Java ile Nesne Merkezli ve Fonksiyonel Programlama

6. Bölüm
Fonksiyonel Programlama (Functional Programming)

Akın Kaldıroğlu

www.javaturk.org

Nisan 2018

Küçük Ama Önemli Bir Konu

- Bu dosya ve beraberindeki tüm, dosya, kod, vb. eğitim malzemelerinin tüm hakları **Selsoft Yazılım, Danışmanlık, Eğitim ve Tic. Ltd. Şti**.'ne aittir.
- Bu eğitim malzemelerini kişisel bilgilenme ve gelişiminiz amacıyla kullanabilirsiniz ve isteyenleri **http://www.selsoft.academy** adresine yönlendirip, bu malzemelerin en güncel hallerini almalarını sağlayabilirsiniz.
- Yukarıda bahsedilen amaç dışında, bu eğitim malzemelerinin, ticari olsun/olmasın herhangi bir şekilde, toplu bir eğitim faaliyetinde kullanılması, bu amaca yönelik olsun/olmasın basılması, dağıtılması, gerçek ya da sanal/Internet ortamlarında yayınlanması yasaktır. Böyle bir ihtiyaç halinde lütfen benimle, **akin.kaldiroglu@selsoft.academy** adresinden iletişime geçin.
- Bu ve benzeri eğitim malzemelerine katkıda bulunmak ya da düzeltme ve eleştirilerinizi bana iletmek isterseniz çok sevinirim.
- Bol Java'lı günler dilerim.

İçerik

- Bu bölümde şu konular ele alınacaktır:
  - Fonksiyonel programlamanın temel kavramları ve yapıları,
  - Groovy'nin closure yapısı ve Python'un fonksiyonları
  - Geri çağırma metotları (callback methods) ve isimsiz sınıflar (anonymous classes),
  - Lambda ifadeleri,
  - Hazır Fonksiyonlar ve
  - Metot Referansları

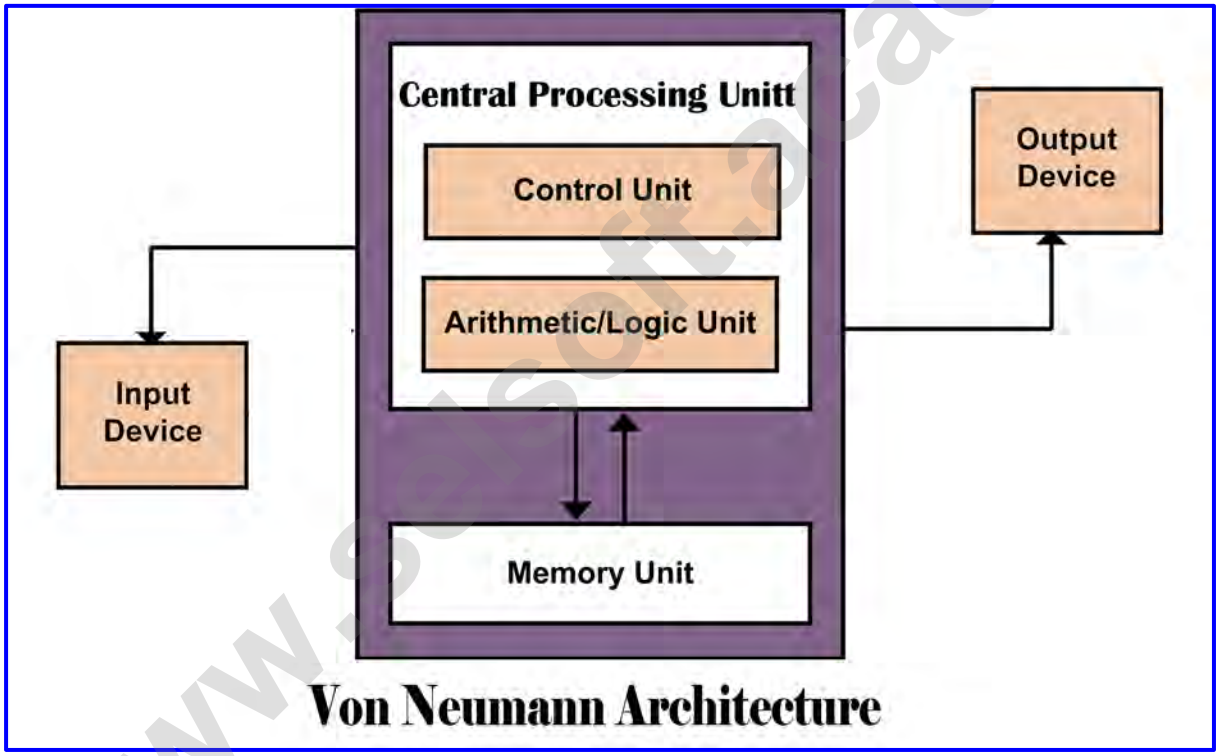
Fonksiyonel Programlamaya Giriş

Fonksiyonel Programlama: Tarih ve Temel Kavramlar

John Backus

- John Backus, Fortran'ın mucididir ve bundan dolayı da 1977'de ACM'in Turing ödülünü kazanmıştır.
- Daha sonra makale olarak da basılan (*Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*) 1978'de yaptığı ödül konuşmasında Backus, Fortran ve Algol gibi komutsal (imperative) dillerin büyümelerine rağmen güçlenmediklerini ifade edip onları"*fat and flappy*" olarak nitelendirdi.
- Ayrıca Backus bu duruma karşı, bir başka programlama pradigması ya da yaklaşımı teklif etti: Functional style programming.

Von Neumann Mimarisi

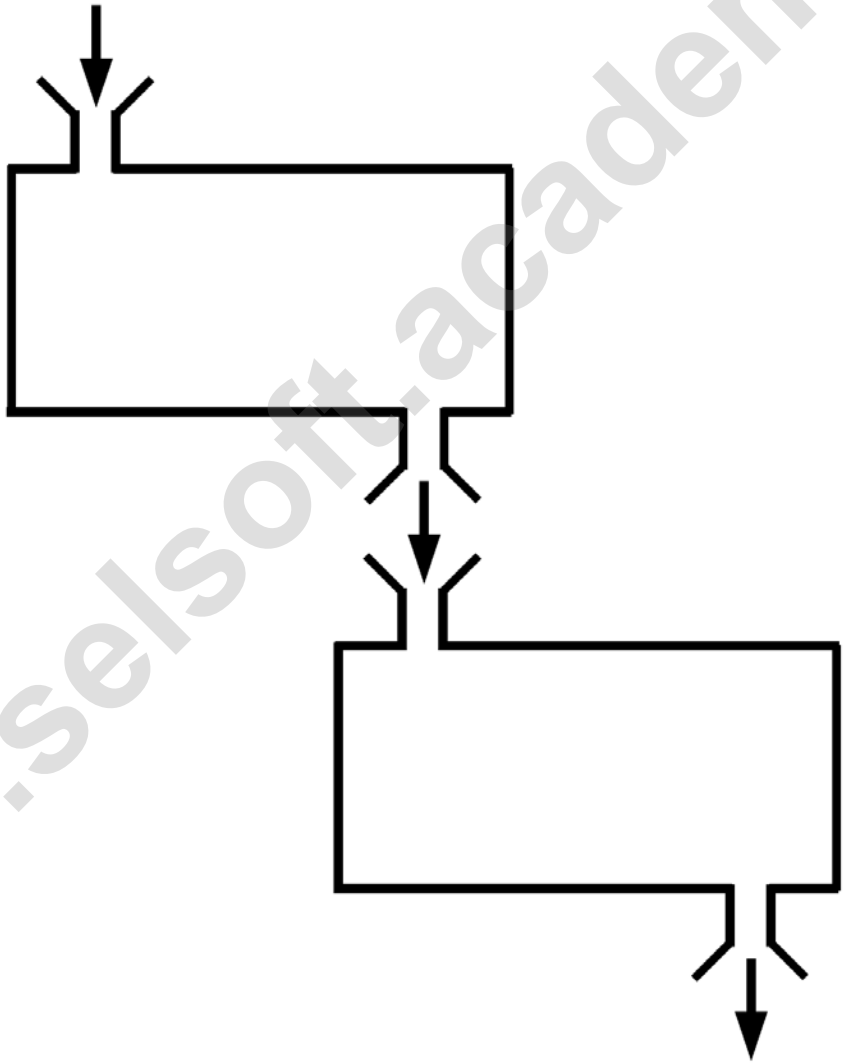


Imperative ve Fonksiyonel Programlama

- Backus, fonksiyonel dillerde programların daha rahat okunur, anlaşılır ve hatalardan uzak olacağını iddia etti.
  - Çünkü teklif ettiği yapıda ne ifadelerin ne de fonksiyonların, bulundukları bağlama bir etkileri olacaktı.
  - Ayrıca değişken dolayısıyla da durum olmayacaktı.

- Komutsal dillerde, değişkenlerle temsil edilen durum bilgisi (state) program boyunca sürekli değiştirilir.
  - Programlar büyüdükçe bu çok zor ve tehlikeli bir iş haline gelir.
- Backus ise sadece matematiksel fonksiyonlardan oluşan bir programlama stili teklif ediyordu.

Function



Lambda İfadeleri & Calculus

- 1941'de Alonzo Church, sadece parametreleri ve işleyişi ile tanımladığı isimsiz fonksiyonlara Lambda İfadeleri (λ Expressions) ismini verdi.

```
λ(x)x * x * x
```

- Bir lambda ifadesine geçilen parametre, ifadede değerlendirilip sonuç üretilebilir.

```
(λ(x)x * x * x)(2) // Produces 8
```

- Church, lambda ifadeleri üzerine kurguladığı hesaplama modeline de Lambda Calculus ismini verdi.

Yüksek Seviyeli Fonksiyon - I

- Bir ya da daha fazla lambda ifadesini parametre olarak almak,
- Bir lambda ifadesini geriye döndürmek.

- Bu iki özellikten herhangi birine sahip fonksiyona Yüksek Seviyeli Fonksiyon (Higher-Order Function) denir.

```
h = f o g
```

- En yaygın Yüksek Seviyeli Fonksiyon türü, bileşke fonksiyonlardır (functional composition).

```
f(x) = x + 2
g(x) = 3*x
h(x) = f o g = f(g(x)) = 3*x + 2
```

Higher-Order Function- II

- Bir başka Yüksek Seviyeli Fonksiyon türü ise, bir listeden başka bir liste üreten, eşleştirme/karşı getirme fonksiyonlarıdır (map function).

```
h(x) = x * x
(h,(2,3,4)) // => (4, 9, 16)
```

- Fonksiyonel programlama dillerinde gerek dilin içinde gerek ise sağladığı kütüphanelerde değişik Yüksek Seviyeli Fonksiyonlar bulunur:
  - **Filtreleme** (filter): Eleme yapar.
  - **İndirgeme** (reduce): Birden fazla argümandan bir değer üretir.
  - **Eşleştirme** (map): Bir değerden başka bir değer üretir.

Higher-Order Function- III

- Aşağıdaki akış (stream) haline getirilmiş dizinin elemanlarına
  - önce filtreleme uygulanarak çift olanlar seçiliyor,
  - sonra, elemanların kareleriyle yeni bir liste oluşturuluyor,
  - sonra listedeki elemanların ortalaması alınıyor,
  - Ve en sonra olarak da ortalama, varsa yazılıyor.
- **`filter()`**, **`map()`**, **`average()`** ve **`ifPresent()`**, birer yüksek seviyeli fonksiyondur.

```
Arrays.stream(new int[]{1, 2, 3, 4, 5, 6, 7, 8})
        .filter(x -> x % 2 == 0)
        .map(n -> n * n)
        .average()
        .ifPresent(System.out::println);
```

Birinci Sınıf Vatandaş

- Programlama dillerinde, bir değişkene atanma, metoda argüman olarak geçilme ya da ondan döndürülme vb. işlemlere muhatap olan yapılara birinci sınıf vatandaş anlamında first-class citizen denir.
- Bu tarihi bir isimlendirmedir ve ilk defa Algol dilinde, değişkenler ile fonksiyonları ayırt etmek için kullanılmıştır.
  - Algol'da int ya da real gibi tipler birinci-sınıf vatandaş, fonksiyonlar (procedure) ise ikinci-sınıf vatandaştırlar (second-class citizen).
    - Çünkü fonksiyonlar, int ya da real tipler gibi bir değişkene atanamazlar vs.

Birinci Sınıf Vatandaş Olarak Fonksiyon

- Fonksiyonel programlama dillerinde, fonksiyonlar da birinci sınıf vatandaştır.
- Çünkü,
  - Fonksiyonların tipleri vardır ve değişkenlere atanabilirler.
  - Fonksiyonlar, yüksek-seviyeli fonksiyonlara geçilebilirler ve onlardan döndürülebilirler.

```
h(x) = x * x
print(h,(2,3,4)) // Prints(4, 9, 16)
```

LISP

- LISP (LISt Processing) ile faktoriel alan ve dizi işlemleri yapan iki kod örneği aşağıdaki gibidir.
- Bu kodları online bir LISP yorumlayıcısında örneğin http://rextester.com/l/common_lisp_online_compiler de çalıştırabilirsiniz.

```
(print "Factorial Algorithm")
(defun factorial(n)
   (if (= n 1)
      1
     (* n (factorial (- n 1)))))
(print (factorial 7))
```

```
(write (count 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (remove 5 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (delete 5 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (substitute 10 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (find 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (position 5 '(1 5 6 7 8 9 2 7 3 4 5)))
```

Fonksiyonel Programlamaya Giriş

Fonksiyonel Programlama İlkeleri

- Fonksiyonel programlamanın en temel prensipleri şunlardır:
  - **Değişmezlik** (**immutability**) ve **saf fonksiyonlar** (**pure functions**): Olabildiğince az durum değişikliği yapmaktır. Bir fonksiyonun saf olması ise, değişmezlik ilkesinin bir uygulaması olarak fonksiyonların, kendilerine geçilen parametreler dışında hiç bir şeyi değiştirmemesidir.

  - **Birinci sınıf vatandaş olarak fonksiyonlar** (**Functions as a first-class citizen**): Fonksiyonların, yüksek seviyeli fonksiyonlara (ya da birbirlerine) parametre olarak geçilebilmeleri ya da onlardan geri döndürülebilmeleri.

Değişmezlik ve Saf Fonksiyonlar

Değişmezlik ve Saf Fonksiyonlar

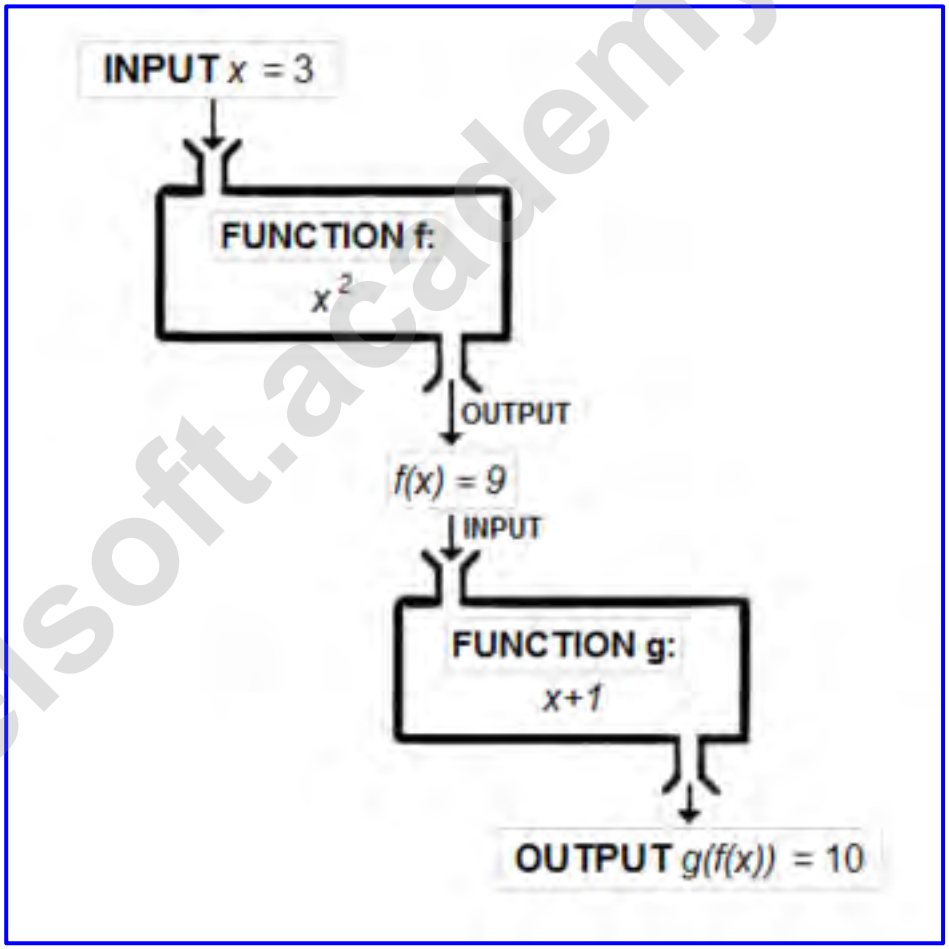
- Fonksiyonel programlama (functional programming) ya da kısaca FP, programı matematiksel fonksiyonlarla kurar.
- Matematiksel fonksiyonun (ya da *saf fonksiyon* (*pure function*) ya da *saf fonksiyon*) en temel özelliği, yan etkisinin (side effect) olmamasıdır.
  - Yani fonksiyon, içinde bulunduğu bağlamdan bağımsızdır, ona etki etmez, onu değiştirmez.
  - Fonksiyonun değiştirebilecekleri, sadece kendi, yerel değişkenleridir.
- Bu durum şu iki prensiple ifade edilir:
  - Değişmezlik (immutability)
  - İlişkisel şeffaflık/saydamlık (referential transparency)

Değişmezlik ve İlişkisel Şeffaflık

- Bir fonksiyon, **saf** (**pure**) olmalı yani kendi yerel değişkenleri (local, automatic, stack variable) dışında hiç bir değişkeni değiştirmemelidir.
- En temel yerel değişkenler ise fonksiyona geçilen parametrelerdir.
- Fonksiyon, yerel olmayan değişkenlere ulaşabilir ama onları değiştiremez.
- İlişkisel şeffaflık/saydamlık (referential transparency) ise bir fonksiyonun aynı parametrelerle çağrılması durumunda daima aynı sonucu üretecek olmasıdır.
  - Bu da fonksiyonun saf olmasından dolayıdır.

```
public int f (int x){
      int y = x * x;
      return y;
}

public int g (int x){
     x += 1;
     return x;
}
```



Değişmezlik - I

- Yandaki Python kodunda **`increment1()`** fonksiyoneldir çünkü sadece parametresini değiştirmektedir ama **`increment2()`** fonksiyonel değildir çünkü yerel olmayan **`a`**'yı değiştirmektedir.

```
a = 0

// Functional function
def increment1(a):
    a += 1  // Changes local a
    return a

// Unfunctional function
def increment2():
    global a
    a += 1 // Changes global a
    return a
```

Değişmezlik - II

- Bir fonksiyon, kendi yerel (local) değişkenleri dışında başka hiç bir değişkeni değiştirmemelidir.
- Fonksiyon, yerel olmayan değişkenlere, örneğin **increment**, ulaşabilir ama onları değiştiremez.
- Yandaki kodda **i** değişebilirken **increment** değişemez.

```
class A{
 static int i = 5;
 static final int increment = 2;

 // Unfunctional function
 public static void increment1(){
   i += increment; // Changes static i
 }

 // Functional function
 public static int increment2(int i){
   i += increment; // Changes local i
   return i;
 }

 public void main(){
    increment1();
    i = increment2(i);
 }
}
```

Değişmezlik - III

- **`increment2()`** fonksiyoneldir çünkü sadece kendisine geçilen parametreyi değiştirmekdir.

- **`increment1()`** fonksiyonel değildir çünkü **`A`** sınıfının bir değişkenini değiştirmektedir.

```
class A{
 static int i = 5;
 static final int increment = 2;

 // Unfunctional function
 public static void increment1(){
   i += increment; // Changes static i
 }

 // Functional function
 public static int increment2(int i){
   i += increment; // Changes local i
   return i;
 }

 public void main(){
    increment1();
    i = increment2(i);
 }
}
```

FP ve Durum Yönetimi - I

- Programlamaya fonksiyonel yaklaşımda, programlarda durumun değiş(tiril)mesinden kaynaklanan karmaşıklığı azaltmak amacına sahiptir.

- Eğer bir fonksiyonun hiç bir yan etkisi yoksa, sadece kendi yerel değişkenlerini değiştiriyor ve aynı girdiyle daima aynı çıktıyı üretiyorsa, hiç bir zaman yerel olmayan değişkenlerin durumlarının tutarlı olup olmadıklarını kontrol etmeye gerek olmayacaktır.

FP ve Durum Yönetimi - II

- FPda yerel olmayan değişkenlerin değerlerinin değişmesinden kaynaklanan riskler ortadan kalkacaktır.
  - Bunu C/C++ ve Python gibi kapsamsız (unscoped) ve korumasız (unprotected) olan global değişken tanımlamaya izin veren dillerde geliştirilen programların riskleriyle kıyaslayın!

FP, Durum Yönetimi ve Java - I

- Bu risk, Java gibi dillerde daha azdır çünkü
  - Global değişkenler yoktur, en geniş kapsam sınıftır (encapsulation),
  - Durumu oluşturan alanlara (field) erişimi, erişim niteleyicilerle (access modifiers) kontrol edebilirsiniz,
    - Örneğin değişkeni **private** yaparak erişilmez kılabilirsiniz,
  - Değiştirilecek alanların değişimini, `set` metotları ile kontrol edebilirsiniz.

- Dolayısıyla Java'da **kontrollu değişim** (**controlled mutability**) söz konusudur.

FP, Durum Yönetimi ve Java - II

- Fpda yerel olmayan değişkenlerin değeri hiç bir zaman değişmiyorsa, bu değişkenler pratikte birer sabite demektir.
- Bu durumda yerel olmayan bir değişken aslen **sabite**dir ve bir değere verilen isimden ibarettir.
  - Java'da sabiteler **final** anahtar kelimesi ile oluşturulur.
- Örneğin aşağıdaki örnekte `i` bir değiken iken `j` aslen "5" değerine verilen bir isimden ibarettir.

```
int i = 3;          // Değişken
final int j = 5; // Sabite
```

FP ve Çok Kanallı Programlama

- Programlamaya fonksiyonel yaklaşım, aynı zamanda çok kanallı (multi-threaded) programların davranışından emin olunmasını sağlar.
- Eğer bir fonksiyonun hiç bir yan etkisi yoksa, sadece kendi yerel değişkenlerini değiştiriyorsa, bu durumda bu fonksiyonu paralel olarak çağırmanın hiç bir riski yok demektir.
  - Bu durumda kilitlemeye (locking ya da synchronizing) gerek yoktur çünkü yarış durumu (race condition) söz konusu değildir.
  - Böylece deadlock riski de oluşmaz.
- Aynı fonksiyon, performans artışı sağlamak amacıyla istenildiği kadar paralel olarak çalıştırılabilir.

FP'nin Diğer Faydaları

- Fonksiyonlar daha anlaşılır olur.
  - Kendi kendini dokümante eder (self-documented),
  - Bir fonksiyonu anlamak için diğer fonksiyonları ve dış durumu anlama ihtiyacı aza iner.
- Fonksiyonel yaklaşımla yazılan kodların birim testi (unit test) çok kolaydır,
- `Set` ya da `Map` gibi yapılarda anahtar olarak rahatlıkla kullanılabilir çünkü durumları dolayısıyla hash kodları değişmez,
- Üretilen sonuçların cache ile saklanması mümkündür, bu da performansı arttırır.

Birinci Sınıf Vatandaş Olarak Fonksiyonlar

Birinci Sınıf Vatandaş

- Fonksiyonların birinci sınıf vatandaş (first-class citizen) olması, fonksiyonların da değişkenler gibi davranabilmesidir.

- Böylece fonksiyonlar, hem yüksek seviyeli fonksiyonlara (ya da birbirlerine) geçilebilirler hem de onlardan geriye döndürülebilirler.

Davranış Parametrikleştirme

- Fonksiyonların, birnci sınıf vatandaşlar gibi kodda dolaştırılabilmesi, **davranış parametrikleştirme** (**behavior parameterization**) olarak isimlendirilebilir.

- Bu şekilde aynı işi farklı değişkenlerle yapmanın yanında, aynı işi farklı şekillerde yapmak da söz konusu olabilir.

- Burada “farklı şekiller”den kasıt, bir değişkenle ifade edilemeyen, farklı algoritmalar, veri işleme vb. kodlardır.

Groovy'de Fonksiyonel Programlama

Groovy

- Groovy, JVM tabanlı, dinamik tipli ve hem nesne-merkezli hem de fonksiyonel bir dildir.
- Groovy'de lambda ifadeleri, "**closure**" denen yapılarda gerçekleştirilir.
- Groovy'de closurelar, anonim, isimsiz fonksiyonlardır öyle ki tanımlandıkları yerdeki değişkenlere ulaşabilirler,
- Ayrıca closurelar değişkenlere atanabilirler ve yüksek seviyeli fonksiyonlara yani metotlara geçilebilirler.

ClosureExamples.groovy

Closure Geçmek

- Davranış parametrikleştirme olarak isimsiz bir fonksiyon olan closureı bir metoda geçmek, kodunuzu, bunu yapamadığınız duruma göre daha kısa ve kolay kılabilir.

- Bunu Michael Feathers şöyle ifade etmişti:

OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.

- Yandaki kodu tekrar kullanımla nasıl daha kısa yaparsınız?
- Renkli kod parçaları switch-case içerisine konulsa bile metotların dönüş tipleri farklı olduğundan, bu dört metot tek bir metoda indirgenemeyecektir!

```
def printEvenNumbers(int n){
    for(int i = 2; i <= n; i += 2)
        print(i)
}

int calculateSumOfEvenNumbers(int n){
    def sum = 0;
    for(int i = 2; i <= n; i += 2)
        sum += i
    return sum;
}

int calculateProductOfEvenNumbers(int n){
    def product = 1;
    for(int i = 2; i <= n; i += 2)
        product *= i
    return product;
}

int[] calculateSquareOfEvenNumbers(int n){
    def square = []
    for(int i = 2; i <= n; i += 2)
        square << i ** 2
    return square;
}
```

Closure ile Çözüm

- Bu koddaki 4 metotta değişken olan kısmın, for çevrimi içerisindeki ifade olduğu kolayca görülebilir.
- Eğer bu metotlara, for çevrimi içerisinde ne yapacağını bildiren bir kod parçası geçilebilirse, bu durumda bu dört metot, tek bir metoda indirgenebilir.
- Bu kod parçası bir closure olabilir.
- Dört farklı closure, tek bir metoda paramere olarak geçilir.

```
static main(args) {
    def obj = new EvenNumberCalculationsWithClosure()
    Closure closure = { println it }
    obj.pickEvenNumbers(10, closure)

    def total = 0
    obj.pickEvenNumbers(10) { total += it }

    println('Total: ' + total)

    def product = 1
    obj.pickEvenNumbers(10) { product *= it }
    println('Product: ' + product)

    def squared = []
    obj.pickEvenNumbers(10) { squared << it ** 2 }
    println('Squared: ' + squared)
}

def pickEvenNumbers(n, block) {
    for(int i = 2; i <= n; i += 2)
        block(i)
}
```

EvenNumberCalculations.groovy

EvenNumberCalculationsWithClosure.groovy

Fonksiyonel Programlamada Zorluk

- Burada, fonksiyonel programlama ile uğraşmamış olmanın getirdiği, "fonksiyonu parametre olarak algılama" zorluğu olduğu açıktır.

- Komutsal ya da nesne-merkezli diller bize bir metodun farklı parametrelerle çağrılabileceğini öğretti ama bir metodun farklı fonksiyonlarla çağrılabileceğini öğretmedi.

- Fonksiyonel programlamaya rahat geçiş için fonksiyonların nasıl kod içerisinde gezebilen parçalar olduğunu iyi anlamak gereklidir.

Python'da Fonksiyonel Programlama

Python

- Python, dinamik tipli, hem komutsal, hem nesne-merkezli hem de fonksiyonel bir dildir.
- Python'da lambda ifadeleri yoktur ama fonksiyonlar zaten birinci sınıf vatandaştırlar.
- Dolayısıyla fonksiyonlar değişkenlere atanabilirler ve diğer fonksiyonlara geçilebilirler.

```
def pickEvenNumbers(n, operation):
    for i in range(2, 11, 2):
        operation(i)
```

```
def printer(k):
    print(k)
pickEvenNumbers(10, printer)

def adder(k):
    global sum
    sum += k

a = adder
sum = 0
pickEvenNumbers(10, a)
print("Sum: ", sum)

def multiplier(k):
    global product
    product *= k

product = 1
pickEvenNumbers(10, multiplier)
print("Product: ", product)

def square(k):
    global squares
    squares.append(k * k)

squares = array('i')
pickEvenNumbers(10, square)
print("Squares: ")
for i in squares:
    print(i, ' ', end='')
```

even_number.py

Uygulama

- Geçilen iki sayı arasında farklı aritmetik işlemler yapan bir fonksiyonu, Groovy, Python ya da tamamen kavramsal bir formatta, yalancı kod (pseudo code) olarak nasıl yazabilirsiniz?

- İşlemler şunlar olabilir:
  - Toplama, çıkarma, çarpma ya da bölme gibi basit işlemler,
  - Üst alma, max/min bulma vb.

Java'da Fonksiyonel Programlama

Fonksiyonel ve Nesne-Merkezli Programlama

Nesne-Merkezli P. ve FP - I

- Nesne-merkezli programlama (NMP) ileFP nasıl bir arada düşünülebilir?
- NMP'nun varlık sebebi, nesnedir.
  - Nesne, duruma ve davranışa sahip olan bir yazılım varlığıdır.
- Fonksiyonel programlamanın durumu değiştirmeme prensibi, nesne-merkezli programlamanın nesnelerine uygulanabilir mi?

Nesne-Merkezli P. ve FP - I

- Nesnenin durumu, tabi olarak değişir:
  - Ürünün fiyatı değişir, stok bilgisi değişir,
  - Öğrencinin aldığı dersler değişir, notları, sınıfı değişir,
  - Oyundaki nesnelerin konumları değişir, arabalar yarışır, askerler koşar, canavarlar vurulur, vs.

- Nesnenin durumu, tabi olarak değişir:
  - Nesnenin üzerindeki metotlar, nesnenin durumunu değiştirir,
  - Nesneler bu metotlarla birbirlerinin durumlarını değiştirir.

```
public class Car {
    private String make;
    private String model;
    private String year;
    private int speed;
    private int distance;

    public Car(String make, ..) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.speed = speed;
        this.distance = distance;
    }

    public void go(int distance){
        this.distance += distance;
    }

    public void accelerate(int speed){
        this.speed = speed;
    }

    public void stop(){
        speed = 0;
    }
}
```

```
public class CarTest {
    public static void main(String[] args) {
        Car car1 = new Car("Mercedes", "C200", ...);

        String infoCar1 = car1.getInfo();
        System.out.println(infoCar1);

        car1.accelerate(80);
        car1.go(10);

        infoCar1 = car1.getInfo();
        System.out.println(infoCar1);

        car1.accelerate(120);
        car1.go(20);

        infoCar1 = car1.getInfo();
        System.out.println(infoCar1);
    }
}
```

```
public class Driver {
    private String name;
    private Car car;

    public Driver(String name, Car car) {
        this.name = name;
        this.car = car;
    }

    public void drive(Place place) {
        int distance = place.getDistance();
        car.accelerate(120);
        car.go(distance);
        car.stop();
    }

    public Car getCar() {
        return car;
    }

    public void setCar(Car car) {
        this.car = car;
    }

    public String getName() {
        return name;
    }
}
```

CarTest.java ve DriverTest.java

- org.javaturk.oofp.ch06.oop.car paketinde.

Nesne-Merkezli P. ve FP - II

- Dolayısıyla FPnın durum değiştirmeme prensibini nesnelere uygulamak her zaman mümkün değildir.

- Nesnenin var olan durumunu değiştirmeden farklı durumda bir nesne oluşturmanın yöntemi, her değişiklik için kopyalama yoluyla eski nesneden, değişikliklerle birlikte yeni bir nesne oluşturmaktır.

- Ama bu hem programlama açısından zordur hem de bellek dolayısıyla da performans açısından sıkıntı doğuracaktır.

Nesne-Merkezli P. ve FP - III

- Yine de değiştirmeme prensibinin faydalarından dolayı, mümkün olan her yerde uygulanması, programları daha temiz kılacaktır.

- Bu amaçla şu önlemler alınabilir.
  - Değişmemesi gereken alanları **final** yapmak, sınıfın metotlarının dahi o alanı değiştirmesini önler,
  - Değişmemesi gereken alanları **private** yapmak ve setter kullanmamak, diğer nesnelerin o alanı değiştirmesini önler.

CarTest.java ve DriverTest.java

- org.javaturk.oofp.ch06.oop.carFp paketinde.

Değişmeyen Nesne - I

- FP'ye uygun olarak durumu değişmeyen nesne (immutable object) oluşturmak için şu önlemler alınabilir:
  - Sınıflar **final** yapmak, dolayısıyla devralınmasını önleyerek metotlarının override edilmesini imkansız kılmak,
  - Bütün alanları **private** ve **final** yapmak,
  - Varsayılan kurucu (default ya da no-arg constructor) sağlamamak varsa **private** yapmak,
  - Sadece argümanlı kurucular, mümkünse bir tane sağlamak,
    - Bu durumda tüm **final** alanların ilk değer atamalarının burada yapılması gereklidir.
  - Setter metotları kullanmamak,
  - Hiç bir metotta durum değişikliği yapmamak.

Değişmeyen Nesne - II

- Nesne üzerindeki torbaların (collection) ise değiştirilemez hale getirildikten sonra getter yoluyla döndürülmesi,
  - Bu amaçla **`Collections`** sınıfı üzerindeki metotlar kullanılabilir.
- Eğer nesne üzerinde değiştirilebilecek nesneler varsa, getter ile bu nesnelerin referanslarını değil, kopyalarını döndürmek.

- Eğer herhangi bir alanının kaçınılmaz olarak değişmesi gerekiyorsa, nesnenin var olan durumunu yeni bir nesneye kopyalayıp, değişen alanın yeni değeriyle birlikte geri döndürülebilir.

Person.java ve Address.java

- org.javaturk.oofp.ch06.oop.address paketinde.

Java API'sinde Değişmeyen Sınıflar - I

- Java API'sinde pek çok değişmeyen nesne (immutable object) vardır:
  - **`java.lang.String`**
  - Tüm wrapper sınıflar: **`java.lang`** paketindeki **`Integer`**, **`Byte`**, **`Character`**, **`Short`**, **`Boolean`**, **`Long`**, **`Double`**, **`Float`**
  - **`java.io.File`**
  - **`java.util.Locale`**
  - **`java.net.URL`** ve **`java.net.URI`**
  - **`java.math.BigInteger`** ve **`java.math.BigDecimal`**

Java API'sinde Değişmeyen Sınıflar - I

- Pek çok **enum** sınıfın nesneleri de değişemezdir:
  - `java.lang.Thread.State`

- Bazı sınıfların nesneleri oluşturulamaz ama statik duruma sahiptirler:
  - `java.lang.Math`

Değiştirmeme ve FP - I

- Aslında fonksiyonel programlamada da değiştirme kaçınılmazdır.
- Bu sebeple, değiştirmeme, ideal ve stratejik bir hedeftir.
- Değiştirmeden tamamen kaçınılamasa bile en aza indirgenebilir.
- Fonksiyonel programlama bu konuda nesne-merkezli programlama yapanlara bir farkındalık ve faydalı bir pratik kazandırmış olur.

Değiştirmeme ve FP - II

- Örneğin aşağıdaki işlemleri yapan fonksiyonlar için ilişkisel şeffaflık söz konusu değildir:
  - Rastgele (random) sayı üretmek,
  - Tarih ve zamanı üretmek,
  - Kullanıcı girdisini almak,
  - Ağdan ya da dosyadan veri okumak vs.
- Bu fonksiyonlar her seferinde farklı değer döndüreceklerdir.
- Ama döndürdükleri değerler değişmeyen (immutable) olabilir, olmalıdır.

Değiştirmeme ve FP - III

- Fonksiyonel programlamanın temel iddiası, veriyi değiştirmeme üzerinedir.
  - Nesne ise veri değildir!
- Fonksiyonel programlama ve prensipleri bu anlamda daha çok veri işleme için kullanılmalıdır.
- Torbalar (collections) ve işlenmeleri söz konusu olduğunda fonksiyonel yapıları kullanmak çok daha uygun ve faydalıdır.

Paradigmalar - I

- Komutsal programlamada (imperative programming), emirler programlama dili vasıtasıyla derleyiciye verilir.
- Nesne-merkezli programlamada (object-oriented programming) ise emirler doğrudan nesnelere verilir.
  - Nesneler iş yaparken diğer nesnelere ya da bilgisayara emirler verir.
- Ayrıca nesneler ile iş ortamındaki kavramlar ve mekanizmalar modellenir.

- Dolayısıyla bir programlama dilinin hem komutsal hem de nesne-merkezli yapılara sahip olması kaçınılmazdır.

Paradigmalar - II

- Dolayısıyla FP, komutsal ve nesne-merkezli programlamaya alternatif olarak görülmemelidir.
- FYı, daha etkin programlama yapmayı sağlayacak bir yaklaşım ya da teknik olarak ele almak daha makuldur.
- Çünkü FP, programlamadaki tüm ihtiyaçları çözemez, belli türden problemlerin çözülmesini kolaylaştırır ve bu tür kodları daha kısa, anlaşılır ve şık hale getirir.

OO pattern/principle	FP pattern/principle
Single Responsibility Principle	Functions
Open/Closed principle	Functions
Dependency Inversion Principle	Functions, also
Interface Segregation Principle	Functions
Factory pattern	Yes, functions
Strategy pattern	Oh my, functions again!
Decorator pattern	Functions
Visitor pattern	Functions

Fonksiyonel Programlama Dilleri

- Tamamen fonksiyonel olan programlama dillerinin (pure fonctional languages) pek popüler olmadığı söylenebilir.

- Genel olarak var olan, komutsal ya da nesne-merkezli dillerin fonksiyonel özelliklere de sahip olmasıdır.
  - İlk fonksiyonel dil olan LISP, hem komutsal hem de fonksiyonel özelliklere sahiptir.
  - Scala, Groovy ve Python hem nesne-merkezli hem de farklı ağırlıklarla, fonksiyonel özelliklere sahip dillerdir.

- Java ve C#, nesne-merkezli diller olarak sonradan eklemeyle fonksiyonel özelliklere sahip olmuşlardır.

Komutsal ve Açıklayıcı

- Komutsal programlamada (imperative programming), programda, dilin yardımıyla, neyin nasıl olması gerektiği adım adım kurulur.
  - Adımlar ise derleyiciye verilen komutlar ya da emirlerdir.
- Açıklayıcı programlamada (declarative programming) ise daha çok neyin istendiğini ifade etmek yeterlidir, kullanılan dil ya da framework nasıllığı sağlar.
- AP, KPdan daha soyut, kolay ve anlaşılırdır.
  - Örneğin C komutsal, SQL ise açıklayıcı bir dildir.
- Diller ya da yapılar soyutlaştıkça komutsaldan açıklayıcı programlamaya doğru ilerlenir ve geliştirme hızı artar.

- Yandaki ilk kod JDBC ile ikinci kod ise JPA ile yazılmıştır.
- İlk kodda "nasıllık" kodda açıkça görülmektedir.
- İkinci kodda ise sadece istenen şey vardır, nasıl olacağı ise saklıdır.

```
String SAVE_PERSON_QUERY = "INSERT INTO PERSONS
                                VALUES(?,?,?,?)";

public void savePerson(Person person){
   PreparedStatement stmt = null;
   try {
      stmt = conn.prepareStatement(SAVE_PERSON_QUERY);
      stmt.setInt(1, person.getId());
      stmt.setString(2, person.getFirstName());
      stmt.setString(3, person.getLastName());
      stmt.setDate(4, person.getDobAsSQlDate());
      stmt.executeUpdate();
   } catch (SQLException e) {
      ...
   }
}
```

```
public void savePerson(Person person){
   em.persist(person);
}
```

Açıklayıcı Programlama

- Öte taraftan, fonksiyonel programlama dilleri, şu iki noktadan dolayı daha açıklayıcı programlama stiline sahip oldukları düşünülür:
  - Fonksiyonların *birinci sınıf vatandaş* olmaları ve
  - Kendi içlerinde ya da kütüphanelerinde barındırdıkları pek çok fonksiyon.
- Fonksiyonların birbirlerine ve daha yüksek seviyeli fonksiyonlara yani metotlara geçilebilmeleri, birden fazla metotta ortak olan kısımların fonksiyon olarak soyutlanabilmesini sağlar.
- Soyutlanan metot eğer dilde ya da kütüphanesinde varsa, bu durum açıklayıcı programlama örneği oluşturur.

```
(write (count 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (remove 5 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (delete 5 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (substitute 10 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (find 7 '(1 5 6 7 8 9 2 7 3 4 5)))
(terpri)
(write (position 5 '(1 5 6 7 8 9 2 7 3 4 5)))
```

Kaynaklar

- Nesnelerin durumlarının değişmemesi ile ilgili şu kaynaklara da bakılabilir:
  - Joshua Bloch'ın **Effective Java 3rd Ed.** kitabı 17. madde
  - GoF'un **Design Patterns** kitabı *Flyweight* kalıbı

Java'da İsimsiz Sınıflar

Java'da İsimsiz Metotlar!

- Bazı gerçekler:
- Java'da fonksiyon yoktur, metot vardır.
  - Java'nın metotları muhakkak sınıf içinde tanımlanır.
  - Java'nın metotlarına, static ise sınıf üzerinden değilse nesneler üzerinden ulaşılır.

- Java'da isimsiz metot yoktur ama isimsiz sınıf (anonymous class) vardır.
  - İsimsiz sınıf bir iç sınıf (inner class) türüdür.
- Ve eğer bir isimsiz sınıf yapıp içine tek bir metot koyarsanız, bu pratikte isimsiz metoda karşı gelebilir.

Geri Çağırma (Call Back) - I

- Yazılım sistemlerinde sıklıkla, bir buttonun tıklanması ya da bir kullanıcının sistemi kullanmaya başlaması (login) gibi bazı olayların takibi gereklidir.
- Bu amaçla genel olarak, olayın kaynağı olan nesneye, olayın olduğunu bildirmesi için bir fonksiyon geçilir.
- İlgilenilen durum oluştuğunda da nesne, kendisine geçilen fonksiyonu çağırır.
- Bu mekanizmaya geri çağırma (callback), geri çağrılan metoda da geri çağırma metodu (callback method) denir.
- **Observer** (event-notification ya da publisher-subscriber) tasarım kalıbı bu problemi ve çözümünü tarif eder.

Geri Çağırma (Call Back) - II

- Geri çağırmayı Java'da kurgulamak için olayın kaynağına, fonksiyon değil, üzerinde belirli bir metot olan nesne geçilir.
  - Çünkü Java nesne merkezlidir ve nesne geçilmesi durumu çok daha geniş bir hareket alanı sağlar.
- Bu durumda olayın kaynağı olan nesnenin, olayın olması durumunda hangi metodu çağıracağını bilmesi gerekir.

- Bu sebeple Java'da geri çağırma nesneleri, üzerinde genelde bir tane geri çağırma metodu bulunduran arayüzlerden türetilir.
- Olayın kaynağı da bu arayüzü bilir.

TimerExample.java

- *org.javaturk.oofp.ch04.callBack* paketi.

İsimsiz Sınıflar - I

- Genelde geri çağırma metotlarının üzerinde bulunduğu sınıfların tek kullanımlık bir nesnesine ihtiyaç duyulur.
  - Yani arayüzü gerçekleştiren sınıfın bir tek nesnesine ihtiyaç vardır ve bu nesne sadece bir yerde kullanılır.

- Bu durumda Java, arayüzü yerine getiren sınıfın isimsiz bir şekilde, hızlıca oluşturulmasına ve bunun yapıldığı yerde tek bir nesnesinin yaratılıp kullanılmasına izin verir.

- Bu şekilde oluşturulan sınıflara isimsiz sınıf (anonymous class) denir.

İsimsiz Sınıflar - II

- İsimsiz sınıflar sıklıkla olayları (event) yakalamada kullanılırlar.
  - Çünkü çoğu zaman özel bir duruma işaret eden olay nesnesi sadece bir yerde yakalanır ve gereği yapılır.

```
Timer t = new Timer(1_000, new ActionListener() {

    @Override
    public void actionPerformed(ActionEvent e) {
        ...
    }
});
```

- Örnekteki **`ActionListener`**, sadece **`actionPerformed()`** metoduna sahip bir arayüzdür.

TimerExample.java

- *org.javaturk.oofp.ch04.anonymous.timer* paketi.

İsimsiz Sınıflar - III

- Benzer şekilde çoğu zaman bir GUI bileşeninin durumundaki bir değişikliğe işaret eden olay nesnesi sadece bir yerde yakalanır ve gereği yapılır.

```
button.setOnAction(new EventHandler<ActionEvent>() {

        @Override
        public void handle(ActionEvent event) {
                ...
        }
});
```

- Örnekteki **`EventHandler`**, sadece **`handle()`** metoduna sahip bir arayüzdür.

MyApplication.java

- *org.javaturk.oofp.ch04.anonymous.event* paketi.

İsimsiz Sınıflar - IV

- Sınıflar, tanımlamaya (class declaration) sahip oldukları halde isimsiz sınıflar ifadedirler (expression).
- İsimsiz sınıf ifadesi, bir kurucu çağrısına benzer ama içinde tekrar tanımlanan (override) metot ya da metotlar vardır.
- İsimsiz sınıflar hem arayüzleri gerçekleştirmede hem de sınıfları genişletmede kullanılabilirler.
- İsimsiz sınıflar genelde sadece bir metota sahip arayüzleri gerçekleştirmede kullanılmalarına rağmen birden fazla metodu yeniden tanımlayacak şekilde kullanılabilirler.
  - Olay yapılarında çağrılacak metot bir tane olduğundan, genelde tek metodu tekrar tanımlamada kullanılırlar.

İsimsiz Sınıflar - V

- İsimsiz sınıf ifadesi şöyledir:
  - new operatörü,
  - Gerçekleştirilecek arayüzün ya da genişletilecek sınıfın ismi,
    - new operatöründen sonra gelen tipin sınıf olması durumunda, kurucuya geçilecek parametreler de sıralanabilir.
    - Eğer tip arayüz ise, arayüzlerin kurucuları olmadığından, sanki varsayılan kurucu çağrılıyormuş gibi içi boş iki parantez bulunur.
  - Sınıf bloğu.
- İsimsiz sınıflar birer ifade olduklarından, bloklarında başka ifadeler olamaz, sadece metot gibi başka bloklar olabilir.
- İsimsiz sınıf ifadesi, arayüz gerçekleştirmesinde new operatöründen sonra arayüzün varsayılan kurucusunu çağırıyor bir görüntüye sahip olduğundan tuhaf görünür.

```
public interface DoerInterface {

        void doIt();

        void doThat();
}
```

```
new DoerInterface(){
    {
        System.out.println("Instance initializer block.");
    }

    @Override
    public void doIt(){
        System.out.println("I!ll always do it :)");
    }

    @Override
    public void doThat(){
        System.out.println("I'll always do that :)");
    }
}).doIt();
```

AnonymousDoerClassTest.java

- *org.javaturk.oofp.ch04.anonymous.doer* paketi.

İsimsiz Sınıflar - VI

- İsimsiz sınıflar, içinde bulundukları sınıfın üyelerine erişebilir.
- İsimsiz sınıflar, içinde bulundukları bloğun yerel değişkenlerine final ya da değeri değişmediği (effectively final) hallerde ulaşabilir.
  - Bu durumda da yerel değişkeni değiştiremez.
- İsimsiz sınıflar, sabite olmaları şartıyla statik alanlar tanımlayabilirler.

İsimsiz Sınıflar - VII

- İsimsiz sınıflar ayrıca şunları tanımlayabilirler:
  - Alanlar,
  - Yerel sınıflar (local classes),
  - Üst tipinde olmayan metotlar,
  - Nesne ilk değer blokları
- İsimsiz sınıflar, statik ilk değer atama blokları ile üye arayüzler tanımlayamazlar.

WeirdAnonymousDoesClassTest.java

- *org.javaturk.oofp.ch04.anonymous.doer* paketi.

Lambda İfadeleri

Arayüz ve Gerçekleştirmeleri

- Elimizde aşağıdaki gibi bir arayüz ve bu arayüzü argüman olarak kabul eden bir metodun olduğunu düşünelim.
- Arayüzün gerçekleştirmelerinin nesnelerini **doMath** metoduna geçmek için en uzun yol olan, **Math** arayüzünü gerçekleştiren sınıflar ve bu sınıfların nesnelerini oluşturmak yerine isimsiz sınıfları kullanabiliriz.

```
public interface Math{

   double calculate(double arg1, double arg2);
}
```

```
public static void doMath(Math math, double arg1, double arg2){

   System.out.println(math.calculate(arg1, arg2));
}
```

İsimsiz Sınıf Gerçekleştirmesi

```
Math adder = new Math(){
    @Override
    public double calculate(double  arg1, double arg2){
      return arg1 + arg2;
    }
};
doMath(adder, 3, 5);

doMath(new Math(){
    @Override
    public double calculate(double  arg1, double arg2){
      return arg1 * arg2;
    }
}, 3, 5);
```

AnonymousMathImplementations.java

- *org.javaturk.oofp.ch06.lambda.math* paketi.

Lambda İfadesi Gerçekleştirmesi

```
Math adder = (double arg1, double arg2) -> { return arg1 + arg2;};
doMath(adder, 3, 5);

Math multiplier = (double arg1, double arg2) -> { return arg1 * arg2;};
doMath(multiplier, 3, 5);

doMath((double arg1, double arg2) -> { return arg1 - arg2;}, 3, 5);

doMath((double arg1, double arg2) -> { return arg1 / arg2;}, 3, 5);
```

LambdaMathImplementations.java

- *org.javaturk.oofp.ch06.lambda.math* paketi.

Lambda İfadesi

- Java'da lambda ifadeleri (lambda expressions), tek bir metoda sahip bir arayüzü gerçekleştiren ifadelerdir.
- Lambda ifadeleri, pratikte arayüz tipinde bir fonksiyon değerine (a function value with an interface type) karşılık gelir.
  - Dolayısıyla tipleri vardır ve gerçekleştirdiği arayüzdür.
  - Bir lambda ifadesi, gerçekleştirdiği arayüz tipinden bir değişkene atanabilir, parametre olarak metoda geçilebilir ya da metottan geri döndürülebilir.
- Lambda ifadeleri, alternatifi olduğu isimsiz sınıflardan çok daha kısa ve anlaşılır ifadelerdir.

Lambda İfadesinin Yapısı - I

- Bir lambda ifadesi şu parçalardan oluşur:
  - Parantez içinde virgül ile ayrılmış formal parametre listesi.
    - Parametre listesi, lambda ifadesinin gerçekleştirdiği arayüzdeki metodun parametreleriyle aynı sayıda ve aynı tipte olmalıdır.
  - Ok "->" ve
  - Tek bir ifade ya da "{ }" ile bir blok.
    - Tek bir ifade varsa bloğa gerek yoktur.
    - Birden fazla ifade var ya da return kullanıyorsa blok gereklidir.

```
(int i) -> System.out.println(i);
(int i) -> {System.out.println(i);};

(int i) -> i > 0;
(int i) -> { return i > 0;};

(int i1, int i2) -> i1 > i2;
(int i1, int i2) -> { return i1 > i2;};
```

Lambda İfadesinin Yapısı - II

- İstenirse parantez içindeki parametre listesinde tipler düşürülebilir.
  - Eğer tek bir parametre varsa, parantezler de "**()**" düşürülebilir.
- Bir lambda ifadesinde "**->**"den sonra blok (**{}**)yok ise ve dönüş tipi varsa, JVM ilk ifadenin sonucunu geri döndürecektir.
  - Bu durumda return gerekli değildir.
- Blok ve metodun döndürdüğü bir tip varsa bu durumda return zorunludur.
- Geri dönüş değeri söz konusu değilse bu durumda ya tek olan ifade ya da varsa blok çalıştırılacaktır.

Lambda İfadesinin Yapısı - III

- Ok "->" zorunludur.
  - Tek istisnası metot referanslarının kullanıldığı haldir.

```
(int i) -> {System.out.println(i);};
(int i) -> System.out.println(i);
(i) -> System.out.println(i);
 i -> System.out.println(i);


 i -> i > 0;

(i1, i2) -> i1 > i2;
(i1, i2) -> { return i1 > i2;};

(i1, i2) -> {
                boolean b = false;
                if (i1 > i2)
                  b = true;
                return b;
            };
```

LambdaVariations1.java

- *org.javaturk.oofp.ch06.lambda* paketi.

Fonksiyonel Arayüz - I

- Tek bir soyut metoda sahip arayüzlere fonksiyonel arayüz (functional interface) denir.
- Lambda ifadelerinin tipi, fonksiyonel arayüz olmak zorundadır.
  - Aksi taktirde derleme zamanı hatası oluşur.
- Yani lambda ifadeleriyle gerçekleştirilecek arayüzlerde sadece ve sadece bir tane soyut metot olmalıdır.
- Lambda ifadesi de o tek olan soyut metoda bir gerçekleştirme verir.

Fonksiyonel Arayüz - II

- Fonksiyonel arayüzlerde default ve static metotlar olabilir.
- default ve static metotların gerçekleştirmeleri zaten yine fonksiyonel arayüz üzerinde olduğundan, bu metotlar arayüzün "fonksiyonel" olmasına bir zarar vermez.
- Fonksiyonel bir arayüzde bir tek metot olması zorunluluğu nasıl açıklanabilir?

FunctionalInterface

- Fonksiyonel olması beklenen arayüzlere birden fazla metot ekleme hatasına düşmemek için, o arayüzün fonksiyonel olması gerektiği **`java.lang.FunctionalInterface`** notu (annotation) ile ifade edilebilir.
  - Zorunlu değildir, bilgi ve hatadan korumak için kullanılır.
- Fonksiyonel olması beklenen arayüzün isminden önce kullanılacak **`@FunctionalInterface`** notu ile arayüze birden fazla metot yazılması engellenir.
  - Bu durumda Java derleyicisi o arayüzde hata verecektir.

Java APIsindeki Fonksiyonel Arayüzler

- Java SE 8 APIsinde pek çok fonksiyonel arayüz vardır.
- Bunların bir kısmı Java SE 8 ile birlikte gelmiş, bir kısmı ise zaten var olan tek metotlu arayüzlerdir.
- Hepsinin APIsinde `@FunctionalInterface` notunu görebilirsiniz.
- Dolayısıyla bu arayüzlerin gerçeklemelerini lambda ifadesi oalrak yazabilirsiniz.
- Örneğin:
  - `java.util.Comparator`

ComparatorLambda.java

- *org.javaturk.oofp.ch06.functions* paketi.

Lambda İfadesi: Birinci Sınıf Vatandaş

- Lambda ifadeleri birinci sınıf vatandaştırlar, yani lambda ifadeleri
  - Bir metoda parametre olarak geçilebilir,
    - Bu durumda parametrenin tipi, lambda ifadesinin gerçeklediği fonksiyonel arayüzün tipidir.
  - Bir metottan geriye döndürülebilir,
    - Bu durumda dönüş tipi, lambda ifadesinin gerçeklediği fonksiyonel arayüzün tipidir.

LambdaVariations2.java

- *org.javaturk.oofp.ch06.lambda* paketi.

Uygulama

- Aşağıdaki arayüzü farklı dillerde (Türkçe, İngilizce vs.) ya da şekillerde gerçekleyen en az üç tane lambda ifadesi yazın.

- Yazdığınız lambda ifadelerini kendisine geçeceğiniz bir metot yazıp, lambda ifadelerini main metottan geçerek çalıştırın.

```
@FunctionalInterface
public interface Selamlama{

    void selamla(String kimi);
}
```

Uygulama

- Celcius, Fahrenheit ve Kelvin sıcaklık sistemleri arasında dönüşümler yapabilecek bir yapıyı lambda ifadeleri kullanarak tasarlayıp kodlayın.

Lambda İfadeleri ve Çevresi - I

- Lambda ifadeleri içinde bulundukları kapsamdaki (scope) değişkenlere erişebilir.

- Lambda ifadeleri statik kapsama (static scope) sahiptirler.
  - Dolayısıyla gerçekledikleri arayüzden herhangi bir değişken devralmazlar,
  - İçinde bulundukları kapsamdaki değişkenleri tekrar tanımlayamazlar,
  - Dolayısıyla herhangi bir gölgeleme (shadowing) de söz konusu değildir.

Lambda İfadeleri ve Çevresi - II

- Eğer bir lambda ifadesi, içinde bulunduğu kapsamdaki bir yerel değişkene ulaşırsa, bu yerel değişkenin final ya da "effectively final" olması gereklidir.
  - Bir değişken tanıtılırken final olarak tanıtılırsa, sabite olur ve değeri değişemez.
  - Eğer bir değişken final olarak tanıtılmadığı halde değeri, yeni bir atama ya da "++" veya "--" gibi ön ya da son arttırma veya azaltma işlemcileriyle değişmiyorsa, gerçekte "effective final"dır yani sabite olarak kalmaya devam ediyordur.

Lambda İfadeleri ve Çevresi - III

- Lambda ifadeleri, içinde bulunduğu kapsamdaki static ve nesne değişkenlerine ulaşabilir.

- Bu değişkenlerin final ya da "effectively final" olmasına gerek yoktur.

- Üzerinde değişkenlerine ulaştığı nesnenin final ya da "effectively final" olduğunu unutmayın!

LambdaScope.java

- *org.javaturk.oofp.ch06.lambda* paketi.

Lambda İfadeleri ve Arayüzleri - I

- Lambda ifadeleri statik kapsama (static scope) sahiptirler.
  - Dolayısıyla gerçekledikleri arayüzden herhangi bir değişken devralmazlar,
- Çünkü lambda ifadeleri, gerçekleştirdiği arayüzün tipinden nesnelerdir.
- Bu yüzden lambda ifadeleri, arayüzü üzerinde var olan, public, static ve final olan durum bilgisi ile varsayılan (default) metotlara ulaşır.
- Lambda ifadeleri static metotlara ulaşamaz çünkü static metotlar sadece ve sadece arayüz üzerinden çağrılabilirler, alt arayüzler tarafından bile devralınamazlar.

Lambda İfadeleri ve Arayüzleri - II

- Fonksiyonel arayüzler birbirlerinden miras devralabilirler.
- Ama alt arayüz, yeni bir abstract metot ekleyemez.
- Fonksiyonel arayüzlerin lambda ifadeleri arasında de **is-a** ilişkisi geçerlidir.

LambdaProperties.java

- *org.javaturk.oofp.ch06.lambda* paketi.

Lambda İfadeleri ve Hedef Tipleri - I

- Bir lambda ifadesinin tipinin, gerçeklediği fonksiyonel arayüz olduğunu belirtmiştik.
- Tip olan arayüz, lambda ifadesinin bir parçası değildir.
  - Dolayısıyla sadece lambda ifadesine bakarak onun tipini anlamak mümkün değildir.
  - Örneğin aşağıdaki lambda ifadesi, hiç bir argüman almayan ve `String` döndüren bir metoda sahip herhangi bir fonksiyonel arayüzün tipinde olabilir.

```
() -> "Done!";
```

- Dolayısıyla derleyici bir lambda ifadesinin tipini nasıl belirler?

Lambda İfadeleri ve Hedef Tipleri - II

- Java'da bir lambda ifadesi daima bir bağlamda bulunur ve bu bağlam ifadenin tipini verir.
- Yani bir lambda ifadesi en temelde aşağıdaki bağlamlardan birinde ifade edilir:
  - Gerçeklediği arayüzün tipinde bir değişkene atanma,
  - Gerçeklediği arayüzün tipinde argüman alan bir metoda ya da kurucuya parametre geçilme,
  - Gerçeklediği arayüzün tipinde bir dönüş değerine sahip metottan döndürülme.

- Bu konuda daha fazla alternatif vardır ama şu an için bu kadar ile yetinelim.

LambdaTargets.java

- *org.javaturk.oofp.ch06.lambda.targeting* paketi.

Lambda İfadeleri ve Sıradışı Durumlar

Lambda İfadelerinde Sıradışı Durum - I

- Lambda ifadeleri sıradışı durum (exception) fırlatabilir.
- Lambda ifadesinin fırlattığı sıradışı durum eğer bir checked exception ise bu durumda bu sıradışı durum, lambda ifadesinin gerçeklediği soyut metodun arayüzünde throws ile belirtilmelidir.
  - Çünkü checked exceptionlar ya yakalanmalı ya da metot arayüzünde throws ile fırlatıldığı belirtilmelidir.

Lambda İfadelerinde Sıradışı Durum - II

- Soyut metodun arayüzünde sıradışı durum fırlatıldığının throws ile belirtilmesi, lambda ifadesinin de sıradışı durum fırlatmasını gerektirmez.
  - Lambda ifadesi hiç sıradışı durum fırlatmamayı ya da arayüzde listelenen sıradışı durumun bir alt sınıfından sıradışı durum fırlatmayı da tercih edebilir.
  - Yerine geçme (substitubility) özelliği geçerlidir.

- Lambda ifadesinde fırlatılan sıradışı durum eğer bir unchecked exception ise bunun zaten soyut metodun arayüzünde ifade edilmesine gerek yoktur.

ExceptionsInLambda.java

- *org.javaturk.oofp.ch06.lambda.exception* paketi.

Örnekler

Örnekler

- *org.javaturk.oofp.ch06.examples* paketindeki örnekleri inceleyin.

Uygulama

- Daha önce Groovy'de closure ile çözdüğümüz problemi Java'da lambda ifadeleriyle nasıl çözersiniz?

- *org.javaturk.oofp.ch06.lambda* paketindeki **`EvenNumberOperations`** arayüzüne ve **`EvenNumberOperationsTest`** sınıfına bakıp, test sınıfındaki dört metodu, lambda ifadeleriyle nasıl tek bir metoda indirgeyebileceğinizi düşünün.

Hazır Arayüzler (Built-in Interfaces)

Yaygın Fonksiyonel Arayüzler - I

- Gerek Java API'sinde gerek ise projelerde sıklıkla aslen aynı yapıda olan fonksiyonel arayüzlerin farklı bağlamlarda farklı amaçlarla kullanıldıkları görürsünüz.
- Örneğin aşağıdaki iki arayüzü göz önüne alın:

```
@FunctionalInterface
public interface UniIntegerChecker{
        boolean method(int i);
}
```

```
@FunctionalInterface
public interface BiIntegerChecker{
        boolean method(int i, int j);
}
```

- Bu arayüzleri hangi farklı amaçlarla farklı lambda ifadeleriyle gerçekleyebilirsiniz?

CommonFunctionalInterfaces.java

- *org.javaturk.oofp.ch06.functions* paketindeki örnekleri inceleyin.

Yaygın Fonksiyonel Arayüzler - II

- Bu örnekteki gibi çok yaygın tiplerden parametre alıp yine yaygın tiplerden dönüş değeri olan ya da hiç bir şey döndürmeyen metotlar çok sık kullanılmaktadır.
- Bu türden metotların içinde bulunduğu fonksiyonel arayüzleri farklı amaçlar için sürekli tekrar tekrar oluşturmak gerekir.
- Java, bu durumda kalmamamız ve kendi APIsindeki böyle ihtiyaçlara cevap vermek amacıyla en temel fonksiyonel arayüzleri **`java.util.function`** paketinin altında tanımlamıştır.
- Bunlara hazır fonksiyonel arayüzler (built-in functional interfaces) denmektedir.

Hazır Fonksiyonel Arayüzler

- Java 8'de ve 9'da **`java.util.function`** paketinde toplam 43 tane hazır fonksiyonel arayüz ya da fonksiyon vardır.
- Bu arayüzler aynı zamanda Java API'sinde de kullanılmaktadır.
- Dolayısıyla, kendi fonksiyonel arayüzünüzü yazmadan önce buraya bakın.

java.util.function

Interfaces

BiConsumer
BiFunction
BinaryOperator
BiPredicate
BooleanSupplier
Consumer
DoubleBinaryOperator
DoubleConsumer
DoubleFunction
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function
IntBinaryOperator
IntConsumer
IntFunction
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer
ObjIntConsumer
ObjLongConsumer
Predicate
Supplier
ToDoubleBiFunction
ToDoubleFunction
ToIntBiFunction
ToIntFunction
ToLongBiFunction
ToLongFunction
UnaryOperator

java.util.function.Function - I

- **`java.util.function.Function`** fonksiyonel arayüzü, tek bir argüman alıp bir değer döndüren her türlü iş için kullanılabilir.

```
@FunctionalInterface
public interface Function<T,R>
```

Represents a function that accepts one argument and produces a result.

This is a functional interface whose functional method is `apply(Object)`.

Since:
1.8

apply

```
R apply(T t)
```

Applies this function to the given argument.

Parameters:
t - the function argument

Returns:
the function result

java.util.function.Function - II

- **`Function<T, R>`** fonksiyonel arayüzü ve tabi olarak üzerindeki tek soyut metot olan **`apply(T t)`** genel (generic) bir metottur ve lambda ifadesini oluştururken gerçek tipler geçilmelidir.

FunctionExample.java

- *org.javaturk.oofp.ch06.functions* paketi.

java.util.function.Function - III

- **`Function<T, R>`** fonksiyonel arayüzü üzerinde soyut olmayan 3 metot daha vardır:
  - **`theAfter`** ve **`compose`**, başka fonksiyonlarla bileşim içindir.
    - İleride ele alınacaklardır.
  - İdentity ise daima argümanını geriye döndüren bir **`Function`** üretir.

Modifier and Type	Method
default <V> Function<T,V>	andThen(Function<? super R,? extends V> after)
R	apply(T t)
default <V> Function<V,R>	compose(Function<? super V,? extends T> before)
static <T> Function<T,T>	identity()

java.util.function.BiFunction

- **`BiFunction<T, U, R>`** fonksiyonel arayüzü, **`Function<T, R>`** arayüzünün iki argüman alan halidir.

- Üzerindeki metot **`apply(T t,U u)`** da genel (generic) bir metottur ve iki argüman alıp bir değer döndürmektedir.

BiFunctionExample.java

- *org.javaturk.oofp.ch06.functions* paketi.

En Temel Hazır Arayüzler

- En çok kullanılan ve en temel hazır arayüzler şunlardır:

Interface name	Arguments	Returns	Example
`Predicate<T>`	`T`	`boolean`	Has this album been released yet?
`Consumer<T>`	`T`	`void`	Printing out a value
`Function<T,R>`	`T`	`R`	Get the name from an `Artist` object
`Supplier<T>`	None	`T`	A factory method
`UnaryOperator<T>`	`T`	`T`	Logical not (`!`)
`BinaryOperator<T>`	`(T, T)`	`T`	Multiplying two numbers (`*`)

Diğer Hazır Fonksiyonel Arayüzler

- Diğer hazır fonksiyonel arayüzlerden bazıları şunlardır:
  - **`Consumer<T>`** ve **`void accept(T t)`**
    - Setter metodu gibi çalışır.
  - **`BiConsumer<T,U>`** ve **`void accept(T t, U u)`**
    - İki argümanın setter metodu gibi çalışır.
  - **`Supplier<T>`** ve **`T get()`**
    - Getter metodu gibi çalışır.
  - **`Predicate<T>`** ve **`boolean test(T t)`**
    - Geçilen argümanı test eder.
  - **`BiPredicate<T,U>`** ve **`boolean test(T t, U u)`**
    - Geçilen iki argümanı test eder.

BuiltinFunctionExamples1.java

- *org.javaturk.oofp.ch06.functions* paketi.

Calculator.java

- *org.javaturk.oofp.ch06.functions.other* paketi.

Basit Tipler İçin Hazır Arayüzler

- Hazır fonksiyonel arayüzler arasında tamamen genel olmayan (non-generic), kısmen ya da bazen tamamen basit tiplere özel olarak oluşturlulmuş olanlar da vardır:
  - `IntFunction<R> - R apply(int)`
  - `IntToDoubleFunction - double applyAsDouble(int)`
  - `IntToLongFunction - long applyAsLong(int)`
  - `IntConsumer – void accept(int)`
  - `LongConsumer – void accept(long)`
  - `DoubleConsumer – void accept(double)`

BuiltinFunctionExamples2.java

- *org.javaturk.oofp.ch06.functions* paketi.

BinaryOperator<T, T, T>

- **`BinaryOperator<T, T, T>`**, **`BiFunction`**'un bir alt arayüzüdür ve üçü de aynı tipten olan nesnelerle çalışır.
- **`BiFunction`**'dan devraldığı **`apply()`** ve **`andThen()`** metotlarına şu iki statik metodu ekler:
  - **`maxBy(Comparator<? super T> comparator)`**: Geçilen **`Comparator`** nesnesine göre, argümanlardan büyük olanı döndüren bir BinaryOperator üreten bir metottur.
  - **`minBy(Comparator<? super T> comparator)`**: Geçilen **`Comparator`** nesnesine göre, argümanlardan küçük olanı döndüren bir BinaryOperator üreten bir metottur.

BinaryOperatorExample.java

- *org.javaturk.oofp.ch06.functions* paketi.

Uygulama

- **`BinaryOperator`**'ün **`maxBy()`** metodunu ve **`Comparator`**'ü kullanarak geçilen iki **`Book`** nesnesinden sayfa sayısı daha çok olanı döndürecek şekilde lambda ifadesi olarak gerçekleyin.

Java APIsinde Fonksiyon Kullanımlar

- Hazır fonksiyonlar Java SE APIsinde yaygın olarak kullanılmaktadır.
- Örneğin
  - `java.lang.Iterable` arayüzündeki `forEach(Consumer<? super T> action)`
  - `java.util.Collection` arayüzündeki `removeIf(Predicate<? super E> filter)`
  - `java.util.List` arayüzündeki `replaceAll(UnaryOperator<E> operator)`

BuiltinFunctionsInAPI.java

- *org.javaturk.oofp.ch06.functions* paketi.

Calculator.java

- *org.javaturk.oofp.ch06.functions.other* paketi.

- Bu örnekte hazır fonksiyonların lambda ifadeleri, Calculator nesnesinin değişkenleri olarak konumlandırılmıştır.

LambdaInLambda.java

- *org.javaturk.oofp.ch06.functions.other* paketi.

- Bu örnekte bir lambda ifadesinin bloğunda başka bir lambda ifadesi kullanılmaktadır.

Hazır Fonksiyonların Bileşimleri

Fonksiyon Bileşimleri

- Hazır arayüzlerin arka arkaya uygulanması için bir zincir şeklinde ifade edilmesi mümkündür.
- Böylece fonksiyon bileşimleri elde edilir.
- Bu amaçla hazır fonksiyonlara şu iki metot ile başka hazır bir fonksiyon geçilebilir:
  - **`andThen()`** : Önce fonksiyonu sonra geçilen fonksiyon çağrılır.
  - **`compose()`** : Önce geçilen fonksiyon, sonra esas fonksiyon çağrılır.
- Fonksiyonların çalışma sırası açısından **`compose()`**, **`andThen()`**'in tersidir.

FunctionComposition.java

- *org.javaturk.oofp.ch06.functions.composition* paketi.

ComparatorComposition.java

- *org.javaturk.oofp.ch06.functions.composition* paketi.

andThen() Bileşimi

- Şu 10 fonksiyonel arayüzde **`andThen()`** metodu vardır:
  - **`Function<T, U>`**
  - **`BiFunction<T,U,R>`**
  - **`Consumer<T>`**
  - **`BiConsumer<T,U>`**
  - **`IntConsumer`**
  - **`LongConsumer`**
  - **`DoubleConsumer`**
  - **`UnaryOperator<T>`**, Function'un alt sınıfıdır.
  - **`IntUnaryOperator`**
  - **`LongUnaryOperator`**
  - **`DoubleUnaryOperator`**

compose() Bileşimi

- Şu 4 fonksiyonel arayüzde **`compose()`** metodu vardır:
  - **`Function<T, U>`**
  - **`UnaryOperator<T>`**, Function'un alt sınıfıdır.
  - **`IntUnaryOperator`**
  - **`LongUnaryOperator`**
  - **`DoubleUnaryOperator`**

BiPredicate<T, U>

- **`BiPredicate<T,U>`**'da aşağıdaki default metotlar vardır:
  - **`and(BiPredicate<? super T,? super U> other)`**: Bu ve geçilen **`BiPredicate`** fonksiyonu arasında kısa devre mantıksal VE çalıştıran bileşik fonksiyon üretir.
  - **`or(BiPredicate<? super T,? super U> other)`**: Bu ve geçilen **`BiPredicate`** fonksiyonu arasında kısa devre mantıksal VE çalıştıran bileşik fonksiyon üretir.
  - **`BiPredicate<T, U> negate()`**: Bu **`BiPredicate`** fonksiyonunun değilini veren fonksiyonu üretir.

DoublePredicate

- **`DoublePredicate`**'de aşağıdaki default metotlar vardır:
  - **`and(DoublePredicate other)`**: Bu ve geçilen **`DoublePredicate`** fonksiyonu arasında kısa devre mantıksal VE çalıştıran bileşik fonksiyon üretir.
  - **`or(DoublePredicate other)`**: Bu ve geçilen **`DoublePredicate`** fonksiyonu arasında kısa devre mantıksal VE çalıştıran bileşik fonksiyon üretir.
  - **`DoublePredicate negate()`**: Bu **`DoublePredicate`** fonksiyonunun değilini veren fonksiyonu üretir.

BiAndDoublePredicateExample.java

- *org.javaturk.oofp.ch06.functions* paketi.

Metot Referansları

Metot Referansları - I

- Lambda ifadeleri bazen sadece var olan metotları çağırırlar.
- Böyle hallerde metot çağrısı yapmak yerine çağrılacak metodu göstermek daha kısa ve anlaşılır olabilir.
- Bu durumda "**()**" ile tip ve parametre bilgisi ve "**->**" kullanılmaz, atama operatörü "**=**" kullanılır.
- Metotlara ulaşmak için "**::**" operatörü kullanılır.
- Bu gösterime metot referansı (method reference) denir.

```
Consumer<String> print1 = (s1) -> System.out.println(s1);
print1.accept("Hey, what's up?");

Consumer<String> print2 = System.out::println;
print2.accept("Hey, what's up?");
```

Metot Referansları - II

- Metot referanslarında dört türlü metot da ifade edilebilir:
  - Statik metotlar,
  - Özel bir nesnenin metotları,
  - Bir tipten herhangi bir nesnenin metotları,
  - Kurucu metotlar.
- Bu metotların ifade şekilleri aşağıda verilmiştir.

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`containingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

MethodReferences.java

- *org.javaturk.oofp.ch06.functions* paketi.

En İyi Pratikler

En İyi Pratikler - I

- Java'da fonksiyonel programlama için bazı en iyi pratiklerden (best practices) bahsedilebilir:
  - Yerel olmayan değişkenlerin değiştirilmesini sınırlayın,
  - Mümkün olan her yerde metotları, saf fonksiyon olarak yazın,
  - Fonksiyonel arayüz oluşturmadan önce hazır arayüzleri araştırın, kullanabiliyorsanız onları tercih edin.
  - `@FunctionalInterface` daima kullanın, sizi hatadan korur.

En İyi Pratikler - II

- Lambda ifadelerini olabildiğince kısa yazın.
  - Blok yerine tek bir ifadeyi tercih edin,
  - Bu durumda tip ve parametre ismini de yazmayın.
- Mümkün olan her yerde metot referanslarını kullanın.
- Lambda ifadelerinde olabildiğince final ve effective final değişkenleri kullanın.