Bölüm 6: Hata Ayıklama

JAVA ile Nesne Yönelimli Programlama

İstisna

- İstisna, programda **beklenmeyen** bir durumu temsil eder.
- Programın **normal akışını bozan** ve programcıya bir **hatanın** gerçekleştiğini bildirir.
- Kural dışı durumlar, Java çalışma zamanı sistemi tarafından otomatik olarak veya kod tarafından manuel olarak oluşturulabilir.
- **Örneğin**, bir dosyayı açma işlemi sırasında dosyanın bulunamaması.
- Java, bir istisna durumuyla karşılaştığında, bu durumu ele almak yerine "Pes ettim, şu an ne yapacağımı bilmiyorum. Sen hallet!" der.
- Programcı, bu istisnayı **ele alarak** hatayı çözebilir veya hatayı çağıran kodun sorunu olarak bırakabilir. İstisna mekanizması, programcılara anormal durumlarla başa çıkma şansı tanır.

İstisna

- **İstisna İle Başa Çıkma:**
  - Metot, belirli bir durumda istisna oluşturabilir ve bu durumu ele alabilir. Örneğin; Dosya okuma metodu dosya bulunamazsa FileNotFoundException fırlatabilir.
- **Çağıran Kodun Sorunu Yapma:**
  - Metot, olası bir istisna durumunu ele almıyorsa, bu istisna metodu çağıran kodun sorunu olur. Çağıran kod, bu istisna durumunu ele almak veya iletmek zorundadır.

İstisna

- Bir programın "bir şeyin ters gittiği" durumları belirtmek için kullanılır.
- Ancak, "yanlış" kelimesi öznel (*subjektif*) bir kavramdır.
- Örnek kod, eşleşme bulunamazsa istisna fırlatmak yerine -1 döndürür.

```
public int indexOf(String[] isimler, String isim) {
  for (int i = 0; i < isimler.length; i++) {
    if (isimler[i].equals(isim)) {
      return i;
    }
  }
  return -1;
}
```

İstisna

- Genellikle, -1 gibi dönüş kodlarından kaçınılmalı ve Java'nın istisna mekanizması tercih edilmelidir.
- İstisna, hata durumlarını açık ve etkili bir şekilde ele almanın bir yoludur.
- Kodun okunabilirliğini artırır ve hata ayıklamayı kolaylaştırır.
- **Öneri**: İstisnaları Kullanın!
  - Daha açık ve tutarlı bir hata işleme sağlar.
  - Kodunuzun güvenilirliğini artırır.

try ve catch İfadeleri

- Hata yönetimi için kullanılan temel yapıdır.
- **try** bloğu içinde potansiyel hata içeren kodlar bulunur.
- **catch** bloğu ise **try** bloğunda bir hata oluştuğunda bu hatayı ele alır.

```
try {
   // Potansiyel hata içeren kodlar
   int sonuc = 10 / 0; // Hata: ArithmeticException
} catch (ArithmeticException e) {
   // Hata durumuyla başa çıkma
   System.out.println("hata oluştu: " + e.getMessage());
}
```

Çoklu catch Bloğu

- Bir kod bloğunda birden çok istisna durumu oluşabilir.
- Çoklu hata türü için farklı *catch* blokları eklenir.
- Hatalar türlerine göre sırayla kontrol edilir ve uygun *catch* bloğu çalıştırılır.

```
try {
   // Potansiyel hata içeren kodlar
} catch (ArithmeticException e) {
   // ArithmeticException ile başa çıkma
} catch (NullPointerException e) {
   // NullPointerException ile başa çıkma
} catch (Exception e) {
   // Diğer hata türleri için genel durum
}
```

Zincirleme catch Blokları

- Farklı türdeki hataların spesifik olarak ele alınmasına imkan tanır.
- catch blokları sırayla eklenerek zincirleme bir yapı oluşturulabilir.
- İstisna durumu fırlatıldığında, her catch ifadesi sırayla kontrol edilir ve eşleşen ilk catch bloğu çalıştırılır.
- Eğer catch bloklarında kullanılan istisna sınıfları arasında kalıtım ilişkisi varsa, sıralama önemlidir. Alt sınıflar, üst sınıflardan önce gelmelidir.
- Her catch bloğu, belirli bir hata türüne özgü işlemleri içerir.
- Farklı hatalara özgü olarak uygun bir tepki verme imkanı sağlar.

finally Bloğu

- try ve catch bloklarının ardından **finally** bloğu eklenir.
- İstisna olsun veya olmasın her durumda çalıştırılır.
- Kodun belirli adımlarının çalıştırılmasını güvence altına alır.
- Kaynak yönetimi ve temizlik işlemleri için önemlidir.

```
try {
   // Potansiyel hata içeren kodlar
} catch (Exception e) {
   // Diğer hata türleri için genel durum
} finally {
   // Her durumda çalıştırılacak kodlar
}
```

İç İçe try Blokları

- Bir try bloğu içinde başka bir try bloğu bulunabilir.
- İçteki try bloğunda bir istisna durumu oluştuğunda:
  - İçteki try bloğunda uygun bir catch **bulunursa**, catch bloğu çalıştırılır.
  - **Bulunmazsa**, dışa doğru, ilk uygun catch bloğu bulunur ve çalıştırılır.
  - Hiçbir catch bloğu eşleşmezse, istisna durumu JVM'in istisna işleyicisi tarafından ele alınır.
  - Her durumda, finally bloğu (varsa) çalıştırılır.

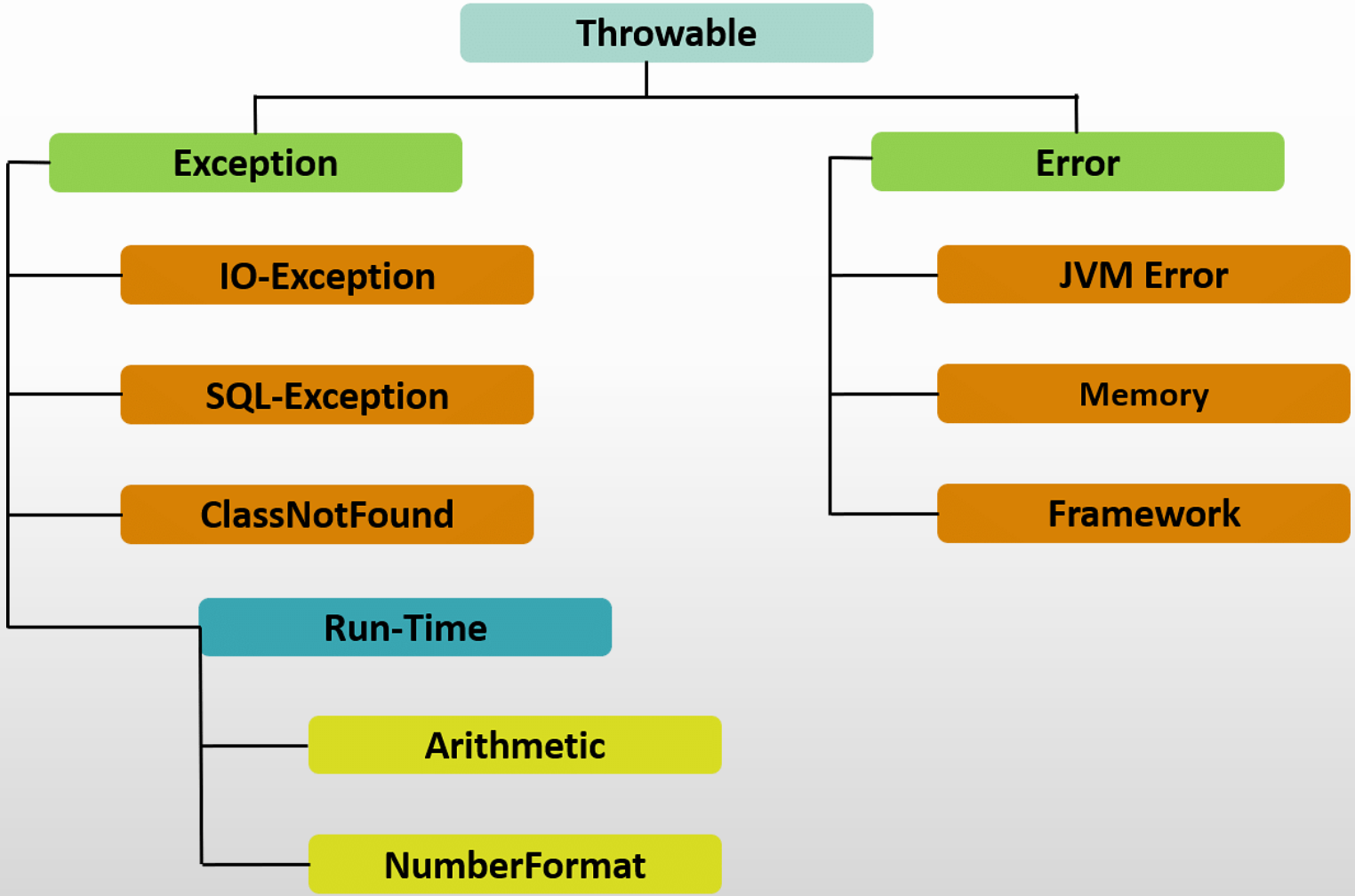
İç İçe try Blokları

```
try {
   // Dıştaki try bloğu
   try {
      // İçteki try bloğu
      // Potansiyel hata içeren kodlar
   } catch (Exception e) {
   // İçteki catch bloğu
   } finally {
   // İçteki finally bloğu
   }
} catch (Exception e) {
   // Dıştaki catch bloğu
} finally {
   // Dıştaki finally bloğu
}
```

İstisnalar Hiyerarşisi

- Java'da istisna sınıfları bir hiyerarşi içindedir.
- *Throwable* sınıfı, tüm istisna sınıflarının atasıdır.
- Java, yeni özel istisna sınıfların oluşturulmasını destekler.
- Uygulamaya özel durumlara daha iyi uyacak istisna sınıfları tanımlanabilir.

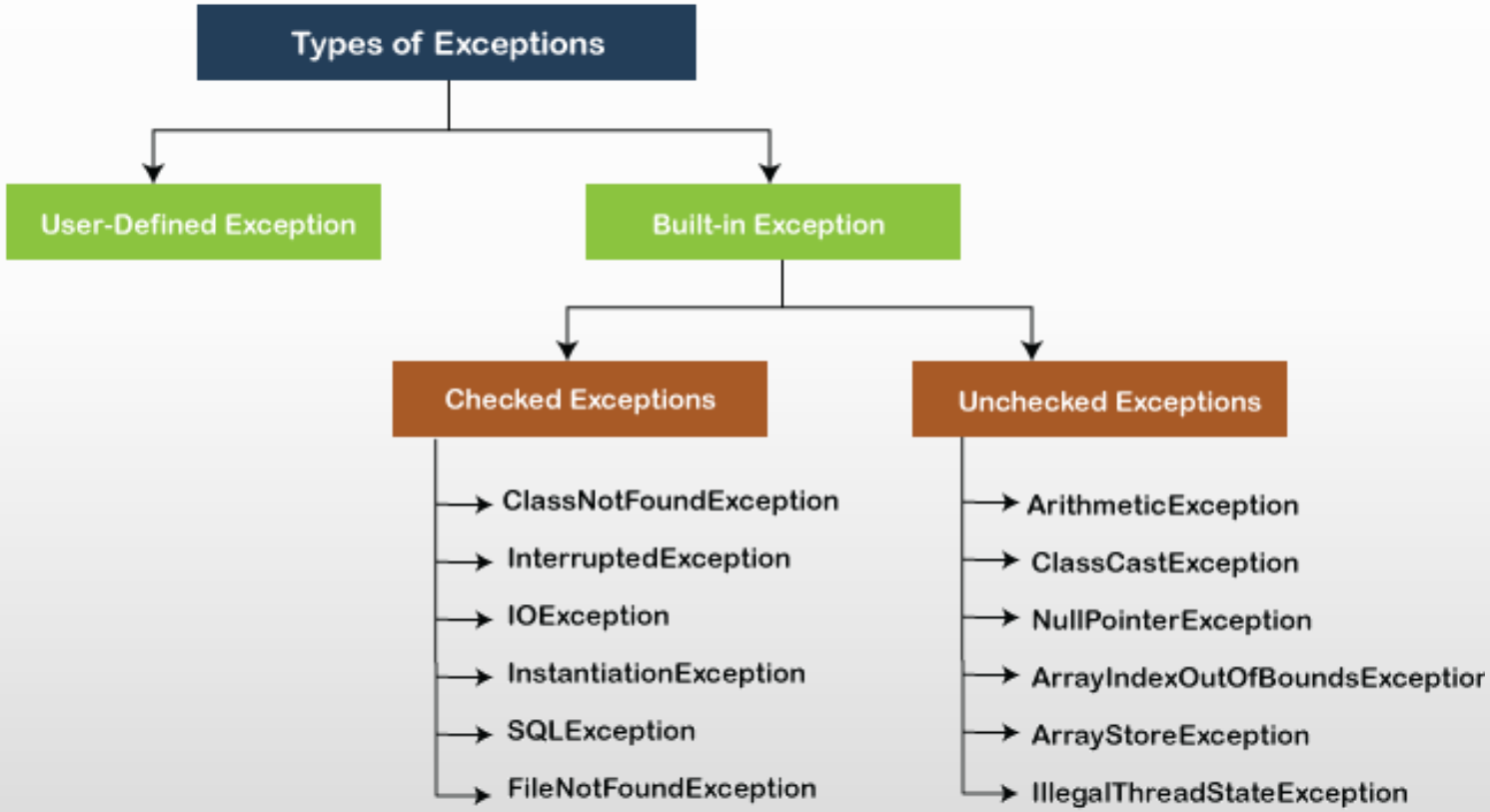
İstisnalar Hiyerarşisi



İstisna Türleri

- **Checked Exceptions:**
  - Derleme zamanında kontrol edilir, programcı tarafından ele alınır.
  - Ortaya çıkabileceği önceden bilinen durumları temsil eder.
- **Unchecked Exceptions:**
  - Derleme zamanında kontrol edilmez, çalışma zamanında ortaya çıkar.

İstisna Türleri



Kontrol Edilen İstisna

- Belirli bir uygulama kodu tarafından ele alınması veya bildirilmesi gerekir.
- Metodun başında throws anahtar kelimesi ile belirtilmeli veya try-catch blokları içinde ele alınmalıdır.
- RuntimeException sınıfından türetilmez.
- Bu tür istisna durumları derleme zamanında kontrol edilir.
- **Örneğin**, var olmayan bir dosyayı okumaya çalışmak olabilir.

```
public void dosyaOku() throws FileNotFoundException {
   // Dosya okuma kodu
}
```

Kontrol Edilen İstisna

```
public void dosyaOku() {
   try {
   // Dosya okuma kodu
   } catch (FileNotFoundException e) {
     // Kontrol edilen istisna durumu ele alınıyor
     System.err.println("Dosya bulunamadı:" + e.getMessage());
     // İstisna durumu tekrar fırlatılabilir
     // throw e;
   }
}
```

Kontrol Edilmeyen İstisna

- Belirli bir uygulama kodu tarafından ele alınması veya bildirilmesi zorunlu olmayan istisna türleridir.
- Genellikle mantık hataları veya programcı hatası sonucu ortaya çıkarlar.
- Metodun başında *throws* anahtar kelimesi ile belirtilmesi veya *try-catch* blokları içinde ele alınması zorunlu değildir.
- *RuntimeException* veya *Error* sınıflarından türetilir.

```
public void bolmeIslemi(int pay, int payda) {
   // Kontrol edilmeyen istisna durumu: ArithmeticException
   int sonuc = pay / payda;
}
```

Error ve Throwable

- *Throwable* sınıfı, tüm Java istisna sınıflarının atasıdır.
- *Error* ve *Exception* sınıfları, *Throwable* sınıfından türetilir.
- *Error*, programın kurtarılamayacak ciddi hata ile karşılaştığını ifade eder.
- Bu durumlarla başa çıkmak mümkün değildir ve genellikle programın sonlanmasına neden olur.
- **Örneğin**, disk sürücüsü "kayboldu" veya program belleği tükendi gibi aşırı durumlar.

İstisna Fırlatma (Throwing an Exception)

- Java çalışma zamanı, beklenmeyen bir durumla karşılaştığında, üst seviyedeki kodlara bildirmek için istisna nesnesi oluşturur ve fırlatır.
- İstisna, normal program akışını değiştirerek hatanın ele alınmasını sağlar.
- İstisna fırlatmak için *throw* anahtar kelimesi kullanılır:
- `throw new Exception("Bu bir istisna durumu!");`
- Java'da fırlatılabilen birçok istisna türü bulunur.
  - Örneğin, IOException, NullPointerException, kendi özel istisna sınıfları.
- Fırlatılan istisna, çağrı yapan kod tarafından *try-catch* blokları ile yakalanabilir. Programda istisna ele alınmamış ise, JVM programı sonlandırır ve neyin yanlış gittiğini ve nerede olduğunu bildirir.

İstisna Fırlatma (Throwing an Exception)

- İstisna türünü seçerken, durumu en iyi temsil eden sınıf seçilmelidir.
- Özel senaryolar için özel istisna sınıfları oluşturulabilir.

```
public void kontrolEt(int deger) throws CustomException {
  if (deger < 0) {
  // Belirli bir durumu kontrol et ve istisna fırlat
    throw new CustomException("Değer negatif olamaz!");
  }
  // Değer pozitifse, normal işlem devam eder
}
```

İstisna Fırlatan Metodları Çağırmak

- Metotları çağırırken, belirtilen istisna durumlarına dikkat edilmesi ve uygun hata yönetimi stratejilerinin kullanılması önemlidir.
- Metot imzalarında belirtilen istisnalar dikkate alınmalıdır.
- Metot çağrısı öncesinde uygun *try-catch* blokları eklenmelidir.
- Metodun çağırıldığı yerde, belirtilen istisna türlerine karşı *throws* anahtar kelimesi kullanılabilir. *try-catch* blokları kullanmaktan daha az tercih edilir.

İstisna Fırlatan Metodları Çağırmak

- Dosya Okuma Metodu:

```
public void dosyaOku() throws FileNotFoundException {
   // Dosya okuma kodu
}
```

- Metodu Çağırma:

```
try {
   dosyaOku();
} catch (FileNotFoundException e) {
   // Dosya bulunamadı durumuyla başa çıkma kodu
   System.err.println("Dosya bulunamadı: " + e.getMessage());
}
```

RuntimeException Sınıfları

RuntimeException Sınıfları

- Programcı hatası veya mantık hataları sonucu ortaya çıkan istisna durumlarını temsil eder. Derleme zamanında kontrol edilmez ve programcının bilinçli olarak ele alması gerekir.
- Yaygın RuntimeException sınıfları:
  - NullPointerException,
  - ArrayIndexOutOfBoundsException,
  - ArithmeticException,
  - IllegalArgumentException,
  - NumberFormatException,
  - ClassCastException

NullPointerException

- Null referanslarla çalışma durumları, programcıların dikkat etmesi gereken yaygın hata kaynaklarından biridir.
- **Objects.requireNonNull:** Belirli bir nesnenin null olup olmadığını kontrol eder ve null ise bir *NullPointerException* fırlatır.
- **Optional** sınıfı, *null* kontrolü yapmadan güvenli şekilde işlem yapmayı sağlar.
- **Objects.requireNonNullElse:** Belirtilen nesnenin null olup olmadığını kontrol eder ve null ise varsayılan bir değer döndürür.
- **Apache Commons Lang** kütüphanesindeki *StringUtils* sınıfı, null güvenli metin işlemleri sağlar.

NullPointerException

```
String str = null;

// NullPointerException
int length = str.length();

// NullPointerException
Objects.requireNonNull(str, "Str null olamaz");

Optional<String> optionalStr = Optional.ofNullable(str);
length = optionalStr.map(String::length).orElse(0);

String defaultValue = Objects.requireNonNullElse(str, "0");
```

ArrayIndexOutOfBoundsException

- Dizinin sınırları dışında bir indekse erişme durumudur.
- **ArrayList** sınıfı, dinamik bir diziye benzese de, dizinin belirli bir boyutunu aşan indekslere erişmeye çalıştığında bu istisna durumu fırlatır.
- **Vector** sınıfı, dinamik bir dizi yapısına sahiptir ancak belirli bir boyutu aşan indekslere erişimde bu istisna durumu ortaya çıkar.
- **Arrays** sınıfı, dizi işlemleri sağlar ve belirli bir boyutu aşan indekslere erişimde bu istisna durumu fırlatabilir.
- **String** sınıfı, karakter dizileri üzerinde işlem yaparken belirli bir boyutu aşan indekslere erişimde bu istisna durumu oluşabilir.

ArrayIndexOutOfBoundsException

```
int[] numbers = {1, 2, 3};
int value = numbers[5]; // Exception

ArrayList<String> list = new ArrayList<>();
String svalue = list.get(5); // Exception

Vector<Integer> vector = new Vector<>();
value = vector.get(10); // Exception

String[] names = {"Alice", "Bob", "Charlie"};
svalue = Arrays.asList(names).get(5); // Exception

String str = "Hello";
char ch = str.charAt(10); // Exception
```

ArithmeticException

- Matematiksel işlemler sırasında ortaya çıkan hata durumunu temsil eder.
- Aritmetik işlem hatası, örneğin, bir sayıyı sıfıra bölmek.
- *Math* sınıfındaki bazı işlemler, *ArithmeticException* fırlatabilir.
- **Random** sınıfı, belirli bir aralıkta rastgele sayı üretirken sıfıra bölme durumlarına dikkat edilmelidir.
- **BigDecimal** sınıfı, yüksek hassasiyetli aritmetik işlemler sağlar, ancak sıfıra bölme durumunda *ArithmeticException* fırlatabilir.

ArithmeticException

```
int result = 10 / 0; // Exception


double dresult = Math.sqrt(-1);
System.out.println(dresult); // NaN


Random random = new Random();
int randomNumber = random.nextInt(0); // Exception


BigDecimal num1 = new BigDecimal("10");
BigDecimal num2 = BigDecimal.ZERO;
BigDecimal bresult = num1.divide(num2); // Exception
```

IllegalArgumentException

- Bir metodun geçersiz bir argüman aldığı durumu ifade eder.
- **Objects** sınıfı, nesne işlemleri sağlar ve geçersiz argümanlar durumunda *IllegalArgumentException* fırlatabilir.
- **File** sınıfı, dosya işlemleri sırasında geçersiz dosya yolları durumunda *IllegalArgumentException* fırlatabilir.
- **Thread** sınıfı, geçersiz bir *Runnable* nesnesi aldığında *IllegalArgumentException* fırlatabilir.
- **Arrays** sınıfı, diziler üzerinde işlemler yaparken geçersiz argümanlar durumunda *IllegalArgumentException* fırlatabilir.

IllegalArgumentException

```
if (age < 0) {
  throw new IllegalArgumentException("Yaş negatif olamaz");
}

if (name.length() < 3) {
  throw new IllegalArgumentException("3 karakter olmalı");
}

File file = new File("invalid/file/path");
FileReader reader = new FileReader(file); // Exception

Runnable invalidRunnable = null;
Thread thread = new Thread(invalidRunnable); // Exception

int[] numbers = {1, 2, 3};
Arrays.copyOfRange(numbers, 5, 2); // Exception
```

NumberFormatException

- Bir dizeyi sayıya dönüştürme işlemi sırasında, dize içinde geçerli bir sayı temsil etmiyorsa ortaya çıkan istisna durumudur.
- **Integer** sınıfındaki bazı dönüştürme yöntemleri, geçerli bir sayıyı temsil etmeyen dizelerle kullanıldığında *NumberFormatException* fırlatabilir.
- **Double** ve **Float** sınıflarındaki dönüştürme yöntemleri, geçerli bir ondalık sayıyı temsil etmeyen dizelerle kullanıldığında *NumberFormatException* fırlatabilir.
- **Short**, **Long**, **Byte** sınıflardaki dönüştürme yöntemleri de geçerli bir sayıyı temsil etmeyen dizelerle kullanıldığında *NumberFormatException* fırlatabilir.

NumberFormatException

```
String strNumber = "abc";
int number = Integer.parseInt(strNumber); // Exception

strNumber = "123abc";
number = Integer.parseInt(strNumber); // Exception

String strDouble = "12.34abc";
double doubleValue = Double.parseDouble(strDouble);

String strShort = "45xyz";
short shortValue = Short.parseShort(strShort); // Exception
```

ClassCastException

- Bir nesnenin beklenen bir türden değil de başka bir türden olduğu durumlarda ortaya çıkan istisna durumudur.
- **ArrayList** gibi koleksiyon sınıfları, yanlış türdeki nesneleri almak istendiğinde ClassCastException fırlatabilir.
- **Map** arayüzünde, yanlış türde anahtar veya değer kullanımı ClassCastException'a neden olabilir.
- **Set** ve **List** arayüzlerinde yanlış türde eleman eklemek veya almak ClassCastException'a yol açabilir.
- Arrays sınıfındaki bazı metotlar, beklenmeyen türde dizilerle çalıştığında ClassCastException fırlatabilir.

ClassCastException

```
Object obj = "Merhaba";
Integer number = (Integer) obj; // ClassCastException
ArrayList list = new ArrayList();
list.add("Merhaba");
number = (Integer) list.get(0); // ClassCastException
Map map = new HashMap();
map.put("anahtar", "Değer");
Integer value = (Integer) map.get("anahtar"); // ClassCastException
Set set = new HashSet();
set.add("Merhaba");
value = (Integer) set.iterator().next(); // ClassCastException
Object[] objArray = new String[5];
Integer[] intArray = (Integer[]) objArray; // ClassCastException
```

Kontrollü İstisna Sınıfları

Kontrollü İstisna Sınıfları

- Kodun hatayı ele alması ve uygun bir şekilde geri dönmesi gereken durumları temsil eder.
- Yaygın kontrollü istisna sınıfları:
  - FileNotFoundException
  - IOException
  - SQLException
  - ParseException
  - ClassNotFoundException

FileNotFoundException

- Bir dosya açılmaya çalışıldığında, belirtilen dosya bulunamazsa ortaya çıkan istisna.

```
try {
   FileReader fileReader = new FileReader("dosya.txt");
} catch (FileNotFoundException e) {
   // Dosya bulunamadı durumuyla ilgili işlemler
}
```

IOException

- Giriş/Çıkış işlemleri sırasında genel bir hata durumunu temsil eden istisna.

```
try {
   BufferedReader reader = new BufferedReader(new FileReader(path));
   String line;
   while ((line = reader.readLine()) != null) {
      System.out.println(line);
   }
   reader.close();
} catch (IOException e) {
   // IOException durumuyla ilgili işlemler
   e.printStackTrace();
}
```

SQLException

- Veritabanı işlemleri sırasında SQL hatası oluştuğunda fırlatılan istisna.

```
try {
  // Sorguyu çalıştırma ve sonuçları alıp işleme
  ResultSet resultSet = preparedStatement.executeQuery();
  while (resultSet.next()) {
    // Sonuçları işleme
    System.out.println("User ID: " + resultSet.getInt("user_id"));
  }
  // Bağlantıyı kapatma
  connection.close();
} catch (SQLException e) {
  // SQLException durumuyla ilgili işlemler
  e.printStackTrace();
}
```

ParseException

- Bir dizeyi belirli bir formata ayrıştırma işleminde hata oluştuğunda fırlatılan istisna.

```
try {
   SimpleDateFormat form = new   SimpleDateFormat("dd/MM/yy");
   Date date = form.parse("01-01-2022");
} catch (ParseException e) {
   // Ayrıştırma istisnası durumuyla ilgili işlemler
}
```

ClassNotFoundException

- Bir sınıfın yüklenirken, belirtilen sınıf bulunamazsa fırlatılan istisna sınıfıdır.

```
try {
   // Var olmayan bir sınıfı yüklemeye çalışma
   Class.forName("com.example.NonExistentClass");
} catch (ClassNotFoundException e) {
   // ClassNotFoundException durumuyla ilgili işlemler
   e.printStackTrace();
}
```

NotSerializableException

- Nesnenin serileştirilememesi durumunda ortaya çıkar.
- Serileştirme, bir nesnenin veri akışına dönüştürülmesi ve daha sonra geri dönüştürülmesi işlemidir.
- *Serializable* arayüzünü uygulayan bir sınıf, serileştirilebilir bir sınıf olarak kabul edilir.

```
try {
   // Nesneyi ObjectOutputStream ile serileştirmeye çalışma
   outputStream.writeObject(notSerializableObj);
} catch (NotSerializableException e) {
   // NotSerializableException durumuyla ilgili işlemler
   e.printStackTrace();
}
```

Error Sınıfları

Error Sınıfları

- Hatalar, programın normal çalışmasını etkiler.
- Sistem düzeyinde sorunları temsil eden kontrolsüz istisna durumlarıdır.
- Uygulama kodundan kaynaklanmaz, genelde sistem düzeyinde sorunlar.
- Bu tür hatalar JVM tarafından fırlatılır ve ele alınması veya bildirilmesi önerilmez. Programın düzgün bir şekilde sonlandırılması daha uygun bir yaklaşımdır.
- Örnek Hatalar:
  - **OutOfMemoryError**: Bellek tükenmesi durumunda ortaya çıkar.
  - **StackOverflowError**: Yığın aşımı durumunda fırlatılır.
  - **AssertionError**: assert ifadesi başarısız olursa ortaya çıkar.

Özel İstisna Sınıfları

- Geliştirici, kendi programında ortaya çıkabilecek beklenmeyen durumları yönetmek için özel istisna sınıfları oluşturabilir.
- Özel istisna sınıfları, *Throwable* sınıfını genişletirse, bu sınıflar bir istisna olarak fırlatılabilir.
- Ancak, genellikle *Exception* veya *RuntimeException* sınıflarını genişletmek tercih edilir.

```
public class OzelHata extends Exception {
   // Constructor
   public OzelHata(String message) {
      super(message);
   }
}
```

Özel İstisna Sınıfları

```
public static void main(String[] args) {
   try {
      fonksiyon();
   } catch (OzelHata e) {
      System.out.println("Yakalandı: " + e.getMessage());
   }
}

public static void fonksiyon() throws OzelHata {
   if (true) {
      throw new OzelHata("Özel hata oluştu.");
   }
}
```

İzleme Listesi (Stack Trace)

- Programın çalıştığı sırada hangi metot çağrımlarının gerçekleştiğini ve istisna durumunun nerede oluştuğunu gösteren rapordur.
- Metodun adı, sınıfı, dosya adı ve hatanın ne olduğu gibi bilgileri içerir.
- Yakalanmayan bir istisna durumunda, JVM tarafından konsola yazdırılır.
- Programcının hatayı anlamasına ve sorunu çözmesine yardımcı olur.

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.MyClass.myMethod(MyClass.java:10)
    at com.example.MyClass.main(MyClass.java:6)
```

SON