BÖLÜM 7

Altuğ B. Altıntaş

ARAYÜZLER VE DAHİLİ SINIFLAR

(Interface and Inner Classes)

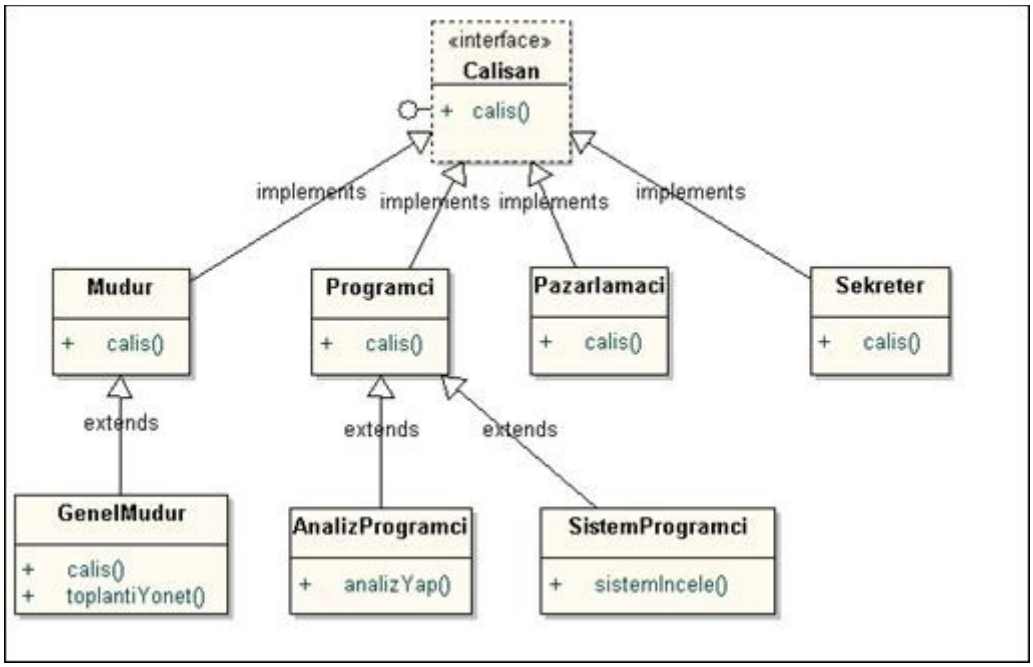
Diğer programlama dillerinde olan çoklu kalıtım (*multiple inheritance*) özelliği Java programlama dilinde yoktur. Java programlama dilinde çoklu kalıtım desteğinden faydalanmak için arayüz (*interface*) ve dahili sınıflar (inner classes) kullanılır. (yorum ekle)

7.1. Arayüz (Interface)

Arayüzler, soyut (abstract) sınıfların bir üst modeli gibi düşünelebilir, soyut sınıfların içerisinde hem iş yapan hem de hiçbir iş yapmayan sadece birleştirici rol üstlenen gövdesiz yordamlar (soyut yordamlar-*abstract methods*) vardı. Bu birleştirici rol oynayan yordamlar, soyut sınıfdan (*abstract class*) türetilmiş alt sınıfların içerisinde iptal edilmeleri (*override*) gerektiğini geçen bölümde incelenmişti. Arayüzlerin içerisinde ise iş yapan herhangi bir yordam (*method*) bulunamaz; arayüzün içerisinde tamamen gövdesiz yordamlar (soyut yordamlar) bulunur. Bu açıdan bakılacak olursak, arayüzler, birleştirici bir rol oynamaları için tasarlanmıştır. Önemli bir noktayı hemen belirtelim; arayüzlere ait gövdesiz (*soyut*) yordamlar otomatik olarak `public` erişim belirleyicisine sahip olurlar ve sizin bunu değiştirme imkanınız yoktur. Aynı şekilde arayüzlere ait global alanlarda otomatik `public` erişim belirleyicisine sahip olurlar ek olarak, bu alanlar yine otomatik olarak `final` ve `statik` özelliği içerirler ve sizin bunlara yine müdahale etme imkanınız yoktur. (yorum ekle)

7.1.1. Birleştiricilik

Bölüm-6'da verilen *BüyükIsYeri.java* örneğini, arayüzleri kullanarak baştan yazmadan önce, yeni UML diyagramını inceleyelim; (yorum ekle)



Şekil-7.1. Birleştiricilik

UML diyagramında görüldüğü üzere, *Calisan* arayüzü (*interface*), birleştirici bir rol oynamaktadır. *Calisan* arayüzünde tanımlanan ve soyut (gövdesiz) `calis()` yordamı (*method*), bu arayüze erişen tüm sınıfların içerisinde iptal edilmek zorundadır (*override*). UML diyagramımızı Java uygulamasına dönüştürülürse; (yorum ekle)

Örnek: *BuyukIsYeri.java* (yorum ekle)

```
interface Calisan { // arayuz
  public void calis() ;
}

class Mudur implements Calisan {
  public void calis() { // iptal etti (override)
       System.out.println("Mudur Calisiyor");
  }
}

class GenelMudur extends Mudur {
  public void calis() { // iptal etti (override)
       System.out.println("GenelMudur Calisiyor");
  }
  public void toplantiYonet() {
       System.out.println("GenelMudur toplanti yonetiyor");
  }
}

class Programci implements Calisan {
  public void calis() { // iptal etti (override)
       System.out.println("Programci Calisiyor");
  }
}

class AnalizProgramci extends Programci {
  public void analizYap() {
       System.out.println("Analiz Yapiliyor");
  }
}
```

```
class SistemProgramci extends Programci {
  public void sistemIncele() {
       System.out.println("Sistem Inceleniyor");
  }
}

class Pazarlamaci implements Calisan {
  public void calis() { // iptal etti (override)
       System.out.println("Pazarlamaci Calisiyor");
  }
}

class Sekreter implements Calisan {
  public void calis() { // iptal etti (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[6];
        // c[0]=new Calisan(); ! Hata ! arayüz olusturulamaz
        c[0]=new Programci();      // yukari cevirim (upcasting)
        c[1]=new Pazarlamaci();    //yukari cevirim (upcasting)
        c[2]=new Mudur();          //yukari cevirim (upcasting)
        c[3]=new GenelMudur();     //yukari cevirim (upcasting)
        c[4]=new AnalizProgramci(); //yukari cevirim  (upcasting)
        c[5]=new SistemProgramci(); //yukari cevirim  (upcasting)
        mesaiBasla(c);
  }
}
```

Yukarıdaki örneğimiz ilk etapta çekici gelmeyebilir, "Bunun aynısı soyut sınıflarla (*abstract class*) zaten yapılabiliyordu. Arayüzleri neden kullanayım ki.... " diyebilirsiniz. Yukarıdaki örnekte arayüzlerin nasıl kullanıldığı incelenmiştir; arayüzlerin sağladığı tüm faydalar birazdan daha detaylı bir şekilde incelenecektir. (yorum ekle)

Arayüzlerin devreye sokulmasını biraz daha yakından bakılırsa.

Gösterim-7.1:

```
class Mudur implements Calisan {
  public void calis() { // iptal etti (override)
       System.out.println("Mudur Calisiyor");
  }
}
```

Olaylara *Mudur* sınıfının bakış açısından bakılsın. Bu sınıf *Calisan* arayüzünün gövdesiz yordamlarını iptal etmek (override) istiyorsa, *Calisan* arayüzüne ulaşması gerekir. Bu ulaşım `implements` anahtar kelimesi ile gerçekleşir. *Mudur* sınıfı bir kere *Calisan* arayüzüne ulaştımı, buradaki gövdesiz yordamları (soyut yordamları) kesin olarak iptal etmek (override) zorundadır. Uygulamanın çıktısı aşağıdaki gibidir; (yorum ekle)

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

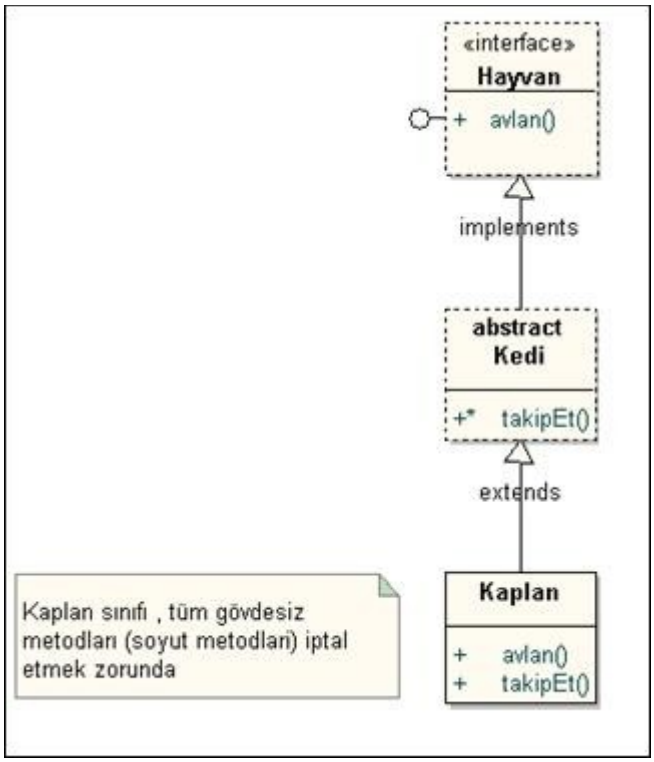
7.1.2. **Arayüz (Interface) ve Soyut Sınıflar (Abstract Classes)**

Eğer bir sınıf (soyut sınıflar dahil) bir arayüze (*interface*) ulaşmak istiyorsa, bunu `implements` anahtar kelimesi ile gerçekleştirebileceğini belirtmiştik. Ayrıca eğer bir sınıf bir kere arayüze ulaştımı artık onun tüm gövdesiz yordamlarını (soyut yordamlar) kesin olarak iptal etmesi (override) gerektiğini de belirttik. Peki eğer soyut bir sınıf (*abstract class*) bir arayüze ulaşırsa, arayüze ait gövdesiz yordamları kesin olarak, kendi içerisinde iptal etmeli mi? Bir örnek üzerinde incelersek; (yorum ekle)

Örnek: *Karisim.java* (yorum ekle)

```
interface Hayvan {
  public void avlan() ;
}

abstract class Kedi implements Hayvan {
}
```

Yukarıdaki örneğimizi derleyebilir (*compile*) miyiz? Derlense bile çalışma anında (*run-time*) hata oluşturur mu? Aslında olaylara kavramsal olarak bakıldığında çözüm yakalanmış olur. Soyut sınıfların amaçlarından biri aynı arayüz özelliğinde olduğu gibi birleştirici bir rol oynamaktır. Daha açık bir ifade kullanırsak, hem arayüzler olsun hem de soyut sınıflar olsun, bunların amaçları kendilerine ulaşan normal sınıflara, kendilerine ait olan gövdesiz yordamları iptal ettirmektir (*override*). O zaman yukarıdaki örnekte soyut olan *Kedi* sınıfı, *Hayvan* arayüzüne (*interface*) ait gövdesiz (soyut) `avlan()` yordamını iptal etmek zorunda değildir. Daha iyi anlaşılması açısından yukarıdaki örneği biraz daha geliştirelim ama öncesinde UML diyagramını çıkartalım; (yorum ekle)



Şekil-7.2. Arayüzler ve Soyut Sınıflar

UML diyagramından görüleceği üzere, *Kaplan* sınıfı, `avlan()` ve `takipEt()` yordamlarını (gövdesiz-soyut yordamlarını) iptal etmek zorundadır. UML diyagramını Java uygulamasına dönüştürülürse; (yorum ekle)

Örnek: *Karisim2.java* (yorum ekle)

```
interface Hayvan {
  public void avlan() ;
}

abstract class Kedi implements Hayvan {
  public abstract void takipEt() ;
}

class Kaplan extends Kedi {
  public void avlan() { // iptal etti (override)
       System.out.println("Kaplan avlaniyor...");
  }

  public void takipEt() { // iptal etti (override)
       System.out.println("Kaplan takip ediyor...");
  }
}
```

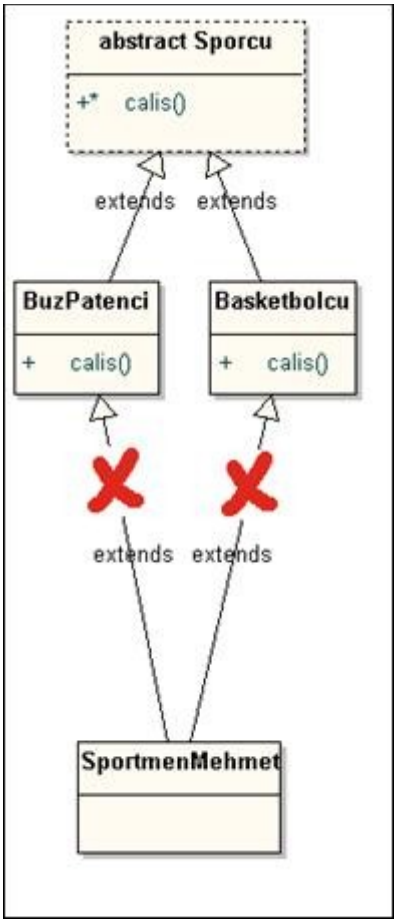
Soyut (*abstract*) olan *Kedi* sınıfının içerisinde, herhangi bir gövdesiz yordam (soyut yordam) iptal edilmemiştir (*override*). İptal edilme işlemlerinin gerçekleştiği tek yer *Kaplan* sınıfının içerisidir. Soru: *Kaplan* sınıfı *Hayvan* arayüzünde (interface) tanımlanmış soyut olan (gövdesiz) `avlan()` yordamını iptal etmek (override) zorunda mı? Cevap: Evet, çünkü *Kaplan* sınıfı *Kedi* sınıfından türetilmiştir. *Kedi* sınıfı ise

Hayvan arayüzüne ulaşmaktadır. Bu sebepten dolayı *Kaplan* sınıfının içerisinde `avlan()` yordamı iptal edilmelidir. (yorum ekle)

En baştaki sorumuzun cevabı olarak, Karisim.java örneğimiz rahatlıkla derlenebilir (*compile*) ve çalışma anında (*run-time*) herhangi bir çalışma-anı istisnasına (*runtime-exception*) sebebiyet vermez. (Not: İstisnaları (Exception) 8. bölümde detaylı bir şekilde anlatılmaktadır.) (yorum ekle)

7.1.3. **Arayüz (Interface) İle Çoklu Kalıtım (Multiple Inheritance)**

İlk önce çoklu kalıtımın (*multiple inheritance*) niye tehlikeli ve Java programlama dili tarafından kabul görmediğini inceleyelim. Örneğin *Sporcu* soyut sınıfından türetilmiş iki adet sınıfımız bulunsun, *BuzPatenci* ve *Basketbolcu* sınıfları. Bu iki sınıftan türetilen yeni bir sınıf olamaz mı? Örneğin *SportmenMehmet* sınıfı; yani, *SportmenMehmet* sınıfı aynı anda hem *BuzPatenci*, hem de *Basketbolcu* sınıfından türetilebilir mi? Java programlama dilinde türetilemez. Bunun sebeplerini incelemeden evvel, hatalı yaklaşımı UML diyagramında ifade edilirse; (yorum ekle)



Şekil-7.3. Çoklu Kalıtımın (Multiple Inheritance) Sakıncaları

Java programlama dili niye çoklu kalıtımı bu şekilde desteklemez? UML diyagramını, hatalı bir Java uygulamasına dönüştürülürse; (yorum ekle)

Örnek: *Spor.java* (yorum ekle)

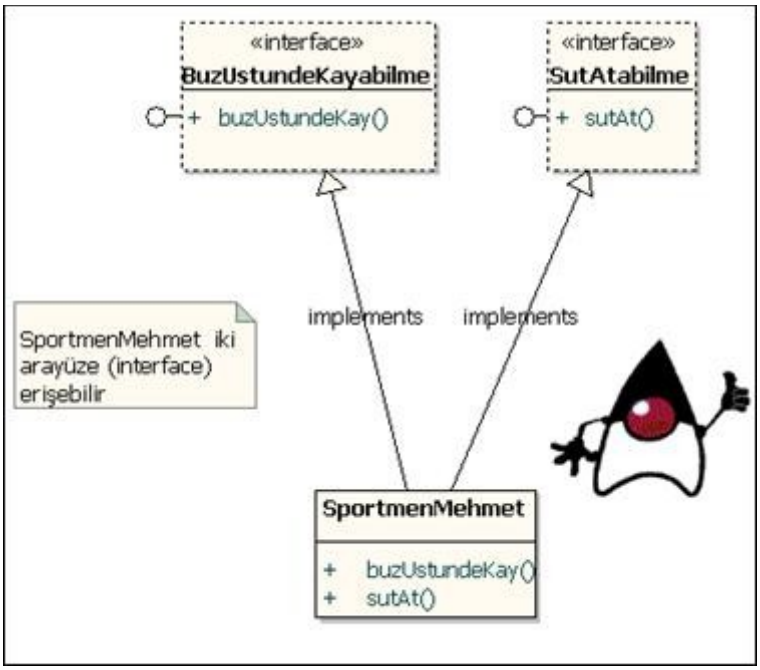
```
abstract class Sporcu {
  public abstract void calis();
}

class BuzPatenci extends Sporcu {
  public void calis() {
       System.out.println("BuzPatenci calisiyor....") ;
  }
}

class Basketbolcu extends Sporcu {
  public void calis() {
       System.out.println("Basketbolcu calisiyor....") ;
  }
}

/*
Bu ornegimiz derleme aninda hata alicaktir.
Java, coklu kalitimi desteklemez
*/
class SportmenMehmet extends BuzPatenci, Basketbolcu {
}
```

Spor.java derleme anında hata alacaktır. Bu ufak ayrıntıyı belirttikten sonra, kaldığımız yerden devam edelim. Java'nın niye çoklu kalıtımı (*multiple inheritance*) desteklemediğini anlamak için aşağıdaki gösterim incelenmelidir. (yorum ekle)

Gösterim-7.2:

```
Sporcu s = new SportmenMehmet(); // yukari dogru cevirim
s.calis(); // ??
```

Herhangi bir yerden, yukarıdaki gösterimde belirtildiği gibi bir ifade yazılmış olsa, sonuç nasıl olurdu? *Sporcu* tipinde olan referans, *SportmenMehmet* nesnesine bağlanmıştır (bağlanabilir çünkü arada kalıtım ilişkisi vardır). Fakat burada `s.calis()` ifadesi yazılırsa, hangi nesnenin `calis()` yordamı çağrılacaktır? *BuzPatenci* nesnesinin mi? Yoksa *Basketbolcu* nesnesinin mi? Sonuçta, `calis()` yordamı, *BuzPatenci* ve *Basketbolcu* sınıflarının içerisinde iptal edilmiştir. Bu sorunun cevabı yoktur. Fakat çoklu kalıtımın bu zararlarından arıtılmış versiyonunu yani arayüzleri (interface) ve dahili sınıflar (inner classes) kullanarak, diğer dillerinde bulunan çoklu kalıtım desteğini, Java programlama dilinde de bulmak mümkündür. 'Peki ama nasıl?' diyenler için hemen örneğimizi verelim. Yukarıdaki örneğimizi Java programlama diline uygun bir şekilde baştan yazalım ama öncesinde her zaman ki gibi işe UML diyagramını çizmekle başlayalım; (yorum ekle)



Şekil-7.4.Arayüzlerin kullanılışı

*SportmenMehmet***,** belki aynı anda hem *BuzPatenci* hemde *Basketbolcu* olamaz ama onlara ait özelliklere sahip olabilir. Örneğin *BuzPatenci* gibi buz üzerinde kayabilir ve *Basketbolcu* gibi şut atabilir. Yukarıdaki UML diyagramı Java uygulamasına dönüştürülürse. (yorum ekle)

Örnek: *Spor2.java* (yorum ekle)

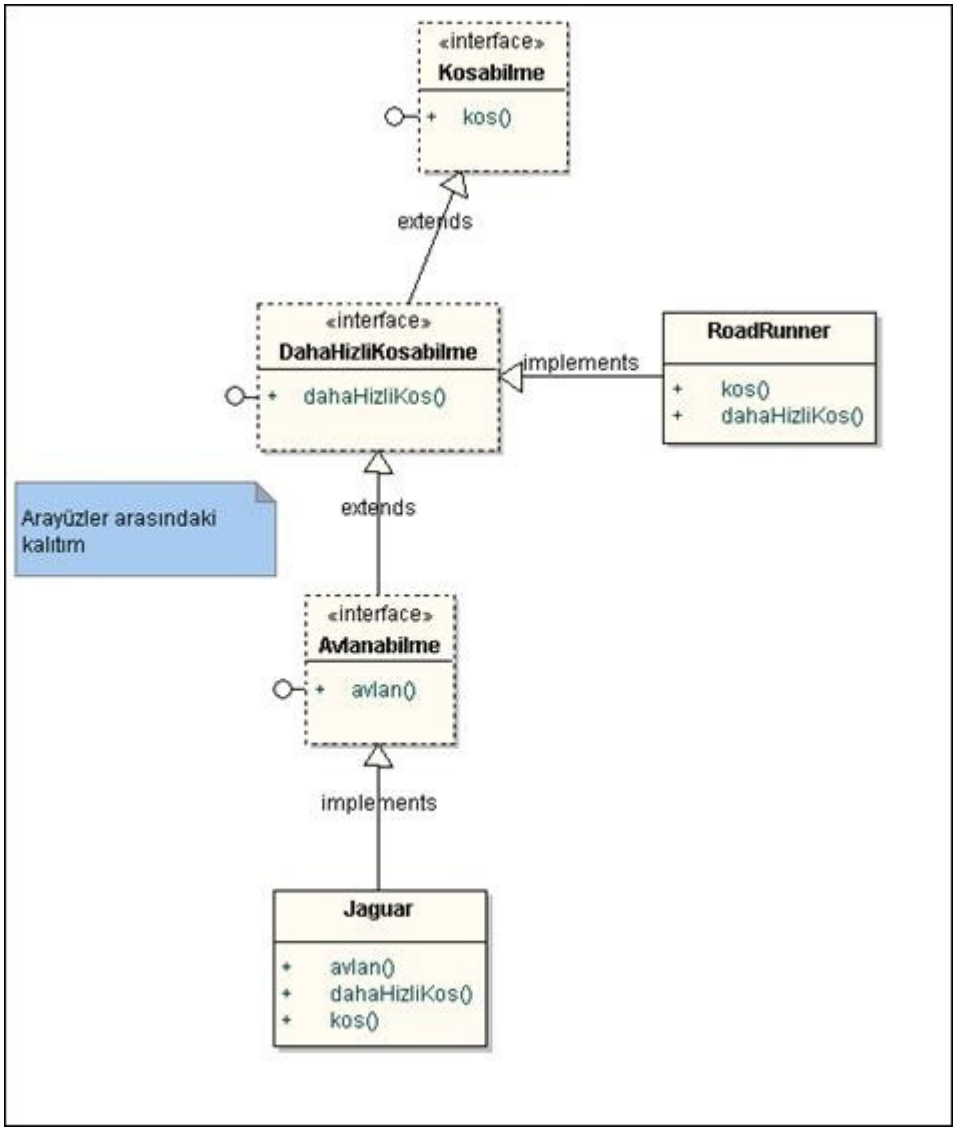
```
interface BuzUstundeKayabilme {
  public void buzUstundeKay();
}

interface SutAtabilme {
  public void sutAt();
}

class SportmenMehmet implements BuzUstundeKayabilme, SutAtabilme {
  public void buzUstundeKay() {
       System.out.println("SportmenMehmet buz ustunde kayiyor");
  }
  public void sutAt() {
       System.out.println("SportmenMehmet sut atiyor");
 }
}
```

Bu örneğimizde *SportmenMehmet*, *BuzUstundeKayabilme* ve *SutAtabilme* özelliklerine sahip olmuştur. Arayüzler içerisindeki (*BuzUstundeKayabilme*,*SutAtabilme*) gövdesiz (soyut) yordamları (`buzUstundeKay()`, `sutAt()`), bu arayüzlere erişen sınıf tarafından kesinlikle iptal edilmelidir (*overrride*). Eğer iptal edilmez ise, derleme anında (*compile-time*) Java tarafından gerekli hata mesajı verilir. (yorum ekle)

Örneğimizden anlaşılabileceği üzere arayüz (interface) ile soyut (abstract) sınıf arasında büyük fark vardır. En başta kavramsal olarak bir fark vardır. Bu kavramsal fark nedir derseniz hemen açıklayalım; Soyut bir sınıftan türetilme yapıldığı zaman, türetilen sınıf ile soyut sınıf arasında mantıksal bir ilişki olması gerekirdi, örnek vermek gerekirse "Yarasa **bir** Hayvandır" gibi veya "Müdür **bir** Çalışandır" gibi....Geçen bölümlerde incelediğimiz **bir** ilişkisi. Fakat arayüzler ile bunlara erişen sınıflar arasında kalıtımsal bir ilişki bulunmayabilir. (yorum ekle)

7.1.4. **Arayüzlerin Kalıtım (İnheritance) Yoluyla Genişletilmesi**

Bir arayüz başka bir arayüzden türetilerek yeni özelliklere sahip olabilir; böylece arayüzler kalıtım yoluyla genişletilmiş olur. Olayları daha iyi anlayabilmek için önce UML diyagramını çizip sonrada Java uygulamasını yazalım. (yorum ekle)



Şekil-7.5. Arayüzlerin Kalıtım Yoluyla Genişletilmesi

Avlanabilme arayüzü, *DahaHizliKosabilme* arayüzünden türemiştir. *DahaHizliKosabilme* arayüzüde *Kosabilme* arayüzünde türemiştir. Yukarıdaki UML diyagramımızı Java uygulamasına dönüştürelim; (yorum ekle)

Örnek: *Jaguar.java* (yorum ekle)

```
interface Kosabilme {
  public void kos();
}

interface DahaHizliKosabilme extends Kosabilme {
  public void dahaHizliKos();
}

interface Avlanabilme extends DahaHizliKosabilme {
  public void avlan();
}

class RoadRunner implements DahaHizliKosabilme {
  public void kos() {
       System.out.println("RoadRunner kosuyor, bip ");
  }

  public void dahaHizliKos() {
       System.out.println("RoadRunner kosuyor, bip bippp");
  }
}

public class Jaguar implements Avlanabilme {
  public void avlan() {
       System.out.println("Juguar avlaniyor");
  }

  public void dahaHizliKos() {
       System.out.println("Juguar daha hizli kos");
  }

  public void kos() {
       System.out.println("Juguar Kosuyor");
  }
}
```

Jaguar sınıfı *Avlanabilme* arayüzüne ulaşarak, `avlan()`, `dahaHizliKos()`, `kos()` yordamlarını iptal etmek (override) zorunda bırakılmıştır. Bunun sebebi *Avlanabilme* arayüzünün *DahaHizliKosabilme* arayüzünden, *DahaHizliKosabilme* arayüzününde *Kosabilme* arayüzünden türemiş olmasıdır. (yorum ekle)

RoadRunner sınıfı ise sadece *DahaHizliKosabilme* arayüzüne erişerek, `kos()` ve `dahaHizliKos()` gövdesiz (soyut) yordamlarını iptal etmek (*override*) zorunda bırakılmıştır. Yine bunun sebebi *DahaHizliKosabilme* arayüzünün *Kosabilme* arayüzünden türemiş olmasıdır. (yorum ekle)

Arayüzlere Özel

Şimdi birazdan inceleyeceğiz olay sadece arayüzler söz konusu olduğunda yapılabilir. İlk olayımız açıklayalım; Bir arayüz (*interface*) birden çok arayüzden türetilebilir. (yorum ekle)

Gösterim-7.3:

```
interface C {
  //..
}

interface B {
  //..
}

interface A extends B, C {
  //..
}
```

Yukarıdaki gösterimde, *A* arayüzü, birbirlerinden bağımsız iki arayüzden türetilmiş oldu. İkinci olay ise daha evvelde incelenmişti (*bkz:Spor2.java*) ama bir kere daha üzerine basa basa durmakta fayda var; bir sınıf birden fazla arayüze rahatlıkla erişebilir. (yorum ekle)

7.1.5. **Çakışmalar**

Arayüzlerin içerisinde dönüş tipleri haricinde her şeyleri aynı olan gövdesiz (soyut) yordamlar varsa, bu durum beklenmedik sorunlara yol açabilir. Yazılanları Java uygulaması üzerinde gösterilirse; (yorum ekle)

Örnek: *Cakisma.java* (yorum ekle)

```
interface A1 {
  public void hesapla();
}

interface A2 {
  public void hesapla(int d);
}

interface A3 {
  public int hesapla();
}

class S1 implements A1,A2 { // sorunsuz
  public void hesapla() { //adas yordamlar(overloaded)
       System.out.println("S1.hesapla");
  }
  public void hesapla(int d) { //adas yordamlar(overloaded)
       System.out.println("S1.hesapla " + d );
  }
}

/*
! Hatali !, adas yordamlar (overloaded)
donus tiplerine gore ayirt edilemez
*/

class S2 implements A1,A3 {
  public void hesapla() {
       System.out.println("S2.hesapla");
  }

  /* ! Hata !
  public int hesapla() {
       System.out.println("S2.hesapla");
       return 123;
  }
  */
}
```

Cakisma.java örneğini derlenirse (*compile*), aşağıdaki hata mesajı ile karşılaşılır: (yorum ekle)

```
Cakisma.java:27: hesapla() in S2 cannot implement hesapla() in A3;
attempting to
  use incompatible return type
found : void
required: int
class S2 implements A1,A3 {
^
1 error
```

Bu hatanın oluşma sebebi, *A1* ve *A3* arayüzlerinin içerisindeki gövdesiz (soyut) yordamlarından kaynaklanır. Bu yordamların isimleri ve parametreleri aynıdır ama dönüş tipleri farklıdır. (yorum ekle)

Gösterim-7.4:

```
public void hesapla(); // A1 arayüzüne ait
public int hesapla(); // A3 arayüzüne ait
```

Bölüm-3'de incelendiği üzere iki yordamın adaş yordam (*overloaded method*) olabilmesi için bu yordamların parametrelerinde kesin bir farklılık olması gerekirdi. İki yordamın dönüş tipleri dışında herşeyleri aynıysa bunlar adaş yordam olamazlar. Olamamalarının sebebi, Java'nın bu yordamları dönüş tiplerine göre ayırt edememesinden kaynaklanır. (yorum ekle)

7.1.6. Arayüzün (Interface) İçerisinde Alan Tanımlama

Arayüzlerin içerisinde gövdesiz (soyut) yordamların dışında alanlarda bulunabilir. Bu alanlar uygulamalarda sabit olarak kullanabilir. Çünkü arayüzün içerisinde tanımlanan bir alan (ilkel tipte veya sınıf tipinde olsun) otomatik olarak hem `public` erişim belirleyicisine hemde `final` ve `static` özelliğine sahip olur. (yorum ekle)

Örnek: *AyBul.java* (yorum ekle)

```
interface Aylar {
  int
  OCAK = 1, SUBAT = 2, MART = 3,
  NISAN = 4, MAYIS = 5, HAZIRAN = 6, TEMMUZ = 7,
  AGUSTOS = 8, EYLUL = 9, EKIM = 10,
  KASIM = 11, ARALIK = 12;
}

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

       int ay = (int)(Math.random()*13) ;
       System.out.println("Gelen ay = " + ay);
       switch ( ay ) {
         case Aylar.OCAK : System.out.println("Ocak");break;
         case Aylar.SUBAT : System.out.println("Subat");break;
         case Aylar.MART : System.out.println("Mart");break;
         case Aylar.NISAN : System.out.println("Nisan");break;
         case Aylar.MAYIS : System.out.println("Mayis");break;
         case Aylar.HAZIRAN : System.out.println("Haziran");break;
         case Aylar.TEMMUZ : System.out.println("Temmuz");break;
         case Aylar.AGUSTOS : System.out.println("Agustos");break;
         case Aylar.EYLUL : System.out.println("Eylul");break;
         case Aylar.EKIM : System.out.println("Ekim");break;
         case Aylar.KASIM : System.out.println("Kasim");break;
         case Aylar.ARALIK : System.out.println("Aralik");break;
         default:System.out.println("Tanimsiz Ay");
       }
  }
}
```

7.1.6.1. **Arayüzün İçerisinde Tanımlanmış Alanlara İlk Değerlerinin Verilmesi**

Arayüzlerin içerisinde tanımlanmış alanların ilk değerleri, çalışma anında da (run-time) verilebilir. Aşağıdaki örneği inceleyelim. (yorum ekle)

Örnek: *Test.java* (yorum ekle)

```
interface A7 {

  int devir_sayisi = (int) ( Math.random() *6 ) ;
  String isim = "A7" ;
  double t1 = ( Math.random() * 8 ) ;
}

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

       System.out.println("devir_sayisi = " + A7.devir_sayisi );
       System.out.println("isim = " + A7.isim );
       System.out.println("t1 = " + A7.t1 );
  }
}
```

A7 arayüzünün içerisindeki ilkel (primitive) int tipindeki `devir_sayisi` ve `t1` alanlarının değerlerini derleme anında bilebilmek imkansızdır. Bu değerler ancak çalışma anında bilenebilir. (yorum ekle)

Dikkat edilmesi gereken bir başka husus ise *A7* arayüzünün içerisindeki alanların ne zaman ilk değerlerini aldıklarıdır. Bir arayüzün içerisindeki alanlar `final` ve `statik` oldukları için, *A7* arayüzüne ilk kez erişildiği zaman, *A7* arayüzünün içerisindeki tüm alanlar ilk değerlerini alırlar. (yorum ekle)

7.1.7. **Genel Bakış**

Arayüzler ve soyut sınıfların bizlere sağlamak istediği fayda nedir? Aslında ulaşılmak istenen amaç çoklu yukarı çevirimdir (*upcasting*). Bir sınıfa ait nesnenin bir çok tipteki sınıf referansına bağlanabilmesi, uygulama içerisinde büyük esneklik sağlar. Bir örnek üzerinde açıklayalım.... (yorum ekle)

Örnek: *GenelBakis.java* (yorum ekle)

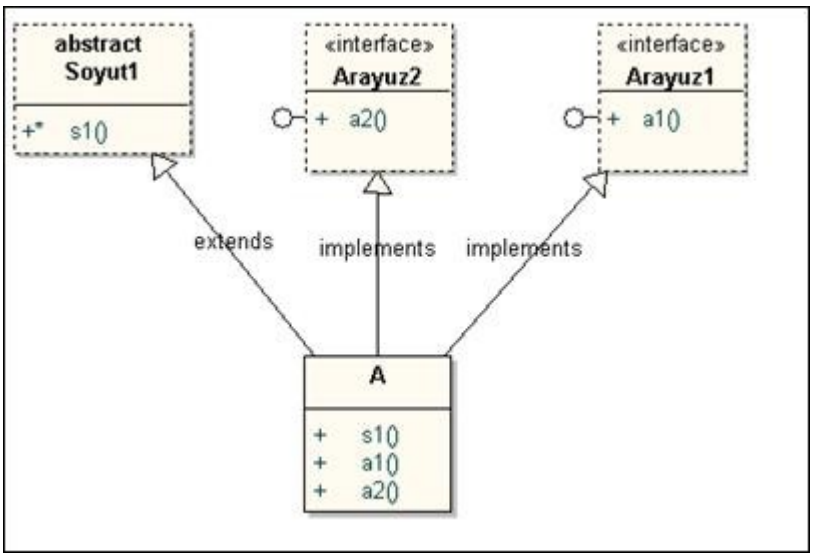
```
interface Arayuz1 {
  public void a1() ;
}

interface Arayuz2 {
  public void a2() ;
}

abstract class Soyut1 {
  public abstract void s1();
}

class A extends Soyut1 implements Arayuz1, Arayuz2 {
 public void s1() { // iptal etti (override)
       System.out.println("A.s1()");
 }
 public void a1() { // iptal etti (override)
       System.out.println("A.a1()");
 }
 public void a2() { // iptal etti (override)
       System.out.println("A.a2()");
 }
}

public class GenelBakis {
  public static void main(String args[]) {
       Soyut1 soyut_1 = new A();
       Arayuz1 arayuz_1 = (Arayuz1) soyut_1 ;
       Arayuz2 arayuz_2 = (Arayuz2) soyut_1 ;
       // Arayuz2 arayuz_2 = (Arayuz2) arayuz_1 ; // dogru

       soyut_1.s1();
       // soyut_1.a1(); // ! Hata !
       // soyut_1.a2(); // ! Hata !

       arayuz_1.a1();
       // arayuz_1.a2(); // ! Hata !
       // arayuz_1.s1(); // ! Hata !

       arayuz_2.a2();
       // arayuz_2.a1(); // ! Hata !
       // arayuz_2.s1(); // ! Hata !
  }
}
```

A sınıfı *Soyut1* soyut sınıfından türetilmiştir, ayrıca *A* sınıfı *Arayuz1* ve *Arayuz2* arayüzlerine (*interface*) erişmektedir. Daha yakından incelenirse. (yorum ekle)

Gösterim-7.5:

```
class A extends Soyut1 implements Arayuz1, Arayuz2 {
```

Yukarıdaki gösterim şunu der: *A* sınıfına ait bir nesne, *Soyut1* sınıfı, *Arayuz1* veya *Arayuz2* arayüzü tipindeki referanslara bağlanabilir. Anlatılanlar UML diyagramına dönüştürülürse; (yorum ekle)



Şekil-7.6. Genel Bakış

Bu örneğimizde görüldüğü üzere, *A* sınıfı ait tek bir nesne oluşturulmuştur. Oluşturulan nesne farklı referanslara bağlanabilmektir. (yorum ekle)

Gösterim-7.6:

```
Soyut1 soyut_1 = new A();
Arayuz1 arayuz_1 = (Arayuz1) soyut_1 ; // tip degisimi
Arayuz2 arayuz_2 = (Arayuz2) soyut_1 ; // tip degisimi
```

Oluşturulan tek bir *A* sınıfına ait nesne, bir çok farklı sınıf ve arayüz tipindeki referansa bağlanabilmektedir. Örneğin yukarıdaki gösterimde, *A* sınıfına ait nesnemiz ilk olarak *Soyut1* sınıfı tipindeki referansa bağlanmıştır. Bağlanabilir çünkü *A* sınıfı *Soyut1* sınıfından türemiştir. *Soyut1* sınıfı tipindeki referansa bağlı olan *A* sınıfına ait nesnenin sadece `s1()` yordamına ulaşılabilir. (yorum ekle)

Daha sonra *Soyut1* referansının bağlı olduğu *A* sınıfına ait nesneyi, *Arayuz1* tipindeki referansa bağlanıyor ama bu bağlama sırasında *A* sınıfına ait nesneyi *Arayuz1* tipindeki referansa bağlanacağını açık açık söylemek gerekir. *A* sınıfına ait nesnemiz *Arayuz1* tipindeki referansa bağlanırsa bu nesnenin sadece `a1()` yordamına erişilebilir. (yorum ekle)

Aynı şekilde *Soyut1* sınıfı tipindeki referansa bağlı olan *A* sınıfına ait nesne, *Arayuz2* tipindeki referansa bağlınabilir. *A* sınıfına ait nesne, *Arayuz2* tipindeki referansa bağlanırsa, bu nesnenin sadece `a2()`

yordamına erişebilir. Gösterim-7.6.'deki ifade yerine aşağıdaki gibi bir ifade de kullanılabilir fakat bu sefer üç ayrı *A* sınıfına ait nesne oluşturmuş olur. (yorum ekle)

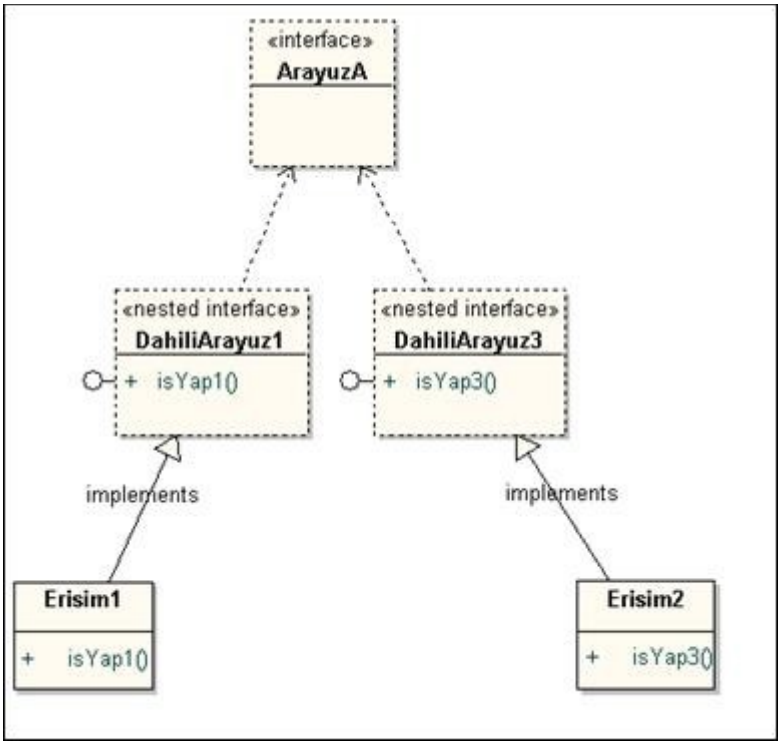
Gösterim-7.7:

```
Soyut1 soyut_1 = new A();
Arayuz1 arayuz_1 = new A();
Arayuz2 arayuz_2 = new A();
```

Görüldüğü üzere, *A* sınıfına ait üç adet nesne oluşturduk ve bu nesnelerin her birini farklı tipteki referanslara bağlayabildik. Bu olay nesneye yönelik tasarımlar yaparken işimize çokça yarayabilecek bir yaklaşımdır. (yorum ekle)

7.1.8. **Dahili Arayüzler (Nested Interface)**

Bir arayüz, başka bir arayüzün veya sınıfın içerisinde tanımlanabilir. Bir arayüzün içerisinde tanınlanan dahili arayüzler, `protected`, `friendly` veya `private` erişim belirleyicisine sahip olamaz. Örneğimize geçmeden evvel UML diyagramını inceleyelim. (yorum ekle)



Sekil-7.7. Dahili arayüzler

UML diyagramımızdan anlaşılacağı üzere, *ArayuzA* arayüzünün içerisinde iki adet dahili arayüz (nested interface) tanımlanmıştır. Dışarıdaki iki sınıfımız, dahili olarak tanımlanmış bu iki arayüze erişebilmektedir. (yorum ekle)

Örnek: *DahiliArayuzTest.java* (yorum ekle)

```
interface ArayuzA { //aslinda public erisim belirleyicisine sahip
  public interface DahiliArayuz1 {
       public void isYap1() ;
  }

  /* // ! Hata !
  protected interface DahiliArayuz2 {
       public void isYap2() ;
  }
  */

  interface DahiliArayuz3 { // aslinda public erisim belirleyicisine sahip
       public void isYap3() ;
  }

  /* // ! Hata !
  private interface DahiliArayuz4 {
       public void isYap4() ;
  }
  */
}

class Erisim1 implements ArayuzA.DahiliArayuz1 {
  public void isYap1() {
       System.out.println("Erisim1.isYap1()");
  }
}

class Erisim2 implements ArayuzA.DahiliArayuz3 {
  public void isYap3() {
       System.out.println("Erisim1.isYap3()");
  }
}

public class DahiliArayuzTest {
  public static void main(String args[]) {
       Erisim1 e1 = new Erisim1();
       Erisim2 e2 = new Erisim2();
       e1.isYap1();
       e2.isYap3();
  }
}
```

Dahili arayüzlere erişen sınıflar açısından olaylar aynıdır. Yine bu dahili arayüzlerin içerisindeki gövdesiz yordamları iptal etmeleri gerekmektedir. Uygulamanın çıktısı aşağıdaki gibidir; (yorum ekle)

```
Erisim1.isYap1()

Erisim1.isYap3()
```

7.1.8.1. Sınıfların İçerisinde Tanımlanan Dahili Arayüzler (*Nested Interface*)

Bir arayüz diğer bir arayüzün içerisinde tanımlandığı gibi, bir sınıfın içerisinde de tanımlanabilir. (yorum ekle)

Örnek: *SinifA.java* (yorum ekle)

```
public class SinifA {

  public interface A1 {
       public void ekranaBas();
  } //arayüz

  public class DahiliSinif1 implements A1 {
       public void ekranaBas() {
```

```
            System.out.println("DahiliSinif1.ekranaBas()");
        }
    } // class DahiliSinif1

    protected class DahiliSinif2 implements A1 {
        public void ekranaBas() {
            System.out.println("DahiliSinif2.ekranaBas()");
        }
    } // class DahiliSinif2

    class DahiliSinif3 implements A1 {
        public void ekranaBas() {
            System.out.println("DahiliSinif3.ekranaBas()");
        }
    } // class DahiliSinif3

    private class DahiliSinif4 implements A1 {
        public void ekranaBas() {
            System.out.println("DahiliSinif4.ekranaBas()");
        }
    } // class DahiliSinif4

    public static void main(String args[]) {
        SinifA.DahiliSinif1 sad1 = new SinifA().new DahiliSinif1();
        SinifA.DahiliSinif2 sad2 = new SinifA().new DahiliSinif2();
        SinifA.DahiliSinif3 sad3 = new SinifA().new DahiliSinif3();
        SinifA.DahiliSinif4 sad4 = new SinifA().new DahiliSinif4();

        sad1.ekranaBas();
        sad2.ekranaBas();
        sad3.ekranaBas();
        sad4.ekranaBas();

        SinifB sb = new SinifB();
        sb.ekranaBas();
    }
}

class SinifB implements SinifA.A1{
    public void ekranaBas() {
        System.out.println("SinifB.ekranaBas()");
    }
}
```

SinifA sınıfının içerisinde tanımlanan *A1* arayüzüne, *SinifB* sınıfından ulaşılabilir. Bir sınıfın içerisinde dahili arayüz tanımlanabildiği gibi dahili sınıfda tanımlanabilir. Bu konu az sonra incelenecektir. Bu örneğimizdeki ana fikir, bir sınıfın içerisinde nasıl dahili arayüzün oluşturulduğu ve bu dahili arayüzün, dahili sınıf olsun veya dışarıdan başka bir sınıf tarafından olsun, nasıl erişilebildiğini göstermektir. (yorum ekle)

7.2. **Dahili Sınıflar (Inner Classes)**

Dahili sınıflar JDK 1.1 ile gelen bir özelliktir. Bu özellik sayesinde bir sınıf diğer bir sınıfın içerisinde tanımlanabilir; böylece mantıksal bir bütünü oluşturan bir çok sınıf tek bir çatı alında toplanır. Dahili sınıflar yapısal olarak 3 gruba ayrılabilir. (yorum ekle)

- Dahili üye sınıflar
- Yerel sınıflar (*Local classes*)
- İsimsiz sınıflar (*Anonymous classes*)

7.2.1. **Dahili Üye Sınıflar**

Bir sınıfın içerisinde, başka bir sınıfı tanımlamak mümkündür; Şöyle ki... (yorum ekle)

Gösterim-7.8:

```
class CevreliyiciSinif {

  class DahiliSinif {
      //....
  }

  //...
}
```

Başka bir sınıfın içerisinde tanımlanan bu sınıfa dahili üye sınıf denir. Dahili sınıfları, çevreleyici sınıfların içerisinde kullanmak, geçen bölümlerde incelediğimiz komposizyondan yönteminden farklıdır. (yorum ekle)

Dahili üye sınıflar, tek başlarına bağımsız sınıflar gibi düşünülebilir. Örnek üzerinde incelersek. (yorum ekle)

Örnek: *Hesaplama.java* (yorum ekle)

```
public class Hesaplama {

  public class Toplama { //Dahili uye sinif
    public int toplamaYap(int a, int b) {
        return a+b ;
    }
  } // class Toplama

  public static void main(String args[]) {
    Hesaplama.Toplama ht = new Hesaplama().new Toplama() ;
    int sonuc = ht.toplamaYap(3,5);
    System.out.println("Sonuc = " + sonuc );
  }
} // class Hesapla
```

Hesaplama sınıfının içerisinde tanımlanmış *Toplama* sınıfı bir dahili üye sınıfıdır. *Hesaplama* sınıfı ise çevreleyici sınıftır. *Toplama* sınıfına ait bir nesne oluşturmak için, önce *Hesaplama* sınıfına ait bir nesne oluşturmamız gerekir. (yorum ekle)

Gösterim-7.9:

```
Hesaplama.Toplama ht = new Hesaplama().new Toplama() ;
```

`ht` referansı *Toplama* dahili üye sınıfı tipindedir; artık bu referansı kullanarak *Toplama* nesnesine ait `toplamaYap()` yordamına ulaşabiliriz. Uygulamanın çıktısı aşağıdaki gibdir; (yorum ekle)

```
Sonuc = 8
```

7.2.1.1. Dahili Üye Sınıflar ve Erişim

Dahili üye sınıflara, `public`, `friendly`, `protected` veya `private` erişim belirleyicileri atanabilir, böylece dahili üye sınıflarımıza olan erişimi kısıtlamış/açmış oluruz. Dikkat edilmesi gereken diğer bir husus ise bir dahili üye sınıf `private` erişim belirleyicisine sahip olsa dahi, çevreleyici sınıf içerisindeki tüm yordamlar tarafından erişilebilir olmasıdır. Bu kısıt ancak başka sınıflar için geçerlidir. (yorum ekle)

Örnek: *Hesaplama1.java* (yorum ekle)

```
public class Hesaplama1 {
  public class Toplama { // Dahili uye sinif - public
       public int toplamaYap(int a, int b) {
                          return a + b ;
       }
  } // class Toplama

  protected class Cikartma { // Dahili uye sinif - protected
       public int cikartmaYap(int a, int b) {
         return a - b ;
       }
  } // class Cikartma

  class Carpma { // Dahili uye sinif - friendly
       public int carpmaYap(int a, int b) {
         return a * b ;
       }
  } // class Carpma

  private class Bolme { // Dahili uye sinif - private
       public int bolmeYap(int a, int b) {
         return a / b ;
       }
  } // class Bolme

  public static void main(String args[]) {
       Hesaplama1.Toplama ht = new Hesaplama1().new Toplama() ;
       Hesaplama1.Cikartma hck = new Hesaplama1().new Cikartma() ;
       Hesaplama1.Carpma hcp = new Hesaplama1().new Carpma() ;
       Hesaplama1.Bolme hb = new Hesaplama1().new Bolme() ;

       int sonuc1 = ht.toplamaYap(10,5);
       int sonuc2 = hck.cikartmaYap(10,5);
       int sonuc3 = hcp.carpmaYap(10,5);
       int sonuc4 = hb.bolmeYap(10,5);

       System.out.println("Toplama Sonuc = " + sonuc1 );
       System.out.println("Cikartma Sonuc = " + sonuc2 );
       System.out.println("Carpma Sonuc = " + sonuc3 );
       System.out.println("Bolme Sonuc = " + sonuc4 );
  }
```

```
} // class Hesaplama
```

Hesaplama1 sınıfımızın içerisinde toplam 4 adet dahili üye sınıf mevcuttur. `public` erişim belirleyicisine sahip *Toplama* dahili üye sınıfı, `protected` erişim belirleyicisen sahip *Cikartma* dahili üye sınıfı, `friendly` erişim belirleyicisine sahip *Carpma* dahili üye sınıfı ve `private` erişim belirleyicisine sahip *Bolme* üye dahili sınıfı. *Hesaplama1* sınıfı, bu 4 adet dahili üye sınıfın çevreliyici sınıfıdır. Çevreleyici olan *Hesaplama1* sınıfının statik olan `main()` yordamına dikkat edilirse, bu yordamın içerisinde tüm (*private* dahil) dahili üye sınıflara erişilebildiğini görülür. Bunun sebebi, `main()` yordamı ile tüm dahili üye sınıfların aynı çevreliyici sınıfın içerisinde olmalarıdır. Uygulamanın çıktısı aşağıdaki gibidir: (yorum ekle)

```
Toplama Sonuc = 15
Cikartma Sonuc = 5
Carpma Sonuc = 50
Bolme Sonuc = 2
```

Yukarıdaki örneğin yeni bir versiyonu yazılıp, dahili üye sınıflar ile bunlara ait erişim belirleyicilerin nasıl işe yaradıklarını incelenirse... (yorum ekle)

Örnek: *Hesaplama2Kullan.java* (yorum ekle)

```
class Hesaplama2 {
  public class Toplama2 { // Dahili uye sinif - public
       public int toplamaYap(int a, int b) {
         return a + b ;
       }
  } // class Toplama2

  protected class Cikartma2 {   // Dahili uye sinif - protected
       public int cikartmaYap(int a, int b) {
         return a - b ;
       }
  } // class Cikartma2

  class Carpma2 { // Dahili uye sinif - friendly
       public int carpmaYap(int a, int b) {
         return a * b ;
       }
  } // class Carpma2

  private class Bolme2 { // Dahili uye sinif - private
       public int bolmeYap(int a, int b) {
         return a / b ;
       }
  } // class Bolme2

} // class Hesaplama2

public class Hesaplama2Kullan {
  public static void main(String args[]) {

       Hesaplama2.Toplama2 ht=new Hesaplama2().new Toplama2() ;
       Hesaplama2.Cikartma2 hck=new Hesaplama2().new Cikartma2() ;
       Hesaplama2.Carpma2 hcp = new Hesaplama2().new Carpma2() ;
       // Hesaplama2.Bolme3 hb = new Hesaplama2().new Bolme2() ;
```

```
      // ! Hata !

         int sonuc1 = ht.toplamaYap(10,5);
         int sonuc2 = hck.cikartmaYap(10,5);
         int sonuc3 = hcp.carpmaYap(10,5);
         // int sonuc4 = hb.bolmeYap(10,5); // ! Hata !

         System.out.println("Toplama Sonuc = " + sonuc1 );
         System.out.println("Cikartma Sonuc = " + sonuc2 );
         System.out.println("Carpma Sonuc = " + sonuc3 );

    }
}
```

Hesaplama2 sınıfımız, toplam 4 adet olan dahili üye sınıflarının çevreleyicisidir. Dahili üye sınıfları ve onlara ait erişim belirleyicileri incelenirse: (yorum ekle)

- *Toplama2* sınıfı, `public` erişim beliryicisine sahip olan dahili üye sınıfıdır. (yorum ekle)
- *Cikartma2* sınıfı, `protected` erişim belirleyicisine sahip olan dahili üye sınıfıdır. (yorum ekle)
- *Carpma2* sınıfı, `friendly` erişim belirleyicisine sahip olan dahili üye sınıfıdır. (yorum ekle)
- *Bolme2* sınıfı, `private` erişim belirleyicisine sahip olan dahili üye sınıfıdır. (yorum ekle)

Hesaplama2Kullan sınıfının `statik` olan `main()` yordamının içerisinden, *Hesaplama2* sınıfının içerisindeki dahili üye sınıflara erişilebilir mi? Erişilebilir ise hangi erişim belirleyicilerine sahip olan dahili üye sınıflara erişilebilir? (yorum ekle)

Normalde bir sınıf `private` veya `protected` erişim belirleyicisine sahip olamaz ancak dahili sınıflar `private` veya `protected` erişim belirleyicisine sahip olabilir. *Hesaplama2Kullan* sınıfı, *Hesaplama2* sınıfı ile aynı paket içerisinde (bkz: Bölüm 4-varsayılan paket) olduğu için, *Hesaplama2Kullan* sınıfı, *Hesapla2* sınıfının içerisinde tanımlanmış olan `public`, `protected` ve `friendly` erişim belirleyicilerine sahip olan dahili üye sınıflara erişebilir ama `private` erişim belirleyicisine sahip olan *Bolme* dahili üye sınıfına erişemez. Uygulamanın çıktısı aşağıdaki gibidir; (yorum ekle)

```
Toplama Sonuc = 15
Cikartma Sonuc = 5
Carpma Sonuc = 50
```

7.2.1.2. **Dahili Üye Sınıflar ve Bunları Çevreleyen Sınıflar Arasındaki İlişki**

Dahili üye sınıflar, içerisinde bulundukları çevreleyici sınıfların tüm alanlarına (`statik` veya değil-`private` dahil) ve yordamlarına (`statik` veya değil-`private` dahil) erişebilirler. (yorum ekle)

Örnek: *Hesaplama3.java* (yorum ekle)

```
public class Hesaplama3 {
  private int sabit1  = 2 ;
  private static int sabit2 = 1 ;

  public class Toplama3 { //Uye dahili sinif
    public  int toplamaYap(int a, int b) {
```

```
            return  (a+b) + sabit1  ; // dikkat
          }
    }  // class Toplama3

    public class Cikartma3 { //Uye dahili sinif
      public  int cikartmaYap(int a, int b) {
            dekontBilgileriGoster();    // dikkat
            return  (a-b) -  sabit2  ; // dikkat
          }
    }  // class Cikartma3

    private void  dekontBilgileriGoster() {
         System.out.println("Dekont Bilgileri Gosteriliyor");
    }

    public void ekranaBas(int a , int b ) {
         int sonuc = new Toplama3().toplamaYap(a,b);
         System.out.println("Sonuc = " + a + " + " + b + " + sabit1 = "
                                                          + sonuc);
    }

    public static void main(String args[]) {

      Hesaplama3 h3 = new Hesaplama3();
      h3.ekranaBas(10,5);

      // Toplama islemi
      Hesaplama3.Toplama3 ht3 = h3.new Toplama3() ;
      int sonuc = ht3.toplamaYap(11,6);
      System.out.println("Sonuc = 11 + 6 + sabit1  =  " + sonuc );

      // Cikartma islemi
         Hesaplama3.Cikartma3 hc3  = h3.new Cikartma3();
         int sonuc1 = hc3.cikartmaYap(10,5);
         System.out.println("Sonuc = 10 - 5 - sabit2 = " + sonuc1);
    }
} // class Hesaplama3
```

Hesaplama3 sınıfının içerisinde iki adet dahili üye sınıf bulunmaktadır. Bunlar *Toplama3* ve *Cikartma3* sınıflarıdır. *Toplama3* dahili üye sınıfı, *Hesaplama3* sınıfı içerisinde global olarak tanımlanmış ilkel (primitive) int tipindeki ve `private` erişim belirleyicisine sahip olan `sabit1` alanına erişebilmektedir. *Toplama3* dahili üye sınıfı, *Hesaplama3* sınıfı içerisinde tanımlanmış olan `sabit1` alanını kullanırken sanki kendi içerisinde tanımlanmış bir alanmış gibi, hiç bir belirteç kullanmamaktadır. (yorum ekle)

Aynı şekilde *Cikartma3* dahili üye sınıfı, *Hesaplama3* sınıfının içerisinde statik olarak tanımlanmış, `private` erişim belirleyicisine sahip ilkel `int` tipindeki `sabit2` alanını ve `private` erişim belirleyicisine sahip `dekontBilgileriGoster()` yordamına direk olarak erişebilmektedir. (yorum ekle)

Hesaplama3 sınıfının, nesne yordamı olan (-bu yordamın kullanılabilmesi için *Hesaplama3* sınıfına ait bir nesne oluşturmak gerekir) `ekranaBas()`, iki adet parametre alıp, geriye hiçbirşey döndürmez (void). Bu yordamın içerinde *Toplama3* dahili üye sınıfına ait nesne oluşturularak, bu dahili üye sınıfın `toplamaYap()` yordamı çağrılmaktadır. *Toplama3* dahili üye sınıfının `toplamaYap()` yordamından dönen cevap, `ekranaBas()` yordamının içerisinde ekrana bastırılır. (yorum ekle)

Dikkat edilmeye değer diğer bir husus ise sadece bir adet çevreleyici sınıfa ait nesne oluşturup, Bu nesneye bağlı referansı kullanarak, çevreleyici sınıf içerisindeki diğer dahili üye sınıflara ait nesnelerin oluşturulmasıdır. Olaylara daha yakından bakılırsa; (yorum ekle)

Gösterim-7.10:

```
Hesaplama3 h3 = new Hesaplama3();
```

```
Hesaplama3.Toplama3  ht3  = h3.new Toplama3() ;
Hesaplama3.Cikartma3 hc3  = h3.new Cikartma3();
```

Sadece bir adet *Hesaplama3* sınıfına ait nesne oluşturuldu. Bu nesneye bağlı referansı kullanarak (`h3`), diğer dahili üye sınıflara ait nesneler oluşturulabilir. Buradaki anafikir, çevreleyici sınıfların içerisinde bulunan her dahili üye sınıfa ait bir nesne oluşturmak için, her seferinde yeni bir çevreleyici sınıfa ait nesne oluşturma zorunluluğu olmadığıdır. Yani çevreleyici sınıfa ait bir nesne, yine çevreleyici sınıf tipindeki bir referansa bağlanırsa, işler daha kestirmeden çözülebilir. Uygulamanın çıktısı aşağıdaki gibidir; (yorum ekle)

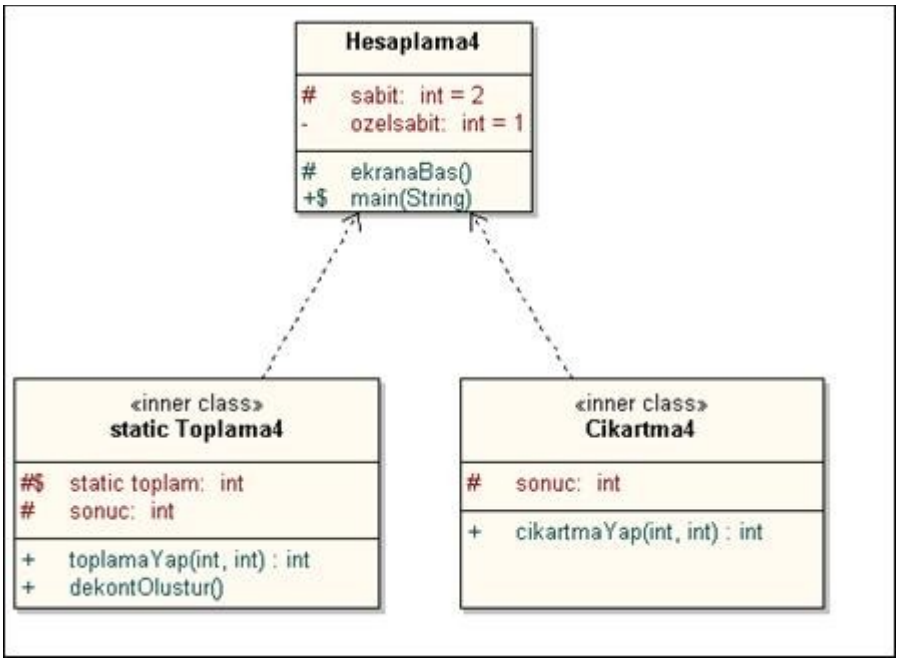
```
Sonuc = 10 + 5 + sabit1 = 17
Sonuc = 11 + 6 + sabit1 = 19
Dekont Bilgileri Gosteriliyor
Sonuc = 10 - 5 - sabit2 = 4
```

7.2.1.3. **Statik Dahili Üye Sınıflar**

Statik (`static`) olarak tanımlanmış dahili üye sınıflar, normal dahili üye sınıflardan farklıdırlar. Bu farklılıklar şöyledir: (yorum ekle)

- Statik dahili üye sınıfına ait nesne oluşturmak için, onu çevreleyen sınıfa ait bir nesne oluşmak zorunda değilizdir. (yorum ekle)
- Statik dahili üye sınıflar, kendilerini çevreleyen sınıfa ait bağlantıyı (-`this`-) kaybederler . (yorum ekle)

Statik dahili üye sınıflar, onları çevreleyen üst sınıfa ait global alanlara (statik veya değil) ve yordamlara (statik veya değil) direk ulaşım şansını kaybeder. Bunun sebebi, kendisini çevreleyen sınıf ile arasındaki bağı kopartmış olmasıdır. Buraya kadar ifade edilenleri örnek üzerinde inceleyelim, ama öncesinde UML diyagramı çizilirse... (yorum ekle)



Şekil-7.8. Statik Dahili Üye Sınıflar

Hesaplama4 sınıfının içerisinde, 2 adet dahili üye sınıf oluşturulacaktır; fakat bu dahili üye sınıflardan biri statik olarak tanımlanacaktır. Bu örnekte statik tanımlanacak olan dahili üye sınıf, *Toplama4* sınıfıdır. *Toplama4* sınıfına ait bir nesne oluşturulmak istenirse, bunun hemen öncesinde *Hesaplama4* sınıfına ait bir nesne oluşturulmaz. UML diyagramı Java uygulamasına dönüştürülürse... (yorum ekle)

Örnek: *Hesaplama4.java* (yorum ekle)

```
public class Hesaplama4 {

    int sabit  = 2 ;
      private int ozelsabit = 1 ;
      public static class Toplama4 {    // Statik uye dahili sinif
        static int toplam ;           // dogru
        int sonuc ;       // dogru
        public  int toplamaYap(int a, int b) {
          // return  (a+b)  + sabit  ;  ! Hata !
          sonuc = toplam = a+b ;
          return sonuc    ;
        }

        public void dekontOlustur() {
          /*  -sabit- alanina  ve
                -ekranaBas() yordamına ulasabilmek icin
                 Hesaplama4 sinifina ait nesne olusturmamiz gerekir.
                  */
                    Hesaplama4 hs4 = new Hesaplama4(); //dikkat
                    int a =  hs4.ozelsabit ; // dogru
                    hs4.ekranaBas() ; //dogru
                    System.out.println("Dekont olusturuyor = " +
                                       hs4.sabit + " - "  +a  );
       }
       }  // class Toplama4

  public class Cikartma4 { //Uye dahili sinif

       int sonuc ;
       // static int sonuc1 ; // ! hata !
       public  int cikartmaYap(int a, int b) {
         ekranaBas();  // dikkat
         sonuc = (a-b) -  ozelsabit ;
```

```
            return sonuc    ; // dikkat
          }
    }   // class Cikartma4

    private void ekranaBas() {
         System.out.println("Hesaplama4.ekranaBas()");
         }

    public static void main(String args[]) {

         //! Hata !
         // Hesaplama4.Toplama4 ht=new Hesaplama4().new Toplama4();
         Toplama4   tp4   =   new Toplama4();
         tp4.dekontOlustur();
         int sonuc = tp4.toplamaYap(10,5);
         System.out.println("Sonuc   =   10 + 5 =   " + sonuc );
    }

} // class Hesaplama4

class Hesaplama4Kullan {
  public static void main(String args[]) {

         //! Hata !
         // Hesaplama4.Toplama4 ht=new Hesaplama4().new Toplama4() ;
         Hesaplama4.Toplama4   tp4   =   new Hesaplama4.Toplama4();
         int sonuc = tp4.toplamaYap(10,5);
         System.out.println("Sonuc   =   10 + 5    =   " + sonuc );
    }

}   // class  Hesaplama4Kullan
```

Statik dahili üye sınıf olan *Toplama4* sınıfını yakın takibe alıp, neleri nasıl yaptığını inceleyelim. *Toplama4* statik dahili sınıfının içerisinde statik global alan tanımlayabiliriz. Statik olmayan dahili üye sınıfların içerisinde statik global alan tanımlanamaz. (yorum ekle)

Toplama4 statik dahili üye sınıfının, `toplamaYap()` yordamının içerisinde, *Hesaplama4* sınıfına ait global olarak tanımlamış ilkel (primitive) int tipindeki `sabit` alanına direk erişilemez. Statik dahili üye sınıflar ile bunları çevreleyen sınıflar arasında `this` bağlantısı yoktur. Eğer statik dahili üye sınıfın içerisinden, onu çevreleyen sınıfa ait bir alan (statik olmayan) veya yordam (statik olmayan) çağrılmak isteniyorsa, bu bizzat ifade edilmelidir. Aynı *Toplama4* statik dahili üye sınıfına ait `dekontOlustur()` yordamının içerisinde yapıldığı gibi. (yorum ekle)

`dekontOlustur()` yordamının içerisinde, *Hesaplama4* sınıfına ait nesne oluşturulmadan, `sabit`, `ozelsabit` alanlarına ve `ekranaBas()` yordamına ulaşamazdık. Buradaki önemli nokta, dahili üye sınıf statik olsa bile, kendisine çevreleyen sınıfın `private` erişim belirleyicisi sahip olan alanlarına (statik veya değil) ve yordamlarına (statik veya değil) erişebilmesidir. (yorum ekle)

Hesaplama4 sınıfının statik olan `main()` yordamının içerisinde, *Toplama4* statik dahili üye sınıfına ait nesnenin nasıl oluşturulduğuna dikkat edelim. *Toplama4* statik dahili üye sınıfına ait nesne oluştururken, onu çevreleyen sınıfa ait herhangi bir nesne oluşturmak zorunda kalmadık. (yorum ekle)

Son olarak *Hesaplama4Kullan* sınıfında statik olarak tanımlanan `main()` yordamının içerisindeki olayları inceleyelim. Başka bir sınıfın içerisinde statik dahili üye sınıfı ulaşmak için, sadece tanımlama açısından, dahili üye sınıfı çevreleyen sınıfın ismi kullanılmıştır. Mantıklı olanda budur, statik de olsa sonuçta ulaşılmak istenen dahili üye bir sınıfıdır. (yorum ekle)

Elimizde iki adet çalıştırılabilir sınıf mevcutur (`-main()` yordamı olan). *Hesaplama4* sınıfını çalıştırdığımızda (java Hesaplama4), sonuç aşağıdaki gibi olur; (yorum ekle)

```
Hesaplama4.ekranaBas()
Dekont olusturuyor = 2 - 1
Sonuc = 10 + 5 = 15
```

Eğer *Hesaplama4Kullan* sınıfı çalıştırılırsa (`java Hesaplama4Kullan`), sonuç aşağıdaki gibi olur; (yorum ekle)

```
Sonuc = 10 + 5 = 15
```

7.2.1.4. **Statik Dahili Üye Sınıflar ve Statik Yordamlar**

Statik dahili üye sınıfların içerisinde statik alanlar bulunduğu gibi, statik yordamlarda bulunabilir. Eğer statik dahili üye sınıfı içerisinde, statik bir yordam oluşturulmuş ise, bu yordamı çağırmak için ne statik dahili üye sınıfına ne de onu çevreleyen sınıfa ait herhangi bir nesne oluşturmak gerekmez. (yorum ekle)

Örnek: *Hesaplama5.java* (yorum ekle)

```
public class Hesaplama5 {
  private static int  x = 3 ;

  public static class Toplama5 { // Statik uye dahili sinif
       static int toplam ; // dogru
       int sonuc ;         // dogru
       public  static int toplamaYap(int a, int b) {
         // sonuc = a+b + x ;  // ! Hata !
       toplam = a + b + x ;
         return toplam    ;
       }
  } // class Toplama5

  public static void main(String args[]) {
       int sonuc = Hesaplama5.Toplama5.toplamaYap(16,8); // dikkat
       System.out.println("Sonuc  =  16 + 8  =  " + sonuc );
  }
} // class Hesaplama5
```

Toplama5 statik dahili üye sınıfının, statik olan `toplamaYap()` yordamından, *Hesaplama5* çevreliyici sınıfına ait ilkel (`primitive`) `int` tipinde tanımlanmış `x` alanına ulaşılabilir. Bunun sebebi `x` alanında statik olarak tanımlanmış olmasıdır. `main()` yordamının içerisinde, `toplamaYap()` yordamının çağrılışına dikkat edilirse, ne *Hesaplama5* sınıfına ait nesne, ne de *Toplama5* statik dahili üye sınıfına ait bir nesnenin oluşturulmadığı görülür. Uygulamanın çıktısı aşağıdaki gibidir; (yorum ekle)

```
Sonuc = 16 + 8 = 27
```

7.2.1.5. **Statik ve Final Alanlar**

Statik olmayan dahili üye sınıfların içerisinde, statik alanlar ve yordamlar tanımlanamaz; ama "`statik` ve `final`" alanlar tanımlanabilir. Bir alanın hem statik hemde `final` olması demek, onun SABİT olması anlamına geldiği için, Statik olmayan dahili üye sınıfların içerisinde `statik` ve `final` alanlar kullanılabilir. (yorum ekle)

Örnek: *StatikFinal.java* (yorum ekle)

```
class CevreliyiciSinif1 {

  class DahiliSinif1 { // Dahili uye siniflar
       // static int x = 10 ; // ! Hata !
  }
}
// Dogru
class CevreliyiciSinif2 {

  class DahiliSinif2 {
       int x; // Dogru
  }
}
// Dogru
class CevreliyiciSinif3 {

  class DahiliSinif3 {
       static final int x = 0;   // Dogru
  }
}
```

7.2.1.6. **Dahili Üye Sınıflar ve Yapılandırıcılar (*Constructors*)**

Dahili üye sınıfların yapılandırıcıları olabilir.

Örnek: *BuyukA.java* (yorum ekle)

```
public class BuyukA {

  public class B {
       public B() { // yapilandirici
         System.out.println("Ben B sinifi ");
       }
  }  // class B

  public BuyukA() {
       System.out.println("Ben BuyukA sinifi ");
  }

  public static void main(String args[]) {
       BuyukA ba = new BuyukA();
  }
}
```

Dahili üye sınıfını çevreleyen sınıfa ait bir nesne oluşturulduğu zaman, dahili üye sınıfına ait bir nesne otomatik oluşturulmaz. Yukarıdaki örneğimizde sadece *BuyukA* sınıfına ait bir nesne oluşturulmuştur ve bu yüzden sadece *BuyukA* sınıfına ait yapılandırıcı çağrılacaktır. Eğer dahili üye sınıf olan *B* sınıfına ait yapılandırıcının çağrılmasını isteseydik, `main()` yordamının içerisine : " `BuyukA.newB()` " dememiz gerekirdi. (yorum ekle)

7.2.1.7. İç içe Dahili Üye Sınıflar

Bir sınıfın içerisinde dahili üye sınıf tanımlayabilirsiniz. Tanımlanan bu dahili üye sınıfın içerisinde, yine bir dahili üye sınıf tanımlayabilirsiniz... bu böyle sürüp gidebilir... (yorum ekle)

Örnek: *Abc.java* (yorum ekle)

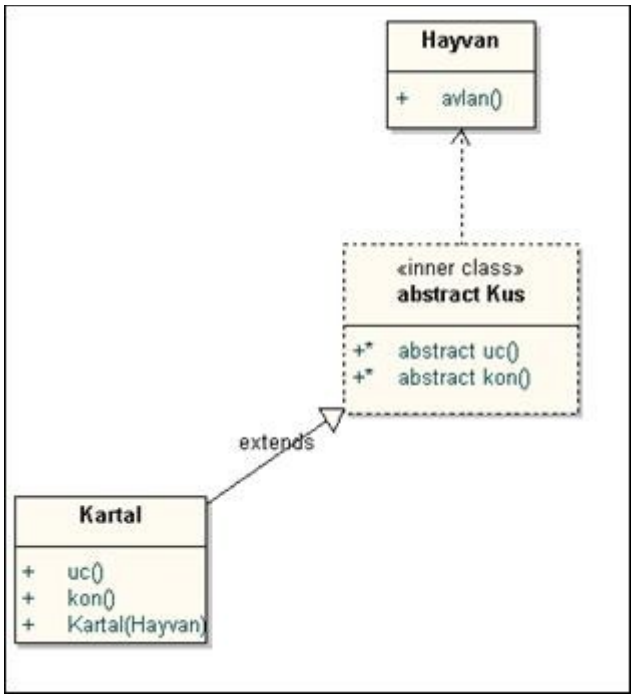
```
public class Abc {

  public Abc() {// Yapilandirici
       System.out.println("Abc nesnesi olusturuluyor");
  }

  public class Def {
       public Def() { // Yapilandirici
         System.out.println("Def nesnesi olusturuluyor");
       }

       public class Ghi {
         public Ghi() {    // Yapilandirici
           System.out.println("Ghi nesnesi olusturuluyor");
         }

       }   // class Ghi

  } //class Def

  public static void main( String args[] ) {
       Abc.Def.Ghi  ici_ice = new Abc().new Def().new Ghi();
  }

} // class Abc
```

Bu örnekte iç içe geçmiş üç adet sınıf vardır. Uygulamanın çıktısı aşağıdaki gibi olur:

```
Abc nesnesi olusturuluyor
Def nesnesi olusturuluyor
Ghi nesnesi olusturuluyor
```

7.2.1.8. Soyut (*Abstract*) Dahili Üye Sınıflar

Dahili üye sınıflar, soyut (*abstract*) sınıf olarak tanımlanabilir. Bu soyut dahili üye sınıflardan türeyen sınıflar, soyut dahili üye sınıfların içerisindeki gövdesiz (soyut) yordamları iptal etmeleri gerekmektedir. Örneğimize geçmeden evvel, UML diyagramını inceleyelim. (yorum ekle)



Şekil-7.9. Soyut Dahili Üye Sınıflar

Hayvan sınıfının içerisinde soyut (*abstract*) dahili üye sınıf olarak tanımlanmış *Kus* sınıfı iki adet gövdesiz (soyut-*abstract*) yordamı olsun, `uc()` ve `kon()`. `Kartal` sınıfı, soyut dahili üye sınıf olan *Kus* sınıfından türetilebilir. (yorum ekle)

Örnek: *HayvanKartal.java* (yorum ekle)

```
class Hayvan {

  abstract class Kus {
    public abstract void uc ();
    public abstract void kon();
  }

  public void avlan() {
    System.out.println("Hayvan avlaniyor...");
  }
}

class Kartal extends Hayvan.Kus {
  public void uc() {
    System.out.println("Kartal Ucuyor...");
  }
  public void kon() {
    System.out.println("Kartal Konuyor...");
  }

  // public Kartal() {  } // ! Hata !

  public Kartal(Hayvan hv) {
    hv.super(); //Dikkat
  }

  public static void main(String args[]) {
    Hayvan h = new Hayvan();  //Dikkat
    Kartal k = new Kartal(h);
    k.uc();
    k.kon();
```

```
  }
}
```

Kartal sınıfının içerisinde, soyut dahili üye sınıf olan *Kus* sınıfının, gövdesiz olan iki yordamı iptal edilmiştir. Olayları sırası ile inceleyelim, *Kartal* sınıfına ait bir nesne oluşturulmak istense bunun öncesinde *Kus* sınıfına ait bir nesnenin oluşturulması gerekir çünkü *Kartal* sınıfı *Kus* sınıfından türetilmiştir. Buraya kadar sorun yok, fakat asıl kritik nokta *Kus* sınıfının dahili üye sınıf olmasıdır. Daha açık bir ifade ile, eğer *Kus* sınıfına ait bir nesne oluşturulacaksa, bunun öncesinde elimizde *Kus* sınıfının çevreleyici sınıfı olan *Hayvan* sınıfına ait bir nesne bulunması zorunluluğudur. *Kus* sınıfı statik dahili üye sınıf olmadığından, *Hayvan* sınıfına bağımlıdır. Uygulamanın çıktısı aşağıdaki gibidir; (yorum ekle)

```
Kartal Ucuyor...
Kartal Konuyor...
```

Kartal sınıfının statik olarak tanımlanmış `main()` yordamının içerisine dikkat edersek, önce *Hayvan* sınıfına ait bir nesne sonrada *Kartal* sınıfına ait bir nesne oluşturduk. Daha sonra *Hayvan* sınıfı tipinde parametre kabul eden, *Kartal* sınıfının yapılandırıcısına, bu referansı pasladık. *Kartal* sınıfına ait yapılandırıcının içerisinde `super()` anahtar kelimesi ile, *Hayvan* sınıfının varsayılan yapılandırıcısını çağrılmıştır. (yorum ekle)

Gösterim-7.11:

```
CevreliyiciSinif.super() ;
```

Eğer *Kus* sınıfı, statik dahili üye sınıfı yapılsaydı, `super()` anahtar kelimesini kullanılmak zorunda değildi. Bunun sebebi, statik olan dahili üye sınıfların onları çevreleyen sınıflara bağımlı olmamasıdır. Yukarıdaki örnek bu anlatılanlar ışığında tekrardan yazılırsa. (yorum ekle)

Örnek: *HayvanKartal1.java* (yorum ekle)

```
class Hayvan1 {

  static abstract class Kus1 {
    public abstract void uc ();
    public abstract void kon();
  }

  public void avlan() {
    System.out.println("Hayvan avlaniyor...");
  }
}

class Kartal1 extends Hayvan1.Kus1 {
  public void uc() {
    System.out.println("Kartal1 Ucuyor...");
  }
  public void kon() {
```

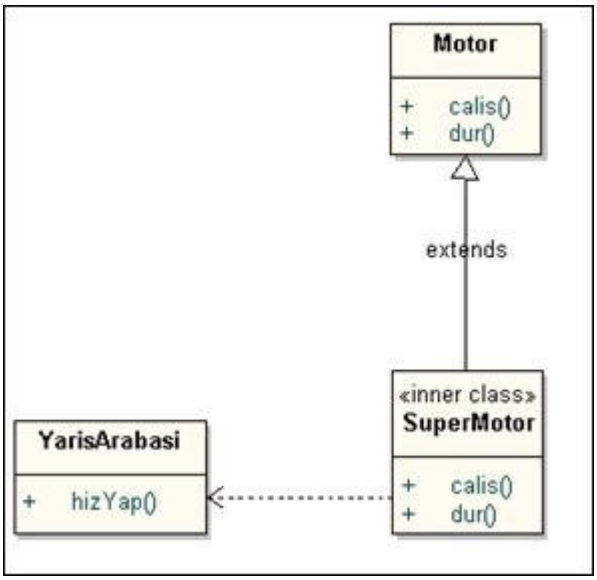
```
    System.out.println("Kartal1 Konuyor...");
  }

  public Kartal1() {  } // dogru

  public static void main(String args[]) {
    Kartal1   k1 = new Kartal1();
    k1.uc();
    k1.kon();
  }
}
```

Yukarıdaki örneğimizden görüldüğü üzere, artık *Kus* sınıfına ait bir nesne oluşturmak istersek, bunun hemen öncesinde *Hayvan* sınıfına ait bir nesne oluşturmak zorunda değilizdir. Bunun sebebi, *Kus* sınıfının statik dahili üye sınıfı olmasından kaynaklanır. Uygulamanın çıktısı aşağıdaki gibidir; (yorum ekle)

```
Kartal1 Ucuyor...
Kartal1 Konuyor...
```

7.2.1.9. **Türetilebilen Dahili Üye Sınıflar**

Dahili üye sınıflar, aynı normal sınıflar gibi başka sınıflardan türetilebilirler. Böylece diğer dillerde olan çoklu kalıtım desteğinin bir benzerini Java programlama dilinde de bulabiliriz. Dahili sınıfların varoluş sebeplerini biraz sonra detaylı bir şekilde inceleyeceğiz. Örneğimize geçmeden evvel, UML diyagramımızı inceleyelim; (yorum ekle)



Şekil-7.10. Türetilebilen Dahili Üye Sınıflar

Dahili üye sınıf olan *SuperMotor* sınıfı, *Motor* sınıfından türetilmiştir. UML diyagramını Java uygulamasını dönüştürüp, olayları daha somut bir şekilde incelersek. (yorum ekle)

Örnek: *YarisArabasi.java* (yorum ekle)

```
class Motor {
  public void calis() {
    System.out.println("Motor Calisiyor");
  }
  public void dur() {
    System.out.println("Motor Durdu");
  }
}

public class YarisArabasi {
  public void hizYap() {
    System.out.println("YarisArabasi hiz yapiyor");
  }
  public class SuperMotor extends Motor {
    public void calis() { // iptal etti (override)
      System.out.println("SuperMotor Calisiyor");
    }
    public void dur() { // iptal etti (override)
      System.out.println("SuperMotor Durdu");
    }
  }
}
```

Dahili üye sınıflar, başka sınıflardan türetilebildiği gibi arayüzlere erişip, bunların içlerindeki gövdesiz yordamları iptal edebilir, aynı normal sınıflar gibi... (yorum ekle)

7.2.2. Yerel Sınıflar (*Local Classes*)

Yerel sınıflar, yapılandırıcıların (*constructor*), sınıf yordamlarının (statik yordam), nesne yordamların, statik alanlara toplu değer vermek için kullandığımız statik bloğun (bkz:bölüm 3) veya statik olmayan alanlara toplu değer vermek için kullandığımız bloğun (bkz:bölüm 3) içerisinde tanımlanabilir. Yerel sınıfların genel gösterimi aşağıdaki gibidir; (yorum ekle)

Gösterim-7.12:

```
public class Sinif {
  public void yordam() {
    public class YerelSinif {
      //...
    }
  }
}
```

Yerel sınıflar, yalnızca içinde tanımlandıkları, yordamın veya bloğun içerisinde geçerlidir. Nasıl ki dahili üye sınıfların çevreleyici sınıfları vardı, yerel sınıfların ise çevreleyici yordamları veya blokları vardır. Yerel sınıflar tanımlandıkları bu yordamların veya blokların dışarısından erişilemezler. (yorum ekle)

Yerel sınıflara ait ilk özellikleri verelim;

- Yerel sınıflar tanımlandıkları yordamın veya bloğun dışından erişilemezler. (yorum ekle)
- Yerel sınıflar başka sınıflardan türetilebilir veya arayüzlere (interface) erişebilir. (yorum ekle)
- Yerel sınıfların yapılandırıcıları olabilir. (yorum ekle)

Yukarıdaki özelikleri Java uygulamasında ispatlanırsa;

Örnek: *Hesaplama6.java* (yorum ekle)

```
interface Toplayici {
  public int hesaplamaYap() ;
}

public class Hesaplama6 {

  public int topla(int a, int b) {
    class Toplama6  implements Toplayici {
      private int deger1 ;
      private int deger2;
      public Toplama6(int deger1, int deger2) { // yapilandirici
        this.deger1 = deger1;
        this.deger2 = deger2;
      }

      public int hesaplamaYap() { // iptal etti (override)
        return deger1+deger2;
      }

    } // class Toplama6

    Toplama6  t6 = new Toplama6(a,b);
    return t6.hesaplamaYap();
  }

  public void ekranaBas() {
    // Toplama6  t6 = new Toplama6(2,6,); // !Hata!-Kapsama alanının dışı
  }

  public static void main(String args[]) {
      Hesaplama6 h6 = new Hesaplama6();
    int sonuc = h6.topla(5,9);
    System.out.println("Sonuc  =  5 + 9  =  " + sonuc );
  }
} // class Hesaplama6
```

Bu örneğimizde *Toplama6* yerel sınıfdır. Yerel bir sınıf, başka bir sınıfdan türetilebilir veya bir arayüze erişip, onun gövdesiz yordamlarını iptal edebilir, aynı normal sınıflar gibi. *Toplama6* yerel sınıfı, *Hesapliyici* arayüzüne eriştiğinden, bu arayüzün gövdesiz yordamı olan `hesaplamaYap()` yordamını iptal etmek zorundadır. (yorum ekle)

Toplama6 yerel sınıfı, *Hesaplama6* sınıfının `topla()` yordamının içerisinde tanımlanmıştır. Bunun anlamı, *Toplama6* yerel sınıfına yanlızca `topla()` yordamının içerisinde erişilebileceğidir. *Hesaplama6* sınıfının nesne yordamı olan (bu yordama ulaşmak için Hesaplama6 sınıfına ait nesne oluşturmamız gerektiği anlamında...) `ekranaBas()` yordamının içerisinden, *Toplama6* yerel sınıfına ulaşılamaz çünkü *Toplama6* yerel sınıfı, `ekranaBas()` yordamının kapsama alanının dışında kalmaktadır. Uygulamamızın çıktısı aşağıdaki gibi olur; (yorum ekle)

```
Sonuc = 5 + 9 = 14
```

Yerel sınıflara diğer özellikler aşağıdaki gibidir;

- Yerel sınıflar, içinde bulundukları yordamın sadece `final` olan değişkenlerine ulaşabilirler. (yorum ekle)
- Yerel sınıflar, `statik` veya `statik` olmayan yordamların içerisinde tanımlanabilirler. (yorum ekle)
- Yerel sınıflar, `private`, `protected` ve `public` erişim belirleyicisine sahip olamazlar sadece `friendly` erişim belirleyicisine sahip olabilirler. (yorum ekle)
- Yerel sınıflar, `statik` olarak tanımlanamaz. (yorum ekle)

Yukarıdaki kuralları, bir örnek üzerinde uygularsak...

Örnek: *Hesaplama7.java* (yorum ekle)

```
public class Hesaplama7 {
  public static int topla(int a, final int b) {
    int a_yedek = a ;
       class Toplama7 {
      private int x ; // dogru
      public int y ;  // dogru
      // protected int z = a_yedek ;   // ! Hata !
      int p ;  // dogru
      public int degerDondur() {
        // int degera = a ;   // Hata
        int degerb = b ;
        return b ;
      }
    }   // class Toplama7

    Toplama7  t7 = new Toplama7();
    return t7.degerDondur();
  }

  public  void ekranaBas()  {

       /*   yerel siniflar sadece friendly erisim
            belirleyicisine sahip olabilirler

         public  class Toplama8 {
           public void  test() {}
         }  // class Toplama8

       */
  }  // ekranaBas

  public void hesaplamaYap()  {

       /* yerel sinif sadece friendly erisim
          belirleyicisine sahip olabilirler

          static class Toplama9 {
            public void abcd() {
            }
          }   // class Toplama9

       */
```

```
    }   // hesaplamaYap

    public static void main(String args[]) {

      int sonuc = Hesaplama7.topla(5,9);
      System.out.println("Sonuc    " + sonuc );
    }
  } // class Hesaplama7
```

Toplama7 yerel sınıfı, *Hesaplama7* sınıfının, statik olan `topla()` yordamının içerisinde tanımlanmıştır ve sadece `topla()` yordamının içerisinde geçerlidir. *Toplama7* yerel sınıfı, `topla()` yordamının içerisindeki `final` özelliğine sahip olan yerel değişkenlere erişip onları kullanabilir. Bu sebepten dolayı, ilkel (`primitive`) `int` tipinde tanımlanmış olan `a` ve `a_yedek` yerel değişkenlerine *Toplama7* yerel sınıfının içerisinden erişilemez, bu bir hatadır. (yorum ekle)

Hesaplama7 sınıfının, nesne yordamı olan `ekranaBas()` içerisinde tanımlanmış olan *Toplama8* yerel sınıfı hatalıdır. Hatanın sebebi *Toplama8* yerel sınıfının `public` erişim belirleyicisine sahip olmasıdır. Yukarıda belirtildiği üzere, yerel sınıflar ancak `friendly` erişim belirleyicisine sahip olabilir. (yorum ekle)

Aynı şekilde *Hesaplama7* sınıfının, nesne yordamı olan `hesaplamaYap()` içerisinde tanımlanmış olan *Toplama9* yerel sınıfı hatalıdır. Bu hatanın sebebi, *Toplama9* yerel sınıfının statik yapılmaya çalışılmasıdır. Az evvel belirtildiği gibi, yerel yordamlar, `statik` olarak tanımlanamazlardı. Uygulamanın çıktısı aşağıdaki gibidir; (yorum ekle)

```
Sonuc    9
```

7.2.3. İsimsiz Sınıflar (*Anonymous Classes*)

İsimsiz sınıflar, isimsiz ifade edilebilen sınıflardır. İsimsiz sınıflar havada oluşturulabildiklerinden dolayı bir çok işlem için çok avantajlıdır, özellikle olay dinleyicilerin (*event listeners*) devreye sokulduğu uygulamalarda sıkça kullanılırlar. İsimsiz sınıfların özellikleri aşağıdaki gibidir; (yorum ekle)

- Diğer dahili sınıf çesitlerinde olduğu gibi, isimsiz sınıflar direk `extends` ve `implements` anahtar kelimelerini kullanarak, diğer sınıflardan türetilemez ve arayüzlere erişemez. (yorum ekle)
- İsimsiz sınıfların herhangi bir ismi olmadığı için, yapılandırıcısında (*constructor*) olamaz. (yorum ekle)

Yukarıdaki kuralları, bir örnek üzerinde uygularsak...

Örnek: *Hesaplama8.java* (yorum ekle)

```
interface Toplayici {
       public int hesaplamaYap() ;
}

public class Hesaplama8 {

       public Toplayici topla(final int a, final int b) {
                 return new Toplayici() {
                           public int hesaplamaYap()  {

                                                // final olan yerel degiskenlere ulasabilir.
                                                return a + b ;
                                     }
                 }; // noktali virgul sart

       } // topla, yordam sonu


       public static void main(String args[]) {

                 Hesaplama8 h8 = new Hesaplama8();
                 Toplayici t  = h8.topla(5,9);
                 int sonuc = t.hesaplamaYap();
                 System.out.println("Sonuc  =  5 + 9  =  " + sonuc );
       }
} // class Hesaplama8
```

Hesaplama8 sınıfının, `topla()` yordamı *Toplayici* arayüzü tipindeki nesneye bağlı bir referans geri döndürmektedir. *Toplayici* arayüzü tipindeki nesneye bağlı bir referans geri döndürmek demek, *Toplayici* arayüzüne erişip onun gövdesiz olan yordamlarını iptal eden bir sınıf tipinde nesne oluşturmak demektir. Sonuçta bir arayüze ulaşan sınıf, ulaştığı arayüz tipinde olan bir referansa bağlanabilirdi. " Buraya kadar tamam ama isimsiz sınıfımız nerede... diyebilirsiniz. Olaylara daha yakından bakılırsa; (yorum ekle)

Gösterim-7.13:

```
return new Toplayici() {
  public int hesaplamaYap()  {

       // final olan yerel degiskenlere ulasabilir.
       return a + b ;
  }
}; // noktali virgul sart
```

İşte isimsiz sınıfımız !! .Yukarıdaki ifade yerine, `topla()` yordamın içerisinde yerel bir sınıf da yazılabilirdi. (yorum ekle)

Gösterim-7.14:

```
public Toplayici topla(final int a, final int b) {
  public class BenimToplayicim implements Toplayici {
       public int hesaplamaYap()  {
```

```
        // final olan yerel degiskenlere ulasabilir.
        return a + b ;
      }
    } // yordam sonu
    return new BenimToplayicim();
  }
```

İsimsiz sınıfları, yerel sınıfların kısaltılmışı gibi düşünebilirsiniz. Yerel sınıflarda `return new BenimToplayicim()` yerine, isimsiz sınıflarda hangi sınıf tipinde değer döndürüleceği en başta belirtilir. (yorum ekle)

Gösterim-7.15:

```
return new Toplayici() { ....
  ...
};
```

İsimsiz sınıflarda, yerel sınıflar gibi içinde bulundukları yordamın sadece `final` olarak tanımlanmış yerel değişkenlerine erişebilirler. (yorum ekle)

Yukarıdaki örneğimizde, isimsiz sınıfımız, *Toplayici* arayüzüne erişip onun gövdesiz sınıflarını iptal etmiştir, buraya kadar herşey normal. Peki eğer isimsiz sınıfımız, yapılandırıcısı parametre olan bir sınıftan türetilseydi nasıl olacaktı? Belirtildiği üzere isimsiz sınıfların yapılandırıcısı olamaz. (yorum ekle)

Örnek: *Hesaplama9.java* (yorum ekle)

```
abstract  class  BuyukToplayici {
  private  int deger ;
  public BuyukToplayici(int x) {
    deger = x ;
  }
  public int degerDondur() {
    return deger ;
  }
  public abstract int hesaplamaYap() ; // iptal edilmesi gerek
}

public class Hesaplama9 {
  public BuyukToplayici degerGoster( int gonderilen ) {
    return new BuyukToplayici( gonderilen ) {
      public int hesaplamaYap()  { //iptal etti (override)
        return  super.degerDondur() + 5  ;
      }
    }; // noktali virgul sart
  } // degerGoster , yordam sonu

  public static void main(String args[]) {

      Hesaplama9 h9 = new Hesaplama9();
    BuyukToplayici bt  = h9.degerGoster(5);
    int sonuc =  bt.hesaplamaYap();
```

```
      System.out.println("Sonuc  =  " + sonuc );
    }
} // class Hesaplama9
```

BuyukToplayici sınıfı soyuttur, bunun anlamı bu sınıfın içerisinde en az bir tane gövdesiz yordam olduğudur. *BuyukToplayici* sınıfının içerisindeki `hesaplamaYap()` gövdesiz yordamını, *BuyukToplayici* sınıfından türetilen alt sınıflar tarafından iptal edilmek zorundadır. (yorum ekle)

Bu örneğimizde ilginç olan iki nokta vardır. Birincisi, isimsiz bir sınıfın, soyut bir yordam dan türetilmesi, ikincisi ise türetilme yapılan *BuyukToplayici* sınıfına ait yapılandırıcısının parametre almasıdır. İsimsiz sınıfımızın yapılandırıcısı olamayacağından dolayı, *BuyukToplayici* sınıfına ait parametre alan yapılandırıcıyı burada çağıramayız. Bu işlemi *BuyukToplayici* sınıfından türetilen isimsiz sınıfımızı oluştururken yapmalıyız. (yorum ekle)

İsimsiz sınıfların içerisinde, onları çevreleyen yordamların final olmayan yerel değişkenleri kullanılamaz. "Peki ama bu örnekte kullanılıyor..." diyebilirsiniz. (yorum ekle)

Gösterim-7.16:

```
public BuyukToplayici degerGoster( int gonderilen ) {
  return new BuyukToplayici( gonderilen ) {
    public int hesaplamaYap()  { //iptal etti (override)
      return  super.degerDondur() + 5  ;
    }
  }; // noktali virgul sart

} // degerGoster , yordam sonu
```

İlkel `int` tipinde tanımlanmış `gonderilen` yerel değişkeni, `degerGoster()` yordamına aittir, isimsiz sınıfımızın içerisinde kullanılmamıştır. `return new BuyukToplayici(gonderilen)` ifadesi, `degerGoster()` yordamına dahil olduğundan bir sorun çıkmaz. Eğer uygulamamızı çalıştırırsak, ekran çıktısı aşağıdaki gibi olacaktır. (yorum ekle)

```
Sonuc = 10
```

7.2.4. Fiziksel İfade

İçerisinde Java kodları olan fiziksel bir dosya derlendiği (*compile*) zaman, bu fiziksel dosya içerisinde tanımlanmış her bir sınıf için, fiziksel bir *.class* dosyası oluşturulur. Peki olaylar dahili sınıflar içinde aynı mıdır? Her bir dahili sınıf için, bir fiziksel *.class* dosyası oluşturulur mu? Eğer oluşturuluyorsa ismi ne olur? (yorum ekle)

Her bir dahili sınıf için (3 çeşit dahili sınıf içinde geçerli) fiziksel *.class* dosyası oluşturulur. Bu *.class* dosyasının ismi ise, çevreleyen sınıfın ismi + **$** + dahili sınıfın ismi şeklindedir. *Hesaplama1.java* örneğimizden bir gösterim yaparsak; (yorum ekle)

Gösterim-7.17:

```
Hesaplama1$1.class
Hesaplama1$Bolme.class
Hesaplama1$Carpma.class
Hesaplama1$Cikartma.class
Hesaplama1$Toplama.class
Hesaplama1.class
```

Hesaplama1 sınıfı, *Bolme*, *Carpma*, *Cikartma* ve *Toplama* sınıflarının çevreliyici sınıfıdır, böyle olunca dahili sınıflarımıza ait *.class* dosyasının ismi de *ÇevreliyiciSınıf$DahiliSınıf* biçiminde olduğunu görürüz. (yorum ekle)

Java, *Hesaplama9.java* örneğimizdeki isimsiz sınıf için nasıl bir *.class* dosyası oluşturur? Cevabı hemen aşağıdadır; (yorum ekle)

Gösterim-7.18.

```
Hesaplama9$1.class
Hesaplama9.class
```

Java, *Hesaplama9.java* içerisinde belirtilmiş isimsiz sınıfa ait *.class* dosyası oluştururken isim olarak *1 (bir)* kullanmıştır. Eğer aynı çevreliyici sınıf içerisinde iki adet isimsiz sınıf olsaydı, bu isimsiz sınıfların ismi 1 ve 2 olacaktı. (yorum ekle)

Gösterim-7.19:

```
Hesaplama9$1.class
Hesaplama9$2.class
Hesaplama9.class
```

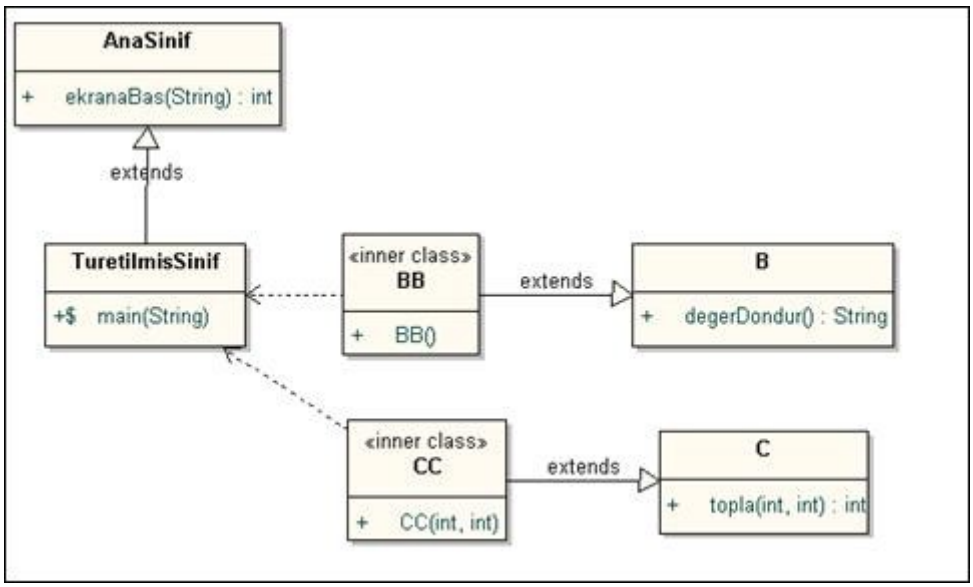
7.2.5. **Neden Dahili sınıflar?**

Dahili üye sınıflar, yerel sınıflar, isimsiz sınıflar hepsi çok güzel ama Java programlama dili neden bunlara ihtiyaç duymuş olabilir? Şimdiye kadar normal sınıflarımızla güzel güzel idare edebiliyorduk diyebilirsiniz. Dahili sınıfların var olmasındaki neden çoklu kalıtıma (*multiple inheritance*) tam desteği sağlamaktır. (yorum ekle)

Arayüzler ile çoklu kalıtım desteğini kısmen bulabiliyorduk ama bu tam değildi. Tam değildi çünkü bir sınıf iki normal sınıftan türetilemiyordu, bunun sakıncalarını tartışmıştık. Fakat bazı zamanlarda, arayüzler dışında, normal sınıflara ihtiyaç duyabiliriz. Normal sınıflar derken, soyut olmayan, problem çözmek için tasarlanmış işleyen sınıflardan bahsediyorum. Bu işleyen sınıfların iki tanesine aynı anda ulaşıp türetme yapılmıyorduk ama bu isteğimize artık dahili sınıflar ile ulaşabiliriz. (yorum ekle)

Java, dahili sınıflar ile çoklu kalıtım olan desteğini güvenli bir şekilde sağlamaktadır. Dahili sınıflar, kendilerini çevreleyen sınıfların hangi sınıftan türetildiğine bakmaksızın bağımsız şekilde ayrı sınıflardan türetilebilir veya başka arayüzlere erişebilir. (yorum ekle)

Örnek Java kodumuzu incelemeden evvel, UML diyagrama bir göz atalım.



Şekil-7.11. Dahili sınıflara neden ihtiyaç duyarız ?

UML diyagramından olayları kuş bakışı görebiliyoruz. *AnaSinif* sınıfından türetilmiş *TüretilmisSinif* sınıfının içerisinde iki adet dahili üye sınıf bulunmaktadır. Dahili üye sınıf olan *BB* ve *CC* sınıfları da, *B* ve *C* sınıflarından türetilmişlerdir. Bu örneğimizdeki ana fikir, bir sınıfın içerisinde dahili üye sınıflar kullanılarak çoklu kalıtımın güvenli bir şekilde yapılabildiğini göstermektir. UML diyagramını Java uygulamasına dönüştürürsek; (yorum ekle)

Örnek: *TuretilmisSinif.java* (yorum ekle)

```
class AnaSinif {
  public void ekranaBas(String deger) {
    System.out.println( deger );
  }
}

class B {
  public String degerDondur() {
    return "B";
  }
```

```
}

class C {
  public int  topla(int a , int b) {
     return a+b ;
  }
}

public class TuretilmisSinif  extends AnaSinif {

  public class BB extends B {
    public BB() {   // yapılandırıcı
      ekranaBas( "Sonuc = " + degerDondur() );
    }

  }

  public class CC extends C {
    public CC( int a , int b ) { // yapılandırıcı
      ekranaBas("Sonuc = " + topla(a,b) );
    }
  }

  public  static void main( String args[] ) {
    TuretilmisSinif.BB tbb= new TuretilmisSinif().new BB();
    TuretilmisSinif.CC tcc= new TuretilmisSinif().new CC(6, 9);

  }
}
```

TuretilmisSinif sınıfımız, *AnaSinif* sınıfından türetilmiştir fakat bu dahili sınıfların başka sınıflardan türetilmelerine engel teşkil etmez. Her bir dahili sınıfın kendine ait bir durumu olabilir. Dahili sınıflar kendilerini çevreleyen sınıflardan bağımsızdır. Dahili sınıflar ile onları çevreleyen sınıflar arasında kalıtımsal bir ilişki olmak zorunda değildir, geçen bölümlerde incelediğimiz "bir" ilişkisi, *Kaplan* **bir** *Kedidir* gibi. (yorum ekle)

Örneğimize geri dönersek, *B* sınıfından türetilmiş *BB* sınıfı ve *C* sınıfından türetilmiş *CC* sınıfı, *Anasinif* sınıfına ait *ekranaBas()* yordamını kullanarak sonuçlarını ekrana yansıtabilmektedirler. Olaylara bu açıdan baklacak olursa, *TüretilmisSinif* sınıfın sanki üç ayrı işleyen (normal) sınıftan güvenli ve kolay bir şekilde türetilmiş olduğu görülür. (yorum ekle)

Uygulamamızın çıktısı aşağıdaki gibi olur;

```
Sonuc = B
Sonuc = 15
```