# Callback

## Asenkron / Senkron Programlama Nedir?

Senkron (Synchronous) programlamada işlemler sırayla gerçekleşir. Başka bir deyişle bir işlem bitmeden diğerine geçiş yapılmaz. Aktif olan devam ederken sıradaki işlem bloklanır.

console.log("Hello,");

const syncExample = () => {

  console.log("How are you?");

};

syncExample();

console.log("World.");

Bildiğiniz gibi Javascript bir **single-threaded**programlama dilidir. Yani kodu satır satır okur ve sırayla işleme sokar. Bu nedenle yukarıda verilen örneğin çıktısı şu şekildedir:

Hello,  
How are you?  
World.

Asenkron (Asynchronous) programlama paralel programlamayı mümkün kılar. Başka bir deyiş ile main akışı engellemeden kod yazabiliriz.  
Şimdi az önceki örneğimizi asynchronous şekilde düzenleyelim.

console.log("Hello,");

const asyncExample = () => {

  setTimeout(() => {

    console.log("How are you?");

  }, 3000);

};

asyncExample();

console.log("World.");

Sizce output nasıl olmalı? Gelin beraber göz atalım.

Hello,  
World.  
How are you?

Gördüğünüz gibi sıralama bu sefer farklı. Peki bu nasıl oldu?

Temel olarak, JavaScript derleyicisi asynchronous işlevle karşılaştığında, bu yürütme bağlamını WebAPI’ye gönderir ve kodun geri kalanı standart çalışma sistemi ile yani kodu satır satır okuyarak çalışmaya devam eder. Bu sırada WebAPI setTimeOut fonksiyonunun takibini yapar ve zamanı geldiğinde işlevi gerçekleşir. Böylece aynı anda iki görevi yerine getirebilir. Bu sayede yukarıdaki örnekte de görüldüğü gibi kod satır satır okunmasına rağmen 3. log daha erken basılmış olur. Çünkü **asyncExample**3 sn. sonra gerçekleşir.

Peki asynchronous fonksiyonun çıktısına main akışta ihtiyaç duyarsak ne yapacağız? Bu duruma da bir örnek ile göz atalım.

Let givenName;

setTimeout(() => {

  givenName = “ABC”;

  console.log(asyncExample());

}, 3000);

function asyncExample() {

  return `Execution of “asyncExample()” function, and givenName is “${givenName}”.`;

}

console.log(asyncExample());

console.log(“World.”);

Bunun çıktısı şu şekilde olur:

Execution of "asyncExample()" function, and givenName is "undefined".

  World.

  undefined

Execution of "asyncExample()" function, and givenName is "ABC".

setTimeout fonksiyonu yüzünden **asyncExample**fonksiyonu 3 saniye sonra gerçekleşti. Ancak o sırada **console.log(asyncExample(“ABC”));** çağırıldığı için sonuç **undefined**oldu. Peki bu sorunu nasıl çözeceğiz?

İşte bu sorunun cevabı bizim asıl konumuz olan Callbacks, Promises ve Async-Awaits yöntemlerini kullanmak. Gelin şimdi de beraber bu başlıklara göz atalım.

# Promise

## Nedir Promise?

Bir örnekle başlayayım. Diyelim ki bir hamburgerciye gittiniz ve kasada siparişinizi verdiniz. Kasiyer de size siparişinizi hazırladıklarında haber verilmek üzere bir elektronik alet teslim etti. Şu anda hamburger bizim için gelecekte elde edebileceğimiz bir değer. Elimizdeki aygıt da bu değere ulaşmamız için müessesenin bize verdiği sözün (işte ***promise***) bir göstergesi. Aygıt bildirim alıncaya kadar **bekleme**(***pending***) durumundadır. Bildirim geldiğinde ise ya hamburgerimiz **başarılı**(***resolved***) bir şekilde hazırlanmıştır, ya da beklenenin dışına çıkarak **başarısız**(***rejected***) olmuştur. Başarılı durumda afiyetle yemeğimize yumuluruz tabii ki. Başarısız durum biraz daha nahoş olacaktır. Kasiyerle kavga ederek veya yeniden sipariş vererek **hatalı durumu yönetmemiz** (***error handling***) gerekir.

## Callback mi, Promise mi?

Öncelikle hatırlayalım *callback* neydi diye. *Callback*, en basit haliyle herhangi bir fonksiyona parametre olarak verdiğimiz ve sonra geri çağıracağımız fonksiyonlara denir. İstenilen değere ulaştığında, bir şart gerçekleştiğinde (event) veya işlem sonlandığında görevini yerine getirir.

Programlama sürecinde Callback ile ilgili bazı sorunlar yaşandı:

* *Callback*’in beklenenden erken çağırılması
* *Callback*’in hiç çağırılmaması
* *Callback*’in beklenilenden az veya çok çağırılması
* Gerekli parametreleri doğru bir şekilde alamaması
* Hataların yutulması

*Promise*, *callback*’lerin sıkıntılı yönlerini düzeltmek amacıyla önerilmiş bir yapıdır. Sözdiziminden bahsettiğimizde daha net akıllarda oturacaktır, ancak şimdiden avantajlarından bahsetmekte fayda var. Şöyle ki:

* Promise istenilen görevi yerine getirdiğinde değeri değişmez (***immutable***)
* Sadece bir kere **başarıya**(***resolved***) ulaşır, veya **başarısız**(***rejected***) olur.
* Öngörülemeyen hatalar otomatik olarak *Promise*’i **başarısız**(*rejected*) sonuca götürür. Bu da hataları kontrol etme noktasında faydalıdır.
* Yapısı gereği, gelecekteki bir değerin göstergesi olduğundan daha güvenilirdir.

## Promise Sözdizimi

Gelelim anlatılan konseptlerin daha elle tutulur tarafına. Kodu görünce soyut kavramlara göre kendinizi daha rahat hissedeceksinizdir.

En basitinden bir promise örneği:

const sozVerdik = new Promise(function (resolve, reject) {

  if (herseyYolunda) {

    resolve('Her şey yolunda!');

  } else {

    reject('Bir sıkıntı var...');

  }

})

sozVerdik.then(function (cevap) {

  console.log(cevap) // 'İşlem tamam!' yazısını basar

}).catch(function (hata) {

  console.log(hata) // 'Bir sıkıntı var...' yazısını basar

})

* *Promise*’leri ***new*** öneki ile tanımlıyoruz.
* **Başarılı**(*resolve*) ve **başarısız**(*reject*) durumlarda çağırılacak iki fonksiyon ile birlikte oluşturuyoruz. (Sıralamayı değiştirmemek kaydıyla bu iki fonksiyona farklı isimler de verebilirsiniz ancak genelde bu isimler tercih edilmektedir.)
* *Promise*’leri bir değişkene atayabiliriz. (Örnekteki ***sozVerdik*** değişkeni gibi.)
* *Promise* beklenilen işlemi gerçekleştirdikten sonra yapılacak adımlar için ***.then()***fonksiyonu çağırılır. İçerisindeki fonksiyonun parametresi ***resolve()*** ile gönderilen parametredir.
* Eğer istek dahilinde ***reject()*** çağırıldığında veya öngörülemeyen bir hata sonucu *promise* başarısız olduğunda ***.then()*** fonksiyonu es geçilerek, ***.catch()*** içerisindeki fonksiyon çağırılır ve hatalı durumda yapılacak adımlar izlenir.

## Promise Zincirleri

Zincir diyerek kastettiğimiz şu: biri diğerini bekleyen asenkron işlemlerin arka arkaya çalıştırılması. Örneğin, cep telefonunda kullandığımız bir uygulamaya güncelleme geldiğinde, önce güncellemenin tamamlanmasını bekleriz, ardından uygulamadan içerik talep edebiliriz. *Promise*yapısı gereği asenkrondur ve uygulamadaki bekleyen diğer kodların çalışmasını bekletmez. Bu yüzden sıralı asenkron işlemlerin birbirini beklemesi için *promise* zinciri diyebileceğimiz yapılar oluşturmamız gerekir.

asenkronIslem()

  .then(sonuc => {

    return baskaAsenkronIslem(sonuc);

  })

  .then(zincirSonuc => {

    return zincirSonuc.json();

  })

  .catch(hata => {

    console.log(hata);

  });

* Birden fazla ***.then()*** arka arkaya eklenerek oluşturulur.
* Zinciri başlatan bir *promise* olduğu gibi ***.then()*** içindeki fonksiyonların dönüş değeri de *promise* olur. **return** değeri *promise*’leştirilerek zincirin diğer halkasına aktarılır. Bu yüzden zincirin her bir halkasını *promise* okuyan ve *promise* çıktısı oluşturan bir yapı olarak düşünebiliriz.
* Hataların yakalanması için tek bir***.catch()*** yeterlidir. Zincir içerisinde ne zaman **başarısız** (*rejected*) bir *promise* veya beklenmedik bir hata oluşursa, sonrasındaki ***.then()*** halkaları atlanılarak ***.catch()*** içerisindeki fonksiyon çalışır. ***asenkronIslem***, ***baskaAsenkronIslem***, birinci veya ikinci ***.then()***’in hatalı sonuçlanması konsola hata mesajının basılması ile sonuçlanır.

## Promise metodları

Promise nesnesinin kendine ait 4 tane metodu bulunmaktadır. Bunlar yeni bir promise (***new Promise()***) oluşturmadan kullanılabilir. Açıklamadan önce, örnek kullanımlarına bakalım:

// Anında başarıya ulaşan(resolve) promise nesnesi

const basarili = Promise.resolve(42);

// Anında başarısız olan(reject) promise nesnesi

const basarisiz = Promise.reject('Tüh ya');

// Promise'lerden oluşan bir dizinin gelecekteki değerlerinin

// hepsinin başarılı sonuçlanması için oluşturulan promise nesnesi

const tumIslerBitti = Promise.all([promise1, promise2, ...]);

// Promise dizisi içinden en hızlı başarılı/başarısız

// olan değeri tutan promise nesnesi

const enHizliYapan = Promise.race(promiseDizisi)

* **Promise.resolve()**: Verilen değeri, başarılı sonuçlanmış *promise* haline getirir. ***.then()*** içindeki fonksiyonların dönüş değeri kendiliğinden *promise* haline getirilir demiştik. Bu durumu `return Promise.resolve(donusDegeri)` şeklinde de düşünebiliriz.
* **Promise.reject()**: Verilen değer hata mesajı olacak şekilde **başarısız**(*rejected*) olmuş bir *promise* nesnesi döndürür.
* **Promise.all():** Bu metodun kullanılma amacını, paralel olarak gerçekleştirilen asenkron işlemlerin hepsinin bitip, bitmediğini anlamak olarak tanımlayabiliriz. Birbirini beklemeyen asenkron işlemleri (*promise*’leri) zincirlersek toplam gerçekleşme süresini uzatmış oluruz. Hem bekleyen tüm işlemlerin bittiğinden emin olmak hem de bunu *promise* kullanarak yapabilmek **Promise.all()** ile mümkün. Dizideki tüm *promise*’ler **başarılı**(*resolved*) olduğunda tuttukları değerler de dizi halinde kullanıcıya döner. Eğer dizideki herhangi bir *promise* **başarısız**(*rejected*) olursa, **Promise.all()** sonucu oluşan *promise* de **başarısız**(*re*jected) olur.
* **Promise.race()**: Bu metod ise adından anlaşılacağı gibi dizi içerisindeki *promise*’lerin yarıştırılmasıdır. Galip olan, yani en hızlı sonuca ulaşan *promise* dönüş değeri olarak alınır. Yalnız **başarılı**(*resolved*) veya **başarısız** (*rejected*) sonuçlanması sonucu etkilemez. Gelecekteki vaat edilen değerine (istenilen değer veya hata mesajı) ilk ulaşan *promise* yarışın kazananı olur.

# Async/Await

Öncelikle bu özellik ES7 kapsamında olup, *promise*’e nazaran tarayıcı desteği daha azdır.

*Async/await* bize neler katacak? Maddelerimiz şöyle:

* Kod daha okunaklı hale gelecek
* Asenkron işlemlerin gerçekleşme sırasını takip etmek kolaylaşacak
* *Promise* zincirleri için birçok ***.then()*** yazmak gerekmeyecek
* Diğer programlama dillerinden aşina olunan **try-catch** yapısı kullanılabilecek

// Promise zinciri

const zincirES6 = () => {

  return asenkronIslem1()

    .then(asenkronIslem2)

    .then(asenkronIslem3)

}

zincirES6()

  .then(cevap => console.log(cevap))

  .catch(hata => console.log(hata))

// Async/await

const zincirES7 = async () => {

  const sonuc1 = await asenkronIslem1();

  const sonuc2 = await asenkronIslem2(sonuc1);

  return asenkronIslem3(sonuc2);

}

try {

  const cevap = await zincirES7();

  console.log(cevap);

} catch (hata) {

  console.log(hata);

}

Yukarıdaki iki farklı kod da aynı işlemleri gerçekleştirmektedir. İki yöntem de *promise* dönen asenkron işlemleri çağırır. ***await***, *promise* dönen fonksiyonların öncesinde kullanılarak değerin gelmesini bekletir. Ancak ***await*** sadece ***async*** ile tanımlanmış bir fonksiyon içerisinde kullanılabilir. ***async***ile tanımlanmış asenkron bir fonksiyonun dönüş değeri de *promise* nesnesi olur.

## async ve await nedir?

async ve await ES7 ile birlikte asenkron programlama yapmayı daha da kolay hale getiren anahtar kelimelerdir.

async anahtar kelimesi fonksiyon sonucunun bir Promise olduğu belirtirken await ise sonucu Promise olan fonksiyonun bitmesini bekler.

Şimdi buraya nereden geldik bakalım.

let adi = null;

// Uzun bir işlem

function islem() {

  setTimeout(() => {

    adi = 'Falan Filan';

  }, 2000);

}

// Yazdırma işlemi

function yazdir() {

  setTimeout(() => {

    console.log(adi);

  }, 1000);

}

islem();

yazdir();

Yapmak istediğimiz şu: 2 sn. sonra bir değişkene bir değer atayalım. 1 sn. sonra da o değişkeni yazdıralım. Ama sonuç istediğimiz gibi çıkmıyor. Biz değer atadığımız fonksiyonu önce yazsak da JavaScript mantıklı olarak ilk bitecek olan fonksiyonu tamamlıyor.

O zaman callback yöntemini kullanalım.

let adi = null;

// Uzun bir işlem

function islem(callback) {

  setTimeout(() => {

    adi = 'Yusuf SEZER';

    callback();

  }, 2000);

}

// Yazdırma işlemi

function yazdir() {

  setTimeout(() => {

    console.log(adi);

  }, 1000);

}

islem(yazdir);

İstediğimiz sonuç ortaya çıktı.

Ancak callback yöntemi fonksiyonların çoğalmasına ve kodların karmaşık olmasına neden oluyor.

Bir de Promise yapısını kullanalım:

let adi = null;

// Uzun bir işlem

function islem() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      try {

        // Uzun işlemler

        adi = 'Falan Filan';

        resolve();

      }

catch (hata) {

        reject(hata);

      }

    }, 2000);

  });

}

// Yazdırma işlemi

function yazdir() {

  setTimeout(() => {

    console.log(adi);

  }, 1000);

}

islem()

  .then(yazdir)

  .catch(hata => console.log(hata));

Promise yapısı ile **islem** sonrası **yazdir** fonksiyonun çalışacağını anlıyoruz sanki kodlar biraz daha anlamlı hale geldi.

Ancak birbirine bağlı birden fazla Promise yapısı bazı sorunlara yol açabiliyor.

Burada ES7 ile birlikte gelen **async** ve **await** anahtar kelimeleri imdadımıza yetişiyor:

let adi = null;

// Uzun bir işlem

function islem() {

  return new Promise((resolve, reject) => {

    setTimeout(() => {

      try {

        // Uzun işlemler

        adi = 'Falan Filan';

        resolve();

      }

catch (hata) {

        reject(hata);

      }

    }, 2000);

  });

}

// Yazdırma işlemi

function yazdir() {

  setTimeout(() => {

    console.log(adi);

  }, 1000);

}

async function calistir() {

  await islem();

  yazdir();

}

calistir();

İşe yarar bir örnek verelim.

async function toplaGel() {

  let sonuc = await fetch('https://jsonplaceholder.typicode.com/users');

  let veriler = await sonuc.json();

  console.log(veriler);

}

toplaGel();

Promise ile kullanımı da şöyle olacaktı:

fetch('https://jsonplaceholder.typicode.com/users')

  .then(sonuc => { return sonuc.json(); })

  .then(veriler => console.log(veriler));

Son olarak işlemi biraz daha karıştıralım:

async function toplaGel() {

  let sonuc = await fetch('https://jsonplaceholder.typicode.com/users');

  let veriler = await sonuc.json();

  return veriler;

}

toplaGel()

  .then(veriler => console.log(veriler));