Nested Loops

Control Flow kategorisinin altıncı ve son topic'i: temel iç içe `for` döngüsü, 2 boyutlu dizi gezinme, etiketsiz break/continue'nun yalnızca en içteki döngüyü etkilemesi, etiketli (labeled) break/continue, ve O(n²) performans farkındalığı.

Başlangıç 22 dk

Son güncelleme: 19 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/nested-loops

---

Nested Loops

Şimdiye kadar gördüğümüz her döngü (`for` , `while` , `do-while`) tek başına, düz bir sırayla çalışıyordu. Ama birçok gerçek problem tek boyutlu değil -- bir tablonun her hücresi, bir görüntünün her pikseli, iki listenin her olası eşleşmesi gibi. Bunun için bir döngünün gövdesine başka bir döngü koyarız: **iç içe döngüler (nested loops)**. Kontrol Akışı kategorisinin bu son topic'i, önceki 5 topic'te öğrenilen her şeyi (`for` , `break` , `continue` , `while`) bir araya getirip iç içe kullanıldıklarında ortaya çıkan yeni davranışları ele alır.

İç İçe Döngüler Nedir?

İç içe döngü, bir döngünün gövdesinin (`for` , `while` ya da `do-while` fark etmez) kendi içinde başka bir döngü çalıştırmasıdır:

```
for (int i = 0; i < 3; i++) {      // dış döngü
    for (int j = 0; j < 3; j++) {  // iç döngü
        // ...
    }
}
```

Dış döngü her bir adım attığında, iç döngü **baştan sona tamamen** çalışır. Yani dış döngü 3 kez, iç döngü her seferinde 3 kez çalışırsa, gövde toplam 3 × 3 = 9 kez çalışır -- bu çarpım ilişkisi, iç içe döngülerin en temel özelliğidir.

Neden Var?

Tek bir döngü, doğrusal (1 boyutlu) bir veri üzerinde gezinmek için yeterlidir -- bir dizi, bir liste. Ama gerçek dünyadaki birçok veri ve problem 2 (ya da daha fazla) boyutludur: bir satranç tahtası, bir Excel tablosu, bir görüntünün genişlik×yükseklik pikselleri, ya da "her

öğrenci için her ders" gibi iki ayrı koleksiyonun tüm kombinasyonları. Tek bir döngüyle bunları ifade etmenin bir yolu yok -- her boyut için ayrı bir döngü seviyesi gerekir. İç içe döngüler, bu çok boyutlu gezinmeyi dilin kendi temel araçlarıyla (yeni bir sözdizimi öğrenmeden) doğal şekilde ifade etmemizi sağlar.

Tarihçe

İç içe döngüler, ayrı bir dil özelliği değildir -- herhangi bir döngü türünün gövdesine başka bir döngü yazılabilmesinin doğal bir sonucudur, bu yüzden Java'nın ilk gününden (1996) beri mevcuttur. Özel bir sözdizimi ya da anahtar kelime gerektirmez; burada asıl "yeni" olan, `break` / `continue`'nun birden fazla döngü seviyesi varken nasıl davrandığı ve bunu netleştirmek için Java'nın sunduğu **etiketli (labeled) break/continue** mekanizmasıdır -- bu da dilin başından beri var olan, C'de bulunmayan bir Java özelliğidir.

Temel İç İçe for Döngüsü

En yalın haliyle, bir dış döngünün her adımında bir iç döngünün tamamen çalışması:

```
public class NestedForBasicsExample {
    public static void main(String[] args) {
        // The outer loop runs 3 times; for EACH of those, the inner loop runs
        // 3 full times -- so the body executes 3 * 3 = 9 times in total.
        int totalIterations = 0;

        for (int row = 1; row <= 3; row++) {
            for (int col = 1; col <= 3; col++) {
                System.out.print("(" + row + "," + col + ") ");
                totalIterations++;
            }
            System.out.println();
        }

        System.out.println("Total iterations: " + totalIterations);
    }
}
```

Beklenen Çıktı

```
(1,1) (1,2) (1,3)
(2,1) (2,2) (2,3)
(3,1) (3,2) (3,3)
Total iterations: 9
```

Çıktıya dikkat edin: dış döngü değişkeni (`row`) sabit kalırken iç döngü değişkeni (`col`) 1'den 3'e kadar tüm değerleri alıyor -- sonra dış döngü bir adım ilerliyor ve iç döngü yeniden baştan başlıyor.

2 Boyutlu Diziyle Çalışma

İç içe döngülerin en doğal kullanım alanlarından biri 2 boyutlu dizilerdir ("Arrays" dersindeki "Çok Boyutlu Diziler" bölümüne bakabilirsiniz) -- dış döngü satırları, iç döngü o satırın sütunlarını gezer:

```
public class TwoDArrayExample {
    public static void main(String[] args) {
        // A 2D array is really an array of arrays -- matrix[row] is itself an
        // int[]. Nested loops are the natural way to visit every cell: the
        // outer loop walks the rows, the inner loop walks the columns of
        // whichever row we're currently on.
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        int sum = 0;
        for (int row = 0; row < matrix.length; row++) {
            for (int col = 0; col < matrix[row].length; col++) {
                System.out.print(matrix[row][col] + " ");
                sum += matrix[row][col];
            }
            System.out.println();
        }

        System.out.println("Sum of all elements: " + sum);
    }
}
```

Beklenen Çıktı

```
1 2 3
4 5 6
7 8 9
Sum of all elements: 45
```

`matrix[row].length` kullanmaya dikkat edin -- `matrix.length` yerine bunu kullanmak, dizinin "jagged" (satırları farklı uzunlukta) olduğu durumlarda da doğru çalışır.

İç İçe Döngülerde break ve continue

`break` ("for Loop" dersindeki "break ile Döngüden Çıkmak" bölümüne bakabilirsiniz) ve `continue` ("for Loop" dersindeki "continue ile Bir Adımı Atlamak" bölümüne bakabilirsiniz), etiketsiz kullanıldıklarında **yalnızca yazıldıkları en içteki döngüyü** etkiler -- dış döngüyü etkilemezler:

```
public class BreakContinueInNestedLoopExample {
    public static void main(String[] args) {
        // An unlabeled break only exits the loop it's directly written
        // inside -- the INNER loop here. The outer loop is completely
        // unaffected and keeps running its remaining iterations.
        System.out.println("break in a nested loop:");
        for (int row = 1; row <= 3; row++) {
            System.out.println("Starting row " + row);
            for (int col = 1; col <= 5; col++) {
                if (col == 3) {
                    System.out.println("  breaking inner loop at col = " + col);
                    break;
                }
                System.out.println("  col = " + col);
            }
            System.out.println("Finished row " + row);
        }

        // Just like break, an unlabeled continue only affects the loop it's
        // directly written inside. It skips the rest of the current inner
        // iteration and moves on -- the outer loop's iteration count is
        // untouched.
        System.out.println("continue in a nested loop:");
        for (int row = 1; row <= 2; row++) {
            for (int col = 1; col <= 4; col++) {
                if (col % 2 == 0) {
                    continue;
                }
                System.out.println("row=" + row + ", col=" + col);
            }
        }
    }
}
```

Beklenen Çıktı

```
break in a nested loop:
Starting row 1
  col = 1
  col = 2
  breaking inner loop at col = 3
Finished row 1
Starting row 2
  col = 1
  col = 2
  breaking inner loop at col = 3
Finished row 2
Starting row 3
  col = 1
  col = 2
  breaking inner loop at col = 3
Finished row 3
continue in a nested loop:
row=1, col=1
row=1, col=3
row=2, col=1
row=2, col=3
```

Çıktıda görüldüğü gibi, `break` iç döngüyü her satırda `col = 3`'te kırıyor ama dış döngü 3 satırın tamamını normal şekilde tamamlıyor; `continue` ise çift sayıları atlıyor ama bu yalnızca iç döngüyü etkiliyor, dış döngü (`row`) hiçbir adım atlamadan ilerliyor. İkisinin de etkisi yalnızca bulunduğu döngü seviyesiyle sınırlı.

Etiketli break ve continue (Labeled break/continue)

Bazen dış döngüyü, içeriden (iç döngünün içinden) doğrudan etkilemek gerekir -- örneğin bir eşleşme bulunduğunda TÜM aramayı durdurmak. Bunun için bir döngünün hemen önüne bir **etiket** (`isimBirTanımlayıcı:`) konur, `break` / `continue` bu etiketle birlikte kullanılır:

```
public class LabeledBreakContinueExample {
    public static void main(String[] args) {
        int[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
            {7, 8, 9}
        };

        // A label (an identifier followed by ':') placed right before a loop
        // lets break/continue target THAT loop specifically, even from
        // inside a nested loop. Without the label, break/continue could only
        // ever affect the innermost loop they're written in.
        int target = 5;
        boolean found = false;

        searchLoop:
        for (int row = 0; row < matrix.length; row++) {
            for (int col = 0; col < matrix[row].length; col++) {
                if (matrix[row][col] == target) {
                    System.out.println("Found " + target + " at [" + row + "][" + col +
"]");
                    found = true;
                    break searchLoop; // exits BOTH loops immediately
                }
            }
        }
        if (!found) {
            System.out.println(target + " not found");
        }

        // Labeled continue: skip straight to the outer loop's next
        // iteration as soon as any negative number is found in a row.
        int[][] withNegatives = {
            {1, 2, -3},
            {4, -5, 6},
            {7, 8, 9}
        };

        rowLoop:
        for (int row = 0; row < withNegatives.length; row++) {
            for (int col = 0; col < withNegatives[row].length; col++) {
                if (withNegatives[row][col] < 0) {
                    System.out.println("Row " + row + " has a negative value, skipping it");
                    continue rowLoop; // jumps straight to row + 1
                }
            }
            System.out.println("Row " + row + " is all non-negative");
        }
    }
}
```

Beklenen Çıktı

```
Found 5 at [1][1]
Row 0 has a negative value, skipping it
Row 1 has a negative value, skipping it
Row 2 is all non-negative
```

`break searchLoop;` hem iç hem dış döngüyü aynı anda sonlandırıyor -- etiketsiz bir `break` yalnızca iç döngüyü durdurup dış döngünün devam etmesine izin verirdi. `continue rowLoop;` ise doğrudan dış döngünün bir sonraki adımına atlıyor, o satırın kalan sütunlarını tamamen atlayarak.

Performans: Neden O(n²)?

İç içe döngülerin bir maliyeti vardır: her ek döngü seviyesi, işi çarpımsal olarak artırır. Tek bir döngü `n` eleman üzerinde `n` adım atar (`O(n)`); aynı büyüklükte ikinci bir döngü onun içine yerleştirildiğinde, gövde artık `n × n` kez çalışır (`O(n²)`):

```
public class NestedLoopPerformanceExample {
    public static void main(String[] args) {
        // Every extra level of nesting multiplies the work. One loop over n
        // elements is O(n). Nest a second loop of the same size inside it,
        // and the body now runs n * n times -- O(n^2). This example doesn't
        // measure wall-clock time (unreliable on a shared machine); instead
        // it counts actual operations to make the growth concrete.
        int[] sizes = {10, 100, 1000};

        for (int n : sizes) {
            long operations = 0;
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    operations++;
                }
            }
            System.out.println("n = " + n + " -> " + operations + " operations (n^2 = " +
((long) n * n) + ")");
        }
    }
}
```

Beklenen Çıktı

```
n = 10 -> 100 operations (n^2 = 100)
n = 100 -> 10000 operations (n^2 = 10000)
n = 1000 -> 1000000 operations (n^2 = 1000000)
```

`n` 10'dan 100'e (10 kat) çıktığında işlem sayısı 100'den 10.000'e (100 kat) çıkıyor -- doğrusal değil, karesel bir büyüme. Büyük veri kümeleri üzerinde çalışırken iç içe döngülerin bu maliyetini göz önünde bulundurmak gerekir.

Uygulamalı Örnek: Yıldızlardan Piramit (Pyramid Printing)

Şimdiye kadar öğrenilenleri tek bir klasik alıştırmada birleştirelim: yıldızlardan ortalanmış bir piramit basmak. Burada iki AYRI döngü -- biri boşluklar için, biri yıldızlar için -- dış döngünün her adımında sırayla çalışır. **Bu hâlâ tek bir iç içe döngü seviyesidir** -- boşluk ve yıldız döngüleri birbirinin İÇİNDE değil, ikisi de yalnızca dış döngünün içinde, art arda çalışıyor:

```
public class PyramidPrintingExample {
    /**
     * Prints a pyramid of stars, centered with leading spaces.
     * For rows = 4:
     *    *
     *   ***
     *  *****
     * *******
     *
     * This is still just ONE level of nesting -- the "spaces" loop and the
     * "stars" loop each sit one level inside the outer row loop and run one
     * after another, not inside each other.
     */
    public static void main(String[] args) {
        int rows = 4;

        for (int i = 0; i < rows; i++) {
            // Row i needs (rows - i - 1) leading spaces...
            for (int j = 0; j < rows - i - 1; j++) {
                System.out.print(" ");
            }
            // ...followed by (2 * i + 1) stars.
            for (int k = 0; k < 2 * i + 1; k++) {
                System.out.print("*");
            }
            System.out.println();
        }
    }
}
```

Beklenen Çıktı

```
   *
  ***
 *****
*******
```

Her satırın matematiği, satır indeksi `i`'ye (0'dan başlayarak, `rows = 4` için) bağlı: `i=0` için 3 boşluk + 1 yıldız, `i=1` için 2 boşluk + 3 yıldız, `i=2` için 1 boşluk + 5 yıldız, `i=3` için 0 boşluk + 7 yıldız.

`i` arttıkça boşluk sayısı azalıyor, yıldız sayısı artıyor -- bu ikisi birlikte piramidin ortalanmış görünümünü oluşturuyor. Bu, iç içe döngülerin yalnızca "tekrar etmek" için değil, bir döngünün sınırlarını başka bir döngünün o anki değerine göre HESAPLAMAK için de kullanılabileceğinin iyi bir örneği.

Best Practices

İç içe döngüleri gerçekten çok boyutlu bir yapı (2D dizi, tüm çift kombinasyonları) gerektiren durumlarla sınırlı tutun -- gerekmediği yerde ekstra bir döngü seviyesi eklemek kodu hem yavaşlatır hem okumayı zorlaştırır. Etiketli break/continue'yu yalnızca gerçekten dış döngüyü etkilemeniz gerektiğinde kullanın; iç döngüyü etkilemek yeterliyse etiketsiz hâli tercih edin -- gereksiz etiket kullanımı kodu karmaşıklaştırır. Döngü değişkenlerine (`i`, `j` yerine `row`, `col` gibi) anlamlı isimler verin -- özellikle 2'den fazla iç içe döngü seviyesinde `i` / `j` / `k` karışıklığı gerçek bir okunabilirlik sorunudur.

Yaygın Hatalar

En sık yapılan hata, etiketsiz bir `break` / `continue`'nun dış döngüyü de etkileyeceğini sanmaktır -- oysa yalnızca en içteki döngüyü etkiler, dış döngüyü etkilemek için mutlaka etiket gerekir. İkinci yaygın hata, iç ve dış döngü değişkenlerini karıştırmaktır -- örneğin iki iç içe döngüde de `i` kullanıp yanlışlıkla iç döngüde dış döngünün değişkenini güncellemek (ya da tam tersi). Üçüncü hata, iç içe döngülerin maliyetini göz ardı edip büyük veri kümelerinde (örn. `n = 100.000`) `O(n²)` bir algoritma yazmaktır -- bu, `n` büyüdükçe uygulamayı fark edilir şekilde yavaşlatabilir.

Özet, Cheat Sheet ve Terimler Sözlüğü

İç içe döngüler, bir döngünün gövdesine başka bir döngü yazmaktır -- dış döngünün her adımında iç döngü tamamen çalışır (çarpımsal ilişki). Etiketsiz `break` / `continue` yalnızca en içteki döngüyü etkiler; dış döngüyü etkilemek için etiketli `break etiket;` / `continue etiket;` gerekir. `n` elemanlı iki iç içe döngü `O(n²)` maliyetlidir.

```
// Temel iç içe döngü
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        // n * n kez çalışır
    }
}

// Etiketli break: dış döngüyü de sonlandırır
outer:
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (kosul) {
            break outer;
        }
    }
}
```

Terimler Sözlüğü

- **İç içe döngü (nested loop):** Bir döngünün gövdesi içinde çalışan başka bir döngü.

- **Dış döngü (outer loop) / İç döngü (inner loop):** İç içe iki döngüde, gövdesi diğer döngüyü içeren döngüye dış, onun içindeki döngüye iç döngü denir.
- **Etiket (label):** Bir döngünün hemen önüne konan, `break` / `continue`'nun o belirli döngü seviyesini hedeflemesini sağlayan tanımlayıcı (`isim:`).
- **O(n²) (karesel zaman karmaşıklığı):** Girdi boyutu `n` ile işlem sayısının `n`'in karesi oranında arttığı durum -- tipik olarak aynı boyuttaki veri üzerinde çalışan iki iç içe döngüden kaynaklanır.
- **Piramit/desen basma (pattern printing):** Bir döngünün sınırlarının (kaç kez çalışacağının), dış döngünün o anki değerine bağlı olarak hesaplandığı klasik iç içe döngü alıştırması.