Paketler ve Erişimler

Paket kavramı

Yazılım geliştirme süreçlerimizde şimdiye kadar karmaşıklığa neden olmaması için sınıf isimlerinin aynı olmamasına dikkat etmiştik. Zaten Eclipse ile bir sınıf oluşturduğumuz anda aynı isimde java ve class uzantılı dosyalar oluşacağından işletim sistemi seviyesinde böyle bir şeye izin verilmeyecektir. Peki çok sayıda sınıf bulunan bir projemizde yine aynı şekilde aynı isimli sınıfların olmamasını mı sağlamamız gerekiyor ? Her seferinde birbirleri ile çakışmayan dosya isimlerini nasıl oluşturabilirim ?

Aslında bu durumda Java' da paket sistemi devreye giriyor ve bizi bu problemden kurtarıyor. Paket sistemi arka planda klasör sistemini kullarak bizim için sınıflarımızın klasörler altında saklanmasını sağlıyor ve bunu sınıf başlarında paket tanımı ile belirtiyor.

Bunu örneklendirmek gerekirse aşağıdaki sınıfımız deneme paketi aldında bulunuyor. Bu durumda aynı zamanda deneme klasörü altında da olması gerekiyor yani paket eşittir klasör diyebiliriz.

```
package deneme;

public class Ornek {

}
```

Paket isimlendirmesinde kullanılan yapı şekli alt klasörlere giderek paketlemedir ki burada her bir klasörü nokta işareti ile belirtebiliyoruz . Aksi durumda yine bir klasör ismi bulmaya çalışacaktık ki bu karmaşıklığı yinede kurtarmayacaktı. Şimdi az önceki örneği aşağıdaki gibi alt dizinlere yayarak yapalım.

```
package deneme.yeni.alt.paket;

public class Ornek {

}
```

Bu örneğe göre bu class dosyamızın duracağı yer deneme\yeni\alt\paket dizini altındadır. Burada alt dizin sayımızın herhangi bir sınırı bulunmamaktadır ve her bir nokta işareti alt dizine geçmeyi ifade eder.

Not: Paketleme yönteminde sınıflar ve fiziksel dosyalar ayrı yerlerde olacağından farklı paketler içerisinde aynı isimde sınıf isimleri yer alabilir. Örnek vermek gerekirse Java içerisinde java.sql.Date ve java.util.Date şeklinde iki ayrı Date sınıfı yer almaktadır.

Peki paketleme konusunda bir standart bulunuyor mu ? Tabiki, aski durumda bir sınıfı paketler içerisinde arayıp bulmak gerçekten zor olabilirdi.

En basitinden standart JSR dahilinde olan java sınıflarını java ve javax paketleri altında bulabiliriz. Mesela dosya işlemleri java.io, veritabanı işlemleri ise java.sql ve javax.sql paketleri içerisindedir.

Bunun dışında Java dünyasında kabül görmüş paketleme standartı da bulunmaktadır. Buna göre domain ve domain uzantıları kullanılarak aşağıdaki şekilde giden bir paketleme vardır.

[domain uzantısı].[domain].[proje veya modül adı]. [modül adı]. [modül adı]

Bunu örneklendirmek gerekirse aşağıdaki gibi paket isimlendirmeleri yapılabilir.

com.merge.egitim.uye
com.merge.crm.kayit.servis
org.lkd.web.uye
gov.enerji.erp.ik
edu.metu.obs.uye

Bu örneklere göre domain uzantıları ilgili kuruma göre com, gov, org gibi tanımlarla başlatılabilir. Bir diğer tanım aşağıdaki örnekteki gibi tr.com.merge seklinde başlatılabilir. Yani domain başlangıcını ülke standartı olan tr.com ile de başlatma gibi bir durum da olabilir.

Farklı paket sınıflarına erişim ve import işlemi

Eğer bir sınıfa farklı bir paket içerisinden erişmek istiyorsanız onu import etmeniz gerekmektedir. Bunu import kelimesi ile aşağıdaki örnekteki gibi yapabilirsiniz.

```
package deneme.yeni.alt.paket;

import java.io.File;
import java.util.Date;

public class Ornek {

	public static void main(String[] args) {
		Date tarih;
		File dosya;
	}
}
```

Yukaridaki örneğe göre Java içerisindeki tarih işlemleri için Date, dosya işlemleri içinse File sınıfında iki adet değişken tanımladık. Fakat bu değişkenler farklı paketler içerisinde olduğundan import etmemiz gerekti.

Import işleminde iki yöntem kullanılabilir. Birincisi yukarıdaki örnekteki gibi direkt olarak sınıfı import edebiliriz. Diğer bir yöntem ise aşağıdaki gibi yıldız ile paketin tamamını import etmektir.

```
package deneme.yeni.alt.paket;

import java.io.*;
import java.util.Date;
```

```
public class Ornek {

	public static void main(String[] args) {
		Date tarih;
		File dosya;
	}
}
```

Buna göre File için java.io içerisindeki sınıfların tamamını import ettik. Date içinse java.util in tamamını almak yerine sadece Date sınıfını import ettiğimizi belirttik.

Dikkat: Bu iki yöntem arasında çalışma anında performans veya bellek kullanımı açısından bir farklılık yoktur ancak derleme anında yavaşlamaya neden olabilir. Bunun dışında kod yazma esnasında örneğin java.util ve java.sql paketlerinin ikisinde de

Not : Eğer farklı paket altında aynı anda aynı isimde iki sınıfı import etmek istiyorsanız bunlardan birini direkt paket ismi ile kullanmanız gerekiyor. Örneğin aşağıdaki sınıf örneğinde hem java.util hemde java.sql altındaki Date sınıfları kullanılmaktadır. Buna göre Date iki kere import edilemeyeceği için java.sql.Date sınıfını direkt paket ile kullanmak zorunda kaldım.

```
package deneme.yeni.alt.paket;

import java.util.Date;

public class Ornek {

	public static void main(String[] args) {
		Date tarih1;
		java.sql.Date tarih2;
	}
}
```

Dikkat : import işlemlerinde yıldız kullanırsanız sadece o paket altındaki sınıflar çağırılır. Örnek vermek gerekirse **import java.util.*** şeklindeki bir tanım sadece **java.util** altındaki sınıfların import işlemini gerçekleştirir dolayısıyla **java.util.zip** altındaki dosyalar için ayrıca import yazmanız gerekecektir.

Dikkat : Java içerisinde java.lang dile özgü bir paket olduğu için otomatik olarak import edilmiştir. Bu yüzden String, Integer veya System gibi sınıflar için java.lang paketini import etmenize gerek yoktur.

Kendi paketlerimize ulaşım

Şimdi standart java paketleri dışında kendi paketimiz içerisindeki bir çağırımı gerçekleştirelim. Aşağıdaki örnekte **SatisServis** isimli sınıfımıza farklı bir paketteki **Ornek** sınıfı içerisinden erişiyorum. Buna göre standart paketlerden faklı bir durum bulunmamaktadır.

```
package com.merge.kitap.servis;

public class SatisServis {
```

```
	public void sepeteEkle(String urun) {
		System.out.println("Ürün sepete eklendi : " + urun);
	}

	public void satinAl() {
		System.out.println("Satin alma islemi tamamlandi");
	}
}
```

```
package deneme.yeni.alt.paket;

import com.merge.kitap.servis.SatisServis;

public class Ornek {

	public static void main(String[] args) {
		SatisServis satisServis = new SatisServis();
		satisServis.sepeteEkle("Kitap");
		satisServis.sepeteEkle("Bilgisayar");
		satisServis.satinAl();
	}
}
```

static import

import yönteminin bir başka kullanımı static import' tur. Amaç static değer veya metodları aşağıdaki örnekteki gibi direkt olarak kullanabilmenizi sağlamaktır.

```
import static java.lang.Math.PI;
import static java.lang.Math.cos;


class Ornek {

	public static void main(String[] args) {
		System.out.println(PI);
		System.out.println(cos(20));
	}
}
```

Bu konuyla ilgili detayli bilgiyi static bölümünde bulabilirsiniz.

Erişimler

Java' da bir kontrol mekanizması olarak sınıflar, metodlar, değişkenler gibi tanımlar için erişim yetkileri bulunmaktadır. Bu erişim yetkileri **public**, **private**, **protected** ve **default** olmak üzere temelinde dört adettir. public, private ve protected birer anahtar kelime iken default isminde bir anahtar kelime bulunmamaktadır. Tanımsız kullanımlara **default** denilmektedir.

Not : default tanımız bazı dokümanlarda **friendly** veya **tanımsız** olarakta kullanılmaktadır.

Erişim kullanımını aşağıdaki gibi örneklendirebiliriz.

```
public class Ornek {

	private String ad;

	protected void hesapla() {
	}

	public void sonuc() {

	}
}
```

Peki bu erişimler neyi ifade ediyor ? Bu tanımları ekran kullanıcılarının erişimleri gibi düşünebilirsiniz ancak kullanıcı erişimleri ile bir alakası olmayıp kodların bir birleri arasında erişimlerini ifade etmektedir.

Bunu aşağıdaki tablo ile açıklayabiliriz.

	Aynı sınıftan	Aynı paketten	Farklı Paketten	Miras durumunda
public	Erişebilir	Erişebilir	Erişebilir	Erişebilir
private	Erişebilir	Erişemez	Erişemez	Erişemez
default	Erişebilir	Erişebilir	Erişemez	Erişemez
protected	Erişebilir	Erişebilir	Erişemez	Erişebilir

public

En açık erişim yetkisine sahiptir. public tanımlara aynı sınıf, aynı paket, farklı paket olmak üzere her yerden erişilebilir.

private

En kısıtlı erişime sahiptir. private tanımlara sadece aynı sınıftaysanız erişebilirsiniz.

default

Bir sınıf, değişken, metod benzeri tanım başına bir erişim eklenmezse bu default erişimdedir. Az önce bahsettiğimiz gibi default erişime friendly veya tanımsız da denmektedir. Aynı sınıf ve aynı paket içerisinden erişime imkan verir.

protected

default ile aynı erişimlere sahiptir. Tek farkı miras alınması anında paket dışından da erişim sağlanabilir. Bu durumda aynı sınıf, aynı paket, miras durumunda da farklı paketten erişim sağlanabilir.

Kurallar

Erişim kullanımları ile ilgili aşağıdaki gibi kurallar bulunmaktadır.

- Sınıflar ve Inteface' ler private ve protected olamaz.
- Inteface içerisindeki metodlar otomatik olarak public' tir.

Şimdi yukarıdaki tablo ve açıklamalar göre aşağıdaki kod bloğundaki erişimleri açıklayalım.

```
package com.merge.kitap.servis;

public class Hesaplama {

	private Double komisyon;
	public String kurTipi;

	public void hesapla(){
	}

	Boolean hesapKapamaBilgisi(){
		return true;
	}

	protected void hesaplamaYontemi(){
	}
}
```

Sınıf public olduğundan herhangi bir yerden erişilebiliyor. Bu durumda sınıf içi sınıf dışı, paket içi ve paket dışındaki herhangi bir kod içeriğinden sınıfa ulaşılabilir.

komisyon tanımı private olduğundan sadece sınıf içerisinden bir kod bloğundan erişilebilir.

kurTipi public olduğundan sınıf içi sınıf dışı, paket içi ve paket dışındaki herhangi bir kod içeriğinden erişilebilir.

hesapla public olduğundan sınıf içi sınıf dışı, paket içi ve paket dışındaki herhangi bir kod içeriğinden erişilebilir.

hesapKapamaBilgisi başında herhangi bir erişim tanımı olmadığıdan bunu default (tanımsız veya friendly de denebilir) olarak düşünebiliriz. Bu durumda sadece sınıf içi veya aynı paketteki kod içeriğinden erişilebilir.

hesaplamaYontemi protected olarak tanımlandığından aynı sınıf, aynı paket, ve miras alınması durumunda farklı bir paketten erişilebilir.

Örnek erişim

Şimdi başka bir örnek yapalım ve burada paket içi ve dışında erişimlerimizi kontrol edelim.

Öncelikle OrnekSinif adında farklı metod erişimleri olan aşağıdaki gibi bir sınıf hazırlıyorum.

```
package com.merge.kitap.paket1;

public class OrnekSinif {
	public void metodBir() {
		System.out.println("Metod bir cagirildi");
	}
```

```
	private void metodIki() {
		System.out.println("Metod iki cagirildi");
	}

	void metodUc() {
		System.out.println("Metod üç cagirildi");
	}

	protected void metodDort() {
		System.out.println("Metod dort cagirildi");
	}

	public void metodBes() {
		System.out.println("Metod bes cagirildi");
		metodIki();
	}
}
```

Şimdi bu örneği aşağıdaki gibi aynı sınıf içerisinde kullanalım.

```
package com.merge.kitap.paket1;

public class AyniPaketIcerisinde {
	public static void main(String[] args) {
		OrnekSinif testSinifi = new OrnekSinif();
		testSinifi.metodBir();
		testSinifi.metodIki(); // Erisimde hata olustu
		testSinifi.metodUc();
		testSinifi.metodDort();
		testSinifi.metodBes();
	}
}
```

Buna göre aynı paket içerisinde olduğundan metodIki dışındaki erişimlerde herhangi bir sıkıntı ile karşılaşmadım. metodIki private olduğundan sadece aynı sınıf içerisinden çağırılabilir. Bu yüzden bu satırda hata alıyoruz.

Burada dikkat etmemiz gereken bir nokta var. Ben metodBes' e erişirken metodBes kendi içerisinde metodIki' ye erişiyor. Bu durumda biz dolaylı yoldan metodIki' ye erişmiş oluyoruz. Metodlar ve değişkenler private gibi kısıtlı erişimlere sahip olsalarda farklı bir metod üzerinden benzeri erişimler sağlanabilir. Bu durum bir sonraki bölümdeki encapsulation kavramında da açıklanacak.

Şimdi sınıfımıza farklı bir paket içerisinden erişelim.

```
package com.merge.kitap.paket2;

import com.merge.kitap.paket1.OrnekSinif;

public class FarkliPaketIcerisindenErisim {
	public static void main(String[] args) {
		OrnekSinif testSinifi = new OrnekSinif();
```

```
            testSinifi.metodBir();
            testSinifi.metodIki(); // Erisimde hata olustu
            testSinifi.metodUc(); // Erisimde hata olustu
            testSinifi.metodDort(); // Erisimde hata olustu
            testSinifi.metodBes();
      }
}
```

Öncelikle farklı bir pakette olduğumuz için kullanacağımız sınıfı import ettik. Şimdi erişimlerimizi kontrol edelim. metodBir ve metodBes' te herhangi bir erişim sıkıntısı olmamasına karşın metodIki, metodUc ve metodDort derleme anında erişim hataları oluşturdu.

Şimdi bu hataları yorumlayalım.

metodIki private olduğundan sadece aynı sınıftan erişilebilir ve paket dışından bir erişime kapalıdır.

metodUc default erişime sahip olduğundan sadece aynı paketten ve sınıftan çağırılabilir, paket dışından erişime kapalıdır.

metodDort protected erişime sahip olduğundan sadece inheritance yani miras anında paket dışından erişilebilir. Direk erişimler sadece sınıf içerisinden ve paket içerisinden yapılabilir.

Encapsulation (Kapsülleme) kavramı

Nesneye dayalı programlama konseptinin özelliklerinden biride encapsulation yani kapsüllemedir. Kapülleme anlamından da anlaşılacağı gibi veriyi kontrol altına almak adına direkt erişimi engellemektir. Bunun iki nedeni vardır.

Birincisi verileri sadece nesne içerisinden erişilebilir hale getirmek. Bu durumda mecvut sınıfın size sunduğu arayüzler dışında veriye erişiminiz olamaz. Yani veri nesnenin kendi yapısı içerisinde saklanır ve direkt erişilemez.

Diğer bir neden ise verilere değer atama esnasında kontrol mekanizması oluşturmak. Bu durumda veriye ulaşırken yetki ve değer kontrolü yapabilirsiniz. Örneğin yaş alanının eksi bir değer olması bu şekilde engellenebir.

Nesneyi korumak için bölümün konusu olan erişimleri kullanırız. Şimdi aşağıdaki gibi bir örnek sınıfımız olsun.

```
public class Ogrenci {
      public String ad;
      public Integer yas;

}
```

Şimdi bu sınıftan bir nesne oluşturup atamalarını gerçekleştirelim.

```
public class Ornek {

	public static void main(String[] args) {
		Ogrenci ogr = new Ogrenci();
		ogr.ad = "M";
		ogr.yas = -30;
	}
}
```

Örneğe göre ad ve yas değerlerini atama sırasında bazı mantık hatalarına neden olduk. Örneğin ad iki karakterden az, yaş ise negatif bir değer olmamalıydı. Şu anda bu durumu kesebileceğimiz bir kontrol mekanizması bulunmuyor. Bunun nedeni ise direkt olarak yas alanının referansının açık olması. Şimdi bu durumu engelleyelim.

```
public class Ogrenci {
	private String ad;
	private Integer yas;

	public void setAd(String ad) {
		if(ad != null && ad.length() > 3)
		this.ad = ad;
	}

	public String getAd() {
		return ad;
	}

	public void setYas(Integer yas) {
		if(yas > 0 && yas < 100)
		this.yas = yas;
	}

	public Integer getYas() {
		return yas;
	}
}
```

Yukarıdaki kod bloğumuza göre değerlere aşağıdaki gibi set ve get metodları ile erişmemiz gerekecektir.

```
public class Ornek {

	public static void main(String[] args) {
		Ogrenci ogr = new Ogrenci();
		ogr.setAd("M");
		ogr.setYas(-33);

		System.out.println(ogr.getAd());
		System.out.println(ogr.getYas());

	}
}
```

Peki buna göre çıktı ne olur ? Çok basit ad ve yas null değerler alacaktır. Sebebiyse set metodlarını çağırdığımız anda kontroller ile beklenen parametreler olmamasından dolayı gönderilen değerleri kabul etmemesi ve atamayı gerçekleştirmemesidir. Direkt olarak ad ve yas alanlarınada erişemeyerceğimize göre değerler metodlar dışında bir erişime kapalıdır.

Yukarıdaki örneğe göre encapsulation sayesinde daha kontrollü bir yapıya sahip olduk.

Dikkat : Nesneye dayalı programlama dünyasında eğer bir nesne tasarımı yapıyorsak ve bu nesne içerisinde değerler bulunuyorsa, bu değer erişimleri aşağıdaki örnekteki gibi get ve set metodları ile yapılmalıdır. Bazı framework' ler ise bunu zorunlu olarak bekler.

```
public class Ogrenci {
	private String ad;
	private Integer yas;

	public void setAd(String ad) {
		this.ad = ad;
	}

	public String getAd() {
		return ad;
	}

	public void setYas(Integer yas) {
		this.yas = yas;
	}

	public Integer getYas() {
		return yas;
	}

}
```