



# Java ile Nesne Merkezli ve Fonksiyonel Programlama

## 9. Bölüm: Java'da Fonksiyonel Programlama



Eğitmen:

**Akın Kaldıroğlu**

Çevik Yazılım Geliştirme ve Java Uzmanı

```
121     Customer unfoundCustomer = service.login(unfoundCustomerTckn, proper
122     });
123 }
124
125 @Test // Test 1
126 public void testUnsuccessfulLoginWithCustomerLockedException() {
127     assertThrows(CustomerLockedException.class, () -> {
128         Customer lockedCustomer = service.login(lockedCustomerTckn, "qwerty");
129     });
130 }
131
132 @Test // Test 2
133 public void testUnsuccessfulLoginWithCustomerAlreadyLoggedInException() {
134     assertThrows(CustomerAlreadyLoggedInException.class, () -> {
135         Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomer
136     });
137 }
138
139 @Test // Test 3
140 public void testUnsuccessfulLoginWithImproperCustomerCredentialsExceptionFor
141     assertThrows(ImproperCustomerCredentialsException.class, () -> {
142         Customer customer = service.login(shortTckn, properPassword);
143     });
144 }
145
146 @Test // Test 4
147 public void testUnsuccessfulLoginWithImproperCustomerCredentialsExceptionFor
148     assertThrows(ImproperCustomerCredentialsException.class, () -> {
149         Customer customer = service.login(properTckn, properPassword);
150     });
151 }
152
153 @Test // Test 5
154 public void testUnsuccessfulLoginWithImproperCustomerCredentialsExceptionFor
155     assertThrows(ImproperCustomerCredentialsException.class, () -> {
156         Customer customer = service.login(properTckn, shortPassword);
157     });
158 }
159
160 @Test // Test 6
161 public void testUnsuccessfulLoginWithImproperCustomerCredentialsExceptionFor
162     assertThrows(ImproperCustomerCredentialsException.class, () -> {
163         Customer customer = service.login(properTckn, shortPassword);
164     });
165 }
```



# Konular

- **Java'da Fonksiyonel Programlama**
  - İsimsiz Sınıflar
  - Lambda İfadeleri
  - Lambda İfadelerinin Özellikleri
  - Lambda İfadeleri ve Sıra Dışı Durumlar
  - **Hazır Fonksiyonel Arayüzler**
- Hazır Arayüzlerde Fonksiyon Bileşimleri
  - Metot Referansları
- **Nesne-Merkezli ve Fonksiyonel Programlama**
  - Açıklayıcı Programlama
  - **En İyi Uygulamalar**
  - **Ödevler**

# Java'da Fonksiyonel Programlama



```
111 public void testSuccessfullLogin() {  
112     assertEquals(CustomerNotFoundException.class, customer.  
113         login("customer", "password").getClass());  
114     assertEquals("Customer", customer.getCustomerName());  
115     assertEquals(true, customer.isLogged());  
116 }  
117  
118 @Test // Test 1  
119 public void testSuccessfullLoginWithNoSuchCustomerException() {  
120     assertEquals(NoSuchCustomerException.class, () -> {  
121         customer.unfoundCustomer = service.login("customer", "password");  
122     }).get();  
123 }  
124  
125 @Test // Test 2  
126 public void testSuccessfullLoginWithCustomerLockedException() {  
127     assertEquals(CustomerLockedException.class, () -> {  
128         customer.lockedCustomer = service.login("lockedCustomer", "password");  
129     }).get();  
130 }  
131  
132 @Test // Test 3  
133 public void testSuccessfullLoginWithCustomerAlreadyLoggedException() {  
134     assertEquals(CustomerAlreadyLoggedException.class, () -> {  
135         customer.alreadyLoggedCustomer = service.login("alreadyLoggedCustomer", "password");  
136     }).get();  
137 }  
138  
139 @Test // Test 4  
140 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
141     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
142         customer = service.login("shortTckn", "properPassword");  
143     }).get();  
144 }  
145  
146 @Test // Test 5  
147 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
148     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
149         customer = service.login("properTckn", "shortPassword");  
150     }).get();  
151 }  
152  
153 @Test // Test 6  
154 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
155     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
156         customer = service.login("properTckn", "properTckn");  
157     }).get();  
158 }  
159  
160 @Test // Test 7  
161 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
162     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
163         customer = service.login("properTckn", "properTckn");  
164     }).get();  
165 }
```

# isimsiz Siniflar

# Java'da İsimsız Metotlar - I



- Fonksiyonel programlamada en temel kavramlardan ikisi saf fonksiyonlar (pure functions) ve saf fonksiyonların birinci sınıf vatandaş (first-class citizen) olarak görülmESİYDİ.
- Java'da ise fonksiyon yoktur, metot vardır.
  - Java'nın metodları muhakkak sınıf içinde tanımlanır.
  - Java'nın metodlarına, **static** ise sınıf üzerinden değilse nesneler üzerinden ulaşılır.

# Java'da İsimsiz Metotlar - II



- Java'da metodların bir değişken gibi birbirlerine geçilmeleri mümkün olmadığı gibi isimsiz metot da yoktur.
- Java'da isimsiz metot yoktur ama **isimsiz sınıf (anonymous class)** vardır.
  - Isimsiz sınıf bir **İç sınıf (inner class)** türüdür.
  - Ve eğer bir isimsiz sınıf yapıp içine tek bir metod koyulursa, bu isimsiz sınıfın nesnesi, pratikte isimsiz metoda karşı gelir.

# Geri Çağırma (Call Back) - I



- Yazılım sistemlerinde sıkılıkla, 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.

# Geri Çağırma (Call Back) - II



- Observer (event-notification ya da publisher-subscriber) tasarım kalığı bu problemi ve çözümünü tarif eder.

# Geri Çağırma (Call Back) - III



- Geri çağrımayı Java'da kurmak 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 çağrıracağını bilmesi gereklidir.

# Geri Çağırma (Call Back) - IV



- Bu sebeple Java'da geri çağırma nesneleri, üzerinde genelde bir tane geri çağrıma metodu bulunan arayüzlerden türetilir.
- Olayın kaynağı da bu arayüzü bilir.
- Java'da bu türden arayüzler genelde **listener** olarak adlandırılır.

# TimerExample1.java



- org.javaturk.oofp.ch03.callback.TimerExample1

# İsimsiz Sınıflar - I



- Geri çağırma metodlarının üzerinde bulunduğu sınıfların nesnesine genelde tek kullanımı olarak 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 oluşturulup kullanılmasına izin verir.
- Bu şekilde oluşturulan sınıflara **isimsiz sınıf (anonymous class)** denir.

# İsimsiz Sınıflar - II



- Isimsiz sınıflar sıkılıkla 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.

# TimerExample2.java



- org.javaturk.oofp.ch03.callback.TimerExample2

# 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.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent event) {
        JButton button = (JButton) event.getSource();
        String buttonLabel = button.getText();
        buttonClicked(buttonLabel);
    }
});
```

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

# MyApplication.java



- org.javaturk.oofp.ch04.anonymous.event.MyApplication

# İsimsiz Sınıflar - IV



- Sınıflar tanımdır (class declaration) ama isimsiz sınıflar ifadedir (expression).
- Isimsiz sınıf ifadesi, bir kurucu çağrısına benzer ama içinde tekrar tanımlanan (override) metot ya da metotlar vardır.
- Isimsiz sınıflar hem arayüzleri gerçekleştirmede hem de sınıfları genişletmede kullanılabilir.

# İsimsiz Sınıflar - V



- Isimsiz sınıflar genelde sadece bir metoda sahip arayüzleri gerçekleştirmelerine rağmen birden fazla metodu yeniden tanımlayacak şekilde kullanılabilirler.
- Olay yapılarında çağrılmak üzere bir tane olduğundan, genelde tek metodu tekrar tanımlamada (overriding) kullanılırlar.

# İsimsiz Sınıflar - VI



- İsimsiz sınıf ifadesi söyledir:
  - **new** operatörü,
  - Gerçekleştirilecek arayüzün ya da genişletilecek sınıfın ismi,
  - **new** operatöründen sonra gelen tipin sınıf olması durumunda, kurucuya gelecek parametreler de sıralanabilir.
  - Eğer tip arayüz ise, arayüzlerin kurucuları olmadığından, sanki varsayılan kurucu çağrılmış gibi içi boş iki parantez bulunur.
  - Sınıf bloğu.



- Isimsiz sınıflar birer ifade olduklarından, bloklarında başka ifadeler olamaz, sadece metot gibi başka bloklar olabilir.
- Isimsiz sınıf ifadesi, arayüz gerçekleştirmesinde `new` operatöründen sonra arayüzün varsayılan kurucusunu çağrıyor 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.  
**AnonymousDoerClassTest**



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

# İsimsiz Sınıflar - IX



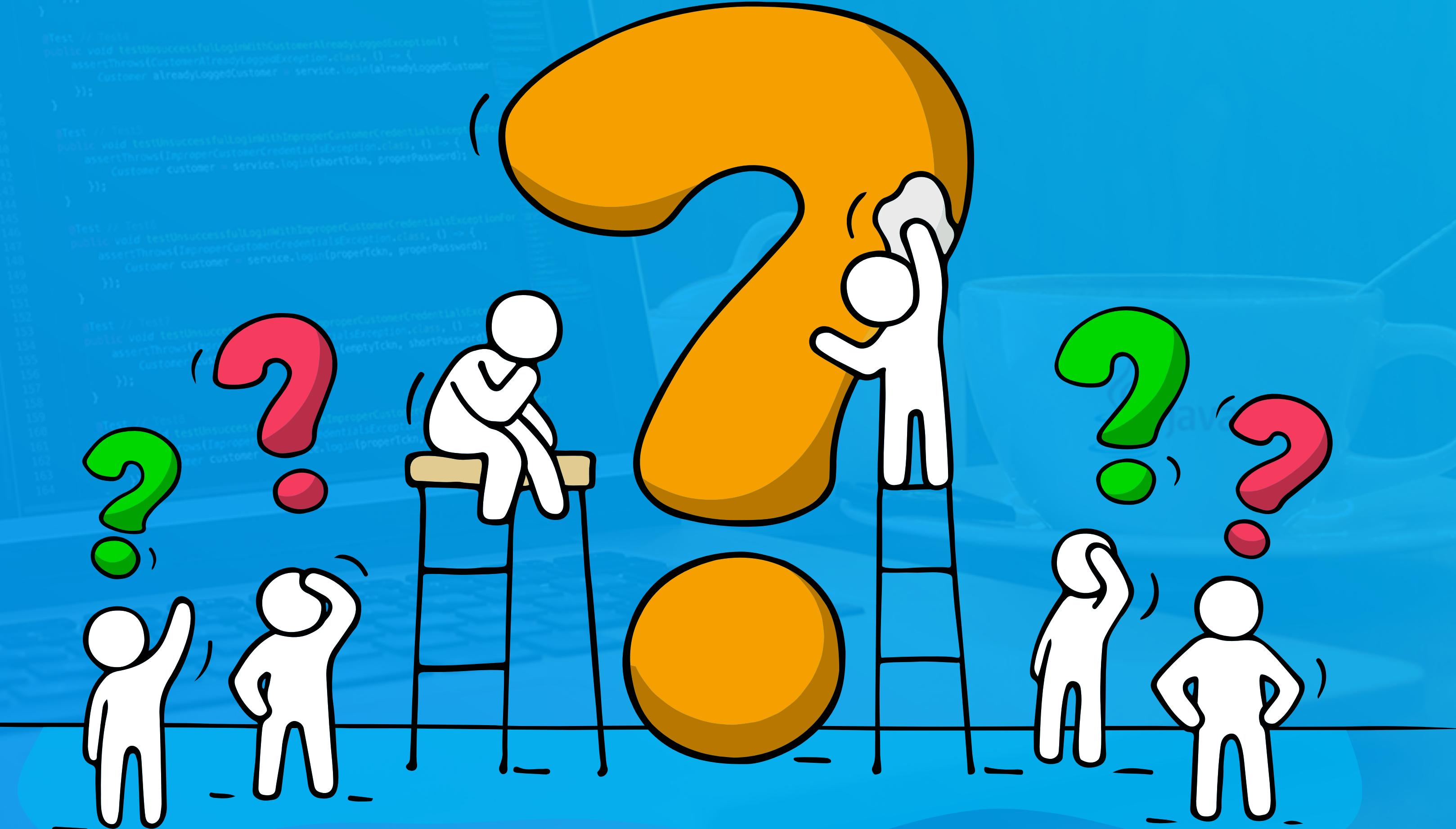
- Isimsiz sınıflar ayrıca şunları tanımlayabilirler:
  - Alanlar,
  - Yerel sınıflar (local classes),
  - Üst tipinde olmayan metodlar,
  - Nesne ilk değer blokları
- Isimsiz sınıflar, statik ilk değer atama blokları ile üye arayüzler tanımlayamazlar.

# WeirdAnonymousDoesClassTest.java



- org.javaturk.oofp.ch04.anonymous.doer.  
WeirdAnonymousDoesClassTest

# Soru ve Cevap Zamani!





```
111 public void testSuccessfullLogin() {
112     assertThrows(NoSuchCustomerException.class, () -> {
113         Customer foundCustomer = service.login(properties, properPassword);
114     });
115 }
116
117 @Test // Test 1
118 public void testSuccessfullLoginWithNoSuchCustomerException() {
119     assertThrows(NoSuchCustomerException.class, () -> {
120         Customer unfoundCustomer = service.login(unfoundCustomerToken, properPassword);
121     });
122 }
123
124 @Test // Test 2
125 public void testSuccessfullLoginWithCustomerLockedException() {
126     assertThrows(CustomerLockedException.class, () -> {
127         Customer lockedCustomer = service.login(lockedCustomerToken, "password");
128     });
129 }
130
131 @Test // Test 3
132 public void testSuccessfullLoginWithCustomerAlreadyLoggedException() {
133     assertThrows(CustomerAlreadyLoggedException.class, () -> {
134         Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomerToken, "password");
135     });
136 }
137
138 @Test // Test 4
139 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {
140     assertThrows(ImproperCustomerCredentialsException.class, () -> {
141         Customer customer = service.login(shortToken, properPassword);
142     });
143 }
144
145 @Test // Test 5
146 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {
147     assertThrows(ImproperCustomerCredentialsException.class, () -> {
148         Customer customer = service.login(propertiesToken, properPassword);
149     });
150 }
151
152 @Test // Test 6
153 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {
154     assertThrows(ImproperCustomerCredentialsException.class, () -> {
155         Customer customer = service.login(propertiesToken, shortPassword);
156     });
157 }
158
159 @Test // Test 7
160 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {
161     assertThrows(ImproperCustomerCredentialsException.class, () -> {
162         Customer customer = service.login(propertiesToken, propertiesToken);
163     });
164 }
```

# Lambda İfadeleri

# Arayüz ve Gerçekleştirmeleri



- Elimizde aşağıdaki `calculate()` metodunda sahip `Math` arayüzü ve bu arayüzü argüman olarak alan `doMath()` metodunun olduğunu düşünelim.
- Arayüzün gerçekleştirmelerinin nesnelerini `doMath()` metoduna geçmek için uzun yol olan `Math` arayüzüni 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(addler, 3, 5);  
  
doMath(new Math(){  
    @Override  
    public double calculate(double arg1, double arg2){  
        return arg1 * arg2;  
    }  
}, 3, 5);
```

# AnonymousMathImplementations.java



- org.javaturk.oofp.ch09.lambda.math.  
**AnonymousMathImplementations**

# Lambda İfadesi ile Arayüz Nesnesi



- **Math** arayüzünün nesnesini, isimsiz sınıf kullanmadan daha basit bir şekilde **Lambda İfadesi** olarak oluşturabiliriz.
- Her lambda ifadesi, **Math** arayüzünün ayrı bir nesnesi olur ve arayüzdeki tek bir metot olan **calculate()** 'e bir gerçekleştirmeye sağlar.

```
Math adder = (double arg1, double arg2) → { return arg1 + arg2;};  
doMath(add, 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);
```



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

# LambdaMathImplementations.java



- org.javaturk.oofp.ch09.lambda.math.  
LambdaMathImplementations

# Lambda İfadesi - I



- Java'da lambda ifadesi (**lambda expressions**), tek bir soyut metoda sahip bir arayüzü gerçekleştiren ifadedir.
- Lambda ifadesi, pratikte **arayüz tipinde bir fonksiyon değerine** (**a function value with an interface type**) karşılık gelir.
- Dolayısıyla tipi vardır ve gerçekleştirdiği arayüzdür.
- Öyle ki bir lambda ifadesi, gerçekleştirdiği arayüz tipinden bir değişkene atanabilir, parametre olarak metoda geçilebilir ya da metottan geri döndürülebilir.

# Lambda İfadesi - II



- Lambda ifadeleri, alternatif olduğu isimsiz sınıflardan çok daha kısa, pratik ve anlaşılırıdır.

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

# Lambda İfadesinin Yapısı - I



- Bir lambda ifadesi şu parçalardan oluşur:
  - Parantez içinde virgül ile ayrılmış formal parametre listesi.
    - Parametre listesi tabi olarak, lambda ifadesinin gerçekleştirdiği arayüzdeki metodun parametreleriyle aynı sayıda ve uyumlu tipte olmalıdır.
    - Parametreler otomatik olarak yükseltilebilen tipte olabilir.

```
(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



- Ok,  $\rightarrow$  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 - III



- İstenirse parantez içindeki parametre listesinde tipler düşürülebilir.
- Eğer tek bir parametre varsa, parantezler de () düşürülebilir.

```
(int i) → System.out.println(i);  
i → System.out.println(i);  
  
i → i > 0;
```

# Lambda İfadesinin Yapısı - IV



- Bir lambda ifadesinde `->`'den sonra blok `{ }` yok ise ve dönüş tipi varsa, JVM ilk ifadenin sonucunu geri döndürecek.
  - Bu durumda **return** gerekli değildir.
- Metodun döndürdüğü bir tip varsa ve blok kullanılıyorsa bu durumda **return** zorunludur.
- Geri dönüş değeri yoksa bu durumda ya tek olan ifade ya da varsa blok çalıştırılacaktır.

# Lambda İfadesinin Yapısı - IV



- 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.ch09.lambda.LambdaVariations1

# Fonksiyonel Arayüz - I



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

# Fonksiyonel Arayüz - II



- Fonksiyonel arayüzlerde **default** ve **static** metodlar olabilir.
- **default** ve **static** metodlarının gerçekleştirmeleri zaten yine fonksiyonel arayüz üzerinde olduğundan, bu metodlar arayüzün “fonksiyonel” olmasına bir zarar vermez.
- Fonksiyonel bir arayüzde bir tek metod olması zorunluluğu nasıl açıklanabilir?

# FunctionallInterface



- Fonksiyonel olması beklenen arayüzlere birden fazla metot ekleme hatasına düşmemek için, o arayüzün fonksiyonel olması gereği `java.lang.FunctionalInterface` notu (annotation) ile ifade edilebilir.
- Zorunlu değildir, bilgi vermek ve hatadan korumak için kullanılır.
- Fonksiyonel olması beklenen arayüzde `@FunctionalInterface` kullanılması, 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.
  - `java.util.Comparator`
- 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 olarak yazabilirsiniz.

# ComparatorLambda.java



- org.javaturk.oofp.ch09.functions.other.  
ComparatorLambda

# Lambda İfadesi: Birinci Sınıf Vatandaş



- Lambda ifadeleri birinci sınıf vatandaştır, 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.ch09.lambda.LambdaVariations2

# 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.

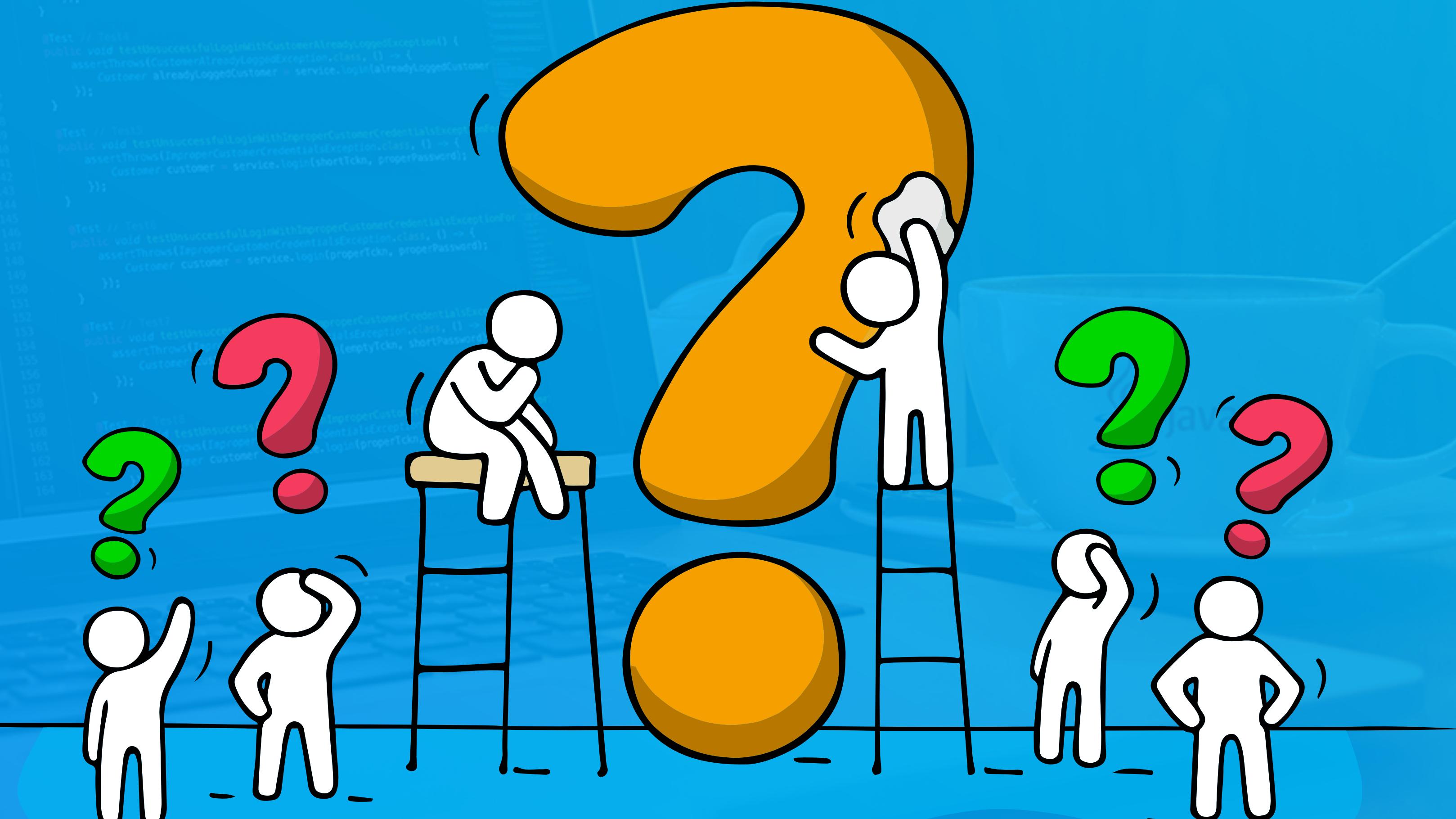
```
@FunctionalInterface  
public interface Selamlama{  
  
    void selamla(String kim);  
}
```

# Uygulama



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

# Soru ve Cevap Zamani!





```
public void testSuccessfulLogin() {
    assertThrows(NoSuchCustomerFoundException.class, () -> {
        Customer loggedCustomer = service.login(unfoundCustomerTkn, properPassword);
        assertEquals(successfulCustomer, loggedCustomer);
    });
}

@Test // Test2
public void testUnsuccessfulLoginWithNoSuchCustomerException() {
    assertThrows(NoSuchCustomerFoundException.class, () -> {
        Customer unfoundCustomer = service.login(unfoundCustomerTkn, properPassword);
    });
}

@Test // Test3
public void testUnsuccessfulLoginWithCustomerLockedException() {
    assertThrows(CustomerLockedException.class, () -> {
        Customer lockedCustomer = service.login(lockedCustomerTkn, "over18");
    });
}

@Test // Test4
public void testUnsuccessfulLoginWithCustomerAlreadyLoggedException() {
    assertThrows(CustomerAlreadyLoggedException.class, () -> {
        Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomerTkn, properPassword);
    });
}

@Test // Test5
public void testUnsuccessfulLoginWithImproperCustomerCredentialsException() {
    assertThrows(ImproperCustomerCredentialsException.class, () -> {
        Customer customer = service.login(shortTkn, properPassword);
    });
}

@Test // Test6
public void testUnsuccessfulLoginWithImproperCustomerCredentialsExceptionForShortTkn() {
    assertThrows(ImproperCustomerCredentialsException.class, () -> {
        Customer customer = service.login(properTkn, properPassword);
    });
}

@Test // Test7
public void testUnsuccessfulLoginWithImproperCustomerCredentialsExceptionForShortPassword() {
    assertThrows(ImproperCustomerCredentialsException.class, () -> {
        Customer customer = service.login(unproperTkn, shortPassword);
    });
}
```

# Lambda ifadelerinin Özellikleri

# Lambda İfadeleri ve Çevresi - I



- Lambda ifadeleri içinde bulundukları kapsamdaki (scope) değişkenlere erişebilir.
- Lambda ifadeleri statik kapsama (static scope) sahiptir.
  - Dolayısıyla gerçekledikleri arayüzden herhangi bir değişken devralamaz,
  - 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 **final** olarak tanıtılsa, sabite olur ve değeri değişmez.
- Eğer bir değişken **final** olarak tanıtılmadığı halde değeri, yeni bir atama ya da `++` veya `--` gibi ön ya da son artırma veya azaltma işlemcileriyle değişmiyorsa, gerçekte **effectively final**'dır yani sabite olarak kalmaya devam ediyor.

# Lambda İfadeleri ve Çevresi - III



- Lambda ifadeleri, içinde bulunduğu kapsamdaki üye değişkenlere yani statik ve nesne değişkenlerine ulaşabilir.
- Bu değişkenlerin `final` ya da `effectively final` olmasına gerek yoktur.
- Üzerinde değişkenlerine ulaştığı nesnenin, yerel değişken olmasından dolayı, `final` ya da `effectively final` olduğunu unutmayın!

# Lambda İfadeleri ve Çevresi - IV



- Görüldüğü gibi lambda ifadeleri, yerel değişkenleri değiştirme konusunda saf fonksiyon olarak davranmakta, kendilerine parametre olarak geçenler dışında hiç bir yerel değişkeni değiştirememektedir.
- Ama aynı durum üye değişkenlerde geçerli değildir, lambda ifadeleri nesne ve sınıf değişkenlerinin değerini değiştirebilir.
- Bu durum fonksiyonel olma özelliğinden bir tavizdir ama kaçınılmazdır:
  - Nesnelerin durumları sıkılıkla değişir, aslolan fonksiyonel yapıları nesnelerle beraber kullanmaktadır.

# LambdaScope.java



- org.javaturk.oofp.ch09.lambda.LambdaScope

# Lambda İfadeleri ve this



- Lambda ifadesinde **this** anahtar kelimesi ile lambda ifadesinin nesnesine değil, ancak kullanıldığı kapsamda (scope) var olan nesneye ulaşılır.
- Dolayısıyla lambda ifadesi bir nesne metodunda tanımlanıyorsa **this**, metodun üzerinde çağrıldığı nesneye ulaşır.
- Lambda ifadesi bir sınıf metodunda tanımlanıyorsa **this** zaten söz konusu değildir, ulaşmaya çalışmak derleme hatasıdır.

# 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.
- Ama tam da bu yüzden lambda ifadeleri, arayüzü üzerindeki **public**, **static** ve **final** olan durum bilgisi ile varsayılan (**default**) metotlara ulaşır.

# Lambda İfadeleri ve Arayüzleri - II



- Ama bir lambda ifadesi içinde kendi arayüzünde tanımlanan varsayılan (`default`) metoda erişilemez, erişmeye çalışıldığında derleyici arayüzdeki metodу tanımayacaktır, `this` kullanılırsa o da çalışmayaçaktır.
- Ancak lambda ifadesinin atandığı referans üzerinde varsayılan metotlara erişilebilir.
- Lambda ifadeleri `static` metotlara ulaşamaz çünkü `static` metodlar sadece ve sadece arayüz üzerinden çağrılabılır, alt arayüzler tarafından bile devralınamazlar.

# Lambda İfadeleri ve Arayüzleri - III



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

# LambdaProperties.java



- org.javaturk.oofp.ch09.lambda.LambdaProperties

# Lambda İfadeleri ve Hedef Tip - I



- Bir lambda ifadesinin tipinin, gerçeklediği fonksiyonel arayüz olduğunu belirtmiştık, bu tipe **hedef tip (target type)** denir.
- Hedef tip, lambda ifadesinin bir parçası değildir dolayısıyla sadece lambda ifadesine bakarak tipini anlamak mümkün değildir.
- Örneğin aşağıdaki lambda ifadesi, hiç bir argüman almayan ve **String** döndüren bir metoda sahip herhangi bir fonksiyonel arayüz tipinde olabilir.

`() → “Done!”;`

- Dolayısıyla derleyici bir lambda ifadesinin tipini nasıl belirler?

# Lambda İfadeleri ve Hedef Tip - II



- Java'da bir lambda ifadesi daima hedef tipiyle birlikte bir bağlamda kullanılır.

```
@FunctionalInterface  
public interface DoneFactory{  
    String createDone();  
}
```

- Ya da bir başka ifadeyle lambda ifadesi daima bir hedef tipe sahiptir, yani lambda ifadesi
  - ya hedef tipinden bir değişkene atanır,

```
DoneFactory factory = () → "Done!";
```

# Lambda İfadeleri ve Hedef Tip - III



- ya hedef tipinden parametre alan bir metoda/kurucuya geçilir,

```
void call(DoneFactory factory){  
    String s = factory.createDone();  
    System.out.println(s);  
}
```

- ya da hedef tipinden dönüşe sahip bir metottan geri döndürülür.

```
DoneFactory getDoneFactory(){  
    return () → “Done!”;  
}
```

# Lambda İfadeleri ve Hedef Tip - IV



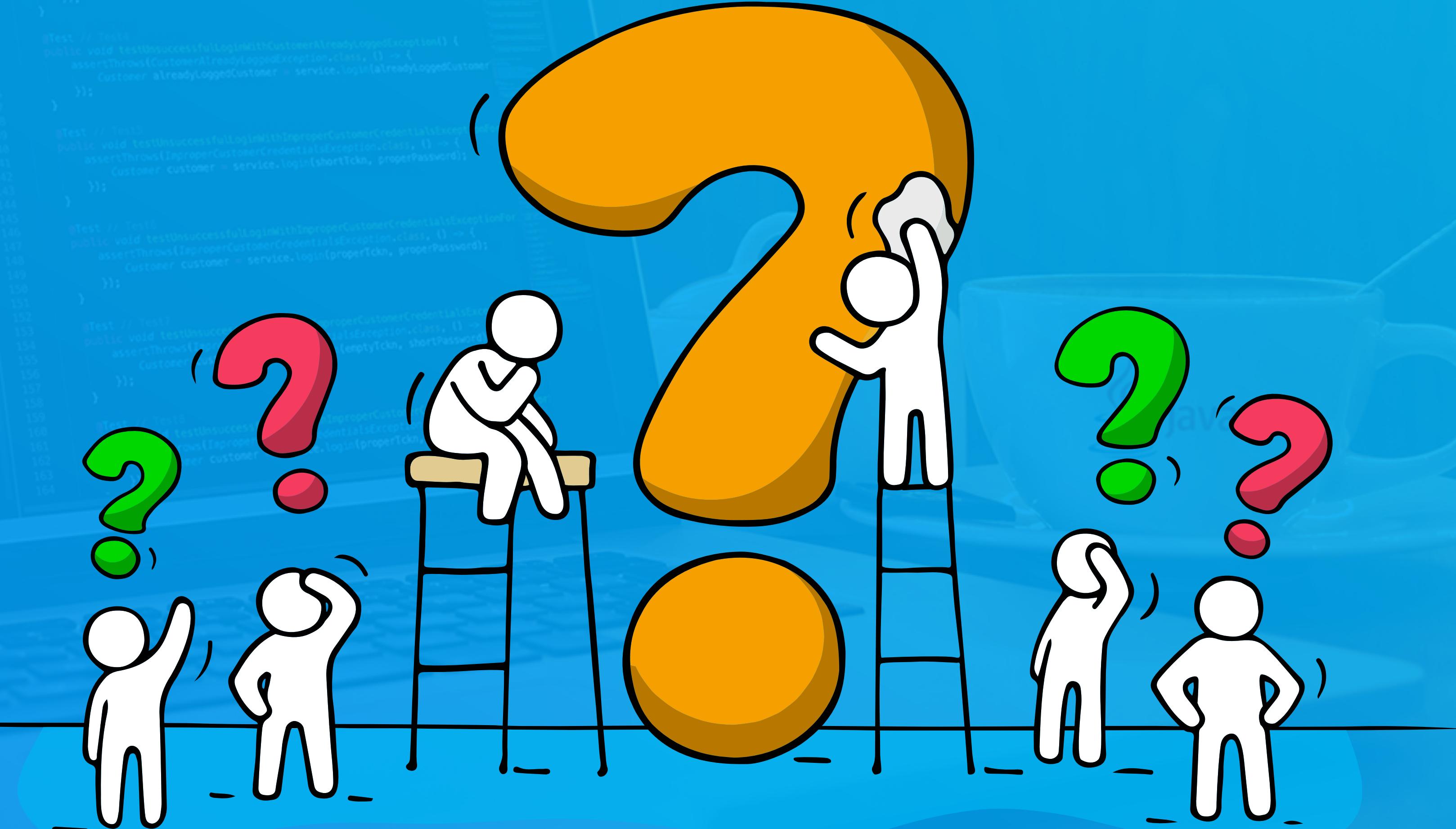
- Bu konuda daha fazla alternatif vardır ama şu an için bu kadar ile yetinelim.
- Detaylı bilgi için: <http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html>

# LambdaTargets.java



- org.javaturk.oofp.ch09.lambda.targeting.  
LambdaTargets

# Soru ve Cevap Zamani!





```
111 public void testSuccessfulLogin() {
112     // Given
113     // When
114     // Then
115     Customer loggedCustomer = service.login(properties, properPassword);
116     assertEquals(successfulCustomer, loggedCustomer);
117 }
118
119 @Test // Test 2
120 public void testSuccessfulLoginWithNoSuchCustomerException() {
121     // Given
122     // When
123     // Then
124     assertThrows(NoSuchCustomerNotFoundException.class, () -> {
125         Customer unfoundCustomer = service.login(unfoundCustomerToken, properPassword);
126     });
127 }
128
129 @Test // Test 3
130 public void testSuccessfulLoginWithCustomerLockedException() {
131     // Given
132     // When
133     // Then
134     assertThrows(CustomerLockedException.class, () -> {
135         Customer lockedCustomer = service.login(lockedCustomerToken, "password");
136     });
137 }
138
139 @Test // Test 4
140 public void testSuccessfulLoginWithCustomerAlreadyLoggedException() {
141     // Given
142     // When
143     // Then
144     assertThrows(CustomerAlreadyLoggedException.class, () -> {
145         Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomerToken, "password");
146     });
147 }
148
149 @Test // Test 5
150 public void testSuccessfulLoginWithImproperCustomerCredentialsException() {
151     // Given
152     // When
153     // Then
154     assertThrows(ImproperCustomerCredentialsException.class, () -> {
155         Customer customer = service.login(shortToken, properPassword);
156     });
157 }
158
159 @Test // Test 6
160 public void testSuccessfulLoginWithImproperCustomerCredentialsSpecificationException() {
161     // Given
162     // When
163     // Then
164     assertThrows(ImproperCustomerCredentialsSpecificationException.class, () -> {
165         Customer customer = service.login(propertiesToken, shortPassword);
166     });
167 }
```

# Lambda ifadeleri ve Sıra Dışı 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 geçebilme (substitutability)** özelliği geçerlidir.
- Lambda ifadesinde fırlatılan sıradışı durum eğer bir unchecked exception ise bunun zaten metodun arayüzünde ifade edilmesine gerek yoktur.

# ExceptionsInLambda.java



- org.javaturk.oofp.ch09.lambda.exception



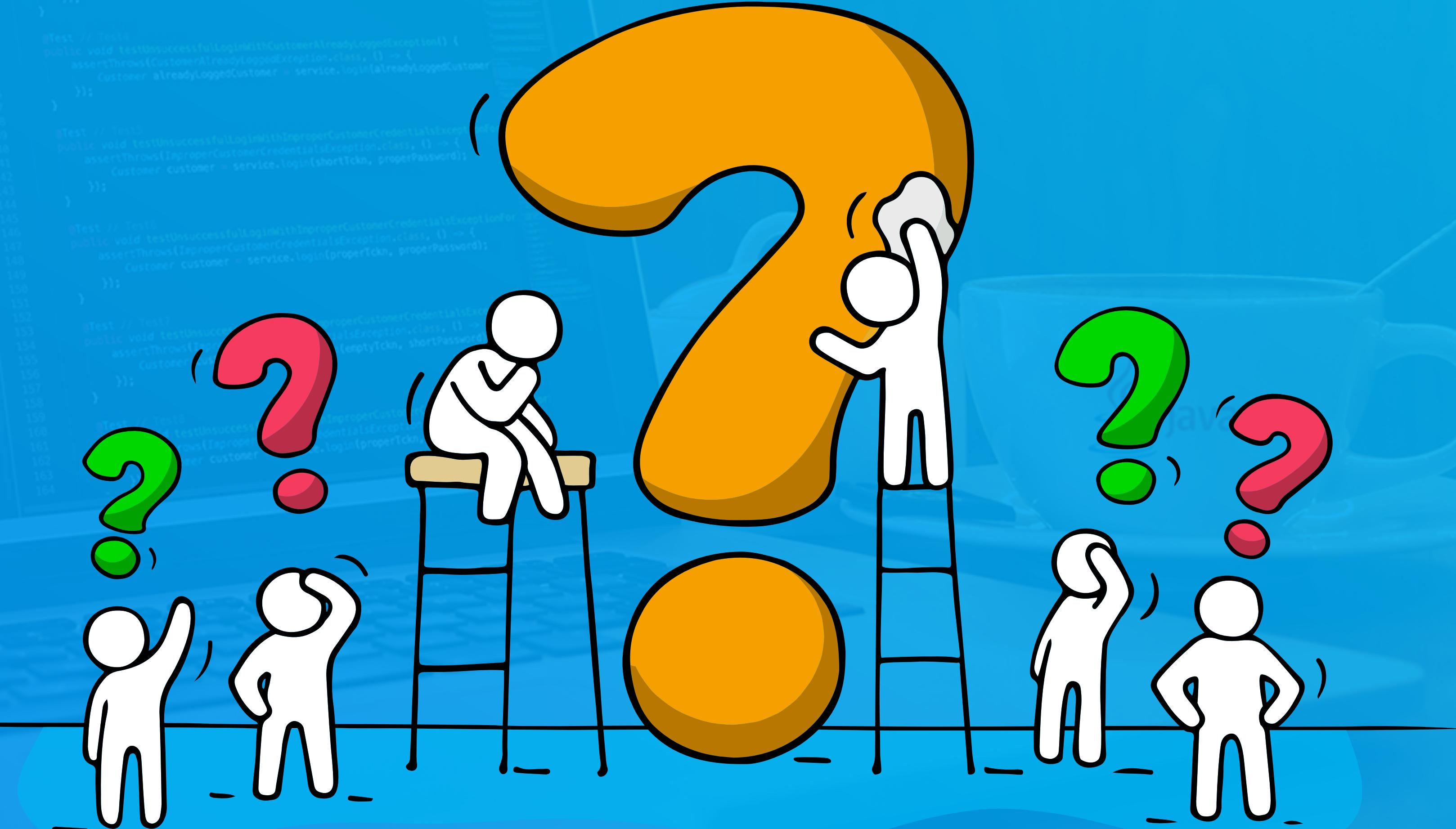
- **org.javaturk.oofp.ch09.examples** paketindeki örnekleri inceleyin.
  - **clock**
  - **doer**
  - **stringAnalyzer**
  - **event**

# Uygulama



- Daha önce Groovy'de closure ile çözdüğümüz problemi Java'da lambda ifadeleriyle nasıl çözersiniz?
- `org.javaturk.oofp.ch09.lambda` paketindeki **EvenNumberOperations** arayüzüne ve **EvenNumberOperationsTest** sınıfına bakıp, test sınıfındaki dört metodu, lambda ifadeleriyle nasıl tek bir metoda indirgeyebileceğinizi düşünün.

# Soru ve Cevap Zamani!



# Hazır Fonksiyonel Arayüzler





```
111 public void testSuccessfulLogin() {
112     assertThrows(NoSuchCustomerException.class, () -> {
113         Customer loggedCustomer = service.login(properties, properPassword);
114     });
115 }
116
117 @Test // Test 1
118 public void testSuccessfulLoginWithNoSuchCustomerException() {
119     assertThrows(NoSuchCustomerException.class, () -> {
120         Customer unfoundCustomer = service.login(unfoundCustomerToken, properPassword);
121     });
122 }
123
124 @Test // Test 2
125 public void testSuccessfulLoginWithCustomerLockedException() {
126     assertThrows(CustomerLockedException.class, () -> {
127         Customer lockedCustomer = service.login(lockedCustomerToken, "password");
128     });
129 }
130
131 @Test // Test 3
132 public void testSuccessfulLoginWithCustomerAlreadyLoggedException() {
133     assertThrows(CustomerAlreadyLoggedException.class, () -> {
134         Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomerToken, "password");
135     });
136 }
137
138 @Test // Test 4
139 public void testSuccessfulLoginWithImproperCustomerCredentialsException() {
140     assertThrows(ImproperCustomerCredentialsException.class, () -> {
141         Customer customer = service.login(shortToken, properPassword);
142     });
143 }
144
145 @Test // Test 5
146 public void testSuccessfulLoginWithImproperCustomerCredentialsException() {
147     assertThrows(ImproperCustomerCredentialsException.class, () -> {
148         Customer customer = service.login(propertiesToken, properPassword);
149     });
150 }
151
152 @Test // Test 6
153 public void testSuccessfulLoginWithImproperCustomerCredentialsException() {
154     assertThrows(ImproperCustomerCredentialsException.class, () -> {
155         Customer customer = service.login(propertiesToken, shortPassword);
156     });
157 }
158
159 @Test // Test 7
160 public void testSuccessfulLoginWithImproperCustomerCredentialsException() {
161     assertThrows(ImproperCustomerCredentialsException.class, () -> {
162         Customer customer = service.login(propertiesToken, propertiesPassword);
163     });
164 }
```

# Genel Fonksiyonel Arayüz

# Genel Fonksiyonel Arayüz - I



- Sıklıkla, aslen aynı yapıda olan fonksiyonel arayüzlerin farklı bağamlarda farklı amaçlarla kullanıldıkları görülür.
- Örneğin aşağıdaki iki arayüzü göz önüne alın:

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

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

- Bu arayüzler bir ve iki `int` argümanla ilgili tüm önermeleri temsil ederler.
- Doğruluk değeri söz konusu olduğundan `boolean` sonuç döndürürler.

# Genel Fonksiyonel Arayüz - II



- Bu arayüzleri hangi amaçlarla farklı lambda ifadeleriyle gerçekleyebilirsiniz?

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

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

# CommonFunctionallInterfaces.java



- org.javaturk.oofp.ch09.functions.  
CommonFunctionalInterfaces

# Genel Fonksiyonel Arayüz - III



- Örneğin pek çok **String** işlemi aşağıdaki arayüzün lambda ifadesi olarak yazılabilir.

```
@FunctionalInterface  
public interface StringAnalyzer {  
    public boolean analyze(String target, String searchStr);  
}
```

# LambdaTest.java



- org.javaturk.oofp.ch09.examples.stringAnalyzer.  
LambdaTest

# Genel Fonksiyonel Arayüz - IV



- Bu örneklerdeki gibi çok yaygın tiplerden parametre alıp yine yaygın tiplerden dönüş yapan (ya da hiç bir şey döndürmeyen) metodlar çok sık kullanılmaktadır.
- Bu türden metodların içinde bulunduğu fonksiyonel arayüzleri farklı amaçlar için sürekli tekrar tekrar oluşturmak gereklidir.
- Projelerde ihtiyaç olduğunda bu türden arayüzlerin bir kere tanımlanması ve herkes tarafından kullanılması çok daha uygun bir çözümüdür.
- Herkes aynı arayüzün farklı lambda ifadelerini farklı amaçlar için yazar.



```
111 public void testSuccessfulLogin() {  
112     assertEquals(CustomerNotFoundException.class, customer.  
113         login("customer", "password").getClass());  
114     assertEquals("Customer", customer);  
115     assertEquals("Customer", customer.getCustomerName());  
116     assertEquals("Customer", customer.getCustomerEmail());  
117     assertEquals("Customer", customer.getCustomerPhone());  
118 }  
119  
120 @Test // Test 1  
121 public void testSuccessfulLoginWithUnfoundCustomer() {  
122     assertEquals(UnfoundCustomerException.class, () -> {  
123         Customer unfoundCustomer = service.login("unfoundCustomer", "proper  
124             password");  
125     }).  
126 }  
127  
128 @Test // Test 2  
129 public void testSuccessfulLoginWithLockedCustomer() {  
130     assertEquals(LockedCustomerException.class, () -> {  
131         Customer lockedCustomer = service.login("lockedCustomer", "proper  
132             password");  
133     }).  
134 }  
135  
136 @Test // Test 3  
137 public void testSuccessfulLoginWithAlreadyLoggedCustomer() {  
138     assertEquals(AlreadyLoggedCustomerException.class, () -> {  
139         Customer alreadyLoggedCustomer = service.login("alreadyLoggedCustomer",  
140             "properpassword");  
141     }).  
142 }  
143  
144 @Test // Test 4  
145 public void testSuccessfulLoginWithImproperCustomerCredentials() {  
146     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
147         Customer customer = service.login("shortTckn", "properpassword");  
148     }).  
149 }  
150  
151 @Test // Test 5  
152 public void testSuccessfulLoginWithImproperCustomerCredentialsAndPassword() {  
153     assertEquals(ImproperCustomerCredentialsAndPasswordException.class, () -> {  
154         Customer customer = service.login("properTckn", "shortPassword");  
155     }).  
156 }  
157  
158 @Test // Test 6  
159 public void testSuccessfulLoginWithImproperCustomerCredentialsAndTckn() {  
160     assertEquals(ImproperCustomerCredentialsAndTcknException.class, () -> {  
161         Customer customer = service.login("properTckn", "properpassword");  
162     }).  
163 }
```

# Hazır Fonksiyonel Arayüzler

# Hazır Fonksiyonel Arayüzler - I



- Java da benzer amaçla, kendi APIsindeki böyle ihtiyaçlara cevap vermek için en temel fonksiyonel arayüzleri `java.util.function` paketinin altında tanımlamıştır.
- Bunlara **hazır fonksiyonel arayüzler (built-in functional interfaces)** denir.
- Az önce belirtilen arayüzleri yazmak yerine biz de bu hazır arayüzleri kullanabiliriz.

# Hazır Fonksiyonel Arayüzler - II



- Java 8'de `java.util.function` paketinde toplam 43 tane hazır fonksiyonel arayüz ya da fonksiyon vardır.
  - Şu ana kadar bir değişiklik de olmamıştır.
- Bu arayüzler aynı zamanda Java API'sinde de sıkılıkla kullanılmaktadır.
- Dolayısıyla, kendi fonksiyonel arayüzünüzü yazmadan önce buraya bakın.

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

# Hazır Fonksiyonel Arayüzler - III



- Her fonksiyonel arayüzde sadece bir tane soyut (**abstract**) metot vardır.
- Bu soyut metoda **fonksiyonel metot (functional method)** denir.
- Lambda ifadesi üzerinde bu metot çağrıılır.
- Fonksiyonel arayüzlerde soyut olmayan, **default** veya **static** metotlar da vardır.
- Bu metotlar bileşim (compositon) vb. amaçlar için konumuştur.

# Hazır Fonksiyonel Arayüzler - IV



- `java.util.function` paketinde 4 tane temel hazır arayüz ve üzerindeki soyut metodlar şunlardır:
  - `Function<T, R>`: `R apply(T t)`
  - `Supplier<T>`: `T get()`
  - `Consumer<T>`: `void accept(T t)`
  - `Predicate<T>`: `boolean test(T t)`

# Hazır Fonksiyonel Arayüzler - V



- Bu 4 temel hazır arayüzün, **Supplier** dışında diğer 3 tanesi için, ismi **Bi** ile başlayan ve aynı işi iki argüman için yapan halleri ve üzerindeki soyut metotlar şunlardır:
  - **BiFunction<T, U, R>**: `R apply(T t, U u)`
  - **BiConsumer<T, U>**: `void accept(T t, U u)`
  - **BiPredicate<T, U>**: `boolean test(T t, U u)`

# Hazır Fonksiyonel Arayüzler - VI



- Ayrıca, **Function<T, R>**'nın alt arayüzü **UnaryOperator<T>** ve **BiFunction<T, R, U>**'nın alt arayüzü **BinaryOperator<T>** vardır.
- **UnaryOperator<T>**, **Function<T, R>**'nın argüman ve dönüş tipi aynı olacak şekilde özelleşmiş hali iken, **BinaryOperator<T>** de **BiFunction<T, R, U>**'nın aynı tipten iki argüman alıp aynı tipten değer döndürecek şekilde özelleşmiş halidir.

# Hazır Fonksiyonel Arayüzler - VII



- **UnaryOperator<T>** ve **BinaryOperator<T>** üzerindeki soyut metodlar şunlardır:
  - **UnaryOperator<T>:** `T apply(T t)`
  - **BinaryOperator<T>:** `T apply(T t1, T t2)`

# Hazır Fonksiyonel Arayüzler - VIII



- En çok kullanılan ve en temel hazır arayüzler ve özellikleri şunlardır:

Hazır Fonksiyon	Foksiyonel Metot	Örnek Kullanım
<b>Function&lt;T, R&gt;</b>	R apply(T t)	Verilen bir tam sayının kare kökünü hesapla ve döndür.
<b>Supplier&lt;T&gt;</b>	T get()	Rastgele (random) bir sayı üret.
<b>Consumer&lt;T&gt;</b>	void accept(T t)	Loglama yap.
<b>Predicate&lt;T&gt;</b>	boolean test(T t)	Verilen sayı çift mi?
<b>UnaryOperator&lt;T&gt;</b>	T apply(T t)	Verilen sayının karesini hesapla ve döndür.

# Hazır Fonksiyonel Arayüzler - VIII



- En çok kullanılan ve en temel hazır arayüzler ve özellikleri şunlardır:

Hazır Fonksiyon	Foksiyonel Metot	Örnek Fonksiyon	Örnek Kullanım
<b>Function&lt;T,R&gt;</b>	<b>R apply(T t)</b>	Function<Integer, Double>	Verilen bir tam sayının kare kökünü döndür.
<b>Supplier&lt;T&gt;</b>	<b>T get()</b>	Supplier<Integer>	Rastgele (random) bir tam sayı üret.
<b>Consumer&lt;T&gt;</b>	<b>void accept(T t)</b>	Consumer<String>	Loglama yap.
<b>Predicate&lt;T&gt;</b>	<b>boolean test(T t)</b>	Predicate<Integer>	Verilen sayı çift mi?
<b>UnaryOperator&lt;T&gt;</b>	<b>T apply(T t)</b>	UnaryOperator<Integer>	Verilen tam sayının karesini döndür.



```
111 public void testSuccessLogin() {
112     // Given
113     // When
114     // Then
115     // ...
116 }
117
118 @Test // Test 1
119 public void testSuccessfulLoginWithNoSuchCustomerException() {
120     // Given
121     // When
122     // Then
123     // ...
124 }
125
126 @Test // Test 2
127 public void testSuccessfulLoginWithCustomerLockedException() {
128     // Given
129     // When
130     // Then
131 }
132
133 @Test // Test 3
134 public void testSuccessfulLoginWithCustomerAlreadyLoggedException() {
135     // Given
136     // When
137     // Then
138 }
139
140 @Test // Test 4
141 public void testSuccessfulLoginWithImproperCustomerCredentialsException() {
142     // Given
143     // When
144     // Then
145 }
146
147 @Test // Test 5
148 public void testSuccessfulLoginWithImproperCustomerCredentialsSpecificationException() {
149     // Given
150     // When
151     // Then
152 }
153
154 @Test // Test 6
155 public void testSuccessfulLoginWithImproperCustomerCredentialsSpecification() {
156     // Given
157     // When
158     // Then
159 }
160
161 @Test // Test 7
162 public void testSuccessfulLoginWithImproperCustomerCredentialsSpecification() {
163     // Given
164     // When
165     // Then
166 }
```

# Function ve BiFunction

# 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

# Function - II



- `Function<T, R>` fonksiyonel arayüzü ve tabi olarak üzerindeki tek soyut metot olan `apply(T t)` genel (generic) bir metottur ve tek bir argüman, `T` alıp bir sonuç, `R` döndürür.
- Lambda ifadesi oluşturulurken `T` ve `R` yerine gerçek tipler verilmelidir.
- Çalışma zamanında da belirtilen tiplere uygun değerler geçilmelidir.

# FunctionExample.java



- org.javaturk.oofp.ch09.functions.FunctionExample
  - **functionExample()**

# Function - III



- `Function<T, R>`'de üzerinde soyut olmayan 3 metot daha vardır:
- `andThen()` ve `compose()` default metotları başka fonksiyonlarla bileşim (composition) içindir.
- `identity()` ise statiktir ve daima argümanını geriye döndüren bir `Function` üretir.

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

# FunctionExample.java



- org.javaturk.oofp.ch09.functions.FunctionExample
  - identityExample()

# 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, T ve U alıp bir değer, R döndürür.

# BiFunctionExample.java



- org.javaturk.oofp.ch09.functions.BiFunctionExample

# Calculator1.java



- `org.javaturk.oofp.ch09.functions.other.Calculator1`
- Bu örnekte **Function** hazır fonksiyonun lambda ifadeleri, **Calculator1** nesnesinin değişkenleri olarak konumlandırılmıştır.



```
111 public void testSuccessfullLogin() {
112     // Given
113     // When
114     // Then
115     Customer loggedCustomer = service.login(properties, properPassword);
116     assertEquals(successfulCustomer, loggedCustomer);
117 }
118
119 @Test // Test 2
120 public void testSuccessfullLoginWithNoSuchCustomerException() {
121     // Given
122     // When
123     // Then
124     assertThrows(NoSuchCustomerNotFoundException.class, () -> {
125         Customer unfoundCustomer = service.login(unfoundCustomerTkn, properPassword);
126     });
127 }
128
129 @Test // Test 3
130 public void testSuccessfullLoginWithCustomerLockedException() {
131     // Given
132     // When
133     // Then
134     assertThrows(CustomerLockedException.class, () -> {
135         Customer lockedCustomer = service.login(lockedCustomerTkn, "password");
136     });
137 }
138
139 @Test // Test 4
140 public void testSuccessfullLoginWithCustomerAlreadyLoggedException() {
141     // Given
142     // When
143     // Then
144     assertThrows(CustomerAlreadyLoggedException.class, () -> {
145         Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomerTkn, properPassword);
146     });
147 }
148
149 @Test // Test 5
150 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {
151     // Given
152     // When
153     // Then
154     assertThrows(ImproperCustomerCredentialsException.class, () -> {
155         Customer customer = service.login(shortTkn, properPassword);
156     });
157 }
158
159 @Test // Test 6
160 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {
161     // Given
162     // When
163     // Then
164     assertThrows(ImproperCustomerCredentialsException.class, () -> {
165         Customer customer = service.login(propertiesTkn, properPassword);
166     });
167 }
```

# Consumer ve BiConsumer

# Consumer ve BiConsumer



- **Consumer<T>: void accept(T t)**
  - Setter metodu gibi çalışır, geçilen argümanı alıp işler ama geriye bir şey döndürmez.
  - Geçilen iki argümanı tüketen yapıdadır (consumer).
- **BiConsumer<T, U>: void accept(T t, U u)**
  - İki argümanlı setter metodu gibi çalışır.

# ConsumerExample.java



- org.javaturk.oofp.ch09.functions.ConsumerExample



```
public void testSuccessfulLogin() {
    throws NoSuchCustomerFoundException, CustomerLockedException, Customer
InproperCustomerCredentialsException, MalformedOrFailedLoggingAttempts
    Customer loggedCustomer = service.login(properTckn, properPassword);
    assertEquals(successfulCustomer, loggedCustomer);
}

@Test // Test 2
public void testUnsuccessfulLoginWithCustomerNotFoundException() {
    assertThrows(NoSuchCustomerFoundException.class, () -> {
        Customer unfoundCustomer = service.login(unfoundCustomerTckn, proper
    });
}

@Test // Test 3
public void testUnsuccessfulLoginWithCustomerLockedException() {
    assertThrows(CustomerLockedException.class, () -> {
        Customer lockedCustomer = service.login(lockedCustomerTcks, "over1");
    });
}

@Test // Test 4
public void testUnsuccessfulLoginWithCustomerAlreadyLoggedException() {
    assertThrows(CustomerAlreadyLoggedException.class, () -> {
        Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomer
    });
}

@Test // Test 5
public void testUnsuccessfulLoginWithImproperCustomerCredentialsException() {
    assertThrows(ImproperCustomerCredentialsException.class, () -> {
        Customer customer = service.login(shortTckn, properPassword);
    });
}

@Test // Test 6
public void testUnsuccessfulLoginWithImproperCustomerCredentialsExceptionFor
    assertThrows(ImproperCustomerCredentialsException.class, () -> {
        Customer customer = service.login(properTckn, properPassword));
    });
}

@Test // Test 7
public void testUnsuccessfulLoginWithImproperCustomerCredentialsExceptionOr
    assertThrows(ImproperCustomerCredentialsException.class, () -> {
        Customer customer = service.login(emptyTckn, shortPassword);
    });
}

@Test // Test 8
public void testSuccessfulLoginWithImproperCustomerCredentialsExceptionOr
    assertThrows(ImproperCustomerCredentialsException.class, () -> {
        Customer customer = service.login(emptyTckn, emptyPassword);
    });
}
```

# Supplier

# Supplier ve BooleanSupplier



- **Supplier<T>: T get()**
  - Getter ta da factory metodu gibi çalışır, bir değer üretip geri döndürür.

# SupplierExample.java



- org.javaturk.oofp.ch09.functions.SupplierExample



```
111 public void testSuccessfullLogin() {  
112     assertEquals(CustomerNotFoundException.class, customer.  
113         login("customer", "password").getClass());  
114     assertEquals("Customer logged in", service.loggedCustomer);  
115     assertEquals(true, successfulCustomer.isLogged());  
116 }  
117  
118 @Test // Test 3  
119 public void testSuccessfullLoginWithUnfoundCustomer() {  
120     assertEquals(UnfoundCustomerException.class, () -> {  
121         Customer.unfoundCustomer = service.login(unfoundCustomerTkn, proper  
122             .password());  
123     }).getClass();  
124 }  
125  
126 @Test // Test 4  
127 public void testSuccessfullLoginWithCustomerLocked() {  
128     assertEquals(CustomerLockedException.class, () -> {  
129         Customer.lockedCustomer = service.login(lockedCustomerTkn, proper  
130             .password());  
131     }).getClass();  
132 }  
133  
134 @Test // Test 5  
135 public void testSuccessfullLoginWithCustomerAlreadyLogged() {  
136     assertEquals(CustomerAlreadyLoggedException.class, () -> {  
137         Customer.alreadyLoggedCustomer = service.login(alreadyLoggedCustomer  
138             .Tkn, properPassword());  
139     }).getClass();  
140 }  
141  
142 @Test // Test 6  
143 public void testSuccessfullLoginWithImproperCustomerCredentials() {  
144     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
145         Customer.customer = service.login(shortTkn, properPassword());  
146     }).getClass();  
147 }  
148  
149 @Test // Test 7  
150 public void testSuccessfullLoginWithImproperCustomerCredentialsAndPassword() {  
151     assertEquals(ImproperCustomerCredentialsAndPasswordException.class, () -> {  
152         Customer.customer = service.login(properTkn, shortPassword());  
153     }).getClass();  
154 }  
155  
156 @Test // Test 8  
157 public void testSuccessfullLoginWithImproperCustomerCredentialsAndPasswordAndTkn() {  
158     assertEquals(ImproperCustomerCredentialsAndPasswordAndTknException.class, () -> {  
159         Customer.customer = service.login(properTkn, properPassword());  
160     }).getClass();  
161 }
```

# Predicate ve BiPredicate

# Diğer Hazır Fonksiyonel Arayüzler - III



- **Predicate<T>:** boolean test(T t)
  - Geçilen argüman ile ilgili bir önermeyi test eder.
- **BiPredicate<T, U>:** boolean test(T t, U u)
  - Geçilen iki argüman ile ilgili bir önermeyi test eder.

# PredicateExample.java



- org.javaturk.oofp.ch09.functions.PredicateExample



```
111 public void testSuccessfullLogin() {  
112     assertEquals(CustomerNotFoundException.class, customer.  
113         login("customer", "password").getClass());  
114     Customer loggedCustomer = service.login(properties, properPassword);  
115     assertEquals(true, successfulCustomer(loggedCustomer));  
116 }  
117  
118 @Test // Test 1  
119 public void testSuccessfullLoginWithUnfoundCustomer() {  
120     assertEquals(UnfoundCustomerException.class, () -> {  
121         Customer unfoundCustomer = service.login(unfoundCustomerToken, proper  
122             Password);  
123     }).getClass();  
124 }  
125  
126 @Test // Test 2  
127 public void testSuccessfullLoginWithCustomerLockedException() {  
128     assertEquals(CustomerLockedException.class, () -> {  
129         Customer lockedCustomer = service.login(lockedCustomerToken, "password");  
130     }).getClass();  
131 }  
132  
133 @Test // Test 3  
134 public void testSuccessfullLoginWithCustomerAlreadyLoggedException() {  
135     assertEquals(CustomerAlreadyLoggedException.class, () -> {  
136         Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomer  
137             Token, properPassword);  
138     }).getClass();  
139 }  
140  
141 @Test // Test 4  
142 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
143     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
144         Customer customer = service.login(shortToken, properPassword);  
145     }).getClass();  
146 }  
147  
148 @Test // Test 5  
149 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
150     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
151         Customer customer = service.login(propertiesToken, properPassword);  
152     }).getClass();  
153 }  
154  
155 }  
156  
157 @Test // Test 6  
158 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
159     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
160         Customer customer = service.login(propertiesToken, shortPassword);  
161     }).getClass();  
162 }
```

# UnaryOperator ve BinaryOperator

# UnaryOperator ve BinaryOperator - I



- Ayrıca, `Function<T, R>`'nın alt arayüzü `UnaryOperator<T>` ve `BiFunction<T, R, U>`'nın alt arayüzü `BinaryOperator<T>` vardır.
- `UnaryOperator<T>`, `Function<T, R>`'nın argüman ve dönüş tipi aynı olacak şekilde özelleşmiş hali iken, `BinaryOperator<T>` de `BiFunction<T, R, U>`'nın aynı tipten iki argüman alıp aynı tipten değer döndürecek şekilde özelleşmiş halidir.

# UnaryOperator ve BinaryOperator - II



- **UnaryOperator<T>** ve **BinaryOperator<T>** üzerindeki soyut metodlar şunlardır:
  - **UnaryOperator<T>:** `T apply(T t)`
  - **BinaryOperator<T>:** `T apply(T t1, T t2)`

# UnaryOperator<T> - I



- **UnaryOperator<T>**, **Function<T, R>**'nın alt arayüzüdür.
- Aralarındaki fark, **Function<T, R>**'da argüman ve dönüş değerinin tipleri farklı olabilirken **UnaryOperator<T>**'de ikisinin de aynı tipten olmasıdır.
- Dolayısıyla **UnaryOperator<T>**, aslen **Function<T, T>** şeklinde bir fonksiyondur.
- Dolayısıyla üzerindeki, **Function<T, R>**'dan devralınan metot da artık **T apply(T t)** şeklindedir.

# UnaryOperator<T> - II



- `UnaryOperator<T>` üzerindeki `identity()` ise statiktir ve daima argümanını geriye döndüren bir `UnaryOperator<T>` üretir.

# BinaryOperator<T> - I



- **BinaryOperator<T>**, **BiFunction<T, U, R>**'nin bir alt arayüzüdür.
- Aralarındaki fark, **BiFunction<T, U, R>**'da argümanlar ile dönüş değerinin tipleri farklı olabilirken **BinaryOperator<T>**'de hepsinin aynı tipten olmasıdır.
- Dolayısıyla **BinaryOperator<T>**, aslen **BiFunction<T, T, T>** şeklinde bir fonksiyondur.
- Dolayısıyla üzerindeki, **BiFunction<T, R>**'dan devralınan metod da artık **T apply(T t1, T t2)** şeklindedir.

# BinaryOperator<T> - II



- **BinaryOperator**, **BiFunction**'dan devraldığı **apply()** ve **andThen()** metodlarına şu iki statik metodu da 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.

# OperatorExample.java



- org.javaturk.oofp.ch09.functions.OperatorExample

# Calculator2.java



- `org.javaturk.oofp.ch09.functions.other.Calculator2`
- Bu örnekte **BinaryOperator** hazır fonksiyonun lambda ifadeleri, **Calculator2** nesnesinin değişkenleri olarak konumlandırılmıştır.



```
111 public void testSuccessfullLogin() {  
112     assertEquals(CustomerNotFoundException.class, customer.  
113         login("customer", "password").getClass());  
114     assertEquals("Customer logged in", customer.loggedCustomer);  
115     assertEquals(true, customer.loggedCustomer);  
116 }  
117  
118 @Test // Test 1  
119 public void testSuccessfullLoginWithNoSuchCustomerException() {  
120     assertEquals(NoSuchCustomerException.class, () -> {  
121         customer.login("customer", "password");  
122     }).  
123 }  
124  
125 @Test // Test 2  
126 public void testSuccessfullLoginWithCustomerLockedException() {  
127     assertEquals(CustomerLockedException.class, () -> {  
128         customer.login("lockedCustomer", "password");  
129     }).  
130 }  
131  
132 @Test // Test 3  
133 public void testSuccessfullLoginWithCustomerAlreadyLoggedException() {  
134     assertEquals(CustomerAlreadyLoggedException.class, () -> {  
135         customer.login("alreadyLoggedCustomer", "password");  
136     }).  
137 }  
138  
139 @Test // Test 4  
140 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
141     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
142         customer.login("customer", "password");  
143     }).  
144 }  
145  
146 @Test // Test 5  
147 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
148     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
149         customer.login("customer", "password");  
150     }).  
151 }  
152  
153 @Test // Test 6  
154 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
155     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
156         customer.login("customer", "password");  
157     }).  
158 }  
159  
160 @Test // Test 7  
161 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {  
162     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
163         customer.login("customer", "password");  
164     }).  
165 }
```

# İlkel Tipler İçin Hazır Arayüzler

# Basit Tipler İçin Hazır Arayüzler



- Hazır fonksiyonel arayüzler arasında tamamen genel olmayan (non-generic) yanı kısmen ya da bazen tamamen basit, ilkel tiplere özel oluşturulmuş olanlar da vardır.
- Bu arayüzlerde kullanılan basit tipler, `int`, `long` ve `double`'dır.

# Basit Tipler İçin Function - I



- `Function<T, R>`'nın ilkel tip alıp R döndüren şu halleri vardır:
  - `IntFunction<R>` ve R `apply(int)`: Bir `int` alıp R döndürür.
  - `LongFunction<R>` ve R `apply(long)`: Bir `long` alıp R döndürür.
  - `DoubleFunction<R>` ve R `apply(double)`: Bir `double` alıp R döndürür.

# Basit Tipler İçin Function - II



- Ayrıca argüman ve dönüş tipi `int`, `long` ve `double` olan şu 6 hali de vardır:
  - `IntToLongFunction: long applyAsLong(int)`
  - `InttoDoubleFunction: double applyAsDouble(int)`
  - `LongToIntFunction: int applyAsInt(Long)`
  - `LongtoDoubleFunction: double applyAsDouble(long)`

# Basit Tipler İçin Function - III



- **DoubleToIntFunction:** int applyAsInt(double)
- **DoubleToLongFunction:** long applyAsLong(double)

# PrimitiveFunctionExample.java



- org.javaturk.oofp.ch09.functions.primitive  
**PrimitiveFunctionExample**

# Basit Tipler İçin BiFunction - I



- BiFunction'un da ilkel tipler için şu 3 hali de vardır:
  - ToIntBiFunction<T, U>: int applyAsInt(T, U)
  - ToLongBiFunction<T, U>: long applyAsLong(T, U)
  - ToDoubleBiFunction<T, U>: double applyAsDouble(T, U)

# PrimitiveBiFunctionExample.java



- `org.javaturk.oofp.ch09.functions.primitive`  
`PrimitiveBiFunctionExample`

# Basit Tipler İçin Consumer



- **Consumer<T>:**
  - **IntConsumer:** void accept(int)
  - **LongConsumer:** void accept(long)
  - **DoubleConsumer:** void accept(double)
  - **ObjIntConsumer:** void accept(Object, double)
  - **ObjLongConsumer:** void accept(Object, long)
  - **ObjDoubleConsumer:** void accept(Object, double)

# Basit Tipler İçin Supplier



- **Supplier<T>:**
  - **IntSupplier:** int `getAsInt()`
  - **LongSupplier:** long `getAsLong()`
  - **DoubleSupplier:** double `getAsDouble()`
  - **BooleanSupplier:** boolean `getAsBoolean()`

# Basit Tipler İçin Predicate



- **Predicate<T>:**
  - **IntPredicate:** void test(int)
  - **LongPredicate:** void test(long)
  - **DoublePredicate:** void test (double)

# PrimitiveBuiltinFunctionExamples.java



- org.javaturk.oofp.ch09.functions.  
**PrimitiveBuiltInFunctionExamples**

# Java APIsinde Hazır Fonksiyon Kullanımı



- Hazır fonksiyonlar Java SE APIsinde yaygın olarak kullanılır.
- `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)`

# BuiltinFunctionsInJavaAPI.java



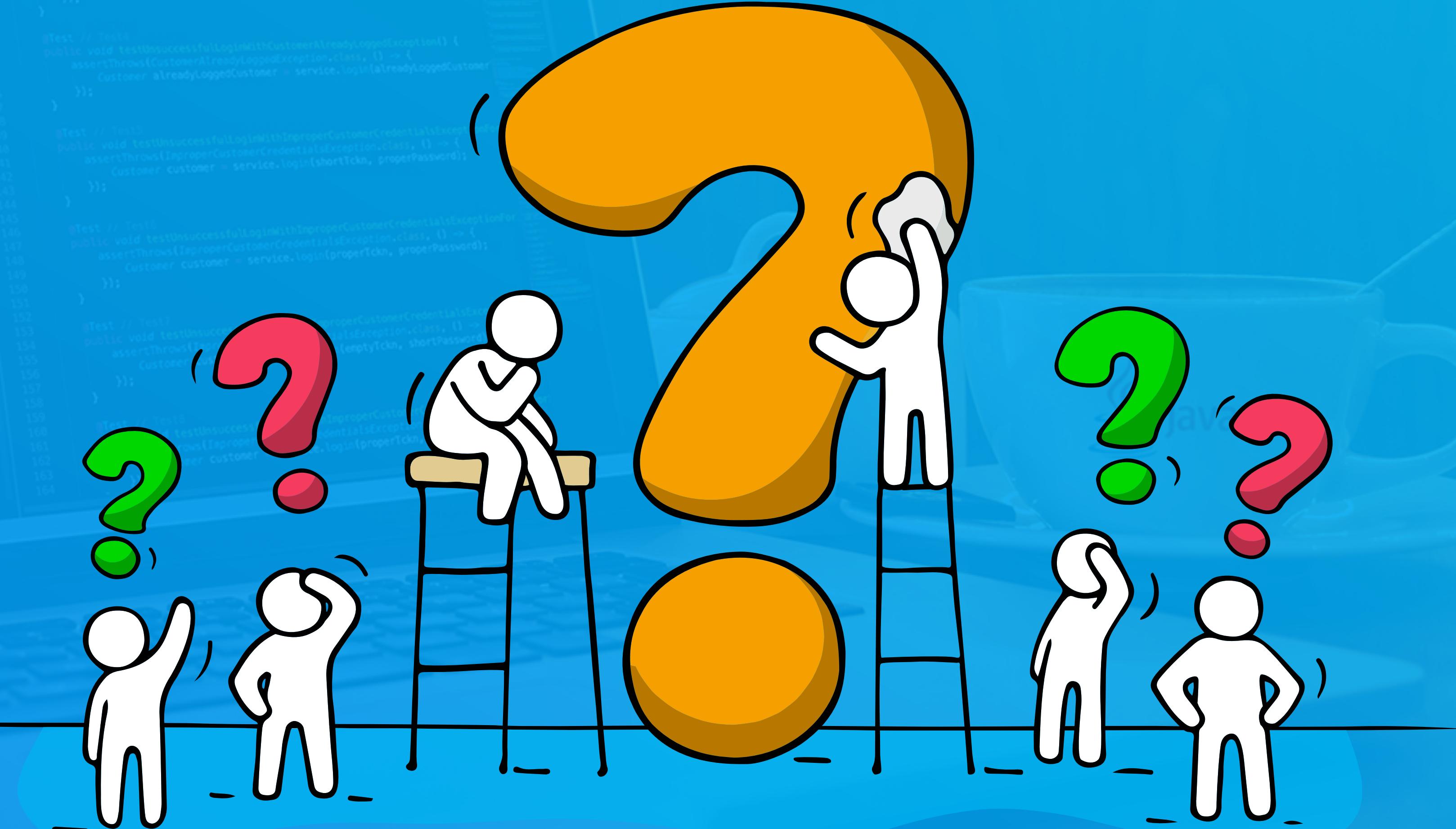
- org.javaturk.oofp.ch09.functions.  
BuiltinFunctionsInJavaAPI

# LambdaInLambda.java



- org.javaturk.oofp.ch09.functions.other.  
LambdaInLambda
- Bu örnekte bir lambda ifadesinin bloğunda başka bir lambda ifadesi  
kullanılmaktadır.

# Soru ve Cevap Zamani!





```
111 public void testSuccessfullLogin() {
112     assertEquals(CustomerNotFoundException.class, customer
113         .logIn("customer", "password").getClass());
114     Customer loggedCustomer = service.logIn("customer", "password");
115     assertEquals(true, successfulCustomer(loggedCustomer));
116 }
117
118 @Test // Test 1
119 public void testSuccessfullLoginWithNoSuchCustomerException() {
120     assertEquals(NoSuchCustomerException.class, () -> {
121         Customer unfoundCustomer = service.logIn("unfoundCustomer", "proper
122         password"));
123     });
124 }
125
126 @Test // Test 2
127 public void testSuccessfullLoginWithCustomerLockedException() {
128     assertEquals(CustomerLockedException.class, () -> {
129         Customer lockedCustomer = service.logIn("lockedCustomer", "proper
130         password"));
131     });
132 }
133
134 @Test // Test 3
135 public void testSuccessfullLoginWithCustomerAlreadyLoggedException() {
136     assertEquals(CustomerAlreadyLoggedException.class, () -> {
137         Customer alreadyLoggedCustomer = service.logIn("alreadyLoggedCustomer",
138         "properpassword"));
139     });
140 }
141
142 @Test // Test 4
143 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {
144     assertEquals(ImproperCustomerCredentialsException.class, () -> {
145         Customer customer = service.logIn("shortTckn", "properpassword"));
146     });
147 }
148
149 @Test // Test 5
150 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {
151     assertEquals(ImproperCustomerCredentialsException.class, () -> {
152         Customer customer = service.logIn("properTckn", "properpassword"));
153     });
154 }
155
156 @Test // Test 6
157 public void testSuccessfullLoginWithImproperCustomerCredentialsException() {
158     assertEquals(ImproperCustomerCredentialsException.class, () -> {
159         Customer customer = service.logIn("properTckn", "shortPassword"));
160     });
161 }
```

# Hazır Arayüzlerde Fonksiyon Bileşimleri

# Fonksiyon Bileşimleri - I



- Hazır arayüzlerin arkaya arkaya çalışacak şekilde bir zincir olarak ifade edilmesi mümkündür.
- Böylece **fonksiyon bileşimi (function composition)** elde edilir.
- Bu amaçla hazır fonksiyonlarda `andThen()` ve `compose()` default metotları vardır.
- Bu metotlar default olduklarından muhakkak hazır bir fonksiyonu gerçekleyen bir lambda ifadesi üzerinde çağrılmalıdır.

# Fonksiyon Bileşimleri - II



- Bu iki metod şöyle kullanılır:
  - **andThen ()** : Önce üzerinde çağrıldığı fonksiyon sonra geçen fonksiyon çalışır.
  - **compose ()** : Önce geçen fonksiyon, sonra üzerinde çağrıldığı fonksiyon çalışır.
- Fonksiyonların çalışma sırası açısından **compose ()** , **andThen ()** 'in tersidir.

# FunctionComposition1.java



- org.javaturk.oofp.ch09.functions.composition.  
FunctionComposition

# ComparatorComposition.java



- `org.javaturk.oofp.ch09.functions.composition.ComparatorComposition`
- Bu örnekte **Comparator** arayüzündeki `thenComparing()` metoduyla birden fazla kriter sahip bileşik kıyaslayıcılar oluşturulmaktadır.

# andThen() Bileşimi



- Şu fonksiyonel arayüzlerde `andThen()` metodu vardır:
  - `Function<T, U>`
  - `BiFunction<T, U, R>`
  - `Consumer<T>`
  - `BiConsumer<T, U>`
  - `IntConsumer`
  - `LongConsumer`
  - `DoubleConsumer`
  - `UnaryOperator<T>`
  - `IntUnaryOperator`
  - `LongUnaryOperator`
  - `DoubleUnaryOperator`

# compose() Bileşimi



- Şu fonksiyonel arayüzlerde `compose()` metodу vardır:
  - `Function<T, U>`
  - `UnaryOperator<T>`
  - `IntUnaryOperator`
  - `LongUnaryOperator`
  - `DoubleUnaryOperator`

# Predicate<T> - I



- **Predicate<T>**'de aşağıdaki **default** metodları vardır:
  - **and (Predicate<? super T> other)**: Bu ve geçilen **Predicate** fonksiyonu arasında kısa devre mantıksal **VE** çalıştırın bileşik fonksiyon üretir.
  - **or (Predicate<? super T> other)**: Bu ve geçilen **Predicate** fonksiyonu arasında kısa devre mantıksal **VEYA** çalıştırın bileşik fonksiyon üretir.
  - **Predicate<T, U> negate ()**: Bu **Predicate** fonksiyonunun dēilini veren fonksiyonu üretir.

# Predicate<T> - I



- **Predicate<T>**'de aşağıdaki statik metodlar da vardır:
  - **isEqual (Object targetRef)**: `java.util.Objects` üzerinde statik olan **equals (Object, Object)** metoduna göre fonksiyonu arasında kısa devre mantıksal VE çalıştırın bileşik fonksiyon üretir.
  - **not (Predicate<? super T> other)**: Geçilen **Predicate**'in tümleyenini üretir.

# BiPredicate<T, U>



- Benzer metodlar **BiPredicate<T, U>** için de vardır:
  - **and(BiPredicate<? super T, ? super U> other)**: Bu ve geçen **BiPredicate** fonksiyonu arasında kısa devre mantıksal **VE** çalıştırınan bileşik fonksiyon üretir.
  - **or(BiPredicate<? super T, ? super U> other)**: Bu ve geçen **BiPredicate** fonksiyonu arasında kısa devre mantıksal **VEYA** çalıştırınan bileşik fonksiyon üretir.
  - **BiPredicate<T, U> negate()**: Bu **BiPredicate** fonksiyonunun deðilini veren fonksiyonu üretir.

# DoublePredicate



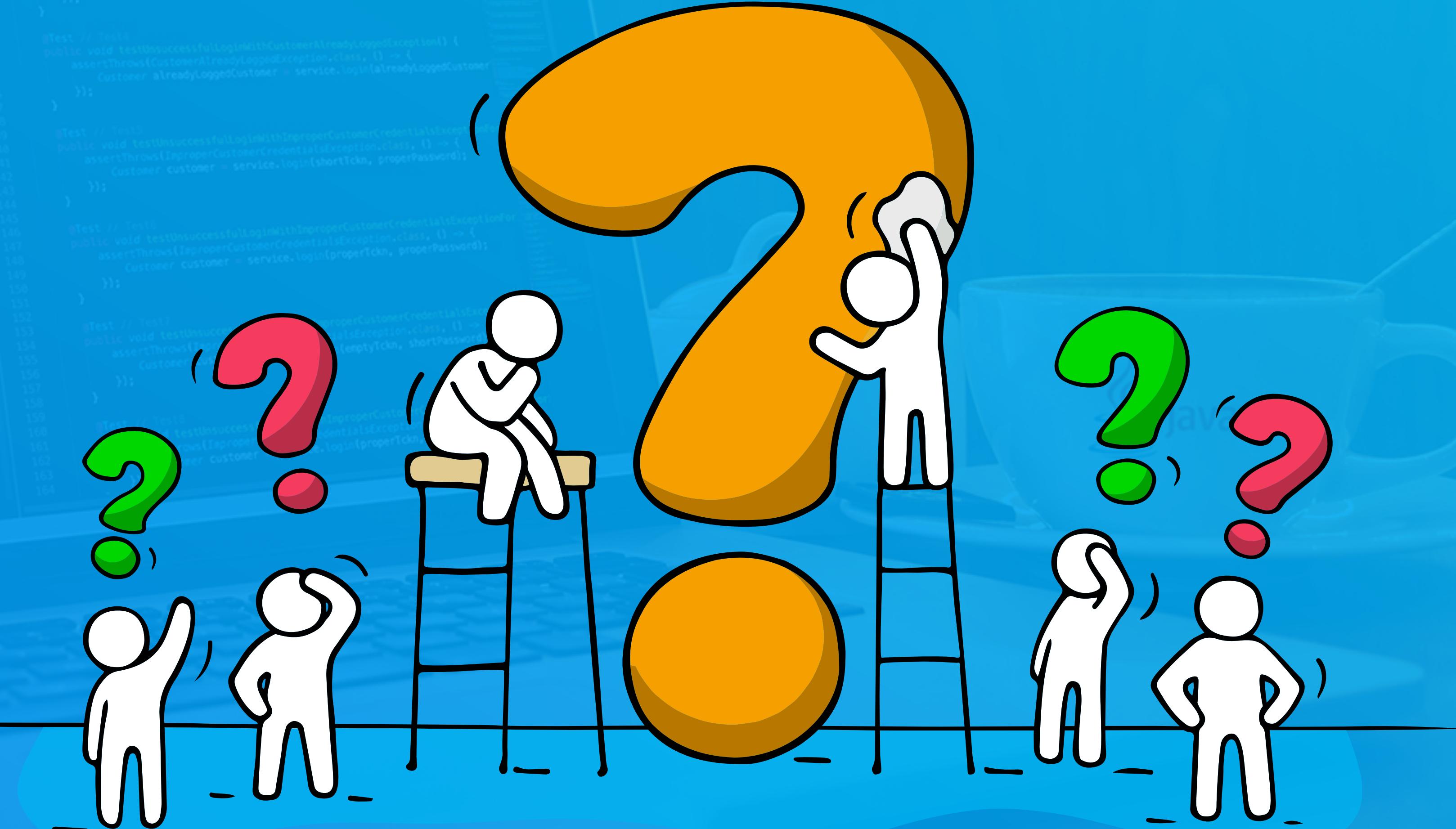
- DoublePredicate'de aşağıdaki default metodlar vardır:
  - **and (DoublePredicate other)** : Bu ve geçilen DoublePredicate fonksiyonu arasında kısa devre mantıksal VE çalıştırın bileşik fonksiyon üretir.
  - **or (DoublePredicate other)** : Bu ve geçilen DoublePredicate fonksiyonu arasında kısa devre mantıksal VEYA çalıştırın bileşik fonksiyon üretir.
  - **DoublePredicate negate ()** : Bu DoublePredicate fonksiyonunun deðilini veren fonksiyonu üretir.

# PredicateCompositionExample.java



- org.javaturk.oofp.ch09.functions.  
PredicateCompositionExample

# Soru ve Cevap Zamani!





```
111 public void testSuccessfulLogin() {  
112     assertEquals(CustomerNotFoundException.class, customer.  
113         login("customer1", "password1").getClass());  
114     assertEquals("Customer locked", customer.  
115         login("customer1", "password1").getErrorMessage());  
116 }  
117  
118 @Test // Test 1  
119 public void testSuccessfulLoginWithUnfoundCustomer() {  
120     assertEquals(UnfoundCustomerException.class, () -> {  
121         Customer unfoundCustomer = service.login("unfoundCustomer1", "pass  
122     }).getClass();  
123 }  
124  
125 @Test // Test 2  
126 public void testSuccessfulLoginWithCustomerLocked() {  
127     assertEquals(CustomerLockedException.class, () -> {  
128         Customer lockedCustomer = service.login("lockedCustomer1", "pass  
129     }).getClass();  
130 }  
131  
132 @Test // Test 3  
133 public void testSuccessfulLoginWithCustomerAlreadyLogged() {  
134     assertEquals(CustomerAlreadyLoggedException.class, () -> {  
135         Customer alreadyLoggedCustomer = service.login("alreadyLoggedCustomer1",  
136     }).getClass();  
137 }  
138  
139 @Test // Test 4  
140 public void testSuccessfulLoginWithImproperCustomerCredentials() {  
141     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
142         Customer customer = service.login("shortTckn", "properPassword");  
143     }).getClass();  
144 }  
145  
146 @Test // Test 5  
147 public void testSuccessfulLoginWithImproperCustomerCredentials() {  
148     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
149         Customer customer = service.login("properTckn", "properPassword");  
150     }).getClass();  
151 }  
152  
153 @Test // Test 6  
154 public void testSuccessfulLoginWithImproperCustomerCredentials() {  
155     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
156         Customer customer = service.login("properTckn", "shortPassword");  
157     }).getClass();  
158 }  
159  
160 @Test // Test 7  
161 public void testSuccessfulLoginWithImproperCustomerCredentials() {  
162     assertEquals(ImproperCustomerCredentialsException.class, () -> {  
163         Customer customer = service.login("properTckn", "properPassword");  
164     }).getClass();  
165 }
```

# Metot Referansları

# Metot Referansları - I



- Lambda ifadeleri bazen sadece var olan metotları çağırır, ifadenin gerçekleştirmesinde başka bir şey olmaz, sadece metot çağrıısı olur.
- Böyle hallerde metot çağrıısı yapmak yerine çağrıılacak metodу göstermek daha kısa ve anlaşılır olabilir.

# Metot Referansları - II



- 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 - III



- 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.

# Metot Referansları - III



Metot Referansı	Gösterim	Örnek Kullanım
Statik metot	Tipİsmi::metotİsmi	System.out::println
Belirli bir nesnenin metodu	nesne::metotİsmi	book::setAuthorFName
Belirli bir tipin nesne metodu	Tip::metotİsmi	Book::toString
Kurucu	Sınıfİsmi::new	Date::new

# Metot Referansları - IV



- Belirli bir tipin nesne metodu referansı kullanırken, üzerinde metot çağrılacak nesne parametre geçilir.

```
Consumer<Book> bookconsumer = Book::printInfo;  
bookconsumer.accept(book1);
```

- Belirli bir nesnenin metot referansı kullanırken, nesne metoduna parametre geçilebilir.

```
UnaryOperator<String> greeting = x → "Hello, ".concat(x);  
System.out.println(greeting.apply("World"));
```

# MethodReferences.java



- org.javaturk.oofp.ch09.functions.methods.  
MethodReferences



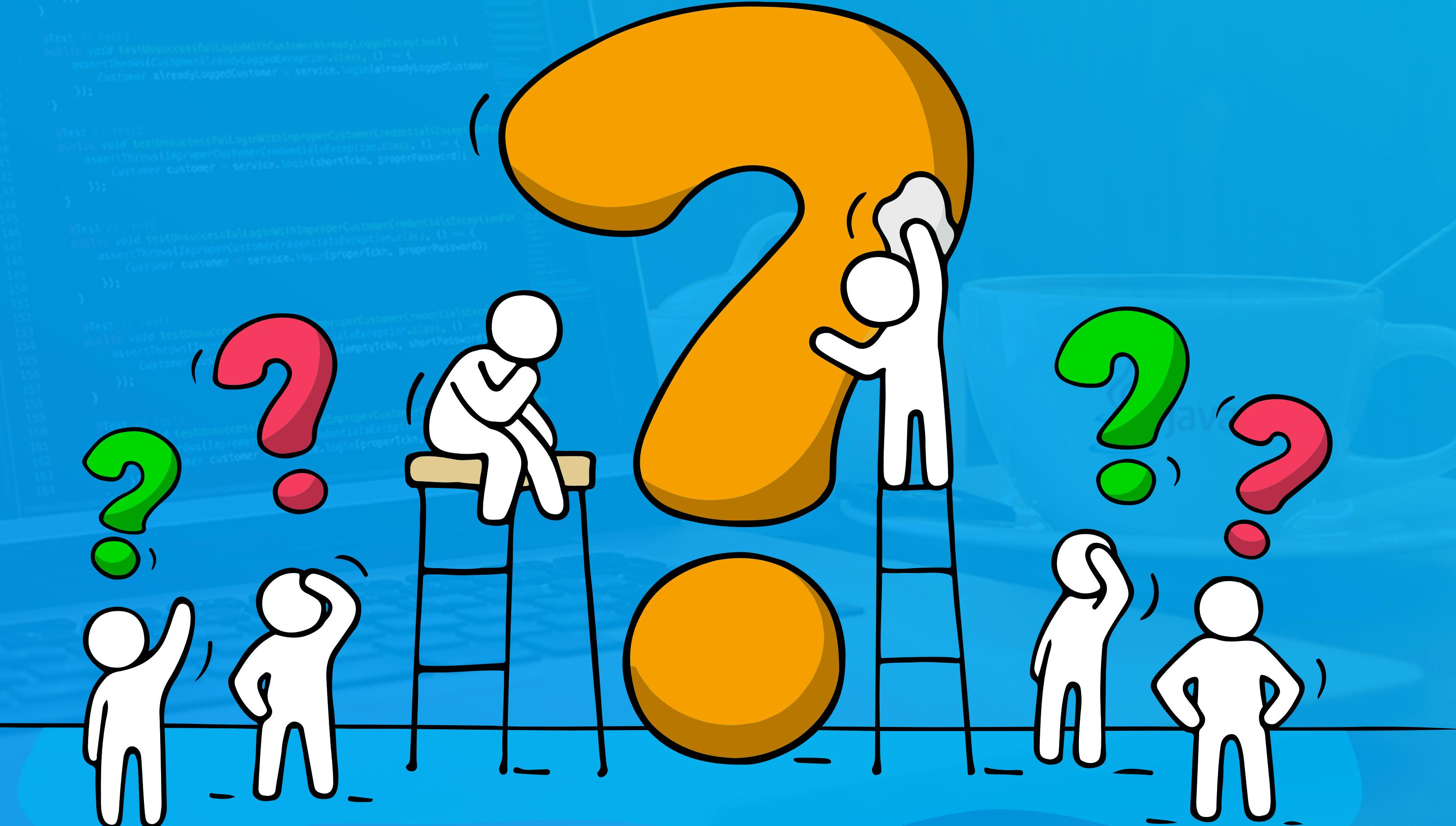
- Metot referanslarına parametre geçmede bazı kısıtlar söz konusudur.
- Metot referanslarına ancak referansı temsil eden hazır fonksiyondaki fonksiyonel metodun izin verdiği şekilde parametre geçilebilir.

# MethodReferences.java



- org.javaturk.oofp.ch09.functions.methods.  
MethodReferences
  - limitations()

# Soru ve Cevap Zamani!



# Nesne-Merkezli ve Fonksiyonel Programlama

# Nesne-Merkezli ve Fonksiyonel P. - I



- Nesne-merkezli programlama (NMP) ile FP nasıl bir arada kullanılabilir?
- NMP'nun varlık sebebi, nesnedir.
  - Nesne, duruma ve davranışa sahip olan bir yazılım birimidir.
  - Fonksiyonel programmanın durumu değiştirmeme prensibi, nesne-merkezli programmanın nesnelerine uygulanabilir mi?

# Nesne-Merkezli ve Fonksiyonel P. - II



- Nesnelerin durumu, tabi olarak değişir:
  - Ürünün fiyatı, stok bilgisi değişir,
  - Öğrencinin aldığı dersler, notları, sınıfı değişir,
  - Oyundaki nesnelerin konumları değişir, arabalar yarışır, askerler koşar, vs.
- Nesnenin üzerindeki metodlar, nesnenin durumunu değiştirir,
- Nesneler bu metodlarla 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.ch09.oop.car.CarTest ve DriverTest



- Dolayısıyla FP'nın 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 oluşturmaktadır.
- Ama bu hem programlama açısından zordur hem de bellek dolayısıyla da performans açısından sıkıntı doğurabilir.



- 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 metodlarının dahi o alanı değiştirmesini öner,
  - Değişmemesi gereken alanları **private** yapmak ve **setter** kullanmamak, diğer nesnelerin o alana ulaşmasını öner.

# CarTest.java ve DriverTest.java



- org.javaturk.oofp.ch09.oop.carFP.CarTest ve  
DriverTest

# Durumu Değişmeyen Nesne - I



- FP'ye uygun olarak durumu **değişmeyen nesne (immutable object)** oluşturmak için şu önlemler alınabilir:
  - Sınıfları **final** yapmak, dolayısıyla devralınmasını önleyerek metodlarının override edilmesini imkansız kılmak,
  - Bütün alanları **private** ve **final** yapmak,
  - Varsayılan kurucu (default ya da no-arg constructor) sağlamamak, varsa **private** yapmak,

# Durumu Değişmeyen Nesne - II



- Sadece argümanlı kurucular, mümkünse bir tane sağlamak,
- Bu durumda tüm **final** alanlarının ilk değer atamalarının burada yapılması gereklidir.
- Setter metotları kullanmamak,
- Hiç bir metotta durum değişikliği yapmamak.

# Durumu Değişmeyen Nesne - III



- Nesne üzerindeki torbaları (collection) değiştirilemez hale getirildikten sonra getter yoluyla döndürmek,
- Bu amaçla `java.util.Collections` üzerindeki metodlar kullanılabilir.
- Eğer nesne üzerinde durumu değiştirebilecek nesneler varsa, getter ile bu nesnelerin referanslarını değil, kopyalarını döndürmek.
- Eğer herhangi bir alanının kaçınılmaz olarak değişmesi gerekiyorsa, nesnenin var olan durumu yeni bir nesneye kopyalanıp, bu yeni nesne, değişen alanın yeni değeriyle birlikte geri döndürülebilir.

# Person.java ve Address.java



- org.javaturk.oofp.ch09.oop.address.Person ve Address

# ImmutableRGB.java



- org.javaturk.oofp.ch09.oopimmutable.ImmutableRGB

# Java API'sinde Değişmeme - 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 API'sinde Değişmeme - II



- `java.math.BigInteger` ve `java.math.BigDecimal`
- Pek çok enum sınıfının nesneleri de değişmezdir:
  - `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.
- 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 döndürür çünkü bağımlarından bağımsız değildir.
  - Ama döndürdükleri değerler değişmeyen (immutable) olabilir, olmalıdır.

# Değiştirmeme ve FP - II



- Dolayısıyla değiştirmeme, ideal ve stratejik bir hedeftir.
- Değiştirmeden tamamen kaçınılamasa bile en aza indirgenebilir.
- Fonksiyonel programlama bu konuda nesne-merkezli programlama yapanlara bir farkındalık ve çok faydalı bir pratik kazandırmış olur.

# Değiştirmeme ve FP - III



- Fonksiyonel programlamanın temel iddiası, veriyi değiştirmeme üzerindedir.
  - 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.

# Paradigmalar - II



- Ayrıca nesneler ile iş ortamındaki kavramlar ve mekanizmalar modellenir.
- Dolayısıyla bir programlama dilinin hem komutsal hem de nesne-merkezli yapılara sahip olması kaçınılmazdır.

# Paradigmalar - III



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



## OO pattern/principle

- Single Responsibility Principle
- Open/Closed principle
- Dependency Inversion Principle
- Interface Segregation Principle
- Factory pattern
- Strategy pattern
- Decorator pattern
- Visitor pattern

## FP pattern/principle

- Functions
- Functions
- Functions, also
- Functions
- Yes, functions
- Oh my, functions again!
- Functions
- Functions □



- Tamamen fonksiyonel olan programlama dillerinin (pure functional languages) çok akademik kaldıkları ve 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 fonksiyonel özelliklere sahip dillerdir.



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



- 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 kalıbü.



```
111 public void testSuccessfulLogin() {
112     assertThrows(NoSuchCustomerException.class, () -> {
113         Customer loggedCustomer = service.login(properties, properPassword);
114     });
115 }
116
117 @Test // Test 1
118 public void testSuccessfulLoginWithNoSuchCustomerException() {
119     assertThrows(NoSuchCustomerException.class, () -> {
120         Customer unfoundCustomer = service.login(unfoundCustomerTkn, properPassword);
121     });
122 }
123
124 @Test // Test 2
125 public void testSuccessfulLoginWithCustomerLockedException() {
126     assertThrows(CustomerLockedException.class, () -> {
127         Customer lockedCustomer = service.login(lockedCustomerTkn, properPassword);
128     });
129 }
130
131 @Test // Test 3
132 public void testSuccessfulLoginWithCustomerAlreadyLoggedException() {
133     assertThrows(CustomerAlreadyLoggedException.class, () -> {
134         Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomerTkn, properPassword);
135     });
136 }
137
138 @Test // Test 4
139 public void testSuccessfulLoginWithImproperCustomerCredentialsException() {
140     assertThrows(ImproperCustomerCredentialsException.class, () -> {
141         Customer customer = service.login(shortTkn, properPassword);
142     });
143 }
144
145 @Test // Test 5
146 public void testSuccessfulLoginWithImproperCustomerCredentialsException() {
147     assertThrows(ImproperCustomerCredentialsException.class, () -> {
148         Customer customer = service.login(propertiesTkn, properPassword);
149     });
150 }
151
152 @Test // Test 6
153 public void testSuccessfulLoginWithImproperCustomerCredentialsException() {
154     assertThrows(ImproperCustomerCredentialsException.class, () -> {
155         Customer customer = service.login(propertiesTkn, shortPassword);
156     });
157 }
```

# Açıklayıcı Programlama

# Komutsal Programlama



- Komutsal programlamada (imperative programming), programda, dilin yardımıyla, neyin nasıl olması gerektiği adım adım kurulur.
  - Adımlar ise derleyiciye verilen komutlar ya da emirlerdir.
- Nesne-merkezli programlamada (object-oriented programming) ise emirler doğrudan nesnelere verilir, nesneler üzerinden adımlar gerçekleştirilir.
- Nesne-merkezli programlama bu açıdan komutsal olarak görülebilir.

# Açıklayıcı Programlama - I



- Açıklayıcı programlamada (declarative programming) ise daha çok neyi istedğini 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.

# Açıklayıcı Programlama - II



- Genel olarak NMP da nesneler üzerinden soyut yapıların kurulmasını teşvik eder.
- Diller ya da yapılar soyutlaşıkça komutsaldan açıklayıcı programlamaya doğru ilerlenir ve geliştirme hızı artar.



## C'de Quicksort

```
qsort(int a[], hi, lo){  
    int h, w, p, t;  
    if (lo < hi) {  
        w = lo;  
        h = hi;  
        p = a[hi];  
        do {  
            while ((w < h) && (a[w] <= p))  
                w = w+1;  
            while ((h > w) && (a[h] >= p))  
                h = h-1;  
            if (w < h) {  
                t = a[w];  
                a[w] = a[h];  
                a[h] = t;  
            }  
        } while (w < h);  
        t = a[w];  
        a[w] = a[hi];  
        a[hi] = t;  
        qsort( a, lo, w-1 );  
        qsort( a, w+1, hi );  
    }  
}
```

## Lisp'te Quicksort

```
(defun quicksort (lis)  
  (if (null lis) nil  
      (let* ((x (car lis)) (r (cdr lis))  
             (fn (lambda (a) (< a x))))  
        (append (quicksort (remove-if-not fn r))(list x)  
                (quicksort (remove-if fn r))))))
```

# Stream ile Veri İşleme



- Aşağıdaki kod, Java'nın fonksiyonel veri işleme yapılarından olan **Stream** sınıfı ve lambda ifadeleriyle yazılmıştır.

```
long count = stream.map(s → Integer.parseInt(s)).
    filter(i → i % 2 == 0).
    map(i → Math.sqrt(Math.sqrt(i))).
    filter(i → i > 5).
    map(i → i * i * i * i).
    filter(i → i > 100).
    count();
```

# JDBC ve JPA

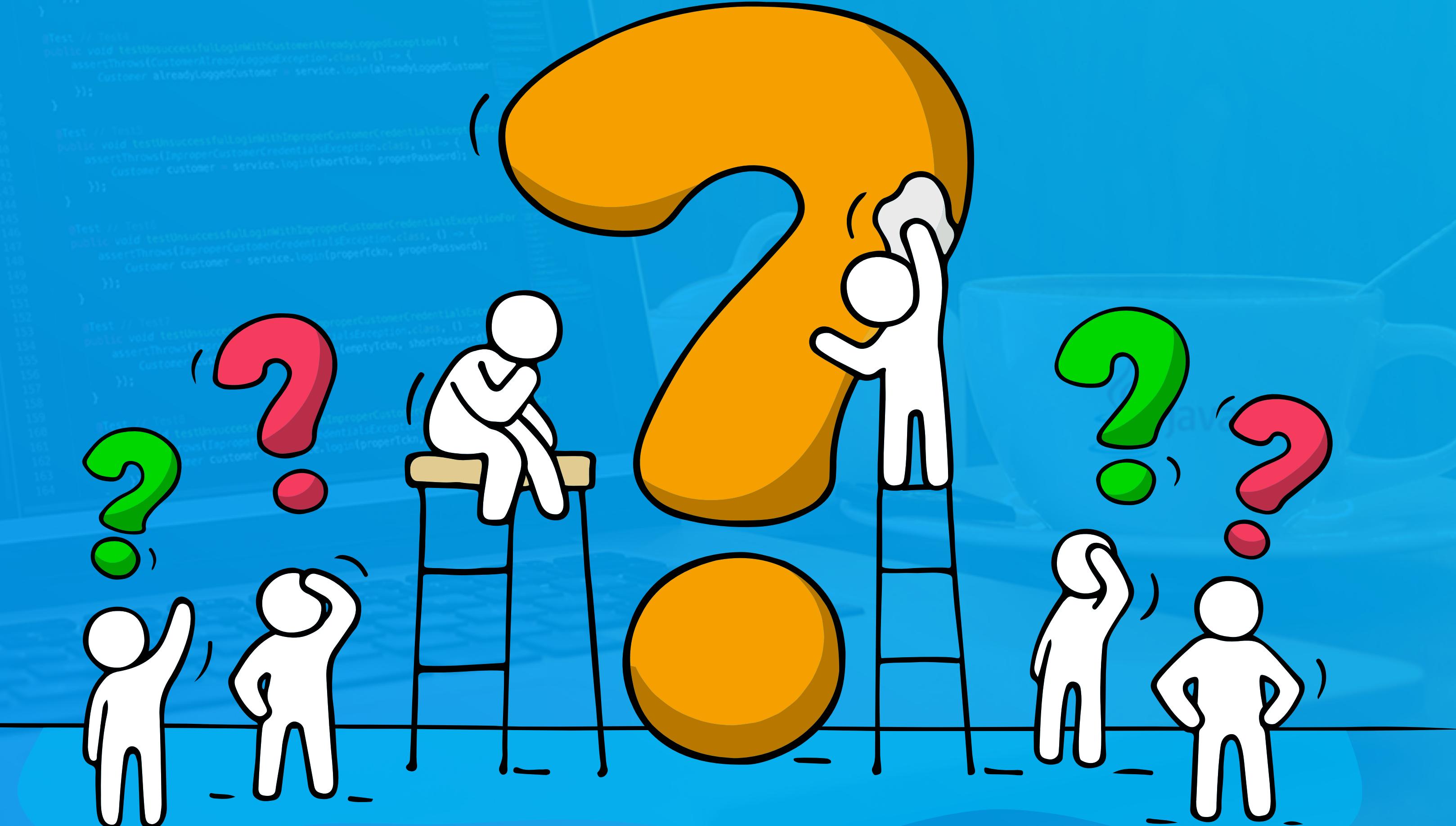


- Yandaki ilk kod JDBC ile ikinci kod ise JPA ile yazılmıştır.
- İlk kodda “nasıllık” kodda açıkça görülmektedir.
- İkinci kodda ise sadece istenen şey vardır, nasıl olacağı ise saklıdır.

```
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);  
}
```

# Soru ve Cevap Zamani!





# En iyi Uygulamalar



# En İyi Uygulamalar - I



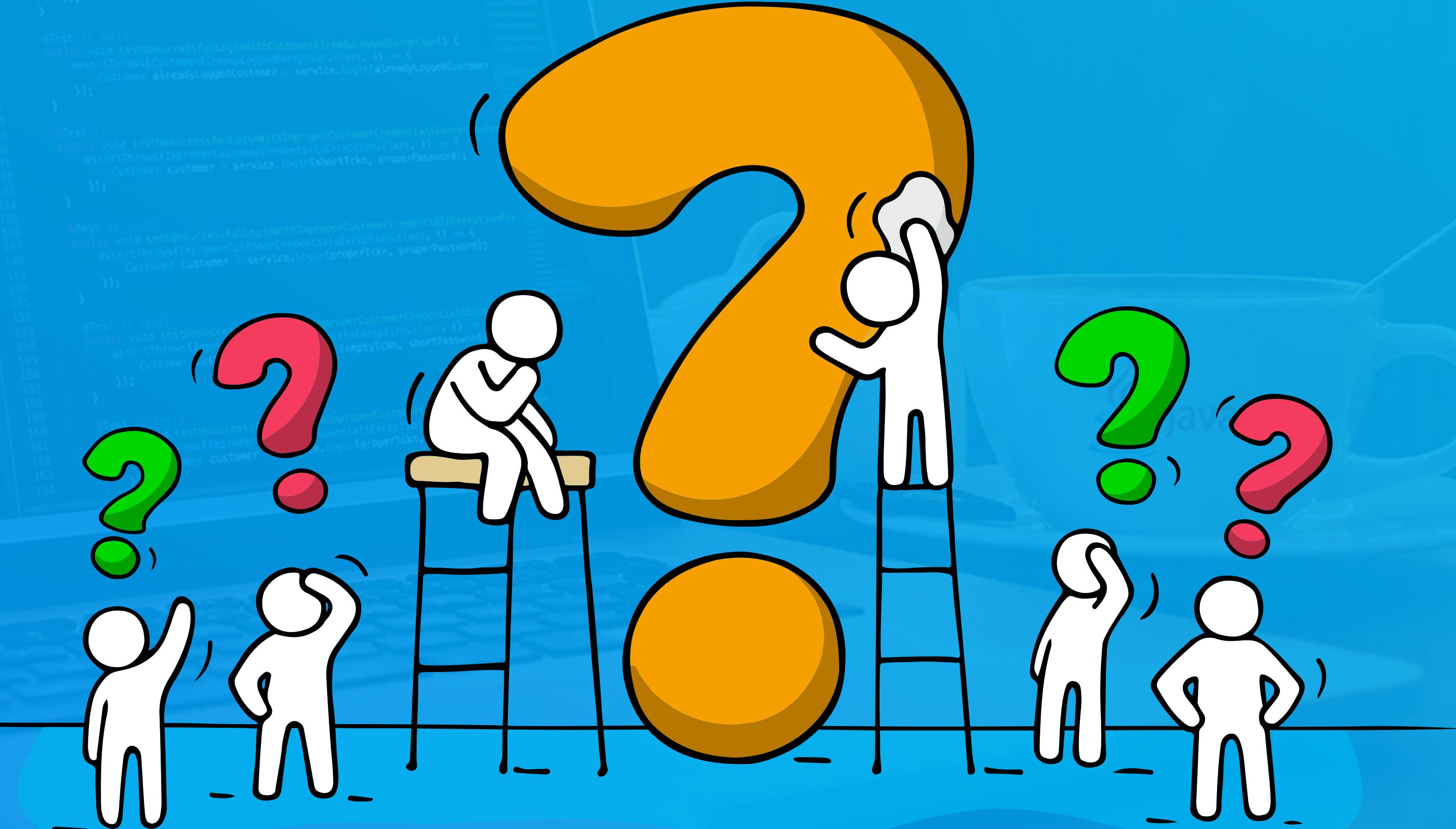
- Java'da fonksiyonel programlama için bazı en iyi pratiklerden (best practices) bahsedilebilir:
  - Yerel olmayan değişkenlerin değiştirilmesini sınırlayın,
  - Mمmkün olan her yerde metodları, saf fonksiyon olarak yazın,
  - Fonksiyonel arayüz oluşturmadan önce hazır arayüzleri araştırın, kullanabiliyorsanız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

# En İyi Uygulamalar - II



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

# Soru ve Cevap Zamani!





# Ödevler

```
112     public void testSuccessfulLogin() {
113         throws NoSuchCustomerFoundException, CustomerLockedException, Customer
114             ImproperCustomerCredentialsException, MaxNumberOffailedLoginsException
115         Customer loggedCustomer = service.login(properticks, properPassword);
116         assertEquals(successfulCustomer, loggedCustomer);
117     }
118
119     @Test // Test1
120     public void testUnsuccessfulLoginWithNoSuchCustomerException() {
121         assertThrows(NoSuchCustomerFoundException.class, () -> {
122             Customer unfoundCustomer = service.login(unfoundCustomerTckn, proper
123         });
124     }
125
126     @Test // Test2
127     public void testUnsuccessfulLoginWithCustomerLockedException() {
128         assertThrows(CustomerLockedException.class, () -> {
129             Customer lockedCustomer = service.login(lockedCustomerTckn, "password");
130         });
131     }
132
133     @Test // Test3
134     public void testUnsuccessfulLoginWithCustomerAlreadyLoggedInException() {
135         assertThrows(CustomerAlreadyLoggedInException.class, () -> {
136             Customer alreadyLoggedCustomer = service.login(alreadyLoggedCustomer
137         });
138     }
139
140     @Test // Test4
141     public void testUnsuccessfulLoginWithInproperCustomerCredentialsExceptionFor
142         assertThrows(InproperCustomerCredentialsException.class, () -> {
143             Customer customer = service.login(shortTckn, properPassword);
144         });
145
146     @Test // Test5
147     public void testUnsuccessfulLoginWithInproperCustomerCredentialsExceptionFor
148         assertThrows(InproperCustomerCredentialsException.class, () -> {
149             Customer customer = service.login(properticks, properPassword);
150         });
151
152     @Test // Test6
153     public void testUnsuccessfulLoginWithInproperCustomerCredentialsExceptionFor
154         assertThrows(InproperCustomerCredentialsException.class, () -> {
155             Customer customer = service.login(emptyTckn, shortPassword);
156         });
157     }
158 }
```



1. Daha önce çözdüğünüz **EvenNumberOperations** arayüzüünü gerçekleştiren lambda ifadeleri problemini hazır arayızlar kullanarak çözün.
  - Bu durumda **EvenNumberOperations** arayüzü yerine **java.util.Function** paketindeki arayızlarından birisini kullanmanız gereklidir.



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

# Bölüm Sonu

## Soru ve Cevap Zamani!

