NESNE YÖNELİMLİ PROGRAMLAMA

Nesne Yönelimli Programlama

Ders notları ve alıştırmalar

0 Neden nesne yönelimli?
1 Java'ya giriş
2 Sınıflar ve nesneler
3 Kapsülleme ve static
4 UML sınıf diyagramları
5 Sahiplik ve toplama
6 Kalıtım ve override
7 Çok biçimlilik
8 Soyut sınıf ve arayüz
9 Hata yönetimi
10 Dosyalar ve akışlar
11 Jenerikler ve koleksiyonlar
12 Enum, iç sınıf, lambda
13 Paralel programlama

HEKZON

hekzon.com/dersler

14 bölüm · 61 konu

İçindekiler

Bu notlar, YouTube'daki Nesne Yönelimli Programlama dersinin yazılı hâlidir. Her konunun yanındaki zaman, uzun videoda o konunun başladığı yeri gösterir. Her bölümün sonunda sorular var; cevaplarını ve çözümlerini en sonda bulabilirsin. Aynı soruları hekzon.com/dersler adresinde, cevabını hemen göreceğin alıştırmalar olarak da çözebilirsin.

0 **Neden nesne yönelimli?** 4
Milyonlarca satır · Dört sözcük

1 **Java'ya giriş** 10
Aynı dosya, iki sistem · Sınıfsız tek satır · 4 mü, 8 mi?

2 **Sınıflar ve nesneler** 20
new yazınca? · Kaç kilometre? · Program neden durdu? · Plaka neden boş?

3 **Kapsülleme ve static** 32
300 lira, 500 lira · Aynı ad, iki sınıf · Hesap numarası

4 **UML sınıf diyagramları** 42
Önce kutu, sonra kod · Çizgiler ne söyler? · Kaç tane?

5 **Sahiplik ve toplama** 51
Silinince ne kalır? · Elmas kodun neresinde?

6 **Kalıtım ve override** 59
Yirmi tür, bir kod? · Neden derlenmedi? · Prim nereye gitti? · Bu yazı nereden geldi? · String'den türemek · Kare bir dikdörtgendir

7 **Çok biçimlilik** 76
Aynı satır, iki sonuç · Üçgen geliyor · Daire ama çapı yok · Daireyle çalışıyor, kareyle çöküyor

8 **Soyut sınıf ve arayüz** 88
Hangi ödeme? · Tek üst sınıf · Yarının ödeme yöntemi · Hangisi önce? · Bozulmayan sınıflar

9 **Hata yönetimi** 103
Hangisi doğru? · Kırk satırlık hata · İki satır, iki hata · Bakiye yetmiyor · Dosyada ne var?

10 **Dosyalar ve akışlar** 118
Kaldığın yer · İki dakika · Nesnenin kendisi · Dosya açmak

11 **Jenerikler ve koleksiyonlar** 132
Kutuya ne girdi? · En büyüğü bulan tek metot · 127 eşit, 128 eşit değil · Beş şehir mi, üç şehir mi? · Konya'nın plakası kaç? · Gelemeyenleri silmek · Çekilişte aynı kişi iki kez · Neyin listesi?

12 **Enum, iç sınıf, lambda** 152
Günler sayıyla · Fiyat nerede durmalı? · Turnike hangi şubenin? · Tek kullanımlık sıra · Altı satırda tek satır iş · Boş yeri olan seanslar

13 **Paralel programlama** 170
Donan pencere · start mı, run mı? · Eksik sayaç · Araya kimse girmesin · Bitmeyen program · 500 fotoğraf, 500 thread?

NESNE YÖNELİMLİ PROGRAMLAMA

0 Neden nesne yönelimli?

Bu bölümde büyük programların neden nesne denen parçalara bölünerek yazıldığını görüyoruz: veriyle onu değiştiren işlemleri bir araya toplayan nesneyi, nesnelerin kalıbı olan sınıfı ve nesne yönelimli programlamanın dört ilkesini tek bir banka hesabı üzerinde inceliyoruz.

Milyonlarca satır

videoda 0:11

Büyük bir uygulama milyonlarca satır koddan oluşur ve üstünde yüzlerce yazılımcı aynı anda çalışır. Bu büyüklükte bir program neden nesne denen parçalara bölünerek yazılır?

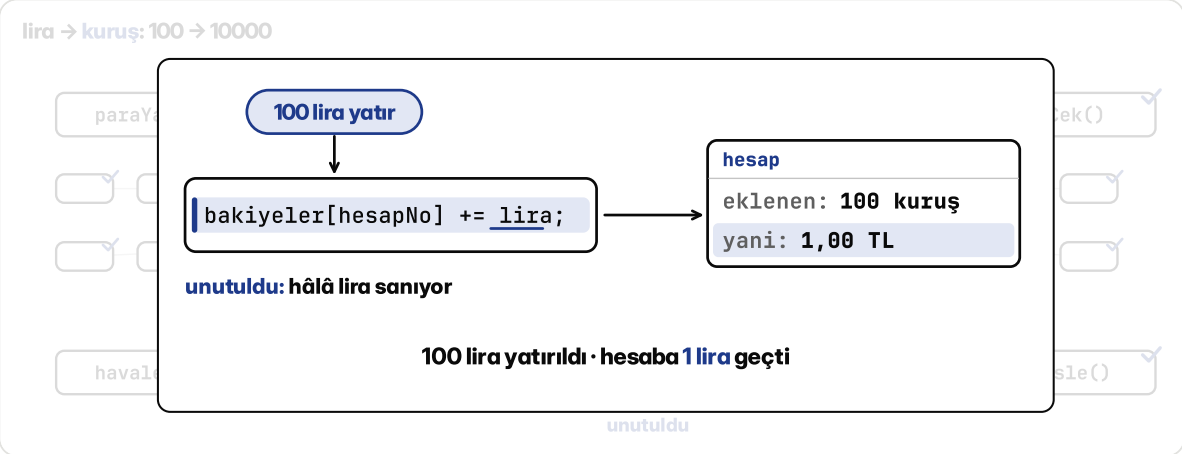
C gibi dillerde veri bir yerde durur, onu değiştiren fonksiyonlar başka yerlerde. Bir banka programı düşünelim: bütün hesapların bakiyesi tek bir dizide tutuluyor. Para yatıran, çeken, havale yapan, faiz işleten fonksiyonlar programın dört bir yanına dağılmış; diyelim 300 fonksiyon, hepsi bu diziye doğrudan dokunuyor.

Bir gün kuruşlar da tutulsun diye bakiye lira yerine kuruş olarak yazılmaya başlanıyor: 100 lira artık 10.000. 300 fonksiyonun her biri bulunup düzeltilmelidir. Biri unutulursa, 100 lira yatıran müşterinin hesabına yalnızca 1 lira geçer, çünkü o fonksiyon hâlâ lira sanıp bakiyeye 100 ekler. Unutulan fonksiyonun içindeki satır şudur:

```
bakiyeler[hesapNo] += lira;
```

Aynı programda bir fonksiyon düzeltilmiş, öteki unutulmuşsa ikisine de 100 lira yatırınca sonuç şöyle çıkar:

```
Şubeden 100 lira:       10000 kuruş = 100,00 TL
Bankamatikten 100 lira: 100 kuruş = 1,00 TL
```



Şekil 0.1 Bakiyeler kuruşa geçmiş; unutulan fonksiyon 100 lira yatırılınca hesaba 100 kuruş, yani 1,00 TL ekliyor.

Nesne yönelimli programlama bu dağınıklığı toplar: veriyi ve o veriyle çalışan işlemleri tek bir yerde birleştirir. Bu bütüne **nesne** denir. Bir banka hesabı nesnesinin bir durumu vardır: bakiyesi, örneğin 500 lira. Bir de davranışları vardır: para yatır ve para çek. Hesabın dışındaki kod bakiyeye kendisi dokunmaz; hesaba "100 lira yatır" der, işi hesap yapar.



Şekil 0.2 Hesap nesnesi: üstte durum (bakiye), altta davranışlar (para yatır, para çek); dışarıdaki kod bakiyeye doğrudan dokunmaz.

Java'da bu hesap aşağıdaki gibi yazılır: üstte bakiye, altında onu değiştiren iki işlem, hepsi aynı süslü parantezlerin içinde. Nesneye ait bu işlemlere **metot** denir.

```
class Hesap {
    int bakiye;
    void paraYatir(int tutar) {
        bakiye += tutar;
    }
    void paraCek(int tutar) {
        bakiye -= tutar;
    }
}
```

Bakiye kuruşa geçerse yalnız bu iki metodun içi değişir: bakiye += tutar * 100; ve bakiye -= tutar * 100;. Gelen tutar 100'le çarpılır; hesaba para yatıran, para çeken 300 fonksiyon olduğu gibi kalır.

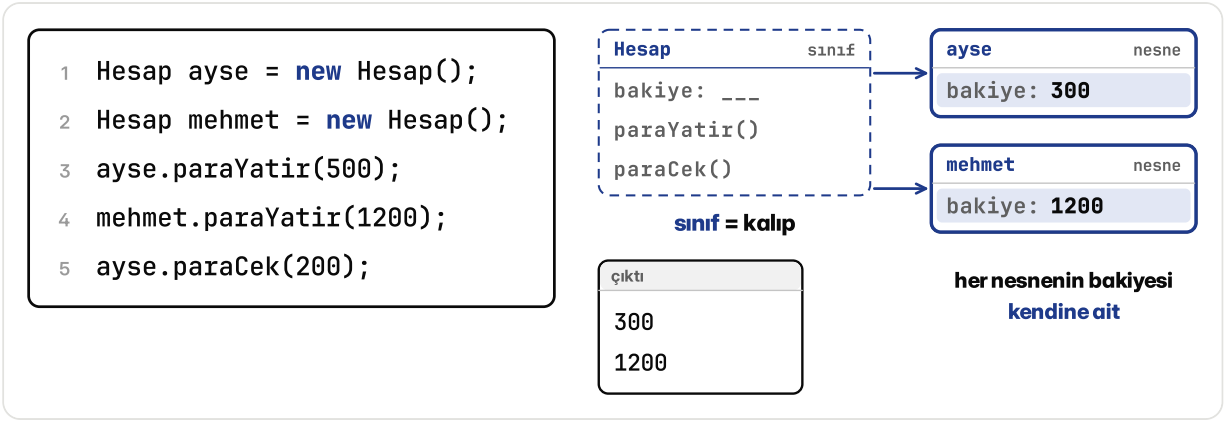
Bu kod tek bir hesap değil, bütün hesapların kalıbıdır; buna **sınıf** denir. Kalıptan istediğimiz kadar hesap üretiriz; Java'da yeni bir nesneyi new sözcüğü üretir. Ayşe'nin hesabına 500, Mehmet'inkine 1.200 lira yatsın, sonra Ayşe 200 lira çeksin:

```
Hesap ayse = new Hesap();
Hesap mehmet = new Hesap();
ayse.paraYatir(500);
mehmet.paraYatir(1200);
ayse.paraCek(200);
```

Ardından iki bakiye yazdırılınca:

```
300
1200
```

Ayşe'nin bakiyesi 300'e iner, Mehmet'inki 1.200'de kalır. İkisi de aynı kalıptan çıktı, ama her nesnenin bakiyesi kendine aittir. Metotlar kuruşa geçirilmiş sınıfla aynı satırlar çalıştırılınca 30000 ve 120000 yazılır; çağıran satırların hiçbiri değişmemiştir.



Şekil 0.3 Hesap sınıfı bir kalıp; ondan üretilen ayse ve mehmet nesnelerinin bakiyeleri ayrı: 300 ve 1200.

Milyonlarca satırlık bir program da böyle binlerce sınıfa bölünür; her sınıf kendi verisinden sorumludur. Bir şey değişince yüzlerce fonksiyonu tek tek aramayız, yalnız o veriyi taşıyan sınıfı açarız.

Dört sözcük

videoda 2:08

Yazılım mülakatlarında sık sorulan bir soru var: nesne yönelimli programlamanın dört ilkesi nedir? Cevap dört sözcük: kapsülleme, kalıtım, çok biçimlilik ve soyutlama. Saymak kolay; peki her biri tek bir banka hesabında neye karşılık gelir?

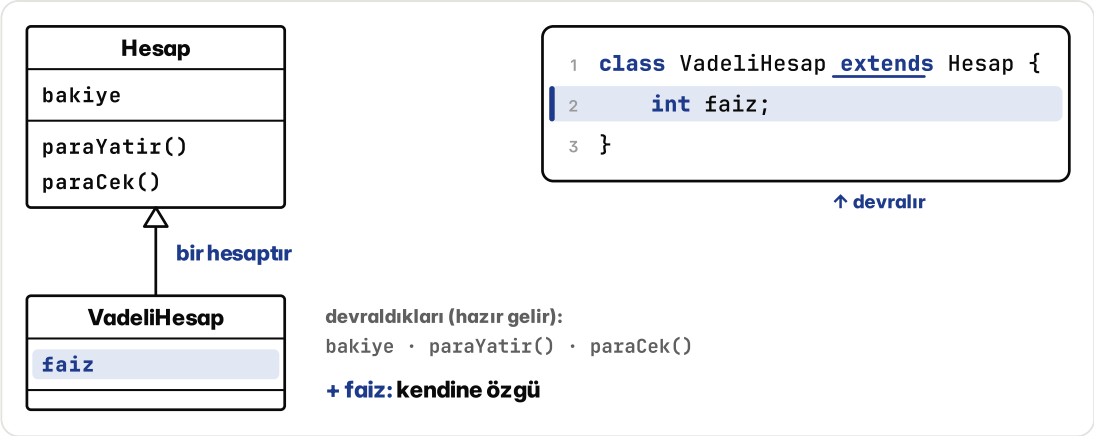
Banka hesabı bir nesnedir: bir bakiyesi ve iki işlemi vardır, para yatır ve para çek. Birinci ilke **kapsülleme** (encapsulation): bakiye hesabın içinde kapalı durur. Dışarıdaki bir kod bakiyeyi doğrudan 1 milyon yapmaya kalkarsa program derlenmez bile:

```
h.bakiye = 1000000;
```

Derleyici bu satır için "bakiye has private access in Hesap" der, yani bakiye Hesap'ın dışına kapalıdır. Bakiye yalnız hesabın kendi işlemleriyle değişir; böylece bakiyeyle ilgili her kural tek bir yerde yazılır.

İkinci ilke **kalıtım** (inheritance). Vadeli hesap da bir hesaptır: bakiyeyi, para yatırmayı ve para çekmeyi sıradan hesaptan hazır devralır. Bunları yeniden yazmayız; üstüne yalnız kendine özgü olanı ekleriz: birikmiş faiz. Java'da bu, extends sözcüğüyle yazılır:

```
class VadeliHesap extends Hesap {
    int faiz;
}
```



Şekil 0.4 VadeliHesap, Hesap'tan bakiye, paraYatir() ve paraCek()'i hazır devralır; üstüne yalnız faiz eklenir.

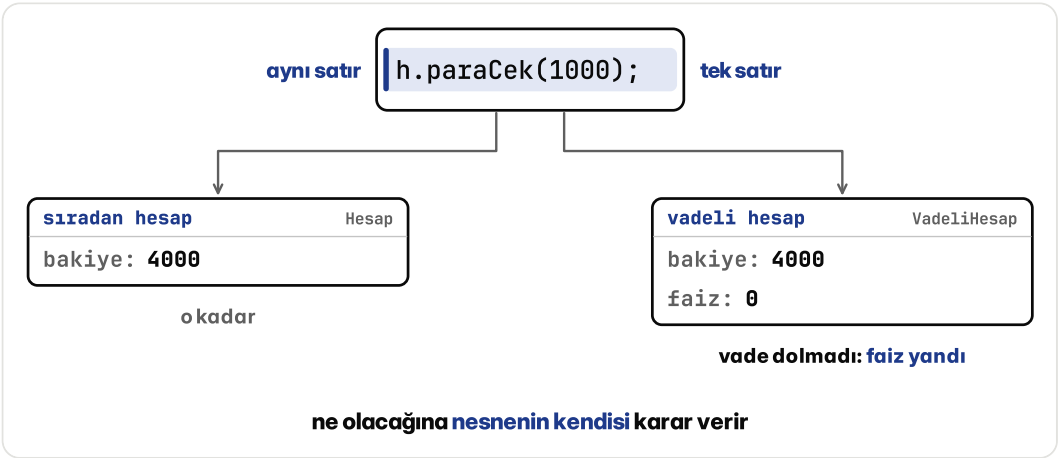
Üçüncü ilke **çok biçimlilik** (polymorphism): aynı istek, nesnenin türüne göre farklı iş yapar. İki hesapta da 5.000 lira var; ikisinden de 1.000 lira çekelim. Sıradan hesapta bakiye 4.000'e iner, o kadar. Vadeli hesapta vade dolmadıysa bakiye yine 4.000'e iner, ama birikmiş 150 lira faiz de yanar. Para çekme isteğini gönderen kod iki hesap için de aynı tek satırdır:

```
h.paraCek(1000);
```

Programın çıktısının son iki satırı:

```
Sonra   sıradan: bakiye 4000
Sonra   vadeli:  bakiye 4000, faiz 0
```

Ne olacağına satırı yazan değil, hesabın kendisi karar verir.



Şekil 0.5 Aynı h.paraCek(1000) satırından sonra sıradan hesapta bakiye 4000; vadeli hesapta bakiye 4000 ve faiz 0.

Dördüncü ilke **soyutlama** (abstraction): gereksiz ayrıntıyı gizleyip yalnız işe yarayan yüzü göstermek. Bankamatikten para çekerken kartı takar ve tutarı gireriz; bankanın içinde hangi kayıtların değiştiğini görmeyiz. Programda da hesabı kullanan kod yalnız para yatırmayı ve para çekmeyi görür; bakiyenin lirayla mı kuruşla mı tutulduğunu bilmez.

Bu fikirler yeni değil. 1967'de Norveç'te Simula dili, adından da anlaşılacağı gibi simülasyon için, yani gerçek dünyadaki sistemleri bilgisayarda canlandırmak için geliştirildi. Sınıf, nesne ve kalıtım ilk kez bu dilde ortaya çıktı. 1970'lerde Smalltalk'ta her şey, sayılar bile, birer nesneydi. 1985'te C++ çıktı: C diline sınıflar eklenmiş hâliydi. 1995'te de Java geldi; baştan nesne yönelimli tasarlanmıştı.



Şekil 0.6 Zaman çizgisi: 1967 Simula, 1970'ler Smalltalk, 1985 C++, 1995 Java.

Java bugün de en çok kullanılan dillerden biridir ve dört ilkenin dördü de dilin içinde hazır gelir. Mülakatta o dört sözcük sorulduğunda cevap tek bir banka hesabında saklıdır: bakiyesini koruyan, vadeli hesaba miras bırakan, aynı isteğe türüne göre cevap veren ve içini göstermeyen bir nesne.

Özet

- Dağınık fonksiyonların hepsi aynı veriye dokunursa verinin biçimi değişince hepsini tek tek bulmak gerekir; biri unutulunca hata çıkar.
- Nesne, veriyi (durum) ve o veriyle çalışan işlemleri (davranış) bir arada tutar; nesneye ait işlemlere metot denir.
- Sınıf nesnelerin kalıbıdır; Java'da new aynı kalıptan istenen sayıda nesne üretir ve her nesnenin verisi kendine aittir.
- Kapsülleme veriyi içeride saklar, kalıtım bir sınıfın ötekinden hazır devralmasıdır.
- Çok biçimlilikte aynı istek nesnenin türüne göre farklı iş yapar; soyutlama yalnız işe yarayan yüzü gösterir.
- Sınıf, nesne ve kalıtım 1967'de Simula ile doğdu; Java 1995'te baştan nesne yönelimli tasarlandı.

Sorular

1 **Bir banka hesabı nesnesinde bakiye ile para çekme işlemi sırasıyla neye örnektir?**

A Davranış ve durum
B Durum ve davranış
C Sınıf ve nesne
D Nesne ve sınıf

2 **Hesap sınıfı ile Hesap ayse = new Hesap(); satırıyla oluşan nesne arasındaki ilişki nedir?**

A Hesap bir kalıptır; ayse o kalıptan üretilmiş, bakiyesi kendine ait bir nesnedir.

B ayse, Hesap sınıfının bir kopyasıdır ve bakiyesi bütün hesaplarla ortaktır.

C Hesap bir nesnedir; ayse onun metotlarından biridir.

D İkisi aynı şeyin iki adıdır.

3 **Bölümdeki Hesap sınıfıyla aşağıdaki satırlar ekrana ne yazar?**

```
Hesap ayse = new Hesap();
ayse.paraYatir(500);
ayse.paraCek(200);
ayse.paraYatir(50);
System.out.println(ayse.bakiye);
```

Cevap:

4 **Bakiyeler kuruşa geçmiş, ama bir fonksiyon unutulmuş: hâlâ lira sanıp yatırılan tutarı bakiyeye olduğu gibi ekliyor. Bu fonksiyonla 250 lira yatırılırsa hesaba kaç lira geçer?**

Cevap:

5 **Bakiye kuruşa geçtiğinde nesne yönelimli yazılmış Hesap sınıfında neyin değişmesi gerekir?**

A Hesaba para yatıran ve çeken bütün satırların

B Yalnız main metodunun

C Yalnız paraYatir ve paraCek metotlarının içinin

D Hiçbir şeyin; Java kuruşu kendisi hesaplar

6 **Hesap sınıfının dışındaki h.bakiye = 1000000; satırının derlenmemesi hangi ilkenin sonucudur?**

A Kalıtım

B Çok biçimlilik

C Soyutlama

D Kapsülleme

7 **Sıradan ve vadeli iki hesapta aynı h.paraCek(1000); satırı çalışıyor; vadeli hesapta bakiyenin yanında birikmiş faiz de yanıyor. Bu hangi ilkedir?**

A Kalıtım

B Çok biçimlilik

C Kapsülleme

D Soyutlama

8 **Sınıf, nesne ve kalıtım ilk kez hangi dilde ortaya çıktı?**

A Simula

B Smalltalk

C C++

D Java

NESNE YÖNELİMLİ PROGRAMLAMA

1 Java'ya giriş

Bu bölümde Java programının nasıl derlenip çalıştığını görüyoruz: aynı dosyanın her sistemde çalışmasını sağlayan bytecode ve Java sanal makinesi, ilk program ve main metodu, klavyeden okuma, sonra C'den gelenler için Java'nın farkları: sabit tip boyları, mantıksal tip, dizi sınırı denetimi, String nesnesi, referans ve çöp toplayıcı.

Aynı dosya, iki sistem

videoda 4:17

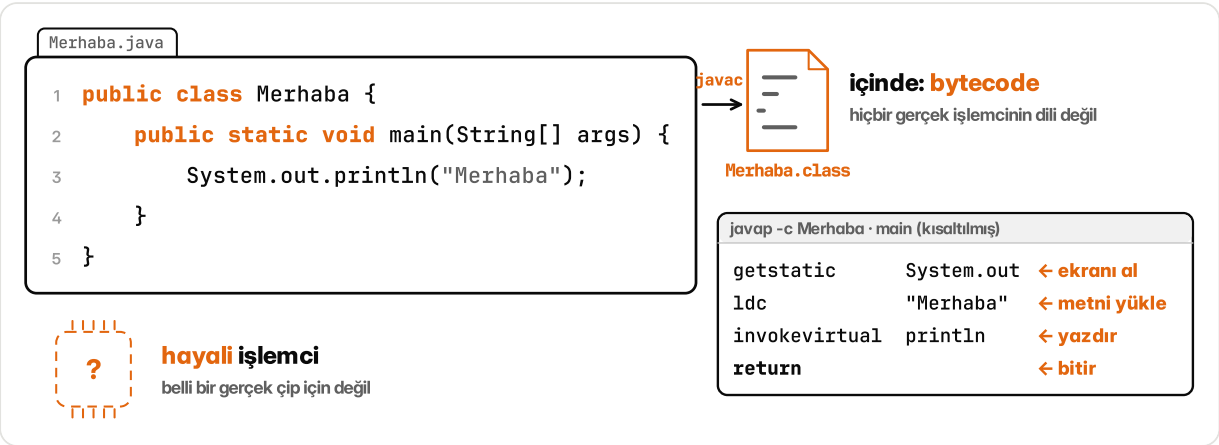
Windows'ta derlenen bir Java programı Linux'a kopyalanır ve hiçbir şeyi değiştirilmeden çalışır. C dilinde derlenmiş bir programla bunu yapamayız. Aynı Java dosyası iki sistemde de nasıl çalışıyor?

C derleyicisi kaynak kodu doğrudan işlemcinin anladığı makine koduna çevirir ve çıkan dosyayı o işletim sisteminin kendi program biçiminde paketler. Windows için derlenen merhaba.exe dosyası 4D 5A baytlarıyla, Linux için derlenen dosya 7F 45 4C 46 baytlarıyla başlar; Windows için derlenen dosyayı Linux tanımaz, programı orada yeniden derlemek gerekir. İşlemci farklıysa, örneğin telefonlardaki gibi, makine kodu da baştan farklıdır.

Java derleyicisi ise kaynak kodu hiçbir gerçek işlemcinin diline çevirmez. Merhaba.java dosyasından Merhaba.class dosyası çıkar; içindeki komutlara **bytecode** denir.

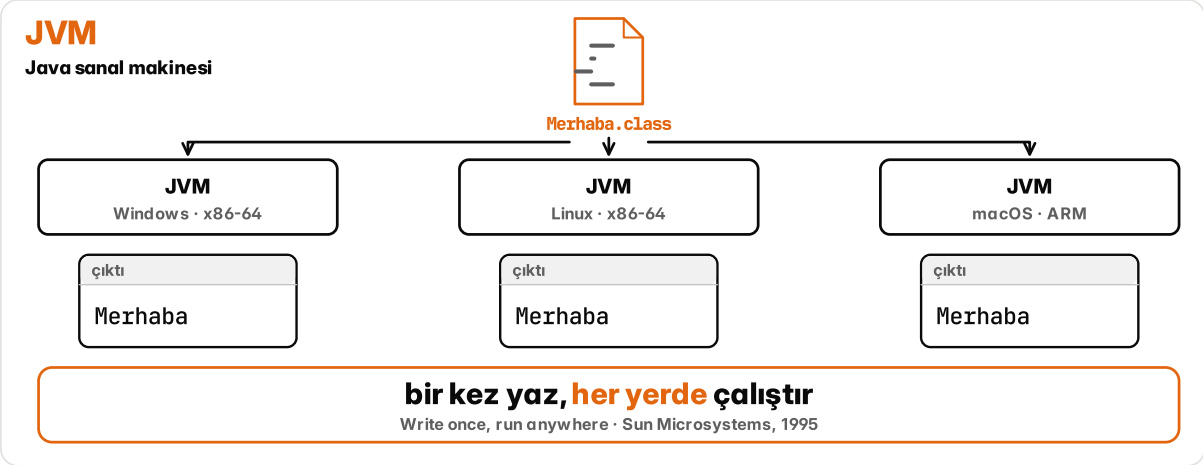
```
public class Merhaba {
    public static void main(String[] args) {
        System.out.println("Merhaba");
    }
}
```

Programın başladığı main metodu yalnızca 4 komuttur: ekranı al (getstatic), metni yükle (ldc), yazdır (invokevirtual) ve bitir (return). Bu komutlar belli bir gerçek işlemci için değil, hayali bir işlemci için yazılmıştır.



Şekil 1.1 Merhaba.java javac ile Merhaba.class'a derlenir; main'in bytecode'u dört komuttur: ekranı al, metni yükle, yazdır, bitir.

O hayali işlemciyi **Java sanal makinesi** (JVM) canlandırır. Sanal makine, Windows için ayrı, Linux için ayrı hazırlanmış bir programdır. Ama hepsi aynı bytecode'u okur ve her komutu kendi işlemcisine yaptırır. Böylece aynı Merhaba.class dosyası iki sistemde de Merhaba yazar. Java'nın ilk günden sloganı buydu: bir kez yaz, her yerde çalıştır.

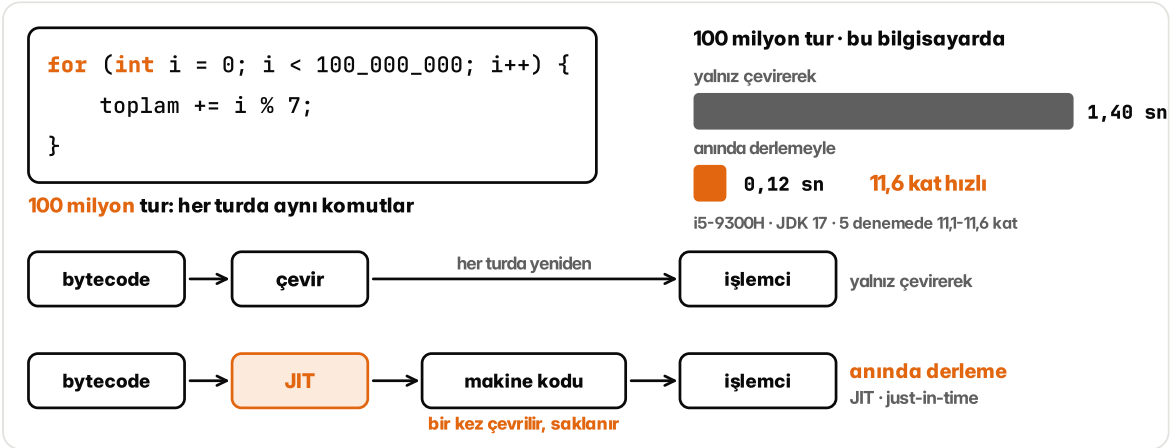


Şekil 1.2 Aynı Merhaba.class; Windows, Linux ve macOS'taki üç ayrı JVM'de çalışıp aynı çıktıyı verir.

Her komutu çalışırken tek tek çevirmek yavaş olmaz mı? Olurdu; bu yüzden sanal makine, çok tekrarlanan kısımları program çalışırken bir kez gerçek makine koduna çevirir ve saklar. Buna **anında derleme** (JIT, just-in-time) denir. Şu döngü 100 milyon tur döner:

```
for (int i = 0; i < 100_000_000; i++) {
    toplam += i % 7;
}
```

Bu bilgisayarda (i5-9300H, JDK 17) döngü yalnızca çevirerek 1,40 saniye sürdü; anında derlemeyle 0,12 saniyede bitti. Fark 11,6 kat, yani 10 kattan fazla.



Şekil 1.3 Yorumlayıcı her turda bytecode'u yeniden çevirir, JIT bir kez çevirip saklar; 100 milyon tur 1,40 sn yerine 0,12 sn sürer.

Bu sayede yazılımcı kodu kendi Windows bilgisayarında yazıp dener, aynı dosya şirketin Linux sunucusunda çalışır. Android uygulamaları da uzun yıllar Java ile yazıldı; Minecraft da ilk olarak Java ile yazılmıştı. Bir banka uygulamasında para gönder düğmesine basınca isteği bankanın sunucusundaki programlar karşılar; bankalarda bu programların önemli bir kısmı Java ile yazılır.

Sınıfsız tek satır

videoda 6:10

Ekrana Merhaba yazdıran satırı bir Java dosyasına tek başına yazalım:

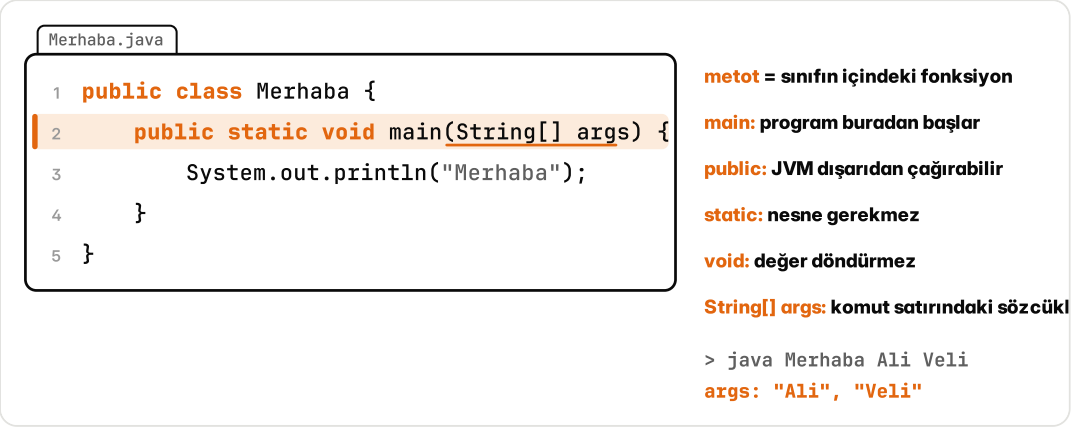
```
System.out.println("Merhaba");
```

Derleyici kabul etmez: "class, interface, enum, or record expected", yani satırdan önce bir sınıf ya da benzeri bir tanım bekler. Tek satırlık bir iş için neden bir sınıf gerekiyor?

Çünkü Java'da her kod bir sınıfın içinde yaşar; sınıfın dışında kod yoktur. Sınıf, birbirine ait veriyi ve işlemleri bir arada tutan bir kalıptır; Java programları bu kalıplardan kurulur. Programı çalıştıran sanal makinenin yüklediği birim de sınıftır; derleyicinin ürettiği dosyanın adı bu yüzden Merhaba.class olur. Dışarıya açık, yani public bir sınıfın adı dosyanın adıyla aynı olmalıdır. Merhaba sınıfını Selam.java dosyasına yazarsak derleyici yine durur: "class Merhaba is public, should be declared in a file named Merhaba.java".

Java'da sınıfın içindeki fonksiyonlara **metot** denir. Her Java programı main metodundan başlar; sanal makine programı açınca ilk bu metodu arar. Satırdaki sözcüklerin her birinin bir işi vardır:

- public: metot dışarıdan, yani sanal makine tarafından çağrılabilir.
- static: onu çağırmak için önce bir nesne oluşturmak gerekmez.
- void: metot geriye bir değer döndürmez.
- String[] args: program açılırken komut satırından verilen sözcükleri taşır; java Argumanlar Ali Veli diye açılan program "2 sözcük: Ali, Veli" yazar.



Şekil 1.4 Merhaba.java'nın main satırı: public, static, void ve String[] args'ın anlamları yanında.

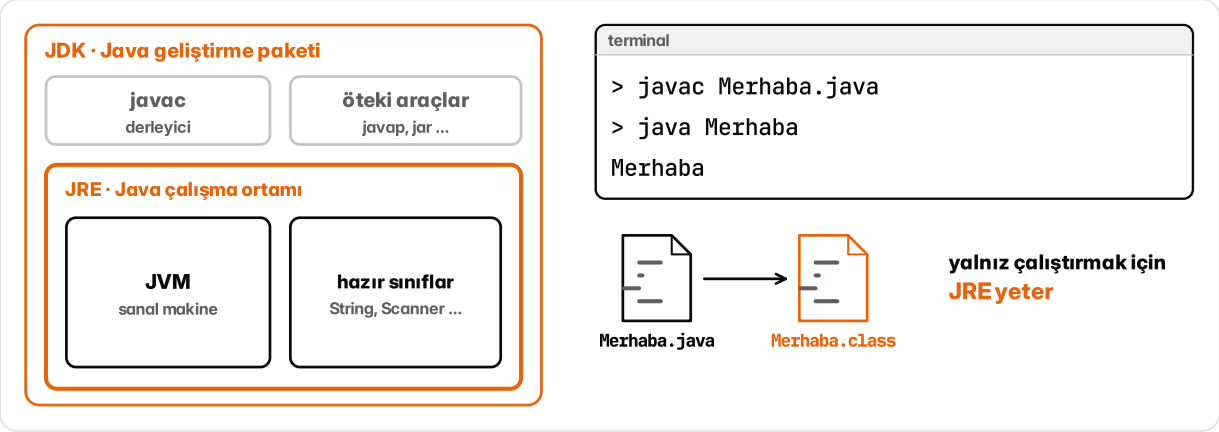
Gövdedeki tek satır ekrana yazar. System.out, programın standart çıktısını, yani ekranı gösterir. println ise verilen metni yazar ve alt satıra kendiliğinden geçer; alt satıra geçmeyen hâli print'tir:

```
public class Yazdir {
    public static void main(String[] args) {
        System.out.print("Mer");
        System.out.print("haba");
        System.out.println();
        System.out.println("Mer");
        System.out.println("haba");
    }
}
```

Çıktı:

```
Merhaba
Mer
haba
```

Bu dosyayı derlemek için bilgisayara **Java geliştirme paketi** (JDK) kurulur; içinde derleyici (javac) de sanal makine de vardır. Derleyici Merhaba.class dosyasını üretir, sanal makine onu çalıştırır ve ekrana Merhaba yazılır. Programı yalnızca çalıştıracak bir bilgisayara derleyici gerekmez; sanal makineyle hazır sınıfları içeren **çalışma ortamı** (JRE) yeter.



Şekil 1.5 JDK, derleyiciyi ve JRE'yi içerir; JRE, JVM ile hazır sınıflardan oluşur ve yalnız çalıştırmak için yeter.

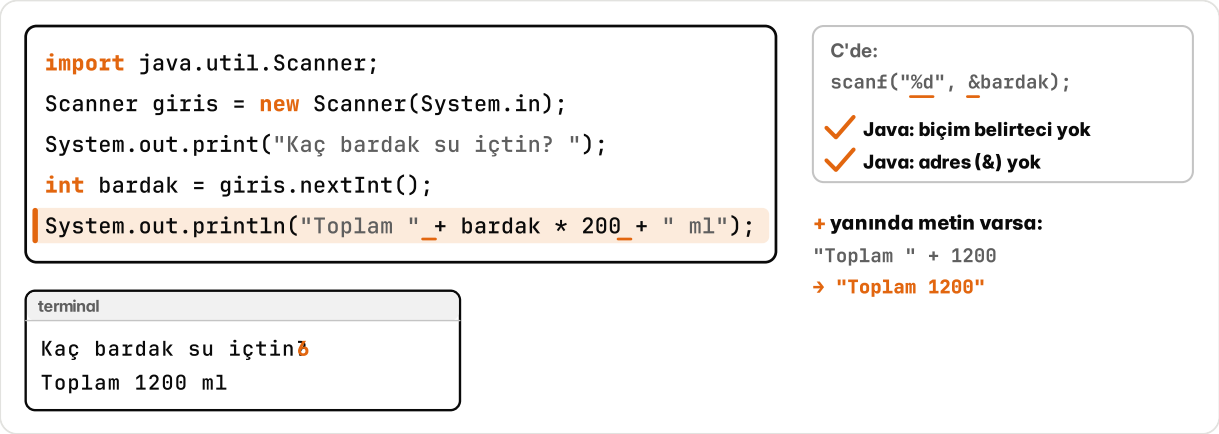
Klavyeden okumayı Scanner adlı hazır sınıf yapar; en üstteki import satırı onu programa tanıtır. Klavyeye bağlı yeni bir okuyucu nesne oluşturur, ondan bir tam sayı isteriz; program, kullanıcı sayıyı yazıp Enter tuşuna basana kadar bekler:

```
import java.util.Scanner;
Scanner giris = new Scanner(System.in);
System.out.print("Kaç bardak su içtin? ");
int bardak = giris.nextInt();
System.out.println("Toplam " + bardak * 200 + " ml");
```

Kullanıcı 6 yazıp Enter'a basınca ekranda şunlar görünür:

```
Kaç bardak su içtin? 6
Toplam 1200 ml
```

C'deki scanf'ten farkı şu: ne biçim belirteci ne de değişkenin adresi gerekir. + işareti, yanında metin varsa sayıyı da metne çevirip uç uca ekler; önce bardak * 200 = 1200 hesaplanır, sonra "Toplam " ile birleşir.



Şekil 1.6 Scanner ile okunan 6 bardak için Toplam 1200 ml; C'deki scanf'in biçim belirteci ve & işareti Java'da yok.

Gerçek projelerde bu komutları elle yazmayız; kodu bir **geliştirme ortamında** (IDE; örneğin IntelliJ IDEA, Eclipse, VS Code) yazarız. Geliştirme ortamı programı tek tuşla derleyip çalıştırır; sınıfsız yazılmış o tek satırı da daha derlemeye gelmeden kırmızıyla işaretler.

4 mü, 8 mi?

videoda 8:21

long tipinin kaç bayt olduğunu yazdıran aynı C programı Windows'ta 4, Linux'ta 8 yazar:

```
printf("%zu\n", sizeof(long));
```

Java'da aynı soru her bilgisayarda 8 cevabını alır:

```
System.out.println(Long.BYTES);
```

Aradaki fark nereden geliyor? C standardı tiplerin boyunu kesin söylemez, yalnızca en az kaç bit olacağını söyler (int en az 16, long en az 32 bit); gerisini derleyici ve işletim sistemi seçer. Java'da boylar dilin kendisinde yazılıdır: byte 8, short 16, int 32, long 64, float 32, double 64 bit, her makinede. Aynı hesap bu yüzden her bilgisayarda aynı sonucu verir.

C'de if (x = 5) satırı derlenir, çünkü orada koşul bir sayı olabilir: x'e 5 atanır ve koşul her seferinde doğru çıkar. Java'da koşul yalnızca doğru ya da yanlış olabilir; bunun için ayrı bir mantıksal tip, **boolean** vardır (değeri true ya da false) ve sayı ona kendiliğinden dönüşmez:

```
public class Kosul {
    public static void main(String[] args) {
        int x = 3;
        if (x = 5) {
            System.out.println("beş");
        }
    }
}
```

Derleyici 4. satırı reddeder: "incompatible types: int cannot be converted to boolean", yani tam sayı mantıksal tipe çevrilemez.

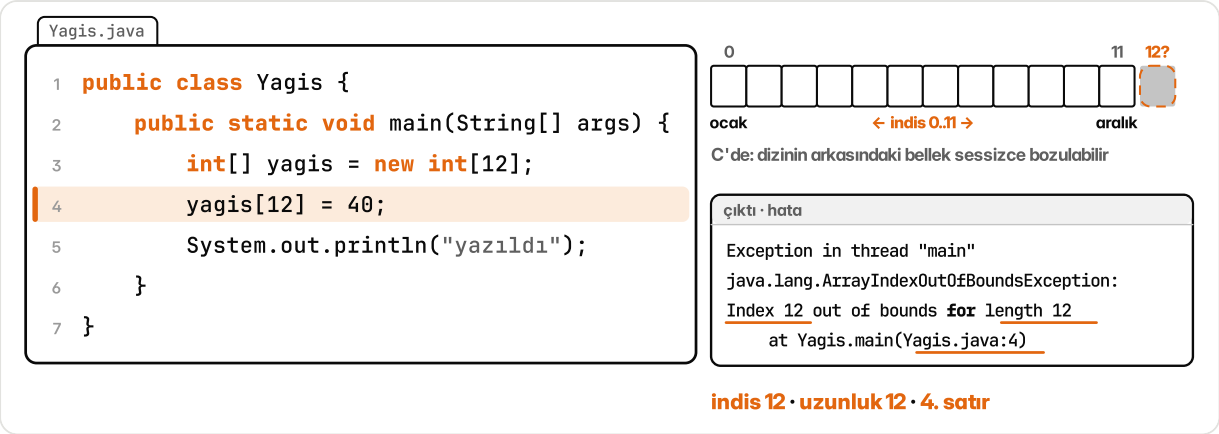
12 ayın yağışını tutan bir dizide ocak 0 numaralı indiste, aralık 11'dedir. 12 numaralı indise yazmak C'de dizinin arkasındaki belleği sessizce bozabilir. Java ise her dizi erişiminde sınırı denetler: program o satırda durur ve hatanın yerini söyler.

```
public class Yagis {
    public static void main(String[] args) {
        int[] yagis = new int[12];
        yagis[12] = 40;
        System.out.println("yazıldı");
    }
}
```

Çıktı:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 12 out of bounds
for length 12
    at Yagis.main(Yagis.java:4)
```

12 numaralı indis, uzunluğu 12 olan dizinin dışındadır; hata 4. satırdadır. "yazıldı" hiç yazılmaz.



Şekil 1.7 12 hücreli yağış dizisi (indis 0-11) ve yagis[12] satırında duran programın hata mesajı: indis 12, uzunluk 12, 4. satır.

Java'da metin, sonu sıfırla biten bir karakter dizisi değildir; her metin bir String nesnesidir. Uzunluğunu bulmak için sonundaki sıfırı aramak gerekmez, nesneye sorarız:

```
String sehir = "Eskişehir";
System.out.println(sehir.length());
```

Çıktı:

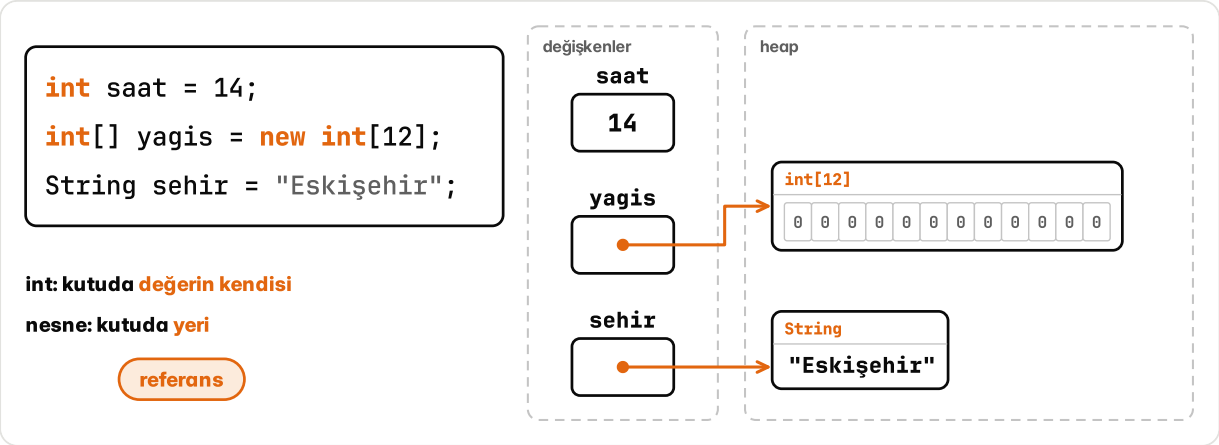
```
9
```

Arama (indexOf), parçalama (split), birleştirme (concat) gibi işleri de nesnenin kendi metotları yapar.

Bu fark bellekte de görünür:

```
int saat = 14;
int[] yagis = new int[12];
String sehir = "Eskişehir";
```

saat değişkeni int tipindedir; kutusunda değerin kendisi durur: 14. Dizi ve metin ise nesnedir; onların değişkeninde nesnenin kendisi değil, bellekte durduğu yer vardır. Değişkende duran bu yer bilgisine **referans** denir.



Şekil 1.8 saat kutusunda 14 durur; yagis ve sehir kutularından heap'teki dizi ve String nesnesine oklar, yani referanslar gider.

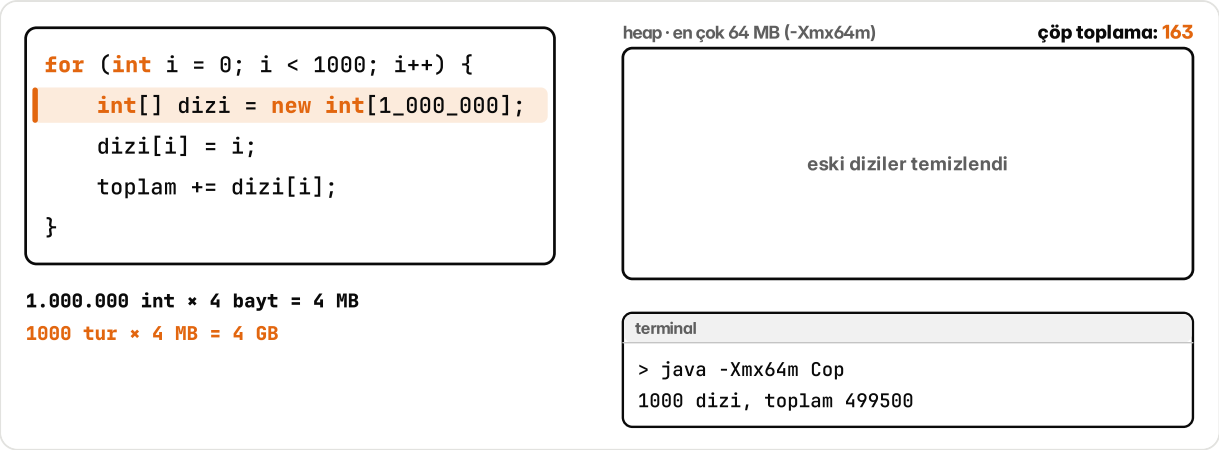
C'de malloc ile alınan belleği free ile geri vermek yazılımcının işidir; unutulursa bellek sızar. Java'da free yoktur: hiçbir referansın göstermediği nesneyi **çöp toplayıcı** (garbage collector) bulur ve belleğini kendisi geri alır. Bir döngüde 1000 kez, 4 megabaytlık yeni bir dizi oluşturalım (1.000.000 int × 4 bayt); toplam 4 gigabayt eder:

```
for (int i = 0; i < 1000; i++) {
    int[] dizi = new int[1_000_000];
    dizi[i] = i;
    toplam += dizi[i];
}
```

Program java -Xmx64m Cop komutuyla, yani 64 megabaytlık bellek sınırıyla çalıştırıldı ve sorunsuz bitti:

```
1000 dizi, toplam 499500
```

Çöp toplayıcı bu sürede 163 kez devreye girip eski dizileri temizledi.



Şekil 1.9 64 MB sınırlı heap'te 4 MB'lık diziler oluşur, eskileri temizlenir; çöp toplama sayacı 163.

Böylece C'de en sık yapılan iki hata, dizinin dışına yazmak ve belleği geri vermeyi unutmak, Java'da baştan engellenir.

Özet

- Java derleyicisi kaynak kodu bytecode içeren .class dosyasına çevirir; her sistemdeki JVM aynı dosyayı çalıştırır ve sık çalışan kodu JIT ile makine koduna çevirir.
- Java'da her kod bir sınıfın içindedir; public sınıfın adı dosyanın adıyla aynıdır ve program public static void main metodundan başlar.
- println yazıp alt satıra geçer, print geçmez; geliştirme için JDK, yalnız çalıştırmak için JRE gerekir.
- Scanner klavyeden okur; biçim belirteci ve adres gerekmez.
- Java'da tip boyları sabittir, koşul yalnız boolean olabilir, dizi sınırı her erişimde denetlenir ve metin bir String nesnesidir.
- Temel tipteki değişken değeri, dizi ve nesne değişkeni referansı tutar; kullanılmayan nesneleri çöp toplayıcı temizler.

Sorular

1 **javac Merhaba.java komutu hangi dosyayı üretir, içinde ne vardır?**

A merhaba.exe; Windows'un makine kodu

B Merhaba.class; hayali bir işlemci için yazılmış bytecode

C Merhaba.class; o bilgisayarın işlemcisinin makine kodu

D Merhaba.jvm; sanal makinenin kendisi

2 **Java sanal makinesinin anında derleme (JIT) ile yaptığı nedir?**

A Programı çalıştırmadan önce .java dosyasını yeniden derler.

B Bytecode'u her turda baştan çevirir.

C Çok tekrarlanan kısımları program çalışırken bir kez makine koduna çevirip saklar.

D Kullanılmayan nesnelerin belleğini geri alır.

3 **public static void main(String[] args) satırında static sözcüğü ne söyler?**

A Metodun değer döndürmediğini

B Metodun dışarıdan çağrılabileceğini

C Komut satırından verilen sözcükleri

D Metodu çağırmak için önce bir nesne oluşturmak gerekmediğini

4 **Aşağıdaki satırlar çalışınca ekranda ne görünür?**

```
System.out.print("A");
System.out.println("B");
System.out.print("C");
```

A AB ve alt satırda C

B A, B ve C ayrı satırlarda

C ABC tek satırda

D A ve alt satırda BC

5 **Bölümdeki su programında kullanıcı 4 yazıp Enter'a basıyor. Programın son satırda yazdığı metni yaz.**

```
int bardak = giris.nextInt();
System.out.println("Toplam " + bardak * 200 + " ml");
```

Cevap: _____

6 **int x = 3; if (x = 5) { ... } satırları Java'da ne olur?**

A Derlenir; x 5 olur ve koşul hep doğru çıkar.

B Derlenir; koşul yanlış çıkar, çünkü x 3'tür.

C Derlenmez; tam sayı mantıksal tipe (boolean) çevrilemez.

D Derlenir ama program o satırda durur.

7 **int[] yagis = new int[12]; tanımından sonra yagis[12] = 40; satırı çalışınca ne olur?**

A Dizinin arkasındaki bellek sessizce bozulur, program devam eder.

B Program o satırda durur; ArrayIndexOutOfBoundsException, indis 12 ve uzunluk 12 yazılır.

C Dizi kendiliğinden 13 elemana büyür.

D 40 değeri son elemana, yagis[11]'e yazılır.

8 **Aşağıdaki satırlar ekrana ne yazar?**

```
String sehir = "Kırşehir";
System.out.println(sehir.length());
```

Cevap: _____

NESNE YÖNELİMLİ PROGRAMLAMA

2 Sınıflar ve nesneler

Bu bölümde bir Araba sınıfı üzerinden nesnelerin bellekte nasıl doğduğunu görüyoruz: new'in yaptığı üç iş, referans, iki değişkenin aynı nesneyi göstermesi, == ile equals farkı, null ve NullPointerException, yapıcı metot, this ve aşırı yükleme.

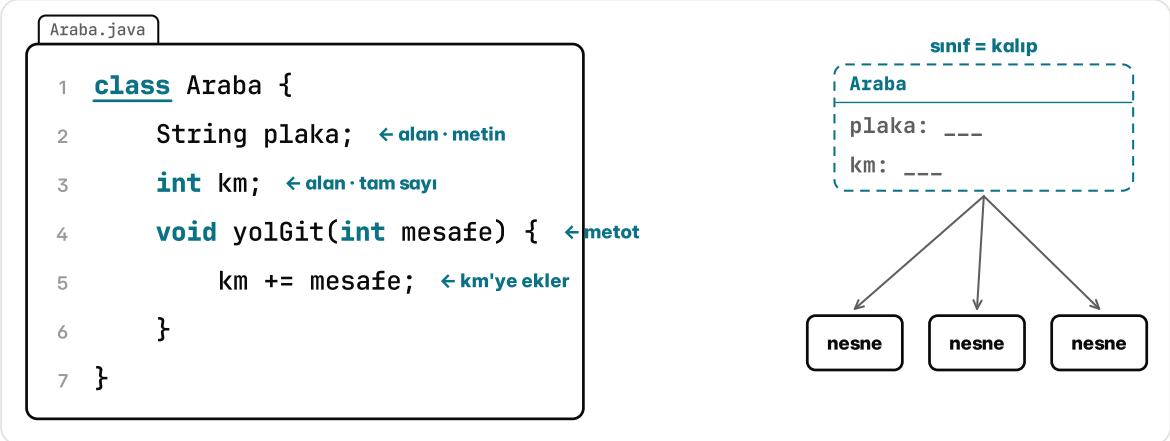
new yazınca?

videoda 10:28

Java'da yeni bir nesne new sözcüğüyle oluşturulur. Peki new yazınca bellekte tam olarak ne olur? Cevap için önce oluşturduğumuz şeyin tarifine bakalım: Araba sınıfına.

```
class Araba {
    String plaka;
    int km;
    void yolGit(int mesafe) {
        km += mesafe;
    }
}
```

Sınıf, bir türden nesnelerin neleri tutacağını ve neler yapabileceğini tanımlar. plaka ve km arabanın **alanlarıdır**, yani her arabanın taşıdığı veri: plaka bir metin (String), km bir tam sayı (int). yolGit ise bir metottur, yani arabanın yapabildiği bir iş: gidilen mesafeyi kilometreye ekler. Sınıf tek başına bir araba değildir, arabanın kalıbıdır; bu kalıptan istediğimiz kadar araba üretebiliriz.



Şekil 2.1 Araba sınıfı: plaka ve km alanları, yolGit metodu; sınıf bir kalıptır ve ondan nesneler üretilir.

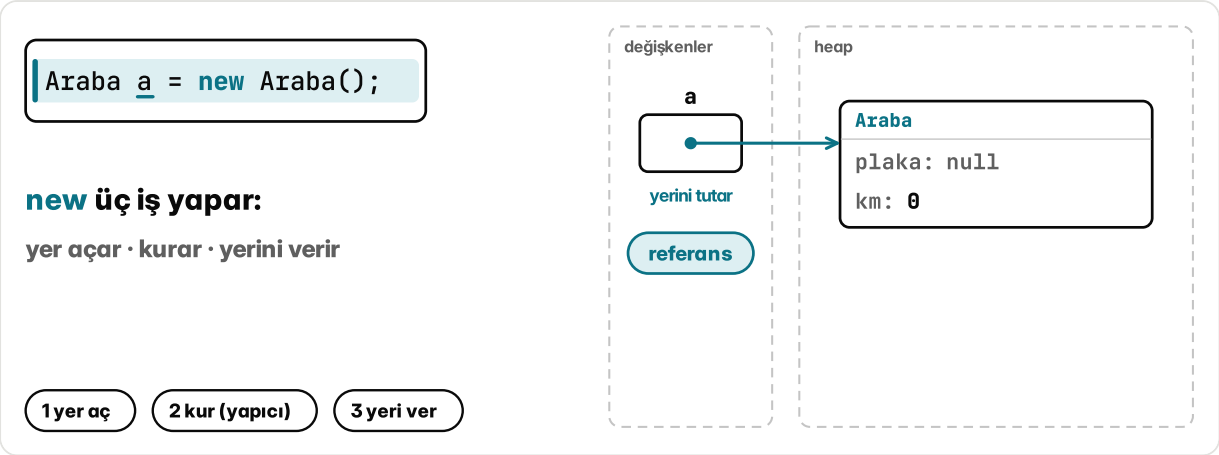
new, kalıptan gerçek bir araba üretir. Önce bellekte, **heap** denen bölgede, nesneye yetecek kadar yer açar; program çalışırken oluşturulan nesneler orada durur. Yeni açılan yerde her alan, tipinin varsayılan değeriyle başlar: tam sayı alanları 0, metin gibi nesne tutan alanlar null ile; yani km 0, plaka null. **null**, plakanın henüz hiçbir metni göstermediği anlamına gelir.

```
Araba a = new Araba();
System.out.println(a.plaka);
System.out.println(a.km);
```

Çıktı:

```
null
0
```

Sınıfta nesneyi kuran bir yapıcı metot varsa, o da bu sırada çalışır. En sonda new, nesnenin bellekteki yerini verir; a değişkenine yazılan bu yerdir. Kısacası new üç iş yapar: heap'te yer açar, nesneyi kurar ve yerini verir. a ise arabanın kendisini değil, nerede olduğunu tutar; bu yüzden onu heap'teki arabayı gösteren bir ok gibi düşünürüz. Böyle değişkenlere **referans** denir.



Şekil 2.2 new'in üç işi: yer aç, kur, yeri ver; a değişkeninden heap'teki Araba nesnesine giden ok referanstır.

Okun ucundaki nesneye nokta ile ulaşırız: a.plaka a'nın gösterdiği arabanın plakası, a.yolGit(120) da o arabanın metodudur. a'nın plakasına bir plaka yazıp a ile 120 kilometre yol gidince, metot a'nın gösterdiği arabanın kilometresini artırır:

```
Araba a = new Araba();
a.plaka = "06 HKZ 07";
a.yolGit(120);
System.out.println(a.km);
```

Çıktı:

```
120
```

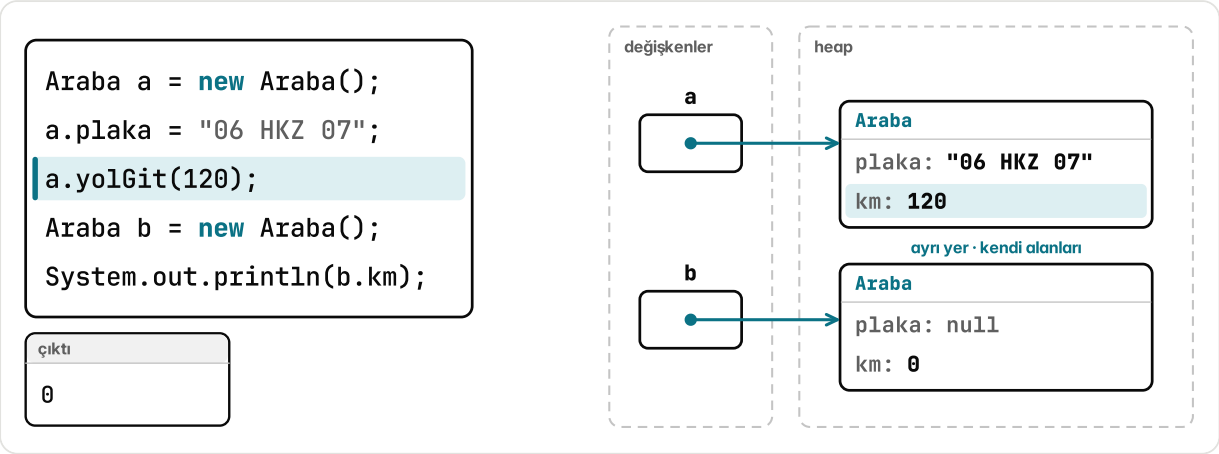
Aynı kalıptan ikinci bir araba üretirsek heap'te ayrı bir yer açılır ve b o yeri tutar:

```
Araba a = new Araba();
a.plaka = "06 HKZ 07";
a.yolGit(120);
Araba b = new Araba();
System.out.println(a.km);
System.out.println(b.km);
```

Çıktı:

```
120
0
```

Bu arabanın kilometresi 0'dır; a'nın 120'si ona geçmez. Her nesnenin alanları kendinedir; metot ise hangi nesne üzerinden çağrılırsa onun alanlarıyla çalışır.



Şekil 2.3 Heap'te iki ayrı Araba nesnesi: a'nınki 06 HKZ 07 plakalı ve 120 km, b'ninki null plakalı ve 0 km.

Kaç kilometre?

videoda 12:09

Bir arabayı gösteren a değişkenini b'ye atadık, sonra b ile 50 kilometre yol gittik. Şimdi a'nın kilometresine bakarsak kaç görürüz?

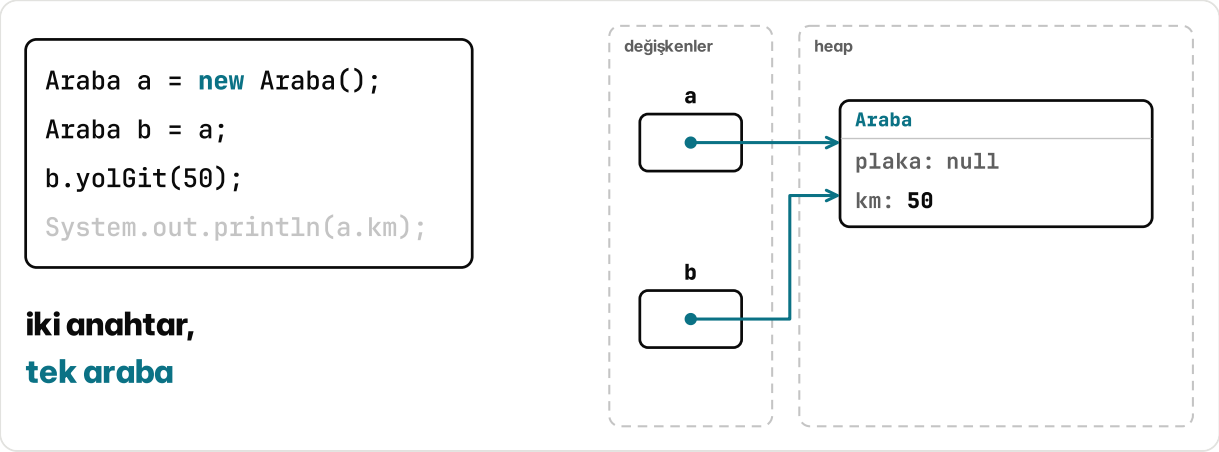
```
Araba a = new Araba();
Araba b = a;
b.yolGit(50);
System.out.println(a.km);
System.out.println(a == b);
```

Çıktı:

```
50
true
```

Cevap 0 değil, 50; çünkü ortada iki değil, tek bir araba var. Programda yalnız bir kez new yazıldı; ikinci bir new olmadıkça yeni nesne oluşmaz. Java'da nesne tutan bir değişken nesnenin kendisini değil, bellekteki yerini tutar. a'yı b'ye atamak arabayı kopyalamaz; yalnız a'daki yeri b'ye kopyalar.

Artık iki ok aynı arabayı gösterir; b ile yapılan değişiklik a ile bakınca da görünür. Bir arabanın iki anahtarı gibi: hangisiyle sürersek sürelim, kilometresi artan aynı arabadır.



Şekil 2.4 a ve b'nin okları heap'teki tek Araba nesnesine gider; km 50: iki anahtar, tek araba.

== de nesnelerin içine bakmaz; iki değişkenin aynı yeri tutup tutmadığına bakar. a ile b aynı yeri tuttuğu için sonuç true çıktı. new ile ayrı ayrı üretilmiş iki araba ise plakaları ve kilometreleri aynı olsa bile eşit çıkmaz: iki ayrı nesne, iki ayrı yer.

```
Araba a = new Araba();
Araba c = new Araba();
System.out.println(a == c);
```

Çıktı:

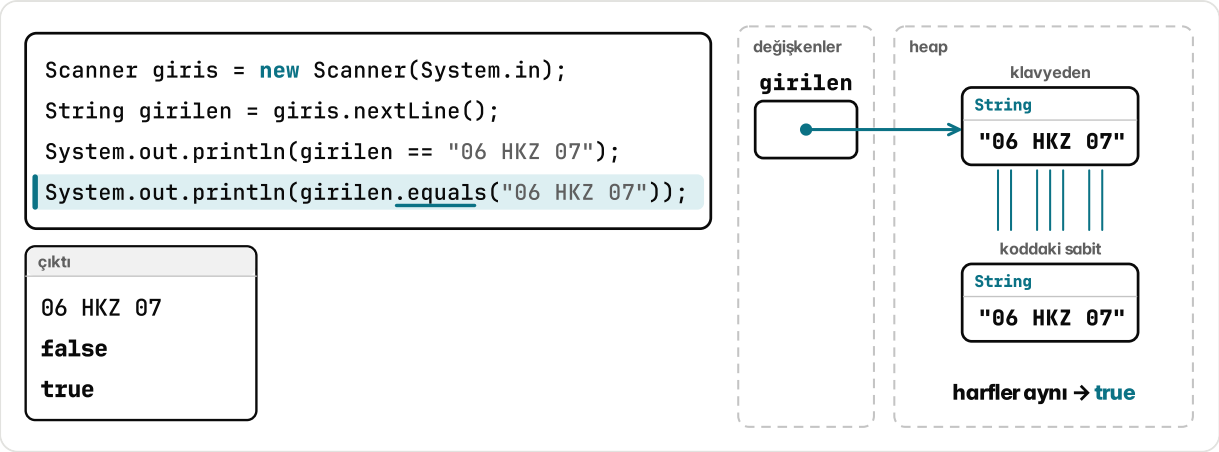
```
false
```

Bu kural metinlerde sinsi bir hataya yol açar, çünkü String de bir nesnedir. Kullanıcı plakayı klavyeden yazsın; Scanner okuduğu metin için bellekte yeni bir nesne oluşturur. Bu metni koddaki plakayla == ile karşılaştırınca harfler birebir aynı olduğu hâlde sonuç false çıkar. İçeriği karşılaştırmak içinse **equals** metodunu kullanırız; equals iki metnin harflerini sırayla karşılaştırır, hepsi aynıysa sonuç true çıkar.

```
Scanner giris = new Scanner(System.in);
String girilen = giris.nextLine();
System.out.println(girilen == "06 HKZ 07");
System.out.println(girilen.equals("06 HKZ 07"));
```

Kullanıcı 06 HKZ 07 yazınca çıktı:

```
false
true
```



Şekil 2.5 Klavyeden gelen ve koddaki "06 HKZ 07" heap'te iki ayrı String nesnesi; == false, equals true verir.

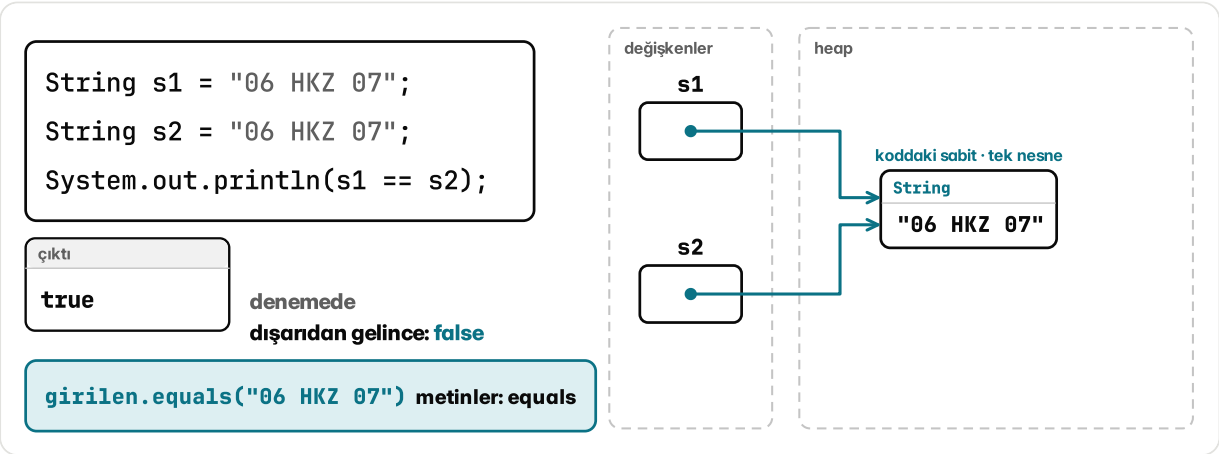
İşin sinsi yanı şu: koda aynı metni iki kez yazarsak, Java bellekte tek bir nesne tutar ve ikisi de onu gösterir.

```
String s1 = "06 HKZ 07";
String s2 = "06 HKZ 07";
System.out.println(s1 == s2);
```

Çıktı:

```
true
```

Bu yüzden koddaki iki sabitle yapılan denemede == true verir ve yöntem doğruymuş gibi görünür; hata ancak metin dışarıdan, klavyeden ya da bir dosyadan gelince ortaya çıkar. Kural basit: metinler karşılaştırılırken == değil, equals kullanılır.



Şekil 2.6 s1 ve s2 koddaki aynı sabiti, tek String nesnesini gösterir; == true çıkar, ama kural metinler için equals'tır.

Program neden durdu?

videoda 13:41

Yeni bir araba oluşturduk ama plakasını yazmayı unuttuk. Sonra plakası 06'yla mı başlıyor, yani Ankara plakası mı diye sorduk:

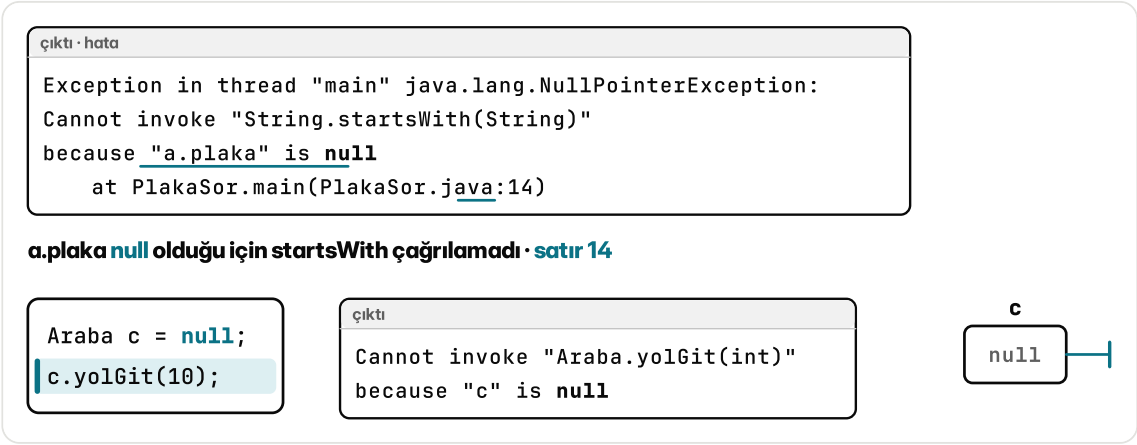
```
Araba a = new Araba();
System.out.println(a.plaka.startsWith("06"));
```

Program tam bu satırda durdu ve ekrana bir hata yazdı; neden? Plakaya hiç değer yazmadığımız için alan varsayılan değeriyle kaldı: null. null, bu alanın hiçbir nesneyi göstermediği demektir; ok hiçbir yere gitmez. startsWith ("ile mi başlıyor?") ise metnin bir metodudur; olmayan bir metnin metodu da çağrılamaz.

Java burada programı durdurur ve **NullPointerException** denen hatayı verir. Mesaj, hatanın hangi satırda çıktığını ve neyin boş olduğunu da söyler:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.startsWith(String)" because "a.plaka" is null
    at PlakaSor.main(PlakaSor.java:14)
```

a'nın plakası null olduğu için metot çağrılamadı. Henüz nesne yok demek için bir değişkene null'ı biz de verebiliriz (Araba c = null;); onu kullanmak da aynı hatayı verir: c.yolGit(10); satırı "Cannot invoke "Araba.yolGit(int)" because "c" is null" mesajıyla durur.



Şekil 2.7 NullPointerException mesajı: a.plaka null olduğu için startsWith çağrılamadı, satır 14; c = null için de aynı hata.

Hatanın kökü arabanın yarım kurulmasıdır; plakasız bir araba hiç oluşmamalıydı. Bunun yolu **yapıcı metottur** (constructor): new nesneye yer açtıktan hemen sonra çalışan ve nesneyi kuran metot. Adı sınıfın adıyla aynıdır ve dönüş tipi yoktur.

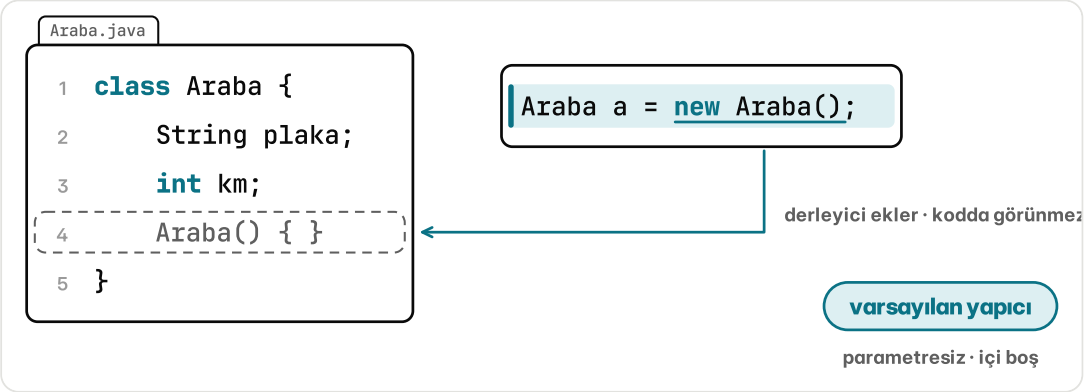
```
class Araba {
    String plaka;
    int km;

    Araba(String p) {
        plaka = p;
    }
}
```

Plaka artık araba oluşturulurken parantezin içinde verilir; aynı soru da hatasız çalışır ve true cevabını verir:

```
Araba a = new Araba("06 HKZ 07");
System.out.println(a.plaka.startsWith("06"));
```

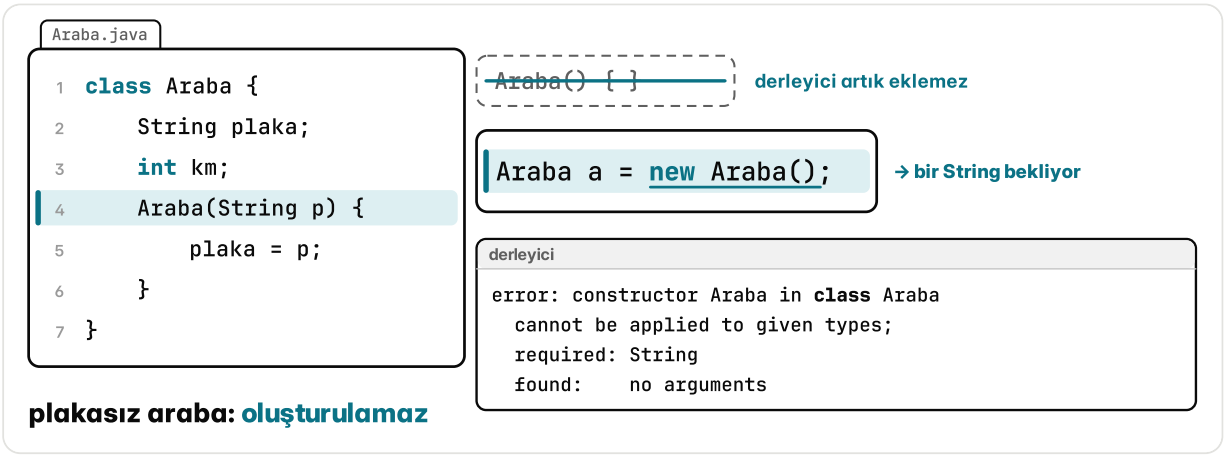
Dikkat: başına void yazarsak (void Araba(String p)) yapıcı olmaz; adı sınıfla aynı, sıradan bir metot olur.

Peki hiç yapıcı yazmadığımızda new Araba() nasıl çalışıyordu? Sınıfta hiç yapıcı yoksa derleyici parametresiz ve içi boş bir yapıcıyı, Araba() { }, kendisi ekler; buna **varsayılan yapıcı** denir.



Şekil 2.8 Yapıcısız Araba sınıfına derleyicinin eklediği, kodda görünmeyen Araba() { } satırı: varsayılan yapıcı.

Biz bir yapıcı yazınca derleyici bunu artık eklemez. Bu yüzden parantezi boş bırakıp plakasız bir araba istersek, Araba a = new Araba(); satırı derlenmez; derleyici bu yapıcının bir metin beklediğini söyler: "required: String", "found: no arguments". Bu bir kayıp değil, tam istediğimiz şey: plakasız araba artık hiç oluşturulamaz.



Şekil 2.9 Araba(String p) yazılınca varsayılan yapıcı eklenmez; new Araba() satırı "required: String" hatasıyla derlenmez.

Plaka neden boş?

videoda 15:14

Arabayı kuran yapıcı metodun parametrelerine alanlarla aynı adı verdik: plaka ve km.

```
class Araba {
    String plaka;
    int km;
    Araba(String plaka, int km) {
        plaka = plaka;
        km = km;
    }
}
```

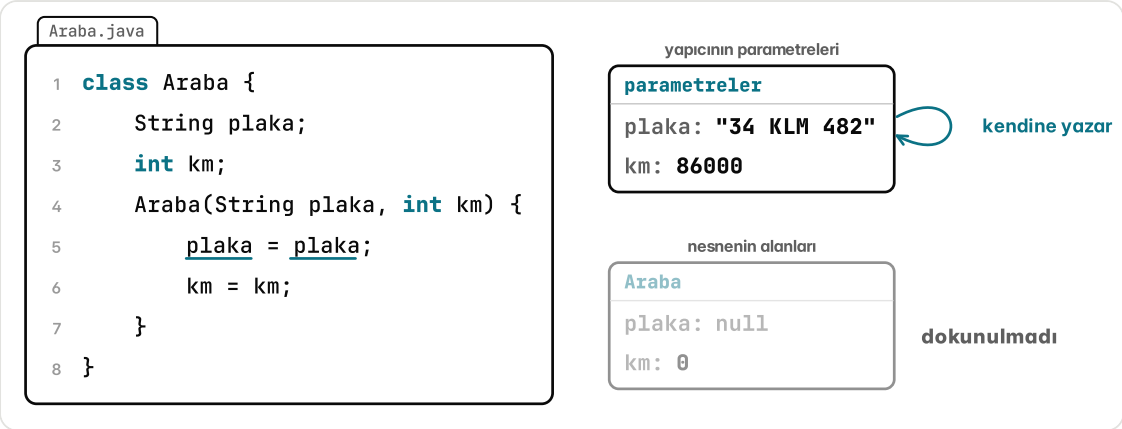
Kod hatasız derleniyor; ama ikinci el bir araba oluşturunca plakası null, kilometresi 0 çıkıyor:

```
Araba a = new Araba("34 KLM 482", 86000);
System.out.println(a.plaka);
System.out.println(a.km);
```

Çıktı:

```
null
0
```

Verdiğimiz değerler nereye gitti? Yapıcının içinde plaka denince Java en yakındaki adı, yani parametreyi anlar. Bu yüzden plaka = plaka satırı parametreyi alıp yine parametreye yazar; alana hiç dokunmaz. Alanlar da varsayılan değerlerinde kalır. Derleyici bu satır için uyarı bile vermez; yanlışlık ancak çıktıya bakınca fark edilir.

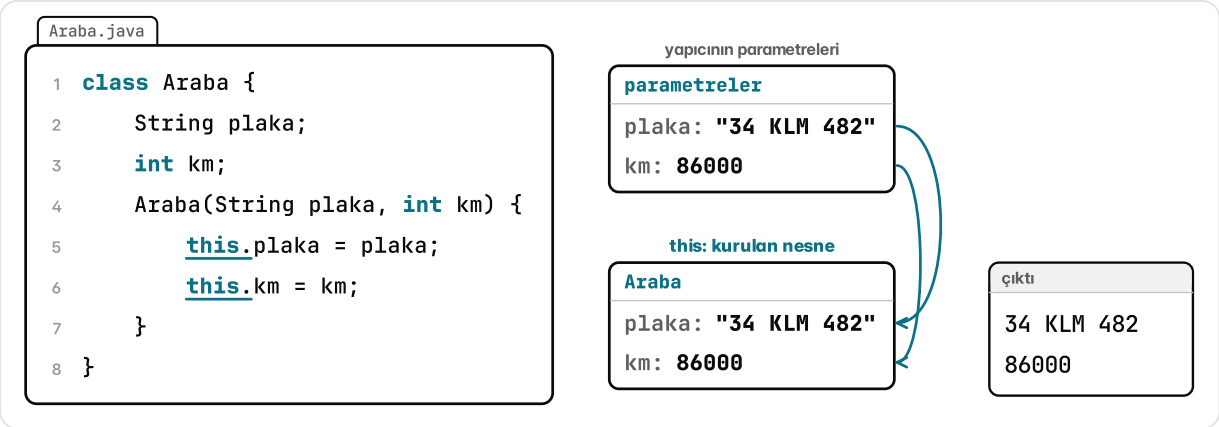


Şekil 2.10 plaka = plaka parametreyi kendine yazar; nesnenin alanları null ve 0 kalır, dokunulmaz.

Alanı göstermenin yolu **this** sözcüğüdür: this, o an kurulan ya da metodu çağrılan nesnenin kendisidir. this.plaka nesnenin alanıdır, tek başına plaka ise parametredir:

```
Araba(String plaka, int km) {
    this.plaka = plaka;
    this.km = km;
}
```

Böylece parametreye anlamlı bir ad verir, alanla karıştırmadan kullanırız; aynı main ile ekrana 34 KLM 482 ve 86000 doğru çıkar.



Şekil 2.11 this.plaka ve this.km ile parametreler kurulan nesnenin alanlarına yazılır; çıktı 34 KLM 482 ve 86000.

Bayiden yeni çıkan araba ise sıfır kilometredir; her seferinde km'ye 0 yazmak gereksiz. Bunun için yalnız plaka alan ikinci bir yapıcı yazarız:

```
Araba(String plaka, int km) {
    this.plaka = plaka;
    this.km = km;
}

Araba(String plaka) {
    this.plaka = plaka;
    this.km = 0;
}
```

Aynı adı taşıyıp parametreleri farklı olan yapıcılara ve metotlara **aşırı yükleme** (overload) denir. Derleyici hangisinin çalışacağını verilen değerlerin sayısına ve tipine bakarak seçer: new Araba("06 HKZ 07") tek parametreli, new Araba("34 KLM 482", 86000) iki parametreli yapıcıyı çağırır. Ekrana yazdırırken de bunu kullanırız: println tam sayı için de metin için de ayrı ayrı yazılmıştır; System.out.println(86000) println(int)'i, System.out.println("34 KLM 482") println(String)'i çalıştırır.

Kurma işi iki yerde yazılmasın diye kısa yapıcı uzun olanı this ile çağırır ve km'ye 0 verir:

```
Araba(String plaka) {
    this(plaka, 0);
}
```

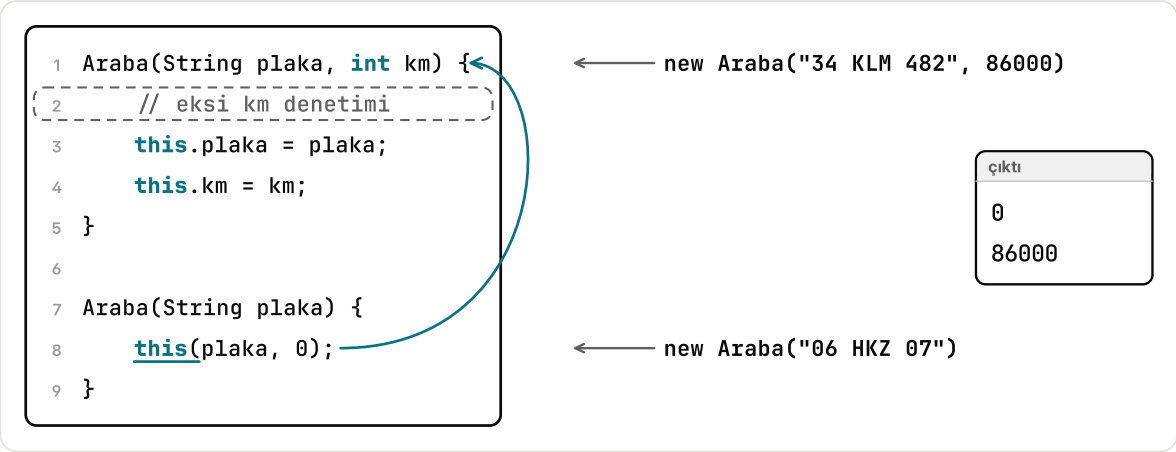
this burada nokta değil parantez alır: aynı sınıfın öteki yapıcısını çağırmak demektir. Böylece kurma işi tek yerde kalır; yarın eksi kilometreyi engelleyen bir denetim eklersek iki yapıcıya birden işlemiş olur.

```
Araba yeni = new Araba("06 HKZ 07");
Araba ikinciEl = new Araba("34 KLM 482", 86000);
System.out.println(yeni.km);
System.out.println(ikinciEl.km);
```

Çıktı:

```
0
86000
```

Sıfır kilometre araba da ikinci el araba da aynı yapıcıdan geçerek kurulur.



Şekil 2.12 Kısa yapıcı this(plaka, 0) ile uzun yapıcıyı çağırır; iki new de aynı yapıcıdan geçer, çıktı 0 ve 86000.

Özet

- Sınıf alanları ve metotları tanımlayan kalıptır; new heap'te yer açar, nesneyi kurar ve yerini verir, alanlar 0 ya da null ile başlar.
- Nesne değişkeni nesnenin kendisini değil yerini, yani referansı tutar; b = a ile iki değişken aynı nesneyi gösterir.
- == iki değişkenin aynı yeri tutup tutmadığına bakar; metinlerin içeriği equals ile karşılaştırılır.
- null hiçbir nesneyi göstermez; null üzerinden metot çağırmak NullPointerException verir.
- Yapıcı metot sınıfla aynı adı taşır, dönüş tipi yoktur; hiç yapıcı yoksa derleyici varsayılan yapıcıyı ekler, biri yazılınca eklemez.
- this nesnenin kendisidir; aynı adla farklı parametreli yapıcılar aşırı yüklemedir ve this(...) aynı sınıfın öteki yapıcısını çağırır.

Sorular

1 **Araba a = new Araba(); satırında a değişkeninde ne durur?**

- **A** Araba nesnesinin kendisi
- **B** Araba nesnesinin heap'teki yeri, yani referans
- **C** Araba sınıfının bir kopyası
- **D** Nesnenin ilk alanının değeri

2 **Bölümdeki Araba sınıfıyla aşağıdaki satırlar ekrana ne yazar?**

```
Araba a = new Araba();
Araba b = a;
a.yolGit(30);
b.yolGit(20);
System.out.println(a.km);
```

Cevap: _____

3 **Araba a = new Araba(); Araba c = new Araba(); System.out.println(a == c); satırları ne yazar?**

A true, çünkü iki arabanın alanları aynı

B Derlenmez, nesneler == ile karşılaştırılamaz

C false, çünkü iki ayrı nesne iki ayrı yerdedir

D null

4 **Klavyeden okunan girilen metninin "06 HKZ 07" olup olmadığına doğru biçimde nasıl bakılır?**

A girilen.equals("06 HKZ 07")

B girilen == "06 HKZ 07"

C girilen = "06 HKZ 07"

D "06 HKZ 07" == girilen

5 **Yapıcısız Araba sınıfıyla Araba a = new Araba(); System.out.println(a.plaka.length()); satırları çalışınca ne olur?**

A 0 yazar

B null yazar

C Satır derlenmez

D Program durur ve NullPointerException verir

6 **Araba sınıfında yalnız Araba(String p) yapıcısı var. Aşağıdaki satırlardan hangisi derlenmez?**

A Araba a = new Araba("06 HKZ 07");

B Araba a = new Araba();

C Araba a = null;

D Araba b = new Araba("34 KLM 482");

7 **Aşağıdaki sınıf ve satırlar çalışınca ikinci println ne yazar?**

```
class Araba {
    String plaka;
    int km;
    Araba(String plaka, int km) {
        plaka = plaka;
        km = km;
    }
}
Araba a = new Araba("34 KLM 482", 86000);
System.out.println(a.plaka);
System.out.println(a.km);
```

Cevap:

8 **Araba sınıfında Araba(String plaka, int km) ve Araba(String plaka) yapıcıları var. new Araba("06 HKZ 07") hangisini çalıştırır ve neden?**

A İki parametreliyi, çünkü sınıfta önce o yazılmıştır

B İkisini de sırayla

C Tek parametreliyi, çünkü derleyici verilen değerlerin sayısına ve tipine bakar

D Hiçbirini; aynı adlı iki yapıcı derlenmez

NESNE YÖNELİMLİ PROGRAMLAMA

3 Kapsülleme ve static

Bu bölümde bir banka hesabının bakiyesini korumayı görüyoruz: private ile alanı kapatmak, kuralı tek bir metotta toplamak, get ve set tuzağı, final ve değişmez nesneler; sonra paketler, import ve dört erişim belirleyici; en son nesneye değil sınıfa ait olan static alan ve metotlar.

300 lira, 500 lira

videoda 16:44

Hesabında 300 lira olan biri 500 lira çekmek istiyor. Banka yazılımı bakiyenin -200'e düşmesini nasıl engeller?

Bakiye alanı herkese açıksa, yani başında public varsa, programın herhangi bir satırı ona doğrudan yazabilir:

```
public class Hesap {
    public int bakiye;
}
```

Başka bir sınıfın içinde:

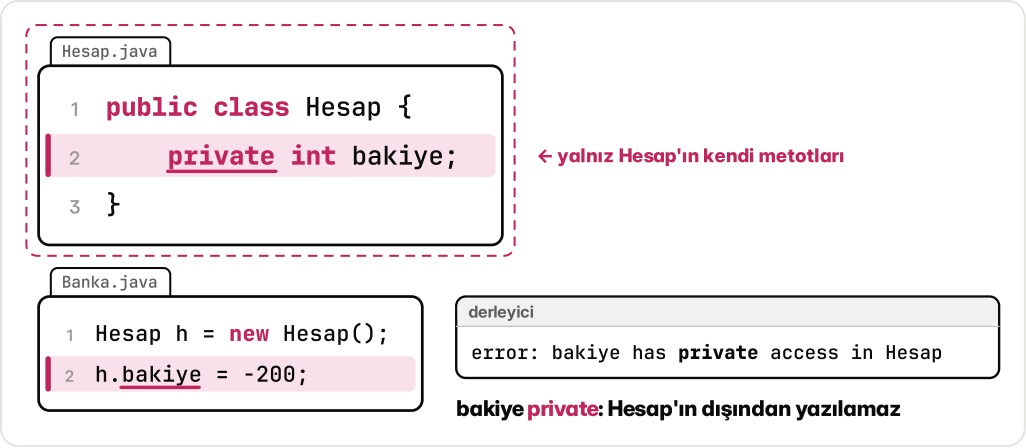
```
Hesap h = new Hesap();
h.bakiye = 300;
h.bakiye = h.bakiye - 500;
System.out.println(h.bakiye);
```

Çıktı:

```
-200
```

500 lirayı düşen satır hiçbir kontrolden geçmedi. Bunu önlemek için kontrolü para çekilen her yere ayrı ayrı koymak gerekir: bankamatiğe, mobil uygulamaya, şubedeki programa. Biri unutulunca bakiye yine eksiye düşer.

Çözüm, alanı dışarıya kapatmaktır: alanın başına **private** yazarız. Artık bakiyeye yalnız Hesap sınıfının kendi metotları dokunabilir. Başka bir sınıftan h.bakiye = -200; diye yazan satır derlenmez bile; derleyici alanın dışarıya kapalı olduğunu söyler: "bakiye has private access in Hesap".



Şekil 3.1 bakiye private olunca Banka.java'daki h.bakiye = -200; satırı "bakiye has private access in Hesap" hatasıyla derlenmez.

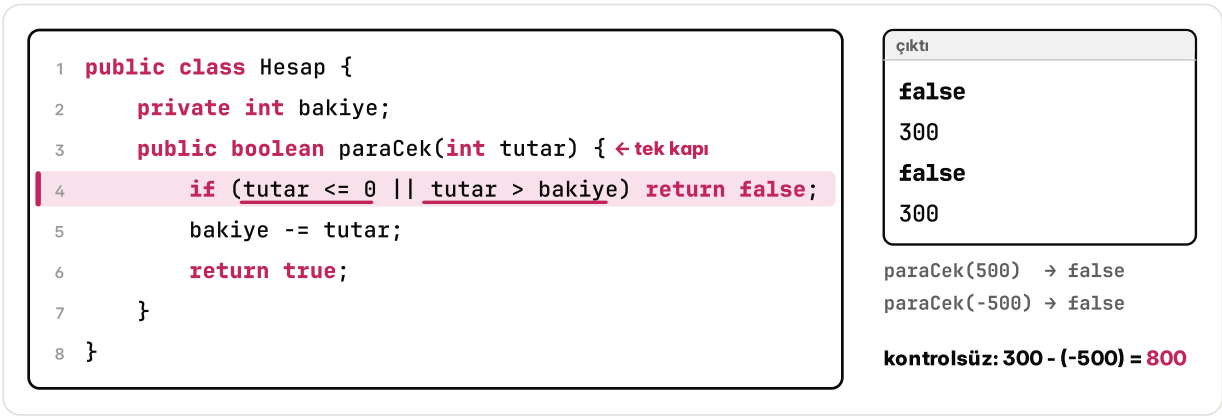
Bakiyeyi değiştirmenin tek yolu artık sınıfın açtığı metotlardır: para yatırmak ve para çekmek. Para çekme metodu önce tutara bakar; tutar bakiyeden büyükse işlemi reddeder:

```
public class Hesap {
    private int bakiye;
    public boolean paraCek(int tutar) {
        if (tutar <= 0 || tutar > bakiye) return false;
        bakiye -= tutar;
        return true;
    }
}
```

Dosyada ayrıca paraYatir ve bakiyeyi okuyan getBakiye metotları da var. 300 yatırılan hesaptan önce 500, sonra -500 çekilmek istenince, her denemenin ardından bakiye yazdırılır:

```
false
300
false
300
```

Bakiye 300'de kalır. Eksi tutarı da reddeder; yoksa -500 lira çekmek, yani bakiye -= -500, hesaba 500 lira eklerdi.



Şekil 3.2 paraCek tek kapıdır: 500 ve -500 isteklerine false döner, bakiye 300'de kalır; kontrolsüz olsa 300 - (-500) = 800 olurdu.

Buna **kapsülleme** (encapsulation) denir: veri sınıfın içinde saklanır, dışarıya yalnız kuralı olan kapılar açılır. Kural değişince düzeltilecek yer de tektir. Banka 1.000 liralık ek hesap tanırsa yalnız bu koşula 1.000 eklenir: tutar > bakiye + 1000. Bankamatik de, mobil uygulama da yeni kuralla çalışır.

Bakiyeyi okumak için de alanı açmayız; değeri geri veren küçük bir metot yazarız. Adı get ile başladığı için buna **get metodu** denir. Sık yapılan hata, her alana bir de set metodu yazmaktır:

```
public class Hesap {
    private int bakiye;
    public int getBakiye() { return bakiye; }
    public void setBakiye(int yeni) { bakiye = yeni; }
}
```

Bakiyeye set yazmak kapattığımız kapıyı yeniden açar: h.setBakiye(-500); çağrısından sonra getBakiye() -500 verir. Dışarıya yalnız gereken işlemler açılır: paraYatir ve paraCek.

Bazı alanlar ise hiç değişmemeli; hesap numarası gibi. Böyle bir alanın başına **final** yazarız. Alan değerini hesap oluşturulurken bir kez alır; sonra onu değiştirmeye çalışan satır derlenmez:

```
public class Hesap {
    private final int no;
    private int bakiye;
    public Hesap(int hesapNo) { no = hesapNo; }
    public void noDegistir(int yeni) { no = yeni; }
}
```

Derleyici 5. satır için "cannot assign a value to final variable no" der.

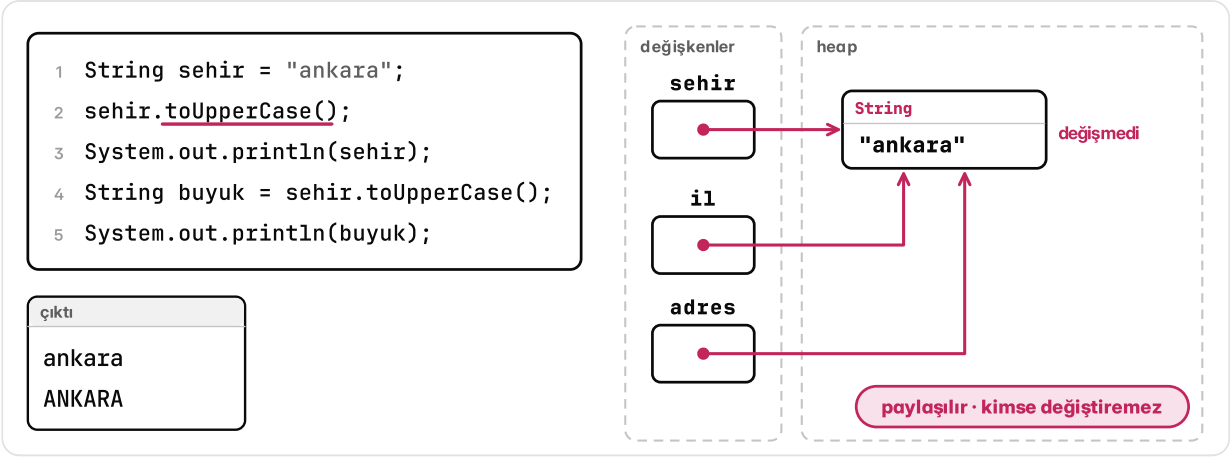
Oluşturulduktan sonra hiç değişmeyen nesneye **değişmez nesne** denir. Java'da String'ler de böyledir. Büyük harfe çevirme metodu toUpperCase eldeki String'i değiştirmez; büyük harfli yeni bir kopya üretip geri verir:

```
String sehir = "ankara";
sehir.toUpperCase();
System.out.println(sehir);
String buyuk = sehir.toUpperCase();
System.out.println(buyuk);
```

Çıktı:

```
ankara
ANKARA
```

Sonucu bir değişkene atamayınca ankara yazısı olduğu gibi kalır. Bu sayede aynı metin pek çok yerde güvenle paylaşılır, çünkü kimse onu değiştiremez.



Şekil 3.3 "ankara" String nesnesi toUpperCase'ten sonra da değişmez; sehir, il ve adres aynı nesneyi güvenle paylaşır.

Aynı ad, iki sınıf

videoda 18:46

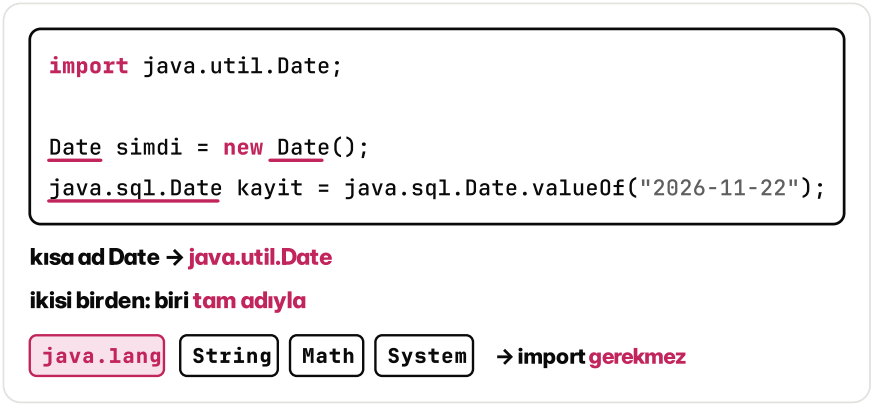
Java'nın kendi kütüphanesinde aynı adı taşıyan iki ayrı sınıf var; ikisinin de adı Date ve ikisi de tarih tutuyor. Kodda bu adı yazdığımızda hangisini kastettiğimizi derleyici nereden bilir?

Java'da her sınıf bir pakette durur; **paket**, birbiriyle ilgili sınıfları bir araya toplayan bir klasördür. Paketin adı sınıfın soyadı gibidir. Biri yardımcı araçlar paketinde (java.util.Date), öteki veritabanı paketinde (java.sql.Date) durur; tam adlarıyla yazılınca hiç karışmazlar. Kendi sınıflarımızı da dosyanın ilk satırına paket adını yazarak bir pakete koyarız: package banka;.

Tam adı her seferinde yazmamak için dosyanın başına import satırı ekleriz. Derleyici, o dosyada kısa adın hangi sınıfı gösterdiğini buradan öğrenir. İkisi birden gerekirse birini tam adıyla yazarız:

```
import java.util.Date;

Date simdi = new Date();
java.sql.Date kayit = java.sql.Date.valueOf("2026-11-22");
```

İkisini birden import etmek ise derlenmez; derleyici Date adının belirsiz olduğunu söyler. En temel sınıfların durduğu java.lang paketi (String, Math, System) import istemez; her dosyada kendiliğinden tanınır.

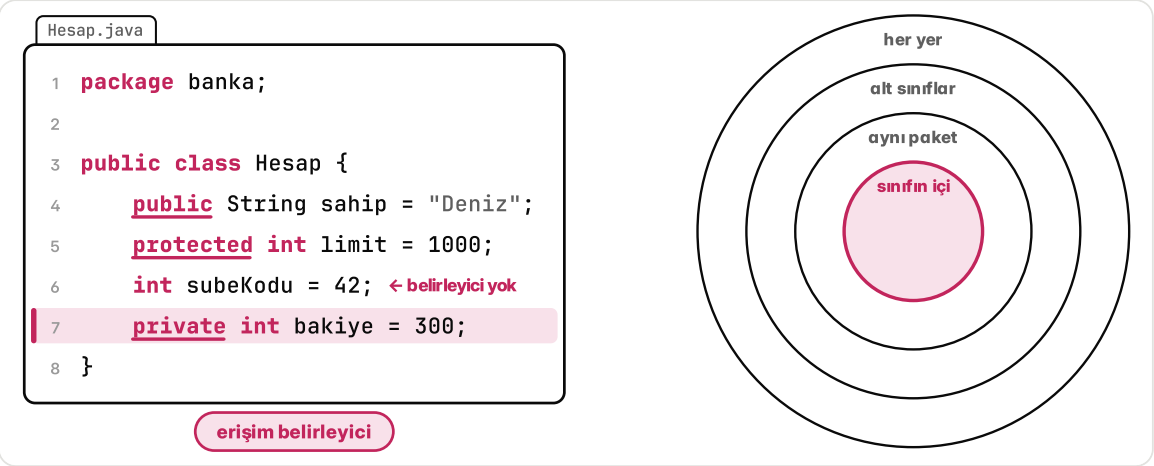


Şekil 3.4 import java.util.Date ile kısa ad Date java.util.Date olur; ikincisi tam adıyla yazılır, java.lang import istemez.

Paket erişimin de sınırını çizer. Bir alana ya da metoda kimin ulaşacağını başına yazılan sözcük belirler; bunlara **erişim belirleyici** denir. Aşağıdaki sınıfta dördü de var:

```
package banka;

public class Hesap {
    public String sahip = "Deniz";
    protected int limit = 1000;
    int subeKodu = 42;
    private int bakiye = 300;
}
```

- public yazılırsa her yerden ulaşılır.
- Hiçbir şey yazılmazsa yalnız aynı paketteki sınıflar ulaşır; buna **paket erişimi** denir.
- private yazılırsa yalnız sınıfın kendi içinden ulaşılır.
- protected aynı paketin yanında, başka paketlerdeki alt sınıflara da açıktır. Alt sınıf, bir sınıftan kalıtımla türetilen yeni bir sınıftır.

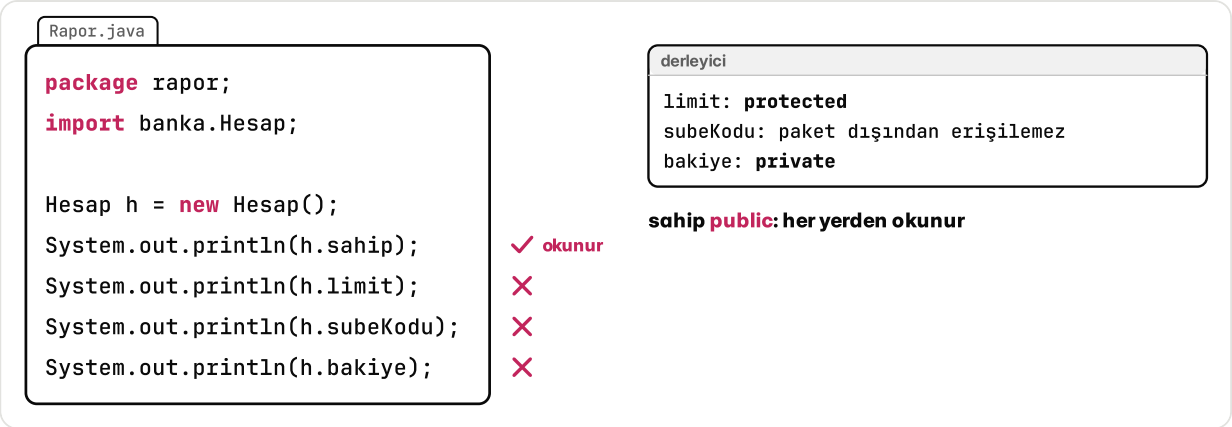


Şekil 3.5 Hesap'ın dört alanı ve iç içe erişim halkaları: sınıfın içi, aynı paket, alt sınıflar, her yer.

Başka paketteki bir rapor sınıfı hesabın sahibini okuyabilir; öteki üç alana dokunan satırlar ise derlenmez:

```
package rapor;
import banka.Hesap;

Hesap h = new Hesap();
System.out.println(h.sahip);
System.out.println(h.limit);
System.out.println(h.subeKodu);
System.out.println(h.bakiye);
```

Derleyici üç hata verir: limit protected, subeKodu paket dışından erişilemez, bakiye private. Aynı paketteki bir sınıf ise sahip, limit ve subeKodu'nu okur (Deniz, 1000, 42).



Şekil 3.6 Rapor.java'da yalnız h.sahip okunur; limit, subeKodu ve bakiye satırları derleyici hatası verir.

Yaygın alışkanlık şudur: alanların hepsi private, dışarının gerçekten kullanacağı metotlar ise public olur. Kapalı bir parça sonradan rahatça değiştirilir, çünkü ona bağlı dış kod yoktur. public bir metodu değiştirmek ise onu kullanan her yeri bozabilir.

Hesap numarası

videoda 20:17

Banka açılan her hesaba sıradaki numarayı veriyor: 1, 2, 3. Hesap sınıfına bir sayaç alanı koyduk; her yeni hesap oluşturulurken çalışan yapıcı metot sayacı bir artırıp numarayı ondan alıyor:

```
public class Hesap {
    private int sayac = 0;
    private int no;
    public Hesap() {
        sayac++;
        no = sayac;
    }
}
```

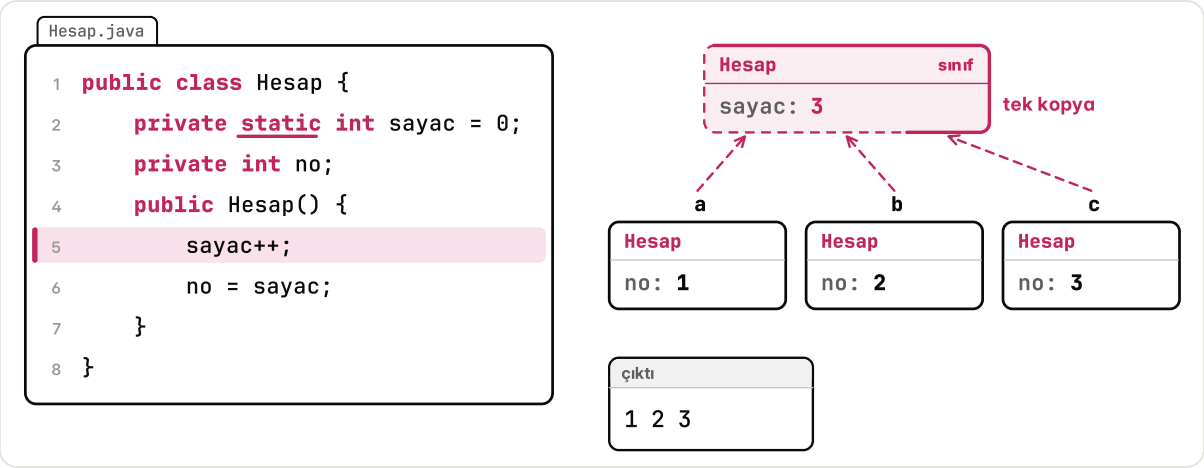
Ama üç hesap açıp numaralarını yazdırınca üçünün de numarası 1 çıkıyor:

```
1 1 1
```

Neden? Çünkü alanlar nesneye aittir: her hesap kendi bakiyesini taşıdığı gibi kendi sayacını da taşır. Her yeni hesap 0'dan başlayan yepyeni bir sayaçla doğar ve onu 1'e çıkarır.

Sayacın başına **static** yazınca alan nesneye değil, sınıfın kendisine ait olur: private static int sayac = 0;. Bellekte tek bir kopyası vardır ve bütün hesaplar onu paylaşır. Her yeni hesap aynı sayacı artırır; numaralar 1, 2, 3 olur:

```
1 2 3
```



Şekil 3.7 static sayac Hesap sınıfında tek kopya; a, b ve c onu sırayla artırır, numaraları 1, 2, 3 olur.

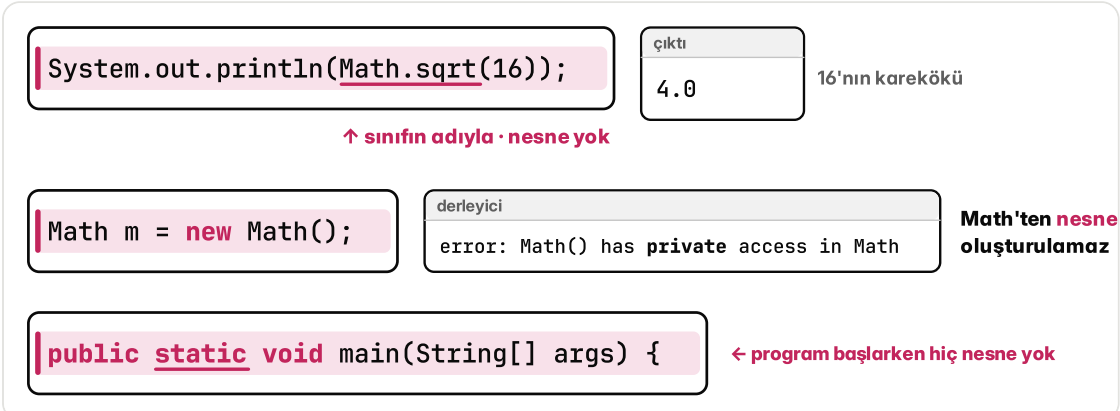
Metotlar da static olabilir; böyle bir metot hiçbir nesneye bağlı değildir, işini yalnız aldığı değerlerle yapar. Karekök iyi bir örnektir: 16'nın karekökünü almak için bir nesneye gerek yoktur. Bu yüzden Math sınıfının karekök metodunu nesne oluşturmadan sınıfın adıyla çağırırız:

```
System.out.println(Math.sqrt(16));
```

Çıktı:

```
4.0
```

O sınıftan nesne oluşturmak zaten yasaktır: Math m = new Math(); satırı "Math() has private access in Math" hatasıyla derlenmez. Programın başladığı main metodu da static yazılır, çünkü program başlarken ortada hiç nesne yoktur.

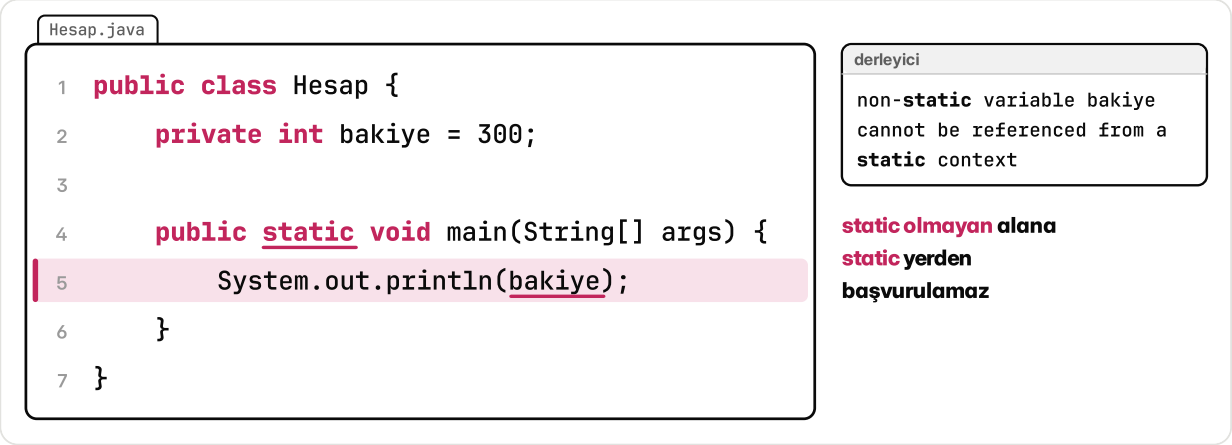


Şekil 3.8 Math.sqrt(16) sınıfın adıyla çağrılır ve 4.0 verir; new Math() derlenmez; main de static'tir.

Bunun bir bedeli de var. static bir metodun içinden bakiye gibi bir nesne alanına doğrudan ulaşılamaz. main'in içinde bakiyeyi yazdırmaya kalkınca derleyici durur:

```
public class Hesap {
    private int bakiye = 300;

    public static void main(String[] args) {
        System.out.println(bakiye);
    }
}
```

Mesaj "non-static variable bakiye cannot be referenced from a static context", yani static olmayan bir alana static bir yerden başvurulamaz.



Şekil 3.9 main'in içindeki println(bakiye) satırı için derleyici: static olmayan alana static yerden başvurulamaz.

Çünkü static bir metot belli bir nesne için değil, sınıf için çağrılır. Bellekte 1.000 hesap olabilir; hangisinin bakiyesini istediğimizi bilemez. Çözüm nesneyi açıkça vermektir: bir hesap oluşturup onun bakiyesini isteriz.

```
public static void main(String[] args) {
    Hesap h = new Hesap();
    System.out.println(h.bakiye);
}
```

Ekrana 300 yazılır. main aynı sınıfın içinde olduğu için private alana erişebilir. Kısacası static olanlar sınıfa, ötekiler her bir nesneye aittir.

Özet

- private alan yalnız kendi sınıfının içinden kullanılır; dışarıdan yazan satır derlenmez.
- Kapsüllemede veri içeride saklanır, değişiklik kuralı olan tek bir metottan geçer; kural değişince düzeltilecek yer tektir.
- Her alana set yazmak kapıyı yeniden açar; final alan bir kez değer alır, String gibi değişmez nesneler hiç değişmez.
- Paket sınıfın soyadıdır, import kısa adın anlamını verir; public her yer, protected paket ve alt sınıflar, belirleyicisiz aynı paket, private yalnız sınıfın içidir.
- static alan sınıfa ait tek kopyadır; static metot nesne olmadan sınıfın adıyla çağrılır.
- static bir metottan nesne alanına doğrudan ulaşılamaz; hangi nesnenin alanı istendiği açıkça verilir.

Sorular

1 **Hesap sınıfında bakiye private int bakiye; diye tanımlı. Başka bir sınıftaki h.bakiye = -200; satırı ne olur?**

A Derlenir; bakiye -200 olur.

B Derlenir ama program o satırda durur.

C Derlenmez; bakiye Hesap'ın dışına kapalıdır.

D Derlenir; satır yok sayılır, bakiye değişmez.

2 **Bölümdeki tek kapılı Hesap sınıfıyla (paraCek'te koşul: tutar <= 0 || tutar > bakiye) aşağıdaki satırlar ekrana ne yazar?**

```
Hesap h = new Hesap();
h.paraYatir(300);
h.paraCek(100);
h.paraCek(500);
System.out.println(h.getBakiye());
```

Cevap: _____

3 **Bakiyeyi koruyan Hesap sınıfına public void setBakiye(int yeni) { bakiye = yeni; } metodunu eklemek neden hatadır?**

A Kapattığımız kapıyı yeniden açar; dışarıdan bakiyeye -500 gibi her değer yazılabilir.

B set metodu olan sınıf derlenmez.

C private alana sınıfın kendi metodu da yazamaz.

D setBakiye yalnız static olabilir.

4 **Aşağıdaki satırlar ekrana ne yazar?**

```
String sehir = "bursa";
sehir.toUpperCase();
System.out.println(sehir);
```

Cevap: _____

5 **private final int no; alanı için hangisi doğrudur?**

A Değeri istenen her metotta değiştirilebilir.

B Değerini nesne oluşturulurken bir kez alır; sonra ona değer atayan satır derlenmez.

C Bütün nesneler aynı no'yu paylaşır.

D Değeri program çalışırken her seferinde sıfırlanır.

6 **banka paketindeki Hesap'ta public String sahip, protected int limit, int subeKodu ve private int bakiye alanları var. rapor paketindeki, Hesap'tan türemeyen Rapor sınıfında hangi satır derlenir?**

A System.out.println(h.limit);

B System.out.println(h.subeKodu);

C System.out.println(h.bakiye);

D System.out.println(h.sahip);

7 **Hesap sınıfında private static int sayac = 0; var ve yapıcı metot sayac++; no = sayac; yapıyor. Art arda 5 hesap oluşturulursa beşinci hesabın no'su kaç olur?**

Cevap: _____

8 **Hesap sınıfının static main metodunda System.out.println(bakiye); yazınca derleyici durur. Sebebi nedir?**

A bakiye private olduğu için main ona ulaşamaz.

B static metot bir nesne için değil sınıf için çağrılır; hangi hesabın bakiyesinin istendiğini bilemez.

C println tam sayı yazdıramaz.

D main'in içinde değişken kullanılamaz.

NESNE YÖNELİMLİ PROGRAMLAMA

4 UML sınıf diyagramları

Bu bölümde yazılımcıların koda başlamadan önce çizdiği sınıf diyagramını bir kütüphane örneğiyle öğreniyoruz: üç bölmeli sınıf kutusu ve görünürlük işaretleri, sınıfları birbirine bağlayan altı çizgi ve çizgi uçlarındaki çokluk; her birinin kodda neye karşılık geldiği.

Önce kutu, sonra kod

videoda 21:51

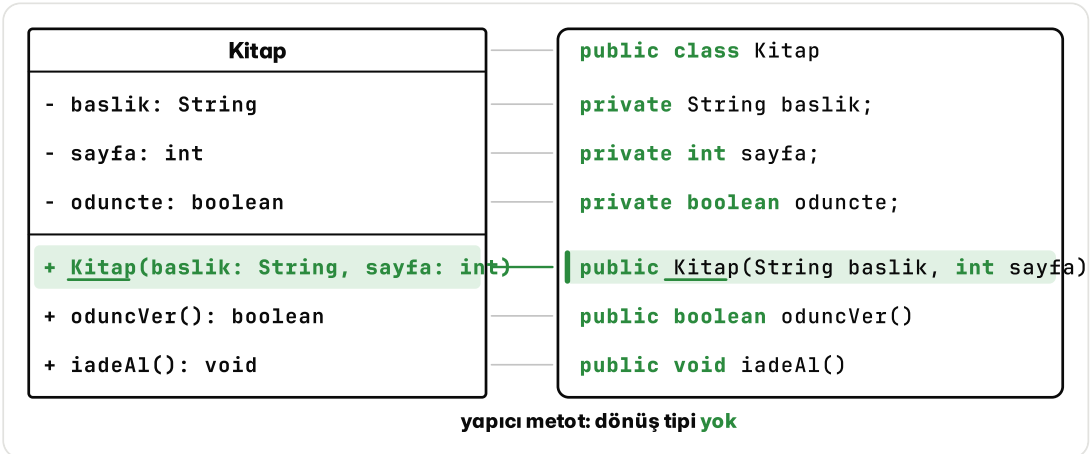
Bir kütüphane programı yazacağız: kitaplar, üyeler ve ödünç verme. Ekip daha tek satır kod yazmadan tahtaya kutular ve oklar çizer: Kitap, Uye, Odunc. Yazılımcılar neden doğrudan koda başlamaz da önce kutu ve ok çizer?

Çünkü tasarımı değiştirmenin en ucuz yeri kâğıttır. Kitabı kimin ödünç aldığı kitabın içinde mi tutulacak, yoksa ayrı bir ödünç kaydında mı? Kâğıtta bu kararı değiştirmek bir satırı silip başka kutuya yazmaktır; kod yazıldıktan sonra ise o bilgiyi kullanan her satırı bulup değiştirmek gerekir. Diyagram bir de büyük sistemi tek sayfada gösterir; ekip aynı sayfaya bakarak konuşur.

Bu çizimlerin ortak bir dili var: **UML** (Unified Modeling Language), yani Birleşik Modelleme Dili. Doksanlı yıllarda üç yazılımcı (Booch, Rumbaugh ve Jacobson) kendi çizim yöntemlerini tek dilde birleştirdi; dil 1997'de standart oldu. 14 tür diyagramı var: kullanım senaryosu diyagramı sistemi kimin ne için kullandığını, sıralama diyagramı nesnelerin hangi sırayla mesajlaştığını gösterir. Koda en yakın olanı **sınıf diyagramıdır**: her kutu bir sınıftır.

Sınıf kutusu üç bölmedir: en üstte ad, ortada alanlar, en altta metotlar. Kitap'ın alanları nesnenin tuttuğu bilgilerdir: başlık, sayfa sayısı ve kitabın ödünçte olup olmadığı. Metotları ise yapabildiği işlerdir: yapıcı metot, ödünç ver ve iade al.

Her satırda tip de yazılır, ama Java'nın tersine: önce ad, sonra iki nokta, sonra tip. Java'daki private String baslik; kutuda - baslik: String olur. Metotta parametreler parantez içinde yine adıyla ve tipiyle yazılır, metodun döndürdüğü tip en sona gelir: + oduncVer(): boolean. oduncVer doğru ya da yanlış döndürür: kitap raftaysa verir ve true der, zaten ödünçteyse false. Hiçbir şey döndürmeyen iadeAl'ın dönüş tipi void; yapıcı metodun dönüş tipi ise hiç yazılmaz: + Kitap(baslik: String, sayfa: int).

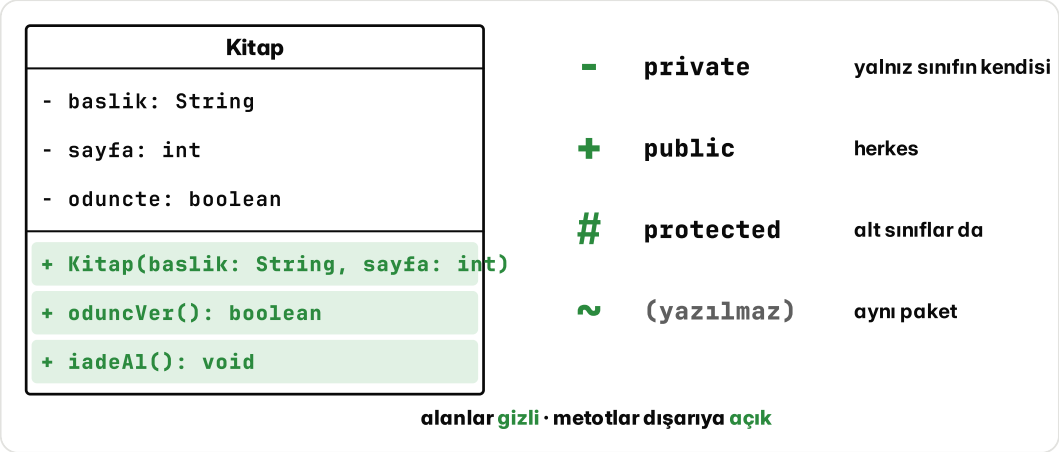


Şekil 4.1 Kitap kutusunun her satırı karşısındaki Java satırıyla eşleşir: UML'de önce ad sonra tip, dönüş tipi en sonda, yapıcıda dönüş tipi yok.

Satırın başındaki işaret, o üyeye kimin erişebileceğini söyler:

- Eksi (-) private demektir: yalnız sınıfın kendisi görür.
- Artı (+) public demektir: herkes kullanır.
- Diyez (#) protected demektir: alt sınıflar da görür.
- Dalga (~) paket erişimidir: aynı paketteki sınıflar görür; Java'da bunun için sözcük yazılmaz.

Kitap'ta alanlar eksi, metotlar artı: bilgiyi sınıf saklar, dışarıya metotlarını açar.



Şekil 4.2 Kitap kutusu ve görünürlük işaretleri tablosu: - private, + public, # protected, ~ aynı paket; alanlar gizli, metotlar açık.

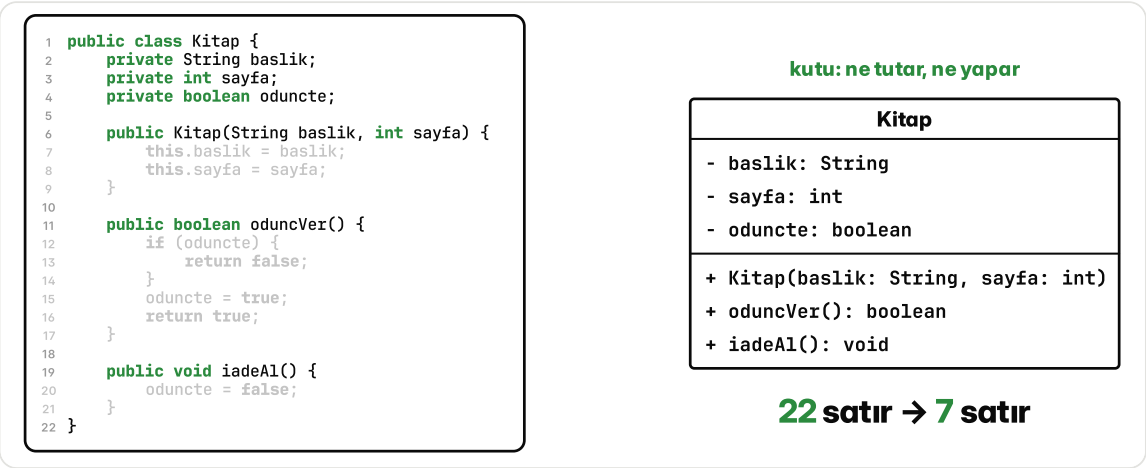
Kodla kutu arasında satır satır bir karşılık var, ama metotların gövdesi diyagrama girmez. Kutu sınıfın ne tuttuğunu ve ne yapabildiğini söyler; nasıl yaptığını kod söyler. Kitap sınıfının ilk kısmı alanlar ve yapıcı metottur:

```
public class Kitap {
    private String baslik;
    private int sayfa;
    private boolean oduncte;

    public Kitap(String baslik, int sayfa) {
        this.baslik = baslik;
        this.sayfa = sayfa;
    }
```

Ardından iki metot ve sınıfın kapanışı gelir:

```
    public boolean oduncVer() {
        if (oduncte) {
            return false;
        }
        oduncte = true;
        return true;
    }

    public void iadeAl() {
        oduncte = false;
    }
}
```

Bu yüzden 22 satırlık Kitap sınıfı diyagramda 7 satırlık bir kutudur (ad, üç alan, üç metot); ekip tasarımı bu kısa kutular üzerinden konuşur.



Şekil 4.3 22 satırlık Kitap.java'nın yalnız imza satırları 7 satırlık kutuya geçer; metot gövdeleri kodda kalır.

Çizgiler ne söyler?

videoda 24:10

Bir sınıf diyagramında her kutu bir sınıftır; kutuları birleştiren çizgilerin kimi düz, kimi kesik, uçlarında da ok, üçgen ya da elmas var. Kodu hiç görmeden, yalnız bu çizgilere bakarak ne söyleyebiliriz? Çok şey, çünkü her çizgi biçimi kodda belli bir yazıma karşılık gelir.

Kütüphanede bir ödünç kaydı hangi kitabın verildiğini bilir: Odunc sınıfında Kitap tipinde bir alan vardır.

```
public class Odunc {
    private Kitap kitap;
}
```

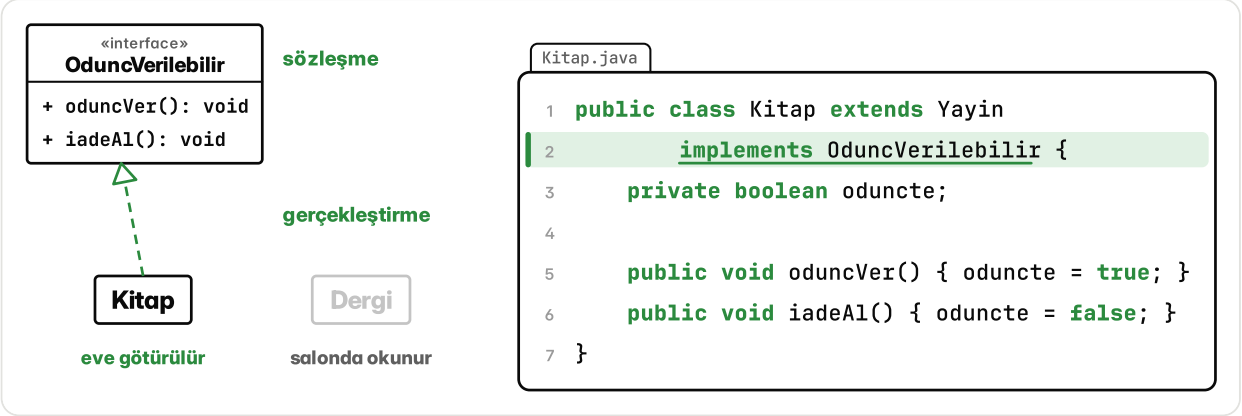
Diyagramda bu, iki kutu arasında düz bir çizgidir; buna **ilişki** (association) denir. Ucundaki açık ok kimin kimi tanıdığını gösterir: bu diyagramda ödünç kaydı kitabı bilir, kitap ödünç kaydını bilmez.

Kitap da dergi de birer yayındır; başlık gibi ortak alanları Yayin sınıfında durur. İkisi de Yayin'dan türer, yani alanlarını ve metotlarını devralır; buna **kalıtım** (inheritance) denir. Kodda extends yazılır; diyagramda ucu içi boş üçgen olan düz bir çizgi çizilir ve üçgen üst sınıfı gösterir.

```
public class Dergi extends Yayin {
    private int sayi;
}
```

Bu kütüphanede kitaplar eve götürülebilir, dergiler yalnız salonda okunur. Ödünç verilebilen her şeyin oduncVer ve iadeAl metotları olmalı; bu söze **arayüz** (interface) denir: sınıfın hangi metotları yazmak zorunda olduğunu söyleyen bir sözleşme. Kitap bu sözleşmeyi gerçekleştirir: kodda implements, diyagramda yine içi boş üçgen, ama bu kez çizgi kesik.

```
public class Kitap extends Yayin
        implements OduncVerilebilir {
    private boolean oduncte;

    public void oduncVer() { oduncte = true; }
    public void iadeAl() { oduncte = false; }
}
```



Şekil 4.4 Kitap'tan «interface» OduncVerilebilir'e kesik çizgi ve içi boş üçgen: gerçekleştirme; Dergi'den çizgi yok.

Görevlinin iadeAl metodu kitabı parametre olarak alır, iade edildi diye işaretler ve bırakır:

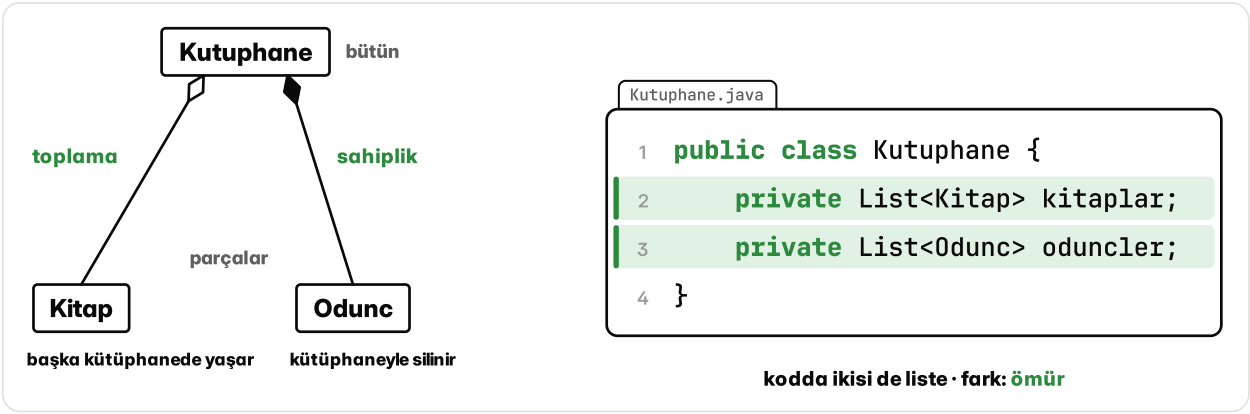
```
public class Gorevli {
    public void iadeAl(Kitap kitap) {
        kitap.iadeAl();
    }
}
```

Gorevli'nin kitap diye bir alanı yoktur; kitabı yalnız o metodun içinde kullanır. Bu kısa kullanıma **bağımlılık** (dependency) denir ve kesik çizgiyle, ucunda açık bir okla çizilir: Kitap değişirse Gorevli de etkilenebilir.

Son iki çizgide elmas var; ikisi de bir bütünü ve parçalarını gösterir, elmas bütünün yanında durur. Kutuphane ile kitapları arasındaki elmasın içi boştur, buna **toplama** (aggregation) denir: kütüphane kapansa da kitaplar başka bir kütüphanede yaşar. Ödünç kayıtlarıyla arasındaki elmasın içi doludur, buna **sahiplik** (composition) denir: kayıtlar kütüphaneyle doğar, onunla birlikte silinir. Kodda ikisi de birer liste alanıdır:

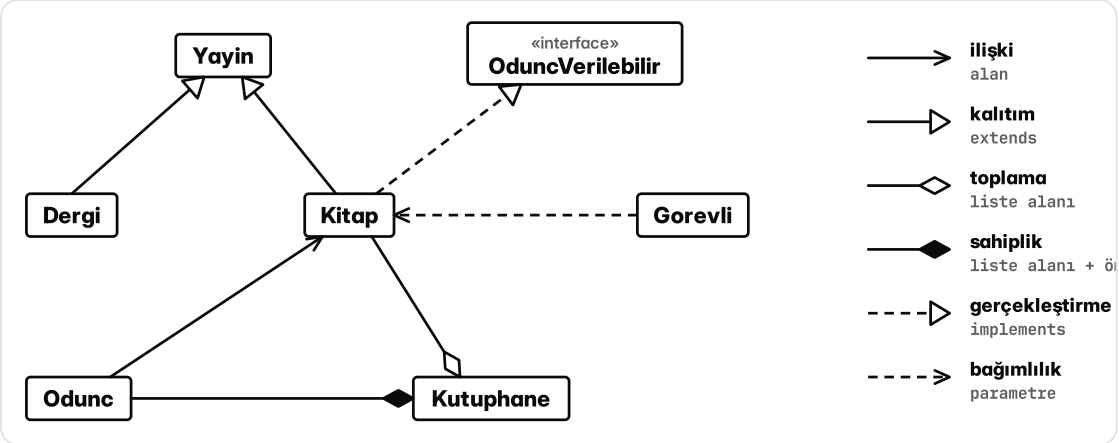
```
public class Kutuphane {
    private List<Kitap> kitaplar;
    private List<Odunc> oduncler;
}
```

Elmas, kodun söylemediği ömür bilgisini ekler.



Şekil 4.5 Kutuphane'den Kitap'a içi boş elmas (toplama), Odunc'a içi dolu elmas (sahiplik); kodda ikisi de liste, fark ömürde.

Bu çizgileri tek kuralla toplayabiliriz. Düz çizgide bağ kodda kalıcıdır: ya bir alan ya da extends. Kesik çizgide alan yoktur: ya yerine getirilecek bir söz (implements) ya da bir metodun içinde kısa bir kullanım. Bu yüzden diyagrama bakan bir yazılımcı kodu açmadan iki sınıfın nasıl bağlandığını bilir.



Şekil 4.6 Tam diyagram ve altı çizginin kodda karşılığı: ilişki alan, kalıtım extends, gerçekleştirme implements, bağımlılık parametre, toplama ve sahiplik liste alanı.

Kaç tane?

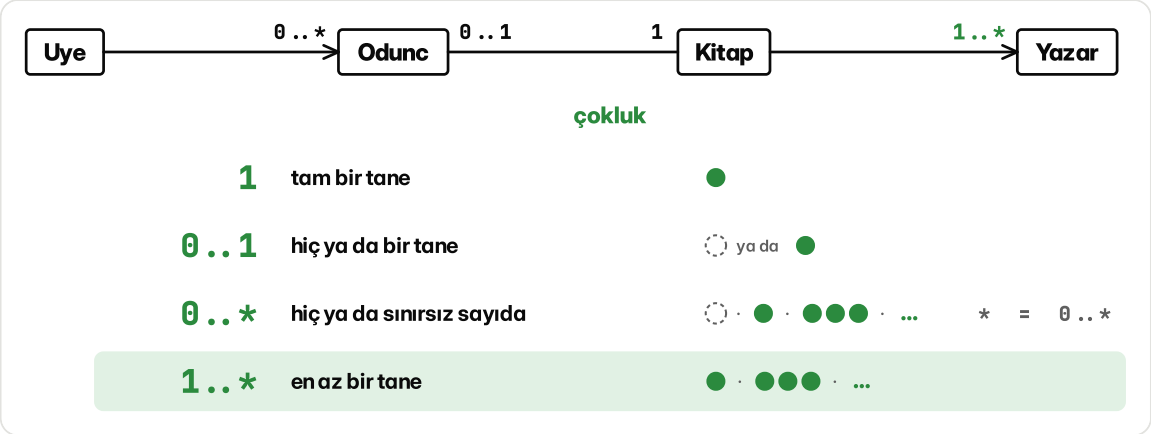
videoda 26:19

Bir kütüphane üyesinin elinde hiç ödünç kitap olmayabilir; ama yazarı olmayan bir kitap olmaz. Sınıf diyagramı bu farkı nasıl gösterir, kod da onu nasıl korur?

Sınıf diyagramında çizgilerin uçlarına küçük sayılar konur; bunlara **çokluk** (multiplicity) denir. Dört kalıp var:

- 1: tam bir tane.
- 0..1: hiç ya da bir tane.
- 0..*: hiç ya da sınırsız sayıda; tek başına * da aynı anlamdadır.

- 1..*: en az bir tane.



Şekil 4.7 Uye, Odunc, Kitap, Yazar zincirinin uçlarında 0..*, 0..1, 1, 1..* ve dört kalıbın anlamları.

Sayı, yanında durduğu kutudan kaç tane olabileceğini söyler. Uye'den Odunc'a giden çizginin ucunda 0..* durur: bir üyenin hiç ödünç kaydı olmayabilir ya da birçok olabilir. Tek alan buna yetmez; kodda bir liste tutulur ve liste boş başlar:

```
public class Uye {
    private List<Odunc> oduncler = new ArrayList<>();
}
```

Hiç kitap almamış üyenin listesi boştur.

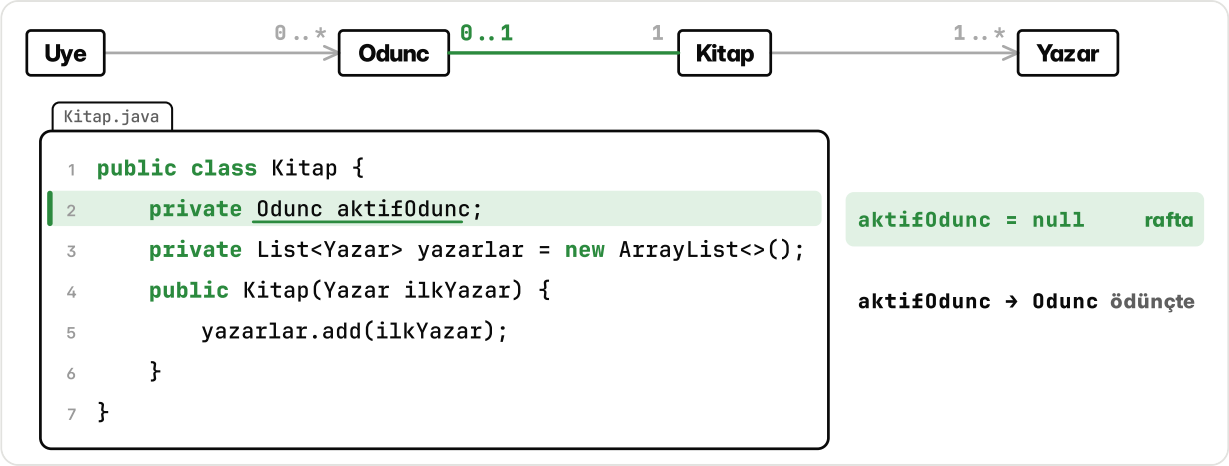
Odunc ile Kitap arasındaki çizginin iki ucunda da sayı var. Kitap tarafında 1 durur: her ödünç kaydı tam bir kitaba aittir, bu yüzden Odunc sınıfı kitabı yapıcı metotta ister:

```
public class Odunc {
    private Kitap kitap;
    public Odunc(Kitap kitap) {
        this.kitap = kitap;
    }
}
```

Odunc tarafında 0..1 durur: kitap ya raftadır ya da tek bir ödünç kaydındadır. Kodda bu, boş kalabilen tek bir alandır, aşağıdaki aktifOdunc; değeri null ise kitap raftadır.

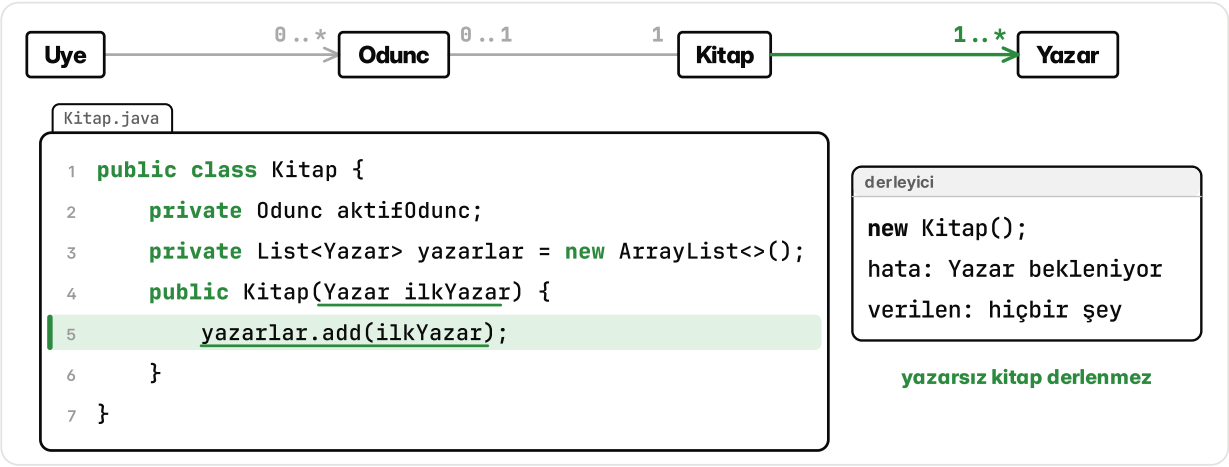
Kitap'tan Yazar'a giden çizginin ucunda 1..* durur: birden çok yazar olabilir, ama en az biri şarttır. Kodda yine bir liste var, ama boş başlayamaz: yapıcı metot ilk yazarı ister ve hemen listeye ekler.

```
public class Kitap {
    private Odunc aktifOdunc;
    private List<Yazar> yazarlar = new ArrayList<>();
    public Kitap(Yazar ilkYazar) {
        yazarlar.add(ilkYazar);
    }
}
```



Şekil 4.8 Kitap.java'da aktifOdunc tek alan: null ise kitap rafta, bir Odunc'u gösteriyorsa ödünçte.

Yazarsız kitap oluşturmaya çalışan new Kitap(); satırı derlenmez bile; derleyici bir Yazar beklediğini, hiçbir şey verilmediğini söyler ("required: Yazar", "found: no arguments").



Şekil 4.9 new Kitap(); için derleyici: Yazar bekleniyor, verilen hiçbir şey; yazarsız kitap derlenmez.

Bu çeviri iki yöne de işler: yazılmış koddan da diyagram çıkarılır. Bir alan başka bir sınıfı gösteriyorsa iki kutu arasına çizgi çekilir; alan bir listeyse ucuna yıldız konur. Birçok geliştirme aracı bu diyagramı koddan kendisi çizebilir; yeni bir projeye katılan biri önce bu tek sayfaya bakar. Baştaki fark da diyagramda iki küçük sayı (0..* ve 1..*), kodda ise boş başlayan bir liste ile ilk yazarı isteyen bir yapıcı metottur.

Özet

- UML'in sınıf diyagramında her kutu bir sınıftır; kâğıtta tasarımı değiştirmek koddan çok daha ucuzdur.
- Sınıf kutusu üç bölmedir: ad, alanlar, metotlar; satırlar "ad: Tip" biçimindedir, dönüş tipi en sona gelir, gövdeler kodda kalır.
- Görünürlük işaretleri: - private, + public, # protected, ~ paket erişimi.
- Düz çizgi kalıcı bir bağdır (ilişkide alan, kalıtımda içi boş üçgenle extends); kesik çizgide alan yoktur (gerçekleştirmede implements, bağımlılıkta açık okla parametre).
- İçi boş elmas toplamadır, içi dolu elmas sahiplik; ikisi de kodda liste alanıdır, elmas ömrü söyler.
- Çokluk 1, 0..1, 0..* ve 1..* ile gösterilir; kodda 0..* boş başlayan bir listeye, 1..* ilk elemanı yapıcıda isteyen bir listeye karşılık gelir.

Sorular

1 **UML sınıf kutusunun üç bölmesi yukarıdan aşağıya hangi sırayla dizilir?**

A Alanlar, ad, metotlar
B Ad, metotlar, alanlar
C Ad, alanlar, metotlar
D Metotlar, alanlar, ad

2 **Java'daki private int sayfa; alanı UML sınıf kutusunda nasıl yazılır?**

Cevap: _____

3 **Java'daki public boolean oduncVer() metodu UML kutusunda nasıl görünür?**

A + oduncVer(): boolean
B + boolean oduncVer()
C - oduncVer(): boolean
D # oduncVer: boolean()

4 **UML'de bir üyenin başındaki # işareti ne demektir?**

A private: yalnız sınıfın kendisi görür
B Paket erişimi: aynı paketteki sınıflar görür
C public: herkes kullanır
D protected: alt sınıflar da görür

5 **İki sınıf arasında kesik çizgi var ve ucunda içi boş üçgen duruyor. Kodda buna ne karşılık gelir?**

A extends: sınıf üst sınıftan türer
B implements: sınıf bir arayüzü gerçekleştirir
C Bir alan: sınıf ötekini tanır
D Bir metot parametresi: kısa bir kullanım

6 **Gorevli sınıfının Kitap tipinde alanı yok; yalnız iadeAl(Kitap kitap) metodu kitabı parametre olarak alıp kullanıyor. Diyagramda bu nasıl çizilir?**

A Kesik çizgi, Kitap ucunda açık ok: bağımlılık

B Düz çizgi, Kitap ucunda açık ok: ilişki

C Düz çizgi, Gorevli ucunda içi boş elmas: toplama

D Düz çizgi, Kitap ucunda içi boş üçgen: kalıtım

7 **Kutuphane ile Odunc kayıtları arasındaki içi dolu elmas ne söyler?**

A Odunc kayıtları Kutuphane'den türer.

B Kutuphane kapansa da kayıtlar başka bir kütüphanede yaşar.

C Kayıtlar kütüphaneyle doğar ve onunla birlikte silinir.

D Kutuphane'de Odunc tipinde alan yoktur.

8 **Bir kitabın birden çok yazarı olabilir, ama en az bir yazarı olmak zorundadır. Kitap'tan Yazar'a giden çizginin Yazar ucuna hangi çokluk yazılır?**

Cevap: _____

NESNE YÖNELİMLİ PROGRAMLAMA

5 Sahiplik ve toplama

Bir nesnenin alanı başka bir nesne olabilir: takımın oyuncuları, siparişin satırları gibi. Bu bölümde bu bağın iki türünü, toplamayı ve sahipliği göreceğiz: aralarındaki farkı, sınıf diyagramındaki elmasları ve bu farkın koddaki karşılığını.

Silinince ne kalır?

videoda 28:01

Bir alışveriş uygulamasında sipariş silinince içindeki satırlar da silinir: 3 paket makarna, 1 şişe zeytinyağı. Bir spor uygulamasında takım silinince ise oyuncular yerinde kalır. İkisinde de bir bütün ve onun parçaları var; biri neden parçalarıyla birlikte gidiyor da öteki gitmiyor?

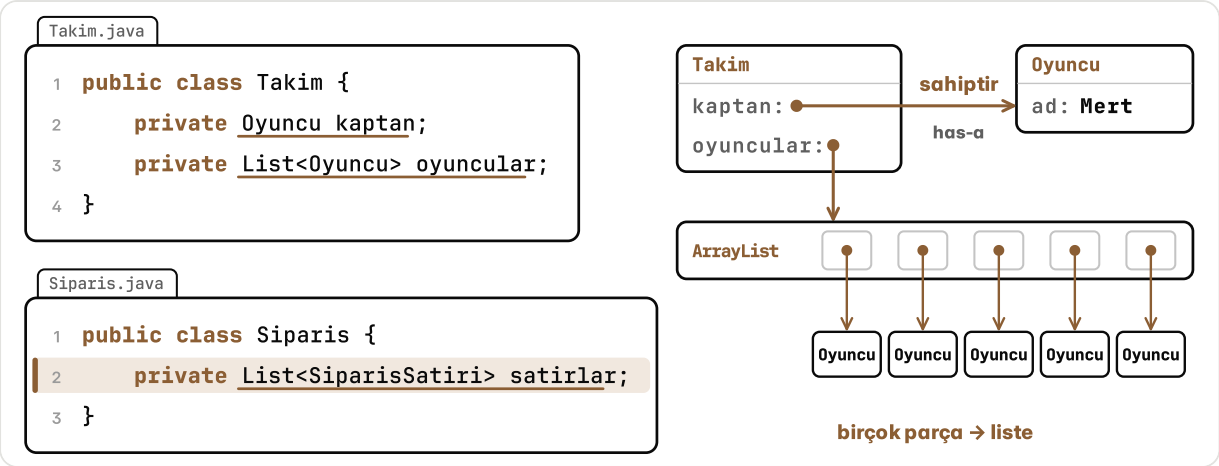
Önce ikisindeki ortak bağa bakalım. Takımın bir kaptanı vardır; kodda bu, Takim sınıfında Oyuncu tipinde bir alandır:

```
public class Takim {
    private Oyuncu kaptan;
    private List<Oyuncu> oyuncular;
}
```

Bir nesnenin alanı başka bir nesne olunca, ikisi arasındaki bağa **sahiptir ilişkisi** (has-a) denir: takım bir kaptana sahiptir. kaptan alanı oyuncunun kendisini değil, bellekteki yerini tutar; yani bir **referanstır**. Takımın içine oyuncunun bir kopyası konmaz; takım yalnız oyuncunun nerede durduğunu bilir.

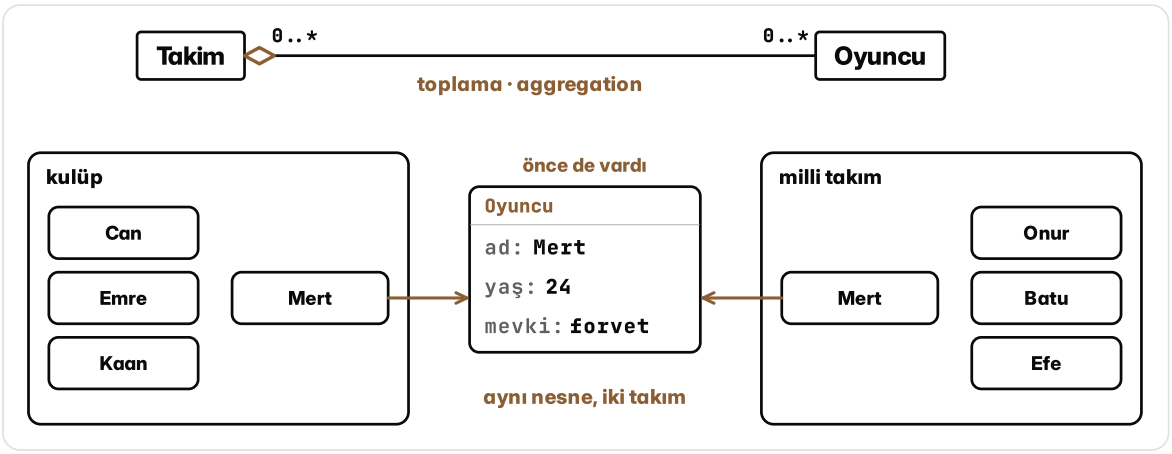
Takımın oyuncuları da böyle bir alanda durur; ama oyuncu tek değil birçok olduğu için bu alan bir listedir. List<Oyuncu>, Oyuncu nesnelerinin referanslarını sırayla tutan bir listedir. Siparişin satırları da aynı biçimde saklanır:

```
public class Siparis {
    private List<SiparisSatiri> satirlar;
}
```



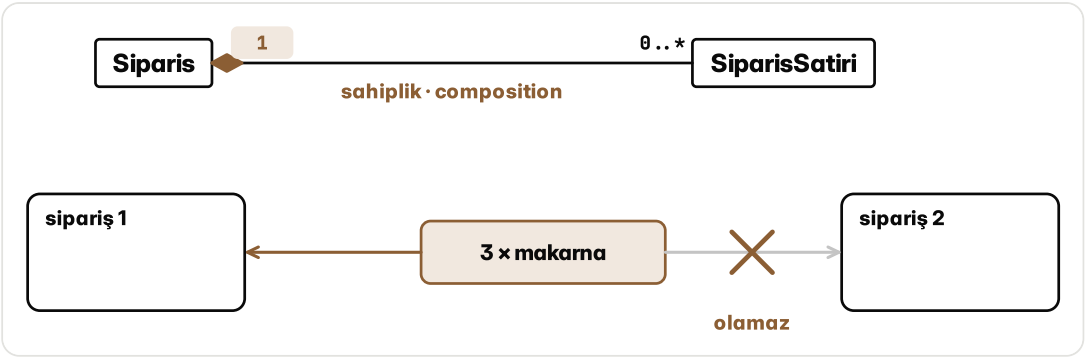
Şekil 5.1 Takim'in kaptan alanı bir Oyuncu nesnesini gösterir (sahiptir, has-a); oyuncular alanı beş oyuncuyu tutan bir listeye gider.

Ama bu bağların hepsi aynı değildir. Oyuncu takımdan önce de vardı; adı, yaşı, oynadığı mevki kendisine aittir. Takım dağılınca başka bir takıma geçer; aynı oyuncu aynı anda hem kulübünde hem milli takımda oynayabilir. Bellekte bu, aynı Oyuncu nesnesine iki ayrı takımın listesinden ulaşılması demektir. Parçanın bütünden bağımsız yaşadığı bu ilişkiye **toplama** (aggregation) denir. Sınıf diyagramında bütünün, yani Takim'in ucuna içi boş bir elmas çizilir. Takim ucundaki 0..* bir oyuncunun birden çok takımda olabileceğini söyler.



Şekil 5.2 Takim ile Oyuncu arasında içi boş elmas ve iki uçta 0..*; aynı Mert nesnesi hem kulübün hem milli takımın kadrosunda.

Sipariş satırı ise tek başına bir şey anlatmaz: 3 paket makarna, ama kimin siparişi, hangi gün? Satır siparişin içinde doğar ve yalnız o siparişe aittir; sipariş silinince satırı tutmanın anlamı kalmaz, o da silinir. Buna **sahiplik** (composition) denir; diyagramda Siparis'in ucuna içi dolu bir elmas çizilir. Bir satır aynı anda 2 siparişte olamaz; bu yüzden diyagramda Siparis'in ucunda 1 durur. SiparisSatiri'nin ucundaki 0..* ise bir siparişte birçok satır olabileceğini söyler.



Şekil 5.3 Siparis ile SiparisSatiri arasında içi dolu elmas; Siparis ucunda 1, satır ucunda 0..*; aynı satır iki siparişe bağlanamaz.

İkisini ayırmak için tek bir soru yeter: bütün silinince, parça başka bir yerde işe yarar mı? Oyuncu yarar, öyleyse toplama; satır yaramaz, öyleyse sahiplik. Aynı soru her yerde işler: çalma listesi silinince şarkılar yerinde kalır, bu toplamadır; bina yıkılınca odaları da onunla gider, bu sahipliktir. Diyagramdaki elmasın içinin boş ya da dolu olması, bu sorunun cevabıdır.

Elmas kodun neresinde?

videoda 29:45

Takım ile oyuncu arasındaki elmasın içi boş, çünkü oyuncu takımdan bağımsız yaşar; sipariş ile satır arasındakinin içi dolu, çünkü satır siparişle birlikte silinir. Ama iki sınıfın kodu neredeyse aynıdır; ikisinde de bir liste alanı var:

```
public class Takim {
    private List<Oyuncu> oyuncular = new ArrayList<>();
}

public class Siparis {
    private List<SiparisSatiri> satirlar = new ArrayList<>();
}
```

Bu fark kodun neresinde? Önce listeye bakalım. Bir siparişte birçok satır olabilir; bu yüzden alan tek bir satır değil, satırları tutan bir listedir. Bir nesneye birçok nesnenin bağlandığı bu ilişkiye **bire çok ilişki** denir. new ArrayList<>() boş bir liste oluşturur; liste boş başlar ve her yeni satır add ile sonuna eklenir. Siparişin toplamını bulan metotta bir for döngüsü listedeki satırları sırayla alır ve her birinin tutarını t'ye ekler:

```
public int toplam() {
    int t = 0;
    for (SiparisSatiri s : satirlar) {
        t += s.tutar();
    }
    return t;
}
```

SiparisSatiri'nin tutar metodu adedi fiyatla çarpar. Siparişe iki satır ekleyip toplamı yazdıralım:

```
Siparis s = new Siparis();
s.satirEkle("makarna", 3, 25);
s.satirEkle("zeytinyağı", 1, 280);
System.out.println(s.toplam());
```

Çıktı:

```
355
```

3 paket makarna 3 × 25 = 75 lira, 1 şişe zeytinyağı 1 × 280 = 280 lira; toplam 75 + 280 = 355 lira.

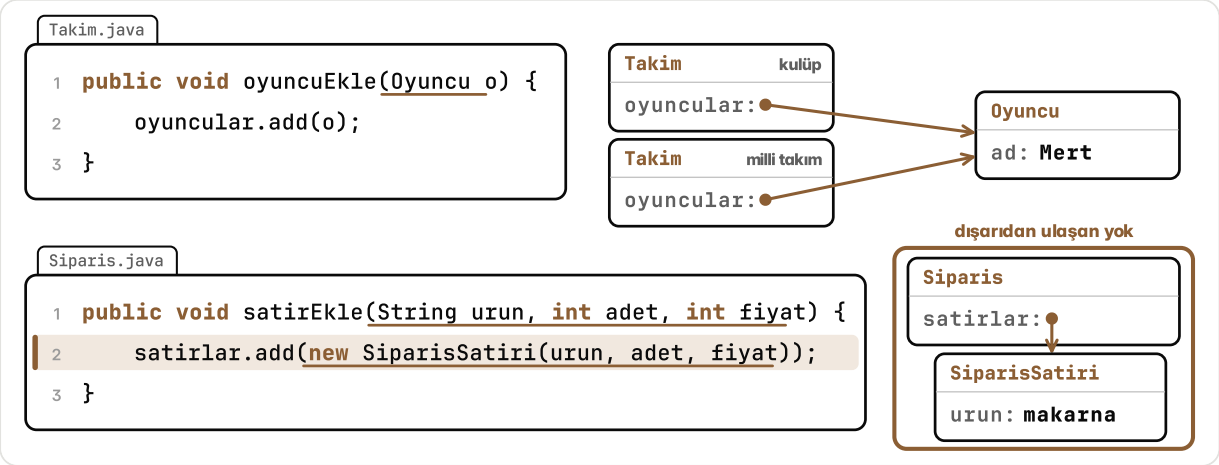
Fark, parçayı kimin oluşturduğundadır. Takımın oyuncu ekleme metodu, dışarıda oluşturulmuş bir oyuncuyu alır ve listesine ekler:

```
public void oyuncuEkle(Oyuncu o) {
    oyuncular.add(o);
}
```

O oyuncuya başka referanslar da ulaşır: aynı oyuncu milli takımın listesinde de durabilir. Siparişin satır ekleme metodu ise ürünü, adedi ve fiyatı alır; satırı kendisi new ile oluşturur:

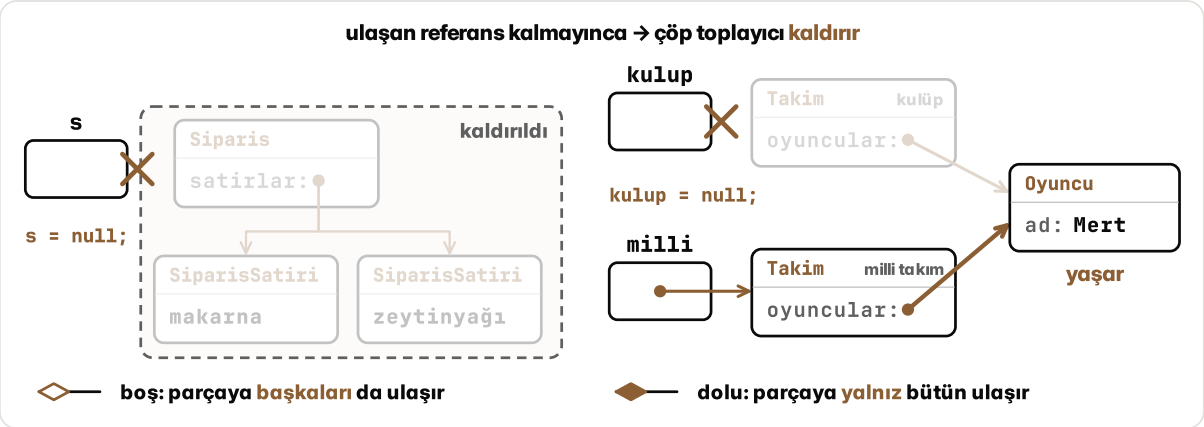
```
public void satirEkle(String urun, int adet, int fiyat) {
    satirlar.add(new SiparisSatiri(urun, adet, fiyat));
}
```

Satırın referansı dışarıya hiç verilmez; satırlara yalnız sipariş ulaşır.



Şekil 5.4 Mert'e hem kulübün hem milli takımın listesi ulaşır; siparişin satırına ise dışarıdan ulaşan yok.

Peki silinmek ne demek? Java'da nesneyi biz silmeyiz; ona ulaşan hiçbir referans kalmayınca **çöp toplayıcı** (garbage collector) onu bellekten kaldırır. Çöp toplayıcının ne zaman çalışacağı önceden bilinmez; bilinen, ulaşılamayan nesnenin kaldırılabilir hâle geldiğidir. Siparişe ulaşan son referans gidince, örneğin s = null; satırıyla, satırlara ulaşan da kalmaz; sipariş, satırlarıyla birlikte kaldırılır. Kulüp takımı silinince ise oyuncuya milli takımın listesi hâlâ ulaşır; oyuncu yaşamaya devam eder. Kodda elmasın anlamı budur: boş elmasta parçaya başkaları da ulaşır, dolu elmasta yalnız bütün ulaşır.



Şekil 5.5 s = null; sonrasında sipariş satırlarıyla birlikte kaldırılır; kulup = null; sonrasında Mert'e milli takım ulaştığı için yaşar.

Sahiplikte parça bazen dışarıdan gelir. Sipariş verilirken teslimat adresi, müşterinin kayıtlı adresinden alınır. Siparişin yapıcı metodu bir Adres alıp teslimat alanına koysun:

```
public class Siparis {
    private Adres teslimat;
    public Siparis(Adres adres) {
        teslimat = adres;
    }
}
```

Müşterinin adresi Ankara iken sipariş verilsin, ardından müşteri adresini değiştirsin. teslimatSehri metodu siparişin teslimat adresindeki şehri döndürür:

```
Adres ev = new Adres("Ankara");
Siparis s = new Siparis(ev);
ev.setSehir("İzmir");
System.out.println(s.teslimatSehri());
```

Çıktı:

```
İzmir
```

Sipariş adres nesnesinin referansını olduğu gibi sakladığı için aynı nesne hem müşterinin hem siparişin olur. Müşteri ertesi ay İzmir'e taşınıp adresini değiştirince, Ankara'ya teslim edilmiş eski siparişin adresi de İzmir olur.

Çözüm, gelen nesnenin kopyasını almaktır. Yapıcı metottaki tek satır değişir:

```
public class Siparis {
    private Adres teslimat;
    public Siparis(Adres adres) {
        teslimat = new Adres(adres);
    }
}
```

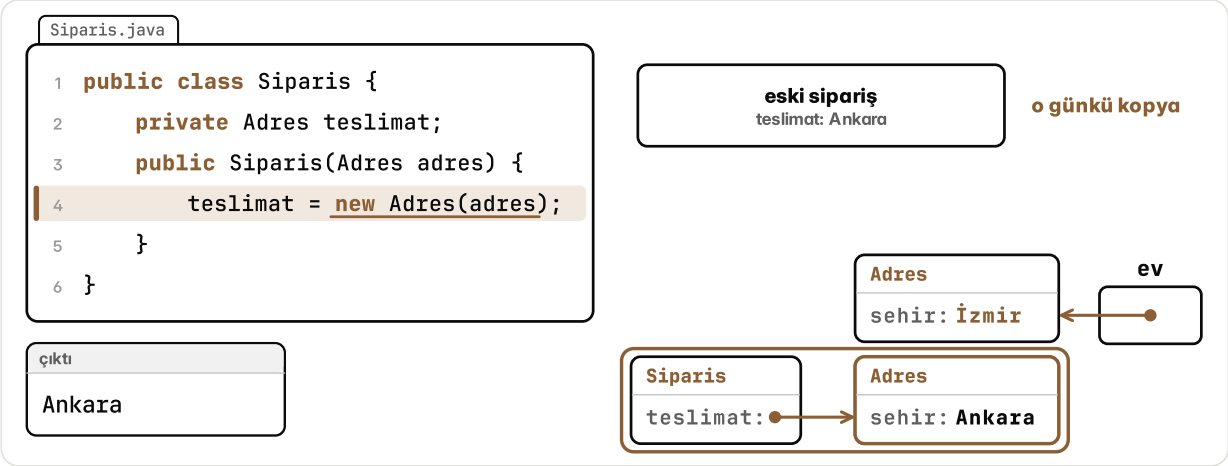
Sipariş, yeni bir Adres nesnesini new ile oluşturur ve şehri ona aktarır. Bu işi Adres sınıfına eklenen, adresi kopyalayan ikinci bir yapıcı metot yapar; sehir alanı private olsa da aynı sınıfın içinden diger.sehir diye okunabilir:

```
public Adres(Adres diger) {
    this.sehir = diger.sehir;
}
```

Aynı deneme programı artık şunu yazar:

```
Ankara
```

Müşteri kendi adresini değiştirse de siparişin adresi Ankara'da kalır; parçanın tek sahibi siparistir. Alışveriş sitelerinin çoğu da siparişe adresin o günkü kopyasını yazar; eski siparişte hep teslim edilen adres görünür.



Şekil 5.6 Kopyalayan yapıcı metotla iki ayrı Adres nesnesi oluşur: ev İzmir olur, siparişinki Ankara kalır.

Özet

- Bir nesnenin alanı başka bir nesne olunca aralarında sahiptir ilişkisi (has-a) vardır; alan nesnenin kendisini değil, referansını tutar.
- Parça bütünden bağımsız yaşıyorsa ilişki toplamadır; diyagramda bütünün ucuna içi boş elmas çizilir.
- Parça bütünle doğuyor ve bütünle siliniyorsa ilişki sahipliktir; bütünün ucunda içi dolu elmas ve 1 durur.
- Ayırt eden soru: bütün silinince parça başka bir yerde işe yarar mı? Evetse toplama, hayırsa sahiplik.
- Birçok parça bir listede tutulur (bire çok ilişki); toplamada parça dışarıdan gelir, sahiplikte bütün onu new ile kendisi oluşturur ve referansını dışarı vermez.
- Java'da ulaşan referansı kalmayan nesneyi çöp toplayıcı kaldırır; dışarıdan gelen parçanın tek sahibi olmak için kopyası alınır.

Sorular

1 **Takim sınıfında private Oyuncu kaptan; diye bir alan var. kaptan alanı neyi tutar?**

- **A** Oyuncu nesnesinin bir kopyasını
- **B** Oyuncunun yalnız adını
- **C** Oyuncu nesnesinin bellekteki yerini, yani referansını
- **D** Oyuncu sınıfının bütün nesnelerini

2 **Aşağıdaki çiftlerden hangisi sahiplik (composition) örneğidir?**

- **A** Takım ve oyuncu
- **B** Bina ve oda
- **C** Çalma listesi ve şarkı
- **D** Milli takım ve oyuncu

3 **Siparis ◆— SiparisSatiri diyagramında Siparis'in ucunda neden 1 yazar?**

- **A** Bir satır aynı anda yalnız bir siparişe ait olabilir.
- **B** Bir siparişte yalnız bir satır olabilir.
- **C** Siparis sınıfından yalnız bir nesne oluşturulabilir.
- **D** Her siparişin bir müşterisi vardır.

4 **Sınıf diyagramında Takim ile Oyuncu arasındaki elmasın içi neden boştur?**

- **A** Takım sınıfının alanı yoktur.
- **B** Oyuncu, takımın içinde new ile oluşturulur.
- **C** Bir takımda yalnız bir oyuncu olabilir.
- **D** Oyuncu takımdan bağımsız yaşar; takım dağılsa da başka bir takımda oynayabilir.

5 **Bu bölümdeki Siparis ve SiparisSatiri sınıflarıyla aşağıdaki satırlar ekrana ne yazar?**

```
Siparis s = new Siparis();
s.satirEkle("çay", 2, 40);
s.satirEkle("şeker", 3, 30);
System.out.println(s.toplam());
```

Cevap: _____

6 **Takim ve Siparis sınıflarının ikisinde de bir liste alanı var. Siparişin satırlarına yalnız siparişin ulaşmasını sağlayan fark hangisidir?**

- **A** Siparis'teki alanın bir liste olması
- **B** toplam metodunun bir for döngüsü kullanması
- **C** satirEkle'nin satırı new ile kendisi oluşturması ve referansını dışarı vermemesi
- **D** SiparisSatiri'nin bir tutar metodu olması

7 **Java'da bir nesne ne zaman bellekten kaldırılabilir hâle gelir?**

A Ona ulaşan hiçbir referans kalmadığında

B Programcı onu delete ile sildiğinde

C Nesneyi oluşturan metot bittiğinde

D Sınıfının bütün nesneleri silindiğinde

8 **Siparis'in yapıcı metodu teslimat = new Adres(adres); satırıyla adresin kopyasını alıyor. Aşağıdaki satırlar ekrana ne yazar?**

```
Adres ev = new Adres("Bursa");
Siparis s = new Siparis(ev);
ev.setSehir("Eskişehir");
System.out.println(s.teslimatSehri());
```

Cevap:

NESNE YÖNELİMLİ PROGRAMLAMA

6 Kalıtım ve override

Birçok sınıf aynı alanları ve aynı metotları taşıyorsa, ortak olanı bir kez yazıp öteki sınıfların ondan türemesini sağlayabiliriz. Bu bölümde kalıtımı (inheritance) göreceğiz: extends ile türemeyi, yapıcıların ve super'in işini, bir metodu alt sınıfta yeniden tanımlamayı, her sınıfın atası Object'i, kalıtımın kapalı kapılarını ve kalıtımın yanlış seçildiği durumları.

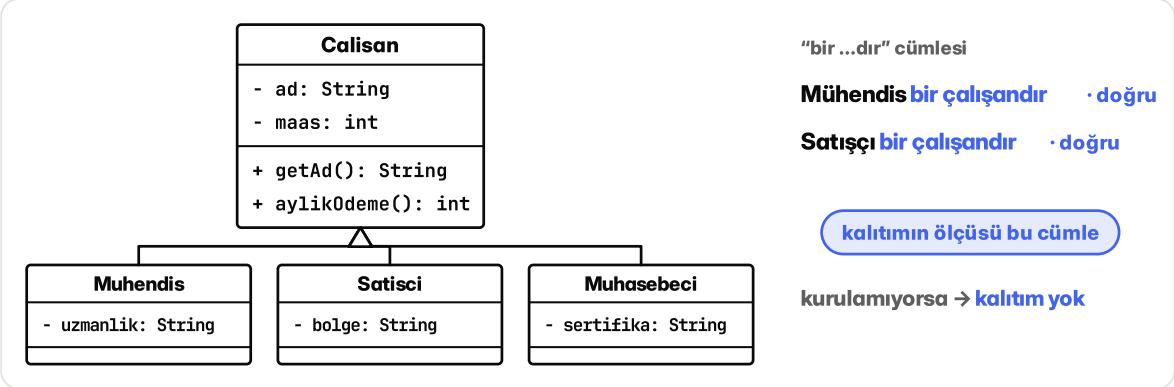
Yirmi tür, bir kod?

videoda 31:57

Bir şirketin yazılımında 20 ayrı çalışan türü var: mühendis, satışçı, muhasebeci ve daha 17'si. Her birinin bir adı ve maaşı var; aylık ödemeleri de aynı kuralla hesaplanıyor. Aynı kodu 20 sınıfa 20 kez yazmadan nasıl çıkarız?

İlk akla gelen, bir sınıfı yazıp 20 kez kopyalamaktır. Ama ödeme hesabında bir hata bulununca, 20 sınıfta ayrı ayrı düzeltmek gerekir; biri atlanırsa, o türdeki çalışanlar yanlış maaş alır.

Doğru yol, ortak olanı tek bir sınıfta toplamaktır: ad, maaş ve aylık ödeme, Calisan sınıfında bir kez yazılır. Mühendis de satışçı da bu sınıftan türer; çünkü ikisi için de aynı cümle doğrudur: mühendis bir çalışandır, satışçı da bir çalışandır. Kalıtımın ölçüsü bu "bir ...dır" cümlesidir; kurulamıyorsa kalıtım da kurulmaz.



Şekil 6.1 Ortak satırlar Calisan sınıfında toplanır; Muhendis, Satisci ve Muhasebeci yalnız kendi alanlarını taşır ve kalıtım okları Calisan'a çıkar.

Kodda bunu **extends** sözcüğü söyler: Muhendis sınıfı, Calisan sınıfını genişletir. Calisan'a **üst sınıf**, Muhendis'e **alt sınıf** denir. Mühendis sınıfına yalnız kendine özgü olanı ekleriz: uzmanlık alanı.

```
public class Muhendis extends Calisan {
    private String uzmanlik;
}
```

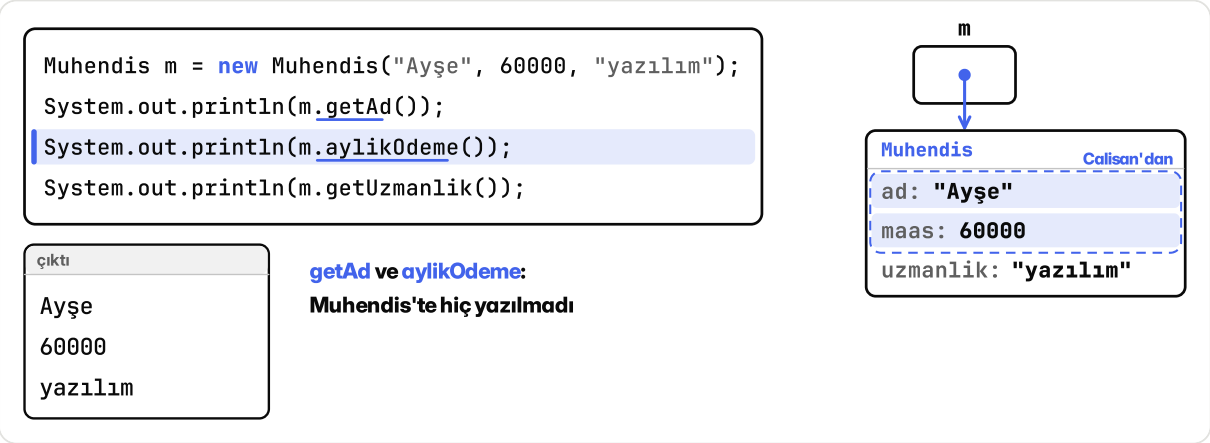
Ayşe'yi, 60.000 lira maaşlı bir yazılım mühendisi olarak oluşturalım:

```
Muhendis m = new Muhendis("Ayşe", 60000, "yazılım");
System.out.println(m.getAd());
System.out.println(m.aylikOdeme());
System.out.println(m.getUzmanlik());
```

Çıktı:

```
Ayşe
60000
yazılım
```

getAd da aylikOdeme de Muhendis sınıfında hiç yazılmadı; yine de çalışırlar, çünkü Calisan'dan devralındılar. Mühendis nesnesinin içinde, Calisan'dan gelen ad ve maas alanları da vardır.

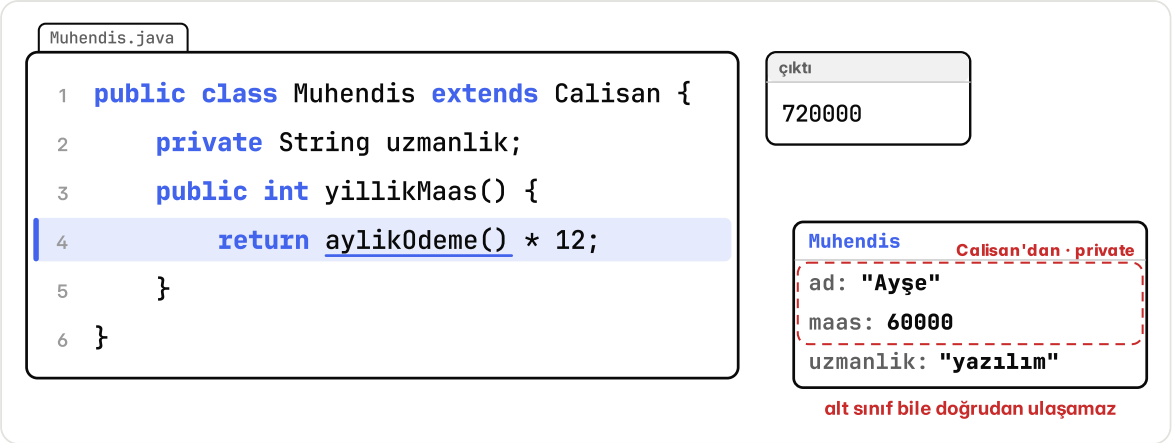


Şekil 6.2 Muhendis nesnesinin üst bölmesinde Calisan'dan gelen ad = "Ayşe" ve maas = 60000, altında kendi alanı uzmanlik = "yazılım".

Şimdi mühendise yıllık maaşı hesaplayan bir metot ekleyelim: maaş × 12.

```
public class Muhendis extends Calisan {
    private String uzmanlik;
    public int yillikMaas() {
        return maas * 12;
    }
}
```

Kod derlenmez; derleyici "maas has private access in Calisan" der, yani maas Calisan sınıfında private'tır. Alan mühendis nesnesinde vardır; ama private olduğu için alt sınıf bile ona doğrudan ulaşamaz. Çözümlerden biri, Calisan'ın açtığı metodu kullanmaktır: satır return aylikOdeme() * 12; olunca kod derlenir ve yillikMaas 720.000 lira verir.



Şekil 6.3 maas alanı Muhendis nesnesinde vardır ama Calisan'ın private alanıdır; alt sınıf bile ona doğrudan ulaşamaz.

Alanı alt sınıflara açmanın yolu ise Calisan'da protected int maas; yazmaktır. **protected** bir alana alt sınıflar, başka bir pakette dursalar bile ulaşır; aynı paketteki sınıflar da ulaşır. Başka paketteki sıradan bir sınıf ise ulaşamaz: derleyici "maas has protected access in Calisan" der. Yine de alışkanlık, alanları private tutmaktır: protected bir alanı her alt sınıf istediği gibi değiştirebilir, maaşı eksiye bile düşürebilir. Alt sınıfa gereken iş, kuralı olan bir metotla açılır.

Neden derlenmedi?

videoda 33:50

Muhendis sınıfı Calisan'dan türüyor; Calisan'ın da ad ve maaş alan bir yapıcı metodu var: Calisan(String ad, int maas). Mühendisi aynı 2 değerle oluşturmak istedik:

```
public class Muhendis extends Calisan {
    private String uzmanlik;
}

Muhendis m = new Muhendis("Ayşe", 60000);
```

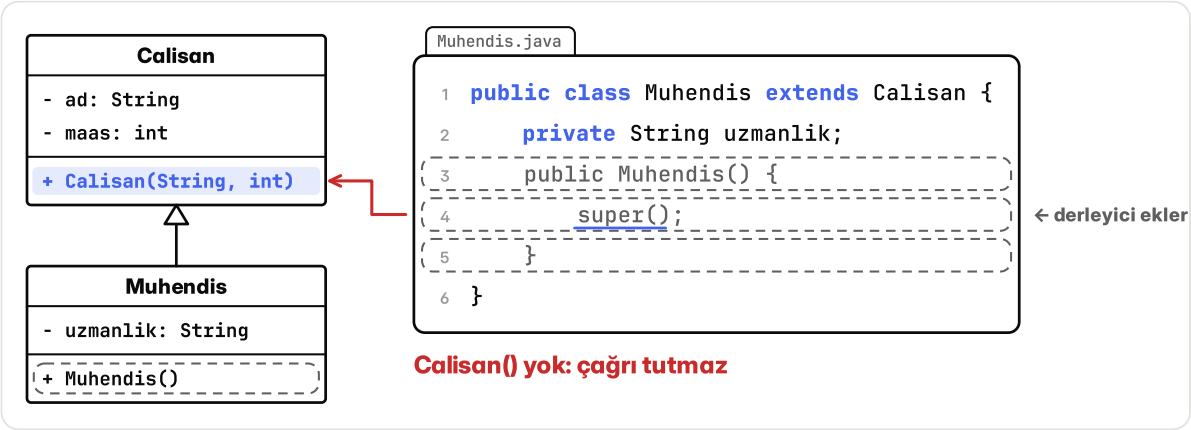
Derleyici 2 hata verdi:

```
Muhendis.java:1: error: constructor Calisan in class Calisan cannot be applied to given
types;
public class Muhendis extends Calisan {
       ^
  required: String,int
  found:    no arguments
  reason: actual and formal argument lists differ in length
Olustur.java:3: error: constructor Muhendis in class Muhendis cannot be applied to given
types;
        Muhendis m = new Muhendis("Ayşe", 60000);
                     ^
  required: no arguments
  found:    String,int
  reason: actual and formal argument lists differ in length
2 errors
```

Mühendis, çalışanın her şeyini devraldıysa, yapıcısını neden kullanamıyor?

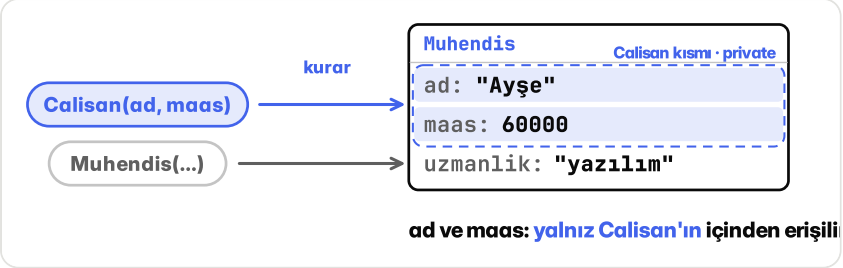
Hatalardan biri, nesneyi oluşturduğumuz satırda: Muhendis'in yapıcısı hiç değer almıyor (required: no arguments), biz ise 2 değer verdik. Çünkü yapıcılar devralınmaz; yapıcı yalnız kendi sınıfının nesnesini kurar. Mühendise yapıcı yazmadığımız için, derleyici ona parametresiz bir **varsayılan yapıcı** eklemiş.

Öteki hata, Muhendis sınıfının ilk satırında: Calisan'ın yapıcısı 2 değer bekliyor (required: String,int), ama hiç değer verilmemiş. Oysa Calisan'ın yapıcısını hiç çağırmadık; bu çağrı nereden çıktı? Her yapıcı işe, üst sınıfın yapıcısını çağırarak başlar; biz yazmazsak derleyici, parametresiz bir çağrıyı kendisi ekler. Derleyicinin eklediği yapıcı public Muhendis() { super(); } gibidir. Calisan'ın parametresiz bir yapıcısı olmadığı için bu çağrı tutmaz.



Şekil 6.4 Derleyici Muhendis'e public Muhendis() { super(); } ekler; Calisan() olmadığı için çağrı tutmaz.

Bunun nedeni şu: mühendis nesnesinin içinde bir de çalışan kısmı vardır, yani ad ve maas. Bu alanlar Calisan sınıfına kapalıdır; onları ancak Calisan'ın yapıcısı kurabilir.



Şekil 6.5 Muhendis nesnesinin Calisan kısmını (ad, maas) Calisan'ın yapıcısı, uzmanlik alanını Muhendis'in yapıcısı kurar.

Çözüm, mühendise kendi yapıcısını yazmaktır. Yapıcının ilk satırında super, adı ve maaşı Calisan'ın yapıcısına iletir; ardından mühendis kendi alanını, uzmanlığı kurar:

```
public class Muhendis extends Calisan {
    private String uzmanlik;
    public Muhendis(String ad, int maas, String uzmanlik) {
        super(ad, maas);
        this.uzmanlik = uzmanlik;
    }
}
```

super, üst sınıf demektir; yanına () gelince üst sınıfın yapıcısını çağırır. Kod artık derlenir; new Muhendis("Ayşe", 60000, "yazılım") ile kurulan nesnede getAd() Ayşe, aylikOdeme() 60000 verir.

Sırayı görmek için 3 kat kuralım: kıdemli mühendis bir mühendistir, mühendis de bir çalışandır. Her yapıcı, kurulduğunu ekrana yazsın:

```
public class Calisan {
    public Calisan() {
        System.out.println("Çalışan kuruldu");
    }
}

public class Muhendis extends Calisan {
    public Muhendis() {
        System.out.println("Mühendis kuruldu");
    }
}
```

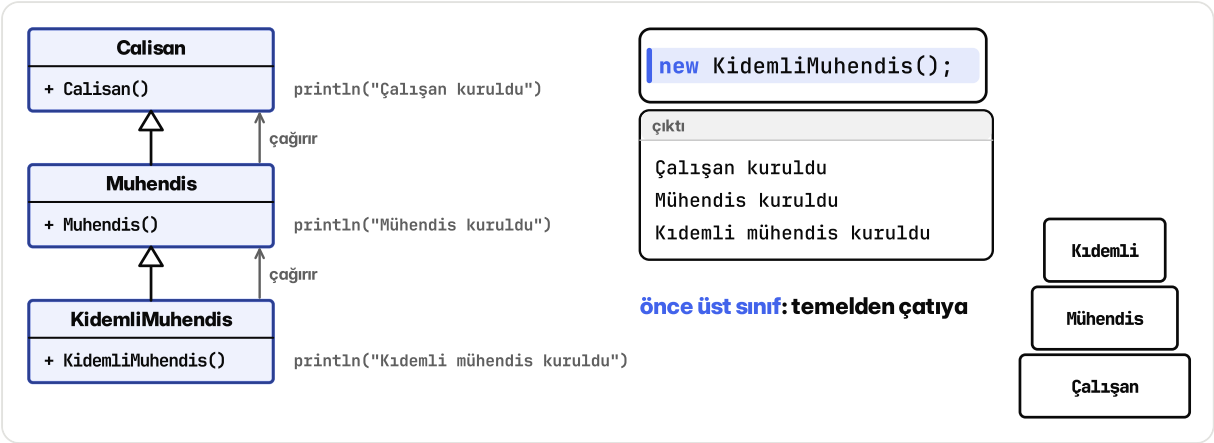
Üçüncü kat:

```
public class KidemliMuhendis extends Muhendis {
    public KidemliMuhendis() {
        System.out.println("Kıdemli mühendis kuruldu");
    }
}
```

new KidemliMuhendis(); satırının çıktısı:

```
Çalışan kuruldu
Mühendis kuruldu
Kıdemli mühendis kuruldu
```

Hiçbir yapıcıya super yazmadık; derleyicinin eklediği super(); çağrıları işi görür. Her yapıcı önce bir üstünü çağırdığı için kurulum en tepeden başlar; bina gibi, temel atılmadan kat çıkılmaz.



Şekil 6.6 KidemliMuhendis Muhendis'i, Muhendis Calisan'ı çağırır; çıktı ise en tepeden başlar: Çalışan, Mühendis, Kıdemli.

Prim nereye gitti?

videoda 35:30

Satışçı Zeynep'in maaşı 40.000 lira; bu ay 15.000 lira da satış primi var. Satisci sınıfı, aylikOdeme metodunu Calisan sınıfından devraldı:

```
public class Satisci extends Calisan {
    private int prim;
}

Satisci z = new Satisci("Zeynep", 40000, 15000);
System.out.println(z.aylikOdeme());
```

Çıktı:

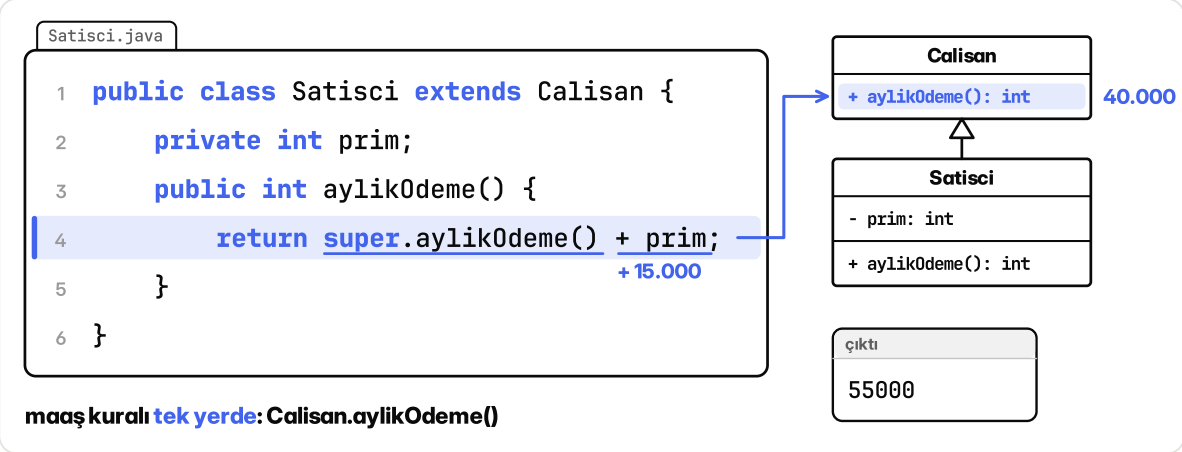
```
40000
```

Prim nereye gitti? Devralınan metot Calisan sınıfı için yazıldı; içinde yalnız return maas; vardır, primden haberi yoktur. Satışçının ödemesi ise başka türlü hesaplanır: maaş + prim. Metot, satışçıya göre değişmeli.

Alt sınıf, devraldığı bir metodu aynı ad ve aynı parametrelerle kendi içinde yeniden yazabilir. Buna metodu **yeniden tanımlamak** denir; yazılımcılar buna **override** der. Artık bir Satisci nesnesinde Calisan'ın metodu değil, Satisci'ninki çalışır. Yeni metot maaşa primi eklesin:

```
public class Satisci extends Calisan {
    private int prim;
    public int aylikOdeme() {
        return maas + prim;
    }
}
```

Ama maas Calisan sınıfında private olduğu için return satırı derlenmez: "maas has private access in Calisan". Maaşı veren hesap zaten üst sınıfta yazılı. super sözcüğüyle üst sınıfın aylikOdeme metodunu çağırırız: satır return super.aylikOdeme() + prim; olur. super.aylikOdeme() 40.000 verir, üstüne primi ekleyince Zeynep'in ödemesi 55.000 olur. Böylece hesap tek yerde kalır; Calisan'ın kuralı değişince, Satisci'ninki de değişir.

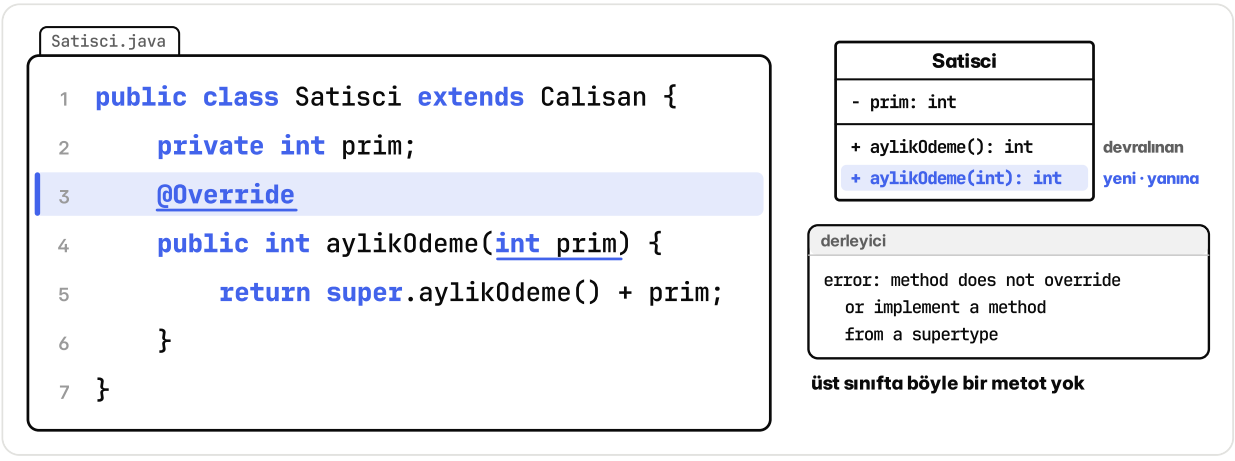


Şekil 6.7 super.aylikOdeme() Calisan'ın metodundan 40.000 getirir, + prim 15.000 eklenir; maaş kuralı tek yerde kalır.

Bir gün metoda, primi dışarıdan alsın diye bir parametre ekledik: public int aylikOdeme(int prim). Kod derlendi; ama Zeynep'e yine 40.000 ödendi. Parametreleri farklı olduğu için bu yeni bir metottur; eskisinin yerine geçmez, yanına eklenir. z.aylikOdeme() çağrısı hâlâ Calisan'dan devralınan metodu bulur.

Bu hatayı yakalamak için metodun üstüne **@Override** yazarız. Derleyici, üst sınıfta yeniden tanımlanacak böyle bir metot bulamayınca durur:

```
Satisci.java:9: error: method does not override or implement a method from a supertype
    @Override
    ^
1 error
```

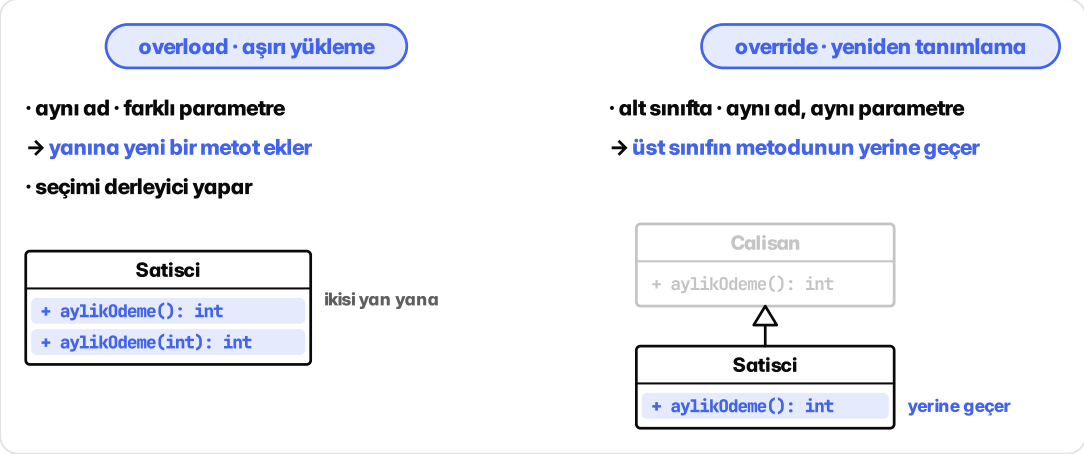


Şekil 6.8 Satisci'de devralınan aylikOdeme() ile yeni aylikOdeme(int) yan yana durur; @Override yazılınca derleyici "üst sınıfta böyle bir metot yok" der.

Doğru hâli şöyledir; çıktısı 55000:

```
public class Satisci extends Calisan {
    private int prim;
    @Override
    public int aylikOdeme() {
        return super.aylikOdeme() + prim;
    }
}
```

Bu iki kavram sık karışır. **Aşırı yükleme** (overload), aynı adla, parametreleri farklı yeni bir metot ekler; hangisinin çalışacağını derleyici, verilen değerlere bakarak seçer. println(int) ve println(String) böyledir. Yeniden tanımlama ise alt sınıfta, aynı ad ve aynı parametrelerle yazılır; üst sınıfın metodunun yerine geçer. Kısacası overload yanına ekler, override yerine geçer.



Şekil 6.9 Overload: aynı ad, farklı parametre, iki metot yan yana. Override: alt sınıfta aynı ad ve parametre, üst sınıfın metodunun yerine geçer.

Bu yazı nereden geldi?

videoda 37:05

Bir çalışan nesnesini doğrudan ekrana yazdıralım. Calisan bu kez sicil numarası ve ad tutuyor:

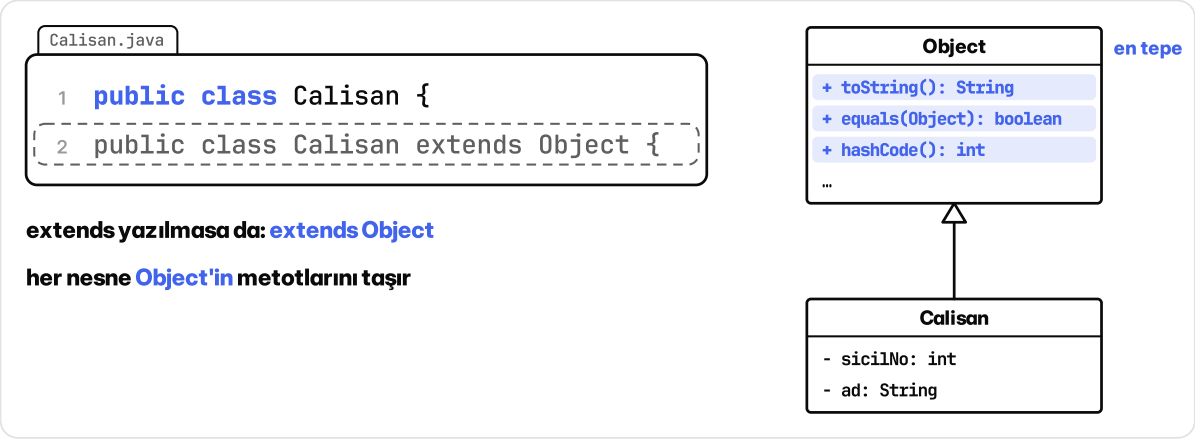
```
public class Calisan {
    private int sicilNo;
    private String ad;
    public Calisan(int sicilNo, String ad) {
        this.sicilNo = sicilNo;
        this.ad = ad;
    }
}

Calisan c = new Calisan(1042, "Ayşe");
System.out.println(c);
```

Bu çalıştırmada çıktı:

```
Calisan@4a574795
```

Ekrana çalışanın adı çıkmadı; sınıfın adı, bir @ işareti ve anlamsız görünen harflerle rakamlar çıktı. Bu yazıyı biz yazmadık; peki nereden geldi?

Java'da her sınıfın bir üst sınıfı vardır. extends yazmadığımız bir sınıf, **Object** adlı sınıftan türer; bütün sınıfların en tepedeki atası odur. Bu yüzden her nesne, Object'ten gelen birkaç metodu hazır taşır; en çok kullanılan üçü toString, equals ve hashCode.



Şekil 6.10 extends yazılmasa da Calisan, Object'ten türer; her nesne Object'in toString, equals ve hashCode metotlarını taşır.

println bir nesneyi yazdırırken, önce onun metin hâlini ister: bu işi **toString** metodu yapar. Object'teki toString, sınıfın adını, bir @ işaretini ve nesneye ait bir sayıyı 16'lık tabanda yazar. Bu sayıya **hash kodu** denir; çoğu kişi onu bellek adresi sanır, oysa nesne bellekte yer değiştirse bile bu sayı aynı kalır. toString'i Calisan sınıfında yeniden tanımlayınca, ekrana sicil numarası ve ad çıkar:

```
@Override
public String toString() {
    return sicilNo + " · " + ad;
}
```

Aynı program artık 1042 · Ayşe yazar.

Aynı çalışanı 2 ayrı yerden okuyup 2 nesne oluşturduk; sicil numaraları da adları da aynı. **equals** ile karşılaştıralım:

```
Calisan a = new Calisan(1042, "Ayşe");
Calisan b = new Calisan(1042, "Ayşe");
System.out.println(a.equals(b));
```

Sonuç false çıkar. Çünkü Object'teki equals, == gibi çalışır: 2 değişkenin aynı nesneyi gösterip göstermediğine bakar. Calisan sınıfında yeniden tanımlarız: gelen nesne bir Calisan ise sicil numaralarını karşılaştırır. instanceof, nesnenin Calisan olup olmadığına bakar; doğruysa onu c adıyla kullanmamıza izin verir:

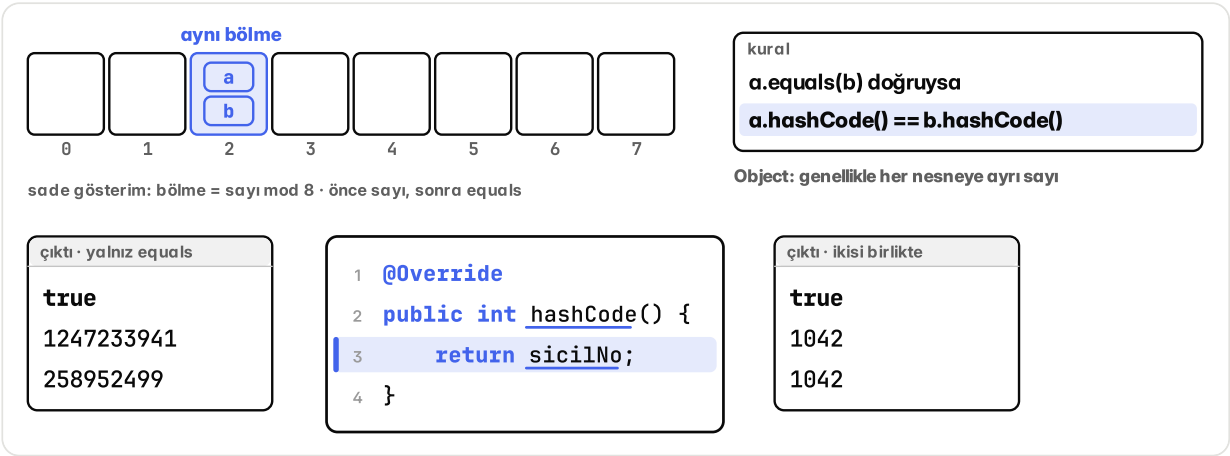
```
@Override
public boolean equals(Object o) {
    return o instanceof Calisan c && sicilNo == c.sicilNo;
}
```

Sonuç artık true. Metinlerde equals'ın harflere bakması da bundandır: String sınıfı onu yeniden tanımlamıştır.

hashCode metodu, nesneden bir tam sayı üretir; hızlı arama yapan koleksiyonlar, nesneyi önce bu sayıyla bulur. Kural şudur: equals'a göre eşit olan 2 nesne, aynı hash kodunu vermeli. Object'teki hashCode ise genellikle her nesneye ayrı bir sayı verir. Yalnız equals'ı yeniden tanımlayıp yukarıdaki üç satıra a.hashCode() ve b.hashCode() satırlarını ekleyince bu çalıştırmada true, 1247233941 ve 258952499 çıkar: eşit 2 çalışanın sayıları farklıdır ve bu koleksiyonlarda 2 ayrı kişi gibi görünürler. Bu yüzden ikisi hep birlikte yazılır; burada hash kodu, sicil numarasının kendisidir:

```
@Override
public int hashCode() {
    return sicilNo;
}
```

Çıktı artık true, 1042 ve 1042 olur.



Şekil 6.11 Kural: a.equals(b) doğruysa a.hashCode() == b.hashCode(); yalnız equals yazılınca eşit iki nesne ayrı bölmelere, ikisi birlikte yazılınca aynı bölmeye düşer.

String'den türemek

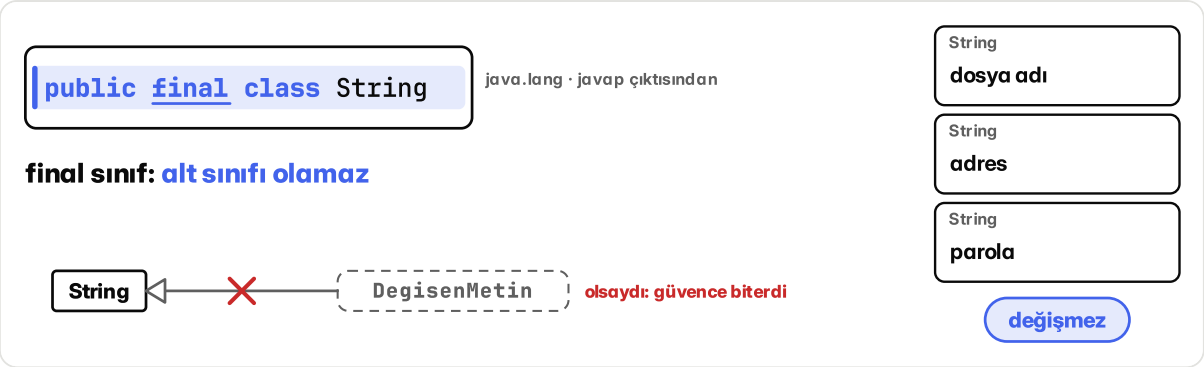
videoda 38:46

String'den türeyen bir sınıf yazıp, ona kendi metodumuzu eklemek istedik. Daha metot eklemeden, 2 satırlık boş sınıf bile derlenmedi:

```
public class Metin extends String {
}
```

Derleyici "cannot inherit from final String" der: final bir sınıftan türenemez. final ne demek; kalıtımın kapısı başka nerelerde kapalı?

Bir sınıfın başına **final** yazılırsa, o sınıftan hiçbir sınıf türeyemez. String sınıfı böyledir: public final class String. Çünkü String'ler hiç değişmez; dosya adı, adres ya da parola gibi bilgiler bu güvenceyle taşınır. Değişebilen bir alt sınıf yazılabilseydi, String bekleyen her kod bu güveni kaybederdi.



Şekil 6.12 String final bir sınıftır; dosya adı, adres ve parola değişmez metinlerdir, değişebilen bir DegisenMetin alt sınıfı bu güvenceyi bitirirdi.

final, metotlarda da kullanılır: final bir metot, alt sınıfta yeniden tanımlanamaz. Turnike, çalışanın kartını Calisan sınıfındaki bir metotla denetliyor: okunan numara, sicil numarasıyla aynı mı?

```
public class Calisan {
    private int sicilNo;
    public boolean kartGecerli(int okunan) {
        return okunan == sicilNo;
    }
}
```

Stajyer sınıfı bu metodu yeniden tanımlayıp her karta evet derse, kartı tutmayan herkes içeri girer:

```
@Override
public boolean kartGecerli(int okunan) {
    return true;
}
```

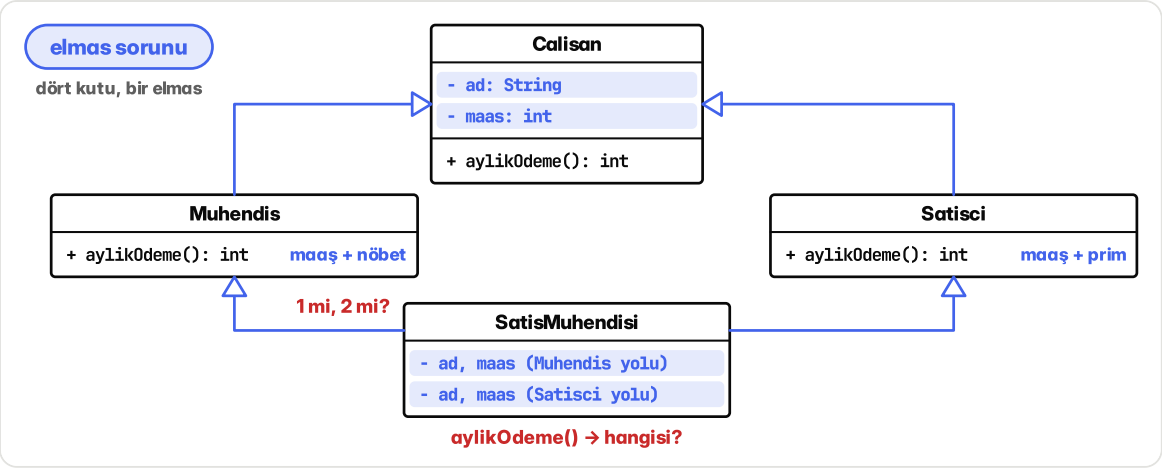
Turnike 9999 numaralı kartı okuyunca sicili 1042 olan çalışan için false, sicili 2051 olan stajyer için true çıkar. Metodun başına final yazınca, yani public final boolean kartGecerli(int okunan) olunca, Stajyer'deki tanım derlenmez; derleyici "overridden method is final" der. Denetim, bütün alt sınıflarda aynı kalır.

Bir sınır daha var: satış mühendisi hem mühendis hem de satışçıdır. İki sınıftan birden türeyebilir mi? Java'da hayır: extends'ten sonra yalnız bir sınıf adı yazılır, ikinci ad derlenmez:

```
public class SatisMuhendisi extends Muhendis, Satisci {
}
```

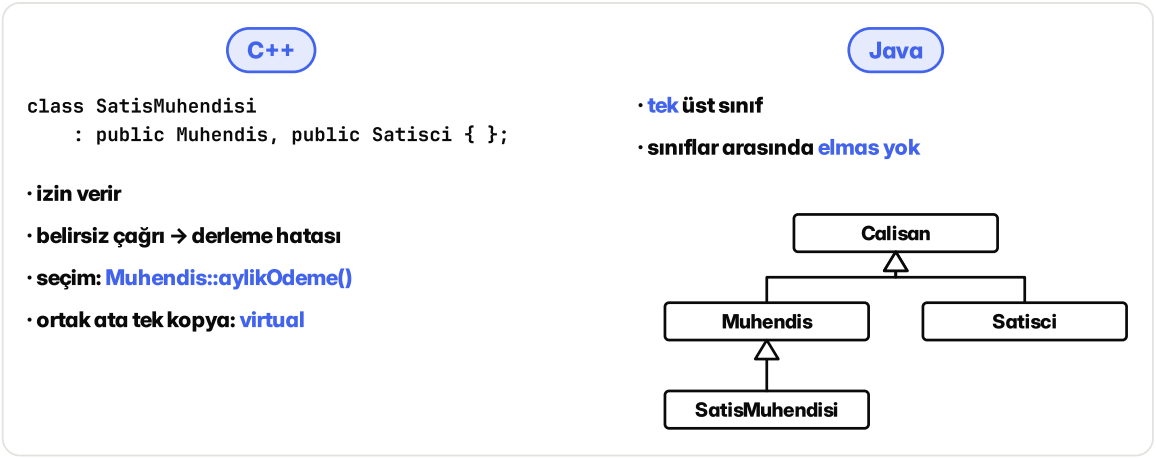
Derleyici virgülü görünce "'{' expected" der. Her sınıfın tek bir üst sınıfı olur.

Nedeni, diyagramı çizince görünür: en üstte Calisan, altında Muhendis ve Satisci, en altta SatisMuhendisi; 4 kutu bir elmas oluşturur. Muhendis'in ödeme metodu maaşa nöbet ücretini, Satisci'ninki primi ekliyorsa, satış mühendisi için hangisi çalışacak? Calisan'dan gelen ad ve maas da 2 yoldan iner: nesnede 1 mi olacak, 2 mi? Buna **elmas sorunu** denir.



Şekil 6.13 Calisan, Muhendis, Satisci ve SatisMuhendisi bir elmas çizer: aylikOdeme() hangisi, ad ve maas 1 mi 2 mi?

C++ birden çok üst sınıfa izin verir (class SatisMuhendisi : public Muhendis, public Satisci) ve cevabı programcıya bırakır. Programcı hangi metodun çalışacağını açıkça yazar; yazmazsa derleyici, çağrının belirsiz olduğunu söyler. Ortak atanın tek kopya olması için de ayrı bir anahtar sözcük (virtual) kullanır. Java bu yükü baştan kaldırır: her sınıfın tek bir üst sınıfı olur, sınıflar arasında elmas hiç oluşmaz.



Şekil 6.14 C++ çoklu kalıtıma izin verir ve seçimi programcıya bırakır; Java'da her sınıfın tek üst sınıfı vardır, elmas oluşmaz.

Kare bir dikdörtgendir

videoda 40:31

Matematikte her kare bir dikdörtgendir: 4 açısı dik, kenarları da eşit olan bir dikdörtgen. O hâlde kodda da Kare sınıfı, Dikdortgen sınıfından türeyebilir; cümle doğru kuruluyor. Peki ne ters gider?

Dikdortgen sınıfında, eni ve boyu ayrı ayrı değiştiren 2 metot var; Kare bunları da devralır. Sınıfın en ve boy alan yapıcısını burada göstermiyoruz:

```
public class Dikdortgen {
    private int en;
    private int boy;
    public void setEn(int en) { this.en = en; }
    public void setBoy(int boy) { this.boy = boy; }
    public int getEn() { return en; }
    public int getBoy() { return boy; }
}
```

Kare'nin yapıcısı tek kenar alır ve onu hem en hem boy olarak üst sınıfa verir:

```
public class Kare extends Dikdortgen {
    public Kare(int kenar) {
        super(kenar, kenar);
    }
}
```

Kenarı 3 olan bir karenin enini 5 yapalım:

```
Kare k = new Kare(3);
k.setEn(5);
System.out.println(k.getEn() + " x " + k.getBoy());
```

Çıktı:

```
5 x 3
```

Boy 3'te kaldı: 5'e 3 bir kare olmaz. Kareyi korumak için 2 metodu Kare'de yeniden tanımlarız: biri değişince öteki de değişsin. setBoy şöyle olur, setEn de aynı biçimde iki alanı birden değiştirir:

```
@Override
public void setBoy(int boy) {
    super.setEn(boy);
    super.setBoy(boy);
}
```



Şekil 6.15 3'e 3 karenin eni 5 yapılınca 5 x 3 çıkar; Kare'de setEn ve setBoy biri değişince ötekini de değiştirecek biçimde yeniden tanımlanır.

Bu kez de dikdörtgenin sözü bozulur. Bir dikdörtgenin enini 5, boyunu 4 yapınca 5'e 4 bir dikdörtgen çıkar: 5 x 4. Aynı 2 satır karede 4 x 4 verir; boyu değiştirmek, eni de değiştirmiştir. Dikdörtgenle çalışan her kod, eni değiştirince boyun yerinde kalacağına güvenir; kare bu güveni bozar. Kural şudur: alt sınıf, üst sınıfın verdiği her sözü tutmalı; tutamıyorsa, cümle doğru görünse de kalıtım yanlıştır.



Şekil 6.16 Aynı setEn(5) ve setBoy(4) dikdörtgende 5 x 4, karede 4 x 4 verir; alt sınıf üst sınıfın her sözünü tutmalı.

Kalıtımın yanlış seçildiği bir yer daha var: zamanla değişen ilişkiler. Ayşe bir mühendis; müdürler için de ayrı bir Mudur alt sınıfı yazdık. Ayşe ekip müdürü olunca ne olacak? Nesnenin sınıfı sonradan değişmez; yeni bir nesne oluşturup bütün bilgilerini taşımak gerekir. Mühendis olan müdür, satışçı olan müdür derken sınıflar da katlanır.

Müdürlük, Ayşe'nin ne olduğu değil, üstlendiği bir görevdir; yani çalışanın bir görevi vardır. Bu yüzden kalıtım yerine sahiptir ilişkisi kurarız: Calisan sınıfına, Gorev tipinde bir alan ekleriz. Gorev, unvanı ve ek ödemeyi tutar. Calisan'ın ad alanı ve yapıcısı aşağıda gösterilmedi:

```
public class Calisan {
    private int maas;
    private Gorev gorev;
    public void gorevVer(Gorev yeni) { gorev = yeni; }
    public int aylikOdeme() {
        if (gorev == null) return maas;
        return maas + gorev.getEkOdeme();
    }
}
```

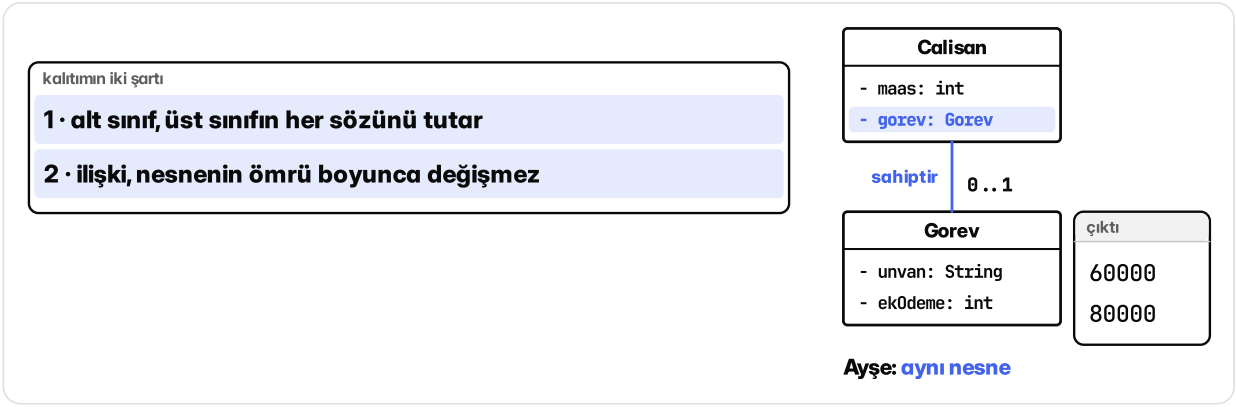
Görevi olmayan çalışana (gorev null iken) yalnız maaşı ödenir. Ayşe'ye ekip müdürlüğü verelim:

```
Muhendis ayse = new Muhendis("Ayşe", 60000, "yazılım");
System.out.println(ayse.aylikOdeme());
ayse.gorevVer(new Gorev("Ekip müdürü", 20000));
System.out.println(ayse.aylikOdeme());
```

Çıktı:

```
60000
80000
```

Aylık ödeme 60.000'den 80.000'e çıkar; Ayşe ise aynı nesne olarak kalır. Kısacası kalıtım 2 şey ister: alt sınıf üst sınıfın her sözünü tutmalı, ilişki de nesnenin ömrü boyunca değişmemeli.



Şekil 6.17 Calisan bir Gorev'e sahiptir (0..1); Ayşe aynı nesne kalır ve ödemesi 80.000 olur; altta kalıtımın iki şartı.

Özet

- Ortak alan ve metotlar üst sınıfta bir kez yazılır; alt sınıf extends ile türer ve yalnız kendine özgü olanı ekler. Ölçü "bir ...dır" cümlesidir.
- private alan alt sınıfın nesnesinde vardır ama alt sınıf ona doğrudan ulaşamaz; protected alt sınıflara ve aynı pakete açar, yine de alanlar çoğunlukla private tutulur.
- Yapıcılar devralınmaz; her yapıcı önce üst sınıfın yapıcısını çağırır, biz yazmazsak derleyici super(); ekler. Kurulum en tepedeki sınıftan başlar.
- Override, devralınan metodu alt sınıfta aynı ad ve parametrelerle yeniden yazar; super.metot() üst sınıfın hâlini çağırır, @Override yanlış imzayı derleyiciye yakalatır. Overload yanına ekler, override yerine geçer.
- Her sınıf Object'ten türer; toString metin hâlini verir, equals ve hashCode birlikte yeniden tanımlanır.
- final sınıftan türenemez, final metot yeniden tanımlanamaz; Java'da her sınıfın tek üst sınıfı vardır. Alt sınıf üst sınıfın her sözünü tutamıyorsa ya da ilişki zamanla değişiyorsa kalıtım yerine sahiplik kurulur.

Sorular

1 **"bir ...dır" ölçüsüne göre aşağıdaki sınıf çiftlerinden hangisinde ilki ikincisinden türeyebilir?**

A SiparisSatiri ve Siparis

B Oyuncu ve Takim

C Muhendis ve Calisan

D Gorev ve Calisan

2 **Calisan sınıfında maas alanı private int maas; diye tanımlı. Muhendis extends Calisan sınıfına return maas * 12; satırı yazılınca ne olur?**

A Derlenmez; maas Muhendis nesnesinde vardır ama private olduğu için alt sınıf ona doğrudan ulaşamaz.

B Derlenir; alt sınıf üst sınıfın bütün alanlarına ulaşır.

C Derlenir ama maas 0 okunur, çünkü private alanlar alt sınıfın nesnesinde bulunmaz.

D Derlenmez; Muhendis'te maas adında bir alan yoktur, nesnede de bulunmaz.

3 **Calisan'ın tek yapıcısı Calisan(String ad, int maas). Muhendis extends Calisan sınıfına hiç yapıcı yazılmadı. Muhendis sınıfı neden derlenmez?**

A Alt sınıfın en az bir alanı olmalıdır.

B Yapıcılar devralındığı için Muhendis'te iki ayrı yapıcı çakışır.

C Calisan'ın yapıcısı public değildir.

D Derleyicinin eklediği varsayılan yapıcı super(); çağırır, ama Calisan'ın parametresiz yapıcısı yoktur.

4 **Calisan'ın parametresiz yapıcısı System.out.print("1"); yazıyor. Muhendis extends Calisan'ınki System.out.print("2"); ve KidemliMuhendis extends Muhendis'inki System.out.print("3"); yazıyor. Hiçbir yapıcıda super yazılı değil. new KidemliMuhendis(); satırı ekrana ne yazar?**

Cevap:

5 **Satisci sınıfındaki metot şöyle:**

```
@Override
public int aylikOdeme() {
    return super.aylikOdeme() + prim;
}
Calisan'ın aylikOdeme metodu maaşı döndürüyor. new Satisci("Deniz", 30000,
5000).aylikOdeme() kaç verir? (İkinci değer maaş, üçüncüsü prim.)
```

Cevap:

6 **Satisci sınıfında @Override yazmadan public int aylikOdeme(int prim) metodu tanımlandı; Calisan'daki aylikOdeme() ise maaşı döndürüyor. Satisci z = new Satisci("Zeynep", 40000, 15000); için z.aylikOdeme() ne verir?**

A 55000

B 40000

C 15000

D Kod derlenmez.

7 **Bir metodun üstüne @Override yazmanın işi nedir?**

A Metodu final yapar, alt sınıflar onu değiştiremez.

B Metodun üst sınıftaki hâlini otomatik olarak çağırır.

C Üst sınıfta aynı imzalı bir metot yoksa derleyicinin hata vermesini sağlar.

D Aynı adla, parametreleri farklı yeni bir metot ekler.

8 **Calisan sınıfında equals sicil numarasına göre yeniden tanımlandı, hashCode ise tanımlanmadı. Bunun sonucu nedir?**

- **A** equals'a göre eşit iki çalışan farklı hash kodları verebilir ve hızlı arama yapan koleksiyonlarda iki ayrı kişi gibi görünür.
- **B** equals artık == gibi çalışır.
- **C** Kod derlenmez; ikisi birlikte yazılmak zorundadır.
- **D** toString artık sicil numarasını yazar.

9 **Calisan final olmayan bir sınıf, kartGecerli metodu ise final. Aşağıdakilerden hangisi derlenir?**

- **A** public class Metin extends String { }
- **B** public class Stajyer extends Calisan { }
- **C** public class SatisMuhendisi extends Muhendis, Satisci { }
- **D** Stajyer sınıfında kartGecerli metodunu yeniden tanımlamak

10 **Kare sınıfı Dikdortgen'den türüyor ve setEn ile setBoy'u biri değişince öteki de değişecek biçimde yeniden tanımlıyor. Aşağıdaki satırlar ekrana ne yazar?**

```
Kare k = new Kare(3);
k.setEn(5);
k.setBoy(4);
System.out.println(k.getEn() + " x " + k.getBoy());
```

Cevap: _____

NESNE YÖNELİMLİ PROGRAMLAMA

7 Çok biçimlilik

Aynı satır, arkasındaki nesneye göre farklı iş yapabilir. Bu bölümde çok biçimliliği (polymorphism) göreceğiz: üst sınıf tipindeki bir değişkenin alt sınıf nesnesi tutmasını, çalışacak metodun program çalışırken seçilmesini, yeni bir türü eski koda dokunmadan eklemeyi, değişkenin tipiyle nesnenin tipi arasındaki farkı ve tip dönüşümlerini.

Aynı satır, iki sonuç

videoda 42:08

Bu programı iki kez çalıştıralım. Kullanıcı bir sayı giriyor, program ona göre bir şekil oluşturup alanını yazıyor:

```
int secim = giris.nextInt();
Sekil s;
if (secim == 1) {
    s = new Daire(1);
} else {
    s = new Kare(2);
}
System.out.println(s.alan());
```

giris, klavyeden okuyan bir Scanner nesnesidir. Kullanıcı 1 yazınca ekrana 3.141592653589793 çıkıyor: yarıçapı 1 olan dairenin alanı, yani pi sayısı. 2 yazınca 4.0 çıkıyor: kenarı 2 olan karenin alanı. İkisini de aynı satır, son satır yazdırıyor. Bu satır, hangi şeklin alanını hesaplayacağını nereden biliyor?

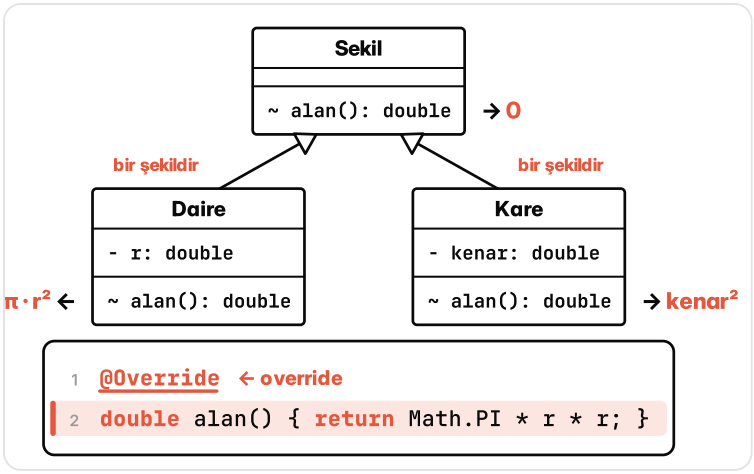
Programda üç sınıf var: Sekil, Daire ve Kare. Daire ve Kare, Sekil'den türetilmiş alt sınıflardır; her daire bir şekildir, her kare de. Üçünde de alan metodu var. Sekil, hangi şekil olduğunu bilmediği için alan olarak 0 verir. Daire pi çarpı yarıçapın karesini hesaplar:

```
class Sekil {
    double alan() { return 0; }
}

class Daire extends Sekil {
    private double r;
    Daire(double r) { this.r = r; }
    @Override
    double alan() { return Math.PI * r * r; }
}
```

Kare de kenarın karesini hesaplar:

```
class Kare extends Sekil {
    private double kenar;
    Kare(double kenar) { this.kenar = kenar; }
    @Override
    double alan() { return kenar * kenar; }
}
```

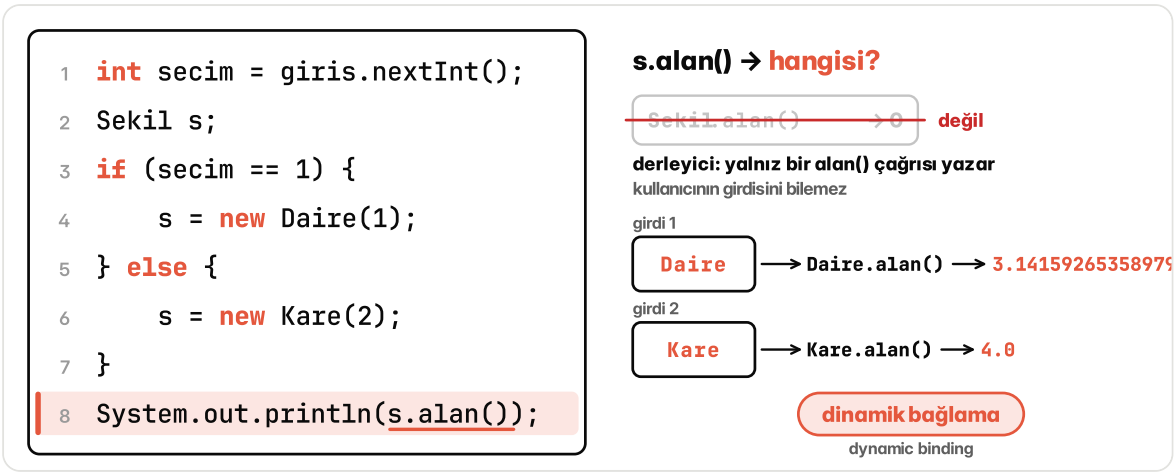
Yani iki alt sınıf, üst sınıftan gelen metodu kendine göre yeniden yazmış; buna override denir.



Şekil 7.1 Sekil'in altında Daire ve Kare; ikisi de "bir şekildir" ve üst sınıftan gelen alan() metodunu override eder.

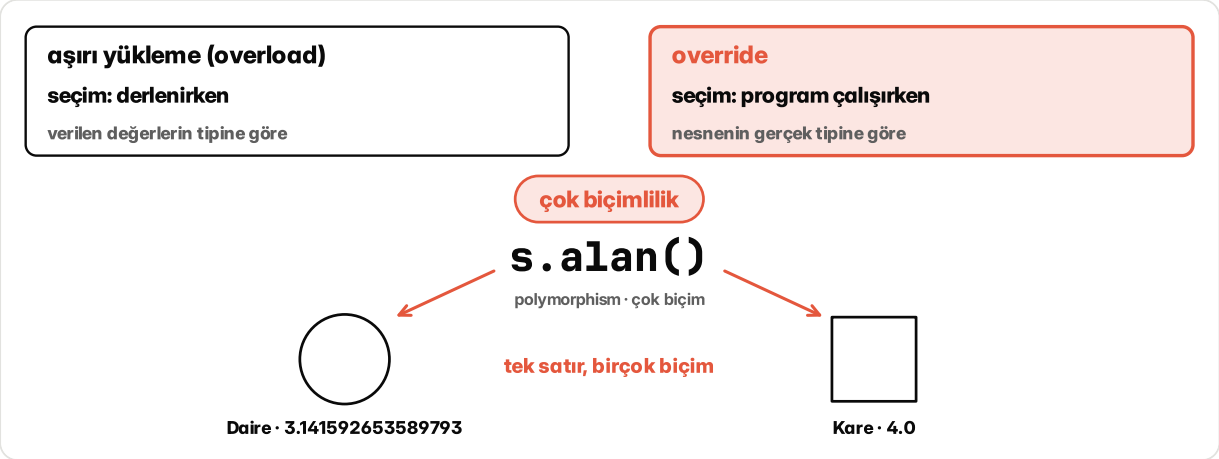
Püf noktası, s değişkeninin tipinde: Daire ya da Kare değil, Sekil. Java'da üst sınıf tipindeki bir değişken, alt sınıflardan birinin nesnesini de tutabilir. Her daire bir şekil olduğu için, şekil beklenen yere bir daire konabilir. Böylece s'ye, kullanıcının seçimine göre bazen bir daire, bazen bir kare yazılır; ikisi de geçerlidir.

Peki s.alan() dendiğinde hangi alan metodu çalışır? Sekil sınıfının 0 veren metodu değil; s'nin o an gösterdiği nesnenin kendi metodu. Bu seçim derleme sırasında yapılamaz, çünkü kullanıcının ne yazacağını derleyici bilemez; bu satıra yalnız bir alan çağrısı yazar. Java, program çalışırken nesnenin gerçek tipine bakar: nesne daireyse Daire'nin, kareyse Kare'nin alan metodunu çalıştırır. Çalışacak metodun program çalışırken, nesneye bakılarak seçilmesine **dinamik bağlama** (dynamic binding) denir.



Şekil 7.2 Derleyici yalnız bir alan() çağrısı yazar; girdi 1'de Daire.alan() 3.141592653589793, girdi 2'de Kare.alan() 4.0 verir.

Aşırı yüklenmiş metotlar arasında seçimi derleyici yapar, verilen değerlerin tipine bakarak. Override edilen metotta ise karar, program çalışırken nesneye bırakılır. Aynı çağrının, nesnenin türüne göre farklı iş yapmasına **çok biçimlilik** (polymorphism) denir. Adı da buradan gelir: tek bir satır, arkasındaki nesneye göre birçok biçimde çalışır.



Şekil 7.3 Aşırı yüklemede seçim derlenirken, override'da program çalışırken yapılır; tek s.alan() satırı Daire'de 3.141592653589793, Kare'de 4.0 verir.

Üçgen geliyor

videoda 43:52

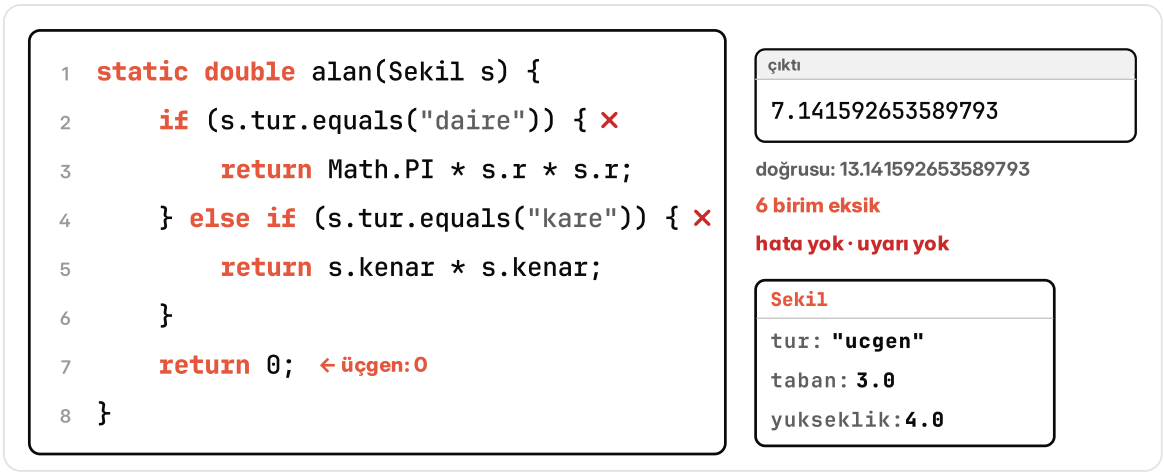
Bir çizim programı, ekrandaki bütün şekillerin toplam alanını hesaplıyor: daireleri ve kareleri. Yarın programa üçgen de eklenecek. Alanları toplayan koda hiç dokunmadan bu mümkün mü?

Çok biçimlilik olmadan bu iş şöyle yazılır: her şekil, türünü bir etikette taşır. Sekil sınıfında tur adlı bir metin alanı ("daire", "kare") ve her türün ölçüleri için alanlar (r, kenar, taban, yukseklik) vardır. Alanı hesaplayan metot da önce etikete bakar: daireyse pi çarpı yarıçapın karesi, kareyse kenarın karesi:

```
static double alan(Sekil s) {
    if (s.tur.equals("daire")) {
        return Math.PI * s.r * s.r;
    } else if (s.tur.equals("kare")) {
        return s.kenar * s.kenar;
    }
    return 0;
}
```

Yarıçapı 1 olan bir daire ve kenarı 2 olan bir karede toplam doğru çıkar: pi + 4, yani 7.141592653589793. Ama her yeni şekil türü, bu if-else zincirine yeni bir dal ekler.

Üçgen gelince zincire yeni bir dal yazmak gerekir; hem de yalnız burada değil, şeklin türüne bakan her metotta: çevrede, çizimde, kaydetmede. Birini unutmak kolaydır ve unutulan dal hata vermez. Alan metodunda üçgenin dalı unutulursa en sondaki return 0; satırı çalışır ve üçgenin alanı 0 sayılır. Tabanı 3, yüksekliği 4 olan üçgenin 6 birimlik alanı toplama hiç girmez; ekrana yine 7.141592653589793 çıkar ve program hiçbir uyarı vermeden çalışmaya devam eder.



Şekil 7.4 Etiketi "ucgen" olan şekil iki dalın da yanından geçer ve return 0'a düşer; çıktı 7.141592653589793 kalır, doğrusu 13.141592653589793; hata ve uyarı yok.

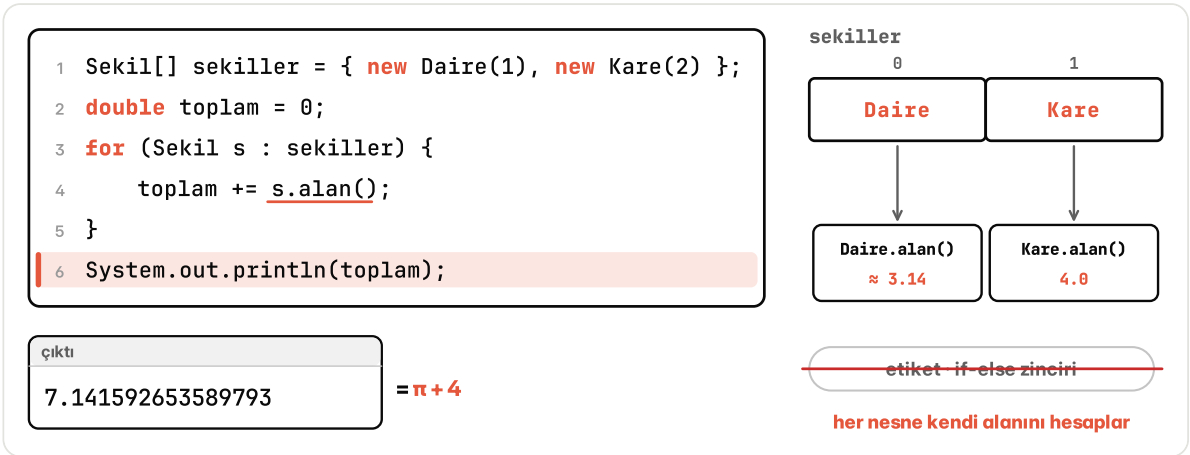
Çok biçimlilikle şekiller, Sekil tipindeki bir dizide durur: daire de kare de bir şekildir. Sekil, Daire ve Kare bir önceki örnekteki sınıflardır:

```
Sekil[] sekiller = { new Daire(1), new Kare(2) };
double toplam = 0;
for (Sekil s : sekiller) {
    toplam += s.alan();
}
System.out.println(toplam);
```

Çıktı:

```
7.141592653589793
```

Döngü, dizideki şekilleri sırayla s'ye verir ve tek bir iş yapar: s'nin alanını toplama ekler. Döngüde ne etiket var ne if; dairenin alanını Daire sınıfı, karenin alanını Kare sınıfı hesaplar. Hangi alan metodunun çalışacağını, program çalışırken nesnenin kendisi belirler. Toplam yine 7.141592653589793.



Şekil 7.5 sekiller dizisinde bir Daire ve bir Kare; her nesne kendi alanını hesaplar (≈ 3.14 ve 4.0), etiket ve if-else zinciri gerekmez.

Üçgen eklenince yeni bir alt sınıf yazarız: Ucgen de Sekil'den türetilir ve alanını taban çarpı yükseklik bölü 2 diye hesaplar:

```
class Ucgen extends Sekil {
    private double taban, yukseklik;
    Ucgen(double taban, double yukseklik) {
        this.taban = taban;
        this.yukseklik = yukseklik;
    }
    @Override
    double alan() { return taban * yukseklik / 2; }
}
```

Diziye bir üçgen eklemek yeter; döngünün tek harfi bile değişmez. Programın yalnız ilk satırı şöyle olur:

```
Sekil[] sekiller = { new Daire(1), new Kare(2), new Ucgen(3, 4) };
```

Üçgenin 3 × 4 / 2 = 6 birimlik alanı toplama girer ve ekrana 13.141592653589793 çıkar. Yeni tür, eski koda dokunmadan eklenir; çünkü kod şeklin türünü sormaz, şekle alanını sorar.

Daire ama çapı yok

videoda 45:34

Bu kez Daire sınıfında çapı veren bir metot da var; Sekil ve Kare öncekilerle aynı:

```
class Daire extends Sekil {
    private double r;
    Daire(double r) { this.r = r; }
    @Override
    double alan() { return Math.PI * r * r; }
    double cap() { return 2 * r; }
}
```

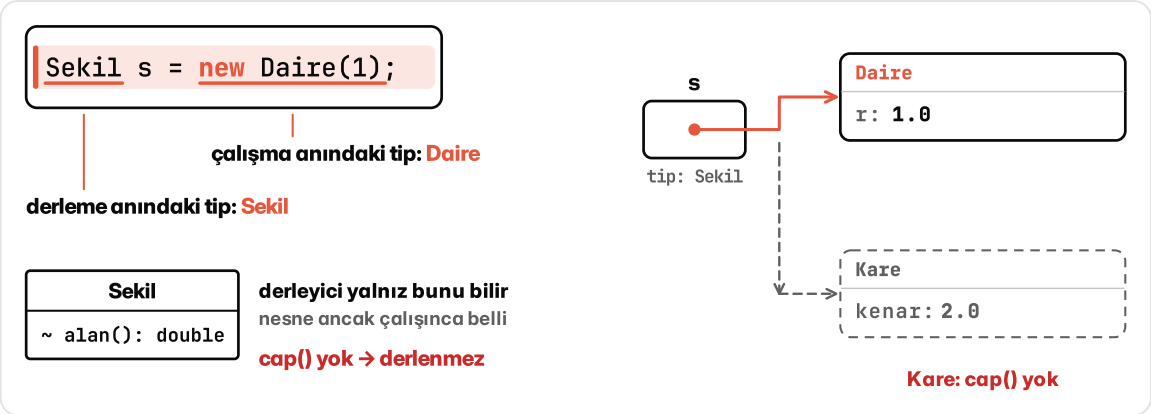
s değişkeni bir daireyi tutuyor; ama s.cap() yazınca kod derlenmiyor:

```
Sekil s = new Daire(1);
System.out.println(s.cap());
```

Derleyici böyle bir metot bulamadığını söylüyor:

```
CapSor.java:4: error: cannot find symbol
        System.out.println(s.cap());
                           ^
  symbol:   method cap()
  location: variable s of type Sekil
1 error
```

Nesne bir daire olduğu hâlde neden? Çünkü bu satırda iki ayrı tip var. **Derleme anındaki tip**, değişkenin tanımında yazan tiptir: s için Sekil. **Çalışma anındaki tip** ise değişkenin o an gösterdiği nesnenin gerçek tipidir: burada Daire. Derleyici yalnız birincisini bilir; ikincisi ancak program çalışınca belli olur. s'ye bir kare de yazılabilirdi ve karenin çapı yoktur. Bu yüzden derleyici, Sekil sınıfında olmayan bir metodun çağrılmasına izin vermez.



Şekil 7.6 s'nin derleme anındaki tipi Sekil, nesnesi Daire; derleyici yalnız Sekil'i bilir ve Sekil'de cap() olmadığı için satır derlenmez.

Kural şöyle: hangi metotların çağrılabileceğini derleme anındaki tip belirler; çağrılan metodun hangi sınıftaki hâlinin çalışacağını ise çalışma anındaki tip. Aynı s ile System.out.println(s.alan()); derlenir, çünkü Sekil sınıfında alan metodu vardır; program çalışınca ise Daire'nin alan metodu çalışır ve pi sayısı, 3.141592653589793 çıkar.

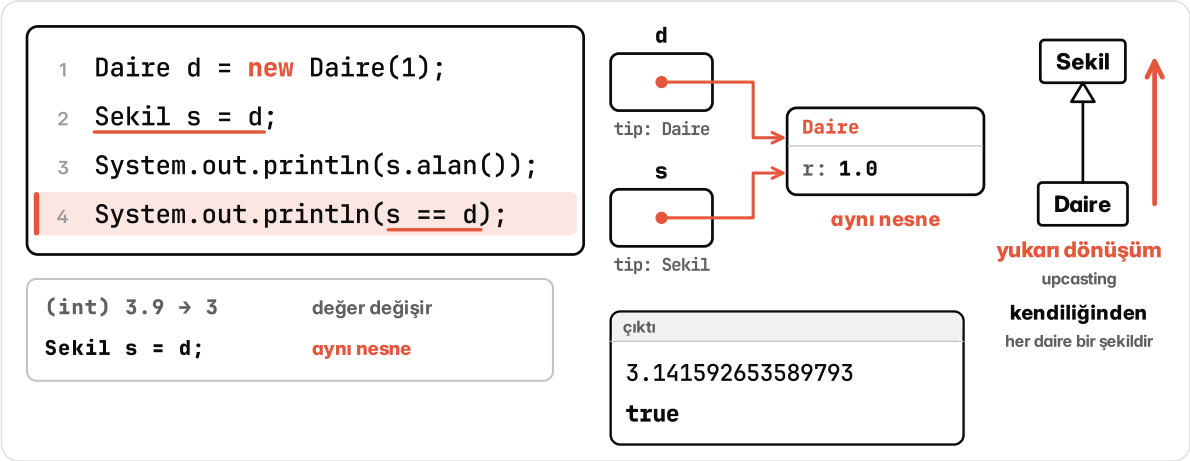
Bir daireyi Sekil tipindeki bir değişkene yazmak, **yukarı dönüşümdür** (upcasting): sınıf ağacında alt sınıftan üst sınıfa çıkarız. Bunu derleyici kendiliğinden yapar, çünkü her zaman güvenlidir: her daire bir şekildir. Bu dönüşümde nesneye hiçbir şey olmaz:

```
Daire d = new Daire(1);
Sekil s = d;
System.out.println(s.alan());
System.out.println(s == d);
```

Çıktı:

```
3.141592653589793
true
```

Sayılarda 3,9'u tam sayıya çevirmek, (int) 3.9, değeri 3'e indirir; nesne dönüşümünde ise aynı nesneye yalnız daha genel bir tipten bakılır. Daire değişkeniyle Sekil değişkenini == ile karşılaştırınca true çıkar: ikisi de aynı nesneyi gösterir.

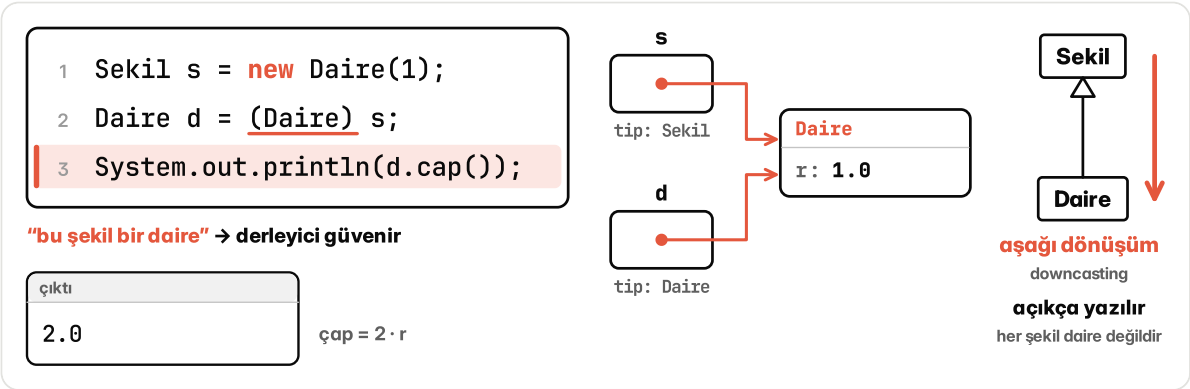


Şekil 7.7 d (tip Daire) ve s (tip Sekil) aynı Daire nesnesini gösterir; yukarı dönüşüm kendiliğinden olur ve nesneyi değiştirmez.

Ters yöne, Sekil'den Daire'ye inmek ise **aşağı dönüşümdür** (downcasting) ve kendiliğinden olmaz. Sekil tipindeki s'yi bir Daire değişkenine atayan Daire d = s; satırı derlenmez; derleyici "incompatible types: Sekil cannot be converted to Daire" der. Dönüşümü açıkça yazmamız gerekir: s'nin önüne, parantez içinde Daire yazarız:

```
Sekil s = new Daire(1);
Daire d = (Daire) s;
System.out.println(d.cap());
```

Bu, derleyiciye "bu şeklin bir daire olduğundan eminim" demektir. Derleyici sözümüze güvenir, çap metodu çağrılır ve ekrana 2.0 çıkar.



Şekil 7.8 Aşağı dönüşümde Sekil tipindeki s, (Daire) ile açıkça Daire tipindeki d'ye çevrilir; her şekil daire değildir.

Daireyle çalışıyor, kareyle çöküyor

videoda 47:19

Daire ve Kare, Sekil sınıfından türetilmiş iki sınıf; çapı veren metot yalnız Daire'de var. capYaz metodu, aldığı şekli daireye çevirip çapını yazdırıyor. main onu önce bir daireyle, sonra bir kareyle çağırıyor:

```
static void capYaz(Sekil s) {
    Daire d = (Daire) s;
    System.out.println(d.cap());
}

capYaz(new Daire(1));
capYaz(new Kare(2));
```

Çıktı:

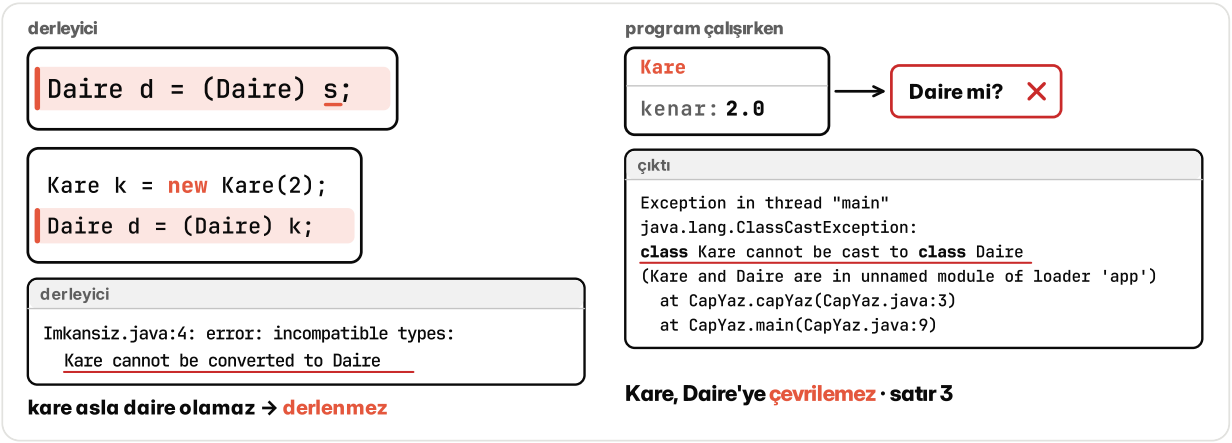
```
2.0
Exception in thread "main" java.lang.ClassCastException: class Kare cannot be cast to class
Daire (Kare and Daire are in unnamed module of loader 'app')
        at CapYaz.capYaz(CapYaz.java:3)
        at CapYaz.main(CapYaz.java:9)
```

Yarıçapı 1 olan bir daire verince ekrana 2.0 çıkıyor. Bir kare verince program duruyor. Derleyici bu dönüşümü hatasız kabul etmişti; neden uyarmadı?

Çünkü derleyici yalnız değişkenin tipini görür: Sekil tipindeki bir değer gerçekten bir daire olabilir, bu yüzden dönüşüme izin verir. Derleyici yalnız imkânsız olanı reddeder: tipi Kare olan bir değişkeni Daire'ye çeviren satır derlenmez, çünkü bir kare hiçbir zaman daire olamaz:

```
Kare k = new Kare(2);
Daire d = (Daire) k;
```

Bu satırda derleyici "incompatible types: Kare cannot be converted to Daire" der. Asıl denetim program çalışırken yapılır: Java nesnenin gerçek tipine bakar; nesne bir daire değilse programı **ClassCastException** hatasıyla durdurur. Mesajdaki "class Kare cannot be cast to class Daire" bölümü, Kare sınıfının Daire sınıfına çevrilemediğini söyler.



Şekil 7.9 Kare'yi Daire'ye çeviren satırı derleyici reddeder; Sekil tipindeki değerde ise denetim program çalışırken yapılır ve kare "Daire mi?" kapısından geçemez.

Çözüm, çevirmeden önce sormaktır. **instanceof**, bir nesnenin bir sınıftan olup olmadığını sorar: nesne o sınıftan ya da onun bir alt sınıfındansa true, değilse false verir:

```
Sekil s = new Kare(2);
System.out.println(s instanceof Daire);
System.out.println(s instanceof Sekil);
```

Çıktı false ve true olur; çünkü kare bir daire değildir ama bir şekildir. Metot artık önce sorar: şekil daireyse çevirir ve çapı yazar, değilse dönüşüme hiç girmez:

```
static void capYaz(Sekil s) {
    if (s instanceof Daire) {
        Daire d = (Daire) s;
        System.out.println(d.cap());
    } else {
        System.out.println("çapı yok");
    }
}
```

Aynı iki çağrının çıktısı:

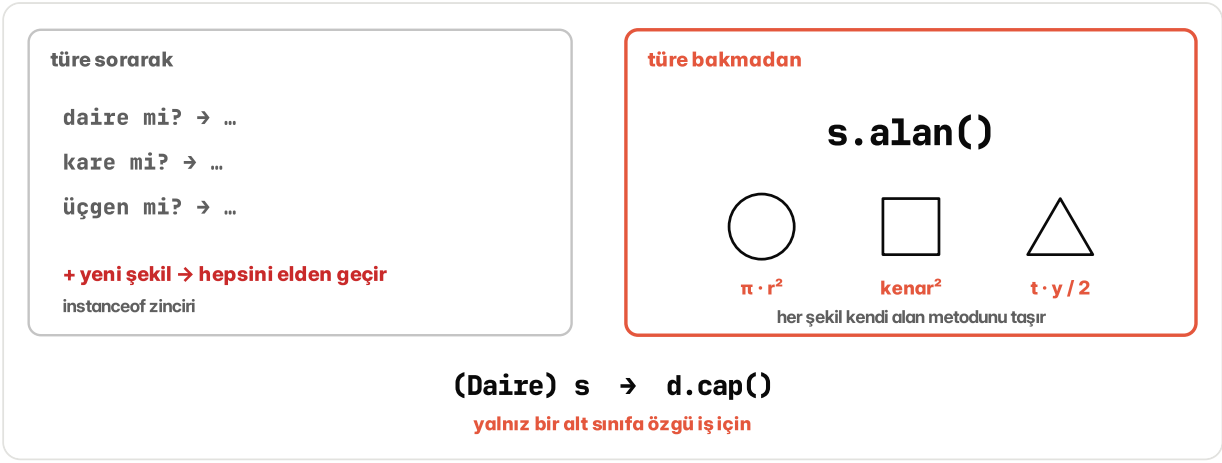
```
2.0
çapı yok
```

Kare verilince program durmaz; ekrana "çapı yok" yazar.

Bu iki adım, Java 16'dan beri tek satırda yazılabilir. instanceof'tan sonra bir değişken adı verilirse, sınama doğru çıktığında nesne o değişkene Daire tipinde yazılır. Ayrı bir dönüşüm satırına gerek kalmaz; sonuç aynıdır:

```
static void capYaz(Sekil s) {
    if (s instanceof Daire d) {
        System.out.println(d.cap());
    } else {
        System.out.println("çapı yok");
    }
}
```

Yine de instanceof'a sık başvuran kod bir uyarı işaretidir: her işi "daire mi, kare mi, üçgen mi" diye sorarak yapan kod, her yeni şekilde baştan elden geçirilir. Alan gibi şekle göre değişen bir işte türe bakılmaz; her şekil kendi alan metodunu taşır. Aşağı dönüşüm, çap gibi yalnız bir alt sınıfta olan bir iş gerektiğinde kullanılır.



Şekil 7.10 Türe sorarak yazılan instanceof zinciri her yeni şekilde elden geçer; s.alan() türe bakmaz, (Daire) s yalnız alt sınıfa özgü iş için kullanılır.

Özet

- Üst sınıf tipindeki bir değişken, alt sınıflardan birinin nesnesini tutabilir: her daire bir şekildir.
- Override edilen metodun hangi hâlinin çalışacağı program çalışırken, nesnenin gerçek tipine bakılarak seçilir; buna dinamik bağlama denir. Aşırı yüklemede ise seçimi derleyici yapar.
- Aynı çağrının nesnenin türüne göre farklı iş yapmasına çok biçimlilik denir; yeni bir tür, eski koda dokunmadan yeni bir alt sınıfla eklenir.
- Hangi metotların çağrılabileceğini derleme anındaki tip, çağrılan metodun hangi hâlinin çalışacağını çalışma anındaki tip belirler.
- Yukarı dönüşüm kendiliğinden olur ve nesneyi değiştirmez; aşağı dönüşüm (Daire) ile açıkça yazılır, nesne o tipten değilse program ClassCastException ile durur.
- instanceof nesnenin tipini sorar; Java 16'dan beri s instanceof Daire d tek satırda sınar ve çevirir. Türe göre değişen iş, türün kendi metoduna bırakılır.

Sorular

1 **Bu bölümdeki Sekil ve Kare sınıflarıyla Sekil s = new Kare(2); System.out.println(s.alan()); satırları çalışınca hangi metot çalışır?**

A Sekil'in alan() metodu; ekrana 0.0 çıkar.

B Kare'nin alan() metodu; ekrana 4.0 çıkar.

C Kod derlenmez, çünkü s'nin tipi Kare değil.

D İkisi de sırayla çalışır; ekrana 0.0 ve 4.0 çıkar.

2 **Aşağıdakilerden hangisinde çalışacak metot program çalışırken seçilir?**

A println(5) ile println("5") çağrılarında hangi println'in kullanılacağı

B Sekil s = new Daire(1); satırının derlenip derlenmeyeceği

C s.cap() çağrısının derlenip derlenmeyeceği

D Sekil tipindeki s için s.alan() çağrısında Daire'nin mi Kare'nin mi metodunun çalışacağı

3 **Bu bölümdeki Sekil ve Kare sınıflarıyla aşağıdaki satırlar ekrana ne yazar?**

```
Sekil[] sekiller = { new Kare(2), new Kare(3), new Sekil() };
double toplam = 0;
for (Sekil s : sekiller) {
    toplam += s.alan();
}
System.out.println(toplam);
```

Cevap:

4 **Daire sınıfında cap() metodu var, Sekil'de yok. Sekil s = new Daire(1); System.out.println(s.cap()); neden derlenmez?**

A s'nin derleme anındaki tipi Sekil'dir ve çağrılabilecek metotları bu tip belirler; Sekil'de cap() yoktur.

B Nesne program çalışırken Sekil'e dönüşür ve cap() metodunu kaybeder.

C cap() metodu private olmadığı için dışarıdan çağrılamaz.

D Daire nesnesi Sekil tipindeki bir değişkene hiç yazılamaz.

5 **Sekil s = new Daire(1); satırından sonra aşağıdaki satırlardan hangisi derlenmez?**

A Daire d = (Daire) s;

B Sekil t = s;

C Daire d = s;

D double a = s.alan();

6 **static void capYaz(Sekil s) { Daire d = (Daire) s; System.out.println(d.cap()); } metodu capYaz(new Kare(2)); ile çağrılınca ne olur?**

A Kod derlenir; program çalışırken ClassCastException hatasıyla durur.

B Kod derlenmez; derleyici Kare'nin Daire'ye çevrilemeyeceğini söyler.

C Ekrana 0.0 yazılır, çünkü karenin çapı yoktur.

D Kare kendiliğinden daireye dönüşür ve ekrana 2.0 yazılır.

7 **Aşağıdaki satırlar ekrana ne yazar?**

```
Sekil s = new Kare(2);
System.out.println(s instanceof Daire);
```

Cevap:

8 **Daire d = new Daire(1);**

```
Sekil s = d;
System.out.println(s == d);
satırları ekrana ne yazar?
```

A false, çünkü s ve d farklı tipte değişkenlerdir.

B true, çünkü s ve d aynı nesneyi gösterir.

C Kod derlenmez, çünkü Daire'den Sekil'e dönüşüm açıkça yazılmalıdır.

D false, çünkü yukarı dönüşüm nesnenin bir kopyasını oluşturur.

NESNE YÖNELİMLİ PROGRAMLAMA

8 Soyut sınıf ve arayüz

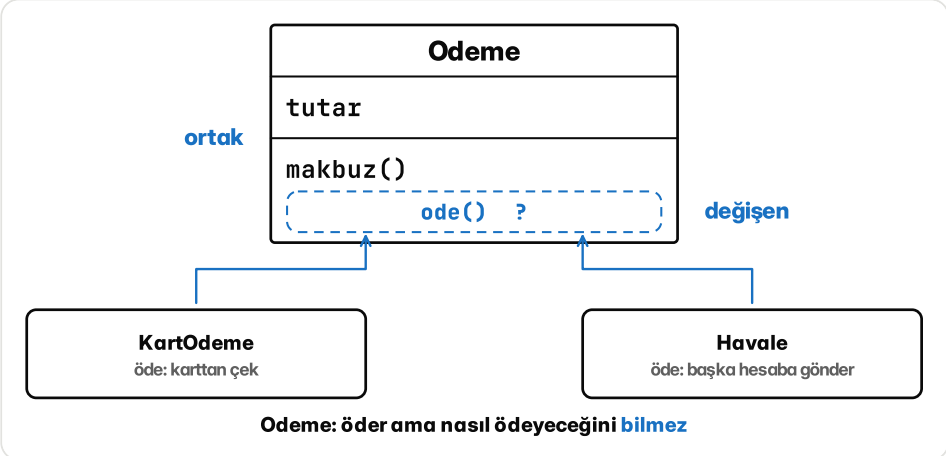
Bazı sınıflar bir işin yapılacağını bilir ama nasıl yapılacağını bilmez. Bu bölümde soyut sınıfı (abstract class) ve arayüzü (interface) göreceğiz: gövdesiz metotları, bir sınıfa birden çok yetenek vermeyi, koda henüz yazılmamış sınıflarla çalışmayı, Comparable ve Comparator ile sıralamayı ve arayüzlere gövdesiyle gelen varsayılan metotları.

Hangi ödeme?

videoda 49:02

Bir alışveriş uygulaması iki yolla ödeme alıyor: kartla ve havaleyle. Kodda ikisinin ortak yanı, yani tutar ve makbuz, Odeme adlı bir üst sınıfta durur; KartOdeme ve Havale ondan türer. Peki biri ne kart ne havale olan, yalnızca bir Odeme nesnesi oluşturursa, 250 lira nereden çekilecek?

Önce ortak olanı ayıralım. Her ödemenin bir tutarı var ve ödeme bitince makbuz hep aynı biçimde yazılır. Değişen tek şey ödemenin kendisi: kart ödemesi tutarı karttan çeker, havale başka bir hesaba gönderir. Yani Odeme sınıfı, ödemenin yapılacağını bilir ama nasıl yapılacağını bilmez.



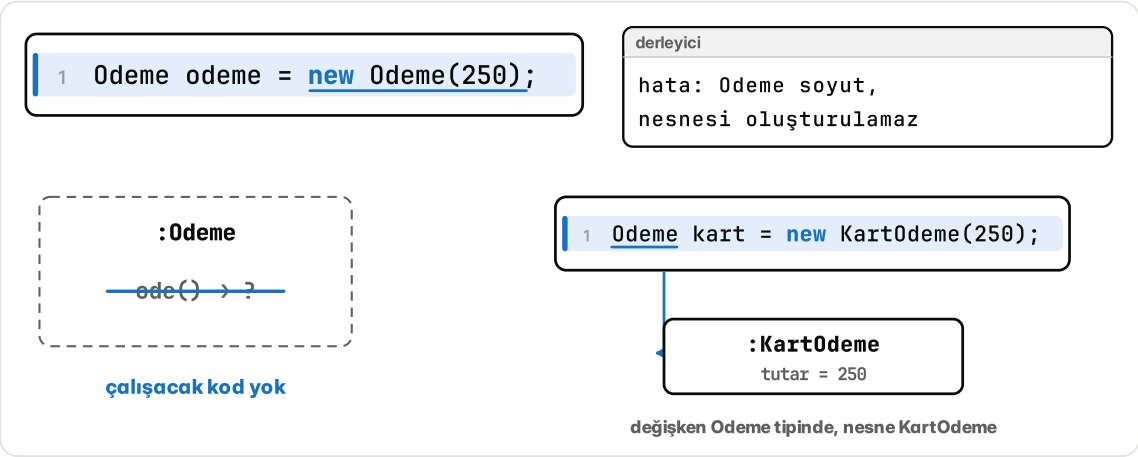
Şekil 8.1 Odeme'de tutar ve makbuz() ortak; ode() yuvası boş, çünkü KartOdeme karttan çeker, Havale başka hesaba gönderir.

Bu bilmediği işi gövdesiz bir metotla yazarız: metodun yalnız başlık satırı durur, süslü parantezlerin yerinde noktalı virgül vardır. Metodun başına **abstract** yazılır; buna **soyut metot** denir:

```
public abstract class Odeme {
    protected int tutar;
    public Odeme(int tutar) { this.tutar = tutar; }
    public abstract void ode();
    public String makbuz() {
        return "Makbuz: " + tutar + " TL";
    }
}
```

Soyut metot bir ödevdir: Odeme'den türeyen her sınıf, ode metodunu kendisi yazacak. İçinde soyut metot olan sınıf da soyut olmak zorundadır; sınıfın başına da abstract yazılır. Böyle sınıfa **soyut sınıf** denir.

Şimdi baştaki soruya dönelim: Odeme odeme = new Odeme(250); satırı derlenmez. Derleyici açıkça söyler: "Odeme is abstract; cannot be instantiated", yani Odeme soyuttur, nesnesi oluşturulamaz. Çünkü o nesnenin ode metodunun gövdesi yok; çağrılsa çalışacak tek satır kod yok. Ama Odeme tipinde değişken tanımlanabilir; o değişken bir kart ödemesini ya da bir havaleyi tutar: Odeme kart = new KartOdeme(250);



Şekil 8.2 :Odeme nesnesinin ode() metodunda çalışacak kod yok; Odeme tipindeki değişken ise bir KartOdeme nesnesini (tutar = 250) tutabilir.

Kart ödemesi ode metodunu yazar: tutarı karttan çeker. Tutarı yerleştirmeyi Odeme'nin yapıcı metoduna, super(tutar) ile bırakır. Soyut sınıfın da yapıcı metodu olur, ama yalnız bir alt sınıfın nesnesi oluşturulurken çalışır:

```
public class KartOdeme extends Odeme {
    public KartOdeme(int tutar) { super(tutar); }
    @Override
    public void ode() {
        System.out.println(tutar + " TL karttan çekildi");
    }
}
```

Havale sınıfı bu metodu yazmayı unutursa ne olur?

```
public class Havale extends Odeme {
    public Havale(int tutar) { super(tutar); }
}
```

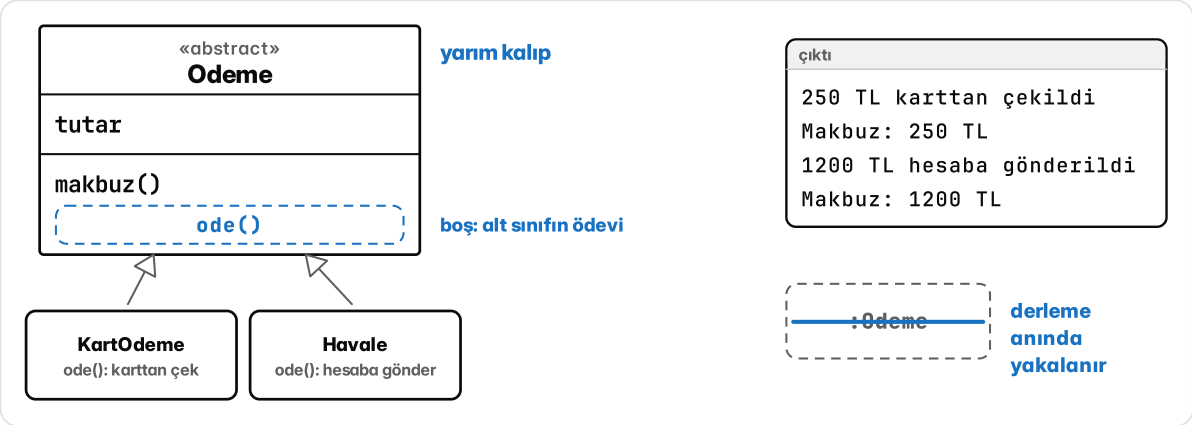
Program çalışmadan, daha derlenirken durur: "Havale is not abstract and does not override abstract method ode() in Odeme", yani Havale soyut değil, ama ode metodunu yazmamış. Alt sınıfın iki yolu var: ya metodu yazar ya da kendisi de soyut olur ve ödevi kendi alt sınıflarına bırakır.

Soyut sınıf yarım bırakılmış bir kalıptır. Tutar ve makbuz gibi ortak kısım hazırdır, alt sınıflar onu olduğu gibi devralır; türe göre değişen kısım ise boştur ve her alt sınıfa zorunlu bir ödevdir. Havale ode metodunu yazınca, 250 liralık bir kart ödemesi ile 1200 liralık bir havalenin ode ve makbuz çağrıları

şunu yazar:

```
250 TL karttan çekildi
Makbuz: 250 TL
1200 TL hesaba gönderildi
Makbuz: 1200 TL
```

Böylece ne kart ne havale olan bir ödeme hiç oluşturulamaz; ode metodunu unutan yeni bir ödeme türü de kullanıcıya ulaşmadan, derleme anında yakalanır.



Şekil 8.3 Yarım kalıp: Odeme'de tutar ve makbuz() hazır, ode() boş; KartOdeme ve Havale boşluğu doldurur, :Odeme nesnesi derleme anında yakalanır.

Tek üst sınıf

videoda 50:59

Java'da bir sınıf yalnız bir sınıftan türeyebilir; KartOdeme de zaten Odeme sınıfından türüyor. Şimdi kart ödemesine iki yetenek daha verelim: iade edilebilsin ve taksite bölünebilsin. Üst sınıfın yeri dolu; bu iki yetenek kart ödemesine nereden gelecek?

Cevap arayüz. **Arayüz** (interface) sınıfa benzer, ama class yerine interface sözcüğüyle yazılır; içinde metotların yalnız başlık satırları durur, gövdeleri yoktur:

```
public interface IadeEdilebilir {
    void iadeEt(int miktar);
}
```

IadeEdilebilir arayüzü tek bir söz verir: beni gerçekleştiren her sınıfın, miktar alan bir iadeEt metodu olacak. Ne yapılacağını söyler, nasıl yapılacağını hiç söylemez. Arayüzdeki metotlar kendiliğinden soyut ve public'tir; bunu ayrıca yazmaya gerek yok. Derlenmiş arayüze javap aracıyla bakınca metot public abstract void iadeEt(int); diye görünür.



Şekil 8.4 IadeEdilebilir arayüzünde yalnız başlık var, gövde yok; derlenmiş hâlinde public abstract sözcüklerini derleyici kendisi ekler.

Bir sınıf bu sözleşmeye **implements** sözcüğüyle katılır ve metodu kendi bildiği gibi yazar. Kart ödemesi iadeyi, parayı karta geri yükleyerek yapar. Yapıcı metot ve ode aşağıda gösterilmedi:

```
public class KartOdeme extends Odeme
        implements IadeEdilebilir {
    @Override
    public void iadeEt(int miktar) {
        System.out.println(miktar + " TL karta iade edildi");
    }
}
```

Metodu yazmayan sınıf derlenmez: "KartOdeme is not abstract and does not override abstract method iadeEt(int) in IadeEdilebilir". Arayüzün kendisinden de nesne oluşturulamaz; new IadeEdilebilir() için derleyici "IadeEdilebilir is abstract; cannot be instantiated" der, çünkü ortada çalışacak kod yoktur.

TaksitYapilabilir de ayrı bir arayüz: taksit sayısını alan bir metot ister:

```
public interface TaksitYapilabilir {
    void taksitlendir(int taksitSayisi);
}
```

Kart ödemesi iki arayüzü birden gerçekleştirir: adları, implements sözcüğünden sonra virgülle sıralanır:

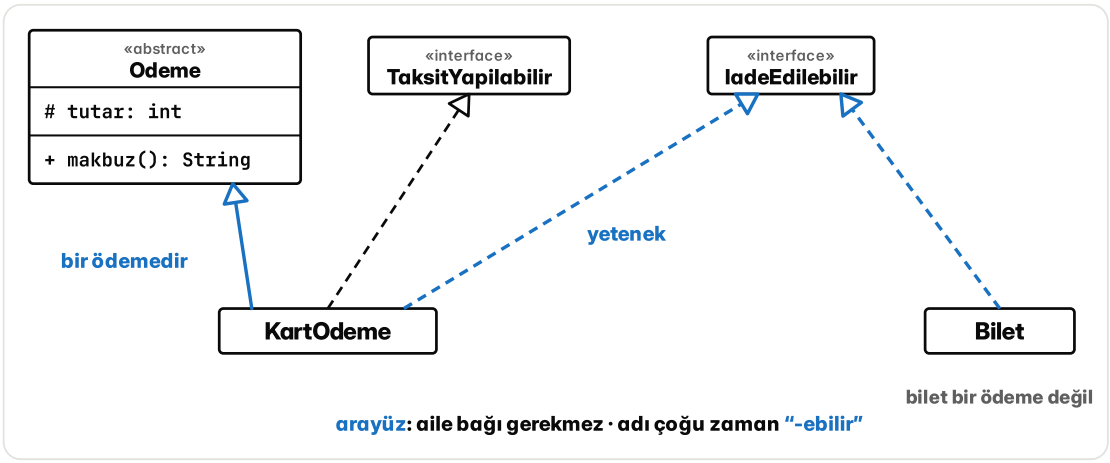
```
public class KartOdeme extends Odeme
        implements IadeEdilebilir, TaksitYapilabilir {
```

Üst sınıf bir tane olabilir, arayüz istediğimiz kadar; her arayüz sınıfa yeni bir söz ekler.



Şekil 8.5 KartOdeme'de extends Odeme yalnız bir üst sınıf alır; implements'ten sonra istenen kadar arayüz yazılır, her biri yeni bir söz ekler.

Peki ne zaman soyut sınıf, ne zaman arayüz? Cümleyi kurup bakarız: kart ödemesi bir ödemedir. Bu bir kimlik; tutar alanı ve makbuz metodu da hazır geliyor: ortak alanı ve gövdeli metotları olan bir aile için soyut sınıf. Kart ödemesi iade edilebilir: bu ise bir yetenek. Bir uçak bileti de iade edilebilir, ama bilet bir ödeme değildir; Bilet sınıfı Odeme'den türemeden IadeEdilebilir'i gerçekleştirir. Aralarında aile bağı olmayan sınıflara ortak bir yetenek vermek için arayüz; bu yüzden yetenek bildiren arayüzlerin adı çoğu zaman "-ebilir" diye biter.



Şekil 8.6 KartOdeme bir ödemedir (Odeme'den türer) ve iki arayüzü gerçekleştirir; Bilet Odeme'ye bağlı değildir ama IadeEdilebilir'i gerçekleştirir.

Kart ödemesinin tanımı artık bir cümle gibi okunur: bir ödemedir; iade edilebilir ve taksit yapılabilir. Kimliği tek bir üst sınıftan gelir, yetenekleri istediği kadar arayüzden. Üst sınıfın yeri dolu olsa da yeni bir yetenek, yeni bir arayüzle eklenir.

Yarının ödeme yöntemi

videoda 52:49

Bir alışveriş uygulamasının kasası bugün kartla ve havaleyle ödeme alıyor. Gelecek yıl yeni bir yöntem eklenecek, diyelim telefondaki dijital cüzdan; onun kodu bugün daha yazılmadı. Bir uygulama henüz yazılmamış bir ödeme yöntemini nasıl kullanabilir?

Önce kasanın gerçekten neye ihtiyacı olduğuna bakalım. Kasa, paranın karttan mı çekildiğini, başka bir hesaba mı gönderildiğini bilmek zorunda değil; tek bir şey ister: şu tutarı öde. Bu tek isteği bir arayüze yazarız. OdemeYontemi arayüzü, tutar alan bir ode metodu ister:

```
public interface OdemeYontemi {
    void ode(int tutar);
}
```

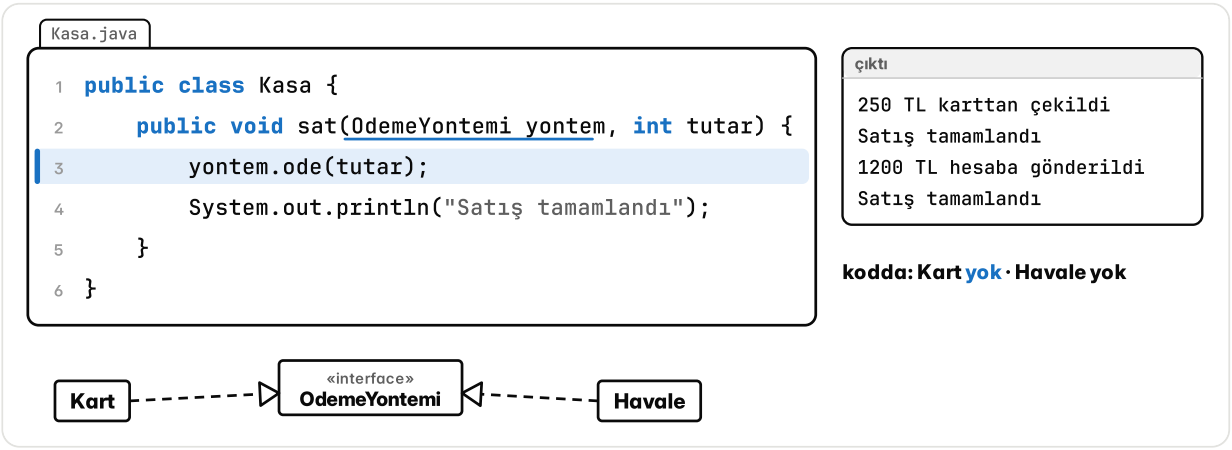
Kasanın sat metodu, parametre olarak OdemeYontemi tipinde bir nesne alır:

```
public class Kasa {
    public void sat(OdemeYontemi yontem, int tutar) {
        yontem.ode(tutar);
        System.out.println("Satış tamamlandı");
    }
}
```

Kart da Havale de bu arayüzü gerçekleştirir; ikisi de bu parametreye verilebilir. Kasa öde der; çalışan, gelen nesnenin kendi ode metodudur. 250 liralık bir kart satışı ve 1200 liralık bir havale satışı şunu yazar:

```
250 TL karttan çekildi
Satış tamamlandı
1200 TL hesaba gönderildi
Satış tamamlandı
```

Kasanın kodunda Kart ya da Havale sözcüğü hiç geçmez.

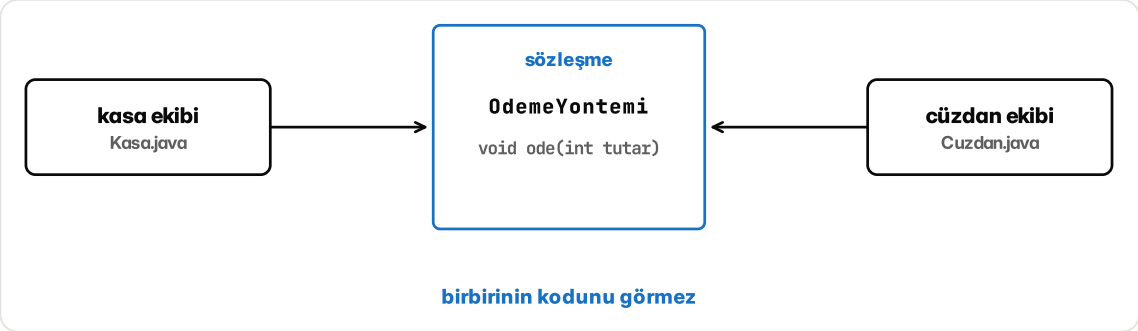


Şekil 8.7 Kasa.java'nın parametresi OdemeYontemi arayüzü; Kart ve Havale bu arayüzü gerçekleştirir, kasanın kodunda ikisinin adı geçmez.

Bir yıl sonra Cuzdan sınıfı yazılır, belki başka bir ekip tarafından; o da OdemeYontemi arayüzünü gerçekleştirir:

```
public class Cuzdan implements OdemeYontemi {
    @Override
    public void ode(int tutar) {
        System.out.println(tutar + " TL cüzdandan ödendi");
    }
}
```

Cüzdanın kartla ya da havaleyle hiçbir akrabalığı yoktur; aynı sözü vermesi yeter. Kasaya tek satır eklenmez, kasa yeniden derlenmez bile: bir yıl önce derlenmiş hâliyle, hiç tanımadığı bu sınıfla ödeme alır. 90 liralık bir satışta ekrana "90 TL cüzdandan ödendi" ve "Satış tamamlandı" çıkar. Arayüz, kasayı yazanla cüzdanı yazan arasındaki sözleşmedir; iki taraf birbirinin kodunu görmeden çalışır.

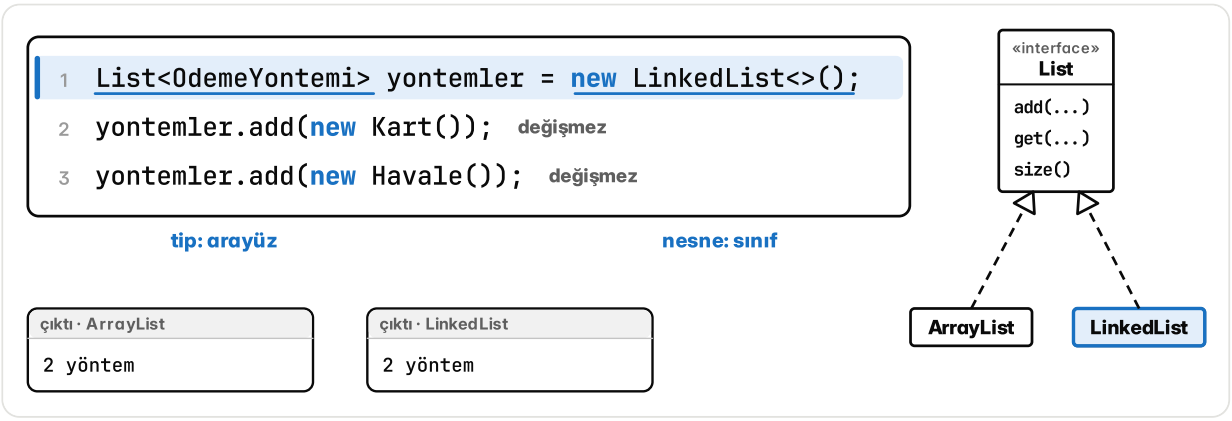


Şekil 8.8 Kasa ekibi ile cüzdan ekibi yalnız OdemeYontemi sözleşmesine bağlıdır; birbirinin kodunu görmez.

Java'nın kendi kütüphanesi de bu alışkanlıkla kullanılır. **List** bir arayüzdür: eleman ekle (add), şu sıradaki elemanı getir (get), kaç eleman var (size) gibi işleri söyler. **ArrayList** ve **LinkedList** onu gerçekleştiren iki sınıf; ikisi de listedir, ama içleri farklı çalışır. Bu yüzden değişkenin tipine arayüzü yazarız, nesneyi sınıfla oluştururuz:

```
List<OdemeYontemi> yontemler = new ArrayList<>();
yontemler.add(new Kart());
yontemler.add(new Havale());
```

Listeyi kullanan satırlar yalnız arayüzü bilir. Sınıfı değiştirmek gerekirse yalnız nesnenin oluşturulduğu yer değişir: new ArrayList<>() yerine new LinkedList<>() yazılır (bir de import satırı); listeyi kullanan satırların hiçbiri değişmez. İki program da aynı çıktıyı verir, ilk satırı "2 yöntem".



Şekil 8.9 List bir arayüzdür, ArrayList ve LinkedList onu gerçekleştirir; değişkenin tipi arayüz, nesne sınıf; sınıf değişince öteki satırlar değişmez.

Bu alışkanlığın adı **arayüze göre programlamak**. Kodu, nesnenin hangi sınıftan olduğuna göre değil, verdiği söze göre yazarız. Bugün yazılan kasa, ödeme yönteminden yalnız bu sözü beklediği için, yarın yazılacak bir cüzdanla da ödeme alır.

Hangisi önce?

videoda 54:37

Ödeme geçmişinde 4 ödeme var: Market 420 TL, Eczane 85 TL, Akaryakıt 1.500 TL ve Kitapçı 260 TL. Bunları tutara göre sıralamak istiyoruz. Bu bölümdeki Odeme sınıfı bir yer adı ve bir tutar tutuyor; ödemeler odemeler adlı bir listede. Java'nın hazır sıralama metodu Collections.sort sayıları da metinleri de sıralar; peki bizim yazdığımız Odeme sınıfında hangi nesnenin önce geleceğini nereden bilecek?

Bilemez; Collections.sort(odemeler); satırı derlenmez bile. Derleyici "no suitable method found for sort(List<Odeme>)" der: Odeme nesneleri karşılaştırılamıyor. Her sıralama yöntemi aynı soruyu tekrar tekrar sorar: bu ikisinden hangisi önce gelir? Sayılarda cevap belli, metinlerde alfabe; ama iki ödemeden hangisinin önce geleceğine sınıfı yazan karar verir. Java'da bu cevabı vermenin iki yolu var, ikisi de birer arayüz.

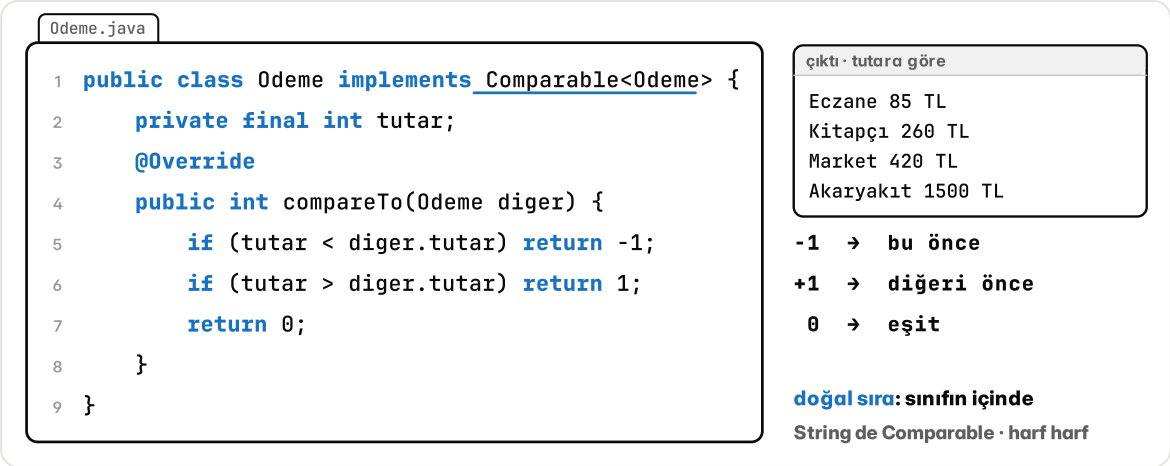
Birincisi **Comparable**, yani karşılaştırılabilir arayüzü. Odeme sınıfı bu arayüzü gerçekleştirir ve tek bir metot yazar: kendini başka bir ödemeyle karşılaştıran compareTo metodu. Yer alanı, yapıcı metot ve öteki metotlar aşağıda gösterilmedi:

```
public class Odeme implements Comparable<Odeme> {
    private final int tutar;
    @Override
    public int compareTo(Odeme diger) {
        if (tutar < diger.tutar) return -1;
        if (tutar > diger.tutar) return 1;
        return 0;
    }
}
```

Metot bir sayı döndürür: bu ödemenin tutarı küçükse -1, büyükse +1, eşitse 0. Collections.sort soruyu artık bu metoda sorar; liste 85 liradan 1.500 liraya dizilir:

```
Eczane 85 TL
Kitapçı 260 TL
Market 420 TL
Akaryakıt 1500 TL
```

Sınıfın içine yazılan bu sıraya **doğal sıra** denir. String sınıfı da bu arayüzü gerçekleştirir; onun doğal sırası harf harf karşılaştırmadır.



Şekil 8.10 compareTo -1 dönerse bu ödeme, +1 dönerse öteki önce gelir, 0 eşittir; liste tutara göre Eczane, Kitapçı, Market, Akaryakıt diye dizilir.

Ama ekstrede ödemeleri yere göre, alfabetik görmek de isteyebiliriz. Sınıfın doğal sırası bir tanedir; aynı adla ikinci bir karşılaştırma metodu yazılamaz: derleyici "method compareTo(Odeme) is already defined in class Odeme" der. Üstelik sınıf başkasının kütüphanesinden geliyorsa, içine hiç dokunamayız.

İkincisi **Comparator** arayüzü: karşılaştırmayı sınıfın dışında, ayrı bir nesne yapar. YereGore adlı küçük bir sınıf iki ödemeyi alır ve yer adlarını, metinlerin doğal sırasıyla karşılaştırır:

```
public class YereGore implements Comparator<Odeme> {
    @Override
    public int compare(Odeme a, Odeme b) {
        return a.getYer().compareTo(b.getYer());
    }
}
```

Sıralama metoduna listeyle birlikte bu nesneyi veririz: Collections.sort(odemeler, new YereGore()); liste akaryakıttan markete dizilir:

```
Akaryakıt 1500 TL
Eczane 85 TL
Kitapçı 260 TL
Market 420 TL
```

Aynı listeye istediğimiz kadar farklı sıra, sınıfa hiç dokunmadan verilir; böyle kısa karşılaştırıcılar tek satırda da yazılabilir, o yazımın adı lambda.

YereGore.java

```java
public class YereGore implements Comparator<Odeme> {
    @Override
    public int compare(Odeme a, Odeme b) {
        return a.getYer().compareTo(b.getYer());
    }
}
```

```java
Collections.sort(odemeler, new YereGore());
```

çıktı · yere göre

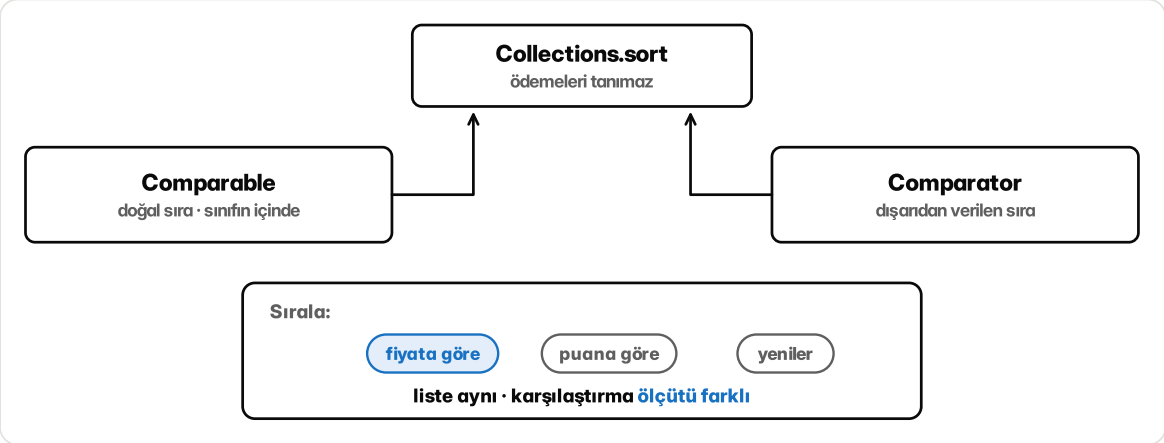
```
Akaryakıt 1500 TL
Eczane 85 TL
Kitapçı 260 TL
Market 420 TL
```

Odeme sınıfına dokunulmadı

kısa yazımı: lambda

Şekil 8.11 YereGore Comparator'ı gerçekleştirir ve yer adlarını karşılaştırır; liste Akaryakıt, Eczane, Kitapçı, Market diye dizilir, Odeme sınıfına dokunulmaz.

Sıralama metodu ödemeleri hiç tanımaz; yalnız bu iki arayüzü tanır. Comparable sınıfın kendi doğal sırasıdır; dışarıdan başka bir sıra vermek içinse Comparator kullanılır. Bir alışveriş sitesindeki fiyata göre ve puana göre sıralama seçenekleri de aynı fikir: liste aynı, karşılaştırma ölçütü farklı.

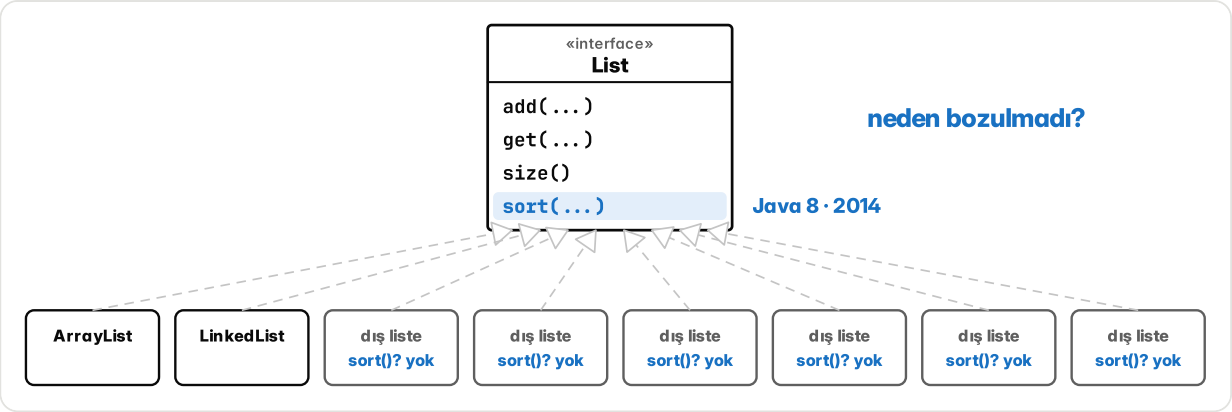


Şekil 8.12 Collections.sort ödemeleri tanımaz: Comparable sınıfın içindeki doğal sırayı, Comparator dışarıdan verilen sırayı getirir.

Bozulmayan sınıflar

videoda 56:31

Java 8, 2014'te çıktığında List arayüzüne yeni bir sıralama metodu, sort eklendi. Oysa bu arayüzü gerçekleştiren, başka şirketlerin ve yazılımcıların yazdığı pek çok liste sınıfı vardı; bu yeni metodu bilmeleri mümkün değildi. Arayüze eklenen metot, sözleşmeye eklenen yeni bir maddedir; peki bu sınıfların hiçbiri nasıl bozulmadı?



Şekil 8.13 Java 8'de List arayüzüne sort(...) eklenir; ArrayList, LinkedList ve başka şirketlerin liste sınıflarında bu metot yoktur.

Önce sorunu küçük bir örnekte görelim. OdemeYontemi arayüzünü Kart ve Havale gerçekleştiriyor; arayüze komisyon hesaplayan yeni bir soyut metot ekleyelim:

```
public interface OdemeYontemi {
    void ode(int tutar);

    int komisyon(int tutar);
}
```

İki sınıf da o anda derlenmez, çünkü ikisi de yeni metodu yazmamış: derleyici her biri için "is not abstract and does not override abstract method komisyon(int) in OdemeYontemi" der. Kendi iki sınıfımızı düzeltmek kolay; ama arayüzü başkalarının yazdığı yüzlerce sınıf gerçekleştiriyorsa, hepsi birden kırılır.

Java 8'in çözümü **varsayılan metot** (default method). Arayüzdeki metodun başına default yazılır ve metot gövdesiyle birlikte gelir: komisyon varsayılan olarak 0'dır:

```
public interface OdemeYontemi {
    void ode(int tutar);

    default int komisyon(int tutar) {
        return 0;
    }
}
```

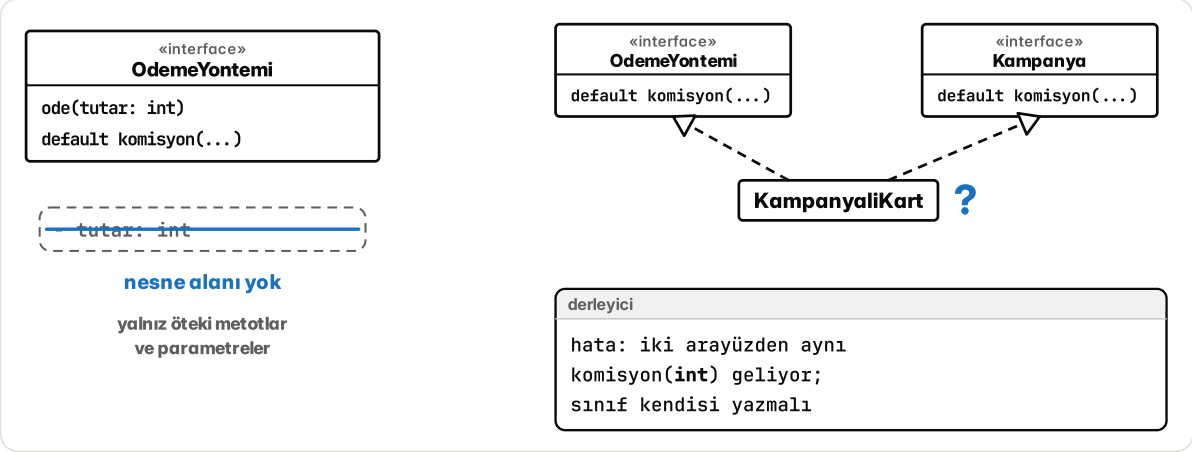
Metodu yazmayan sınıf bu gövdeyi hazır kullanır: Havale hiç değişmeden derlenir ve komisyonu 0 çıkar. Kendi hesabı olan sınıf metodu yeniden yazar: Kart, tutarın %2'sini döndürür:

```
public class Kart implements OdemeYontemi {
    @Override
    public void ode(int tutar) {
        System.out.println(tutar + " TL karttan çekildi");
    }
    @Override
    public int komisyon(int tutar) {
        return tutar * 2 / 100;
    }
}
```

250 liralık ödemede komisyonlar şöyle çıkar (250 × 2 / 100 = 5):

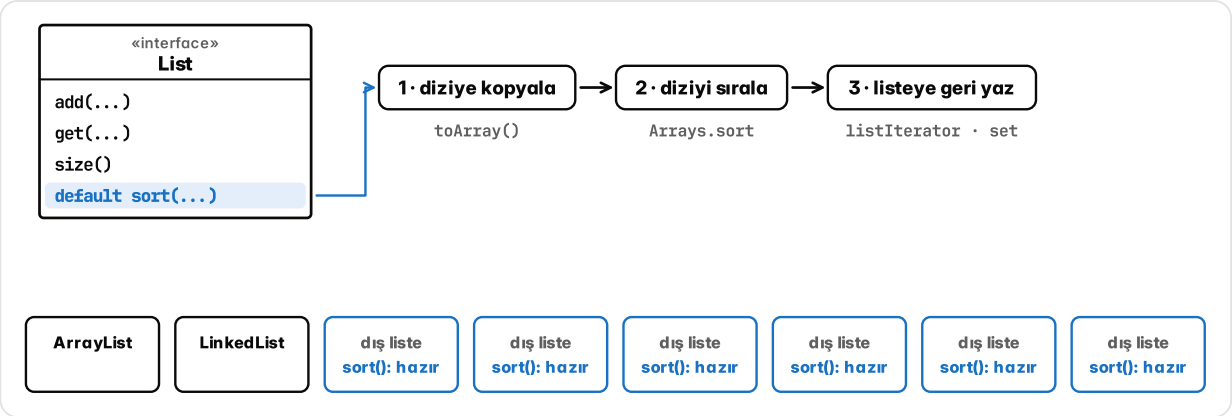
```
Kart komisyonu: 5 TL
Havale komisyonu: 0 TL
```

Varsayılan metot arayüzü soyut sınıfa çevirmez: arayüzde yine nesneye ait alan tutulamaz, bu metot yalnız arayüzün öteki metotlarıyla ve parametreleriyle çalışır. İki arayüz aynı adla birer varsayılan metot getirirse derleyici birini seçmez; sınıfın o metodu kendisinin yazmasını ister. OdemeYontemi ve Kampanya arayüzlerini birlikte gerçekleştiren KampanyaliKart için derleyici "inherits unrelated defaults for komisyon(int)" der.



Şekil 8.14 Varsayılan metodun sınırı: arayüzde nesne alanı yok; iki arayüzden aynı adlı default komisyon(...) gelirse KampanyaliKart onu kendisi yazmalı.

List arayüzündeki sort metodu da böyle, gövdesiyle birlikte eklendi. Gövde, listenin zaten var olan metotlarıyla çalışır: elemanları bir diziye kopyalar, diziyi sıralar, sonra listeye sırayla geri yazar. Eski liste sınıfları bu gövdeyi hazır buldu; tek satırları değişmeden yeni metoda kavuştu.



Şekil 8.15 List'in default sort(...) gövdesi üç adımda çalışır: diziye kopyala, diziyi sırala, listeye geri yaz; bütün liste sınıflarında hazırdır.

Özet

- Soyut metot gövdesizdir ve abstract ile yazılır; soyut metodu olan sınıf da soyuttur ve nesnesi oluşturulamaz, ama o tipte değişken tanımlanabilir.
- Soyut sınıfı genişleten sınıf soyut metotları yazmak ya da kendisi de soyut olmak zorundadır; unutulan metot derleme anında yakalanır.
- Arayüz yalnız ne yapılacağını söyler; sınıf onu implements ile gerçekleştirir. Bir sınıfın tek üst sınıfı, istediği kadar arayüzü olur: kimlik ("bir ...dır") soyut sınıftan, yetenek ("...ebilir") arayüzden gelir.
- Arayüze göre programlanan kod, sözü veren her sınıfla, sonradan yazılanlarla bile, yeniden derlenmeden çalışır; değişkenin tipine arayüz (List), nesneye sınıf (ArrayList) yazılır.
- Comparable sınıfın içine doğal sırayı, Comparator dışarıdan başka bir sırayı koyar; Collections.sort yalnız bu arayüzleri tanır.
- Varsayılan (default) metot arayüze gövdesiyle gelir; arayüze yeni metot eklenirken eski sınıflar kırılmaz.

Sorular

1 **Odeme soyut bir sınıf, KartOdeme ondan türüyor ve ode metodunu yazıyor. Aşağıdaki satırlardan hangisi derlenir?**

A Odeme o = new Odeme(250);

B Odeme o = new KartOdeme(250);

C KartOdeme k = new Odeme(250);

D Odeme o = Odeme(250);

2 **Odeme sınıfında public abstract void ode(); var. Havale extends Odeme sınıfı ode metodunu yazmayı unutmuş ve kendisi soyut değil. Ne olur?**

A Kod derlenir; havale.ode() çağrılınca hiçbir şey yapmaz.

B Kod derlenir; havale.ode() çağrılınca Odeme'deki ode çalışır.

C Kod derlenmez; Havale soyut değil ama soyut ode metodunu yazmamış.

D Kod derlenir ama program çalışırken ode çağrılınca durur.

3 **KartOdeme zaten Odeme'den türüyor. Bir uçak bileti de kart ödemesi gibi iade edilebilsin isteniyor; bilet bir ödeme değil. Bu yetenek en doğru nasıl verilir?**

A Bilet sınıfı Odeme'den türetilir.

B KartOdeme hem Odeme'den hem Bilet'ten türetilir.

C iadeEt metodu Odeme sınıfına gövdesiyle yazılır.

D IadeEdilebilir arayüzü yazılır; KartOdeme ve Bilet onu implements ile gerçekleştirir.

4 **Kasa sınıfının sat metodu OdemeYontemi tipinde bir parametre alıyor. Bir yıl sonra Cuzdan implements OdemeYontemi sınıfı yazılıyor. Kasa'nın cüzdanla ödeme alabilmesi için ne gerekir?**

A Hiçbir şey; Kasa yeniden derlenmeden de cüzdanla ödeme alır.

B sat metoduna cüzdan için yeni bir if dalı eklenir.

C Cuzdan sınıfı Kart sınıfından türetilir.

D Kasa'ya Cuzdan tipinde ikinci bir sat metodu yazılır.

5 **Bu bölümdeki Odeme sınıfının compareTo metodu tutarı küçükse -1, büyükse 1, eşitse 0 döndürüyor. Aşağıdaki satırlar ekrana ne yazar?**

```
Odeme a = new Odeme("Eczane", 85);
Odeme b = new Odeme("Market", 420);
System.out.println(b.compareTo(a));
```

Cevap: _____

6 **Odeme sınıfı başka bir şirketin kütüphanesinden geliyor ve doğal sırası tutara göre. Ödemeleri yer adına göre sıralamak için ne yapılır?**

A Odeme sınıfına ikinci bir compareTo(Odeme) metodu eklenir.

B Comparator<Odeme> arayüzünü gerçekleştiren bir sınıf yazılır ve nesnesi Collections.sort'a listeyle birlikte verilir.

C Odeme sınıfının compareTo metodu silinir.

D Collections.sort(odemeler); satırı iki kez çağrılır.

7 **OdemeYontemi arayüzünde default int komisyon(int tutar) { return 0; } var. Kart sınıfı komisyon metodunu return tutar * 2 / 100; diye yeniden yazıyor, Havale ise hiç yazmıyor. new Kart().komisyon(1000) + new Havale().komisyon(1000) kaç verir?**

Cevap: _____

8 **Yüzlerce sınıfın gerçekleştirdiği bir arayüze yeni bir metot eklemek gerekiyor. Bu sınıfların hiçbiri değişmeden derlenmeye devam etsin isteniyor. Yeni metot nasıl yazılır?**

A Gövdesiz, soyut bir metot olarak

B Arayüz soyut bir sınıfa çevrilerek

C Başına default yazılıp gövdesiyle birlikte

D Arayüzde nesneye ait yeni bir alanla birlikte

NESNE YÖNELİMLİ PROGRAMLAMA

9 Hata yönetimi

Programlar yanlış girdiyle, bulunamayan dosyayla, yetmeyen bakiyeyle karşılaşır. Bu bölümde Java'nın hatayı nasıl bildirdiğini göreceğiz: hatayı yakalamayı, çökünce yazılan dökümü okumayı, derleyicinin hangi hatalara hazırlıklı olmamızı istediğini, kendi hata sınıfımızı yazmayı ve hata çıksa da dosyanın kesin kapanmasını.

Hangisi doğru?

videoda 58:04

Bugün kaç adım attığımızı soran bir programa kullanıcı, rakam yerine yazıyla "yedi bin" giriyor. C ile yazılmış program ekrana "0 adım" yazıp yoluna devam ediyor; Java ile yazılmış olan ise o satırda duruyor ve bir hata yazıyor. Hangisi doğru davrandı?

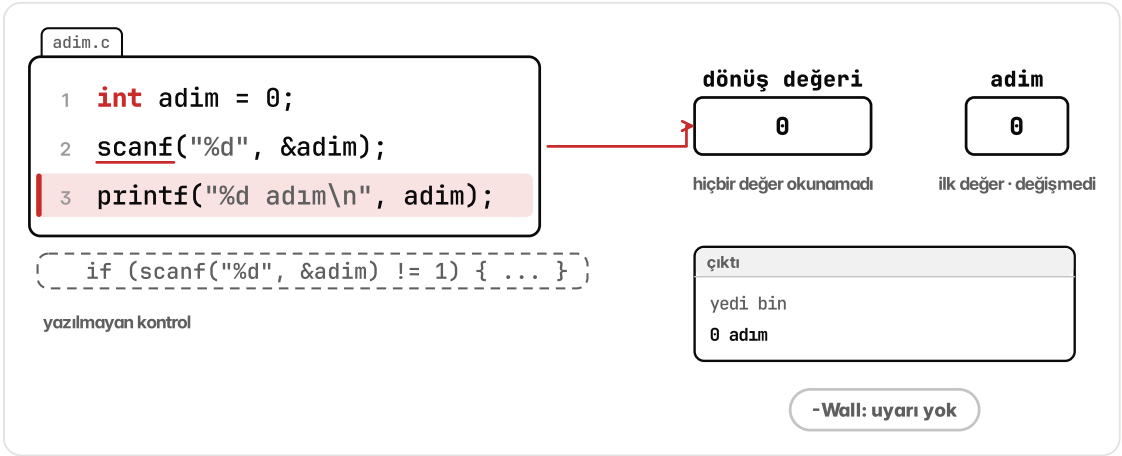
C'de bir fonksiyon hatayı dönüş değeriyle bildirir. Sayıyı okuyan scanf burada 0 döndürdü, yani hiçbir değer okuyamadığını söyledi:

```
int adim = 0;
scanf("%d", &adim);
printf("%d adım\n", adim);
```

Çıktı:

```
0 adım
```

Ama bu dönüş değerine bakan bir satır yok. adim ilk değeri olan 0'da kalıyor ve program hiçbir şey olmamış gibi sürüyor. scanf'in 1 döndürüp döndürmediğine bakan kontrolü yazmak bize kalmıştır; unutmak çok kolaydır ve derleyici çoğu zaman uyarmaz. Bu örnek -Wall seçeneğiyle de uyarısız derlendi.



Şekil 9.1 scanf 0 döndürür ama bakan satır yoktur; adim ilk değeri olan 0'da kalır ve çıktı 0 adım olur.

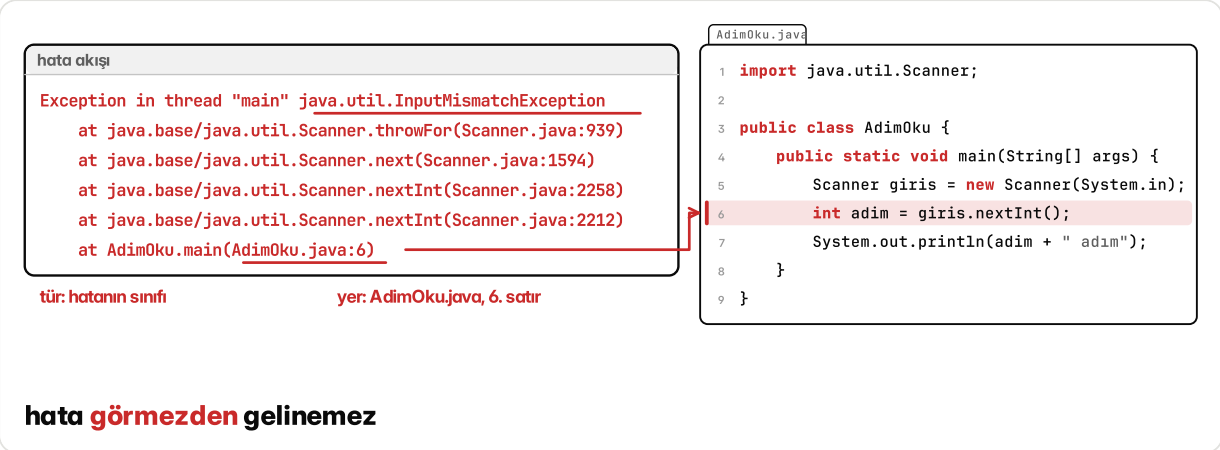
Java'da hata dönüş değeriyle değil, bir nesneyle bildirilir; bu nesneye **istisna** (exception) denir. Hatanın çıktığı yer bu nesneyi fırlatır. Klavyeden tam sayı okuyan nextInt metodu, "yedi bin" yazısını sayıya çeviremeyince bunu yaptı:

```
import java.util.Scanner;

public class AdimOku {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int adim = giris.nextInt();
        System.out.println(adim + " adım");
    }
}
```

Fırlatılan hatayı kimse yakalamazsa program durur ve ekrana hatanın türünü ve yerini yazar:

```
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at AdimOku.main(AdimOku.java:6)
```

İlk satır hatanın türüdür: InputMismatchException, girdinin beklenen kalıba uymadığını söyler. Son satır yeridir: AdimOku.java'nın 6. satırı. 7. satır hiç çalışmaz. Hata sessizce geçip gidemez.



Şekil 9.2 nextInt'in fırlattığı hata yakalanmayınca program durur; döküm hatanın türünü ve yerini, 6. satırı gösterir.

Programın durması yerine kullanıcıyı uyarmak istersek hatayı yakalarız. Hata çıkabilecek satırları **try** bloğuna yazarız. Altına **catch** gelir: parantezin içine hangi hatayı yakalayacağımızı, bloğun içine de o hata çıkınca ne yapılacağını koyarız:

```
Scanner giris = new Scanner(System.in);
try {
    int adim = giris.nextInt();
    System.out.println(adim + " adım");
} catch (InputMismatchException e) {
    System.out.println("Lütfen rakamla yazın.");
}
System.out.println("Program sürüyor.");
```

Kullanıcı rakamla 8432 yazarsa try bloğu sonuna kadar çalışır, catch hiç çalışmaz.

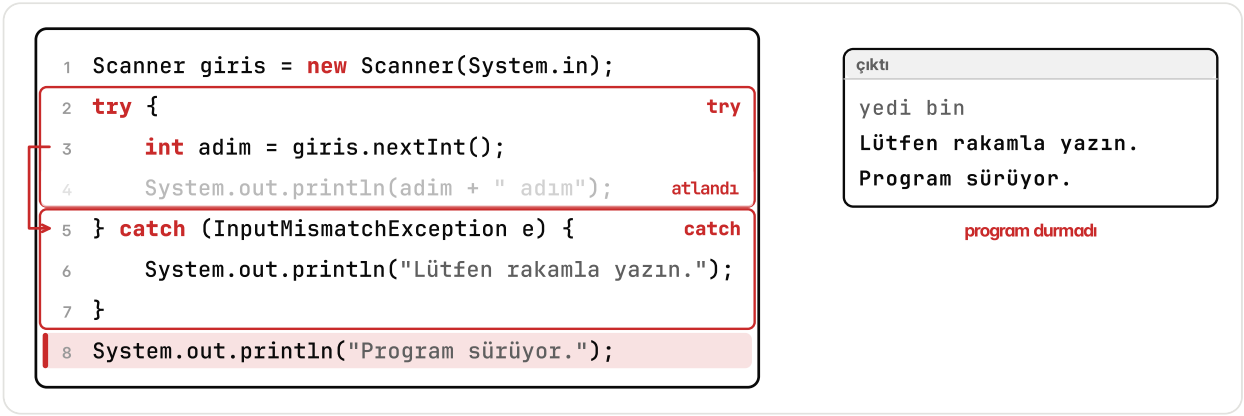
Çıktı:

```
8432 adım
Program sürüyor.
```

Yazıyla yazarsa hata nextInt'in satırında çıkar. try bloğunda o satırdan sonrası atlanır ve sıra catch bloğuna gelir. catch bitince program durmaz; blokların altından yoluna devam eder.

Çıktı:

```
Lütfen rakamla yazın.
Program sürüyor.
```



Şekil 9.3 Hata nextInt'in satırında çıkınca try'ın geri kalanı atlanır, catch çalışır ve program blokların altından sürer.

Hangisi doğru davrandı? Java. Yanlış bir değerle sessizce süren program, hatalı sonucu kimse fark etmeden kayda geçirir. Java'da hata görmezden gelinemez: ya yakalayıp ne yapılacağını söyleriz ya da program durur ve hatanın yerini gösterir.

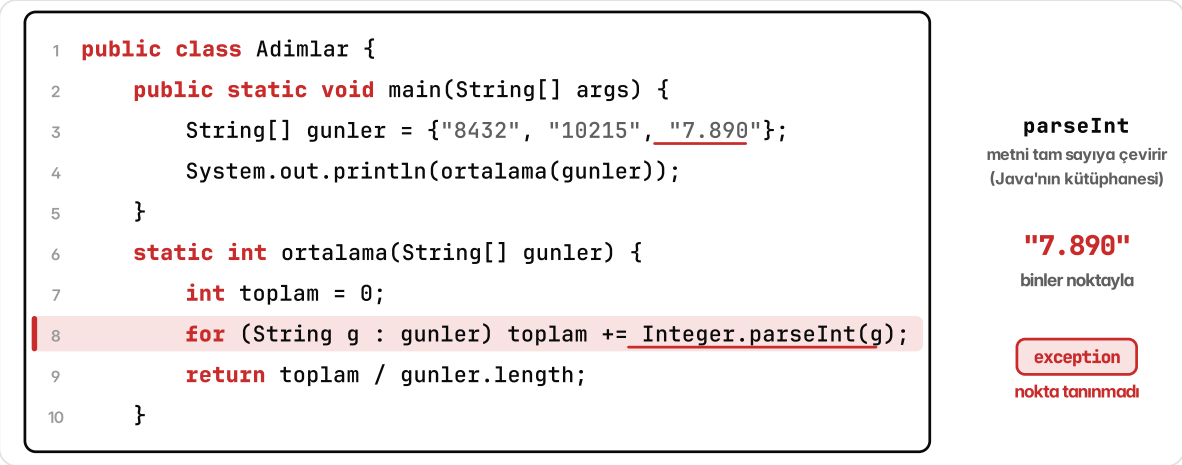
Kırk satırlık hata

videoda 59:41

Java ile yazılmış bir program çökünce ekrana bazen kırk satırlık bir hata yazar ve çoğumuz okumadan geçeriz. Oysa bu dökümün hangi satırına bakılacağı bellidir. Önce dökümün nereden çıktığını görelim.

Programımız üç günün adım ortalamasını buluyor. ortalama metodu, metin olarak duran sayıları Java'nın kendi kütüphanesindeki Integer.parseInt metoduyla tam sayıya çeviriyor. Üçüncü gün, "7.890", binlerden sonra noktayla yazılmış; parseInt noktayı tanımıyor ve hata fırlatıyor. Satır numaraları birazdan işe yarayacak: 4. satır main'deki çağrı, 8. satır çevirmenin yapıldığı satırdır.

```
public class Adimlar {
    public static void main(String[] args) {
        String[] gunler = {"8432", "10215", "7.890"};
        System.out.println(ortalama(gunler));
    }
    static int ortalama(String[] gunler) {
        int toplam = 0;
        for (String g : gunler) toplam += Integer.parseInt(g);
        return toplam / gunler.length;
    }
}
```



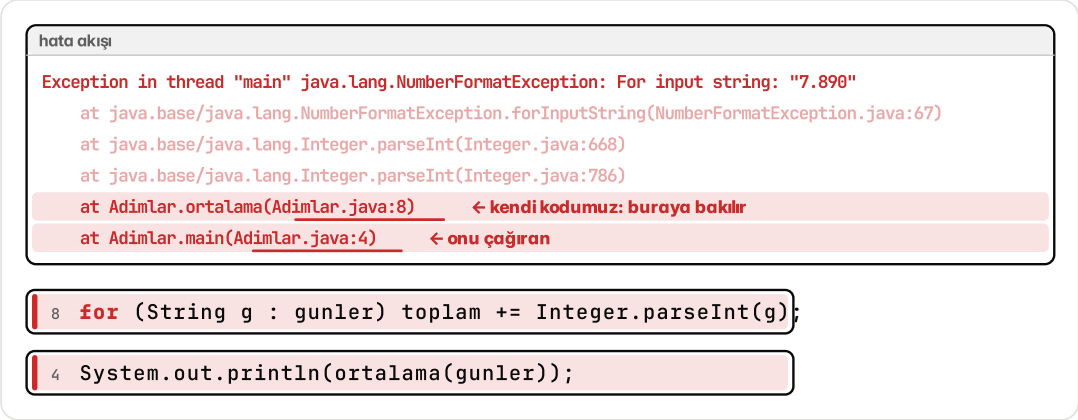
Şekil 9.4 Adimlar.java: üçüncü gün "7.890" binler noktayla yazılmış; 8. satırdaki parseInt noktayı tanımaz ve hata fırlatır.

Program, çağrılan her metot için **çağrı yığınına** bir çerçeve koyar; en son çağrılan en üsttedir. Burada yığında alttan üste main, ortalama ve parseInt durur. Hatayı fırlatan metot onu yakalamazsa orada biter ve hata, onu çağıran metoda geçer. ortalama da yakalamıyor; o da yarıda kalıyor, 9. satırdaki return hiç çalışmıyor. Hata main'e geçiyor; main de yakalamayınca program duruyor ve Java yığının o anki dökümünü ekrana yazıyor:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "7.890"
    at
java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:668)
    at java.base/java.lang.Integer.parseInt(Integer.java:786)
    at Adimlar.ortalama(Adimlar.java:8)
    at Adimlar.main(Adimlar.java:4)
```

Bu döküme **çağrı yığını dökümü** (stack trace) denir. İlk satır ne olduğunu söyler: hatanın türü, burada NumberFormatException yani sayı biçimi hatası, ve hangi metnin çevrilemediği. Altındaki her satır yığındaki bir çerçevedir: en üstte hatanın fırladığı yer, aşağı indikçe oraya kimin çağırarak geldiği.

Üstteki üç satır java.base ile başlar; Java'nın kendi kütüphanesine aittir. Hata orada değil, ona verdiğimiz değerdedir. Bu yüzden kendi sınıfımızın adının geçtiği ilk satıra ineriz: Adimlar.ortalama, 8. satır. Düzeltmeye buradan başlarız. Bir alttaki satır da onu kimin çağırdığını söyler: main, 4. satır. Kütüphane satırlarındaki numaralar Java'nın sürümüne göre değişebilir; bizim satırlarımız değişmez.



Şekil 9.5 Dökümde kütüphanenin üç satırı geçilir; bakılacak yer kendi kodumuzun ilk satırı (8) ve onu çağıran main'dir (4).

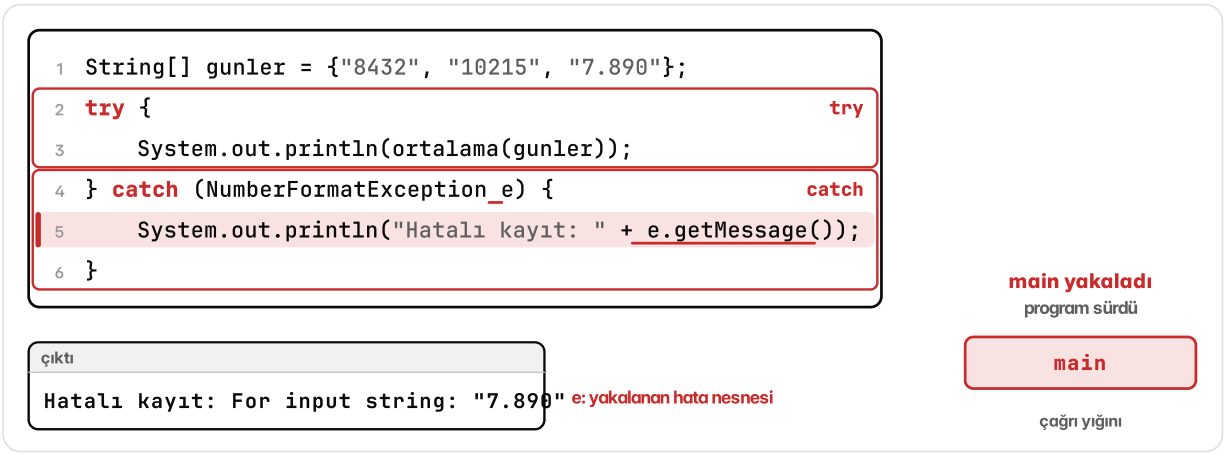
Hatayı yığındaki herhangi bir metot yakalayabilir. main'deki çağrıyı try bloğuna alırsak ortalama yine yarıda kalır, ama program durmaz:

```
String[] gunler = {"8432", "10215", "7.890"};
try {
    System.out.println(ortalama(gunler));
} catch (NumberFormatException e) {
    System.out.println("Hatalı kayıt: " + e.getMessage());
}
```

Çıktı:

```
Hatalı kayıt: For input string: "7.890"
```

catch parantezindeki e, yakalanan hata nesnesidir. getMessage metodu hatanın mesajını verir; hatalı kaydı ondan alıp ekrana yazarız.



Şekil 9.6 Çağrı main'de try bloğuna alınınca hatayı main yakalar; ekrana hatalı kayıt yazılır ve program sürer.

Gerçek projelerde araya kullandığımız kütüphanelerin satırları girer ve döküm kırk satırı bulur; ama bakılacak yer aynıdır: ilk satır ne olduğunu, kendi kodumuzun ilk satırı nerede olduğunu söyler. Dökümün ortasında "Caused by:" ile başlayan ikinci bir başlık varsa, bir kütüphane hatayı kendi hatasının içine koymuştur; asıl sebep en alttaki başlıktadır.

İki satır, iki hata

videoda 1:01:29

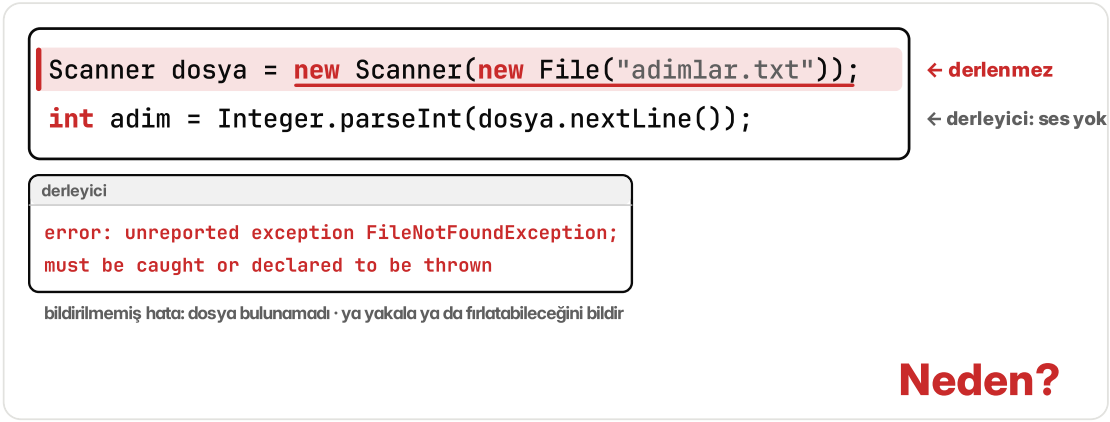
Bir dosyadan adım sayısını okuyan iki satır yazdık. Birincisi adimlar.txt dosyasını Scanner ile açıyor; File nesnesi dosyanın adını taşır. İkincisi okunan satırı sayıya çeviriyor:

```
Scanner dosya = new Scanner(new File("adimlar.txt"));
int adim = Integer.parseInt(dosya.nextLine());
```

İkisi de hata fırlatabilir: dosya olmayabilir, metin bozuk olabilir. Ama derleyici yalnız birinci satıra itiraz ediyor ve kodu derlemiyor; ikinciye hiç ses etmiyor. Derleyicinin mesajı şu:

```
error: unreported exception FileNotFoundException; must be caught or declared to be thrown
```

Türkçesi: bildirilmemiş hata, FileNotFoundException; yakalanmalı ya da fırlatılabileceği bildirilmeli. Neden yalnız biri?



Şekil 9.7 Dosyayı açan satır derlenmez: FileNotFoundException ya yakalanmalı ya da bildirilmeli; parseInt'in satırına derleyici ses etmez.

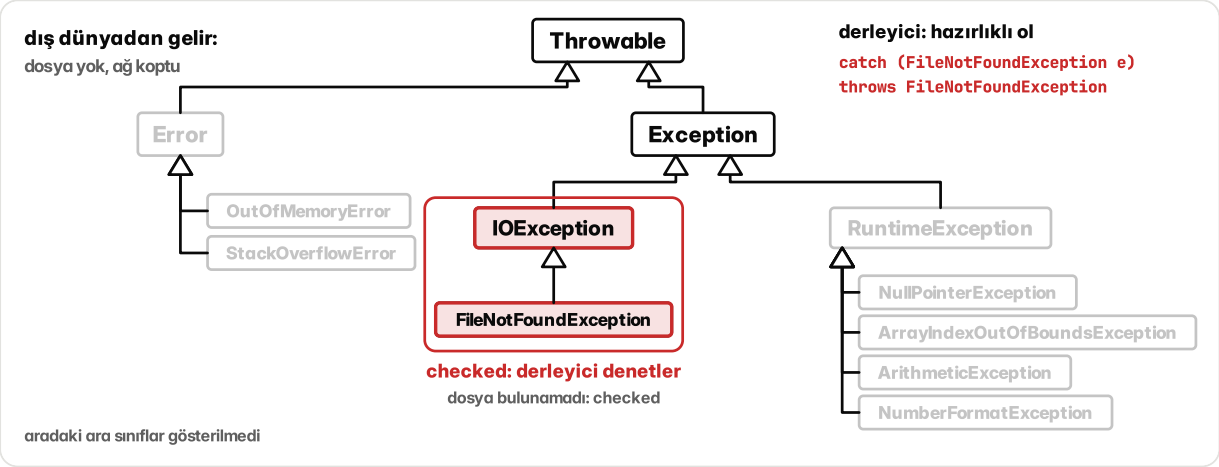
Cevap hata sınıflarının ağacında. Java'da fırlatılabilen her şey Throwable sınıfından türer ve ağaç iki kola ayrılır. Biri Error: belleğin bitmesi (OutOfMemoryError), çağrı yığınının taşması (StackOverflowError) gibi, sanal makinenin kendi ciddi sorunları; program bunlardan çoğu zaman kurtulamaz. Öteki Exception: programın ele alabileceği hatalar.

Exception'ın altında özel bir dal var: RuntimeException, yani çalışma anı hataları. NullPointerException, dizi sınırının aşılması (ArrayIndexOutOfBoundsException), sıfıra bölme (ArithmeticException) ve sayı biçimi hatası (NumberFormatException) bu daldadır; bunlar çoğu zaman kodun kendi hatasıdır. Bazılarıyla RuntimeException arasında ara sınıflar da vardır, örneğin NumberFormatException IllegalArgumentException'dan türer; diyagram onları göstermiyor. Ağaçtaki yerin bir önemi daha var: catch bir sınıfı yakalayınca altındaki bütün sınıfları da yakalar. catch (RuntimeException e) bu dört hatanın hepsini yakalar.

Exception'dan türeyip bu dalın dışında kalanlar ise dış dünyadan gelir: dosyanın bulunamaması, ağ bağlantısının kopması. Kod kusursuz olsa bile çıkabilirler. Derleyici bu yüzden bunlara hazırlıklı olmamızı ister: ya hatayı yakalarız ya da metodun başına **throws** yazıp bu hatayı fırlatabileceğini bildiririz. Bildiren main derlenir:

```
public static void main(String[] args) throws FileNotFoundException {
```

Bunlara **denetlenen hata** (checked exception) denir, çünkü derleyici denetler. IOException'ın altındaki FileNotFoundException da bunlardandır.



Şekil 9.8 Hata sınıflarının ağacı: Throwable'ın altında Error ve Exception; Exception'ın altında RuntimeException dalı ve dış dünyadan gelen IOException, FileNotFoundException.

RuntimeException dalı ve Error dalı ise **denetlenmeyen hatalardır** (unchecked exception); derleyici yakalamaya zorlamaz. Zorlasaydı nesne kullanan her satırı try bloğuna almamız gerekirdi, çünkü her nesne null olabilir. Bu hataların çaresi yakalamak değil, kodu düzeltmektir: eksik kontrolü eklemek, değeri baştan doğrulamak.

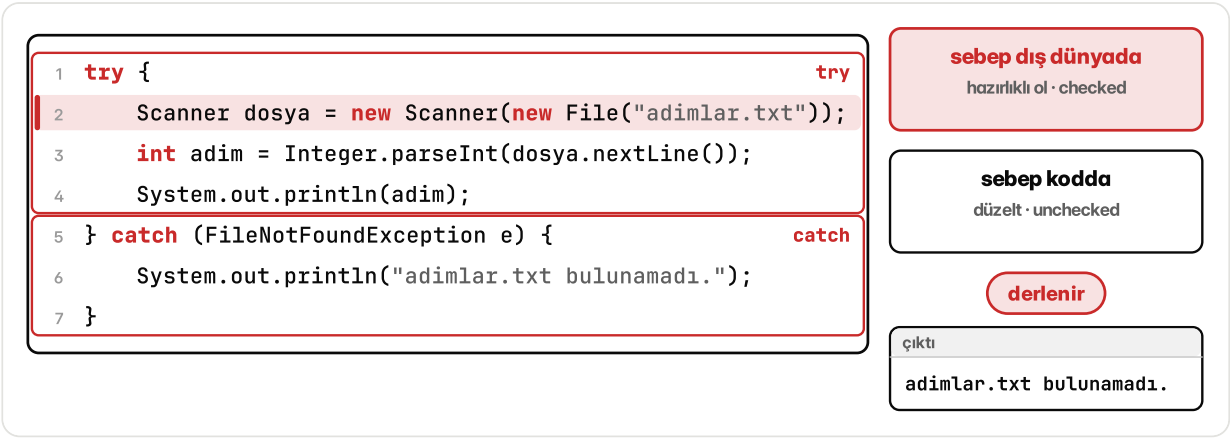
Kural kısa: sebep dış dünyadaysa derleyici hazırlıklı olmaya zorlar; kodun kendisindeyse düzeltmeyi bize bırakır. Dosyayı açan satırı try bloğuna alıp FileNotFoundException'ı yakalayınca kod derlenir:

```
try {
    Scanner dosya = new Scanner(new File("adimlar.txt"));
    int adim = Integer.parseInt(dosya.nextLine());
    System.out.println(adim);
} catch (FileNotFoundException e) {
    System.out.println("adimlar.txt bulunamadı.");
}
```

Çalışma klasöründe adimlar.txt yokken çıktı:

```
adimlar.txt bulunamadı.
```

Dosya varsa ve içinde 8432 yazıyorsa program 8432 yazar.



Şekil 9.9 Sebep dış dünyadaysa hazırlıklı olunur (checked), koddaysa düzeltilir (unchecked); try ve catch'le kod derlenir.

Bakiye yetmiyor

videoda 1:03:16

Hesapta 750 TL var, müşteri 1.000 TL çekmek istiyor. Para çekme metodu bu isteği reddetmeli; ama reddettiğini, kendisini çağıran koda nasıl bildirecek?

Birinci yol, işlemin olmadığını false döndürerek söylemek:

```
boolean paraCek(int tutar) {
    if (tutar > bakiye) {
        return false;
    }
    bakiye -= tutar;
    return true;
}
```

Bu çalışır, ama dönüş değerine bakmak çağırana kalmıştır. Bakmayı unutan kod, para çekilmiş gibi devam eder:

```
HesapFalse h = new HesapFalse();
h.paraCek(1000);
System.out.println("Paranızı alabilirsiniz.");
```

Çıktı:

```
Paranızı alabilirsiniz.
```

İkinci yol, metodun içinden ekrana "Bakiye yetersiz" yazmak. Ama aynı metodu bankamatik de mobil uygulama da kullanır; uyarının nerede, nasıl gösterileceğine metot karar veremez, çağıran verir.

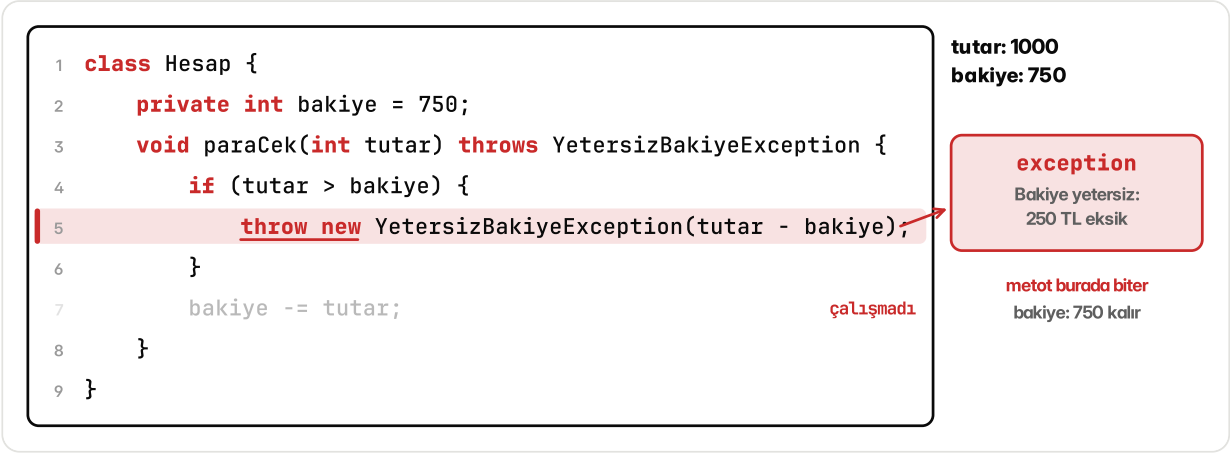
Üçüncü yol hata fırlatmak. Önce hatanın kendisini bir sınıf olarak yazarız:

```
class YetersizBakiyeException extends Exception {
    YetersizBakiyeException(int eksik) {
        super("Bakiye yetersiz: " + eksik + " TL eksik");
    }
}
```

Bu sınıf Exception'dan türer. Yapıcı metodu eksik tutarla mesajı hazırlar ve super ile üst sınıfın yapıcısına verir; getMessage sonra bu mesajı döndürür. Para çekme metodunda tutar bakiyeden büyükse **throw** ile bu hatanın yeni bir nesnesini fırlatırız:

```
class Hesap {
    private int bakiye = 750;
    void paraCek(int tutar) throws YetersizBakiyeException {
        if (tutar > bakiye) {
            throw new YetersizBakiyeException(tutar - bakiye);
        }
        bakiye -= tutar;
    }
}
```

Metot throw satırında biter; bakiyeyi azaltan satır hiç çalışmaz ve bakiye 750'de kalır.



Şekil 9.10 tutar 1000, bakiye 750: throw satırı hatayı fırlatır, metot orada biter, bakiye -= tutar çalışmaz.

Metodun başındaki throws ise bir bildiridir: bu metot YetersizBakiyeException fırlatabilir. Kısacası throw fırlatma işinin kendisidir, metodun içinde bir nesneyle yazılır; throws metodun başında, fırlatılabilecek hatanın türünü adıyla bildirir. Bu hata sınıfı Exception'dan türediği için derleyicinin denetlediği bir hatadır. paraCek'i çağırıp bu hatayı ne yakalayan ne de bildiren kod derlenmez; h.paraCek(1000); satırı için derleyici "unreported exception YetersizBakiyeException; must be caught or declared to be thrown" der.

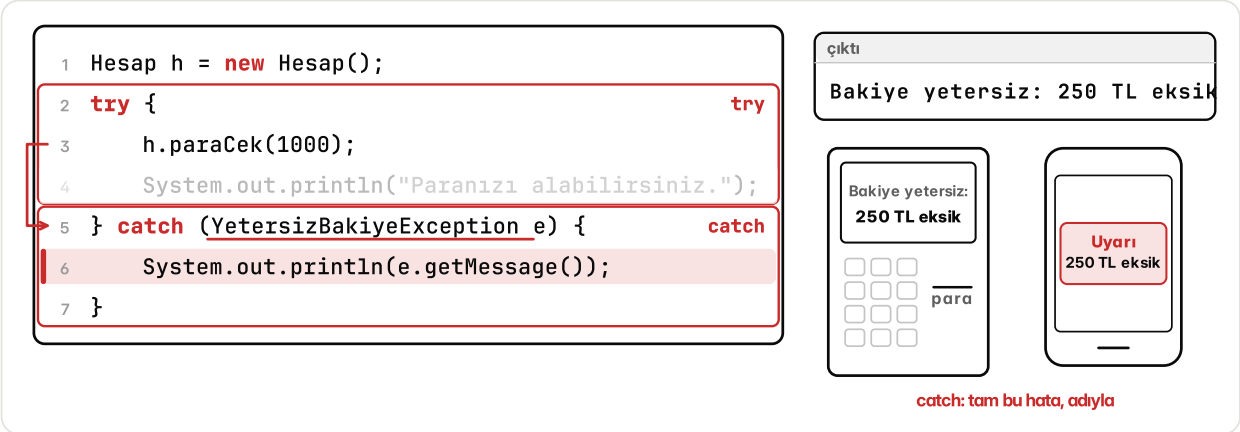
Çağıran kod para çekme satırını try bloğuna alır:

```
Hesap h = new Hesap();
try {
    h.paraCek(1000);
    System.out.println("Paranızı alabilirsiniz.");
} catch (YetersizBakiyeException e) {
    System.out.println(e.getMessage());
}
```

Çıktı:

```
Bakiye yetersiz: 250 TL eksik
```

Bakiye yetmeyince "Paranızı alabilirsiniz." satırı atlanır; catch bloğu çalışır ve hatanın mesajını yazar: 1.000 − 750 = 250 TL eksik. Bankamatik bu mesajı ekranına yazar, mobil uygulama bir uyarı penceresinde gösterir. Kendi hata sınıfımızın bir yararı daha var: catch tam bu hatayı adıyla yakalar.



Şekil 9.11 Çağıran try ve catch'le karar verir: paraCek hata fırlatınca sonraki satır atlanır, catch mesajı yazar.

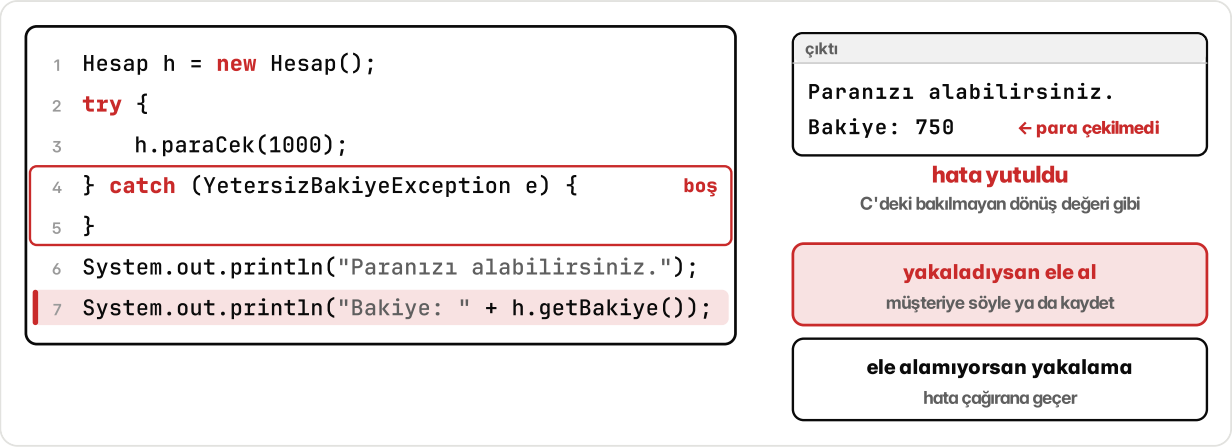
En kötü yol ise catch bloğunu boş bırakmaktır. Bu örnekteki Hesap'a bakiyeyi döndüren getBakiye metodu eklendi:

```
Hesap h = new Hesap();
try {
    h.paraCek(1000);
} catch (YetersizBakiyeException e) {
}
System.out.println("Paranızı alabilirsiniz.");
System.out.println("Bakiye: " + h.getBakiye());
```

Çıktı:

```
Paranızı alabilirsiniz.
Bakiye: 750
```

Derleyici susar, program da susar: bakiye yetmediği hâlde ekrana "Paranızı alabilirsiniz." yazılır, oysa hesaptan tek kuruş çekilmemiştir. Buna hatayı yutmak denir; C'deki bakılmayan dönüş değerinden farkı yoktur. Hatayı yakalıyorsak ele alırız: müşteriye söyleriz ya da en azından kayda geçiririz. Ele alamıyorsak hiç yakalamayız; hata, onu ele alabilecek bir çağırana kadar yukarı çıkar.



Şekil 9.12 Boş catch hatayı yutar: ekrana Paranızı alabilirsiniz yazılır ama bakiye 750'de kalmıştır.

Dosyada ne var?

videoda 1:05:08

Bir program her günün adım sayısını sayıya çevirip rapor dosyasına yazıyor. FileWriter bir dosyaya metin yazan sınıftır: write yazar, close dosyayı kapatır. main'in başında throws IOException var, çünkü dosya işlemleri bu denetlenen hatayı fırlatabilir.

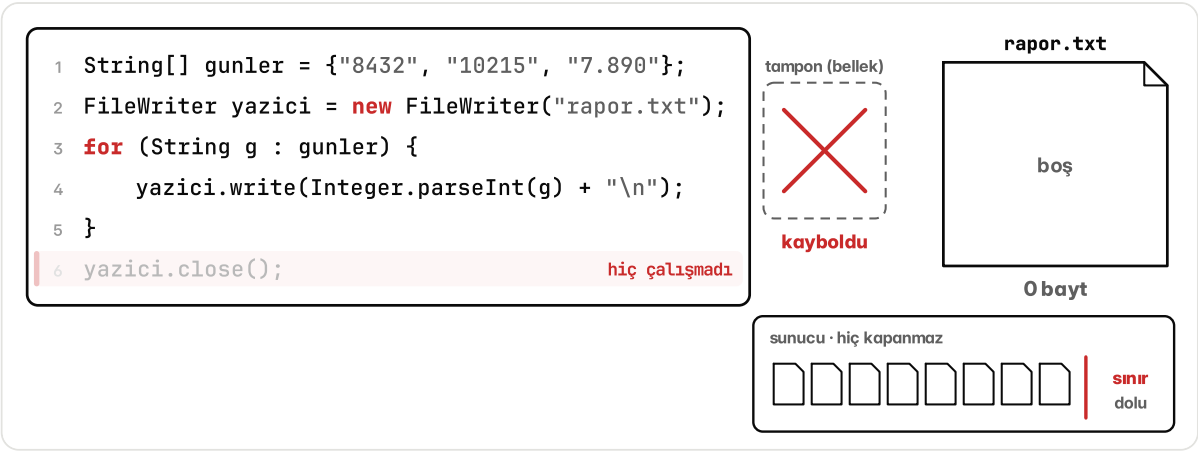
```
String[] gunler = {"8432", "10215", "7.890"};
FileWriter yazici = new FileWriter("rapor.txt");
for (String g : gunler) {
    yazici.write(Integer.parseInt(g) + "\n");
}
yazici.close();
```

Program ilk iki günü yazdı; üçüncü günde, noktayla yazılmış sayıda hata verip durdu. Dökümün ilk satırı:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "7.890"
```

Dosyada ilk iki günü görmeyi bekleriz; oysa rapor.txt bomboş, 0 bayt. Yazılanlar nereye gitti?

Açılan dosya, işletim sisteminden ödünç alınmış bir **kaynaktır**; işimiz bitince kapatıp geri veririz. Kapatmanın bir işi daha var: yazdıklarımız önce bellekteki bir **tamponda** (buffer) bekler, dosya kapanınca diske yazılır. Hata close satırından önce çıktı ve o satıra hiç gelinmedi; tampondaki iki gün diske yazılmadan kayboldu. Sunucu gibi hiç kapanmayan bir programda açık kalan dosyalar birikebilir. İşletim sistemleri bir programın aynı anda açık tutabileceği dosya sayısını sınırlar; sınır dolunca yeni dosya açılamaz.

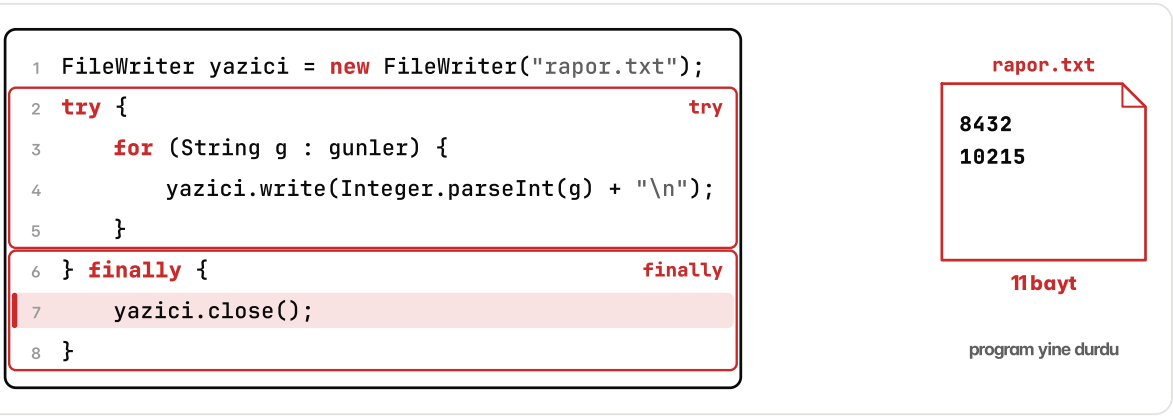


Şekil 9.13 close satırı hiç çalışmadı: tampondaki iki gün diske yazılmadan kayboldu, rapor.txt 0 bayt kaldı.

close satırı, hata çıksa da çıkmasa da çalışmalı. Bunun yeri **finally** bloğudur. try bloğundan sonra yazılır ve try nasıl biterse bitsin çalışır: normal bittiğinde de, hata yakalanmasa bile, hatta try'ın içinden return ile metottan dönülse bile.

```
FileWriter yazici = new FileWriter("rapor.txt");
try {
    for (String g : gunler) {
        yazici.write(Integer.parseInt(g) + "\n");
    }
} finally {
    yazici.close();
}
```

Program yine aynı hatayla durur, ama durmadan önce finally çalışır ve dosya kapanır. rapor.txt'de ilk iki gün vardır: 8432 ve 10215, satır sonlarıyla birlikte 11 bayt.

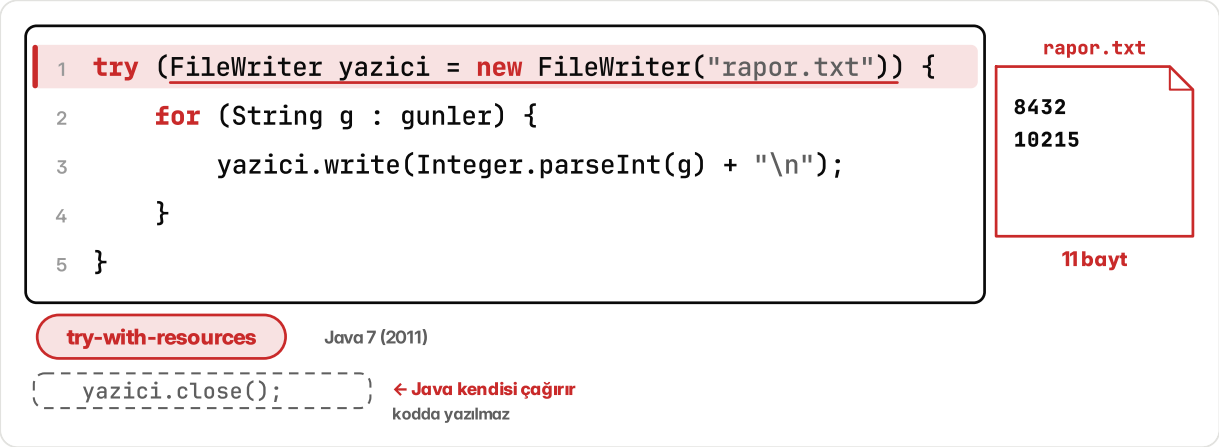


Şekil 9.14 close finally bloğunda: program yine durur ama dosya kapanır; rapor.txt'de 8432 ve 10215, 11 bayt.

finally işi görür, ama kod uzar ve kapatmayı yazmak yine bize kalır. Java 7'den (2011) beri kısa bir yol var: dosyayı try'ın yanındaki parantezin içinde açarız. Buna **try-with-resources** denir:

```
try (FileWriter yazici = new FileWriter("rapor.txt")) {
    for (String g : gunler) {
        yazici.write(Integer.parseInt(g) + "\n");
    }
}
```

Parantezde açılan kaynağı, blok nasıl biterse bitsin, Java kendisi kapatır; close satırı yazmaya gerek kalmaz. Program yine hatayla durur, ama rapor.txt'de yine ilk iki gün, 11 bayt vardır.



Şekil 9.15 try-with-resources: parantezde açılan dosyayı Java kendisi kapatır; kodda close yazılmaz, rapor.txt yine 11 bayt.

Bu yol yalnız dosyalar için değil, kapatılması gereken her kaynak için çalışır: ağ bağlantısı (Socket), veritabanı bağlantısı (Connection). Bunların hepsi, kapatılabilir olduklarını söyleyen aynı arayüzü, AutoCloseable'ı gerçekleştirir. Böylece hata ne zaman çıkarsa çıksın, yazılanlar kaybolmaz ve hiçbir kaynak açık kalmaz.

Özet

- Java'da hata bir nesnedir (istisna) ve fırlatılır; yakalanmazsa program durur ve hatanın türünü ve yerini yazar.
- try bloğunda hata çıkınca o satırdan sonrası atlanır, uygun catch çalışır ve program blokların altından sürer.
- Çağrı yığını dökümünde ilk satır ne olduğunu, kendi kodumuzun ilk satırı nerede olduğunu söyler.
- Dış dünyadan gelen hatalar denetlenir (checked): ya yakalanır ya throws ile bildirilir; RuntimeException ve Error dalının çaresi kodu düzeltmektir.
- Kendi hata sınıfı Exception'dan türer ve throw ile fırlatılır; boş catch hatayı yutar.
- finally ve try-with-resources, hata çıksa da dosyanın kapanmasını ve yazılanların diske gitmesini sağlar.

Sorular

1 **Aşağıdaki kod ekrana ne yazar?**

```
try {
    int x = Integer.parseInt("12a");
    System.out.println("A");
} catch (NumberFormatException e) {
    System.out.println("B");
}
System.out.println("C");
```

A A ve C

B B ve C

C A, B ve C

D Yalnız B

2 **Bir program şu dökümle durdu:**

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "7.890"
    at
java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)
    at java.base/java.lang.Integer.parseInt(Integer.java:668)
    at java.base/java.lang.Integer.parseInt(Integer.java:786)
    at Adimlar.ortalama(Adimlar.java:8)
    at Adimlar.main(Adimlar.java:4)
Düzeltmeye hangi satırdan başlanır?
```

A at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)

B at java.base/java.lang.Integer.parseInt(Integer.java:786)

C at Adimlar.ortalama(Adimlar.java:8)

D at Adimlar.main(Adimlar.java:4)

3 **Aynı dökümde hatanın türü hangi sınıftır? Yalnız sınıfın adını yaz.**

Cevap: _____

4 **Aşağıdaki hatalardan hangisi denetlenen (checked) bir hatadır; derleyici onu ya yakalamamızı ya da throws ile bildirmemizi ister?**

A NullPointerException

B ArithmeticException

C NumberFormatException

D FileNotFoundException

5 **catch (RuntimeException e) bloğu aşağıdaki hatalardan hangisini yakalamaz?**

A NumberFormatException

B FileNotFoundException

C ArrayIndexOutOfBoundsException

D NullPointerException

6 **throw ile throws arasındaki fark nedir?**

A throw metodun içinde bir hata nesnesini fırlatır; throws metodun başında fırlatılabilecek hatanın türünü bildirir.

B throw metodun başında hatanın türünü bildirir; throws metodun içinde hata nesnesini fırlatır.

C İkisi aynı işi görür; throws yalnız birden çok hata için kullanılır.

D throw yalnız kendi yazdığımız hata sınıflarıyla, throws yalnız Java'nın hatalarıyla kullanılır.

7 **Hesap sınıfında bakiye 750 ve paraCek, tutar bakiyeden büyükse throw new YetersizBakiyeException(tutar - bakiye); ile hata fırlatıyor. Hata sınıfının yapıcısı super("Bakiye yetersiz: " + eksik + " TL eksik"); diyor. Aşağıdaki kod ekrana ne yazar?**

```
Hesap h = new Hesap();
try {
    h.paraCek(900);
    System.out.println("Paranızı alabilirsiniz.");
} catch (YetersizBakiyeException e) {
    System.out.println(e.getMessage());
}
```

Cevap:

8 **Bir program FileWriter ile rapor.txt'ye iki satır yazdıktan sonra, close satırına gelmeden yakalanmayan bir hatayla duruyor. Dosyada ne olur?**

A İki satır da dosyadadır; write her satırı hemen diske yazar.

B Yalnız birinci satır dosyadadır.

C Dosya boştur (0 bayt); yazılanlar tamponda kalmış, dosya kapanmadığı için diske gitmemiştir.

D Dosya hiç oluşmaz.

NESNE YÖNELİMLİ PROGRAMLAMA

10 Dosyalar ve akışlar

Program kapanınca bellekteki her şey silinir; kalıcı olması gereken veri diske yazılır. Bu bölümde Java'nın dosyayla nasıl konuştuğunu göreceğiz: akışları, Türkçe harflerin neden bozulabildiğini, tamponun okumayı nasıl hızlandırdığını, satır satır okuyup yazmayı, bir nesneyi olduğu gibi kaydetmeyi ve dışarıdan gelen kayıt dosyasının neden tehlikeli olabildiğini.

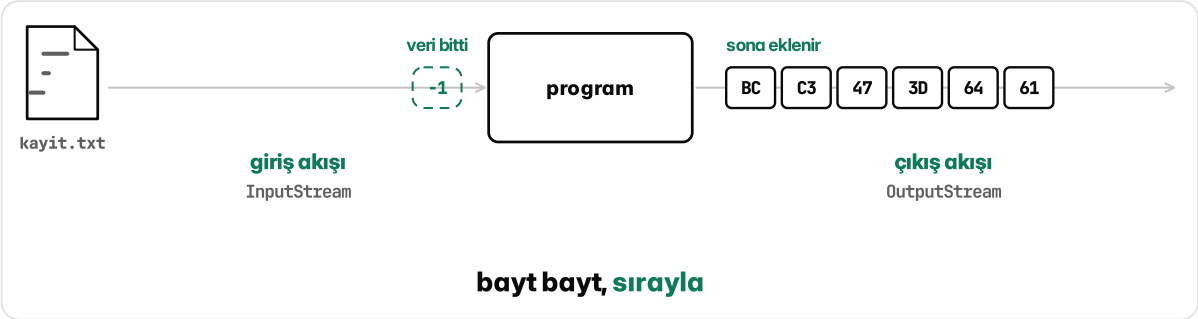
Kaldığın yer

videoda 1:06:42

Bir oyunu kapatıp ertesi gün açıyoruz; kahramanımız aynı seviyede, aynı canla bizi bekliyor. Oysa oyun kapanınca bellekteki her şey silinmişti. Oyun kaldığımız yeri nereden hatırladı?

Oyun kapanmadan önce durumunu diskteki bir dosyaya yazar, açılırken okur. Bu oyunun kayıt dosyası kayit.txt üç satırlık bir metindir: ad=Gülşen, seviye=7, can=3. Java'da dosyaya yazmanın da dosyadan okumanın da temel yolu **akıştır** (stream).

Akış, verinin bir uçtan ötekine bayt bayt, sırayla aktığı bir yoldur. Programa veri getiren yola **giriş akışı**, programdan veri götüren yola **çıkış akışı** denir; Java'daki sınıfları InputStream ve OutputStream'dir. Giriş akışından her okumada sıradaki bayt gelir; veri bitince okuma −1 döndürür. Çıkış akışına yazılan her bayt da sıranın sonuna eklenir.



Şekil 10.1 kayit.txt'nin baytları giriş akışıyla programa sırayla gelir, veri bitince okuma −1 döndürür; çıkış akışında yeni bayt sona eklenir.

Kopyalayan bir metot yazalım: giriş akışından okuduğu her baytı çıkış akışına yazsın, −1 gelince dursun. Döngünün koşulu okunan baytı b'ye koyar ve −1 olup olmadığına bakar.

```
static void kopyala(InputStream giris, OutputStream cikis)
        throws IOException {
    int b;
    while ((b = giris.read()) != -1) {
        cikis.write(b);
    }
}
```

Kayıt dosyasını okuyan bir giriş akışı (FileInputStream) açıp metoda, ekrana yazan çıkış akışıyla (System.out) birlikte veririz:

```
try (InputStream dosya = new FileInputStream("kayit.txt")) {
    kopyala(dosya, System.out);
}
```

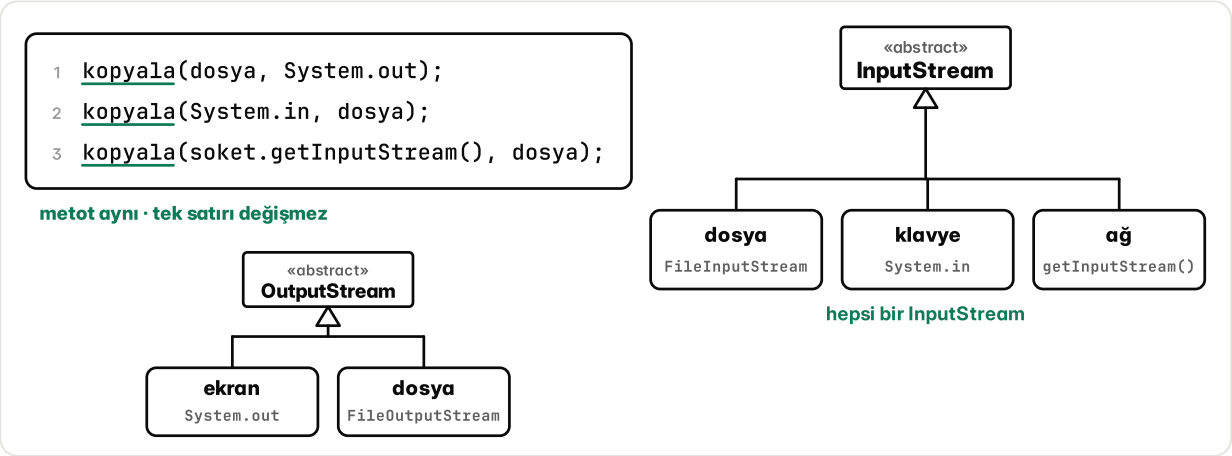
Çıktı:

```
ad=Gülşen
seviye=7
can=3
```

Metot baytın nereden geldiğini bilmez. InputStream soyut bir sınıftır; dosyadan, klavyeden ya da bir ağ bağlantısından okuyan bütün akışlar bu tiptedir. Aynı metotla klavyede yazılanı bir dosyaya, bulut sunucusundan gelen kaydı da diske kopyalarız (üç ayrı programdan birer satır; son ikisinde dosya bir çıkış akışıdır):

```
kopyala(dosya, System.out);
kopyala(System.in, dosya);
kopyala(soket.getInputStream(), dosya);
```

Metodun tek satırı bile değişmez. Çıkış tarafı da öyledir: ekrana (System.out) da dosyaya (FileOutputStream) da bir OutputStream ile yazılır. Oyun da kapanırken durumunu bir çıkış akışıyla yazar, açılırken bir giriş akışıyla okur.



Şekil 10.2 Dosya, klavye ve ağ akışlarının hepsi bir InputStream, ekran ve dosya bir OutputStream; kopyala metodu üçünde de aynı kalır.

Aynı dosyayı bu kez harf harf okuyan FileReader ile açalım:

```
try (Reader okuyucu = new FileReader("kayit.txt")) {
    int c;
    while ((c = okuyucu.read()) != -1) {
        System.out.print((char) c);
    }
}
```

Java 17'de, Türkçe Windows'ta çıktı:

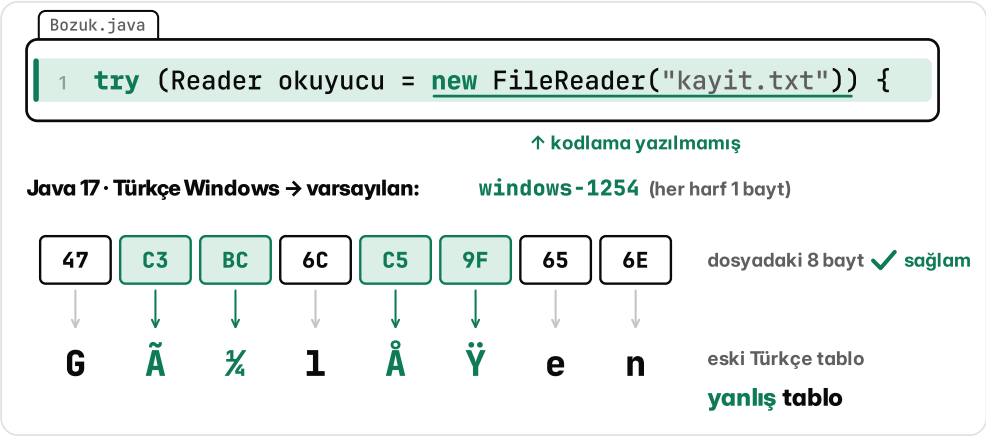
```
ad=GÃ¼lÅŸen
seviye=7
can=3
```

Ü'nün ve ş'nin yerinde ikişer tuhaf işaret var; Not defteri ise adı doğru gösteriyor. Bozuk olan dosya mı, onu okuyan program mı?

Dosyada harf yoktur, bayt vardır; harfleri baytlarla yazmanın kuralına **kodlama** (encoding) denir. Bu dosya UTF-8 ile yazılmış: İngiliz alfabesindeki harfler birer bayt, ü ve ş gibi Türkçe harfler ikişer bayt tutar. Altı harflik Gülşen 8 bayttır: G 47, ü C3 BC, l 6C, ş C5 9F, e 65, n 6E.

Java'da bu yüzden iki tür akış var. **Bayt akışları** (InputStream, OutputStream) baytları olduğu gibi taşır: resim, ses ya da kayıt dosyası. **Karakter akışları** (Reader, Writer) ise baytları harflere çevirip metinle çalışır; bunun için dosyanın hangi kodlamayla yazıldığını bilmeleri gerekir.

Dosyayı açan satırda kodlama yazılmamış. Java 17 bu durumda işletim sisteminin varsayılanını kullanır; Türkçe Windows'ta bu, her harfi tek baytla yazan eski Türkçe tablodur: windows-1254. Okuyucu ü'nün iki baytını iki ayrı harf sanır, C3 Ã, BC ¼ olur; ş'nin iki baytı da Å ve Ÿ olur. Dosya sağlam; yanlış olan, okurken seçilen tablo.



Şekil 10.3 Kodlama yazılmayınca Java 17 Türkçe Windows'ta windows-1254'ü kullanır; dosyadaki 8 bayt sağlamdır ama ü'nün baytları Ã ve ¼, ş'ninkiler Å ve Ÿ diye okunur.

Çözüm, dosyayı açan satıra kodlamayı yazmaktır (UTF_8, StandardCharsets sınıfının sabitidir). Değişen tek satır:

```
try (Reader okuyucu = new FileReader("kayit.txt", UTF_8)) {
```

Çıktı:

```
ad=Gülşen
seviye=7
can=3
```

Java 18'den beri varsayılan kodlama zaten UTF-8; ama program eski bir sürümde de çalışabilir. Kodlamayı her zaman açıkça yazarız; kaydı yazan da okuyan da aynı kodlamada anlaşmalıdır.

İki dakika

videoda 1:09:32

Oyunun 20.000.000 baytlık harita dosyasını, harita.dat'ı bayt bayt okuyan program bu bilgisayarda iki dakikadan uzun sürdü. Okuyan akışı tek bir sınıfla sarınca aynı iş bir saniyenin altına indi. Döngü aynı, dosya aynı; fark nereden geliyor? İlk programın okuma kısmı şöyle (süreyi ölçen satırlar gösterilmiyor):

```
int sayi = 0;
try (InputStream giris = new FileInputStream("harita.dat")) {
    while (giris.read() != -1) {
        sayi++;
    }
}
```

Bu çalıştırmada çıktı:

```
20000000 bayt, 149,4 sn
```

Dosya akışı her okumada işletim sistemine gider ve sıradaki tek baytı ister. Diske yalnız işletim sistemi erişir; programdan işletim sistemine geçip geri dönmek, bir baytı kopyalamaktan kat kat uzun sürer. Sayılınca istek sayısı 20.000.001 çıktı; sonuncusu −1 döndüren okuma.

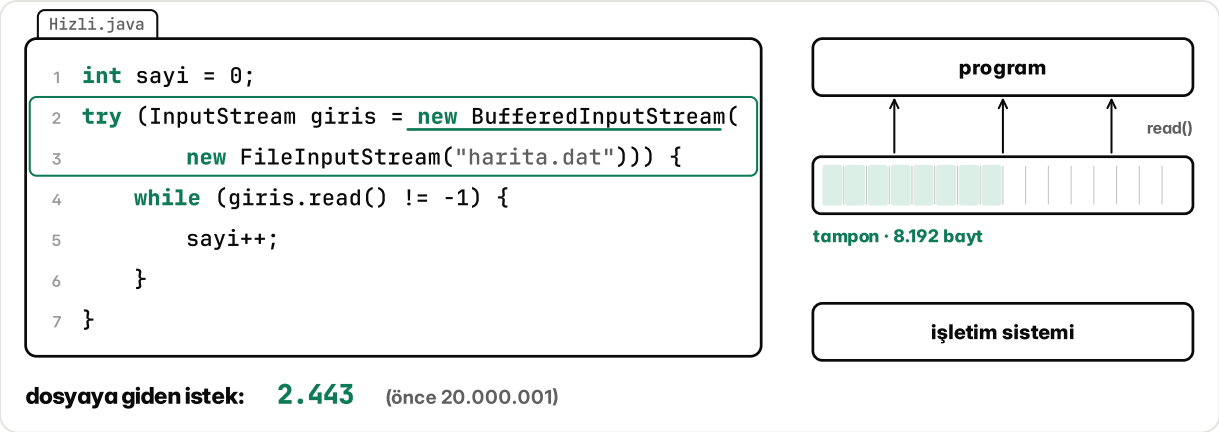
Tamponlu akış (BufferedInputStream) dosya akışını sarar:

```
int sayi = 0;
try (InputStream giris = new BufferedInputStream(
        new FileInputStream("harita.dat"))) {
    while (giris.read() != -1) {
        sayi++;
    }
}
```

Bu çalıştırmada çıktı:

```
20000000 bayt, 0,3 sn
```

Tamponlu akış işletim sisteminden tek bayt değil, 8 kilobaytlık (8.192 bayt) bir parça ister ve bellekteki tamponuna koyar. Sonraki okumalar tampondan gelir; tampon boşalınca bir sonraki parça istenir. Böylece dosyaya giden istek sayısı 20 milyondan 2.443'e iner: 20.000.000 / 8.192 ≈ 2.441,4, yani 2.442 parça ve sondaki −1.



Şekil 10.4 Program ile işletim sistemi arasına 8.192 baytlık tampon girer; dosyaya giden istek 20.000.001'den 2.443'e iner.

Java'da akışlar böyle iç içe sarılır; her katman içindekine bir yetenek ekler. Kayıt dosyasındaki metni okurken en içte dosyadan bayt okuyan akış (FileInputStream) durur; onu baytları harfe çeviren okuyucu (InputStreamReader) sarar, en dışta da tamponlu okuyucu (BufferedReader):

```
try (BufferedReader okuyucu = new BufferedReader(
        new InputStreamReader(new FileInputStream("kayit.txt"), UTF_8))) {
    System.out.println(okuyucu.readLine());
}
```

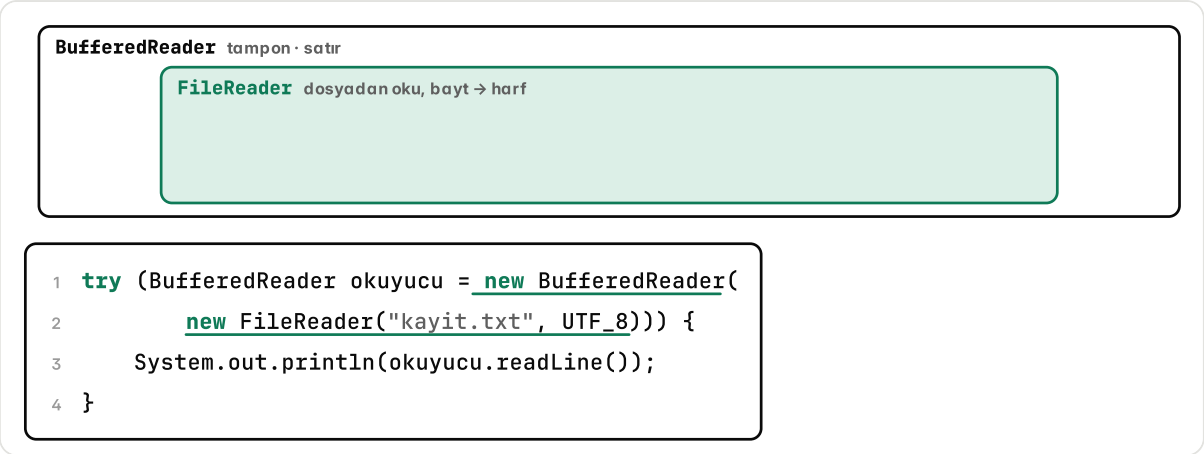
FileReader ilk iki katmanı tek sınıfta birleştirir; InputStreamReader'dan türer. Onu tamponlu okuyucuyla sarmak yeter:

```
try (BufferedReader okuyucu = new BufferedReader(
        new FileReader("kayit.txt", UTF_8))) {
    System.out.println(okuyucu.readLine());
}
```

İki yolun çıktısı da aynı:

```
ad=Gülşen
```

Tamponlu okuyucu bir yetenek daha katar: bir satırı tek seferde okumak. readLine, satır sonuna kadar olan harfleri tek bir metin olarak verir. Yazarken de aynısı geçerli: tamponlu yazıcı (BufferedWriter) yazılanları biriktirip diske parça parça gönderir.



Şekil 10.5 İç içe akışlar: FileReader dosyadan okuyup baytı harfe çevirir, dışındaki BufferedReader tampon ve satır okuma ekler.

Yazarken bir tuzak var. Oyun her bitişte skoru skorlar.txt dosyasına yeni bir satır olarak yazıyor:

```
try (BufferedWriter yazici = new BufferedWriter(
        new FileWriter("skorlar.txt", UTF_8))) {
    yazici.write("4730");
    yazici.newLine();
}
```

Dördüncü oyun bitti; dosyada dört skor (4200, 3850, 5100, 4730) olmalıydı, ama yalnız sonuncusu var: 4730. FileWriter dosyayı açarken içini boşaltır ve baştan yazar; her oyun öncekilerin skorunu sildi. Sonuna eklemek için yapıcı metoda bir değer daha veririz: true, yani sonuna ekle. Değişen tek satır:

```
        new FileWriter("skorlar.txt", UTF_8, true))) {
```

write skoru yazar, newLine satırı bitirir; eski skorlar yerinde kalır ve dosyada dört satır olur.

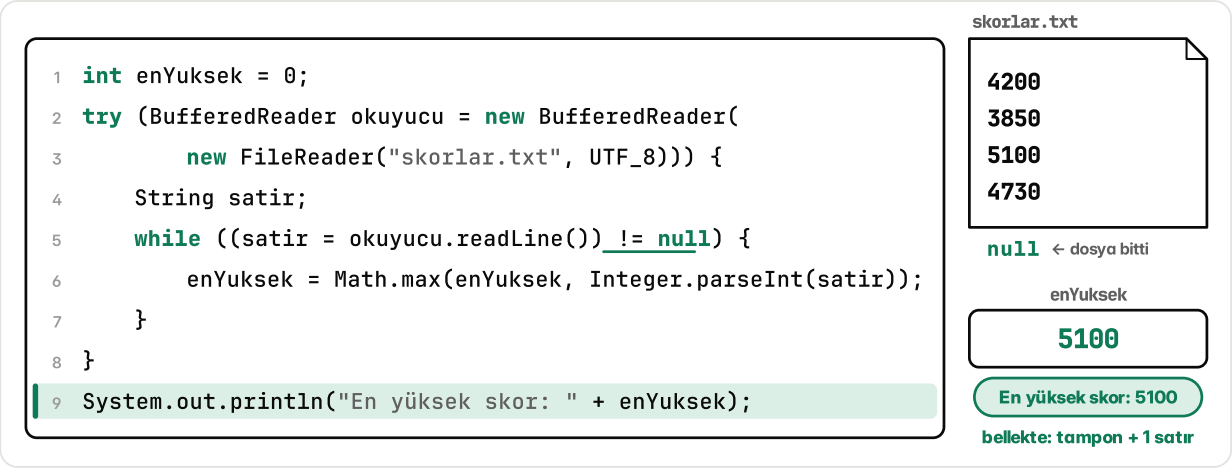
En yüksek skoru bulmak için dosyayı satır satır okuruz:

```
int enYuksek = 0;
try (BufferedReader okuyucu = new BufferedReader(
        new FileReader("skorlar.txt", UTF_8))) {
    String satir;
    while ((satir = okuyucu.readLine()) != null) {
        enYuksek = Math.max(enYuksek, Integer.parseInt(satir));
    }
}
System.out.println("En yüksek skor: " + enYuksek);
```

Çıktı:

```
En yüksek skor: 5100
```

Döngü her turda bir satır okur, onu sayıya çevirir ve o ana kadarki en büyükle karşılaştırır: enYuksek sırayla 4200, 4200, 5100, 5100 olur. Dosya bitince readLine null döndürür ve döngü durur. Bellekte her an yalnız tampon ve o satır durur; dosya milyonlarca satır olsa da program az bellekle çalışır.



Şekil 10.6 readLine döngüsü dört skoru sırayla okur, enYuksek 5100'de kalır; dosya bitince null gelir ve bellekte yalnız tampon ile bir satır durur.

Aynı iş daha kısa da yazılır. Java'nın dosya işleri için hazır bir sınıfı var, Files; sık yapılan işleri tek satıra indirir. Path.of dosyanın yolunu verir:

```
int enYuksek = 0;
for (String satir : Files.readAllLines(Path.of("skorlar.txt"))) {
    enYuksek = Math.max(enYuksek, Integer.parseInt(satir));
}
System.out.println("En yüksek skor: " + enYuksek);
```

readAllLines bütün satırları bir liste olarak verir; dosyayı açmak, tampon ve kapatma onun içindedir. Kodlama verilmezse UTF-8 kullanır. Dosyanın sonuna satır eklemek de tek satırdır; APPEND sona ekle demektir:

```
Files.write(Path.of("skorlar.txt"), List.of("4730"), APPEND);
```

Bedeli şu: bütün satırlar aynı anda belleğe gelir. Küçük dosyada kısa yol; çok büyük dosyada tamponlu okuyucuyla satır satır okuma.

Nesnenin kendisi

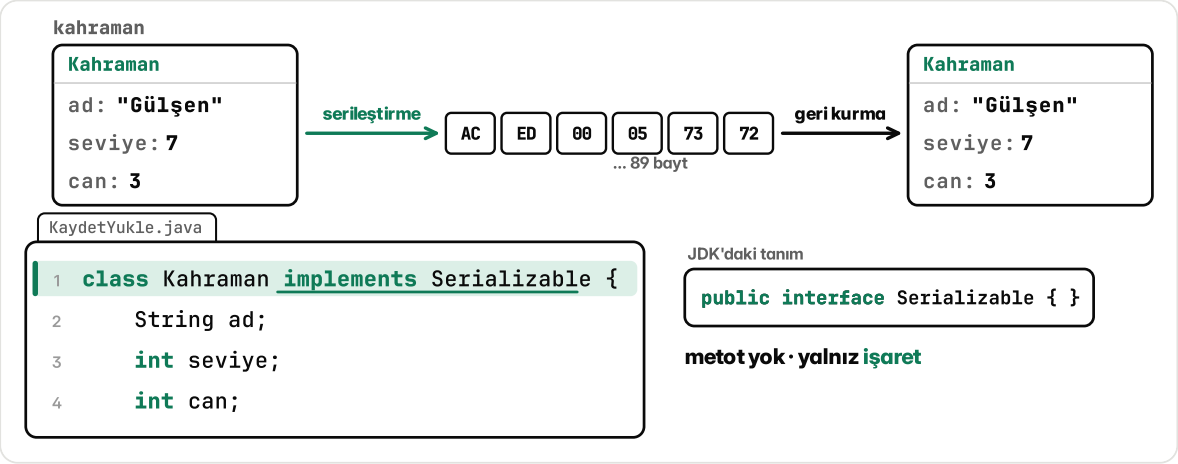
videoda 1:12:07

Kahraman nesnesinin adı, seviyesi ve canı var. Her alanı dosyaya tek tek yazmak, açılışta tek tek okuyup nesneyi yeniden kurmak uzun iş; oyuna yeni bir alan eklenince yazan kod da okuyan kod da değişir. Nesnenin kendisini olduğu gibi dosyaya yazamaz mıyız?

Yazabiliriz. Nesneyi bütün alanlarıyla bir bayt dizisine çevirmeye **serileştirme** (serialization) denir; o baytlardan nesneyi yeniden kurmak da işin tersidir. Java bunu kendisi yapar; sınıfın Serializable arayüzünü gerçekleştirmesi yeter:

```
class Kahraman implements Serializable {
    String ad;
    int seviye;
    int can;
```

Sınıfta ayrıca üç parametreli bir yapıcı ve toString vardır. Serializable'ın hiç metodu yoktur; JDK'daki tanımı gövdesi boş bir arayüzdür: public interface Serializable { }. Yalnız "bu sınıfın nesneleri kaydedilebilir" diyen bir işarettir.



Şekil 10.7 Kahraman nesnesi serileştirmeyle bayt dizisine çevrilir, baytlardan yeniden kurulur; Serializable metodu olmayan bir işaret arayüzüdür.

Kaydetmek için dosya akışını nesne yazan akışla (ObjectOutputStream) sararız ve kahramanı tek satırda yazarız. kahraman, new Kahraman("Gülşen", 7, 3) ile oluşturulmuştur:

```
try (ObjectOutputStream cikis =
        new ObjectOutputStream(new FileOutputStream("kayit.ser"))) {
    cikis.writeObject(kahraman);
}
```

Yüklemek için nesne okuyan akışı (ObjectInputStream) kullanırız. readObject'in döndürdüğü değerin tipi Object'tir; aşağı dönüşümle Kahraman'a çevririz:

```
try (ObjectInputStream giris =
        new ObjectInputStream(new FileInputStream("kayit.ser"))) {
    Kahraman yuklenen = (Kahraman) giris.readObject();
    System.out.println(yuklenen);
    System.out.println(yuklenen == kahraman);
}
```

Çıktı:

```
Gülşen · seviye 7 · can 3
false
```

Yüklenen kahramanın içi aynı, ama bellekte yeni bir nesne; == ile karşılaştırınca sonuç false.

Kayıt dosyası kayit.ser 89 bayttır. Baytların arasında sınıfın adı Kahraman, alanların adları can, seviye, ad ve Gülşen yazısı okunur. Java nesneyi geri kurabilmek için sınıfın adını ve her alanın adını, tipini (I tam sayı, L nesne), değerini yazar. Alanlarda başka nesneler varsa onlar da birlikte yazılır: ad alanındaki String de bir nesnedir. O nesnelerin sınıfı da kaydedilebilir olmalıdır.



Şekil 10.8 kayit.ser'in 89 baytı: sınıfın adı Kahraman, alanların adları ve tipleri, ardından değerler 3, 7 ve Gülşen'in baytları.

Kahramana bir müzik çalar ekleyelim; oyunda o an çalan şarkıyı, "Orman teması"nı tutuyor. MuzikCalar sınıfı Serializable değil:

```
class Kahraman implements Serializable {
    String ad;
    int seviye;
    int can;
    MuzikCalar calar = new MuzikCalar();
```

Kaydetmeye çalışınca program, bu nesnenin serileştirilemediğini söyleyen bir hatayla durur:

```
Exception in thread "main" java.io.NotSerializableException: MuzikCalar
```

Müzik çaları zaten kaydetmek istemeyiz; oyun her açılışta onu yeniden kurar. Bunun için alanın başına **transient** yazarız: bu alan kaydedilmez.

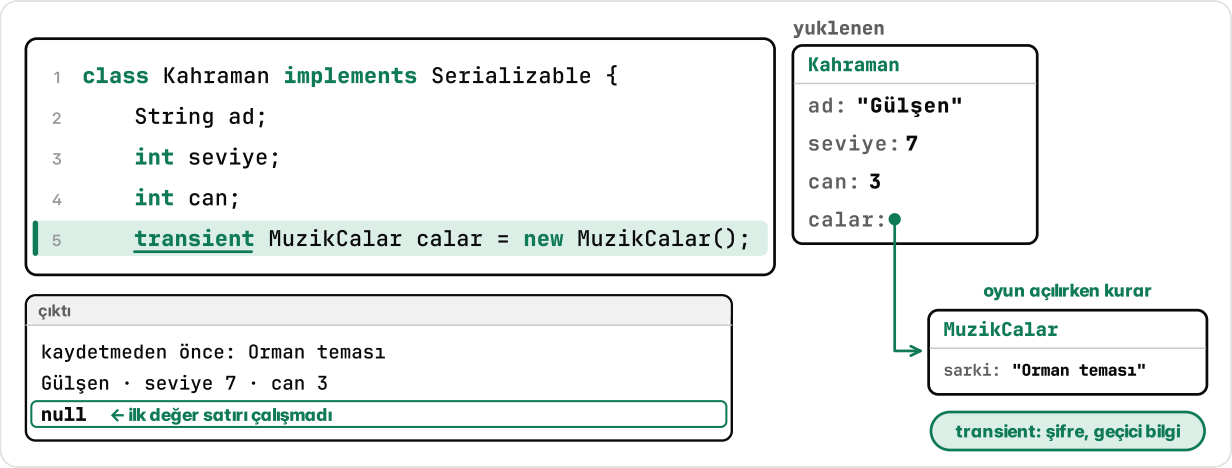
```
class Kahraman implements Serializable {
    String ad;
    int seviye;
    int can;
    transient MuzikCalar calar = new MuzikCalar();
```

Kayıt artık sorunsuz yazılır. Program kaydetmeden önce çalan şarkıyı, sonra yüklenen kahramanı ve onun calar alanını yazdırır.

Çıktı:

```
kaydetmeden önce: Orman teması
Gülşen · seviye 7 · can 3
null
```

Yüklenen kahramanda müzik çalar alanı boştur, null. Alana ilk değer veren satır bile çalışmaz, çünkü nesne yüklenirken sınıfın yapıcısı ve alanların ilk değerleri çalışmaz. Oyun açılırken müzik çaları yeniden kurar. Şifre gibi dosyaya hiç girmemesi gereken alanlar da transient ile işaretlenir.



Şekil 10.9 transient alan kaydedilmez: yüklenen kahramanda calar null'dır, ilk değer satırı çalışmaz; oyun müzik çaları açılırken kurar.

Dosya açmak

videoda 1:13:48

Bir oyun sunucusu düşünelim: oyuncuların yüklediği kayıt dosyalarını Kahraman nesnesi diye açıyor. Bir gün açtığı bir dosya, sunucuda başkasının yazdığı bir kodu çalıştırıyor. Bir dosyayı açmak nasıl olur da kod çalıştırır?

Serileştirilmiş baytlarda sınıfın adı yazılıdır; nesne okuyan akış, hangi sınıftan nesne kuracağını dosyadan öğrenir. Bazı sınıflar okunurken çalışan kendi kodunu taşır: sınıfta private void readObject metodu varsa akış, nesneyi kurarken bu metodu çağırır. Burada sahte bir sınıfı biz yazdık; okunurken ekrana bir satır yazdırıyor:

```
class Sahte implements Serializable {
    private void readObject(ObjectInputStream giris)
            throws IOException, ClassNotFoundException {
        giris.defaultReadObject();
        System.out.println("Bu satırı, dosyayı hazırlayan yazdırdı.");
    }
}
```

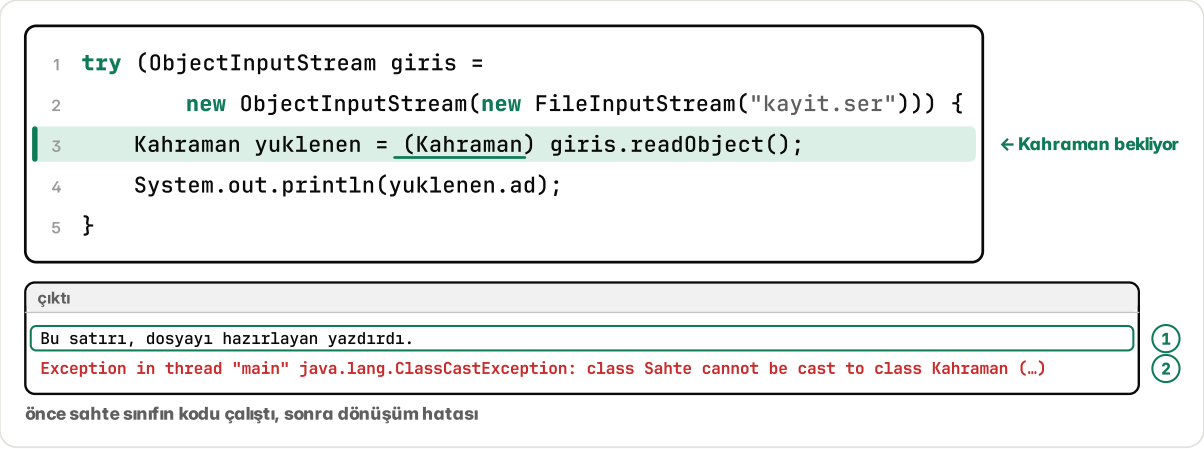
kayit.ser'e bu sınıftan bir nesne yazılmış; dosyanın 26 baytı arasında sınıfın adı, Sahte okunur. Sunucu dosyayı kahraman kaydı sanıp açar:

```
try (ObjectInputStream giris =
        new ObjectInputStream(new FileInputStream("kayit.ser"))) {
    Kahraman yuklenen = (Kahraman) giris.readObject();
    System.out.println(yuklenen.ad);
}
```

Çıktı:

```
Bu satırı, dosyayı hazırlayan yazdırdı.
Exception in thread "main" java.lang.ClassCastException: class Sahte cannot be cast to class
Kahraman (Sahte and Kahraman are in unnamed module of loader 'app')
```

Sunucu Kahraman bekliyordu ve dönüşüm hatası (ClassCastException) verdi; ama ancak sahte sınıfın kodu çalıştıktan sonra.



Şekil 10.10 Önce sahte sınıfın satırı yazdırılır, sonra dönüşüm hatası gelir: kod, nesne Kahraman'a çevrilmeden çalışmıştır.

Gerçek bir saldırıda saldırgan kendi sınıfını sunucuya sokamaz; sunucuda zaten bulunan kütüphanelerin sınıflarını kullanır. Tek başına zararsız sınıflar, okunurken birbirini çağıracak biçimde dizilir; zincirin sonunda bir komut çalışır. 2015'te Java dünyasında çok yaygın bir kütüphanede, Apache Commons Collections'ta böyle bir zincir gösterildi; dışarıdan gelen veriyi bu yolla açan WebLogic, WebSphere, JBoss ve Jenkins gibi büyük kurumsal sunucu yazılımları etkilendi. Java'nın kendi belgesi de ObjectInputStream için açıkça uyarır: güvenilmeyen veriden nesne kurmak doğası gereği tehlikelidir ("Deserialization of untrusted data is inherently dangerous and should be avoided.").

Bugün uygulamalar veriyi çoğunlukla **JSON** denen metin biçimiyle saklar ve alıp verir. JSON alanları adlarıyla ve değerleriyle düz metin olarak yazar:

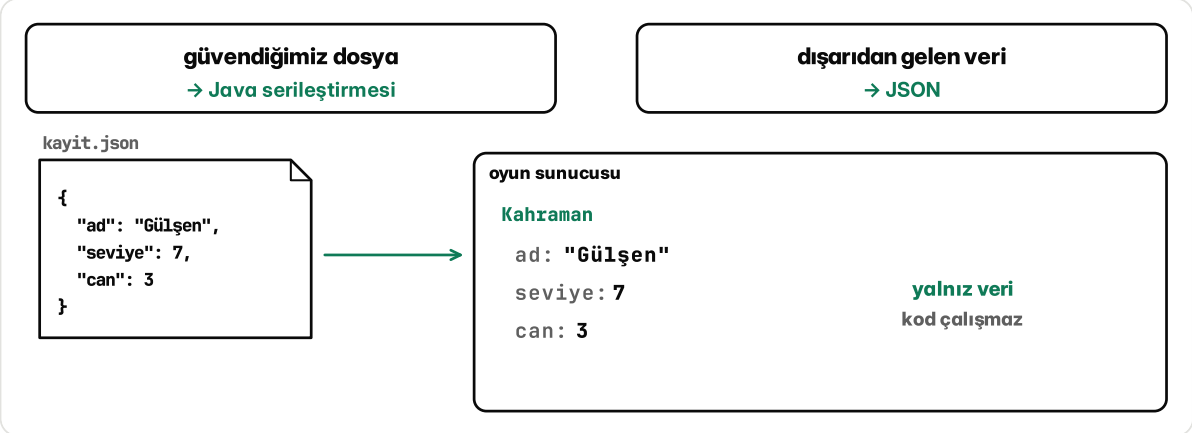
```
{
  "ad": "Gülşen",
  "seviye": 7,
  "can": 3
}
```

Dosyayı açan herkes okur, her programlama dili de okur. İçinde sınıf adı da kod da yoktur; veriyi hangi sınıfa çevireceğine dosya değil, okuyan program karar verir. Java'nın kendi kütüphanesinde JSON okuyan bir sınıf yok; bunun için Jackson ve Gson gibi yaygın açık kaynak kütüphaneler kullanılır.



Şekil 10.11 JSON kaydı düz metindir: alan adları ve değerler var, sınıf adı ve kod yok; Java, Python, JavaScript ve telefon uygulamaları okur.

Kısacası Java'nın kendi serileştirmesi yalnız güvendiğimiz dosyalarda kullanılır. Dışarıdan gelen veri, JSON gibi yalnız veri taşıyan bir biçimle alınır. Oyun sunucusu da kayıt dosyasını böyle açınca dosyadan yalnız ad, seviye ve can okur; çalışacak bir kod yoktur.



Şekil 10.12 Güvendiğimiz dosyada Java serileştirmesi, dışarıdan gelen veride JSON; sunucu JSON'dan yalnız ad, seviye ve can okur.

Özet

- Akış verinin bayt bayt, sırayla aktığı yoldur; InputStream ve OutputStream soyut sınıflardır ve dosya, klavye, ağ akışları aynı tiptedir.
- Dosyada harf değil bayt vardır; karakter akışları kodlamayı bilmelidir, bu yüzden kodlama (UTF-8) her zaman açıkça yazılır.
- Tamponlu akış işletim sisteminden 8 KB'lık parçalar ister; akışlar iç içe sarılır ve BufferedReader satır satır okur.
- FileWriter dosyayı baştan yazar, sona eklemek için true verilir; Files kısa yoldur ama bütün satırları belleğe alır.
- Serializable sınıfın nesnesi ObjectOutputStream ile tek satırda kaydedilir; transient alan kaydedilmez ve yüklenince null olur.
- Serileştirilmiş dosya hangi sınıfın kurulacağını kendisi söyler; dışarıdan gelen veri JSON gibi yalnız veri taşıyan bir biçimle alınır.

Sorular

1 **static void kopyala(InputStream giris, OutputStream cikis) metodu bir dosyayı ekrana kopyalamak için yazıldı. kopyala(System.in, dosya); çağrısı neden de çalışır?**

A System.in de bir InputStream'dir; metot baytın nereden geldiğini bilmez.

B Derleyici System.in'i kendiliğinden bir dosyaya çevirir.

C kopyala metodu klavye için ayrıca aşırı yüklenmiştir.

D System.in bir OutputStream olduğu için ikinci parametreye uyar.

2 **"Gülşen" adı UTF-8 ile yazılınca kaç bayt tutar?**

Cevap:

3 **UTF-8 ile yazılmış kayit.txt, Java 17'de Türkçe Windows'ta new FileReader("kayit.txt") ile okununca ad=Gülşen satırı ad=GÃ¼lÅŸen diye çıkıyor. Sebebi nedir?**

A Dosya diske yazılırken bozulmuştur.

B FileReader yalnız İngiliz alfabesindeki harfleri okuyabilir.

C Kodlama yazılmadığı için okuyucu varsayılan windows-1254 tablosunu kullanır ve Türkçe harflerin iki baytını iki ayrı harf sanır.

D System.out.print Türkçe harfleri ekrana yazamaz.

4 **20.000.000 baytlık bir dosya, tamponu 8.192 bayt olan BufferedInputStream ile sonuna kadar okunuyor. Tampon her seferinde işletim sisteminden bir parça istiyor; veri bitince de bir istek daha −1 döndürüyor. İşletim sistemine toplam kaç istek gider?**

Cevap:

5 **skorlar.txt'de 4200, 3850 ve 5100 satırları var. Aşağıdaki kod çalışınca dosyada ne kalır?**

```
try (BufferedWriter yazici = new BufferedWriter(
        new FileWriter("skorlar.txt", UTF_8))) {
    yazici.write("4730");
    yazici.newLine();
}
```

A 4200, 3850, 5100 ve 4730

B Yalnız 4730

C 4730, 4200, 3850 ve 5100

D Yalnız 4200, 3850 ve 5100; dosya değişmez

6 **Kahraman Serializable'dır. kahraman nesnesi kayit.ser'e writeObject ile yazılıyor, sonra aynı dosyadan Kahraman yuklenen = (Kahraman) giris.readObject(); ile okunuyor. System.out.println(yuklenen == kahraman); ekrana ne yazar?**

Cevap:

7 **Kahraman sınıfındaki transient MuzikCalar calar = new MuzikCalar(); alanı için hangisi doğrudur?**

A Alan kaydedilir, ama yüklenirken yeniden new MuzikCalar() ile kurulur.

B MuzikCalar Serializable olmadığı için kayıt yine NotSerializableException ile durur.

C Alan kaydedilmez; yüklenen kahramanda calar null olur, ilk değer satırı çalışmaz.

D transient yalnız sayı tipindeki alanlarda kullanılabilir.

8 **Dışarıdan gelen bir dosyayı ObjectInputStream ile açmak neden tehlikelidir?**

A Dosya JSON biçiminde olduğu için Java onu yanlış okur.

B Hangi sınıfın kurulacağını dosya söyler; okunurken o sınıfların kodu çalışır ve zararsız sınıflar zincirlenerek bir komut çalıştırılabilir.

C ObjectInputStream dosyayı okurken diskteki öteki dosyaları siler.

D Serileştirilmiş dosyalar şifreli olduğu için içi denetlenemez.

NESNE YÖNELİMLİ PROGRAMLAMA

11 Jenerikler ve koleksiyonlar

Bu bölümde ne tuttuğunu kullananın söylediği sınıf ve metotları, yani jenerikleri göreceğiz: hatayı derleme anına taşımalarını, sınırlı tipleri ve çalışma anında tiplerin neden silindiğini. Java'nın hazır koleksiyonlarını da tanıyacağız: liste, küme ve eşleme; sayıların listeye nasıl girdiğini, dolaşırken silmenin doğru yolunu ve HashSet'in equals ile hashCode'a nasıl dayandığını.

Kutuya ne girdi?

videoda 1:15:27

Bir kutu sınıfı yazıyoruz: içine bir şey konur, sonra geri alınır. Bu kutuya yalnız şehir adı koymak istiyoruz, ama biri yanlışlıkla bir sayı koyuyor. Bu hata ne zaman ortaya çıkmalı: kod yazılırken mi, yoksa program kullanıcının elindeyken mi?

Jenerikler olmadan iki yol var. Birincisi her tip için ayrı bir kutu sınıfı yazmak: MetinKutusu, SayiKutusu, TarihKutusu. Hepsinin kodu aynı, yalnız tip farklı; bir hata düzeltilince hepsi tek tek elden geçer. İkincisi tek bir kutu yazıp içine Object koymak. Java'da bütün sınıfların atası Object olduğu için bu kutuya her şey sığar:

```
public class Kutu {
    private Object icerik;
    public void koy(Object yeni) { icerik = yeni; }
    public Object al() { return icerik; }
}
```

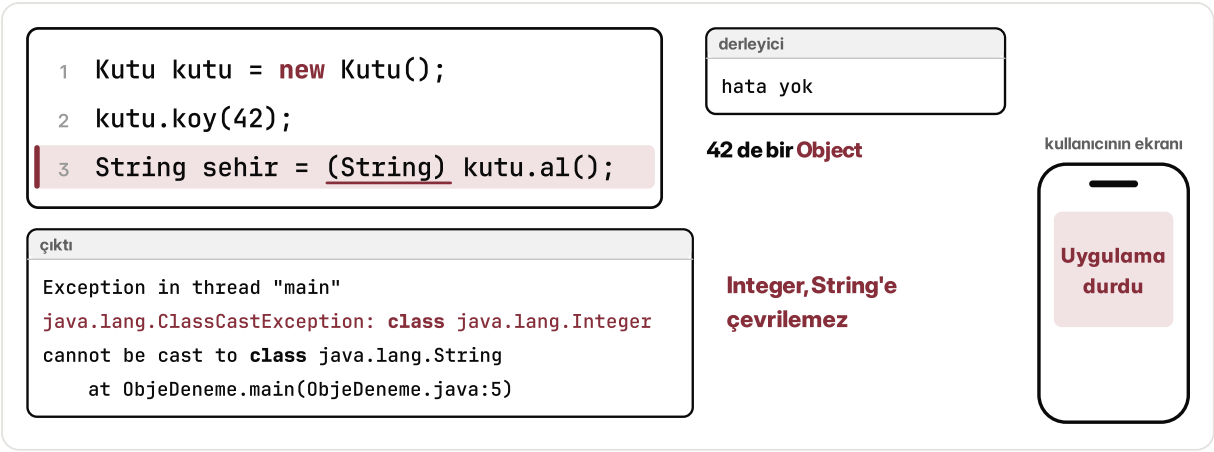
Ama bu kutudan çıkan şeyin tipi de Object'tir; onu metin olarak kullanmak için her seferinde (String) ile dönüştürmek gerekir. Biri kutuya sayı koyduysa derleyici hiçbir şey demez, çünkü 42 de sonuçta bir Object'tir:

```
Kutu kutu = new Kutu();
kutu.koy(42);
String sehir = (String) kutu.al();
```

Hata ancak program çalışırken, dönüştürme satırında çıkar:

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be
cast to class java.lang.String (java.lang.Integer and java.lang.String are in module
java.base of loader 'bootstrap')
    at ObjeDeneme.main(ObjeDeneme.java:5)
```

Java programı ClassCastException ile durdurur ve tam sayının (Integer) metne (String) çevrilemediğini söyler. Yani hata kullanıcının ekranında patlar.

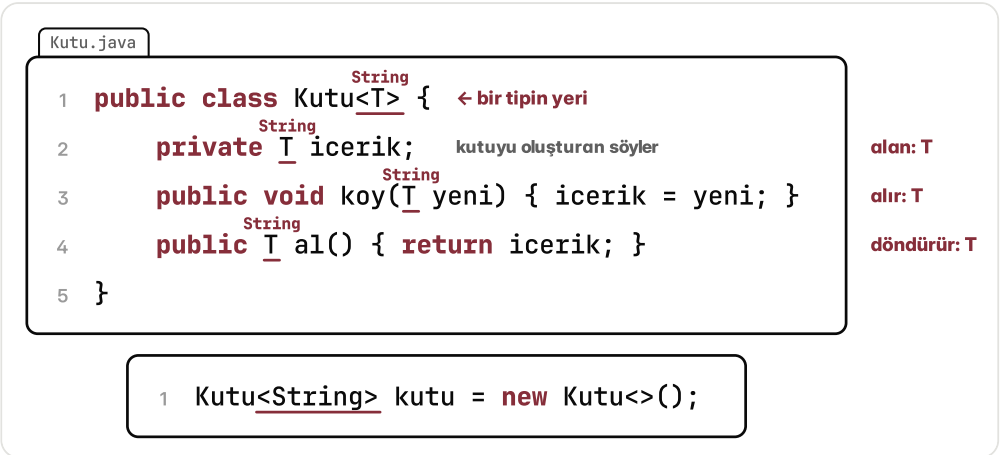


Şekil 11.1 Object kutusuna 42 konunca derleyici ses etmez; hata ancak çalışırken, dönüştürme satırında ClassCastException olarak çıkar.

Jenerik sınıf üçüncü bir yol açar. Sınıfın adının yanına, açılı parantez içinde bir harf yazılır: T. T bir tipin yerini tutar; anlamı şu: bu kutunun ne tuttuğunu, kutuyu oluşturan söyleyecek. Sınıfın içinde alanın tipi T'dir, koy metodu T alır, al metodu T döndürür:

```
public class Kutu<T> {
    private T icerik;
    public void koy(T yeni) { icerik = yeni; }
    public T al() { return icerik; }
}
```

T'ye **tip parametresi** denir. Şehir adı tutacak kutuyu oluştururken açılı parantezin içine String yazarız; bu kutuda T'nin yerini String alır.

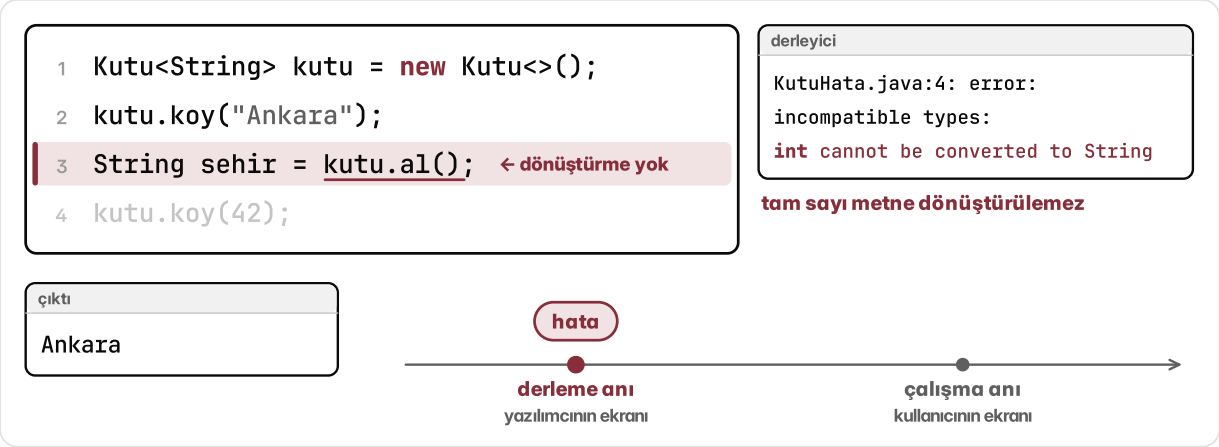


Şekil 11.2 Kutu<T>: alanın tipi, koy'un aldığı ve al'ın döndürdüğü T'dir; Kutu<String> oluşturulunca dört T'nin yerini String alır.

Şimdi aynı yanlışı yapalım:

```
Kutu<String> kutu = new Kutu<>();
kutu.koy("Ankara");
String sehir = kutu.al();
kutu.koy(42);
```

Son satır derlenmez. Derleyici açıkça söyler: "incompatible types: int cannot be converted to String", yani tam sayı metne dönüştürülemez. Kutudan çıkan şey de doğrudan metindir; üçüncü satırda dönüştürme yazmaya gerek kalmaz. Tip denetimi çalışma anından derleme anına taşındı: aynı hata artık kullanıcının ekranında değil, yazılımcının ekranında, program daha çalışmadan yakalanır.



Şekil 11.3 Kutu<String>'e 42 koyan satır derlenmez; kutudan çıkan dönüştürmesiz metindir, hata derleme anına taşınmıştır.

Java'nın listeleri de böyle jeneriktir; List<String> yalnız metin tutar:

```
List<String> sehirler = new ArrayList<>();
sehirler.add("Ankara");
String ilk = sehirler.get(0);
```

Jenerikler Java'ya 2004'te, Java 5 ile geldi. Ondan önce listelerde Object duruyordu, her okumada dönüştürme gerekiyordu ve yanlış eleman ancak program çalışırken fark ediliyordu. Sağ taraftaki boş açılı parantezin (<>) anlamı da kısa: tipi yeniden yazmayız, derleyici soldaki tipten çıkarır. Kutuya ne gireceğini artık kutuyu oluşturan söyler, derleyici de her satırda bu sözü denetler.

En büyüğü bulan tek metot

videoda 1:17:33

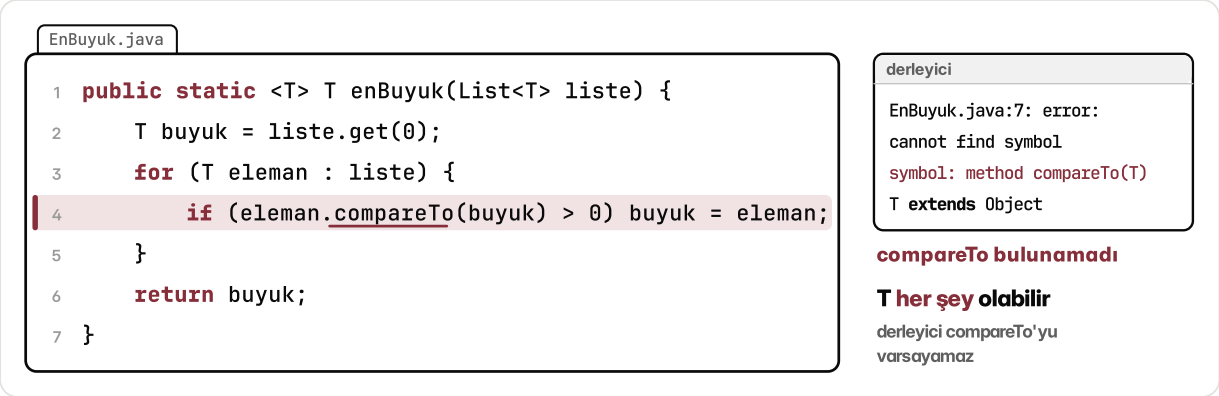
Bir hava durumu uygulaması haftanın en yüksek sıcaklığını arıyor; aynı uygulamada şehir adlarından alfabede en sonda geleni de gerekiyor. İkisi için tek bir enBuyuk metodu yazmak istiyoruz: tam sayı listesi verilince en büyük sayıyı, metin listesi verilince alfabede en sondakini versin. Ama "büyük" sözü her tip için anlamlı değil; anlamsız bir tip verilirse ne olacak?

Yalnız sınıflar değil, metotlar da jenerik olabilir. **Jenerik metodun** başına, dönüş tipinden önce, açılı parantez içinde T yazılır: bu metot T tipinde elemanları olan bir liste alır ve T tipinde bir eleman döndürür. T'nin ne olduğunu çağıran yazmaz; derleyici verilen listeden çıkarır: sıcaklık listesinde T Integer, şehir listesinde String olur.

Metodun içi kısa: ilk elemanı en büyük sayar, sonra listedeki her elemana sırayla bakar, daha büyüğünü bulursa onu tutar:

```
public static <T> T enBuyuk(List<T> liste) {
    T buyuk = liste.get(0);
    for (T eleman : liste) {
        if (eleman.compareTo(buyuk) > 0) buyuk = eleman;
    }
    return buyuk;
}
```

Ama karşılaştırma satırı derlenmez: "cannot find symbol, symbol: method compareTo(T)", yani iki elemanı karşılaştıran compareTo metodu bulunamadı. Mesaj T'yi "T extends Object" diye anar: T her şey olabilir ve derleyici her tipin bir karşılaştırma metodu olduğunu varsayamaz.



Şekil 11.4 Sınırsız T'de compareTo satırı derlenmez: T her şey olabilir, derleyici karşılaştırma metodunu varsayamaz.

Çözüm, T'ye bir sınır koymak. Açılı parantezde T'nin yanına extends Comparable<T> yazılır: T, kendini başka T'lerle karşılaştırabilen bir tip olmalı. Comparable, yani karşılaştırılabilir arayüzü, compareTo metodunu zorunlu kılar; sınırda extends sözcüğü arayüz için de kullanılır. Buna **sınırlı tip** (bounded type) denir:

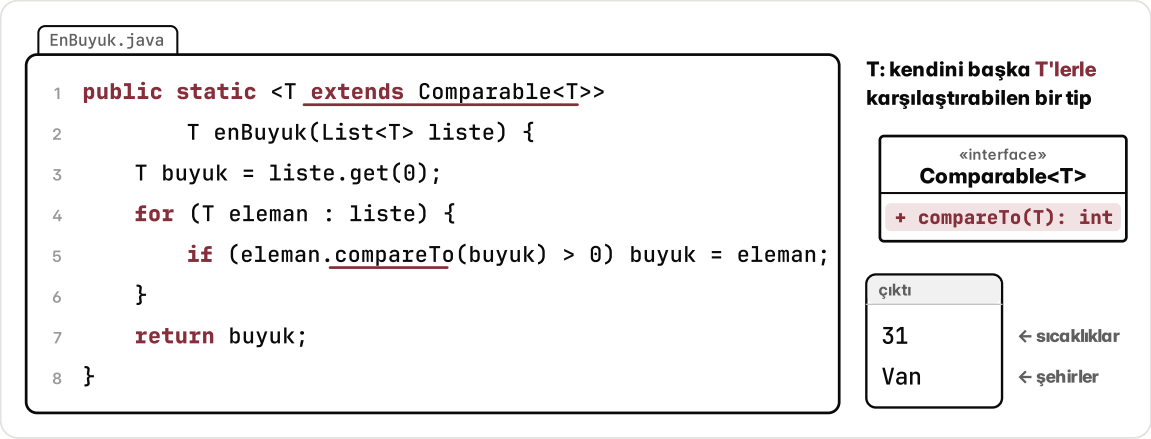
```
public static <T extends Comparable<T>>
        T enBuyuk(List<T> liste) {
    T buyuk = liste.get(0);
    for (T eleman : liste) {
        if (eleman.compareTo(buyuk) > 0) buyuk = eleman;
    }
    return buyuk;
}
```

Artık derleyici compareTo'nun var olduğunu bilir ve satır derlenir. Tam sayılar da metinler de karşılaştırılabilir:

```
List<Integer> sicakliklar = List.of(12, 31, 25, 18, 22, 27, 19);
List<String> sehirler = List.of("Erzurum", "Antalya", "Van");
System.out.println(enBuyuk(sicakliklar));
System.out.println(enBuyuk(sehirler));
```

Çıktı:

```
31
Van
```

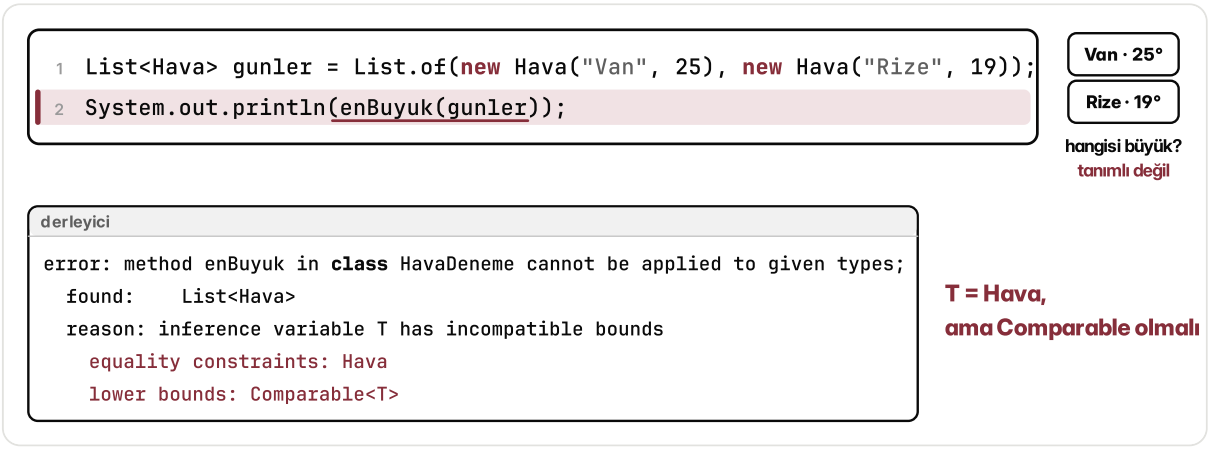


Şekil 11.5 <T extends Comparable<T>>: Comparable compareTo'yu zorunlu kılar; metot sıcaklıklarda 31, şehirlerde Van döndürür.

Şimdi anlamsız tipi deneyelim: her biri bir şehir ve bir derece taşıyan Hava nesnelerinden oluşan bir liste. Hava sınıfı Comparable değil; iki Hava nesnesinden hangisinin büyük olduğu tanımlı değil:

```
List<Hava> gunler = List.of(new Hava("Van", 25), new Hava("Rize", 19));
System.out.println(enBuyuk(gunler));
```

İkinci satır derlenmez. Derleyicinin uzun mesajındaki "inference variable T has incompatible bounds" sözü bunu söyler: T, Hava olarak çıkarıldı, ama Hava Comparable sınırını karşılamıyor.



Şekil 11.6 Comparable olmayan Hava listesiyle çağrı derlenmez: T = Hava çıkar ama sınırı karşılamaz.

Sınırlı tip, metotla onu çağıran arasında bir sözdür: metot yalnız karşılaştırılabilir tipleri kabul eder, derleyici her çağrıda bu sözü denetler. Java'nın kendi en büyük ve sıralama metotları, Collections.max ve Collections.sort da böyle sınırlı tiplerle yazılmıştır. Yanlış tip verilirse hata yine program çalışmadan, derlenirken yakalanır.

127 eşit, 128 eşit değil

videoda 1:19:26

Bir depo programı iki ürünün stok adedini bir listede tutuyor ve stoklar eşit mi diye == ile bakıyor:

```
List<Integer> stok = new ArrayList<>();
stok.add(128);
stok.add(128);
System.out.println(stok.get(0) == stok.get(1));
```

İki stok da 127'yken cevap true; ikisi de 128 olunca false. Aynı kod, bir sayı fark yüzünden neden ters sonuç verir?

Önce listeye bakalım. Tam sayı listesi, açılı parantezin içine int yazılarak kurulamaz:

```
List<int> stok = new ArrayList<>();
```

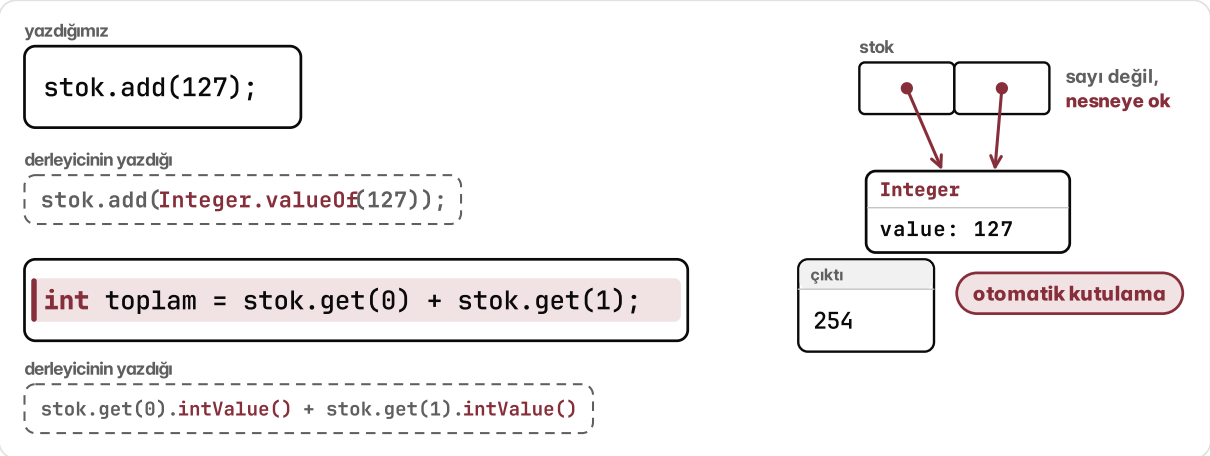
Bu satır derlenmez: "unexpected type, required: reference, found: int", yani burada bir referans tipi bekleniyordu, int bulundu. Jenerik tipler yalnız nesne tutar; int ise bir temel tiptir, değerin kendisidir, nesne değildir.

Bu yüzden her temel tipin bir **sarmalayıcı sınıfı** (wrapper class) vardır; int'in sarmalayıcısı Integer'dır. Integer, içinde tek bir tam sayı değeri taşıyan küçük bir nesnedir; nesne olduğu için listeye girebilir. Öteki temel tiplerin de birer sarmalayıcısı var: double için Double, char için Character, boolean için Boolean. Tam sayı listesi bu yüzden List<Integer> diye kurulur.

Listeye 127 eklenirken kutuyu biz yazmayız; derleyici sayıyı kendisi Integer'a çevirir: stok.add(127); satırını stok.add(Integer.valueOf(127)); olarak derler. Buna **otomatik kutulama** (autoboxing) denir. Listeden okurken de kutuyu kendisi açar ve içindeki sayıyı intValue ile alır; iki stok toplanınca 254 çıkar:

```
int toplam = stok.get(0) + stok.get(1);
```

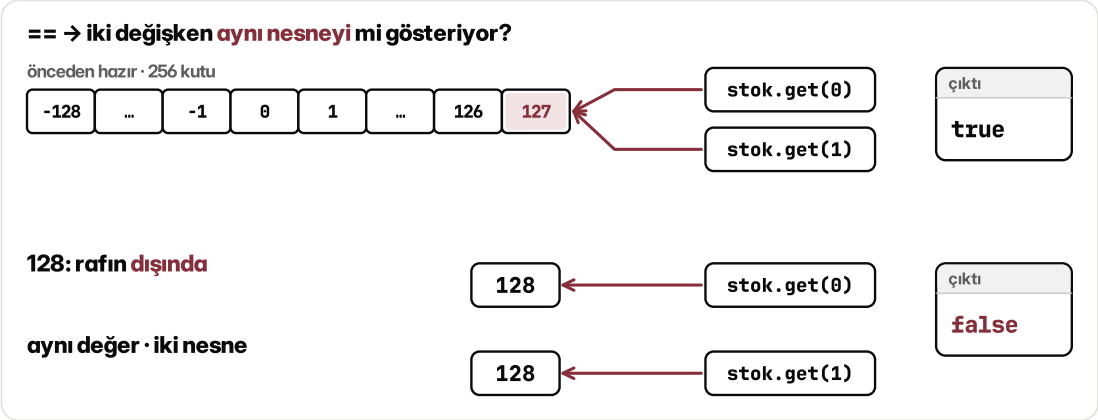
Yani listede sayıların kendisi değil, sayı taşıyan nesneler durur.



Şekil 11.7 Derleyici add(127)'yi Integer.valueOf(127)'ye, toplamayı intValue çağrılarına çevirir; listede sayı değil, nesneye ok durur.

İki nesne == ile karşılaştırılınca değerlere değil, iki değişkenin aynı nesneyi gösterip göstermediğine bakılır. Kutulama yapan Integer.valueOf, −128'den 127'ye kadar her sayının kutusunu önceden hazırlar ve her istenişte aynı kutuyu verir, çünkü küçük sayılar çok kullanılır. 127 iki kez kutulanınca iki kez aynı

nesne gelir: == true der. 128 bu aralığın dışındadır; her kutulamada yeni bir nesne oluşur ve aynı değeri taşıyan iki ayrı nesne çıkar: == false der.



Şekil 11.8 –128..127 arası 256 kutu önceden hazırdır: 127'de iki değişken aynı nesneyi gösterir, 128'de aynı değerli iki ayrı nesne çıkar.

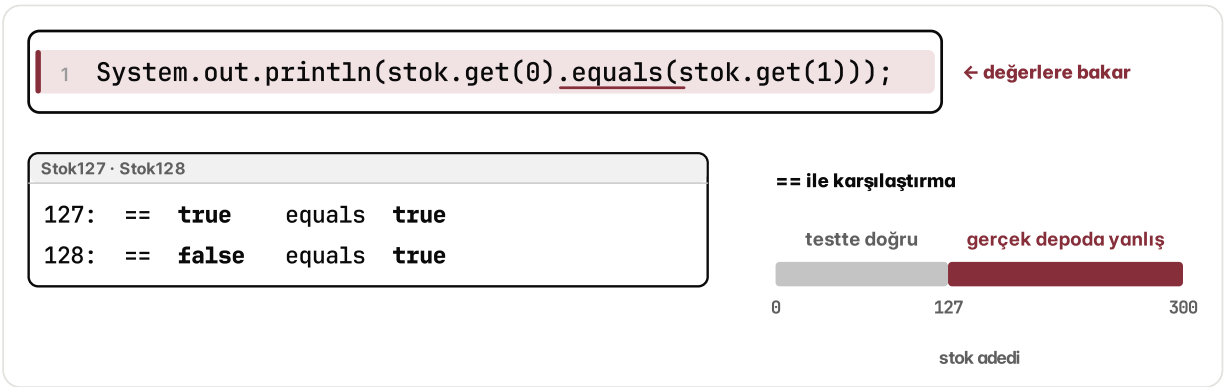
Bu yüzden sarmalayıcıları karşılaştırmanın doğru yolu equals'tır; equals değerlere bakar:

```
System.out.println(stok.get(0).equals(stok.get(1)));
```

128 ile çalışan programın üç satırı (==, equals ve toplam):

```
false
true
256
```

127 ile aynı program true, true ve 254 yazar. Bu hata sinsidir: küçük sayılarla denenen kod doğru çalışır, gerçek depoda stok 127'yi geçince yanlış sonuç vermeye başlar.



Şekil 11.9 equals değerlere bakar ve iki durumda da true verir; == yalnız 127'ye kadar doğru görünür.

Beş şehir mi, üç şehir mi?

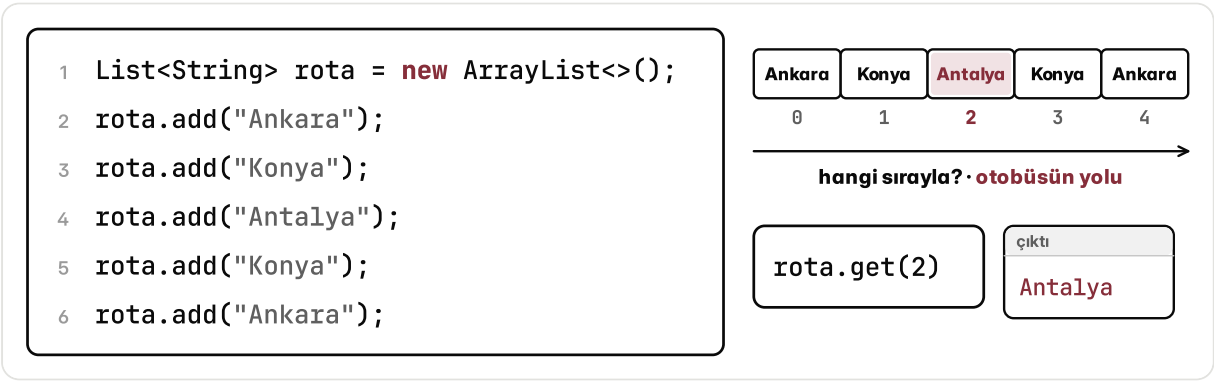
videoda 1:21:18

Bir tur otobüsü bir hafta boyunca şu şehirlerden geçti: Ankara, Konya, Antalya, yine Konya ve yine Ankara. Bu beş adı bir listeye koyunca beş eleman çıkıyor, bir kümeye koyunca üç. Aynı veri neden iki koleksiyonda farklı sayılır?

Liste (List), elemanları eklendikleri sırayla tutar ve aynı elemanın tekrar girmesine izin verir; Konya da Ankara da iki kez yer alır:

```
List<String> rota = new ArrayList<>();
rota.add("Ankara");
rota.add("Konya");
rota.add("Antalya");
rota.add("Konya");
rota.add("Ankara");
```

rota.size() 5 verir. Her elemanın sıfırdan başlayan bir sıra numarası vardır: rota.get(2) Antalya'dır. Liste "hangi sırayla?" sorusunun koleksiyonudur: burada otobüsün izlediği yol.



Şekil 11.10 Liste beş adı eklenme sırasıyla, tekrarlarıyla tutar; sıra numaraları 0'dan başlar ve rota.get(2) Antalya'dır.

List bir arayüzdür; onu gerçekleştiren iki sınıf ArrayList ve LinkedList'tir. İkisi de aynı metotlarla kullanılır; programda new ArrayList<>() yerine new LinkedList<>() yazınca bu örnekte çıktı birebir aynı kalır. Farkları içlerindeki yapıdan gelir. Bu bilgisayarda bir denemede ArrayList'e 100.000 elemanı listenin başına eklemek 934 ms, yani bir saniyeye yakın sürdü, LinkedList'e 17 ms. Ortadaki elemanı 10.000 kez okumaksa tam tersi: ArrayList'te 3 ms, LinkedList'te 2.761 ms. Çoğu işte ArrayList seçilir; LinkedList, listenin başına sık sık eleman eklenip çıkarılan işlerde öne geçer.



Şekil 11.11 Bir denemenin ölçümü: başa ekleme LinkedList'te, ortadan okuma ArrayList'te çok daha hızlı.

Küme (Set) ise her elemanı bir kez tutar. Listedeki beş adla bir küme oluşturunca içinde üç şehir kalır:

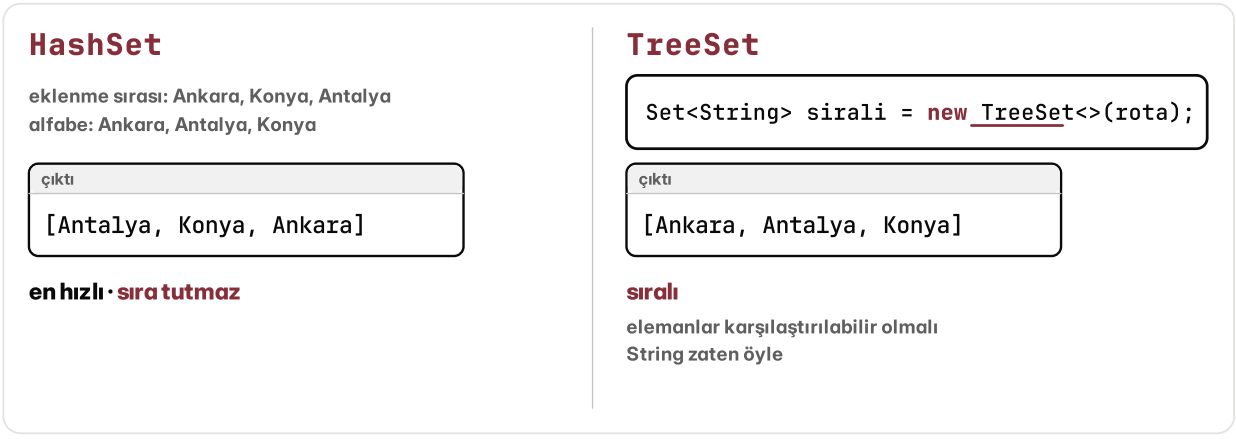
```
Set<String> sehirler = new HashSet<>(rota);
```

sehirler.size() 3 verir. Konya'yı bir kez daha eklemeye çalışınca küme değişmez; sehirler.add("Konya") eklemediğini bildirmek için false döndürür. Küme "hangileri var?" ve "bu var mı?" sorularının koleksiyonudur: sehirler.contains("Antalya") true verir.

Kümenin de sık kullanılan iki sınıfı var. HashSet en hızlısıdır, ama sıra tutmaz: bu örnekte şehirler [Antalya, Konya, Ankara] diye çıktı; ne eklenme sırası ne alfabe. TreeSet elemanları sıralı tutar:

```
Set<String> sirali = new TreeSet<>(rota);
```

Bu küme [Ankara, Antalya, Konya] diye çıkar. Bunun için elemanların karşılaştırılabilir olması gerekir; metinler zaten öyledir.



Şekil 11.12 HashSet sıra tutmaz: [Antalya, Konya, Ankara]; TreeSet elemanları sıralı tutar: [Ankara, Antalya, Konya].

Aynı beş ad, listede otobüsün izlediği yolu, kümede görülen şehirleri anlatır. Hangi koleksiyonun seçileceğini, verinin hangi soruya cevap vereceği belirler: sıra ve tekrar önemliyse liste; her elemanın bir kez bulunması ve "var mı?" sorusu önemliyse küme.

Konya'nın plakası kaç?

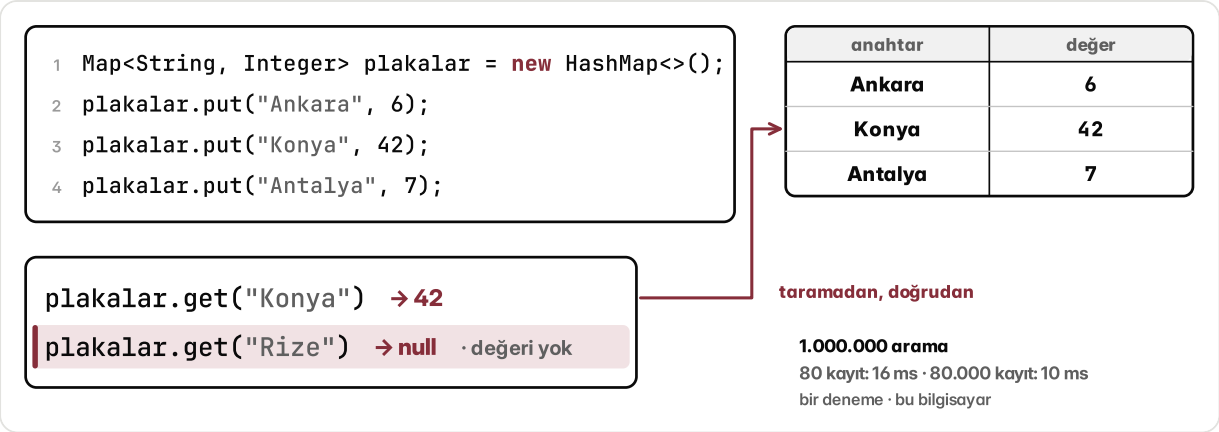
videoda 1:23:15

81 ilin adı ve plakası bir listede dursa, Konya'nın plakasını bulmak için liste baştan taranır; Konya hangi sıradaysa ona kadar her elemana tek tek bakılır. Plaka sırasıyla dizilmiş listede bu 42 bakış demektir. Bir şeyi adıyla, doğrudan bulmanın yolu var mı?

Var: **eşleme**; Java'daki adı Map. Eşlemede her kayıt bir çifttir: bir **anahtar** ve ona bağlı bir **değer**. Burada anahtar ilin adı, değer plakası: Ankara 6, Konya 42, Antalya 7. Açılı parantezde bu kez iki tip yazılır: önce anahtarın tipi, sonra değerin; burada String ve Integer.

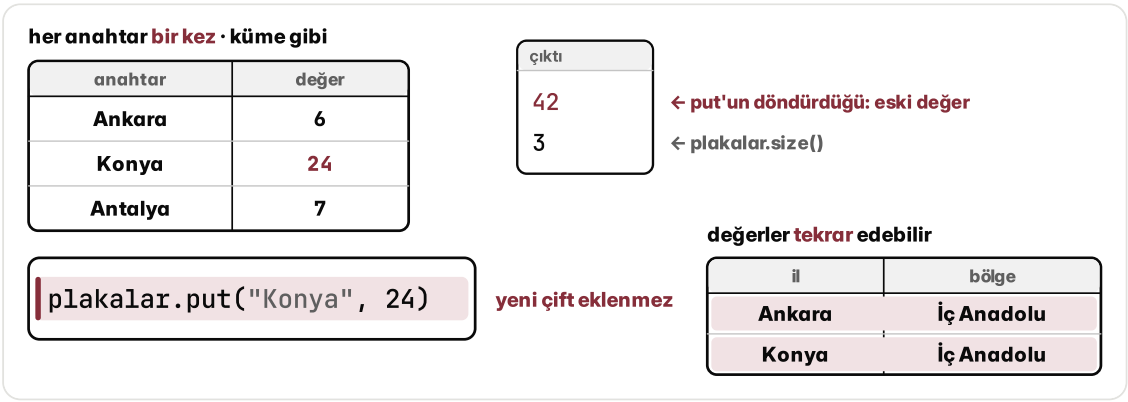
```
Map<String, Integer> plakalar = new HashMap<>();
plakalar.put("Ankara", 6);
plakalar.put("Konya", 42);
plakalar.put("Antalya", 7);
```

Çiftler put ile tek tek eklenir; sonra get'e anahtar verilir, değer gelir: plakalar.get("Konya") 42 verir. HashMap bu aramayı listeyi taramadan yapar: kayıt sayısı 80 de olsa 80.000 de olsa anahtarın yerine doğrudan gider. Bir denemede 1.000.000 arama, 80 kayıtlı eşlemede 16 ms, 80.000 kayıtlıda 10 ms sürdü; süre kayıt sayısıyla büyümedi. Eşlemede olmayan bir anahtar sorulursa, örneğin plakalar.get("Rize"), değer yerine null döner: bu anahtarın bir değeri yok demektir.



Şekil 11.13 Anahtar verilir, değer gelir: get("Konya") 42, eşlemede olmayan get("Rize") null; arama listeyi taramadan yapılır.

Eşlemede her anahtar bir kez bulunur, tıpkı bir kümedeki elemanlar gibi. Konya'yı bu kez yanlış bir plakayla bir daha ekleyelim: plakalar.put("Konya", 24). Yeni bir çift eklenmez, eski değerin yerine yenisi geçer. Eşlemede yine üç çift vardır, plakalar.size() 3 verir ve Konya artık 24 der. put, silinen eski değeri, yani 42'yi döndürür. Değerler ise tekrar edebilir: iki farklı anahtarın değeri aynı olabilir; illeri bölgelerine eşleyen bir tabloda Ankara da Konya da İç Anadolu'dur.



Şekil 11.14 put("Konya", 24) yeni çift eklemez, eski değeri değiştirir ve 42'yi döndürür; boyut 3 kalır, değerler tekrar edebilir.

Bir koleksiyon seçerken veriye üç soru sorulur. Sıra önemli mi, aynı eleman tekrar edebilir mi: liste. Önemli olan her elemanın bir kez bulunması ve "bu var mı?" sorusu mu: küme. Bir şeyi anahtarıyla bulmak mı gerekiyor: eşleme. Sonra sınıf seçilir: çoğu işte ArrayList, HashSet ve HashMap; elemanları sıralı tutmak gerekirse TreeSet.



Şekil 11.15 Üç soru, üç koleksiyon: sıra ve tekrar için List, bir kez ve var mı için Set, anahtarla bulmak için Map.

Bir mesajlaşma uygulamasında üçü yan yana çalışır:

```
List<String> mesajlar = new ArrayList<>();
Set<String> engellenenler = new HashSet<>();
Map<String, Integer> okunmamis = new HashMap<>();
```

Bir sohbetteki mesajlar bir listedir: sıra önemli, aynı mesaj birkaç kez yazılabilir. Engellenen kişiler bir kümedir: bir kişi ya engellidir ya değildir; yeni mesaj gelince yalnız "bu kişi engelli mi?" diye sorulur. Kimden kaç okunmamış mesaj olduğunu tutan da bir eşlemedir: kişiyi veririz, sayı gelir. Üçü de aynı kişiler ve mesajlar üstünde çalışır, ama her biri başka bir soruya cevap verir.

Gelemeyenleri silmek

videoda 1:25:11

Bir doğum günü için davetli listesi hazır: Ali, Ayşe, Can ve Deniz; Ayşe gelemeyeceğini söyledi. davetliler bir ArrayList, gelemeyenler yalnız Ayşe'yi içeren bir küme. Listeyi dolaşıp gelemeyenleri silen kısa bir döngü yazıyoruz:

```
for (String ad : davetliler) {
    if (gelemeyenler.contains(ad)) {
        davetliler.remove(ad);
    }
}
```

Program çalışırken hatayla duruyor:

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
    at Davet.main(Davet.java:9)
```

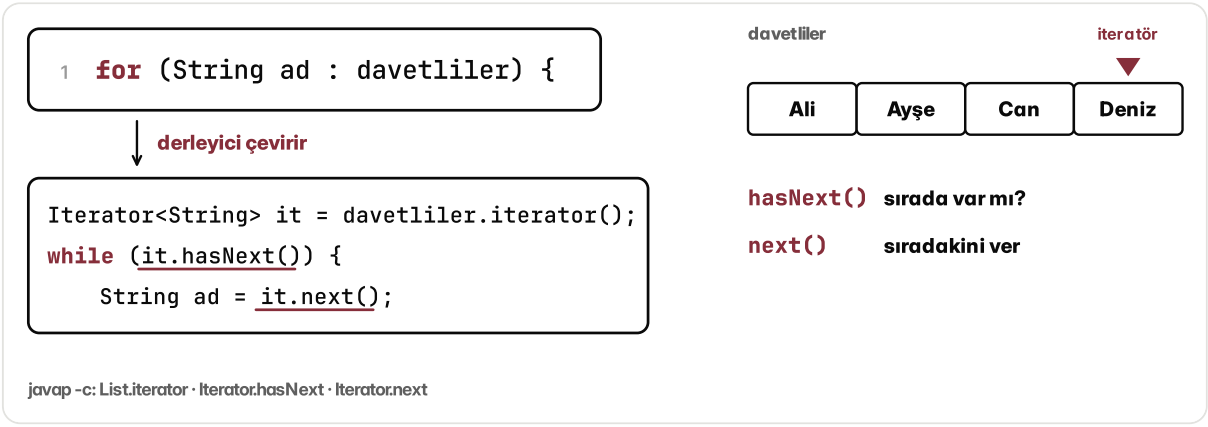
Bir listeyi dolaşırken ondan eleman silmek neden sorun?

Önce dolaşmanın kısa yolu: **for-each** döngüsü. for (String ad : davetliler) listedeki her elemanı sırayla ad değişkenine verir; sıra numarasıyla uğraşmayız. Aynı döngü kümelerde de dizilerde de böyle çalışır.

Koleksiyonlarda bu kısa döngü, perde arkasında **iteratör** (Iterator) denen bir nesne kullanır: koleksiyonun üstünde ilerleyen bir imleç. İteratörün iki temel metodu var: hasNext sırada eleman var mı diye sorar, next sıradakini verir. Derleyici for-each döngüsünü bu iki metodu çağıran bir döngüye çevirir:

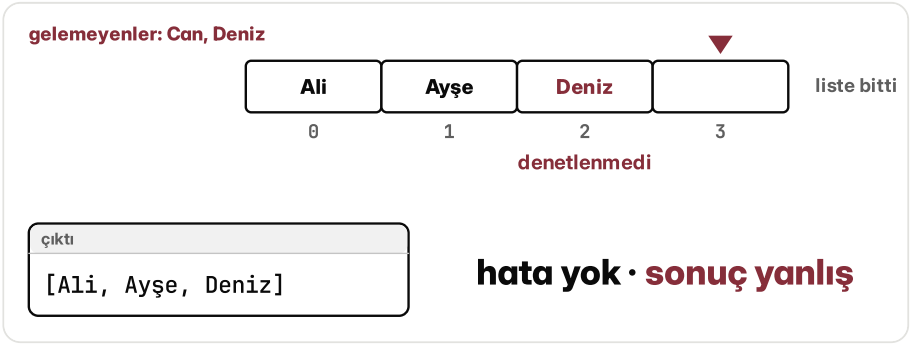
```
Iterator<String> it = davetliler.iterator();
while (it.hasNext()) {
    String ad = it.next();
```

Derlenmiş kodda kısa döngü değil, iteratör vardır.



Şekil 11.16 Derleyici for-each'i iteratöre çevirir: hasNext sırada eleman var mı diye sorar, next sıradakini verir.

Döngünün içinde listeden doğrudan silince liste değişir, ama iteratörün bundan haberi olmaz. Ayşe silinince arkasındaki elemanlar birer sıra öne kayar; iteratör kaldığı sıradan devam etse Can'ı atlardı. Java bu değişikliği sıradakini isteyen next'te fark eder ve programı durdurur: liste, dolaşılırken değiştirilmiş. Ama bu denetim her zaman işe yaramaz. Gelemeyenler Can ve Deniz olsaydı, Can silinince liste üç elemana iner, hasNext sırada eleman yok der ve döngü hata vermeden biter. Sonuç [Ali, Ayşe, Deniz] olur: Deniz listede kalır. Program durmaz, ama yanlış sonuç verir.

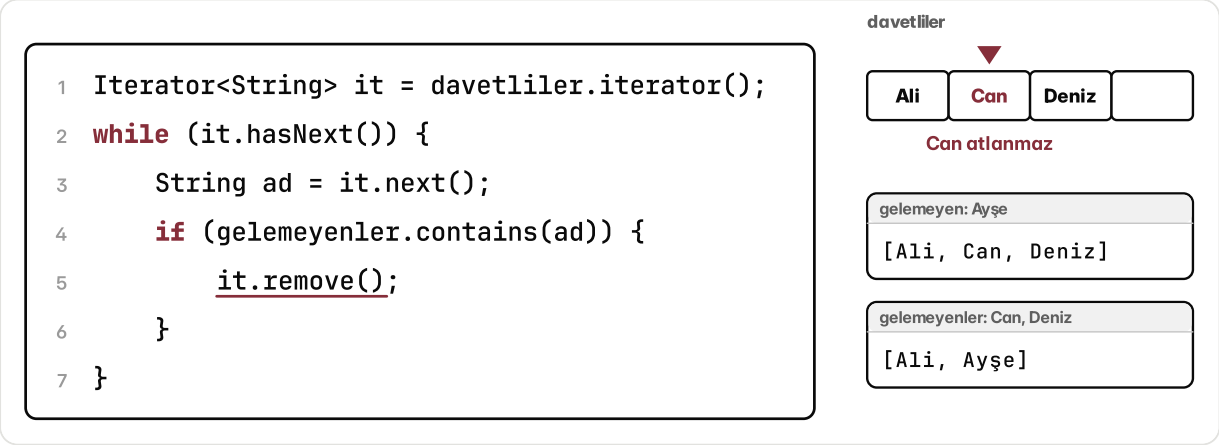


Şekil 11.17 Gelemeyenler Can ve Deniz: Can silinince Deniz kayar ve döngü biter; hata yok ama Deniz listede kalır.

Doğru yol, silmeyi iteratörün kendisine yaptırmaktır. Döngüyü iteratörle açıkça yazarız; silinecek elemanda iteratörün kendi remove metodunu çağırırız:

```
Iterator<String> it = davetliler.iterator();
while (it.hasNext()) {
    String ad = it.next();
    if (gelemeyenler.contains(ad)) {
        it.remove();
    }
}
```

İteratör listeyi kendisi değiştirdiği için yerini bilir, kayan elemanları atlamaz. Ayşe gelemezse liste [Ali, Can, Deniz] olur; Can ile Deniz gelemezse [Ali, Ayşe] kalır.



Şekil 11.18 it.remove() ile silen iteratör Can'ı atlamaz: Ayşe gelemezse [Ali, Can, Deniz], Can ve Deniz gelemezse [Ali, Ayşe].

Kural kısa: bir koleksiyonu dolaşırken onu yalnız dolaşan değiştirir. Yalnız okumak için for-each yeter; dolaşırken silmek gerekiyorsa iteratör.

Çekilişte aynı kişi iki kez

videoda 1:26:51

Bir çekilişe herkes bir kez katılabilir; katılımcılar, tekrarı kabul etmeyen bir kümede, HashSet'te tutuluyor. Elif aynı telefon numarasıyla iki kez kaydoluyor:

```
Set<Kisi> katilimcilar = new HashSet<>();
katilimcilar.add(new Kisi("Elif", "0555 000 11 22"));
katilimcilar.add(new Kisi("Elif", "0555 000 11 22"));
System.out.println(katilimcilar.size());
```

Çıktı:

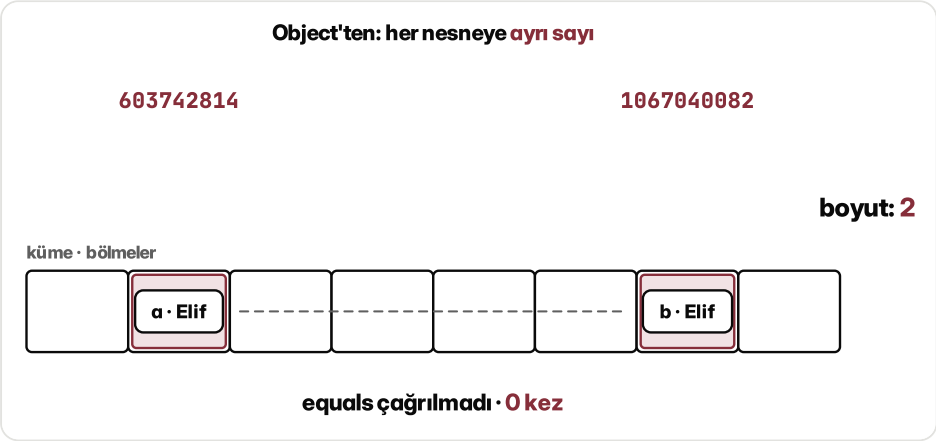
```
2
```

Tekrarı atması gereken küme aynı kişiyi neden iki kez sayar? Her kayıt yeni bir Kisi nesnesi oluşturur: içerikleri aynı, ama bellekte iki ayrı nesne. Kisi sınıfı equals metodunu yeniden yazmış; iki kişi, telefon numaraları aynıysa eşittir:

```
@Override
public boolean equals(Object o) {
    return o instanceof Kisi k && telefon.equals(k.telefon);
}
```

o instanceof Kisi k, o bir Kisi ise onu k adıyla kullanmamızı sağlar. Aynı ad ve telefonla oluşturulan a ve b için a.equals(b) true verir; yani sorun eşitlikte değil.

HashSet bir eleman eklenirken önce onun sayısını, yani **haş kodunu** (hashCode) ister; bu sayı kümenin hangi bölmesine bakılacağını seçer. equals yalnız aynı sayıyı veren elemanlar arasında çağrılır. Kisi sınıfı hashCode yazmamış; o hâlde Object'teki hashCode çalışır ve her nesneye ayrı bir sayı verir. Bu çalıştırmada iki kaydın sayıları 603742814 ve 1067040082 çıktı. Sayılar farklı olunca equals hiç çağrılmaz; küme onları iki ayrı kişi sanır.

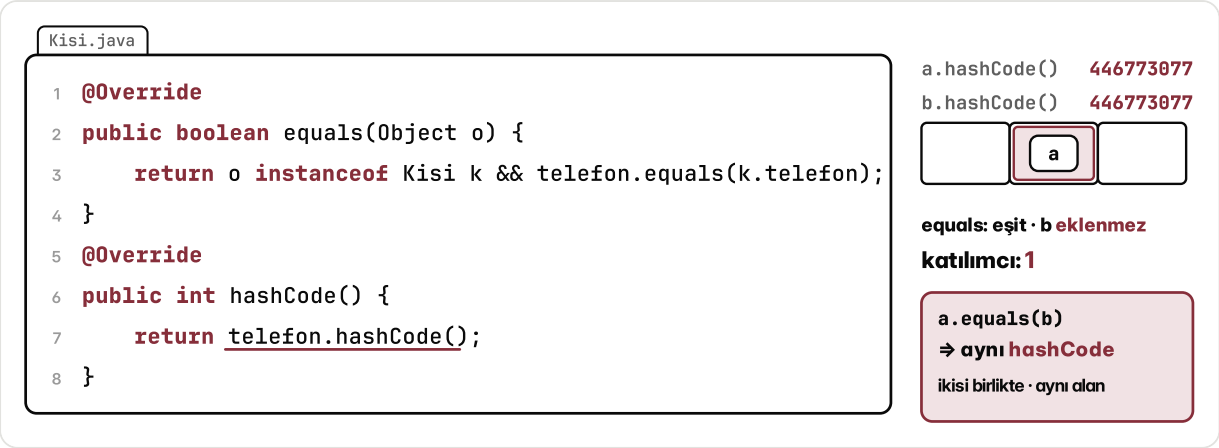


Şekil 11.19 Object'in hashCode'u iki nesneye ayrı sayılar verir; iki kayıt ayrı bölmelere düşer, equals hiç çağrılmaz ve boyut 2 olur.

Çözüm, hashCode'u da aynı alandan, telefon numarasından üretmek:

```
@Override
public boolean equals(Object o) {
    return o instanceof Kisi k && telefon.equals(k.telefon);
}
@Override
public int hashCode() {
    return telefon.hashCode();
}
```

Artık eşit iki kayıt aynı sayıyı verir; bu çalıştırmada ikisi de 446773077. Küme aynı bölmeye bakar, equals eşit der ve ikinci kayıt eklenmez: kümede bir katılımcı kalır, çıktı 1 olur. Kural şudur: equals'a göre eşit olan iki nesne aynı hashCode'u vermeli; ikisi hep birlikte ve aynı alanlara göre yazılır.

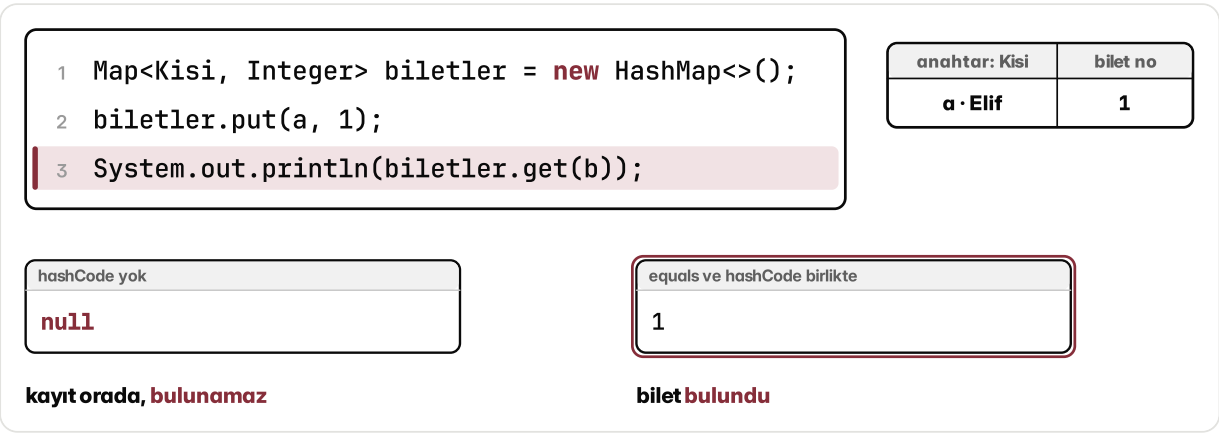


Şekil 11.20 hashCode telefondan üretilince iki kayıt aynı sayıyı verir, equals eşit der ve ikinci kayıt eklenmez: katılımcı 1.

HashMap'in anahtarları için de aynı kural geçerli. Kişiyi anahtar yapıp ona bir bilet numarası bağlayalım:

```
Map<Kisi, Integer> biletler = new HashMap<>();
biletler.put(a, 1);
System.out.println(biletler.get(b));
```

hashCode yazılmamışsa, aynı telefonlu b nesnesiyle sorunca eşleme null döndürür: kayıt orada, ama bulunamaz. İki metot birlikte yazılınca aynı soru 1 numaralı bileti getirir.



Şekil 11.21 HashMap'te de aynı kural: hashCode yoksa get(b) null döner, equals ve hashCode birlikte yazılınca bilet 1 bulunur.

HashSet'in tekrarı atabilmesi ve HashMap'in aradığını bulabilmesi, elemanların bu iki metodu birlikte yazmasına dayanır; çekilişin adil olması da.

Neyin listesi?

videoda 1:28:27

Bir metin listesi ve bir tam sayı listesi oluşturalım, sonra program çalışırken ikisinin sınıfını getClass ile soralım:

```
List<String> metinler = new ArrayList<>();
List<Integer> sayilar = new ArrayList<>();
System.out.println(metinler.getClass() == sayilar.getClass());
```

Çıktı:

```
true
```

İkisinin sınıfı da java.util.ArrayList çıkar; açılı parantezlerin izi bile yoktur. Program çalışırken bir listenin neyin listesi olduğunu soramıyorsak, açılı parantezdeki tipler nereye gitti?

Derleyici açılı parantezdeki tipleri yalnız denetim için kullanır; denetim bitince onları siler. Buna **tip silme** (type erasure) denir. Örneğin Kutu<T> sınıfı derlenince alanın ve metotların tipi Object olur: icerik Object'tir, koy Object alır, al Object döndürür. T'ye Comparable sınırı konduysa T'nin yerini Comparable alır. Kutudan metin alan her satıra, örneğin String sehir = kutu.al(); satırına, (String) dönüştürmesini derleyici kendisi ekler; tipi zaten denetlediği için bu dönüştürme güvenlidir.



Şekil 11.22 Önce denetle, sonra sil: Kutu<T>'nin derlenmiş hâlinde T yerine Object, sınırlı tipte Comparable; okuma satırına (String) derleyici ekler.

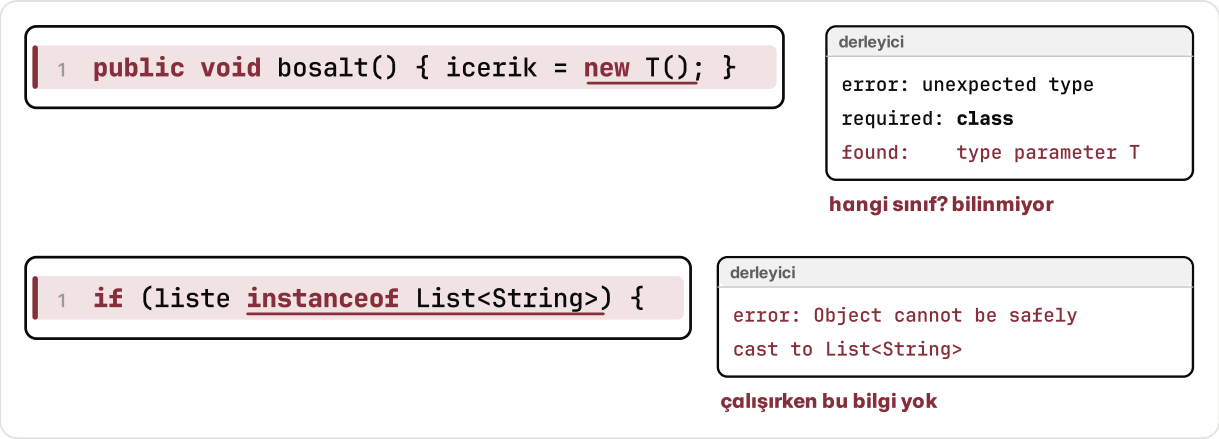
Çalışma anında T'nin ne olduğu bilinmediği için iki tür satır yazılamaz. Birincisi, jenerik sınıfın içinde T tipinde yeni bir nesne oluşturmak:

```
public void bosalt() { icerik = new T(); }
```

Hangi sınıfın nesnesinin oluşturulacağını bilen yoktur; derleyici "unexpected type, required: class, found: type parameter T" der. İkincisi, Object tipindeki bir değişkenin metin listesi olup olmadığını instanceof ile sormak:

```
if (liste instanceof List<String>) {
```

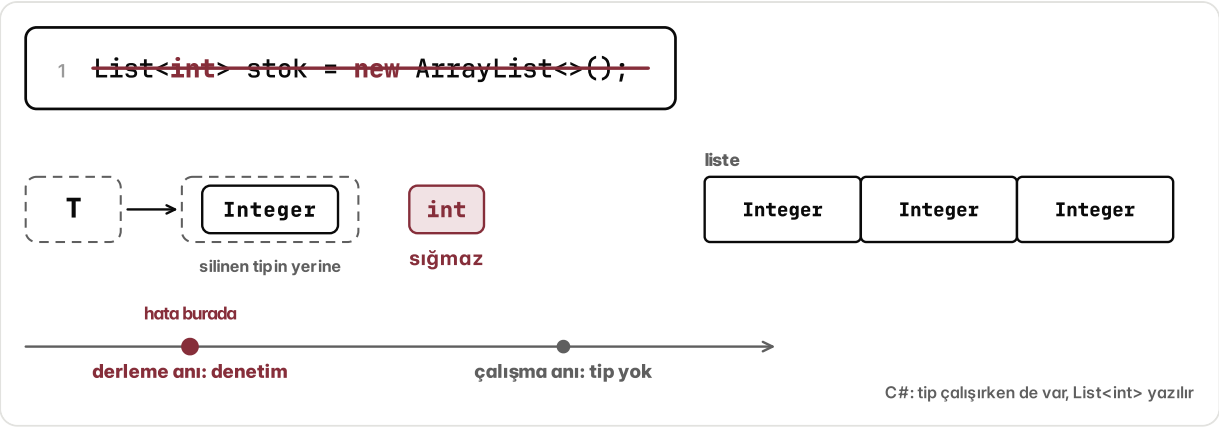
Çalışırken bu bilgi yoktur; derleyici "Object cannot be safely cast to List<String>" diyerek satırı reddeder.



Şekil 11.23 Yazılamayan iki satır: new T() (hangi sınıf bilinmiyor) ve instanceof List<String> (çalışırken bu bilgi yok).

Neden böyle? Jenerikler Java'ya 2004'te, Java 5 ile geldi. O güne kadar yazılmış milyonlarca satır kod ve derlenmiş kütüphaneler vardı; tip silme sayesinde yeni jenerik kod, eski kodla ve eski kütüphanelerle değişmeden birlikte çalıştı. Bedeli, tipin çalışma anında bilinmemesi.

List<int> yazılamamasının nedeni de bu. Silinen tipin yerine Object geçer; int ise bir nesne değildir, Object'in yerine oturamaz. Bu yüzden listeler sayıları Integer nesnelerine sarılmış olarak tutar. Her dil böyle değildir: C#'ta tip çalışırken de bilinir ve List<int> yazılabilir. Java'da jenerikler, derleyicinin bizim yerimize yaptığı bir denetimdir: hata derlenirken yakalanır, program çalışırken o tipler artık yoktur.



Şekil 11.24 Silinen T'nin yerine Object geçer; int bu yere sığmaz, Integer sığar: denetim derleme anında, çalışırken tip yok.

Özet

- Jenerik sınıf ve metotta T bir tipin yerini tutar; tip denetimi derleme anına taşınır ve dönüştürme yazmaya gerek kalmaz.
- Sınırlı tip (<T extends Comparable<T>>) metodun yalnız karşılaştırılabilir tipleri kabul etmesini sağlar; derleyici her çağrıyı denetler.
- Jenerikler yalnız nesne tutar; int listeye Integer olarak girer, derleyici kutular ve açar. Sarmalayıcılar == ile değil equals ile karşılaştırılır.
- Liste sıra ve tekrar tutar, küme her elemanı bir kez tutar, eşleme anahtarla değere doğrudan gider; çoğu işte ArrayList, HashSet ve HashMap seçilir.
- Koleksiyonu dolaşırken yalnız dolaşan değiştirir: silmek için iteratörün remove metodu kullanılır.
- HashSet ve HashMap önce hashCode'a bakar; equals ve hashCode birlikte, aynı alanlardan yazılır. Jenerik tipler derlendikten sonra silinir.

Sorular

1 **Kutu<T> jenerik bir sınıf; koy metodu T alıyor. Aşağıdaki satırlar için hangisi doğrudur?**

```
Kutu<String> kutu = new Kutu<>();
kutu.koy(42);
```

A Kod derlenir, program çalışırken ClassCastException verir.

B İkinci satır derlenmez: tam sayı String'e dönüştürülemez.

C Kod derlenir ve 42 kutuya "42" metni olarak konur.

D Kod derlenir; hata ancak kutu.al() çağrılınca çıkar.

2 **public static <T> T enBuyuk(List<T> liste) metodunun içindeki eleman.compareTo(buyuk) çağrısı derlenmiyor. Metodun başı nasıl yazılırsa çağrı derlenir?**

A public static T enBuyuk(List<T> liste)

B public static <T extends Object> T enBuyuk(List<T> liste)

C public static <Comparable> T enBuyuk(List<T> liste)

D public static <T extends Comparable<T>> T enBuyuk(List<T> liste)

3 **Sınırlı tiple yazılmış enBuyuk metodu ilk elemanı en büyük sayıp compareTo ile daha büyüğünü arıyor. System.out.println(enBuyuk(List.of("Bursa", "Adana", "Mersin"))); ekrana ne yazar?**

Cevap:

4 **Aşağıdaki kod ekrana ne yazar?**

```
List<Integer> stok = new ArrayList<>();
stok.add(1000);
stok.add(1000);
System.out.println(stok.get(0) == stok.get(1));
System.out.println(stok.get(0).equals(stok.get(1)));
```

A true ve true

B true ve false

C false ve true

D false ve false

5 **List<int> stok = new ArrayList<>(); satırı neden derlenmez?**

A int bir temel tiptir, nesne değildir; jenerik tipler yalnız nesne tutar, silinen tipin yerine geçen Object'in yerine int oturamaz.

B ArrayList yalnız metin tutabilir.

C Tam sayı listesi için açılı parantez yazılmaz; List stok yazmak gerekir.

D int'in sarmalayıcı sınıfı olmadığı için listeye giremez.

6 **Bir rota listesine sırayla "Rize", "Trabzon", "Rize", "Artvin", "Rize" eklendi. Set<String> s = new HashSet<>(rota); satırından sonra s.size() kaç verir?**

Cevap: _____

7 **Map<String, Integer> plakalar = new HashMap<>(); eşlemesine put("Ankara", 6) ve put("Konya", 42) yapıldı. Sonra plakalar.put("Konya", 24) çağrılıyor. Bu put neyi döndürür ve plakalar.size() kaç olur?**

A null döndürür, boyut 3 olur.

B 24 döndürür, boyut 2 olur.

C 42 döndürür, boyut 3 olur.

D 42 döndürür, boyut 2 olur.

8 **Bir ArrayList'i dolaşırken bazı elemanları silmek gerekiyor. Doğru yol hangisidir?**

A for-each döngüsünün içinde liste.remove(eleman) çağırmak.

B Iterator ile dolaşıp silinecek elemanda it.remove() çağırmak.

C for-each içinde silip hatayı try ve catch ile yakalamak.

D for-each içinde silip döngüyü olduğu gibi bırakmak; Java kayan elemanları kendisi düzeltir.

9 **Kisi sınıfı equals'ı telefon numarasıyla yeniden yazmış, hashCode'u yazmamış. Aynı ad ve telefonla oluşturulan iki Kisi nesnesi bir HashSet'e eklenince küme 2 diyor. Neden?**

A equals yanlış yazılmıştır; iki nesne için false döndürür.

B HashSet aynı elemanı iki kez tutmaya her zaman izin verir.

C Object'in hashCode'u iki nesneye farklı sayılar verir; küme farklı bölmelere bakar ve equals hiç çağrılmaz.

D HashSet elemanları yalnız == ile karşılaştırır, equals'ı hiç kullanmaz.

10 Aşağıdaki son satır ekrana ne yazar?

```
List<String> metinler = new ArrayList<>();
List<Integer> sayilar = new ArrayList<>();
System.out.println(metinler.getClass() == sayilar.getClass());
```

Cevap: _____

12

NESNE YÖNELİMLİ PROGRAMLAMA

Enum, iç sınıf, lambda

Bu bölümde değerleri baştan belli olan şeyler için enum'u, bir sınıfın içine yazılan sınıfları, adı olmayan sınıfları ve kısa bir işi tek satırda yazmayı sağlayan lambdayı göreceğiz. Bütün örnekler bir spor salonunun uygulamasından: haftalık program, üyelik türleri, şubeler ve seanslar. Bölümün sonunda bir listenin elemanlarını stream ile süzüp dönüştüreceğiz.

Günler sayıyla

videoda 1:30:01

Bir spor salonunun uygulaması haftalık seans programını tutuyor: yoga salı akşamı saat 18'de, yüzme cumartesi sabahı. Kodda günleri tutmanın en kolay yolu sayıdır: pazartesi 1, salı 2, pazar 7. Peki haftanın günlerini 1'den 7'ye sayıyla tutmak neden hataya açık?

Seans ekleyen metot iki tam sayı alıyor, gün ve saat:

```
static void seansEkle(int gun, int saat) {
    System.out.println(gun + ". gün, saat " + saat);
}
```

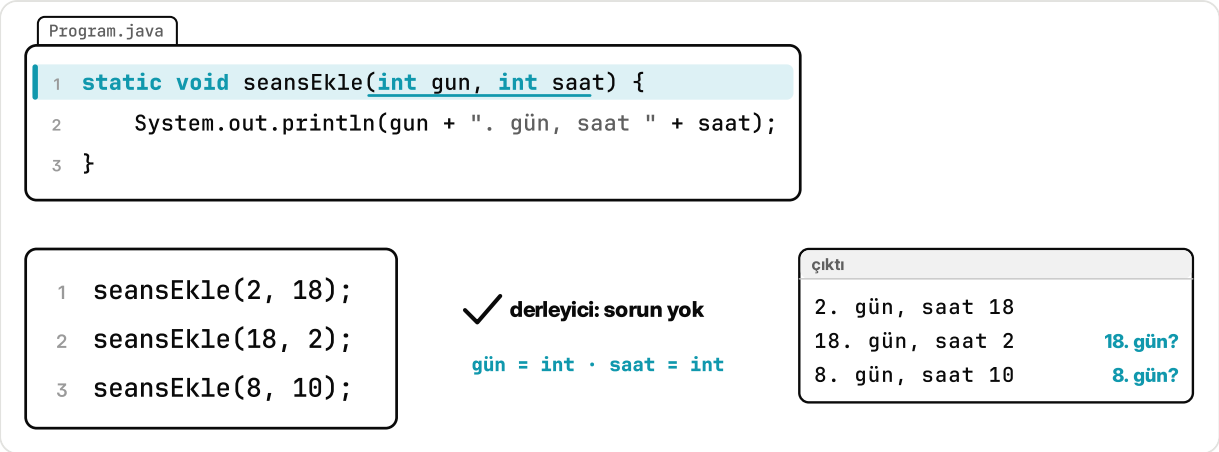
Salı saat 18 için 2 ve 18 yazarız. Ama sayıların yeri karışırsa metot 18. gün saat 2'ye seans ekler; 8. gün yazılırsa onu da kabul eder:

```
seansEkle(2, 18);
seansEkle(18, 2);
seansEkle(8, 10);
```

Çıktı:

```
2. gün, saat 18
18. gün, saat 2
8. gün, saat 10
```

Üç satır da sorunsuz derlenir ve çalışır, çünkü derleyici için gün de saat de yalnızca birer int'tir. Yanlışı ancak programı çalıştırıp çıktıya bakan biri fark eder.



Şekil 12.1 Gün ve saati int olarak alan seansEkle; yeri karışan ve 8. güne düşen çağrılar da derlenir, derleyici için ikisi de birer tam sayıdır.

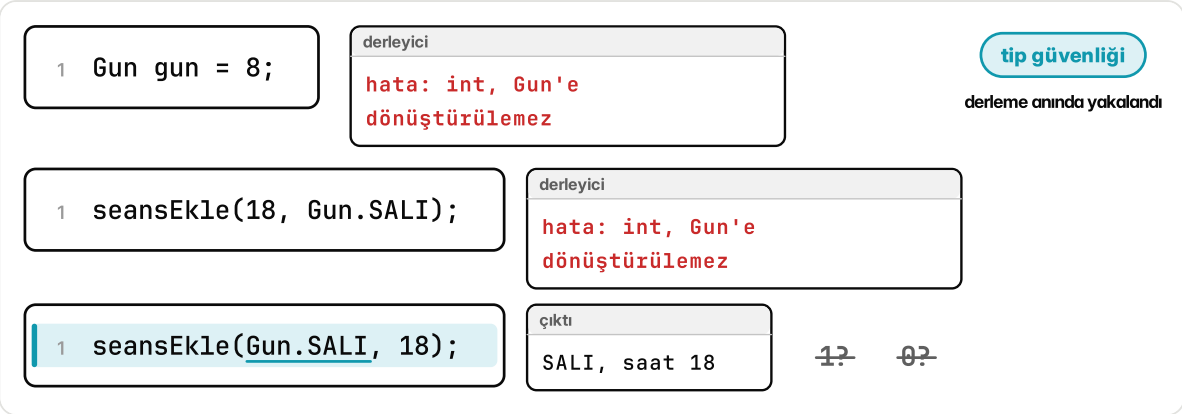
İkinci sorun, sayımın nereden başladığıdır. Bir yazılımcı pazartesiyi 1 sayar, öteki 0; bir başkası haftayı pazardan başlatır. Java'nın kendi eski takvim sınıfı Calendar'da bile pazar 1, pazartesi 2'dir; aylar ise 0'dan başlar, ocak 0. aydır: Calendar.SUNDAY 1, Calendar.MONDAY 2, Calendar.JANUARY 0 verir. Bu hatalar derlenirken değil, program çalışırken ortaya çıkar; belki aylar sonra, yanlış güne düşen bir seansla.

Çözüm, gün için yeni bir tip tanımlamaktır. **enum**, alabileceği değerleri adlarıyla, tek tek sayılmış bir tiptir:

```
public enum Gun {
    PAZARTESI, SALI, CARSAMBA, PERSEMBE,
    CUMA, CUMARTESI, PAZAR
}
```

Değerlerin her biri bir sabittir; bu yüzden büyük harfle yazılır. Gun tipindeki bir değişken bu yedi değerden birini tutar, başka hiçbir şeyi değil. Seans ekleyen metot da artık gün için bir sayı değil, bir Gun değeri ister: static void seansEkle(Gun gun, int saat).

Aynı hataları yeniden deneyelim. Gun gun = 8; satırı derlenmez; javac "incompatible types: int cannot be converted to Gun" der, yani bir tam sayı Gun tipine dönüştürülemez. Gün ile saatin yeri karışınca, seansEkle(18, Gun.SALI); satırında da aynı hata gelir. Doğru çağrı seansEkle(Gun.SALI, 18); ise derlenir ve SALI, saat 18 yazar. Yanlış değer, program daha çalışmadan, derleme anında yakalanır; buna **tip güvenliği** (type safety) denir. Sayımın başı da sorun olmaktan çıkar, çünkü kodda 1 ya da 0 yoktur, SALI vardır.



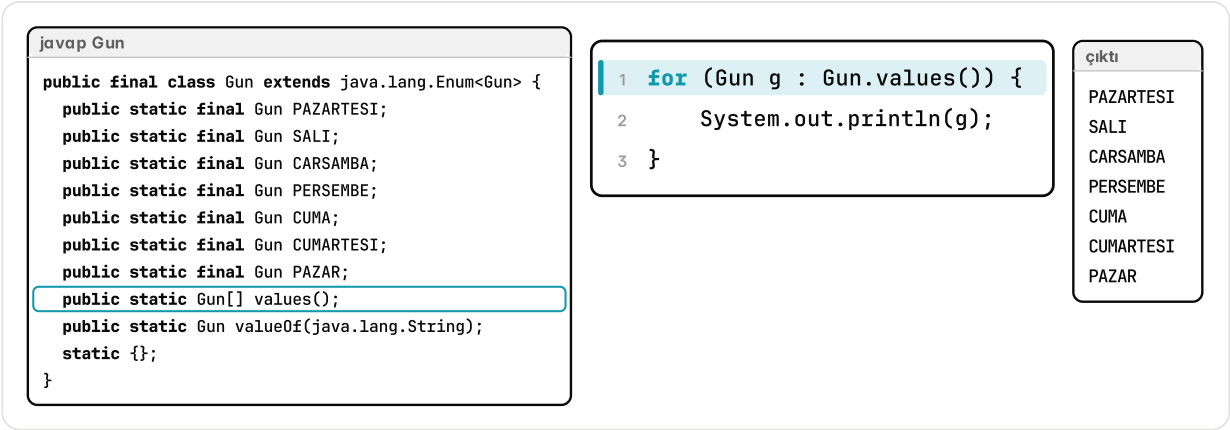
Şekil 12.2 Gun gun = 8; ve yeri karışan seansEkle(18, Gun.SALI); derlenmez; doğru sıradaki çağrı SALI, saat 18 yazar.

Perde arkasında enum bir sınıftır. Derlenmiş sınıfın üyelerini listeleyen javap aracıyla Gun'a bakınca public final class Gun extends java.lang.Enum<Gun> satırı ve yedi tane public static final Gun alanı görünür: derleyici her değeri o sınıfın sabit bir nesnesi olarak üretir. Her değer programda yalnız bir kez oluşturulur; bu yüzden iki SALI == ile güvenle karşılaştırılır, ikisi de aynı nesneyi gösterir. Enum'un values() metodu bütün değerleri tanımlandıkları sırayla bir dizi olarak verir; bir döngüyle haftanın yedi günü dolaşılır:

```
Gun bugun = Gun.SALI;
System.out.println(bugun == Gun.SALI);
for (Gun g : Gun.values()) {
    System.out.println(g);
}
```

Çıktı:

```
true
PAZARTESI
SALI
CARSAMBA
PERSEMBE
CUMA
CUMARTESI
PAZAR
```



Şekil 12.3 javap çıktısında Gun bir sınıftır ve her değer sabit bir nesnedir; values() üzerinde dönen döngü yedi günü tanımlandıkları sırayla yazar.

Java 8'le gelen yeni tarih kütüphanesi java.time da bu yolu seçti: orada haftanın günleri DayOfWeek, aylar Month adlı birer enum'dur. Değerleri baştan belli olan her şey, günler, aylar ya da bir üyeliğin türü, enum'la tutulur; yanlış değer daha derlenirken durur.

Fiyat nerede durmalı?

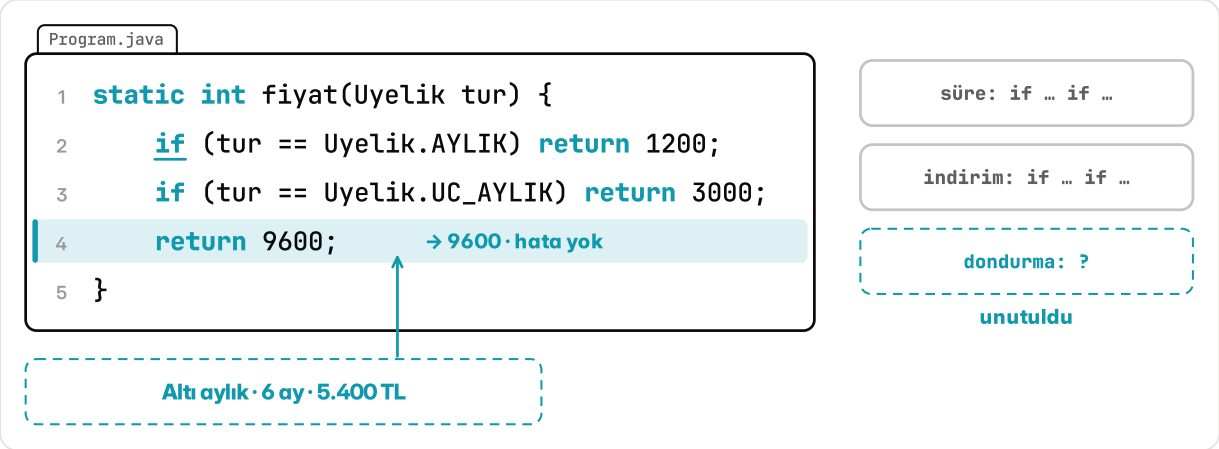
videoda 1:32:12

Spor salonunun üç üyelik türü var: aylık, üç aylık ve yıllık. Türleri Uyelik adlı bir enum'la tutuyoruz (AYLIK, UC_AYLIK, YILLIK); ama her türün bir süresi ve bir fiyatı da var: aylık 1.200 lira, üç aylık 3.000, yıllık 9.600 lira. Fiyat gereken her yerde "aylıksa şu kadar, üç aylıksa bu kadar" diye mi soracağız?

İlk akla gelen, if ile türü sormaktır. Fiyatı bulan metot türe bakar:

```
static int fiyat(Uyelik tur) {
    if (tur == Uyelik.AYLIK) return 1200;
    if (tur == Uyelik.UC_AYLIK) return 3000;
    return 9600;
}
```

Süre, indirim ve dondurma hakkı için de aynı sorular başka yerlerde yazılır. Altı aylık üyelik gelince bu yerlerin hepsi bulunup düzeltilmelidir; biri unutulursa altı aylık üyelik, hiçbir hata vermeden yıllığın fiyatını alır. Enum'a ALTI_AYLIK eklenip bu metot değiştirilmeyince fiyat(Uyelik.ALTI_AYLIK) 9600 döndürür.

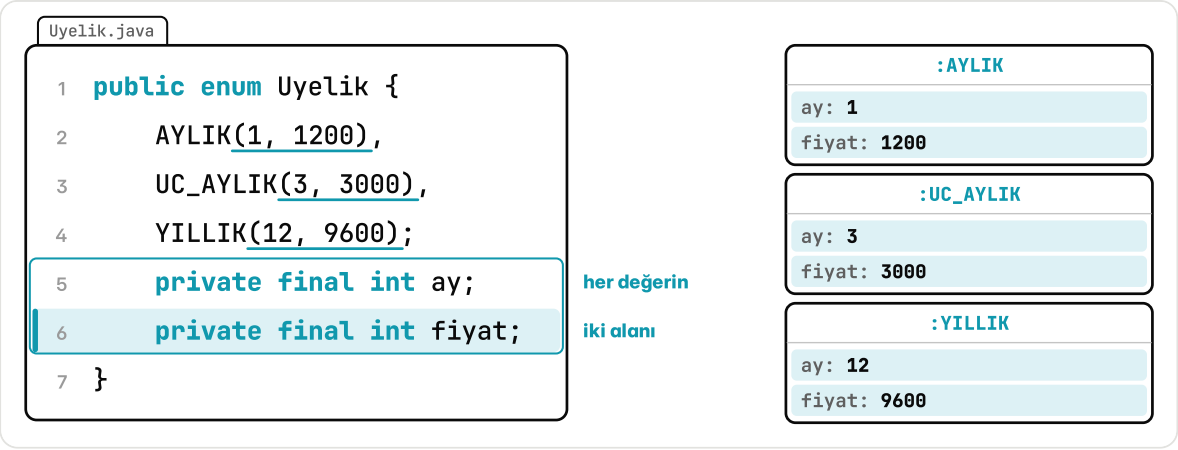


Şekil 12.4 if'li fiyat metodu; metoda yazılması unutulan altı aylık üyelik, hata vermeden yıllığın fiyatını (9600) alır.

Oysa enum'un değerleri birer nesnedir ve nesnenin alanları olur. Her değerin yanına, parantez içinde kendi bilgisi yazılır: aylık 1 ay ve 1.200 lira, üç aylık 3 ay ve 3.000, yıllık 12 ay ve 9.600. Bu sayılar her değerin ay ve fiyat adlı iki alanına yerleşir:

```
public enum Uyelik {
    AYLIK(1, 1200),
    UC_AYLIK(3, 3000),
    YILLIK(12, 9600);

    private final int ay;
    private final int fiyat;

    Uyelik(int ay, int fiyat) {
        this.ay = ay;
        this.fiyat = fiyat;
    }
```

Değer listesinin ardından başka üyeler geldiği için liste noktalı virgülle kapanır. Parantezdeki sayıları alanlara, enum'un yapıcı metodu yerleştirir. Ama bu yapıcı dışarıdan çağrılamaz: Uyelik altiAylik = new Uyelik(6, 5400); satırı derlenmez, javac "enum types may not be instantiated" der, yani enum'dan new ile nesne oluşturulamaz. Enum'un yapıcı metodu kendiliğinden private'tır. Üyelik türleri, enum'un içinde sayılan üç değerden ibarettir ve her biri yalnız bir kez oluşturulur.



Şekil 12.5 Uyelik'in her değeri parantezdeki sayılarla kendi ay ve fiyat alanlarını taşıyan bir nesnedir.

Enum'a metot da yazılır. Sınıfın geri kalanı aylikFiyat metodudur; fiyatı ay sayısına böler:

```
    public int aylikFiyat() {
        return fiyat / ay;
    }
}
```

Fiyatı soran kod artık türü sormaz, değerin kendisine sorar: u.aylikFiyat(). Altı aylık üyelik gelince enum'a tek bir değer eklenir, ALTI_AYLIK(6, 5400); öteki kod değişmez. Her türün aylık fiyatını yazdıran döngü şunu verir:

```
AYLIK: ayda 1200 TL
UC_AYLIK: ayda 1000 TL
YILLIK: ayda 800 TL
ALTI_AYLIK: ayda 900 TL
```

Yıllık üyelik ayda 800 liraya gelir (9.600 / 12), üç aylık 1.000'e, aylık 1.200'e; altı aylığın aylık fiyatı kendiliğinden 900 çıkar (5.400 / 6).

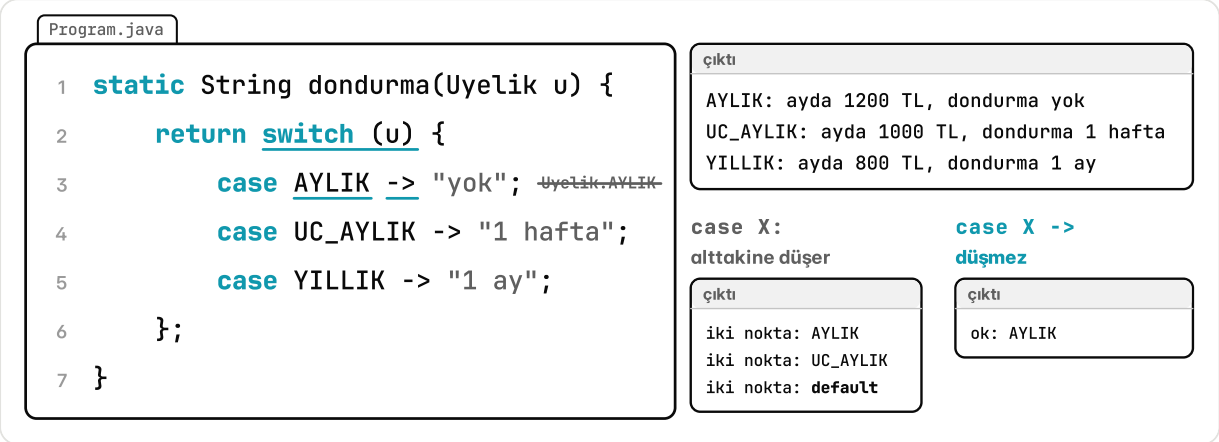
Bazen değere göre değişen şey bir sayı değil, yapılacak iştir; bunun için switch kullanılır. Dondurma hakkı aylıkta yok, üç aylıkta 1 hafta, yıllıkta 1 ay:

```
static String dondurma(Uyelik u) {
    return switch (u) {
        case AYLIK -> "yok";
        case UC_AYLIK -> "1 hafta";
        case YILLIK -> "1 ay";
    };
}
```

Çıktı:

```
AYLIK: ayda 1200 TL, dondurma yok
UC_AYLIK: ayda 1000 TL, dondurma 1 hafta
YILLIK: ayda 800 TL, dondurma 1 ay
```

case satırlarına enum'un adı yazılmaz, yalnız değerin adı yazılır: Uyelik.AYLIK değil, AYLIK. Ok işaretli (→) bu yazımda, C'deki gibi alttaki duruma düşme olmaz; her durum yalnız kendi satırını çalıştırır. Aynı değerle (AYLIK) iki noktalı eski yazım üç satır yazar, çünkü AYLIK'tan sonra alttaki UC_AYLIK ve default durumlarına da düşer; oklu yazım yalnız AYLIK'ın satırını yazar.



Şekil 12.6 Değer döndüren switch; case'lerde yalnız değerin adı yazılır, iki noktalı yazım alttaki durumlara düşer, oklu yazım düşmez.

Bu switch bir değer döndürüyor; değer döndürdüğü için de enum'un bütün değerlerini kapsamak zorundadır. ALTI_AYLIK enum'a eklenir de switch'e yazılmazsa kod derlenmez; javac "the switch expression does not cover all possible input values" der. Enum böylece bir ad listesinden fazlası olur: her değer kendi bilgisini taşır, bilgi tek yerde durur ve unutulan yeri derleyici gösterir.

Turnike hangi şubenin?

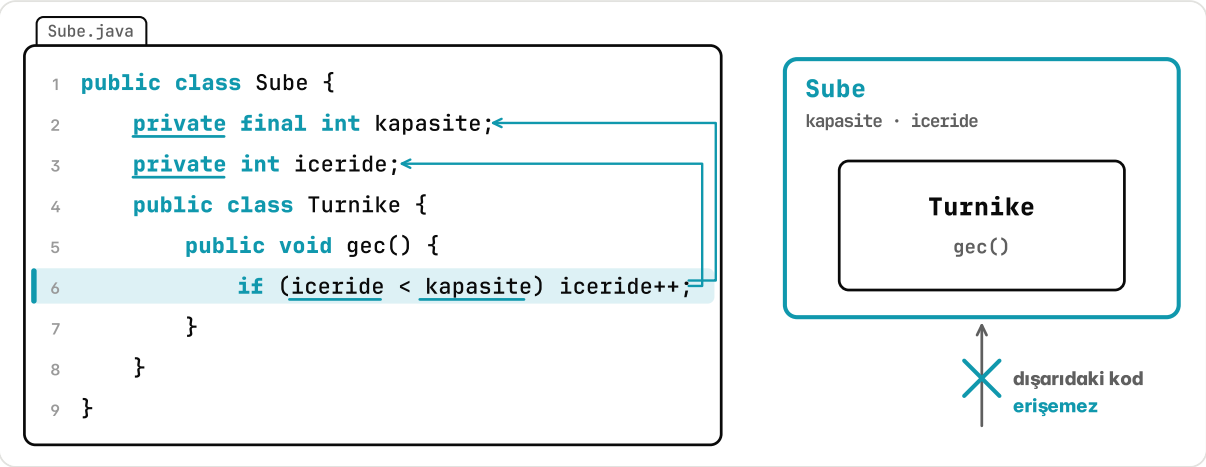
videoda 1:34:26

Bir spor salonu zincirinin her şubesinde birkaç turnike var. Üye geçince turnike, şubenin içerideki kişi sayısını bir artırır; şube dolduysa kapıyı açmaz. Bu sayı, şubenin dışarıya kapalı, private bir alanı. Peki turnike hangi şubenin sayacını artıracağını nereden bilir ve bu kapalı alana nasıl ulaşır?

Turnike sınıfını, Sube sınıfının içine yazarız; buna **iç sınıf** (inner class) denir:

```
public class Sube {
    private final int kapasite;
    private int iceride;
    public class Turnike {
        public void gec() {
            if (iceride < kapasite) iceride++;
        }
    }
}
```

İç sınıf, dıştaki sınıfın private alanlarını bile doğrudan kullanır: turnikenin gec metodu şubenin kapasite ve iceride alanlarına kendi alanlarıymış gibi ulaşır. Kapsülleme bozulmaz; turnike şubenin bir parçasıdır, şubenin dışındaki kod o alanlara yine ulaşamaz.



Şekil 12.7 Turnike, Sube'nin içinde yazılmış bir iç sınıftır; gec metodu şubenin private alanlarını doğrudan kullanır.

Her turnike nesnesi, onu oluşturan şube nesnesine gizli bir bağ taşır. Turnikeyi şube kendi metoduyla üretir:

```
public Turnike turnikeEkle() {
    return new Turnike();
}
```

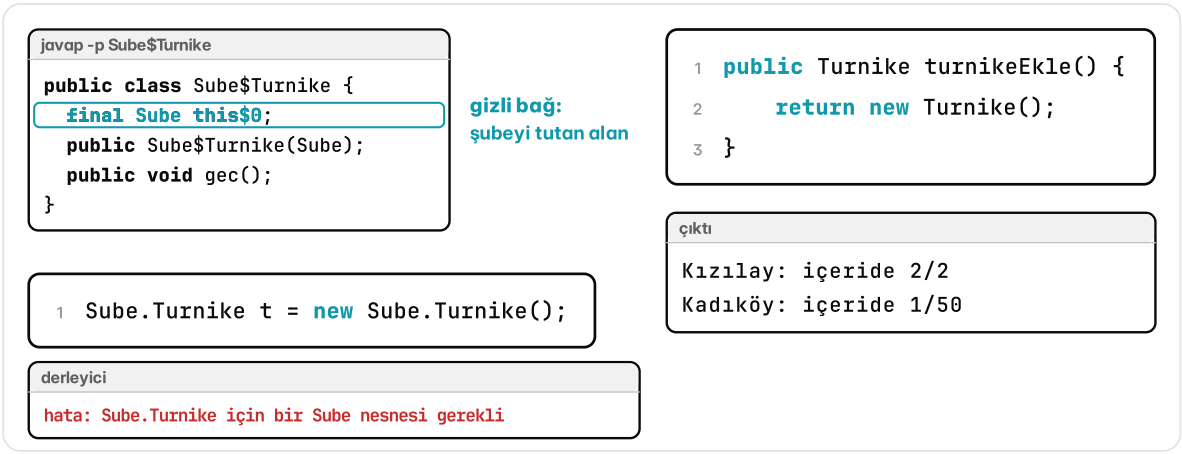
Denemede Kızılay şubesinin kapasitesi 2, Kadıköy'ünki 50. Kızılay'ın iki turnikesi t1 ve t2, Kadıköy'ün turnikesi t3; geçişler sırasıyla t1, t2, t1 ve t3'ten yapılıyor. Kızılay'ın iki turnikesi aynı sayacı artırır, Kadıköy'ünki kendi sayacını. Kızılay'da üçüncü geçişte içeride zaten 2 kişi vardır; sayı artmaz, kapı açılmaz. İki şube yazdırılınca:

```
Kızılay: içeride 2/2
Kadıköy: içeride 1/50
```

Derlenmiş turnike sınıfına javap -p ile bakınca bağ görünür:

```
public class Sube$Turnike {
  final Sube this$0;
  public Sube$Turnike(Sube);
  public void gec();
}
```

Derleyici iç sınıfa Sube$Turnike adını vermiş ve şubeyi tutan gizli bir alan (this$0) eklemiştir; yapıcı metodu da bir Sube alır. Bu yüzden iç sınıfın nesnesi bir dış nesne olmadan oluşturulamaz. main'den Sube.Turnike t = new Sube.Turnike(); yazılırsa javac "an enclosing instance that contains Sube.Turnike is required" der: turnike için bir şube nesnesi gerekir.



Şekil 12.8 javap çıktısında Sube$Turnike'nin şubeyi tutan gizli alanı (this$0); şube nesnesi olmadan new Sube.Turnike() derlenmez.

Bazen içteki sınıfın bir dış nesneye hiç ihtiyacı yoktur. Şubenin adresi semtten ve şehirden oluşur; içerideki kişi sayısını bilmesi gerekmez. Adres sınıfı da şubenin içine yazılır, ama başına static eklenir; buna **static iç sınıf** (static nested class) denir:

```
public class Sube {
    private int iceride;
    public static class Adres {
        private final String semt;
        private final String sehir;
    }
}
```

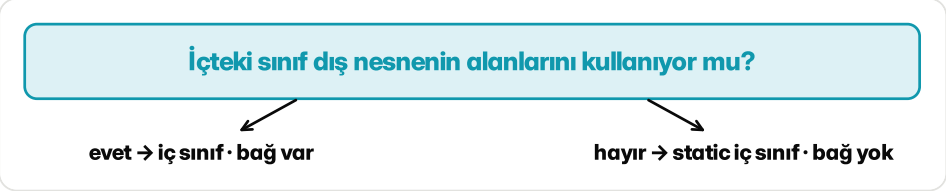
Adres'in yapıcı metodu ve toString metodu da dosyada vardır. static iç sınıfın nesnesi şube olmadan oluşturulur; adı da şubenin adıyla birlikte anılır: Sube.Adres adres = new Sube.Adres("Kadıköy", "İstanbul"); satırından sonra adres yazdırılınca Kadıköy, İstanbul çıkar.

static iç sınıfın gizli bağı yoktur; bu yüzden şubenin içerideki sayısına ulaşamaz:

```
public class Sube {
    private int iceride;
    public static class Adres {
        public int kalabalik() {
            return iceride;
        }
    }
}
```

Bu kod derlenmez; javac "non-static variable iceride cannot be referenced from a static context" der: static olmayan bir alan static bir yerden kullanılamaz. Soru zaten cevapsızdır: hangi şubenin sayısı? Adresin bağlı olduğu bir şube nesnesi yoktur.

Kural basit: içteki sınıf dış nesnenin alanlarını kullanıyorsa iç sınıf, kullanmıyorsa static iç sınıf. static olan daha hafiftir; her nesnesi dış nesneye fazladan bir bağ taşımaz. Java'nın kendi listeleri de böyle yazılmış. ArrayList'in elemanlarını sırayla dolaşan nesnenin sınıfı (Itr) bir iç sınıftır; hangi listeyi dolaştığını bu bağla bilir. LinkedList'in her düğümü (Node) ise static iç sınıftır; yalnız kendi değerini ve komşularını tutar, listenin kendisine bağlı değildir. Seçimi tek soru belirler: içteki sınıfın, dıştaki nesneye ihtiyacı var mı?



Şekil 12.9 Seçim kuralı: içteki sınıf dış nesnenin alanlarını kullanıyorsa iç sınıf (bağ var), kullanmıyorsa static iç sınıf (bağ yok).

Tek kullanımlık sıra

videoda 1:36:35

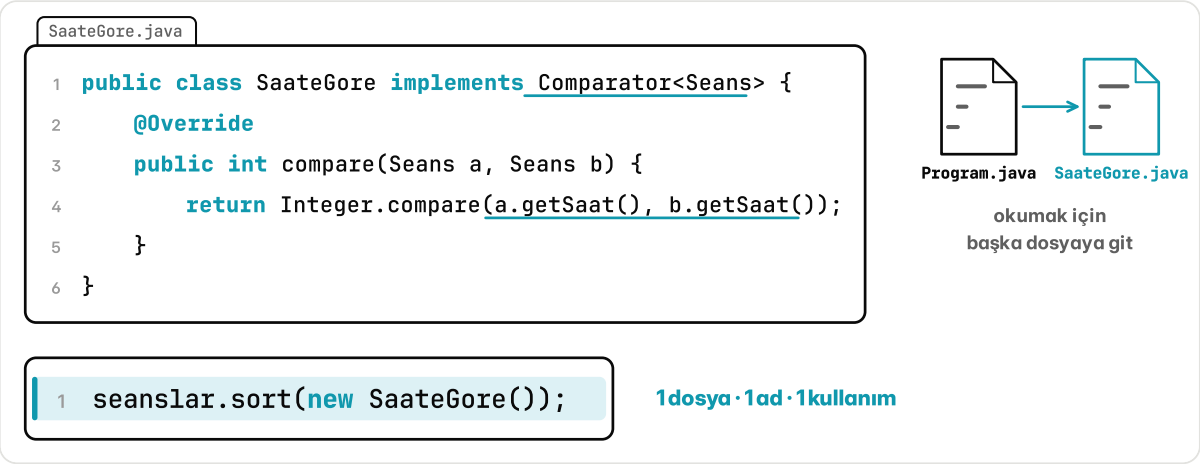
Spor salonunun uygulamasında dört seans var: Yoga saat 18.00'de, Pilates 9.00'da, Yüzme 7.00'de, Boks 20.00'de. Her seans Seans sınıfının bir nesnesidir; adını, saatini ve boş yer sayısını getAd, getSaat ve getBosYer metotlarıyla verir, yazdırılınca "Yoga 18.00" biçiminde görünür. Programı gösteren ekran seansları saate göre sıralıyor ve bu sıra yalnız orada, tek bir yerde kullanılıyor. Bunun için ayrı bir dosyada, adı olan bir sınıf yazmak şart mı?

Listenin sort metodu, iki seanstan hangisinin önce geleceğini söyleyen bir nesne ister; bu nesne Comparator arayüzünü gerçekleştirir. Bilinen yol ayrı bir sınıftır: SaateGore, iki seansın saatini karşılaştırır.

```
public class SaateGore implements Comparator<Seans> {
    @Override
    public int compare(Seans a, Seans b) {
        return Integer.compare(a.getSaat(), b.getSaat());
    }
}
```

Integer.compare iki sayıyı karşılaştırır: ilki küçükse eksi, eşitse 0, büyükse artı bir sayı döndürür; sort da buna bakarak sıralar. Kullanımı tek satırdır: seanslar.sort(new SaateGore());. Ama bu sınıfın kendi dosyası ve adı var, oysa programda yalnız bu satırda kullanılıyor; kodu okuyan, sıranın nasıl

belirlendiğini görmek için başka bir dosyaya gider.

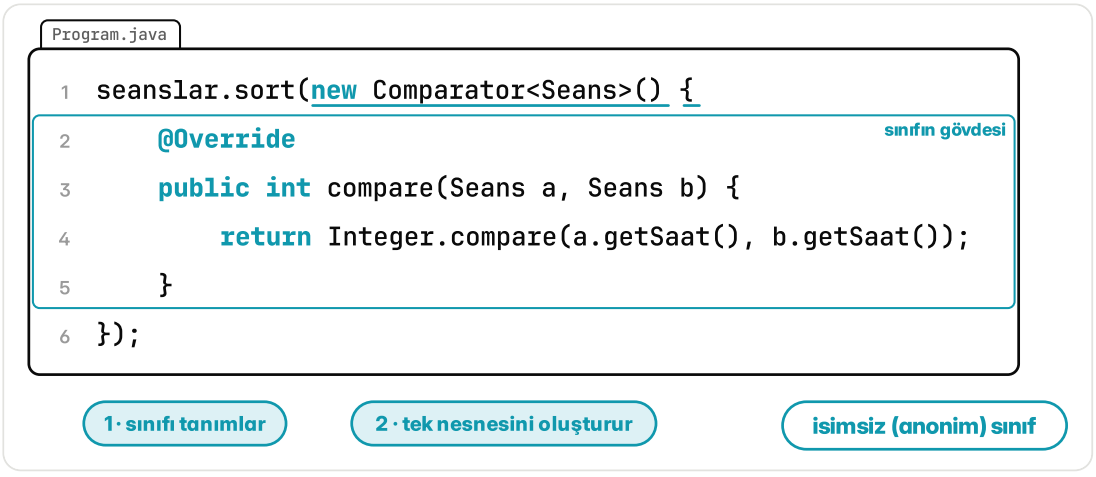


Şekil 12.10 Ayrı SaateGore sınıfı: kendi dosyası ve adı olan, programda tek satırda kullanılan bir Comparator.

Bunun yerine sınıfı, kullanıldığı yerde ve adını hiç koymadan yazabiliriz. sort'un parantezinde new, ardından arayüzün adı ve boş parantez gelir; hemen arkasından süslü parantez içinde sınıfın gövdesi, yani compare metodu:

```
seanslar.sort(new Comparator<Seans>() {
    @Override
    public int compare(Seans a, Seans b) {
        return Integer.compare(a.getSaat(), b.getSaat());
    }
});
```

Bu tek ifade iki iş yapar: sınıfı tanımlar ve o sınıfın tek nesnesini oluşturur. Buna **isimsiz sınıf** denir; anonim sınıf (anonymous class) adı da kullanılır.

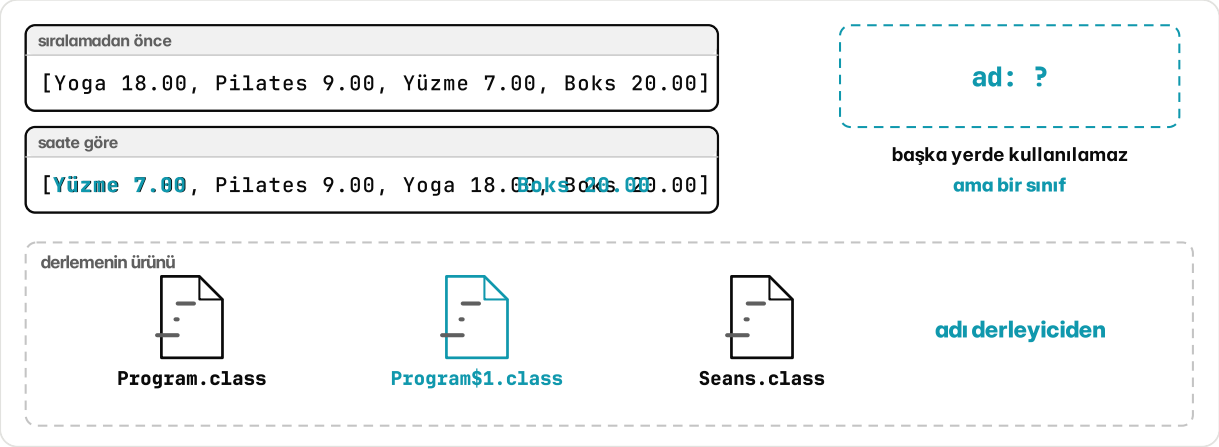


Şekil 12.11 sort'un parantezinde new Comparator<Seans>() ve ardından gelen gövde: sınıfı tanımlar ve tek nesnesini oluşturur.

Listeyi sıralamadan önce ve sonra yazdırınca:

```
[Yoga 18.00, Pilates 9.00, Yüzme 7.00, Boks 20.00]
[Yüzme 7.00, Pilates 9.00, Yoga 18.00, Boks 20.00]
```

Seanslar saat 7'deki yüzmeden saat 20'deki boksa dizildi. İsimsiz sınıfın adı olmadığı için başka bir yerde yeniden kullanılamaz; ama yine de bir sınıftır. Derleyici onun için ayrı bir sınıf dosyası üretir ve adını kendisi koyar: derleme klasöründe Program.class ve Seans.class'ın yanında Program$1.class da durur.



Şekil 12.12 Sıralamadan önceki ve sonraki liste; derleyici isimsiz sınıf için Program$1.class dosyasını üretir.

İsimsiz sınıf, bir arayüzü tek bir yerde ve kısa bir iş için gerçekleştirmek gerektiğinde kullanılır. Java'nın masaüstü programlarında bir düğmeye tıklanınca çalışacak kod da yıllarca böyle yazıldı: düğmeye, tıklamayı dinleyen ActionListener arayüzünü gerçekleştiren isimsiz bir nesne verildi. Swing kütüphanesinin kaydet adlı düğmesi için:

```
kaydet.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Kaydedildi");
    }
});
```

Düğmeye tıklanınca (denemede tıklama doClick metoduyla yapıldı) ekrana Kaydedildi yazılır. Kod kullanıldığı yerde durur; okuyan, ne yapıldığını aynı satırlarda görür.

Altı satırda tek satır iş

videoda 1:38:07

Seansları saate göre sıralayan isimsiz sınıf altı satır tutuyor. Oysa asıl işi yapan tek bir satır var: iki seansın saatini karşılaştıran return satırı. Geri kalan beş satır, yani arayüzün adı, metodun adı, tipler ve parantezler gerçekten gerekli mi?

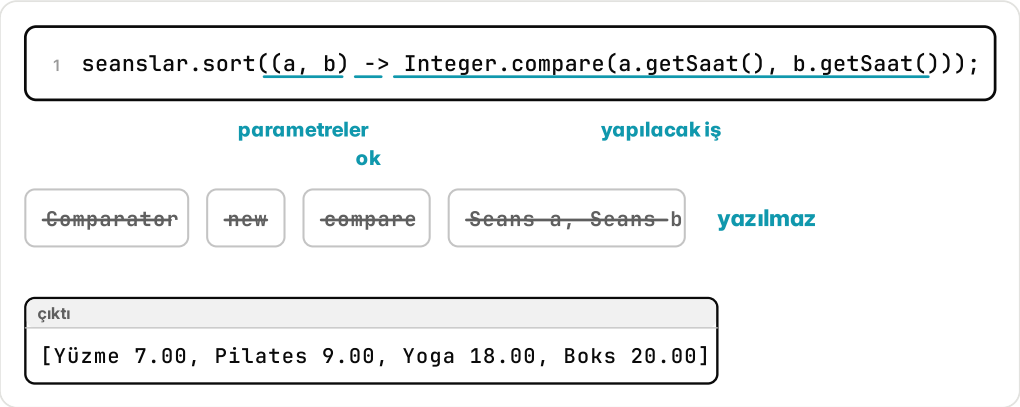
Java 8 bunun için yeni bir yazım getirdi: **lambda**. Parantez içinde parametreler, a ve b; ardından ok işareti (→) ve yapılacak iş, yani iki seansın saatini karşılaştırmak:

```
seanslar.sort((a, b) -> Integer.compare(a.getSaat(), b.getSaat()));
```

Çıktı:

```
[Yüzme 7.00, Pilates 9.00, Yoga 18.00, Boks 20.00]
```

Arayüzün adı, new, metodun adı ve parametrelerin tipleri yazılmaz; sıralama aynı sonucu verir.



Şekil 12.13 Lambda satırı üç parçadan oluşur: parametreler (a, b), ok ve yapılacak iş; Comparator, new, compare ve tipler yazılmaz.

Peki yazılmayanları derleyici nereden biliyor? sort, Comparator tipinde bir nesne bekliyor ve Comparator'ın tek bir soyut metodu var: compare. Lambda o metodun gövdesidir; a ile b'nin Seans olduğu da compare'in parametrelerinden çıkar. Tek soyut metodu olan arayüze **fonksiyonel arayüz** (functional interface) denir; lambda yalnız böyle bir arayüzün beklendiği yere yazılır. İki soyut metodu olan bir arayüze yazılırsa derlenmez:

```
public interface Karsilastirici {
    int karsilastir(Seans a, Seans b);
    String ad();
}
```

Karsilastirici k = (a, b) → Integer.compare(a.getSaat(), b.getSaat()); satırı için javac "Karsilastirici is not a functional interface" der ve gerekçesini ekler: arayüzde birden çok soyut metot bulundu. Derleyici gövdenin hangi metoda ait olduğunu bilemez.

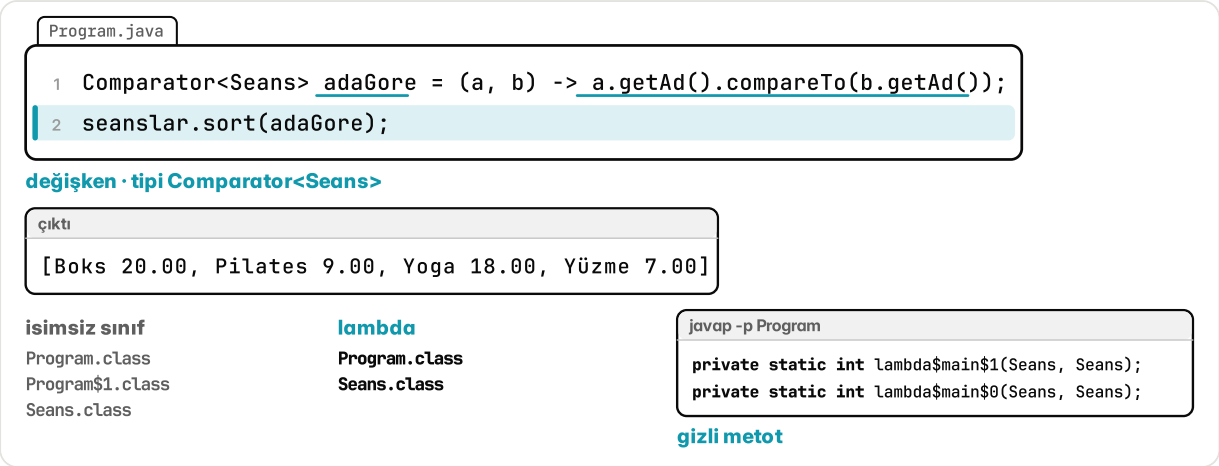
Lambda bir değerdir; bir değişkene de konabilir. adaGore adlı değişkende ikinci bir karşılaştırma durur, bu kez seansların adlarına göre. String'in compareTo metodu iki metni harf harf karşılaştırır:

```
Comparator<Seans> adaGore = (a, b) -> a.getAd().compareTo(b.getAd());
seanslar.sort(adaGore);
```

Çıktı:

```
[Boks 20.00, Pilates 9.00, Yoga 18.00, Yüzme 7.00]
```

İsimsiz sınıftan bir farkı da var: derleyici lambda için ayrı bir sınıf dosyası üretmez. İki lambdalı bu programın derlenmesinden yalnız Program.class ve Seans.class çıkar; javap -p Program ile bakınca gövdelerin programın kendi sınıfına lambda$main$0 ve lambda$main$1 adlı gizli (private) metotlar olarak eklendiği görülür.



Şekil 12.14 Değişkene konan adaGore ile ada göre sıralama; isimsiz sınıf Program$1.class üretir, lambda ise Program'a gizli bir metot olarak eklenir.

Lambdanın tek işi bir metodu çağırmaksa yazım daha da kısalır. Önce sınıfın adı, sonra iki kez iki nokta üst üste, sonra metodun adı: buna **metot referansı** (method reference) denir. Seans::getSaat, "her seans için saati getir" demenin kısa yoludur; s → s.getSaat() lambdasıyla aynı işi yapar. Comparator'ın hazır comparing metodu bu referansı alır ve saate göre karşılaştıran yeni bir Comparator üretir:

```
seanslar.sort(Comparator.comparing(Seans::getSaat));
```

Çıktı:

```
[Yüzme 7.00, Pilates 9.00, Yoga 18.00, Boks 20.00]
```

Aynı sıralama böylece üç yazımda görüldü: isimsiz sınıfla 6 satır, lambdayla tek satır, metot referansıyla daha da kısa. Her adımda iş aynıdır; azalan yalnız kalıp satırlardır. Geriye yalnız asıl soru kalır: seanslar neye göre sıralanacak?



Şekil 12.15 Aynı sıralamanın üç yazımı: isimsiz sınıf, lambda ve metot referansı; üçü de Yüzme'den Boks'a dizer.

Boş yeri olan seanslar

videoda 1:40:02

Spor salonunun dört seansından bazılarında boş yer kaldı: yogada 3, yüzmede 5; pilates ve boks dolu. Üyenin ekranında yalnız boş yeri olan seansların adları görünecek. Bunun için her seferinde döngü, koşul ve yeni bir liste mi yazmak gerekir?

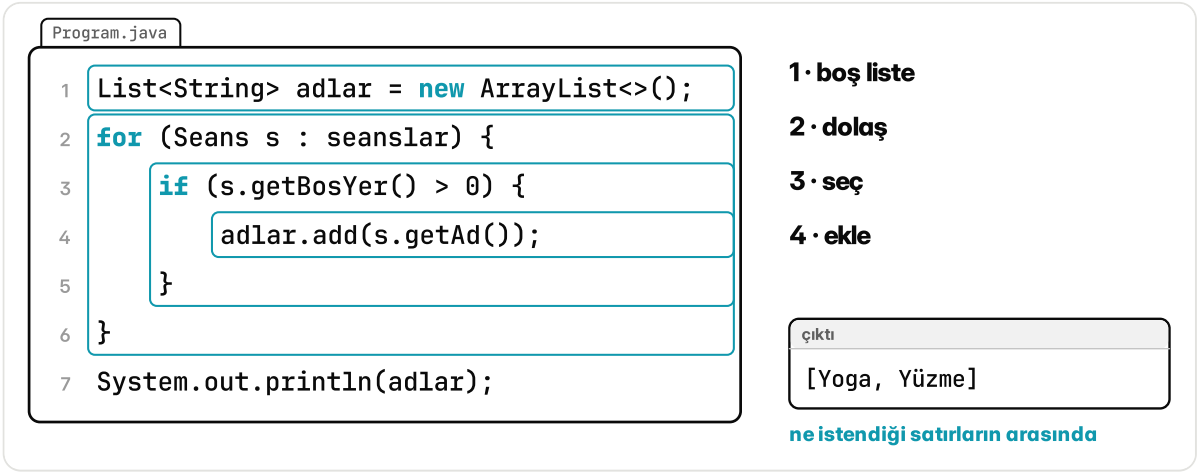
Alışık olduğumuz yol şöyle: boş bir ad listesi açılır, döngü seansları tek tek dolaşır, koşul boş yeri 0'dan büyük olanı seçer, seçilenin adı listeye eklenir:

```
List<String> adlar = new ArrayList<>();
for (Seans s : seanslar) {
    if (s.getBosYer() > 0) {
        adlar.add(s.getAd());
    }
}
System.out.println(adlar);
```

Çıktı:

```
[Yoga, Yüzme]
```

Sonuç doğru. Ama dört iş iç içe geçmiş; kodun ne istediğini anlamak için satırları tek tek okuyup, nasıl yapıldığının içinden ne yapıldığını çıkarmak gerekir.

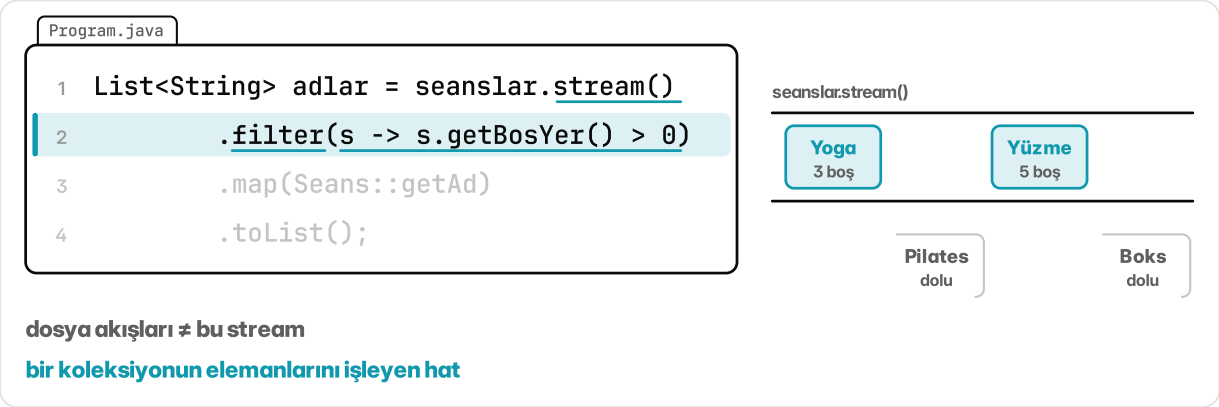


Şekil 12.16 Döngülü çözümde dört iş iç içe: boş liste, dolaşma, seçme ve ekleme; sonuç [Yoga, Yüzme].

Java 8'le bu işin yeni bir yolu geldi: **stream**. Liste, stream metoduyla elemanlarını sırayla akıtan bir hat açar. filter koşulu sağlayanları geçirir, öbürlerini durdurur; koşulu yine bir lambda verir: boş yeri 0'dan büyük mü? map, hattan geçen her elemanı başka bir şeye çevirir: seans girer, adı çıkar. Neye çevrileceğini bir metot referansı söyler, Seans::getAd. Hattın sonundaki toList akanları yeni bir listede toplar:

```
List<String> adlar = seanslar.stream()
        .filter(s -> s.getBosYer() > 0)
        .map(Seans::getAd)
        .toList();
```

Sonuç yine [Yoga, Yüzme]. Buradaki stream, dosyaya yazan akışlarla aynı adı taşısa da başka bir şeydir: bir koleksiyonun elemanlarını işleyen bir hattır. Stream Java 8'le geldi; sondaki toList metodu ise Java 16'da eklendi.



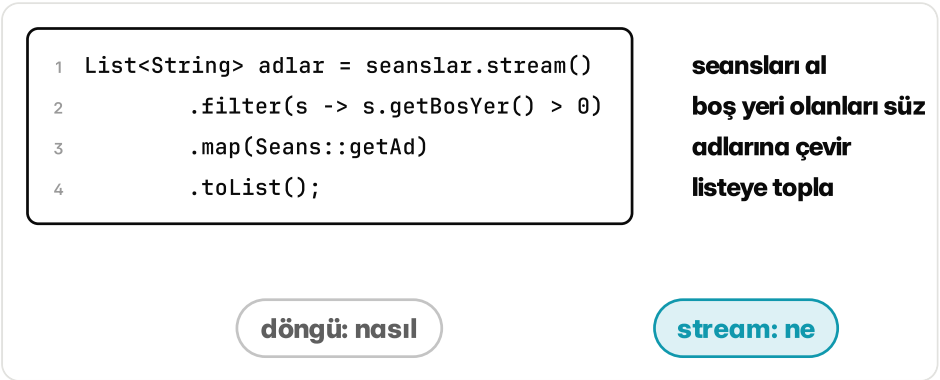
Şekil 12.17 stream bir hat açar; filter boş yeri olmayan Pilates ile Boks'u durdurur, Yoga ile Yüzme geçer.

Seansların asıl listesi hiç değişmez; hattan sonra yazdırılınca dört seans olduğu gibi durur: [Yoga 18.00, Pilates 9.00, Yüzme 7.00, Boks 20.00]. Aynı hat toplamak için de kurulur. mapToInt her seansın boş yerini bir tam sayıya çevirir, sum bunları toplar:

```
int toplam = seanslar.stream()
        .mapToInt(Seans::getBosYer)
        .sum();
```

toplam yazdırılınca Boş yer: 8 çıkar (3 + 0 + 5 + 0).

Kod artık bir cümle gibi okunur: seansları al, boş yeri olanları süz, adlarına çevir, listeye topla. Döngüde nasıl yapılacağını adım adım yazıyorduk; stream'de ne istediğimizi yazarız, adımları hat kendisi yürütür.



Şekil 12.18 Stream satırları cümle gibi okunur: seansları al, boş yeri olanları süz, adlarına çevir, listeye topla.

Özet

- enum, değerleri adlarıyla sayılmış bir tiptir; yanlış değer derleme anında yakalanır (tip güvenliği).
- Enum perde arkasında bir sınıftır: her değer bir kez oluşturulan sabit bir nesnedir, alanı, private yapıcısı ve metodu olabilir; değer döndüren switch bütün değerleri kapsamak zorundadır.
- İç sınıf dış nesneye gizli bir bağ taşır ve onun private alanlarını kullanır; static iç sınıfın böyle bir bağı yoktur.
- İsimsiz sınıf, bir arayüzü kullanıldığı yerde tanımlar ve tek nesnesini oluşturur; derleyici onun için Program$1.class gibi bir dosya üretir.
- Lambda, fonksiyonel arayüzün (tek soyut metotlu arayüz) beklendiği yere yazılan kısa gövdedir; metodu yalnız çağıran lambda, Seans::getSaat gibi bir metot referansıyla kısalır.
- stream, bir koleksiyonun elemanlarını filter, map ve toList gibi adımlardan geçiren hattır; asıl listeyi değiştirmez.

Sorular

1 **public enum Gun { PAZARTESI, SALI, CARSAMBA, PERSEMBE, CUMA, CUMARTESI, PAZAR } tanımlıyken Gun gun = 8; satırı için ne olur?**

A Derlenir; gun PAZARTESI olur, çünkü sayım baştan döner.

B Derlenir ama program çalışırken hata verir.

C Derlenmez; bir int, Gun tipine dönüştürülemez.

D Derlenir; gun 8. gün olarak saklanır.

2 **Enum değerlerini == ile karşılaştırmak neden güvenlidir?**

A Her enum değeri programda yalnız bir kez oluşturulan sabit bir nesnedir; iki SALI aynı nesneyi gösterir.

B == enum'larda içerik karşılaştırması yapacak biçimde değiştirilmiştir.

C Enum değerleri perde arkasında tam sayıdır.

D Derleyici enum'larda == yazılınca onu kendiliğinden equals'a çevirir.

3 **Uyelik enum'unun her değeri (ay, fiyat) alanlarını taşıyor ve aylikFiyat() metodu return fiyat / ay; satırından oluşuyor. Enum'a IKI_YILLIK(24, 16800) değeri eklenirse Uyelik.IKI_YILLIK.aylikFiyat() kaç döndürür?**

Cevap:

4 **Uyelik bir enum'dur. main'deki Uyelik altiAylik = new Uyelik(6, 5400); satırı neden derlenmez?**

A Uyelik'in yapıcı metodu iki değil, tek parametre alır.

B Enum'dan new ile nesne oluşturulamaz; yapıcı metodu kendiliğinden private'tır.

C Değişkenin adı büyük harfle yazılmalıdır.

D Enum'un ay ve fiyat alanları final olduğu için yeni değer alamaz.

5 **Aşağıdaki kod ne yazar?**

```
Uyelik u = Uyelik.UC_AYLIK;
switch (u) {
    case AYLIK -> System.out.println("A");
    case UC_AYLIK -> System.out.println("B");
    default -> System.out.println("C");
}
```

A Alt alta B ve C

B Alt alta A, B ve C

C C

D B

6 **Bölümdeki Sube sınıfında Turnike bir iç sınıftır ve gec() metodu if (iceride < kapasite) iceride++; satırını çalıştırır. Aşağıdaki kod ne yazar?**

```
Sube s = new Sube("Ulus", 3);
Sube.Turnike t1 = s.turnikeEkle();
Sube.Turnike t2 = s.turnikeEkle();
t1.gec();
t2.gec();
t1.gec();
t2.gec();
System.out.println(s);
```

Cevap:

7 **Bir sınıfın içinde yazılacak yardımcı sınıf, dış nesnenin hiçbir alanını kullanmıyor. Nasıl yazılması önerilir ve neden?**

A static iç sınıf olarak; nesneleri dış nesneye gizli bir bağ taşımaz ve dış nesne olmadan oluşturulabilir.

B İç sınıf olarak; dış sınıfın private alanlarına ancak böyle ulaşabilir.

C İsimsiz sınıf olarak; iç sınıflar yalnız arayüzden türetilebilir.

D İç sınıf olarak; static iç sınıfın nesnesi hiç oluşturulamaz.

8 **Program.java'nın main metodunda seanslar.sort(new Comparator<Seans>() { ... }); biçiminde tek bir isimsiz sınıf var. Derleme klasöründe Program.class'ın yanında hangi dosya da çıkar?**

A Comparator.class

B Anonim.class

C Program$1.class

D Hiçbiri; isimsiz sınıf için dosya üretilmez.

9 **Bir lambda, aşağıdaki arayüzlerden hangisinin tipinde bir değişkene atanabilir?**

A İki soyut metodu olan bir arayüz

B Tek soyut metodu olan bir arayüz

C Hiç soyut metodu olmayan bir arayüz

D Herhangi bir sınıf

10 **seanslar listesi sırasıyla Yoga (saat 18), Pilates (saat 9), Yüzme (saat 7) ve Boks (saat 20) seanslarını tutuyor. Aşağıdaki satır ne yazar?**

```
System.out.println(seanslar.stream().filter(s -> s.getSaat() <
10).map(Seans::getAd).toList());
```

Cevap: _____

NESNE YÖNELİMLİ PROGRAMLAMA

13 Paralel programlama

Bu bölümde bir programın birkaç işi aynı anda nasıl yürütebildiğini göreceğiz: pencerenin neden donduğunu, Java'da yeni bir iş parçacığının nasıl başlatıldığını, paylaşılan bir sayacın neden eksik saydığını ve kilitle nasıl korunduğunu, iki kilidin bir programı nasıl sonsuza kadar durdurabildiğini ve iş parçacığı havuzunu. Örnekler bir fotoğraf uygulamasından.

Donan pencere

videoda 1:41:30

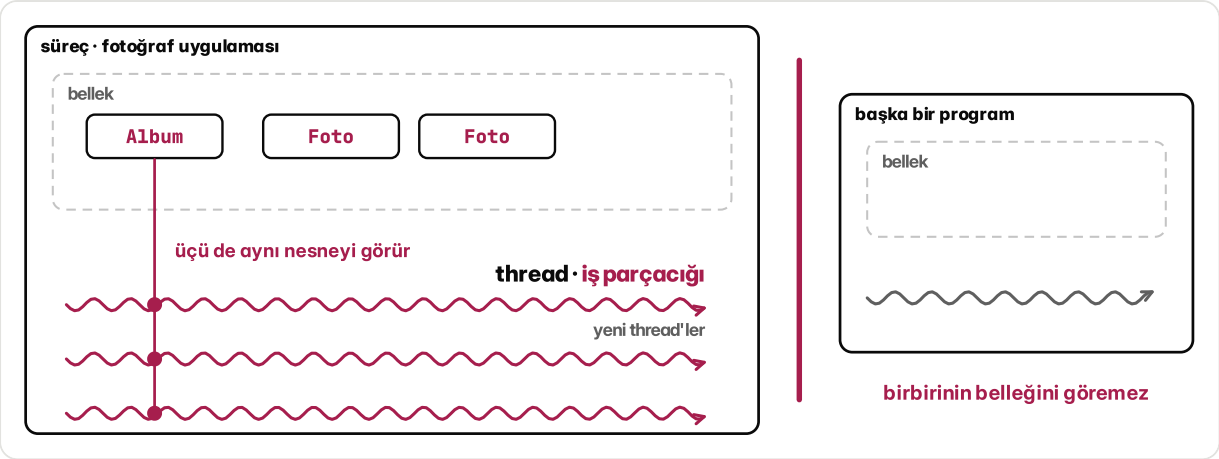
Fotoğraf uygulamasında 500 fotoğraf seçip Küçült düğmesine basıyoruz. İş bitene kadar pencere donuyor: düğmeler tepki vermiyor, pencere yerinden bile oynamıyor. Üstelik işlemci aynı anda 8 iş yürütebildiği hâlde yalnız biri çalışıyor, 7'si boşta; işlemcinin %12,5'i (8'de 1) kullanılıyor. Pencere neden donuyor, işlemcinin geri kalanı neden boşta?

Sebep, programın tek bir akışla çalışmasıdır. Pencereli bir programda tıklamaları karşılayan, ekranı çizen ve küçültme işini yapan aynı akıştır; satırları sırayla yürütür. Akış 500 fotoğrafla meşgulken sıradaki tıklamaya gelemez; tıklamalar sırada bekler ve pencere donmuş görünür. Tek akış bir anda yalnız tek bir iş olarak yürüyebildiği için işlemcinin öteki 7 iş yeri de boşta kalır. Telefonda bu bekleme 5 saniyeyi geçerse Android "uygulama yanıt vermiyor" penceresini açar.



Şekil 13.1 Tıklama, çizim ve 500 fotoğraflık küçültme aynı akışta sırayla yürür; tıklamalar sırada bekler, telefonda 5 saniyeden sonra "yanıt vermiyor" penceresi açılır.

Bu akışa **iş parçacığı** (thread) denir. Çalışan her programa ise **süreç** denir: işletim sistemi her sürece kendi belleğini verir ve süreçler birbirinin belleğini göremez. İş parçacığı, sürecin içinde satırları sırayla yürüten akıştır; her süreçte en az bir tane bulunur, gerekirse süreç yenilerini açar. Aynı süreçteki iş parçacıkları aynı belleği paylaşır: aynı nesneleri görür, aynı alanları değiştirebilir.



Şekil 13.2 Bir süreç, kendi belleği ve onu paylaşan iş parçacıkları; başka bir programın süreci bu belleği göremez.

Java'da en basit program da bir akışla çalışır: main metodu, main adlı iş parçacığında başlar. Thread.currentThread() o satırı yürüten iş parçacığını, getName() onun adını verir; Runtime.getRuntime().availableProcessors() ise işlemcinin aynı anda kaç iş yürütebildiğini söyler:

```
System.out.println(Thread.currentThread().getName());
System.out.println(Runtime.getRuntime().availableProcessors());
```

Çıktı:

```
main
8
```

Program çökünce yazılan hata dökümünün ilk satırında, Exception in thread "main" ifadesinde main sözünün geçmesi de bundandır. Bu bilgisayarın işlemcisinde 4 çekirdek var ve her çekirdek aynı anda 2 iş yürütebiliyor; 8 buradan gelir. Ama her şeyi main'de yapan program bu 8'den yalnız birini kullanır.

Çözüm, uzun işi yeni bir iş parçacığına vermektir. Pencereye bakan akış tıklamaları karşılamaya devam eder, küçültme arkada sürer; pencere donmaz. İşi dörde bölüp 4 ayrı iş parçacığına verirsek her birine 125 fotoğraf düşer; işletim sistemi onları 4 çekirdeğe dağıtır ve 4 fotoğraf gerçekten aynı anda küçülür. Buna **paralel programlama** denir: iş 4 çekirdeğe bölününce, en iyi durumda dörtte bir sürede biter. Dörtte bir, ulaşılabilecek en iyi sonuçtur; bölünen iş ancak bu kadar hızlanabilir.



Şekil 13.3 Küçültme ayrı iş parçacığında sürerken pencere donmaz; dört iş parçacığına bölünen iş dört çekirdekte aynı anda yürür.

Bugün telefonlarda bile 8 çekirdek var; tek akışla yazılan program işlemcinin büyük kısmını boş bırakır. Ama iş parçacıkları aynı belleği paylaştığı için iş burada bitmez: iki iş parçacığı aynı veriye aynı anda dokunduğunda ne olacağını da düşünmek zorundayız.

start mı, run mı?

videoda 1:43:32

Java'da uzun bir işi yeni bir iş parçacığına vermek istiyoruz. Thread nesnesinin bunun için iki metodu var: start ve run. İkisi de aynı işi çalıştırıyor; ama biri pencereyi dondurur, öteki dondurmaz. Hangisi hangisi, neden?

Önce yapılacak işi yazarız. Java'da iş parçacığına verilen iş, Runnable arayüzünü gerçekleştiren bir sınıftır. Bu arayüzün tek metodu var, run: iş parçacığı çalışınca yapılacak her şey onun içine yazılır. Kucult sınıfının run metodu şimdilik yalnız hangi iş parçacığında çalıştığını ekrana yazıyor:

```
class Kucult implements Runnable {
    public void run() {
        String ad = Thread.currentThread().getName();
        System.out.println("küçültülüyor: " + ad);
    }
}
```

Sonra işi, Thread sınıfından oluşturduğumuz nesneye veririz: yeni Thread oluştururken parantezin içine Kucult nesnesini koyarız. Yani iş başka, işi yürütecek olan başka: biri Kucult nesnesi, öteki Thread nesnesi. Önce run'ı doğrudan çağıralım:

```
Thread t = new Thread(new Kucult());
t.run();
System.out.println("düğmeler hazır");
```

Çıktı:

```
küçültülüyor: main
düğmeler hazır
```

İşi yürütenin adı main: iş, main'in kendisinde çalıştı. run'ı çağırmak sıradan bir metot çağrısıdır: yeni iş parçacığı açılmaz, main işin bitmesini bekler ve "düğmeler hazır" satırına ancak sonra gelir. Küçültme uzun sürseydi pencere yine donacaktı.



Şekil 13.4 t.run() işi main'in kendisinde çalıştırır: önce küçültülüyor: main, ardından düğmeler hazır yazılır.

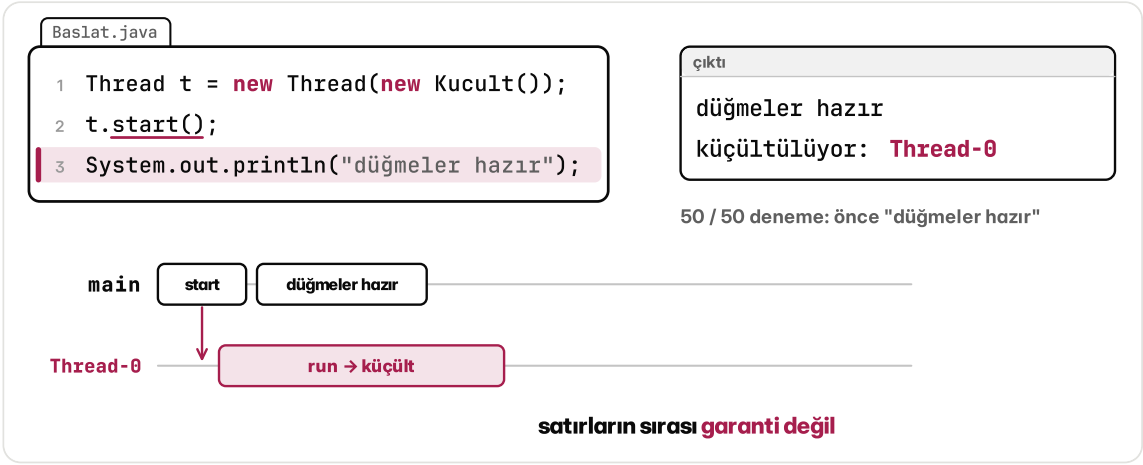
Şimdi ikinci satırda run yerine start'ı çağıralım. start yeni bir iş parçacığı açar, run'ı o iş parçacığında çalıştırır ve beklemeden döner:

```
Thread t = new Thread(new Kucult());
t.start();
System.out.println("düğmeler hazır");
```

Çıktı:

```
düğmeler hazır
küçültülüyor: Thread-0
```

İşi yürütenin adı değişti: Thread-0. Java, ad verilmeden oluşturulan iş parçacıklarına Thread-0, Thread-1 gibi adlar verir. main ise beklemeden sıradaki satıra geçti; bu bilgisayarda 50 denemenin 50'sinde de "düğmeler hazır" önce çıktı. Ama hangi satırın önce yazılacağının garantisi yoktur; sırayı işletim sistemi belirler.

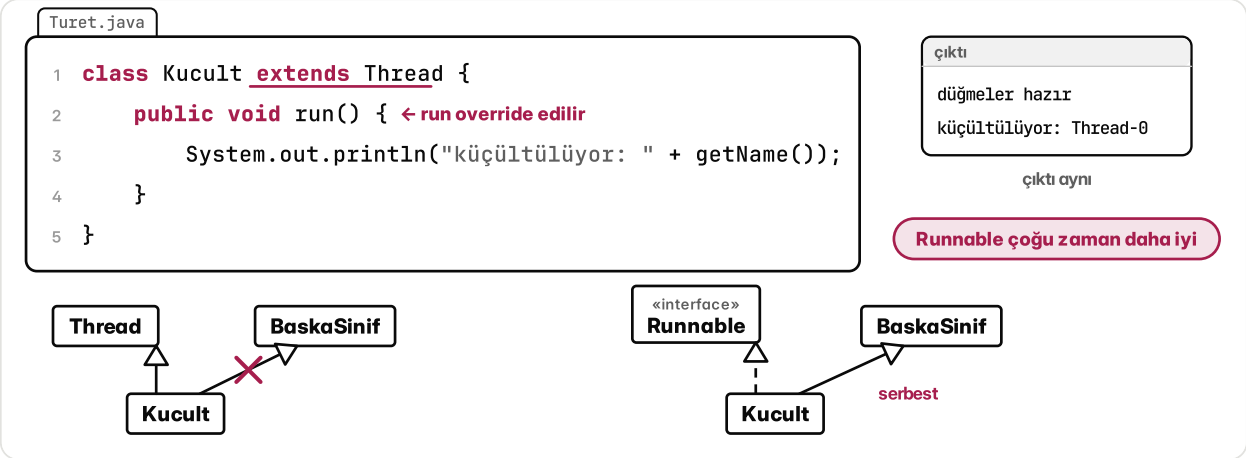


Şekil 13.5 t.start() yeni bir iş parçacığı (Thread-0) açar; main beklemeden devam eder ve iki satırın sırası garanti değildir.

İşi vermenin ikinci yolu, Thread sınıfından türeyip run metodunu override etmektir:

```
class Kucult extends Thread {
    public void run() {
        System.out.println("küçültülüyor: " + getName());
    }
}
```

Sınıf bir Thread olduğu için getName doğrudan çağrılır. main'de Kucult t = new Kucult(); ve t.start(); yazılır; çıktı aynıdır, bu bilgisayarda 20 denemenin hepsinde önce düğmeler hazır, sonra küçültülüyor: Thread-0. Bu da çalışır; ama Runnable çoğu zaman daha iyidir. Java'da tek kalıtım olduğundan, Thread'den türeyen sınıf başka bir sınıftan türeyemez; Runnable ise yalnız bir arayüzdür ve sınıfın elini bağlamaz.



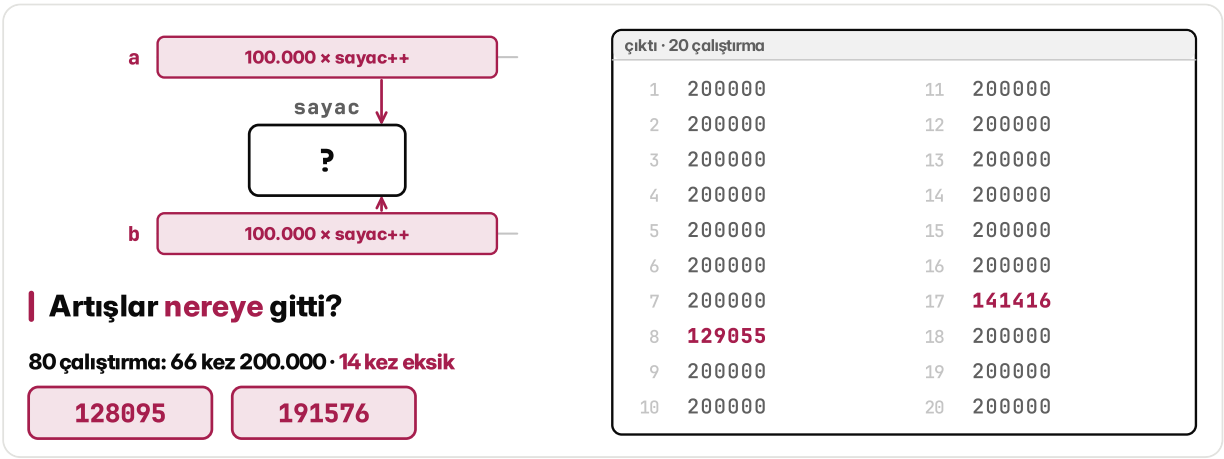
Şekil 13.6 Thread'den türeyen Kucult başka bir sınıftan türeyemez; Runnable'ı gerçekleştiren Kucult ise bir üst sınıftan türeyebilir.

Kısacası run işin kendisidir; start ise o işi yeni bir iş parçacığında başlatan çağrıdır. run'ı biz çağırmayız: start'ı çağırırız, run'ı ise yeni iş parçacığının içinden Java kendisi çağırır.

Eksik sayaç

videoda 1:45:17

İki iş parçacığı aynı sayacı 100.000'er kez artırıyor; sonuç 200.000 olmalı. Programı 80 kez çalıştırdık: 66'sında tam 200.000 çıktı; ama 14 denemede eksik, birinde 128.095, bir başkasında 191.576. Kod aynı, girdi aynı; artışlar nereye gitti?



Şekil 13.7 Çalıştırmaların çoğunda 200000, bazılarında eksik bir sayı çıkar; 80 çalıştırmada 66 kez 200.000, 14 kez eksik.

Sayaç, Artir sınıfının static alanıdır; iki Artir nesnesi de aynı alanı görür. İki iş parçacığının her biri run metodunda sayacı 100.000 kez bir artırır:

```
class Artir implements Runnable {
    static int sayac = 0;
    public void run() {
        for (int i = 0; i < 100000; i++) {
            sayac++;
        }
    }
}
```

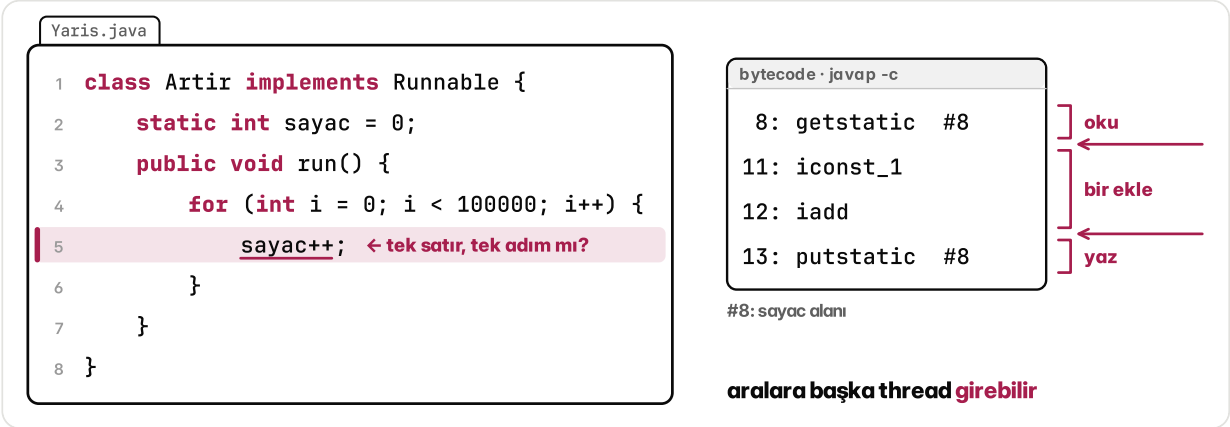
main iki iş parçacığını başlatır, join ile ikisinin de bitmesini bekler ve sayacı yazar. join, çağrıldığı iş parçacığı bitene kadar bekler; bu bekleme kesilebildiği için main'in başlığında throws InterruptedException yazılıdır:

```
Thread a = new Thread(new Artir());
Thread b = new Thread(new Artir());
a.start();
b.start();
a.join();
b.join();
System.out.println(Artir.sayac);
```

Sorun, sayac++ satırının tek adım olmamasıdır. Satırın derlenmiş hâline, yani derleyicinin ürettiği ve JVM'in yürüttüğü bytecode'a javap -c ile bakınca satır dört komuta ayrılır:

```
       8: getstatic     #8                  // Field sayac:I
      11: iconst_1
      12: iadd
      13: putstatic     #8                  // Field sayac:I
```

getstatic sayacın değerini bellekten okur, iconst_1 ile iadd bir ekler, putstatic sonucu belleğe geri yazar. Yani üç ayrı iş var: oku, bir ekle, yaz. Bu işlerin arasına başka bir iş parçacığı girebilir.

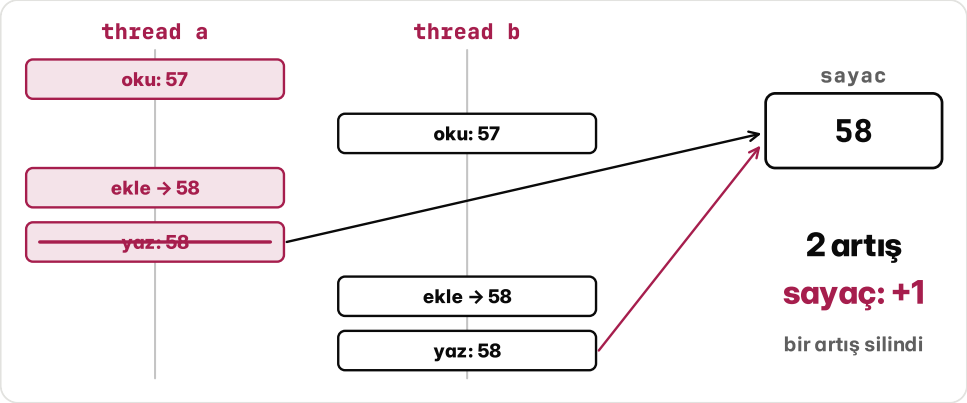


Şekil 13.8 sayac++ satırının bytecode'u: getstatic okur, iconst_1 ve iadd bir ekler, putstatic yazar; aralara başka bir iş parçacığı girebilir.

Sayaç 57'deyken iki iş parçacığı, a ve b, aynı anda gelsin (57 anlatım için seçilmiş bir değer):

- a 57'yi okur; o daha yazmadan b de 57'yi okur.
- a bir ekler ve 58 yazar.
- b de bir ekler ve o da 58 yazar.

İki artış yapıldı, sayaç yalnız bir arttı: birinin artışı ötekinin yazmasıyla silindi.



Şekil 13.9 İki iş parçacığı da 57'yi okur ve 58 yazar; iki artıştan sonra sayaç yalnız bir artmıştır.

Buna **yarış durumu** (race condition) denir: sonuç, iş parçacıklarının hangi sırayla çalıştığına bağlıdır ve bu sıra her çalıştırmada değişebilir. İş parçacıkları ayrı çekirdeklerde gerçekten aynı anda çalışabilir; tek çekirdekte bile işletim sistemi iş parçacıklarından birini okumakla yazmak arasında durdurup sırayı ötekine verebilir.

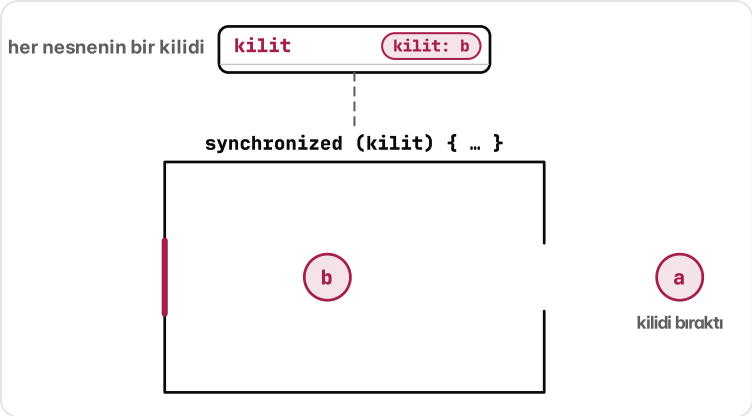
Bu hatanın en sinsi yanı saklanmasıdır. Aynı program sayacı 1.000'er kez artırınca 80 denemenin 76'sında doğru sonucu, 2000'i verdi: iş o kadar kısaydı ki çoğu zaman ikinci iş parçacığı başlayana kadar birinci işini bitiriyordu. Hata küçük denemede nadiren görünür, iş büyüdükçe sıklaşır; bu yüzden testten geçen kod canlıda yanlış sayabilir.

Araya kimse girmesin

videoda 1:46:52

İki iş parçacığı aynı sayacı 100.000'er kez artırıyor; sonuç çoğu zaman 200.000, ama arada bir eksik çıkıyor. Çünkü sayac++ aslında üç ayrı iş: oku, bir ekle, geri yaz; iş parçacıkları bu işlerin arasına girip birbirinin artışını siliyor. İş parçacıklarından biri üç işi bitirmeden ötekinin araya girmesini nasıl engelleriz?

Java'da her nesnenin bir kilidi vardır. synchronized bloğuna girmeden önce iş parçacığı, parantezdeki nesnenin kilidini ister. Kilit boştaysa alır ve girer; başkası tutuyorsa kilit bırakılana kadar kapıda bekler. Bloktan çıkınca iş parçacığı kilidi bırakır ve bekleyenlerden biri içeri girer.

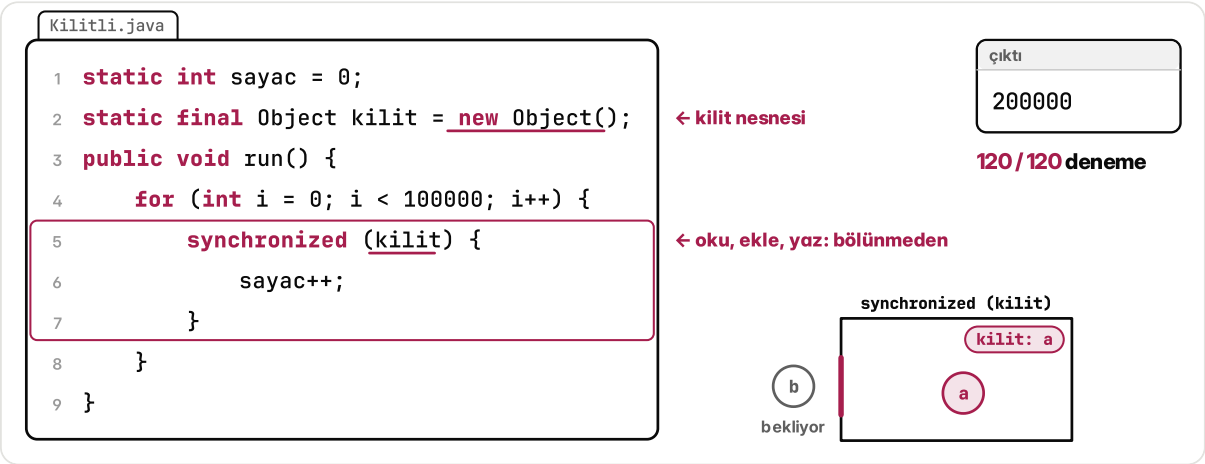


Şekil 13.10 synchronized bölgenin kapısında kilit: a kilidi alıp girer, b kapıda bekler; a çıkınca kilit b'ye geçer.

Kodda, kilit olarak kullanmak için ayrı bir nesne oluşturduk ve sayac++ satırını synchronized bloğuna aldık; parantezde bu kilit nesnesi duruyor. Artir sınıfının gövdesi şöyle (main değişmedi):

```
static int sayac = 0;
static final Object kilit = new Object();
public void run() {
    for (int i = 0; i < 100000; i++) {
        synchronized (kilit) {
            sayac++;
        }
    }
}
```

kilit de static'tir; iki Artir nesnesi aynı kilidi kullanır. Şimdi iş parçacıklarından biri okuyup yazarken öteki kapıda bekliyor; üç iş hiç bölünmeden yapılıyor. Program 120 kez çalıştı; hepsinde de sonuç tam 200000.



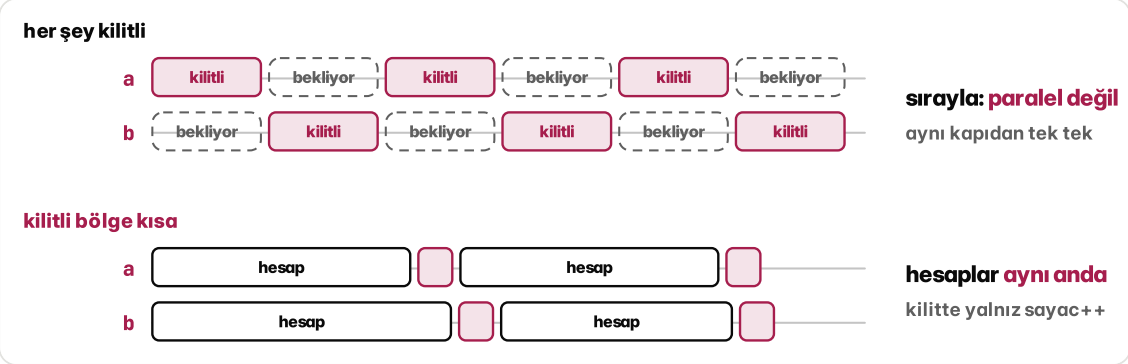
Şekil 13.11 Kilit nesnesiyle synchronized bloğa alınan sayac++; 120 denemenin hepsinde sonuç 200000.

Aynı işin kısa yolu metodun başına synchronized yazmaktır:

```
static synchronized void artir() {
    sayac++;
}
```

run'daki döngü artık sayac++ yerine artir() çağırır; bu sürüm de 20 denemenin hepsinde 200000 verdi. Böyle bir metotta kilit, nesne metodunda nesnenin kendi kilididir (this); static metotta ise sınıfın kilidi. artir static olduğu için iki iş parçacığı da aynı kilidi, Artir sınıfının kilidini kullanır. Bir iş parçacığı böyle bir metodu çalıştırırken ötekiler aynı kilidi kullanan metotların hiçbirine giremez.

Kilidin bir bedeli var: kilidi almak zaman alır, beklemedeki iş parçacığı da o sırada hiç iş yapmaz. Bu örnekte iki iş parçacığı aynı kapıdan sırayla geçiyor; iş paralel olmaktan çıkıyor. Bu yüzden kilitli bölge kısa tutulur: yalnız paylaşılan veriye dokunan satırlar kilidin içine alınır, uzun hesaplar dışarıda kalır ve iş parçacıkları onları aynı anda yürütür.



Şekil 13.12 Her şey kilitliyken iki iş parçacığı sırayla geçer; kilitli bölge kısa tutulunca hesaplar aynı anda yürür, kilitte yalnız sayac++ kalır.

Kısacası synchronized, paylaşılan veriye aynı anda iş parçacıklarından yalnız birinin dokunmasını sağlar. Sonuç her seferinde doğru çıkar; bedeli, iş parçacıklarının kapıda bekleyerek geçirdiği süredir.

Bitmeyen program

videoda 1:48:26

Program hata vermiyor, ekrana hiçbir şey yazmıyor, işlemci de boşta; ama hiç bitmiyor. Ne oldu?

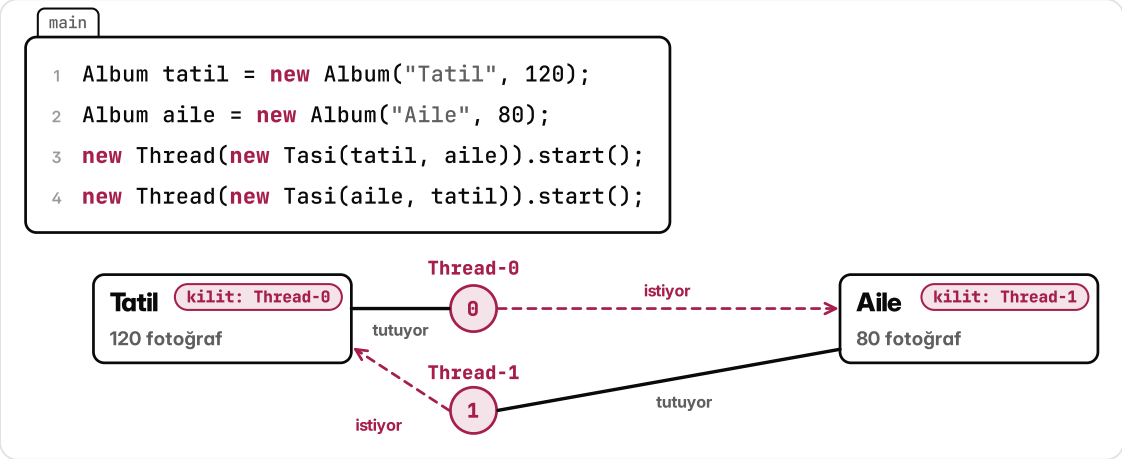
Fotoğraf uygulamamızda bir fotoğrafı bir albümden ötekine taşıyoruz. Album sınıfının iki alanı var: ad ve fotoğraf sayısı (foto). Taşırken iki albümü de kilitleriz: kaynaktan eksiltip hedefe eklerken araya kimse girmesin. Tasi sınıfının run metodu önce kaynak albümün kilidini alır, kısa bir süre bekler, sonra hedef albümün kilidini alır:

```
public void run() {
    synchronized (kaynak) {
        Bekle.ms(100);
        synchronized (hedef) {
            kaynak.foto--;
            hedef.foto++;
            System.out.println(kaynak.ad + " → " + hedef.ad + " taşındı");
        }
    }
}
```

Bekle.ms(100), iş parçacığını 100 milisaniye uyutan küçük bir yardımcıdır; içinde Thread.sleep çağrılır. Aynı anda iki taşıma başlıyor:

```
Album tatil = new Album("Tatil", 120);
Album aile = new Album("Aile", 80);
new Thread(new Tasi(tatil, aile)).start();
new Thread(new Tasi(aile, tatil)).start();
```

Birinci iş parçacığı Tatil'den Aile'ye taşıyor ve önce Tatil albümünü kilitliyor. İkinci iş parçacığı Aile'den Tatil'e taşıyor ve önce Aile albümünü kilitliyor. Sonra birinci, Aile albümünün kilidini istiyor; ama onu ikinci tutuyor. İkinci de Tatil'in kilidini istiyor; onu da birinci tutuyor.



Şekil 13.13 İki taşıma ters sırayla kilit alır: Thread-0 Tatil'i, Thread-1 Aile'yi tutar ve her biri ötekinin albümünün kapısında bekler.

İkisi de ötekinin bırakmasını bekliyor, hiçbiri de tuttuğunu bırakmıyor; çünkü bırakmak için önce işini bitirmesi gerek. Buna **kilitlenme** (deadlock) denir. Beklemedeki iş parçacığı işlemciyi kullanmaz ve hata da fırlatılmaz; program sonsuza kadar öylece durur. Java'yla gelen jstack aracı, çalışan programın iş parçacıklarına bakıp kilitlenmeyi bulur: hangi iş parçacığının hangi kilidi tutup hangisini beklediğini yazar. Dökümün kilitlenme bölümü (adresler kısaltıldı):

```
Found one Java-level deadlock:
"Thread-0":
  waiting to lock monitor … (object …, a Album),
  which is held by "Thread-1"
"Thread-1":
  waiting to lock monitor … (object …, a Album),
  which is held by "Thread-0"
```

Thread-0, Thread-1'in tuttuğu Album kilidini; Thread-1 de Thread-0'ın tuttuğu Album kilidini bekliyor. Dökümün sonunda Found 1 deadlock. satırı durur.

jstack <pid> · JDK ile gelen araç

```
Found one Java-level deadlock:
"Thread-0":
  waiting to lock monitor … (object …, a Album),
  which is held by "Thread-1"
"Thread-1":
  waiting to lock monitor … (object …, a Album),
  which is held by "Thread-0"
```

Thread-0, Thread-1'in tuttuğu Album kilidini bekliyor

Thread-1, Thread-0'ın tuttuğu Album kilidini bekliyor

Şekil 13.14 jstack dökümü: Thread-0 ve Thread-1, her biri ötekinin tuttuğu Album kilidini bekliyor.

Koddaki kısa bekleme, kilitlenmeyi her çalıştırmada gösterebilmek için konuldu. Gerçek programda bekleme olmaz; kilitlenme ancak iki taşıma tam aynı anlara denk gelirse olur. Beklemesiz hâli 30 denemenin yalnız birinde kilitlendi. Bu yüzden testte çoğu zaman görünmez, bir gün canlıda olur.

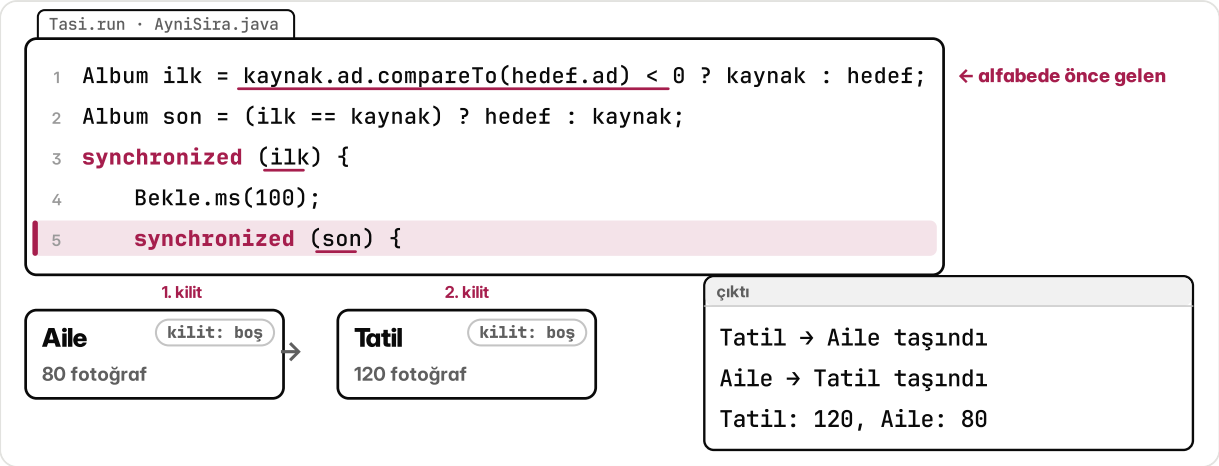
Çözüm, kilitleri herkesin aynı sırayla almasıdır. Burada kural şu: adı alfabede önce gelen albüm önce kilitlenir. run'ın ilk satırları değişir, gerisi aynı kalır:

```
Album ilk = kaynak.ad.compareTo(hedef.ad) < 0 ? kaynak : hedef;
Album son = (ilk == kaynak) ? hedef : kaynak;
synchronized (ilk) {
    Bekle.ms(100);
    synchronized (son) {
```

kaynak.ad.compareTo(hedef.ad) < 0, kaynağın adının alfabede önce gelip gelmediğini sorar; ? : işleci koşul doğruysa ilk, yanlışsa ikinci değeri seçer. "Aile", "Tatil"den önce geldiği için iki taşıma da önce Aile albümünü ister. Kilidi ilk alan iki kilidi de alıp taşır, öteki kapıda bekler; kilitler bırakılınca o da taşır ve program biter. main iki iş parçacığını join ile bekleyip albümleri yazınca:

```
Tatil → Aile taşındı
Aile → Tatil taşındı
Tatil: 120, Aile: 80
```

Program 20 kez çalıştı ve hepsinde bitti; ilk iki satırın sırası birinde tersti, son satır hep aynıydı. İki taşıma da yapıldı: Tatil'de yine 120, Aile'de 80 fotoğraf var.



Şekil 13.15 Kilitler alfabe sırasıyla alınır, önce Aile sonra Tatil; iki taşıma da yapılır, program biter ve albümler 120 ile 80'de kalır.

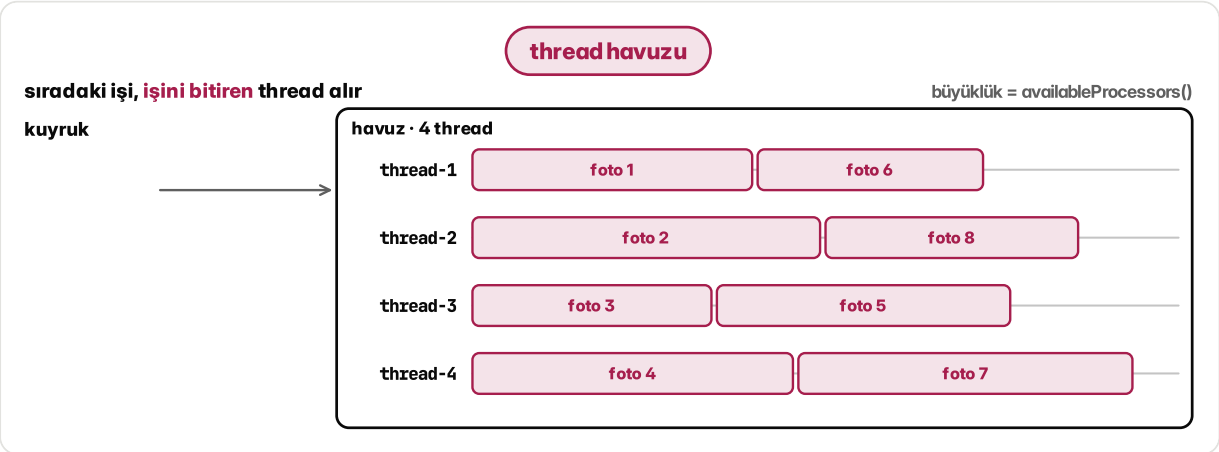
500 fotoğraf, 500 thread?

videoda 1:50:02

500 fotoğrafı küçültmek için her fotoğrafa ayrı bir iş parçacığı açmak akla ilk gelen yoldur. Neden kötü bir fikir?

İş parçacıklarının her birinin bir bedeli var: her biri için bellekte ayrı bir çağrı yığını ayrılır, iş parçacığını açıp kapatmak da zaman alır. Üstelik işlemci aynı anda ancak birkaç iş yürütebilir. 500 iş parçacığı açılırsa çekirdekler onların arasında sürekli geçiş yapar ve bu geçişler de zaman yer. Fotoğraf küçültmek gibi hesap işlerinde, çekirdeklerden fazla iş parçacığı açmak işi hızlandırmaz.

Bunun yerine **iş parçacığı havuzu** (thread pool) kullanılır: baştan belli sayıda iş parçacığı açılır ve bütün işleri onlar yapar. İşler bir kuyruğa girer; elindeki iş bitince iş parçacığı kuyruktaki sıradaki işi alır. Havuzun büyüklüğü, hesap işlerinde genellikle işlemcinin aynı anda yürütebildiği iş sayısı kadar, yani availableProcessors() kadar seçilir.



Şekil 13.16 Havuz: işler kuyrukta bekler, havuzdaki dört iş parçacığından işini bitiren kuyruğun başındaki işi alır.

Java'nın hazır havuzunu kullanıyoruz. Executors.newFixedThreadPool(4), içinde 4 iş parçacığı olan bir havuz oluşturur; ExecutorService bu havuzun tipidir. submit, verilen işi havuzun kuyruğuna ekler. Çıktı kısa olsun diye 8 fotoğraf verdik:

```
ExecutorService havuz = Executors.newFixedThreadPool(4);
for (int i = 1; i <= 8; i++) {
    havuz.submit(new Kucult(i));
}
havuz.shutdown();
havuz.awaitTermination(1, TimeUnit.MINUTES);
System.out.println("hepsi bitti");
```

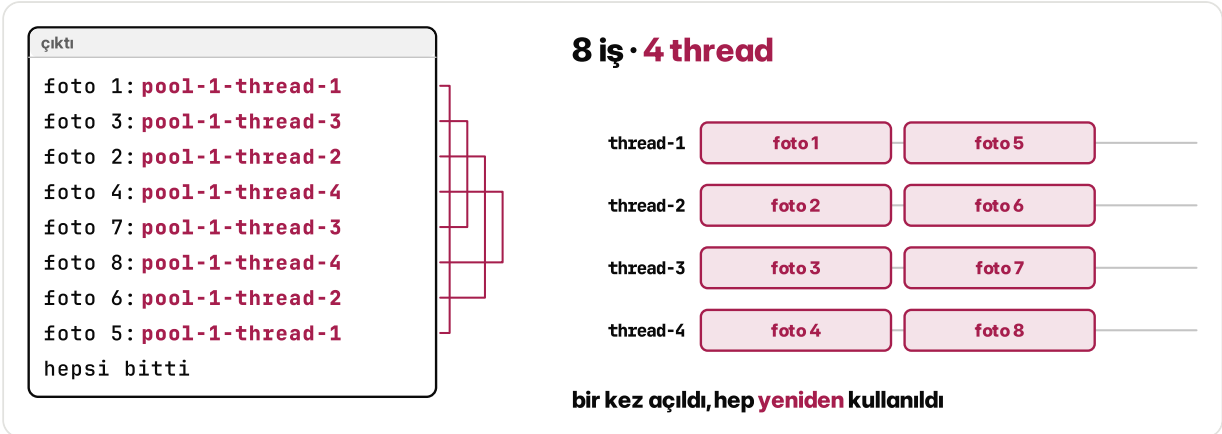
Kucult'un yapıcısı fotoğrafın numarasını no alanına koyar. Her iş 100 milisaniye bekleyip kendisini hangi iş parçacığının yürüttüğünü yazar:

```
public void run() {
    Bekle.ms(100);
    System.out.println("foto " + no + ": " + Thread.currentThread().getName());
}
```

Bir çalıştırmanın çıktısı (satırların sırası her çalıştırmada değişir):

```
foto 1: pool-1-thread-1
foto 3: pool-1-thread-3
foto 2: pool-1-thread-2
foto 4: pool-1-thread-4
foto 7: pool-1-thread-3
foto 8: pool-1-thread-4
foto 6: pool-1-thread-2
foto 5: pool-1-thread-1
hepsi bitti
```

Çıktıda 8 fotoğraf var ama iş parçacığı adı yalnız 4 tane; bu denemede her biri ikişer fotoğraf küçülttü, örneğin foto 1 ile foto 5'i pool-1-thread-1 yürüttü. İlk 4 fotoğraf havuzdaki iş parçacıklarına dağıldı; işini bitiren kuyruktan sıradakini aldı. İş parçacıkları bir kez açıldı ve bütün işlerde yeniden kullanıldı. Dağılım çalıştırmadan çalıştırmaya değişir: başka bir denemede pool-1-thread-4 tek başına 4 iş yaptı. Ama ad hep bu dördünden biridir.



Şekil 13.17 8 işin çıktısında yalnız 4 iş parçacığı adı var; aynı adı taşıyan satırlar aynı iş parçacığının yürüttüğü işlerdir.

İşler verilince havuzu kapatırız. shutdown'dan sonra havuz yeni iş almaz, eldeki işleri bitirir ve iş parçacıkları kapanır. awaitTermination işlerin bitmesini en çok verilen süre kadar, burada 1 dakika bekler; main de son işin bitmesini bekleyip hepsi bitti yazar. Kapatmayı unutursak havuzdaki iş parçacıkları yeni iş beklemeye devam eder ve main bitse bile program kapanmaz. Havuzun kapatılmadığı bir denemede ekrana önce main bitti, ardından fotoğraf satırları yazıldı ve program çalışmayı sürdürdü.

Havuz.java

```
1 ExecutorService havuz = Executors.newFixedThreadPool(4);
2 for (int i = 1; i <= 8; i++) {
3     havuz.submit(new Kucult(i));
4 }
5 havuz.shutdown();
6 havuz.awaitTermination(1, TimeUnit.MINUTES);
7 System.out.println("hepsi bitti");
```

çıktı · Kapatmasiz.java

```
main bitti
foto 1
foto 4
foto 5
foto 6
foto 7
foto 8
foto 3
foto 2
```

program: hâlâ çalışıyor

Şekil 13.18 shutdown yeni işi keser, awaitTermination işlerin bitmesini bekler; havuz kapatılmazsa main bittiği hâlde program çalışmayı sürdürür.

Java'yla yazılmış sunucular da gelen istekleri böyle karşılar: her istek için yeni iş parçacığı açılmaz; havuzdaki iş parçacıklarından biri isteği alır, cevabı gönderir ve sıradakine geçer.

Özet

- İş parçacığı (thread), bir sürecin içinde satırları sırayla yürüten akıştır; aynı süreçteki iş parçacıkları aynı belleği paylaşır, Java'da main de bir iş parçacığıdır.
- İş, Runnable'ın run metoduna yazılır; start yeni bir iş parçacığı açıp run'ı orada çalıştırır, run'ı doğrudan çağırmak ise sıradan bir metot çağrısıdır.
- sayac++ oku, ekle, yaz diye üç işe ayrılır; iki iş parçacığı araya girince artışlar kaybolur ve sonuç çalıştırma sırasına bağlı olur (yarış durumu).
- synchronized, bir nesnenin kilidiyle paylaşılan veriye aynı anda tek iş parçacığının dokunmasını sağlar; bedeli bekleme olduğu için kilitli bölge kısa tutulur.
- İki iş parçacığı birbirinin tuttuğu kilidi beklerse program hatasız ve sonsuza kadar durur (kilitlenme); kilitleri herkesin aynı sırayla alması bunu önler.
- İş parçacığı havuzu az sayıda iş parçacığını kuyruktaki işlerde yeniden kullanır; işler verilince havuz shutdown ile kapatılır.

Sorular

1 **Aynı süreçteki iş parçacıkları (thread) için hangisi doğrudur?**

A Her biri işletim sisteminden ayrı bir bellek alır ve ötekilerin nesnelerini göremez.

B Hepsi aynı anda yalnız tek bir çekirdekte çalışabilir.

C Aynı belleği paylaşır; aynı nesneleri görür ve aynı alanları değiştirebilir.

D Biri çalışırken öteki kendiliğinden bekler.

2 **Hiç iş parçacığı açmayan bir programın main metodunda `System.out.println(Thread.currentThread().getName());` satırı ne yazar?**

Cevap:

3 **Kucult sınıfı Runnable'ı gerçekleştiriyor ve run metodu "küçültülüyor: " ile birlikte o anki iş parçacığının adını yazıyor. Aşağıdaki kodun çıktısının ilk satırı nedir?**

```
Thread t = new Thread(new Kucult());
t.run();
System.out.println("düğmeler hazır");
```

A küçültülüyor: Thread-0

B küçültülüyor: main

C düğmeler hazır

D Kod derlenmez; run doğrudan çağrılamaz.

4 **İki iş parçacığı `static int sayac` alanını `sayac++` ile artırıyor. Bu satır neden güvenli değildir?**

A sayac++ değeri oku, bir ekle, geri yaz diye üç işe ayrılır ve bu işlerin arasına öteki iş parçacığı girebilir.

B static alanlar iş parçacıkları arasında paylaşılmaz; her iş parçacığı kendi kopyasını artırır.

C ++ işleci yalnız tek iş parçacıklı programlarda derlenir.

D int, 100.000'den büyük sayıları tutamaz.

5 **Sayaç 40'ken iki iş parçacığı sırasıyla şunları yapıyor: a 40'ı okur; b 40'ı okur; a bir ekleyip 41 yazar; b bir ekleyip 41 yazar. İki artıştan sonra sayaç kaçtır?**

Cevap:

6 **İki iş parçacığı, static ve ortak bir kilit nesnesiyle synchronized (kilit) { sayac++; } bloğunu 100.000'er kez çalıştırıyor. Program iki iş parçacığını join ile bekleyip sayacı yazıyor. Çıktı için hangisi doğrudur?**

A Çoğu zaman 200000, arada bir daha küçük bir sayı

B Her çalıştırmada 100000

C Program kilitlenir ve hiç bitmez.

D Her çalıştırmada 200000

7 **Birinci iş parçacığı önce Tatil'in, sonra Aile'nin kilidini; ikinci iş parçacığı önce Aile'nin, sonra Tatil'in kilidini istiyor. İkisi de ilk kilidini aldıktan sonra ne olabilir?**

A JVM bir istisna fırlatır ve program hatayla biter.

B İkisi de ötekinin tuttuğu kilidi bekler; program hata vermeden sonsuza kadar durur.

C İşlemci %100 çalışarak kilitlerin açılmasını sağlar.

D Kilitler 100 milisaniye sonra kendiliğinden bırakılır.

8 **İki albüm arasında iki yönde taşıma yapan iş parçacıklarının kilitlenmesini bölümdeki çözüm nasıl önler?**

A Kilitleri herkes aynı sırayla alır: adı alfabede önce gelen albüm önce kilitlenir.

B İki synchronized bloğu da kaldırılır.

C Bekle.ms(100) satırı 1.000 milisaniyeye çıkarılır.

D İkinci iş parçacığı birincinin join'iyle başlatılır.

9 **Executors.newFixedThreadPool(4) ile oluşturulan havuza işler verildi ama havuz.shutdown() çağrılmadı. main'in son satırı çalıştıktan sonra ne olur?**

A Program hemen kapanır, verilmiş işler yarıda kalır.

B Havuz yeni işleri reddeder ve bir istisna fırlatır.

C İşler çalışır ama havuzdaki iş parçacıkları yeni iş beklemeyi sürdürür; program kapanmaz.

D Havuz kendiliğinden kapanır, çünkü main bitmiştir.

10 **Executors.newFixedThreadPool(3) ile oluşturulan havuza 12 iş veriliyor ve her iş kendisini yürüten iş parçacığının adını yazıyor. Çıktıda en çok kaç farklı iş parçacığı adı görünebilir?**

Cevap: _____

Cevaplar ve çözümler

0 Neden nesne yönelimli?

1. **B** · Durum ve davranış
   Bakiye nesnenin o anki bilgisidir, yani durumudur. Para çekme nesnenin yapabildiği bir iştir, yani davranışıdır; Java'da bu işlemlere metot denir.

2. **A** · Hesap bir kalıptır; ayse o kalıptan üretilmiş, bakiyesi kendine ait bir nesnedir.
   Sınıf bütün hesapların kalıbıdır; new bu kalıptan yeni bir nesne üretir. Aynı kalıptan çıkan nesnelerin her birinin bakiyesi kendine aittir: Ayşe'nin çektiği para Mehmet'in bakiyesini değiştirmez.

3. **350**
   Yeni hesabın bakiyesi 0'dır. 500 yatınca 500, 200 çekilince 300, 50 yatınca 350 olur.

4. **2,5**
   Fonksiyon bakiyeye 250 ekler, ama bakiye artık kuruştur: 250 kuruş = 2,50 lira. Doğrusu 250 × 100 = 25.000 kuruş eklemekti.

5. **C** · Yalnız paraYatir ve paraCek metotlarının içinin
   Bakiyeye yalnız sınıfın kendi metotları dokunduğu için gelen tutarı 100'le çarpmak bu iki metodun içinde yapılır. ayse.paraYatir(500) gibi çağıran satırlar olduğu gibi kalır.

6. **D** · Kapsülleme
   Kapsüllemede veri nesnenin içinde kapalı durur ve yalnız nesnenin kendi işlemleriyle değişir. Dışarıdan doğrudan yazmaya kalkan satırı derleyici reddeder.

7. **B** · Çok biçimlilik
   Çok biçimlilikte aynı istek, nesnenin türüne göre farklı iş yapar. İsteği gönderen satır tektir; ne olacağına hesabın kendisi karar verir. Kalıtım ise vadeli hesabın Hesap'ın her şeyini hazır devralmasıdır.

8. **A** · Simula
   Simula 1967'de Norveç'te simülasyon için geliştirildi ve bu üç fikri ilk kez getirdi. Smalltalk 1970'lerde her şeyi nesne yaptı, C++ 1985'te C'ye sınıf ekledi, Java 1995'te geldi.

1 Java'ya giriş

1. **B** · Merhaba.class; hayali bir işlemci için yazılmış bytecode
   Java derleyicisi hiçbir gerçek işlemcinin diline çevirmez; Merhaba.class dosyasına bytecode yazar. Bu dosyayı her sistemdeki Java sanal makinesi çalıştırır; bu yüzden aynı dosya Windows'ta da Linux'ta da çalışır.

2. **C** · Çok tekrarlanan kısımları program çalışırken bir kez makine koduna çevirip saklar.
   Her turda yeniden çevirmek yavaştır; JIT sık çalışan kodu bir kez gerçek makine koduna çevirir ve saklar. Bölümdeki 100 milyon turluk döngü böylece 1,40 saniye yerine 0,12 saniyede bitti. Belleği geri almak çöp toplayıcının işidir.

3. **D** · Metodu çağırmak için önce bir nesne oluşturmak gerekmediğini
   static, metodun bir nesneye bağlı olmadığını söyler; sanal makine programı açarken ortada nesne yoktur. Değer döndürmemek void'in, dışarıdan çağrılabilmek public'in, komut satırı sözcükleri String[] args'ın işidir.

4. **A** · AB ve alt satırda C

   print yazar ve aynı satırda kalır: A. println B'yi aynı satıra yazıp alt satıra geçer: AB. Son print C'yi yeni satıra yazar.

5. **Toplam 800 ml**

   Önce çarpma yapılır: 4 × 200 = 800. Sonra + işareti, yanında metin olduğu için 800'ü metne çevirip uç uca ekler: Toplam 800 ml.

6. **C** · Derlenmez; tam sayı mantıksal tipe (boolean) çevrilemez.

   Java'da koşul yalnız true ya da false olabilir ve int boolean'a kendiliğinden dönüşmez. Derleyici "int cannot be converted to boolean" der. İlk seçenek C'de olan şeydir.

7. **B** · Program o satırda durur; ArrayIndexOutOfBoundsException, indis 12 ve uzunluk 12 yazılır.

   12 elemanlı dizinin indisleri 0'dan 11'e kadardır. Java her erişimde sınırı denetler ve hatanın yerini söyleyerek programı durdurur. Sessizce bozulma C'de olabilecek şeydir.

8. **8**

   length() metnin karakter sayısını verir: K, ı, r, ş, e, h, i, r. Türkçe harfler de tek karakter sayılır; sonuç 8.

2 Sınıflar ve nesneler

1. **B** · Araba nesnesinin heap'teki yeri, yani referans

   new heap'te yer açar, nesneyi kurar ve yerini verir; a'ya yazılan bu yerdir. a, arabayı gösteren bir ok gibidir; böyle değişkenlere referans denir.

2. **50**

   b = a nesneyi kopyalamaz, yalnız yeri kopyalar; a ve b aynı arabayı gösterir. Araba 0'dan başlar, önce 30, sonra 20 km gider: 50.

3. **C** · false, çünkü iki ayrı nesne iki ayrı yerdedir

   == nesnelerin içine bakmaz; iki değişkenin aynı yeri tutup tutmadığına bakar. new iki kez çalıştığı için heap'te iki ayrı nesne vardır; içerikleri aynı olsa da sonuç false olur.

4. **A** · girilen.equals("06 HKZ 07")

   Scanner okuduğu metin için yeni bir nesne oluşturur; == iki ayrı yer gördüğü için harfler aynı olsa da false verir. equals harfleri sırayla karşılaştırır. Tek eşittir ise atamadır.

5. **D** · Program durur ve NullPointerException verir

   plaka'ya değer yazılmadığı için alan null kalır; hiçbir metni göstermez. Olmayan bir metnin length metodu çağrılamaz, Java programı o satırda durdurur. Kod derlenir; hata çalışırken çıkar.

6. **B** · Araba a = new Araba();

   Sınıfa bir yapıcı yazılınca derleyici varsayılan yapıcıyı artık eklemez. new Araba() parametresiz bir yapıcı arar, bulamaz; derleyici "required: String" der.

7. **0**

   Yapıcının içinde km denince Java en yakın adı, yani parametreyi anlar; km = km parametreyi kendine yazar, alana dokunmaz. Alan varsayılan değeri 0'da kalır. Doğrusu this.km = km; yazmaktır.

8. **C** · Tek parametreliyi, çünkü derleyici verilen değerlerin sayısına ve tipine bakar

   Aynı adı taşıyıp parametreleri farklı yapıcılara aşırı yükleme denir ve geçerlidir. Tek bir String verildiği için derleyici tek parametreli yapıcıyı seçer.

3 Kapsülleme ve static

1. **C** · Derlenmez; bakiye Hesap'ın dışına kapalıdır.
   private alana yalnız sınıfın kendi metotları dokunabilir. Derleyici satırı reddeder ve "bakiye has private access in Hesap" der; hata çalışırken değil, derlerken çıkar.

2. **200**
   300 yatar. 100 çekilir, çünkü 100 bakiyeden büyük değildir: bakiye 200. 500 bakiyeden büyük olduğu için paraCek false döner ve bakiye 200'de kalır.

3. **A** · Kapattığımız kapıyı yeniden açar; dışarıdan bakiyeye -500 gibi her değer yazılabilir.
   set metodu hiçbir kural denetlemeden alana yazar; h.setBakiye(-500) bakiyeyi eksiye düşürür. Dışarıya yalnız kuralı olan işlemler, paraYatir ve paraCek, açılmalıdır.

4. **bursa**
   String değişmez bir nesnedir. toUpperCase eldeki metni değiştirmez, büyük harfli yeni bir kopya döndürür; sonuç bir değişkene atanmadığı için sehir hâlâ "bursa"yı gösterir.

5. **B** · Değerini nesne oluşturulurken bir kez alır; sonra ona değer atayan satır derlenmez.
   final alan değerini bir kez, örneğin yapıcı metotta alır. noDegistir gibi sonradan atama yapan bir metot "cannot assign a value to final variable no" hatasıyla derlenmez. Paylaşılan tek kopya static'in işidir.

6. **D** · System.out.println(h.sahip);
   public her yerden okunur. protected aynı paket ve alt sınıflar içindir; Rapor ikisi de değildir. Belirleyicisiz subeKodu yalnız aynı pakete, private bakiye yalnız sınıfın içine açıktır.

7. **5**
   static alan sınıfa aittir; tek bir kopyası vardır ve bütün hesaplar onu paylaşır. Her yeni hesap aynı sayacı bir artırır: 1, 2, 3, 4, 5. static olmasaydı her hesap kendi sayacıyla doğar ve hepsi 1 alırdı.

8. **B** · static metot bir nesne için değil sınıf için çağrılır; hangi hesabın bakiyesinin istendiğini bilemez.
   Mesaj "non-static variable bakiye cannot be referenced from a static context" der. Çözüm nesneyi açıkça vermektir: Hesap h = new Hesap(); ve h.bakiye. main aynı sınıfta olduğu için private alana erişebilir; sorun private değildir.

4 UML sınıf diyagramları

1. **C** · Ad, alanlar, metotlar
   En üstte sınıfın adı, ortada nesnenin tuttuğu bilgiler (alanlar), en altta yapabildiği işler (metotlar) durur.

2. **- sayfa: int**
   private eksi (-) işaretiyle gösterilir. UML'de Java'nın tersine önce ad, sonra iki nokta, sonra tip yazılır: - sayfa: int.

3. **A** · + oduncVer(): boolean
   public artı (+) ile gösterilir ve dönüş tipi en sona, iki noktadan sonra yazılır. Eksi private'tır; ikinci seçenek Java'nın sırasıdır.

4. **D** · protected: alt sınıflar da görür
   Diyez (#) protected'tır. Eksi (-) private, artı (+) public, dalga (~) paket erişimidir; Java'da paket erişimi için sözcük yazılmaz.

5. **B** · implements: sınıf bir arayüzü gerçekleştirir
   Kesik çizgide alan yoktur. Üçgenli kesik çizgi arayüz gerçekleştirmedir (implements); üçgenli düz çizgi kalıtımdır (extends). Ucunda açık ok olan kesik çizgi bağımlılıktır.

6. **A** · Kesik çizgi, Kitap ucunda açık ok: bağımlılık

   Kitap yalnız bir metodun içinde kısa süre kullanılıyor; alan olmadığı için çizgi kesiktir. Buna bağımlılık denir: Kitap değişirse Gorevli de etkilenebilir. Düz çizgi kalıcı bir bağ, yani alan ya da extends ister.

7. **C** · Kayıtlar kütüphaneyle doğar ve onunla birlikte silinir.

   İçi dolu elmas sahipliktir: parçaların ömrü bütüne bağlıdır. İkinci seçenek içi boş elmasın, yani toplamanın anlamıdır. Kodda ikisi de liste alanıdır; elmas ömür bilgisini ekler.

8. **1..***

   1..* en az bir tane demektir. 0..* olsaydı kitabın hiç yazarı olmayabilirdi. Kodda bu, ilk yazarı yapıcı metotta isteyip listeye ekleyen Kitap(Yazar ilkYazar) ile korunur.

5 Sahiplik ve toplama

1. **C** · Oyuncu nesnesinin bellekteki yerini, yani referansını

   Nesne tipindeki bir alan nesnenin kendisini değil, bellekteki yerini tutar; bu bir referanstır. Takımın içine oyuncunun kopyası konmaz, takım yalnız oyuncunun nerede durduğunu bilir.

2. **B** · Bina ve oda

   Ayırt eden soru: bütün silinince parça başka bir yerde işe yarar mı? Bina yıkılınca odaları da onunla gider, bu yüzden sahipliktir. Oyuncu takım dağılınca başka takıma geçer, şarkı çalma listesi silinince yerinde kalır; bunlar toplamadır.

3. **A** · Bir satır aynı anda yalnız bir siparişe ait olabilir.

   Çokluk karşı uçtan bakılarak okunur: Siparis ucundaki 1, bir satırın kaç siparişe bağlı olduğunu söyler. Sahiplikte parça aynı anda iki bütüne ait olamaz. Bir siparişteki satır sayısını SiparisSatiri ucundaki 0..* söyler.

4. **D** · Oyuncu takımdan bağımsız yaşar; takım dağılsa da başka bir takımda oynayabilir.

   İçi boş elmas toplamayı gösterir: parça bütünden bağımsız yaşar. Aynı oyuncu aynı anda hem kulübünde hem milli takımda olabilir. İçi dolu elmas ise parçanın bütünle silindiği sahipliktir.

5. **170**

   tutar() adedi fiyatla çarpar: 2 × 40 = 80 ve 3 × 30 = 90. toplam() listedeki satırları sırayla dolaşıp tutarları ekler: 80 + 90 = 170.

6. **C** · satirEkle'nin satırı new ile kendisi oluşturması ve referansını dışarı vermemesi

   oyuncuEkle dışarıda oluşturulmuş bir oyuncuyu alır; o oyuncuya başka referanslar da ulaşabilir. satirEkle ise ürünü, adedi ve fiyatı alıp satırı kendisi oluşturur; satırın referansı dışarıya hiç verilmez. İki sınıfta da alan bir liste olduğu için fark liste değildir.

7. **A** · Ona ulaşan hiçbir referans kalmadığında

   Java'da nesneyi programcı silmez. Ona ulaşan hiçbir referans kalmayınca çöp toplayıcı onu bellekten kaldırabilir; ne zaman çalışacağı önceden bilinmez.

8. **Bursa**

   Sipariş kendi Adres nesnesini oluşturup şehri ona kopyaladı; ev ile siparişin adresi artık iki ayrı nesnedir. ev'in şehri Eskişehir olur, siparişinki Bursa kalır. Yapıcı metot teslimat = adres; deseydi iki referans aynı nesneyi gösterir ve çıktı Eskişehir olurdu.

6 Kalıtım ve override

1. **C** · Muhendis ve Calisan

   "Mühendis bir çalışandır" cümlesi doğru kurulur, bu yüzden Muhendis Calisan'dan türeyebilir. Sipariş satırı bir sipariş değildir, oyuncu bir takım değildir, görev de bir çalışan değildir; bunlar sahiptir ilişkileridir.

2. **A** · Derlenmez; maas Muhendis nesnesinde vardır ama private olduğu için alt sınıf ona doğrudan ulaşamaz.

   Alan mühendis nesnesinin içinde vardır, ama private olduğu için yalnız Calisan sınıfı ona doğrudan ulaşır. Derleyici "maas has private access in Calisan" der. Çözüm return aylikOdeme() * 12; gibi üst sınıfın açtığı bir metodu kullanmak ya da alanı protected yapmaktır.

3. **D** · Derleyicinin eklediği varsayılan yapıcı super(); çağırır, ama Calisan'ın parametresiz yapıcısı yoktur.

   Yapıcılar devralınmaz; yapıcısı olmayan sınıfa derleyici parametresiz bir yapıcı ekler ve bu yapıcı super(); ile üst sınıfın parametresiz yapıcısını çağırır. Calisan'da böyle bir yapıcı olmadığı için çağrı tutmaz. Çözüm, Muhendis'e kendi yapıcısını yazıp ilk satırında super(ad, maas); çağırmaktır.

4. **123**

   Her yapıcı önce üst sınıfın yapıcısını çağırır; super yazılmadığı için derleyici super(); ekler. KidemliMuhendis Muhendis'i, o da Calisan'ı çağırır; gövdeler ise en tepeden başlayarak çalışır: önce 1, sonra 2, en son 3.

5. **35000**

   super.aylikOdeme() üst sınıfın metodunu çağırır ve maaşı, 30.000'i verir. Üstüne prim eklenir: 30.000 + 5.000 = 35.000.

6. **B** · 40000

   Parametreleri farklı olduğu için yeni metot eskisinin yerine geçmez, yanına eklenir; bu aşırı yüklemedir (overload). Parametresiz çağrı Calisan'dan devralınan metodu bulur ve yalnız maaşı, 40000'i verir. Kod derlenir, hata ancak @Override yazılınca yakalanır.

7. **C** · Üst sınıfta aynı imzalı bir metot yoksa derleyicinin hata vermesini sağlar.

   @Override derleyiciye verilen bir sözdür: bu metot üst sınıftaki bir metodun yerine geçecek. Üst sınıfta yeniden tanımlanacak böyle bir metot bulunmazsa derleyici "method does not override or implement a method from a supertype" der. Üst sınıfın hâlini çağırmak için ise super.metot() yazılır.

8. **A** · equals'a göre eşit iki çalışan farklı hash kodları verebilir ve hızlı arama yapan koleksiyonlarda iki ayrı kişi gibi görünür.

   Kural: equals'a göre eşit olan iki nesne aynı hash kodunu vermeli. Object'teki hashCode genellikle her nesneye ayrı bir sayı verir; bölümdeki örnekte equals true derken hash kodları 1247233941 ve 258952499 çıktı. Kod derlenir; bu yüzden ikisi hep birlikte yazılır.

9. **B** · public class Stajyer extends Calisan { }

   Stajyer, final olmayan Calisan'dan türeyebilir. String final bir sınıftır, ondan türenemez; final metot alt sınıfta yeniden tanımlanamaz; Java'da extends'ten sonra yalnız bir sınıf adı yazılır.

10. **4 x 4**

    setEn(5) eni de boyu da 5 yapar; ardından setBoy(4) ikisini de 4 yapar. Sonuç 4 x 4. Dikdörtgende aynı satırlar 5 x 4 verirdi; kare, dikdörtgenin "en değişince boy yerinde kalır" sözünü bozar.

7 Çok biçimlilik

1. **B** · Kare'nin alan() metodu; ekrana 4.0 çıkar.
   Override edilen metodun hangi hâlinin çalışacağı program çalışırken, nesnenin gerçek tipine bakılarak seçilir (dinamik bağlama). s'nin gösterdiği nesne bir Kare olduğu için Kare'nin metodu çalışır: 2 × 2 = 4.0.

2. **D** · Sekil tipindeki s için s.alan() çağrısında Daire'nin mi Kare'nin mi metodunun çalışacağı
   Override edilen metotta karar program çalışırken nesneye bırakılır. Aşırı yüklenmiş metotlar (println(int), println(String)) arasında seçimi derleyici, verilen değerlerin tipine bakarak yapar; bir çağrının derlenip derlenmediği de derleme anında belli olur.

3. **13.0**
   Her nesne kendi alan metodunu çalıştırır: Kare(2) 4.0, Kare(3) 9.0 verir; Sekil'in kendi alan metodu ise 0 döndürür. 4 + 9 + 0 = 13; toplam double olduğu için ekrana 13.0 çıkar.

4. **A** · s'nin derleme anındaki tipi Sekil'dir ve çağrılabilecek metotları bu tip belirler; Sekil'de cap() yoktur.
   Derleyici yalnız değişkenin tanımında yazan tipi bilir; s'ye bir kare de yazılabilirdi ve karenin çapı yoktur. Bu yüzden Sekil'de olmayan bir metodun çağrılmasına izin vermez. Nesne değişmez; hâlâ bir dairedir.

5. **C** · Daire d = s;
   Sekil'den Daire'ye inmek aşağı dönüşümdür ve kendiliğinden olmaz; derleyici "incompatible types: Sekil cannot be converted to Daire" der. Dönüşüm (Daire) s diye açıkça yazılınca derlenir. Sekil t = s; ve s.alan() ise sorunsuzdur.

6. **A** · Kod derlenir; program çalışırken ClassCastException hatasıyla durur.
   Derleyici yalnız parametrenin tipini, Sekil'i görür; bir şekil gerçekten daire olabileceği için dönüşüme izin verir. Asıl denetim çalışırken yapılır: nesne bir Kare olduğu için Java programı "class Kare cannot be cast to class Daire" mesajıyla durdurur.

7. **false**
   instanceof, nesne o sınıftan ya da onun bir alt sınıfındansa true verir. s'nin gösterdiği nesne bir Kare'dir; kare bir şekildir ama daire değildir. Bu yüzden false çıkar; s instanceof Sekil sorusu ise true verirdi.

8. **B** · true, çünkü s ve d aynı nesneyi gösterir.
   Yukarı dönüşümü derleyici kendiliğinden yapar, çünkü her daire bir şekildir. Nesneye hiçbir şey olmaz, kopyası da alınmaz; aynı nesneye yalnız daha genel bir tipten bakılır. == aynı nesneyi gösterip göstermediklerine bakar ve true verir.

8 Soyut sınıf ve arayüz

1. **B** · Odeme o = new KartOdeme(250);
   Soyut sınıfın nesnesi oluşturulamaz; derleyici "Odeme is abstract; cannot be instantiated" der. Ama Odeme tipinde değişken tanımlanabilir ve bu değişken bir alt sınıfın, örneğin KartOdeme'nin nesnesini tutabilir.

2. **C** · Kod derlenmez; Havale soyut değil ama soyut ode metodunu yazmamış.
   Soyut metot alt sınıfa bir ödevdir. Havale ya ode metodunu yazmalı ya da kendisi de abstract olmalıdır; ikisini de yapmadığı için hata daha derlenirken yakalanır. Odeme'deki ode'nin gövdesi olmadığı için çalışacak bir kod da yoktur.

3. **D** · IadeEdilebilir arayüzü yazılır; KartOdeme ve Bilet onu implements ile gerçekleştirir.
   Kimlik ("bir ödemedir") soyut sınıftan, yetenek ("iade edilebilir") arayüzden gelir. Arayüz aralarında aile bağı olmayan sınıflara ortak bir yetenek verir. Bilet bir ödeme olmadığı için Odeme'den türemez; Java'da bir sınıf iki sınıftan birden de türeyemez.

4. **A** · Hiçbir şey; Kasa yeniden derlenmeden de cüzdanla ödeme alır.
   Kasa yalnız OdemeYontemi'nin sözünü bilir: ode(int) metodu. Cuzdan bu sözü verdiği için parametreye verilebilir ve çalışan, cüzdanın kendi ode metodudur. Cüzdanın kartla akraba olması gerekmez; arayüz iki ekip arasındaki sözleşmedir.

5. **1**
   b'nin tutarı 420, karşılaştırılan a'nın tutarı 85. 420 > 85 olduğu için ikinci if çalışır ve 1 döner: sıralamada b, a'dan sonra gelir.

6. **B** · Comparator<Odeme> arayüzünü gerçekleştiren bir sınıf yazılır ve nesnesi Collections.sort'a listeyle birlikte verilir.
   Sınıfın doğal sırası bir tanedir; aynı imzalı ikinci compareTo derlenmez, başkasının sınıfına zaten dokunamayız. Comparator karşılaştırmayı sınıfın dışında yapar: Collections.sort(odemeler, new YereGore()); gibi.

7. **20**
   Kart kendi hesabını kullanır: 1000 × 2 / 100 = 20. Havale metodu yazmadığı için arayüzdeki varsayılan gövdeyi hazır kullanır ve 0 döner. 20 + 0 = 20.

8. **C** · Başına default yazılıp gövdesiyle birlikte
   Soyut bir metot eklenirse onu yazmayan her sınıf derlenmez. Varsayılan (default) metot gövdesiyle gelir; metodu yazmayan sınıf bu gövdeyi hazır kullanır. Java 8'de List arayüzüne sort metodu böyle eklendi. Arayüzde nesneye ait alan da tutulamaz.

9 Hata yönetimi

1. **B** · B ve C
   "12a" sayıya çevrilemez; parseInt hata fırlatır ve try bloğunda o satırdan sonrası, yani A'yı yazan satır atlanır. catch çalışıp B'yi yazar; catch bitince program durmaz, blokların altından sürer ve C'yi yazar.

2. **C** · at Adimlar.ortalama(Adimlar.java:8)
   java.base ile başlayan satırlar Java'nın kendi kütüphanesine aittir; hata orada değil, ona verilen değerdedir. Bakılacak yer kendi sınıfımızın adının geçtiği ilk satırdır: ortalama metodu, 8. satır. main'in satırı yalnız onu kimin çağırdığını söyler.

3. **NumberFormatException**
   Dökümün ilk satırı ne olduğunu söyler: hatanın türü NumberFormatException, yani sayı biçimi hatası; mesajı da çevrilemeyen metni, "7.890"ı gösterir.

4. **D** · FileNotFoundException
   FileNotFoundException, IOException'ın altında, RuntimeException dalının dışındadır; sebebi dış dünyadadır (dosya yok), bu yüzden derleyici denetler. Öteki üçü RuntimeException dalındadır; denetlenmez, çaresi kodu düzeltmektir.

5. **B** · FileNotFoundException
   catch bir sınıfı yakalayınca altındaki bütün sınıfları da yakalar. NumberFormatException, ArrayIndexOutOfBoundsException ve NullPointerException RuntimeException'ın altındadır. FileNotFoundException ise IOException üzerinden Exception'dan türer, RuntimeException'ın altında değildir.

6. **A** · throw metodun içinde bir hata nesnesini fırlatır; throws metodun başında fırlatılabilecek hatanın türünü bildirir.

   throw fırlatma işinin kendisidir ve bir nesneyle yazılır: throw new YetersizBakiyeException(...). throws ise metodun başındaki bildiridir: void paraCek(int tutar) throws YetersizBakiyeException.

7. **Bakiye yetersiz: 150 TL eksik**

   900 > 750 olduğu için paraCek hatayı fırlatır; eksik tutar 900 − 750 = 150'dir. "Paranızı alabilirsiniz." satırı atlanır, catch hatanın mesajını yazar.

8. **C** · Dosya boştur (0 bayt); yazılanlar tamponda kalmış, dosya kapanmadığı için diske gitmemiştir.

   Yazılanlar önce bellekteki tamponda bekler ve dosya kapanınca diske yazılır. close hiç çalışmadığı için tampondakiler kaybolur; dosya FileWriter açılırken oluşmuştur ama 0 bayttır. close finally bloğuna alınsa ya da try-with-resources kullanılsa iki satır dosyada kalırdı.

10 Dosyalar ve akışlar

1. **A** · System.in de bir InputStream'dir; metot baytın nereden geldiğini bilmez.

   InputStream soyut bir sınıftır; dosyadan, klavyeden ya da ağdan okuyan bütün akışlar bu tiptedir. Metot yalnız read ile bayt ister, baytın nereden geldiğiyle ilgilenmez; bu yüzden tek satırı değişmeden klavyeden de okur.

2. **8**

   UTF-8'de İngiliz alfabesindeki harfler 1, ü ve ş gibi Türkçe harfler 2 bayt tutar. G, l, e, n 4 bayt; ü ve ş 2 + 2 = 4 bayt; toplam 8 bayt.

3. **C** · Kodlama yazılmadığı için okuyucu varsayılan windows-1254 tablosunu kullanır ve Türkçe harflerin iki baytını iki ayrı harf sanır.

   Dosya sağlamdır; Not defteri adı doğru gösterir. Java 17 kodlama verilmezse işletim sisteminin varsayılanını kullanır; Türkçe Windows'ta bu, her harfi tek baytla yazan windows-1254'tür. Çözüm kodlamayı yazmaktır: new FileReader("kayit.txt", UTF_8).

4. **2443**

   20.000.000 / 8.192 ≈ 2.441,4; son parça eksik de olsa ayrı bir istektir, yani 2.442 parça. Veri bitince −1 döndüren son istekle toplam 2.443 olur. Tamponsuz okumada bu sayı 20.000.001'dir.

5. **B** · Yalnız 4730

   FileWriter dosyayı açarken içini boşaltır ve baştan yazar; önceki skorlar silinir. Sona eklemek için yapıcıya üçüncü değer olarak true verilir: new FileWriter("skorlar.txt", UTF_8, true).

6. **false**

   Yüklenen nesnenin içi aynıdır (Gülşen, seviye 7, can 3), ama bellekte baytlardan yeniden kurulmuş yeni bir nesnedir. == iki değişkenin aynı nesneyi gösterip göstermediğine baktığı için false çıkar.

7. **C** · Alan kaydedilmez; yüklenen kahramanda calar null olur, ilk değer satırı çalışmaz.

   transient, alanın kaydedilmeyeceğini söyler; kayıt sorunsuz yazılır. Nesne yüklenirken sınıfın yapıcısı ve alanların ilk değerleri çalışmaz, bu yüzden calar null kalır; oyun müzik çaları açılırken kendisi kurar.

8. **B** · Hangi sınıfın kurulacağını dosya söyler; okunurken o sınıfların kodu çalışır ve zararsız sınıflar zincirlenerek bir komut çalıştırılabilir.

   Serileştirilmiş baytlarda sınıfın adı yazılıdır ve bazı sınıflar okunurken kendi kodunu çalıştırır; dönüşüm hatası ancak bu koddan sonra gelir. Bu yüzden Java serileştirmesi yalnız güvenilen dosyalarda kullanılır; dışarıdan gelen veri JSON gibi yalnız veri taşıyan bir biçimle alınır.

11 Jenerikler ve koleksiyonlar

1. **B** · İkinci satır derlenmez: tam sayı String'e dönüştürülemez.
   Kutu<String>'de T'nin yerini String almıştır; koy artık yalnız String kabul eder. Derleyici "int cannot be converted to String" der ve satır derlenmez. Tip denetimi çalışma anından derleme anına taşınmıştır; ClassCastException, jeneriksiz Object kutusunda görülen hatadır.

2. **D** · public static <T extends Comparable<T>> T enBuyuk(List<T> liste)
   Sınırsız T her şey olabilir (T extends Object); derleyici her tipin compareTo metodu olduğunu varsayamaz. <T extends Comparable<T>> sınırı T'nin karşılaştırılabilir olmasını şart koşar; Comparable arayüzü compareTo'yu zorunlu kıldığı için çağrı derlenir.

3. **Mersin**
   Metinlerde compareTo alfabe sırasına bakar; en büyük, alfabede en sonda gelendir. B, A ve M harflerinden en sonda M gelir: Mersin.

4. **C** · false ve true
   1000 otomatik kutulanırken Integer.valueOf'un önceden hazırladığı −128..127 aralığının dışındadır; her eklemede yeni bir Integer nesnesi oluşur. == iki ayrı nesne gördüğü için false, equals değerlere baktığı için true yazar.

5. **A** · int bir temel tiptir, nesne değildir; jenerik tipler yalnız nesne tutar, silinen tipin yerine geçen Object'in yerine int oturamaz.
   Derleyici "required: reference, found: int" der: burada bir referans tipi beklenir. Tip silmede T'nin yerine Object geçtiği için yalnız nesneler uyar. Çözüm sarmalayıcı sınıftır: List<Integer>.

6. **3**
   Liste beş elemanı tekrarlarıyla tutar; küme ise her elemanı bir kez tutar. Rize üç kez geçse de kümede bir kez bulunur: Rize, Trabzon, Artvin, yani 3.

7. **D** · 42 döndürür, boyut 2 olur.
   Eşlemede her anahtar bir kez bulunur; aynı anahtarla put yeni çift eklemez, eski değerin yerine yenisini koyar ve silinen eski değeri, 42'yi döndürür. Çift sayısı değişmez: 2.

8. **B** · Iterator ile dolaşıp silinecek elemanda it.remove() çağırmak.
   Listeden doğrudan silince elemanlar kayar ama iteratörün haberi olmaz: ya ConcurrentModificationException çıkar ya da döngü bir elemanı atlayıp yanlış sonuç verir. it.remove() ile silen iteratör yerini bilir; kural, dolaşırken yalnız dolaşanın değiştirmesidir.

9. **C** · Object'in hashCode'u iki nesneye farklı sayılar verir; küme farklı bölmelere bakar ve equals hiç çağrılmaz.
   HashSet önce hashCode'a bakar; equals yalnız aynı sayıyı veren elemanlar arasında çağrılır. hashCode da telefondan üretilince eşit iki kayıt aynı sayıyı verir ve ikincisi eklenmez. Kural: equals'a göre eşit iki nesne aynı hashCode'u vermelidir.

10. **true**
    Derleyici açılı parantezdeki tipleri yalnız denetim için kullanır, sonra siler (tip silme). Çalışırken ikisinin sınıfı da aynı java.util.ArrayList'tir; bu yüzden true yazılır.

12 Enum, iç sınıf, lambda

1. **C** · Derlenmez; bir int, Gun tipine dönüştürülemez.
   Gun tipindeki değişken yalnız enum'da sayılan yedi değerden birini tutabilir. javac "incompatible types: int cannot be converted to Gun" der; hata program çalışmadan, derleme anında yakalanır. Buna tip güvenliği denir.

2. **A** · Her enum değeri programda yalnız bir kez oluşturulan sabit bir nesnedir; iki SALI aynı nesneyi gösterir.
   Enum bir sınıftır ve derleyici her değeri o sınıfın public static final bir nesnesi olarak üretir. Değer bir kez oluşturulduğu için aynı değeri gösteren iki değişken aynı nesneyi gösterir; == bu yüzden doğru sonucu verir.

3. **700**
   Parantezdeki sayılar yapıcı metotla değerin alanlarına yerleşir: ay 24, fiyat 16800. aylikFiyat fiyatı ay sayısına böler: 16800 / 24 = 700.

4. **B** · Enum'dan new ile nesne oluşturulamaz; yapıcı metodu kendiliğinden private'tır.
   javac "enum types may not be instantiated" der. Enum'un değerleri yalnız enum'un içinde sayılanlardır ve her biri bir kez oluşturulur; yeni bir tür eklemek için enum'a değer yazılır.

5. **D** · B
   Ok işaretli (→) yazımda alttaki duruma düşme olmaz; yalnız eşleşen durumun satırı çalışır ve B yazılır. B ile C'yi birlikte yazan, case UC_AYLIK: biçimindeki iki noktalı eski yazım olurdu.

6. **Ulus: içeride 3/3**
   İki turnike de aynı şubeden üretildiği için aynı iceride alanını artırır. İlk üç geçişte sayı 1, 2, 3 olur; dördüncüde 3 < 3 yanlıştır, sayı artmaz. toString ad + ": içeride " + iceride + "/" + kapasite verir: Ulus: içeride 3/3.

7. **A** · static iç sınıf olarak; nesneleri dış nesneye gizli bir bağ taşımaz ve dış nesne olmadan oluşturulabilir.
   Kural: içteki sınıf dış nesnenin alanlarını kullanıyorsa iç sınıf, kullanmıyorsa static iç sınıf. static iç sınıf daha hafiftir; Sube.Adres gibi, şube nesnesi olmadan new Sube.Adres(...) ile oluşturulur.

8. **C** · Program$1.class
   İsimsiz sınıfın adı olmasa da o bir sınıftır; derleyici onun için ayrı bir sınıf dosyası üretir ve adını kendisi koyar: Program$1.class. Sınıf dosyası üretmeyen, lambdadır.

9. **B** · Tek soyut metodu olan bir arayüz
   Lambda, beklenen arayüzün tek soyut metodunun gövdesi olur; tek soyut metotlu arayüze fonksiyonel arayüz denir. İki soyut metot olursa derleyici gövdenin hangisine ait olduğunu bilemez ve kod derlenmez.

10. **[Pilates, Yüzme]**
    filter yalnız saati 10'dan küçük olanları geçirir: Pilates (9) ve Yüzme (7). map her seansı adına çevirir, toList listede toplar. Stream elemanları listedeki sırayla akar, sıralamaz; Pilates listede önce geldiği için sonuç [Pilates, Yüzme] olur.

13 Paralel programlama

1. **C** · Aynı belleği paylaşır; aynı nesneleri görür ve aynı alanları değiştirebilir.
   Kendi belleğini alan süreçtir; süreçler birbirinin belleğini göremez. Bir sürecin içindeki iş parçacıkları ise aynı belleği paylaşır. Bu paylaşım hem paralel çalışmayı kolaylaştırır hem de aynı veriye aynı anda dokunma sorununu doğurur.

2. **main**

   Thread.currentThread() o satırı yürüten iş parçacığını, getName() onun adını verir. Java'da main metodu main adlı iş parçacığında başlar.

3. **B** · küçültülüyor: main

   run'ı doğrudan çağırmak sıradan bir metot çağrısıdır; yeni iş parçacığı açılmaz, iş main'in kendisinde çalışır ve main işin bitmesini bekler. Yeni iş parçacığı (Thread-0) açan, start'tır.

4. **A** · sayac++ değeri oku, bir ekle, geri yaz diye üç işe ayrılır ve bu işlerin arasına öteki iş parçacığı girebilir.

   Bytecode'da sayac++ satırı getstatic (oku), iconst_1 ve iadd (bir ekle), putstatic (yaz) komutlarına ayrılır. Biri okuduktan sonra, yazmadan önce öteki de okursa iki artıştan biri kaybolur.

5. **41**

   İkisi de yazmadan önce aynı 40'ı okudu. a'nın yazdığı 41'in üstüne b de 41 yazar; iki artış yapıldığı hâlde sayaç yalnız bir artmıştır. Sonucun çalışma sırasına bağlı olmasına yarış durumu denir.

6. **D** · Her çalıştırmada 200000

   Kilidi tutan iş parçacığı oku, ekle, yaz işlerini bitirene kadar öteki kapıda bekler; üç iş hiç bölünmeden yapılır ve artış kaybolmaz. Bölümdeki denemede 120 çalıştırmanın hepsinde 200000 çıktı. Bedeli, bekleme yüzünden işin sıraya girmesidir.

7. **B** · İkisi de ötekinin tuttuğu kilidi bekler; program hata vermeden sonsuza kadar durur.

   Hiçbiri işini bitirmeden tuttuğu kilidi bırakmaz; ikisi de ötekini bekler. Buna kilitlenme denir. Bekleyen iş parçacıkları işlemciyi kullanmaz ve hata fırlatılmaz; jstack aracı kilitlenmeyi bulabilir.

8. **A** · Kilitleri herkes aynı sırayla alır: adı alfabede önce gelen albüm önce kilitlenir.

   İki taşıma da önce Aile'nin kilidini ister; ilk alan iki kilidi de alıp taşır, öteki kapıda bekler ve kilitler bırakılınca o da taşır. Kilitleri kaldırmak araya girme sorununu geri getirir, beklemeyi uzatmak ise ters sırayı değiştirmediği için kilitlenmeyi önlemez.

9. **C** · İşler çalışır ama havuzdaki iş parçacıkları yeni iş beklemeyi sürdürür; program kapanmaz.

   Havuzdaki iş parçacıkları eldeki işleri bitirince kuyrukta yeni iş beklemeye devam eder. Bölümdeki denemede main bitti yazıldı, fotoğraf satırları geldi ve program çalışmayı sürdürdü. shutdown yeni iş almayı keser ve iş parçacıklarının kapanmasını sağlar.

10. **3**

    Havuz baştan 3 iş parçacığı açar ve bütün işleri onlar yapar; işini bitiren kuyruktaki sıradaki işi alır. 12 iş olsa da adlar pool-1-thread-1, -2 ve -3'ten ibarettir; hangisinin kaç iş yapacağı çalıştırmadan çalıştırmaya değişir.

Öğrendiğini pekiştir

Bu notlardaki bütün soruları ve daha fazlasını sitede, her cevabın hemen ardından çözümüyle birlikte çözebilirsin.

hekzon.com/dersler

HEKZON