ENUM'LAR

Enum'lar, en kısa ifadesiyle, önceden belirlenmiş sınırlı sayıda sabit değerleri alabilen yeni bir tür oluşturmak için kullanılır. Yani, enum türünde bir değişken yalnızca kendi listesinde belirlenmiş değerlerden birini alabilir, başka bir değere izin verilmez. Başka bir deyişle, enum'lar alabileceği değerleri kendiniz belirlediğiniz yeni bir veri türü oluşturmanızı sağlar. Enum'lar JDK 5 ile dile eklenmiştir.

Aşağıdaki örneği inceleyelim. Bu örnekte haftanın günlerini enum olarak belirttik:

```
enum WeekDays
{
      MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}
```

Enum'lar **enum** deyimiyle oluşturulur ve her bir değer birbirinden virgül ile ayrılır. Yukarıdaki kod sayesinde, *WeekDays* adında yeni bir tür oluşturduk: bu türün yalnızca 7 farklı değeri olabilir.

Şimdi bu değerleri değişken olarak nasıl kullanacağımızı görelim:

```
WeekDays midDay = WeekDays.THURSDAY;
```

Bir enum değerini kullanmak için enum türünün adından sonra nokta koyulur ve değerin ismi yazılır.

Enum'lar switch ifadelerini kontrol etmek için kullanılabilir:

```
WeekDays day;

switch (day)
{
      case MONDAY:
            System.out.println("Pazartesi sendromu");
            break;
      case FRIDAY:
            System.out.println("Cuma sevinci");
            break;
}
```

values() ve valueOf() metotları

Bütün enum türlerinde ortak olan iki static metot vardır. Bu metotların tanımı aşağıdaki gibidir:

public static *enum-türü*[] values()

public static *enum-türü* valueOf(String isim)

values() metodu bir enum türünün bütün değerlerini dizi halinde verir. Örneğin, aşağıdaki kodu çalıştırdığımızda konsola 7 yazar:

```
System.out.println(WeekDays.values().length);
```

valueOf() metodu ismini bildiğimiz bir enum değerine ulaşmamızı sağlar:

```
WeekDays monday = WeekDays.valueOf("MONDAY");
```

Şuna dikkat etmelisiniz: *valueOf()* metodu, eğer parametre olarak verdiğiniz isimde bir enum değeri yoksa *IllegalArgumentException* hatası fırlatır:

```
WeekDays.valueOf("SALI");    // Bu satır hata fırlatır
```

name() ve ordinal() metotları

Bir enum değerinin ismini elde etmek için *name()* metodunu, enum türü içinde belirtildiği sırayı elde etmek için *ordinal()* metodunu kullanırız.

```
System.out.println(WeekDays.MONDAY.name());    // Konsola MONDAY yazar
```

```
System.out.println(WeekDays.MONDAY.ordinal());    // Konsola 0 yazar
```

Şuna dikkat etmelisiniz: enum değerlerinin ordinal sıralaması 0'dan başlar.

Java enum'ları sınıflara benzer

Diğer birçok programlama dilinde enum'lar sabit değerlerden ibarettir. Java'da ise enum'lar birer sınıftır. Enum değerlerini **new** deyimiyle oluşturmasanız da sınıfların çoğu özelliğine sahiptirler. Tıpkı sınıflar gibi, enum'ların da alanları ve metotları olabilir. Fakat enum'lar miras alıp veremez.

Aşağıdaki örnekte *WeekDays* enum'unu yeniden yazalım:

```
public enum WeekDays
{
    MONDAY("Pazartesi"),
    TUESDAY("Salı"),
    WEDNESDAY("Çarşamba"),
    THURSDAY("Perşembe"),
    FRIDAY("Cuma"),
    SATURDAY("Cumartesi"),
    SUNDAY("Pazar")
    ;

    private final String turkishName;

    WeekDays(String turkishName)
    {
        this.turkishName = turkishName;
    }

    public String getTurkishName()
    {
        return turkishName;
    }

}
```

Bu örnekte enum'a *turkishName* adında final bir alan ekledik. Bu değişken sabit olduğu için değer atamak adına bir yapılandırıcı oluşturduk. Enum değerlerini tanımlarken yapılandırıcıları nasıl kullandığımıza dikkat edin.

```
System.out.println(WeekDays.MONDAY.getTurkishName());
```

Yukarıdaki kodu çalıştırdığımızda konsola "Pazartesi" yazar.

Java enum'larının sınıflara benzemesinin enum'lara ne kadar esneklik kattığını fark etmişsinizdir.

HATA YÖNETİMİ (EXCEPTION HANDLING)

Bu bölümde Java'nın hata yönetim mekanizmasını inceleyeceğiz. Java dilinde hatalara **istisna** (*exception*) denir. İstisna, adından da anlaşılacağı üzere, programın çalışması sırasında meydana gelen istisnai (anormal) durumları belirtir. Diğer bir deyişle, çalışma zamanında meydana gelen hatalara istisna denir.

Program yazarken belli kurallar yazmış oluruz. Java çalışma ortamı, programımızı bu kurallara göre çalıştırır. Hata ise istisnai (yani kural dışı) bir durumdur. Böyle bir durumda Java çalışma ortamı ne yapacağını bilemez, çareyi programı sonlandırmakta bulur. Kısacası, bir hata oluştuğu zaman programın çalışması sona erer. Bunun önüne geçebilmek için, kodun yazılış aşamasında hata yönetiminin iyi yapılması gerekir.

Hata yönetiminden kastımız, hatanın meydana gelmesini engellemek değildir. Hata yönetimi, en basit ifadesiyle, çalışma sırasında bir hata meydana gelse bile programın çalışmasına devam etmesini sağlamaktır. Java'da bu mümkündür. Java'nın hata yönetim mekanizması sayesinde, program normal akışında çalışır; eğer bir hata olursa, yazdığımız koda uygun olarak bir aksiyon alınır (kullanıcıya hata bilgilendirmesi yapılır, hata kayıt altına alınır vs.) ve sonra program çalışmasına devam eder.

Java'nın hata yönetim mekanizması şu şekilde işler: Programın çalışması sırasında istisnai bir durum oluşursa bu durumla ilgili bir nesne oluşturulur ve **throws** deyimiyle fırlatılır. Böyle bir durumda, programın olağan akışı durdurulur ve bu hatanın yakalanması beklenir. Hatanın yakalanabilmesi için, hataya sebep olan kodun **try-catch** bloğu içine yazılması gerekir. Bu durumda Java çalışma ortamı, meydana gelen hatayı yakalayabilecek bir **catch** bloğu arar, eğer bulursa bu catch bloğu çalıştırılır. Son olarak, eğer bir **finally** bloğu yazılmışsa bu blok çalıştırılır ve program normal akışına devam eder.

Java hataları yayılımcı hatalardır. Bunun anlamı şudur: hatanın meydana geldiği metot içinde yakalanması gerekir, aksi halde hata bir üst metoda (çağıran metoda) aktarılır. Hata yakalanmadığı sürece bir üst metoda aktarılmaya devam eder. Eğer yazdığımız kodun hiçbir yerinde hatayı yakalamadıysak, hata Java çalışma ortamına aktarılır. Java çalışma ortamı bize

varsayılan bir hata yakalama mekanizması sunar. Bu mekanizma hata meydana geldiğinde programı sonlandırır.

Java'nın hata sınıflarının yapısı şu şekildedir:

- *Throwable*
  - *Error*
  - *Exception*
    - *RuntimeException*

En üst hata sınıfı *Throwable* sınıfıdır. Java'da bir sınıfın hata olarak değerlendirilebilmesi için *Throwable* sınıfının alt sınıfı olması gerekir. Bu sınıfın iki alt sınıfı vardır: *Error* ve *Exception*. *Exception* sınıfı programcıların yakalaması gereken hataları ifade eder. Aynı zamanda, kendi hata türümüzü oluşturmak istediğimiz zaman bu sınıftan kalıtım alırız. *Exception* sınıfının *RuntimeException* adında bir alt sınıfı vardır. Bu sınıf çalışma zamanında meydana gelen ve yakalamak zorunda olmadığımız hataları ifade eder.

Error sınıfı ise normal şartlar altında sıkça karşılaşmayacağımız hataları ifade eder. Bu tür hatalar genelde programımızla ve yazdığımız kodla alakalı olmayıp çalıştırılan ortamla ilgilidir. Hafızanın dolması hatası buna bir örnektir.

Şimdi **try-catch** bloğunun yapısını inceleyelim:

```
try
{
      // Hata oluşması muhtemel kodlar buraya yazılır
}
catch (ExceptionSınıfı1 ex)
{
      // ExceptionSınıfı1 hatası meydana gelirse burası çalıştırılır
}
catch (ExceptionSınıfı2 ex)
{
      // ExceptionSınıfı2 hatası meydana gelirse burası çalıştırılır
}
finally
{
      // Son olarak burası çalıştırılır
}
```

İlk olarak **try** bloğu yazılır. Buraya hata oluşturması muhtemel kodlar yazılır. Böylelikle hata yönetim mekanizmasını başlatmış oluruz. Daha sonra **catch** blokları yazılır. Birden fazla catch bloğu yazabilirsiniz. Bu blokların her biri belli hatalara yöneliktir. catch bloğunu açtıktan sonra parantez içinde hatanın türünü yazar ve sonra bir değişken ismi belirleriz. Bu türde bir hata meydana gelirse bu catch bloğu çalıştırılır ve hatayla ilgili *Exception* nesnesi bize bu değişken içinde verilir. Son olarak **finally** bloğu yazılır. Bu bloğu yazmak zorunlu değildir. Anlamamız gereken önemli bir nokta, finally bloğunun hata oluşsa da oluşmasa da çalıştırılacağıdır.

Şimdi bir hata yakalama örneği yapalım:

```
int a = 0;
System.out.println(5 / a);     // Bu satır hataya sebep olur
System.out.println("Burası çalıştırılmaz.");
```

Yukarıdaki kod çalıştırıldığında sıfıra bölme işlemi yapıldığı için hata meydana gelir ve program sonlanır. Dolayısıyla en alttaki satır çalıştırılmaz. Çıktısı aşağıdaki gibi olur:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at Exception1.main(Exception1.java:7)
```

Şimdi bu kodları hata yönetimi mekanizmasıyla yeniden yazalım:

```
try
{
    int a = 0;
    System.out.println(5 / a);
}
catch (ArithmeticException ex)
{
    System.out.println("Sıfıra bölme hatası meydana geldi.");
}
finally
{
    System.out.println("Burası çalıştırılır.");
}
```

Bu kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Sıfıra bölme hatası meydana geldi.
Burası çalıştırılır.
```

Gördüğünüz gibi, hataya sebep olan satırı **try** bloğu içinde yazdığımız ve hatayla ilgili bir **catch** bloğu oluşturduğumuz için programımız çalışmasına devam etmiştir.

Hata fırlatmak

Fırlatılan hataları **try-catch** blokları içinde yakalayabiliriz. Diğer yandan, kendimiz de hata oluşturup fırlatabiliriz. Hata kelimesi olumsuz bir anlam çağrıştırdığından, hata fırlatmanın kötü bir yöntem olduğunu düşünebilirsiniz. Fakat yeri geldiğinde hata fırlatmak da faydalı olabilir. Hataları fırlatmak için **throw** deyimini kullanırız. Hata fırlatmak için hatayla ilgili bir *Exception* nesnesi oluşturmalıyız. Aşağıdaki örneği inceleyelim:

```
public class Person
{

    private int age;

    public void setAge(int age)
    {
        if (age < 0)
        {
            throw new IllegalArgumentException("Yaş sıfırdan küçük
                  olamaz!");
        }

        this.age = age;
    }

}
```

Yukarıdaki örnekte *Person* isminde bir sınıf oluşturduk. Bu sınıfın *age* isminde bir alanı vardır. Kişinin yaşını ayarlamak istediğinizde, *setAge()* metodunu kullanır ve parametre olarak yaş değerini veririz. Fakat yaş değeri sıfırdan küçük olamayacağı için, negatif değerler için *IllegalArgumentException* türünde bir hata fırlattık. Bu hataya parametre olarak bir hata mesajı verdik. *Exception* türünde nesneleri oluştururken parametre olarak *String* türünde hata mesajı verebilirsiniz.

Hatayı metot tanımında belirtmek

Bir metot yazarken hata fırlatabilecek bir metot çağırıyorsak, ya metodun içerisinde **try-catch** bloğuyla bu hatayı yakalamalı ya da hata yakalamayı bir üst metoda bırakmalıyız. Fakat bu durumda, çağıran metodun bu hatadan haberdar olabilmesi için metodun hata fırlatabileceğini metodun tanımında belirtmeliyiz. Bunu **throws** deyimiyle yaparız. Yukarıdaki örneği aşağıda tekrar yazalım:

```
public class Person
{

    private int age;

    public void setAge(int age) throws IllegalArgumentException
    {
        if (age < 0)
        {
            throw new IllegalArgumentException("Yaş sıfırdan küçük
                  olamaz!");
        }

        this.age = age;
    }

}
```

Gördüğünüz gibi, *setAge()* metodunun hata fırlatabilecek bir metot olduğunu **throws** deyimiyle metot tanımında belirttik. Artık bu metodu çağıran metotlar da bu hatayı yakalamak veya bir üst metoda bırakmak zorundadır.

Java'daki bazı hata sınıfları

Java'da bazı ortak hatalar için önceden tanımlanmış hata sınıfları vardır. Bunlardan en çok karşılaşılanları kısaca inceleyelim:

- **ArithmeticException:** Sıfıra bölme başta olmak üzere matematiksel hataları belirtir.
- **ArrayIndexOutOfBoundsException:** Bir dizinin aralığı dışında elemanına erişmeye çalışıldığında fırlatılır.
- **ClassCastException:** Geçersiz tür dönüşümü işlemlerinde fırlatılır.
- **IllegalArgumentException:** Metoda verilen parametrelerden biri hatalı olduğunda fırlatılır.
- **IndexOutOfBoundsException:** Hatalı indeks erişimlerinde fırlatılır.
- **NullPointerException:** Henüz değer ataması yapılmamış değişkenler üzerinde işlem yapmaya çalışıldığında fırlatılır. Java'da en çok karşılaşılan hatalardan biridir. Bu hataya karşı önlem almak için yaptığımız kontrollere **null kontrolü** (*null-check*) denir.
- **NumberFormatException:** Bir *String* değerini sayısal bir türe dönüştürmeye çalıştığımızda, eğer *String* değer düzgün bir sayı ifade etmiyorsa fırlatılır.

- **UnsupportedOperationException:** Desteklenmeyen bir iş yapmaya çalışıldığında fırlatılır.

JENERİKLER (GENERICS)

Jenerikler, kelime anlamı itibariyle **parametrelendirilmiş tür** anlamına gelir. Jenerikler sayesinde, sınıf, arayüz veya metot yazarken tek bir türe bağlı kalmayıp üzerinde işlem yapacağınız türü parametre olarak alabilirsiniz. Bu sayede, farklı türler üzerinde çalışan tek bir sınıf yazmak mümkündür. Bu şekilde yazılan sınıflara **jenerik sınıf**, metotlara **jenerik metot** denir.

Jenerikler JDK 5 ile dile eklenmiştir. Buna rağmen, Java'nın en temel özelliklerinden biridir ve dili temelden etkilemiştir. Bu yüzden, jenerikleri iyi anlamak Java'yı öğrenmek açısından büyük önem taşır.

Jenerikler **tür güvenliğini** (*type-safety*) sağlamak amacıyla dile eklenmiştir. Java'nın **tür kesinliği olan** (*strongly typed*) bir dil olduğunu daha önce belirtmiştik. Fakat bu iki kavram birbirinden farklıdır. Tür güvenliği kavramını ve neden gerekli olduğunu anlamak için bir örnek yapalım.

Nullable adında basit bir sınıf yazalım. Bu sınıfı *NullPointerException* hatalarının önüne geçmek amacıyla kullanacağız. İlk olarak bu sınıfı *String* değerler üzerinde kullanalım:

```
public class Nullable
{

    private final String value;

    public Nullable(String value)
    {
        this.value = value;
    }

    public String getValue()
    {
        return value;
    }

    public boolean isNull()
    {
        return value == null;
    }

    @Override
    public String toString()
    {
        return isNull() ? "null" : value;
    }

}
```

String nesneler üzerinde uğraşırken null hatalarının önüne geçmek için bu sınıfı kullanacağız. Sınıfı oluştururken parametre olarak bir *String* değer vereceğiz. Bu değerin **null** olup olmadığını *isNull()* metoduyla kontrol edeceğiz. Şimdi örnek bir kullanım gösterelim:

```
Nullable x = new Nullable("null olmayan değer");
if (!x.isNull())
{
    System.out.println(x.getValue());
}

String nullString = null;
Nullable y = new Nullable(nullString);
if (y.isNull())
{
    System.out.println("y değişkeni null");
}
```

Yukarıdaki kodu çalıştırdığınız zaman çıktısı aşağıdaki gibi olur:

```
null olmayan değer
y değişkeni null
```

Gördüğünüz gibi yazdığımız sınıf güzel bir şekilde çalışmaktadır. Peki ya bu sınıfı *String* türünden başka türler için de kullanmak istersek? Mevcut durumda *Nullable* sınıfı yalnızca *String* değerleri kabul ediyor. Yani bu sınıfı *Date* türündeki değerler üzerinde kullanamayız. O halde bu sınıfın adını *NullableString* olarak değiştirelim ve *NullableDate* adında başka bir sınıf oluşturalım:

```
import java.util.Date;

public class NullableDate
{
    private final Date value;

    public NullableDate(Date value)
    {
        this.value = value;
    }

    public Date getValue()
    {
        return value;
    }

    public boolean isNull()
    {
        return value == null;
    }

    @Override
    public String toString()
    {
        return isNull() ? "null" : value.toString();
    }

}
```

Gördüğünüz gibi, aynı sınıfı sadece *String* türünü *Date* olarak değiştirerek tekrar yazdık. Bunun iyi bir yöntem olmadığını kabul etmelisiniz. Yalnızca kod tekrarı yapmakla kalmadık, aynı zamanda birbiriyle alakalı olmalarına rağmen sınıflara farklı isimler vermek zorunda kaldık. Artık *String* türü için *NullableString* sınıfını, *Date* türü için *NullableDate* sınıfını kullanabiliriz.

Yine de hâlâ ideal bir çözüm bulamadık. Çünkü *Nullable* sınıfını yalnızca 2 tür için kullanabiliyoruz. Peki ya bu sınıfı *Integer*, *Double*, *Boolean* gibi diğer türler için de kullanmak istersek? Hepsi için aynı kodu kopyalayıp farklı sınıflar oluşturmamız gerekir.

Yapmak istediğimiz şey, bütün türler için geçerli olacak bir *Nullable* sınıfı yazmak. Bunu şu şekilde başarabiliriz: *Nullable* sınıfının *Object* türü üzerinde çalışmasını sağlayalım. Bildiğiniz gibi, *Object* sınıfı bütün sınıfların atasıdır. Dolayısıyla bütün türleri *Object* türünden ifade edebiliriz. Şimdi sınıfı düzenleyip tekrar yazalım:

```
public class Nullable
{

    private final Object value;

    public Nullable(Object value)
    {
        this.value = value;
    }

    public Object getValue()
    {
        return value;
    }

    public boolean isNull()
    {
        return value == null;
    }

    @Override
    public String toString()
    {
        return isNull() ? "null" : value.toString();
    }

}
```

Şimdi bu sınıfı farklı türler üzerinde kullanalım:

```
Nullable nullableString = new Nullable("abc");
Nullable nullableDate = new Nullable(new Date());
Nullable nullableInt = new Nullable(2020);
Nullable nullableDouble = new Nullable(3.14);
Nullable nullableBoolean = new Nullable(true);
```

Yukarıda görebileceğiniz gibi, *Nullable* sınıfını farklı türler üzerinde kullanabiliriz. Fakat hâlâ bir sorunumuz var: *getValue()* metodunu çağırdığımız zaman çıkan değeri dönüştürmek zorundayız:

```
Nullable nullableString = new Nullable("abc");
if (!nullableString.isNull())
{
      String value = (String) nullableString.getValue();
}
```

Bu önemli bir açıktır. Bu açık yüzünden farkında olmadan hataya sebebiyet verebiliriz. Örneğin, aşağıdaki kodu inceleyelim:

```
Nullable nullableString = new Nullable("abc");
if (!nullableString.isNull())
{
      boolean value = (boolean) nullableString.getValue();
}
```

Bu örnekte *Nullable* sınıfına parametre olarak verdiğimiz değer *String* iken, bu değeri **boolean** türüne dönüştürmeye çalışıyoruz. Bu durumda yukarıda da gördüğümüz *ClassCastException* hatasıyla karşılaşırız.

Bu örnekten şunu anlıyoruz: *Nullable* sınıfını bütün türleri kapsayacak şekilde geliştirmemize rağmen tür güvenliğini sağlayamadık. JDK 5'ten önce bu gibi durumlar sıkça yaşanıyordu ve tür güvensiz kodlar yazılıyordu. Jenerikler ile bunun önüne geçmek mümkün olmuştur. Jenerikler, bir yandan farklı türler için tek bir kod yazmamızı sağlarken, diğer yandan tür güvenliğini sağlar.

Jenerik sınıf yazmak

Jeneriklerin parametrelendirilmiş tür olduğunu söylemiştik. Jenerik bir sınıf yazarken sınıf adından sonra küçüktür ve büyüktür işaretleri arasında bir tür parametresi belirtilir. Bu parametreyi Java'nın değişken isimlendirme kurallarına uygun olarak adlandırabilirsiniz; yine de geleneksel olarak tek ve büyük bir harf verilir. Şimdi *Nullable* sınıfını jenerikleri kullanarak tekrar yazalım:

```
public class Nullable<T>
{

    private final T value;

    public Nullable(T value)
    {
        this.value = value;
    }

    public T getValue()
    {
        return value;
    }

    public boolean isNull()
    {
        return value == null;
    }

    @Override
    public String toString()
    {
        return isNull() ? "null" : value.toString();
    }

}
```

Gördüğünüz gibi, sınıfımızı jenerik bir hale getirdik. T adında bir tür parametresi aldık ve gerekli yerleri bu türe göre yeniden düzenledik. Öncelikle, artık sınıfımızın içinde tuttuğumuz veri T türünde olacaktır. Aynı şekilde, *getValue()* metodunu da T türünde bir değer döndürecek şekilde düzenledik.

Nasıl ki bir metodu çağırırken metodun parametrelerini vermek zorundaysak, jenerik sınıfları kullanırken de parametre olarak bir tür vermek zorundayız. Şimdi bu sınıfı kullanalım:

```
Nullable<String> nullableString = new Nullable<String>("abc");
if (!nullableString.isNull())
{
    String value = nullableString.getValue();
}
```

Bu örnekte *Nullable* sınıfının bir örneğini aldık, fakat bu örneği *String* türü için oluşturduk. Artık *nullableString* nesnesindeki metotları yalnızca *String*

türü için kullanabiliriz. Başka bir tür için kullanmaya çalıştığımızda kodumuz derlenmeyecektir:

```
Nullable<String> nullableString = new Nullable<String>("abc");
if (!nullableString.isNull())
{
    int value = nullableString.getValue();    // Bu satır hata fırlatır
}
```

Örneğin yukarıdaki kod hataya sebep olur; çünkü *nullableString* nesnesinin *getValue()* metodu *String* döndürür. Fakat biz bu metodun sonucunu **int** türünde bir değişkene aktarmaya çalışıyoruz.

Şimdi başka bir örneğe bakalım:

```
Nullable<Date> nullableDate = new Nullable<Date>(new Date());
if (!nullableDate.isNull())
{
    Date value = nullableDate.getValue();
}
```

Yukarıdaki örnekte *Nullable* sınıfını *Date* türü için oluşturduk, dolayısıyla *getValue()* metodu *Date* türünde bir değer döndürmektedir.

Gördüğünüz gibi, jenerikler sayesinde **tür güvenliğini** (*type-safety*) sağlamış olduk.

Jenerikler ilkel veri türleri üzerinde çalışmaz

Jenerik sınıflara veya metotlara ilkel veri türlerini parametre olarak veremezsiniz:

```
Nullable<int> nullable = new Nullable<int>(2020);
// Yukarıdaki satır hataya neden olur
```

Yukarıdaki örnek hataya sebep olur; çünkü tür parametresi olarak ilkel bir veri türü olan int'i veriyoruz.

İlkel veri türlerini de jeneriklerle birlikte kullanabilmek için JDK 5 ile **sarmalayıcı sınıflar** (*wrapper classes*) eklenmiştir. 8 ilkel veri türüne karşılık olarak 8 sarmalayıcı sınıf oluşturulmuştur:

İlkel veri türü	Sarmalayıcı sınıfı
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

Sarmalayıcı sınıfları ilkel veri türlerini sarmalamak için aşağıdaki gibi kullanırız:

```
byte primitiveByte = 1;
Byte byteObject = new Byte(primitiveByte);
byte byteValue = byteObject.byteValue();

short primitiveShort = 1;
Short shortObject = new Short(primitiveShort);
short shortValue = shortObject.shortValue();

int primitiveInt = 1;
Integer intObject = new Integer(primitiveInt);
int intValue = intObject.intValue();

long primitiveLong = 1L;
Long longObject = new Long(primitiveLong);
long longValue = longObject.longValue();

float primitiveFloat = 1.0f;
Float floatObject = new Float(primitiveFloat);
float floatValue = floatObject.floatValue();

double primitiveDouble = 1.0d;
Double doubleObject = new Double(primitiveDouble);
double doubleValue = doubleObject.doubleValue();

char primitiveChar = 'a';
Character charObject = new Character(primitiveChar);
char charValue = charObject.charValue();

boolean primitiveBoolean = true;
Boolean booleanObject = new Boolean(primitiveBoolean);
boolean booleanValue = booleanObject.booleanValue();
```

Kutulama (autoboxing) ve kutudan çıkarma (unboxing)

Yukarıda ilkel veri türleri ve sarmalayıcı sınıfları arasında nasıl dönüşüm yapılabileceğini gördük. Bunu daha kısa bir şekilde yapabilmek için JDK 5 ile **kutulama** ve **kutudan çıkarma** özelliği getirilmiştir. Bu sayede ilkel veri türlerini sarmalayıcı sınıflarına hiçbir ek işlem yapmadan atayabilirsiniz. Yukarıdaki kodu aşağıdaki gibi yazabiliriz:

```
byte primitiveByte = 1;
Byte byteObject = primitiveByte;
byte byteValue = byteObject;

short primitiveShort = 1;
Short shortObject = primitiveShort;
short shortValue = shortObject;

int primitiveInt = 1;
Integer intObject = primitiveInt;
int intValue = intObject;

long primitiveLong = 1L;
Long longObject = primitiveLong;
long longValue = longObject;

float primitiveFloat = 1.0f;
Float floatObject = primitiveFloat;
float floatValue = floatObject;

double primitiveDouble = 1.0d;
Double doubleObject = primitiveDouble;
double doubleValue = doubleObject;

char primitiveChar = 'a';
Character charObject = primitiveChar;
char charValue = charObject;

boolean primitiveBoolean = true;
Boolean booleanObject = primitiveBoolean;
boolean booleanValue = booleanObject;
```

Sınırlandırılmış türler (bounded types)

Jenerik sınıf veya metot tanımlarken parametre olarak aldığınız türü sınırlandırabilirsiniz. Örneğin, parametre olarak aldığım tür *Number* sınıfının alt sınıflarından biri olsun, diyebilirsiniz. Böyle bir durumda *Number* sınıfından türetilmemiş hiçbir sınıfı parametre olarak veremezsiniz.

Tür sınırlandırması yaparken **extends** veya **super** deyimi kullanılır. Bu deyimlerden sonra bir sınıf veya arayüz ismi verilir. Bu deyimleri kullanarak parametre olarak jenerik türünü sınırlandırmış oluruz:

- **extends** deyimini kullandıysak yalnızca belirttiğimiz türü veya alt sınıflarını kullanabiliriz. Buna **üst sınır** (*upper-bound*) denir.
- **super** deyimini kullandıysak yalnızca belirttiğimiz türün üst sınıflarını kullanabiliriz. Buna **alt sınır** (*lower-bound*) denir.

Şimdi yukarıda yazdığımız *Nullable* sınıfını güncelleyerek bir örnek yapalım:

```
public class Nullable<T extends Number>
{

    private final T value;

    public Nullable(T value)
    {
        this.value = value;
    }

    public T getValue()
    {
        return value;
    }

    public boolean isNull()
    {
        return value == null;
    }

    @Override
    public String toString()
    {
        return isNull() ? "null" : value.toString();
    }

}
```

Yukarıda da gördüğünüz gibi, jenerik *Nullable* sınıfına aldığımız tür parametresini **extends** deyimiyle sınırlandırdık. Buna göre bu sınıfı yalnızca *Number* veya alt türleriyle kullanabiliriz.

```
Nullable<Integer> nullableInteger = new Nullable<Integer>(2020);
// Bu kullanım uygundur; çünkü Integer, Number sınıfından türetilmiştir

Nullable<String> nullableString = new Nullable<String>("2020");
// Bu kullanım uygun değildir; String, Number sınıfından türetilmemiştir
```

Joker argümanını (wildcard argument) kullanmak

Jenerikler tür güvenliğini mükemmel bir biçimde sağlar; buna rağmen bazen daha genel düşünmek gerekebilir. Örneğin, yukarıda oluşturduğumuz *Nullable* sınıfıyla ilgili bir metot yazdığımızı düşünelim. Bu metot *Nullable* türünde herhangi bir nesneyi parametre olarak alacak ve içindeki değerin **null** olup olmadığını test edecek. *Nullable* sınıfı jenerik bir sınıf olduğu için, parametre olarak tanımlarken bir tür belirtmek gerekir. Örneğin aşağıda bu metodun bir örneğini görebilirsiniz:

```
public static boolean isNullableHasValue(Nullable<String> nullable)
{
    return nullable != null && !nullable.isNull();
}
```

Bu metot sorunsuz bir şekilde çalışır; fakat yalnızca *String* türündeki *Nullable* nesnelerini kabul eder. Takdir edersiniz ki, biz buraya genel bir metot yazmak istiyoruz. Dolayısıyla yalnızca *String* değil, bütün türleri kapsayan bir *Nullable* nesnesini parametre olarak almak isteriz.

Bu gibi durumlarda **joker argümanını** (*wildcard argument*) kullanırız. Joker argümanını soru işareti (?) ile yazarız. Bu argüman tür parametresi yerine geçer ve bütün türleri temsil eder. Şimdi yukarıdaki metodu joker argümanı kullanarak tekrar yazalım:

```
public static boolean isNullableHasValue(Nullable<?> nullable)
{
    return nullable != null && !nullable.isNull();
}
```

Artık bu metodu bütün *Nullable* nesneleri için kullanabiliriz.

Sınırlandırılmış joker argümanı

Joker argümanına da alt veya üst sınır verebilirsiniz. Örneğin yukarıdaki metodu aşağıdaki gibi yazsaydık yalnızca *Number* sınıfının alt türleri için *Nullable* nesnelerini parametre olarak alabilecektik:

```
public static boolean isNullableHasValue(Nullable<? extends Number>
      nullable)
{
    return nullable != null && !nullable.isNull();
}
```

JENERİK METOTLAR

Bir metot yazarken, içinde bulunduğu sınıf jenerik olmasa metodu jenerik hale getirebilirsiniz. Bunun için metodun dönüş türünden önce tür parametresini belirtmeniz yeterlidir. Örneğin *ArrayUtil* adında bir sınıf yazalım. Bu sınıfın içinde *arrayContains()* adında bir metodumuz olsun. Bu metot herhangi bir elemanın dizi içinde olup olmadığını test etsin.

```
public class ArrayUtil
{

    public <T> boolean arrayContains(T[] array, T elem)
    {
        for (T item : array)
        {
            if (item != null && item.equals(elem))
            {
                return true;
            }
        }

        return false;
    }

}
```

Gördüğünüz gibi, sınıf jenerik olmasa bile metodumuzu jenerik hale getirdik. Artık bu metodu herhangi bir türdeki diziler için kullanabiliriz.

Jenerik sınıf yazarken geçerli olan özellikleri jenerik metotlarda da kullanabilirsiniz. Örneğin, jenerik türü sınırlandırabilir veya joker parametresi kullanabilirsiniz. Ayrıca, jenerik yapılandırıcılar da oluşturabilirsiniz.

Diamond operatörünün kullanılması

JDK 7 ile diamond operatörünü kullanabiliriz. Bu operatörü kullanarak Java derleyicisinin jenerik türü otomatik olarak tespit etmesini sağlarız. Örneğin, aşağıdaki kodu inceleyelim:

```
Nullable<Integer> nullable = new Nullable<Integer>();
```

Burada öncelikle *Nullable<Integer>* türünde bir değişken oluşturuyoruz. Daha sonra bu değişkene yine bu türde bir atama yapıyoruz. Dikkat ederseniz, *Nullable* nesnesini oluştururken de parametre olarak *Integer* yazdık. Fakat JDK 7 ile artık bunu belirtmemize gerek yoktur; çünkü Java derleyicisi değişkene bakarak türün ne olduğunu anlayabilir. Dolayısıyla, yukarıdaki kodu daha kısa bir şekilde yazabiliriz:

```
Nullable<Integer> nullable = new Nullable<>();
```

Gördüğünüz gibi, nesneyi oluştururken tür parametresini boş bıraktık. Bu operatöre (<>) **diamond operatörü** denir.

Jenerik kısıtlamaları

Jeneriklerle ilgili bası kısıtlamalar mevcuttur. Bunlardan kısaca bahsedelim.

Jenerik türlerin bir örneği alınamaz

Tür parametreleri, türün derleme aşamasında değil çalışma zamanında belirlenmesi için kullanılır. Dolayısıyla kodun yazım aşamasında türün ne olacağını önceden kestirmek mümkün değildir. Bu nedenle tür parametresi kullanarak yeni bir nesne oluşturamazsınız:

```
public class GenericClass<T>
{

	private T obj;

	public GenericClass()
	{
		obj = new T();	// Bu satır hataya sebep olur
	}

}
```

Jenerik sınıfların statik üyeleri tür parametresine erişemez

Statik üyelerin sınıfların oluşmasından bağımsız olduğunu daha önce anlatmıştık. Dolayısıyla jenerik bir sınıf içindeki statik üyeler tür parametresine erişemez:

```
public class GenericClass<T>
{

      public static createGenericClass(T param)
      {
            // Yukarıdaki parametre nedeniyle bu kod derlenmez
            // Statik metot içinde T parametresine erişemeyiz
      }

}
```

Jenerik dizi oluşturamazsınız

Dizi oluşturduğunuz zaman hafızada dizinin boyutu kadar yer ayrılır. Jenerik bir türün ne olduğu önceden bilinemeyeceği için jenerik dizi oluşturamazsınız:

```
public class GenericClass<T>
{

      public GenericClass()
      {
            T[] array = new T[];
            // Bu satır hataya sebep olur; jenerik dizi oluşturamazsınız
      }

}
```

Jenerik *Exception* sınıfı oluşturamazsınız

Kendinize özel hata sınıfları oluşturabilirsiniz. Fakat bu hata sınıfları jenerik olamaz.

```
public class MyException<T> extends Exception
{
}
```

Yukarıdaki sınıf henüz derleme aşamasında hata alır; çünkü jenerik bir hata sınıfı oluşturmaya çalışıyoruz.

KOLEKSİYON SINIFLARI

Aynı türden birden fazla nesneyi tek bir değişkende tutmak için dizileri kullandığımızı daha önce anlatmıştık. Fakat diziler dile entegre halde bulunan ilkel yapılardır. Diziler üzerinde işlem yapmak için genellikle çok fazla kod yazmak gerekir. Ayrıca diziler üzerinde yapılan işlemlerde hata olasılığı yüksektir.

Aynı türde birçok nesneyi tutmak için diziler yerine, daha gelişmiş ve esnek yapıda bulunan koleksiyon sınıflarını kullanırız. JDK 5 ile gelen jenerik özelliğiyle birlikte koleksiyon sınıfları da jenerik hale getirilmiş, yeni koleksiyon sınıfları oluşturulmuş ve önceki koleksiyon sınıflarının kullanımı büyük ölçüde azalmıştır.

Bu bölümde en çok kullanılan koleksiyon sınıflarını ve arayüzleri inceleyeceğiz.

Collection arayüzü

Java'nın koleksiyon sınıfları hiyerarşisinde en üstte bu arayüz bulunur. Bütün koleksiyon sınıfları bu arayüzü uygular. Bu arayüz, koleksiyon sınıflarının hepsinde kullanacağımız genel metotları belirler. Bu arayüzün sahip olduğu bazı metotları aşağıda inceleyelim:

interface Collection<**E**>	
int size()	Koleksiyonun içindeki eleman sayısını döndürür.
boolean isEmpty()	Koleksiyonun boş olup olmadığı bilgisini döndürür.
boolean contains(**Object** o)	Parametre olarak aldığı elemanın koleksiyonda olup olmadığı bilgisini döndürür.

boolean add(**E** item)	Parametre olarak aldığı elemanı koleksiyona ekler.
boolean remove(**Object** o)	Parametre olarak aldığı elemanı koleksiyonda arar, varsa siler.
void clear()	Koleksiyondaki bütün elemanları siler.
Object[] toArray()	Koleksiyondaki elemanları içeren bir dizi döndürür.

List arayüzü

Listeleri tanımlamak için *List* arayüzü kullanılır. Listeler dizinin gelişmiş hali olarak düşünülebilir. Nesneler liste içinde arka arkaya dizilirler ve her bir nesneye sıra numarasıyla erişilir. Sıra numaraları (*index*) 0'dan başlar. Bir listenin aralığı dışında bir sıra numarasıyla işlem yapmak istendiği zaman *IndexOutOfBoundsException* hatası fırlatılır.

List arayüzü, *Collection* arayüzünü miras alır. Dolayısıyla *Collection* arayüzündeki bütün metotlar *List* arayüzünde de vardır.

List arayüzünün bazı metotlarını inceleyelim:

interface List<**E**>	
E get(**int** index)	Listenin belirtilen sırasındaki elemanı döndürür. Burada parametre olarak belirtilen sıra numarası 0 veya 0'dan büyük ve listenin eleman sayısından küçük olmalıdır.
E set(**int** index, **E** item)	Parametre olarak verilen indeksteki elemanı ikinci argümandaki elemanla değiştirir. Değiştirme işleminden önceki değeri döndürür.
void add(**int** index, **E** item)	Parametre olarak verilen elemanı listenin belirtilen sırasına ekler. Bu işlem araya sokma işlemidir. Eğer eklenen sırada ve devamında başka elemanlar varsa bu elemanlar birer sıra ileriye kaydırılır.
E remove(**int** index)	Belirtilen indekste bulunan elemanı siler.

int indexOf(**Object** o)	Parametre olarak belirtilen elemanı listede arar, bulursa sıra numarasını döndürür. Eleman listede bulunamazsa -1 döndürür.
int lastIndexOf(**Object** o)	Parametre olarak belirtilen elemanı listede arar, bulursa sıra numarasını döndürür. Yukarıdaki metottan farkı şudur: bir eleman bir liste içinde birden fazla kez bulunabilir. Bu durumda bu metot son sıra numarasını döndürür. Eleman listede bulunamazsa -1 döndürür.
List<E> subList(**int** from, **int** to)	Parametre olarak iki indeks değeri verilir. Listenin bu indeksler arasında bulunan elemanlarını alır ve yeni bir liste olarak döndürür.

Set arayüzü

Kümeleri tanımlamak için *Set* arayüzü kullanılır. Kümeler listelere çok benzer; fakat küme içinde bir eleman birden fazla kez bulunamaz. Ayrıca kümelere **null** eleman eklenemez.

Set arayüzü, *Collection* arayüzünü miras alır. Dolayısıyla *Collection* arayüzündeki bütün metotlar *Set* arayüzünde de vardır.

SortedSet arayüzü

İçindeki elemanları otomatik olarak küçükten büyüğe sıralayan kümeler *SortedSet* arayüzünü uygular. *Set* arayüzünden miras alır. Buna ek olarak iki önemli metot tanımlar:

interface SortedSet<**E**>	
E first()	Kümenin başındaki (en küçük) elemanı döndürür.
E last()	Kümenin sonundaki (en büyük) elemanı döndürür.

Queue arayüzü

Kuyruk tanımlamak için *Queue* arayüzü uygulanır. Kuyruklar listelere ve kümelere benzer; fakat özel yapıları vardır. Bu arayüzü uygulayan kuyruklar genelde **ilk-giren-ilk-çıkar** (*FIFO – first in first out*) mantığını kullanır. Yani, kuyruktan ilk çıkaracağınız eleman kuyruğa ilk giren elemandır.

Queue arayüzü, *Collection* arayüzünü miras alır. Dolayısıyla *Collection* arayüzündeki bütün metotlar *Queue* arayüzünde de vardır.

Queue arayüzünün bazı metotlarını inceleyelim:

interface Queue<**E**>	
E element()	Kuyruğun en üstündeki elemanı döndürür; fakat silmez. Bu eleman kuyruğa ilk eklenen elemandır. Bu sıralama eklenme sırasına göre belirlenir. Kuyruk boşsa *NoSuchElementException* hatası fırlatılır.
boolean offer(**E** item)	Belirtilen elemanı kuyruğa eklemeye çalışır; eklerse **true**, aksi halde **false** döndürür.
E peek()	Kuyruğun en üstündeki elemanı döndürür; fakat silmez. Kuyruk boşsa **null** döndürür.
E poll()	Kuyruğun en üstündeki elemanı döndürür ve siler. Kuyruk boşsa **null** döndürür.
E remove()	Kuyruğun en üstündeki elemanı döndürür ve siler. Kuyruk boşsa *NoSuchElementException* hatası fırlatılır.

Deque arayüzü

Deque arayüzü *Queue* arayüzünden miras alır ve kuyruk tanımlamak için kullanılır. Bu arayüzü uygulayan sınıflar, *Queue* arayüzünden farklı olarak **son-giren-ilk-çıkar** (*LIFO – last in first out*) mantığını da uygulayabilir. Yani, kuyruktan ilk çıkaracağınız eleman kuyruğa son eklenen elemandır.

Deque arayüzünün bazı metotlarını inceleyelim:

interface Deque<**E**>	
void addFirst(E item)	Belirtilen elemanı kuyruğun başına ekler.
void addLast(E item)	Belirtilen elemanı kuyruğun sonuna ekler.
E getFirst()	Kuyruğun başındaki elemanı döndürür; fakat silmez. Kuyruk boşsa *NoSuchElementException* hatası fırlatılır.
E getLast()	Kuyruğun sonundaki elemanı döndürür; fakat silmez. Kuyruk boşsa *NoSuchElementException* hatası fırlatılır.
void push(E item)	Belirtilen elemanı kuyruğun en üstüne ekler.
E pop()	Kuyruğun en üstündeki elemanı döndürür ve siler. Kuyruk boşsa *NoSuchElementException* hatası fırlatılır.
E peekFirst()	Kuyruğun başındaki elemanı döndürür; fakat silmez. Kuyruk boşsa **null** döndürür.
E peekLast()	Kuyruğun sonundaki elemanı döndürür; fakat silmez. Kuyruk boşsa **null** döndürür.
E pollFirst()	Kuyruğun başındaki elemanı döndürür ve siler. Kuyruk boşsa **null** döndürür.
E pollLast()	Kuyruğun sonundaki elemanı döndürür ve siler. Kuyruk boşsa **null** döndürür.

SIK KULLANILAN BAZI KOLEKSİYON SINIFLARI

Koleksiyon sınıflarını oluşturan temel arayüzleri yukarıda inceledik. Şimdi çok sık kullanacağımız bazı koleksiyon sınıflarını inceleyelim.

ArrayList

AbstractList sınıfının alt sınıfıdır ve *List* arayüzünü uygular. Bu sınıfı kullanarak gerektiğinde büyüyebilen dinamik diziler oluşturabiliriz. Daha önce gördüğümüz gibi, Java'da dizi oluştururken sabit bir boyut vermek

gerekir ve bu boyut daha sonra değiştirilemez. *ArrayList* sınıfı kendi içinde dizileri kullanır; fakat gerektiğinde bu diziyi büyütür ve yenisiyle değiştirir. Aynı zamanda dizinin arasına eleman eklediğimiz zaman diğer elemanları gerektiği gibi kaydırır. Bu gibi işlemleri kolaylıkla yapmamızı sağladığı için *ArrayList* sınıfı Java'da çokça kullanılır.

Aşağıdaki örneği inceleyelim:

```
ArrayList<String> list = new ArrayList<>();

list.add("A");
list.add("B");
list.add("C");
list.add("D");
list.add("E");
list.add("F");

System.out.println("İlk 6 elemanı ekledik:");
System.out.println(list + "\n");

list.add(3, "G");
list.add(4, "H");

System.out.println("Araya 2 eleman ekledik:");
System.out.println(list + "\n");

list.set(0, "J");
System.out.println("İlk elemanı değiştirdik:");
System.out.println(list + "\n");

list.set(7, "K");
System.out.println("Son elemanı değiştirdik:");
System.out.println(list + "\n");

System.out.println("Listenin içinde B var mı: ");
System.out.println(list.contains("B") + "\n");

list.remove("B");
System.out.println("B'yi sildik:");
System.out.println(list + "\n");

System.out.println("Listenin içinde B var mı: ");
System.out.println(list.contains("B") + "\n");

System.out.println("E kaçıncı sırada: ");
System.out.println(list.indexOf("E") + "\n");

System.out.println("B kaçıncı sırada: ");
System.out.println(list.indexOf("B") + "\n");

System.out.println("4. indekste hangi eleman var: ");
System.out.println(list.get(4) + "\n");

list.remove(4);
System.out.println("4. indeksteki elemanı sildik: ");
System.out.println(list + "\n");

System.out.println("Listede kaç eleman var: ");
System.out.println(list.size() + "\n");

list.clear();
System.out.println("Listeyi temizledik: ");
System.out.println(list + "\n");
```

Burada *ArrayList* sınıfının en çok kullanılan metotlarını vermeye çalıştık. Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
İlk 6 elemanı ekledik:
[A, B, C, D, E, F]

Araya 2 eleman ekledik:
[A, B, C, G, H, D, E, F]

İlk elemanı değiştirdik:
[J, B, C, G, H, D, E, F]

Son elemanı değiştirdik:
[J, B, C, G, H, D, E, K]

Listenin içinde B var mı:
true

B'yi sildik:
[J, C, G, H, D, E, K]

Listenin içinde B var mı:
false

E kaçıncı sırada:
5

B kaçıncı sırada:
-1

4. indekste hangi eleman var:
D

4. indeksteki elemanı sildik:
[J, C, G, H, E, K]

Listede kaç eleman var:
6

Listeyi temizledik:
[]
```

LinkedList

AbstractSequentialList sınıfının alt sınıfıdır ve *List*, *Deque* ve *Queue* arayüzlerini uygular. Tıpkı *ArrayList* gibi liste belirtir; fakat elemanları dizi şeklinde tutmaz. Aşağıdaki örneği inceleyelim:

```
LinkedList<String> list = new LinkedList<>();

list.add("A");
list.add("B");
list.add("C");
list.add("D");
list.add("E");
list.add("F");

System.out.println("İlk 6 elemanı ekledik:");
System.out.println(list + "\n");

System.out.println("Listenin başında hangi eleman var: ");
System.out.println(list.peekFirst() + "\n");

System.out.println("Listenin sonunda hangi eleman var: ");
System.out.println(list.peekLast() + "\n");

list.pollFirst();
System.out.println("Listenin başındaki elemanı sildik: ");
System.out.println(list + "\n");

list.pollLast();
System.out.println("Listenin sonundaki elemanı sildik: ");
System.out.println(list + "\n");

list.addFirst("G");
System.out.println("Listenin başına G ekledik:");
System.out.println(list + "\n");

list.addLast("H");
System.out.println("Listenin başına H ekledik:");
System.out.println(list + "\n");

list.clear();
System.out.println("Listeyi temizledik: ");
System.out.println(list + "\n");
```

Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
İlk 6 elemanı ekledik:
[A, B, C, D, E, F]

Listenin başında hangi eleman var:
A

Listenin sonunda hangi eleman var:
F

Listenin başındaki elemanı sildik:
[B, C, D, E, F]

Listenin sonundaki elemanı sildik:
[B, C, D, E]

Listenin başına G ekledik:
[G, B, C, D, E]

Listenin başına H ekledik:
[G, B, C, D, E, H]

G
Listeyi temizledik:
[]
```

TreeSet

AbstractSet sınıfının alt sınıfıdır ve *SortedSet* arayüzünü uygular. Elemanları küme halinde tutar. En önemli özelliği elemanların sıralı olmasıdır. *TreeSet* içindeki elemanları otomatik olarak sıralar. Aşağıdaki örneği inceleyelim:

```
TreeSet<String> set = new TreeSet<>();

set.add("F");
set.add("E");
set.add("D");
set.add("C");
set.add("B");
set.add("A");

System.out.println("Elemanları ekledik: ");
System.out.println(set + "\n");
```

Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Elemanları ekledik:
[A, B, C, D, E, F]
```

Gördüğünüz gibi, elemanları sıralı olarak eklememiş olsak bile *TreeSet* otomatik olarak sıralamıştır.

TreeSet oluştururken parametre olarak bir *Comparator* (sıralayıcı) nesnesi verebilirsiniz. Bu durumda *TreeSet*, elemanları sıralarken bu sıralayıcıyı kullanır. Aşağıdaki örneği inceleyelim:

```
TreeSet<String> set = new TreeSet<>(Comparator.reverseOrder());

set.add("A");
set.add("B");
set.add("C");
set.add("D");
set.add("E");
set.add("F");

System.out.println("Elemanları ekledik: ");
System.out.println(set + "\n");
```

Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Elemanları ekledik:
[F, E, D, C, B, A]
```

Comparator.reverseOrder() metodu, elemanları büyükten küçüğe doğru sıralayan bir sıralayıcı döndürür. Bu sıralayıcıyı kullandığımız için elemanlar büyükten küçüğe doğru sıralanmıştır.

HashSet

AbstractSet sınıfının alt sınıfıdır ve *Set* arayüzünü uygular. Elemanları hash tablosu şeklinde tutar. *HashSet* ile ilgili dikkat etmemiz gereken husus şudur: elemanların sırası önceden tahmin edilemez. Aşağıdaki örneği inceleyelim:

```
HashSet<String> set = new HashSet<>();

set.add("Alpha");
set.add("Beta");
set.add("Epsilon");
set.add("Eta");
set.add("Gamma");
set.add("Omega");

System.out.println("Elemanları ekledik: ");
System.out.println(set + "\n");
```

Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Elemanları ekledik:
[Gamma, Eta, Alpha, Epsilon, Omega, Beta]
```

Gördüğünüz gibi, *HashSet* sınıfının yapısı gereği, eklenen elemanların sırası beklediğimizden farklıdır.

Stack

Yığın oluşturmak için *Stack* sınıfı kullanılır. Son-giren-ilk-çıkar mantığını uygular, dolayısıyla bu gibi durumlarda sıkça kullanılır. Aşağıdaki örneği inceleyelim:

```
Stack<String> stack = new Stack<>();

stack.push("A");
stack.push("B");
stack.push("C");
stack.push("D");
stack.push("E");
stack.push("F");

System.out.println("Elemanları çıkarıyoruz: ");
while (!stack.isEmpty())
{
    String item = stack.pop();
    System.out.println(item);
}
```

Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Elemanları çıkarıyoruz:
F
E
D
C
B
A
```

Gördüğünüz gibi, yığından ilk çıkardığımız eleman yığına en son eklediğimiz elemandır.

Iterator arayüzü

Java'da bir koleksiyonun elemanlarını tek tek gezmek için *Iterator* arayüzü oluşturulmuştur. Bu arayüzün metotlarını aşağıdaki tabloda inceleyelim:

interface Iterator<**E**>	
boolean hasNext()	Koleksiyonun sonuna gelip gelinmediğini belirtir.
E next()	Sıradaki elemanı döndürür.
void remove()	Eğer arayüzü uygulayan *Iterator* nesnesi destekliyorsa, şu anki elemanı koleksiyondan siler.

Collection arayüzünde tanımlanmış *iterator()* metodu vardır. Bu metot koleksiyonla ilgili *Iterator* nesnesini elde etmek için kullanılır. Aşağıdaki örneği inceleyelim:

```
TreeSet<String> set = new TreeSet<>();

set.add("D");
set.add("C");
set.add("A");
set.add("F");
set.add("E");
set.add("B");

System.out.println("Elemanları ekledik: ");

Iterator<String> setIterator = set.iterator();
while (setIterator.hasNext())
{
    String item = setIterator.next();
    System.out.println(item);
}
```

Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Elemanları ekledik:
A
B
C
D
E
F
```

Iterable arayüzü

Elemanları üzerinde gezilebilecek bir nesne belirtmek için *Iterable* arayüzü kullanılır.

interface Iterable<**E**>	
Iterator<**E**> iterator()	Iterator nesnesi döndürür.

Collection arayüzü *Iterable* arayüzünden miras alır, dolayısıyla bütün koleksiyon sınıflarında *iterator()* metodu bulunur.

Daha önce **for-each** döngüsünü görmüştük. Java'da bir nesnenin üzerinde for-each döngüsüyle gezebilmek için o nesnenin *Iterable* arayüzünü uygulaması gerekir. Diğer bir deyişle, bütün koleksiyonların elemanlarını for-each döngüsüyle gezebilirsiniz.

Örneğin, yukarıdaki kodu değiştirip daha kısa bir şekilde aşağıdaki gibi yazabiliriz:

```
TreeSet<String> set = new TreeSet<>();

set.add("D");
set.add("C");
set.add("A");
set.add("F");
set.add("E");
set.add("B");

System.out.println("Elemanları ekledik: ");
for (String item : set)
{
    System.out.println(item);
}
```

Bu kodun çıktısı bir öncekiyle aynı olacaktır. Burada anlatmak istediğimiz nokta şudur: for-each döngüsüyle yazılan kodları Java derleyicisi arka planda *Iterator* kodlarına dönüştürür.

Map sınıfları

Nesneleri **anahtar-değer** ikilileri (*key-value pairs*) halinde tutmak için *Map* arayüzü tanımlanmıştır.

interface Map<**K**, **V**>	
boolean containsKey(**Object** key)	Belirtilen anahtarın koleksiyonda olup olmadığı bilgisini döndürür.
boolean containsValue(**Object** value)	Belirtilen değerin koleksiyonda olup olmadığı bilgisini döndürür.
V get(**K** key)	Belirtilen anahtarla ilişkilendirilmiş değeri döndürür. Böyle bir anahtar-değer ikilisi yoksa **null** döndürür.
V put(**K** key, **V** value)	Belirtilen değeri, belirtilen anahtarla ilişkilendirerek koleksiyona ekler.
V remove(**K** key)	Belirtilen anahtarla ilişkilendirilmiş değeri koleksiyondan siler.
Set<**K**> keySet()	Koleksiyondaki bütün anahtarları küme halinde döndürür.

Collection<V> values()	Koleksiyondaki bütün değerleri döndürür.
Set<Map.Entry<K,V>> entrySet()	Koleksiyondaki bütün girdileri küme halinde döndürür.

TreeMap

Nesneleri anahtar-değer ikilileri halinde tutar. Girdiler anahtarlara göre otomatik olarak sıralanır. Örneğin, aşağıdaki örnekte kişilerin yaşlarını isimleriyle beraber tutmak için TreeMap kullandık:

```
TreeMap<String, Integer> map = new TreeMap<>();

map.put("Veli", 35);
map.put("Mehmet", 30);
map.put("Ahmet", 25);

System.out.println("Elemanları ekledik: ");
System.out.println(map + "\n");

System.out.println("Ahmet'in yaşı kaç: ");
System.out.println(map.get("Ahmet") + "\n");

System.out.println("Mehmet'in yaşı kaç: ");
System.out.println(map.get("Mehmet") + "\n");

System.out.println("Mustafa'nın yaşı kaç: ");
System.out.println(map.get("Mustafa") + "\n");

map.remove("Veli");
System.out.println("Veli'nin yaşını sildik: ");
System.out.println(map + "\n");
```

Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Elemanları ekledik:
{Ahmet=25, Mehmet=30, Veli=35}

Ahmet'in yaşı kaç:
25

Mehmet'in yaşı kaç:
30

Mustafa'nın yaşı kaç:
null

Veli'nin yaşını sildik:
{Ahmet=25, Mehmet=30}
```

Dikkat etmişsinizdir, elemanları büyükten küçüğe doğru eklememize rağmen *TreeMap* otomatik olarak sıralamıştır.

HashMap

TreeMap sınıfıyla benzer özelliklere sahiptir. En önemli farkı, anahtarları hash tablosu halinde tuttuğu için elemanlarının sırasının belirsiz olduğudur. *HashMap* sınıfının elemanları hangi sırayla tutacağını önceden öngöremeyiz. Örneğin, yukarıdaki kodun aynısını *HashMap* için yazalım:

```
HashMap<String, Integer> map = new HashMap<>();

map.put("Veli", 35);
map.put("Mehmet", 30);
map.put("Ahmet", 25);

System.out.println("Elemanları ekledik: ");
System.out.println(map + "\n");

System.out.println("Ahmet'in yaşı kaç: ");
System.out.println(map.get("Ahmet") + "\n");

System.out.println("Mehmet'in yaşı kaç: ");
System.out.println(map.get("Mehmet") + "\n");

System.out.println("Mustafa'nın yaşı kaç: ");
System.out.println(map.get("Mustafa") + "\n");

map.remove("Veli");
System.out.println("Veli'nin yaşını sildik: ");
System.out.println(map + "\n");
```

Yukarıdaki kodu çalıştırdığınızda çıktısı aşağıdaki gibi olur:

```
Elemanları ekledik:
{Ahmet=25, Veli=35, Mehmet=30}

Ahmet'in yaşı kaç:
25

Mehmet'in yaşı kaç:
30

Mustafa'nın yaşı kaç:
null

Veli'nin yaşını sildik:
{Ahmet=25, Mehmet=30}
```

TreeMap ile *HashMap* arasındaki farkın elemanların sırası olduğunu fark etmişsinizdir.

Comparator arayüzü

Daha önce gördüğümüz gibi, bazı koleksiyon sınıfları elemanları kendiliğinden sıralar. Sıralamayı desteklemeyen koleksiyon sınıflarında veya

kendi yazacağınız koleksiyon sınıflarında elemanları sıralamak için bir sıralama algoritması belirtmeniz gerekir. Bunu *Comparator* arayüzüyle yaparız.

interface Comparator<T>	
int compare(**T** obj1, **T** obj2)	Parametre olarak iki nesne alır ve bu nesneleri karşılaştırır. Eğer birinci nesne sıralamada ikinci nesneden önce gelmeliyse -1 döndürür. İkinci nesne birinci nesneden önce gelmeliyse 1 döndürür. Eğer iki nesne birbirine eşitse 0 döndürür.

Şimdi *Comparator* ile ilgili bir örnek yapalım. Öncelikle *Person* adında yeni bir sınıf tanımlayalım. Bu sınıfta kişilerin ismini ve yaşlarını tutmak için iki alanımız olacak:

```
class Person
{

    String name;
    int age;

    public Person(String name, int age)
    {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString()
    {
        return String.format("%s=%d", name, age);
    }

}
```

Person türündeki nesneleri karşılaştırabilmek için *PersonComparator* adında bir sınıf oluşturalım. Bu sınıf *Comparator* arayüzünü uygulayacaktır:

```
class PersonComparator implements Comparator<Person>
{

    @Override
    public int compare(Person o1, Person o2)
    {
        if (o1.age == o2.age)
        {
            return 0;
        }

        return o1.age < o2.age ? -1 : 1;
    }

}
```

Buna göre, *PersonComparator* sınıfı *Person* türündeki nesneleri kişilerin yaşına göre küçükten büyüğe doğru sıralamaktadır.

Şimdi bu sınıfları kullanarak aşağıdaki örneği yapalım:

```
PersonComparator comparator = new PersonComparator();
TreeSet<Person> set = new TreeSet<>(comparator);

set.add(new Person("Ahmet", 40));
set.add(new Person("Mehmet", 35));
set.add(new Person("Veli", 30));

System.out.println("Kişileri kümeye ekledik: ");
System.out.println(set);
```

Yukarıdaki kodu çalıştırırsanız çıktısı aşağıdaki gibi olur:

```
Kişileri kümeye ekledik:
[Veli=30, Mehmet=35, Ahmet=40]
```

Gördüğünüz gibi elemanlar, yazdığımız sıralama algoritmasına göre sıralanmıştır.