

MODÜL - 2  
**İLERİ JAVA VE  
VERİTABANI**

**PAKET - 4  
İLERİ JAVA**



**BÖLÜM-1 FONKSİYONEL PROGRAMLAMA**

**T E C H P R O E D**

# PROGRAMLAMA PARADİGMALARI

- **Java8'den önce Java, 3 programlama paradigmalarını** desteliyordu.
  - Prosedürel(Yapısal) Programlama
  - Nesne Yönelimli Programlama (OOP)
  - Jenerik (Generic) programlama
- **Java8 ile birlikte Fonksiyonel programlama** özelliği sınırlı da olsa Java diline girmiş oldu.
  - Problem çözümünü **matematiksel fonksiyonların** üzerinden gerçekleştiren programlama stili.



**imperatif Programlama:** Belirli adımlar üzerinden ilerleyen ve akış kontrolüne dayanan programlama stili.

**Declarative:** Akış kontrolüne odaklanmadan neyin yapılacağıının tanımlandığı programlama stili.

T E C H P R O E D

# FONKSİYONEL PROGRAMLAMA

Java'da 1 den 10 kadar sayıların toplamı

```
int toplam = 0;  
for (int i = 1; i ≤ 10; i++)  
    toplam += i;
```

Hesaplama metodu **değişken ataması**

1 den 10 kadar tamsayıların toplamı  
Fonksiyonel yaklaşım (Haskell)

```
sum [1..10]
```

Hesaplama Metodu **Fonksiyon uygulaması**

T E C H P R O E D

# NEDEN FONKSİYONEL PROGRAMLAMA?

- Imperatif programlamaya göre daha anlaşılır ve daha açıklayıcı program yazmaya imkan tanır.
- Kod yazmaktan ziyade probleme odaklanmaya izin verir.
- Paralel hesaplamaya daha uygundur.
- Yan etkileri azaltır. (Verilerin değiştirilemez gibi davranışını sağlar)

T E C H P R O E D

# FONKSİYON NEDİR?

- **Matematiksel fonksiyon**

İsim

$$f(x) = x^2$$

Girdi

Cıktı

Bir fonksiyon her bir girdiye ait tek bir çıktıının olduğu matematiksel bir ifadedir.

X	f(x)
3	9
1	1
0	0
4	16
-4	16

# FONKSİYONEL PROGRAMLAMA VE LAMBDA

- Fonksiyonel Programlamanın temeli **Lambda Hesaplamasına** dayanmaktadır.
  - **Lambda Hesaplama** teorisi, 1930 yılında **Alonzo Church** tarafından geliştirildi.
- **Lambda hesaplama** teorisi, **Lisp**, **Haskell**, **Ocaml** ve **C++** gibi fonksiyonel dillerin hesaplama modeline esas teşkil etti.
- Sonunda, **Java8** ile birlikte Java diline de eklenmiş oldu.



T E C H P R O E D

# LAMBDA İFADELERİ

- Lambda ifadeleri:
  - **Parametrelere ve bir vücuda sahip, isimsiz** fonksiyonlar.

(parameterler) -> {body}

Lambda  
Syntax

- **Lambda** operatörü: ->

T E C H P R O E D

# LAMBDA İFADELERİ

- . Bir Lambda ifadesi **0 veya daha fazla** sayıda parametre alabilir ve parametreler **virgülle** ayrılır
  - **(int x, int y) -> { return x + y; }** → **explicit**
  - **x -> x \* x** → **implicit**
  - **() -> x**
- . Parametre tipleri açıkça (**explicit**) tanımlanabileceği gibi örtülü (**implicit**) olarak da tanımlanabilir. Derleyici context'e bakarak veri tipini anlayabilir.
- . Eğer parametre tek ise etrafında bir parantez şart değildir.
- . **( )** Sıfır parametre için kullanılır.
- . Süslü parantezler tek bir ifade için şart değildir.

T E C H P R O E D

# LAMBDA İFADELERİ NASIL ÇALIŞIR?

- . Aslında, Java Derleyicisi ilk olarak **Lambda** ifadelerini bir fonksiyona çevirmektedir. Sonrasında da çevrilen bu fonksiyonu çağrılmaktadır.

- . Örneğin:

**x -> System.out.println(x)**

- . Derleyici bu Lambda ifadesini bir static fonksiyona çevrilebilir.

```
public static void genName(Integer x) {  
    System.out.println(x);  
}
```

T E C H P R O E D

# LAMBDA BİR NESNE MİDİR?

- Tam olarak değildir. (Kimliksiz Nesnedir)
- **Object** sınıfından türetilememiştir. Dolayısıyla .equals(), .hashcode(), v.b metotları desteklemez.
- Lambda ifadelerinde **new** anahtar kelimesi kullanılamaz.
- **this** anahtar kelimesini de desteklemez.

T E C H P R O E D

# JAVADA NEREDE VE NASIL KULLANILABİLİR?

- Maalesef, Java Fonksiyonel programlamayı %100 desteklemediği için Lambda ifadelerini her durumda kullanmak **mümkün değil**.
- Java'da **Lambda ifadeleri** genelde **Fonksiyonel Arayüz (Functional Interface)** içerisinde kullanılmaktadır.
  - **SADECE** bir tek **abstract (soyut)** metodu olan arayüzlerdir.
  - default metodlara sahip olabilir.
- Sadece tek metot şartı, Lambda ifadelerine uygun olmasını sağlamak içindir.

**NOT:** Javada metodlar kendi başlarına bir eleman değildir. Metodlar, diğer bir dil elemanın (Sınıf, enum, interface ..) içinde tanımlanırlar. Bu sebeple, Java ortamında, bir Lambda ifadesinden bir metoda dönüşüm için **fonksiyonel arayüzlerin** kullanılması tercih edilmiştir.

TECHPROED

# FONKSİYONEL ARAYÜZLER

- Java'da Kullanıcı, **kendi Fonksiyonel arayüzlerini tanımlayabilir.**
- Ancak, Java8 içerisinde 4 kategoride toplam 43 adet **hazır** arayüz tanımlanmıştır.
  - Kategori isimleri: **Function, Supplier, Consumer, Predicate**
  - Kütüphane: **java.util.function**
  - Java8 öncesinde olan bazı arayüzler de vardır. (**Runnable** gibi)
- **Consumer<T>**      ==> **void** tipinde bir fonksiyonel arayüzdür.
- **Supplier<T>**      ==> **parametre almayan** fonksiyonel arayüzdür.
- **Predicate<T>**      ==> Bir **şartın** değerlendirilmesini sağlayan fonksiyonel arayüzdür.
- **Function<T,R>**      ==> **T tipinde** parametre alan, **R tipinde** sonuç döndüren arayüzdür.
- **BiFunction<T, U, R>** ==> **2 parametre** alıp bir sonuç döndüren arayüzdür.

T E C H P R O E D

# FONKSİYONEL ARAYÜZ KULLANIM ÖRNEKLERİ

- Javada **fonksiyonel arayüzlerin** kullanıldığı bazı **yüksek seviyeli fonksiyonlar** (High Order Function-HOF) bulunmaktadır.
  - **HOF:** Bir fonksiyonu (metodu) parametre olarak alabilen ve / veya sonuçları bir fonksiyon olarak döndürebilen fonksiyonlardır.

Fonksiyonel Arayüz	Kullanım Örneği (HOF)
Consumer	<code>forEach(Consumer), peek(Consumer)</code>
Predicate	<code>filter(Predicate)</code>
Function	<code>map(Function)</code>
Supplier	<code>reduce(Supplier), collect(Supplier)</code>

- Javada fonksiyonel programlama genel olarak bu gibi fonksiyonlar yardımıyla yapılmaktadır.

T E C H P R O E D

# LAMBDA ÖRNEKLERİ-1

- **forEach** fonksiyonu ile lambda ifadelerini kullanabiliriz.

## Bir Listedeki verileri yazdıralım

```
List<Integer> sayılar = Arrays.asList(1,2,3);  
sayılar.forEach(x -> System.out.println(x));
```

```
sayılar.forEach(x -> {  
    x += 2;  
    System.out.println(x);  
});
```



İfade sayısı birden fazla olduğu için  
{ } kullanmalıyız.

```
sayılar.forEach((Integer x) -> {  
    x += 2;  
    System.out.println(x);  
});
```



İstersek parametrenin veri tipini  
belirtebiliriz.

T E C H P R O E D

# LAMBDA ÖRNEKLERİ-1

```
sayılar.forEach(x -> {  
    int y = x * 2;  
    System.out.println(y);  
});
```



Lambda ifadesinde **lokal değişken** kullanılabilir.

```
int dışDeğişken = 5;  
sayılar.forEach(x -> System.out.println(x + dışDeğişken));
```



İfade dışı parametre de kullanabiliriz. Ancak bu değişken **final** gibi düşünülmelidir.

```
sayılar.forEach(System.out::println);
```



Daha kısa kullanım için  
**Metot Referansı**  
tercih edilebilir.

T E C H P R O E D

# METOT REFERANSININ KULLANIMI

- **Metot referansı**, bir **Lambda** ifadesine, var olan bir fonksiyonu argüman olarak geçirmek için kullanılabilir.
- Referans gösterilen metotodun imzası, fonksiyonel arayüz metotodunun imzası ile örtüşmelidir.

Metot Referans Türü	Syntax	Örnek
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

**EK BÖLÜM:**  
**FONKSİYONEL ARAYÜZLER İÇİN**  
**DETAYLI ÖRNEKLER**

**T E C H P R O E D**

# FONKSİYONEL ARAYÜZLER

Kendi Fonksiyonel Arayüzümüzü yazabiliriz.

```
@FunctionalInterface  
interface MesajVer{  
    void merhaba(String mesaj);          // abstract metot  
    default void bye(){                  // Default metot  
        System.out.println("Güle Güle");  
    }  
}  
  
public class FonksiyonelArayuzOrnek {  
    public static void main(String[] args) {  
        MesajVer mesajVer = (msg) -> System.out.println(msg);  
        mesajVer.merhaba ("Merhaba fonksiyonel arayuzler.");  
        mesajVer.bye();  
    }  
}
```

## NOT:

@FunctionalInterface anotasyonunu kullanmak zorunlu değildir.

Kullanılması durumunda anlaşılması kolaylaştırır.

Ayrıca, arayüze 2. bir abstract metot eklenmesi durumunda derleme hatası verilmesini sağlar.

T E C H P R O E D

# FONKSİYONEL ARAYÜZLER - FUNCTION

## Function Fonksiyonel Arayüz Örneği

```
public static void main(String[] args) {  
    System.out.print("Virgülle ayrılmış olarak sayıları giriniz:");  
    Scanner scanner = new Scanner(System.in);  
    String[] giriş = scanner.nextLine().split(",");  
  
    Function<String, Integer> çevirici = x -> Integer.parseInt(x);  
    int toplam = 0;  
    for (String s : giriş) {  
        toplam += çevirici.apply(s);  
    }  
  
    System.out.println("Sayısı = " + giriş.length);  
    System.out.println("Toplamı = " + toplam);  
}
```

**NOT: Function**  
arayüzünde  
tanımlanan  
fonksiyon  
.apply() metodu  
çAĞRıLAbiLir.

T E C H P R O E D

# FONKSİYONEL ARAYÜZLER - PREDICATE

## Predicate Fonksiyonel Arayüz Örneği

```
public class PredicateFonksiyonelArayüzü {  
    public static void main(String[] args) {  
        Predicate<Kişi> predicate = (kişi) -> kişi.getYaş() > 18;  
        boolean sonuç = predicate.test(new Kişi("Yusuf", 20));  
        System.out.println(sonuç);  
    }  
}
```

**NOT:** **Predicate** arayüzünde fonksiyon **.test()** metodu ile çağrılabılır.

```
public class Kişi {  
    private String isim;  
    private int yaşı;  
    public Kişi(String isim, int yaşı) {  
        this.isim = isim;  
        this.yaş = yaşı;  
    }  
    public String getIsim() {  
        return isim;  
    }  
    public void setIsim(String isim) {  
        this.isim = isim;  
    }  
    public int getYaş() {  
        return yaşı;  
    }  
    public void setYaş(int yaşı) {  
        this.yaş = yaşı;  
    }  
}
```

T E C H P R O E D

# FONKSİYONEL ARAYÜZLER - SUPPLIER

## Supplier Fonksiyonel Arayüz Örneği

```
public class SupplierFonksiyonelArayüzü {  
  
    public static void main(String[] args) {  
        Supplier<Kişi> supplier = () -> new Kişi("Hasan", 30);  
  
        Kişi kişi = supplier.get();  
        System.out.println("Kişi Detayı:\n" + kişi.getİsim() + ", " + kişi.getYaş());  
    }  
}
```

**NOT:** **Supplier** arayüzünde fonksiyon, **.get()** metodu ile çağrılabılır.

# FONKSİYONEL ARAYÜZLER - CONSUMER

## Consumer Fonksiyonel Arayüz Örneği

```
public class ConsumerFonksiyonelArayüzü {
    public static void main(String[] args) {
        List<Kişi> kişilerListesi = new ArrayList<Kişi>();
        kişilerListesi.add(new Kişi("Ali", 27));
        kişilerListesi.add(new Kişi("Veli", 26));

        Consumer<Kişi> consumer = (k) -> {
            System.out.println(k.getİsim());
            System.out.println(k.getYaş());
        };
        consumer.accept(new Kişi("Ayşe", 22));
        consumer.accept(new Kişi("Mehmet", 45));
    }
}
```

**NOT:** Consumer arayüzünde fonksiyon, **.accept()** metodu ile çağrılabılır.

# FONKSİYONEL ARAYÜZLER - BIFUNCTION

## BiFunction Fonksiyonel Arayüz Örneği

```
public class BiFunctionFonksiyonelArayüzü {  
    public static void main(String[] args) {  
  
        BiFunction<Kişi, Kişi, Integer> biFunction = (k1, k2) -> k1.getYaş() + k2.getYaş();  
        int toplamYaş = biFunction.apply(new Kişi("John", 40),  
                                         new Kişi("Jeniffer", 20));  
        System.out.println(toplamYaş);  
    }  
}
```

**NOT:** **BiFunction** arayüzünde fonksiyon, **.apply()** metodu ile çağrılabılır.