksiyon içinden çağrılmasını engelleyebiliriz;

```
>>> var merhaba = function(){
           if(merhaba.caller!=null)
               return "Bu fonksiyona başka bir fonksiyonun içinden erişemezsiniz."
          return "Merhaba!"
```

Dizi konusu gibi, ileride detaylıca inceleyeceğimiz konulardan biri de koşullar.Yukarıdaki örneğin ilk iki satırında, "merhaba.caller" değerini kontrol ettik ve eğer "null" değilse, yani bir başka fonksiyonun içinden çağrı yapıldıysa, "Merhaba" yerine uyarı metni döndürdük."return" komutu verildikten sonra yorumcunun fonksiyondan çıktığını hatırlayalım.

Internet Explorer ve eval() İstisnası

"Caller" Internet Explorer, Mozilla gibi popüler tarayıcıların desteklediği, aslında standart dışı bir özelliktir.Fakat Internet Explorer "eval()" komut bloğunda çalıştırılan fonksiyonlar için de caller değerini "null" döndürmektedir.Kontroller yapılırken bu durumu dikkate almak gerekir.

Bölüm Sonu Pratikleri

SORULAR

Soru 1) Dinamik fonksiyon yazımına örnek vermek için, sayfa başlığını statik string olarak döndüren "sayfaBasligi" adında fonksiyon yazınız.

Soru 2)Javascript yazarken sıkça karşılaşılan sorunlardan biri, çalıştırılan fonksiyonların istenen scope'a sahip olmamasıdır.Bu tür problemlerin çözümüne örnek vermek için, herhangi bir fonksiyonun scope değerini herhangi bir objeyle değiştiriniz.

Soru 3) Önceki sorudan devam.Scope problemlerinin çözümünü daha pratik hale getirecek "curry" adında bir fonksiyon methodu yazınız.

CEVAPLAR

Cevap 1) Fonksiyonlar `Function` sınıfından türer.Klasik yazılışın yanında `new` operatörüyle Function sınıfından fonksiyon oluşturulabilir;

```
var fonksiyon = new Function(argüman1,argüman2,.....,fonksiyonKodları);
```

Buna göre sorunun cevabı;

```
var sayfaBasligi = new Function("return '"+document.title+"'");
```

Cevap 2) Bu problem hakkında detaylı bilgiyi `call` ve `apply` başlıklarında bulabilirsiniz.Sorunun yanıtı ise şöyle:

```
var deneme = function(){ alert(this.adres); }
deneme(); // undefined
deneme.call({ "adres":"google.com" }); // google.com
```

Deneme adında bir fonksiyon oluşturduk ve ikinci satırda çalıştırdık.Fonksiyonumuz çalıştığında, kendisine ait "adres" adında bir methodu aradı ve

böyle bir method tanımlanmadığından undefined değer döndü.Üçüncü satırda ise deneme fonksiyonunun `call` methoduna bir obje gönderdik.Deneme fonksiyonu bu kez çalıştığında kendisinin etki alanını gönderdiğimiz

Bu problemin nasıl aşılacağını call ve apply başlıklarında detaylarıyla anlattım, şimdiyse bu işi oldukça pratik bir hale getirelim. Örneğin, herhangi bir fonksiyonu şu biçimde çağıralım ve istediğimiz scope'u uygulayalım:

```
>>> var deneme = {
          kontrol: false
}
>>> deneme.kontrol
false
```

`kontrossl` adında bir boolean içeren deneme objesi oluşturduk, şimdi bu objeye iki ayrı yolla erişelim:

```
>>>var deneme = {
          kontrol: true
};
>>>var kontrolDegeri = function(){ return this.kontrol; };
kontrolDegeri(); // undefined döner
kontrolDegeri.call(deneme); // "true" döner
```

Kontrol değerini ilk çağırışımızda bize değer dönmedi ancak ikinci çağırma biçiminde istediğimiz değer döndü.Burada call methoduna scope değeri olarak deneme objesini gönderdik ve böylece fonksiyonun içindeki this değeri, deneme objesine referanslandı.İlk satırda ise fonksiyonun içeriği scope olarak window objesine eriştiğinden, istediğimiz değer dönmedi.

Cevap 3) Bir veri türüne method atamak için, o veri türünün constructor'ı üzerinde işlem yapmak gerekir.Daha ayrıntılı bilgi almak için, prototype başlığına dönün.

Soruda bizden isteneni daha açık hale getirelim; her fonksiyonun "curry" adında bir methodu olacak ve biz bu methoda istediğimiz scope'u gönderdiğimizde, bize yeni bir fonksiyon değeri gönderecek.Elde ettiğimiz yeni fonksiyonu dilediğimiz gibi çalıştırabileceğiz, yani fonksiyonumuz apply veya call işlemlerindeki gibi, scope gönderilince hemen çalışmayacak.Önce örnek kullanımı inceleyin:

```
>>> var kontrolDegeri = kontrolDegeri.curry(deneme);
>>> kontrolDegeri(); // true değeri döndü
```

Bu kez`curry` methodunu kullanarak, deneme objesini kontrolDegerinin scope'u olarak belirledik ve yeni bir fonksiyon değeri elde ettik.Eğer kontrolDegeri ve deneme objesinin nerede oluşturduğumu kaçırdıysanız, ikinci sorunun çözümüne gözatın.

`curry` Methodunun yazımında öncelikle dikkat etmemiz gereken, bir fonksiyon değeri döndürecek olmamız, bir diğer önemli unsursa, argümanları da kullanacak olmamız.call ve apply başlıklarında, argümanların nasıl kullanıldığını anlatmıştım, tekrar gözatabilirsiniz.Yazmaya başlayalım:

```
>>> Function.prototype.curry = function(scope){
          var fn = this;
          var scope = scope||window;
          return function(){
                    fn.apply(scope,arguments);
          }
}
```

Gördüğünüz gibi pek karmaşık bir çözüm olmadı.İlk satırda, fn adında bir değer tanımladık bunun sebebi, geri döndürdüğümüz fonksiyon değerinin içinden, üst fonksiyona erişemeyişimiz.Ancak üst fonksiyon bloğunda tanımladığımız değerler, birbirine erişebilir.Bu yüzden, scope değerini de tekrar tanımladık.Genellikle mantıksal sorgulamalarda kullandığımız "veya" ifadesinin görevi ise, argüman olarak gelen scope değeri null,false veya undefined olduğunda, window değerini döndürmektir.Daha sonraki satırda ise, bir fonksiyon değeri döndürüyoruz.Yeni fonksiyonun içeriğine baktığımızda bu kez call değil apply kullandığımızı görürsünüz.Bunun sebebi, call methodunun argümanları array tipinde değil, tek tek alması.`apply` ise bunun tam tersine, ilk argümandan sonra bir argümanları içeren dizi değeri bekler.Gönderdiğimiz arguments değeri, yeni fonksiyona gelen argümanları içeren dizi değeridir.Bunun hakkında daha fazla bilgi istiyorsanız, argümanlar başlığına dönün.

Diziler: array

İstenen veri parçalarını tür ayrımı yapmadan bir araya getirip gruplayan veri türüdür.ECMAScript dizi elemanlarının tanıtıcılarını sıra numarası olarak tanımlar.Değer atama işlemi köşeli parantezler içinde dizi elemanlarını virgülle ayırarak yapılır:

```
var tekSayi = [1,'Bir',3,'Üç',5,'Beş']
```

Örnekte integer ve string türündeki verilerle bir dizi oluşturduk.Fakat istenirse, bir dizi elemanının tipi herhangi bir veri türü olabilir, örneğin `number`, `boolean`, `function`, `object` veya yine `array` türünde bir dizi elemanı oluşturabiliriz.

Program içinde diziye erişmek istendiğinde, dizi elemanlarının anahtarlarını kullanmamız gerekir.Burada anahtar, dizi elemanının sıra numarasıdır ve sıra numaraları sıfırdan başlar, yani son dizi elemanının sıra numarası dizi uzunluğundan bir eksiktir.Örnek olarak, az evvel oluşturduğumuz tekSayi dizisinin ikinci elemanına erişelim:

```
>>> tekSayi[1];
'Bir'
```

Pek çok kaynakta dizilerin şu biçimde tanımlandığına rastlayabilirsiniz:

```
tekSayi = new Array("Bir","İki","Üç");
```

Fakat görüldüğü gibi bu pratik bir tanımlama biçimi değildir. Az evvel herhangi bir türde veri tutulabildiğine değinmiştik, bu konuda bir örnek hazırlamak için, fonksiyon bilgimizi de kullanarak toplama ve çıkarma yapan bir dizi oluşturalım:

```
var islem = [
function(sayi0,sayi1){
return sayi0+sayi1;
},
function(sayi0,sayi1){
return sayi0-sayi1;
}
]
```

Oluşturduğumuz dizi birbirinden virgülle ayrılmış iki ayrı fonksiyondan oluşuyor, birinci dizi elemanı toplama işlemi yaparken diğeriyse çıkarma işlemi yapıyor.Şimdi bu dizinin elemanlarına tek tek erişip işlem yaptıralım:

```
>>> islem[0](4,3)
7
>>> islem[1](4,3)
1
```

Diziler, `Array` sınıfından türediğinden ötürü, her dizi aynı zamanda alt method ve değerleri olan bir objedir.Ve doğal olarak, dizi objeleri `Array` iskeletini taşır ve bu iskelete serbestçe ekleme/çıkarma yapılabilir.Bu konuda daha detaylı bilgi ve örnekler için fonksiyonlar konusundaki prototype altbaşlığını inceleyin.

Dizi değerlerin mevcut alt methodlarını inceleyelim:

concat

İki veya daha fazla diziyi birleştirmek için kullanılır, örnek:

```
>>> ["deneme"].concat(["merhaba"],["dünya"])
["deneme","merhaba","dünya"]
```

`concat` browser'ın fazladan bir dizi daha oluşturmasına neden olduğundan pek tercih edilen sağlıklı bir method değildir.Yerine daha kullanışlı olan "push" methodu kullanılilir.Birkaç başlık sonra push methodunu inceleyeceğiz.

join

Verilen ayracı kullanarak diziyi string değere çevirir.Örnek,

```
>>> ["merhaba","dünya"].join("#")
"merhaba#dünya"
>>> ["merhaba","dünya"].join("!")
"merhaba!dünya"
```

length

Dizinin uzunluğunu, yani ne kadar dizi elemanı tuttuğunu veren değer.Örnek;

```
>>> var deneme = ["merhaba","dünya"]
>>> deneme.length;
2
```

pop

Dizinin son elemanını siler ve sildiği değeri döndürür.

```
>>> var deneme = ["merhaba","dünya"]
>>> deneme.pop();
"dünya"
>>> deneme
["merhaba"]
```

push

Argüman tünelinden gelen değerleri dizinin sonuna ekleyen method.Geriye dizinin yeni uzunluğunu döndürür.

```
>>> var deneme = []
>>> deneme.push("merhaba","dünya");
2
>>> deneme
["merhaba","dünya"]
```

Concat yerine Push methodunun kullanımı

Push fonksiyonunu concat yerine kullanmak için, fonksiyonların apply methodundan ve prototype objesinden faydalanılır.Bu iki konuda detaylı bilgiyi fonksiyonlar başlığında bulabilirsiniz.Gelelim örneğe;

```
>>> var dizi1 = [1,2];
>>> var dizi2 = [3,4];
>>> var dizi3 = [5,6];
>>> var dizi4 = [7,8];
>>> dizi1.concat(dizi2);
[1,2,3,4]
>>> Array.prototype.push.apply(dizi3,dizi4);
4
>>> dizi3
[5,6,7,8]
```

Örnekte concat methodunun kullanımını hatırladık ve ardından apply methodunu çağırarak dizi3 ile dizi4 ü birleştirdik.

slice

Dizi değerlerinin içinden istenen index aralıklarını çağırmayı sağlar;

```
>>> [3,6,9,12].slice(1,3)
[6, 9]
>>> [3,6,9,12].slice(1,4)
[6, 9, 12]
>>> [3,6,9,12].slice(0,2)
[3, 6]
```

Bölüm Sonu Pratikleri

ECMAScript standartına göre, diziler ve nesneler farklı değişkenlere aktarılamaz, referanslanır.Bunu daha iyi anlamak için bir örnek yapalım:

```
>>> var a = [3,2,1];
>>> var b = a;
>>> b
[3, 2, 1]
>>> b.push(0);
4
>>> b
[3, 2, 1, 0]
>>> a
[3, 2, 1, 0]
```

Örnekte, b değerine ilk oluşturduğumuz a değerini atadığımızda, iki değer birbirine referanslandı.Bunu anlamak için b dizisine yeni bir sayı ekledik, a dizisini kontrol ettiğimizde aynı sayının a'ya da eklendiğini gördük.Aslında b'ye eklendiğimiz a'ya eklenmedi, daha doğrusu biz hem a'ya hem b'ye ekleme yaptık.

Bu özelliğin açtığı problemlerle ilgili bir örnek yapmak için dizi tipinde alt değeri olan bir sınıf tanımlayalım:

```
>>> var deneme = function(){}
>>> deneme.prototype.dizi = [];
[]
>>> var nesne1 = new deneme();
>>> var nesne2 = new deneme();
>>> nesne1.dizi.push(1);
1
>>> nesne1.dizi
[1]
>>> nesne2.dizi
[1]
```

Örnekte gördüğünüz gibi, deneme iskeletinden türeyen tüm objelerin dizi tipindeki alt değeri birbirine referanslı.Biri değişince diğeri de değişmekte.Şimdi sorulara geçelim:

SORULAR

Soru 1) Dizilerin klonlanmasını sağlayan, `dizi.clone()` şeklinde çalıştırılabilen bir method yazın.Dizilerin içiçe olabileğini dikkate alın.

Soru 2) Birinci sorunun cevabını, fonksiyonların iskeletinde yaşanan referans probleminin çözümü için uygulayın.

CEVAPLAR

Cevap 1) Bu problemin çözümü için bu dökümanda bahsetmediğim döngüleri ve kontrol ifadelerini biliyor olmalısınız.Sınıfların iskeletine method ekleme işlemi hakkında bilgi almak için Fonksiyonlar konusundaki prototype başlığına dönün. Problemin cevabına geçelim:

```
Array.prototype.clone = function(){
          var gecici = [];
          for(var i=0; i<this.length; i++){
                    gecici.push(
                              this[i] instanceof Array? this[i].clone():this[i]
                    );
          }
          return gecici;
```

```
    }
```

Örnekte, `gecici` adında yeni bir dizi oluşturduk ve dizi elemanlarını tek tek bu yeni diziye ekledik.Ve yeni diziyi geri döndürdük.Bu örnekteki en önemli satır, push fonksiyonuna veri gönderirken yaptığımız kontrol:

```
this[i] instanceof Array? this[i].clone():this[i]
```

Eğer dizinin elemanı da bir diziyse, onu da klonlamamız gerekir.Bunun için sıradaki dizi değerinin Array sınıfından türeyip türemediğini `instanceof` operatörünü kullanarak kontrol ettik.

Cevap 2) Eğer iskelete dışarıdan dizi değeri eklediysek, fonksiyonun constructor'ında bu değeri klonlamamız gerekir.

```
>>> var deneme = function(){
          this.dizi = this.dizi.clone();
}
```

Nesneler: object

Objeler, tıpkı diziler gibi, istenen veri parçalarını gruplayan veri türüdür fakat dizi elemanlarının aksine, nesnelerdeki elemanlar için tanıtıcı belirlenir.

Dizilerdeki köşeli parantezlerin yerini süslü parantezler alır.Nesne elemanları birbirlerinden yine virgülle ayrılırlar.Fakat bu kez her veri parçası için bir tanıtıcı belirlenir, bu işlem için "var" ifadesi ve eşit işareti kullanılmaz, tanıtıcı yazılıp hemen ardında ":" kullanılır. Örneğin,

```
>>> var merhaba = { ingilizce:"hello", ispanyolca:"holla" }
```

Nesnelerin elemanları birer method olarak kabul edilir, dizilerde görülen çağırma işlemi dışında, diğer programlama dillerindeki sınıf yapılarından alışık olduğumuz method biçimindeki çağırma işlemi de yapılabilir, her iki biçimde de çağırma işlemi yapalım:

```
>>> merhaba.ingilizce
hello
>>> merhaba["ispanyolca"]
holla
```

Method biçimindeki çağırma işleminde tanıtıcının kullanım şekli özel karakter yazımına ve satıriçi dinamik kullanıma uygun olmaması, dizi biçimindeki tanımlamaNes şeklinin önemini artırır;

```
>>> var meslekler = { "azer koçulu":"öğrenci", "nuray koçulu":"öğretmen" };
>>> meslekler.azer koçulu // hatalı tanıtıcı yazdık.
SyntaxError: missing ; before statement
>>> meslekler["azer koçulu"]
öğrenci
```

Nesne tanımlandıktan sonra yeni bir eleman ekleme-çıkarma işlemi yapmak için dizilerdeki push ve pop fonksiyonları yerine, yine dizilerden alışık olduğumuz tanımlama biçimi kullanılır:

```
>>> var meslekler = { "azer koçulu":"öğrenci", "nuray koçulu":"öğretmen" };
>>> meslekler["arda koçulu"] = "öğrenci"
```

Bu veri tipinin esnek yapısı sayesinde, ECMAscript'in OOP eksikliği tamamlanır.Dizilerdeki basit işlem dizisini, nesneleri kullanarak tekrar yazalım:

```
>>> var islem = {
            topla: function(sayi0,sayi1){
                        return sayi0+sayi1;
                    },
            cikar: function(sayi0,sayi1){
                        return sayi0-sayi1;
                    }
            }
>>> islem.topla(2,3);
5
>>> islem.cikar(2,3);
-1
>>> islem.cikar(islem.topla(2,3),1);
4
```

Yaptığımız örnekte, "islem" adında bir nesnenin içinde, fonksiyon tipinde "topla" ve "cikar" adlarını taşıyan iki method oluşturduk, ve birkaç değerle test ettik.

Örnekteki kodlama türünün, işlem yaptırdığımız değerleri gruplama dışında özel bir işlevi yok.Peki diğer programlama dillerinde alışık olduğumuz "new" operatörüyle oluşturulan (klonlanan da denilebilir) sınıflar oluşturulamaz mı? Elbette oluşturulur, örnekten ilerleyelim:

```
>>> var islemci = function(sayi0){
              return {
                    topla: function(sayi1){
```

```
                                        return sayi0+sayi1;
                                    },
                        cikar: function(sayi1){
                                        return sayi0-sayi1;
                                    }
                        }
            }
>>> var islem1 = new islemci(5);
>>> islem1.topla(3); // 5+3 = 8
8
>>> islem1.cikar(3); // 5-3 = 8
2
```

Uyguladığımız son iki örnekteki farka dikkat edelim, ilk örnekte işlem yapan fonksiyon tipindeki verileri saklayan bir obje vardı, ikinci örnekteyse, birinci örnekte saklanan objeyi "return" operatörüyle geriye döndüren bir fonksiyon var.

Örnekteki "islem1", 'islemci`'nin argüman tünelinden aldığı "5" değeri için döndürdüğü "topla" ve "cikar" fonksiyonundan oluşan nesne değerini aldı.Ardındaki satırlarda bu fonksiyonları test ettik.

Bölüm Sonu Pratikleri

Diziler için geçerli olan referans problemi objeler için de geçerlidir.İlk soru aynı konudan geliyor:

SORULAR

1) Object fonksiyonunun altında, argüman tünelinden aldığı objeleri klonlayan Class adında bir method yaratın.Eğer objenin içindeki dizi değerleri varsa, onları da klonlamayı unutmayın.Zorlandığınızda diziler konusundaki çözümden faydalanabilirsiniz.

2) Obje döndüren bir fonksiyon yaratın.Fakat bu objenin Init adında bir methodu olsun ve fonksiyonun içinde döndürme işleminden önce çalışsın.

CEVAPLAR

1)

```
Object.Clone = function(){
          var gecici = [];
          for(var i=0; i<this.length; i++){
                    gecici.push(
                              this[i] instanceof Array? this[i].clone():(
                                        typeof this[i]=="object"?
                                        Object.Clone(this[i]):
                                        this[i]
                              )
                    );
          }
          return gecici;
}
```

Diziler başlığında kullandığımız klonlama fonksiyona bir kontrol daha ekledik:

```
typeof this[i]=="object"?Object.Clone(this[i]):this[i]
```

Burada aslında değeri instanceof ile Obje sınıfından türeyip türemediğini de kontrol edebilirdik.Fakat browser bu kontrolde fonksiyonlar için de pozitif değer döndürecektir.

2) Bu problemin çözümü aslında oldukça basittir, objeyi önce tanımlar sonra return operatörüyle geri döndürürüz.Fakat bu çözüm düşünmeyi pek gerektirmez, biraz daha düşünüp daha iyi yöntem bulmalıyız.Bu tip kolay problemleri nasıl daha pratik şekilde çözebileceğimizi, tekniğimizi ilerletmek için önemsemeliyiz.Çözüme geçelim:

```
function deneme(){
          return ({
                    "Init": function(){
                              alert("Init fonksiyonundan merhaba!");
                              return this;
                    }
          }).Init()
}
```

Parantez bloklarını kullanarak biraz daha iyi bir çözüm kullandık.Buradaki çözümün özü Init fonksiyonunun içinde bulunduğu objeyi döndürmesi.

Fonksiyonları, dizileri ve objeleri pekiştirdiğimize göre, İleri OOP Teknikleri konusuna geçebiliriz.

İleri Seviye OOP Teknikleri

bu başlık altındaki kod örneklerini ingilizce hazırladım.bunun sebebi kod standartlarını daha iyi anlamanızı sağlamak

Bu başlığa gelmeden önce, klasik obje yapılarını ve yeni popülerleşen obje tekniklerini pratikleştirdik.Neydi bu teknikler, birincisi fonksiyonlar başlığında gördüğümüz sınıf oluşturma tekniği.İkincisi, objeler başlığında incelediğimiz, çok daha pratik olan sınıf oluşturma tekniği.Fonksiyonlar

başlığındaki yapılara tekrar gözatalım:

```
function deneme(){
          this.özellik1 = değer;
}
var yeni = new deneme();
```

```
function deneme(){
}
deneme.prototype.özellik1 = değer;
var yeni = new deneme();
```

```
function deneme(){
}
deneme.prototype = {
          "özellik1":değer;
}
var yeni = new deneme();
```

Objeler başlığından çıkaracağımız yapılarsa şunlar:

```
var deneme = {
          "özellik1":değer;
}
var yeni = Object.Clone(deneme);
```

```
function deneme(){
          return {
                    "özellik1":değer
          }
}
```

Her iki tekniğin de profesyonel web projelerinin pek çoğunda kullanıldığı görülür, ancak kod standartına göre yazım stilleri biraz daha değişim gösterecektir.Örnek olarak şu an çalıştığım şirketin kod standartına göre bir sınıf oluşturalım.

Obje: Banka Hesabı **Değerler:** ad soyad, bakiye, borç limiti **İşlemler:** Para Gönder, Para Yatır

```
function account(){
}
account.prototype = {
         "_balance":0, "_limit":0, "_name":"",
         "getBalance":function(){
                   return this._balance;
         },
         "setBalance":function(value){
                   this._balance = value;
         },
         "getLimit":function(){
                   return this._limit;
         },
         "setLimit":function(value){
                   this._limit = value;
         },
         "getName":function(){
                   return this._name;
         },
         "setName":function(value){
                   this._name = value;
         },
         "SendMoney": function(value){
                   this.setBalance(this.getBalance()-value);
         },
         "Invest":function(value){
                   this.setBalance(this.getBalance()+value);
         }
}
```

Bu yazım biçimini anlamaya çalışalım, neden değişken adlarının önünde underscore işareti var? Neden her değişken için get/set methodları yazılıyor?

Underscore koymamızın nedeni browser'ın reserveli kelimeleriyle çakışma olmamasını sağlamak.Örneğin "class" adında bir değer oluşturmamız gerekirse, webkit ve internet explorer yazdığımız kodu durduracaktır.Bunun için underscore ile işareti kullanırız.

Peki, bir değeri isterken, veya ayarlarken, başka bir işlem yapmamız gerekiyorsa ne yaparız.Veya esas değeri, istenen şekle göre süzerek vermemiz gerekiyorsa? Bu tip durumlarda standart bozulur, veya kodun akışı değişir.Böyle durumlar için get/set özellikleri yazmak daha doğru olacaktır.Peki, oluşturduğumuz bu objede get ve set fonksiyonları içinde süzme vb. işlemler yapılıyor mu? Hayır.Bu unsuru aklımızda tutalım, birazdan tekrar düşüneceğiz.

Son iki satırda işlem yapan fonksiyonları, baş harfleri büyük şekilde yazdık.Çünkü ortam değerlerinden ayrılmaları gerekiyordu.

Elimizde iki önemli unsur var: birincisi önemsiz yere get ve set kullanmamak.Yani gerektiğinde yazmalıyız.Fakat yazmasak ta standart bozulmamalı, makine kendisi bizim yerimize rutin işlemleri yerine getirmeli.Diğer husus, işlem yapan fonksiyonları ortam değerlerinden ayırmak.Bu iki unsuru toparlarsak nasıl bir obje yapısına ihtiyacımız olduğuna karar verelim.Genel kullanım şekli şöyle olsun:

```
var azer = new account();
azer.environment.setName("Azer Koçulu");
azer.environment.setBalance(1000);
azer.environment.setLimit(2000);
// işlem yapan fonksiyonları kullanalım:
azer.sendMoney(500);
```

Örnekte gördüğünüz gibi, ortam değişkenlerini alt objeye yerleştirerek işlem yapan fonksiyonlardan arındırıp biraz daha düzenledik.

Bu kullanım örneğinden ilerlemek istediğimizde karşımıza çıkacak ilk sorun, environment objelerinin referanslanarak birbirine bağlı hale gelmesidir.Sadece environment objesini düşünmeyelim, normalde prototype üzerinden sınıf oluşturmaya çalıştığımızda, dizi ve obje türü değerlerin hepsi birbirine referanslanacak ve programımızı bozacaktır.

Bu problemleri aşmak için istediğimiz gibi sınıf oluşturmamızı sağlayan bir arabirim gereklidir.

Bölüm Sonu Pratiği

Her sınıfın Constant,Environment,Event adında alt objeleri olmalı.Program yazarken, sabitleri, ortam değişkenleri ve event yakalayıcıları bu şekilde ayırmalıyız.Ve tabii, gerek olmadığında get/set yazmamalıyız, bizim yazmadığımız get ve set methodlarını kod standartının bozulmaması için program bizim yerimize oluşturmalı.En önemlisi de, yapacağımız arabirim referans problemini otomatik olarak çözmeli.Arabirimin örnek kullanımını inceleyin:

```
var account = new Interface;
account.body = {
          "Invest":function(){
          },
          "SendMoney":function(){

          }
}
account.environment = {
          "_balance", "_limit", "_name"
}
account = account.build();
```

Bu örneğe ek olarak, extend ve constructor adlı iki methoda ihtiyacımız var.extend, oluşturacağımız sınıf başka bir sınıfı kapsıyorsa kullanacağımız method olacak.consturctor ise, oluşturacağımız sınıfın constructor'ı olacak.Bu noktada obje referans problemini constructor'da çözdüğümüzü hatırlamalısınız.

Okuduğunuz döküman bu sınıfı yazabilmenizi sağlamaya çalıştı.Bu pratiği yaparken kod standartınızı geliştireceğinize inanıyorum, umarım döküman faydalı olmuştur. Sorunun cevabını mı merak ediyorsunuz :) Interface adını verdiğim sınıfı, hazırladığım Pi adlı javascript kütüphanesi için yapmıştım.İşte kodları:

```
var interface = function(){
        this.constants = {};
        this.environment = {};
        this.events = {};
        this.body = {};
}
interface.prototype = {
        "constant":{},
        "constructor":function(){},
        "environment":{},
        "event":{},
        "body":{},
        "extend":function(OBJECT){
                var body = Object.Clone(OBJECT.prototype);
                Object.Concat(this.body,body);
                Object.Concat(this.constants,body.constants);
                Object.Concat(this.environment,body.environment);
                Object.Concat(this.event,body.event);
        },
        "build":function(){
                var link = this;

                var fn = function(){

                        this.constants = Object.Clone(this.constants);
                        this.environment = Object.Clone(this.environment);

                        this.environment._parent_ = this;
                        this.event._parent_ = this;
                        link.constructor.apply(this,arguments);
                };
```

```
                        for(var item in this.environment){
                                if(item.substring(0,1)!="_")continue;
                                var name = item, title = name.substring(1,2).toUpperCase()+name.substring(

                                if(Boolean(this.environment["get"+title])==false)
                                        this.environment["get"+title] = new Function("return this."+name);
                                if(Boolean(this.environment["set"+title])==false)
                                        this.environment["set"+title] = new Function("VALUE","this."+name+
                        }

                        this.body.constants = this.constants;
                        this.body.environment = this.environment;
                        this.body.event = this.event;
                        fn.prototype = this.body;
                        return fn;
                }
        };
```

Sizin için faydalı bir döküman olduğunu umut ediyorum.Sabrınızın ve gayretinizin karşılığını almanızı dilerim.

Azer Koçulu