POLIMORFIZM

Polimorfizm, nesneye yönelik programlamanın önemli kavramlarından biridir ve sözlük anlamı olarak "bir çok şekil" anlamına gelmektedir. Polimorfizm ile kalıtım konusu iç içedir. Kalıtım konusunu geçen bölüm incelenmişti; kalıtım konusunda iki taraf bulunmaktadır, ana sınıf ve bu sınıftan türeyen alt sınıf/sınıflar. (yorum ekle)

6.1. **Detaylar**

Alt sınıf, türetildiği ana sınıfa ait tüm özellikleri alır; yani, ana sınıf ne yapıyorsa türetilen alt sınıfta bu işlemlerin aynısını yapabilir ama türetilen alt sınıfların kendilerine ait bir çok yeni özelliği de olabilir. Ayrıca türetilen alt sınıfa ait nesnenin, ana sınıf tipindeki referansa bağlamanın yukarı doğru (*upcasting*) işlemi olduğu geçen bölüm incelenmişti. Burada anlatılanları bir örnek üzerinde açıklarsak; (yorum ekle)

Örnek: *PolimorfizmOrnekBir.java* (yorum ekle)

```
class Asker {
 public void selamVer() {
      System.out.println("Asker Selam verdi");
 }
}

class Er extends Asker {
 public void selamVer() {
      System.out.println("Er Selam verdi");
 }
}

class Yuzbasi extends Asker {
 public void selamVer() {
      System.out.println("Yuzbasi Selam verdi");
 }
}

public class PolimorfizmOrnekBir {

 public static void hazirOl(Asker a) {
      a.selamVer(); // ! Dikkat !
 }

 public static void main(String args[]) {
      Asker a = new Asker();
      Er e = new Er();
      Yuzbasi y = new Yuzbasi();
      hazirOl(a); // yukarı cevirim ! yok !
      hazirOl(e); // yukarı cevirim (upcasting)
```

```
        hazirOl(y); // yukarı cevirim (upcasting)
    }
}
```

Yukarıdaki örnekte üç kavram mevcuttur, bunlardan biri yukarı çevirim (*upcasting*) diğeri polimorfizm ve son olarak da geç bağlama (*late binding*). Şimdi yukarı çevirim ve polimorfizm kavramlarını açıklayalım. Bu örneğimizde ana sınıf *Asker*sınıfıdır; bu sınıfdan türeyen sınıflar ise *Er* ve *Yuzbasi* sınıflarıdır. Bu ilişki "bir" ilişkisidir ; (yorum ekle)

- Er **bir** Askerdir, veya
- Yüzbası **bir** Askerdir, diyebiliriz.

Yani *Asker* sınıfının yaptığı her işi *Er* sınıfı veya *Yuzbasi* sınıfı da yapabilir artı türetilen bu iki sınıf kendisine has özellikler taşıyabilir, *Asker* sınıfı ile *Er* ve *Yuzbasi* sınıflarının arasında kalıtımsal bir ilişki bulunmasından dolayı, *Asker* tipinde parametre kabul eden `hazirOl()` yordamına *Er* ve *Yuzbasi* tipindeki referansları paslayabildik, bu özelliğinde yukarı çevirim (*upcasting*) olduğunu geçen bölüm incelenmişti. (yorum ekle)
Polimorfizm ise `hazirOl()` yordamının içerisinde gizlidir. Bu yordamın *(method)* içerisinde *Asker* tipinde olan `a` referansı kendisine gelen 2 değişik nesneye (Er ve Yuzbasi) bağlanabildi; bunlardan biri *Er* diğeri ise *Yuzbasi*'dır. Peki bu yordamın içerisinde neler olmaktadır? Sırası ile açıklarsak; ilk önce *Asker* nesnesine bağlı *Asker* tipindeki referansı, `hazirOl()` yordamına parametre olarak gönderiyoruz, burada herhangi bir terslik yoktur çünkü `hazirOl()` yordamı zaten *Asker* tipinde parametre kabul etmektedir. (yorum ekle)
Burada dikkat edilmesi gereken husus, `hazirOl()` yordamının içerisinde *Asker* tipindeki yerel `a` değişkenimizin, kendi tipinden başka nesnelere de (*Er* ve *Yuzbasi*) bağlanabilmesidir; yani, *Asker* tipindeki yerel `a` değişkeni bir çok şekle girmiş bulunmaktadır. Aşağıdaki ifadelerin hepsi doğrudur: (yorum ekle)

- `Asker a = new Asker() ;`
- `Asker a = new Er();`
- `Asker a = new Yuzbasi();`

Yukarıdaki ifadelere, *Asker* tipindeki `a` değişkenin açısından bakarsak, bu değişkenin bir çok nesneye bağlanabildiğini görürüz, bu özellik polimorfizm 'dir -ki bu özelliğin temelinde kalıtım (*inheritance*) yatar. Şimdi sıra geç bağlama (*late binding*) özelliğinin açıklanmasında.... (yorum ekle)

6.2. Geç Bağlama (*Late Binding*)

Polimorfizm olmadan, geç bağlamadan bahsedilemez bile, polimorfizm ve geç bağlama (*late binding*) bir elmanın iki yarısı gibidir. Şimdi kaldığımız yerden devam ediyoruz**,** *Er* nesnesine bağlı *Er* tipindeki referansımızı (`e`) `hazirOl()` yordamına parametre olarak gönderiyoruz. (yorum ekle)

Gösterim-6.1:

```
hazirOl(e); // yukari dogru cevirim (upcasting)
```

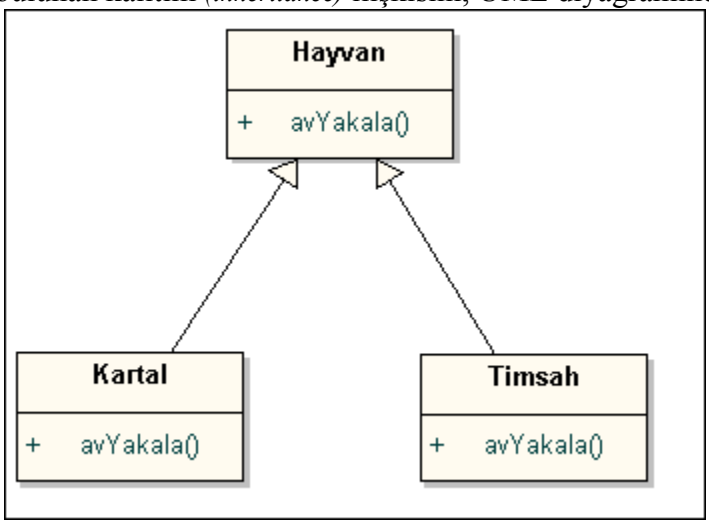
Bu size ilk başta hata olarak gelebilir, ama arada kalıtım ilişkisinden dolayı (Er **bir** Askerdir) nesneye yönelik programlama çerçevesinde bu olay doğrudur. En önemli kısım geliyor; şimdi, hangi nesnesin `selamVer()` yordamı çağrılacaktır? *Asker*nesnesinin mi? Yoksa *Er* nesnesinin mi ? Cevap: *Er* nesnesinin `selamVer()` yordamı çağrılacaktır. Çünkü *Asker* tipindeki yerel değişken (`a`) *Er* nesnesine bağlanmıştır. Eğer *Er* nesnesinin `selamVer()` yordamı olmasaydı o zaman *Asker* nesnesine ait olan `selamVer()` yordamı çağrılacaktı fakat *Er* sınıfının içerisinde, ana sınıfa ait olan (*Asker* sınıfı) `selamVer()` yordamı iptal edildiğinden (*override*) dolayı, Java, *Er* nesnesinin `selamVer()` yordamını çağırılacaktır. Peki hangi nesnesinin `selamVer()` yordamının çağrılacağı ne zaman belli olur? Derleme anında mı (*compile-time*)? Yoksa çalışma anında mı (*run-time*)? Cevap; çalışma anında (*run-time*). Bunun sebebi, derleme anında `hazirOl()` yordamına hangi tür nesneye ait referansın gönderileceğinin belli olmamasıdır. (yorum ekle)
Son olarak, *Yuzbasi* nesnesine bağlı *Yuzbasi* tipindeki referansımızı `hazirOl()` yordamına parametre olarak gönderiyoruz. Artık bu bize şaşırtıcı gelmiyor... devam ediyoruz. Peki şimdi hangi nesneye ait `selamVer()` yordamı çağrılır? *Asker*nesnesinin mi? Yoksa *Yuzbasi* nesnesinin mi? Cevap *Yuzbasi* nesnesine ait olan `selamVer()` yordamının çağrılacağıdır çünkü *Asker* tipindeki yerel değişkenimiz *heap* alanındaki *Yuzbasi* nesnesine bağlıdır ve `selamVer()` yordamı *Yuzbasi* sınıfının içerisinde iptal edilmiştir (*override*). Eğer `selamVer()` yordamı *Yuzbasi* sınıfının içerisinde iptal edilmeseydi o zaman *Asker* sınıfına ait (ana sınıf) `selamVer()` yordamı çağrılacaktı. Aynı şekilde Java hangi nesnenin `selamVer()` yordamının çağrılacağına çalışma-anında (*run-time*) da karar verecektir yani geç bağlama özelliği devreye girmiş olacaktır. Eğer bir yordamın hangi nesneye ait olduğu çalışma anında belli oluyorsa bu olaya geç bağlama (*late-binding*) denir. Bu olayın tam tersi ise erken bağlamadır (*early binding*); yani, hangi nesnenin hangi yordamının çağrılacağı derleme anında bilinmesi. Bu örneğimiz çok fazla basit olduğu için, "Niye ! derleme anında hangi sınıf tipindeki referansın `hazirOl()` yordamına paslandığını bilemeyelim ki, çok kolay, önce *Asker* sınıfına ait bir referans sonra *Er* sınıfına ait bir referans ve en son olarak da *Yuzbasi* sınıfına ait bir referans bu yordama parametre olarak gönderiliyor işte..." diyebilirsiniz ama aşağıdaki örneğimiz için aynı şeyi söylemeniz bu kadar kolay olmayacaktır (yorum ekle)

Örnek: *PolimorfizmOrnekIki.java* (yorum ekle)

```
class Hayvan {
  public void avYakala() {
      System.out.println("Hayvan av Yakala");
  }
}

class Kartal extends Hayvan {
  public void avYakala() {
      System.out.println("Kartal av Yakala");
```

```
  }
}

class Timsah extends Hayvan{
  public void avYakala() {
      System.out.println("Timsah av Yakala");
  }
}

public class PolimorfizmOrnekIki {

  public static Hayvan rasgeleSec() {
      int sec = ( (int) (Math.random() *3) ) ;
      Hayvan h = null ;
      if (sec == 0) h = new Hayvan();
      if (sec == 1) h = new Kartal();
      if (sec == 2) h = new Timsah();
      return h;
  }

  public static void main(String args[]) {
      Hayvan[] h = new Hayvan[3];
      // diziyi doldur
      for (int i = 0 ; i < 3 ; i++) {
        h[i] = rasgeleSec(); //upcasting
      }
      // dizi elemanlarini ekrana bas
      for (int j = 0 ; j < 3 ; j++) {
        h[j].avYakala(); // !Dikkat!
      }
  }
}
```

Yukarıdaki örnekte bulunan kalıtım *(inheritance)* ilişkisini, UML diyagramında gösterirsek:



Şekil-6.1. Kalıtım, Polimorfizm ve Geç Bağlama

PolimorfizmOrnekIki.java örneğimizde `rasgeleSec()` yordamı, rasgele *Hayvan* nesneleri oluşturup geri döndürmektedir. Geri döndürülen bu *Hayvan* nesneleri, Hayvan tipindeki dizi içerisine atılmaktadır. Hayvan dizisine atılan *Kartal* ve *Timsah*nesnelerine Java'nın kızmamasındaki sebep kalıtımdır. *Kartal* **bir** *Hayvan'dır* diyebiliyoruz aynı şekilde *Timsah* **bir** *Hayvandır* diyebiliyoruz; olaylara bu açıdan bakarsak Hayvan tipindeki dizi içerisine eleman atarken yukarı çevirim (upcasting) olduğunu fark edilir. (yorum ekle)
Geç bağlama ise, *Hayvan* dizisinin içerisindeki elemanlara ait `avYakala()` yordamını çağırırken karşımıza çıkar. Buradaki ilginç nokta hangi nesnenin `avYakala()` yordamının çağrılacağının derleme anında (*compile-time*) bilinemiyor olmasıdır. Nasıl yani diyenler için konuyu biraz daha açalım. `rasgeleSec()` yordamını incelersek, `Math.random()` yordamının her seferinde 0 ile 2 arasında rasgele sayılar ürettiği görülür. Bu üretilen sayılar doğrultusunda *Hayvan* nesnesi *Kartal* nesnesi veya *Timsah*nesnesi döndürülebilir; bu sebepten dolayı uygulamamızı her çalıştırdığımızda *Hayvan* tipindeki dizinin içerisine değişik tipteki nesnelerin, değişik sırada olabilecekleri görülür. Örneğin *PolimorfizmIki* uygulamamızı üç kere üst üste çalıştırıp çıkan sonuçları inceleyelim; Uygulamamızı çalıştırıyorum. (yorum ekle)

Gösterim-6.2:

```
java  PolimorfizmIki
```

Uygulamanın çıktısı aşağıdaki gibidir;

```
Kartal avYakala
Hayvan avYakala
Kartal avYakala
```

Aynı uygulamamızı tekrardan çalıştırıyorum;

```
Timsah avYakalaTimsah avYakalaHayvan avYakala
```

Tekrar çalıştırıyorum;

```
Timsah av Yakala
Hayvan av Yakala
Kartal av Yakala
```

Görüldüğü üzere dizi içerisindeki elemanlar her sefersinde farklı olabilmektedir, dizi içerisindeki elemanlar ancak çalışma anında (*runtime*) belli oluyorlar. `h[j].avYakala()` derken, derleme anında (*compile-time*) hangi nesnenin `avYakala()` yordamının çağrılacağını Java tarafından bilinemez, bu olay ancak çalışma anında (*run-time*) bilinebilir. Geç bağlama özelliği bu noktada karşımıza çıkar. Geç bağlamanın (*late-binding*) diğer isimleri, dinamik bağlama (*dynamic-binding*) veya çalışma anında bağlamadır. (*runtime-binding*). (yorum ekle)

6.3. **Final ve Geç Bağlama**

5. bölümde, `final` özelliğinin kullanılmasının iki sebebi olabileceğini belirtmiştik. Bunlardan bir tanesi tasarım diğeri ise verimliliktir. Verimlilik konusu geç bağlama (*late binding*) özelliği ile aydınlamış bulunmaktadır, şöyle ki, eğer biz bir sınıfı `final`yaparsak, bu sınıfa ait tüm yordamları `final` yapmış oluruz veya eğer istersek tek başına bir yordamı da `final` yapabiliriz. Bir yordamı `final` yaparak şunu demiş oluruz, bu yordam, türetilmiş olan alt sınıfların içerisindeki diğer yordamlar tarafından iptal edilemesin (*override*) Eğer bir yordam iptal edilemezse o zaman geç bağlama *(late binding)* özelliği de ortadan kalkar. (yorum ekle)
Uygulama içerisinde herhangi bir nesneye ait normal bir yordam (final olmayan) çağrıldığında, Java, acaba doğru nesnenin uygun yordam mu çağrılıyor diye bir kontrol yapar, daha doğrusu geç bağlamaya *(late-binding)* ihtiyaç var mı kontrolü yapılır. Örneğin *Kedi* sınıfını göz önüne alalım. *Kedi* sınıfı `final` olmadığından dolayı bu sınıftan türetilme yapabiliriz. (yorum ekle)

Örnek: *KediKaplan.java* (yorum ekle)

```
class Kedi {

    public void yakalaAv() {
        System.out.println("Kedi sinifi Av yakaladi");
    }

}

class Kaplan extends Kedi {

  public static void goster(Kedi k) {
      k.yakalaAv();
  }

  public void yakalaAv() {
        System.out.println("Kaplan sinifi Av yakaladi");
  }

  public static void main(String args[] ) {
        Kedi k = new Kedi() ;
        Kaplan kp = new Kaplan();
        goster(k);
        goster(kp); // yukari dogru cevirim (upcasting)

  }
}
```

Kaplan sınıfına ait statik bir yordam olan `goster()` yordamının içerisinde *Kedi* tipindeki `k` yerel değişkene bağlı olan nesnenin, `yakalaAv()` yordamı çağrılmaktadır ama hangi nesnenin `yakalaAv()` yordamı? *Kedi* nesnesine ait olan mı? Yoksa *Kaplan*nesnesine ait olan mı? Java, `yakalaAv()` yordamını çağırmadan evvel geç bağlama (late-binding) özelliğini devreye sokarak, doğru nesneye ait uygun yordamı çağırmaya çalışır tabii bu işlemler sırasından

verimlilik düşer. Eğer biz *Kedi* sınıfını `final`yaparsak veya sadece `yakalaAv()` yordamını `final` yaparsak geç bağlama özelliğini kapatmış oluruz böylece verimlilik artar. (yorum ekle)

Örnek: *KediKaplan2.java* (yorum ekle)

```
class Kedi2 {

      public final void yakalaAv() {
        System.out.println("Kedi sinifi Av yakaladi");
      }

}

class Kaplan2 extends Kedi2 {

  public static void goster(Kedi2 k) {
      // k.yakalaAv(); // ! dikkat !
  }

  /* iptal edemez
  public void yakalaAv() {
      System.out.println("Kaplan sinifi Av yakaladi");
  }
  */
  public static void main(String args[] ) {
        Kedi2 k = new Kedi2() ;
        Kaplan2 kp = new Kaplan2();
        goster(k);
        goster(kp);
      }
}
```

KediKaplan2.java örneğimizde, `yakalaAv()` yordamını `final` yaparak, bu yordamın *Kaplan* sınıfının içerisinde iptal edilmesini engelleriz; yani, geç bağlama (*late binding*) özelliği kapatmış oluruz. (yorum ekle)

6.4. **Neden Polimorfizm?**

Neden polimofizm sorusuna yanıt aramadan evvel, polimorfizm özelliği olmasaydı olayların nasıl gelişeceğini önce bir görelim. Nesneye yönelik olmayan programlama dillerini kullanan kişiler için aşağıdaki örneğimiz gayet normal gelebilir.`mesaiBasla()` yordamının içerisindeki `if-else` ifadelerine dikkat lütfen. (yorum ekle)

Örnek: *IsYeriNon.java* (yorum ekle)

```
class Calisan {
      public String pozisyon = "Calisan";
```

```
      public void calis() {
      }
}

class Mudur  {

      public String pozisyon = "Mudur";

      public Mudur () { // yapılandırıcı
               pozisyon = "Mudur" ;
      }
      public void calis() {  // iptal etme (override)
               System.out.println("Mudur Calisiyor");
      }
}

class Programci {

      public String pozisyon = "Programci";

      public  Programci() {  // yapılandırıcı
               pozisyon = "Programci" ;
      }
      public void calis() { // iptal etme (override)
               System.out.println("Programci Calisiyor");
      }
}

class Pazarlamaci {

      public String pozisyon = "Pazarlamaci";

      public  Pazarlamaci() { // yapilandirici
               pozisyon = "Pazarlamaci" ;
      }

      public void calis() {  // iptal etme (override)
               System.out.println("Pazarlamaci Calisiyor");
      }
}


public class IsYeriNon {

      public static  void mesaiBasla(Object[] o ) {

          for ( int i = 0 ; i < o.length ; i++ ) {

            if ( o[i] instanceof Calisan ) {
                Calisan c = (Calisan) o[i] ; //aşağıya çevirim
                 c.calis();
              } else if ( o[i] instanceof Mudur ) {
                 Mudur m = (Mudur) o[i] ; //aşağıya çevirim
```

```
                m.calis();
              } else if ( o[i] instanceof Programci ) {
                  Programci p = (Programci) o[i] ; //aşağıya çevirim
                p.calis();
            } else if ( o[i] instanceof Pazarlamaci ) {
                  Pazarlamaci paz = (Pazarlamaci) o[i]; //aşağıya
                   //çevirim
                paz.calis();
            }

              //...
          }
       }

      public static void main(String args[]) {
          Object[] o = new Object[4];
          o[0] = new Calisan();     // yukarı cevirim (upcasting)
          o[1] = new Programci();   // yukarı cevirim (upcasting)
          o[2] = new Pazarlamaci(); // yukarı cevirim (upcasting)
          o[3] = new Mudur();    // yukarı cevirim (upcasting)
          mesaiBasla(o);
      }
}
```

Yukarıdaki örneğimizde nesneye yönelik programlamaya yakışmayan davranıştır (OOP), `mesaiBasla()` yordamının içerisinde yapılmaktadır. Bu yordamımızda, dizi içerisindeki elemanların teker teker hangi tipte oldukları kontrol edilip, tiplerine göre`calis()` yordamları çağrılmaktadır. *Calisan* sınıfından türeteceğim her yeni sınıf için `mesaiBasla()` yordamının içerisinde ayrı bir `if-else` koşul ifade yazmak gerçekten çok acı verici bir olay olurdu. Polimorfizm bu durumda devreye girerek, bizi bu zahmet veren işlemden kurtarır. Öncelikle yukarıdaki uygulamamızın çıktısı inceleyelim. (yorum ekle)

```
Programci CalisiyorPazarlamaci CalisiyorMudur Calisiyor
```

IsYeriNon.java örneğimizi nesneye yönelik programlama çerçevesinde tekrardan yazılırsa (yorum ekle)

Örnek: *IsYeri.java* (yorum ekle)

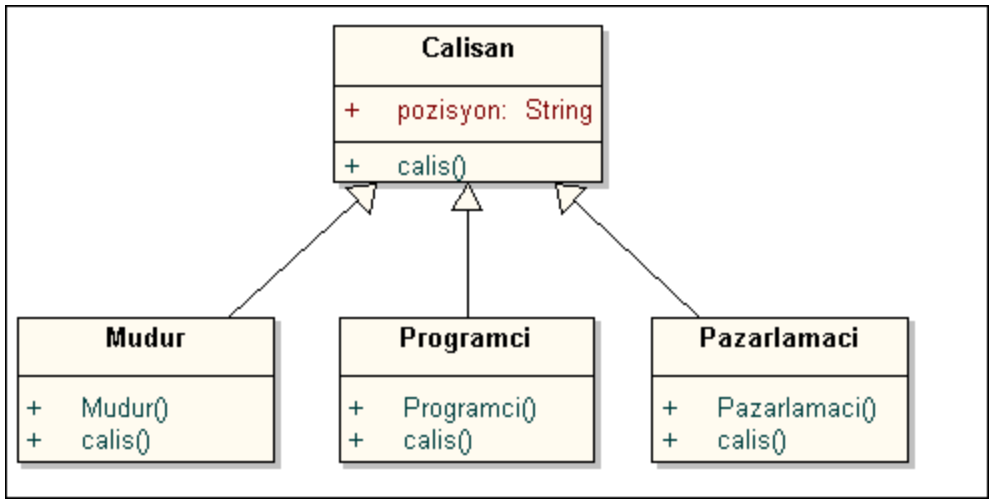
```
class Calisan {
  public String pozisyon="Calisan" ;
  public void calis() {}
}

class Mudur extends Calisan {

  public Mudur () { // yapılandırıcı
      pozisyon = "Mudur" ;
  }
  public void calis() { // iptal etme (override)
```

```
      System.out.println("Mudur Calisiyor");
  }
}

class Programci extends Calisan {
  public Programci() { // yapılandırıcı
      pozisyon = "Programci" ;
  }
  public void calis() { // iptal etme (override)
      System.out.println("Programci Calisiyor");
  }
}

class Pazarlamaci extends Calisan {
  public Pazarlamaci() { // yapılandırıcı
      pozisyon = "Pazarlamaci" ;
  }
  public void calis() { // iptal etme (override)
      System.out.println("Pazarlamaci Calisiyor");
  }
}

public class IsYeri {
  public static void mesaiBasla(Calisan[] c ) {
      for (int i = 0 ; i < c.length ; i++) {
        c[i].calis(); // !Dikkat!
      }
  }
  public static void main(String args[]) {
        Calisan[] c = new Calisan[4];
        c[0] = new Calisan(); // yukarı cevirim gerekmiyor
        c[1] = new Programci(); // yukarı cevirim (upcasting)
        c[2] = new Pazarlamaci(); // yukarı cevirim (upcasting)
        c[3] = new Mudur(); // yukarı cevirim (upcasting)
        mesaiBasla(c);
  }
}
```

Görüldüğü üzere `mesaiBasla()` yordamı artık tek satır, bunu polimorfizm ve tabii ki geç bağlamaya borçluyuz. Bu sayede artık *Calisan* sınıfından istediğim kadar yeni sınıf türetebilirim, yani genişletme olayını rahatlıkla yapabilirim hem de mevcut yapıyı bozmadan. Uygulamanın çıktısında aşağıdaki gibidir. (yorum ekle)

```
Programci CalisiyorPazarlamaci CalisiyorMudur Calisiyor
```

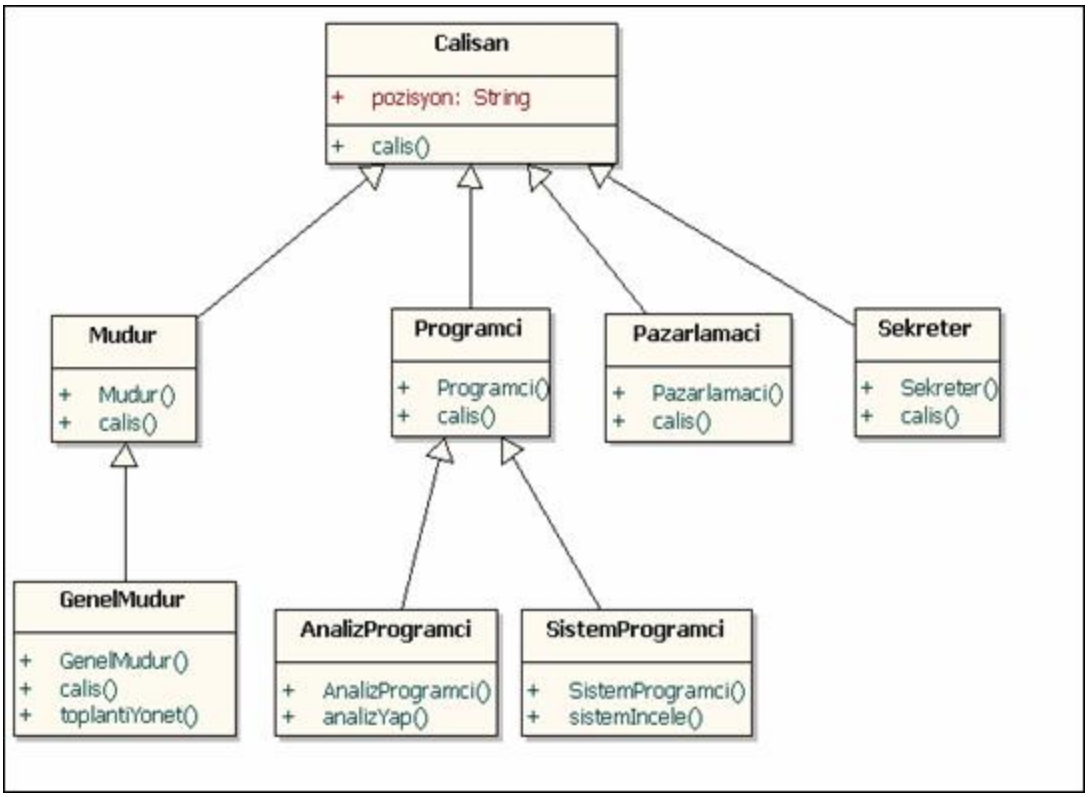
Bu uygulamadaki sınıflara ait UML diyagramı aşağıdaki gibidir.



Şekil-6.2. Kullanılan sınıf yapısı

6.5. **Genişletilebilirlik** ***(Extensibility)***

Polimorfizm sayesinde genişletebilirlik olayı çok basite indirgenmiş bulunmaktadır. Genişletebilirlik, mevcut hiyerarşiyi kalıtım yolu ile genişletmedir. Şimdi *IsYeri.java* örneğimizi biraz daha genişletelim; Yeni uygulamamızın adını *BuyukIsYeri.java*yapalım, bu uygulamamız için, sınıflara ait UML diyagramı aşağıdaki gibidir; (yorum ekle)



Şekil-6.3. Büyük İş Yeri ve Çalışanlar

Yukarıdaki UML diyagramında görüldüğü üzere mevcut hiyerarşiyi genişlettik ve toplam 4 adet yeni sınıfı sistemimize eklendik (*GenelMudur*, *AnalizProgramci*, *SistemProgramci*, *Sekreter*). Yukarıdaki UML şeması Java uygulamasına çevrilirse: (yorumekle)

Örnek: *BuyukIsyeri.java* (yorum ekle)

```
class Calisan {
  public String pozisyon ;
  public void calis() {}
}

class Mudur extends Calisan {

  public Mudur () { // yapılandırıcı
      pozisyon = "Mudur" ;
  }
  public void calis() { // iptal etme (override)
      System.out.println("Mudur Calisiyor");
  }
}

class GenelMudur extends Mudur {

  public GenelMudur () { // yapılandırıcı
      pozisyon = "GenelMudur" ;
```

```
  }
  public void calis() { // iptal etme (override)
      System.out.println("GenelMudur Calisiyor");
  }
  public void toplantiYonet() {
      System.out.println("GenelMudur toplanti yonetiyor");
  }
}

class Programci extends Calisan {

  public Programci() { // yapılandırıcı
      pozisyon = "Programci" ;
  }
  public void calis() { // iptal etme (override)
      System.out.println("Programci Calisiyor");
  }
}

class AnalizProgramci extends Programci {

  public AnalizProgramci() { // yapılandırıcı
      pozisyon = "AnalizProgramci" ;
  }
  public void analizYap() {
      System.out.println("Analiz Yapiliyor");
  }
}

class SistemProgramci extends Programci {

  public SistemProgramci() { // yapılandırıcı
      pozisyon = "SistemProgramci" ;
  }
  public void sistemIncele() {
      System.out.println("Sistem Inceleniyor");
  }
}

class Pazarlamaci extends Calisan {

  public Pazarlamaci() { // yapılandırıcı
      pozisyon = "Pazarlamaci" ;
  }
  public void calis() { // iptal etme (override)
      System.out.println("Pazarlamaci Calisiyor");
  }
}

class Sekreter extends Calisan {

  public Sekreter() { // yapılandırıcı
      pozisyon = "Sekreter" ;
```

```
  }
  public void calis() { // iptal etme (override)
      System.out.println("Sekreter Calisiyor");
 }
}

public class BuyukIsYeri {

  public static void mesaiBasla(Calisan[] c ) {
      for (int i = 0 ; i < c.length ; i++) {
        c[i].calis(); // ! Dikkat !
      }
  }

  public static void main(String args[]) {
   Calisan[] c = new Calisan[7];
   c[0]=new Calisan(); //yukarı cevirim gerekmiyor
   c[1]=new Programci(); //yukarı cevirim (upcasting)
   c[2]=new Pazarlamaci(); // yukarı cevirim (upcasting)
   c[3]=new Mudur(); // yukarı cevirim (upcasting)
   c[4]=new GenelMudur(); // yukarı cevirim (upcasting)
   c[5]=new AnalizProgramci(); // yukarı cevirim (upcasting)
   c[6]=new SistemProgramci(); // yukarı cevirim (upcasting)
   mesaiBasla(c);
  }
}
```

Yukarıdaki örnekte dikkat edilirse `mesaiBasla()` yordamı hala tek satır. Uygulamanın çıktısı aşağıdaki gibidir; (yorum ekle)

```
Programci Calisiyor
Pazarlamaci Calisiyor
Mudur Calisiyor
GenelMudur Calisiyor
Programci Calisiyor
Programci Calisiyor
```

Burada yapılan iş, *Calisan* sınıfından yeni sınıflar türetmektir, bu yeni türetilmiş sınıfların (*GenelMudur*, *AnalizProgramci*, *SistemProgramci*, *Sekreter*) `calis()` yordamlarını çağırmak için ekstra bir yük üstlenilmedi (`mesaiBaslat()` yordamının içerisine dikkat edersek). Polimorfizm ve tabii ki geç bağlama sayesinde bu işler otomatik olarak gerçekleşmektedir. (yorum ekle)

6.6. **Soyut Sınıflar ve Yordamlar (Abstract Classes and Methods)**

Soyut kavramını anlatmadan evvel, *IsYeri.java* ve *BuyukIsyeri.java* örneklerini inceleyelim. Bu uygulamaların içerisinde hiç bir iş yapmayan ve sanki boşuna oraya yerleştirilmiş hissi veren bir sınıf göze çarpar; evet bu *Calisan* sınıfıdır. *Calisan* sınıfını daha yakından bakılırsa; (yorum ekle)

Gösterim-6.3:

```
class Calisan {
  public String pozisyon = "Calisan";
  public void calis() {}
}
```

Yukarıda görüldüğü üzere *Calisan* sınıfı hiç bir iş yapmamaktadır. Akıllara şöyle bir soru daha gelebilir "Madem ki *Calisan* sınıfı hiç bir iş yapmıyor, ne diye onu oraya yazdık"; cevap: birleştirici bir rol oynadığı için, *Calisan* sınıfını oraya yazdık diyebilirim. Olayları biraz daha detaylandıralım. (yorum ekle)

Soyut sınıflar, şu ana kadar bildiğimiz sınıflardan farklıdırlar. Soyut (*abstract*) sınıflarımızı direk `new()` ile oluşturamayız. Soyut sınıfların var olmasındaki en büyük sebeplerden biri birleştirici bir rol oynamalarıdır. Soyut bir sınıftan türetilmiş alt sınıflara ait nesneler, çok rahat bir şekilde yine bu soyut sınıf tipindeki referanslara bağlanabilirler (yukarı çevirim). Böylece polimorfizm ve geç bağlamanın kullanılması mümkün olur. (yorum ekle)

Bir sınıfın soyut olması için, bu sınıfın içerisinde en az bir adet soyut yordamının bulunması gerekir. Soyut yordamların gövdesi bulunmaz; yani, içi boş hiçbir iş yapmayan yordam görünümündedirler. Soyut bir sınıfdan türetilmiş alt sınıflar, bu soyut sınıfın içerisindeki soyut yordamları kesin olarak iptal etmeleri (*override*) gerekmektedir. Eğer türetilmiş sınıflar, soyut olan ana sınıflarına ait bu soyut yordamları iptal etmezlerse, derleme anında (*compile-time*) hata ile karşılaşılır. (yorum ekle)

Gösterim-6.4:

```
abstract void calis() ; // gövdesi olmayan soyut yordam
```

Soyut sınıfların içerisinde soyut yordamlar olacağı gibi, gövdeleri olan, yani iş yapan yordamlarda bulunabilir. Buraya kadar anlattıklarımızı bir uygulama üzerinde pekiştirelim; (yorum ekle)

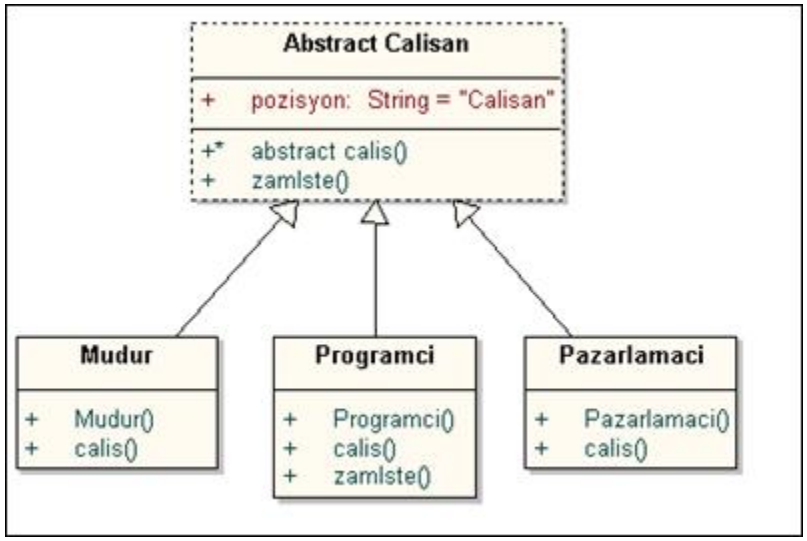
Örnek: *AbIsYeri.java,* (yorum ekle)

```
abstract class Calisan {
  public String pozisyon="Calisan" ;
  public abstract void calis() ;// soyut yordam
  public void zamIste() { // soyut olmayan yordam
      System.out.println("Calisan zamIste");
  }
}

class Mudur extends Calisan {

  public Mudur () { // yapılandırıcı
      pozisyon = "Mudur" ;
  }
```

```
  public void calis() { // iptal etme (override)
      System.out.println("Mudur Calisiyor");
  }
}

class Programci extends Calisan {

  public Programci() { // yapılandırıcı
      pozisyon = "Programci" ;
  }
  public void calis() { // iptal etme (override)
      System.out.println("Programci Calisiyor");
  }

  public void zamIste() { // iptal etme (override)
      System.out.println("Programci Zam Istiyor");
  }
}

class Pazarlamaci extends Calisan {

  public Pazarlamaci() { // yapılandırıcı
      pozisyon = "Pazarlamaci" ;
  }
  public void calis() { // iptal etme (override)
      System.out.println("Pazarlamaci Calisiyor");
  }
}

public class AbIsYeri {

  public static void mesaiBasla(Calisan[] c ) {
      for (int i = 0 ; i < c.length ; i++) {
        c[i].calis(); // !Dikkat!
      }
  }

  public static void main(String args[]) {
      Calisan[] c = new Calisan[3];
      // c[0] = new Calisan(); // soyut sınıflar  new ile
direk  oluşturulamazlar
      c[0] = new Programci(); // yukarı cevirim
 (upcasting)
      c[1] = new Pazarlamaci();// yukarı cevirim
(upcasting)
      c[2] = new Mudur(); // yukari  cevirim (upcasting)
     mesaiBasla(c);
  }
}
```

Yukarıdaki örneğimizi incelersek, soyut (*abstract*) olan *Calisan* sınıfının 2 adet yordamı olduğu görürüz. Bu iki yordamdan bir tanesi olan

soyut (*abstract*) `calis()` yordamıdır. *Calisan* sınıfından türeyen sınıflar tarafından bu yordam kesin kes iptal edilmek (*override*) zorundadır. Baştan açıklarsak, eğer bir sınıfın içerisinde soyut (*abstract*) bir yordam varsa o zaman bu sınıf da soyut (*abstract*) olmak zorundadır. Fakat soyut olan sınıfların içerisinde normal yordamlarda bulunabilir aynı `zamIste()` yordamının *Calisan* sınıfının içerisinde bulunduğu gibi. (yorum ekle)
Calisan sınıfından türeyen diğer sınıfları incelenirse, bu sınıfların hepsinin, `calis()` yordamını iptal ettikleri (*override*) görülür ama aynı zamanda `zamIste()` yordamı sadece *Programci* sınıfının içerisinde iptal edilmiştir (*override*). Eğer ana sınıfın içerisindeki bir yordamın türemiş sınıflar içerisinde iptal edilmeleri (*override*) şansa bırakılmak istenmiyorsa; o zaman bu yordamın soyut olarak tanımlanması gerekir. (yorum ekle)
Dikkat edilmesi gereken bir başka nokta, soyut sınıfların direk olarak `new()` ile oluşturulamıyor olmasıdır. Soyut sınıf demek birleştirici rol oynayan sınıf demektir. *AbIsYeri.java* uygulamasındaki sınıflara ait UML diyagramı aşağıdaki gibidir; (yorumekle)



Şekil-6.4. AbIsYeri.java uygulamasında kullanılan sınıflar

UML diyagramından daha net bir biçimde görmekteyiz ki türemiş sınıflar, ana sınıfa ait `calis()` yordamını iptal etmek (*override*) zorunda bırakılmışlardır. Fakat `zamIste()` yordamı ise sadece *Calisan* sınıfından türemiş olan *Programci* sınıfı içerisinde iptal edilmiştir (*override*). (yorum ekle)

6.6.1. **Niye Soyut Sınıf ve Yordamlara İhtiyaç Duyarız?**

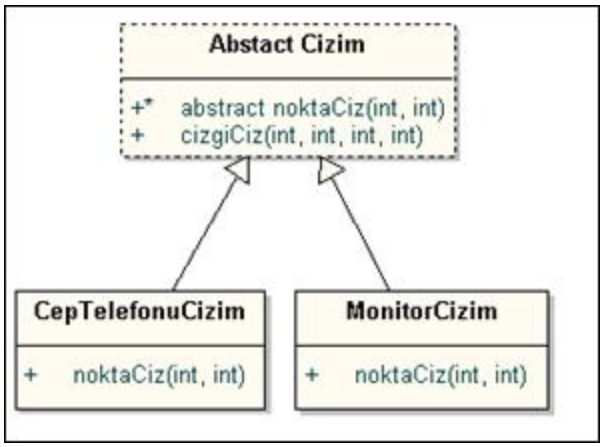
Örneğin hem cep telefonunun ekranına hem de monitörün ekranına çizgi çizdirmek istiyoruz fakat cep telefonu ekranının özellikleri ile monitör ekranının özelliklerinin birbirinden tamamen farklı olması, karşımızda büyük bir problemdir. Bu iki ekrana çizgi çizdirmek için değişik sınıflara ihtiyaç duyulacağı kesindir. Peki nasıl bir yazılım tasarımı yapılmalıdır. (yorum ekle)
Örnek: *CizimPrograma.java* (yorum ekle)

```
abstract class Cizim {
```

```
  // soyut yordam
  public abstract void noktaCiz(int x , int y) ;

  // soyut olmayan yordam
  public void cizgiCiz(int x1 , int y1 , int x2 , int y2)
{
      // noktaCiz(x,y); // yordamını kullanarak ekrana
cizgi ciz
  }
}

class CepTelefonuCizim extends Cizim {
  // iptal ediyor (override)
  public void noktaCiz(int x, int y) {
      // cep telefonu ekrani icin nokta ciz.....
  }
}

class MonitorCizim extends Cizim {
  // iptal ediyor (override)
  public void noktaCiz(int x, int y) {
      // Monitor ekrani icin nokta ciz.....
  }
}

public class CizimProgrami {
  public void baslat(int x1 , int y1 , int x2 , int y2) {
      // cep telefonunun ekranina cizgi cizmek icin
      Cizim c1 = new CepTelefonuCizim();
      c1.cizgiCiz(x1 , y1 , x2 , y2);
      // Monitor ekranina cizgi cizmek icin
      Cizim c2 = new MonitorCizim();
      c2.cizgiCiz(x1 , y1 , x2 , y2 );
 }
}
```

Cizim sınıfımızın içerisinde bulunan `cizgiCiz()` yordamı soyut (*abstract*) değildir fakat `noktaCiz()` yordamı soyuttur, neden? Sebebi, `cizgiCiz()` yordamının ekranlara çizgi çizmek için `noktaCiz()` yordamına ihtiyaç duymasından kaynaklanır.`cizgiCiz()` yordamının ihtiyaç duyduğu tek şey, ekran üzerinde tek bir noktanın nasıl çizileceğini bilmektir, bu bilgiler `cizgiCiz()` yordamına verildiği sürece sorun yaşanmayacaktır. Ekrana tek bir noktanın nasıl çizileceğini, *Cizim* sınıfından türemiş alt sınıflar tarafından verilmektedir. (yorum ekle)

Cizim sınıfından türemiş sınıflara dikkat edilirse (*CepTelefonuCizim* ve *MonitorCizim*), bu sınıfların içerisinde, ana sınıfa ait olan `noktaCiz()` yordamının iptal edilmiş (*override*) olduğunu görürüz. Bunun sebebi her bir ekrana (Monitörün ve Cep telefonu ekranı) ait nokta çiziminin farklı olmasından kaynaklanır. Yukarıdaki uygulamamıza ait sınıflar için UML diyagramı aşağıdaki gibidir. (yorum ekle)



Şekil-6.5. CizimPrograma.java uygulamasında kullanılan sınıflar

Yukarıdaki örneğimizden çıkartılacak olan ana fikir şöyledir; eğer bir işlem değişik verilere ihtiyaç duyup aynı işi yapıyorsa, bu işlem soyut (*abstract*) sınıfın içerisinde tanımlanmalıdır. (yorum ekle)

6.7. **Yapılandırıcılar İçerisindeki İlginç Durumlar**

Yapılandırıcılar içerisinde ne gibi ilginç durumlar olabilir ki diyebilirsiniz? Biraz sonra göstereceğimiz örnek içerisinde polimorfizm ve geç bağlamanın devreye girmesiyle olaylar biraz karışacaktır. Öncelikle yapılandırıcıların ne işe yaradıklarını bir tekrar edelim. (yorum ekle)
Bir nesne kullanıma geçmeden evvel bazı işlemler yapması gerekebilir, örneğin global olan alanlarına ilk değerlerinin verilmesi gerekebilir veya JDBC (ilerleyen bölümlerde inceleyeceğiz) bağlantısı ile veritabanı bağlanıp bazı işlemleri yerine getirmesi gerekebilir, örnekleri çoğaltmak mümkündür... Tüm bu işlemlerin yapılması için gereken yer yapılandırıcılardır. Buraya kadar sorun yoksa örneğimizi incelemeye başlayabiliriz. (yorum ekle)

Örnek: *Spor.java* (yorum ekle)

```
 abstract class Sporcu {
  public abstract void calis();
  public Sporcu() {    // yapılandırıcı   yordam
      System.out.println("calis() cagrilmadan evvel");
      calis(); // ! Dikkat !
      System.out.println("calis() cagrildiktan sonra");
  }
}

class Futbolcu extends Sporcu {
  int antraman_sayisi = 4 ;
  public void calis() {
      System.out.println("Futbolcu calis() " +
antraman_sayisi );
  }
  public Futbolcu() { // yapılandırıcı yordam
      System.out.println("Futbolcu yapilandirici" );
```

```
        calis();
    }
}

public class Spor {
    public static void main( String args[] ) {
        Futbolcu f = new Futbolcu();
        // Sporcu s = new Sporcu(); // ! Hata soyut sınıf !
    }
}
```

Soyut sınıflara ait yapılandırıcılar olabilir. 5. bölümde de incelediği üzere, sınıflara ait nesneleri oluştururken işin içerisinde bir de kalıtım (*inheritance*) özelliği girdiğinde olayların nasıl değiştiği incelemiştik. Bir sınıfa ait nesne oluşturulacaksa, önce bu sınıfın ana sınıfı var mı diye kontrol edilir; yani, bu sınıf türetilmiş bir sınıf mı kontrolü yapılır. Eğer bu sınıfın türetildiği ana bir sınıf var ise önce bu ana sınıfa ait nesne oluşturulur daha sonra sıra türeyen sınıfımıza ait nesnenin oluşturulmasına gelir. Yukarıdaki örneğimizde kalıtım kavramı kullanılmıştır. Ana sınıf *Sporcu* sınıfıdır, bu sınıftan türetilmiş olan ise *Futbolcu* sınıfıdır - Futbolcu **bir** Sporcudur. Biz *Futbolcu* sınıfına ait bir nesne oluşturmak istersek, bu olayın daha öncesinde *Sporcu* sınıfına ait bir nesnesinin oluşacağını açıktır. (yorum ekle)

Bu örneğimizdeki akıl karıştırıcı nokta, soyut bir sınıfa ait yapılandırıcı içerisinden soyut bir yordamın çağrılıyor olmasıdır. *Sporcu* sınıfının yapılandırıcısına dikkat ederseniz, bu yapılandırıcı içerisinden soyut bir yordam olan `calis()` yordamı çağrılmıştır. `calis()` yordamı hangi amaçla soyut yapılmış olabilir? Bu yordamın soyut yapılmasındaki tek amaç, alt sınıfların bu yordamı iptal etmelerini kesinleştirmek olabilir. Bu örneğimizdeki yanlış, soyut bir sınıfa ait yapılandırıcının içerisinden soyut bir yordamın çağrılmasıdır. Peki böyle bir yanlış yapıldığında nasıl sonuçlar oluşur? Uygulamamızın çıktısı aşağıdaki gibidir. (yorum ekle)

```
calis() cagrilmadan evvelFutbolcu calis() 0 --> dikkat calis() cagrildiktan
sonraFutbolcu yapilandiriciFutbolcu calis() 4
```

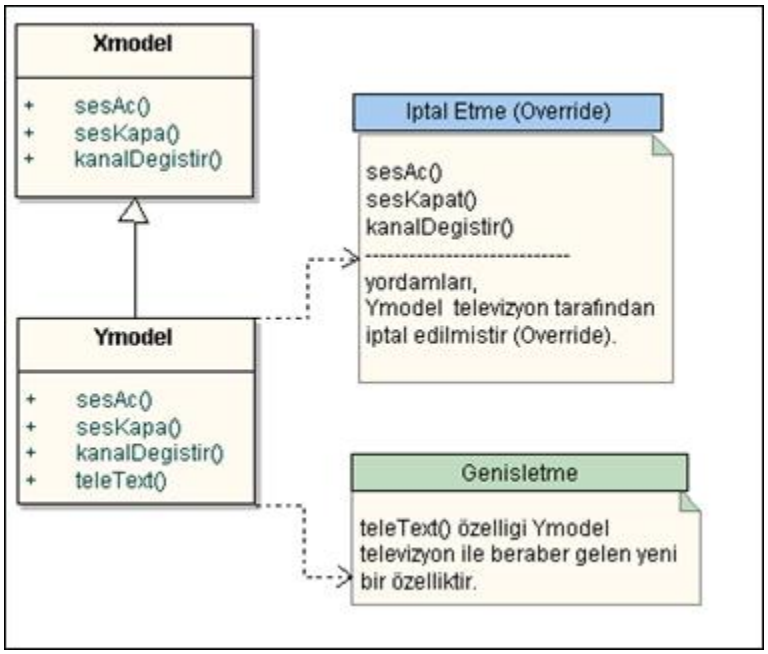
Uygulama çıktısındaki 2. satır çok ilginçtir. Oluşan olayları adım adım açıklarsak;

1. *Futbolcu* sınıfına ait bir nesne oluşturulmak istendi ama *Futbolcu* sınıfı *Sporcu* sınıfından türetildiği için, ilk önce **Sporcu** sınıfına ait yapılandırıcı cağrılacaktır. (yorum ekle)
2. *Sporcu* sınıfına ait yapılandırıcının içerisinde soyut olan `calis()` yordamı çağrıldı. Soyut olan `calis()` yordamı *Futbolcu* sınıfının içerisinde iptal edildiği (override) için, *Futbolcu* sınıfına ait olan `calis()` yordamı çağrılacaktır fakat temel(*primitive*) int tipinde olan `antreman_sayisi` alanına henüz ilk değeri atanmadığından, Java tarafından varsayılan değer (*default value*) olan `0` sayısı verilmiştir. Buradaki ilginç olan nokta *Futbolcu* sınıfının içerisindeki *calis()* yordamı, *Futbolcu*sınıfına ait yapılandırıcıdan bile önce çağrılmış olmasıdır (-ki bu istenmeyen bir durumdur). (yorum ekle)

3. Ana sınıf olan *Sporcu* sınıfına ait yapılandırıcının çalışması sona erdikten sonra türemiş sınıf olan *Futbolcu* nesnesine ait global alan olan `antreman_sayisi` alanının ilk değeri verilmiştir. (yorum ekle)
4. En son olarak *Futbolcu* sınıfına ait yapılandırıcı içerisindeki kodlar çalıştırılarak işlem sona erer. (yorum ekle)

6.8. **Kalıtım ve Yukarı Çevirim (Upcasting)**

Yukarı çevirim (upcasting) her zaman güvenlidir, sonuçta daha özellikli bir tipten daha genel bir tipe doğru çevirim gerçekleşmiştir. Örneğin elimizde iki tip televizyon bulunsun, biri Xmodel televizyon diğeri Xmodel'den türetilmiş ve daha yeni özelliklere sahip olan Ymodel televizyon. Bu ilişkiyi UML diyagramında gösterirsek. (yorum ekle)



Şekil-6.6. Kalıtım ve Yukarı Çevirim

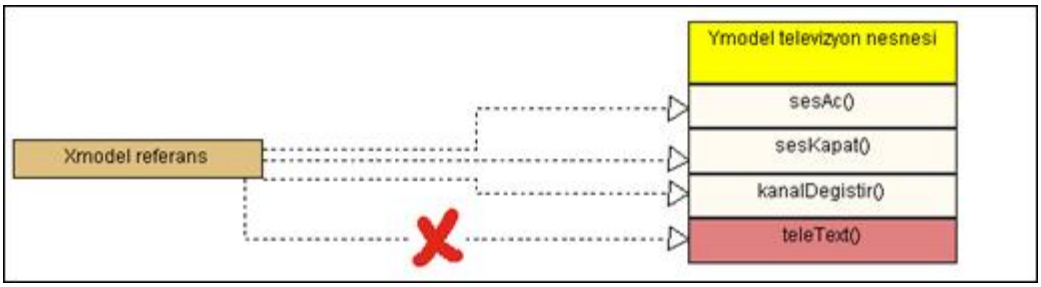
UML diyagramı Java uygulamasına dönüştürürse;

Örnek: *Televizyon.java* (yorum ekle)

```
class Xmodel {
 public void sesAc() {
     System.out.println("X model televizyon sesAc()");
 }
 public void sesKapa() {
     System.out.println("X model televizyon sesKapa()");
 }
 public void kanalDegistir() {
     System.out.println("X model televizyon
kanalDegistir()");
```

```
  }
}

class Ymodel extends Xmodel {
  public void sesAc() { // iptal etme (override)
    System.out.println("Y model televizyon sesAc()");
  }
  public void sesKapa() { // iptal etme (override)
     System.out.println("Y model televizyon sesKapa()");
  }
  public void kanalDegistir() { // iptal etme (override)
   System.out.println("Y model televizyon
kanalDegistir()");
  }

  public void teleText() {
      System.out.println("Y model televizyon teleText()");
  }
}

public class Televizyon {
  public static void main(String args[]) {

      // yukarı  cevirim ( upcasting )
      Xmodel x_model_kumanda = new Ymodel();
      x_model_kumanda.sesAc();
      x_model_kumanda.sesKapa();
      x_model_kumanda.kanalDegistir();

      //!! hata !!, bu kumandanın böyle bir düğmesi yok :)
      // x_model_kumanda.teleText() ;
  }
}
```

Yukarı çevirim (*upcasting*) olayında iki taraf vardır, bir tanesi *heap* alanında nesnenin kendisi diğer tarafta yığın (*stack*) alanında bulunan referans. Olaylara televizyon ve kumanda boyutunda bakarsak işin sırrı çözülmeye başlar. Elimizde *Xmodel*televizyon kumandası olduğunu düşünün ama kumanda *Ymodel* bir televizyonu gösterirsin (gösterebilir çünkü arada kalıtım ilişkisi vardır (*Ymodel* televizyon bir *Xmodel* televizyondur), o zaman karşımızda duran *Ymodel* televizyonun `teleText()` özelliği olmasına rağmen bunu kullanamayız çünkü *Xmodel* bir televizyon kumandası, *Xmodel* televizyon için tasarlandığından, bu kumandanın üzerinde `teleText()` düğmesi olmayacaktır. Anlattıklar şekil üzerinde gösterilirse: (yorum ekle)



Şekil-6.7. Kalıtım ve Yukarı Çevirim

6.9. **Aşağıya Çevirim** (***Downcasting)***

Aşağıya çevirim (*downcasting*), yukarı çevirim (*upcasting*) işleminin tam tersidir. Aşağıya çevirim (*downcasting*), daha genel bir tipten, daha özellikli bir tipe doğru geçiş demektir ve tehlikelidir. Tehlikelidir çünkü çevrilmeye çalışılan daha özellikli tipe doğru çevirim esnasında sorun çıkma riski yüksektir. Java programlama dilinde aşağıya çevirim (*downcasting*) yaparken, hangi tipe doğru çevirim yapılacağı açık olarak belirtmelidir. Fakat yukarı çevirim (*upcasting*) işleminde böyle bir belirteç koyma zorunluluğu yoktur çünkü oradaki olay daha özellikli bir tipten daha genel bir tipe doğru çevirimdir; yani, güvenlidir. Anlattıkları örnek üzerinde gösterirsek: (yorum ekle)

Örnek: *Televizyon2.java* (yorum ekle)

```
 class Xmodel {
  public void sesAc() {
      System.out.println("X model televizyon sesAc()");
  }
  public void sesKapa() {
      System.out.println("X model televizyon sesKapa()");
  }
  public void kanalDegistir() {
      System.out.println("X model televizyon"
              +"kanalDegistir()");
  }
}

class Ymodel extends Xmodel {
  public void sesAc() { // iptal etme (override)
      System.out.println("Y model televizyon sesAc()");
  }
  public void sesKapa() { // iptal etme (override)
      System.out.println("Y model televizyon sesKapa()");
  }
  public void kanalDegistir() { // iptal etme (override)
      System.out.println("Y model televizyon "
         +" kanalDegistir() ");
  }
  public void teleText() {
      System.out.println("Y model televizyon teleText()");
```

```
  }
}

public class Televizyon2 {
  public static void main(String args[]) {

      Object[] ob = new Object[2] ;
      ob[0] = new Xmodel() ; // yukari cevirim (upcasting)
      ob[1] = new Ymodel() ; // yukari cevirim (upcasting)
      for (int i = 0 ; i < ob.length ; i++) {
        // asagiya cevirim (Downcasting)
        Xmodel x_model_kumanda = (Xmodel) ob[i] ;
        x_model_kumanda.sesAc();
        x_model_kumanda.sesKapa();
        x_model_kumanda.kanalDegistir();
        // x_model_kumanda.teleText(); // bu kumanda da
        // böyle bir  düğme yok

       System.out.println("----------------------------");
     }
  }
}
```

Yukarıdaki uygulamanın çıktısı aşağıdaki gibidir;

```
X model televizyon sesAc()
X model televizyon sesKapa()
X model televizyon kanalDegistir()
----------------------------------------------
Y model televizyon sesAc()
Y model televizyon sesKapa()
Y model televizyon kanalDegistir()
----------------------------------------------
```

Bu örneğimizde, *Xmodel* ve *Ymodel* nesnelerimiz, *Object* sınıfı tipindeki dizinin içerisine atılmaktadır. Her sınıf, *Object* sınıfından türetildiği göre, *Object* sınıfı tipindeki bir dizisinin içerisine *Xmodel* ve *Ymodel* nesnelerini rahatlıkla atabiliriz buradaki olay yukarı doğru çevirimdir (*upcasting*). Artık elimizde *Object* sınıfı tipinde bir dizisi var. Bunun içerisindeki elemanları tekrardan eski hallerine sokmak için aşağıya doğru çevirim (*downcasting*) özelliğini kullanmak gereklidir. Daha evvelden de ifade edildiği gibi aşağıya doğru çevirim (*downcasting*) yapılacaksa, bunun hangi tipe olacağı açık olarak belirtilmelidir. (yorum ekle)
`for` ifadesinin içerisine dikkat edilirse, *Object* tipindeki dizisinin içerisindeki elemanları *Xmodel* nesnesine dönüştürmek için aşağıdaki ifade kullanılmıştır. (yorum ekle)

Gösterim-6.5:

```
// asagiya dogru cevirim (Downcasting)
Xmodel x_model_kumanda = (Xmodel) ob[i] ;
```

Yukarıdaki ifade sayesinde *Object* sınıfı tipinde olan dizimizin içerisindeki elemanları, *Xmodel* nesnesine çevirmiş bulunmaktayız. Aslında bu örneğimizde zarardayız, niye derseniz hemen açıklayalım. Sonuçta bizim iki adet hem *Xmodel* hem de *Ymodel*nesnelerimiz vardı. Biz bunları *Object* sınıfı tipine çevirerek yani yukarı çevirim yaparak *Object* sınıfı tipindeki dizisinin içerisine yerleştirdik, buraya kadar sorun yok. Fakat *Object* dizisinin içerisinden elemanlarımızı çekerken, bu elemanlarımızın hepsini*Xmodel* nesnesine çevirdik yani aşağıya çevirim yapmış olduk. Biz böyle yapınca *Ymodel* nesnemiz yok olmuş gibi oldu. (yorum ekle)

Yukarı doğru çevirim (*upcasting*) yaparken, asıl nesnelerimiz değerlerinden birşey kaybetmezler. Örneğin bu uygulamamızda bir *Object* sınıfı tipindeki dizimizin içerisine *Xmodel* ve *Ymodel* nesnelerini atabildik (bunun sebebinin kalıtımdır). *Xmodel* ve*Ymodel* nesnelerini, *Object* nesnelerine çevirerek, bu nesnelerimizin asıl tiplerini değiştirmeyiz, sadece *Object* dizisinin içerisine atma izni elde ederiz; yani, *Object* sınıfı tipindeki dizimizin içerisindeki nesnelerimizin orijinal halleri hala *Xmodel* ve *Ymodel*tipindedir. (yorum ekle)

Java çalışma anında (*run-time*) nesnelerin tiplerini kontrol eder.Eğer bu işlemlerde bir uyumsuzluk varsa bunu hemen kullanıcıya *ClassCastException* istisnası fırlatarak bildirir. Nesne tiplerinin çalışma anından (*run-time*) tanımlanması *(RTTI : Run Time Type Identification)*, kodu yazan kişi açısından büyük faydalar içerir. Bu açıklamalardan yola çıkılarak, yukarıdaki uygulama değiştirilirse. (yorum ekle)

Örnek: *Televizyon3.java* (yorum ekle)

```
class Xmodel {
  public void sesAc() {
      System.out.println("X model televizyon sesAc()");
  }

  public void sesKapa() {
      System.out.println("X model televizyon sesKapa()");
  }

  public void kanalDegistir() {
      System.out.println("X model televizyon"
        +  " kanalDegistir()");
  }
}

class Ymodel extends Xmodel {
  public void sesAc() { // iptal ediyor (override)
      System.out.println("Y model televizyon sesAc()");
  }
```

```
  public void sesKapa() {   // iptal ediyor (override)
      System.out.println("Y model televizyon sesKapa()");
  }

  public void kanalDegistir() { // iptal ediyor (override)
     System.out.println("Y model televizyon "
         +"kanalDegistir() ");
  }

  public void teleText() {
      System.out.println("Y model televizyon teleText()");
  }
}

public class Televizyon3 {
  public static void main(String args[]) {

      Object[] ob = new Object[2] ;
      ob[0] = new Xmodel() ;
      ob[1] = new Ymodel() ;

      for (int i = 0 ; i < ob.length ; i++) {

        Object o = ob[i] ;
        if (o instanceof Ymodel) { // RTTI

               Ymodel y_model_kumanda = (Ymodel) o ;
               y_model_kumanda.sesAc();
               y_model_kumanda.sesKapa();
               y_model_kumanda.kanalDegistir();
               y_model_kumanda.teleText() ;
        } else if (o instanceof Xmodel) { // RTTI
               Xmodel x_model_kumanda = (Xmodel) o;
               // artik guvenli
               x_model_kumanda.sesAc();
               x_model_kumanda.sesKapa();
               x_model_kumanda.kanalDegistir();

        }
      }
  }
}
```

Object sınıfı tipindeki dizi içerisinde bulunan elemanların hepsinin *Object* sınıfı tipinde olma zorunluluğu olduğunu 3. bölümdeki diziler başlığında incelemiştik. *Xmodel* ve *Ymodel* televizyonları yukarı doğru çevirim özelliği sayesinde *Object* sınıfı tipine çevirerek, *Object* sınıfı tipindeki dizi içerisine atabildik. Peki *Xmodel* ve *Ymodel* nesnelerimizin özelliklerini sonsuza kadar geri alamayacak mıyız? Geri almanın yolu aşağıya çevirimdir (*downcasting*). (yorum ekle)

Aşağıya çevirimin (*downcasting*) tehlikeli olduğunu biliyoruz. Eğer yanlış bir tipe doğru çevirim yaparsak, çalışma anında Java tarafından *ClassCastException* istisnası (ilerleyen bölümlerde inceleyeceğiz) ile durduruluruz. Çalışma anında (run-time) yanlış bir tipe çevirimden korkuyorsak `instanceof` anahtar kelimesini kulanmamız gerekir. Yukarıdaki örneğimizde (*Televizyon3.java*) `instanceof` anahtar kelimesi sayesinde çalışma anında, *Object* sınıfı tipindeki dizi içerisindeki elemanların asıl tiplerini kontrol ederek aşağıya doğru çevirim yapma imkanına sahip oluruz. Böylece hata oluşma riskini minimuma indirgeriz. Uygulamamızın çıktısı aşağıdaki gibidir; (yorum ekle)

```
X model televizyon sesAc()
X model televizyon sesKapa()
X model televizyon kanalDegistir()
Y model televizyon sesAc()
Y model televizyon sesKapa()
Y model televizyon kanalDegistir()
Y model televizyon teleText()
```