Hata Yönetimi

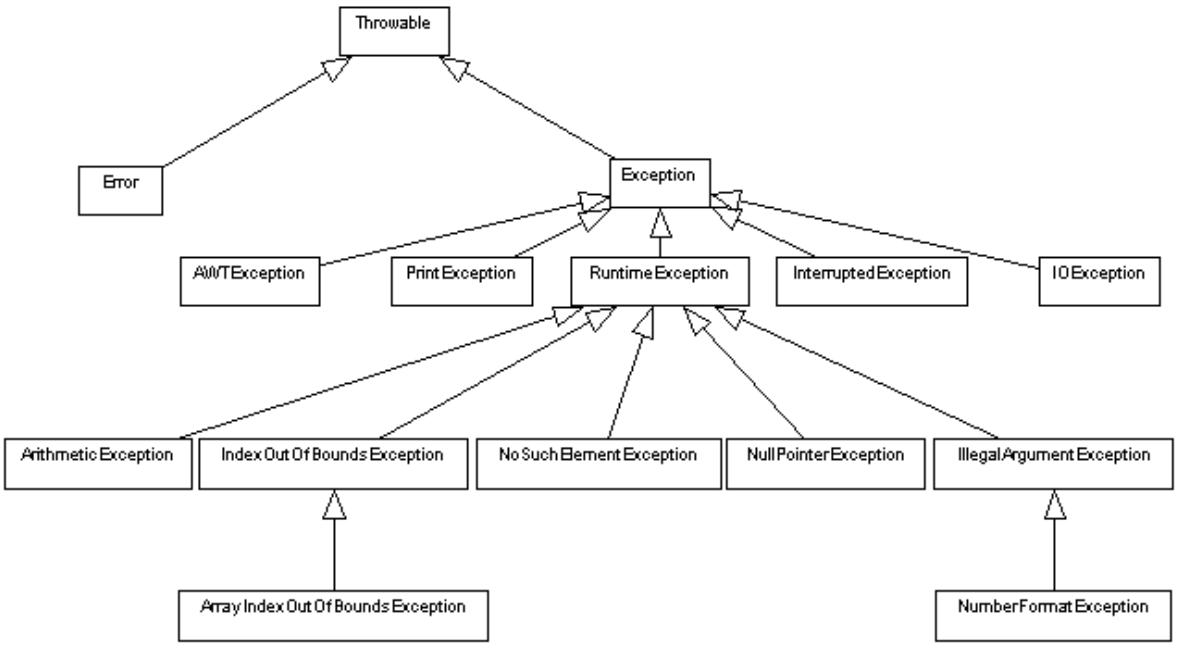
Java dünyasında uygulama geliştirme anında hatalarla karşılaşabiliriz. Bu eğer syntax yani söz dimi hatası değilse bizim yazdığımız bir kod bloğundan bazense kullandığımız bir kütüphaneden kaynaklanıyor olabilir. Bu gibi durumlarda hataya kızmak yerine hatayı doğru yönetmeyi ve hata durumda ne yapacağımızı kontrol etmeliyiz. Bu yaklaşıma hata veya istisna yönetimi yani exception handling denir.

Özellikle dışa bağımlı olarak tasarladığımız uygulamalarda bu durum potansiyel olarak söz konusudur. Örnekleyecek olursak geliştirdiğimiz bir CRM sisteminde bilgiyi bir web servisinden alıp veritabanına kaydediyor olabiliriz. Peki web servise baglanma durumda bir hata olursa ? Örneğin ağ bağlantısı kopması, web servis sunucusunun kapanması gibi durumlarda ne yapılabilir ? Bir başka durumda veritabanı bağlantısı anında gerçekleşebilir ? Veritabanı kapanabilir, kullanici yetkisi alınmış olabilir veya ilgili tablo artık olmayabilir ? İşte tüm bu durumları biz uygulama geliştirici olarak göz önüne alıp hata durumda bir kontrol mekanizması oluşturmalıyız.

Dikkat : Hata yakalama mekanizmalarında derleme anındaki hatalar kontrol edilemez. Örneğin bir syntax yani söz dizimi hatası sizin Java' yı yanlış yazmanızdan dolayı oluşan bir hatadır ve kontrolü mümkün değildir.

Java' da tüm exception tipleri **Throwable** sınıfından türemişlerdir. Bu sınıftan türeyen iki önemli sınıf yani **Error** ve **Exception** sınıfları bulunmaktadır.

Error Java içerisindeki kritik hatalarda fırlatılır. Buradaki kritik hata bizim müdahalemiz dışında Java' nın kendi içerisinde gerçekleşen hata olabilir. Örneğin aşağıdaki hiyerarşide de görebileceğiniz gibi **OutOfMemoryError** tipi **Error** altında **VirtualMachineError** sınıfı altındadır.



Exception sınıfı ise diğer exception tiplerini ifade eder ki bunlar uygulama içerisinde herhangi bir akış anında karşılaşılabilecek hatalardır diyebiliriz.

Exception sınıfı içerisinde yer alan **RuntimeException** bizim en çok karşılaştığımız kontrollü olmayan tipteki hataları ifade eder. Bu tip hatalar dışa bağımlı olmayan ve Java içerisindeki olası hatalardır ve derleme anında herhangi bir kontrol gerektirmez. Bu yüzden **uncheck exception** olarak tanımlanırlar.

Not : **RuntimeException** dışında **Exception** sınıfından türeyen tüm sınıflar **checked** yani kontol gerektiren exceptionlardır.

Hata kontrolü

Java içerisinde kod bloğu içerisinde hata yönetimi için kullanılan yapı try-catch-finally bloklarıdır. Buna göre try içerisinde kod bloğu kontrol edilir. Bir hata durumu oluşursa bu catch bloğu içerisinde yakalanır. finally bloğu ise çalıştırılması kesin olan bloktur.

```
try {
	// Kontrol edilen blok
} catch (Exception e) {
	// Hata oluştuğunda çalışacak blok
} finally {
	// Her durumda çalışacak blok
}
```

Not : Herhangi bir hata oluşmazsa cathc bloğuna uğranmayacaktır.

Şimdi bu durumu örneklendirelim öncelikle aşağıdaki gibi bir kod bloğumuz olsun.

```
public class Ornek {

	public static void main(String[] args) {

		String[] dizi = new String[4];
		dizi[6] = "Ankara";

		System.out.println("Dizi atamaları tamamlandı...");

	}
}
```

Bu örneğe göre 4 uzunluğunda bizi dizi tanımımız var ancak biz bunun 6. elemanına atama yapmaya çalışıyoruz. Doğal olarak aşağıdaki gibi ArrayOutOfBoundException tipinde bir hata oluşuyor.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 6
	at com.merge.kitap.paket.Ornek.main(Ornek.java:8)
```

Peki bu hatayı aldık ve hiç bir şey yapamadan uygulamamız sonlandı. Aslında bu durum ne yazılım geliştiricinin nede uygulamayı kullananın istediği bir şey değil. Doğru olan ise kullanıcıdan doğru bilgiyi istemek veya hatayı bir şekilde tolere etmek.

Şimdi buna göre kod bloğumu aşağıdaki gibi tekrar düzenleyelim.

```
public class Ornek {

	public static void main(String[] args) {

		String[] dizi = new String[4];
		try {
			dizi[6] = "Ankara";
		} catch (Exception e) {
			System.out.println("hatalı bir durum oluştu...");
		}

		System.out.println("Dizi atamaları tamamlandı...");

	}
}
```

Peki değişen ne oldu ? Aslında bunu aşağıdaki çıktımızdan görebiliriz.

```
hatalı bir durum oluştu...
Dizi atamaları tamamlandı...
```

Buna göre hata oluştu ancak biz bu hatayı catch içerisinde kontrol ettik ve kullanıcıya ilettik. Sonrasında ise uygulamamız yarıda kesildemeden işlemlerimize devam edip tamamladık. Yani hata olşsa bile bunu bir şekilde kontrol etmiş olduk.

İşte exception handling yani hata yönetiminin temeli budur. Java içerisinde olası hataları kontrol edip bu durumları kontrol etme yöntemidir.

Peki aynı kod bloğu içerisinde birden fazla hata kontrolü yapılabilir mi ? Tabiki evet. Örneğin aşağıdaki kod bloğunda hem dizi indeks erişimi hemde bu dizi indeks değerine göre dosya varlığı kontrol ediliyor.

```
import java.io.FileNotFoundException;
import java.io.FileReader;

public class Ornek {

	public static void main(String[] args) {

		try {
			String[] dosyaListesi = new String[4];
			new FileReader(dosyaListesi[6]);
		} catch (ArrayIndexOutOfBoundsException e) {
			System.out.println("Dizi boyutu aşıldı...");
		} catch (FileNotFoundException e) {
			System.out.println("Böyle bir dosya bulunamadı...");
		}

	}
}
```

Bu kodu çalıştırdığımızda öncelikli olarak dizi indeksi kontrol edileceğinden ArrayIndexOutOfBoundsException hatası yakalanacak ve sadece bu blok çalıştırılacaktır.

Dikkat : Birden fazla catch bloğu olması durumunda bu bloklardan sadece bir tanesi çalıştırılacaktır.

Peki iç içe try-catch blokları olması durumu olabilir mi ? Tabiki olabilir ve bunu aşağıdaki örnekle görelim.

```
try {
	FileReader kaynak1 = new FileReader("c:/sistem/dosya1.txt");
} catch (FileNotFoundException e) {
	System.out.println("Dosya okunamadı yeni dosya okunuyor...");
	try {
		FileReader kaynak1 = new FileReader("c:/sistem/dosya1.txt");
	} catch (FileNotFoundException e1) {
		System.out.println("İkinci dosya da okunamadı...");
	}
}
```

Bu örneğe göre dosya okuma öncelikle ilk try bloğu içerisinde yapılmaya çalışılıyor ancak ilgili dizin içerisinde dosya bulunamadığından **FileNotFoundException** fırlatılıyor ve catch bloğu işletiliyor. catch bloğu içerisinde yine bir dosya okunmaya çalışıyor ancak hata kontrolü gerektiğinden yine try-catch içerisine alınmak durumunda kalıyor.

Dikkat : try, cathc, finally gibi blokların içerisinde istediğiniz kadar try-catch-finally kullanabilirsiniz.

Hata detaylarını okuma

Hatayı yakaldıktan sonra gerek log mekanizmaları gerekse uygulama geliştirme ortamında bunu görmek için hata detaylarını okumak isteyebiliriz. Bunun için aşağıdaki üç yöntem kullanılabilir.

```
try {
	FileReader kaynak1 = new FileReader("c:/sistem/dosya1.txt");
} catch (FileNotFoundException e) {
	System.out.println(e.getMessage());
	System.out.println(e.toString());
	e.printStackTrace();
}
```

Buna göre hata okuma detayları aşağıdaki gibidir.

getMessage() : Bize hatanın ürettiği mesaji verir.
toString() : Hata mesajıyla birlikte hata sınıfını verir.
printStactTrace() : Hata sınıfı ve ürettiği mesajla birlikte hatanın geçtiği adımları yazdırır.

Buna göre yukarıdaki kod bloğumuzda aşağıdaki gibi bir çıktı veriyor olacaktır.

```
c:\sistem\dosya1.txt (The system cannot find the path specified)
java.io.FileNotFoundException: c:\sistem\dosya1.txt (The system cannot find
the path specified)
java.io.FileNotFoundException: c:\sistem\dosya1.txt (The system cannot find
the path specified)
	at java.io.FileInputStream.open(Native Method)
	at java.io.FileInputStream.<init>(Unknown Source)
	at java.io.FileInputStream.<init>(Unknown Source)
	at java.io.FileReader.<init>(Unknown Source)
	at com.merge.kitap.paket.Ornek.main(Ornek.java:11)
```

Gördüğümüz üzere son çıktımız System.err nesnesini kullanarak geçtiğimiz adımlarla birlikte hatanın oluştuğu kaynaklarıda belirtmektedir.

Checked ve unchecked Exception tipleri

Daha önce de belirttiğimiz üzere hata tiplerini checked ve unchecked olarak ikiye ayırabiliriz. Unchecked hatalar kontrolü zorunlu olmayan hatalardır ve compile-time yani derleme anında kontrol edilme zorunlulukları yoktur. Checked yani kontrollü hatalar ise kaynak kod seviyesinde kontrol edilmelidirler.

Bunu aşağıdaki gibi bir kod bloğu ile örneklendirelim.

```
import java.io.FileReader;

public class Ornek {

	public static void main(String[] args) {
		FileReader dosyaOkuma = new FileReader("Merhaba Dünya");
	}
```

```
}
```

Bu örneğe göre derleme anında bir hata alıyor olacağım. Bu hatanın sebebi FileReader işlemi sırasında FileNotFound hatası kontrolü beklenmesidir. Bu hatanın derleme anında verilmesinin nedeni checked yani kontrollü olmasıdır. Bizim try-catch bloğunu zorunlu olarak kullanmamızı sağlıyor.

Not : Checked yani kontrollü hatalar Exception sınıfından türeyen RuntimeException dışındaki tüm hata tipleridir. Örneğin **FileNotFoundException Exception** sınıfı altındaki **IOException**' dan türetilmiştir.

Peki aynı durumu aşağıdaki gibi farklı bir şekilde yaratalım.

```
public class Ornek {

	public static void main(String[] args) {
		int sayi = Integer.parseInt("abc");
	}
}
```

Buna göre derleme anında herhangi bir hata almayacağız. Aslında parseInt metodu bir **NumberFormatException** bekliyor olmasına karşın bu hata tipi unchecked olduğundan bizim için kontrol zorunuluğu ifade etmiyor. Bu yüzden try-catch kullanımımız zorunlu değildir.

Finally kelimesi

Hata yönetimlerinde hata olsada olmasada her durumda çalıştırılmasını isteğim blokları finally içerisine ekleriz ve kullanımı aşağıdaki gibidir.

```
public class Ornek {

	public static void main(String[] args) {
		int sayi = 0;
		try {
			sayi = Integer.parseInt("aa");
		} catch (Exception e) {
			sayi = 1;
		} finally {
			System.out.println(sayi);
		}
	}
}
```

Bu kod bloğuna göre try içerisinde sayıyı bir String değere çevirerek okumaya çalıştım ancak hata aldım ve catch içerisinde direkt olarak 1 değerini atadım. Son olarak finally bloğunda sayı değerini yazdırmış oldum. Bu durumda hata oluşsa da oluşmasa da finally bloğu çalışacaktır.

Peki önemli bir nokta ! Aşağıdaki iki kod arasında sonuç itibari ile ne fark vardır ?

```java
int sayi = 0;
try {
	sayi = Integer.parseInt("aa");
} catch (Exception e) {
	sayi = 1;
} finally {
	System.out.println(sayi);
}
```

```java
int sayi = 0;
try {
	sayi = Integer.parseInt("aa");
} catch (Exception e) {
	sayi = 1;
}
System.out.println(sayi);
```

Her iki kodu da incelediğimde ikisi arasında finally kullanımı dışında bir fark bulunmuyor ve catch blokları bitiminde değerler finally içinde ve ikincisi dışında olmak üzere yazdırılıyor.

Bu iki kod çıktı olarak farklı bir şey vermeyecektir. Bu yüzdendir ki finally bir anlam içermiyor gibi görünebilir. Ancak aşağıdaki gibi bir kod bloğunda işler daha farklı olacaktır.

```java
public class Ornek {

	public static void main(String[] args) {
		int sayi = 0;
		try {
			sayi = Integer.parseInt("aa");
		} catch (Exception e) {
			sayi = 1;
			return;
		}
		System.out.println(sayi);
	}
}
```

Buna göre catch bloğundan **return** kelimesi ile metodtan çıkılmasını istiyoruz. Böyle bir durum oluşsa dahi return işlemi yapılmadan hemen önce finally garantili olarak çalıştırılır.

Dikkat : finally nin en önemli özelliği garantili olarak çalıştırılmasıdır. Ancak Java işlemini sonlandıracak seviyede bir şey yaparsanız ki bu **System.exit(-1)** olabilir finally çalıştırılmayacaktır.

Hata fırlatmak

Hata mekanizması otomatik olarak çalıştırılmaz. Bir şekilde hatayı yaratan bir ortam olması gerekmektedir. Bunu throw kelimesi ile sağlıyoruz. Örnek vermek gerekirse aşağıdaki kod bloğu içerisinde gelen değer 0 dan küçük veya 5 ten büyükse bir hata oluşturuluyor.

```java
public class Ornek {

	public static void main(String[] args) {
		int x = Integer.parseInt(args[0]);
		if (x < 0 || x > 5)
			throw new IllegalArgumentException("Hatali bir deger
girdiniz");

		System.out.println("Girdiginiz deger : " + x);
	}
```

```
}
```

Normal şartlarda bir sayının 0' dan küçük 5' ten büyük olması herhangi bir hataya neden olmaz. Bu durumda biz bir hata durumu oluşturmalıyız. Bunun için throw kelimesi ile birlikte new diyerek yeni bir hata nesnesi oluşturuyor yani hata fırlatmış oluyoruz.

Şimdi örneği biraz farklı bir hale getirelim.

```
import java.io.FileNotFoundException;

public class Ornek {

	static void sayiKontrol(int x) {
		if (x < 0 || x > 5)
			throw new IllegalArgumentException("Hatali bir deger
girdiniz");
		System.out.println("Girdiginiz deger : " + x);
	}

	static void dosyaOkuma(String dosyaYolu) throws FileNotFoundException {
		if (!dosyaYolu.startsWith("c:/izinliDosyalar/"))
			throw new FileNotFoundException("Izinsiz bir dizine
erismeye calisiyorsunuz");
		System.out.println("Dosya okuma gerceklesti");
	}

	public static void main(String[] args) {
		try {
			sayiKontrol(2);
			dosyaOkuma("c:/windows/deneme");
		} catch (IllegalArgumentException e) {
			e.printStackTrace();
		} catch (FileNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}

}
```

Yukarıdaki örneğe göre sayıKontrol ve dosyaOkuma metodları içerisinde birer kontrol mekanizması oluşturup duruma uyulmadığı halde hata fırlatılmasını sağladık. Burada dikkat çekmenizi istediğim şey dosyaOkuma metodu içerisinde `throws FileNotFoundException` tanımını kullanmam. Bu tanımı kullanma nedenimiz **FileNotFoundException**' ın checked bir exception olması ve dolayısıyla kontrolünün zorunlu hale gelmesidir.

throws kelimesi

Herhangi bir metod tanımı sonunda throws deyimi ile exception belirtimi kullanmak o blok içerisinde exception' ları pas geçmeyi ve bir üst metoda göndermeyi sağlar. Bu özellikle checked exception lar açısından çok faydalıdır.

Eğer aşağıdaki örnekte **throws** `FileNotFoundException` kullanmamış olsaydık metod içerisinde try-catch-finally blokları ile kontrol oluşturmak zorunda kalırdık.

```
static void dosyaOkuma(String dosyaYolu) throws FileNotFoundException {
      if (!dosyaYolu.startsWith("c:/izinliDosyalar/"))
            throw new FileNotFoundException("Izinsiz bir dizine erismeye
calisiyorsunuz");
      System.out.println("Dosya okuma gerceklesti");
}
```

Peki metod içerisinde görmezdek gelmek istediğimiz hata sayısı birden fazla olsaydı ne olacaktı ? Bu durumda throws kelimesi ile birden fazla hata tipini aşağıdaki örnekteki gibi virgüllerle ayırarak yazabilirdik.

```
public void sorgulamaYap(Ogrenci ogr) throws IOException,
                                    SQLException, NullPointerException {

}
```

Özel hata sınıfları oluşturmak

Java içerisinde var olan hata tipleri dışında kendi hata tiplerimizi oluşturma şansınızda bulunuyor. Bunun için Throwable ve alt tiplerini miras alan bir sınıf oluşturmamız yeterlidir.

Peki özel Exception tiplerine neden ihtiyaç duyalım. Bunu örneklendirmek gerekirse aşağıdaki kod bloğu içerisinde fiyat ve mail adresi bilgisi alıyoruz. Bilgiler bilgiKontrol metoduna gönderiliyor eğer fiyat veya mail bilgimiz hatalıysa bizim için IllegalArgumentException tipinde bir hata fırlatıyor. Aslında bu yöntemi seçmemizin sebebi çok basit her iki hatada hatalı argüman gonderiminden kaynaklanıyor. Bu durumda Java' da daha uygun bir Exception tipi bulunmuyor.

```
public class Ornek {
      private static void bilgiKontrol(String mail, Double fiyat) {
            if (mail.indexOf("@") == -1) {
                  throw new IllegalArgumentException();
            } else if (fiyat <= 0.0) {
                  throw new IllegalArgumentException();
            }
      }

      public static void main(String[] args) {
            String mail = "melih.sakarya@gmail.com";
```

```
		Double fiyat = -2.0;

		try {
			bilgiKontrol(mail, fiyat);
		} catch (IllegalArgumentException e) {
			System.out.println("Bir hata oluştu");
		}

		System.out.println("Islem basarili olarak tamamlandi...");

	}

}
```

Bu kod içerisinde herhangi bir sorun olmamasına karşın hatanın neden kaynaklandığını tam olarak algılayamayabiliriz. Şimdi bu hata durumlarına özel aşağıdaki gibi kendi hata sınıflarımızı oluşturalım.

```
public class HataliFiyatException extends IllegalArgumentException{
	public HataliFiyatException() {
		super("Fiyat i yanlis girdiniz");
	}
}
```

```
public class HataliMailException extends IllegalArgumentException {

	public HataliMailException() {
		super("Mail adresinizi hatali girdiniz");
	}
}
```

Şimdi az önceki örneğimizi bu hata tipleriyle gerçekleştirelim.

```
public class Ornek {
	private static void bilgiKontrol(String mail, Double fiyat) {
		if (mail.indexOf("@") == -1) {
			throw new HataliMailException();
		} else if (fiyat <= 0.0) {
			throw new HataliFiyatException();
		}
	}

	public static void main(String[] args) {
		String mail = "melih.sakarya@gmail.com";
		Double fiyat = -2.0;

		try {
			bilgiKontrol(mail, fiyat);
		} catch (HataliMailException e) {
			System.out.println("Mail bilgisi hatali gonderildi");
		} catch (HataliFiyatException e) {
			System.out.println("Fiyat bilgisi hatali gonderildi");
		}
```

```
            System.out.println("Islem basarili olarak tamamlandi...");

      }

}
```

Buna göre artık hatanın tam olarak nereden geldiğini ayırt edebiliyoruz. Bu durumda daha özelleştirilmiş ve kontrol edilebilir bir hata yönetimi ortaya çıkıyor.

Java 7 ile gelen hata yönetimi özellikleri

Java 7 ile hata yönetimi konusunda bir kaç yeni özellik gelmiştir. Bunları aşağıdaki gibi inceleyelim.

Birden fazla kontrol

Eğer bir blok içerisinde birden fazla hata tipi ile karşılaşma riskimiz varsa bu durumda her hata tipine uygun catch blokları yazmamız gerekiyordu. Bir başka alternatifimizse Exception yada bir üst sınıfları ile bu hata tiplerini ortak yakalama şansımızın olmasıdır. Ancak ayrıştırılmış hataları tek bir catch bloğunda yakalamak için aşağıdaki kod örneğini kullanabiliriz. Bu özellik Java 7 ile dile eklenmiştir.

```
try {
      bilgiKontrol(veritabani, dosya);
} catch (FileNotFoundException | SQLException e) {
      System.out.println("Kayit isleminde hata olustu...");
}
```

try-with-resources

Java 7 ile gelen bir başka exception özelliği **try-with-resources** dediğimiz hata yönetiminin kaynaklarla yapılmasıdır. Buradaki önemli nokta Java 7 ile gelen **AutoCloseable** interaface' ini kullanan herhangi bir sınıfın try içerisinde kullanılıp sonunda garantili olarak kapatılmasıdır.

Aşağıdaki örnekte FileReader ile bir dosya okuma işlemi başlatışmış ve kapatma işlemi **try-with-resource** ile otomatik olarak sağlanmıştır. Daha önce ise bu işlemi finally bloklarında yapıyorduk.

```
import java.io.FileReader;
import java.io.IOException;

public class Ornek {
      public static void main(String[] args) {
            try(FileReader f = new FileReader("c:/merge/kitap/ornek.txt")) {
                  System.out.println((char)f.read());
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
}
```

Eğer **try-with-resource** ile kullanılacak kaynak birden fazlaysa aşağıdaki gibi bir kullanım sağlanabilir.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class Ornek {
	public static void main(String[] args) {
		try(	FileReader f = new FileReader("c:/merge/kitap/ornek.txt");
			FileWriter w = new FileWriter("c:/merge/kitap/ornek2.txt")
		) {
			System.out.println((char)f.read());
			w.write("Merhaba Dunya");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Not : AutoCloseable interface' i Connection, ResultSet gibi bir çok sınıfta kullanıldığı gibi bizde kendi sınıflarımızda bu interface' i gerçekleştirip try-with-resources ile birlikte kullanabiliriz.

Dikkat: Java dünyasında hata loglarının yönetimi ve diğer işleyişleri görmek açısından log4j ve slf4j gibi yapılar kullanılabilir.