Polymorphism – Çok Biçimlilik

Nesneye dayalı programlama dünyasının önemli konularından biride polymorphism yani çok biçimliliktir. İsim olarak latince bir kelimeden gelip bir nesnenin farklı tiplerde kullanılabilmesini sağlamak olarak açıklanabilir. Aslına bakarsanız polymorphism için kullanılan yada rezerve edilen bir kelime bulunmamaktadır. Kullanım şekli olarak bu tarz kullanım özelliğin olması dilin polymorphism yeteneğinin olması anlamına gelir.

Şimdi bunu bir örnekle açıklayalım. Öncelikle aşağıdaki gibi bir tasarımımız olsun. Buna göre Dokuman isminde bir sınıfım var ve bunu miras alan Fatura ve Sozlesme sınıflarımız bulunuyor. Tasarımıma göre miras durumundan olayı Fatura ve Sozlesme sınıfları da aynı zamanda bir Dokuman olarak nitelendirilmiş oluyor. Bunun yanında her doküman tipinin kendi yazdırma metodları olduğundan Dokuman sınıfı içerisindeki yazdır metodunu ezmiş oluyorlar.

```
public class Dokuman {

	void dokumanOlustur() {
		System.out.println("Dokuman fiziksel olarak oluşturuldu...");
	}

	void yazdir() {
		System.out.println("Doküman yazdirma islemi yapildi...");
	}
}
```

```
public class Fatura extends Dokuman{

	void odemeGerceklestir(){
		// Fatura odeme islemleri...
	}

	@Override
	void yazdir() {
		System.out.println("Fatura yazdiriliyor...");
	}
}
```

```
public class Sozlesme extends Dokuman {

	void baslatma() {
		// Sozlesme baslatma islemleri...
	}

	@Override
	void yazdir() {
		System.out.println("Sozlesme yazdiriliyor...");
	}
}
```

Öncelikle polymorphism kullanmadan bir örnek gerçekleştiriyorum.

```
public class Ornek {

	void yazdirmaIslemi(Fatura fatura) {
		fatura.yazdir();
	}

	void yazdirmaIslemi(Sozlesme sozlesme) {
		sozlesme.yazdir();
	}

	public static void main(String[] args) {
		Fatura fatura = new Fatura();
		Sozlesme sozlesme = new Sozlesme();

		Ornek orn = new Ornek();
		orn.yazdirmaIslemi(fatura);
		orn.yazdirmaIslemi(sozlesme);
	}
}
```

Buna göre Fatura ve Sozlesme tipinde iki nesne oluşturdum ve bu nesneler için yazdırma operasyonu gerçekleştiren iki adet metod hazırlıyorum. Uygulamayı çalıştırdığım zamansa aşağıdaki gibi bir çıktı oluşuyor.

```
Fatura yazdiriliyor...
Sozlesme yazdiriliyor...
```

Halbuki iki nesne içinde aynı işlevi yapmama rağmen neden bu işi tek seferde halledemedim eğer Dokuman sınıfını miras alan onlarca sınıf olsaydı yine herbiri için bu kod bloklarını tekrar mı edecektim ? Aslında çözüm çok basit bunun cevabu Java' da varken ben kullanmadım ? Şimdi bu yeteneği yani polymorphism' i kullanıp aynı işlemi tekrar yapalım.

```
public class Ornek {

	void yazdirmaIslemi(Dokuman dokuman) {
		dokuman.yazdir();
	}

	public static void main(String[] args) {
		Fatura fatura = new Fatura();
		Sozlesme sozlesme = new Sozlesme();

		Ornek orn = new Ornek();
		orn.yazdirmaIslemi(fatura);
		orn.yazdirmaIslemi(sozlesme);
	}
}
```

Evet şimdi kod bloğum daha da kısaldı ve kodun tekrar kullanılabilirliği arttı. yazdirmaIslemi metodunda az önceki kod bloğunda her bir tip için karşılık beklerken bu sınıf içerisinde ortak alan yani Dokuman tipinde bir değer beklediğimi belirttim. Artık Dokuman sınıfından türeyen tüm tipler bu metod içerisine gönderme şansına sahibim. İşte bu kullanım özelliğine poymorphism denir.

Bunun yanında az önceki programın çıktısı aşağıdaki şekilde diğerinden farklı olmadı. Yeni siz metod içerisinde alınan parametreyi Dokuman olarak göstersenizde nesnenin kendisi Fatura veya Sozlesme ozelliğini halen koruyor.

```
Fatura yazdiriliyor...
Sozlesme yazdiriliyor...
```

Bunu kontrol etmek için aşağıdaki gibi bir kod bloğu kullanalım. Buna göre gelen nesnenin sınıf tipini yazdırıyorum.

```
public class Ornek {

	void yazdirmaIslemi(Dokuman dokuman) {
		System.out.println(dokuman.getClass().getSimpleName());
	}


	public static void main(String[] args) {
		Fatura fatura = new Fatura();
		Sozlesme sozlesme = new Sozlesme();

		Ornek orn = new Ornek();
		orn.yazdirmaIslemi(fatura);
		orn.yazdirmaIslemi(sozlesme);
	}
}
```

Sonuç olarak çıktımız aşağıdaki gibi yani döküman değil nesnenin kendi sınıf tipinde olacaktır.

```
Fatura
Sozlesme
```

Dikkat: Polymorphism de göndereceğiniz geçerli tipler o sınıftan türetilen tipledir. Örneğin az önceki örnekte Dokuman parametresi beklenen metoda sadece Dokuman sınıfından turetilen nesneler gönderilebilir. Bir diğer nokta Java' da gizli kalıtımdan dolayı Object sınıfını bekleyen metodlara ilkel tipler dışındaki tüm nesneleri gönderebilirsiniz.

Polymorphism' in bir diğer kullanım şekli aşağıdaki gibi olabilir.

```
public class Ornek {
```

```
	public static void main(String[] args) {
		Fatura fatura = new Fatura();
		Sozlesme sozlesme = new Sozlesme();

		Dokuman d1 = fatura;
		Dokuman d2 = sozlesme;
	}
}
```

Bu örneğe göre Fatura ve Sozlesme tipinde tanımladığım nesneleri Dokuman tipindeki bir değişkene atayabiliyorum. Bu yine polymorphism sayesinda oluşan bir durumdur. Bunun yine aşağıdaki gibi bir örneği sağlanabilir.

```
public static void main(String[] args) {
	Fatura fatura = new Fatura();
	Sozlesme sozlesme = new Sozlesme();

	List<Dokuman> dokumanListesi = new ArrayList<Dokuman>();
	dokumanListesi.add(fatura);
	dokumanListesi.add(sozlesme);
}
```

Sonuç itibariyle tüm örneklerin ortak noktası nesne tiplerinin miras alabildiği biçimlere girebilmesidir. Bu özellikle tekrar kullanım ve esneklik açısından çok önemli bir durumdur.

Interface ve Polymorphism İlişkisi

Interface ve Abstraction bölümünde önemli kullanım nedenlerinden birinin polymorphism olduğunu belirtmiştik. Peki soyutluk kavramının polymorphism ile ne ilişkisi olabilir ?

Aslında yanıt az önceki örneğimizde gizli. Ben az önceki örneğe göre yazdir metodunu miras ile kullandığım tüm sınıflarda ezdim. Bunun nedeni her doküman tipi için yazdırma işlemlerinin farklı olmasıydı. Bu durumda yazdir metodu Dokuman sınıfı içerisinde soyut bir kavram olmalıdır ki bunu soyut yani abstract metod olarak tanımlıyoruz.

Ben yeni tasarım olarak döküman tipinin tamamını aşağıdaki gibi abstract hale getiriyorum be bunu bir interface' e çeviriyorum.

```
public interface Dokuman {

	void olustur();
	void yazdir();
}
```

Bu durumda Dokuman benim için sadece bir tanım haline geldi. Şimdi aşağıdaki gibi bu Interface' ten türetilmiş iki adet gerçekleştirim sınıfı yaratıyorum.

```
public class Fatura implements Dokuman {

	@Override
	public void yazdir() {
		System.out.println("Fatura yazdiriliyor...");
	}

	@Override
	public void olustur() {
		System.out.println("Fatura dokumani olusturuluyor...");
	}
}
```

```
public class Sozlesme implements Dokuman {

	@Override
	public void yazdir() {
		System.out.println("Sozlesme yazdiriliyor...");
	}

	@Override
	public void olustur() {
		System.out.println("Sozlesme dokumani olusturuluyor...");
	}
}
```

Peki şimdi kullanım nasıl olacak ? İşte Dokuman Interface ile soyut olarak tanımladığımız doküman kavramı bizim için aşağıdaki polymorphism özelliğini kazandıracaktır.

```
public class Ornek {

	public void baslatVeYazdir(Dokuman dokuman) {
		dokuman.olustur();
		dokuman.yazdir();
	}

	public static void main(String[] args) {
		Fatura fatura = new Fatura();
		Sozlesme sozlesme = new Sozlesme();

		Ornek ornek = new Ornek();
		ornek.baslatVeYazdir(fatura);
		ornek.baslatVeYazdir(sozlesme);
	}
}
```

Gerçek dünyadan bir örnek

Java' içerisinde veritabanı bağlantılarını sağlayan yapı JDBC' dir. Bunu detaylı olarak JDBC konusunda görüyor olacağız. Ancak bağlantı esnasında hangi veritabanına nasıl bağlanacağını Java nasıl biliyor ?

Aslına bakarsanız Java içerisinde örneğin MySql için hazır bir bağlantı kodu bulunmuyor. Eğer MySql' e bağlanmak istiyorsanız ilgili bağlantı kütüphanesini eklemeniz gerekiyor. İşte bu kütüphane JDBC arayüzlerinin gerçekleştirimini barındırıyor. Bu yüzdendir ki Java ile JDBC kütüphanesi bulunan tüm veritabanlarına bağlanma şansınız bulunuyor. Bu genişletilebilirlik anlamında soyutlama ve polymorphism konusunda bizim için oldukça güzel bir örnek.