



# JS ile Fonksiyonel Programlama (pdf)

**Son Güncellenme Tarihi:** 12.12.2022

**©Copyright:** OnurDayibasi

## INDEKS

1. Giriş
2. Imperative/Declarative Programlama
  - 2.1 Imperative ve Declarative Kavramları
  - 2.2 Imperative ve Declarative Örnek Kullanımları
3. Fonksiyonel Programlamaya Örnek
4. Fonksiyonel Programlamanın İlk Oluşumu
5. Fonksiyonel Programlamanın Gelişimi
  - 5.1 Fonksiyonel Programlamanın İlk Dönemleri
  - 5.2 Fonksiyonel Programlamanın Arka Plana Atıldığı Dönemler. 1975–2005
  - 5.3 iPhone, iPad SDK ve Dağıtık Kodlamanın Yaygınlaşması (2005–2015)
  - 5.4 ES6 ile Birlikte (2015..)
6. Pure Function (Saf Fonksiyon)

- 7. High-Order Functions
  - 7.1 Döngü (Loop) İşlemlerinde Herkes Tarafından Çok Kullanılan Mantıkları Belli Fonksiyonlarda Toplama
  - 7.2 Sabit İşi Yapan Template Fonksiyon Oluşturma
- 8. Currying Functions
  - 8.1 Birbirlerini Kapsayan Fonksiyonların Çağrılması
- 9. Function Accepting Functions
- 10. Partial Function Applications
- 11. Point-Free Style
- 12. Soyutlama ve Kapsama (Abstraction & Composition)
  - 12.1 Soyutlaştmak Aslında Bir Basitleştirme İşlemidir.
  - 12.2 Fonksiyonel Programlama'da Soyutlama ve Kapsama
  - 12.3 Sonuç
- 13. ADT (Abstract Data Types)
  - 13.1. Data Types (Veri Tipleri)
  - 13.2. ADT (Abstract Data Types) Nedir?
- 14. Functors ve Categories
  - 14.1. Functor Nedir ?
  - 14.2. Functor Yasaları
  - 14.3. Functor'ı Ne İçin Kullanabiliriz ?
  - 14.4 Yorum
- 15. Monad Nedir?
  - 15.1 Monad Ne İşimize Yarıyor ?
  - 15.2 Monad Nedir?
- 16. Object-Oriented Programlama ve Tarihçesi
  - 16.1 Büyük Fikir
  - 16.2 Object-Oriented Programmanın Özü
  - 16.3 İyi Bir MOP(Monitoring-Oriented Programming) Nasıl Olmalı?
- 17. Object Composition (Nesneleri Birleştirme)
  - 17.1 React Composition vs. Inheritance
  - 17.2 React Code Sharing Yöntemleri
  - 17.3 Abstraction (Soyutlama)
  - 17.4 Inheritance ile Soyutlama (Abstraction)

18. Kalıtım Türleri (Inheritance Types)
  - 18.1 Functional inheritance (Fonksiyonel Kalıtım)
  - 18.2 Concatenative inheritance (Birleştirici Kalıtım)
  - 18.3 Prototype Delegation ( Delegasyon ile Kalıtım)
  - 18.4 Inheritance over Composition
19. Factory Functions — Obje Üretim Fonksiyonları
  - 19.1 Obje Tanımlama
  - 19.2 Destructuring İşlemleri
  - 19.3 Default Parameters
  - 19.4 Rest Kullanmak
20. Factory Functions - Functional Mixins
  - 20.1 Property Üzerinden Obj Oluşturma
  - 20.2 Factory Functions for Mixin Composition
21. Sınıflar üzerinden Composition Neden Zor ?
  - 21.1 React Hooks Nedir?
  - 21.2 The Delegate Prototype
  - 21.3 The .constructor Property
  - 21.4 Sınıftan → Factory Büyük Değişim
  - 21.5 Sınıf ile Factory Fonksiyonun Kıyaslaması
  - 21.6 Performance and Memory
  - 21.7 Tip Kontrolü (Type Checking)
  - 21.8 Sınıfları Dikkatli ve Doğru Kullanmak.
22. Fonksiyonlar ile Birleştirilebilir (Composable) Veri Türleri
  - 22.1 Bunu Herhangi Bir Veri Türüyle Uygulayabilirsiniz.
23. Lenses
  - 23.1 Lensleri Farklı Şekillerde Tanımlama
  - 23.2 Lensler Functor özelliği ile Map Yapıları Üzerinde Çalışır.
  - 23.3 Özet
24. Transducers
25. JS'in Güçlü ve Doğru Kullanılması
  - 25.1 Make the function the unit of composition. One job for each function.
  - 25.2. Omit needless code.
  - 25.3. Use Active Voice
  - 25.4. Avoid a Succession of Loose Statements

25.5 Keep related code together.

25.6 Put statements and expressions in positive form.

Referanslar

## 1. GİRİŞ

Fonksiyonel programlama nedir? Şu ana kadar bize okullarda öğretilen programlamadan bir farkı bulunuyor mu? Öğrenmemiz avantajlı mı? Nerelerde kullanılır? Bu ve benzeri soruları cevaplamaya çalışacağız.

JavaScript konusunda öğrenme sırasında ES6 (EcmaScript 6) ile gelen bir çok güncellemenin, yani JavaScript'deki büyük değişiklikler ile kendisine sevdiren dilin aslında temellerinde fonksiyonel programlama yapılarının olduğunu öğrenmem beni fonksiyonel programlama anlamında daha fazlayı öğrenmeye itti.

### EcmaScript (ES):

JavaScript'in temelini oluşturan standarttır. İlk olarak tarayıcılar üzerinde çalışan bir dil olarak ortaya çıktığı için ilk çıkışında bu standart içerisinde Flash'ın ActionScript, Microsoft'un JScript barındırmaktaydı. Fakat sonrasında diğer dillerin iyi özellikleri JavaScript eklendikçe, eski dillerden gelen özellikler zamanla dilin içerisinde kaldırıldı.

Benzer şekilde sadece JS dili değil incelediğimde popüler birçok JavaScript kütüphanesinde de benzer **Fonksiyonel Programlama Örütütlerini** görmeye başladım.

Aşağıdaki kitapçık aslında öğrenme arayışının bir çıktısı olarak oluşmuştur.

Bu konuda türkçe birkaç yazı bulmama rağmen bu yazılar çok kısa sadece belli noktalara değinen yazılırdı. Bu nedenden bu konuyu daha detaylı nasıl inceleyebilirim diye düşünmeye başladım.

Konuyu araştırdıkça bu konuyu öğrenmenin o kadar da kolay olmadığını farkettim.  
Nasıl ki;

- Procedural programlamadan → OOP(Object Oriented Programming),

- OOP(Object Oriented Programming) yaklaşımından → TDD yaklaşımıma geçişte bir sürü zorluk yaşadıysak,
- bir benzerini de OOP → Fonksiyonel Programlamaya geçişte yaşayacağız.

Gerçi JavaScript dili multi-paradigm yani çoklu paradigmayı sayesinde istediğiniz yaklaşımıla kod geliştirebileceğiniz bir yapıya sahip. Yani hem Object Oriented yapıda, Hem de Fonksiyonel yapıda kod geliştirebilirsiniz. Benim anladığım kütüphane gibi altyapısal geliştirmeler için fonksiyonel programlama daha avantajlı iken, enterprise bir uygulama geliştirirken Procedural veya OOP(Prototype based inheritance) yaklaşım daha anlamlı olabiliyor. Hatta TypeScript, Flow benzeri type değişkenler ile çalışmak daha önemli olabiliyor.

Bu konuyu araştırırken analiz ederken bana bir rehber gerekiyordu. Bu konuda ücretsiz olarak yayınlanan [Composing Software: The Book \(from Eric Elliott\)](#) kitabından faydalandım. İsteyen burdan da konuyu okuyabilir. Tabii amacım öğrenmek olduğu için kitabı bire bir çevirmeye çalışmıyorum. Bu kitapla birlikte konuyu anlamak için çevresinde araştırma yaptığım kaynaklar ile bu içerikleri birleştirdim, bazı kısımlarını atladım, bazı kısımlarını zenginleştirerek kendi öğrendiğimi anlatabileceğim bir dile çevirdim. Aşağıdaki kitapçık yukarıdaki Eric Elliott kitabı ile birçok kaynağın benim kafamda harmanlanmış hali olarak düşünebilirsiniz.

Öncelikle Imperative ve Declarative Programlamayı anlamak önemli. Çünkü bu ayrıştırma QBasic, Pascal, C, C++, Delphi, Java ile Scala, Haskel, SQL, Lua, JavaScript arasında ne gibi yapısal farklar olduğunu öğrenmemiz açısından da önemli.

## 2. Imperative/Declarative Programlama

Imperative ile Declarative Programlama Arasında Ne Gibi Farklar Bulunur?  
Structural, Procedural, Nesne Tabanlı Programlama Imperative yaklaşımı kullanırken, Neden Fonksiyonel Programlama Declarative yaklaşımı kullanır?  
Avantaj ve dezavantajları nelerdir?

Öncelikle Fonksiyonel programlamalar Declarative programlamadır. Bu yöntemin Imperative'e göre farkının ne olduğunu anlatarak **Fonksiyonel Programlamaya giriş**

yapmanın daha doğru olacağını düşünüyorum.

## 2.1 Imperative ve Declarative Kavramları

**Imperative** Türkçede emir, buyruk, zorunluluk anlamına geliyor. **Declarative** ise bildiren, açıklayan anlamındadır. Yazılımda ise imperative işlemi **nasıl yapacağını** anlattığın, declarative ise **ne yapacağını** anlattığın programlama şekli olarak tanımlıyorlar.

**Nasıl ile Ne** bu kadar farklı programlama paradigmaları nasıl ortaya çıkarabilir diye düşünebilirsiniz ? Ama işleri oldukça değiştiriyor. Örneğin bir çizim programı tasarlayalım.

### Imperative Yöntem

```
// Diktörgen Çizme
kalemiSuPozisyonaNötür()
kalemiBastır()
kalemiSuPozisyonoSürükle()
kalemiSuPosizyondaDurdur()
kalemiSuPozisyonoSürükle()
kalemiSuPosizyondaDurdur()
kalemiSuPozisyonoSürükle()
kalemiSuPozisyonaNötür()
kalemiSuPozisyonoSürükle()
kalemiKaldır()
```

Imperative yöntemde açıklayıcı emirlerle işlemi detaylı bir şekilde gerçekleştiririz.

### Declarative Yöntem

```
dikdortgen(ciz)
```

Declarative yöntemde sadece yapacağınız şeyi anlatırsınız. Tabi bu soyutlama arka planda dikdörtgen fonksiyonu mantığını bu fonksiyonda çiz fonksiyon mantığını bu da arka planda kalemin işletilme mantığını sizden soyutlar.

Ama matematikte olduğu gibi  $f(g(x))$  fonksiyonlar ile kodu geliştirmenizi sağlar. Bu iki yöntem farklı programlama paradigmalarının oluşmasına neden olmuştur.

## Declarative Yaklaşım

Implementation(gerçekleştiririm) detaylarının anlaşılmasıının çok zor olduğu bir DSL(Domain Specific Language) ile gerçekleştirmi yönetme/soyutlama ihtiyacı olan programlamalarda **Declarative Yaklaşımı** tercih etmelisiniz.

**Database Processing (SQL)** sizi veritabanının birçok gerçekleştirim detayından kurtarır. Bu üst seviye Query Dili sayesinde istediğiniz gibi veritabanını kontrol edebilirsiniz. Sizi alttaki birçok network, işletim sistemi, ve dosya sistemi gibi detaylardan kurtan bir standarttır.

Bu gerçekleştirmi Oracle, MySQL, MSSQL, PostgreSQL, Aurora vs.. kimin yapacağı ve nasıl yaptığı sizin ilgilendirmez. Bilmeniz gereken aşağıdaki sorgu dilinin arka planda veriler ve tablolar üzerinde nasıl bir yönetim sağladığıdır.

```
select * from students where score>80
```

**HTML:** Web sayfası oluşturmanız için size bir markup dili verir. Bunu tarayıcılarınız nasıl render edeceği sizin ilgilendirmez. Bu markup dili sizin tanımlarınızdan DOM, CSS, JS birleştirerek Web sayfanızı renderler ve kullanıcı etkileşimlerini handle edecek kodları ve styling kodlarını yükler. Bunun için alttaki detaylar ile uğraşmanız gerekmektedir. Üst seviye declarative tanımları bilmeniz yeterlidir.

```
<html>
  <head></head>
  <body>
    <h1>Declarative Programlama</h1>
    <p>Lorem ....</p>  <body>
</html>
```

**Regex:** Str içerisinde bir takım pattern/örüntüler ile arama yapacaksınız. Bunu kod ile yapmaya çalışırsanız bunun için bir çok fonksiyon yazmanız lazım ve çok performanslı da çalışmamayabilir. Regex declarative dili sayesinde bu işi çok basit bir şekilde kendiniz bir String operasyon kodu yazmadan, dilin kendi yetenekleri ile yapmasını sağlayabiliyorsunuz.

```
function escapeRegExp(str) {
  var regex = /\$/!(.*?)!\$/g;
  return str.replace(regex, "");
}
```

Aynı örnekleri React JSX yapısı içinde söyleyebiliriz. Aslında arkapanda React API çağrılmaktadır. Ama JSX ile bu geliştiriciden soyutlanır.

Tüm yukarıda verdiğim örneklerde kompleks altyapı gerçekleştirimini sizden saklayan daha üst bir Domain Dili ve bunun altında bu dilin belirtiklerini çalıştırın bir Engine/Motor vb.. olduğunu görürsünüz.

Bu dilleri öğrenmesi zordur ama öğrendikten sonra daha az hataya sebep olurlar çünkü kullanıcıyı olabildiğince belli bir kümeye içerisinde kısıtlarlar.

## Imperative Yaklaşım

Business application veya karmaşık iş akışı uygulamalar geliştirmeniz, bunu da **Imperative Yaklaşımı** geliştirmeniz daha mantıklı olacaktır. Çünkü bu tip iş kurgularını belli matematiksel fonksiyonlarda toplamanız pek mümkün olmaz.

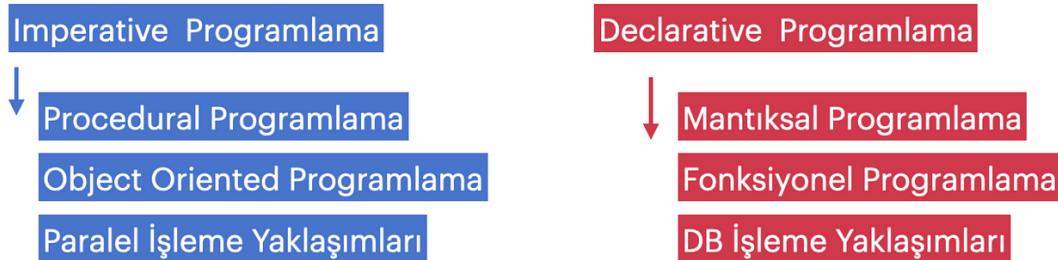
Basic, Pascal, Delphi, C, C++, Java, C# gibi diller ile bu tip çok karmaşık yapılı akışı uygulamalar çok daha basit şekilde geliştirilebilir ve daha sonradan daha iyi bakım yapılabilir ve daha okunabilirde olabilirler.

Yukarıdaki yaklaşılara bir benzerlikte İstatistiksel Modellemelerde kullandığımız Makine Öğrenmesi algoritmalarından verelim;

**Fonksiyonel programlama** veriniz öyle bir şekildedir ki arka plandaki istatistiksel modeliniz bir formül çıkarır. Logistic Regresyon , Linear Regresyon vb.. bunlar bana Fonksiyonel yaklaşımı, Bazen de öyle bir veri yapınız olur ki buda ancak Decision Tree ile bu modeli tanımlayabilirsiniz bu da bana **Imperative Programlamayı** anımsatıyor. Modelin beyinde tutulan modellerin de Fonksiyonel / Procedural olanları var.

## 2.2 Imperative ve Declarative Örnek Kullanımları

## Programlama Paradigmaları



### Programlama Paradigmaları

- **Imperative:**

Procedural Programlama Dilleri (Basic, Pascal, C, C++, Java)

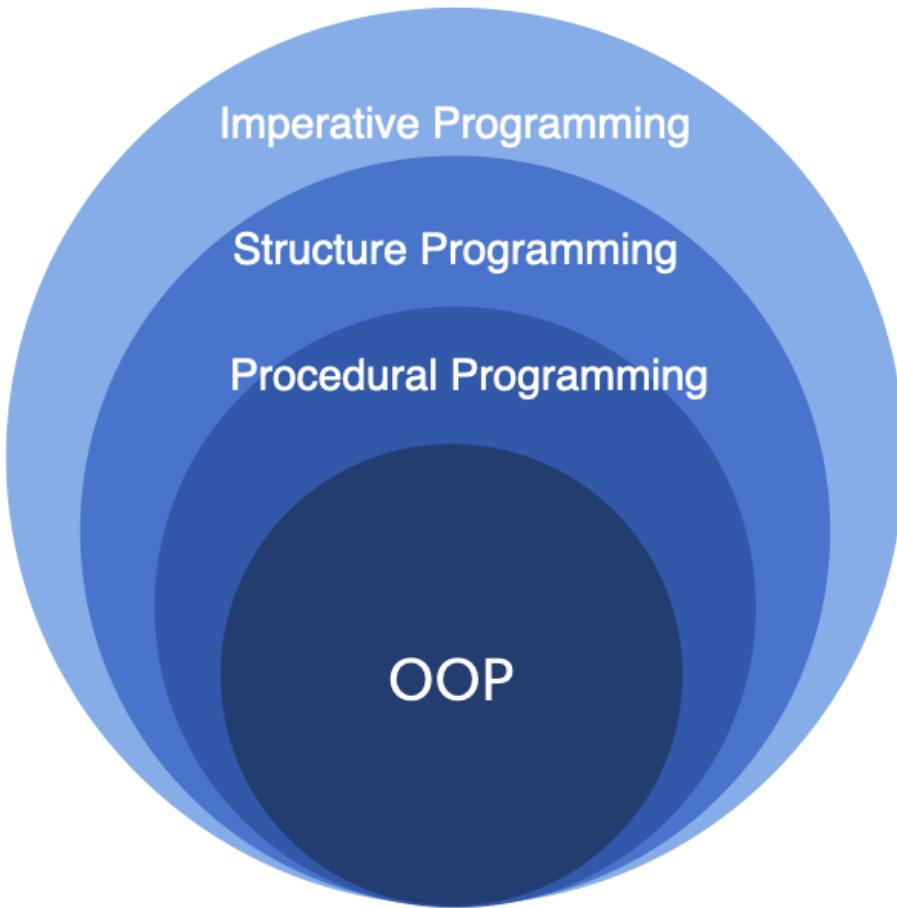
- **Declarative:**

Fonksiyonel Programlama Dilleri ve Markup, DSL diller (SQL, Regular Expression, HTML)

- **Hybrid :**

Javascript, Python

### 2.3. Imperative Programlama Yaklaşımının Tarihsel Gelişimi



## Imperative Programlama Paradigma Kümesi

Öncelikle Imperative dillere bir bakalım. Imperative Programlama yaklaşımı zaman içerisinde Programlama dillerinin gelişimi ile birlikte GoTo ile Programlama → Structured Programlama → Procedural Programlama → Object Oriented Programlama içerisinde yer almıştır.

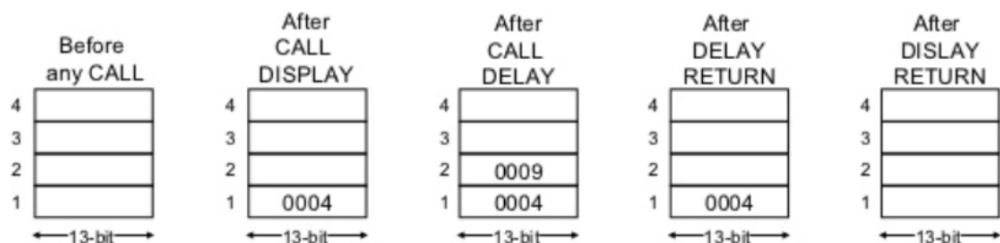
### GOTO

İlk Assembly yani makine dilinde Register(yazmaç) veriyi taşı, toplama işlemi yap sonra bunu diske yaz emirlerinin olduğu kod blokları arka arka yazıldığını görebilirsiniz.

```

00020 ;delay subroutine
          00021      ORG    30H      ;put delay at address 30H
0030 30FF  00022 DELAY   MOVLW 0xFF  ;WREG = 255
0031 0091  00023      MOVWF MYREG
0032 0000  00024 AGAIN   NOP      ;no operation wastes clock cycles
0033 0000  00025      NOP
0034 0000  00026      NOP
0035 0B91  00027      DECFSZ MYREG,F ;decrease until MYREG
becomes 0
0036 2???  00028      GOTO   AGAIN   ;repeat decrement process
0037 0008  00029      RETURN
          00030      END

```



## Assembly Code

Zaman içerisinde High Level Language (Yüksek seviyeli diller ile birlikte) makine dilinden uzaklaşıkça GOTO kullanımının yazılım geliştirmeye zarar verdiği ve kodun spaghetti gibi karıştırdığını belirtmişlerdir. Go to Statement Considered Harmfull . (Dijkstra 1968)

## Structured Programlama

Daha sonrasında dillerin gelişimi ile birlikte yazılımı yönetme olayı için (structured) yapısal programlama yöntemi geliştirilmiştir. Bu yöntemde GOTO metoduna yerine

- **Sequences:** Ardı ardına işletilen kod blokları ve subroutine (subroutine — statement, method, procedure, function)
- **Selections:** Belli koşullarda (sistemin o anki state) göre subroutine çalıştırılması (Condition — if, switch)
- **Iterations:** Bir döngü ile sistemin ilgili state gelinceye kadar subroutine çalıştırılması (Loops — for, while)

Peki işlem sırasında acil çıkışlar yapabilmek veya kodun sequence kodunu 1 atlayarak çalıştırması için bazı özel anahtar sözcükler mevcuttur. **return, break, continue, exit**

Subroutine incelediğimizde

- **statement** kodun en ufak işletildiği satır,
- **procedure** basic gibi dillerde tekrar şeklinde çalıştırılan statement grubundan olduğu işletildiği grup.
- **function** ise bir grup statement işletilip geriye dönüş yapmasıdır. (**pure function**) ise bu işlem sırasında dışarıdan aldığı değerler dışında etkilenmez ve dışarı etkilemez.
- **method** ise sınıfı bağlı fonksiyondur.

```
a=b+3; //statement//procedure bir deger dönmez.
procedure topla(&toplam,a,b){ &toplam=a+b}//fonksiyon bir işlemi yapıp
//geriye bir değer döndürür.
//procedure bir deger dönmez.function topla(a,b){return a+b;} .class calculator{
    private result;
    publicmethod topla(a,b) {
        this.result = a+b;
    }
}
```

## Procedural Programlama

Procedural Programlamada değişkenler , veri-yapıları ve sub rutinlerle uğraşan bir yapıda programlama geliştirilir. Prosedürler kendilerine verilen veri yapılarını işler. Sistemin genel bir state prosedürler arasında işletilir.

## Object-Oriented Programlama

Objelerin tuttuğu state/attribute ve bu objelerin metodları üzerinden veriler işlenir. Objeler arasındaki iletişim için interface(arayüzler) kullanılır. Bu sayede objelerin birbirleri iletişimde olduğu bir yapı ortaya çıkar.

Aslında Procedural Programlamadan → Object-Oriented programlamaya geçiş sırasında (abstraction ve cohesion) genel uygulama kapsamından nasıl objeler içerisinde taşındığını ve artık daha kontrollü bir şekilde nasıl objeler ile kontrol edildiğini görebilirsiniz.

Bu geçişler, yeni programlama yöntemlerinin çıkması, önceden işletim sistemi, driver yazılımı vb uygulama geliştirmeden yavaş yavaş daha enterprise(kurumsal) yazılımlara daha büyük uygulama geliştirme ihtiyaçlarının ve aynı projelerde daha fazla kişinin çalışması gereği için ortaya çıkmıştır.

## 3. Fonksiyonel Programlamaya Örnek

Kısaca Fonksiyonel programlamada procedural programlama gibi aynı structured programlama yapısındaki işlemleri yapmaya çalışmaktadır. Fakat fonksiyonel programlamada imperative elemanlar ile gerçekleştirilen **condition**, **loop** kavramlarının fonksiyonel programlarda kaldırılmıştır. Sistem state kullanmadan pure function ve bunun girdileri ile bu akış gerçekleştirmeye çalışılır.

Bunu bir kod ile anlatalım. Aşağıdaki kodda ben her bir elemanı tek tek dönen bir for döngüsü ve içerisinde console.log komutunun çalıştırıldığını görebilirsiniz.

```
for(let i=0;i<arr.length;i++) {  
    const element=arr[i];  
    console.log(element);  
}
```

fonksiyonel programlamada sizi bir loop döngüsü yapmaktan kurtarır. kontrollü bir forEach içerisinde işleminizi yaparsınız. For döngüsünü geliştiriciden soyutlamıştır.

```
arr.forEach(el=>console.log(el));
```

## 4. Fonksiyonel Programlamanın İlk Oluşumu

Fonksiyonel Programlama yeni yeni popüler olmaya başlamış gibi gözüksede aslında tarihi oldukça eski. Hatta günümüz yazılım programlama paradigmaları oluşturan Alan Turing ve Alonzo Chruch ilk olarak ortaya attığı programlama modeli fonksiyonel programlamanın temeli olan **Lambda Calculus**'tur.

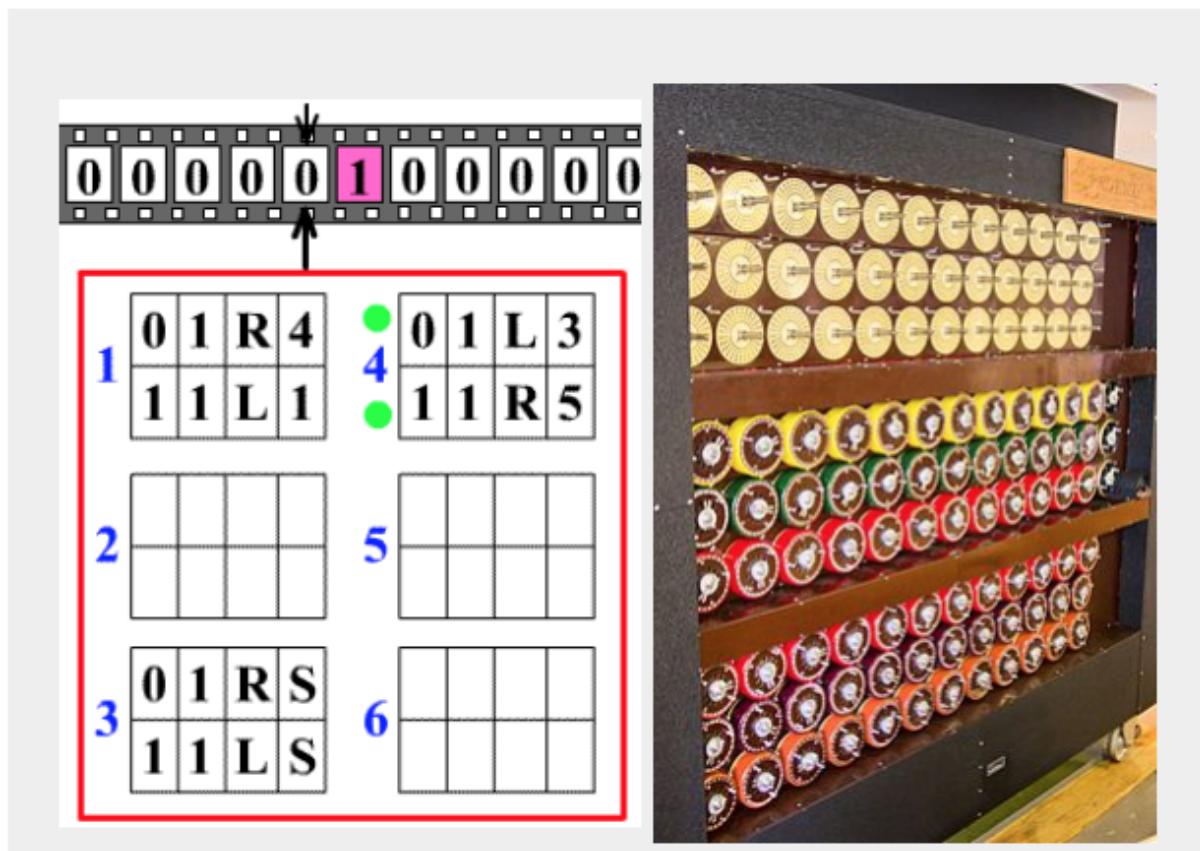
Fonksiyonel Programlama'nın başlangıcı aslında ilk evrensel programlama makinesinin (TM) tasarılanması ve teorinin atılması ile başlıyor.

Yani **Charles Babbage (Analytic Engine)** → **Ada Lovelace (First Program on Baggage)** → **Hermann Hollerith (Punch Card)** → **Alan Turing (Turing Machine)** → .... bir yolculuk.

Fonksiyonel Programlama hikayesi Alan Turing ve Alonzo Chruch oluşturduğu Chruch-Turing Tezi sonrasında ortaya çıkıyor.

Alan Turing **Evrensel Turing Makinesi** ile David Hilbert and Wilhelm Ackermann 1928 yılında **Entscheidungsproblem** denilen karar verme problemini matemetiksel olarak formülasyonunu işletecek çözebilecek evrensel bir programlama

aracı üretme sonucu tekrar programlanabilir ilk bilgisayar prototipini ortaya çıkarmıştır.



Turing Mekanizması ve “Bombee” Enigma kodunu deşifre eden makina

TM(Turing Machine) bir mekanizma. Teyip üzerinde bilgileri koyabileceğiniz şeritler mevcut. Kontrol mekanizması bu şeritleri ileri ve geri hareket ettirerek ve okuyucu/yazıcı kafanın bulunduğu şeritteki veriyi Okuyabilen(READ), Yazabilen(WRITE) bir mekanizmaya sahip. Sistem sizin programınızı işletirken sürekli olarak bu basit mekanizmayı işletiyor. Tabi bunun makineye göre program oluşturmak hiç de kolay değil.

Turing Makinesinin nasıl işlediği ile ilgili aşağıya birkaç link ekliyeceğim

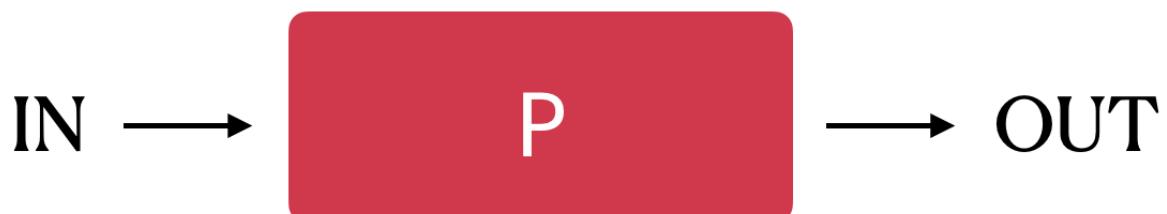
- [Turing Machine](#)
- [How Turing Machine Works](#)
- [Turing Machine Simulator](#)

Aynı dönemde **Alonzo Crunch** Lambda( $\lambda$ ) Calculus sistemini oluşturuyor. Bu sistem matematiksel mantık ile bilgisayar bilimini hesaplama oluşturmak için soyutlamayı(abstraction) kullanarak değişkenleri birleştirerek hesaplama yapmasını sağlar.

**Toplama** için  $\text{sum}(a,b)$  , **çarpma** için ayrı bir  $\text{multiply}(a,b)$  bir mekanizmaya ihtiyaç var.

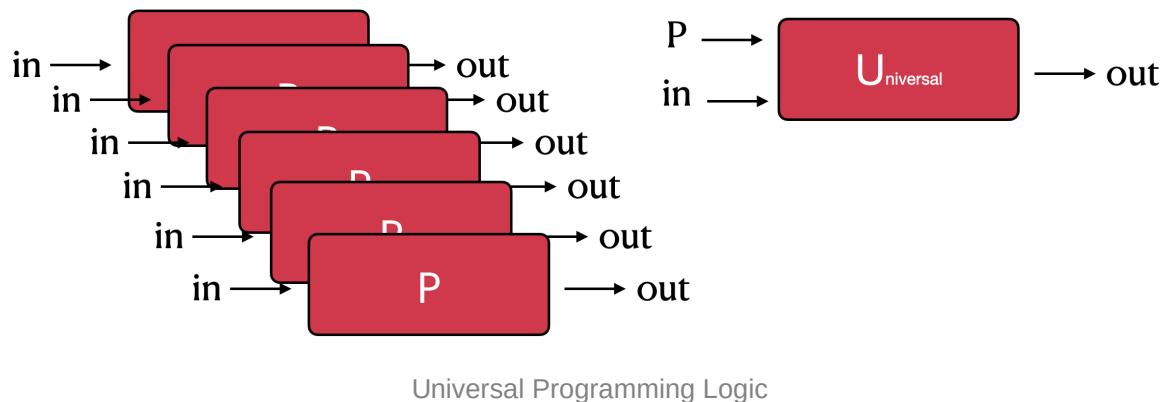


Sonuçta girdisi olan ve bunu işleyen ve çıktıya dönüştüren **P** programına ihtiyaç duyuyoruz.



Peki bu programların sayısı arttığında her bir fonksiyon için ayrı bir program mekanizması olması yerine, ortak tekrar programlanabilen evrensel bir mekanizmaya ihtiyacı var. İşte Universal Turing Machine burada devreye giriyor.

Burada kastedilen punch kart tek seferde işletilen programlar yerine tekrar tekrar işletilebilen aynı mekanizma içerisinde farklı programların çalıştırılabilğini, bir programın çıktısının diğer program tarafından işletilebildiği mekaniklerin oluşturduğu TM (Turing Machine) .



Bu yapının üzerinde Lambda( $\lambda$ ) Calculus bir fonksiyonun diğer fonksiyonu kapsaması ve işletmesi üzerine çalışır. Modern JS ES6 karşılığı bu.

```

sum=(a,b)=>a+b
multiply=(a,b)=>a*b
A=3, B=2, C=1
result=multiply(sum(B,C),A) //9
  
```



Function Composition Nasıl Gerçekleşir

Burda Turing Machine (TM) problemi çözmek için bir yöntem oluşturulmuştur. Bunun üzerinde **Alonzo Crunch** geliştirdiği Lamda( $\lambda$ ) Calculus ile matematiksel spesifikasyonu ortaya konulmuştur.

Lambda ifadesi (+ 4 5) işlemi Lambda ifadesi ile soyutlanarak matematiksel işleme dönüştürülür.

Lambda Expression: Function application written in prefix form.

**Ex1:** “Add four and five” is

(+ 4 5)

Lambda abstraction:

( $\lambda x. + x 1$ )

( $\lambda$

x

.

+

the function of

x

that

adds

x to

|  
Lambda Expression

Sizin yapmak istediğiniz işlemler aşağıdaki örnekte olduğu gibi Lambda ifadesi olarak yazılmış sonrasında Turing Makinesi üzerinde işletilecek fonksiyonlara dönüştürülürler.

Example 2

$5 * 6 + 8 * 3$  is

( $+ (* 5 6) (* 8 3)) \rightarrow (+ 30 (* 8 3))$   
 $\rightarrow (+ 30 24)$   
 $\rightarrow 54$

Another example

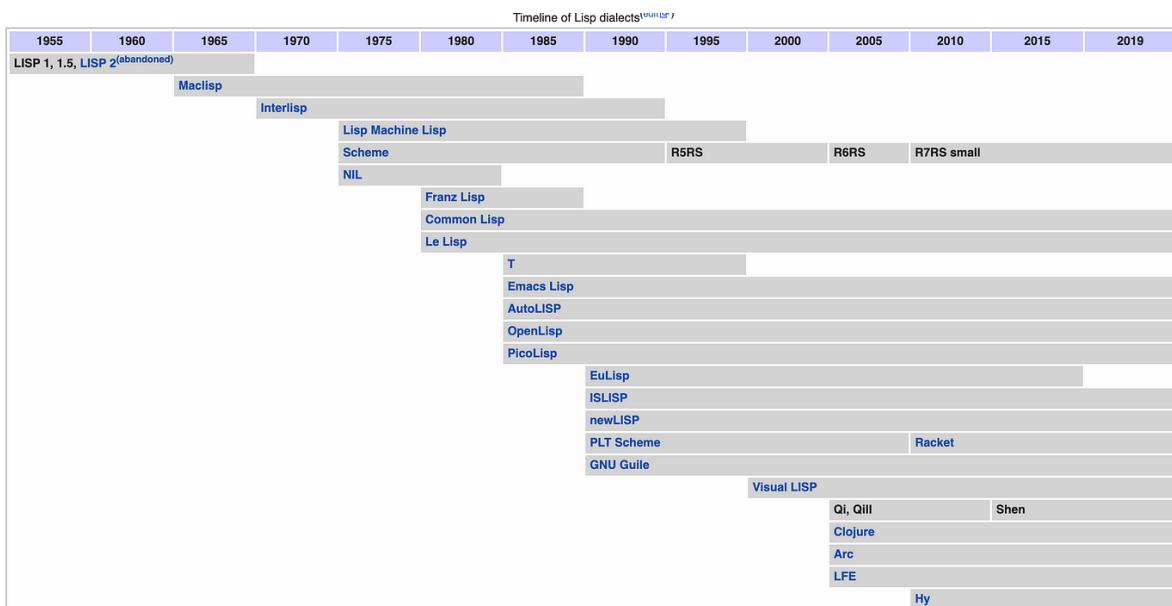
( $\lambda x . \lambda y . +x((\lambda x.-x3)y))5 6 \rightarrow (\lambda y . +5((\lambda x.-x3)y))$   
 $\rightarrow +5((\lambda x.-x3)6)$   
 $\rightarrow +5(-6 3)$   
 $\rightarrow + 5 3$   
 $\rightarrow 8$

Matematiksel Bir İşlemin Lambda İfadesi ile Yazımı

# 5. Fonksiyonel Programlamanın Gelişimi

## 5.1 Fonksiyonel Programlamanın İlk Dönemleri

- **1930 yılında** Fonksiyonel Programlamanın başlangıcına baktığımızda ilk Alonzo Crunch Lambda Calculus, Alan Turing oluşturduğu Turing Machine üzerinde çalışabilen matematiksel bir soyutlama ve hesaplama dili oluşturmuştur. Bu dil fonksiyonların diğer fonksiyonları kapsaması yöntemi ile fonksiyonel programlamanın temelini oluşturdu.
- **1958 yılında** Lisp dili oluşturdu. Lisp dili yapay zeka terimini oluşturan John McCarthy tarafından oluşturuldu. Bu uzunca bir . Lisp dili üzerinde Scheme (ileride JS bundan esinlenerek oluşturuldu), AutoCAD'in Autolisp, Racket ve Clojure benzeri diller aynı dialect üzerine türeyen dillerdir.



[https://en.wikipedia.org/wiki/Lisp\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Lisp_(programming_language))

- **1972 yılında** Alan Kay's Smalltalk objelerden atomic birimler oluşturma ve bunların birbirini kapsaması üzerine nesne tabanlı programlamayı ortaya attı. Buradaki objeler birbirleri iletişimini bir birlerine geçirdikleri mesajlar üzerinden gerçekleştirir. Bu yönyle Smalltalk fonksiyonel nesne tabanlı programlamaya örnek verebiliriz.

## 5.2 Fonksiyonel Programmanın Arka Plana Atıldığı Dönemler. 1975–2005

Bu dönemde Procedural diller çok ön plana çıktı. İlk önce C dili sonrasında Windows ile birlikte Basic, Pascal gibi diller giderek popülerleşti. Fortran, Cobol gibi dillerde bankacılık alanında yaygınlaştı

- **1975–1985 yılları arasında** C (K&R) ve Basic, Pascal

Mikrobilgisayarlar ve işletim sistemlerinin gelişmesi, Desktop uygulamalarının gelişmesini sağladı, bu daha büyük projeler geliştirme ihtiyacını doğdu.

- **85 sonrasında** Visual Basic, Delphi, C++ Builder kod geliştirme arayüz araçları ortaya çıktı.

**Microsoft Windows** ortamı için Win32 , C++ dilleri, **Sun Microsystem** tüm ortamlar için Unix, MacOS, Windows için Java OOP geliştirdi. **MacOS** için Apple firması Objective C (Smalltalk üzerine C dilini nesne tabanlı hale getirerek kullandı).

Bu zamanlarda Java dilinin;

- her platformda çalışabilme özelliği
- JVM mimarisinin popülerliği,
- sunucu ve client tarafında aynı dilin kullanabilme imkanı,
- Sunucu tarafında güçlü olan IBM, Oracle, Sun gibi büyük firmaların veritabanı üzerine kurdukları ApplicationServer kütüphanelerinde ve projelerinde Java'yı tercih etmesi
- Java'nın backward compatible (geri uyumlu)

inanılmaz bir Java popüleritesi oluşturdu ve büyük bir geliştirici topluluğu ve büyük şirketler Java dilini kullanmaya başladı. Diğer diller Java dilinin gölgesinde kaldı.

Bu dönemde oyun programlama, windows işletim sistemi dll, lib yazılımları(C, C++) haricinde tüm piyasa Java dilini kullanmaya başladı.

Bu zaman aralığında çıkan fonksiyonel diller Java, C, Basic popülerliği arkasında kaldılar.

ObjectiveC (1984), Erlang (1986), Haskell(1990)

## 5.3 iPhone, iPad SDK ve Dağıtık Kodlamanın Yaygınlaşması (2005–2015)

Steve Jobs'un iPhone, iPad duyurmasından ve çok yaygınlaşmasından sonra Apple iOS SDK'sını duyurdu. iOS SDK ObjectiveC dili üzerine geliştirildiği için bu dil bir an içinde popülerleşti. Birçok uygulama geliştiricisi bir anda ObjectiveC dilini öğrenmeye başladı. ObjectiveC adından da anlaşılacağı gibi nesne tabanlı C dili ama C++ yapısında değil Smalltalk dil yapısındaki objeler arasında mesaj ile iletişim kuran bir yapıya sahip.

Bu süreçte nesne tabanlı programlamanın alternatifisi bir çok nesne tabanlı fonksiyonel programlamayı destekleyen programlama dili ortaya çıkmıştır.

- [scala-lang.org](http://scala-lang.org) (2004)
- [clojure.org](http://clojure.org) (2007)
- [rust-lang.org](http://rust-lang.org). (2010)
- [elixir-lang.org](http://elixir-lang.org) (2011)
- [swift.org](http://swift.org). (2014)
- [fsharp.org](http://fsharp.org) (2014)

## 5.4 ES6 ile Birlikte (2015..)

2015 yılından itibaren ES6 ile birlikte Javascript fonksiyonel programlama paradigmalarını Javascript daha çok kullanmaya başladı ve bunun haricinde zaten fonksiyonel geliştirme yapmanızı sağlayan JS kütüphaneleri Ramda, Lodash/FP , Immutable JS ve UI geliştirme kütüphanelerinin React/Hook/Redux fonksiyonel programlama temellerini içeriyor olması, geliştirici topluluklarının zaman içerisinde Fonksiyonel programlamaya ilgisini daha çok arttırmış durumda.

**Swift** iOS, **Kotlin** Android gibi diller de multi-paradigm dillerden. Fonksiyonel programlama gerçekleştirebiliyorsunuz.

# 6. Pure Function (Saf Fonksiyon)

Saf(pure) fonksiyonlar, fonksiyonel programlamada önemli bir yer teşkil eder, Concurrency (eş-zamanlılık), Test edilebilir kod, tahmin edilebilir kod, deterministik bir kod geliştirmek için pure fonksiyonlar oluşturmanız gereklidir.

Fonksiyon belli **girdiler** alıp, bu işletip **sonuç** dönen mekanizmalardır ve 3 amaç için kullanılırlar.

- **Mapping:** Fonksiyona giren girdileri sonuçlara map eder. Bu bizim bildiğimiz  $f(x)=y$  işlemidir.

- **Procedures:** Ardışık instruction/komutları çağırduğumız fonksiyonlar.
- **I/O:** Network işlemleri, Kullanıcı girdileri yakalama, ekrana çizim, console log yazdırma, dosya sistemine erişim vb..

## Saf Fonksiyon Nedir ?

- Aynı girdiler için her zaman aynı sonucu verecek.
- Herhangi bir side effect (yan etkisi) olmayan

fonksiyona Pure(Saf) fonksiyon denir ve Mapping işlemleri Pure fonksiyon şeklinde yazılabılır.

## Referential Transparency Nedir

Algebra işlemlerinde ilgili fonksiyonun karşılığında bir değer koyabiliyorsanız Örneğin **max → 12 , double → 6** değeri koyabilirsiniz. Bunu fonksiyon bu aşamada herhangi bir console, ekran, network işlemi yapmadığı için ve sonuç ne olursa olsun aynı olacağı için diyebilirsiniz.

```
//Max
const max=Math.max;
console.log(max(3,2,8,12));

console.log(12)//Double
const double=x=>x*x;
console.log(double(3));
console.log(6)
```

Bu şekilde programın sonucunu değiştirmeden yerine eşdeğer değerine koyma işlemine **referential transparency** denir.

Pure fonksiyonlar paralel programlama için çok uygun mekanizmalardır. State tutmadıkları ve dış etkileri olmadığı için paralel olarak rahat bir şekilde işletilebilirler.

Bir bağımlılıkları olmadığı için kod içerisinde organize edilmeleri, refactor ile bir yerden başka bir yere taşınmaları oldukça kolaydır. Bakım aşamasındaki değişikliklerden az etkilenirler.

Mapping fonksiyonlarının Pure fonksiyon olmasındaki zorluklar nelerdir dersek

- Ortak Durum Paylaşımı (Shared State)
- Random ve Date gibi aynı değeri oluşturmayan fonksiyonların kullanımı
- Mutability

## Problemler

### Ortak Durum Paylaşımı (Shared State)

Aynı veri üzerine birden fazla process yazıyorsa burada ortak durum paylaşımı olur. Örneğin AJAX call düşünelim. Arama yaparken 3 karakterden sonra arama işlemi başlatılsın. 3 karakter ayrı bir arama isteği, 4ncü karakterde ayrı bir arama isteği, 5 karakter yazıldığında ayrı bir arama AJAX request yapılsın.

4ncü karakterin response diğerlerinden sonra/geç geldiğini düşünelim. Bu durumda sistem 5 karakterin aramanın sonucunu değil de 4 karakter aramanın yanıtını sisteme yansıtacak, bu da hatalı bir gösterim olacaktır. Bunu engellemek için Redux'ta olduğu gibi veri üzerinde bir karar mekanizması olup bunun 3,4 aramanın yanıtlarını iptal edecektir. Çünkü 5 için arama yapacağına bunun yanıtının sistem için anlamlı olması sağlanır.

Paralel işleme sırasında bir durum oluşturuluyorsa bu deterministik olmayan sonuçların ortaya çıkmasına sebep olur.

```
non-determinism = parallel processing + mutable state(değişken durum)
```

Bu tarz paralel işleyen yapıların mutable(değişken) state üretmesi sonucunda beklenmedik sonuçlar ile karşılaşabilirsiniz.

### Fonksiyona Giren Girdiler Aynı Sonucu Vermez İse?

Bu durum fonksiyonun içerisinde her çağrılığında başka değerler üreten aşağıdaki fonksiyonlar varsa ve bunlar sonuç üzerinde etkiliyse her fonksiyon çağrılığınızda farklı bir sonuç ile karşılaşırırsınız.

- fonksiyon içerisinde **Math.random()** değer üretiliyorsa
- **new Date()** zaman değeri üretiliyorsa

### Fonksiyon Yan Etkisi Var İse

Geliştirdiğiniz fonksiyon parametre olarak verdığınız değeri değiştiriyorsa bunun çok sakıncası vardır. Dışarıda bir state olarak tanımladığınız user nesnesinin nerede hangi fonksiyonun içerisinde nerede değiştirildiğini bilemezsiniz.

```
const addAge=(user,age)=>{user.age=age; return user;};
const user={name:'onur'}
console.log(addAge(user,12));
```

```
const addElement=(arr,el)=>{arr.push(el); return arr};  
const arr=[1,2,3];  
console.log(addElement(arr,5));
```

Bundan dolayı state mutable hale gelmiştir. Fonksiyon içerisinde değerin immutable olması için, nesneden yeni bir nesne türetmeniz gereklidir. Burada deep copy(derin kopyalama) yöntemi nesnenin kopyası alınır. **Not:** Bunun yerine daha performanslı tries veri yapısını kullanan kütüphanelerde mevcuttur. Örneğin [immutable.js](#) veya [Mori](#) kullanabilirsiniz.

## 7. High-Order Functions

Örnek bir yazılımı fonksiyonel olarak geliştirerek. Mevcut programlama paradigmamızdan ne gibi farklar olduğunu analiz edelim.

Procedural, Fonksiyonel veya Nesne Tabanlı programlamada her zaman amacımız ortak mantıkları soyutlaştırmak ve bunları belli soyutlamalar içerisinde tutarak tekrar tekrar kodlamamaktır. Bu sayede hem daha az satır kod yazılmış olur hem de iş mantığı bir kapsam içerisinde toplanmış olur.

High Order Functions fonksiyonların davranışlarındaki ortak örüntülerini bir üst fonksiyona taşımanızı yardımcı olan bir yöntemdir.

Argümanlarında 1 yada 1 den fazla function referansını parametre olarak alan veya return değerini geriye fonksiyon olarak dönen fonksiyonlara denir.

**Array** üzerinden sıkça kullandığımız aşağıdaki fonksiyonları HoF türünde fonksiyonlardır.

```
[].forEach (e=> ...) //iterate every elements  
[].find(e=>...) //return an element according to condition  
[].findIndex(e=>...) //return an element index according condition  
[].filter(e=>...) return items according to condition  
[].sort(e=>...) sorts according to given condition  
[].map(e=>...) transform existing array to other array  
[].reduce(e=>...) combines elements and return an element  
[].some(e=>...) check some elements  
[].every(e=>...)
```

## 7.1 Döngü (Loop) İşlemlerinde Herkes Tarafından Çok Kullanılan Mantıkları Belli Fonksiyonlarda Toplama

### forEach → myEach

**Procedural Programlama yaklaşımında** For döngüsündeki tüm elemanlara ulaşıp bunlar üzerinde işlem yapmak istiyorsanız, for döngüsünü kodun her kısmında dönmeniz gerekir.

```
const arr=['red','green','blue'];
for(let i=0;i<arr.length;i++){
  console.log(arr[i]);
}
```

**Fonksiyonel Programlama yaklaşımında** For döngüsünü myEach fonksiyonu içerisine encapsulate ediyoruz. Bu sayede sürekli tüm elemanlarını dönecek kodları her yerde yazmamıza gerek kalmaz.

```
Array.prototype.myEach=function(fn){
  for(let i=0;i<this.length;i++){
    fn(this[i])
  }
}arr .myEach(el=>console.log(el));
```

Diğer **filter, find, findIndex, sort, map, reduce, some, every** hepsi de yukarıdaki forEach benzeri. Tek bir mantığı işletiyor ve soyutluyor. Sizin sadece yapmanız gereken filtrelemeyi sağlayan logic vermeniz. Tek tek dönme işlemini ve bunun filtreye uyup uymadığını anlama ve bunu array ekleme işini abstract(soyut) hale getirdiğimiz fonksiyon sağlayacak.

## 7.2 Sabit İşi Yapan Template Fonksiyon Oluşturma

Örneğin verilen sayıyı 5 ve 10 ile toplayan 2 tane fonksiyon yapmak isteyelim.

### Procedural Programlamada

Procedural programlamada her bir Adder fonksiyonu için toplama işlemi dağıttığımızı görebilirsiniz.

```
const fiveAdder(x){return x+5;}
const tenAdder(x){return x+10;}console.log(fiveAdder(7)); //12
console.log(fiveAdder(12)); //22
```

## Fonksiyonel Programlamada

Bu tip fonksiyon toplama işlemini template'e dönüştürmek için High Order Functions kullanabilirsiniz. Toplama işlemi içinde **makeAdder** şeklinde bir fonksiyon yazmamız yeterli.

```
makeAdder=(x)=>(y)=>x+y;
const fiveAdder=makeAdder(5);
const tenAdder=makeAdder(10);

console.log(fiveAdder(7)); //12
console.log(tenAdder(12)); //22
```

# 8. Currying Functions

İster nesne tabanlı programlama olsun, ister fonksiyonel programlama olsun, yazılım sürekli olarak temel elemanların birbirini kapsaması, kullanımı ve birleşimi sonucu oluşur. Aşağıda da bu composition kavramının nasıl işletildiğini anlatmaya çalışacağım.

“Composition: the act of breaking a complex problem down into smaller problems, and composing simple solutions to form a complete solution to the complex problem.”

## 8.1 Birbirlerini Kapsayan Fonksiyonların Çağrılması

2 tane fonksiyonumuz olsun. İlkinin çıktısını, diğerine verdığımız fonksiyon türlerini deneyelim ve her fonksiyon çalıştığında bunu console log olarak basalım.

Örneğin **f(g(x))**

- Birincisi gelen girdi değerini 1 ile toplayan
- İkincisi gelen değeri 2 ile çarpan

## Procedural Programlama

Procedural programlama benim alıştığım yöntem. En az ortak değişkeni kullanarak buradaki ara çıktılar için **result** değişkeni kullandım.

```
function addOne(x){return x+1;}
function multiplyTwo(x){return x*2;}
function doStuff(x){
  let result=addOne(x);
  console.log(`after addOne ${result}`)
  result=multiplyTwo(result);
  console.log(`after multiplyTwo ${result}`)
}
doStuff(20);
//after addOne 21
//after multiplyTwo 42
```

## Fonksiyonel Programlama

Fonksiyonel programlamada ise amacımız arka arka çağrıma işlemini sizin yapmamızın yerine **pipe** fonksiyonu ile soyutlamaktır. Pipe fonksiyonun çalışması için her seferinde 1 parametre alarak çalışan fonksiyonlara ihtiyacımız var. Trace fonksiyonumuzda 2 parametre alıyor bunu tek parametre alan hale nasıl getirebiliriz **Currying** mekanizması bunu sağlıyor.

```
const g = n => n + 1;
const f = n => n * 2;
const trace = label => value => {console.log(` ${label}: ${value}`); return value;};
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);
const doStuffFunctional = pipe(
  g,
  trace('after g'),
  f,
  trace('after f'));
doStuffFunctional(20);
```

## Currying Functions Nedir?

Curried fonksiyonlar birden fazla parametre alan fonksiyonlar yerine her seferinde bir parametre alacak şekilde fonksiyonun yapılandırılmasıdır. Bunu da 1 parametre alıp geriye diğer parametreyi alacak fonksiyonu dönerek gerçekleştirir.

```
function sum(a,b){return a+b};
function sumCurried(a){return function(b){return a+b};};
const sumES6=(a,b)=>a+b;
const sumCurriedES6=(a)=>(b)=>a+b;console.log(sum(3,2));
```

```
console.log(sumCurried(3)(4));console.log(sumES6(1,2));
console.log(sumCurriedES6(2)(4));
```

## 9. Function Accepting Functions

Currying fonksiyonların nasıl kullanıldığını bir önceki bölümde anlatmıştık. Aşağıdaki fonksiyonların her biri birbirinin aynısı ve soldan → sağa doğru işaretlerek ilerletilebilir.

```
sum(a,b) yerine const sum=a=>b=>a+b;
multiply(a,b) const multiply=a=>b=>a*b;
//ve diğer subtraction ve divide vb fonksiyonları
//currying fonksiyon olarak yazabiliriz.
```

Yukarıdaki sistemi biraz daha ileriye götürüp toplama, çıkarma ve çarpma ve bölme işlemlerini de **calc** fonksiyonu ile soyutlayabiliriz.

İşlem yapmasını istediğimiz fonksiyonu parametre olarak geçirip bunu da dinamik hale getirebiliriz.

```
const add=a=>b=>a+b;
const sub=a=>b=>b-a;
const multi=a=>b=>a*b;
const div=a=>b=>b/a;
const calc=fn=>a=>b=>fn(a)(b)

var addTwo = calc(add)(1); //Function Accepting Functions
var multiTwo= calc(multi)(2); //Function Accepting Functions
var divTwo = calc(div)(2); //Function Accepting Functions
var subTwo = calc(sub)(2); //Function Accepting Functions

console.log(addTwo(10));
console.log(multiTwo(10));
console.log(divTwo(10));
console.log(subTwo(10));
```

**Not:** Buradaki bir problem fonksiyon parametre sayısını calc fonksiyonu biliyor yani tüm fonksiyonlarımız en sonunda 2 parametre alıyor. Peki argüman sayısı değişken fonksiyonları desteklemek için ne yapmalıyız ? Örneğin **add(a,b,c,d,e ...)** için ne yapmalıyız ? Bir sonraki konuda Partial Applications konusunda bunun üzerinde duracağım.

# 10. Partial Function Applications

Bir önceki bölümde Function Accepting Functions yazısında nasıl aynı parametredeki işlemleri fonksiyonları da parametre olarak geçirip işletebileceğimizi anlatmıştık. Ama fonksiyonumuz 2 parametre almak yerine 3 parametre, 4 parametre alan fonksiyon olduğunda sistemimizi nasıl genişleteceğiz. addTwo fonksiyonunu aşağıdaki **sum2**, **sum3**, **sum4** destekleyecek hale nasıl getiririz.

```
sum2=(a,b)=>a+b;
sum3=(a,b,c)=>a+b+c;
sum4=(a,b,c,d)=>a+b+c+d
```

Bunu ES6 ile array argümanı alacak şekilde geliştirebiliriz.

```
const sumReducer = (acc, val) => acc + val;
const sum=args=>args.reduce(sumReducer);
console.log(sum([1,2])); //3
console.log(sum([1,2,3])); //6
console.log(sum([1,2,3,4])); //10
```

Öncelikle sum fonksiyonunu birden fazla sayıyı toplayabilen bir hale getiriyoruz. JS Spread yeteneği bunu bize sağlayacaktır. **Partial Applications** belli mikardaki argümanın bir kısmını önceden sabit alan ve geriye döndüğü fonksiyon ile daha az argüman alarak işlemleri tamamlamanız sağlayan fonksiyonlardır.

Aşağıdaki fonksiyonda 1 ile toplama işlemi sabit kısımdır. Bir ile toplamayı kalıp haline getirip içerisinde geçen parametreleri topladıktan sonra 1 rakamına bunu ekliyoruz

```
const sumReducer = (acc, value) => acc + value;
const sum= (...args)=>args.reduce(sumReducer);

console.log(sum(1,2)); //3
console.log(sum(1,2,3)); //6
console.log(sum(1,2,3,4)); //10

const partial=(fn,...args)=>(...arg2)=>fn.apply(null,[...args,...arg2]);
const addOne=partial(sum, 1);

console.log(addOne(4)); //5
console.log(addOne(10,2,3)); //16
```

Bu işlemi ilerletmek ve **sum** yanına **multi** koymak istersek tek yapmamız gereken multi mantıklarını eklemektir. Burada 2 ile çarpmak sabit.

```
const sumReducer = (acc, value) => acc + value;
const multiReducer = (acc, value) => acc * value;
const sum= (...args)=>args.reduce(sumReducer);
const multi= (...args)=>args.reduce(multiReducer);
const partial=(fn,...args)=>(...arg2)=>fn.apply(null,[...args,...arg2]);
const addOne=partial(sum, 1);
const multiTwo=partial(multi, 2);console.log(addOne(10,2,3)); //16
console.log(multiTwo(3,5)); //30
```

**partial** vs **partialRight** arasında ne gibi farklar vardır. Verilen argümanların fonksiyonun sağında mı yoksa soldan → sağa sıra ile mi kullanılacağı arasındaki faktır. Aşağıdaki örnekte bir cümle kalıbü oluşturan bir **fill** fonksiyonu var. a, b parametreleri sırasının değişmesi cümleye farklı anlamlar veriyor. Toplama işlemindeki durumdan farklıdır. **Partial** işlemi ilk parametreyi kalemi veren haline getirirken **partialRight** kalemi alan kişi haline getiriyor.

```
const fill=(a, b)=>`${a} gives ${b} a pencil.`;
const partial=(fn,...args)=>(...arg2)=>fn.apply(null,[...args,...arg2]);
const partialRight=(fn,...args)=>(...arg2)=>fn.apply(null,[...arg2,...args]);
const aliGives=partial(fill,'Ali');

console.log(aliGives('Veli'));//Ali gives Veli a pencil.
console.log(aliGives('Ahmet'));
//Ali gives Ahmet a pencil.
const aliReceives=partialRight(fill,'Ali');
console.log(aliReceives('Veli'));//Veli gives Ali a pencil.
console.log(aliReceives('Ahmet'));//Ahmet gives Ali a pencil.
```

Partial placeholder (    ) tanımlayıp uygulamanın istediğiniz kısmını sabit , istediğiniz kısmını dinamik hale getirebilirsiniz. Aşağıdaki örnekte kırmızı, yeşil ve mavi renk tonlarını oluşturabilirsiniz.

```
const hex=(r, g, b)=> '#' + r + g + b;
console.log(hex('11', '22', '33')) // "#112233"
const partialAny = (function() {
    function partialAny(fn ,...orig) {
        return (...partial) => {
            const args = [];
            for (let i = 0; i < orig.length; i++) {
                args[i] = orig[i] === partialAny._ ? partial.shift() : orig[i];
            }
            return fn.apply(this, args.concat(partial));
        };
    }
    partialAny._ = {};
})
```

```

        return partialAny;
    }());

const __ = partialAny._;
const maxRed=partialAny(hex,"ff",__,__); //ffaabb
console.log(maxRed('aa','bb'));
const greenMax=partialAny(hex,__,'ff',__);//aaffbb
console.log(greenMax('aa','bb'));
const blueMax=partialAny(hex,__,__,"ff");//aabbff
console.log(blueMax('aa','bb'));

```

## 11. Point-Free Style

Function Composition, Currying, High Order Function'ın bir sonucu olarak Point Free Style fonksiyonlar yazılabilir.

Fonksiyon tanımlamasında hiç bir zaman üzerinde işlenecek data bahsini kullanmamak. Bu şekilde fonksiyon yazarak kod geliştirmeye **Point Free Style** denir.

Point-Free style means functions that never mention the data upon which they operate. First class functions, currying, and composition all play well together to create this style. (from )

Peki fonksiyonu nasıl tanımlıyoruz? Fonksiyon girdi parametreleri alıp, bu parametreleri işletip geri sonuç olarak dönen fonksiyonlardır.

```

function testFunc(param1,param2,...) { //codereturn}
const testFunc= function(param1,param2,...) { //codereturn}
const testFunc=(param1,param2,...)=>{// codereturn}

```

Bu durumda fonksiyonu girdi parametreleri olmadan nasıl tanımlayabiliriz ? Aslında bu fonksiyonların parametre almayacağı anlamına gelmiyor sadece bu fonksiyon alan fonksiyonların başka bir fonksiyonla kapsanarak bunun parametre almayan stil ile yazılmıştır.

Örneğin; Toplama işlemi normal bir şekilde yazıldığında fonksiyon a, b değerlerini alan bir fonksiyon olarak yazılır.

```
const add=a=>b=>a+b;
console.log(add(2)(3));
```

Bunu Point Free Style yazmak istediğimizde dışarıdan değişken almayan fonksiyonlar oluşturuyoruz. fiveAdder ve tenAdder dışarıdan bir değişken aldığıını görebilirsiniz.

```
const fiveAdder=add(5);
const tenAdder=add(10);
console.log(tenAdder(4));//14
```

Aşağıdaki örnek [Function Composition point-free style](#) blog yazısından alınmıştır. Anlatım açısından konuya çok iyi bir örnek olduğu için burada bahsetmek istiyorum. Aşağıdaki örnekte **name** dışardan parametre olarak aldığı için burada bu yazım stili Point Free Style olmuyor.

```
const initials =name => name.split(' ')
    .map(compose(toUpperCase, head))
    .join('. ');
```

Bu fonksiyonu aşağıdaki şekilde yazdığımızda fonksiyonları parametrelerini kaybedip **compose**, **join**, **toUpperCase**, **split**, **head** fonksiyonlarını işleyen bir dış kapsayıcıya dönüştürerek yazdığınızda buna Point Free Style deniyor.

```
const initials =compose(join('. '), map(compose(toUpperCase,head)), split(' '));
initials('hunter stockton thompson'); // 'H. S. T'
```

## 12. Soyutlama ve Kapsama (Abstraction & Composition)

Yazılım aynı gerçek dünyadaki gibi soyutlamalar üzerine inşa edilmiştir. İnsanlık gelişimini nasıl bir takım işlerde özelleşerek hizmet verdiği müşterilerinden işlerini detaylarını nasıl yapıldığını soyutluyor ise, yazılım geliştirme de bundan farklı değildir.

Örneğin arabanızı bakıma servise götürüyorsunuz, veya bir restorana yemeğe gidiyorsunuz, size sunulan hizmetlerin sonucunu siz görüyorsunuz ama bu

hizmetlerin içerisindeki detayları harcanan eforları vs görmüyorsunuz.

İşte **SOYUTLAMA (Abstraction)** dediğimiz kavram yapılan işin **NE** olduğu ile ilgilenmek, **NASIL** yapıldığı ile ilgilenmemektir.

Bunu yazılım geliştirmeye uyarladığımızda bizim geliştirdiğimiz kodlar ile ekip arkadaşlarımıza, diğer takımlara veya başka firmalardaki geliştiricilere sağlamış olduğumuz soyutlamalar (abstraction) bugünkü yazılım ekosisteminin, bugünkü büyük yazılım sistemlerinin, uygulamalarının oluşmasını sağlayan temel yapıdır, diyebiliriz. Daha açık konuşmak gerekirse Yazılımda ;

- Fonksiyon (Utility functions)
- Algoritma (Algorithms)
- Veri Yapıları (Data Structures)
- Modüller (Modules)
- Sınıflar (Classes)
- Kütüphane. (Library)
- Çerçeve Yazılımları (Frameworks)

Tüm bu saylıklarımız geliştiricilerin büyük yazılımlar , altyapılar oluşturması için imkan sağlar. Hatta günümüzde sağlanan bu hizmet soyutlamaları aynı makinede derlenip tek bir kod bloğu olacak diye zorlamıyor.

Örneğin SaaS servisleri sayesinde birçok hizmeti Stripe(Ödeme), Auth0(Yetkilendirme) vb servisleri uygulamanıza bağlayıp istediğiniz servisleri arayüzler API üzerinden kullanabiliyorsunuz.

Peki **KAPSAMA(Composition)** neden bu kadar önemli. Yukarıda bahsettiğimiz gibi hizmetleri soyutlayarak belli problemleri, sorunları çözen yapılarımızı oluşturduk.

Aynı gerçek hayatı olduğu gibi araba servisinde olduğu gibi bir kişi gidiyor arabasını bu servise bırakıyor ve akşam üzeri bakımları yapılmış, aksamları tamamlanmış şekilde alıyor. Gerçek yaşam işler bu kadar basit değil, yazılımlar da günümüzde bu seviyenin çok çok üzerinde. Örneğin bir Havalimanını ele alalım. İşlerin ne kadar karmaşık olduğunu siz de göreceksiniz. (Rezervasyon Sistemi, Otopark Sistemi, Güvenlik Sistemi, Kafeler/Mağazalar, Kargo Sistemi, Bilet Sistemi, Uçak Sistemi, Havaalanı vb...) birçok hizmetin kesiştiği bir yer. Bu durumda siz nasıl birçok hizmetten faydalaniyorsanız, yazılım geliştirirken de bu soyutlanan parçaları uygulamanızın kullanımını **KAPSAMA**'sını sağlayarak büyük yazılımlar, sistemler geliştirebilirsiniz.

Buraya kadar Abstraction & Composition (Soyutlama ve Kapsama) kavramından detaylara girmeden bahsettim. Ama bu işlemlerin doğru ve kaliteli yapılmadığında kurduğumuz sistemler giderek karmaşıklaşacak ve içinden çıkışlamaz yapımlara dönüşecektir. Bundan dolayıdır ki bu konuda yazılmış birçok kitap, örnek ve makale bulunur. Yazının devamında bu konulara, yazılımda geçen bazı kavamlara değinmek istiyorum.

## 12.1 Soyutlaştmak Aslında Bir Basitleştirme İşlemidir.

Soyutlaştmak hizmetin veya servisin arkasında olan birçok işlemin, problem çözümünün karmaşıklığın size belli basit arayüzler üzerinden sunularak basitleştirilmesidir.

Arayüz derken, Nesne Tabanlı Programlamada Javada **interface** ve **abstract keyword** bu dildeki soyutlama arayzlarının oluşumunu sağlar, JAR paketleri olsun NPM paketleri olsun, C++ dll modülleri import, require kullanarak h dosyalarını, interface, veya fonksiyon export kullanılarak biz geliştiricilerin arka planda gerçekleştirm detayı dediğimiz karmaşıklıklardan kurtulmamızı sağlar. API (Application Programming Interface) bizim bu kütüphaneler ile haberleşmemizi iletişim kurmamızı sağlar.

Bu soyutlaşma ilk düşünce ortaya çıktıgı dönemlerden Aristoteles (theory of knowledge) çağdaş yöntem bilim ve bilgi kuramını ortaya attığı dönemlere kadar eskiye gider aslnda. Tümdengelim ve Tümevarım akıl yürütme yöntemleri ile bu soyutlamaları insanoğlu önce düşünme şekillerinde gerçekleştirmiştir.

**Tümdengelim:** Genelden özele giderek akıl yürütmeaktır.

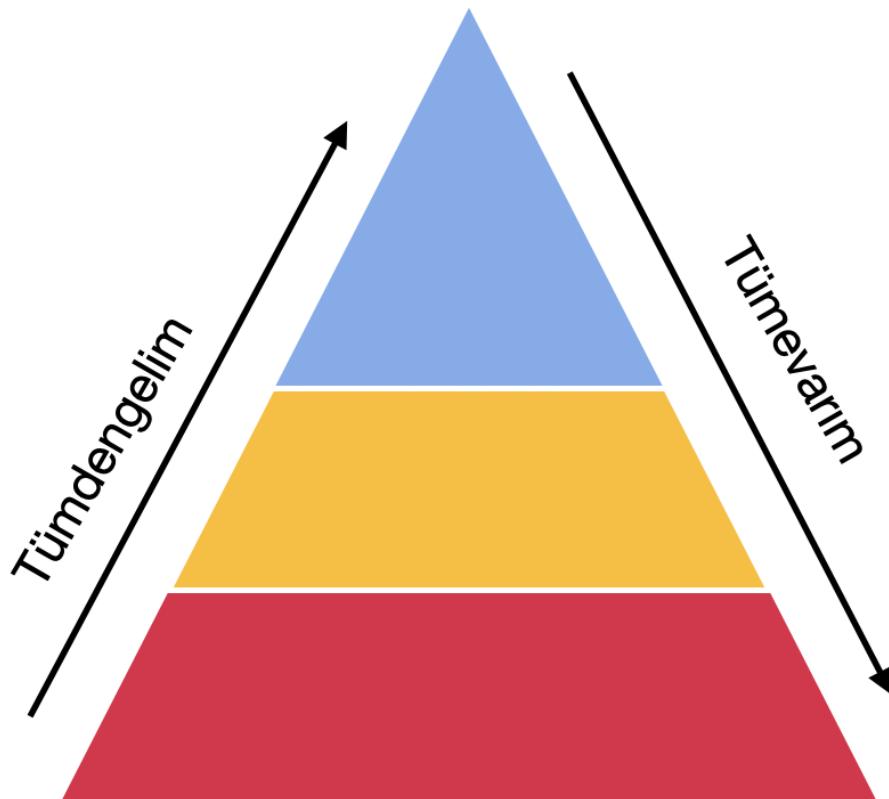
- Bütün balıklar denizde yaşar. Hamsi bir balıktır. O halde hamsi denizde yaşar.
- Bütün insanlar ölümlüdür. Sokrates insandır. O halde Sokrates ölümlüdür.

**Tümevarım:** Özelden genele giderek akıl yürütmeaktır.

- Sokrates insandır. Sokrates öldü. O halde tüm insanlar ölecektir.

Tümevarım özelden genele gittiği için örnek sayısı arttıkça ve sonuç hep aynı çıkar ise çıkarımıza doğru olma ihtimali artacaktır.

Şimdi gelelim yazılımda buna karşılık gelen **Generalization** ve **Specialization** kavamlarının neler olduğuna.



**Generalization (Genelleme)** : Tekrar eden örüntüler ve işlemleri çıkarıp bu benzerlikleri bir soyutlama arkasında saklamaktır. Farz edelim tüm aldığımız ürünlerde %18 kdv ekleniyor. Tüm muhasebe yazılımı yapan uygulamaların bunu yapması yerine Bunu yapan bir fonksiyon, sınıf veya kütüphane yazarak bu muhasebe işlemini soyutlayabiliriz.

**Specialization (Özelleşmek)** : Özelleştirmek'te genelleştirmenin aksine sadece o duruma özel işlemleri , mantıkları oluşturarak soyutlamak anlamına gelir.

Soyutlama (Abstraction) işlemi yaparken bu iki yöntemi ne kadar iyi algılayabildiğimiz ve ne ölçüde başarılı şekilde yapabildiğimize kalacaktır. Buna göre yazılımımız ya çok karmaşık yada istenilen basitlik seviyesinde olacaktır.

## 12.2 Fonksiyonel Programlama'da Soyutlama ve Kapsama

Bizim nesil Java'daki Nesne tabanlı programlamadan öğrendi Abstraction, Encapsulation, Aggregation, Composition, Generalization, Inheritance,

Polymorphism etc.. birçok kavramı ama burada Fonksiyonel Programlama üzerine durduğumuz için Fonksiyonel Soyutlama ve Kapsama'dan bahsediyor olacağım.

Yazının ilerleyen kısmında Nesne Tabanlı Programlama'da bu Soyutlama ve Kapsama'da ne tür yanlışlıklar yapılabildiği üzerinde de duracağız

Fonksiyonlar farklı farklı bağamlarda farklı isimler atayıp(**identity**) kullanabildiğiniz ve bu fonksiyonları birbirine kapsayacak şekilde data

**kapsayıcı(composition)** kompleks fonksiyonlar yazabildiğiniz için fonksiyonlar başarılı soyutlama araçlarıdır diyebiliriz.

Pure(Saf) fonksiyonlar soyutlama için en kullanışlı fonksiyonlardır.

```
f(x)= y , g(w)=z fonksiyonları olsun (f(g)(x)) olan bir h fonksiyonu  
nasıl yazacağız.
```

Bunu gerçek bir örnek ile göstermek istersek. Toplama işlemi yapan bir fonksiyonumuz olsun.

```
const add = (a, b) => a + b;//Kullanım  
const a = add(1, 1);  
const b = add(a, 1);  
const c = add(b, 1);
```

Bunu bu şekilde yazmak yerine Curried fonksiyon şeklinde yazdığımızda kendi özelleşmiş +1 yapan fonksiyonumuz geliştirebiliriz.

```
const add = a => b => a + b;  
const inc = add(1);
```

Örneğin Higher Order Function dediğimiz **map,reduce,filter** kendi uygulamamıza göre özelleştirip kullanırtabiliriz. doubleAll fonksiyonu yapıp kullanabiliriz.

```
const map = f => arr => arr.map(n=>n*n); yerine const f = n => n * 2;  
const doubleAll = map(f);
```

## 12.3 Sonuç

Sonuç olarak yazılım geliştiriciler sürekli olarak bu soyutlama ve kapsama işlemini gerçekleştirerek yazılımlarını geliştirirler. Burada önemli olan konu bunları nasıl

yapmalılar ki rahatça ve herkes tarafından kolaylıkla kullanılabilisin ? `map` , `filter` , and `reduce` fonksiyonlarını analiz ederek bu soyutlaştırmanın olması gereken temel özelliklerini çıkarabilirsiniz;

- Simple (Basit)
- Concise (Öz)
- Reusable (Tekrar Kullanılabilir)
- Independent (Bağımsız)
- Decomposable (Ayırıtırılabilen)
- Recomposable (Yeniden kapsayabilen)

## 13. ADT (Abstract Data Types)

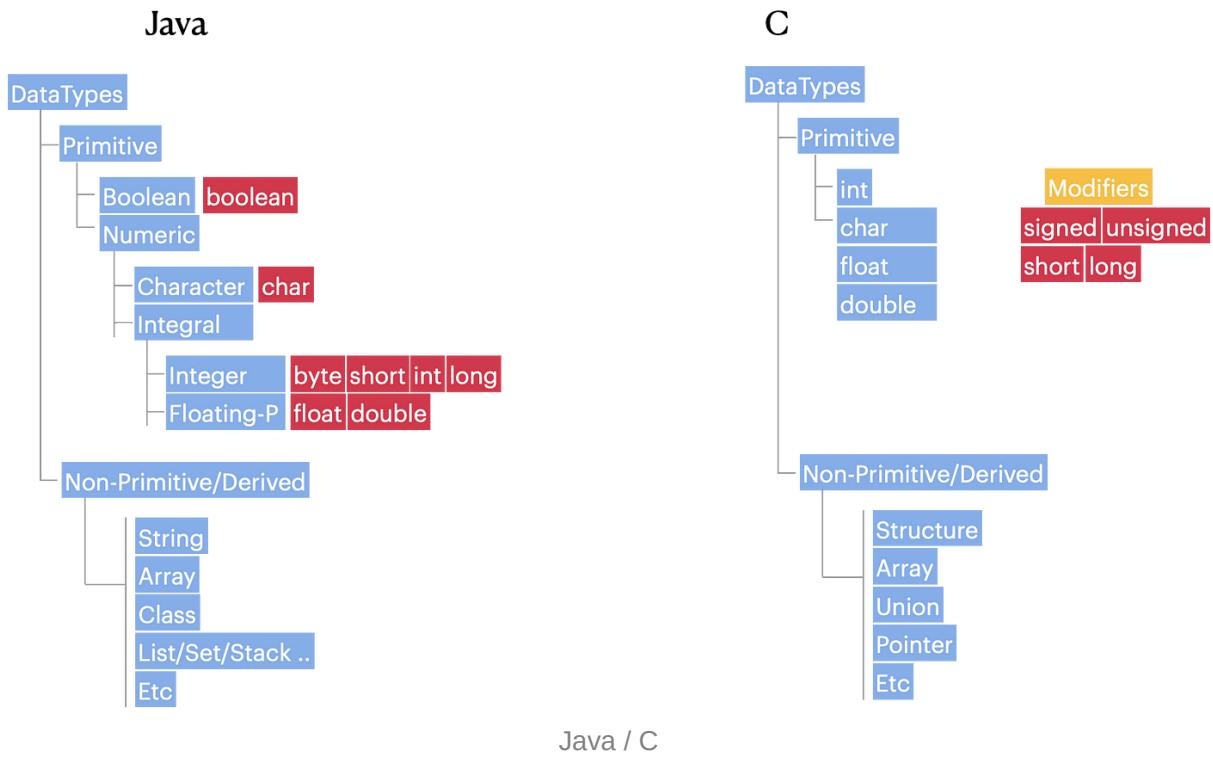
Veri Yapıları(DS) ile Soyut Veri Tipleri(ADT) arasında ne tür farklar bulunuyor. Sıkça kullanılan ADT Türleri nelerdir ? Bu soyut veri tipleri cebirsel işlemlerde ne gibi avantajlar sağlar ?

Bu yazında ADT yani **Soyut Veri Tiplerinin** ne olduğunu ve yazılım için neden önemli olduğunu anlatmaya çalışacağım.

### 13.1. Data Types (Veri Tipleri)

Programlama dillerinde type diller ise 3 aşağı , 5 yukarı aynı türde veri tiplerini içerir. Siz matematiksel bir sayı hesabı yapmak için **integer**, daha detaylı para hesabı için **float**, adres, isim vb işlemler için **string** gibi veri tiplerini kullanırsınız.

Dilleri incelediğimizde **Primitive** ve **Non Primitive** olarak veri türleri 2 'ye ayrılır. Bazlarında **Primary**, **Derived** ve **UserDefined** olarak ayrılabilir. Ama sonuçta dillerin veri tipinin tutacağı verinin türü ve bellekteki boyutu sabit ve değiştirilemez olan (Primitive) ile geliştiriciler tarafından yeni veri tiplerinin oluşturulmasını sağlayan(Non-Primitive) yapılar bulunur.



Sonuçta yeni veri tipleri oluşturacaksak bunları önceden dil tarafından tanımlanmış sabit Primitive ile kullanıcı tarafından daha önceden oluşturulmuş Non-Primitive Kapsayacak şekilde yeni veri tipleri oluşturarak **Kar Topunun** kar üzerinde ilerlerken büyümesi gibi veri tiplerinin diğer veri tipleri kapsayarak büyümesi ile büyük yazılımlar ortaya çıkar.

Geliştiricilere sağlanan bu esneklik sayesinde sonsuz sayıda veri tipi oluşturulabilir. Peki (ADT) Soyut Veri Tipi ne oluyor ? Henüz bu konuya girmeden önce Veri Yapıları (Data Structures) konusunu da biraz dejinmek istiyorum. Bilgisayar Mühendisliğinde 1 veya 2nci sınıfta karşımıza çıkan bu derste Algoritma bilgimizi geliştirmek için Node tanımlaması ile başlayan dersin ilerleyen süreçlerinde LinkList, DoubleLinkList gerçekleştirip bunların fonksiyonlarını implement(gerçekleştirmemizi) etmemizi isterler.

Çünkü yazılımda;

- bir öğrenci değil birden çok öğrenci vardır.
- bir sınıf değil birden fazla sınıf vardır.
- bir öğretmen değil birden fazla öğretmen vardır.
- bir ders değil birden fazla ders vardır.

Sizden istenen bu öğrencilere erişmek, yeni öğrenciler eklemek, silmek gibi işlemlerdir. Bunları yapabilmeniz için bilindik bazı veri yapılarını Örneğin Array, LinkedList, Vector, Stack vb.. nasıl oluşturabileceğinizi ve kullanacağınızı bilmeniz gereklidir.

Tabi gerçek iş yaşamında sıkça kullanılan bu veri yapılarını kendi başına geliştirmeniz gerekmeyez. Bunun için dillerin utility paketleri veya farklı kütüphaneleri içerisinde hazır Collection, Template Library vb veri yapıları kütüphaneleri bulunur.

**Java'da Collection paketi** içerisinde ArrayList, LinkedList, HashSet, HashMap, TreeSet, Vector, PriorityQueue Stack, ConcurrentHashMap

**CPP için Standart Template Library** pair, vector, list, queue, set, queue, map, bitset, valarray

**Javascript'de** Array, Object, Map/Set, WeakMap/WeakSet gibi hazır gerçekleştirmeleri görebilirsiniz.

Tüm dillerde hazır benzer veri yapıları mevcuttur ve bunların bazen isimleri farklı olabilir. Örneğin HashTable Objective C dilinde karşımıza **NSDictionary** olarak çıkabilir. Önemli olan bu veri yapılarının nasıl çalışıklarını anlayıp ihtiyaçlarınız doğrultusunda kullanabilmenizdir.

Farklı diller de olsa bunlar hep gerçekleştirmeleridir. Biz geliştiriciler bu gerçekleştirmeleri kullanırız ama başka bir dili geçsek de diğer gerçekleştirmi kullanmayı öğrenmemiz pek zaman almaz, peki bu **NASIL** olabiliyor ? Cevabı ADT yani **Abstract Data Type** 'dır.

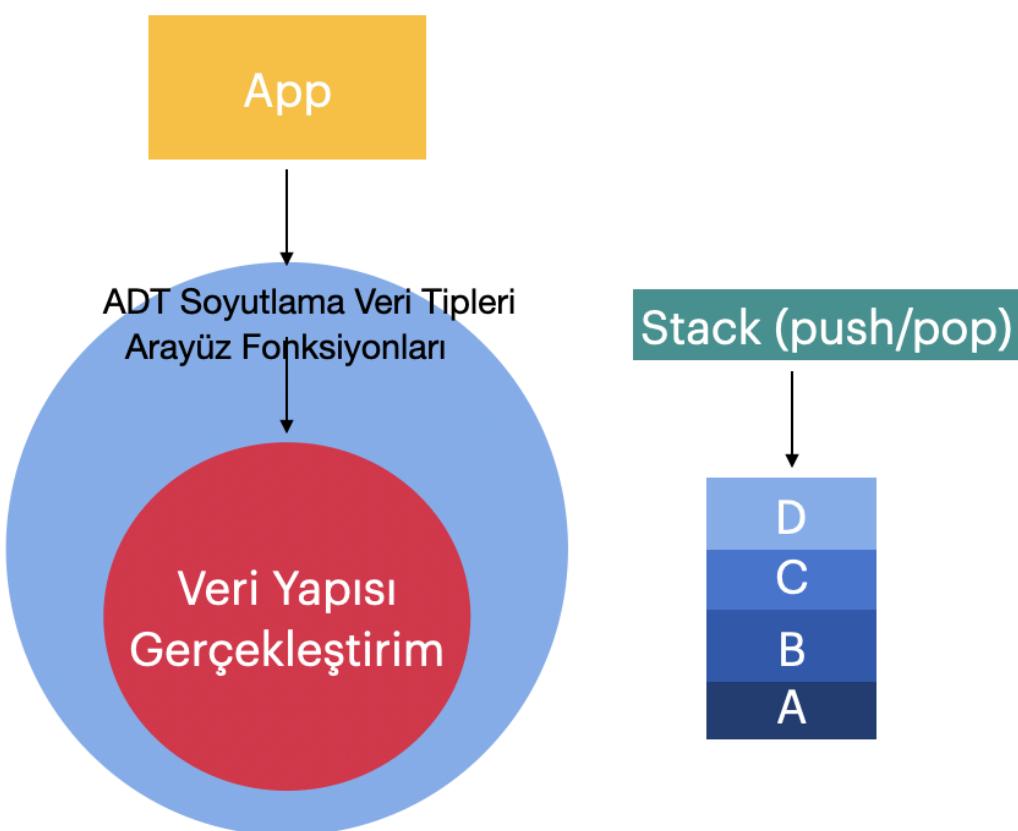
## 13.2. ADT (Abstract Data Types) Nedir?

Yukarıda bahsettiğim farklı dillerdeki veri yapıları;

- gerçekleştirmeleri ,
- verilerin bellekte nasıl tutulduklarını,
- thread bu verilere nasıl erişiklerini, vs vs bir çok detayı içerir.

Ama biz Soyut Veri yapılarından bahsediyorsak Gerçekleştirme detaylarından **SOYUTLANMIŞ** bir kullanımından bahsediyoruzdur. Veri Yapısı üzerindeki bazı verileri ve işlemleri(ops) temsil eden aksiyonlar tarafından tanımlanan soyutlamadır.

Açıklama biraz karmaşık gelmiş olabilir. Aşağıdaki resmin durumu daha iyi anlatacağınu düşünüyorum.



### Örnek Bir ADT ve VeriYapısı(DS) gösterimi

Arayüz kısmına soyutlamasına ADT , gerçekleştirimine DS diyoruz. Sıkça ve çokça kullanılan ADT ler;

- List
- Stack
- Queue
- Set
- Map
- Stream

Ama buradan sadece sıkça kullanılan ADT var diye anlaşılmasın. ADT herhangi bir veri kümesindeki işlemlerin soyutlaması olduğu için sonsuz sayıda ADT vardır diyebiliriz. ADT'ler, semigroups, monoidler, functorlar, monadlar vb. dahil olmak

üzere birçok yararlı cebirsel yapıyı ifade edebilir. Örneğin JS için [The Fantasyland Specification](#) bunun ile ilgili kataloga ulaşabilir.

## Neden ADT?

Aslında bu sorunun cevabını yukarıda da biraz vermiştim. Biz geliştiricilerin kütüphane(library), çerçeve yazılımı(framework) veya dilden bağımsız şekilde yazılım blokları oluşturabilmek için ortak dil oluşturmamızı sağlar. Bu sayede yazılımlarda veriler üzerinde çalışırken tekrar kullanılabilen modülleri matematiksel olarak tanımlamıza olanak sağlar.

Yazılım dediğimiz kavram her geçen gün giderek büyümekte ve milyonlarca satır kodu barındıran uygulamalar günümüzde çalışmaktadır; Google, Facebook, Amazon, Microsoft hatta Tesla, SpaceX hepsinin içerisinde milyonlarca satırlık kodlar var. Yazılım mühendisleri donanımlar üzerinde çalışan kodlarını her ne kadar çalıştığını iddia etseler de gerçekte yazılım projeleri çok kompleks, kırılgan ve karmaşık yazıldığından şu temel problemlerle karşı karşıya kalır.

- Bütçeyi Aşmak
- Gecikme
- Hatalı
- Eksik Gereksinimli
- Bakımı/Devamlılığı zor

Yazılımın modüler parçalardan olduğunu düşünürseniz, tüm sistemi anlamana gerek yoktur. Yazılım'da bu prensibe **locality** denir. Yazılımda bu locality prensibini yakalamak için modüle ihtiyacınız var. Bu sizin geliştirdiğiniz kısmı tüm sistemden izole ederek tüm sistemi anlamak ile uğraşmadan kendi yapınıza odaklanma imkanı verir. ADT bu modül gerçekleştirmelerini yapmadan problemi çözmenize olanak sağlar.

1960'lardan bu yana birçok ünlü bilgisayar bilimci Barbara Liskov, Alan Kay, Bertrand Meyer ve diğerleri ADT'yi içeren Nesne Tabanlı Programlama, ADT ve Tasarım kontratlı modüler yazılım geliştirme kurallarını prensiplerini oluşturdu. [SOLID](#) 'deki **Liskov Substitution Principle** bunlardan birisidir.

## 1.2 ADT için Spesifikasyonlar

ADT spesifikasyonunun uygunluğunu değerlendirmek için çeşitli kriterler bulunur. Aşağıdaki 1975 yılında Liskov and Zilles tarafından yazılmış bu makalede de bu durumlar değerlendirilmektedir.

## Specification Techniques for Data Abstractions.

**Formal:** Spesifikasyonlar formal olmalıdır. Her bir elemanın anlamı hedef kitle açısından uygulama geliştirmesine uygun olmalıdır. Spekteki her aksiyom için kodda bir cebirsel kanıt uygulamak mümkün olmalıdır.

**Applicable(Uygulanabilir):** Geniş çapta uygulanabilir olmalı. Farklı farklı veri tipleri , dillerde veya ortamlarda aynı şekilde çalışabilmeli, kullanılabilirmeli.

**Minimal(Öz/Sade):** Olabildiğince öz olmalı, fazla veya gereksiz hiç bir detay içermemelidir. Kesin ve açık olmalı , hatta çoğu ADT spesifikasyonu aksiyom kullanılarak kanıtlanabilmelidir.

**Extensible(Genişleyebilir):** ADT'ler genişleyebilir olmalıdır. Gereksinimlerdeki ufak değişiklikler ADT küçük değişikliklere yol açmalı büyük yapısal değişiklikler gerektirmemelidir.

**Declarative:** Nasıl yaptığını değil ne yaptığını tanımlamadır. Veri yapısındaki girdi işlemi ile çıktı işlemi arasındaki ilişki bağlantısını anlatması gereklidir. Örn : **stack.push** dediğimizde eklenen elemanı veri yapısının en üstüne ekleyeceğini ve **pop** deyince en üstekini alacağımızı anlamak bu ilişkili tanımlamaktır. Ama bunu bellekte nasıl tuttuğu ve nasıl yaptığı ile ilgilenmemektir.

# 14. Functors ve Categories

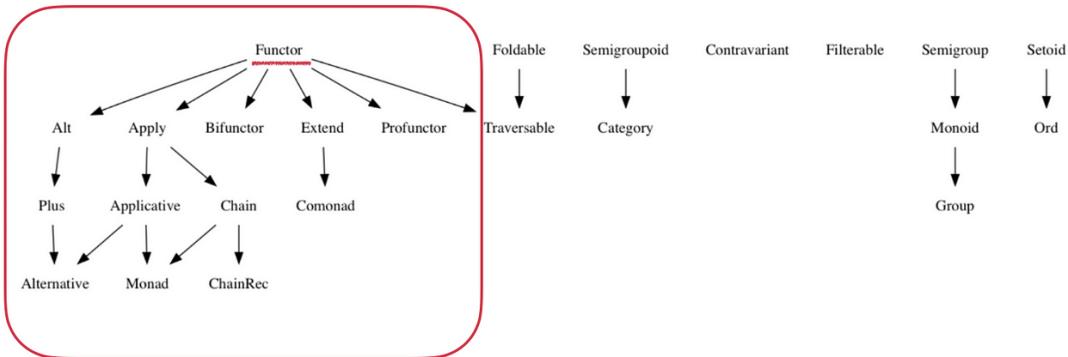
Functors mappable arayüzünden türemiş map() fonksiyonuna sahip yapısını map ile değiştirmeyen ama içeriğini güncelleten veri türüdür.

Bu yazıyı daha önceden yazmış olduğum [JS ile Fonksiyonel](#)

[Programlama](#) yazısının bir devamı olarak yazıyorum. Birçok kavramı tek bir yazında ele almanın yaratacağı karmaşıklıktan ve okunma zorluğundan kaçmak için yazılarımı bölümler halinde yazma kararı verdim.

## 14.1. Functor Nedir ?

Bir önceki yazımada ADT (Abstract Data Types) bahsetmiştim. **Functor** da bir veri tipidir. Bir Değer, String, Array, Object, Stream, Promise vb yapısını bozmadan içeriğini başka bir Değer, String, Array, Object, Stream, Promise **Map etmeyi** sağlıyorsa bu Functor veri tipindedir.



<https://github.com/fantasyland/fantasy-land>

Konunun karmaşık geldiğinin farkındayım. Daha basit bir şekilde anlatmak için örnekler ile anlatmaya çalışacağım. **mappable** interface turemiş **.map()** fonksiyonuna sahip olan **veri tipleri/container** diyebiliriz.

Konuya biraz örnekler ile anlatırsak daha iyi bir şekilde anlaşılacağını düşünüyorum.

## Array Functor

**Array** Functor veri tipidir. Çünkü map fonksiyonu vardır ve buna istediğimiz fonksiyonları vererek arka arkaya **map** fonksiyonu çağırabiliriz. Array yapısı ve eleman sayısı değişmez ama içeriği değişmiş olur. Önce square fonksiyonu ile karesini alıp sonra bu sayılarla +1 ekliyoruz.

```

const square=x=>x*x;
const addOne=x=>x+1;
console.log([ 2, 4, 6 ].map(square).map(addOne));
  
```

## Array Functor

Bunun diğer bir yöntemi de **map().map()** yapmak yerine fonksiyonların birbirini compose etmesi yani sağlayarak tek bir map fonksiyonu içerisinde istenileni yapabilirsiniz.

```
const square=x=>x*x;
const addOne=x=>x+1;
const g=square; const f=addOne;
const h=x=>f(g(x))
console.log([ 2, 4, 6 ].map(h));
```

$h = f(g(x))$

## Function Functor

$h$  fonksiyonu  $f(g(x))$  ile oluşturabildik ama acaba fonksiyonuda `map()` ile yapısını dönüştürmeden çalışma mantığını değiştirebilir miyiz ? Cevap Evet.

```
-| Function.prototype.map = function (f) {
  const g = this
-|   return function () {
    return f(g.apply(this, arguments))
  }
}

const square=x=>x*x;
const addOne=x=>x+1;
const h = square.map(addOne);
console.log([2,4,6].map(h))
```

Function Functor

## String Functor

Örneğin String her bir karakterini alıp bunu bir uppercase fonksiyonundan sonra da Sesli Harfleri 0 dönüştürme fonksiyonundan geçirebiliriz.

```

▼ String.prototype.map = function (f) {
  let result="";
  ▼ for (let i = 0; i < this.length; i++) {
    result += f(this[i]);
  }
  return result
}

const f=x=>x.toUpperCase();
const g=x=>"EA".includes(x) ? 0 : x
console.log("Merhaba".map(f).map(g));

```

## Promise Functor

Promises **.map()** yerine **.then()** metodunu kullanarak async iç içe birbirini içeren promise kaplamaları oluşturarak aynı yapıda promise döner.

```

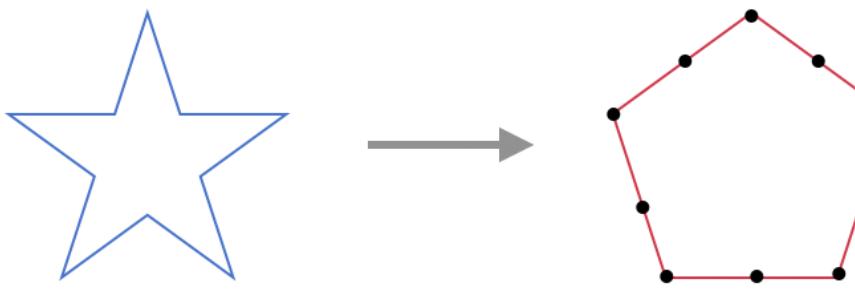
const waitThenCall = (msg) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {resolve(msg + "success")}, 1000)});
}

waitThenCall("Hello1_")
  .then((data) => { console.log(data)
    return waitThenCall("Hello2_"),(err)=>{}}
  .then((data) => { console.log(data)
    return waitThenCall("Hello3_"),undefined)
  .then((data) => { console.log(data)})
  .catch(err) => {console.log(err+"X")})

```

## 14.2. Functor Yasaları

Hayat birçok pattern (örüntülerden) oluşmaktadır. Çiçek yapraklarında, gözde, kar tanesinde vb doğada bir çok görüntü bulunur. Kategori bir yapısı olmalıdır. Örneğin sırası olmalı Array sıralı yapılardır örneği .1 elemanlı bir bir kategori, 2 elemanlı ayrı bir kategoridir. Kategori Teorisi de bu örüntülerin tanınmasını sağlar. Örneğin bir geometrik şeklin başka bir geometrik şekle map/dönüştürülmesidir. Bunu yapan Functor fonksiyonudur. Bunu yaparken yapısının yani nokta sayısının, eleman sayısının değişmediğini görebilirsiniz.

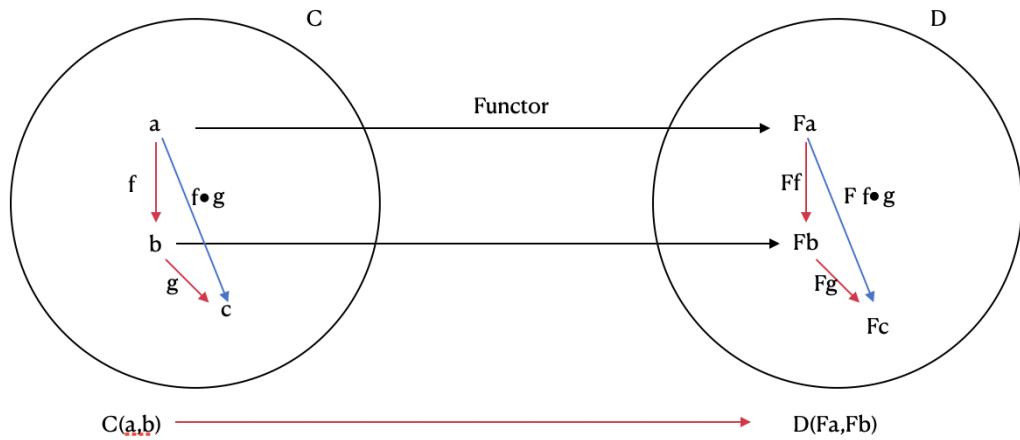


## Morphism

Kategori Functor olabilmesi için aşağıdaki 2 temel yasayı koruması gereklidir.

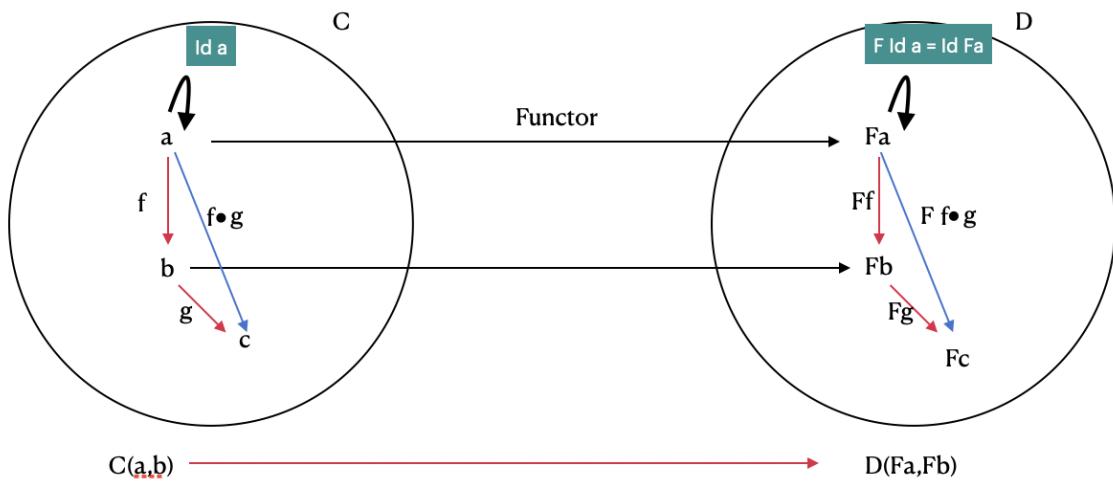
1. Identity (Birim/Kendine dönme):  $f(x) \Rightarrow x$
2. Composition (Kapsama):  $f(g(x)) = map(g).map(f)$

**Composition yasası** bu bahsettiğimiz yapının (structure) korunduğuunun göstergesidir. Aşağıdaki resimde bu durumu göstermeye çalıştım. Functor sayesinde ( $f . g$ ) vektorel eşitliğini görebilirsiniz. Bu sayede map fonksiyonlarını arka arkaya çağırırmak fonksiyon composition ile aynı değeri verir.



Functor Composition (Bartosz Milewski Category Theory)

Identity yasası yani birim fonksiyonun kendine gitmesi . :  $f(x) \Rightarrow x$



### Functor Composition (Bartosz Milewski Category Theory)

```


const Identity = value => ({
  map: fn => Identity(fn(value))
});
const trace = x => { console.log(x);  return x;};
const u = Identity(2);

// Identity law
u.map(trace);           // 2
u.map(x => x).map(trace); // 2


```

Identity by Eric Elliott

### 14.3. Functor’ı Ne İçin Kullanabiliriz ?

addOne fonksiyonu aşağıda kendisine verilen bir sayıyı +1 değeri ile toplayıp geriye dönen pure function

```


const addOne = x => x + 1;
addOne(3);
addOne(100);


```

4  
101

addOne fonksiyonu

Peki her zaman istediğimiz gibi çalışıyor mu ? Örneğin verdığımız argüman sayı değil de Object, String veya Array tipinde de olabilir bunların fonksiyonda düzgün sonuçlar üretmeyeceği bir gerçekdir.

```
const addOne = x => x + 1;  
addOne({val:3})  
addOne("xxxx");  
addOne([1,2])
```

```
'[object Object]1'  
'xxxx1'  
'1,21'
```

Invalid Arguments

Bu tip girdileri kontrol etmek için **number** check koyabiliriz.

```
const addOne = x => {  
  if (typeof x !== "number") {  
    return NaN;  
  }  
  return x+1;  
}  
  
addOne(3);  
addOne({val:3})  
addOne("xxxx");  
addOne([1,2])
```

```
4  
NaN  
NaN  
NaN
```

Ama yine de her matematiksel işlem (**square**, **addTen**, **sum**, **multiply**) için fonksiyonların içerisinde bu kontrollerin olması gerekecektir. Bunu daha yapısal bir hale nasıl dönüştürebiliriz.

- Argümanlar doğru tiplerde olmayabilir.
- Doğru tipte olsa bile içerisinde bir hata oluşturuyor olabilir.
- İçeride kullanılacak değerler async callback elde ediliyor olabilir ve o an için bu değer hazır olmayı bilir.

## Type Functor (Örnek NumFunctor)

NumFunctor sayesinde numeric değerler için işlem yapabılırken , numeric olmayan işlemler için NaN döndürmesini sağlayan fonksiyonu ortaklaşaştırmış olduk. Siz bunu String, Array vb. versiyonlarını yapabilirsiniz.

```

> const NumFunctor = x => ({
>   map: fn => {
>     const val = typeof x == 'number' ? fn(x) : NaN;
>     return NumFunctor(val);
>   },
>   val: x
> });

const square=x=>x*x;
const addOne=x=>x+1;
console.log(NumFunctor(2).map(square).map(addOne).val);
console.log(NumFunctor([2]).map(square).map(addOne).val);
console.log(NumFunctor("2").map(square).map(addOne).val);
console.log(NumFunctor({x:2}).map(square).map(addOne).val);

```

5  
NaN  
NaN  
NaN

Numeric Functor

## Maybe Functor

İç içe olan nesne composition kapsamlarının null ve undefined check yaptığımız birçok kodu bulunur. Bunları Maybe Functor ile giderebiliriz. Bu sayede kod içerisinde defalarca bu kontrolleri yapan kodları tekrarlamamış oluruz.

```

if(user && user.univercity && user.univercity.name) // kontrolü koymamıza gerek kalma
z. Bu ve bunun gibi bütün null undefined check yapısal olarak kapsayabilecek bir Funct
or var.

```

```

const isNothing = value => value === null || typeof value === "undefined";

> const MayBeFunctor = x => ({
>   map: fn => {
>     const val= isNothing(x) ? null : fn(x);
>     return MayBeFunctor(val),
>   val: x
> });

const onur={ univercity: { name:"EGE" }};
const ugur={ univercity: { name:"ITU" }};
const ahmet={ univercity1: { name:"ITU" }};
console.log(MaybeFunctor(onur).map(m => m.univercity).map(u => u.name).val)
console.log(MaybeFunctor(ugur).map(m => m.univercity).map(u => u.name).val)
console.log(MaybeFunctor(ahmet).map(m => m.univercity).map(u => u.name).val)
|_

```

'EGE'  
'ITU'  
null

Maybe Functor

## Either Functor

Left Identity olduğu, Right istenen fonksiyonu gerçekleştirdiği Either Functor oluşturabiliriz.

```
const Left = x => ({x , map: fn => Left(x )}); //Identity
const Right = x => ({x , map: fn => Right(fn(x)))};

const validatePassword = pass => {
  if (pass!=="123") return Left(new Error("password is invalid"));
  return Right(pass);
};

console.log(validatePassword("123").map(v=>v+ " password success").x);
console.log(validatePassword("1234").map(v=>v).x);
```

'123 password success'  
Error: password is invalid  
at validatePassword (eval at <anonymous> (:7:47), <anonymous>:14:35)  
at eval (eval at <anonymous> (:7:47), <anonymous>:19:13)  
at <anonymous>:17:47  
at <anonymous>:125:60  
at WebFrame../lib/renderer/api/web-frame.ts.WebFrame.<computed> [as executeJavaScript]  
(electron/js2c/renderer\_init.js:1590:33)  
at electron/js2c/renderer\_init.js:2844:43  
at EventEmitter.<anonymous> (electron/js2c/renderer\_init.js:2419:57)  
at EventEmitter.emit (events.js:210:5)  
at Object.onMessage (electron/js2c/renderer\_init.js:2188:16)

Either Left/Right

Peki bu Either ile tryCatch mekanizması kurmak istesek Right Functor, Left Functor ve bunları yöneten tryCatch fonksiyonu olsun. Bu tryCatch fonksiyonunu istenilen her exception fırlatma fonksiyonunda kullanabiliriz artık.

```

▼ const Right = x => ({
  map: fn => Right(fn(x)), //Success icin Fonksiyon Calisiyor
  catch: () => Right(x), //Fail icin Identity Kendini Donuyor
  x,
});

const rResult= Right(5).catch(error => error.message).x;
console.log(rResult);

▼ const Left = value => ({
  map: fn => Left(value), //Success icin Identity
  catch: fn => Right(fn(value)), //Fail icin ilgili fonksiyonu cagiriyor
  value,
});

const lResult=Left(new Error("error")).catch(error => error.message).x;
console.log(lResult);

▼ const tryCatch = fn => x => {
  ▼ try {
    return Right(fn(x)); // everything OK
  ▼ } catch (error) {
    return Left(error); // Error
  }
};

▼ const validatePassword = tryCatch(pass => {
  if (pass==="123") return pass;
  throw new Error("password is invalid");
});

validatePassword("123").map(v=>v+" is valid").catch(v=>v+" is invalid").x;
validatePassword("124").map(v=>v+" is valid").catch(v=>v+" Problem").x;

```

try/catch mekanizması

## 14.4 Yorum

Fonksiyonel Programlamada Object Oriented programlamaya göre daha fazla kodlama pattern soyutlaştırma çabası var. Bu sayede olabildiğince tek bir şekilde ve kontrollü kod yazmak mümkün hale geliyor. Tüm kodunuza bu tip kontrol ekleyip bunlar dışında kod yazılmasına izin vermezseniz kodlayıcıları olabildiğince belli bir yolun içerisinde kod yazmaya zorlamış oluyorsunuz.

Object Oriented nesne/domain modelleri, hiyerarşileri ve ilişkileri üzerinde zorunluluklar getirirken, Functional Programming davranış modelleri, ortak kodlama tekrarlarını engelleyecek soyutlamalar üzerine zorunluluklar getirmeyi sağlıyor.

## 15. Monad Nedir?

Fonksiyonel Programlanın önemli konularından Monad nedir konusunu işleyeceğiz. Adını duyduğumuzda anlam ifade etmeyen Monad'ı örnek ile anlatırsak JS Promise bu yapıdadır dediğimde herkesin daha ilgisini çekebilecek bir konu.

Monad konusu internet üzerinde araştırdığınızda çok farklı konular ile karşılaşabilirsiniz.

- Felsefe **Leibniz'in Monad'ına** bir göz gezdirebilirsiniz.
- Matematik **Kategori Teorisi** videolarına bakabilirsiniz.

Bizim konumuz yazılım olduğu için temeli matematikteki **Kategori Teorisi** bu işin temelini oluşturuyor. Ama biz yazılımcılar için bayağı karmaşık ve soyut. En iyisi mi gelin biz bu işe yazılım perspektifinden JavaScript ile bakalım.

Konu zaten karmaşık olduğu için en anlaşılır kısmından başlayalım. Monad'ı nerelerde kullanıyoruz, nerede işimize yarıyor ?

### 15.1 Monad Ne İşimize Yarıyor ?

Monad yapmak istediğiniz işlemler ve hesaplamalardaki tuhaflık ve karmaşıklıklardan bizi soyutlayan katmandır. Bu sayede geliştirici bu işlemlerde esas odağını kaybetmemiş olur.

Örneğin;

#### Promis/Future Monad

Sonucu işlem yaptığınız sırada henüz belli olmayan işlemlerin **sonucunu tutan kap**

```
Promise.resolve($.getJSON('/path.....'))
  .then(function(data) {
    // Do something here...
  });
});
```

## Maybe Monad

Yine bir fonksiyon sonucunda bir değer veya değer üretmediğinden emin olamadığımız durumlarda (undefined veya null) olmasından kaynaklı uygulamamızın **crash (çökmemesini) sağlayacak kap**

```
export class Maybe<T> {
  private constructor(private value: T | null) {}

  static some<T>(value: T) {
    if (!value) {
      throw Error("Provided value must not be empty");
    }
    return new Maybe(value);
  }

  static none<T>() {
    return new Maybe<T>(null);
  }

  static fromValue<T>(value: T) {
    return value ? Maybe.some(value) : Maybe.none<T>();
  }

  getOrElse(defaultValue: T) {
    return this.value === null ? defaultValue : this.value;
  }
}
```

Maybe Monad (<https://codewithstyle.info/advanced-functional-programming-in-typescript-maybe-monad/>)

## List Monad

Rastgele sayıda sonuç döndürebilen kesin olmayan hesaplamaları soyutlamak **icin kullanilan kap**

## I/O Monad

I/O Network işlemi yaparken sonucu kesin olmayan dönüşümleri **soyutlamak icin kullanilan kap**

**Not:** Burada diyebilirsiniz I/O Monad neden Promise Monad farklı. Aşağıdaki yazıları zaman bulup daha detaylı okuyunca bu konuyu tekrardan daha açarak anlatmaya

çalışacağım.

- [The IO monad in Javascript — How does it compare to other techniques](#)
- [Algebraic Effects for the Rest of Us](#)

## Düger Monad

Bunun gibi kullandığımız bir çok monad bulunuyor. Bunların listesini [buradaki linkten](#) erişebilirsiniz

## 15.2 Monad Nedir?

Monad fonksiyonları kapsamanın farklı bir yöntemidir. Bu yöntemde bir context ve işletim, dallanma ve I/O işlemleri Fonksiyonel Programlamadan Side Effect kısmına giriyor. Bu kısmda nasıl bir kap örüntüsü oluşturmalıyız ki bu SideEffect etkilerini azaltabilelim.

JS kullanılan Monad Operasyonlarından bazılarını listelersek;

**lift:** Tipi normal değerden array dönüştürmek ileride yapacağımız işlemlerde birçok kolaylık ve Array High Order Function özelliklerini kullanma imkanı sunacaktır.

```
const x = 20;           // Some data of type `a`  
const arr = Array.of(x); // The type lift.
```

**flatten:** Array yapısında hiyerarşik yapıları tek bir array içerisinde düzleştiren operasyonları içerir.

```
[[1], [2, 3], [4]].flat(); // [1, 2, 3, 4] or  
[].concat.apply([], [[1], [2, 3], [4]]); // [1, 2, 3, 4]
```

**map:** Array içerisindeki her bir değeri bir işleme tabi tutup bunun sonucunu başka benzer bir yapı içerisinde oluşturur.

```
const arr=[1,2,3]  
const f = n => n * 2;      // A function from `a` to `b`  
const result = arr.map(f); // [40]
```

**context:** Monadın hesaplama detayıdır. **Functor / Monad API** ve çalışmaları, Monad'ı uygulamanın geri kalanıyla oluşturmanıza olanak tanıyan bağımlı sağlar.

Functors ve monad'ın amacı, bu bağlamı soyutlamaktır, böylece bir şeyler oluştururken onun için endişelenmememize gerek kalmaz.

## 16. Object-Oriented Programlama ve Tarihçesi

Object Oriented Programlama acaba ilk başta bu programlama paradigmاسını kuran kişilerin kafasında ki yapı mı, yoksa şu anda bambaşka bir şekilde anlaşılmış farklı bir şekilde OOP uygulanıyor. Acaba Java, C++ veya .NET Object Oriented Programlamayı farklı mı yorumladılar ?

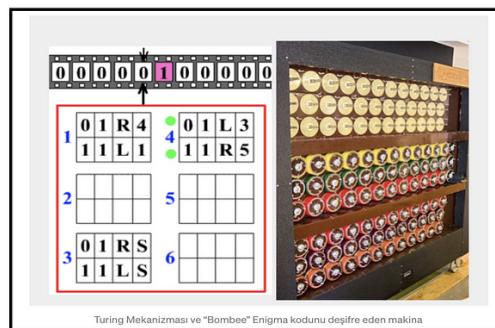
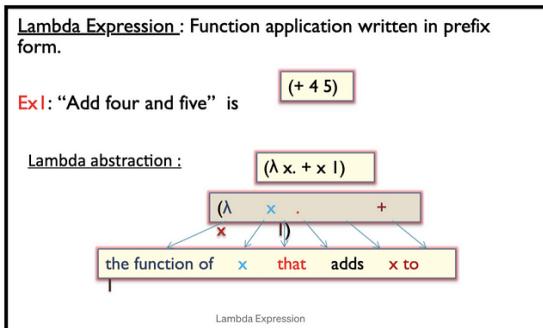
Bugün kullandığımız fonksiyonel ve imperative programlama paradigmaları 1930'larda matematikteki Lambda Calculus ve Turing Machine ile evrensel formülasyon işletimine bir alternatif olarak ortaya çıktı.

Church Turing Tezi bize Turing Makinesi ile Lambda Calculus eşdeğer olduğunu , Turing makinesinde işletilebilen bir kodun aynı zamanda Lambda Calculus içinde işletilebileceğini gösterdi.

Turing Makinesi ile ilgili yanlış bir kanıda her şeyin işletilebilir olduğunu . Ama bazı durumlardan dolayı bu mümkün değil (**örneğin**: halting problem, durdurulamama sorunu). Yazında “işretebilir(computable)” olarak bahsedilen tüm kavramlar Turing Makinesinde çalıştırılabilir kod anlamında bahsedilecektir.

```
while (true) continue//infinite loop  
    print "Hello, world!"
```

Lambda calculus yukarıdan aşağıya fonksiyonların işletimi yaklaşımını benimserken, Turing makinesinde kayan şerit / kayıt makinesinde formülasyonu, hesaplamaya aşağıdan yukarıya, (adım adım) bir yaklaşımı temsil eder.



Lambda Calculus ve Turing Makinesi

Low level programlama dilleri (makine kodu ve assembly) ilk olarak 1940'ların başlarında gözükmeye başladı. 1950'lerin sonuna doğru high-level programlama dilleri gözüktü. Günümüzde de hale kullanılmaya devam eden Lisp dialect devamı olan Clojure, Scheme, AutoLisp vb.... ve bunun yanında günümüz imperative dillerin temelini oluşturan FORTRAN ve COBOL ilk olarak 1950'lerde ortaya çıkmış, daha sonraki yıllarda FORTRAN ve COBOL yazılan uygulamlar C ve C++ ve C aile türevi diller ile tekrardan yazılmıştır.

Dijital bilgisayar çağının ilk evrelerindeki tüm imperative ve fonksiyonel programlama dillerinin kökleri matematik teki **computation theory (hesaplama, işletim teorisine)** dayanır. Object-Oriented Programming OOP (Nesneye-Dayalı Programlama) ise 1966/67 yılında Alan Kay tarafından yüksel okuldayken ortaya atıldı.

Ivan Sutherland's 1961/62 yıllarında geliştirdiği Sketchpad uygulaması ve 1963 yılında yazmış Sketchpad Tezi OOP için öncül ilham kaynağı olmuştur. Objeler veri yapıları olarak grafik resimlerde osiloskop ekranda gösterilip, **dynamic delegation** ve **inheritance kavramlarına** tezinde bahsetmiştir.

Herhangi bir object → (master), bu objenin oluşturulmuş örneklerde → (occurrences) olarak adlandırılmıştır. Sketchpad's masters → örneklerindeki kalıtım yapısı , JavaScript prototype inheritance mekanığınde kullanılmıştır.

MIT Lincoln Laboratuvarı'ndaki TX-2, bir ışık kalemi kullanarak doğrudan ekran etkileşimi kullanan bir grafik bilgisayar monitörünün ilk kullanıcılarından biriydi.



Whirlwind: Preparing the Way for SAGE

1948–1958 arasında çalışan EDSAC, bir ekranda grafikleri görüntüleyebiliyordu. MIT'deki Whirlwind, 1949'da çalışan bir osiloskop ekranına sahipti. Projenin amacı, birden çok uçağı içeren genel bir uçuş simülatörü yaratmaktı. Bu, SAGE hesaplama sisteminin geliştirilmesine yol açtı. The TX-2 was a test computer for SAGE.

İlk object-oriented programlama dili olarak bilinen Simula (Simulation Programming Language) 1965 yılında ortaya çıktı. Simula'da Object, Class, Class Inheritance, Subclass ve Virtual Method kavramlarını kullanmıştır.

Virtual Method bir sınıfın alt sınıflarınca gerçekleştirileceklerinin ezilebileceği metodların tanımlanmasıdır. Java, .NET interface tanımlaman veya abstract sınıfta tanımlanan metodlar alt sınıflar tarafından gerçekleştirilmek zordundadır, veya bir sınıfın subclass içerisinde türediği sınıfa ait bir metodu ezerek implementasyonu değiştirebilir. İşte bu durumda kod compile sırasında değil de, bu kod çalıştırılırken o ana sınıfın hangi subclass implementation çağrılacığı belli olur ve program dallanmaya başlar. JavaScript ise tamamen dinamik bir dil olduğu için tüm

metodları (run-time) belirleyebildiğiniz için bu tür Virtual Method destegine ihtiyaç duymamaktadır.

## 16.1 Büyük Fikir

object-oriented” terimini ortaya atarken hiç C++ düşüncesinde  
değildim~ Alan Kay, OOPSLA ‘97

Alan Kay, Object-Oriented Programlama terminolojisini ortaya 1966/67 yüksek lisans döneminde ortaya atıyor. Buradaki büyük fikir mini-computers bulunan yazılımların encapsulated(dışarıdan soyutlanmış biçimde) birbirlerine mesaj aktarımı ile iletişim kurması ve veriyi paylaşması olarak düşünmüştür. Bu sayede programı veri yapıları (data-structures) ve prosedürler (procedures) olarak bölme zorunluluğu ortadan kalkacaktı.

Recursive tasarımının temel ilkesi, parçaların bütün olarak aynı  
güce sahip olmasını sağlamaktır.~ Bob Barto

Smalltalk Alan Kay, Dan Ingalls, Adele Goldberg ve diğerleri tarafından Xerox PARC kullanılmak için geliştirildi. Smalltalk Simula göre daha Object-Oriented özellikler içeriyordu. Smalltalk hersey objeydi. Sınıf yapıları, integer ve block(closures). 1972 yılındaki original Smalltalk **subclassing** içermiyordu. Subclassing 1976 yılında Dan Ingalls tarafından Smalltalk içerisinde eklendi.

**Not:** Eric Elliott yukarıda subclassing olayına özellikle atıfta bulunmuş. 1972'de ilk Smalltalk içerisinde yoktu diye. Ben bunu şöyle alglııyorum. Bu programlama dili Xerox UI için geliştirildiği için React, Vue ve yeni nesil UI Framework Sınıf ve Sınıf hiyerarşilerinden kurtulma çabaları, Inheritance yerine → Composition kullanmaları. Bu **subclassing** konusunda aşağıdaki yazıları inceleyebilirsiniz.

Smalltalk sınıf ve alt-sınıfları desteklerken bile temel konusu bu sınıf olgusu ve hiyerarşik alt-sınıflar kavramları değildi. Lisp ve Simula'dan ilham alarak ortaya çıkan fonksiyonel bir dildi. Alan Kay burada Endüstri'nin **subclassing odaklımasını** OOP ana faydalardan uzaklaşması olarak görüyor.

OOP , Objects kelimesini icat ettiği için üzgün olduğunu buradaki asıl konu ve büyük fikrin Messaging olduğunu söylüyor. ~ Alan Kay

2003 yılında Email-Exchange Alan Kay Object-Oriented Programming neyi kastettiğini daha açıkça belirtmiş.

“OOP benim için sadece messaging , process state saklayan — local retention ,protection ve extreme late-binding~ Alan Kay

**Not:** Eskiden iOS Programla yaparken ObjectiveC Late-Binding özellikli olduğunu biliyorum. ObjectiveC DynamicTyping yazısını okuyabilirsiniz.

Bir başka deyişle Alan Kay'e göre Object-Oriented Programlamanın temel bileşenleri;

- Message Passing (Mesaj ile Veri aktarımı)
- Encapsulation
- Dynamic Binding

Özetle sektörün üzerinde çok durduğu ve ilk öğretilmeye başlatılan **inheritance**, **subclass**, **polymorphism** Alan Kay'a göre OOP temel bileşenleri değildir.

## 16.2 Object-Oriented Programlamanın Özü

**Message Passing ve Encapsulation** birleşimi bazı önemli amaçlara hizmet eder.

**Not:** Aslında buradaki yazılım içerisindeki programdaki Nesneler ve İletişim kavramları daha büyük ölçekte Yazılım Mimarısında bahsettiğimiz Integration Örüntülerinin'de temelini oluşturan konular. Her ne kadar programlamayı önceden Monolitik yapıda düşünüyor olsak da günümüzde bu sistemler dağıtık , microservice mimari yapısında çalışır bir hale gelmiş oldu.

Tekrar yazıya dönersek **Message Passing ve Encapsulation** önemli amaçlar neydi ?

### Avoiding Shared Mutable State (Paylaşılmış State Kullanma)

Bileşenlerin içerisinde kapsadıkları state(durumları) dışarıdaki diğer objelerden izole et. Obje içerisindeki state değişimi için sadece diğer objeler mesaj yoluyla istekte

bulunabilsinler. Emir vermesinler. State değişimleri local, hücresel seviyede kontrol edilsin, global erişim ve paylaşım yerine..

**Not :** Burada istekte bulunsunlar(request mesajı göndersinler), emir vermesinler çok önemli bir kavram, yani istekte bulunan, istekte bulunduğu objenin iç state bilmemesi gerekiyor).

Bu kısmın daha iyi anlaşılması için React'ın bileşen yapısından örnek vermek istiyorum. React bileşenlerinin bir **internal state** bir de dışarıdan aldığı **props** mekanizması bulunur.

Props üzerinden bilgi alır veya bilgi sorar veya dışarıya bir etkide bulunur. Bu konuda daha detaylı bilgi sahibi olmak için [React Bileşenlerinde State ve Props](#) yazısını okumanızı öneririm.

```
import './App.css';
import FooView from '../src/components/FooView';

function App() {
  return (
    <div className="App">
      <FooView year={2020}/>
    </div>
  );
}

export default App;
```

```
import React, {Component} from 'react';
class FooView extends Component {
  constructor(props) {
    super(props);
    this.state = {age: 30}
  }

  handleClick = (e) => {
    this.setState({age: this.state.age + 1});
  }

  render() {
    const {age} = this.state;
    const {year} = this.props;
    const birthYear = year - age;
    return (
      <div>
        <span>Yaş:{age}</span>
        <span>Doğum Yılı:{birthYear}</span>
        <button onClick={this.handleClick}>Yaşı Artır</button>
      </div>
    )
  }
}

export default FooView;
```

Internal State ve Props Kavramı...

## Decoupling (Ayrışma)

Objeleri birbirinden ayristırma — Message API üzerinden mesaj gönderimi ve alımı yöntemleri ile Objelerin haberleşmesi.

**Not:** Burada 1nci madde ve 2nci Madde için şöyle bir **sitemde de** bulunabilirsiniz. Böyle bileşenler birbirinden tamamı ile bağımsız ve mesajlaşma sistemi üzerinden kendi state yaptığı kurumsal yazılımlar, büyük ölçekli yazılımlar geliştirmek mümkün mü ?

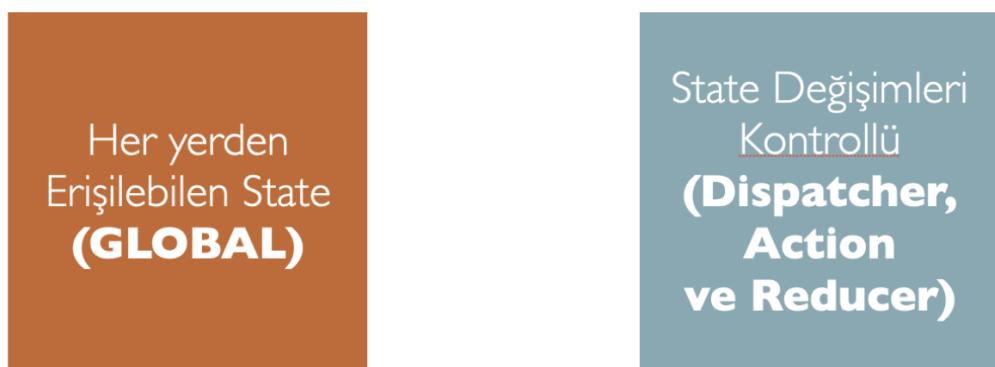
Gerçek dünyadan ve gerçek React App örnek vermek gerekirse; veri-tabanında veya başka bir API üzerinden aldığınız Entity , uygulamanızdaki UI birebir eşleşmiyor. Yani UI bileşenlerinin kendi internal state olmasına karşın bir Entity birden fazla UI tüketiyor ve değiştiriyor ise;

- Bu yapıları nasıl kuracağız ?
- Genel uygulama state nasıl tutulacak?
- Bu yapı gerçek hayatı çalışır mı şeklinde sorular olabilir.

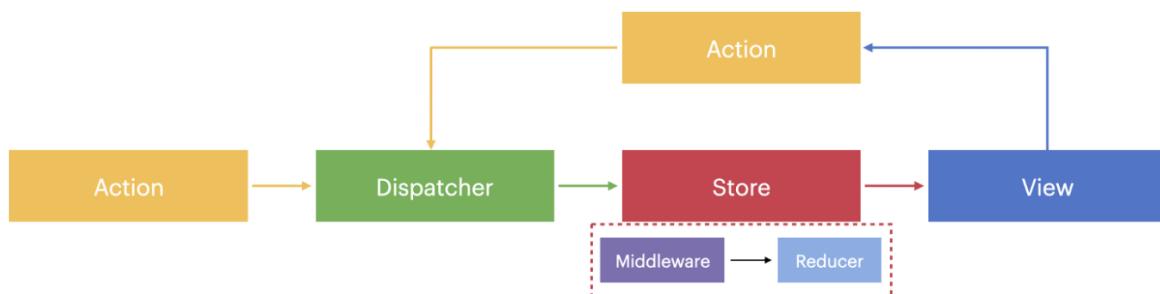
Bu konuda React Bileşen yapısı ve bunların büyük uygulamalardaki kullanımlarından yani **State Management** yaklaşımlarından bahsetmek istiyorum. Bunun için Redux veya GraphQL Apollo gibi kütüphaneler ile bu durumları çözmeye çalışıyorlar.

**Örneğin Redux'ta :**

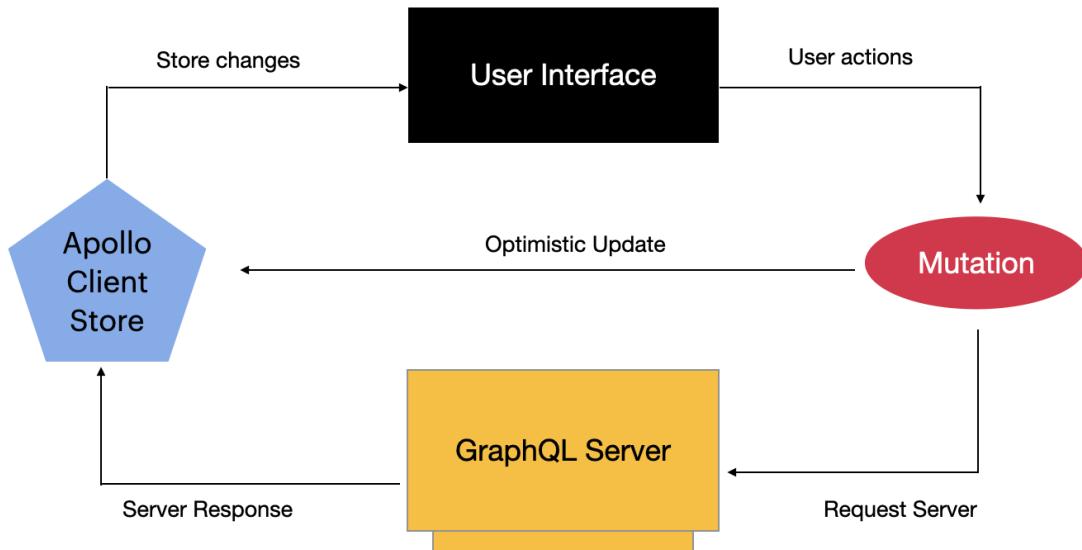
- **Global:** Her yerden Erişilebilen State (Okuma/Read)
- **State Değişimi Kontrollü:** Store State değişimi tek yönden gerçekleşiyor Action → Dispatcher → Store şeklinde



ve bu işlemleri bir mesajlaşma yapısı ve state üzerinden kontrol etme mekanığı var.



**Örneğin Apollo GraphQL de;**



## Adaptability and Resilience to Changes (Değişikliklere Adaptasyon ve Dayanıklılık)

Late-Binding, Dynamic Binding yöntemleri ile bileşenlerin uygulamaya runtime da(kod işletilirken) bağlanabilir mekanizmalara sahip olması Alan Kay'ın OOP için gerekli olduğunu düşündüğü birçok büyük fayda sağlar.

**Not:** JavaScript dinamik bir dil bileşenlerin Runtime bağlayabilecek bir altyapısı bulunuyor. Bu konuya daha sonra detaylı bir örnek ile açıklayacağım.

Alan Kay bu fikirleri biyolojideki hücrelerden ve network üzerindeki bireysel bilgisayarların iletişiminden etkilenderek ortaya atmıştır.

Alan Kay, bu kadar erken bir tarihte bile, bilgisayarların biyolojik hücreler gibi davranışlığı, kendi yalıtılmış durumlarında bağımsız olarak çalıştığı ve mesaj传递 yoluyla iletişim kurduğu dev, dağıtık bir bilgisayarda (internet) çalışan bir yazılım olarak hayal etmiştir. (**Not:** Günümüz Microservis ve Serverless Mimarisi ile sanki o hayalleri gerçekliğe kavuşmuş gibi 😊)

“I realized that the cell/whole-computer metaphor would get rid of data[...]"~ Alan Kay

Alan Kay yukarıdaki cümlede hücre/bilgisayar ana odağın veriden kurtulma yani veri paylaşımının , paylaşılan verinin neden olduğu sıkı bağlantılı yapıların ne tarz zorluklara neden olduğunun farkındaydı.

Ancak 1960'ların sonlarında, ARPA programcılar, yazılım geliştirmeden önce programları için bir veri modeli temsili seçme ihtiyacı nedeniyle hayal kırıklığına uğradılar. Belirli veri yapılarına çok sıkı bir şekilde bağlanan prosedürler, değiştirilmeye karşı dirençli değildi. Verilerin daha homojen bir şekilde işlenmesini istediler.

“[...] the whole point of OOP is not to have to worry about what is inside an object. Objects made on different machines and with different languages should be able to talk to each other [...]” ~ Alan Kay

- Objeler, veri yapısı uygulamalarını soyutlayabilir ve gizleyebilir.
- Bir nesnenin dahili uygulaması, yazılım sisteminin diğer bölümlerini bozmadan değişimdir.
- Aslında, aşırı geç bağlama ile, tamamen farklı bir bilgisayar sistemi bir nesnenin sorumluluklarını devralabilir ve yazılım çalışmaya devam edebilir.
- Bu arada nesneler, nesnenin dahili olarak kullandığı herhangi bir veri yapısı ile çalışan standart bir arayüz ortaya çıkarabilir. Aynı arayüz bağlantılı bir liste, ağaç, akış vb. ile çalışabilir.

Alan Kay ayrıca objeleri cebirsel yapılar olarak görüyordu, bu da davranışlarını matematiksel olarak kanıtlanabilir deterministik mekanikler haline getiriyordu.

“My math background made me realize that each object could have several algebras associated with it, and there could be families of these, and that these would be very very useful.”~ Alan Kay

Bu kanıtlanmış bir doğrudur. Object formunun temelini kategori teorisinden gelen promises veya lenses yapıları oluşturur.

Alan Kay'ın objelere yönelik vizyonunun cebirsel doğası,

- nesnelerin biçimsel doğrulamalara,
- deterministik davranışa
- ve gelişmiş test edilebilirliğe sahip olmasına izin verir

çünkü cebirler(algebras) temelde birkaç kurala uyan denklem formunda işlemlerdir.

Cebir(algebras), sayılar teorisini, geometriyi ve analizi içine alan geniş bir matematik dalıdır. Temel matematik işlemlerinden, çember ve daire alanları bulmayı kapsayan geniş bir çemberi vardır. Temel cebir bilimi, mühendislik ve eczacılık gibi birçok alanda kullanılmaktadır. [Vikipedi](#)

Programcı diliyle **cebir**:

fonksiyonların (aksiyomlar / denklemler) geçmesi gereken birim testler tarafından uygulanan belirli yasaların eşlik ettiği fonksiyonlardan (operasyonlardan) oluşan soyutlamalar gibidir.

Bu fikirler, C ++, Java, C # vb. Dahil olmak üzere çoğu C ailesi OO dilinde onlarca yıldır unutulmuştu, ancak en yaygın kullanılan OO dillerinin son sürümlerinde geri dönmeye başlıyorlar.

Programlama dünyasının OO dilleri bağlamında işlevsel programlamanın ve mantıklı düşünmenin faydalarını yeniden keşfettiğini söyleyebilirsiniz.

JavaScript ve Smalltalk ondan önceki gibi, çoğu modern OO dili giderek daha fazla “çok paradigmalı diller” haline geliyor. Fonksiyonel programlama ve OOP arasında seçim yapmak için hiçbir neden yoktur. Her birinin tarihsel özüne baktığımızda, bunlar sadece uyumlu değil, aynı zamanda tamamlayıcı fikirlerdir.

Ortak birçok özelliği paylaştıkları için, JavaScript'in Smalltalk'in asıl devam edeni olarak söyleyebiliriz.

- Objects
- First-class functions and closures.
- Dynamic types
- Late binding (functions/methods changeable at runtime)
- OOP without class inheritance

Alan Kay'a göre OOP esas olan temeller neydi ?

- Encapsulation
- Message Passing
- Dynamic binding (the ability for the program to evolve/adapt at runtime)

Peki Alan Kay'a göre önemli olmamayan ama Object Oriented Tasarım ve Geliştirmede hep en çok bahsedilen konular nelerdir ?

- Classes
- Class inheritance
- Special treatment for objects/functions/data
- The `new` keyword
- Polymorphism
- Static types
- Recognizing a class as a “type”

Daha önceden Java or C# ağırlıklı yazılım geliştirmişseniz, Nesne-Tabanlı(Object-Oriented) Programlama için temel kavamların aşağıdaki kavamlar olduğunu düşünürsünüz

- Statik Type kullanımı
- Polymorphism
- Inheritance

Fakat Alan Kay programı matematiksel, cebirsel bir denkleme benzetip bu aksiyomlar üzerinden uğraşmanın cebirsel(algebraic) formdaki genel yaklaşımalar ile uğraşmak gerektiğini düşünüyordu. Örneğin;

## Haskell

```
fmap :: (a -> b) -> f a -> f b
```

Yukarıdaki tanımlama **Haskell Functor Map** yapısı , Functor kavramı map edilebilir anlamında yapılar için kullanılıyor. Örneğin JS array'de bulunan `map()` gibi. Burada map **statik bir tür ihtiyacı duymadan** bir veri yapısının içerisindeki değerlere bir fonksiyon etkisi oluşturup başka bir değere dönüştürüyor.

Bu Map Functor yeteneğini Örneğin Frontend Development sırasında sunucudan elde ettiğimiz JSON nesnelerini → JSX React Element dönüştürken sıkça kullanıyoruz.

## JavaScript

Aşağıdaki JavaScript örneğinde de array içerisinde bulunan değerler çift sayımı(2'ye bölünebiliyor mu) yoksa tek mi sorunun cevabını oluşturup başka bir array içerisinde bunun cevaplarını yazmasını istiyoruz.

Burada verdiğimiz array içerisindeki değerler **number**, sonuç array içerisindeki değerler ise **boolean**

```
// isEven = Number => Boolean
const isEven = n => n % 2 === 0; const nums = [1, 2, 3, 4, 5, 6];
// map takes a function `a => b` and an array of `a`s (via `this`)
// and returns an array of `b`s.
// in this case, `a` is `Number` and `b` is `Boolean` const results = nums.map(isEven);
console.log(results); // [false, true, false, true, false, true]
```

Yani **map** fonksiyonu bu algoritmayı çalıştırırken kendisini array ile soyutlamıştır. Array içerisinde tutulan değer ile ilgilenmemektedir. Array içerisinde başka bir tür olması **map** fonksiyonunu etkilememektedir. Bu değerleri işleyecek olan **f** fonksiyonu farklı olacaktır.

```
[1, 2, 3, 4, 5, 6] -> f -> [false, true, false, true, false, true]
```

Örneğin bu array içerisinde haftanın günleri olsaydı, fonksiyonda örneğin **isWorkingDay** olsaydı, **map** fonksiyonunu değiştirmemiz gereklidir miydi? Cevap hayır. İşte generic type'ın sağlamış olduğu bu tür avantajlar bulunur.

**.map()** genel type çalışma işlevini iyi işletir çünkü diziler(array), cebirsel(algebraic) functor yasalarını uygulayan veri yapılarıdır.

Type(Veri Türleri) **.map()** için önemli değildir çünkü onları doğrudan değiştirmeye çalışmaz, bunun yerine uygulama için doğru türleri bekleyen ve döndüren fonksiyonları uygular.

```
// matches = a => Boolean
// here, `a` can be any comparable type
const matches = control => input => input === control;
const strings = ['foo', 'bar', 'baz'];
const results = strings.map(matches('bar'));
console.log(results);
// [false, true, false]
```

Bu genel tür ilişkisinin TypeScript gibi bir dil ile doğru ve kapsamlı bir şekilde ifade edilmesi zordur. Ama Haskell'de Hindley Milner Tür Sistemi daha yüksek türler

desteğiyle ifade edilmesi oldukça kolaydır.

**Not:** Bu blog yazıldıktan sonra TypeScript'e birçok güncelleme geldi. Burada anlatılan TypeScript ve [Hindley Milner Type System](#) arasındaki temel farkların ne olduğunu henüz tam anlamış değilim. İlerleyen yazınlarda TypeScript konularında belki bu kısma biraz daha detaylı değiniriz.

Çoğu Tür Sistemi (Type System) aşağıdaki birçok konunun tasarımda ve gerçekleştirmesinde ciddi sınırlandırmalar getirir.

- özgür dinamik tanımlamalar (free expression of dynamic),
- fonksiyonel fikirler (functional ideas),
- fonksiyonel kapsamaları (function composition),
- object kapsamaları (free object composition),
- runtime object extension,
- combinators,
- lenses, etc,

Düğer bir deyişle Type Sistemleri Composable Software yazmayı zorlaştırmır.

Eric Eliotta göre

“Tür sisteminiz çok kısıtlayıcıysa (ör. TypeScript, Java), aynı hedefleri gerçekleştirmek için daha fazla dolambaçlı kod yazmak zorunda kalırsınız. Bu, statik türlerin kötü bir fikir olduğu veya tüm statik tür uygulamalarının eşit derecede kısıtlayıcı olduğu anlamına gelmez.(Örn Hindley Milner Type System)”

- Statik Type kullanıyor ve seviyorsanız, bunların fanı iseniz,
- Static Type kısıtlamalarına aldırmış etmiyorsanız,
- Bu yazındaki cebirsel düşünmek ve genelleştirmeyi zor buluyorsanız,

Bu yukarıdaki fikirlerin uygulanabilirliğinden çok Type System'in sizi sınırlamasından kaynaklıdır. Daha özgür ve dinamik kodlama için daha fazla serbestlik derecesine sahip araçlara ihtiyacınız var. Sizi sınırlandırmaması gerekiyor.

**Kendi Düşüncelerim:** Burada benim görüşüm ise, TypeScript ve Type sistemleri özellikle Enterprise(Kurumsal) büyük projeler geliştirirken birçok kişinin beraber çalıştığı, modüllerin kendi içerisinde birbirini kullandığı, FE ve BE entegrasyonlarının olduğu yerlerde Type Sistemlerinin sınırlandırmaları ürünün sağlıklı gelişip büyütübilmesi için önem teşkil ediyor.

Bu yüzden öncelikle çok dinamik olmak isteyip istemediğinize karar vermeniz daha önemli.

İkinci bir nokta Java, C++ Object Oriented yaklaşımlarını anlamak çok daha basit, öğrenme eğrisi ve düşünme şekli insan aklına daha yatar. Fonksiyonel Programlarda bu matematiksel ve cebir düşünce tarzı herkesi kolay bir şekilde algılayıp uygulayabilecegi kavramlar değil. Biraz matematik background(altyapısı) ile programlamaya başlamak lazım.

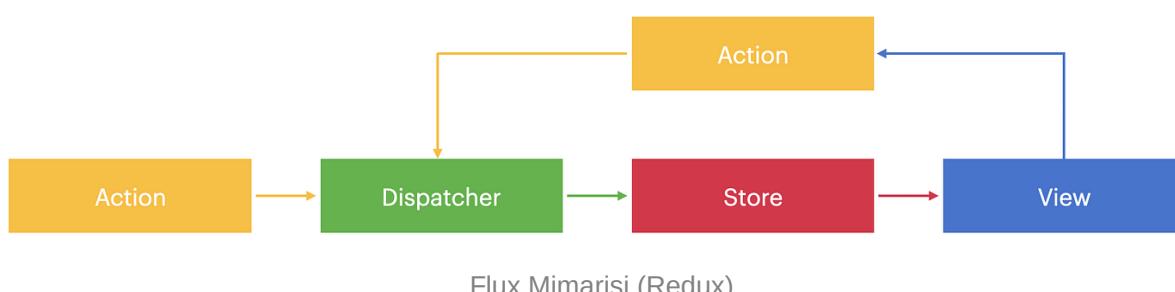
## 16.3 İyi Bir MOP(Monitoring-Oriented Programming) Nasıl Olmalı?

Çoğu modern yazılımda,

- UI 'dan sorumlu, Kullanıcı etkileşiminden sorumlu kısım
- Uygulama State Yönetmekten sorumlu kısım
- Sistem ve Network I/O yönetmekten sorumlu kısım

Bahsedilen kısımlar(sistemler), event listener ile ağ durumu, UI bileşen state durumları, Uygulama durumları vb. kısımları dinlemek ve gerekli aksiyonları almak için uzun-süre yaşayan processlerdir.

İyi bir Monitoring-Oriented Programlama gerçekleştirebilmek için sistemdeki bileşenlerin birbirlerine direkt ulaşıp manipüle etmek yerine birbirine mesaj gönderimi yöntemi ile ulaşması şeklinde olmalıdır.



Kullanıcı bir kaydetme düğmesine tıkladığında, bir uygulama durumu bileşeninin yorumlayabileceği ve bir durum güncelleme işleyicisine (pure reducer) aktarabileceği bir “KAYDET” mesajı gönderilebilir.

Durum güncelledikten sonra, durum bileşeni bir UI bileşenine “STATE\_UPDATED” mesajı gönderebilir, bu da durumu yorumlayacak, UI’nin hangi bölümlerinin güncellenmesi gerektiğini birleştirerek ve güncellenmiş durumu alt bileşenlere iletecektir.

Ağ bağlantısı bileşeni, kullanıcının ağdaki diğer bilgisayarlar ile ilgili iletişimini ve uzak makinede bu UI state’e ilettilip, kaydedilmesi veya çekilmesi ve UI gösterilmesi işleminden sorumlu olur.

Asıl mantık bileşen(sistemlerin) birbirlerinin işlevlerinden ve ayrıntılarından haberdar olmaması önemlidir. Her modül bireysel kaygıları ile vereceği hizmeti mesaj yoluyla alarak işletmekten sorumlu olmalıdır.

Birlikte çalışabilmeleri için standartlaştırılmış arayüzler üzerinde çalışmalıyız. Arayüz tatmin edici olduğu sürece, aynı şeyi farklı şekillerde yapabilen güncellemeleri veya tamamen farklı gerçekleştirimi aynı mesajlarla yapabilirsiniz. Bunu çalışma zamanında bile yapabilirsiniz ve her şey düzgün çalışmaya devam eder. (**Not:** Bu kısmda örneğin Backend API ile iletişimde GraphQL → Query/Mutation üzerinden Schema Mesaj ile birçok işi dinamik ve runtime değiştirilebilir yapılabılır.)

Aynı yazılıma ait bileşenlerin aynı makinede yer alınmasına bile gerek yoktur. Sistem decentralized bir biçimde çalıştırılabilir. (**Not:** Bu konuşulan bana biraz **React Server Component** gelen kabiliyetleri hatırlatıyor) . Bunun yanında veri depolama kısmı bile decentralized bir biçimde ağ üzerinde dağıtık bir şekilde tutulabilir. (Örneğin IPFS)

OOP(Object-Oriented Programlama) Arpanet esinlenerek geliştirilmiştir. Arpanet önemli bir hedefi de decentralized ağ üzerinde oluşacak ataklara karşı dayanıklı bir sistem geliştirmektir. (Stephen J. Lukasik Why the Arpanet Was Built)

İyi bir MOP (Monitoring-Oriented Programming) sisteminde internetin sağlam bir şekilde işletilebilmesi için uygulamalar çalışırken bileşenlerin değiştirebilir olmalıdır. Örneğin uygulama çalışmasına telefonundan devam ederken offline olsa bile uygulamanın çalışmaya devam etmesi (**Not:** Service Worker- PWA — Progressive Web App) . Örneğin veri merkezi bir anda çökse de fonksiyonların işlerine devam etmesidir.

Günümüz teknolojisi ve gereksinimleri (Dağıtık ve Serverless Çalışma İhtiyacı) artık yazılım dünyasının başarısız sınıf kalıtım (**Class Inheritance**) deneyini bırakıp, başlangıçtaki OOP ruhunu tanımlayan matematik ve bilim ilkelerini dönenmenin zamanı.

**Not:** JavaScript ile Modern UI Framework, Görselleştirme Kütüphaneleri veya Yapay Zeka konuları biz geliştiricileri bu yönde ilerlemeye zorlayacaktır.

Uyum içinde çalışan MOP ve işlevsel programlama ile daha esnek, daha dayanıklı, daha iyi oluşturulmuş yazılımlar oluşturmaya başlamanın zamanı geldi.

## 17. Object Composition (Nesneleri Birleştirme)

Nesneleri birleştirme becerisi, bir yazılımcının en çok ihtiyaç duyduğu yeteneklerden birisidir. Bu birleştirme(composition) yöntemlerini teknikleri nelerdir ve nasıl kullanılmalıdır ?

Soyutlama ve Kapsama/Birleştirme(Abstraction & Composition) konusunda daha önceden detaylı bir yazı yazmıştık. Bugün bu konunun teknik detaylarına anlamaya çalışacağımız.

“Object Composition Assembling or composing objects to get more complex behavior.” ~ Gang of Four, “Design Patterns: Elements of Reusable Object-Oriented Software.”

Object Composition yönteminde, günümüz kompleks projelerini oluşturmak için nesneleri birleştirme veya birbirine kapsama yöntemi ile büyüterek istediğimiz projeleri oluşturuyoruz.

Burada Gang of Four “*Design Patterns*” kitabında üzerine basarak bahsettiği bir konuda

“Favor object composition over class inheritance.” ~ Gang of Four, “*Design Patterns*”.

Object-Oriented Sınıf üzerinden kalıtım yerine Object Composition yöntemimi tercih etmemizi öneriyor. Ama genelde Kalıtım mecburi gibi düşünerek her yerde bunu

kullanmaya çalışıyoruz.



### Inheritance vs Composition

Domain nesnelerinde anlamlı olabilen bu yaklaşımın (yani Properties) , Behaviour paylaşımı için çok uygun olmadığını hatta birçok zararının bulunduğu düşünüyorum. Zamanında bu sınıf hiyerarşinin oluşturduğu problemlerden dolayı ciddi refactoring yapmam gerekmisti. (Refactoring hakkında detaylı bilgi için [bu Link'e](#) gidin).

Bu nedenle UI kütüphanelerinde Inheritance yerine → Composition yapılarını tercih ediyorlar. Bu konuyu daha önceki yazılardan örnekler vererek React kapsamında anlatayım.

## 17.1 React Composition vs. Inheritance

**Composition:** Bileşenlerin birbirinin referanslarını tutarak onlara bir iş vermeleri veya onları kullanmaları sonucundaki ilişkidir.

Örneğin Table bileşeni Header, Rows, Columns, Sorting , Search, Pagination gibi bir sürü bileşeni içerip bunları koordineli bir şekilde çalıştırmasına deriz. React yolu da genellikle budur.

**Inheritance** ise React tarafından çok da tercih edilen bir yöntem değil. Hatta Class Component yerine Function component ve Hook kullanımı sayesinde Kalıtım olayına çok da gerek kalmayacaktır. Örneğin ben bir fonksiyon yazacağım; renderShape() bunu bir ana sınıfa yazıp bundan türeyen Rect, Triangle sınıflarında bu renderShape kullanabilirim. Bu daha çok Java'nın kullandığı OOP gelen bir yöntemdir. JS bunu daha çok fonksiyon olarak dışında tanımlar ve diğer sınıflardan kullanmana olanak sağlayan yöntemi tercih eder.

## 17.2 React Code Sharing Yöntemleri

Bu konuda önceden yazmış olduğum [React Code Sharing](#)

Örnekleri yazısında **Default**(Code Duplication), **Inheritance**, **HOC**(High Order

Component), **RenderProps** ve **Hooks** yöntemlerini denemişti.

Default			
Inheritance			
HoC			
RenderProps			
Hooks			RED

Default, Inheritance, HOC, RenderProps, Hooks

Eğer siz uygulama geliştirme yapınızı **is-a ilişkilendirmesi** ile yani **ördek bir kuşur ilişkisi** üzerinden yapıyorsanız bunun çok büyük sıkıntıları olacaktır. Bu ilişki kalıtım aldığı objeye çok sıkı bir ilişki ile bağlandığı için;

- **The fragile base class problem:** Kalıtım alınan ana sınıfın değişikliklere açık olması. Bu durumda bu ana sınıfın türeyen ve bunun alt sınıflarından türeyen tüm sınıfların bu değişimden etkilenenecek olması.
- **The gorilla/banana problem:** Bu konuda Joe Armstrong Erlang Programlama dilinin tasarımcısının aşağıdaki cümlesindeki problem ortaya çıkar. Aslında siz ufak bir objeyi kullandığınızı zannederken arkaplarda bu objeyinin bağımlı olduğu bir goril hatta bir orman kadar çok nesne olabilir.

I think the lack of reusability comes in object-oriented languages, not functional languages. Because the problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

If you have referentially transparent code, if you have pure functions — all the data comes in its input arguments and everything goes out and leave no state behind — it's incredibly reusable. You can just reuse it here, there, and everywhere.

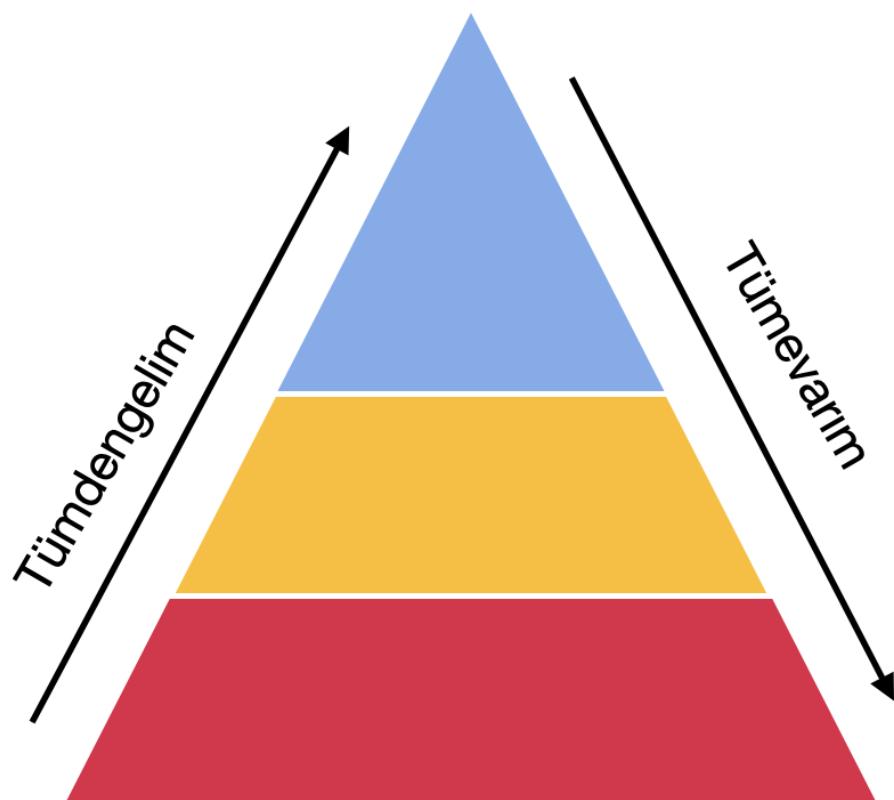
- **The duplication by necessity problem:** Normalde yapısal olarak hiyerarşi sayısı arttığında yeni bir türeyen oluşturmak istediğimizde sadece bir türden

türetemedigimiz 2 farklı ataya sahip olacak yapıların oluşması bu durumda da kodun duplicate edilmesi gerekecektir. Bu da bizim ***DRY prensibine*** aykırı bir durumdur.

DRY (Don't repeat yourself): Kodunuzda farklı modülün içerisinde benzer fonksiyonallitede yapılar bulunur, bunları her modülde tekrar yazmanız kod tekrarına(code dublication) neden olacaktır. Olabildiğince belli işlerin sorumluluğunu ilgili modüllere vererek abstraction iyi sağlamak gerekiyor.

### 17.3 Abstraction (Soyutlama)

Tekrar kullanabilirlik için bir yapıyı soyutlamak(abstraction) gereklidir. Soyutlamanın 2 temel parçası bulunur. ***Generalization ve Specialization***



Kodun içerisinde tasarım yaparken ve yazılımı iyileştirirken sürekli bu 2 kavram üzerinden ilerleriz.

**Generalization (Genelleme)** : Tekrar eden örüntüleri ve işlemleri çıkarıp bu benzerlikleri bir soyutlama arkasında saklamaktır. Farz edelim tüm aldığımız ürünlerde %18 kdv ekleniyor. Tüm muhasebe yazılımı yapan uygulamaların bunu yapması yerine bunu yapan bir fonksiyon, sınıf veya kütüphane yazarak bu muhasebe işlemini soyutlayabiliriz.

**Specialization (Özelleşmek)** : Özelleştirmek'de genelleştirmenin aksine sadece o duruma özel işlemleri , mantıkları oluşturarak soyutlamak anlamına gelir.

## 17.4 Inheritance ile Soyutlama (Abstraction)

Inheritance kullanıyorsak;

- **Genelleme(Generalization) kısmını**, birbirine benzeyen sınıflar için ortak arayüzlerden(interface) oluşturulmasını ve ortak fonksiyonların abstract veya ata sınıfa taşınması ile gerçekleşir
- **Özelleştirme (Specialization)** kısmını ise atasındaki benzemeyen davranışlar için ilgili metodu override ederek kendi davranışını oraya enjekte etmesi ile gerçekleşir.

**Not:** İlerleyen yazınlarda Soyutlama ve (Generalization, Specialization) için tek yöntemin Inheritance olmadığını bunun yerine Object Composition faydalananabileceğimizi anlatacağım.

## 18. Kalıtım Türleri (Inheritance Types)

Kaç farklı türde inheritance (kalıtım) bulunuyor ? Prototypal Inheritance ne demek ? Kalıtım türlerinin yan etkileri var mı ? Bu yazıda bu konular üzerinde duruyor olacağız.

Favor composition over inheritance → Composition, kalıtımı tercih edin derken Java'daki sınıflar için düşünüldüğünde bu kavram çok doğru, çünkü sınıf kalıtım konusunun çok ciddi yan etkileri ve zararları bulunuyor. Aşağıdaki konulara bir önceki yazımız Object Composition konusunda değinmiştik.

Aşağıdaki kısım Eric Elliott What “you were taught was not prototypal inheritance” ile başlayan cevabından bir alıntıdır.

Eric Elliott göre tüm kalıtım yöntemleri kötü ve zararlı değil. Sadece java, .Net gibi dillerde olan **extends** yöntemini kullanan Class Inheritance yöntemlerini zararlı buluyor. Bu tür bir sınıflandırma baştan yaptığımız dinamikliği kaybetmemiz ve aşağıdaki tasarım ve kod problemlerinin oluşmasına neden oluyor.

- **The tight coupling problem:** Ata sınıfa çok sıkı bağımlılık
- **The fragile base class problem:** Ata sınıfındaki değişkenlik problemi
- **Inflexible hierarchy problem:** Hiyerarşinin esnekliği yok etmesi
- **The duplication by necessity problem:** Aynı yeteneğin farklı ata sınıflarda yer alma ihtiyacı davranışın yer aldığı kodun duplicate edilmesine neden oluyor.
- **The Gorilla/banana problem :** Siz uygulamanızda ufak bir objeyi kullanmak istediğinizde aslında arka planda bunun bağımlı olduğu çok fazla yapının olması. Çünkü hiyerarşisinden dolayı ihtiyacınız olmayan bir çok yapıyı zorunlu tutmak problemi.

Peki tamam bu problemler iyi de buna alternatif kalıtım yöntemleri neler ? Başka kalıtım türleri neler ?

- **Functional inheritance**
- **Concatenative inheritance**
- **Prototype delegation**

Diğer kalıtım türlerini aşağıda analiz etmeye çalışalım.

## 18.1 Functional inheritance (Fonksiyonel Kalıtım)

Fonksiyonları **constructor** nesne oluşturucu olarak kullanma. Örneğin aşağıdaki kodu inceleyelim. Cat → Animal türeyen bir yapıda olmasını istiyoruz. Eninde sonunda elimizde belli özellikleri olan ve davranışları olan bellekte yer tutan objelerimiz olacak. Örneğin;

- Animal (Hayvan) → name
- Cat (🐱) → sayHello

olduğu durumda Cat nesnesi önce elindeki veriyle Animal Objesini oluşturup, o Objeye kendi fonksiyonlarını ekleyerek fonksiyonel olarak kalıtımı gerçekleştirir.

```

// Base object constructor function
function Animal(data) {
    var that = {}; // Create an empty object
    that.name = data.name; // Add it a "name" property
    return that; // Return the object
};

// Create a child object, inheriting from the base Animal
function Cat(data) {
    // Create the Animal object
    var that = Animal(data);
    // Extend base object
    that.sayHello = function() {
        return 'Hello, I\'m ' + that.name;
    };
    return that;
};

// Usage
var myCat = Cat({ name: 'Rufi' });
console.log(myCat.sayHello());
// Output: "Hello, I'm Rufi"

```

<https://www.jstips.co/en/javascript/what-is-a-functional-inheritance/>

## 18.2 Concatenative inheritance (Birleştirici Kalıtım)

Object.assign yöntemleri ile ilgili fonksiyon iskelet yapısını klonlayarak yeni objeler oluşturma imkanı sağlar. Yapıları birleştirme altyapısı çok esnek olduğu için ***mixin*** yapılarını oluşturabilme özgürlüğünə sahipsinizdir.

<pre> 1 const proto = { 2     sayHello: function hello() { 3         console.log(`My name is \${this.name} and I am 4             \${this.age} years old.`) 5     } 6     function User(name,age){ 7         const user=Object.assign({}, proto, {name,age}); 8         return user; 9     } 10 const onur=User("Onur",39); 11 onur.sayHello(); </pre>	<pre> 1 2 3 'My name is Onur and I am 39 years old.' </pre>
--	---

Object.assign

## 18.3 Prototype Delegation (Delegasyon ile Kalıtım)

JavaScript objeler içerisinde prototype isminde özel bir yapı bulunur. Bir objenin içerisinde ilgili prop veya metod bulunmadığında bu prototype aracılığıyla delegation yöntemi ile içerde bulunan \_\_proto\_\_ nesnesinde varmı diye bakılır. Bu konuda aşağıdaki yazınlarda bu konulara detaylıca dejindim.

- [JavaScript Kalıtım Nasıl Gerçekleşir ?](#)
- [Javascript' te prototype , \\_\\_proto\\_\\_ Kavramlarını Anlamak](#)



```
1 user.prototype.sayHello=function(){           1 'My name is Onur'
2   console.log('My name is ${this.name}');      2 'My name is Deniz'
3 }
4
5 function user(name){
6   const obj=Object.create(user.prototype);
7   obj.name=name;
8   return obj;
9 }
10
11 user('Onur').sayHello();
12 user('Deniz').sayHello();
```

```
< f user(name){  
  const obj=Object.create(user.prototype);  
  obj.name=name;  
  return obj;  
}  
> user.prototype  
< {sayHello: f, constructor: f} ✖  
  > sayHello: f ()  
  > constructor: f user(name)  
  > __proto__: Object  
> user()  
< user {name: undefined} ✖  
  > name: undefined  
  > __proto__:  
    > sayHello: f ()  
    > constructor: f user(name)  
    > __proto__: Object  
>
```

JavaScript Prototype Delegation Yöntemi..

Hatta siz ne kadar JS class, extends gibi yapılar kullanırsınızda JavaScript bunu arkaplanda Prototye ve Delegation yapısı üzerinden yönetir.

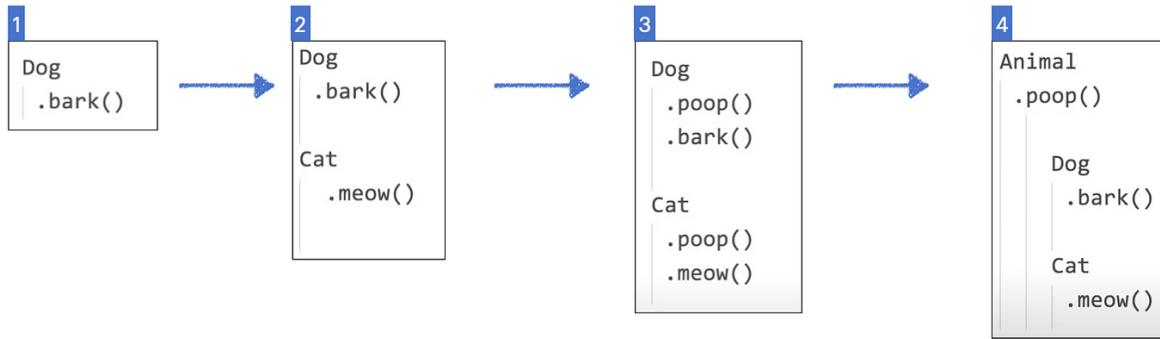
## 18.4 Inheritance over Composition

Inheritance over Composition, kalıtımın dil içerisinde konulmuş extends, class gibi reserved keyword ile değilde başka yöntemler ile diğer objeleri kapsayacak hale gelmesidir ? Peki nasıl gerçekleştirilir ?

Bu yazışta bildiğimiz Class ve extends yöntemi ile kalıtım yönteminin neden sorunlar çıkardığını bunun yerine Inheritance over Composition nasıl yapacağımızı analiz edeceğiz. Aşağıdaki video bunu çok iyi bir şekilde örneklemiş o yüzden bu videodaki örnek üzerinden anlatıyor olacağız.

### Composition over Inheritance

Yukarıdaki videoda gerçek bir yazılım geliştirme yapısını küçük bir örnekle simule eden bir çalışma yapmış, bunu gerçekleştirken öncelikle OOP Class Inheritance ile nasıl yapılacağını yorumlamış;

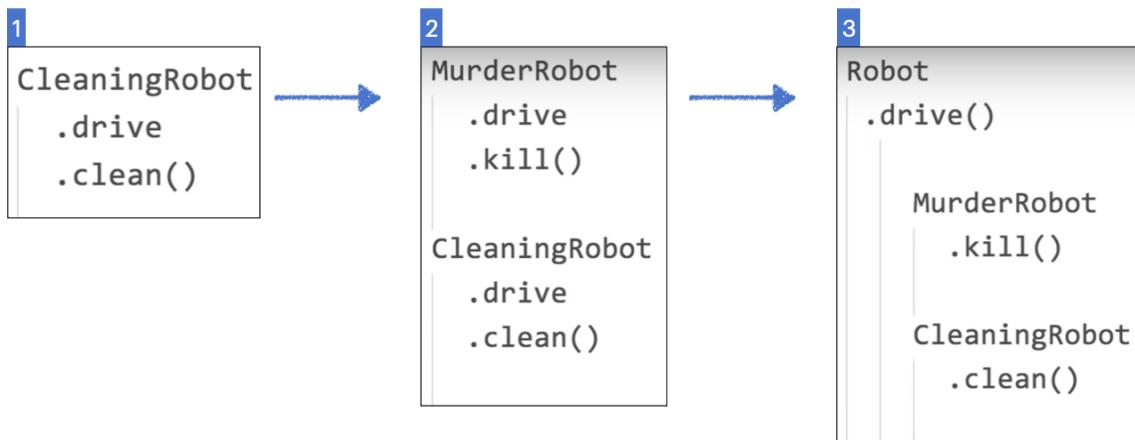


Dog, Cat, Animal

Burada bu adımların projenize aynı anda değil de belli zaman aralıkları ile eklendiğini düşünün

- Önce sistemde havlayan bir köpek olsun.
- Sonra sistemde miyavlayan bir kedi olsun isteniyor,
- Daha sonrasında kedi ve köpek kaka yapsın, gidiyor 2 nesneye bu **poop()** metodlarını ekliyor
- **poop()** duplicate oldu bunu engellemek için bir parent sınıf oluşturup (Animal), içerisinde poop taşıyıp, bunun altına nesnelerimizi hiyerarşik olarsak taşıdık.

Zaman içerisinde sistem biraz daha gelişti bu durumda, hayvanların kakalarını temizleyecek bir temizleme robotu oluşturduğu durumu yaptığını varsayalım.

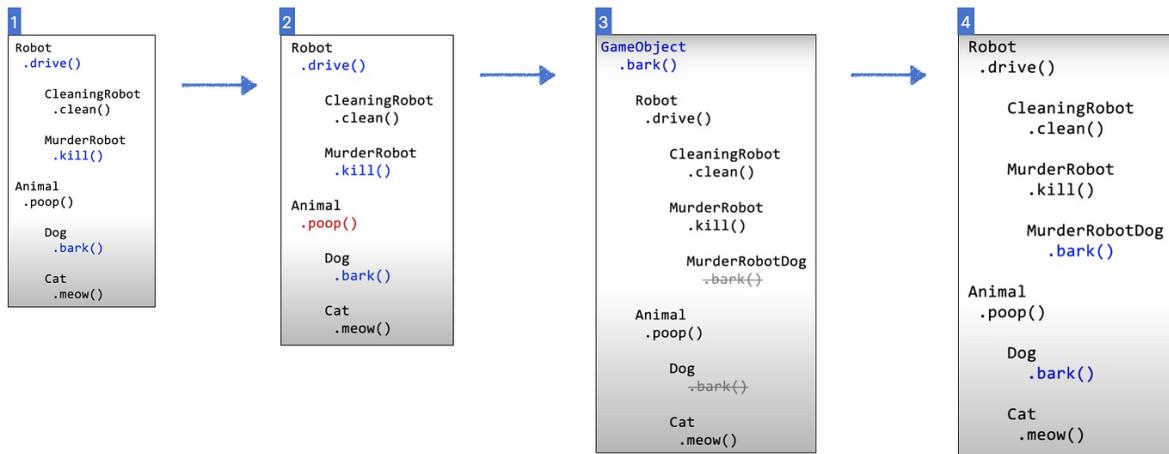


Cleaning, Murder, Robot

- Temizleme Robot(Cleaning Robot)un sürüsü ve temizleme özelliği olsun.
- Katil Robot (Murder Robot)un sürüsü ve öldürme özelliği olsun istendi..

- Burada robotun sürüş yetenekleri ortak ozaman Robot parent sınıfı altında bu drive() fonksiyonu taşıyarak gerçekleşir.

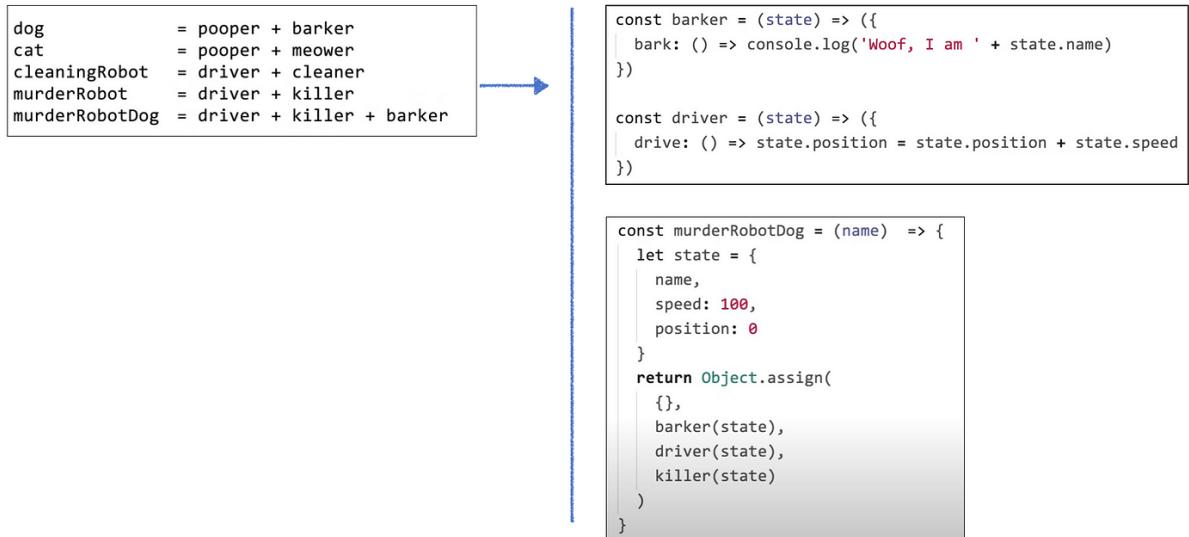
Peki zaman ilerledi sistem güzel bir şekilde işledi, ileride müşterinin veya yöneticilerin isteği ile **Havlayan, Katil, Köpek Robot olsun** isteniyor. Bu durumda bizim kurduğumuz hireyarşik yapıya uymuyor



### GameObject

- 1nci resimde **kill()** ve **bark()** Robot ve Animal içerisinde dağılmış durumda.
- 2nci resimde bu köpek robotun poop() özelliği olmasın isteniyor.
- 3üncü resimde Bu durumda 2 parent nesneden türediği bir GameObject ve bunun bark özelliği olsun dedik ama buda çok mantıklı değil çünkü en üstte aldığımız **bark()** havlama bir çok alt sınıf için uygun değil.
- Bu durumda 4ncü resmi oluşturuyoruz ama **bark()** fonksiyonalitesi duplicate etti. DRY felsefesine ters.

Bizim hiyerarşik yapıdan çok davranışları ekleme yapılmasına ihtiyacımız var. Bu durumda Concatenative inheritance (Birleştirici Kalıtım) Object.assign ile arada istediğimiz davranışı oluşturup kalıtımı çok daha esnek hale getirebiliriz.



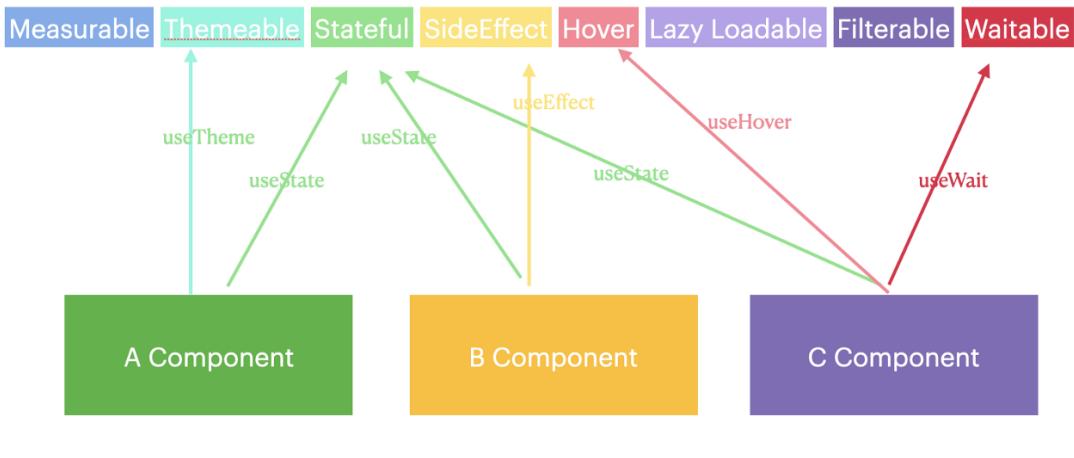
Concatenative Inheritance

Kalıtım Türlerinde bahsettiğimiz fonksiyonlar üzerinden dinamik yapı oluşturmaları bize çok büyük esneklikler sağlayacaktır. Burada bahsi geçen **Composition Over Inheritance**, kalıtımı bu yönde kullanmaya çalışılmasıdır.

**Not:** Composition over inheritance yöntemi UI kütüphanelerinde sıkça kullanılan bir method'dur, ister JQuery bakın, ister backbone hepsinin altyapsında **extends** yerine Composition Over Inheritance tercih ettiğini göreceksiniz.

React bu gidişatını Class Component → Hooks API'si yönünde geçiş'i ile gerçekleştirdi.

Bu konuda da aşağıdaki resim yukarıda anlattığım duruma benzer bir ihtiyacın Component dünyasındaki yansımıası olarak görebilirsiniz.



Hooks Kullanımı

## 19. Factory Functions — Obje Üretim Fonksiyonları

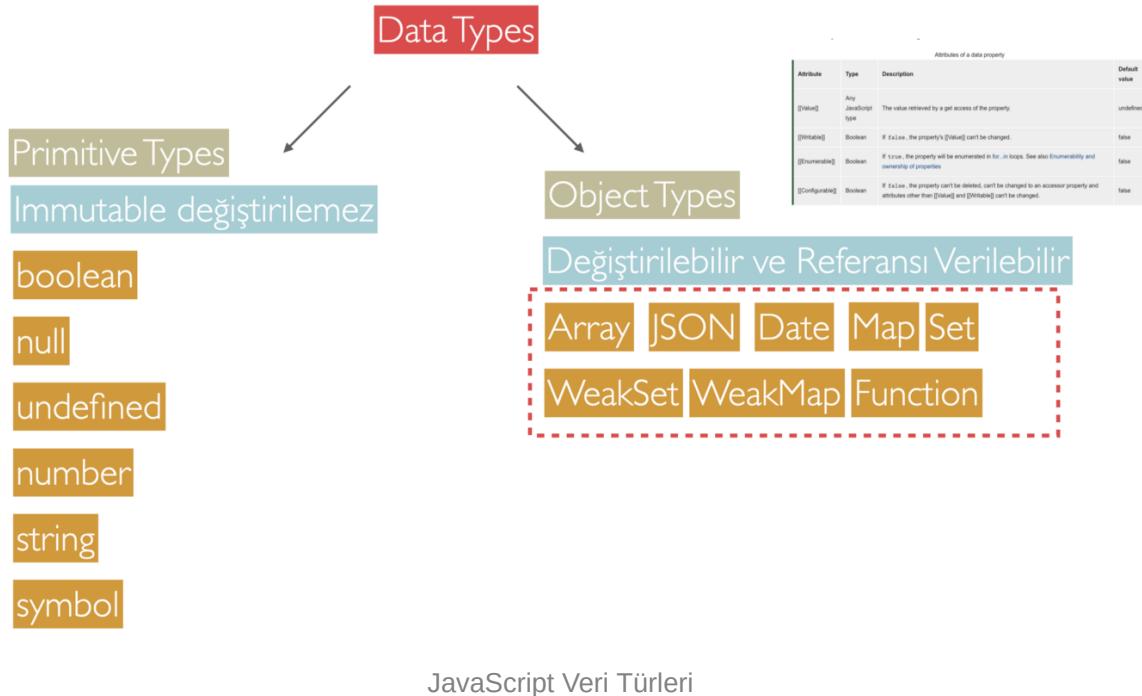
JavaScript'de class , constructor, new kullanmadan obje üretmeye odaklanmış fonksiyonlara Factory Functions deniyor.

Yine Eric Elliott **Composing Software** kitabından devam ediyoruz. Yukarıda da belirttiğim gibi Class, new ve constructor kullanmadan yeni nesne oluşturan fonksiyonlara **Factory Functions** denir.

Bu konuda daha önceden yazmış olduğum yazılarından alıntılar yaparak konuyu anlatmaya çalışacağım. Obje üretmek ve bunları yönetmek her dilde en büyük problemlerden bir tanesidir. Bu konuda OOP(Nesne Tabanlı Programlama) Tasarım Örüntülerinin Creational Pattern amacı `new` yönetmek üzerindedir.

Bunun yanında JS üretim olayını daha dinamik ve basit yaklaşabildiği için karmaşık nesneleri üretebilmek çok daha kısa adımda gerçekleştirilmektedir. Object neden önemlidir.

JavaScript Veri Türlerinde Object Types büyük bir yer oluşturduğunu daha önceki yazılarımızda anlatmıştım.



JavaScript Veri Türleri

Ve bu objeleri oluşturmak için Object içerisinde key , value değerlerinde value → obje/array/primitif type olabilecek şekilde iç içe object yapıları oluşturabilir.

Örneğin aşağıdaki Address objesinde olduğu gibi. Bu sayede çok karmaşık obje tanımlamaları yapabilirsiniz.

**Address:**

```

{
  Street: 'Main',
  Number: 100
  Apartment:
  {
    Floor: 3,
    Number: 301
  }
}

```

Address Object

JavaScript nesne oluşturma avantajı ise dilin bunu JSON nesnesi gibi tanımlamayı izin vermesinden gelir. Örneğin aşağıda bir tane address nesnesini oluşturduk.

```
const address = {  
    street: 'Main',  
    number: 100,  
    Apartment: {  
        Floor:3,  
        Number:301  
    }  
};
```

## 19.1 Objeler Tanımlama

Boş bir obje tanımlayalım.

```
const emptyObject={};
```

Aşağıdaki objemizin içerişine **isim** ve **yas** değişken olarak tutmak isteyelim. Daha nesneyi tanımlarken JSON şeklinde tanımlayabiliriz.

```
const onurObj={isim:"onur", yas:12};
```

veya boş bir obje oluşturup ona **.degiskenIsmi** ile değişken değerlerimizi atayabiliriz.

```
const onurObj2={}  
onurObj2.isim="Onur"  
onurObj2.yas=39
```

veya bir map objesi gibi **[“degiskenIsmi”]** erişerek atama yaparız.

```
const onurObj={}  
onurObj[“isim”] = “Onur”  
onurObj[“yas”] = 39
```

Eğer amacımız tek bir obje oluşturup bunu kullanmak ise yukarıda bahsettiğimiz yöntemler oldukça kullanışlıdır. Fakat amacınız bunun gibi bir çok benzer obje oluşturmak ise örneğin ali, veli, ahmet, ayşe vb.. alt alta bu kodları tekrar tekrar

kopyalamamız anlamına gelir ki bu istediğimiz bir şey değildir. Burada aklımızda bu nesneleri soyutlayacak bir **Function Constructor** oluşturabiliriz.

Aşağıdaki örnekte kullanıcı (User) olarak bir fonksiyon yazıyorum ve bu fonksiyon parametrelerinde değişkenleri alıyor. Bu soyutlama sayesinde kod tekrarından kurtuluyoruz.

```
function User(name,age){  
    let user={};  
    user.name=name;  
    user.age=age;  
    return user;  
}  
  
const onurObj=User("Onur",39);
```

Bizim burada aynı sınıflarda olduğu gibi nesne oluşturmayı ve bu nesnenin fonksiyonlarının olmasını dahi **Factory Functions** ile gerçekleştirebilirsiniz.

```
1 const createUser = ({ name, age }) => ({  
2     name,  
3     age,  
4     setName(name) {  
5         this.name = name;  
6         return this;  
7     },  
8     setAge(age) {  
9         this.age = age;  
10        return this;  
11    },  
12});  
13 console.log(createUser({ name: 'onur', age: '40' }));  
14 console.log(createUser({ name: 'ali', age: '42' }));  
15 const veli = createUser({ name: 'veli', age: '40' });  
16 console.log(veli.setAge(41));  
17  
18 }  
19 {  
20     name: 'onur',  
21     age: '40',  
22     setName: f setName(),  
23     setAge: f setAge()  
24 }  
25 {  
26     name: 'ali',  
27     age: '42',  
28     setName: f setName(),  
29     setAge: f setAge()  
30 }
```

createUser fonksiyonu.. ([Kaynak Kod](#))

**Ekstra Not:** Arrow fonksiyonun çevresinde () parantez olması içerisindeki logic çalışarak return etmesini sağlar. Bunu çağrımadığınız taktirde değeriniz undefined olacaktır. Aşağıdaki resimde bunun çalıştırıldığı örneği görebilirsiniz.

```
18 const user = () => {  
19   name: 'ali';  
20 };  
21 const user2 = () => ({  
22   name: 'ali2',  
23 });  
24  
25 console.log(user()); // undefined  
26 console.log(user2()); // ali2  
27
```

```
18  
19  
20  
21  
22  
23  
24  
25 undefined  
26 { name: 'ali2' }
```

Arrow Functions ([Kaynak kod](#))

## 19.2 Destructuring İşlemleri

Aynı Obje ve Array tanımlamada kullandığımız kısımları tanımlamanın sağ kısmında değil de , sol kısmında kullanırsak yaptığımız obje ve array'i bozabiliriz yani oluşturulma parçalarına kolayca erişebiliriz.

### Obj Destructuring İşlemi

```
obj={name:'onur', age:'40'};  
const{name,age}=obj //Destructuring
```

### Array Destructuring İşlemi

```
colors=['red','green','blue'];  
let [val1,val2,val3]=colors //>> let val1=colors[0] ...
```

## 19.3 Default Parameters

Javascript geçirilen değerlerin her zaman undefined olup olmamasını kontrol etmek ve buna default değerler atamak için yaptığımız if kontrolleri vardır. Bu tip durumlarda Default Parameters sizin kodunuzu daha güzel ve okunabilir hale getirir.

```
//Without Default Parameter  
function ekle(arr,val){
```

```

if(arr==undefined) arr=[];
arr.push(val);
return arr;
}

//With Default Parameter
function ekle(arr=[],val){
  arr.push(val);
  return arr;
}

```

Bazen değişken ilk aşamada bir default değer atanıp daha sonra içerisinde geçirilen değişkenlerden bu değerin oluşacağı durumlar olabilir. Örneğin **Redux store** bu tip durumlar için store default değerleri default parameter olarak fonksiyona geçirilir. Aşağıdaki toplama işleminde undefined değer geçtiğimizde bunları default parametrelerden kullandığını görebilirsiniz.

```

function sum(a=10, b=2){
  return a+b;
}

sum (2,2) //4
sum () //12
sum (3) //5

```

Son olarak aşağıdaki örnekte sadece değer geçilmemiği ve undefined geçildiği durumda default parameter çalıştığını görebilirsiniz. Diğer falsy değerler için bu durum geçerli değildir.

```

function test(num = 1) {
  console.log(typeof num);
}

test();          // 'number' 1
test(undefined); // 'number' 1
test('');        // 'string' ''

```

## 19.4 Rest Kullanmak

Sadece belli değişkenlerle uğraşıp diğer kalanları rest içerisinde atmak isteyebilirsiniz. Obje örneği

```

const props={name:'Deniz',surname:'Dayibasi',other:{age:8, height:128} }
const {name,...rest}=props;
console.log(rest)

```

Array örneği

```
const colors=['red','green','blue'];
let [val1,...rest]=colors
console.log(rest) //green , blue
```

## 20. Factory Functions - Functional Mixins

JavaScript'de class , constructor, new kullanmadan obje üretmeye odaklanmış fonksiyonlara Factory Functions deniyor.

Bu yazıyı 2 parçada incelemek istedim. 19ncu bölümde daha basit anlamda düz, statik yöntemler ile obje oluşturma üzerindeyken, bu yazı daha çok JS obje üretmeyi daha dinamik nasıl yapılabildiğini anlatmak üzerine olacak.

### 20.1 Property Üzerinden Obj Oluşturma

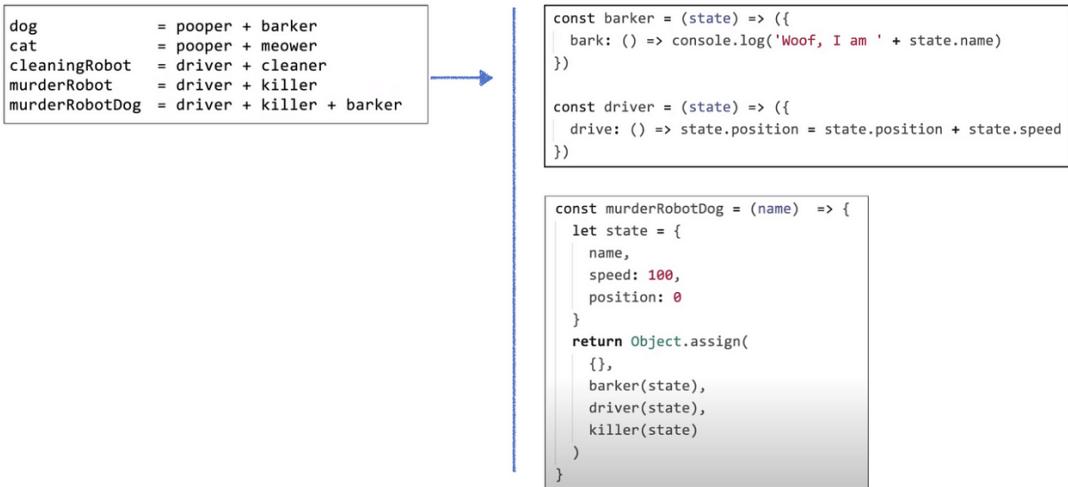
Konuyu anlatmadan kodla bu durumu gösterelim. Bir user nesnesi oluşturmak isteyelim ama user objesinin ne olacağını o an için bilmiyoruz. User tanımını bir dosyadan veya db'den okuduğumuzu düşünelim. Elimize geçen meta veriyle nasıl **user objesi** oluşturduğumuzu görebilirsiniz.

```
1 const meta = { key: 'name', val: 'onur' };
2 const user = {};
3 user[meta.key] = meta.val;
4 console.log(user);
5 |
```

User Objesi

### 20.2 Factory Functions for Mixin Composition

Önceki yazılarımıza [Object Composition \(Inheritance over Composition\)](#) yapısını kullanmanın direk extends yöntemleri yerine kullanmaya göre daha avantajlı olduğunu anlatmıştım. Yapışal hiyerarşi yerine objeleri ve fonksiyonlarını birbirine ekleme yöntemi ile oluşturabileceğimizi anlatmıştık.



Concatenative Inheritance

Bunu diğer bir yöntem olan Mixin Composition ile nasıl gerçekleştirebiliriz?

Eric Elliott yazısındaki örnekten gidersek

```

> const withConstructor = (constructor) => (o) => ({
  __proto__: { constructor },
  ...o,
});

const pipe =
  (...fns) =>
  (x) =>
    fns.reduce((y, f) => f(y), x);
// or `import pipe from 'lodash/fp/flow';`
// Set up some functional mixins
const withFlying = (o) => {
  let isFlying = false;
  return {
    ...o,
    fly() {
      isFlying = true;
      return this;
    },
    land() {
      isFlying = false;
      return this;
    },
    isFlying: () => isFlying,
  };
};
const withBattery =
  ({ capacity }) =>
  (o) => {
    let percentCharged = 100;
    return {
      ...o,
      draw(percent) {
        const remaining = percentCharged - percent;
        percentCharged = remaining > 0 ? remaining : 0;
        return this;
      },
      getCharge: () => percentCharged,
      getCapacity: () => capacity,
    };
  };
};

const createDrone = ({ capacity = '3000mAh' }) =>
  pipe(withFlying, withBattery({ capacity }), withConstructor(createDrone))({});

const myDrone = createDrone({ capacity: '5500mAh' });
console.log(`
  can fly: ${myDrone.fly().isFlying() === true}
  can land: ${myDrone.land().isFlying() === false}
  battery capacity: ${myDrone.getCapacity()}
  battery status: ${myDrone.draw(50).getCharge()}%
  battery drained: ${myDrone.draw(75).getCharge()}% remaining
`);
console.log(`
  constructor linked: ${myDrone.constructor === createDrone}
`);

const aDrone = myDrone;
console.log(aDrone);

  can fly: true
  can land: true
  battery capacity: 5500mAh
  battery status: 50%
  battery drained: 0% remaining

  constructor linked: true
  ▼ createDrone {fly: f, land: f, isFlying: f, draw: f, getCharge: f, ...} ⓘ
    ► draw: f draw(percent)

```

## Mixin Composition Örneği (Kaynak Kod)

Aşağıdaki ana fonksiyonumuz ve kendisini `__proto__` içerisinde constructor olarak alacak.

```
const withConstructor = constructor => o => ({
  // create the delegate [[Prototype]]
  __proto__: {
    // add the constructor prop to the new [[Prototype]]
    constructor
  },
  // mix all o's props into the new object
  ...o
});
```

Bir diğer ihtiyaç ise arka fonksiyonları reduce ederek bize önceki fonksiyonların çıktısını bir sonraki fonksiyona geçirebilecek bir yapıdır. Bunun için pipe fonksiyonunu kullanıyoruz.

```
const pipe =
(...fns) =>
(x) =>
  fns.reduce((y, f) => f(y), x);
// or `import pipe from 'lodash/fp/flow';
// Set up some functional mixins
```

```
const createDrone = ({ capacity = '3000mAh' }) =>
  pipe(withFlying, withBattery({ capacity }), withConstructor(createDrone))({});
```

### createDrone fonksiyonu (örnekten)

Yukarıdaki kod bize sırası ile fonksiyonları arka arkaya girdi olarak çağrıarak bu çağrımlar sonucunda tek bir fonksiyon olarak döndürmesini sağlar. Pipe fonksiyonun nasıl çalıştığını biraz analiz edelim. [Pipe Function in JS](#) yazısı içerisindeki örneği alıp daha iyi anlaşılması için trace fonksiyonunu ekledim.

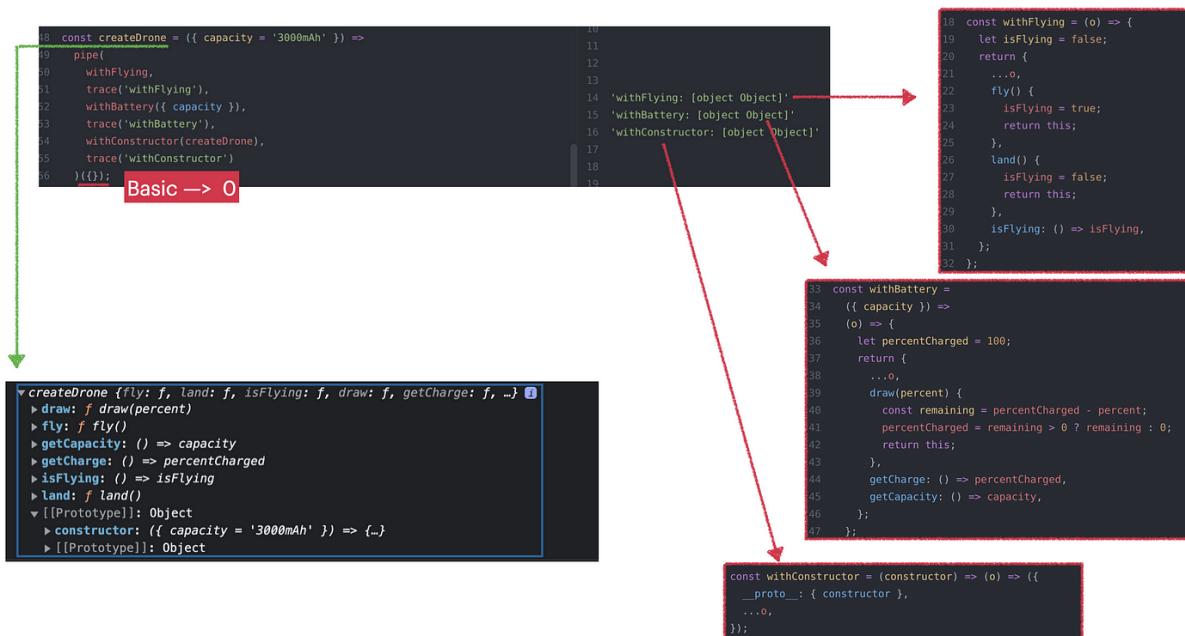
Aşağıdaki örnekte bir hesaplama işlem silsilesi var. Bu işlemleri arka arkaya işletmek istiyoruz. Sırası ile önce 5\$

- `calcTotalWithTax` işletilir çıktı
- `costForTwo` geçirilir bu işletilir bunun çıktı
- `cadToUSD` geçirilir bu işletilir ve sonuç

```
1 const _pipe = (a, b) => (arg) => b(a(arg));
2 const pipe = (...ops) => ops.reduce(_pipe);
3
4 const calcTotalWithTax = (pizzaCost) => pizzaCost * 1.13;
5 const costForTwo = (itemCost) => Math.round((itemCost / 2) * 100) / 100;
6 const cadToUSD = (cad) => Math.round(cad * 0.753653 * 100) / 100;
7 const trace = (label) => (value) => {
8   console.log(`\${label}: \${value}`);
9   return value;
10 };
11
12 const costPerPersonUsd = pipe(
13   calcTotalWithTax,
14   trace('calcTotalWithTax g'),
15   costForTwo,
16   trace('costForTwo f'),
17   cadToUSD,
18   trace('cadToUSD c')
19 );
20 console.log(`You have to pay \$ \${costPerPersonUsd(5)} US.`); // You have to pay
21   $2.13 in usd
```

## Pipe Function Nasıl Çalışıyor ([Kaynak Kod](#))

Mevcut örneğimizde de aynı ama bu sefer amacımız sadece hesaplama yapmak değil ara objeler oluşturup, ara objelerin fonksiyonlarını parametre olarak geçirip yavaş yavaş fonksiyon toplaması gerçekleştirmek. Sırası ile withFlying, withBattery ve withConstructor çalışarak istediğimiz mixin createDrone nesnesini oluşturmuş oluruz.



## Pipe İşleminin Çalışması

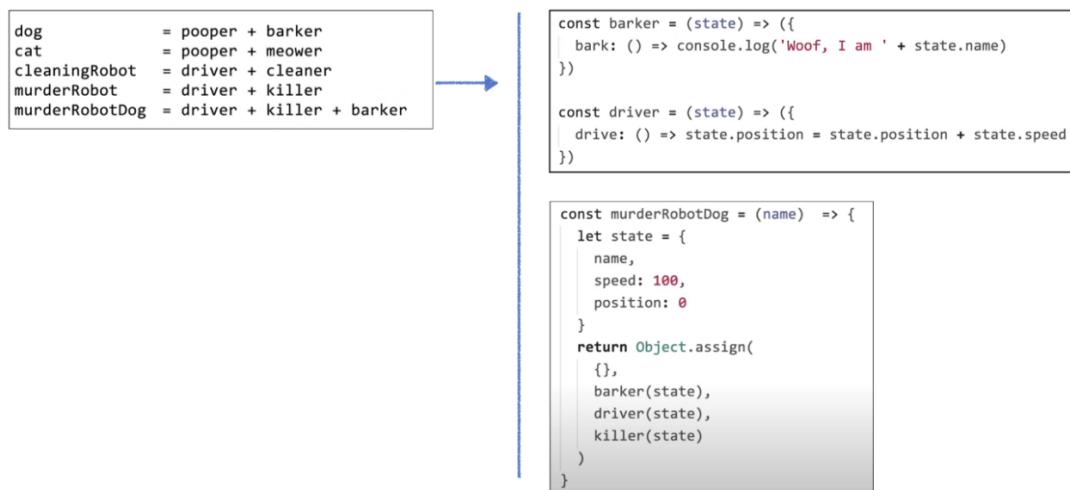
# 21. Sınıflar üzerinden Composition Neden Zor ?

Bu yazıda ise Function Factory ve Mixin ile yaptığımız işlemleri Sınıflar(Class) ile yaparken ne tip zorluklarla neden olduğu üzerinde duracağız.

Bu yazının bazı kısımları Eric Elliott [Why Composition is Harder with Classes](#) yazısından çevirisidir. Yazının orijinal haline erişip direkt okuyabilirsiniz. Ben konuyu anlamak açısından önceden yazmış olduğum yazılardan bazı kısımları bu yazıya ekledim.

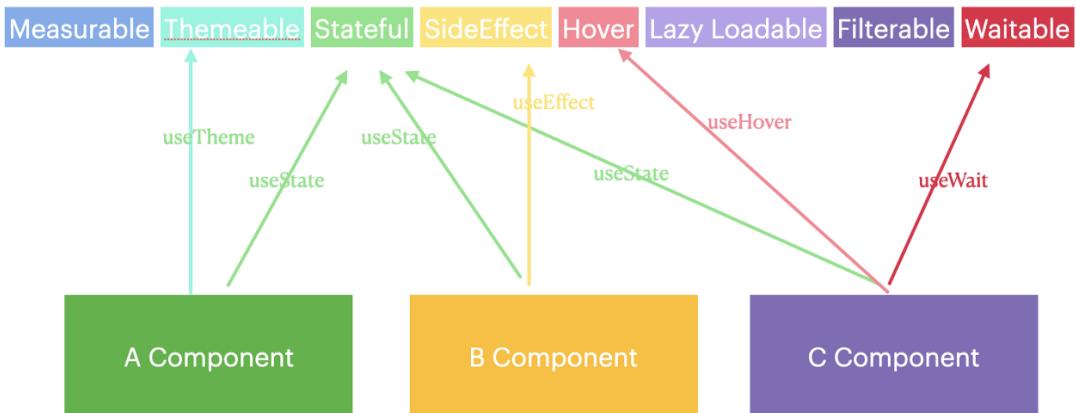
Bir önerim daha olacak. Bu yazı [Factory Functions\(Functional Mixins\)](#) yazısı ile ilişkili bir yazı, bundan dolayı öncesinde yukarıdaki linkteki yazıyı okumanızı öneririm.

Factory Functions yazısındaki temel mantık farklı farklı yeteneklere sahip özellikleri extends etmeden bir objede nasıl bir araya getirebileceğimiz üzerineydi.



Concatenative Inheritance

Aslında yukarıda bahsettiğim konuda React Hooks konusundaki konseptler ile benzer, React Hooks, bir nevi Class(Sınıf) Bileşenlerden kurtarabilmek için sadece durumu farklı şekilde ele alacak bir yapı geliştirdi.



## 21.1 React Hooks Nedir?

Fonksiyonel Programlama temelleri üzerine kurulan Hooks'lar belli standartlar kullanarak React üzerinde ortak kullanılacak kodları daha kolay yönetmemizi sağlıyor.

Ve bu sınıflardan kurtarma isteği bu yazıda bahsedeceğimiz benzer nedenlere dayanıyor. Bu konuda [React Hook neden çıktı ?](#) yazısını okuyabilirsiniz.

ES6 ile birlikte JavaScript dünyasında hayatımıza giren **class** anahtar sözcüğünden normal fonksiyonlara göre kullanımında 2 temel fark var

- *constructor*
- *new*

ama arkaplanda

- yeni bir obje oluşturup *this* anahtar sözcüğünü *constructor* fonksiyonu içerisinde bind ediyor.
- geriye bu *this* objesini dönüyor.
- *instance.\_\_proto\_\_ === Constructor.prototype* atıyor.
- *instance.constructor === Constructor*

Aslında tüm yapılan işlemler **functional mixin** yönteminin aksine oluşturulan sınıfı belli kısıtlamalar getirirken, bu işlemleri daha kompleks ve ekstra maliyetler ile yerine getiriyor.

## 21.2 The Delegate Prototype

En nihayetinde **`new`** ile sınıf oluşturmak yerine bu yeteneği factory function taşımak istediğimizde client kodunda değişiklik ihtiyacı olacak. Birincil olarak sınıf ve constructor yapılarının aksine factory fonksiyonları delegate prototype otomatik link oluşturmazlar.

Sınıflarda yer alan **[[Prototype]]** linki prototype delegation için hafızada milyonlarca nesne varsa , hafızayı koruma yöntemi veya 16 ms'lık bir oluşturma döngüsü içinde bir nesne üzerindeki on binlerce özelliğe erişmeniz gerekiyorsa, programınızdan bir mikro performans artışı elde etmek için kullanılır. (**RxJS ve ThreeJS gibi özel rendering ihtiyacı olaran kütüphanelerde**)

Belleği veya performansı mikro olarak optimize etmeniz gerekmeyor ise **[[Prototype]]** linki faydalanan çok zarar getirebilir. Bunun yerine JS defaultunda yer alan Prototype Chain (Prototip Zinciri) **`instanceof`** operatörüne güç sağlar ama 2 neden dolayı yanlış duruma düşürebilir.

ES5 `Constructor.prototype` linki dinamik ve reconfigurable dır. Bu sizin abstract factory oluşturabilmeniz için kullanılabilir bir özelliktir. Ama bunun üzerinden **`instanceof`** kullanacaksanız bu size yanlış bir sonuç döner.

```
Constructor.prototype !== [[Prototype]]
```

Bunun nedeni de yukarıdaki 2 nesnenin bellekte aynı noktayı işaret etmemesinden kaynaklanır.

```
class User {  
  constructor ({userName, avatar}) {  
    this.userName = userName;  
    this.avatar = avatar;  
  }  
}  
  
const currentUser = new User({  
  userName: 'Foo',  
  avatar: 'foo.png'  
});  
  
User.prototype = {};  
console.log(  
  currentUser instanceof User, // <-- false -- Oops!// But it clearly has the correct shape:  
  // { avatar: "foo.png", userName: "Foo" }  
  currentUser  
);
```

Chrome bu sorunu Constructor.prototype özelliğini **configurable: false** yaparak çözmüştür. Buna karşın Babel bu davranışını benzer şekilde yansıtamamaktadır.

### I don't recommend reassigning . (Eric Elliott)

Yukarıdaki cümleden de anlaşılacağı üzere Constructor.prototype tekrar atama yapılmaması önerilir.

Daha yaygın bir sorun, JavaScript'in birden çok execution context'e sahip olmasıdır - aynı kodun farklı fiziksel bellek konumlarına erişeceği bellek sanal alanlarında bu problem ortaya çıkar. Örneğin Parent Frame(Window) ve bunun içерdiği iFrame **Constructor.prototype** bellekte farklı alanlara bakması ve bunların karşılaşmalarında `==` kullanılan kontrollerin hata vermesi.

Burada **instanceof** kontrolünün structural(yapısal) olmayıp nominal bir type check olmasından kaynaklanıyor. Diyelim ki class bileşenler geliştirerek başladık sonrasında aynı interface ile bunları factory çevirdik buradaki new yapısı oluşturulanlar instanceof ile çalışmadığı için kod aynı arayüzde olsa bile kırılganlık oluşturuyor.

Örneğin ekip, bir müzik oynatıcısı geliştirmenizi istediler. Sonrasında burada videoda görüntülemek istediler, daha sonra farklı video formatları için destekler eklemenizi istediler, aslında baktığınızda player interface için arayüz hiç değişmedi → Play, Stop, Forward, .... fakat aynı yapıya sahip olmalarına karşın videoInterface instanceof AudioInterface sonucu false olacaktır.

Burada 2 yapının aynı olduğunu nasıl ifade edebiliriz. Bazı diller Sharable Interface ile bu problemi çözüyor ama JS için bir çözümü yok.

JavaScript'te **instanceof** ile başa çıkanın en iyi yolu, gerekli değilse temsilci prototip(delegate prototype) bağlantısını kesmek ve her çağrı için **instanceof'un** başarısız olmasına izin vermektedir. Bu şekilde yanlış bir güvenilirlik duygusu elde edemezsiniz. instanceof'u dinlemeyin, size asla yalan söylemez.

## 21.3 The .constructor Property

**constructor** özelliği, JavaScript'te nadiren kullanılan bir özelliktir, ancak çok faydalı olabilir ve onu nesne örneklerinize dahil etmek iyi bir fikirdir. Tür denetimi için kullanmaya çalışmazsanız (bu, instanceof'un güvensiz olmasına aynı nedenlerle güvenli değildir) çoğunlukla zararsızdır.

**Teoride**, **.constructor**, ilettiğiniz nesnenin yeni bir örneğini döndürebilen genel işlevler yapmak için yararlı olabilir.

**Pratikte**, JavaScript'te şeylerin yeni örneklerini yaratmanın birçok farklı yolu vardır - yapıcıya bir referans vermek, onunla yeni bir nesnenin nasıl başlatılacağını bilmekle aynı şey değildir - boş bir nesne oluşturmak gibi görünüşte önemsiz amaçlar için bile.

Örneğin **Array için** aşağıdaki kodun çalıştığını görebilirsiniz.

```
// Return an empty instance of any object type?
const empty = ({ constructor } = {}) => constructor ?
  new constructor() :
  undefined
;

const foo = [10];
console.log(
  empty(foo) // []
);
```

Promise için

```
// Return an empty instance of any type?
const empty = ({ constructor } = {}) => constructor ?
  new constructor() :
  undefined;

const foo = Promise.resolve(10);
console.log(
  empty(foo) // [TypeError: Promise resolver undefined is
              // not a function]
);
```

Sınıflarda en önemli kırılganlığa neden olacak anahtar sözcük **new** dir. Bunu herhangi bir factory ile birlikte kullanmak mümkün olmadığından kodun kırılgan olmasına neden olacaktır.

Bu işi yapmak için ihtiyacımız olan şey, yeni gerektirmeyen standart bir fabrika işlevi kullanarak bir değeri yeni bir örneğe geçirmek için standart bir yola sahip olmaktadır. Bunun için **.of()** ile herhangi bir factory veya constructor üzerinde statik bir yöntem ile bunu gerçekleştirebilirsiniz.

Array için **.of** kullanımı

```
// Return an empty instance of any type?
const empty = ({ constructor } = {}) => constructor.of ?
  constructor.of() :
  undefined;const foo = [23];
console.log(
```

```
empty(foo) // []
);
```

## Promise için `.of` kullanımı

```
// Return an empty instance of any type?
const empty = ({ constructor } = {}) => constructor.of ?
  constructor.of() :
  undefined; const foo = Promise.resolve(10);
console.log(
  empty(foo) // undefined
);
```

Benzer şekilde **strings, numbers, objects, maps, weak maps, or sets** veri yapıları içinde `.of` yoktur.

`.of()` yöntemi için diğer standart JavaScript veri türlerinde de kullanılabilirse, `.constructor` özelliği sonunda dilin çok daha kullanışlı bir özelliği haline gelebilir.

Çeşitli işlevler, monadlar ve diğer cebirsel veri türleri üzerinde hareket edebilen zengin bir yardımcı utility kütüphanesi oluşturmak için kullanabiliriz.

Bir factory modülüne `.constructor` and `.of()` yapısını eklemek oldukça kolaydır.

```
const createUser = ({
  userName = 'Anonymous',
  avatar = 'anon.png'
} = {}) => ({
  userName,
  avatar,
  constructor: createUser
});

createUser.of = createUser; // testing .of and .constructor:
const empty = ({ constructor } = {}) => constructor.of ?
  constructor.of() :
  undefined
;

const foo = createUser({ userName: 'Empty', avatar: 'me.png' });
console.log(
  empty(foo), // { avatar: "anon.png", userName: "Anonymous" }
  foo.constructor === createUser.of, // true
  createUser.of === createUser // true
);
```

`.constructor` enumerable olmaması için delegate prototype ekleyerek bu işlevi gerçekleştirilebilirsiniz.

```
const createUser = ({  
  userName = 'Anonymous',  
  avatar = 'anon.png'  
} = {}) => ({  
  __proto__: {  
    constructor: createUser  
  },  
  userName,  
  avatar  
});
```

## 21.4 Sınıftan → Factory Büyük Değişim

Factory (Objenin Üretim Fonksiyonları) esnekliği arttırmır;

- Nesnenin veya bir yapının oluşturulması ile bunu çağrıran yerin ayırtılması.
- Nesne Havuzundan rastgele nesneleri döndürebilmenizi ve bunların belli kullanılmama durumlarında otomatik silinmelerini sağlayabilirsiniz.
- Çağırılan yere bir tip garantisini sağlamaya çalışmayı bu sayede abstract factory geçiş sonrasında instanceof ile type check yapmak zorunda kalmazsınız.
- Fabrikalar, tip garantileri sağlamamış gibi davranışları zaman , abstract factory uygulama içerisinde dinamik olarak değiştirebilirler. örneğin, farklı medya türleri için `.play()` yöntemini değiştiren bir medya oynatıcısını verebiliriz.
- Factory (Üretim Fonksiyonları) içerisinde composition (kapsayarak) nesne oluşturmak çok daha kolaydır.

Bu hedeflerin çoğu sınıfları kullanarak ulaşmak mümkün olsa da, bunu factory fonksiyonları ile yapmak daha kolaydır. Daha az potansiyel hata, daha az karmaşıklık ve kod ile gerçekleştirilebilir.

Yukarıdaki nedenlerden dolayı genellikle **class → factory** refactoring işlemi gerçekleştiriliyor. Tüm OO(Object Oriented) dillerde bu tip refactor sorunsuz bir şekilde gerçekleşmesi için bu kitap okunabilir. ["Refactoring: Improving the Design of Existing Code"](#) (by Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts.)

Bu dönüşümde new, class yapıları ile birlikte this binding problem çıkacaktır.

**Örneğin** class new olmadan kullanamıyoruz.

```
class Foo {};// TypeError: Class constructor Foo cannot be invoked without 'new'  
const Bar = Foo();
```

Arrow fonksiyonda da **new kullanmamak** gerekiyor.

```
const foo = () => ({})// TypeError: foo is not a constructor  
const bar = new foo();
```

Bu 2 birbirinden farklı yapı yani class ve arrow function da en temel farklardan birisi arrow fonksiyonunu this binding yapmaması. Bu konuda [React Action Handler neden bind etmek gerekiyor](#) yazısını okuyabilirsiniz. Burada fonksiyonların kendilerine this olarak bulundukları bir üst scope nesnesini bind etmesinden kaynaklı problemleri çözmek için ya ayrıca bind ile mevcut yapıları ezmek veya arrow fonksiyon kullanarak default bind önlemeye çalışmak gereklidir.

## 21.5 Sınıf ile Factory Fonksiyonun Kıyaslaması

Sınıf ile üretimi incelediğinizde **class**, içerisinde **constructor ,this ve new** anahtar sözcüklerini görürsünüz.

```
class User {  
  constructor ({userName, avatar}) {  
    this.userName = userName;  
    this.avatar = avatar;  
  }  
}  
const currentUser =new User({  
  userName: 'Foo',  
  avatar: 'foo.png'  
});
```

Bunun benzeri bir yapıyı factory function ile yapmak oldukça basit ve daha temiz bir syntax ile yapmak mümkün.

```
const createUser = ({ userName, avatar })=> ({  
  userName,  
  avatar  
});  
  
const currentUser =createUser({  
  userName: 'Foo',  
  avatar: 'foo.png'  
});
```

## 21.6 Performance and Memory

Delegate Prototypes ait kullanım senaryoları oldukça nadirdir. Sınıflarda 2 türde performans optimizasyonu bulunur;

- Shared memory for properties stored on the delegate prototype
- property lookup optimizations

Bu tip optimizasyon ihtiyaçları RxJS veya ThreeJS gibi binlerce nesne ve özelliğine 16ms render döngülerinde erişme ihtiyacı olduğunda sınıflar kullanılabilir ve bunlar closure scope göre çok daha hızlıdır.

Uygulamalar bağlamında, erken optimizasyondan kaçınmalı ve uğraşımızı yalnızca büyük etki yaratacakları yere odaklamalıyız.

Çoğu uygulama için bu, networking, animation, asset caching strategy vb.. alanlardadır.

Bir performans sorunu fark etmedikçe, uygulama kodunuzun profilini çıkarmadıkça ve gerçek bir darboğaz belirlemedikçe performans için mikro optimizasyon yapmayın.

Bunun yerine, bakım ve esneklik için kodu optimize etmelisiniz.

## 21.7 Tip Kontrolü (Type Checking)

JavaScript de Sınıflar(Class) dinamiktir. `instanceof` kontrolleri, execution context üzerinde etkisiz olacaktır. TypeChecking sınıflar üzerinden başlamak iyi ve güvenilir bir yöntem değildir. Bu durum hem hatalara hem de uygulamanızı gereksiz yere kırılgan katı bir yapıya dönüştürür.

### `extends` Kullanarak Sınıf Kalıtımı Gerçekleştirmek

Sınıf kalıtımlarının neden olduğu problemlerden daha önce bahsetmiştik. Özette;

- **The tight coupling problem:** Ata sınıfına çok sıkı bağımlılık
- **The fragile base class problem:** Ata sınıfındaki değişkenlik problemi
- **Inflexible hierarchy problem:** Hiyerarşinin esnekliği yok etmesi
- **The duplication by necessity problem:** Aynı yetenek ve özelliğin farklı ata sınıflarda yer alma ihtiyacı davranışın yer aldığı kodun tekrar edilmesine neden oluyor.
- **The Gorilla/banana problem :** Siz uygulamanızda ufak bir objeyi kullanmak istediğinizde aslında arka planda bunun bağımlı olduğu çok fazla yapının olması.

Çünkü sorunu hiyerarşisinden dolayı ihtiyacınız olmayan bir çok yapıyı zorunlu tutmaktır.

Bundan dolayı extends ile sınıf kalıtımları yapmak yerine Object Composition bu yeteneklerin sağlanması siz geliştiricilere birçok esneklikler sunacaktır.

## 21.8 Sınıfları Dikkatli ve Doğru Kullanmak.

Bütün bu uyarı yapılarına karşı sınıfları açık kullanım yönergeleri ile daha güvenli şekilde kullanmak mümkün.

- **instanceof** kullanımından kaçınmak gerekiyor. JS dinamik ve çoklu execution context çalışabilme yeteneğine sahip. instanceof tüm bu durumlarda hata ve problemlerin oluşmasına neden oluyor.
- **extends** kalıtım yapısından kaçının. Çoklu kalıtım hiyerarşileri oluşturmayın. Bunun yerine sınıf kalıtımlarını object composition üzerinden kurgulayın.
- Dışarıdan **new** ile herkesin sınıf oluşturabilmesini engelleyin. Bunu oluşturacağınız factory üzerinden yapılmasını sağlayarak yeni nesne üretimini kontrol altına alın.

Sınıfları Kullanmanın Problem Olmadığı Yerler;

- **UI Framework kullandığınız yerlerde**, örneğin React, Angular gibi yapılar önceden Component sınıflarından türeyen veya belli framework sınıflarından türeyen nesneler oluşturmak. Burada bu bileşenleri kullanmak için **new** kullanmaya gerek yok ve türeyen sınıflarınızda belli state ve fonksiyonları kullanmayı zorunlu tutması açısından oldukça başarılı. **Not:** React son dönemde Hook ve functional programlamaya geçmiş durumda.
- **Asla kendi sınıflarınızdan veya bileşenlerinizden türetme yapmayın**, yukarıda bahsettiğimiz durumlar haricinde diğer bileşenlerdeki yapıyı kullanmak için → object composition, function composition, high order functions, higher order components ve modülleri kullanarak reusability sağlamak gerekiyor.
- **Performansı optimize etmelisiniz**, işlevlerinizi bir factory üzerinden gerçekleştirip kullananlara *new* anahtar sözcüğünü ve *extends* tuzağına düşmeden geliştirme yapmanızı sağlamalısınız.

## 22. Fonksiyonlar ile Birleştirilebilir (Composable) Veri Türleri

Composable yani matematiksel olarak birleştirilebilir, üzerinde fonksiyon çalıştırılıp, diğer fonksiyon ile tekrar tekrar kullanabilir anlamında kullanılıyor. Composition işlemleri uygulamanızın esnek ve üzerinde rahat işlem yapabilir hale getirecektir. Aşağıdaki yazı bu tarz veri türlerini nasıl oluşturduğumuz üzerinedir.

Bu yazı Eric Elliott Composable Software kitabındaki Composable Datatypes with Functions bölümünü anlatmaya çalışacağız.

JavaScript'de birleştirerek, kapsayarak yazılım oluşturanın en kolay yolu, fonksiyonlardır ve fonksiyonlarında bir obje olması ve method ve özellik eklenmesi aşağıdaki duruma benzer yapılar kurmanızı sağlar.

Aşağıdaki örnekte **t bir fonksiyon, fn bir fonksiyon, to string başka bir fonksiyon.**

```
const t = value => {
  const fn = () => value; 2
  fn.toString = () => `t(${ value })`; 3
  return fn; 4
};

const someValue = t(2); 1

console.log(
  someValue.toString() // "t(2)"
);
```

- $t(2)$  ile bize bir fonksiyon dönen bir fonksiyonu çağrırlıyorsuz.
- bize fonksiyon dönen fonksiyon aynı zamanda fonksiyona farklı fonksiyonlar ekleyerek onları kullanabilme imkanı sunuyor (Örn `toString` gibi)

**İlk olarak, bazı kurallar oluşturalım;**

Aşağıdaki örnekte  $t$  fonksiyonu  $x+0$  matematiksel yaklaşımı  $x+1$  de de yapıyor olabilmek.

- $t(x)(t(0)) === t(x)$

yukarıda önce  $t(0)$  işletilir çıkan sonuç  $t(x)$  dönen fonksiyona verilir.

- $t(x)(t(1)) === t(x + 1)$

aynısı yukarıdaki içinde geçerli önce  $t(1)$  işletilir çıkan sonuç  $t(x)$  dönen fonksiyona verilir.

Bu durumu JS'de **toString** görebiliriz.

- `t(x)(t(0)).toString() === t(x).toString()`
- `t(x)(t(1)).toString() === t(x + 1).toString()`

Bu fonksiyonu  $t$  için işlettiğimizde ilk durum için  $t$  farklı olduğunu, 2nci durum için toplama işleminin gerçekleşmediğini görebilirsiniz.

```
1 const t = (value) => {
2   const fn = () => value;
3   fn.toString = () => `t(${value})`;
4   return fn;
5 };
6   • t(x)(t(0)).toString() === t(x).toString()
7 console.log(t(4)(t(0)).toString());
8 console.log(t(4).toString());
9
10 console.log('\n');
11   • t(x)(t(1)).toString() === t(x + 1).toString()
12 console.log(t(4)(t(1)).toString());
13 console.log(t(4 + 1).toString());
14
```

Mevcut  $t$  fonksiyonu add özelliği yok

Peki bu fonksiyonu nasıl çalışır hale getireceğiz.

1.  $t$  fonksiyonunun döndüğü `fn` fonksiyonun içerisinde `add` fonksiyonu eklenir ve bu `t(value + n)` şeklinde kendisine geçilen  $n$  değerini mevcut  $value$  ile toplar.
2. `valueOf` fonksiyonu `t()` parametre olarak geçen `n` değerini almakta ve toplama yapmak için kullanılır.
3. Oluşturulan `add` metodunu `fn` eklemek için `Object.assign()` kullanır.

```

1 const t = (value) => {
2   const add = (n) => t(value + n);
3   return Object.assign(add, {
4     toString: () => `t(${value})`,
5     valueOf: () => value,
6   });
7 };
8   • t(x)(t(0)).toString() === t(x).toString()
9 console.log(t(4)(t(0)).toString());
10 console.log(t(4).toString());
11
12 console.log('\n');
13   • t(x)(t(1)).toString() === t(x + 1).toString()
14 console.log(t(4)(t(1)).toString());
15 console.log(t(4 + 1).toString());
16

```

▶ Run

```

1 't(4)'
2 't(4)'
3 '
4 '
5 't(5)'
6 't(5)'

```

mevcut t fonksiyonuna add özelliğini ekledik.

fonksiyonu bu şekilde işletebilen matematiksel hale getirdiğinizde **pipe** gibi yapılar ile tüm bu fonksiyonları birbirini kapsayacak ve birleştirilecek halde (composition) yapıları kurabilmemizi imkan tanır.

```
// Compose functions from top to bottom:
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x); // Sugar to kick off the pipeline with an initial value:
const sumT = (...fns) => pipe(...fns)(t(0)); sumT(
  t(2),
  t(4),
  t(-1)
).valueOf(); // 5
```

## 22.1 Bunu Herhangi Bir Veri Türüyle Uygulayabilirsiniz.

Mantıklı bir composition işlemi olduğu sürece verilerinizin hangi yapıda olduğu önemli değildir.

- List ve Stringler için birleştirme
- DSP için sinyal toplama

Elbette aynı veriler için birçok farklı işlem anlamlı olabilir. Soru şu ki, kompozisyon kavramını en iyi hangi işlem temsil ediyor? Başka bir deyişle, bu şekilde ifade edilen en çok hangi işlem fayda sağlar?

```
const result = compose(
  value1,
```

```
    value2,  
    value3  
);
```

## Composable (Birleştirilebilir) Para Birimi

JS Number değerinin tutulma şeklinden kaynaklı  $0.1 + 0.2$  doları centi topladığımızda istediğimiz rakama ulaşamayız.

```
1 console.log(0.1 + 0.2);  
2
```

▶ Run 1 0.3000000000000004

Bunu birleştirilebilir bir para birimine dönüştürmek için fonksiyon composition faydalananabiliriz.

```
import { $ } from 'moneysafe';  
$(.1) + $(.2) === $(.3).cents; // true
```

Bu kütüphaneyi nasıl yazdığını konusunda videosuna aşağıdaki referanslarda bulunan blog yazısından ulaşabilirsiniz.

Aynı zamanda kaynak kodlara <https://github.com/ericelliott/moneysafe> bu linkten ulaşabilirsiniz.

## 23. Lenses

Fonksiyonel Programlamada Object, Array vb veri yapıları üzerindeki belli alanda işlem yapmanızı sağlayan set, get(view), ve over fonksiyonları ile bu işlemleri soyutlayarak ortak davranışlara dönüştüren yaklaşımındır.

Normalde basit bir obje ile çalıştığımızı düşünelim. Aşağıdaki örnekte olduğu gibi.

```
const obj= {a:1, b:2, c:3}
```

Bu objenin b elemanını okumak veya bunun üzerine bir şeyler yazıp değiştirmek istediğimizde get ve set fonksiyonlarına ihtiyaç duyuyoruz.

```
//get işlemini  
const bVal=obj.b
```

```
//set işlemini  
const obj[b]=5
```

peki bunu property bazında nasıl fonksiyon haline getirebiliriz.

```
const getB=(obj)=>obj[b];  
const setB=(obj, val)=>obj[b]=val;
```

bunu daha genel nasıl yazabiliriz.

```
const view=(obj, prop)=>obj[prop];  
const set=(obj, prop, val)= obj[prop]=val
```

Tabii ben burada bu işlemi oldukça basit anlamda soyut hale getirdim. Ramda kütüphanesi içerisinde **lens konusu** daha detaylı anlatılıyor. Ve konuyu bunun üzerinden anlatmanın daha doğru olacağını düşünüyorum.

Aşağıda gördüğünüz gibi **view**, **set** ve **over** fonksiyonları bulunuyor. Bunlar lens üzerinde işlem yaptığı fonksiyonlar.

```
const xLens = R.lens(R.prop('x'), R.assoc('x'));  
  
R.view(xLens, {x: 1, y: 2}); //=> 1  
R.set(xLens, 4, {x: 1, y: 2}); //=> {x: 4, y: 2}  
R.over(xLens, R.negate, {x: 1, y: 2}); //=> {x: -1, y: 2}
```

- **view** → okuma
- **set** → yazma
- **over** → önce okuyup, üzerinde bir fonksiyon çalıştırıp sonra üzerine yazma işlemi yapıyor.

Tabi birde **R.lens** fonksiyonu bulunuyor. Bu da objenin hangi alanı üzerinde çalışacağını belirtiyor.

Soyutlama işleminin fonksiyonlar ile ne kadar başarılı bir şekilde yapıldığını görebilirsiniz. x prop üzerinde view, set ve over işlemleri çalıştırılıyor.

## 23.1 Lensleri Farklı Şekillerde Tanımlama

## LensIndex

Örneğin ben indeks üzerinden bir tanımlama yapmak istiyorum. Aşağıdaki örnek array birinci indeksi üzerinde çalışmasını belirtiyor.

```
const headLens = R.lensIndex(0);

R.view(headLens, ['a', 'b', 'c']);           //=> 'a'
R.set(headLens, 'x', ['a', 'b', 'c']);       //=> ['x', 'b', 'c']
R.over(headLens, R.toUpperCase, ['a', 'b', 'c']); //=> ['A', 'b', 'c']
```

## LensProp

Aşağıdaki örnek objenin tanımlanmış bir property üzerinde çalışmayı belirtiyor. Aşağıdaki örnek x property üzerinde çalışmaya odaklanıyor.

```
const xLens = R.lensProp('x');

R.view(xLens, {x: 1, y: 2});           //=> 1
R.set(xLens, 4, {x: 1, y: 2});       //=> {x: 4, y: 2}
R.over(xLens, R.negate, {x: 1, y: 2}); //=> {x: -1, y: 2}
```

## LensPath

Daha kompleks obje türlerinde örneğin x property array ve bu arrayin elemanları bulunuyor. Bizde x arrayindeki 0 elemanın y özelliği ile ilgileniyoruz. Bu durumlar için lensPath kullanmalıyız.

```
const xHeadYLens = R.lensPath(['x', 0, 'y']);

R.view(xHeadYLens, {x: [{y: 2, z: 3}, {y: 4, z: 5}]});
//=> 2
R.set(xHeadYLens, 1, {x: [{y: 2, z: 3}, {y: 4, z: 5}]});
//=> {x: [{y: 1, z: 3}, {y: 4, z: 5}]}
R.over(xHeadYLens, R.negate, {x: [{y: 2, z: 3}, {y: 4, z: 5}]});
//=> {x: [{y: -2, z: 3}, {y: 4, z: 5}]} 
```

## 23.2 Lensler Functor özelliği ile Map Yapıları Üzerinde Çalışır.

Daha önceki yazınlarda Functor bahsetmiştim. Daha önceden okumamış olanlar [Functors ve Categories](#) blog yazısını okuyabilirler.

Aşağıdaki örnekte elimizde bir array bulunuyor. Biz amount değerini başka bir currency değeri ile değiştirmek istiyoruz.

```
const quotes = [
  {symbol: 'AAPL', amount: 150, name: 'Apple'},
  {symbol: 'GOOG', amount: 200, name: 'Google'},
  {symbol: 'MSFT', amount: 250, name: 'Microsoft'}
];
const amtLens = R.lensProp('amount');
const otherCurrency = R.over(amtLens, a => a * 1.7);
const result = R.map(otherCurrency, quotes);
console.log('result:', result);
```

<https://youtu.be/AYyMGdxgCTY?t=1294>

Bu durumda R.over lens işlemini bir değişkene atayıp map fonksiyonu içerisinde tüm objeler bunun uygulanmasını sağlayabiliriz.

## 23.3 Özet

Lensler de görüleceği üzere amaç set, get ve over ile objenin şeklini veya yapısında bir takım değişiklikleri daha kontrollü ve soyutlayarak gerçekleştirmek

## 24. Transducers

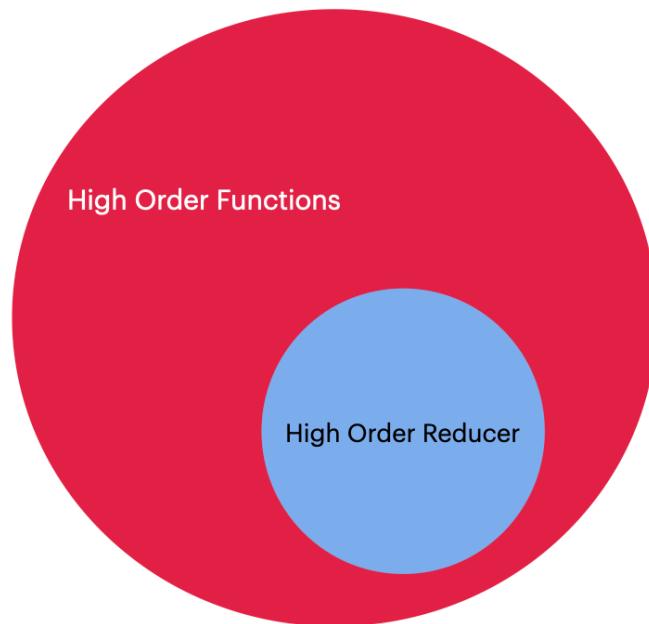
Yazılımda bir girdiyi dönüştürerek istenen farklı bir çıktılar oluşturan fiziksel parçalara Transducers (Transdüber) denir. Elektrik devrelerinde sinyal işleme, veya Yazılımda Data Flow (Veri Akışlarındaki) dönüştürme işlemlerinin soyutlanmasıdır.

Transducers daha iyi anlaşılmasına için aşağıdaki alıntıyla başlamak istiyorum.  
Transdüber aslında bizim hayatı sıkılıkla kullandığımız fiziksel cihazlar.

Mikrofonlarda sesi algılamak için ses sensörleri, araçlarda egzoz gazlarını algılamak için oksijen sensörleri, ortam sıcaklığını algılamak için sıcaklık sensörleri, fabrikalarda gaz veya sıvı basıncını algılamak için basınç sensörleri veya otomatik kapıarda hareketle değişen ışık miktarını algılamak için kullanılan fotoelektrik sensörler (MühendisBeyinler.net)

Yazılım anlamında aslında genişleyebilir (composable), HoR(High Order Reducer) başka Reducer içерine alıp, geriye Reducer dönen yapılardır.

Bizim yukarıda bahsettiğimiz ise biraz daha özelleşmiş bir şey → High Order Reducer, yani High Order Function bir alt kümesi olarak düşünülebiliriz.



Peki **Reducer dediğimiz şey** nedir ? Birden fazla değer alıp bunu birleştirip tek bir sonuç olarak dönen yapılardır.

```
// Sums: (1, 2) = 3
const add = (a, b) => a + b;
// Products: (2, 4) = 8
const multiply = (a, b) => a * b;
// String concatenation: ('abc', '123') = 'abc123'
const concatString = (a, b) => a + b;
// Array concatenation: ([1,2], [3,4]) = [1, 2, 3, 4]
const concatArray = (a, b) => [...a, ...b];
```

High Order Reducer yani Transducers denilen şeyler enumerable veri kaynakları üzerinde → arrays, trees, streams, graphs, vb üzerinde Reducer genişleyebilir şekilde devam ettirebilir yapılardır.

```
T(Reducer) → Reducer.  
G(Reducer) -> T(Reducer) -> Reducer
```

Gerçek uygulamalardan bir örnek verdiğimizde durum daha iyi anlaşılacaktır. Aşağıdaki örnekteki **filter**, **map** fonksiyonları Transducerslar dır.

```
const cities=[  
  {name: 'Ankara', value: 2}  
  {name: 'Izmir', value: 1}  
  {name: 'Istanbul', value: 4}  
]  
  
const filterFn = ({value})= value>1;  
const viewFn = ({name}) => name;  
  
const result=cities.filter(filterFn).map(viewFn);  
console.log(result); // [Ankara, Istanbul];
```

Bir önceki konuda Lens anlattığım gibi bizim aldığımız girdi bir stream, array vb.

- filtrelememiz

```
[10] → (filter) → [5]
```

- array içerisindeki objenin yapısını azaltmamız dışarıya çıkarmamız gerekebilir.

```
[ {name: 'Ankara', value: 2}, {name: 'Istanbul', value: 4}] → [ 'Ankara,Istanbul']
```

- veya bu objenin içerisindeki değeri daha fazla arttırmamız gerekebilir

```
[ {name: 'Ankara', value: 2}, {name: 'Istanbul', value: 4}] →  
[ {name: 'Ankara', value: 2, ab:'Ankara2'},  
  {name: 'Istanbul', value: 4, ab:'Istanbul4'}]
```

Tüm bu çabalar enumerable veri yapısında çalışmaya döngüleri, filtreleri, dönüştürmeleri olabildiğince soyutlamaktır.

Bu soyutlama işlemini yapan `.filter()`, `.map()`, `.reduce()`, `.concat()` fonksiyonlar Transducers diyoruz.

Ramda daki [EricElliottJS blog](#) yazısında yer alan örneği inceleyelim. Aşağıdaki örnek Ramda kütüphanesini kullanıyor.

```
import {
  compose,
  filter,
  map,
  into
} from 'ramda';

const isEven = n => n % 2 === 0;
const double = n => n * 2;

const doubleEvens = compose(
  filter(isEven),
  map(double)
);

const arr = [1, 2, 3, 4, 5, 6];// into = (structure, transducer, data) => result
// into transduces the data using the supplied
// transducer into the structure passed as the
// first argument.
const result = into([], doubleEvens, arr);console.log(result); // [4, 8, 12]
```

However, it does offer a different style of coding, a style that's taken for granted in purely functional programming languages: Ramda makes it simple for you to build complex logic through functional composition. Note that any library with a `compose` function will allow you do functional composition; the real point here is: "makes it simple". (Scott Sauyet)

Yukarıda gördüğünüz gibi herşeyin bir fonksiyon olduğunu görebilirsiniz. Fonksiyonel programlama'da veriler üzerinde işleyen ve immutable çıktı üreten fonksiyonlar bulunur.

- **isEven** : Kendisine verilen değer çift minin cevabını veriyor (True/False)
- **double**: Kendisine verilen değeri 2 ile çarpıyor  $n * 2 \Rightarrow 2n$
- **filter(isEven)**: Enumerable bir veri yapısındaki değerlerin çiftin katları olacak şekilde filtrelemesi

- **map(double)** : Enumerable bir veri yapısında değerleri 2 ile çarpar ve aynı enumerable yapıyı döner.
- **doubleEvens : [array] → filter(isEven) → map(double) → [array]**

Burada Transducer arka arkaya çağrılarak bir veri flow oluşturduğunu görebilirsiniz. Bunu müzik sinyali, video stream, veri akışı üzerinde uygulayabilirsiniz. ,

```
[ Source ] -> [ Mic ] -> [ Filter ] -> [ Mixer ] -> [ Recording ]
[ Enumerator ]->[ Transducer ]->[ Transducer ]->[ Accumulator ]
```

## 25 JS'nin Güçlü ve Doğru Kullanılması

"The Elements of Style" by William Strunk Jr 1920 yılında İngilizce'nin yazılarında, hikayelerde doğru kullanılması için bir rehber niteliğinde stilleri belirtir. Aynısını JavaScript için de uygulayabiliriz.

Aşağıda belirttikleri kesin değişmez yasalar veya kurallar değil, zaman içerisinde doğruluğu ve başarısı kanıtlanmış stillerdir.

İngilizce dili için söylemiş her prensip/stil aynısını kaynak kod içinde kullanabiliriz.

### 25.1 Make the function the unit of composition. One job for each function.

- Paragrafı kompozisyon birimi yapın: Her konuya bir paragraf ile anlatın.
- Fonksiyonu composition unit yapın. Her fonksiyon bir iş yapsın.

The essence of software development is composition. We build software by composing modules, functions, and data structures together.

*Understanding how to write and compose functions is a fundamental skill for software developers.*

Yukarıdaki söylemler yazılım geliştirmede composition(kapsama) çok çok önemli bir konu,

Modüller fonksiyonları ve veri yapılarını içeriyor, fonksiyonlar da diğer fonksiyonları kapsıyor. Ve esas olay uygulama state üzerinde çalışan fonksiyonların veriler

üzerinde uygulanması ile gerçekleştiriliyor.

JavaScript 3 tip fonksiyon bulunuyor;

- **Communicating functions** : I/O girdi çıktıları işleyen fonksiyonlar.
- **Procedural functions**: Birçok komut'un belli bir sıra ve grup içerisinde işletilmesi
- **Mapping functions**: Verilen bir girdiyi başka bir çıktıya dönüştüren fonksiyonlar

Genelde yazılımlarda I/O işlemini → procedural fonksiyon grubu → sonrasında mapping adımları izler. Burada I/O ve mapping işlemlerini birbirine karışmayan ayrı fonksiyonlar içerisinde gerçekleştirebilmek olabildiğince önemlidir.

Yukarıdaki cümleden anlaşılacağı gibi composition yapmayı bilmek ve composition elemanlarını yani I/O ve Mapping fonksiyonlarını ayırip, üzerinde işlem yaptığınız state/veri yapılarının Side Effecti olmasını engelleyecek işlemler yaptırmak oldukça önemlidir.

## 25.2. Omit needless code.

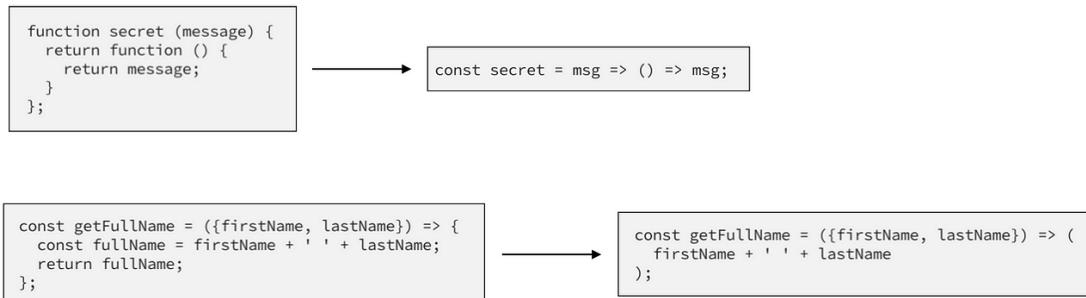
- Gereksiz Kelimeleri Silin
- Gereksiz Kodları Silin

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all sentences short, or avoid all detail and treat subjects only in outline, but that every word tell. [Needless words omitted.]~ William Strunk, Jr., The Elements of Style

Güçlü yazı özlündür. Bir cümle gereksiz kelimeleri, bir paragraf gereksiz cümleleri içermez , resmin gereksiz bir çizgi, makinenin gereksiz parçaları içermemesi , tüm bunlar ürünün kaliteli olmasını sağlar, yazılımda da ne kadar az ve öz o kadar az o kadar az bug ve kodun anlaşılır olması anlamına gelir. Burada az gereksiz olanların çıkarıldığı öz kısmının kaldığı kod parçasıdır.

### Return Kaldırılması

Örneğin aşağıdaki örneklerde return ortadan kaldırınak ve ara bir değişkende işlem sonuçlarını tutmamak, kodu daha anlaşılır ve kısa tutmanızı sağlar.



## Point Free Style

Cury fonksiyonlar ile fonksiyonları parametrik ve içerde veri saklayabilen hale getirebiliriz. Bu sayede `add2(1)` gibi verdığınız sayı ile `inc+1` fonksiyonu elde edebilirsiniz. Bunun gibi istediğiniz fonksiyonları oluşturabilirsiniz.

```
const add2 = a => b => a + b;  
// Now we can define a point-free inc()  
// that adds 1 to any number.  
const inc = add2(1);  
inc(3); // 4
```

## Function Composition

2 fonksiyonun ortak kullanması gereken bir değişken olduğunda bunları ara değerlerde tutmak yerine function composition ile bu değişkenden kurtulabilirsiniz.

```
const g = n => n + 1;  
const f = n => n * 2;
```

```
function calculate(n){  
  const result=g(n);  
  const result2=f(n);
```

```
    return result;  
}
```

Yukarıda bu işlemi result değişkenleri üzerinde tutarak gerçekleştirdik. Halbuki bir değişkene atamadan bu işlemi aşağıdaki şekilde gerçekleştirebiliriz.

```
function calculate(n){  
    return f(g(n))  
}
```

Bunun yerine **compose veya pipe** ile fonksiyonları arka sırada verebilirsiniz.

```
const compose2 = (f, g) => x => f(g(x));  
compose2(20) //42  
  
const pipe = (...fns) => x => fns.reduce((acc, fn) => fn(acc), x);  
pipe(g, f)(20); // 42
```

### 25.3. Use Active Voice

“The active voice is usually more direct and vigorous than the passive.” ~ William Strunk, Jr., The Elements of Style

Burada İngilizce **passive** kelimeler ve eylemler ile fonksiyonları anlatmak yerine aktif eylemler ile dili daha kuvvetli hale getirirsiniz.

```
myFunction.hasBeenCalled() → yerine → myFunction.wasCalled()  
User.create() --> createUser()  
Notifier.doNotification() --> notify()  
  
//Sonucu Boolean dönen fonksiyonlarda  
getActiveStatus(user) --> isActive(user)  
firstRun = false; --> isFirstRun = false;  
  
//Name Function yerine  
plusOne() --> inc()  
filesFromZip() --> unzip()  
matchingItemsFromArray(fn, array) --> filter(fn, array)
```

**EventHandlers** eylemin ne zaman yapıldığını belirttiği için yukarıdaki gibi Active Voice belirtmesinden öte zaman kavramını belirtmesi önemlidir.

```
element.click(handleClick) --> element.onClick(handleClick)
component.onDragStart(handleDragStart) --> component.startDrag(handleDragStart)
```

### LifeCycle metodunda

```
componentWillBeUpdated(doSomething) --> component.beforeUpdate(doSomething)
```

Functional Mixinde sıfat olan eylemler \*ing veya \*able alabilen fiiller tercih edilmelidir.

```
const duck = composeMixins(flying, quacking);
const box = composeMixins(iterable, mappable);
```

## 25.4. Avoid a Succession of Loose Statements

- Art arda gevşek cümlelerden kaçının.
- Art arda gevşek ifadelerden kaçının

Aslında dilde de benzer problemler olabiliyor. Art arda birbiri ile ilişkili olmayan cümleler gibi, prosedürel olarak arka arkaya birbiri ile ilişkilisi güçlü olmayan komutları sıraladığımızda bu monoton bir yapı oluşturur.

Bu tür diziler, her biri ustaca ve bazen beklenmedik bir şekilde farklı olan birçok paralel form tarafından sıkılıkla tekrarlanır.

Örneğin, bir kullanıcı arayüzü bileşeni, neredeyse tüm diğer kullanıcı arayüzü bileşenleri ile aynı temel ihtiyaçları paylaşır. Ve bu yapılar yaşam döngüleri şeklinde parçalanabilir ve ayrı fonksiyonlar ile yönetilebilir.

```
const drawUserProfile = ({ userId }) => {
  const userData = loadUserData(userId);
  const dataToDisplay = calculateDisplayData(userData);
  renderProfileData(dataToDisplay);
};
```

Yukarıdaki kod örneğin 3 farklı durumu içeriyor

- Loading Data
- Process Data
- Render Data

Bunların farklılaştırılması aslında arka planda bir çok avantajlar sağlar ve modern frontend mimarileri veya framework'leri bu yapıları gözeterek bir takım altyapılar sunar.

Bu sayede Rendering katmanını istediğiniz bir Renderer ile değiştirebilirsiniz (ReactNative, WebVR, ReactDOM/Server, VirtualDOM, TestRenderer vb..)

Aynı şekilde Loading katmanını istediğiniz bir fetching kütüphanesi ile Axios, Fetching, XMLHttpRequest WebAPI'ler veya GraphQL, ReactQuery vb.. şema veya caching için ekstra yapılar kullanılabilir.

ProcessData kısmında farklı business logicler uygulanabilir.

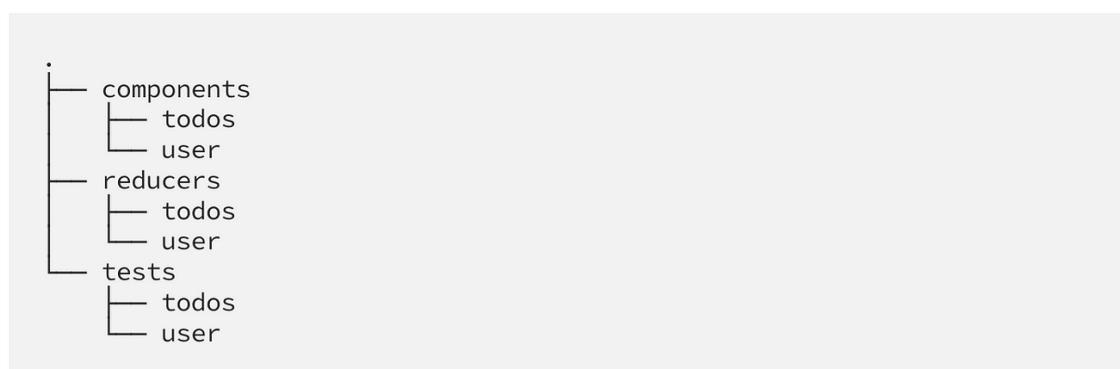
Bu şekilde farklı işlevlere sahip fonksiyonları ayırtılabilirsek hem test edilmesi kolay birimler oluşturabiliriz, hem de bunları farklı opsiyonlar ile istediğimiz zaman değiştirebiliriz.

## 25.5 Keep related code together.

- Birbiri ile ilişkili kelimeleri birbirine yakın tutmak.
- Birbiri ile ilişkili kodları birbirine yakın tutmak.

Bu benim mevcut projelerimde sık sık karşıma çıkan bir durum. Mevcut projelerde dosyaları type göre ayırmadan daha mantıklı olduğunu düşündük. Buna göre çoğu yerde aşağıdaki gibi hatta daha fazla gruplayarak (containers, components, utils, stores, vb...) ayırmalar gerçekleştirdik.

**Grouped by type:**

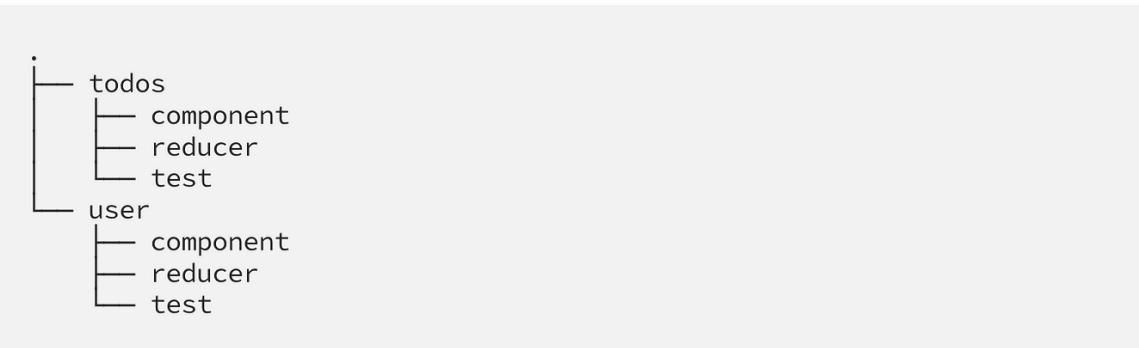


**Avantajı** benzer şekilde kodları belli klasörler altında toplayıp, bunların farklı implementasyonla değiştirmenin sadece klasör değişikliği ile yapılabiliyor olması.

**Dezavantajı** ise giderek bölünmüş bu type kendi içerisinde büyüyerek tek bir feature için bir birinden farklı klasörlerde bölünmüş yapılara ekstra kodlar eklemek.

Diğer bir yaklaşım ise Feature (yetenekler) bazında uygulama parçalarını bölümlemek.

**Grouped by feature:**



Ve o yeteneklerin altında type göre bölümleme yapmak. Avantajı , scroll ile farklı işlevsel klasörler içerisinde aradığını bulmak yerine daha odaklı çalışabilme imkanı. Dezavantajı, birbirine ortak yapıları tutmada karşılaşılacak zorluklar. Buna rağmen yeni yaklaşım Feature bazlı graplama yönünde.

## 25.6 Put statements and expressions in positive form.

“Make definite assertions. Avoid tame, colorless, hesitating, non-committal language. Use the word not as a means of denial or in antithesis, never as a means of evasion.”~ William Strunk, Jr.,  
The Elements of Style

Kodu daha positive komutlar ve koşullar üzerine inşaa etmek.

- **isNotFlying** yerine **isFlying**
- **notOnTime** yerine **late**

gibi daha positive değişken ve fonksiyon isimleri vermeye çalışalım...

### If Statement

Negative ile başlamak yerine hata yoksa ne yapacağımıza başta else durumunda da negative olarak ele alalım..

```
if (err) return reject(err);
// do something...
```



```
if (!err) {
  // ... do something
} else {
  return reject(err);
}
```

## Tenary

```
{  
  [Symbol.iterator]: (!iterator) ? defaultIterator : iterator  
}
```



```
{  
  [Symbol.iterator]: iterator ? iterator : defaultIterator  
}
```

vb... positive yaklaşımları önceliklendirin.

## Referanslar

- [Composing Software: The Book \(from Eric Elliott\)](#)
- [Imperative vs Declarative Programming \(Tyler McGinnis\)](#)
- [Founding Fathers and Mothers of Programming \(Siarhei Kulich\)](#)
- [Turing Machine\(EngMicroLectures\)](#)
- [How Turing Machine Works](#)
- [Turing Machine Simulator](#)
- [Equivalent Formalism for Turing Machine](#)
- [https://en.wikipedia.org/wiki/Timeline\\_of\\_programming\\_languages#1990s](https://en.wikipedia.org/wiki/Timeline_of_programming_languages#1990s)
- <https://purelyfunctional.tv/functional-programming-languages>
- <http://benalman.com/news/2012/09/partial-application-in-javascript/>
- <http://benalman.com/news/2012/09/partial-application-in-javascript/>
- [FP and Category Theory -1](#)
- [FP and Category Theory -2](#)
- [What's Functor](#)

- [Introduction to Functor and Monads](#)
- [Category Theory — Functors](#)
- [Aristoteles ve Yöntem -Tümevarım ve Tümdeğelim](#)
- [Tanım, Aksiyom, Teorem, İspat](#)
- [Abstract Data Types](#)
- [Monads in Functional Programming: a Practical Note](#)
- [The marvellously mysterious javascript maybe monad](#)
- [What's Monad](#)
- [Early History Of Smalltalk](#)