İstisnalar (Exception)

"Diğerlerinin yazdığı programda hata olabilir ama benim yazdığım programda hata olmaz...." ***- Anonim***

Bu bölümde istisnalar üzerinde durulacaktır. İstisna deyince aklınıza ne geliyor? Yanlış yazılmış uygulama mı? Beklenmeyen durum mu? Yoksa her ikisi de mi? İstisna demek işlerin sizin kontrolünüzden çıkması anlamına gelir. Yani kaos ortamı, önceden kestirilemeyen... Birşeylerin ters gitmesi sonucu uygulamanın normal akışına devam edememesi demektir. Bu ters giden bir şeyler ne olabilir? Örneğin kullanıcının uygulamanıza istemeyen veri girmesi olabilir veya açmak istediğiniz dosyanın yerinde olmaması olabilir, örnekleri çoğaltmak mümkündür. (yorum ekle)

8.1. İstisnalara Giriş

Gerçekten tam bir uygulama yazmak ne demektir? Uygulamadan beklenen görevleri yerine getirmesi onu tam bir uygulama yapar mı? Tabii ki yapmaz. Uygulama zaten kendisinden beklenen işi yapmalı, aksi takdirde zaten uygulama olmaz. Bir uygulamanın tam olmasının iki şartı vardır; Birincisi uygulamanın kendisinden beklenen görevleri doğru bir şekilde yerine getirmesidir yani doğruluk, ikincisi ise hatalı davranışlara karşı dayanıklı olmasıdır, sağlamlık. Örneğin bizden iki sayıyı bölmek için bir uygulama istense ne yapılmalıdır, A/ B - A bölüm B çok basit değil mi?. İlk etapta karşı tarafın bizden istediği şey, girilen iki sayının doğru şekilde bölünmesidir - doğruluk, bu öncelikli şarttır, bunda herkes hemfikir. Peki ikinci şart nedir? İkinci şart ise sağlamlıktır, ikinci şart olan sağlamlık genellikle önemsenmez. Bu örneğimizde karşı tarafın bizden istediği olay, iki sayının bölünmesidir ama dikkat edin sayı dedim, kullanıcı `int`, `double` veya `short` ilkel tiplerinde sayı girilebilir. Peki ya kullanıcı *String* bir ifadeyi uygulamanıza yollarsa ne olur? veya A=5, B=0 girince uygulamanız buna nasıl bir tepki verir? (Not :5/0=sonsuz) Uygulamanız direk olarak kapanır mı? Veya uygulamanız bu anlamsız ifadeleri bölmeye mi çalışır? Eğer siz uygulamayı tasarlayan kişi olarak, bu hataları önceden tahmin etmiş ve önlemleri almışsanız sorun ortaya çıksa bile, uygulama için sorun olmaz ama gerçek dünyada her şeyi öngörebilmek imkansızdır. (yorum ekle)
Java programlama dili, oluşabilecek hatalara karşı sert bir yaptırım uygular. Dikkat edin, oluşabilecek diyorum. Java programlama dili, ortada hata oluşmasına sebebiyet verebilecek bir durum var ise yazılan Java dosyasını derlemeyerek (*compile*) kodu yazan kişiye gerekli sert tavrı gösterir. Java programlama dilinin bu tavrı doğru mudur? Kimileriniz diyebilir ki, "Java sadece üstüne düşen görevi yapsın, oluşabilecek hataları bana söyleyerek canımı sıkmasın". Bu yaklaşım yanlıştır, Java programlama dilinin amacı kodu yazan kişiye maksimum şekilde yardımcı olmaktır, daha doğrusu insana dayalı oluşabilecek hataları kendi üstüne alıp, hatalı uygulama üretimini minimuma indirgemeyi amaçlayarak tasarlanmıştır. Bunun ilk örneğini çöp toplama(*garbage collector*) mekanizmasında görmüştük. Diğer dillerde oluşturulan nesnelerin, daha sonradan işleri bitince bellekten silinmemelerinden dolayı bellek yetmezlikleri oluşmaktadır. " Kodu yazan insan, oluşturduğu nesneyi bellekten temizlemez mi? Ben bunu şahsen hiç yapmam. O zaman dalgın insanlar kod yazmasın aaa! " diye bir söz sakın demeyin, çünkü insanoğlu yeri geldiğinde çok dalgın olabilir ve bu dalgınlık uygulamayı

bir bellek canavarına dönüştürebilir ayrıca bu tür hataları, uygulamanın içerisinden ayıklamak cidden çok zor bir iştir. Bu yüzden Java programlama dilinde, bir nesnenin bellekten silinmesi kodu yazan kişiye göre değil, çöp toplama algoritmalarına göre yapılır (bkz:Bölüm3). Java'nın oluşabilecek olan hatalara karşı bu sert tutumu da gayet mantıklıdır. Bu sert tutum sayesinde ileride oluşabilecek ve bulunması çok güç olan hataların erkenden engellenmesini sağlar. (yorum ekle)

8.1.1. İstisna Nasıl Oluşabilir?

İstisna oluşumuna en basit örnek olarak, yanlış kullanılmış dizi uygulamasını verebiliriz. Java programlama dilinde dizilere erişim her zaman kontrollüdür. Bunun anlamı, Java programlama dilinde dizilerin içerisine bir eleman atmak istiyorsak veya var olan bir elemana ulaşmak istiyorsak, bu işlemlerin hepsi Java tarafından önce bir kontrolden geçirilir. Bunun bir avantajı, bir de dezavantajı vardır. Avantaj olarak güvenli bir dizi erişim mekanizmasına sahip oluruz, dezavantaj olarak ufakta olsa hız kaybı meydana gelir. Fakat böyle bir durumda hız mı daha önemlidir yoksa güvenlik mi? Bu sorunun cevabı Java programlama dili için güvenliktir. Aşağıdaki örneğe dikkat edelim; (yorum ekle)

Örnek: *DiziErisim.java* (yorum ekle)

```
public class DiziErisim {

   public static void main(String args[]) {

     int sayilar[] = {1, 2, 3, 4};
     System.out.println("Basla");
     for (int i=0 ; i < 5 ; i++) {
         System.out.println("--> " + sayilar[i]);
     }
     System.out.println("Bitti");
  }
}
```

`sayilar[]`, ilkel (*primitive*) int tipinde dizi değişkenidir ve bağlı bulunduğu dizi nesnesinin içerisinde 4 adet int tipinde eleman vardır. `for` döngüsü sayesinde dizi içerisindeki elemanları ekrana bastırmaktayız. Bu örneğimizdeki hata, `for` döngüsünün fazla dönmesiyle dizinin olmayan elemanına ulaşmak istememizden kaynaklanmaktadır. Böyle bir hareket, çalışma-anında (*run-time*) hata oluşmasına sebebiyet verip uygulamamızın aniden sonlanmasına sebebiyet verecektir. Uygulamayı çalıştırıp, sonuçları hep beraber görelim. (yorum ekle)

```
Basla
--> 1
--> 2
--> 3
--> 4
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
    at DiziErisim.main(DiziErisim.java:10)
```

Bu örneğimizdeki istisna, *ArrayIndexOutOfBoundsException* istisnasıdır. Bu istisnanın sebebi, bir dizinin olmayan elemanına erişmeye çalıştığımızı ifade eder. Fark edildiği üzere Java programlama dilinde, oluşan istisnaları anlamak ve yerlerini belirlemek çok zor değildir. Örneğin bu uygulamada istisnanın 10. satırda ortaya çıktığı anlaşılabilmektedir. (yorum ekle)

8.1.2. **Başka İstisnalar Neler Olabilir?**

Bir uygulama içerisinde, başka ne tür istisnalar oluşabilir ? Bir kaç örnek verirsek;

- Açmak istediğiniz fiziksel dosya yerinde olmayabilir. (yorum ekle)
- Uygulamanıza kullanıcılar tarafında, beklenmedik bir girdi kümesi gelebilir. (yorum ekle)
- Ağ bağlantısı kopmuş olabilir. (yorum ekle)
- Yazmak istediğiniz dosya, başkası tarafından açılmış olduğundan yazma hakkınız olmayabilir. (yorum ekle)

Olabilir, olmayabilir, belki... Yukarıdaki istisnaların, bir uygulamanın başına gelmeyeceğini kim garanti edebilir? Kimse, peki Java program içerisinde tam bir uygulama nasıl yazılır. Başlayalım... (yorum ekle)

8.1.3. **İstisna Yakalama Mekanizması**

Bir istisna oluştuğu zaman uygulamamız aniden kapanmak zorunda mı? Oluşan bu istisnayı daha şık bir şekilde yakalayıp uygulamanın devam etmesini sağlamak mümkün mü? Cevap olarak evet; (yorum ekle)

Gösterim-8.1:

```
try {
  // Istisnaya sebebiyet verebilecek olan kod
} catch(Exception1 e1) {
  //Eger Exception1 tipinde istisna firlatilirsa buraya
} catch(Exception2 e2) {
  //Eger Exception2 tipinde istisna firlatilirsa buraya
}
```

İstisnaya sebebiyet verebilecek olan kod, `try` bloğunun içerisinde tutularak güvenlik altına alınmış olur. Eğer istisna oluşursa, istisna yakalama mekanizması devreye girer ve oluşan bu istinanın tipine göre, uygulamanın akışı `catch` bloklarından birinin içerisine yönlenerek devam eder. (yorum ekle)
İstisnalar nesnedir. Bir istisna oluştuğu zaman bir çok olay gerçekleşir. İlk önce yeni bir istisna nesnesi belleğin heap alında `new()` anahtar kelimesi ile oluşturulur. Oluşan bu istisna nesnesinin içerisine hatanın oluştuğu satır yerleştirilir. Uygulamanın normal seyri durur ve oluşan bu istisnanın yakalanması için `catch` bloğunun olup olmadığına bakılır. Eğer `catch` bloğu varsa uygulamanın akışı uygun `catch` bloğunun içerisinden devam eder. Eğer `catch` bloğu tanımlanmamış ise hatanın oluştuğu yordamı (*method*)cağıran yordama

istisna nesnesi paslanır, eğer bu yordam içerisinde de istisnayı yakalamak için `catch` bloğu tanımlanmamış ise istina nesnesi bir üst yordama paslanır, bu olay böyle devam eder ve en sonunda `main()` yordamına ulaşan istisna nesnesi için bir `catch` bloğu aranır eğer bu yordamın içerisinde de `catch` bloğu tanımlanmamış ise, uygulananın akışı sonlanır. Bu olayları detaylı incelemeden evvel temel bir giriş yapalım; (yorum ekle)

Örnek: *DiziErisim2.java* (yorum ekle)

```
public class DiziErisim2 {

 public void calis() {

   int sayilar[] = {1,2,3,4};
   for (int i=0 ; i < 5 ; i++) {
     try {
       System.out.println("--> " + sayilar[i]);
     } catch (ArrayIndexOutOfBoundsException ex) {
       System.out.println("Hata Olustu " + ex);
     }
   } // for
 }

 public static void main(String args[]) {

   System.out.println("Basla");
   DiziErisim2 de2 = new DiziErisim2();
   de2.calis();
   System.out.println("Bitti");
 }
}
```

Yukarıdaki uygulamamızda, dizi elemanlarına erişen kodu `try` bloğu içerisine alarak, oluşabilecek olan istinaları yakalama şansına sahip olduk. Sahip olduk da ne oldu diyenler için gereken açıklamayı hemen yapalım. `try-catch` istisna yakalama mekanizması sayesinde istisna oluşsa bile uygulamanın akışı aniden sonlanmayacaktır. *DiziErisim.java*_ ile *DiziErisim2.java* uygulamalarının çıktısına bakılırsa aradaki kontrolü hemen fark edilecektir. *DiziErisim2.java* uygulama örneğimizin çıktısı aşağıdaki gibidir. (yorum ekle)

```
 Basla--> 1--> 2--> 3--> 4Hata Olustu
java.lang.ArrayIndexOutOfBoundsExceptionBitti
```

Kontrol nerede? Yukarıdaki *DiziErisim2.java* uygulamasının çıktısının son satırına dikkat ederseniz, "Bitti" yazısının ekrana yazıldığını görürsünüz oysaki bu ifade *DiziErisim.java* uygulamasının çıktısında görememiştik. İşte kontrol buradadır. Birinci kuralı daha net bir şekilde ifade edersek; `try-catch` istisna yakalama mekanizması sayesinde, istisna oluşsa bile uygulamanın akışı aniden sonlanmaz. (yorum ekle)

Yukarıdaki örneğimizde, `try-catch` mekanizmasını `for` döngüsünün içerisine koyulabileceği gibi, `for` döngüsünü kapsayacak şekilde de tasarlanıp yerleştirilebilir. (yorum ekle)

Örnek: *DiziErisim3.java* (yorum ekle)

```
public class DiziErisim3 {

 public void calis() {
```

```
    try {
      int sayilar[] = {1,2,3,4};
      for (int i=0 ; i < 5 ; i++) {
        System.out.println("--> " + sayilar[i]);
      }
    } catch (ArrayIndexOutOfBoundsException ex) {
        System.out.println("Hata Yakalandi");
    }

  }

  public static void main(String args[]) {

    System.out.println("Basla");
    DiziErisim3 de3 = new DiziErisim3();
    de3.calis();
    System.out.println("Bitti");
  }
}
```

Bu uygulama örneği ile *DiziErisim2.java* örneğimiz arasında sonuç bakımından bir fark yoktur. Değişen sadece tasarımdır, `try-catch` bloğunun daha fazla kodu kapsamasıdır. (yorum ekle)

8.1.4. İstisna İfadeleri

Bir yordam hangi tür istisna fırlatabileceğini önceden belirtebilir veya belirtmek zorunda kalabilir. Bu yordamı (*method*) çağıran diğer yordamlar da, fırlatılabilecek olan bu istisnayı, ya yakalarlar ya da bir üst bölüme iletirler. Bir üst bölümden kasıt edilen, bir yordamı çağıran diğer bir yordamdur. Şimdi bir yordamın önceden hangi tür istisna fırlatacağını nasıl belirtmek zorunda kaldığını inceleyelim. (yorum ekle)

Örnek: *IstisnaOrnek1.java* (yorum ekle)

```
import java.io.*;

public class IstisnaOrnek1 {

  public void cokCalis() {
    File f = new File("ornek.txt");
    BufferedReader bf = new BufferedReader( new FileReader( f
) );
    System.out.println(bf.readLine());
  }

  public void calis() {
    cokCalis();
  }

  public static void main(String args[]) {
    IstisnaOrnek1 io1 = new IstisnaOrnek1();
    io1.calis();
  }
```

```
}
```

java.io paketinin içerisindeki sınıfları henüz incelemedik ama bu örneğimizde kullanılan sınıfların ne iş yaptıklarını anlamak çok zor değil. Burada yapılan iş, aynı dizinde bulunduğu farz edilen *ornek.txt* dosyasının ilk satırını okumaya çalışmaktır. Yukarıdaki uygulamamızı derlemeye çalışırsak, derleyicinin bize vereceği mesaj aşağıdaki gibi olur. (yorum ekle)

```
IstisnaOrnek1.java:9: unreported exception
java.io.FileNotFoundException;
must be caught or declared to be thrown new FileReader(f));
                                         ^
IstisnaOrnek1.java:10: unreported exception java.io.IOException;
must be caught or declared to be thrown
System.out.println(bf.readLine());
                                                          ^
2 errors
```

Biz diskimizde bulunduğu varsayılan bir dosyaya erişip onun ilk satırını okumaya çalışmaktayız. Çok masum gibi gözüken ama tehlikeli istekler. Peki daha detaylı düşünelim ve oluşabilecek olan istisnaları tahmin etmeye çalışalım. (yorum ekle)
İlk oluşabilecek olan istisna, o dosyanın yerinde olmayabileceğidir. Bu beklenmeyen bir durum oluşturabilir, başka neler olabilir? Bundan ayrı olarak biz sanki o dosyanın orada olduğundan eminmişiz gibi birde onun ilk satırını okumaya çalışıyoruz, bu isteğimizde istisnaya sebebiyet verebilir çünkü dosya yerinde olsa bile dosyanın ilk satırı olmayabilir. Dikkat ederseniz hep olasılıklar üzerinde durmaktayım ama güçlü olasılıklar. Peki bu uygulamayı derlemenin bir yolu yok mu? (yorum ekle)
Az önce bahsedildiği gibi bir yordam içerisinde oluşmuş olan istisnayı bir üst bölüme yani o yordamı çağıran yordama fırlatabilir. Eğer bir istisna oluşursa bu anlattıklarımıza göre bir yordamın iki şansı vardır diyebiliriz. Birincisi oluşan bu istisnayı ya yakalayıp gereken işlemleri kendi içerisinde sessizce gerçekleştirebilir veya bu istisna ile ben ne yapacağımı bilmiyorum beni çağıran yordam düşünsün diyip, istisna nesnesini bir üst bölüme fırlatabilir. (yorum ekle)
Aşağıdaki örnekte, oluşan istisnayı aynı yordamın içerisinde yakalanmaktadır; bu yüzden yordamın hangi istisnayı fırlatabileceğini açıklamasına gerek yoktur. Bir yordamın hangi tür istisnayı nasıl fırlatabileceğini açıklama olayını az sonra göreceğiz ama önce aşağıdaki örneğimizi inceleyelim. (yorum ekle)

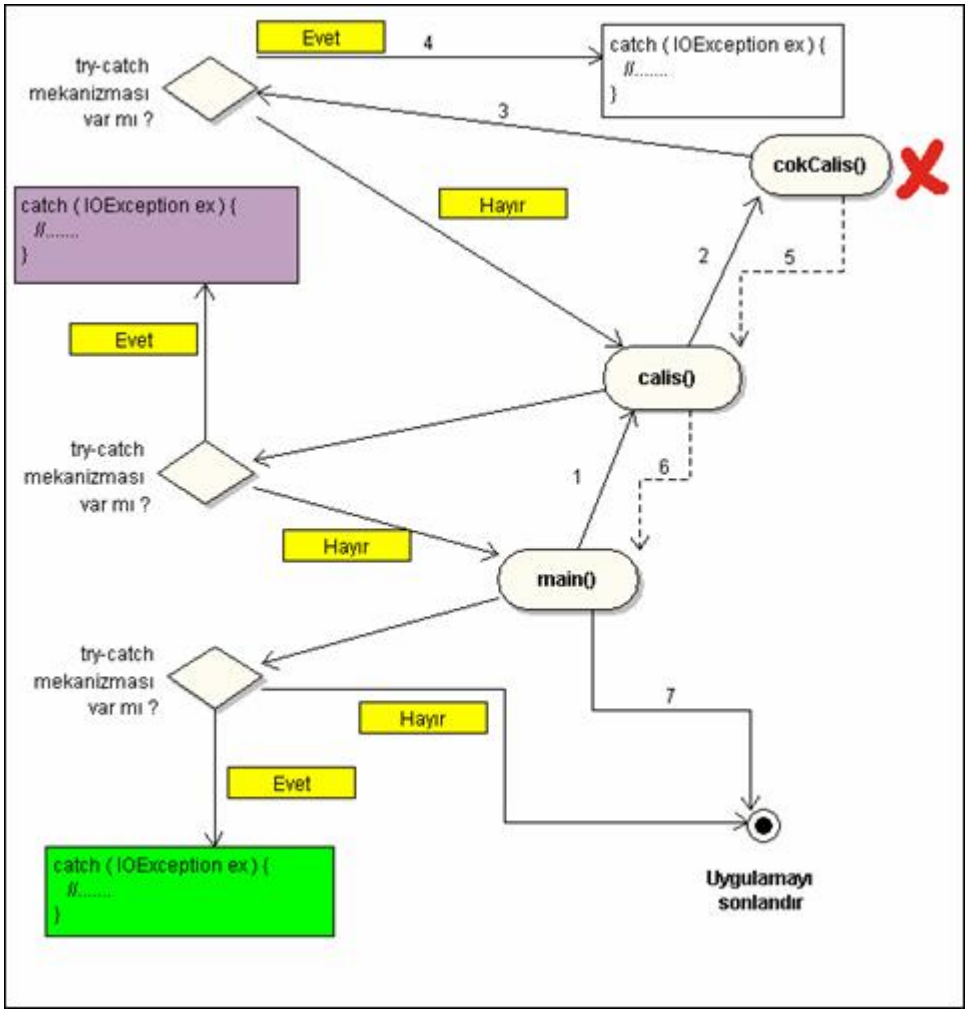
Örnek: *IstisnaOrnek2.java* (yorum ekle)

```
import java.io.*;

public class IstisnaOrnek2 {

  public void cokCalis() {
    try {
     File f = new File("ornek.txt");
     BufferedReader bf=new BufferedReader(new FileReader(f) );
      System.out.println(bf.readLine());
    } catch (IOException ex) {
      System.out.println("Hata Yakalandi =" + ex);
    }
```

```
  }

  public void calis() {
    cokCalis();
    System.out.println("calis() yordamı");
  }

  public static void main(String args[]) {
    IstisnaOrnek2 io2 = new IstisnaOrnek2();
    io2.calis();
    System.out.println("main() yordamı");
  }
}
```

Verilen örnekte, dosyaya erişirken veya ondan birşeyler okumak isterken oluşabilecek olan istisnalar; *java.io.IOException* istisna tipini kullanarak yakalanabilir. Zaten *IstisnaOrnek1.java* uygulamasının derlemeye çalışırken alınan hatadan hangi tür istisna tipinin kullanılması gerektiğini de çıkartabiliriz. *java.io.FileNotFound Exception* istina tipini, *java.io.IOException* tipi kullanılarak yakalanabilir bunun nasıl olduğunu biraz sonra göreceğiz. (yorum ekle)

Yukarıdaki uygulama güzel bir şekilde derlenir çünkü oluşabilecek olan tüm istisnalar için tedbir alınmıştır. Olayların akışını inceliyelim, bu uygulamayı çalıştırdığımız zaman (`java IstisnaOrnek2`) ilk olarak `main()` yordamından akışa başlanır. Daha sonra `calis()` yordamının ve `cokCalis()` yordamının çağrılması şeklinde akış devam ederken olanlar olur ve `cokCalis()` yordamının içerisinde istisna oluşur. Çünkü *ornek.txt* diye bir dosya ortalarda yoktur (yok olduğunu varsayın) ama olsun içimiz rahat çünkü `try-catch` hata yakalama mekanizmamız mevcuttur. Anlattıklarımızı akış diyagramında incelersek..... (yorum ekle)



Şekil-9.1. İstisna Yakalama Mekanizması – I

Akış şemasında numaralandırılmış olan okları takip ederseniz olayların gelişimini çok rahat bir şekilde kavrayabilirsiniz. Akış diyagramımızı açıklamaya başlayalım; (yorum ekle)

1. Öncelikle akış, `main()` yordamının içerisinden başlar. Bu uygulamamızda `main()` yordamının içerisinden `calis()` yordamı çağrılmıştır. (yorum ekle)
2. `calis()` yordamının içerisinden `cokCalis()` yordamı çağrılmıştır. (yorum ekle)
3. `cokCalis()` yordamının içerisinde istisna oluşmuştur çünkü uygulamamızın yer aldığı dizinin içerisinde *ornek.txt* dosyası aranmış ve bulunamamıştır. Şimdi kritik an geldi, `cokCalis()` yordamının içerisinde `try-catch` mekanizması var mı? (yorum ekle)
4. Evet, `cokCalis()` yordamının içerisinde `try-catch` mekanizması olduğu için, `catch` bloğuna yazılmış olan kodlar çalışır. Bu uygulamamızda ekrana " *Hata Yakalandi =java.io.FileNotFoundException: ornek.txt (The system cannot find the file specified)* " basılır, yani dosyanın olmayışından dolayı bir istisna olduğu belirtilir. Not: *java.io.IOException* istisna tipi, *java.io.FileNotFound Exception* istisna tipini kapsadığından bir sorun yaşanmaz bunun nasıl olduğunu biraz sonra inceleyeceğiz. (yorum ekle)

5. Bitti mi? Tabii ki hayır, uygulamamız kaldığı yerden devam edecektir. Şimdi sıra `calis()` yordamının içerisindeki henüz çalıştırılmamış olan kodların çalıştırılmasına. Burada da ekrana "`calis()` yordamı" basılır. (yorum ekle)
6. Son olarak akış `main()` yordamına geri döner ve `main()` yordamının içerisinde çalıştırılmamış olan kodlar çalıştırılır ve ekrana "`main()` yordamı" basılır. (yorum ekle)
7. Ve uygulamamız normal bir şekilde sona erer. (yorum ekle)

Uygulamamızın toplu olarak ekran çıktısı aşağıdaki gibidir.

```
Hata Yakalandi =java.io.FileNotFoundException: ornek.txt (The
system cannot find
the file specified)
calis() yordamı
main() yordamı
```

Akıllara şöyle bir soru gelebilir, "Eğer *ornek.txt* dosyası gerçekten olsaydı yine de `try-catch` mekanizmasını yerleştirmek zorundamıydık". Cevap evet, az önce bahseldiği gibi ortada istisna oluşma tehlikesi varsa bile bu tehlikenin önlemi Java programla dilinde önceden kesin olarak alınmalıdır. (yorum ekle)
IstisnaOrnek2.java uygulamamızda, oluşan istisna aynı yordamın içerisinde yakalanmıştır ve böylece uygulamanın akışı normal bir şekilde devam etmiştir. Peki oluşan bu istisnayı aynı yordamın içerisinde yakalamamak gibi bir lüksümüz olabilir mi? Yani oluşan istisna nesnesini -ki bu örneğimizde oluşan istisnamız *java.io.FileNot FoundException* tipindeydi, bir üst kısma fırlatılabilir mi? Bir üst kısma fırlatmaktan kasıt edilen, istisnanın meydana geldiği yordamı çağıran yordama bu istisna nesnesini fırlatmaktır. "Peki ama niye böyle bişeye ihtiyaç duyalım ki?" diyebilirsiniz. Bunun başlıca sebebi, istisnanın oluştuğu yordam içerisinde, o istisna nesnesi ile ne yapılabileceğinin bilenememesi olabilir. Bir üst kısımda elimizde daha fazla bilgi olabilir, ve bu bilgi çerçevesinde, elimizdeki istisna nesnesini daha güzel bir şekilde değerlendirip, uygulamanın akışını ona göre yönlendirebiliriz. (yorum ekle)

Örnek: *IstisnaOrnek3.java* (yorum ekle)

```
import java.io.*;

public class IstisnaOrnek3 {

  public void cokCalis() throws IOException{

    File f = new File("ornek.txt");
    BufferedReader bf= new BufferedReader( new FileReader(f) );
    System.out.println(bf.readLine());

  }

  public void calis() {
    try {
      cokCalis();
      System.out.println("calis() yordamı");
    } catch(IOException ex) {
      System.out.println("Hata Yakalandi-calis() =" + ex);
    }
```

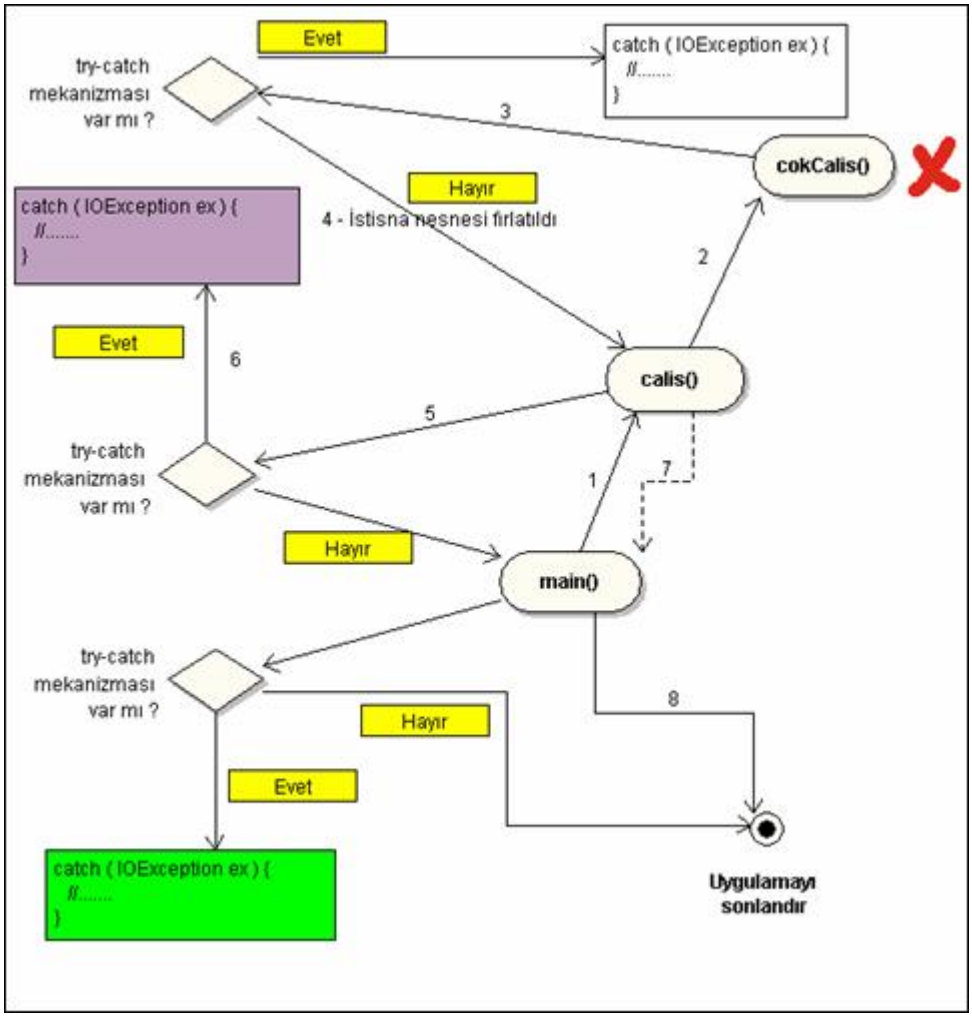
```
  }

  public static void main(String args[]) {
    IstisnaOrnek3 io3 = new IstisnaOrnek3();
    io3.calis();
    System.out.println("main() yordamı");
  }
}
```

IstisnaOrnek3.java örneğimizde oluşan istisna oluştuğu yordam içerisinde yakalanmamıştır. Peki nasıl olurda derleyici buna kızmaz, cevabı hemen aşağıdadır. (yorum ekle)

Gösterim-8.2:

```
public void cokCalis() throws IOException {  //..}
```

Eğer bir istisna oluşursa, istisnanın oluştuğu yordamın yapacağı iki şey vardır demiştik. Birincisi oluşan istisnayı kendi içerisinde `try-catch` mekanizmasıyla yakalayabilir. İkincisi ise oluşacak olan istisnayı bir üst bölüme (kendisini çağıran yordama)fırlatabilir. Örneğin `cokCalis()` yordamı "`throws IOException`" diyerek, kendisini çağıran yordamlara şöyle bir mesaj gönderir, "Bakın benim içimde istisnaya yol açabilecek kod var ve eğer istisna oluşursa ben bunu fırlatırım, bu yüzden başınız çaresine bakın". Buraya kadar anlattıklarımızı akış diyagramında incelersek... (yorum ekle)



Şekil-8.2. İstisna Yakalama Mekanizması - II

Akış şemasında numaralandırılmış olan okları takip ederseniz olayların gelişimini çok rahat bir şekilde kavrayabilirsiniz. Akış diyagramımızı açıklamaya başlayalım; (yorum ekle)

1. Öncelikle akış, `main()` yordamının içerisinden başlar. Bu uygulamamızda `main()` yordamının içerisinden `calis()` yordamı çağrılmıştır. (yorum ekle)
2. `calis()` yordamının içerisinden `cokCalis()` yordamı çağrılmıştır. (yorum ekle)
3. `cokCalis()` yordamının içerisinde istisna oluşmuştur çünkü uygulamamızın yer aldığı dizinin içerisinde *ornek.txt* dosyası aranmış ve bulunamamıştır. Şimdi kritik an geldi, `cokCalis()` yordamının içerisinde `try-catch` mekanizması var mı? (yorum ekle)
4. Hayır, `cokCalis()` yordamının içerisinde oluşan istisnayı yakalama mekanizması yoktur(`try-catch`) ama *java.io.IOException* tipinde bir hata nesnesi fırlatacağını "`throws IOException`" diyerek belirtmiştir. İstisna oluşmuş ve istisna nesnesi (*java.io.IOException*) bir üst bölüme yani `calis()` yordamına fırlatılmıştır. (yorum ekle)

5. Artık istisna nesnemiz `calis()` yordamının içerisindedir, şimdi sorulması gereken soru "`calis()` yordamının içerisinde hata yakalama mekanizması var mıdır? " (yorum ekle)
6. `calis()` yordamının içerisinde hata yakalama mekanizması vardır (`try-catch`) bu yüzden `catch` bloğunun içerisindeki kod çalıştırılır ve ekrana "*Hata Yakalandi-calis() =java.io.FileNotFoundException: ornek.txt (The system can not find the file specified)* " basılır, yani dosyanın olmayışından dolayı bir istisna olduğu belirtilir. Dikkat edilirse ekrana "`calis()` yordamı " basılmadı bunun sebebi istisnanın oluşmasından dolayı akışın catch bloğuna dallanmasıdır. *Not: java.io.IOException* istisna tipi, *java.io.FileNotFoundException* istisna tipini kapsadığından bir sorun yaşanmaz bunun nasıl olduğunu biraz sonra inceleyeceğiz. (yorum ekle)
7. Son olarak akış `main()` yordamına geri döner ve `main()` yordamının içerisinde çalıştırılmamış olan kodlar çalıştırılır ve ekrana "`main()` yordamı" basılır. (yorum ekle)
8. Ve uygulamamız normal bir şekilde sona erer. (yorum ekle)

Uygulamamızın toplu olarak çıktısı aşağıdaki gibidir.

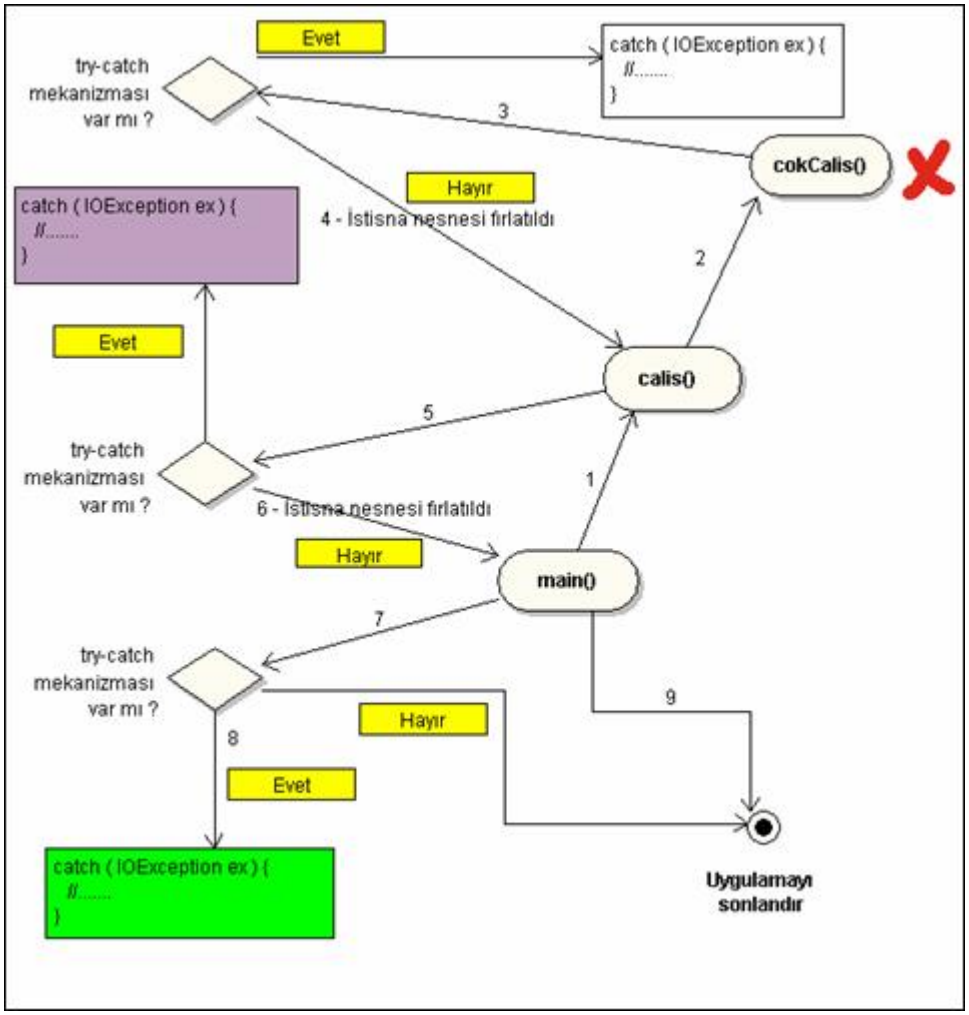
```
Hata Yakalandi-calis() =java.io.FileNotFoundException: ornek.txt (The
system can not find the file specified)main() yordamı
```

Bu örneğimizdeki ana fikir, bir istisna kesin olarak oluştuğu yordamın içerisinde yakalanmayabileceğidir. Fırlatma özelliği sayesinde istisna nesnesi (eğer istisna oluşmuş ise) bir üst bölüme yani istisna oluşan yordamı çağıran yordama fırlatılabilir. (yorum ekle)
Peki bu istisna nesnesi (java.io.IOException) `calis()` yordamın yakalanmasaydı ne olurdu? Cevap: O zaman `main()` yordamın yakalanırdı. Nasıl? Hemen gösterelim. (yorum ekle)

Örnek: *IstisnaOrnek4.java* (yorum ekle)

```
import java.io.*;
public class IstisnaOrnek4 {
  public void cokCalis() throws IOException {
    File f = new File("ornek.txt");
    BufferedReader bf = new BufferedReader( new FileReader( f ) );
    System.out.println(bf.readLine());
  }
  public void calis() throws IOException {
    cokCalis();
    System.out.println("calis() yordamı");
  }
  public static void main(String args[]) {
    try {
      IstisnaOrnek4 io4 = new IstisnaOrnek4();
      io4.calis();
      System.out.println("main() yordamı");
    } catch(IOException ex) {
      System.out.println("Hata Yakalandi-main() =" + ex);
    }
  }
}
```

Bu sefer biraz daha abartıp, oluşan istisna nesnesini son anda `main()` yordamında yakalıyoruz. Bu örneğimizde hem istisnanın meydana geldiği `cokCalis()` yordamı hem de `calis()` yordamı oluşan istisnayı fırlatmışlardır. Buraya kadar anlattıklarımızı akış diyagramında incelersek... (yorum ekle)



Şekil-8.3. İstisna Yakalama Mekanizması - III

Akış şemasında numaralandırılmış olan okları takip ederseniz olayların gelişimini çok rahat bir şekilde kavrayabilirsiniz. Akış diyagramımızı açıklamaya başlayalım; (yorum ekle)

1. Öncelikle akış, `main()` yordamının içerisinden başlar. Bu uygulamamızda `main()` yordamının içerisinden `calis()` yordamı çağrılmıştır. (yorum ekle)
2. `calis()` yordamının içerisinden `cokCalis()` yordamı çağrılmıştır. (yorum ekle)
3. `cokCalis()` yordamının içerisinde istisna oluşmuştur çünkü uygulamamızın yer aldığı dizinin içerisinde *ornek.txt* dosyası aranmış ve bulunamamıştır. Şimdi kritik an geldi, `cokCalis()` yordamının içerisinde `try-catch` mekanizması var mı? (yorum ekle)

4. `cokCalis()` yordamının içerisinde oluşan istisnayı yakalama mekanizması yoktur (`try-catch`) ama *java.io.IOException* tipinde bir hata nesnesi fırlatacağını "`throws IOException`" diyerek belirtmiştir. İstisna oluşmuş ve istisna nesnesi(*java.io.IOException*) bir üst bölüme yani `calis()` yordamına fırlatılmıştır. (yorum ekle)
5. Artık istisna nesnemiz `calis()` yordamının içerisindedir, şimdi sorulması gereken soru " `calis()` yordamının içerisinde hata yakalama mekanizması var mıdır? " (yorum ekle)
6. Cevap hayırdır. `calis()` yordamı da oluşan istisna nesnesini bir üst bölüme yani kendisini çağıran `main()` yordamına fırlatmıştır. (yorum ekle)
7. İstina nesnemiz `main()` yordamının içerisine geldi. Sorulması gereken soru " `main()` yordamının içerisinde hata yakalama mekanizması var mıdır? " (yorum ekle)
8. Cevap evettir. Böylece akış `main()` yordamının içerisindeki `catch` bloğuna dallanır ve catch bloğunun içerisindeki kod çalıştırılır.
9. Ve uygulamamız normal bir şekilde sona erer. (yorum ekle)

Uygulamanın toplu olarak çıktısı aşağıdaki gibidir.

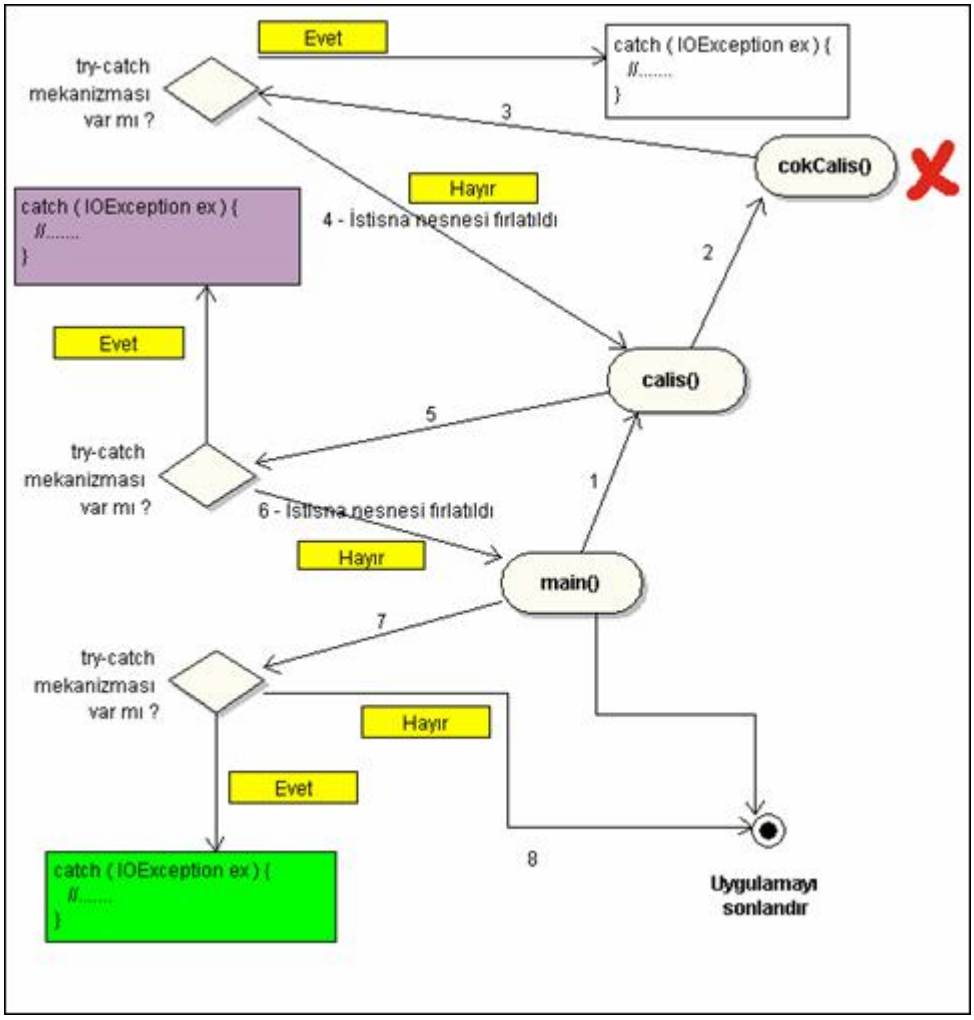
```
 Hata Yakalandi-main() =java.io.FileNotFoundException: ornek.txt(The system cannot find the file specified)
```

Oluşan bir istisna nesnesini catch bloğunda yakalamanın ne gibi avantajları olabilir? Bu sorunun cevabına değinmeden evvel olaylara eğer istisna nesnesi **main()** yordamında yakalanmasaydı neler olacağını inceleyerek başlayalım. (yorum ekle)

Örnek: *IstisnaOrnek5.java* (yorum ekle)

```
import java.io.*;
public class IstisnaOrnek5 {
  public void cokCalis() throws IOException {
    File f = new File("ornek.txt");
    BufferedReader bf = new BufferedReader( new FileReader(f));
    System.out.println(bf.readLine());
  }
  public void calis() throws IOException {
    cokCalis();
    System.out.println("calis() yordamı");
  }
  public static void main(String args[]) throws IOException {
    IstisnaOrnek5 io5 = new IstisnaOrnek5();
    io5.calis();
    System.out.println("main() yordamı");
 }
}
```

Görüldüğü üzere `cokCalis()` yordamının içerisinde oluşan istisna hiçbir yordam içerisinde hata yakalama mekanizması kullanılarak yakalanmamıştır (`try-catch`). Bunun yerine tüm yordamlar bu istisna nesnesini fırlatmayı seçmiştir, buna `main()` yordamıda dahildir. Böyle bir durumda akışın nasıl gerçekleştiğini, akış diyagramında inceleyelim..... (yorum ekle)



Şekil-8.4. İstisna Yakalama Mekanizması - IV

Akış şemasında numaralandırılmış olan okları takip ederseniz olayların gelişimini çok rahat bir şekilde kavrayabilirsiniz. Akış diyagramımızı açıklamaya başlayalım; (yorum ekle)

1. Öncelikle akış, `main()` yordamının içerisinden başlar. Bu uygulamamızda `main()` yordamının içerisinden `calis()` yordamı çağrılmıştır. (yorum ekle)
2. `calis()` yordamının içerisinden `cokCalis()` yordamı çağrılmıştır. (yorum ekle)
3. `cokCalis()` yordamının içerisinde istisna oluşmuştur çünkü uygulamamızın yer aldığı dizinin içerisinde ornek.txt dosyası aranmış ve bulunamamıştır. Şimdi kritik an geldi, `cokCalis()` yordamının içerisinde `try-catch` mekanizması var mı? (yorum ekle)
4. `cokCalis()` yordamının içerisinde oluşan istisnayı yakalama mekanizması yoktur (try-catch) ama java.io.IOException tipinde bir hata nesnesi fırlatacağını "`throws IOException`" diyerek belirtmiştir. İstisna oluşmuş ve istisna nesnesi(java.io.IOException) bir üst bölüme yani `calis()` yordamına fırlatılmıştır. (yorum ekle)

5. Artık istisna nesnemiz `calis()` yordamının içerisindedir, şimdi sorulması gereken soru " `calis()` yordamının içerisinde hata yakalama mekanizması var mıdır? " (yorum ekle)
6. Cevap hayırdır. `calis()` yordamı da oluşan istisna nesnesini bir üst bölüme yani kendisini çağıran `main()` yordamına fırlatmıştır. (yorum ekle)
7. İstina nesnemiz `main()` yordamının içerisine geldi. Sorulması gereken soru " `main` yordamının içerisinde hata yakalama mekanizması var mıdır? " (yorum ekle)
8. Cevap hayırdır. Peki ne olacak? Çok basit, uygulama doğal olarak sonla-nacaktır. (yorum ekle)

Uygulamanın toplu olarak çıktısı aşağıdaki gibidir.

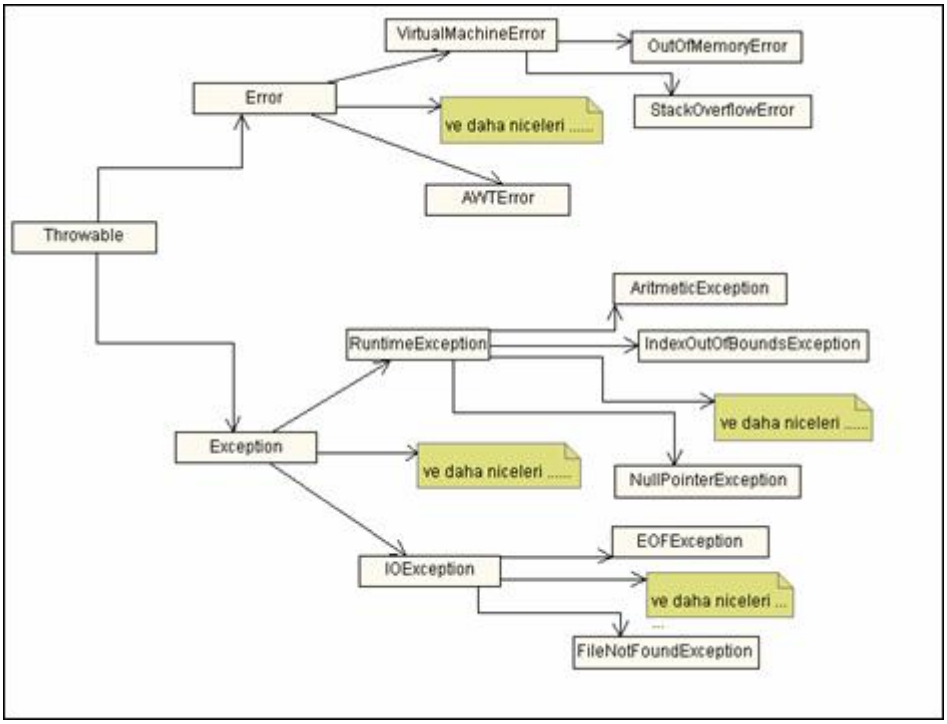
```
Exception in thread "main" java.io.FileNotFoundException:
ornek.txt (The system
cannot find the file specified)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:103)
  at java.io.FileReader.<init>(FileReader.java:51)
  at IstisnaOrnek5.cokCalis(IstisnaOrnek5.java:8)
  at IstisnaOrnek5.calis(IstisnaOrnek5.java:13)
  at IstisnaOrnek5.main(IstisnaOrnek5.java:19)
```

"Hata yakalama mekanizması koyduğumuzda da uygulama sonlanıyordu, şimdide sonlandı bunda ne var ki" diyebilirsiniz. Haklı olabilirsiniz ama önce oluşan bir istisna nesnesi `catch` bloğunda yakalamanın ne gibi avantajları olabilir? (yorum ekle)
Oluşan bir istisna nesnesini catch bloğundan yakalamak, daha doğrusu hata yakalama mekanizması kullanmak uygulamayı yazan kişilere büyük kolaylıklar sağlar. En büyük avantaj oluşan hatayı `catch` bloğunun içerisinde kaydedilirsiniz (*logging*) (dosyaya ama veri tabanına... gibi gibi...) . Örneğin iyi işleyen bir uygulama yazdınız ve bu uygulama yaptığınız tüm -daha doğrusu aklınıza gelen- testlerden geçmiş herşey harika, kendinize güveniniz gelmiş, dünya gözünüze artık bambaşka bir yer gibi geliyor ama bir gün bir bakıyorsunuz ki uygulamanız çalışması durmuş!! ilk yapacağınız şey "bu uygulamayı kim kapattı!" diye etrafa sormak oysaki kimsenin günahı yok, kimse elini uygulamanıza sürmemiştir zaten böyle bir riski kim alabilir ki? Asıl gerçek, uygulamada ters giden birsey olmuş ve uygulama kapanmıştır. İşte tam o anda tutunacağınız tek dal dosyaya veya veri tabanına kayıt ettiğiniz hata mesajlarıdır. Bu bakımdan `catch` bloğunun içerisine oluşan hata ile alakalı ne kadar detaylı bilgi gömerseniz, bu bilgi sizi ileride -eğer hata oluşursa- o kadar yardımcı olacaktır. (yorum ekle)
IstisnaOrnek5.java kötü bir uygulama örneğidir. Oluşabilecek olan bir istisna, hata yakalama mekanizması (`try-catch`) ile sizin öngördüğünüz bir yerde yakalanmalıdır. Bir istisna meydana geldiği zaman uygulama mutlaka sonlanmak zorunda değildir. Eğer bir telafisi var ise bu `catch` bloğunun içerisinde yapılmalı ve uygulama tekrardan ayağa kaldırılmalıdır ama çok ölümcül bir hata ise o zaman hata mesajını kaydetmekten (dosyaya veya veri tabanına.. gibi gibi...) başka yapılacak pek fazla birşey yoktur. (yorum ekle)

8.1.5. **İstisna Tip Hiyerarşisi**

Nasıl olurda java.io.IOException istisna tipi, *java.io.FileNotFoundException* istisna tipini kapsayabilir? Kapsamak ne demektir? Kapsamak demek, eğer

uygulamanızda *java.io.FileNotFoundException* tipinde bir istisna nesnesi oluşmuşsa (bir istisna oluşmuşsa) bu istisna tipini *java.io.IOException* tipini kullanarak da `catch` bloğunda yakalayabileceğiniz anlamına gelir. (yorum ekle)



Şekil-8.5. İstisna Tip Hiyerarşisi

Yukarıdaki şemamızdan görüleceği üzere, *FileNotFoundException* istisna tipi, *IOException* istisnasının alt kümesi olduğu için, *FileNotFoundException* tipinde bir istisna nesnesini `catch` bloğunun içerisinde *IOException* istisna tipiyle yakalayabiliriz. (yorum ekle)
Throwable istisna nesnesi, tüm istisna nesnelerinin atasıdır. Yukarıdaki şemamızıa bakarak istisnaları 3 gruba ayırabiliriz. (yorum ekle)

- **Error** istisna tipi ölümcül bir hatayı işarettir ve telafisi çok zordur, neredeyse imkansızdır. Örneğin *OutOfMemoryError* (yetersiz bellek) istisnası oluşmuş ise uygulamanın buna müdahele edip düzeltmesi imkansızdır. (yorum ekle)
- **RuntimeException** istisna tipleri, eğer uygulama normal seyrinde giderse ortaya çıkmaması gereken istisna tipleridir. Örneğin *ArrayIndexOutOfBoundsException* istisna tipi, bir dizinin olmayan elemanına eriştiğimiz zaman ortaya çıkan bir istisnadır. *RuntimeException* istisna tipleri, kontrolsüz kodlamadan dolayı meydana gelen istisna tipleri diyebiliriz. Biraz sonra bu istisna tipini detaylı biçimde inceleyeceğiz. (yorum ekle)
- Ve **diğer Exception tipleri**. Bu istisna tipleri çevresel koşullardan dolayı meydana gelebilir. Örneğin erişmeye çalışan dosyanın yerinde olmaması (*FileNotFoundException*) veya network bağlantısının kopması sonucu

ortaya çıkabilecek olan istisnalardır ve bu istisnalar için önceden bir tedbir alınması şarttır. (yorum ekle)

8.1.5.1. Tüm Diğer Exception İstisna Tiplerini Yakalamak

Bir uygulama içerisinde oluşabilecek olan tüm istisna tiplerini yakalamak için aşağıdaki ifadeyi kullanabilirsiniz. (yorum ekle)

Gösterim-8.3:

```
catch (Exception ex) {  //......}
```

Tüm istisnaları yakalamak (*Error*, *RuntimeException* ve diğer *Exception* türleri) için *Throwable* istisna tipini kullanmak iyi fikir değildir. Bunun yerine bu üç gruba ait daha özellikli istisna tiplerinin kullanılmasını önerilir. (yorum ekle)

8.1.5.2. RuntimeException İstisna Tipleri

DiziErisim.java uygulama örneğimiz içerisinde istisna oluşma riski olmasına rağmen nasıl oldu da Java buna kızmayarak derledi? Peki ama *IstisnaOrnek1.java* uygulamasını niye derlemedi? Bu soruların cevapları istisna tiplerinin iyi bilenmesi ile ortaya çıkar. (yorum ekle)
DiziErisim.java uygulama örneğinde istisna oluşma riski vardır. Eğer uygulamayı yazan kişi dizinin olmayan bir elemanına erişmeye kalkarsa *ArrayIndexOutOfBounds Exception* hatası alacaktır, yani RuntimeException (çalışma-anı hatası). Peki bunun sebebi nedir? Bunun sebebi kodu yazan arkadaşın dikkatsizce davranmasıdır. Bu tür hatalar derleme anında (*compile-time*) fark edilemez. Java bu tür hatalar için önceden bir tedbir alınmasını şart koşmaz ama yine de tedbir almakta özgürsünüzdür. Bir dosyaya erişirken oluşacak olan istisnaya karşı bir tedbir alınmasını, Java şart koşar çünkü bu tür hatalar diğer Exception istisna tipine girer. Genel olarak karşılaşılan *RuntimeException* istisna türlerine bir bakalım; (yorum ekle)

· ***AritmeticException***: Bir sayının sıfıra bölünmesiyle ortaya çıkabilecek olan bir istisna tipidir.

Gösterim-8.4:

```
int i = 16 / 0 ; // AritmeticException ! hata !
```

· ***NullPointerException***: Bir sınıf tipindeki referansı, o sınıfa ait bir nesneye bağlamadan kullanmaya kalkınca alınabilecek bir istisna tipi. (yorum ekle)

Gösterim-8.5:

```
 String ad == null;// NullPointerException ! hata !System.out.println("Ad
= " + ad.trim() );
```

Bu hatayı almamak için;

Gösterim-8.6:

```
 String ad = " Java Kitap Projesi "; // baglama islemiSystem.out.println("Ad = "
+ ad.trim() ); //dogru
```

· ***NegativeArraySizeException***: Bir diziyi negatif bir sayı vererek oluşturmaya çalışırsak, bu istisna tipi ile karşılaşırız. (yorum ekle)

Gösterim-8.7:

```
// NegativeArraySizeException ! hata !int dizi[] = new dizi[ -100 ];
```

· ***ArrayIndexOutOfBoundsException***: Bir dizinin olmayan elemanına ulaşmak istendiği zaman karşılaşılan istisna tipidir. Daha detaylı bilgi için *DiziErisim.java* uygulama örneğini inceleyiniz. (yorum ekle)
· ***SecurityException***: Genellikle tarayıcı (*browser*) tarafından fırlatılan bir istisna tipidir. Bu istisnaya neden olabilecek olan sebepler aşağıdaki gibidir; (yorum ekle)

- Applet içerisinden, yerel (*local*) bir dosyaya erişilmek istendiği zaman. (yorum ekle)
- Appletin indirildiği sunucuya (*server*) değilde değişik bir sunucuya bağlantı kurulmaya çalışıldığı zaman. (yorum ekle)
- Applet içerisinde başka bir uygulama başlatmaya çalışıldığı zaman. (yorum ekle)

SecurityException istisna tipi fırlatılır.
Önemli noktayı bir kez daha vurgulayalım, *RuntimeException* ve bu istisna tipine ait alt tipleri yakalamak için, Java derleme anında (*compile-time*) bizlere bir bir zorlama yapmaz. (yorum ekle)

8.1.6. İstisna Mesajları

Bir istisna nesnesinden bir çok veri elde edebilirsiniz. Örneğin istisna oluşumunun yol haritasını izleyebilirsiniz veya istisna oluşana kadar hangi yordamların çağrıldığını öğrenebilirsiniz. (yorum ekle)
Bu bilgileri elde etmek için kullanılan *Throwable* sınıfına ait `getMessage()`, `getLocalizedMessage()` ve `toString()` yordamlarının ne iş yaptıklarını örnek uygulama üzerinde inceleyelim. (yorum ekle)

Örnek: *IstisnaMetodlari.java* (yorum ekle)

```
public class IstisnaMetodlari {
  public void oku() throws Exception {
    throw new Exception("istisna firlatildi"); // dikkat
  }
  public static void main(String args[]) {
    try {
      IstisnaMetodlari im = new IstisnaMetodlari();
      im.oku();
    } catch (Exception ex) {
      System.out.println("Hata- ex.getMessage() : " +
                                          ex.getMessage() );
```

```
      System.out.println("Hata-ex.getLocalizedMessage() : " +
                                          ex.getLocalizedMessage() );
      System.out.println("Hata- ex.toString() : " + ex );
    }
  }
}
```

`oku()` yordamının içerisinden bilinçli olarak *Exception* (istisna) nesnesi oluşturulup fırlatılmıştır. Bu istisna sınıfının yapılandırıcısına ise kısa bir not düştüm. `main()` yordamının içerisindeki `catch` bloğunda *Exception* istisna sınıfına ait yordamlar kullanılarak, oluşan istisna hakkında daha fazla bilgi alınabilir. *Exception* sınıfı *Throwable* sınıfından türediği için, *Throwable* sınıfı içerisindeki erişilebilir olan alanlar ve yordamlar otomatik olarak *Exception* sınıfının içerisinde de bulunur. Bu yordamların detaylı açıklaması aşağıdaki gibidir. (yorum ekle)

`String getMessage()`
Oluşan istisnaya ait bilgileri *String* tipinde geri döner. Bu örneğimizde bilgi olarak "istisna firlatildi" mesajını yazdik. Mesajın *String* olmasından dolayı bu yordam bize bu bilgiyi *String* tipinde geri döndürecektir. Eğer *Exception* sınıfının yapılandırıcısına birşey gönderilmeseydi; o zaman `null` değeri döndürülürdü. (yorum ekle)

`String getLocalizedMessage()`
Bu yordam, *Exception* sınıfından türetilmiş alt sınıflar tarafından iptal edilebilir (*override*). Biraz sonra kendi istisna sınıflarımızı nasıl oluşturacağımızı gördüğümüzde, bu yordam daha bir anlam taşıyacaktır. Eğer bu yordam alt sınıflar tarafından iptal edilmemiş ise `getMassage()` yordamı ile aynı sonucu döndürür. (yorum ekle)

`String toString()`
Oluşan istisna hakkında kısa bir açıklamayı *String* tipinde geri döner. Eğer istisna sınıfına ait nesne; bir açıklama ile oluşturulmuş ise - `new Exception ("hata fırlatıldı")` - bu açıklamayı da ekrana basar. `toString()` yordamı oluşan istisna ile ilgili bilgiyi belli bir kural içerisinde ekrana basar. (yorum ekle)

- Oluşan istisna nesnesinin tipini ekrana basar. (yorum ekle)
- ": " iki nokta üst üste koyar ve bir boşluk bırakır. (yorum ekle)
- Son olarak `getMassege()` yordamı çağrılır ve buradan - eğer bilgi varsa- ekrana basılır. (yorum ekle)

Eğer oluşan istisna sınıfına ait nesne bir açıklama ile oluşturulmamış ise yani direk - `new Exception()` - diyerek oluşturulmuş ise son adımda hiçbirsey basmaz. (yorum ekle)
Uygulamamızın çıktısı aşağıdaki gibi olur.

```
Hata- ex.getMessage() : istisna firlatildi
Hata-ex.getLocalizedMessage() : istisna firlatildi
Hata- ex.toString() : java.lang.Exception: istisna firlatildi
```

`Throwable getCause()`

Java 1.4 ile gelen *Throwable* sınıfına ait bir başka yordam ise `getCause()` yordamıdır. Bu yordam *Throwable* nesnesine bağlı referans geri döner. Buradaki amaç, oluşmuş olan istisnanın -eğer varsa- sebebini daha detaylı bir biçimde yakalamaktır. (yorum ekle)

Örnek: *IstisnaMetodlari2.java* (yorum ekle)

```
import java.io.*;

public class IstisnaMetodlari2 {

  public void oku() throws Exception {
    throw new Exception("istisna firlatildi",
                                new IOException() ); // dikkat
  }

  public static void main(String args[]) {
    try {
      IstisnaMetodlari2 im2 = new IstisnaMetodlari2();
      im2.oku();
    } catch (Exception ex) {
      System.out.println("Hata-ex.getCause():" + ex.getCause());
    }
  }
}
```

Bu örnek için *java.io.IOException*_ kullanıldığı için `import java.io.*` denilmeliydi. Bu kısa açıklamadan sonra detayları vermeye başlayalım. (yorum ekle)
`getCause()` yordamın işe yaraması için, istisna sınıfına ait yapılandırıcının içerisine bu istisnaya sebebiyet vermiş olan istisna tipini yerleştirmemiz gerekmektedir. Tekrardan belirtelim bu yordam *Throwable* nesnesine bağlı bir referans geri döndürür. (yorum ekle)

Gösterim-8.8:

```
throw new Exception("istisna firlatildi",new IOException()); // dikkat
```

Böyle bir ifade nerede işimize yarar ki diyebilirsiniz. Bu olay aslında aynı anda iki tip istisna fırlatabilmenize olanak tanır ve bu çoğu yerde işinize yarayabilir. Eğer istisnanın oluştuğu yerde alt istisna nesnesi belirtilmemiş ise - `throw new Exception ("istisna firlatildi")` gibi- `getCause()` yordamı "`null`" dönecektir. (yorum ekle)
Uygulamanın çıktısı aşağıdaki gibi olur.

```
Hata- ex.getCause() : java.io.IOException
```

`Throwable initCause (Throwable cause)`
Java 1.4 ile gelen bir başka yenilik ise `initCause()` yordamıdır. Bu yordam iki istisna tipini birleştirmeye yarar. (yorum ekle)
Örnek: *IstisnaMetodlari3.java* (yorum ekle)

```
import java.io.*;
```

```
public class IstisnaMetodlari3 {

  public void oku() throws Throwable {
    Exception ioEx = new IOException(); // dikkat
    Exception fnfEx = new FileNotFoundException(); // dikkat
    Throwable th = ioEx.initCause(fnfEx);
    throw th;
  }

  public static void main(String args[]) {
    try {
      IstisnaMetodlari3 im3 = new IstisnaMetodlari3();
      im3.oku();
    } catch (Throwable th) {
      // Throwable th = ex.getCause();
      //Throwable th2 = ex.initCause(th); // hata
      System.out.println("Hata-th.initCause():"+ th );
      System.out.println("Hata-th.getCause():"+ th.getCause() );
    }
  }
}
```

`oku()` yordamının içerisinde iki ayrı tipte istisna nesnesi oluşturulmuştur. Bu istisna nesneleri birleştirilerek tek bir Throwable tipinde nesne oluşturmak mümkündür. Hatanın yakalandığı yerde birleştirilen bu iki istisna tipine ayrı ayrı ulaşılabilir. (yorum ekle)
Eğer bir istisna `Throwable (Throwable)` veya `Throwable (String, Throwable)` ile oluşturulmuş ise `initCause()` yordamı çağrılamaz. (yorum ekle)
Uygulamanın çıktısı aşağıdaki gibi olur.

```
 Hata - th.initCause() : java.io.IOExceptionHata - th.getCause() :
java.io.FileNotFoundException
```

`printStackTrace()` :

Oluşan bir hatanın yol haritasını `printStackTrace()` yordamı sayesinde görebilirsiniz. (yorum ekle)

Örnek: *IstisnaMetodlari4.java* (yorum ekle)

```
public class IstisnaMetodlari4 {

 public void cokOku() throws Exception {
   System.out.println("cokOku() yordamı cagrildi");
   throw new Exception("istisna olustu"); // dikkat
 }
 public void oku() throws Exception {
   System.out.println("oku() yordamı cagrildi");
   cokOku();
 }
 public static void main(String args[]) {
   try {
     IstisnaMetodlari4 im4 = new IstisnaMetodlari4();
     im4.oku();
   } catch (Exception ex) {
     ex.printStackTrace();
   }
 }
}
```

Yol haritasına, bir istisna oluşmuş ise bunun hangi satırda meydana gelmiş, istisnanın oluştuğu yordamı hangi yordam çağırmış gibi soruların cevaplarının bulunduğu bir çeşit bilgi kümesi diyebiliriz. `printStackTrace()` yordamı hatayı `System.err`kullanarak kullanıcıya iletir. Bunun ne gibi avantajları var derseniz hemen açıklayalım: Eğer bir uygulamanın çıktısını dosyaya veya buna benzer bir yere yönlendirmiş iseniz `System.out` kullanarak yazılmış ifadeler yine bu dosyalara ve buna benzer yerlere yazılacaktır. (yorum ekle)

```
$ java Test > a.txt
```

Fakat `System.err` kullanılarak yazılmış bir ifade, uygulama nereye yönlendirilmiş olursa olsun kesin olarak konsola yazılır ve kullanıcının dikkatine sunulur. (yorum ekle)

`printStackTrace (PrintStream s)`
PrintStream sınıfına ait nesne kullanılarak, oluşan istisnanın yol haritasını konsol yerine başka bir yere bastırmanız mümkündür. Başka bir yer derken, örneğin bir dosya veya ağ (*network*) bağlantısı ile başka bir bilgisayara oluşan bu istisnanın yol haritasını gönderebilirsiniz. (yorum ekle)

`printStackTrace (PrintWriter s)`
PrintWriter sınıfına ait nesne kullanılarak, oluşan istisnanın yol haritasını konsol yerine başka bir yere bastırmanız mümkündür. Özellikle JSP ve Servlet kullanırken oluşan bir istisnanın yol haritasını HTTP/HTTPS kanalı ile kullanıcılara gösterilebilir. (yorum ekle)
IstisnaMetodlari4.java uygulamamızın çıktısı aşağıdakidir.

```
oku() yordamı cagrildi
cokOku() yordamı cagrildi
java.lang.Exception: istisna olustu
  at IstisnaMetodlari4.cokOku(IstisnaMetodlari4.java:10)
  at IstisnaMetodlari4.oku(IstisnaMetodlari4.java:15)
  at IstisnaMetodlari4.main(IstisnaMetodlari4.java:22)
```

`Throwable fillInStackTrace()`
Oluşan bir istisnanın yol haritasını *Throwable* nesnesi içerisinde elde etmeniz için `fillInStackTrace()` yordamını kullanmalısınız. Bu olay istisnanın tekrardan fırlatılması söz konusu olduğunda - biraz sonra inceleyeceğiz - faydalı olabilir. (yorumekle)

Örnek: *IstisnaMetodlari5.java* (yorum ekle)

```
public class IstisnaMetodlari5 {

 public void cokOku() throws Exception {
   System.out.println("cokOku() yordamı cagrildi");
   throw new Exception("istisna olustu");
 }
 public void oku() throws Exception {
   System.out.println("oku() yordamı cagrildi");
   cokOku();
 }
 public static void main(String args[]) {
   try {
     IstisnaMetodlari5 im5 = new IstisnaMetodlari5();
     im5.oku();
```

```
    } catch (Exception ex) {
      Throwable t = ex.fillInStackTrace();
      System.err.println( t.getMessage() );
    }
  }
}
```

Bu method oluşan istisnanın yol haritasına müdahale ederek değiştirir ve değiştirilen bilgiler ışığında yeni bir *Throwable* nesnesi oluşturulur. (yorum ekle)
Uygulamanın çıktısı aşağıdaki gibidir.

```
oku() yordamı cagrildi
cokOku() yordamı cagrildi
istisna olustu
```

`StackTraceElement[] getStackTrace()`

Yine Java 1.4 ile birlikte gelen `getStackTrace()` yordamı, `printStackTrace()` yordamı ile oluşan hata satırlarını *StackTraceElement* tipindeki dizi nesnesine çevirir. (yorum ekle)

Örnek: *IstisnaMetodlari6.java* (yorum ekle)

```
public class IstisnaMetodlari6 {

 public void cokOku() throws Exception {
   System.out.println("cokOku() yordamı cagrildi");
   throw new Exception("istisna olustu");
 }

 public void oku() throws Exception {
   System.out.println("oku() yordamı cagrildi");
   cokOku();
 }
 public static void main(String args[]) {
   try {
     IstisnaMetodlari6 im6 = new IstisnaMetodlari6();
     im6.oku();
   } catch (Exception ex) {
     StackTraceElement[] ste = ex.getStackTrace(); // dikkat
     for(int i=0 ;i < ste.length;i++) {
       System.err.println("-->"+ ste[i].getFileName() +" - "+
                                 ste[i].getMethodName() +" - "+
                                 ste[i].getLineNumber() );
     }
   }
 }
}
```

Oluşan istisnanın yol haritası bilgilerine ulaşmak için `getStackTrace()` yordamı kullanılmalıdır. Bu yordam, oluşan istisnaya ait yol bilgilerini bir *StackTraceElement* dizisi şeklinde sunar. `printStackTrace()` yordamının çıktısı göz önüne getirirsek, buradaki ilk satır, *StackTraceElement* dizisinin ilk elemanına denk gelir. Bu dizinin son elemanında, oluşan istisna yol haritasının son satırına denk gelir. (yorum ekle)
Uygulamamızın çıktısı aşağıdaki gibi olur.

```
oku() yordamı cagrildi
cokOku() yordamı cagrildi
--> IstisnaMetodlari6.java - cokOku - 8
--> IstisnaMetodlari6.java - oku - 13
```

```
--> IstisnaMetodlari6.java - main - 20
```

`setStackTrace (StackTraceElement[] stackTrace)`
Son olarak inceleyeceğimiz yordam yine Java 1.4 ile birlikte gelen `setStackTrace()` yordamıdır. Bu yordam sayesinde oluşan istisnanın yol haritası değiştirelebilir. (yorum ekle)

Örnek: *IstisnaMetodlari7.java* (yorum ekle)

```
public class IstisnaMetodlari7 {

 public void cokOku() throws Exception {
   System.out.println("cokOku() yordamı cagrildi");
   Exception eE = new Exception("istisna olustu-1"); // dikkat
   System.out.println("------------");
   Exception eE2 = new Exception("olusan istisna-2"); // dikkat
   eE2.setStackTrace( eE.getStackTrace() ); // dikkat
   throw eE2; // dikkat

 }

 public void oku() throws Exception {
   System.out.println("oku() yordamı cagrildi");
   cokOku();
 }

 public static void main(String args[]) {
   try {
     IstisnaMetodlari7 im7 = new IstisnaMetodlari7();
     im7.oku();
   } catch (Exception ex) {
     StackTraceElement[] ste = ex.getStackTrace(); // dikkat
     for(int i=0 ;i < ste.length;i++) {
       System.err.println("-->"+ ste[i].getFileName() +" - "+
                                 ste[i].getMethodName()+" - "+
                                 ste[i].getLineNumber() );
     }
   }
 }
}
```

Bu değişimden sonra `getStackTrace()` veya `printStackTrace()` gibi benzeri yordamlar artık değişen bu yeni yol haritasını basacaklardır. (yorum ekle)
Uygulamamızın çıktısı aşağıdaki gibi olur.

```
oku() yordamı cagrildi
cokOku() yordamı cagrildi
------------
--> istisnametodlari7.java - cokOku - 6
--> istisnametodlari7.java - oku - 16
--> istisnametodlari7.java - main - 23
```

Yukarıdaki örneğimizde fırlatılan istisnanın çıkış noktası 9. satırda olmasına rağmen, `setStackTrace()` yordamı kullanarak oluşan istisnanın yol haritasında değişiklik yapabildik. Artık fırlatılan istisnanın yeni çıkış noktasını 6. satır olarak gösterilmektedir. (yorum ekle)

8.1.7. **Kendi İstisnalarımızı Nasıl Oluşturabiliriz?**

Javanın kendi içerisinde tanımlanmış istisna tiplerinin dışında, bizlerde kendimize özgü istisna tiplerini oluşturup kullanabiliriz. Sonuçta istisnalar da birer nesnedir ve kendilerine has durumları ve özellikleri olabilir. İlk istisna sınıfını oluşturalım; (yorumekle)

Örnek: *BenimHatam.java* (yorum ekle)

```
public class BenimHatam extends Exception {
 private int id ;

 public BenimHatam() {
 }

 public BenimHatam(String aciklama) {
   super(aciklama); // dikkat
 }

 public BenimHatam(String aciklama , int id) {
   super(aciklama); //dikkat
   this.id = id ;
 }

 public String getLocalizedMessage() { // iptal etme (override)

   switch(id) {
     case 0 : return "onemsiz hatacik" ;
     case 1 : return "hata" ;
     case 2 : return "! onemli hata !" ;
     default: return "tanimsiz hata";
   }
 }

 public int getId() {
   return id;
 }
}
```

İlk istisna sınıfımızın ismi *BenimHatam*. Şimdi olaylara geniş açıdan bakıldığında görülmesi gereken ilk şey, bir sınıfın istisna tipleri arasında yer alabilmesi için *Exception* sınıfından türetilmesi gerekiğidir. (yorum ekle)

Gösterim-8.9:

```
public class BenimHatam extends Exception {  //..}
```

Exception sınıfından türetilen bir sınıf artık istisna sınıfları arasında yerini almaya hazırdır. Fakat önce bir kaç önemli noktayı gün ışığına çıkartalım. Öncelikle *BenimHatam* istisna sınıfının yapılandırıcılarına (*constructor*) dikkat etmenizi istiyorum.*BenimHatam* istisna sınıfının iki adet yapılandırıcısı (*constructor*) bulunmaktadır, bunlardan biri *String* tipinde diğeri ise bir *String* birde ilkel `int` tipinde parametre kabul etmektedir. Bu yapılandırıcıların ortak olarak aldıkları parametre tipi *String* tipidir. Niye? (yorum ekle)

BenimHatam istisna sınıfından anlaşılacağı üzere, bu sınıfımızın içerisinde aynı diğer sınıflarımızda oldu gibi yordamlar tanımlayabildik, örneğin `getId()` yordamı. Bu yordam hataya ait `Id` numarasını dönmektedir. "Hangi hata numarası, bu da ne ?" demeyin çünkü bunu yordamlara zenginlik katması için ekledim. (yorum ekle)
Bu sınıfımızın içerisinde *Throwable* (*Exception* sınıfının da *Throwable* sınıfından türetildiğini unutmayalım) sınıfının bir yordamı olan `getLocalizedMessage()` yordamını iptal ettik (*override*). Yukarıdaki açıklamalardan hatırlayacağınız üzere eğer`getLocalizedMessage()` yordamı iptal edilmez ise `getMessage()` ile aynı açıklamayı dönerdi (bkz: istisna yordamları). Fakat biz burada sırf heyacan olsun diye her numarayı bir açıklama ile eşleştirip geri döndürmekteyiz. Örneğin "sıfır=önemsiz hata", "bir= !önemli hata!" gibi gibi, tabii bunlarda tamamen hayal ürünü olarak yazılmıştır. (yorum ekle)
İstisna sınıflarının yapılandırıcılarına *String* ifade göndermenin amacı `getMessage()` yordamının yaptığı işi anlamaktan geçer. Tabii sadece istisna sınıfının yapılandırıcısına *String* tipte parametre göndermek ile iş bitmez. Bu gönderilen parametre eğer`super(String)` komutu çağrılırsa amacına ulaşır. Peki amaç nedir? Diyelim ki bir dosya açmak istediniz ama açılmak istenen dosya yerinde değil? Böyle bir durumda *java.io.FileNotFoundException* tipinde bir istisna nesnesi oluşturulup fırlatılacaktır. Oluşan istisna ile (dosyanin bulunamaması) oluşan istisna tipinin ismi (java.io.FileNot FoundException) arasında bir ilişki kurabiliyor musunuz? "FileNotFoundException" ifadesinin Türkçesi "dosya bulunamadı istisnası" demektir, bingo!. Böyle bir durumda oluşan istisnayı, bu istisna için oluşturulan bakarak aklımızda bir ışık yanabilir. Peki her istisna tipinin ismi bu kadar açıklayıcı mıdır? Örneğin *SQLException*, bu ifadenin Türkçesi "SQL istisnası" demektir. Böyle bir istisna bazı zamanlarda anlamsız gelebilir. Evet SQL istisnası çok güzel ama niye? Ne ters gitti de ben bu hatayı aldım, ek açıklama yok mu diyeceğimiz anlar olmuştur ve olacaktır. Sonuçta ek açıklamalara ihtiyaç olduğu bir gerçektir. İşte bu ek açıklamaları bu istisna sınıflarının yapılandırıcılarına gönderebiliriz. Böylece oluşan istisna nesnesinin ismi ve bizim vereceğimiz ek açıklamalar ile sır perdesini aralayabiliriz. Ek açıklamanın *String* tipinde olmasına herhalde kimsenin bir itirazı yoktur. İşte bu yüzden iyi tasarlanmış bir istisna sınıfının *String* tipinde parametre kabul eden yapılandırıcıları vardır. Ama ek olarak yukarıdaki örneğimizde olduğu gibi hem *String* hem de ilkel (*primitive*) `int` tipinde parametre kabul eden yapılandırıcılar olabilir. Fazla parametre göz çıkartmaz. (yorum ekle)
Şimdi ikinci istisna sınıfımız olan *SeninHatan* sınıfını inceleyelim;

Örnek: *SeninHatan.java* (yorum ekle)

```
public class SeninHatan extends Exception {
 public SeninHatan() {
 }
 public SeninHatan(String aciklama) {
   super(aciklama); // dikkat
 }
}
```

SeninHatan istisna sınıfı bir öncekine (*BenimHatam*) göre daha sadedir. Şimdi tek eksiğimiz bu istisna sınıflarımızın kullanıldığı bir kobay örnek. Onu da hemen yazalım. (yorum ekle)

Örnek: *Kobay.java* (yorum ekle)

```
public class Kobay {
 public void cikart(int a,int b) throws BenimHatam, SeninHatan{
   if(a == 0) {
```

```
      throw new SeninHatan("a parametresi sifir geldi");
    }
    if(b == 0) {
      throw new SeninHatan("b parametresi sifir geldi");
    }
    if( (a<0) || (b<0) ) {
      throw new SeninHatan(); // kotu, aciklama yok
    }
    int sonuc = a - b ; // hesaplama islemi

    if(sonuc < 0) {
      throw new BenimHatam("sonuc eksi",2);
    }else if( sonuc == 0) {
      throw new BenimHatam("sonuc sifir",1);
    }
  }
  public static void main(String args[]) {
    System.out.println("----------------------");
    try {
      Kobay it = new Kobay();
      it.cikart(1,2);
    } catch (BenimHatam ex1) {
      System.out.println( "Hata Olustu-1:"+ ex1.getMessage() );
      System.out.println(ex1.getLocalizedMessage());
      System.out.println(ex1.getId());
    } catch (SeninHatan ex2) {
      System.out.println("Hata Olustu-2:"+ ex2);
    }
    System.out.println("----------------------");
    try {
      Kobay it = new Kobay();
      it.cikart(1,0);
    } catch (BenimHatam ex1) {
      System.out.println("Hata Olustu-1:"+ ex1.getMessage());
      System.out.println(ex1.getLocalizedMessage());
      System.out.println(ex1.getId());
    } catch (SeninHatan ex2) {
      System.out.println("Hata Olustu-2:"+ ex2);
    }

    System.out.println("----------------------");
    try {
      Kobay it = new Kobay();
      it.cikart(1,-124);
    } catch (BenimHatam ex1) {
      System.out.println("Hata Olustu-1:"+ ex1.getMessage());
      System.out.println(ex1.getLocalizedMessage());
      System.out.println(ex1.getId());
    } catch (SeninHatan ex2) {
      System.out.println("Hata Olustu-2:"+ ex2);
    }
  }
}
```

Yukarıdaki örnekte üç adet harekete kızılmaktadır. Bunlar sırasıyla:

- Sonucun eksi çıkması durumunda *BenimHatam* tipinde istisna oluşmaktadır. (yorum ekle)
- Parametrelerden birinin sıfır gönderilmesi durumunda *SeninHatan* tipinde istisna oluşmaktadır (yorum ekle)
- Parametrelerden birinin eksi gönderilmesi durumunda *SeninHatan* tipinde istisna oluşmaktadır (yorum ekle)

Eğer *BenimHatam* tipinde bir istisna oluşursa nasıl detaylı bilgi alınacağına lütfen dikkat edin. Aynı şekilde *SeninHatan* tipinde bir istisna oluşursa ekrana sadece `toString()` yordamından geri dönen açıklama gönderilecektir. *SeninHatan* istisnasının fırlatıldığı yerlere dikkat ederseniz, ek açıklamaların ne kadar hayati bir önem taşıdığını göreceksiniz. Uygulamanın çıktısı aşağıdaki gibidir. (yorum ekle)

```
----------------------Hata Olustu-1:sonuc eksi! onemli hata !2------------
----------Hata Olustu-2:SeninHatan: b parametresi sifir geldi--------------
--------Hata Olustu-2:SeninHatan
```

SeninHatan istisna tipinin nasıl meydana geldiğini gönderilen ek açıklama ile daha iyi kavrayabiliyoruz ama son `try-catch` bloğunda yakalanan *SeninHatan* istisna tipinin sebebi açık değildir. Ortada bir istisna vardır ama bu istisnayı nasıl giderebileceğimiz konusunda bilgi yoktur. Bu uygulama karmaşık olmadığı için kolaylıkla " burada oluşan istisnanın sebebi parametrenin eksi gönderilmesidir " diyebilirsiniz ama çok daha büyük uygulamalarda bu tür çıkarımlar yapmak zannedildiği kadar kolay olmayabilir. (yorum ekle)

8.1.8. **finally Bloğu**

Bir işlemin her koşulda - istisna olsun ya da olmasın - kesin olarak yapılmasını istiyorsak finally bloğu kullanmalıyız. (yorum ekle)

Gösterim-8.10:

```
try {
 // riskli kod
   // bu kod BenimHatam,SeninHatan
   // OnunHatasi, BizimHatamiz
   //  tipinde istisnalar firlatabilir
} catch(BenimHatam bh) {
  //  BenimHatam olusursa buraya
} catch(SeninHatan sh) {
  //  SeninHatan olusursa buraya
} catch(OnunHatasi oh) {
  //  OnunHatasi olusursa buraya
} catch(BizimHatamiz bizh) {
  //  BizimHatamiz olusursa buraya
} finally {
  // ne olursa olsun calisacak kod buraya
}
```

Ne olursa olsun çalışmasını istediğiniz kodu `finally` bloğuna yazabilirsiniz. Bir uygulama üzerinde açıklanmaya çalışılırsa. (yorum ekle)

Örnek: *FinallyOrnek1.java* (yorum ekle)

```
public class FinallyOrnek1 {
 public static void a(int deger) throws  SeninHatan {
   if(deger < 0 ) {
     throw new SeninHatan();
```

```
    }
  }
  public void hesapla() {
    for(int i=-1 ; i < 1 ; i++ ) {
      try {
        System.out.println("a() cagriliyor");
        a(i);
      } catch(SeninHatan shEx) {
        System.out.println("SeninHatan olustu : " + shEx);
      } finally {
        System.out.println("finally blogu calistirildi");
      }
    }
  }
  public static void main(String args[]) {
    FinallyOrnek1 fo1 = new FinallyOrnek1();
    fo1.hesapla();
  }
}
```

Bu uygulamada dikkat edilmesi gereken yer `hesapla()` yordamıdır. `for` döngüsü –1'den 0 'a kadar ilerlemektedir, ilerleyen bu değerler `a()` yordamına parametre olarak gönderilmektedir. `a()` yordamının içerisinde ise gelen parametrenin değeri kontrol edilip eğer bu değer 0 dan küçükse *SeninHatan* istisnası fırlatılmaktadır. Buradaki amaç bir istisnalı birde istisnasız koşulu yakalayarak; her koşulda `finally` bloğuna girildiğini ispatlamaktır. Uygulamanın çıktısı aşağıdaki gibidir. (yorum ekle)

```
 a() cagriliyor
SeninHatan olustu: SeninHatan
finally blogu calistirildi
a() cagriliyor
finally blogu calistirildi
```

Ayrıca `finally` bloğunun daha bir çok faydası bulunur. Örneğin birşey aramak için geceleyin bir odaya girdiğinizde ilk olarak ne yaparsanız? Genelleme yaparak ışığı yakarsanız diyelim. Aynı şekilde odanın içerisinde arama işlemi bittiğinde ve odaya terk edeceğiniz zaman ne yaparsanız? Açık olan ışığı kapatırsınız değil mi? Sonuçta odadan çıkarken ışığın kapatılması gerekir bunun her zaman olması gereken bir davranış olarak kabul edip aşağıdaki uygulamayı inceleyelim. (yorum ekle)

Örnek: *Oda.java* (yorum ekle)

```
 class BeklenmeyenHata1 extends Exception {
  public BeklenmeyenHata1(String ekAciklama) {
    super(ekAciklama);
  }
}

class BeklenmeyenHata2 extends Exception {
  public BeklenmeyenHata2(String ekAciklama) {
    super(ekAciklama);
  }
}

public class Oda {
  public void isiklariKapat() {
    System.out.println("isiklar kapatildi");
  }
  public void isiklariAc() {
```

```
    System.out.println("isiklar acildi");
  }
  public void aramaYap() throws BeklenmeyenHata1, BeklenmeyenHata2 {
      // istisna firlatabilecek olan govde
      //...
  }
  public void basla() {
    try {
      // riskli kod
      isiklariAc();
      aramaYap();
      isiklariKapat(); // dikkat
    } catch(BeklenmeyenHata1 bh1) {
      System.out.println("BeklenmeyenHata1 yakalandi");
      isiklariKapat(); // dikkat
    } catch(BeklenmeyenHata2 bh2) {
      System.out.println("isiklar acildi");
      isiklariKapat(); // dikkat
    }
  }
  public static void main(String args[]) {
    Oda o = new Oda();
    o.basla();
  }
}
```

Bu örneğimizde `basla()` yordamında gelişen olaylara dikkat edelim. Karanlık bir odada arama yapmak için ilk önce ışıkları açıyoruz daha sonra aramayı gerçekleştiriyoruz ve en sonunda ışıkları kapatıyoruz. Fakat dikkat edin ışıkların kapanmasını garantilemek için üç ayrı yerde `isiklariKapat()` yordamı çağrılmaktadır, peki ama niye? (yorum ekle)
`try` bloğunun içerisine yerleştirilen `isiklariKapat()` yordamı, eğer herşey yolunda giderse çağrılacaktır. *BeklenmeyenHata1* istisnasının
yakalandığı `catch` bloğundaki `isiklariKapat()` yordamı, eğer *BeklenmeyenHata1* istisnası oluşursa ışıkların kapatılması unutulmasın diye yerleştirilmiştir. Aynı şekilde *BeklenmeyenHata2* istisnasının
yakalandığı `catch` bloğundaki `isiklariKapat()` yordamı, eğer *BeklenmeyenHata2* istisnası oluşursa ışıkların kapatılmasını garantilemek amacı için buraya yerleştirilmiştir. Bir işi yapabilmek için aynı kodu üç farklı yere yazmak ne kadar verimlidir olabilir? Daha karmaşık bir yapıda belki de ışıkları söndürmek unutulabilir. İşte böyle bir durumda `finally` bloğu hem verimliliği artırmak hem de çalışması istenen kodun çalışmasını garantilemek amacıyla kullanılabilir. Yukarıdaki uygulama örneğimizin doğru versiyonunu tekrardan yazarsak. (yorum ekle)

Örnek: *Oda2.java* (yorum ekle)

```
public class Oda2 {
 public void isiklariKapat() {
   System.out.println("isiklar kapatildi");
 }
 public void isiklariAc() {
   System.out.println("isiklar acildi");
 }
 public void aramaYap() throws BeklenmeyenHata1, BeklenmeyenHata2 {
     // istisna firlatabilecek olan govde
 }
 public void basla() {
   try {
     // riskli kod
     isiklariAc();
     aramaYap();
   } catch(BeklenmeyenHata1 bh1) {
     System.out.println("BeklenmeyenHata1 yakalandi");
```

```
    } catch(BeklenmeyenHata2 bh2) {
      System.out.println("isiklar acildi");
    } finally {
      isiklariKapat(); // dikkat
    }
  }
  public static void main(String args[]) {
    Oda2 o2 = new Oda2();
    o.basla();
  }
}
```

Bu uygulama örneğimizde `isiklariKapat()` yordamı sadece `finally` bloğunun içerisine yazılarak her zaman ve her koşulda çalıştırılması garantili hale getirilmiştir. Artık herhangi bir istisna oluşsun veya oluşmasın ışıklar kesin olarak söndürülecektir. (yorum ekle)
`finally` bloğunun kesin olarak çağrıldığını aşağıdaki uygulamamızdan da görebiliriz. (yorum ekle)

-

Örnek: *FinallyOrnek2.java* (yorum ekle)

```
public class FinallyOrnek2 {

 public static void main(String args[]) {
   try {
     System.out.println("1- try blogu");
     try {
       System.out.println("2- try blogu");
       throw new Exception();
     } finally {
       System.out.println("2- finally blogu");
     }
   } catch(Exception ex) {
     System.out.println("1- catch blogu");
   } finally {
     System.out.println("1- finally blogu");
   }
  }
}
```

Bu örneğimizde dip taraftaki `try` bloğunun içerisinde bir istisna oluşturulmuştur. Dip taraftaki `try` bloğunun `catch` mekanizması olmadığı için bu oluşan istisna dış taraftaki `catch` mekanizması tarafından yakalanacaktır. Fakat bu yakalanma işleminin hemen öncesinde dipte bulunan `finally` bloğunun içerisindeki kodlar çalıştırılacaktır. Uygulamanın çıktısı aşağıdaki gibidir. (yorum ekle)

```
1- try blogu
2- try blogu
2- finally blogu
1- catch blogu
1- finally blogu
```

8.1.8.1. **return ve finally Bloğu**

`finally` bloğu her zaman çalıştırılır. Örneğin bir yordam hiçbir şey döndürmüyorsa (*void*) ama bu yordamın içerisinde yordamı sessizce terk etmek amacı ile `return` ifadesi kullanılmış

ise, `finally` bloğu içerisindeki kodlar bu `return` ifadesi devreye girmeden hemen önce çalıştırılır. Uygulama üzerinde gösterilirse. (yorum ekle)

Örnek: *ReturnOrnek.java* (yorum ekle)

```
public class ReturnOrnek {
 public void calis(int deger) {
   try {
     System.out.println("calis yordamı cagrildi, gelen deger: "
                                                          + deger);
     if(deger == 0) {
       return; // yordamı sessizce terk et
     }
     System.out.println("-- calis yordamı normal bir sekilde bitti--");
   } catch (Exception ex) {
     System.out.println("catch blogu icerisinde");
   } finally {
     System.out.println("finally blogu cagrildi");
     System.out.println("----------------------");
   }
 }
 public static void main(String args[]) {
   ReturnOrnek ro = new ReturnOrnek();
   ro.calis(1);
   ro.calis(0); // dikkat
 }
}
```

`calis()` yordamına gönderilen parametre eğer sıfırsa, bu yordam çalışmasını sona erdiriyor fakat `finally` bloğu içerisindeki kodlar bu durumda bile çalıştırılmaktadır. Dikkat edilmesi gereken bir başka nokta ise `calis()` yordamının bilerek birşey döndürmemesidir -void- olmasıdır. Çünkü eğer `calis()` yordamı birşey - ör: *String* tipi- döndüreceğini söyleseydi, geri döndürme (*return*) işlemini `finally` bloğunun içerisinde yapması gerekirdi, aksi takdirde derleme anında (*compile-time*) uyarı alınırdı. Yani bir yordamın içerisinde `try - finally` blok sistemi tanımlanmış ise `try` bloğunda `return` ile bir değer geri döndürülmesine izin verilmez. Uygulamanın çıktısı aşağıdaki gibidir. (yorum ekle)

```
 calis yordamı cagrildi, gelen deger: 1-- calis yordamı normal bir sekilde
bitti--finally blogu cagrildi----------------------calis yordamı cagrildi,
gelen deger: 0finally blogu cagrildi----------------------
```

8.1.8.2. **Dikkat System.exit();**

Eğer *System* sınıfının statik bir yordamı olan `exit()` çağrılırsa `finally` bloğuna hiç girilmez. `System.exit()` yordamı uygulamanın içerisinde çalıştığı JVM'i (*Java virtual machine*) kapatır. Anlatılanları bir uygulama üzerinde incelersek. (yorumekle)

Örnek: *SystemExitOrnek.java* (yorum ekle)

```
public class SystemExitOrnek {
 public void calis(int deger) {
   try {
     System.out.println("calis yordamı cagrildi, gelen deger: "
                                                          + deger);
     if(deger == 0) {
       System.exit(-1); // JVM'i kapat
     }
     System.out.println("-- calis yordamı normal bir sekilde bitti--");
   } catch (Exception ex) {
     System.out.println("catch blogu icerisinde");
```

```
    } finally {
      System.out.println("finally blogu cagrildi");
      System.out.println("----------------------");
    }
  }
  public static void main(String args[]) {
    SystemExitOrnek seo = new SystemExitOrnek();
    seo.calis(1);
    seo.calis(0); // dikkat
  }
}
```

Bu örneğimizin bir öncekine göre tek farkı `return` yerine `System.exit()` komutunun yazılmış olmasıdır. `System.exit()` komutu, uygulamanın içerisinde çalıştığı JVM'i kapattır. `exit()` yordamına gönderilen eksi bir değer JVM'in anormal bir sonlanmış yapacağını ifade `eder`. Bu çok ağır bir cezalandırmadır. Normalde uygulamanın bu şekilde sonlandırılması pek tercih edilmemektedir ancak tek başına çalışan (*standalone*) uygulamalarda kullanıcının yanlış parametre girmesi sonucu kullanılabilir. (yorum ekle)

8.1.9. İstisnanın Tekrardan Fırlatılması

Oluşan bir istisnayı `catch` bloğunda yakaladıktan sonra tekrardan bir üst kısma fırlatmanız mümkündür. Genel gösterim aşağıdaki gibidir. (yorum ekle)

Gösterim-8.11:

```
try {
  // riskli kod
} catch (Exception ex){
  System.out.println("istisna yakalandi: " + ex);
  throw ex; // dikkat
}
```

Oluşan bir istisnayı bir üst kısma fırlatırken istisna nesnesinin içerisindeki bilgiler saklı kalır. Bir uygulama üzerinde incelersek. (yorum ekle)

Örnek: *TekrarFirlatimOrnek1.java* (yorum ekle)

```
public class TekrarFirlatimOrnek1 {
 public void cokCalis() throws Exception {
   try {
     throw new Exception("oylesine bir istisna"); // istisnanin olusumu
   } catch(Exception ex) {
     System.out.println("cokCalis() istisna yakalandi: " + ex);
     throw ex; // dikkat
   }
 }
 public void calis() throws Exception {
   try {
     cokCalis();
   } catch(Exception ex) {
     System.out.println("calis() istisna yakalandi: " + ex);
     throw ex; // dikkat
   }
 }
 public void basla() {
   try {
     calis();
   } catch(Exception ex) {
```

```
      ex.printStackTrace(); // bilgi alimi
    }
  }
  public static void main(String args[]) {
    TekrarFirlatimOrnek1 tfo1 = new TekrarFirlatimOrnek1();
    tfo1.basla();
  }
}
```

Yukarıdaki örneğimizde istisna `cokCalis()` yordamının içerisinde oluşmaktadır. Oluşan bu istisna `catch` mekanizması sayesinde yakalandıktan sonta tekrardan bir üst kısma fırlatılmaktadır. `calis()` yordamının içerisinde de aynı şekilde fırlatılan istisna`catch` mekanizması sayesinde yakalanıp tekrardan bir üst kısma fırlatılmaktadır. `basla()` yordamına kadar gelen istisna nesnesi burada yakalanıp içerisinde saklı bulunan bilgiler `printStackTrace()` yordamıyla gün ışığına çıkarılmaktadır. Uygulamanın çıktısı aşağıdaki gibidir. (yorum ekle)

```
cokCalis() istisna yakalandi: java.lang.Exception:
                                oylesine bir istisna
calis() istisna yakalandi: java.lang.Exception:
                                oylesine bir istisna
java.lang.Exception: oylesine bir istisna
  at TekrarFirlatimOrnek1.cokCalis(TekrarFirlatimOrnek1.java:7)
  at TekrarFirlatimOrnek1.calis(TekrarFirlatimOrnek1.java:17)
  at TekrarFirlatimOrnek1.basla(TekrarFirlatimOrnek1.java:29)
  at TekrarFirlatimOrnek1.main(TekrarFirlatimOrnek1.java:38)
```

Dikkat edilirse oluşan istisnaya ait bilgiler `basla()` yordamının içerisinde ekrana basılmasına karşın, orijinalliğini hiç kaybetmedi. Orijinallikten kasıt edilen istisnanın gerçekten nerede oluştuğu bilgisidir. Oluşan bir istisnayı yakalayıp yeniden fırlatmadan evvel, onun içerisindeki bilgilere müdahale etmeniz mümkündür. Şöyle ki... (yorum ekle)

Örnek: *TekrarFirlatimOrnek2.java* (yorum ekle)

```
public class TekrarFirlatimOrnek2 {
  public void cokCalis() throws Exception {
    try {
      throw new Exception("oylesine bir istisna");
    } catch(Exception ex) {
      System.out.println("cokCalis() istisna yakalandi: " + ex);
      throw ex; // dikkat
    }
  }
  public void calis() throws Throwable {
    try {
      cokCalis();
    } catch(Exception ex) {
      System.out.println("calis() istisna yakalandi: " + ex);
      throw ex.fillInStackTrace(); // dikkat
    }
  }
  public void basla() {
    try {
```

```
      calis();
    } catch(Throwable th) {
      th.printStackTrace(); // döküm
    }
  }
  public static void main(String args[]) {
    TekrarFirlatimOrnek2 tfo2 = new TekrarFirlatimOrnek2();
    tfo2.basla();
  }
}
```

Bu örneğimizde istisnanın orijinal oluşma yeri `cokCalis()` yordamıdır ama `calis()` yordamı içerisinde, istisna nesnesinin içindeki bilgilere `fillInStackTrace()` müdahale edilip değiştirmektedir. Uygulamanın çıktısı aşağıdaki gibidir. (yorum ekle)

```
 cokCalis() istisna yakalandi: java.lang.Exception:
                                          oylesine bir istisna
calis() istisna yakalandi: java.lang.Exception:
                                              oylesine bir istisna
java.lang.Exception: oylesine bir istisna
  at TekrarFirlatimOrnek2.calis(TekrarFirlatimOrnek2.java:20)
  at TekrarFirlatimOrnek2.basla(TekrarFirlatimOrnek2.java:28)
  at TekrarFirlatimOrnek2.main(TekrarFirlatimOrnek2.java:36)
```

Artık istisnanın oluşma yeri olarak 20. satırı yani `fillInStackTrace()` yordamının devreye girdiği yer gösterilmektedir. Böylece oluşan istisnanın içerisindeki bilgilere müdahale etmiş bulunmaktayız. `calis()` yordamında niye *Throwable* tipinde bir istisna fırlatıldığına gelince, bunun sebebi `fillInStackTrace()` yordamının *Throwable* tipinde bir istisna nesnesi geri döndürmesidir. Bu sebebten dolayı `basla()` yordamının içerisindeki `catch` bloğunda *Exception* istisna tipi yerine *Throwable* tipi belirtilmiştir. Eğer bu `catch` bloğunda *Exception* tipi belirtilseydi derleme anında (*compile-time*) *Throwable* yakalanmalı diye hata alınırdı. Şekil-8.5.'e dikkat ederseniz *Throwable* istisna tipi en üste bulunmaktadır. Bunun anlamı eğer bir *Throwable*tipinde istisna fırlatılmış ise bunu kesin olarak `catch` bloğunun içerisinde *Throwable* tipi belirterek yakalayabileceğimizdir. (yorum ekle)

Örnek: Rutbe.java (yorum ekle)

```
public class Rutbe {
  public static void main(String args[]) {
    try {
      throw new Throwable();
    } catch ( Exception ex ) {
      System.out.println(" istisna yakalandi: " + ex);
    }
  }
}
```

Yukarıdaki örnek derlenmeye (*compile*) çalışılırsa; aşağıdaki hata mesajı ile karşılaşılır:

```
Rutbe.java:7: unreported exception java.lang.Throwable; must be
caught or declared to be thrown
           throw new Throwable();
```

```
            ^
1 error
```

Bunun anlamı, *Throwable* tipindeki fırlatılmış bir istisna nesnesini `catch` bloğunun içerisinde *Exception* tipi belirtilerek yakalanamayacağıdır. (yorum ekle)

8.1.10. printStackTrace() ve Hata Mesajlarının Kısaltılması

Java 1.4 ile beraber gelen bir başka özellik ise *Throwable* sınıfının yapılandırıcısına bir başka istisna tipini parametre olarak gönderebiliyor olmamızdır. Bu özellikten daha evvel bahsetmiştik, esas ilginç olan bu özelliğin fazla kullanılmasıyla aynı hata mesajlarının tekrarlamasıdır. Java tekrarlayan bu hata mesajları için bir kısaltma kullanır. (yorum ekle)

Örnek: *Kisaltma.java* (yorum ekle)

```
class YuksekSeviyeliIstisna extends Exception {
      YuksekSeviyeliIstisna(Throwable cause) {
               super(cause);
      }
}

class OrtaSeviyeliIstisna extends Exception {
      OrtaSeviyeliIstisna(Throwable cause) {
               super(cause);
      }
}

class DusukSeviyeliIstisna extends Exception {
}

public class Kisaltma {
      public static void main(String args[]) {
               try {
                        a();
               } catch(YuksekSeviyeliIstisna e) {
                        e.printStackTrace();
               }
      }
      static void a() throws YuksekSeviyeliIstisna {
               try {
                        b();
               } catch(OrtaSeviyeliIstisna e) {
                        throw new YuksekSeviyeliIstisna(e);
               }
      }
      static void b() throws OrtaSeviyeliIstisna {
               c();
      }
      static void c() throws OrtaSeviyeliIstisna {
               try {
                        d();
               } catch(DusukSeviyeliIstisna e) {
                        throw new OrtaSeviyeliIstisna(e);
               }
      }
      static void d() throws DusukSeviyeliIstisna {
               e();
      }
      static void e() throws DusukSeviyeliIstisna {
               throw new DusukSeviyeliIstisna(); // baslangic
      }
}
```

Yukarıdaki örneğimizde üç adet istisna tipi bulunmaktadır. `e()` yordamının içerisinde başlayan istisnalar zinciri `main()` yordamının içerisinde son bulmaktadır. Buradaki olay oluşan bir istisnayı diğerine ekleyerek aynı tip hata mesajları elde etmektir. Uygulamanın çıktısı aşağıdaki gibidir. (yorum ekle)

```
YuksekSeviyeliIstisna: OrtaSeviyeliIstisna: DusukSeviyeliIstisna
  at Kisaltma.a(Kisaltma.java:29)
  at Kisaltma.main(Kisaltma.java:20)
Caused by: OrtaSeviyeliIstisna: DusukSeviyeliIstisna
  at Kisaltma.c(Kisaltma.java:39)
  at Kisaltma.b(Kisaltma.java:33)
  at Kisaltma.a(Kisaltma.java:27)
  ... 1 more
Caused by: DusukSeviyeliIstisna
  at Kisaltma.e(Kisaltma.java:46)
  at Kisaltma.d(Kisaltma.java:43)
  at Kisaltma.c(Kisaltma.java:37)
  ... 3 more
```

Uygulamanın çıktısından da anlaşılacağı üzere, tekrar eden kısmın kaç kere tekrar ettiği bilgisi de verilmektedir. Mesela:

at Kisaltma.c(Kisaltma.java:39)
at Kisaltma.b(Kisaltma.java:33)
at Kisaltma.a(Kisaltma.java:27)

Yukarıdaki kısım 1 kere tekrar etmiştir

```
at Kisaltma.e(Kisaltma.java:46)
at Kisaltma.d(Kisaltma.java:43)
at Kisaltma.c(Kisaltma.java:37)
```

Bu kısım ise 3 kere tekrar etmiştir.

8.1.11. **İlginç Gelişme**

Oluşan bir istisna her zaman fırlatılmayabilir. Aşağıdaki uygulamamızı inceleyelim (yorum ekle)

Örnek: *FirlatimOrnek1.java* (yorum ekle)

```
public class FirlatimOrnek1 {

 public void basla(int a, int b) throws Exception {
     int       sonuc = 0;
     try {
       sonuc = a / b;
     } catch(Exception ex) {
       System.out.println("basla() istisna yakalandi");
       throw ex;
     } finally {
       System.out.println("sonuc: "+ sonuc);
     }
 }
```

```
  public static void main(String args[]) {
      try {
        FirlatimOrnek1 fo1 = new FirlatimOrnek1();
        fo1.basla(1,0);
      } catch(Exception ex) {
        System.out.println("main() istisna yakalandi");
      }
  }
}
```

Yukarıdaki örneğimizde akışın nasıl olmasını bekleriz ? İlkel (*primitive*) `int` tipinde bir sayının sıfıra bölünmesi sonuçu *ArithmeticException* tipinde bir istisna oluşur aynı bizim bu örneğimizde olduğu gibi. Daha sonra ekrana `finally` bloğunun içerisinde tanımlanmış ifade yazılır ve en son olarak istisna nesnesi bir üst kısma fırlatılır. Herşey beklendiği gibi gitmekte! Uygulamamızın çıktısı aşağıdaki gibidir. (yorum ekle)

```
basla() istisna yakalandi
sonuc: 0
main() istisna yakalandi
```

Önce `basla()` yordamının içerisinde yakalanan istisna, `finally` bloğunun çalıştırılmasından sonra bir üst kısma fırlatılabilmiştir. Fırlatılan bu istisna `main()` yordamı içerisinde yakalanmaktadır. (yorum ekle)
Peki ya `basla()` yordamı bir değer döndürseydi olaylar nasıl değişirdi? (yorum ekle)

Örnek: *FirlatimOrnek2.java* (yorum ekle)

```
public class FirlatimOrnek2 {
 public int basla(int a, int b) throws Exception {
      int sonuc = 0;
      try {
        sonuc = a / b;
      } catch(Exception ex) {
        System.out.println("basla() istisna yakalandi");
        throw ex;
      } finally {
        System.out.println("sonuc: "+ sonuc);
        return sonuc; // dikkat
      }
 }
 public static void main(String args[]) {
      try {
        FirlatimOrnek2 fo2 = new FirlatimOrnek2();
        fo2.basla(1,0);
      } catch(Exception ex) {
        System.out.println("main() istisna yakalandi");
      }
  }
}
```

Uygulamamızın çıktısı nasıl olacaktır? Bir önceki uygulama ile aynı mı? (yorum ekle)

```
basla() istisna yakalandisonuc: 0
```

Oluşan istisna, `basla()` yordamında yakalanmıştır ama daha sonra ortalardan kaybolmuştur. Aslında bu olay hata gibi algılanabilir ve haklı bir algılamadır. Fakat olaylara birde Java tarafından bakarsak anlayış gösterilebilir. Bir yordamın bir seferde sadece tek bir şey döndürme hakkı vardır. Ya bir değer döndürebilir veya bir istisna fırlatabilir, sonuçta fırlatılan

bir istisna da değer niteliği taşır. Bu uygulamamızda `basla()` yordamı `int` tipinde değer döndüreceğini söylediği ve `finally` bloğu kullanıldığı için, oluşan bir istisnanın tekrardan fırlatılması olanaksızdır. Bu işin bir çözümü var mı? Düşünelim... Bir yordam bir değer döndürse bile eğer bir istisna oluşursa bu oluşan istisnayı öncelikli olarak nasıl fırlatabilir? Böyle bir ikilem ile er ya da geç karşı karşıya kalınacaktır. Aşağıdaki gibi bir çözüm iş görecektir. (yorum ekle)

Örnek: *FirlatimOrnek3.java* (yorum ekle)

```
 public class FirlatimOrnek3 {  public int basla(int a, int b) throws
Exception {      public class FirlatimOrnek3 {
  public int basla(int a, int b) throws Exception {
      int sonuc = 0;
      Exception globalEx = null;
      try {
        sonuc = a / b;
      } catch(Exception ex) {
        System.out.println("basla() istisna yakalandi");
        globalEx = ex; // aktarim
      } finally {
        System.out.println("sonuc: "+ sonuc);
        if(globalEx != null) { // eger istisna olusmus ise
                throw globalEx; // tekrardan firlatim
        }
        return sonuc; // degeri geri dondur
      }
  }
  public static void main(String args[]) {
      try {
        FirlatimOrnek3 fo3 = new FirlatimOrnek3();
        fo3.basla(1,1);
        fo3.basla(1,0);
      } catch(Exception ex) {
        System.out.println("main() istisna yakalandi");
      }
  }
}
```

Yukarıdaki örneğimizde, eğer bir istisna oluşmuş ise *Exception* tipinde tanımlanan `globalEx` alanına, `catch` bloğu içerisinde değer aktarılmaktadır. `finally` bloğunun içerisinde `globalEx` alanına bir istisna nesnesinin bağlı olup olmadığı kontrol edilmektedir. Eğer `globalEx`, `null` değerinden farklıysa, bu `catch` bloğunda bir istisna nesnesine bağlandığı anlamına gelir yani bir istisnanın oluştuğunu ifade eder. Eğer `globalEx` `null` değerine eşitse sorun yok demektir. Böylece istisna oluşmuş ise`finally` bloğunda istisna fırlatılır, değilse de yordam normal dönmesi gereken değeri geri döndürür. Uygulamamızın çıktısı aşağıdaki gibidir. (yorum ekle)

```
sonuc: 1
basla() istisna yakalandi
sonuc: 0
main() istisna yakalandi
```

8.1.12. İptal Etme (Overrride) ve İstisnalar

İptal etme (*override*) konusunu 5. bölümde incelemişti. Bir sınıftan türetilen bir alt sınıfın içerisinde, üst (ana) sınıfa ait bir yordamı iptal edebilmesi için bir çok şart aranmaktaydı, bunlar sırasıyla, iptal eden yordamın, iptal edilen yordam ile aynı parametrelere, aynı isme ve üst sınıfa ait yordamın erişim belirleyicisinden daha erişilebilir veya aynı erişim belirleyicisine sahip olması gerekirdi. Buraya kadar anlattıklarımızda hemfikirsek esas soruyu sorabilirim; İptal edilme (*override*) ile istisnalar arasında bir bağlantı olabilir mi? Bu konu için bir başlık ayrıldığına göre herhalde bir bağlantı var ama nasıl? Bir uygulama üzerinde incelersek. (yorum ekle)

Örnek: AB.java (yorum ekle)

```
import java.io.*;
class A {
  public void basla() throws FileNotFoundException, EOFException {
      //...
  }
}
public class AB extends A {
  public void basla() throws IOException {
      //...
  }
}
```

AB.java uygulamasını derlemeye (*compile*) çalıştığımız zaman aşağıdaki hata mesajını alırız. (yorum ekle)

```
AB.java:12: basla() in AB cannot override basla() in A;
overridden method does not throw java.io.IOException
   public void basla() throws IOException {
               ^
1 error
```

Bu hata mesajının anlamı nedir? *AB* sınıfının içerisindeki `basla()` yordamının, *A* sınıfının içerisindeki `basla()` yordamını iptal edemediği çok açıktır, bunun sebebi erişim belirleyiciler olabilir mi? Hayır olamaz çünkü hem iptal eden hem de edilen yordam aynı erişim belirleyicisine sahip (public erişim belirleyicisine). Hımm peki sorun nerede? Sorun istisna tiplerinde. *A* sınıfına ait `basla()` yordamı iki adet istisna nesnesi fırlatıyor (*FileNotFoundException* ve *EOFException*) ama bunu iptal etmeye çalışan *AB* sınıfına ait `basla()` yordamı sadece bir tane istisna nesnesi fırlatıyor (*IOException*), sorun bu olabilir mi? (yorum ekle)

İptal edememe sorununu anlamak için Şekil-8.5.'deki yapıyı incelemek gerekir. Bu şeklimizden görüleceği üzere *FileNotFoundException* ve *EOFException* istisna tipleri, *IOException* istisna tipinden türetilmişlerdir. Kötü haberi hemen verelim, iptal ederken (*override*) artık yeni bir kuralımız daha oldu, şöyle ki: iptal edilen yordamının (*A* sınıfının içerisindeki `basla()` yordamı) fırlatacağı istisna tipi, iptal eden yordamın (*AB* sınıfı içerisindeki `basla()` yordamı) fırlatacağı istisna tiplerini kapsamalıdır. Aşağıdaki uygulamamız bu kuralı doğru bir şekilde yerine getirmektedir. (yorum ekle)

Örnek: *CD.java* (yorum ekle)

```
import java.io.*;
class C {
  public void basla() throws IOException {
      //...
  }
}
public class CD extends C {
  public void basla() throws FileNotFoundException, EOFException {
      //...
  }
}
```

İşte doğru bir iptal etme (*override*) örneği. C sınıfının `basla()` yordamı (iptal edilen) sadece bir adet istisna fırlatmaktadır (*IOException*) fakat *CD* sınıfının `basla()` yordamı (iptal eden) iki adet istisna fırlatmaktadır (*FileNotFoundException* ve*EOFException*). Buradan çıkarılacak sonuç doğru bir iptal etme işlemi için fırlatılan istisna sayısı değil, tiplerinin önemli olduğudur. Şekil-8.5.'e bir kez daha bakılırsa, *IOException* istisna tipinin, *FileNotFound Exception* ve *EOFException* istisna tiplerini kapsadığını görürsünüz; yani, *FileNotFoundException* ve *EOFException* tipinde istisna tipleri fırlatılırsa bu istisnaları *IOException* tipi ile `catch` bloğunda yakalanabilir ama bunun tam tersi olanaksızdır. Daha ilginç bir örnek verelim. (yorumekle)

Örnek: *EF.java* (yorum ekle)

```
 import java.io.*;
class E {
 public void basla() throws IOException {
      //...
 }
}
public class EF extends E {
  public void basla() {
      //...
  }
}
```

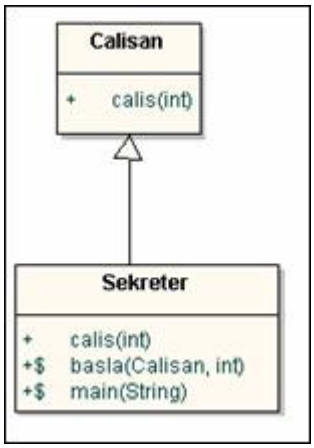
İptal edilen yordam *IOException* tipinde bir istisna fırlatmasına karşın, iptal eden yordamın hiç bir istisna fırlatmama lüksü vardır. İptal eden yordamın hiç bir istisna fırlatmaması bir soruna yol açmaz. Niye iptal edilen yordamın daha kapsamlı bir istisna fırlatması gerekir? Bu sorunun cevabını daha kapsamlı bir örnek üzerinde inceleyelim. (yorum ekle)

Örnek: *Sekreter.java* (yorum ekle)

```
 import java.io.*;
class Calisan {
  public void calis(int deger) throws IOException {
      System.out.println("Calisan calisiyor "+ deger);
  }
}

public class Sekreter extends Calisan {

  public void calis(int deger) throws FileNotFoundException,
                                                  EOFException {

      System.out.println("Calisan calisiyor "+ deger);
      if(deger == 0) {
        throw new FileNotFoundException("Dosyayi bulamadim");
```

```
      } else if(deger == 1) {
        throw new EOFException("Dosyanin sonuna geldim");
      }
  }

  public static void basla(Calisan c, int deger) {
      try {
        c.calis(deger);
      } catch (IOException ex) {
        System.out.println("Istisna olustu: "+ ex);
      }
  }

  public static void main(String args[]) {
      Sekreter s1 = new Sekreter();
      Sekreter s2 = new Sekreter();
      Sekreter s3 = new Sekreter();
      basla(s1,2); // sorunsuz
      basla(s1,1); // EOFException
      basla(s3,0); // FileNotFoundException
  }
}
```

Bu örneğimizde *Sekreter* sınıfı *Calisan* sınıfından türemiştir. Ayrıca *Sekreter* sınıfının `calis()` yordamı, kendisinin ana sınıfı olan *Calisan* sınıfının `calis()` yordamını iptal etmiştir (*override*). *Calisan* sınıfına ait `calis()` yordamının fırlatacağı istisna daha kapsamlı olmasındaki sebep yukarı çevrimlerde (upcasting) sorun yaşanmaması içindir. Şimdi `basla()` yordamına dikkat edelim. Bu yordam *Calisan* tipinde parametre kabul etmektedir; yani, `main()` yordamının içerisinde oluşturulan *Sekreter*nesneleri `basla()` yordamına parametre olarak gönderilebilir çünkü arada kalıtım (*inheritance*) ilişkisi vardır. (yorum ekle)



Şekil-8.6. İptal etme (*override*) ve İstisnalar

Fakat bu gönderilme esnasında bir daralma (yukarı çevirim) söz konusudur, *Sekreter* nesneleri *heap* alanında dururken onlara bağlı olan referansların tipi *Calisan* tipindedir. Burada bir ayrıntı saklıdır, bu ayrıntı şöyledir: `c.calis()` komutu çağrıldığı zaman *Calisan* sınıfının `basla()` yordamına ait **etiketin** altında *Sekreter* sınıfında tanımlı olan `basla()` yordamına ait kodlar çalıştırılır. Bu uygulamamızda kullanılan etiket aşağıdadır. (yorum ekle)

Gösterim-8.12:

```
public void calis(int deger) throws IOException { // etiket
```

Çalıştırılacak gövde aşağıdadır.

Gösterim-8.13:

```
 System.out.println("Calisan calisiyor "+ deger);if(deger == 0) {
throw new FileNotFoundException("Dosyayi bulamadim");} else if(deger == 1)
{      throw new EOFException("Dosyanin sonuna geldim");}
```

Bu yüzden iptal edilen yordamın olabilecek en kapsamlı istisnayı fırlatması gerekir -ki yukarı çevirim işlemlerinde (*upcasting*) iptal eden yordamların gövdelerinden fırlatılabilecek olan istisnalara karşı aciz kalınmasın. Olabilecek en kapsamlı istisna tipi bu uygulama örneğimizde *IOException* istisna tipindedir çünkü bu istisna tipi hem *FileNotFoundException* istisna tipini hem de *EOFException* kapsamaktadır (bkz:Şekil-85.). (yorum ekle)

Uygulamamızın çıktısı aşağıdaki gibidir.

```
 Calisan calisiyor 2Calisan calisiyor 1Istisna olustu:
java.io.EOFException: Dosyanin sonuna geldimCalisan calisiyor 0Istisna
olustu: java.io.FileNotFoundException: Dosyayi bulamadim
```

8.1.13. İstisnaların Sıralanması

Bir istisna `catch` bloğunda veya `catch` bloklarında yakalanırken, istisnaların hiyarerşik yapılarına dikkat edilmelidir. (yorum ekle)

Örnek: *IstisnaSiralamasi.java* (yorum ekle)

```
class IstisnaBir extends Exception {
}

class IstisnaIki extends IstisnaBir {
}

public class IstisnaSiralamasi {
  public static void main(String args[]) {
      try {
        throw new IstisnaIki(); // dikkat
      } catch (IstisnaIki is2) {
        System.out.println("istisna yakalandi IstisnaIki: " );
      } catch (IstisnaBir is1) {
        System.out.println("istisna yakalandi IstisnaBir: " );
      }
  }
}
```

Bu örneğimizde kendimize özgü iki adet istisna tipi vardır. *IstisnaIki* sınıfı, *IstisnaBir* sınıfından türetilmiştir. Bunun anlamı eğer *IstisnaIki* tipinde bir istisna fırlatılırsa bunun *IstisnaBir* tipiyle `catch` bloğunda yakalanabileceğidir. Yukarıdaki örneğimizde*IstisnaIki* tipinde bir istisna fırlatılmaktadır, fırlatılan bu istisna ilk `catch` bloğunda yakalanmaktadır. Bir istisna bir kere yakalandı mı artık diğer `catch` bloklarının bu istisnayı bir daha tekrardan yakalama şansları yoktur (tekrardan

fırlatılmadıkları varsayılarak). Yani bir istisna bir kerede ancak bir `catch` bloğu tarafından yakalanabilir. Uygulamanın çıktısı aşağıdaki gibidir. (yorum ekle)

```
istisna yakalandi IstisnaIki:
```

Az önce bahsettiğimiz gibi eğer *IstisnaIki* tipinde bir istisna fırlatılırsa bu *IstisnaBir* tipiyle `catch` bloğunda yakalanabilir. (yorum ekle)

Örnek: *IstisnaSiralamasi2.java* (yorum ekle)

```
public class IstisnaSiralamasi2 {
 public static void main(String args[]) {
     try {
       throw new IstisnaIki(); // dikkat
     } catch (IstisnaBir is1) {
       System.out.println("istisna yakalandi IstisnaBir: " );
     }
  }
}
```

Yukarıdaki örneğimiz doğrudur. Uygulamamızın çıktısı aşağıdaki gibidir. (yorum ekle)

```
istisna yakalandi IstisnaBir:
```

Eğer *IstisnaIki* tipinde bir istisna fırlatılırsa ve bu ilk etapda *IstisnaBir* tipiyle catch bloğunda ve ikinci etapda ise *IstinsaIki* tipiyle catch bloğunda yakalanmaya çalışırsa ilginç bir olay meydana gelir. (yorum ekle)

Örnek: *IstisnaSiralamasi3.java* (yorum ekle)

```
public class IstisnaSiralamasi3 {
 public static void main(String args[]) {
     try {
       throw new IstisnaIki(); // dikkat
     } catch (IstisnaBir is1) {
       System.out.println("istisna yakalandi IstisnaBir: " );
     } catch (IstisnaIki is2) {
       System.out.println("istisna yakalandi IstisnaIki: " );
     }
  }
}
```

Yukarıdaki örneğimizi derlemeye (*compile*) çalıştığımız zaman aşağıdaki hata mesajını alırız. (yorum ekle)

```
IstisnaSiralamasi3.java:9: exception IstisnaIki has already been caught
 } catch (IstisnaIki is2) {
 ^
1 error
```

Bu hata mesajının anlamı, ikinci `catch` bloğunun boşu boşuna konulduğu yönündedir çünkü zaten ilk `catch` bloğu, bu istisnayı yakalayabilir, bu yüzden ikinci `catch` bloğuna (yorum ekle)