Java ile Nesne Merkezli ve Fonksiyonel Programlama

7. Bölüm
Sıra Dışı Durum Yönetimi (Exception Handling)

Akın Kaldıroğlu

www.javaturk.org

Aralık 2016

Küçük Ama Önemli Bir Konu

- Bu dosya ve beraberindeki tüm, dosya, kod, vb. eğitim malzemelerinin tüm hakları **Selsoft Yazılım, Danışmanlık, Eğitim ve Tic. Ltd. Şti**.'ne aittir.
- Bu eğitim malzemelerini kişisel bilgilenme ve gelişiminiz amacıyla kullanabilirsiniz ve isteyenleri **http://www.selsoft.academy** adresine yönlendirip, bu malzemelerin en güncel hallerini almalarını sağlayabilirsiniz.
- Yukarıda bahsedilen amaç dışında, bu eğitim malzemelerinin, ticari olsun/olmasın herhangi bir şekilde, toplu bir eğitim faaliyetinde kullanılması, bu amaca yönelik olsun/olmasın basılması, dağıtılması, gerçek ya da sanal/Internet ortamlarında yayınlanması yasaktır. Böyle bir ihtiyaç halinde lütfen benimle, **akin.kaldiroglu@selsoft.academy** adresinden iletişime geçin.
- Bu ve benzeri eğitim malzemelerine katkıda bulunmak ya da düzeltme ve eleştirilerinizi bana iletmek isterseniz çok sevinirim.
- Bol Java'lı günler dilerim.

İçerik

- Bu bölümde şu konular ele alınacaktır:
  - Geleneksel sıra dışı durum yönetimi,
  - Java'da sıra dışı durum yönetimi,
  - Throwable sınıf hiyerarşisi,
  - Exception ve Error sınıfları,
  - assert anahtar kelimesi ve savunmacı programlama,
  - Sıra dışı durum yönetiminin en iyi kullanımları.

Sıra Dışı Durum (Exception)

Sıra Dışı Durum Nedir?

- Yazılımda **sıra dışı durum** (**exception**), normal çalışmadan bir sapmadır.

- Sıra dışı durumlar, yazılımların çalışması sırasında, icra edilen süreçlerde beklenen ve olması gerekenler dışında meydana gelen anormal hallerdir.
  - Kredi kartıyla ödeme yaparken kartın limitinin yetmemesi bir sıra dışı durumdur.

- Yazılımlara bu halleri yönetmek için yapılar bulunur.
- Bu yapılara **sıra dışı durum yönetimi** (**exception handling**) denir.

Sıra Dışı Durum Neden Olur?

- Önemli olan sıra dışı durumlardan kaçınmak değildir, çünkü sıra dışı durumlar aslında o kadar da sıra dışı değildir, işin bir parçasıdır:
  - Kredi kartıyla ödeme yaparken kartın limitinin yetmemesi,
  - Bir dosyayı açmaya çalışırken açma yetkisine sahip olunmadığının ortaya çıkması,
  - Başvuru yaparken bir bilginin eksik olması
- gibi durumlar, aslen süreçlerin tabi parçası olan alternatif hallerdir.
- Yanlış olan, iş ve ihtiyaç analizi sırasında süreçler detaylandırılırken bu durumların hiç düşünülmemesidir.

Sıra Dışı Durum Olduğunda

- Önemli olan sıra dışı durum oluştuğunda yazılımın nasıl devam edeceğidir.

- Yazılımlarda sıra dışı durum oluştuğunda temelde iki seçenek söz konusu olur:
  - Durumu kullanıcıya bildirmek ve onun kararına göre devam etmek:
    - Dosya bulunamadığında yeni bir dosya ismi sormak örneğin.
  - Karar alıp, kullanıcıdan bir inisiyatif almasını beklemeden ve ona hissettirmeden çalışmaya devam etmek.
    - Dosya bulunamadığında yeni bir dosya oluşturup devam etmek.

Sıra Dışı Durum Yönetimi

- Sıra dışı durum yönetiminin temel kavramları şunlardır:
  - **Sıra dışı durum** (**exception**): Sıra dışı durumun kendisidir. Durum ile ilgili bilgileri de taşır.
  - **Fırlatma** (**throw**): Sıra dışı durumu oluşturup JVM'e bildirmektir.
  - **Yükseltme** (**raise**): Sıra dışı durumu bir üst bağlama göndermektir.
  - **Yakalama** (**catch, handle**): Fırlatılan sıra dışı durumun, yönetilmek üzere özel bir kod parçasına girmesidir.
  - **Çağrı zinciri** (**call chain**): Metotların birbirlerini çağırmalarından doğan zincirdir.
  - **Yığın izi** (**stack trace**): Herhangi bir anda aktif olan metot pencerelerinin (method frame) yığındaki durumudur.

Sıra Dışı Durum ve Hata

- Java açısından sıra dışı durum, **hata** (**error**) değildir.
- Hata hangi sebeple olursa olsun, genelde geri dönüşü olmayan bir durumdur.
- Yazılımlarda farklı tipte hatalar söz konusudur:
  - **Derleme hataları**: Söz dizimi (syntax) hatalarıdır.
  - **Çalışma zamanı hataları**: Programın çalışması sırasında olan hatalardır: JVM'deki bir durumdan kaynaklanan hata, "out of memory" gibi bellek hataları, vs.
  - **Mantık hataları**: Yazılımın süreçlerindeki hatalardır ve "bug" olarak adlandırılır. Bu hatalar tamamen yazılım takımının sorumluluğundadır.

Mantık Hataları

- Mantık hatalarının bir kısmı iş mantığıyla ilgilidir:
  - Satın alınan malın, kupon, indirim, vergi vs.den sonraki fiyatını yanlış hesaplamak bu cinsten, "bug" denince akla gelen hatalardır.
- Bazı mantık hataları ise iş mantığından ziyade, programlama dili yapılarıyla ilgilidir:
  - null olan bir referansın üzerinde metot çağrısı yapmak,
  - *n* odaya sahip bir dizide *n.* odaya ulaşmaya çalışmak.
- Bu türden hatalar, doğrudan programcının hatasıdır ve düzeltilmelidir.

Geleneksel Sıra Dışı Durum Yönetimi

Geleneksel Sıra Dışı Durum Yönetimi

- Her tür yazılımda sıra dışı durumlar söz konusudur ve yönetilmelidir.
- Dillerde bu tür durumlar için özel yapılar olmasa bile programatik olarak sıra dışı durumlar tespit edilip yönetilir.
- Aşağıdaki gibi bir "**readFile()**" metodu olsun:

```
// Pseudo code
readFile (fileName) {
   open the file;
   determine its size;
   allocate that much memory;
   read the file into memory;
   edit the file;
   save the file;
   close the file;
}
```

Muhtemel Sıra Dışı Durumlar

- Verilen örnekte aşağıdaki sıra dışı durumlar söz konusu olabilir:
  - Dosya açılamazsa?
  - Dosyanın boyutu belirlenemezse?
  - Dosyayı açmak için yeterince bellek yoksa?
  - Dosya belleğe okunurken problem çıkarsa?
  - Dosya kapatılamazsa?

- Tüm bu "`readFile()`" metodunda olabilecek sıra dışı durumlardır ve geleneksel şekilde de olsa yönetilmelidir.

- Geleneksel sıra dışı durum yönetimi, int ya da String hata kodları üzerinden ve genel mekanizmalarla yapılır.

```
// Pseudo code
errorCodeType readFile {
   initialize errorCode = 0;
   open the file;
   if (fileIsOpen) {
     determine the length of the file;
     if (gotTheFileLength) {
       allocate that much memory;
       if (gotEnoughMemory) {
         read the file into memory;
         if (readFailed) { errorCode = -1; } // Can't read
       } else { errorCode = -2; } // Not enough memory
     } else { errorCode = -3; }   // File length unavailable
     close the file;
     if (fileDidntClose && errorCode == 0){ errorCode = -4; }
     else { errorCode = errorCode and -4; }
   } else { errorCode = -5; } // File can't open
   return errorCode;
}
```

- "`readFile()`" metodunun aşağıdaki gibi bir metot zincirde çağrıldığını düşünün.
- `readFile()` metodunun fırlattığı sıra dışı durum kodunun `method3()` ve `method2()` üzerinden `method1()`'e yükseltildiği durumda yandaki gibi bir kod yapısı söz konusu olacaktır.

```
// Pseudo code
method1 {
   call method2;
}
method2 {
   call method3;
}
method3 {
   call readFile;
}
```

```
// Pseudo code
method1 {
  errorCodeType error;
  error = call method2;
  if (error)
    doErrorProcessing;
  else proceed;
}

errorCodeType method2 {
  errorCodeType error;
  error = call method3;
  if (error) return error;
  else proceed;
}

errorCodeType method3 {
  errorCodeType error;
  error = call readFile;
  if (error) return error;
  else proceed;
}
```

Geleneksel Yaklaşımın Problemleri

- Geleneksel sıra dışı durum yönetiminin en problemli iki özelliği şunlardır:
  - Dilde sıra dışı durum yönetiminin mekanizmaları yoktur, yönetim tamamen program yazanlar tarafından kurgulanır.
    - Sıra dışı durumun fark edilmesi, oluşturulması, yığındaki metotlara ulaştırılması ve yakalanması, tamamen dildeki genel yapılarla halledilir.
    - Bu durum programları daha karmaşık ve anlaşılmaz kılar.
  - Sıra dışı durumlar ancak int ya da String tipinde değişkenlerle ifade edilirler.
    - Bu da prosedürel yapıların en temel problemi olan "anlam" problemini tekrar gündeme getirir.

Java'da Sıra Dışı Durum Yönetimi

Java'da Sıra Dışı Durum Yönetimi - I

- Java, nesne-merkezli bir dil olarak, sıra dışı durumlara has kontrol yapısı ve sıra dışı durumlar için oluşturduğu nesne hiyerarşisi ile sıra dışı durum yönetimi sağlamaktadır.

- Örnek olarak ele alınan "`readFile()`" metodunun Java'nın sıra dışı durum yönetimi çerçevesinde kavramsal olarak şöyle olduğunu düşünülebilir:

```
// Pseudo code
method1 {
    call method2;
}
method2 {
    call method3;
}
method3 {
    call readFile;
}
```

Java'da Sıra Dışı Durum Yönetimi - II

- Java'da sıra dışı durum yönetimi için **try-catch** bloğu kullanılır.
- Sıra dışı durum fırlatma ihtimali olan kod try bloğunda, fırlatılabilecek sıra dışı durumları yakalayacak kod ise catch bloğuna konur.

```
// Pseudo code
readFile {
  try {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
  }catch (fileOpenFailed){
      doSomething;
  }catch (sizeDeterminationFailed){
      doSomething;
  }catch (memoryAllocationFailed){
      doSomething;
  }catch (readFailed){
      doSomething;
  }catch (fileCloseFailed){
      doSomething;
  }
  ...
}
```

Java'da Sıra Dışı Durum Yönetimi - III

- Ya da sıra dışı durumu yakalamayı-yönetmeyip, bir üst bağlama yükseltmek de söz konusu olabilir.
- Bu durumda **try-catch** bloğu kullanılmaz, oluşabilecek sıra dışı durumlar metodun arayüzünde listelenir ki, bu metodu çağıranlar yakalamak ya da yükseltmekten birini seçebilsin.
- Bunun için **throws** anahtar kelimesi kullanılır.

```
// Pseudo code
readFile throws fileOpenFailed,
  sizeDeterminationFailed,
  memoryAllocationFailed,
  readFailed,
  fileCloseFailed  {

    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Java'da Sıra Dışı Durum Yönetimi - IV

- Metotlar sıra dışı durum fırlattıklarını **throws** ifadesiyle belirtirler.
- **readFile()** metodunun fırlattığı sıra dışı durumun **method3()** ve **method2()** üzerinden **method1()**'e yükseltildiği durumda yandaki gibi bir kod yapısı söz konusu olacaktır.
- Bu durumda sıra dışı durumu yakalayan ve yöneten **method1()**'dir.

```
// Pseudo code
method1 {
   try {
      call method2;
   } catch (exception){
        doExpProcessing;
   }
}

method2 throws exception{
   call method3;
}

method3 throws exception{
   call readFile;
}
```

Java'da Sıra Dışı Durum Yönetimi - IV

- Sıra dışı durumu **`method1()`** yerine **`method2()`** yakalarsa kod yandaki gibi olur.
- Dolayısıyla soru şudur: Sıra dışı durumu kim yakalar?
  - Sıra dışı durumu kimler yükseltir (raise)?
- Sıra dışı durumu yakalayan, o durumu düzeltmek için bilgiye sahip olandır.
- Diğerleri ise sıra dışı durumu, yakalayana yükseltirler.

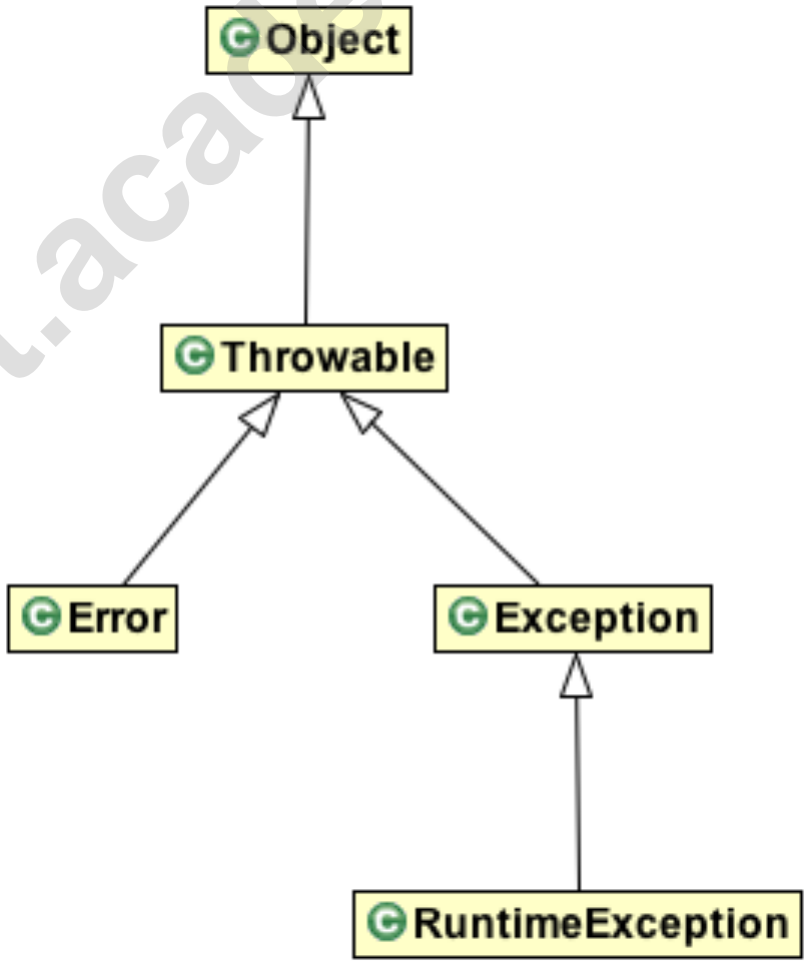
```
// Pseudo code
method1 {
    call method2;
}

method2 {
  try {
    call method3;
  } catch (exception){
      doExpProcessing;
  }
}

method3 throws exception{
  call readFile;
}
```

java.lang.Throwable Sınıfı ve Hiyerarşisi

Java'da Hata ve Sıra Dışı Durum

- Java'nın hata ve sıra dışı durumlarla ilgili nesneleri yandaki gibidir.

- Hata ve sıra dışı durumların atası **`Throwable`** sınıfıdır.
- Hatalar **`java.lang.Error`** sınıfıyla, sıra dışı durumlar ise **`java.lang.Exception`** sınıfıyla ifade edilir.



Throwable

- Java'da tüm hatalar ve sıra dışı durumlar **`java.lang.Throwable`** sınıfıyla temsil edilir.
- **`Throwable`** isim olarak bir arayüz gibi görünmesine rağmen bir sınıftır.
- İster JVM tarafından ister kod içinde programatik olarak fırlatılsın, Java'da fırlatılan tüm nesneler bu sınıfın ya da alt sınıflarının bir nesnesi olmak zorundadır.
- **`Throwable`**'ın sadece iki tane alt sınıfı vardır: **`Error`** ve **`Exception`**

Throwable API - Kurucular

- **`Throwable`** sınıfının 5 tane kurucusu vardır.
- En sık kullanılan 4 kurucu metodu şunlardır:

```
Throwable()

Throwable(String message)

Throwable(Throwable cause)

Throwable(String message, Throwable cause)
```

Throwable API: Metotlar

- **`Throwable`** nesnesi yakalanınca üzerinde metotlar yardımıyla bazı bilgilere ulaşılır:

```
String getMessage()

String getLocalizedMessage()

void initCause(Throwable t)

Throwable getCause()

void printStackTrace()

void printStackTrace(PrintStream s)

void printStackTrace(PrintWriter s)

StackTraceElement[] getStackTrace()
```

StackTraceExample.java

- Örneği önce “throwable.printStackTrace();” satırı olmadan çalıştırın.

- Sonra “throwable.printStackTrace();” satırı etkin iken çalıştırın ve yığınla alakalı bilgiyi aldığınız noktadan itibaren geriye doğru, çalışmanın en başına kadar, o anda yığında olan metotların nasıl sıralandığına dikkat edin.

throw Anahtar Kelimesi

- **Throwable** sınıfının nesneleri fırlatılır.
- Bu amaçla **throw** anahtar kelimesi kullanılır.
- throw anahtar kelimesi, **Throwable** sınıfının ya da alt sınıflarından birisinin nesnesini argüman olarak alır.

```
Throwable throwable = new Throwable("Just kidding!");
throw throwable;

Ya da

throw new Throwable("Just kidding!");
```

Throwable Fırlatılınca Ne Olur? -I

- Var olan sıra dışı durumu throw ile Throwable sınıfının ya da alt sınıflarından birisinin nesnesini fırlatarak ifade eden kod parçası için şu iki durumdan birisi söz konusudur:
  - İçinde bulunulan metodun arayüzünde bu durumun yani bir Throwable nesnesinin fırlatıldığının belirtilmesi,
    - Öyle ki bu metodu çağıranlar, bu durumun olabileceğinden haberdar olsunlar,
  - Fırlatılan `Throwable` nesnesinin yakalanıp gereğinin yapılması.
- Bu iki durumdan ilkine, yükseltme (raising ya da propogating), ikincisine de yakalama (catching) denir.

Yükseltme (Raising-Propogating) - I

- Bir metotta fırlatılan **Throwable** nesnesinin, o metotta yakalanmayıp bir üst bağlama yükseltilmesi (raising), fırlatılan nesnenin metodun arayüzünde **throws** anahtar kelimesi kullanarak ifade edilmesiyle olur.
- Bir üst bağlam, **Throwable** nesnesinini fırlatan metodu çağıran ve yığında (stack) bir altta ya da çağrı zincirinde (call chain) bir üstte bulunan metottur.

```
public static void throwAThrowable() throws Throwable {
  Throwable throwable = new Throwable("Just kidding!");
  throw throwable;
}
```

Yükseltme (Raising-Propogating) - II

- Bu durumda üst bağlam da aynı iki seçeneğe sahiptir: Yükseltmek veya yakalamak.

- Yani metotlar, çağırdıkları metotların kendilerine yükselttikleri **`Throwable`** nesnesini bir üst bağlama yükseltebilirler.
  - Ya da yakalarlar.

- Eğer fırlatılan **`Throwable`** nesnesini çağrı zincirindeki hiç bir metot yakalamazsa JVM çalışmasını durdurur.

  - Dolayısıyla fırlatılan **`Throwable`** nesnesi muhakkak yakalanmalıdır.

RaisingExample.java

- Program çalıştığında ekrana basılan çıktıya dikkat edin.
- "After doSomething()" yazıldı mı?
- "throw throwable;" satırını kaldırıp tekrar çalıştırın, farkı gözlemleyin.

Yakalama (Catching)

- throw ile **Throwable** sınıfının ya da alt sınıflarından birisinin nesnesini fırlatan kod parçası için söz konusu ikinci durum, fırlatılan nesneyi yakalamaktır.

- Bu amaçla **try - catch** bloğu kullanılır.

```
private static void doSomething() {
    try {
        doSomethingElse();
    } catch (Throwable e) {
        System.out.println("Catching a Throwable");
    }
}
```

try-catch Bloğu

- **`Throwable`** nesnesini fırlatan kod parçası **try** blokunda bulunur, fırlatılan nesnenin tipinde argümana sahip **catch** bloku ise hemen ardından gelir.

- **`Throwable`** sınıfının alt sınıfları söz konusu olduğunda birden fazla farklı **`Throwable`** nesnesi fırlatılabilir.

- Bu durumda birden fazla farklı tipte nesneyi yakalamak için birden fazla catch bloğu alt alta bulunabilir.

- try ile catch ve birden fazla olduğu durumda catch blokları arasına başka bir kod bloğu giremez.

CatchingExample.java

java.lang.Error Sınıfı

Error

- **`java.lang.Error`** sınıfı, uygulamaların yakalayıp düzeltmeye çalışmayacağı ciddi hataları ifade eder.
- Bir uygulamada hiç bir şekilde bu tipten bir nesnenin oluşmasına izin verilmemelidir.
- Ama oluşan **`Error`** nesnelerini yakalamak söz konusu değildir.
  - Çünkü bu sınıfın nesnelerini JVM fırlatır.

- Bir metodun fırlattığı **`Error`** nesnelerini throws ifadesiyle belirtmesi gerekmediği gibi bu tipten nesnelerin try-catch ile yakalanması da gerekmez.

ErrorExample.java

Java API'sindeki Error Sınıfları - I

- **`Error`** sınıfları, her pakette ayrı bir başlık altında sıralanır.
- Java API'sinde Exception sınıflarından sonra gelir.

- **`java.lang`** paketindeki bazı **`Error`** sınıfları şunlardır:
  - **`AbstractMethodError`**: Soyut bir metodu çağırmaya çalıştığınızda oluşur. Normalde çalışma zamanında yakalanır ama bazen daha önce somut olan bir metodu soyut yaparak sınıfı tekrar derleyip kullanmaya çalışıldığında oluşur.
  - **`NoSuchFieldError`** ve **`NoSuchMethodError`** **`AbstractMethodError`** gibi, bir sınıf, eski durumla uygun olmayan bir şekilde değiştirilip, derlenip kullanıldığında oluşur.

Java API'sindeki Error Sınıfları - II

- **`NoClassDefFoundError`**: Yüklenecek sınıf bulunamadığında fırlatılır.
- **`VirtualMachineError`**: JVM'in çalışması sırasında oluşan hatalı durumlar için fırlatılır.
- **`StackOverflowError`**: Yığının (stack), yeni bir metot çağrısı için gerekli belleğe sahip olmadığında fırlatılır.
- **`OutOfMemoryError`**: Heapin, yeni bir nesne yaratmak için gerekli belleğe sahip olmadığında fırlatılır.

java.lang.Exception Sınıfı

Exception

- **`java.lang.Exception`** sınıfı, yakalanması düşünülen her türlü sıra dışı durumu temsil eder.

- Uygulamalarda oluşan sıra dışı durumlar ya **`Exception`** sınıfının ya da alt sınıflarının nesnesi olarak fırlatılır.

- **`Exception`** nesneleri JVM tarafından, Java API'sinin bir parçası olan metotlarda fırlatılabildiği gibi programatik olarak da oluşturulup fırlatılabilir.
  - İkinci halde Exception sınıfının alt sınıflarının oluşturulması da söz konusudur.

Exception API: Kurucular

- **`Exception`** sınıfının varsayılan dahil 5 tane kurucusu vardır.
- En sık kullanılan, sıra dışı durumu betimleyen **`String`** parametre alan kurucudur.
- Bazen bir **`Exception`** nesnesinin yakalanıp, bir başka **`Exception`** nesnesini oluşturmakta kullandığı da görülür.

```
Exception(String message)

Exception(Throwable cause)

Exception(String message, Throwable cause)
```

Exception API: Metotlar

- **`Exception`** nesnesi yakalanınca üzerinde metotlar yardımıyla bazı bilgilere ulaşılır:

```
String getMessage()

String getLocalizedMessage()

Throwable getCause()

void printStackTrace()

void printStackTrace(PrintStream s)

void printStackTrace(PrintWriter s)
```

Java API'sindeki Exception Sınıfları - I

- Exception sınıfları, her pakette ayrı bir başlık altında, sınıflardan ve enumlardan sonra sıralanır.

- **`java.lang`** paketindeki bazı **`Exception`** sınıfları şunlardır:
  - **`java.lang.ClassCastException`**: Nesnelerin yanlış alt sınıfa dönüştürülmelerinde (down cast) fırlatılır.
  - **`java.lang.IndexOutOfBoundsException`**: String ya da dizi (array) gibi yapılarda yanlış erişimi ifade eder.
  - **`java.lang.NumberFormatException`**: Girilen String nesnesinin sayı formatına dönüştürülmesi sırasında oluşan sıra dışı durumları ifade eder.

Java API'sindeki Exception Sınıfları - II

- Diğer paketlerdeki bazı sık kullanılan **`Exception`** sınıfları:
  - **`java.io.IOException`**: JVM ile dış dünya arasındaki giriş-çıkış işlemleri sırasında oluşan sıra dışı durumlar içindir.
  - **`java.sql.SqlException`**: JVM ile veri tabanı arasındaki iletişim sırasında oluşan sıra dışı durumlar içindir.

ExceptionExample1.java

- Önce “openFile()” metodunu çalıştırın.
- Sonra “openFile()” metodunu kaldırıp “openAndCloseFile()” metodunu çalıştırın.
  - “openAndCloseFile()” metodunda iki tane catch bloku olduğuna dikkat edin.
- Sonra da iki metodu arka arkaya çalıştırın.
- Geçerli ve geçersiz girdiler vererek sıra dışı durumun fırlatıldığı ve fırlatılmadığı halleri gözlemleyin.

try-catch ile Sıra Dışı Durum Yönetimi

try-catch Bloğu ve Exception - I

- **`Exception`** nesnesini fırlatan kod parçası **try** blokunda bulunur, fırlatılan nesnenin tipine uygun argümana sahip **catch** bloğu ise hemen ardından gelir.

- **`Exception`** sınıfının alt sınıfları söz konusu olduğunda birden fazla farklı **`Exception`** nesnesi fırlatılabilir.

  - Bu durumda birden fazla farklı tipte nesneyi yakalamak için birden fazla catch bloğu alt alta bulunabilir.

- try ile catch ve birden fazla olduğu durumda catch blokları arasına başka bir kod bloğu giremez.

try-catch Bloğu ve Exception - II

- Eğer try bloğu varsa en az bir tane catch bloğu olmalıdır.
- Eğer kod try bloğuna girerse, sıra dışı durum oluşmazsa akış catch bloğuna girmez, normal çalışma devam eder.
- Eğer kod try bloğuna girerse ve sıra dışı durum oluşursa, akış var olan catch bloklarından bir tanesine dallanır.
- Dallanılan catch bloğu, fırlatılan nesnenin tipinden ya da uygun üst sınıfından nesneyi argüman olarak almalıdır.

try-catch Bloğu ve Exception - III

- Bir **`Exception`** nesnesi için sadece ve sadece bir catch bloğuna dallanma olabilir.
- catch bloğu argüman olarak **`Exception`** nesnesi alan bir metot gibi düşünülebilinir.

```
try {
   // Code that might generate exceptions
} catch(Type1 id1) {
   // Handle exceptions of Type1
} catch(Type2 id2) {
   // Handle exceptions of Type2
} catch(Type3 id3) {
   // Handle exceptions of Type3
}
```

DivisionByZero1.java

- "divide()" metodunu önce ikinci argümanı "0" olmayacak şekilde, örneğin "divide(20, 5)" çağırın.

- Sonra ikinci argümanı "0" olacak şekilde, örneğin "divide(20, 0)" çağırın.
  - Bu durumda fırlatılan **`ArithmeticException`** nesnesini yakalayacak şekilde örneği değiştirin => DivisionByZero2.java

DivisionByZero2.java

- DivisionByZero2, DivisionByZero1'in sıra dışı durum yönetimini içeren halidir.
- "divide()" metodunu önce ikinci argümanı "0" olmayacak şekilde, örneğin "divide(20, 5)" çağırın.
- Sonra ikinci argümanı "0" olacak şekilde, örneğin "divide(20, 0)" çağırın.

try-catch Bloğu ve Exception - III

- Fırlatılan **`Exception`** nesnesine uygun tipte bir catch bloğu bulunamazsa, bulununcaya kadar aramaya devam edilir.
  - Bu, sıra dışı durumun metotlar arasında, yığında en yukarıdan en aşağıya doğru gezdirilmesidir (propagating).
  - Bu durumda metodun, yakalanması gerekip de yakalanmayan nesneyi "throws" ifadesi ile belirtmesi gereklidir.

- Bu aramada-gezdirmede önce try bloğunu takip eden catch blokları sorgulanır, uygun catch bloğu bulunamazsa, bir üst bağlam olan çağıran metoda geçilir ve aynı arama orada da yapılır.

try-catch Bloğu ve Exception - IV

- Eğer çağrı zincirindeki tüm metotlar arandığı halde fırlatılan **`Exception`** nesnesine uygun bir catch bloğu bulunamamışsa bu durumda JVM çalışmasını durdurur (JVM crashes and exits).

- Yani Java'nın sıra dışı durumları yok görülemez, muhakkak yakalanmalı ve gereği yerine getirilmelidir.

ExceptionExample2.java

- “openAndCloseFile()” metodunda fırlatılan iki farklı **`Exception`** nesnesinin iki farklı yerde yakalandığına dikkat edin.

try-catch Bloğu ve Exception - IV

- Fırlatılan birden fazla **`Exception`** nesnesi olduğunda birden fazla catch bloğu olabileceği gibi, bir metot throws ile birden fazla **`Exception`** nesnesini fırlatacağını belirtebilir.
- Bu durumda fırlatılan nesnenin tipleri, throws anahtar

```
private static void openAndCloseFile(String path)
        throws IOException, FileNotFoundException {
  File file = new File(path);
  // throws FileNotFoundException
  InputStream in = new FileInputStream(file);
  System.out.println("File opened!");
  in.close();  // throws IOException
  System.out.println("File closed!");
}
```

ExceptionExample3.java

- “openAndCloseFile()” metodunun nasıl birden fazla sıra dışı durum fırlattığını gözlemleyin.

- Bu durumda “openAndCloseFile()” metodunun çağrıldığı yerde birden fazla catch bloğu olabilir.

try-catch Bloğu ve Exception - V

- “Fırlatılan **`Exception`** nesnesine uygun tipte bir catch bloğu bulunamazsa, bulununcaya kadar aramaya devam edilir.” dendi.
  - “Fırlatılan **`Exception`** nesnesine uygun tipte bir catch bloğu” ne demektir?

- Java API’si içerisinde **`Exception`** sınıfının pek çok alt sınıfı vardır.
  - Yenilerini de siz oluşturabilirsiniz.

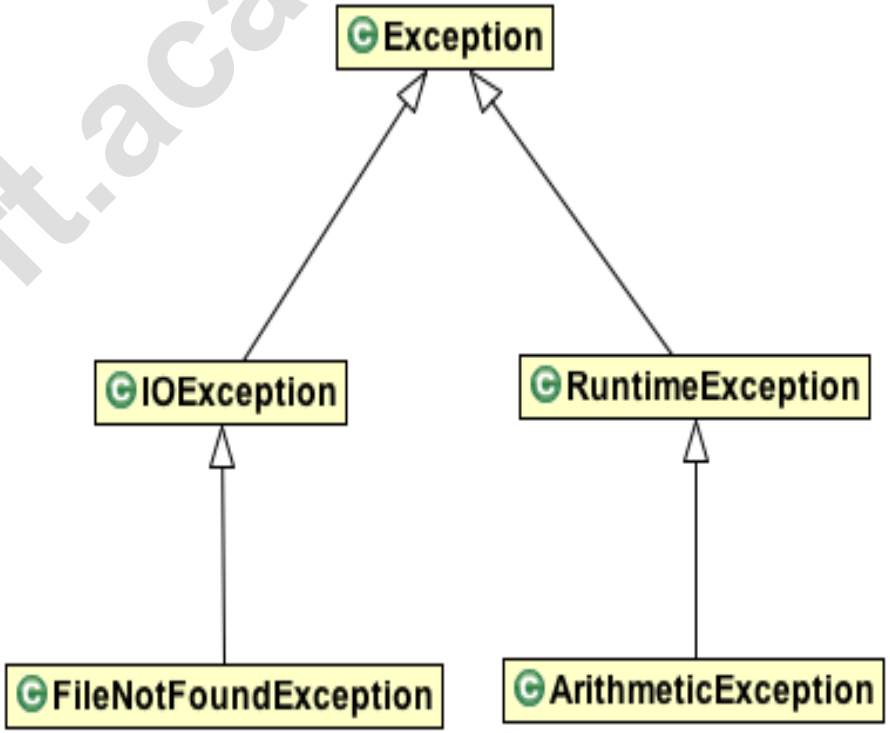
Yerine Geçebilme - I

- Yerine geçebilme (substitutability) özelliğinden dolayı, her sıra dışı durum nesnesi, üst sınıfından bir nesne bekleyen bağlamda kullanılabilir.

- Bu cümlenin bir kaç farklı sonucu vardır, ilki:
  1. Bir catch bloğu, aldığı **`Exception`** nesnesi ve tüm alt sınıflarının nesnelerini yakalar.
     - Her sıra dışı durum nesnesi, üst sınıfından bir nesne kabul eden catch bloğu tarafından yakalanır.

Uygun Exception

- Yandaki hiyerarşi göz önüne alındığında,
  - fırlatılan **`FileNotFoundException`** nesnesi, **`IOException`** hatta **`Exception`** nesnesi bekleyen catch bloğunda,
  - ya da fırlatılan **`ArithmeticException`** nesnesi, **`RuntimeException`** hatta **`Exception`** nesnesi bekleyen catch bloğunda yakalanabilir.



DivisionByZero3.java

- **`ArithmeticException`** fırlatan “divide()” metodunun nesnesinin, nasıl farklı catch bloklarıyla yakalanabildiğini gözlemleyin.

Yerine Geçebilme - II

- Yerine geçebilme (substitutability) özelliğinden dolayı, her sıra dışı durum nesnesi, üst sınıfından bir nesne bekleyen bağlamda kullanılabilir.

- Bu cümlenin ikinci sonucu:
  2. Bir metot, gerçekte fırlatılan **`Exception`** nesnesi yerine daha daha genel olanını fırlatabilir.
     - Ama daha özel olanını ya da aralarında bir is-a ilişkisi olmayanı fırlatamaz.

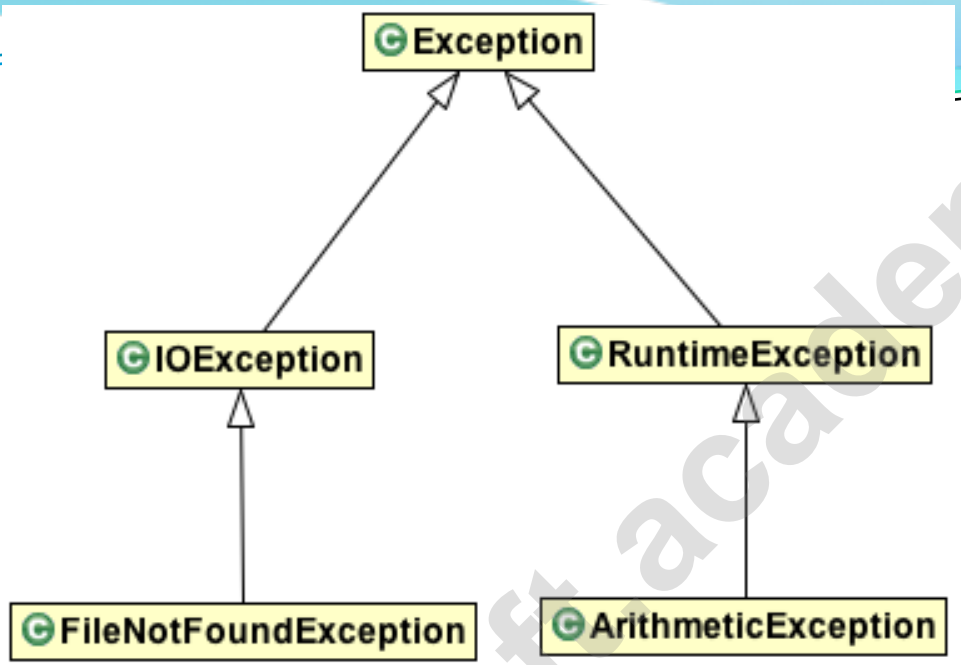
ExceptionExample4.java

- Aynı koda sahip olan “openAndCloseFile()” metodunun farklı şekillerinin nasıl farklı türden sıra dışı durum nesneleri fırlattığını gözlemleyin.

- Ama “openAndCloseFile()” metodunun, kodunda fırlatılan sıra dışı durum nesneleri ile bir is-a ilişkisine sahip olmayan örneğin **`ArithmeticException`** fırlatamayacağını da gözlemleyin.

Yerine Geçebilme - III

- Yerine geçebilme (substitutability) özelliğinden dolayı, her sıra dışı durum nesnesi, üst sınıfından bir nesne bekleyen bağlamda kullanılabilir.
- Bu cümlenin üçüncü sonucu:
  3. Birden fazla catch bloğunun varlığı halinde, bu bloklar, daha özel **`Exception`** nesnesi alanından daha genel olanına doğru sıralanmalıdır.
     - Aksi taktirde erişilemeyen kod (unreachable code) oluşur ve bu da bir derleme zamanı hatasıdır.
     - catch blokları mutlaka farklı türden **`Exception`** nesneleri almalıdırlar, eğer aynı try bloğununa ait iki catch bloğu aynı nesneyi alırsa bir derleme hatası oluşur.

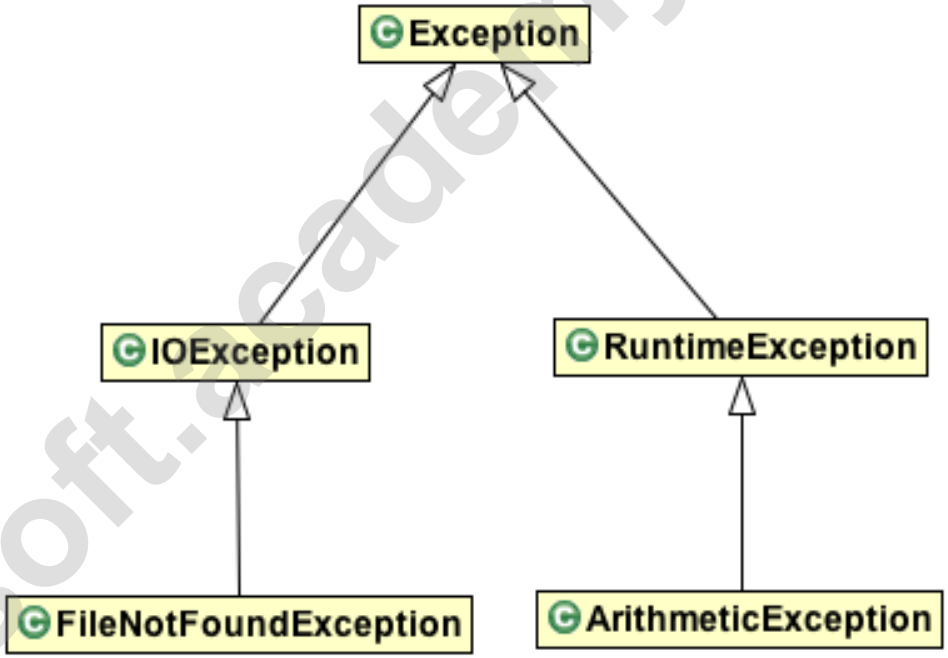



```
try {
    openAndCloseFile(path);
}
catch(IOException e){ … }
catch(FileNotFoundException e){
    // unreachable code
}
```

```
try {
    openAndCloseFile(path);
}
catch(Exception e){ … }
catch(IOException e){
    // unreachable code
}
catch(FileNotFoundException e){
    // unreachable code
}
```

Yerine Geçebilme - III

- Benzer bir durum, bir metodun fırlattığı sıra dışı nesneleri tanıtırken söz konusu değildir.




```
void openAndCloseFile(String path)
                throws IOException, FileNotFoundException

// Ya da

void openAndCloseFile(String path)
                throws FileNotFoundException, IOException
```

ExceptionExample5.java

Human.java

- B. Eckels'in "Thinking in Java" kitabından.

Sıra Dışı Durum Yakalandıktan Sonra

- Sıra dışı durum yakalandıktan sonra ne olur?
  - Akış, sıra dışı durumu yakalayan catch bloğundan devam eder.
  - Akış, sıra dışı duruma sebep olan bağlama geri dönmez.
- Dolayısıyla sıra dışı durumu yakaladıktan sonra şunlar yapılabilir:
  - Sıra dışı duruma sebep olan kodu, şartları değiştirerek tekrar çağırmak,
  - Ya da sıra dışı duruma sebep olan kodu bir daha çağırmadan devam etmek.
- Her halükarda sıra dışı durumu log ile kayıt altına almak da bir seçenektir.

DivisionByZero4.java

- **`ArithmeticException`** nesnesi yakalandıktan sonra akışın nasıl devam ettiğini gözlemleyin.

catch Bloğunda Çoklu Exception - I

- Birden fazla sıra dışı durum fırlatılma ihtimali olması durumunda, istenirse birden fazla catch bloğu yazmak yerine tek bir catch bloğuyla yetinmek mümkündür.
- Java SE 7 ile gelen bu özellik sayesinde catch bloğuna "|" ile yakalanacak birden fazla **Exception** nesnesinin tipi geçilebilir.

```
try {…}
catch(IOException e){…}
catch(ArithmeticException e){…}
```

```
try {…}
catch(IOException | ArithmeticException e){…}
```

DivisionByZero5.java

catch Bloğunda Çoklu Exception - II

- catch bloğunda çoklu **Exception** kullanılması durumunda, her tür nesne için aynı kodun geçerli olduğuna dikkat edilmelidir.
- Ayrıca catch bloğuna geçilen **Exception** nesneleri arasında bir is-a ilişkisi olması durumunda, sadece daha geniş olan tip kullanılabilir.
  - Bu durum zaten bir derleme hatasıdır.

```
try {…}
catch(IOException | FileNotFoundException e){…}
// Compilation error
```

try-catch ile Kaynak Yönetimi

try-catch Bloğu ve Kaynaklar

- Sıra dışı durumları try-catch ile yönetiminde eğer try bloğunda bazı kaynaklar açılıyorsa ve try bloğunda sıra dışı durum fırlatılırsa bu kaynaklar kapanmayabilir.

```
private static void openFile(String path) {
    File file = new File(path);
    try {
        InputStream in = new FileInputStream(file);
        System.out.println("File opened!");
        doSomethingWithFile(in);
        in.close();  // Can't get here to close file
    } catch (FileNotFoundException e) { … }
      catch (IOException e) { … }
}
```

Kaynaklı try Bloğu

- Bu türden açılan kaynakların kapanmama riskini ortadan kaldırmak için Java SE 7'de **kaynaklı try bloğu** (**try-with-resources**) yapısı geldi.
- Bu yapıyla, try ifadesinde, blokta kullanılacak ve ne şekilde olursa olsun blok çıkışında otomatik olarak kapatılacak kaynaklar try ifadesinde listelenir.

```
try (ResType1 res1 = initialization1,
     ResType2 res2 = initialization2,…){
   doSomething();
}
catch(SomeException e){ … }
catch(OtherException e){ … }
catch(AnotherException e){ … }
```

AutoClosable

- Bir kaynağın, kaynaklı try bloğunda listelenebilmesi için **`java.io.AutoClosable`** arayüzünü gerçekleştirmesi ya da bu arayüzü gerçekleştiren bir sınıftan miras devralması gereklidir.

- **`AutoClosable`** arayüzünde bir tane metot vardır.

```
public void close()
```

- Bu metot, try bloğunda tanımlanan kaynakların, try bloğundan ne şekilde çıkılırsa çıkılsın kapatılması için otomatik olarak çağrılır.

ClosableExample.java

- Önce “run0()” metodunu çalıştırın ve kaynaklı try kullanılmadığı hali inceleyin.

- Sonra “run1()” metodunu çalıştırın ve aynı işin kaynaklı try ile nasıl daha kolay yapılabildiğini gözlemleyin.

Kaynakları Kapatma

- try ifadesinde birden fazla kaynak tanımlanırsa, kapatılmaları tanım sırasının tersine olur.
- Dolayısıyla ilk tanımlanan en son kapanır.
- Birbirlerine bağımlı yapılarda bu durumu göz önüne almak gereklidir.

Kaynağı Oluştururken Problem

- Eğer kaynak açılırken yani nesnesi oluşturulurken problem olursa ve bir sıra dışı durum fırlatılırsa bu durumda zaten kaynak oluşturulmayacak demektir.

ClosableExample.java

- “run2()” metodunu çalıştırın. Burada kaynakların hangi sıralayla kapanacağını gözlemleyin.

- “run3()” metodunu çalıştırın. Burada kaynakların oluşturulurken bir sıra dışı durum fırlatılması halinde ne olacağını gözlemleyin.

finally

finally Bloğu - I

- **finally**, try-catch yapısında kullanılabilen bir diğer bloktur.
  - Zorunlı değildir, olursa sadece bir tane olabilir.
  - Muhakkak en son catch bloğundan sonra gelir.
  - try-catch-finally blokları arasına bir başka kod parçası giremez.
- try bloğuna giren kod, catch bloğuna girsin ya da girmesin, daima finally bloğuna girer.

```
try {
    doSomething;
}
catch(SomeException e){ … }
catch(OtherException e){ … }
catch(AnotherException e){ … }
finally{ … }
```

finally Bloğu - II

- Dolayısıyla finally bloğu, sıra dışı durum oluşmasına bağlı olmadan, try bloğuna girmekten kaynaklanan durumlar için kullanılır.

- Akış try bloğuna girildikten sonra ister normal olarak çıksın ister bir sıra dışı durumun fırlatılmasıyla çıksın, nihayetinde muhakkak finally bloğuna girer.

FinallyExample.java

finally Bloğu – Kaynaklı try

- finally bloğunun kaynakları otomatik olarak kapatan kaynaklı try bloğundan farkı nedir?

- Kaynaklı try bloğu sadece `AutoClosable` olan yapılar için geçerlidir.

- Bunun dışındaki her türlü kaynağın yönetimi için finally bloğu kullanılmalıdır.

Neden finally Bloğu?

- finally bloğu kaynak yönetimi açısından önemlidir.
- Öncelikle **`AutoClosable`** olan kaynaklar için kaynaklı try kullanılmalıdır.
- try içinde açılan ve **`AutoClosable`** olmayan ve her türlü kaynağı kapatmanın yeri finally bloğudur
  - Dolayısıyla finally bloğu, try bloğunda yapılan her türlü iş için bir temizlik yeri olarak düşünülmelidir.
- Bu şekilde sıra dışı durum fırlatılmasa bile özellikle return, break ve continue gibi kontrol yapıları yüzünden çalışmayan ve temizlik yapan kod parçalarının finally bloğunda çalışması sağlanır.

CleanUpWithFinally.java

Dikkat!

- finally bloğunda kaynakları kapatırken sıra dışı durumların tekrar oluşabileceği de düşünülmelidir.

- Dolayısıyla finally bloğunda da try-catch-finally kullanılabilir.

Sıra Dışı Durumu Tekrar Fırlatmak

Tekrar Fırlatmak - I

- Bazen sıra dışı durum yakalanır ama tekrar fırlatılır.
- Bu durumlarda yakalama amacı çoğunlukla loglamaktır.

```
try {
   // access the database
} catch (SQLException ex) {
    ...
    logger.log(logLevel, message, ex);
    throw ex;
}
```

- Ama sıra dışı durumun nasıl yönetileceği bilinmediğinden, gerçekten onu yönetecek koda gitmesi için sıra dışı durum tekrar fırlatılır.

Tekrar Fırlatmak - II

- Bazen de sıra dışı durum yakalanıp, bir başka sıra dışı durumun içine koyup tekrar fırlatılır.
- Bu duruma sıra dışı durumları zincirleme (chaining exceptions) denir.
- Bu amaçla **`void initCause(Throwable t)`** metodu kullanılabilir.

```
try {
   // access the database
} catch (SQLException ex) {
    logger.log(logLevel, message, ex);
    Throwable e = new  NoSuchProductException("No such product!");
    e.initCause(ex);  // Optional!
    throw e;
}
```

Tekrar Fırlatmak - III

- Bazen bir sıra dışı durum yakalanıp tamamen farklı bir sıra dışı durumun fırlatıldığı da olur.
- Bu amaçla **`void initCause(Throwable t)`** metodu çağrılmaz ya da kurucu metoda yakalanan sıra dışı durum geçilmez.
  - Bu yaklaşım, çok teknik olan sıra dışı durumların yakalanmasında kullanılır.
  - Teknik sıra dışı durum bu yolla daha iş merkezli sıra dışı duruma döndürülmüş olur.

```
try {
   // access the database
} catch (SQLException ex) {
    logger.log(logLevel, message, ex);
    Throwable e = new  NoSuchProductException("No such product!");
    throw e;
}
```

Tekrar Fırlatmak - IV

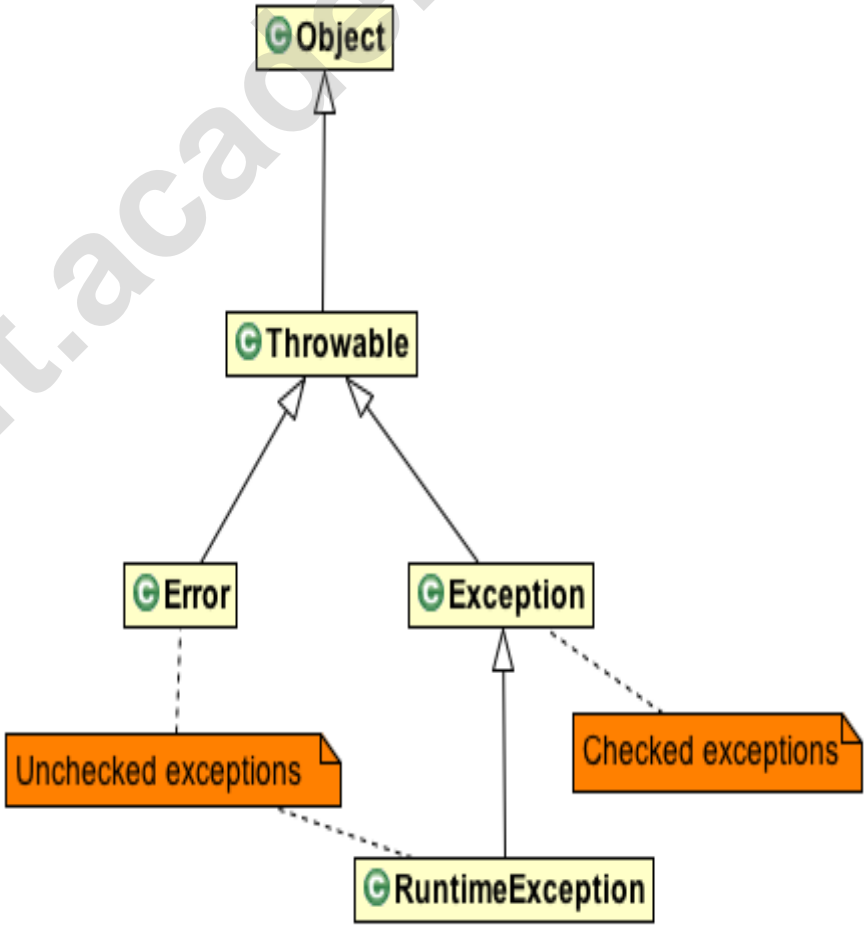
- Bunun en güzel örneği, **`java.sql.SQLException`** gibi bazı sıra dışı durum nesnelerinin, tek bir nesne ile olabilecek pek çok sıra dışı durumu temsil etmesidir.
- Böyle durumlarda bu sıra dışı durum yakalanır ve içindeki bilgilerden yola çıkarak farklı iş sıra dışı durumları fırlatılabilir.

```
int                     getErrorCode()

SQLException            getNextException()

String                  getSQLState()

Iterator<Throwable>     iterator()
```

Checked-Unchecked Exceptions

Checked ve Unchecked

- Java'da sıra dışı durumlar ikiye ayrılır: checked ve unchecked.
- **`Exception`** sınıfının alt sınıfı olup da **`RuntimeException`** olmayan sınıflar, checked exception olarak adlandırılırlar
- **`RuntimeException`** ve alt sınıfları ise unchecked exceptiondur.
  - `Error` de unchecked exceptiondur.



Checked Exceptions

- Checked exceptionlar ya yakalanmalı ya da metot arayüzünde throws ile fırlatıldığı belirtilmelidir.
- Daha önce geçen "Java'nın sıra dışı durumları yok görülemez, muhakkak yakalanmalı ..." cümlesi sadece checked exceptionlar için geçerlidir.
- Eğer bir kod parçası checked exception fırlatıyorsa bu durumda iki seçenek vardır:
  - Ya try-catch ile yakalamak,
  - Ya da yakalamayıp arayüzde throws ile fırlatıldığını belirtmek.
- Bu durum derleme zamanında kontrol edilir ve gerekirse derleme hatası verilir.

Unchecked Exceptions - I

- Unchecked exceptionlar yakalanmak ya da metot arayüzünde throws ile fırlatıldığı belirtilmek zorunda değildir.
- Çünkü unchecked exceptionlar programcı hatasıdır, düzeltilmelidirler.
- Tüm unchecked exceptionlar **`lava.lang.RuntimeException`** sınıfının ya da alt sınıflarının nesneleridir.
- **`RuntimeException`** sınıfının (ya da alt sınıflarının) nesnelerinin fırlatılması derleme zamanında kontrol edilmez ve yakalanmadığı ya da belirtilmediğinde de hata oluşmaz.

Unchecked Exceptions - II

- **`RuntimeException`** sınıfının (ya da alt sınıflarının) nesnelerinin fırlatılması derleme zamanında kontrol edilmez ve yakalanmadığı ya da belirtilmediğinde de hata oluşmaz.
  - Bu yüzden bu tür sıra dışı durumların atası olan sınıfa **`RuntimeException`** adı verilmiştir.

- Aslen checked olsun ya da olmasın tüm sıra dışı durumlar zaten çalışma zamanında oluşur.

Checked - Unchecked

- Checked exceptionlar umulan ve düzeltilebilecek durumları ifade eder, örneğin **`FileNotFoundException`**.
  - Bu yüzden çalışma zamanında yakalanıp düzeltilmelidirler.
- Ama unchecked exceptionlar programcı hatasıdır, çalışma zamanında oluşması beklenmemelidir.
  - Örneğin **`NullPointerException`**
- Eğer bir unchecked exception çalışma zamanında oluşuyorsa, kodda düzeltilmelidirler.
- Nihayetinde bir yazılım sisteminin kodu, çalışma zamanında hiç bir unchecked exception fırlatmayacak hale getirilmelidir.

Yaygın Unchecked Exceptionlar

- Yaygın unchecked exceptionlar:
  - **`java.lang.NullPointerException`**: Muhtemelen en yaygın unchecked exceptiondur. null referansı ifade eder.
  - **`java.lang.ArithmeticException`**: Matematiksel hesaplamalarda ortaya çıkabilecek durumlardır. "0"a bölmek gibi.
  - **`java.lang.ClassCastException`**: Nesnelerin yanlış alt sınıfa dönüştürülmelerinde (down cast) fırlatılır.
  - **`java.lang.IndexOutOfBoundsException`**: String ya da dizi (array) gibi yapılarda yanlış erişim durumlarını ifade eder.
  - **`java.lang.NumberFormatException`**: Girilen **`String`** nesnesinin sayı formatına dönüştürülmesi sırasında oluşan sıra dışı durumları ifade eder. **`IllegalArgumentException'un`** alt sınıfıdır.

RuntimeExceptionExample.java

Sıra Dışı Durum Sınıfı Oluşturmak

Sıra Dışı Durum Sınıfı Oluşturmak - I

- Kodunuzdaki sıra dışı durumları yönetmek için önce Java'nın ve kullandığınız diğer bileşenlerin ya da kütüphanelerin sıra dışı durumlarını kullanın.
  - Örneğin Hibernate'in **`Session`** sınıfındaki **`load()`** metodu veri tabanında girilen bir idye karşılık nesne bulunamadığında **`org.hibernate.ObjectNotFoundException`** sıra dışı durumu fırlatılır ve mesaj olarak da "*No row with the given identifier exists*" yazar.
  - Bu durumda bu sıra dışı durum yakalanıp işlenebileceği gibi, yakalanan nesneden alınacak bilgilerle "*BoyleBirUrunBulunamadı*" ya da "*NoSuchProduct*" gibi sıra dışı durumlar fırlatılabilir.

Sıra Dışı Durum Sınıfı Oluşturmak - II

- Bunlar yeterli değilse kendi sıra dışı durum sınıflarınızı oluşturun.
- Sıra dışı durum sınıfı oluşturmak için **`Exception`** sınıfından miras devralınır.
  - Tipik olarak kurucusuna **`String`** argüman geçilir.
- İhtiyaca göre sınıf üzerinde farklı değişkenler ve gerekli set/get metotları da oluşturulabilir.
  - Çünkü sıra dışı durum nesneleri bilgi taşıyabilir.

ShapeTest.java

Exception ve Overriding

Ezilen Metotların Sıradışı Durumları

- Üst tipteki bir metodun fırlatığı sıra dışı durumlar ile bu metodun devralınıp, override edildiği durumdaki fırlattığı sıra dışı durumlar arasındaki ilişki nedir?
- Üst tipteki metot override edildiğinde, override eden metot, ebeveyndeki metodun fırlattığı sıra dışı durumlardan fazlasını fırlatamaz ama daha azını fırlatabilir.
- Çünkü, ebeveynin istemcisi en çok ebeveyndeki metodun fırlattığı sıra dışı durumu beklemektedir, daha çoğunu beklememektedir.
  - Alt tipte, ebeveyndeki metodun fırlattığından daha çoğunu fırlatmak, yerine geçme özelliğini bozar.

Ezilen Metotların Sıradışı Durumları

- Dolayısıyla override eden metot şu üç durumdan birini seçebilir:
  - Aynı sıra dışı durumu fırlatmak,
  - Ebeveynde fırlatılan sıra dışı durumun bir alt tipini fırlatmak,
  - Hiç sıra dışı durum fırlatmamak.

HR.java & HRForManagerX.java

- *overriding* package

Assertion

Defensive Programming

- http://c2.com/cgi/wiki?DefensiveProgramming de : "Defensive programming defends against the currently impossible".

- Savunmacı programlama, olabilecek durumları önceden sezerek onlara karşı önlem almayı öngören bir programlama yaklaşımıdır.

Assertion

- Assertion kelime olarak öne sürme, iddia etme gibi anlamlara gelir.
  - Programlamada ise savunmacı programlamanın (defensive programming) bir tekniğidir.
  - Assertion, bir boolean değer üreten ve her zaman true olması beklenen durumdur.
    - Aksi taktirde hata (error) oluşur.

assert Anahtar Kelimesi - I

- **assert** anahtar kelimesi Java'ya 1.5 sürümüyle birlikte katılmıştır.
- assert kullanımının iki şekli vardır:
  - İlkinde assert kendisini takip eden ifadenin doğruluğunu test etmek için kullanılır.

```
assert expression;
```

  - İfade doğruysa çalışma devam eder, yanlışsa **java.lang.AssertionError** fırlatılır.
  - İkinci şekilde ise assert'ten sonra iki ifade vardır:

```
assert expression1 : expression2;
```

    - Çalışma ilk şekildeki gibidir. Tek fark ilk ifadenin yanlış olması halinde bir değer üreten ikinci ifadenin sonucunun **AssertionError**'in uygun bir kurucusuna geçilmesidir.

assert Anahtar Kelimesi - II

- assert kullanımı, maliyetli oluşundan dolayı çalışma zamanı için açılıp kapatılabilen bir özelliktir.

- Varsayılan durumda assert kullanımı kapalıdır.
  - Açmak için JVM'e **–enableassertions** ya da **–ea** seçeneklerini geçmek gereklidir, aksi taktirde assert ifadeleri çalışmaz.

- Bu yüzden assert ifadeleri kod geliştirme aşamasında, kodun doğru çalıştığından emin olmak için kullanılır ve canlı (live) ortamda kapatılır.

AssertionExample.java

Test Amaçlı assert Kullanımı - I

- assert kullanımının açılır-kapatılır bir özellik olması, bu yapının programcı tarafından test amacıyla serbestçe kullanılabilmesine imkan tanır.

- Bu yüzden assert genel olarak bir verinin geçerliliğini (invariant) test etmek için kullanılır.
  - Bu veri bir metottaki yerel değişken olabileceği gibi bir nesnenin durumu da olabilir.
  - Ön ve son şartları (pre & post conditions) test etmek için de kullanılır.

SqrtCalculator.java

PrePostConditionsExample.java

SwitchDemoWithAssertion.java

assert ve Sıra Dışı Durumlar

- assert kullanımının açılır-kapatılırdır ama sıra dışı durum yönetimi kalıcıdır.
- Dolayısıyla assert canlı ortamda olması muhtemel durumları yakalamak için kullanılmaz.
  - Zaten sıra dışı durum oluşunca yakalanır ve çalışma devam eder.
  - Ama **`AssertionError`** oluşunca JVM çalışmasını durdurur.
- assert kodun doğru çalıştığından emin olmak amacıyla, bir geliştirme zamanı yapısı olarak, bir durumun olmadığını test etmek için kullanılır ve canlı ortamda kapatılır.

Sıra Dışı Durum İçin Tavsiyeler

Tavsiyeler - I

- Sıra dışı durumları iş modelinizin parçası haline getirin.
  - Bu amaçla kendi **`Exception`** nesnelerinizi oluşturun.
- **`Exception`** nesnelerini kullanırken olabildiğince özel olun, genel olmayın.
  - Örneğin **`Exception`** nesnesini hiç bir zaman fırlatmayın, daima daha özel alt nesnelerini fırlatın.
- Fırlattığınız sıra dışı durum nesnelerine gerekli detayda veriyi koyun.
- Fırlatılan sıra dışı durumları API'de @throws ile belirtin ve açıklayın.

Tavsiyeler - II

- Ya kaynaklı try kullanın ya da finally bloğunda kaynaklarınızı kapatın.
- Sıra dışı durumları loglayın.
  - Bu amaçla bir loglama yapısı kurgulayın.

Özet

- Bu bölümde sıra dışı durum yönetimi ele alındı.
- **`Throwable`**, **`Error`** ve **`Exception`** sınıfları incelendi.
- try-catch-finally yapısı ele alındı.
- assert anahtar kelimesi ve savunmacı programlama ele alındı.
- Sıra dışı durum yönetimiyle ilgili tavsiyeler ele alındı.

Ödevler

Ödevler I

- Employee hiyerarşisinin bulunduğu örnekte, maaşı 7.000 TL ve üzerine olanlar için **`SalaryPaidOnBankException`** isimli bir sıra dışı durum nesnesi oluşturup örneği tekrar düzenleyin.

- Bazı Java frameworkleri checked exception kullanmayıp, tüm exceptionları **`RuntimeException`** yani unchecked exception olarak tanımlamaktadır. Bu tercihi tartışın.