

NESNE YÖNELİMLİ PROGRAMLAMA

İçindekiler

[SORU 1: object array \(nesne dizisi\)](#)

[SORU 2: Observer Design Pattern \(Gözlemci Tasarım Kalıbı\)](#)

[SORU 3: Nesne Yönelimli Programlama Dersi Bütünleme İmtihanı Çözümü](#)

[SORU 4: Java dilinde vektörler](#)

[SORU 5: JAVA da nihai uygulması \(Final\)](#)

[SORU 6: JAVA dilinde çoklu kalıtım \(miras\)](#)

[SORU 7: Nesne Akışı \(Casting\)](#)

[SORU 8: Nesne Kopyalama](#)

[SORU 9: Nesne Yönelimli Programlama Dersi Quiz Çözümü](#)

[SORU 10: C++ dilinde üzerine bindirme](#)

[SORU 11: C++ üzerinde çok şekillilik](#)

[SORU 12: C++ Nesne Yönelimli Programlama İlişki Türleri](#)

[SORU 13: Örnek C++ Sınıf İlişkileri](#)

[SORU 14: DOM \(DNM\)](#)

[SORU 15: C2 Üslûbu \(C2 Style\)](#)

[SORU 16: Nesne serileme ve dizme \(Object Serialization , Marshalling\)](#)

[SORU 17: Fabrika Metotları \(Factory Methods\)](#)

[SORU 18: Sabit Metotlar \(Static Methods\)](#)

[SORU 19: Birleşim Noktaları \(JoinPoints\)](#)

[SORU 20: Bağlam Örücüler \(Apect Weavers\)](#)

SORU 1: object array (nesne dizisi)

Bu yazının amacı, nesne yönelimli programlama konusu altında, sıklıkla geçen nesne dizilerini (object array) anlatmaktır. Kısaca bir dizinin elemanlarına nesne atamanın nasıl olacağı anlatılacaktır.

Öncelikle nesne yönelimli programlama dillerinde, kullanılan [nesne atıflarının \(object referrer\)](#) birer [gösterici \(pointer\)](#) olduklarını bilmemiz gerekir.

Örnek olarak bir sınıf tanımı ile başlayalım:

```
class insan{
    int yas;
    String isim;
}
```

Şimdi yukarıdaki sınıftan üretilen bir nesneyi ele alalım:

```
insan ali = new insan();
```

Burada ali ismi ile geçen [değişken \(variable\)](#) bir [nesne atıfıdır \(object referrer\)](#) ve insan sınıfından yeni bir nesne üretilmiş, ve bu nesne gösterilmiştir.

Ayrıca JAVA dilindeki dizi tanımlamasını da hatırlatalım:

```
int x[] = new int[5];
```

Yukarıdaki tanım ile, bir int dizisi tanımlanmıştır. Bu dizinin kendisi de aslında bir nesnedir ve kendisine özgü bazı fonksiyon ve özellikleri bulunur. Ancak konumuz kapsamında 5 elemanlı (0'dan başlayarak 4'e kadar giden (0 ve 4 dahil)) elemanları olduğunu ve int tipinde tanımlandığı için her elemanına birer tam sayı konulabilen (int) bir [dizi \(array\)](#) olduğunu söylememiz yeterlidir.

Artın hatırladığımız dizi tanımı ile nesne tanımını birleştirebiliriz:

```
insan x[] = new insan[5];
```

Elemanlarının her biri birer nesne atfı olan bir dizi tanımlanmıştır. Buna göre dizinin her elemanına, insan tipinden bir nesne göstericisi konulabilir.

Ancak nesne göstericileri için geçerli olan kurallar burada da geçerlidir. Örneğin yukarıdaki tanımdan hemen sonra aşağıdaki gibi bir kullanım hata verecektir:

```
x[2].yas = 20;
```

Buradaki hatanın sebebi henüz üretilmemiş olan bir nesnenin bir özelliğine erişiliyor olmasıdır. x[2] ile gösterilen dizinin RAM'de henüz kapladığı yer belirtilmemiş ve dolayısıyla bu alana erişime izin verilmemiştir. Çözüm olarak önce new komutu ile hafızada bu nesnenin yaşayacağı alan tanıtılmalıdır.

```
x[2] = new insan();
```

Ancak yukarıdaki şekilde bir tanımlamadan sonra bir önceki adımda hata veren satırı yazıp çalıştırabiliriz.

JAVA dilinde ayrıca for dizilerinin özel bir kullanımı aşağıdaki şekildedir:

```
for(insan i : x)
```

Yukarıdaki kod satırı, x dizisinin eleman sayısı kadar döner. Örneğin dizimiz 10 elemanlı ise, döngü 0'dan 9'a kadar dönecek ve her dönüşünde ilgili eleman değerini i isimli nesne göstericisine atayacaktır.

Yukarıda anlatılan bilgiler ışığında, derste yazdığım bir kodu aşağıda yayınlıyorum:

```
class insan{
    int yas;
    String isim;
    public String toString(){
        return "yas: "+yas+" isim: "+isim;
    }
}
```

Yukarıdaki kodda, bir insan sınıfı tanımlanmıştır. Bu tanıma göre insanın yaşı ve ismi bulunmaktadır. Ayrıca toString ismi verilen özel bir fonksiyon yazılmıştır. Bu fonksiyonun amacı, bu [sınıftan \(class\)](#) üretilen bir [nesnenin \(object\)](#), [dizgiye \(string\)](#) çevrilmesi durumunda otomatik olarak çalışıyor olmasıdır. Aslında bu fonksiyon, her [nesne \(object\)](#) için otomatik olarak tanımlıdır. Bunun sebebi java.lang.Object sınıfında tanımlı bir fonksiyon olmasıdır ve bu fonksiyon her sınıf tanımında [ezilebilir \(override\)](#). Örnek kodumuzda da üzerine yükleme (override) işlemi yapılmış ve fonksiyon ezilmiştir. Böylece insan sınıfından bir nesne, dizgi olarak kullanıldığında fonksiyonda belirtildiği üzere “yas: “ yazısına insanın yaşı eklenecek ardından “isim:” yazısının ardında insanın ismi eklenerek döndürülecektir.

Test amacıyla yazılmış örnek bir kod aşağıda verilmiştir:

```
public static void main(String args[]){
    insan i[];
    i = new insan[5];
    insan ali = new insan();
    ali.e = x;
    ali.yas = 20;
    ali.isim= "ali yilmaz";
    System.out.println(" alinin adresi : "+
ali.e.adres);

    i[0] = ali;
    i[1] = new insan();
    i[1].yas = 30;
    i[1].isim = "veli demir";
    System.out.println("i[0]:" +i[0] );
    i[2] = new insan();
    i[3] = new insan();
    i[4] = new insan();
    i[2].isim = "ahmet yildiz";
    i[3].isim = "veli yilmaz";
    i[4].isim = "cem yildiz";
    for(int j=0;j<i.length;j++){
        i[j].yas =10+ (int)( Math.random() * 40);
    }
    for(insan in : i){
        System.out.println(in.toString());
    }
}
```

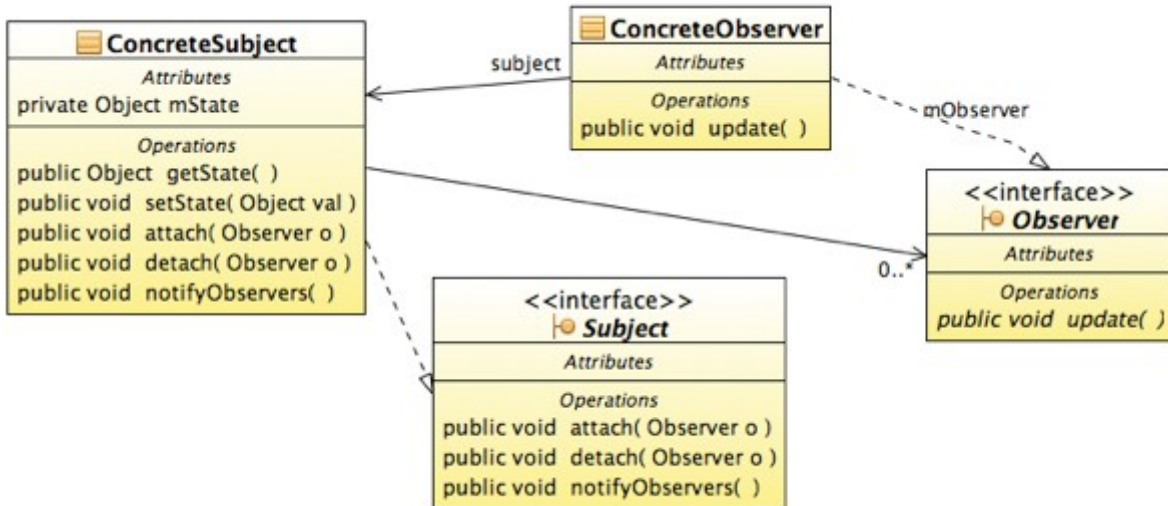
Yukarıdaki kodda örnek olarak 5 elemanlı bir insan dizisi oluşturulmuştur. Bu diziye önce ali isimli bir nesne göstericisinin içerisine atanan nesne yerleştirilmiş (dizinin 0. sırasına) ardından dizinin 1. elemanı için yeni bir nesne oluşturulmuştur. Son olarak dizinin 3., 4. ve 5.

elemanları için arka arkaya nesne oluşturulmuş (hafızada yer açılmış) ve isim ataması yapılmıştır. Kodun bundan sonraki kısmında bir döngü ile dizideki bütün insanlara rast gele yaş ataması yapılmıştır (random fonksiyonu) ve ikinci bir döngü ile insanlar ekrana bastırılmıştır. Buradaki bastırma işlemi sırasında insanlar daha önceden tanımlı olan toString fonksiyonu marifetiyle önce [dizgiye \(string\)](#) çevrilmiş ve ardından ekrana bastırılmıştır.

SORU 2: Observer Design Pattern (Gözlemci Tasarım Kalıbı)

Bu yazının amacı, [nesne yönelimli programlama ortamlarında](#) kullanılan bir tasarım kalıbı (design pattern) olan gözlemci tasarım kalıbını (observer design pattern) açıklamak ve kullanımına dair bir örnek vermektir. Nesne yönelimli olmayan programlama dillerinde (örneğin C) aynı yapı, [geri çağırım \(callback\)](#) ismi verilen yaklaşım ile yapılabilir.

Konuyla, klasik bir gözlemci tasarım kalıbını açıklayarak başlayalım. Geliştirme ortamı olarak sıklıkla kullandığım Netbeans içerisinde bir observer pattern eklenmesi halinde aşağıdakine benzer bir [sınıf diyagramı \(class diagram\)](#) görülmektedir:

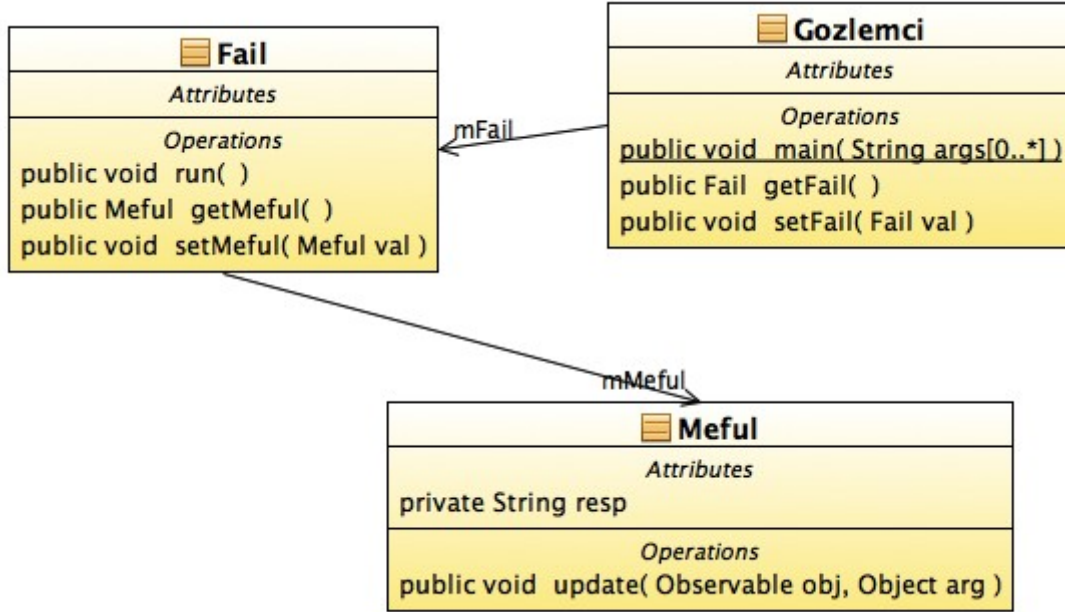


Yukarıdaki kalıpta görüldüğü üzere 4 temel unsur bu kalıbı oluşturmaktadır:

- ConcreteObserver : Somut bir gözlemci
- ConcreteSubject: Somut bir fail (özne)
- Subject : Somut failden (öznenen) türemiş olan diğer failler
- Observer: Somut gözlemciden türemiş olan diğer gözlemciler./
-

Gözlemci tasarım kalıbı, yukarıdaki şekilde görülebileceği üzere, aslında bir Fail / Meful ilişkisi (subject / object, özne / nesne) üzerine kuruludur. Somut fail ile kastedilen, gözlemci yapısındaki herhangi bir fiile sebep olan (eylemi yapan) faildir (özne). Buna karşılık, nesne yönelimli programlama dillerinde, üzerine bir fiil uygulanan şey genelde nesne olarak ifade edilir. Yani zaten nesne yönelimli dillerde herhangi bir fiil, sadece nesneler üzerine uygulanmaktadır. Bu durumda fail ile kastedilen, yukarıdaki kalıpta concreteSubject ve bu yapıdan türeyen bütün subject sınıfı nesneler, meful ise bu nesnelerin metotlarına parametre olarak geçirilen ve üzerinde işlem yapılan nesnelerdir.

Bu yaklaşımı bir kod örneği üzerinden açıklamaya çalışalım. Öncelikle tasarımımızın [sınıf diyagramını \(class diagram\)](#) vererek konuya başlayalım:



Yukarıdaki şekilde, modelimizde bulunacak olan üç unsur içinde, yani gözlemci, fail ve meful için birer sınıf tanımlı yaptık. Bu sınıfların kodu aşağıda verilmiştir:

```
8      *
9      * @author sadievreseker
10     */
11     public class Gozlemci {
12
13         /**
14          * @param args the command line arguments
15          */
16         public static void main(String args[]) {
17             System.out.println("Fiil >");
18
19             // standart inputtan okuyoruz
20             final Fail failNesne = new Fail();
21
22             // bir gozlemci üretiyoruz
23             final Meful mefulNesne = new Meful();
24
25             // fail üzerinde gözlenen bir meful bağlıyoruz
26             failNesne.addObserver( mefulNesne );
27
28             // thread olarak faili çalıştırıyoruz
29             Thread thread = new Thread(failNesne);
30             thread.start();
31         }
32     }
```

İlk olarak gözlemci [sınıfı \(class\)](#) ile konuya başlayalım. Sınıfın tanımında, bir main fonksiyonu bulunuyor ve bu main fonksiyonu aslında fail , meful ve gözlemci kavramlarının bir araya geldiği nokta oluyor. Zaten sınıf diyagramında da görüleceği üzere gözlemci bu iki kavramı bir araya getirir. Buna göre fail ve meful sınıflarından üretilen failNesne ve mefulNesne [nesneleri \(objects\)](#) (kodun 20. ve 23. satırlarında), kodun 26. satırında, addObserver fonksiyonu ile birbirine bağlanmıştır. Bu basit işlem, meful üzerinde bir failin gözlemini tanımlar. Ayrıca kodun 29. satırında klasik thread üretmek için kullanılan ve

Runnable arayüzü üzerinden Thread çağırımı ile failNesne'mizi bir thread şeklinde başlatıyoruz.

Gelelim Fail sınıfının tanımına:

```
7  /**
8  *
9  * @author sadievrenseker
10 */
11 import java.util.Observable;           //Faili Observable yapan
12 import java.io.BufferedReader;
13 import java.io.IOException;
14 import java.io.InputStreamReader;
15
16 public class Fail extends Observable implements Runnable {
17     public void run() {
18         try {
19             final InputStreamReader isr = new InputStreamReader( System.in );
20             final BufferedReader br = new BufferedReader( isr );
21             while( true ) {
22                 String response = br.readLine();
23                 setChanged();
24                 notifyObservers( response );
25             }
26         }
27         catch (IOException e) {
28             e.printStackTrace();
29         }
30     }
31 }
```

Yukarıdaki kodda, basit bir şekilde, kullanıcıdan, ekranda yazı okumak için BufferedReader, InputStreamReader'a ve InputStreamReader da System.in'e bağlanmış. Bu sayede, System.in üzerinden girilen bir veri, BufferedReader tarafından okunabilecektir. response isimindeki String [değişkeni \(variable\)](#) bir sonsuz döngü içerisinde (21. satırdaki while) kullanıcıdan veri okumakta ve sırasıyla setChanged() ve notifyObservers() fonksiyonlarını çağırılmaktadır. Buradaki notifyObservers fonksiyonu aslında bütün bu yazının kalbini oluşturuyor. Dikkat edilirse bu fonksiyona klavyeden okuduğumuz response isimli değişkeni parametre olarak veriyoruz.

Kodda dikkat edilecek diğer iki husus ise, Observable sınıfını [miras almamız \(inheritance\)](#) ve bir thread olarak çalışmasını istediğimiz için Runnable [arayüzünü \(interface\)](#) implements etmemizdir.

Son olarak meful sınıfını açıklayıp, kodun çalışmasına bakalım:

```
7  /**
8  *
9  * @author sadievrenseker
10 */
11 import java.util.Observable;
12 import java.util.Observer; /* Event Handler için */
13
14 public class Meful implements Observer {
15     private String mesaj;
16     public void update (Observable obj, Object arg) {
17         if (arg instanceof String) {
18             mesaj = (String) arg;
19             System.out.println( "\nFiil Cevabı: " + mesaj );
20         }
21     }
22 }
23
```


Kodda görüldüğü üzere, Meful isimli sınıfımız, basitçe Observer [arayüzünü](#) kendi üzerine uygulamıştır (implements).

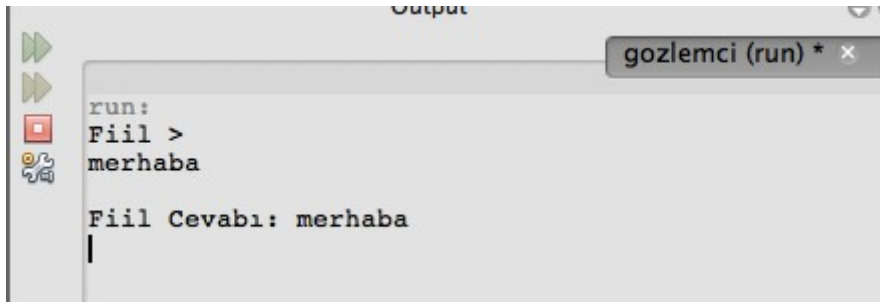
Meful sınıfının yegane fonksiyonu update fonksiyonudur ve parametre olarak bir Observable nesnesi bir de arg isimli Object tipinde (yani JAVA açısından tipsiz) bir parametre almaktadır.

Burada tipsiz tipler ile ilgili aşağıdaki yazıyı okumakta yarar olabilir

Son olarak update fonksiyonu içerisinde alınan mesaj, [tip inkılabı \(typecasting\)](#) ile Object tipinden String tipine inkılap ettirilmiş ve ardından ekrana basılmıştır.

Şimdi yukarıdaki senaryoyu toparlayacak olursak, gözlemci sınıfımızdaki main fonksiyonu çalıştırılarak bütün senaryo başlamaktadır. Bu main fonksiyonu içerisinde bir fail bir de meful nesnesi üretilmekte ve üretilen nesneler birbirine addObserver fonksiyonu ile bağlanmaktadır. Yani artık meful sınıfından türeyen nesneler, fail sınıfının birer observer'ı olmaktadır. Ardından fail sınıfından üretilen nesne bir thread olarak çalıştırılmakta ve kendi içerisinde bulundurduğu, ve threadlerin ilk çalışan fonksiyonu olan run() fonksiyonunda tanımlı olan kullanıcıdan bir dizgi (String) okuma işlemini gerçekleştirmektedir. Hemen ardından okunan mesajı, aslında hiçbir fikri olmayan ismini, ne tür bir parametre aldığını dahi bilmediği notifyObservers () fonksiyonuna parametre geçirmektedir. Bu fonksiyon, bir sevki tabii olarak meful sınıfındaki update isimli fonksiyonu tetiklemektedir. Update fonksiyonu ise kendisine parametre olarak gelen nesneyi String sınıfına çevirmekte ve ekrana bamaktadır.

Yukarıdaki kodu çalışması sonucu ekran görüntüsü aşağıdaki şekildedir:



```
run:
Fiil >
merhaba

Fiil Cevabı: merhaba
|
```

Görüldüğü üzere kodumuz bir fiil sorusu sormakta ve klavyeden benim yazdığım “merhaba” yazısını ekrana geri basmaktadır. Buradaki mesajı okuyan fail isimli sınıftan türeyen nesne iken, ekrana cevabı yazan meful isimli sınıftan türeyen nesnedir

Yukarıdaki bu tasarım kalıbını, daha gerçekçi bir örnek üzerinden açıklamaya çalışalım.

Örneğin bir müzayede (açık arttırma) ortamını, gözlemci tasarım kalıbı ile modellemek isteyelim. Buna göre müzayedeyi yöneten bir kişi ve bu müzayedeye katılan müzayedeciler bulunacaktır. Buna göre müzayedeyi yöneten kişi bir gözlemci olmakta, müzayedecilerin her biri birer fail (fiili yapan kişi) olmakta ve müzayedede sırasında verilen her teklif (fiyat) bir meful (fiilin etkisindeki değer) olmaktadır.

Bu yaklaşımı aşağıdaki şekilde göstermeye çalışalım:



Yukarıdaki yaklaşımda, müzayede sırasında verilen değerler, gözlemcinin update fonksiyonuna birer parametre olarak geçirilmekte ve nihayetinde gözlemciden bağımsız olarak ilgili fonksiyonlar çağırılmaktadır.

Gözlemci tipi tasarım kalıbının bir dezavantajı, bu noktada çıkmaktadır. Gözlemci tasarım kalıbında, meful yapısında olan ve fiilden etkilenen nesnelerin sistemde etkin rol oynaması beklenir. Diğer bir deyişle, açık arttırmaya katılan her bir katılımcı, fail olarak fiyatlarını belirtirken, gözlemci bu durumu sadece gözlemekte ve bir fiilde bulunmamakta ancak fiileri gözlemektedir. Bu anlamda, gözlemci tasarım kalıbı, çekme iletişim modeli (pull interaction model) olarak düşünülebilir. Alternatifi olan itme iletişim modelleri (push interaction model) failin doğrudan eylemde bulunması ile sonuca ulaşmakta ve dışarıdaki nesneleri çalışmaya zorlamaktadır. Yani gözlemci örüntü modelinde, dışarıda bulunan nesnelerin çalışmaları sonuçları toplanmaktayken, itme iletişim modeli olan diğer kalıplarda, bir nesne, diğer nesnelerin sonuç üretmesini zorlayabilir. Burada, tasarıma bağlı olarak bir seçim yapmak gerekir.

SORU 3: Nesne Yönelimli Programlama Dersi Bütünleme İmtihanı Çözümü

Çözüme geçmeden önce, sizlerle görüşme imkanımız olmadığı için bir iki noktayı buradan açıklama ihtiyacı hissettim. Öncelikle, bilinmelidir ki bu sınavda sorulan bütün sorular, derste anlatılan konular kapsamında hazırlanmıştır. Bazı sorulara çözüm olabilecek derste birden fazla yol gösterilmiştir, buradaki cevaplarda bir tanesini seçip (genelde en uzun ve karmaşık olanını vermeye çalışıyorum) anlatıyorum ancak diğer yollar da doğru kabul edilecektir. Hatta derste anlatılmayan ama JAVA dilinde doğru olan yöntemler kullandıysanız (sonuçta derste anlatmadığım ama JAVA Tutorial içinde bulunan ve sizin ders dışı çalışarak öğrendiğiniz konular olabilir) bunları da doğru kabul edeceğim.

Ayrıca sınavlarınızın bir kısmını okudum ve genel olarak çalışmadığınız görülüyor. Nasıl bir sınav bekliyordunuz bilmiyorum ama bu dönem itibariyle hayal kırıklığına uğradığımı söyleyebilirim. Özellikle final sınavında birebir derste anlattığım kodları sormuştum. Finaldeki toplam 5 sorunun tamamını derste adım adım kodlayıp çalıştırıp sizlere anlatmıştım. Özellikle finaldeki 0. soruyu çözemeyip, ekrana ismini ve numarasını bastıramayan kişi sayısı sınıfın çoğunluğunu oluşturuyor. Bu durum da gösteriyor ki bu derse hiç çalışılmamış. Bütünleme sınavlarının henüz tamamını okumadım ama görülen durum, derste anlatılan konuları, bildiğiniz veya bilmenizi beklediğimiz konuları, farklı problemlere uygulayamadığınız. Örneğin Thread sorusunda, derste anlattıklarımın dışında yeni hiçbir bilgiye ihtiyacınız olmadığı halde genelde bu soruda durum çok kötü. Bu dönem itibariyle ilk defa üniversitenizde ders verdiğim için sanırım benim hatam sizlere fazla güvenmem ve çalışacağınıza olan inancımdı. Bu dönemden sonra, her hafta ödev, dönem projesi, quiz ve sıkı bir laboratuvar programı uygulamamız gerektiğini, ve sizleri çalışmaya zorlamamız gerektiğini anlamış bulunuyorum. Aşağıdaki çözümleri web sayfası olarak yayınlarken bazan hatalar olabiliyor. Kod hassas bir yazı olduğu için bu tip hatalara mahal vermemek adına, çalışan bir örnek kodu aşağıdaki linkten indirebilirsiniz:

[Sınavın çözüm kodu \(netbeans projesi olarak\) indirmek için tıklayın.](#)

Soru 0) Bir kesir (fraction) sınıfı kodlayınız. Kodunuza iki kesiri toplayan fonksiyonu ekleyiniz. (15 puan)

Öncelikle bilinmelidir ki rasyonel sayılar kümesi reel sayılar kümesinden farklıdır. Ayrıca kesirli sayıların paydası 0 olamaz. Sınavda bu problemi çözmek için istisna yakalama (exception handling) kullanabileceğiniz gibi, basit bir if ile kontrolünüz de doğru kabul edilecektir. Buna göre kodlanmış hali aşağıdaki şekildedir:

```
package butunleme;
class kesir{
    public int getPay() {
        return pay;
    }
    public void setPay(int pay) {
        this.pay = pay;
    }
    public int getPayda() {
        return payda;
    }
    public void setPayda(int payda) {
        if(payda==0) // kesirli sayıların paydası 0 olamaz.
            payda = 1;
    }
}
```

```

        this.payda = payda;
    }
    public kesir(int pay, int payda) {
        setPayda(payda);
        setPay(pay);
    }
    int obeb(int a,int b){
        if(b==0)
            return a;
        else
            return obeb(b,a%b);
    }
    public void topla(kesir k){
        int obeb = obeb(k.payda, payda);
        k.pay *= payda/obeb;
        pay *= k.payda/obeb;
        pay += k.pay;
        payda = (k.payda/obeb ) * (payda/obeb);
    }
    int pay;
    int payda;
}
public class Butunleme {
    public static void main(String[] args) {
        // TODO code application logic here
        kesir k = new kesir(3,4);
        kesir m = new kesir(4,5);
        k.topla(m);
        System.out.println("toplam : "+k.pay + " / "+ k.payda);
    }
}

```

Ekran çıktısı :

toplam : 31 / 20

Soru 1) 0. Soruda verdiğiniz cevabı kullanarak, bir çok terimli (polynom) sınıfı kodlayınız. Toplama (summation) , çarpma (multiplication), türev (derivation) ve çözme (evaluation) metotlarını kodlayınız. Kodunuzda, kapsülleme (encapsulation), çok şekillilik (polymorphism), yapıcı metotlar (constructor methods) bulunmalı ve kodunuzu çalıştıran main fonksiyonunu yazmalısınız. Ayrıca kodunuzu yazdıktan sonra konsol üzerinde nasıl derlenip çalıştırıldığını da yazınız.

Örnek:

$(\frac{3}{4}x^2 + \frac{1}{2}x + 5)$ çok terimlisi ile $(5x^3 + \frac{1}{3}x + 2)$ çok terimlisinin toplamı :

$(5x^3 + \frac{3}{4}x^2 + \frac{5}{6}x + 7)$ olarak bulunmalıdır. Yukarıdaki iki soru için tek bir kod yazabilirsiniz. (35 puan)

Sorunun çözümünü dosya olarak ekliyorum. Soru için genel olarak birden fazla çözüm yolu bulunuyor. Örneğin iki çok terimlinin toplamını 3. bir çok terimliye döndürmek veya mevcut çok terimliye bir parametre eklemek arasındaki tercih notlamayı etkilemez. Örnek kodda çarpma için üçüncü bir çok terimli döndürdüm ve toplama için mevcut çok terimliye ilave ettim, dolayısıyla iki yolda görülebiliyor. Ayrıca çok terimlileri sadeleştirmek veya sadeleştirmemek de notlama için önemli değil. Yine örnek çözümde terim isminde bir sınıf

kullanıp, çok terimlinin her terimini bu sınıftan bir nesne halinde vektöre yerleştirdim. Derste birden fazla nesneyi tutmak için iki yol görmüştük, vektör yerine diğer yol olan dizi kullanılabilir veya terim sınıfı yerine 3 farklı dizi paralel şekilde tutulabilirdi (ders veri yapıları olmadığı için bu tip, aslında veri yapısı hatası olan kodlamaları kabul edebiliriz). Yine örnek kodda, obeb (gcd) hesabı için özyineli (recursive) fonksiyon kullandım, iteratif kullanım da yapılabilirdi. Anlatmaya çalıştığım üzere sonucu doğru olan birden fazla yol bulunmakta ve hangisi ile yapılması istendiği, soruda belirtilmediği için önemli olmamaktadır.

Soru 2)Grafik arayüzü kullanarak (swing kütüphanesi), bir çok terimliyi okuyan ve yukarıdaki kodunuzu kullanarak bir nesne oluşturan kodu yazınız. (20 puan)

Bu soruda da farklı seçimler yapılabilir. Örneğin swing kütüphanesi kullanılarak basit bir arayüz tasarlayıp tek bir metin kutusundan çok terimli okunabilir. Okuduğunuz çok terimliyi daha sonra tokenize etmeniz gerekeceği için swing kısmı basit ama arkadaki string işleme kısmı kuvvetli bir kod yazabileceğiniz gibi, çok terimlinin her terimini okuyan ayrı birer kutu da yerleştirilebilir. Sonuçta amacımız bir çok terimliyi ekrandan okumak.

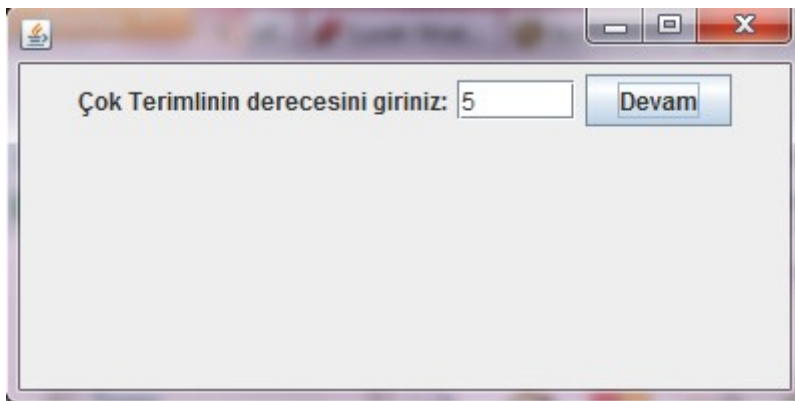
```
package butunleme;
import javax.swing.*.*;
import java.awt.*.*;
import java.awt.event.*;
public class ekran implements ActionListener{
    //www.bilgisayarkavramlari.com
    JButton dugme;
    JButton tamam;
    JFrame x;
    int derece;
    JTextField tf;
    JLabel jl;
    JTextField ctparam [];
    public ekran(){
        x = new JFrame();
        x.setLayout(new FlowLayout());
        jl = new JLabel("Çok Terimlinin derecesini giriniz:");
        x.add(jl);
        tf = new JTextField(5);
        x.add(tf);
        dugme=new JButton("Devam");
        x.add(dugme);
        dugme.addActionListener(this);
        x.setSize(400,200);
        x.setVisible(true);
    }
    public static void main(String args[]){
        new ekran();
    }
    public void actionPerformed (ActionEvent e){
        if(e.getSource()==dugme){
            derece = Integer.parseInt(tf.getText());
            x=new JFrame();
            x.setLayout(new GridLayout(derece+2,2));
            x.setSize(150,300);
            x.add(new JLabel("pay"));
            x.add(new JLabel("payda"));
            x.add(new JLabel("üst"));
            ctparam = new JTextField [derece*3];
```

```

        for(int i = 0;i<derece;i++){
            ctparam[3*i]=new JTextField();
            ctparam[3*i+1]=new JTextField();
            ctparam[3*i+2]=new JTextField();
            x.add(ctparam[3*i]);
            x.add(ctparam[3*i+1]);
            x.add(ctparam[3*i+2]);
        }
        tamam = new JButton("Tamam");
        x.add(tamam);
        tamam.addActionListener(this);
        x.setVisible(true);
    }
    if(e.getSource() == tamam){
        cokTerimli ct = new cokTerimli();
        for(int i = 0;i<derece;i++){
            ct.ct.add(new
terim(Integer.parseInt(ctparam[3*i+2].getText()),
            new kesir(Integer.parseInt(ctparam[3*i].getText()),
            Integer.parseInt(ctparam[3*i+1].getText()))));
        }
        ct.goster();
    }
}
}

```

Kodun ekran çıktısı aşağıdaki şekildedir:

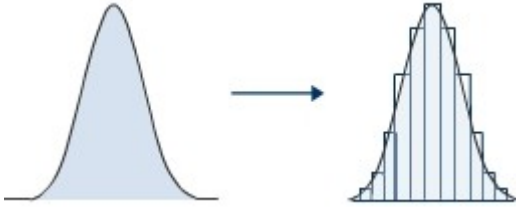


pay	payda	üst
2	3	4
5	6	5
3	1	2
3	4	3
1	1	1

Ve kod konsoldan aşağıdaki çok terimliyi basar:

$$2/3x^4 + 5/6x^5 + 3/1x^2 + 3/4x^3 + 1/1x^1$$

Soru 3) Bir çok teriminin x eksenini ile arasında kalan alanın hesaplanması için, kullanılan yöntemlerden birisi de, belirli aralıklarla x değerleri verilerek dikdörtgenler elde etmek ve bu dikdörtgen alanlarının toplamını hesaplamaktır.



Örneğin yukarıda, şekli verilen çok teriminin alanı hesaplanırken, belirli aralıklara sahip x değerleri verilmiş ardından bu değerlerin çok terimlideki sonuçları ile aralık miktarları çarpılarak elde edilen dikdörtgenlerin alanları bulunmuştur. Çok teriminin alanı, bu dikdörtgenlerin alanlarının toplamı olarak bulunabilir.

Bu hesaplama işlemi, karmaşık çok terimliler için, çok küçük aralıklarla hesaplama istendiğinde, bilgisayara yüksek miktarda yük getirebilmektedir. Sizden istenen hesaplama işlemi çok life bölmenizdir.

Çok lifli (multi threaded) kod yazarak hesaplanacak alanı en az 3 life (thread) bölünüz ve sonucu ekranda gösteren main fonksiyonunu kodlayınız.

Not: kolaylık olması için bu soruda, hesaplanacak alanın başlangıç ve bitiş değerlerini (x'in alt ve üst limitlerini), aralık değerini (delta) ve lif sayısını, kullanıcıdan alabilirsiniz.

(30 puan)

Yukarıdaki soru için yazılabilecek çözümlerden birisi aşağıdaki şekildedir:

```
package butunleme;
class alanLif extends Thread {
    cokTerimli ct ;
    int basla,bit;
    int delta;
    anaLif al ;
    public alanLif(cokTerimli ct,int basla, int bit, int delta,anaLif al){
        this.ct = ct;
        this.basla = basla;this.bit=bit;this.delta=delta;
        this.al = al;
    }
    public void run(){
        int hareket = basla;
        int toplam = 0;
        while(hareket < bit){
            toplam += ct.coz(hareket)*delta;
            hareket += delta;
        }
        al.getSonuc(toplam);
        System.out.println(" alan : "+ toplam + " aralık " + basla + " - "+
bit);
    }
}
public class anaLif {
    int toplam=0;
    int sonucGelen = 0;
    public anaLif(){
        cokTerimli ct = new cokTerimli();
        ct.ct.add(new terim(2,new kesir(2,1)));
        ct.ct.add(new terim(1,new kesir(3,1)));
        ct.ct.add(new terim(0,new kesir(5,1)));
        ct.goster();

        int basla = 0;
        int bit = 100;
        int ara= (bit-basla)/5,arabasla=0,arabit=ara;
        for(int i = 0;i<5;i++){

            new alanLif(ct,arabasla,arabit,1,this).start();
            arabasla+=ara;
            arabit += ara;
        }
    }
    public static void main(String args[]){
        anaLif al = new anaLif();
        while(al.sonucGelen < 5)
            try{ Thread.sleep(50); } catch(Exception e){}
        System.out.println("alanın toplam değeri:"+al.toplam);
    }
    public void getSonuc(int sonuc){
        toplam += sonuc;
        sonucGelen ++;
    }
}
```

Yukarıdaki kodda, alanLif isimli thread sınıfı, verilen aralık ve delta değeri için alan hesaplamaktadır. Bu sırada, çok terimli sınıfında bulunan çöz fonksiyonunu kullanmaktadır.

Ayrıca analif sınıfı, 5 adet (bu örnek için lif sayısını ben 5 kabul ettim, sayı farklı olabilir, soruda 3'ten fazla olması istenmiştir) lif (thread) oluşturmaktadır.

Son aşamada, main fonksiyonu altında, bütün threadler bitene kadar (ki bittiklerini hepsinin sonuç döndürmesinden anlıyoruz) bekliyoruz. Bütün threadler sonuçlarını analif'e döndürünce, her threadden gelen değeri alıp sonuç toplamını hesaplıyor ve ekrana basıyoruz.

Yukarıdaki kod için örnek ekran çıktısı aşağıdaki şekildedir:

$2/1x^2 + 3/1x^1 + 5/1x^0$ alan : 5610 aralık 0 – 20

alan : 102410 aralık 40 – 60

alan : 38010 aralık 20 – 40

alan : 198810 aralık 60 – 80

alan : 327210 aralık 80 – 100

alanın toplam değeri:672050

SORU 4: Java dilinde vektörler

Bu yazının amacı, Java dilindeki vektör sınıfının kullanılmasını ve yapısını anlatmaktır.

Java dilindeki vektörlerin yapısından bahsederek başlayalım. Klasik veri yapısı olarak [dizilerin \(array\)](#) ve [bağlı listelerin \(linked list\)](#) özelliklerini birleştirmiştir.

Bir vektör boyutu belli olmadan tanımlanıp içerisine veri konuldukça hafızada kapladığı yeri arttırmaktadır. Bu anlamda bağlı listelere benzer özellik göstermektedir. Yani bir vektörün boyutunun önceden tanımlanması gerekmez. Aslında vektörün boyu yoktur ve herhangi bir anda vektör için bir limit belirtilmez.

Boyutunu belirsiz tutabilmemizin yanında, bir vektörün herhangi bir elemanına doğrudan erişmek (random acces) de mümkündür. Bu sayede vektör yapısı diziler ve bağlı listelerden daha avantajlı bir hal almış olur.

Vektörler hakkında bilmemiz gereken son özellik ise, vektörlerin tipsiz olduğudur. Yani vektörler içlerinde istenilen tipte veri tutabilir. Bunu, JAVA dilinde bulunan nesne mirası sayesinde yapmaktadırlar. JAVA dilinde, bütün nesneler, Object sınıfından türemiştir. Yani aslında bütün nesneler ile Object sınıfı arasında bir [miras \(inheritance\)](#) ilişkisi bulunmaktadır.



Dolayısıyla JAVA dilindeki bütün nesneler, Object sınıfına yukarı [akış yapılarak \(upcasting\)](#) bu tipte tutulabilir. Aşağıdaki örneklerde bu durumu gösteren kodlar sunulacaktır.

Gelelim vektörlerin, JAVA dilindeki kullanılışlarına

Basit bir uygulama ile bu sınıfı tanımaya başlayalım.

```
13
14 import java.util.Vector;
15
16 class insan {
17     int yas;
18     int boy;
19     public insan(int yas,int boy){
20         this.yas=yas;
21         this.boy= boy;
22     }
23 }
24
25 public class vektorDeneme {
26     public static void main(String args[]){
27         Vector vektor = new Vector();
28         insan ali = new insan(10,100);
29         vektor.add(ali);
30         System.out.println("vektör boyutu: "+vektor.size());
31         System.out.println("vektörün 0. elemanı : " + vektor.elementAt(0));
32         System.out.println("0. elemanın insan olarak boyu : "+ ((insan)vektor.elementAt(0)).boy);
33         System.out.println("yasi : "+ ((insan)vektor.elementAt(0)).yas );
34     }
35 }
36 }
```

Öncelikle java dilindeki vektör sınıfı, java.util paketi altında bulunabilir. Dolayısıyla vektörler ile çalışırken kodun başına

import java.util.Vector;

yazmak gerekir. Yukarıdaki kodun 14. satırında da bu import işlemi ile koda başlanmıştır.

Kodun ana çalışan sınıfı (class) vektorDeneme isimli sınıftır ve bu sınıfın içerisinde bir adet main fonksiyonu bulunur. Bu fonksiyonun içinde victor isimli bir Vector tanımlanmıştır.

Ayrıca vektörün içerisinde tutulacak veri tipini belirlemek için bir oluşum sınıfı (composition class) tanımı yapılmış ve insan tipinin, boy ve yaş bilgilerinden oluştuğu kodlanmıştır. (kodun 14-23. satırları arası). Bu sınıf içerisinde bir [inşa fonksiyonu \(yapıcı fonksiyon, constructor\)](#) tanımlanmış ve iki tam sayı (int) alarak bir insanı tanımlamaya imkan sağlamıştır.

Kodun 28. satırında, insan tipinde bir ali nesnesi (object) üretilmiş ve bu nesne, 29. satırda, daha önceden tanımladığımız victor ismindeki vektöre yerleştirilmiştir.

Kodun 30. satırında, victor isimli vektörün boyutunun nasıl alındığı gösterilmiştir. Buna göre bir vektör nesnesinin .size() şeklinde metodu çağrılarak, boyutu alınabilir.

Ayrıca vektörün herhangi bir sıradaki elemanına erişmek için .elementAt() fonksiyonunu çağırarak ve bu fonksiyona parametre olarak kaçınıcı elemana erişilmek istendiğini vermek yeterlidir.

Ne yazık ki 31. satırdaki çağırma işlemi sonucunda JAVA anlamlı bir şey basamaz. Bunun sebebi, vektörün 0. elemanı olarak konulan nesnenin tipinin belirsiz olması ve şu anda nesne sınıfına (object) çevrilmiş olmasıdır. Vektörün 0. sırasında bulunan bu elemanın üyelerine (members) erişmek için öncelikle insan tipine geri çevrilmesi gerekir. Bunun için tip inkılabı yapıp nesnenin başına parantezler içerisinde (insan) yazılmıştır. Ancak bu çevirim yapıldıktan sonra boy ve yas üyelerine erişilebilmiş ve ekrana basılmıştır.

Yukarıdaki kodun çıktısı aşağıda verilmiştir:

- vektör boyutu: 1
- vektörün 0. elemanı : javaapplication21.insan@190d11
- elemanın insan olarak boyu : 100 yasi : 10

Görüldüğü üzere, çıktının ikinci satırında, JAVA nesneyi ekrana basmıştır. (aslında ekrana basılan nesnenin dizgiye (string) çevrilmiş ve serilenmiş halidir, daha fazla bilgi için [nesne serileme \(serialization\)](#) başlıklı yazıya bakabilirsiniz).

Vektörlerin, JAVA dilinde bir tip kısıtlaması ile kullanılması da mümkündür.

Örneğin yukarıdaki kodda bulunan victor isimli vektöre, yeni elemanlar eklenirken herhangi bir kısıt bulunmaz. İkinci elemanı olarak öğrenci, üçüncü elemanı olarak ders tipinde nesneler eklenebilir. Bu nesnelerin hepsi Object tipine çevrilerek yerleştirilir. Ancak biz kısıtlamak istersek bir [şablon tanımı \(template\)](#) yapmamız mümkündür.

```
Vector<insan>kisitli = new Vector<insan>();
```

Şeklindeki tanım, sadece insan tipinden nesneler alabilir. Farklı tiplerdeki nesnelerin bu vektöre konulması mümkün değildir. Aynı zamanda tip dönüşümü yapmak da gerekmez. Yani bir önceki kodda bulunan ve vektörün herhangi bir elemanının boyu veya yaşını bastırmak için başına eklediğimiz (insan) ibaresinin bu kodda bulunması artık gerekmez. Çünkü JAVA bu sınıfların tiplerini bilmektedir ve bu tiplere göre erişime izin verir.

SORU 5: JAVA da nihai uygulması (Final)

Bu yazının amacı, java dilinde bulunan final kelimesini açıklamaktır. Temel olarak final kelimesi, java dilinde 3 farklı yerde bulunabilir:

- Sınıf tanımında
- Metot (fonksiyon) tanımında
- Değişken tanımında

Sırasıyla bu durumları uygulamalar üzerinden görelim.

```
public final class sonsınıf{ }
```

yukarıdaki tanımın yapılması halinde, [sınıf \(class\)](#) nihai bir sınıf olur (final class) ve artık bu sınıftan [kalıtım yapılamaz \(miras, inheritance\)](#).

Örneğin yukarıdaki tanımdan sonra, aşağıdaki şekilde bir kod yazılması hatadır:

```
public class yenisınıf extends sonsınıf{ }
```

sebebi ise sonsınıf'ın nihai sınıf olarak tanımlanmış olmasıdır (final class).

Benzer şekilde bir metot tanımının başına final kelimesi yazılarak, metodun nihai olması durumunda, bu metodun [üzerine bindirme yapılamaz \(overriding\)](#).

Örneğin aşağıdaki kodu ele alalım:

```
public class deneme{  
    public final int toplama(int a,int b){  
        return a + b;  
    }  
}
```

Yukarıdaki kodda, toplama fonksiyonunun nihai hali kodlanmıştır. Bu sınıftan [miras alan hiçbir sınıf](#) toplama fonksiyonunu değiştiremez.

Örneğin aşağıdaki kod hatalıdır:

```
public class yenisınıf extends deneme{  
    public int toplama(int a, int b){ }  
}
```

Yukarıdaki bu kodun hatası, final olarak tanımlanan bir metodun [üzerine bindirme yapıyor olmasıdır \(overriding\)](#).

Son olarak bir değişkenin nihai olmasını inceleyelim. (final variables)

Bir değişken nihai tanımlandıysa, bu durumda içerisine atama sadece bir kere yapılabilir (assignment).

Örneğin aşağıdaki tanımlı ele alalım:

```
final int x = 10;
```

yukarıdaki koddan sonra, kodun herhangi bir yerinde x değişkenine yeni değer atanamaz. Örneğin

```
x= 7;
```

yazılması durumunda hata alınır.

Nihayi değişkenler, sabit değişkenlerden farklıdır (constant variables, const)

En önemli farkı ise, derleme zamanında (compile time) değişkenin değerinin bilinmesi gerekmektedir.

Örneğin aşağıdaki kodu ele alalım:

```
public class deneme{
    final int x;
    public deneme(int x){
        this.x = x;
    }
}
```

Yukarıdaki kod doğru ve çalışan bir koddur çünkü değişkene sadece bir kere, [yapıcı fonksiyon marifetiyle \(constructor, inşa fonksiyonu\)](#) atama yapılmıştır. Buna karşılık aşağıdaki kod hatalıdır:

```
public class deneme{
    const int x;
    public deneme(int x){
        this.x = x;
    }
}
```

Çünkü yukarıdaki kodda, değişkenin tanımı sırasında değer atanması yapılması gerekir. Örneğin

```
const int x = 10;
```

gibi ve bu atamadan sonra değer değiştirilemez. Örneğin yukarıdaki kodda bulunan final kelimesinin const kelimesi ile değişmesi halinde, [yapıcı fonksiyonda \(constructor\)](#) hala hata bulunmaktadır çünkü sabit bir değişkenin değeri değiştirilmeye çalışılmaktadır.

SORU 6: JAVA dilinde çoklu kalıtım (miras)

Bu yazının amacı, JAVA programlama dilinde bir sınıfın, birden fazla sınıftan nasıl miras aldığını anlatmaktır. Temel olarak [JAVA dilinde](#) doğrudan [çoklu kalıtım \(multiple inheritance\)](#) bulunmaz. Yani aşağıdaki gibi bir kod hatalıdır:

```
public class bilgisayar_mühendisliği_öğrencisi extends sayısal_öğrenci, üniversite_öğrencisi{
}
```

Yukarıdaki tanıma göre bir bilgisayar mühendisliği öğrencisi hem üniversite öğrencisidir hem de sayısal bölüm öğrencisidir. Ancak JAVA, yukarıdaki bu tanıma izin vermez ve extends kelimesinden sonra sadece tek bir sınıf ismi yazılmasına izin verir.

Buna rağmen, JAVA programlama dilinde [arayüz kullanarak \(interface\)](#) bir sınıfın birden fazla sınıftan miras almasını sağlayabiliriz.

Çoklu mirasa geçmeden önce tek bir sınıfı, arayüz kullanarak nasıl miras aldığımızı görelim ardından birden fazla sınıf için bu yöntemi geliştirelim.

```
interface sayısal_öğrencisi_olma_şartı{  
  
public void analitik_düşünmek();  
  
}
```

Yukarıda bir sayısal öğrencisi olma şartı arayüzü tanımlanmıştır. Bu arayüz, herhangi bir sınıf tarafından uygulanırsa (implements) bu sınıfın analitik_düşünmek şeklinde bir fonksiyonu bulunması gerekir.

Şimdi bir bilgisayar mühendisliği öğrencisinin bu şartı sağladığını JAVA dilinde modelleyelim:

```
Public class bilgisayar_mühendisliği_öğrencisi implements sayısal_öğrencisi_olma_şartı{  
  
public void analitik_düşünmek(){  
  
// fonksiyon içeriği...  
  
}  
  
}
```

Yukarıda kodda görüldüğü üzere, uygulama şartı olarak (implements) arayüzde bulunan fonksiyonun içeriği yazılmıştır. Bu kodlamadan sonra artık bilgisayar mühendisliği öğrencisinin, sayısal öğrencisi olma şartını sağladığını söyleyebiliriz.

Şimdi bilgisayar mühendisliğindeki öğrencileri sayısal öğrencisi olarak tanıtabiliriz:

```
public class sayısal_öğrencisi {  
  
double ortalaması;  
  
public sayısal_öğrencisi( sayısal_öğrencisi_olma_şartı s){  
  
s.analitık_düşünmek();  
  
}  
  
}
```

```
public class bilgisayar_mühendisliği_öğrencisi implements sayısal_öğrencisi_olma_şartı{

public void analitik_düşünmek(){

System.out.println("düşünüyorum...");

}

}

public class test{

public static void main(String args[]){

sayısal_öğrencisi ali = new sayısal_öğrencisi( new bilgisayar_mühendisliği_öğrencisi());

}

}
```

Yukarıdaki kod sonucunda ekranda, "düşünüyorum..." yazısı belirir. Bunun anlamı, ali isimli sayısal öğrencisinin, düşünmek isminde bir fonksiyonu olmasıdır.

Yazının başında da belirtildiği gibi JAVA doğrudan bir kalıtım ilişkisini desteklemez. Ancak yukarıdaki şekilde sınıflar arasında bir arayüz tanımlayarak bu iki sınıf arasında kalıtım ilişkisi kurulabilir.

Yukarıdaki kodda, yapılan eylem iki farklı yanlış anlaşılmaya açıktır. Genelde sık yapılan bu hataları aşağıda anlatmaya çalışalım.

1. Kalıtım değildir.

Yukarıdaki kod modeli, tam olarak kalıtım değildir. Sadece fonksiyonlar üzerinde tanımlıdır. Örneğin [arayüzler \(interface\)](#) üzerinde değişken tanımlanamaz.

2. HASA ve ISA arasındaki fark.

HASA ilişki modeli ile bir nesnenin diğer bir nesneyi içermesi, ISA ilişki modelinden farklıdır.

Bu durumu aşağıdaki şekil üzerinden açıklamaya çalışalım. Örneğin aralarında ISA ilişkisi bulunan iki sınıfı ele alalım:



Bu çizimi bir küme modeli ile modellemek istersek aşağıdaki şekilde bir sonuç çıkar:



Yukarıdaki model, nesne varlığı açısından kurulmuştur. Bu modele göre, insan, öğrenci'nin bir alt sınıfıdır. Yani öğrenci, insanda olan bütün özellikleri ve eylemleri kapsar ilave olarak kendisine özgü bazı özellikleri de bulunabilir. (bu modelleme aslında çok sağlıklı değildir çünkü genelde kavram-varlığı açısından karmaşaya sebep olur. Modele kavram-varlığı açısından bakılırsa aslında öğrenci, insanın bir alt kümesidir ve insan kümesinde farklı varlıklar da vardır (örneğin öğrenci olmayan diğer insanlar) ancak yukarıdaki model tekrar ediyorum nesne-varlığı açısından çizilmiştir).

Yukarıdaki modeli elde etmenin birinci yolu [miras ilişkisi kurmaktır \(kalıtım inheritance\)](#) diğer yolu ise HASA ilişkisi tanımlamaktır.

```
public class öğrenci extends insan{  
  
}
```

Şeklindeki bir tanım, insan sınıfındaki bütün tanımları otomatik olarak öğrenci sınıfına taşıyacak ve ardından yukarıda çizilen küme modelini gerçekleyecektir.

```
public class öğrenci{  
  
    insan a;  
  
}
```

Ancak bu gerekleme bir kalıtım ilişkisi olarak kabul edilemez. Bu gereklemede aslında iki farklı varlık vardır ve bu varlıklar arasında var oluştan kaynaklanan bir fark yoktur.

```
public class test{  
  
    public static void main(String args[]){  
  
        insan ali = new insan();  
  
        öğrenci veli = new öğrenci();  
  
        veli.a= ali;  
  
    }  
}
```

Yukarıdaki bu model sonucunda, ali, velinin bir alt kümesi halini alır ancak buradaki ali'nin varlığı ile veli'nin varlığı arasında bağlayıcılık yoktur. Örneğin veli nesnesi (object) yok edilmesi sonucunda ali yaşamına devam edebilir. Bu tip ilişkilere [münasebet \(aggregation\)](#) ismi verilmektedir ve yukarıdaki yazıda anlatılan arayüz marifetiyle kalıttan tamamen farklı bir yapıdır.

SORU 7: Nesne Akışı (Casting)

Nesne yönelimli programlama dillerinde, [sınıflar \(class\)](#) arasında [miras ilişkisi bulunması halinde \(inheritance\)](#), bu sınıflardan türetilen nesnelerin birbirine akıtılması durumudur.

Bu durumu aşağıdaki örnekler üzerinden anlamaya çalışalım:

```
public class insan{  
    int boy;  
    int kilo;  
}  
public class öğrenci extends insan{  
    int sınıf;  
    String bölüm;  
}
```

Örnek olarak verilen yukarıdaki iki sınıf arasında miras ilişkisi bulunmaktadır ve insan sınıfı, öğrenci sınıfının atası olarak kabul edilebilir.



Şimdi bu sınıflardan birer nesne tanımlayarak akış işlemini göstermeye çalışalım:

```
public class test{  
    public static void main(String args[]){  
        öğrenci ali = new öğrenci();  
        insan veli = new insan();  
        ali= veli;  
        veli = new öğrenci();  
    }  
}
```

Yukarıdaki kodun son iki satırında bulunan atama işlemleri birer akış işlemidir (casting) ve ali=veli atamasında, öğrenci tipinden bir nesne atfina, insan sınıfından bir nesne konulduğu için bu akışa, aşağı akış (down casting) ismi verilir.

Bu akışın tam ters yönlüsü olan ve son satırda yer alan atamada ise veli isimli insan tipinden bir nesne atfının içerisine öğrenci sınıfından bir nesne yerleştirilmektedir. Bu durum ise yukarı akış (up casting) olarak isimlendirilmektedir.

Klasik olarak nesne yönelimli programlama dillerinde aşağı akış (downcasting) yapılması halinde, nesnenin bütün bilgilerine erişilebilirken, yukarı akış (upcasting) yapılması durumunda nesnenin sadece nesne atfının üyeleri kadar bilgisine erişilebilir. Örneğin yukarıdaki kod için veli isimli değişken, insan tipinde olduğu için, öğrencinin boy ve kilosuna erişilebilirken, öğrencinin sınıfı veya bölümüne erişemez. Bu bilgiler daha önceden atanmışsa hafızada durur ama erişilemez. Tekrar öğrenci tipinde bir nesne atfı tarafından karşılanması durumunda bu bilgilere erişilebilir. Örneği yukarıdaki kodun sonuna aşağıdaki satır eklenseydi:

```
ali = veli;
```

Bu durumda veli tarafından atıfta bulunulan öğrenci, kendi tipinden bir nesne atfı tarafından gösterilecek ve veli tarafından erişilemeyen değerlere erişilebilecekti.

Atama işlemleri sırasında unutulmaması gereken bir husus da, [tip inkılabında olduğu gibi \(type casting\)](#) aşağı akış sırasında (downcasting) mutlaka tip dönüşümü yapılması gerektiğidir. Örneğin yukarıdaki kodda bulunan bu satır JAVA dili açısından hatalıdır:

```
ali= veli;
```

Ancak bunun tam tersi atama hatalı değildir:

```
veli = ali;
```

Yukarıdaki hatalı olan atamayı düzeltmek için tip dönüşümü yapılabilir:

```
ali = (öğrenci) veli;
```

şeklinde.

SORU 8: Nesne Kopyalama

Bu yazının amacı, nesne yönelimli programlama ortamlarında (object oriented programming) kullanılan iki temel kopyalama işlemini anlatmaktır.

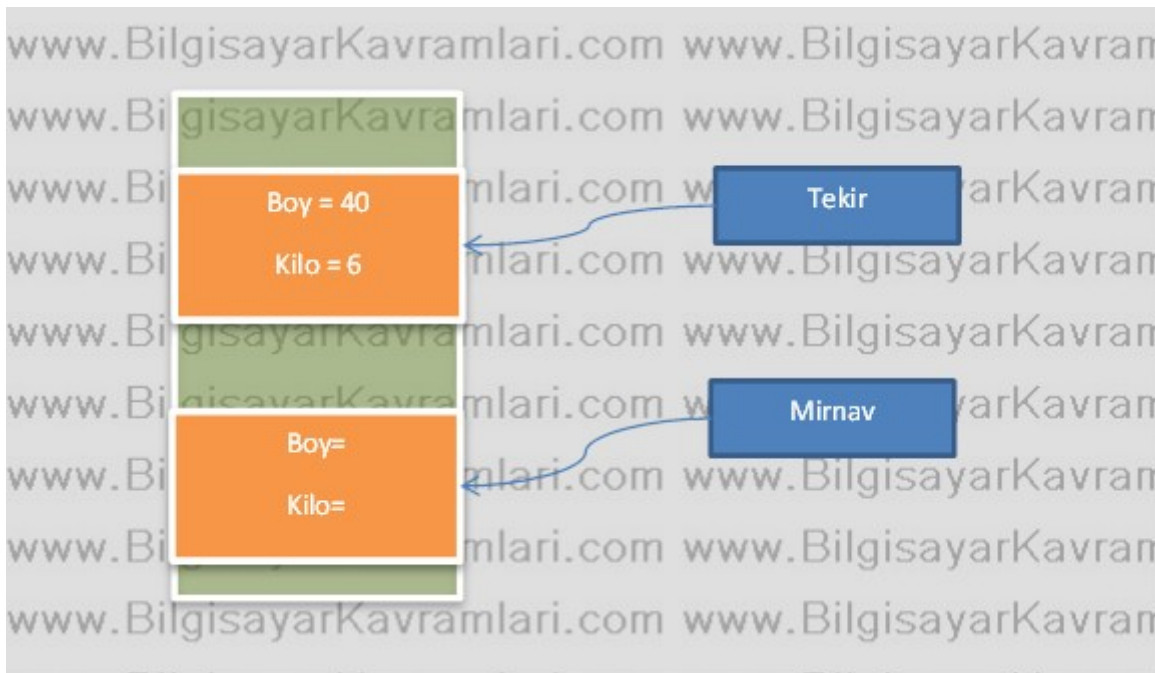
Literatürde bir nesnenin kopyalanması için geçen iki terim bulunur:

- Sığ kopyalama (shallow copying)
- Derin kopyalama (deep copying)

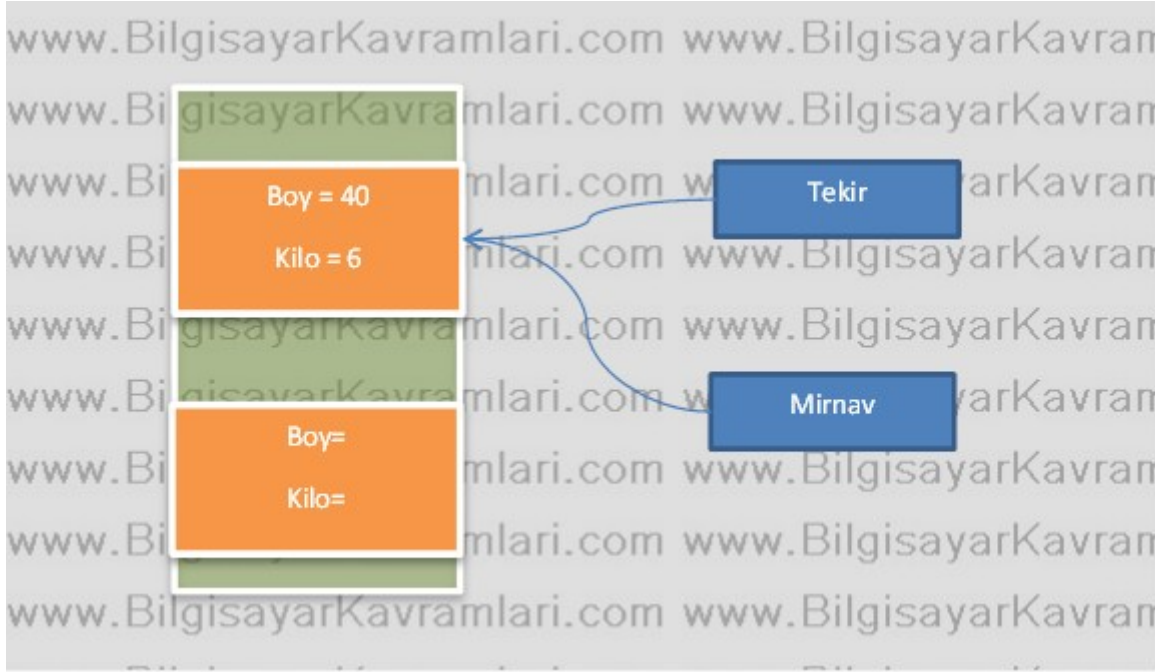
Sığ kopyalama basitçe [bir nesnenin referansını gösteren nesne atfının](#), farklı bir nesne atfına atanmasıdır. Örneğin aşağıdaki kodu inceleyelim :

```
public class kedi{  
    int boy;  
    int kilo;  
}  
  
public class test{  
    public static void main(String args[]){  
        kedi tekir = new kedi();  
        tekir.boy = 40;  
        tekir.kilo=6;  
        kedi mırnav = new kedi();  
        mırnav = tekir;  
    }  
}
```

Yukarıdaki kodun son satırında bulunan mırnav = tekir; ataması (Assignment) yapılmadan önce hafızdaki (RAM) durum aşağıdaki şekildedir:



İlgili atama satırı çalıştıktan sonra mırnav ile tekir aynı adresi göstermeye başlar:



Yukarıdaki bu atamaya sıg atama (shallow copying) ismi verilir çünkü hafızada iki [nesne atıfı](#) tarafından da aynı nesne gösterilmektedir ve birisinde yapılan değişiklik diğerini etkiler. Örneğin aşağıdaki satırı ele alalım:

```
mırnav.boy = 35
```

Bu satırdan sonra tekir isimli nesne atfının da boy üyesi 35 değerine sahip olmaktadır.

Peki, atamadan önceki mırnav tarafından gösterilen ve henüz değerleri bulunmayan nesne ne olur? Sonuçta bu nesnenin içerisindeki üyelere (members) değer konulmamış olsa da (boy ve kilo değeri verilmemiştir), new operatörü ile tanımlanmış bir hafız alanı söz konusudur ve bu alan hafızada yer kaplar. Sorunun cevabı programlama diline göre değişir. Örneğin java için garbage collector isimli bir işlem devreye girerek bu alanı temizler. C++ dilinde ise bu alan uzayda kaybolur (lost in space) ve C++ programcılarının bu tip hatalara karşı dikkatli olması ve free() fonksiyonunu kullanarak hafızadan kaldırması gerekir.

Gelelim derin kopyalamaya (deep copying)

Bu durumu da aynı sınıf tanımı üzerinden (kedi sınıfı) aşağıdaki kod ile anlamaya çalışalım:

```
public class kedi{  
  
    int boy;  
  
  
    int kilo;
```

```
public void kopyala(kedi x){

    this.boy = x.boy;

    this.kilo= x.kilo;

}

}

public class test{

    public static void main(String args[]){

        kedi tekir = new kedi();

        tekir.boy = 40;

        tekir.kilo=6;

        kedi mirnav = new kedi();

        mirnav.kopyala(tekir);

    }

}
```

Yukarıdaki kopyala satırı, parametre olarak aldığı, kedi sınıfından bir nesnenin içeriğini kendisine kopyalar. Örneğimizdeki mirnav nesnesi, tekir nesnesinin içeriğini kendisine kopyalamıştır.

Yukarıdaki bu işlemin hafızdaki (RAM) görüntüsü ise aşağıdaki temsili resimde verilmiştir:



Yukarıda gösterilen derin kopyalama işleminin bir benzerini yapmak için [yapıcı fonksiyonlardan \(constructor function, inşa fonksiyonu\)](#) faydalanılabilir. Bunun için aşağıdaki kodda olduğu gibi bir yapıcı fonksiyon tanımlamak yeterlidir.

```
public class kedi{  
  
    int boy;  
  
    int kilo;  
  
    public kedi(kedi x){  
  
        this.boy =x. boy;  
  
        this.kilo= x.kilo;  
    }  
}
```



```
    }

}

public class test{

    public static void main(String args[]){

        kedi tekir = new kedi();

        tekir.boy = 40;

        tekir.kilo=6;

        kedi mırnav = new kedi(tekir);

    }

}
```

Yukarıdaki kodda bulunan bu yapıcı fonksiyona ayrıca özel olarak kopyalama yapıcısı (copy constructor) ismi de verilmektedir.

SORU 9: Nesne Yönelimli Programlama Dersi Quiz Çözümü

Quiz soruları ve çözümleri aşağıdaki şekildedir:

1. Bir oylama için program yazmanız isteniyor. Oylamaya katılan 5 aday bulunuyor ve bu adayların numarası (1'den 5'e kadar bir sayı) oy pusulasına yazılarak oy kullanılıyor. Programınızda 10 adet oyu okuyup ekrana adayların aldıkları oy miktarını basan bir program yazınız. Kullanılan oyun 1-5 arasında olmaması halinde problemi istisna yakalama ile çözünüz.

Çözüm

Oylama programında basitçe her oyun tutulduğu bir oy pusulası ve bir de oy verilen adaylar olacağını düşünebiliriz. Bu durumda oy pusulası ve aday isimli iki adet sınıf tanımlayıp bu sınıfların kodlamasını aşağıdaki şekilde yapabiliriz.

```
1  #include <iostream>
2  using namespace std;
3  class aday{
4  private:
5      int adayno;
6      int oysayisi;
7  public:
8      aday(){
9          oysayisi=0;
10     }
11     int getOysayisi(){
12         return oysayisi;
13     }
14     int getAday(){
15         return adayno;
16     }
17     void setaday(int adayno){
18         this->adayno= adayno;
19     }
20     void oysayisiniArttir(){
21         oysayisi++;
22     }
23 };;
```

Yukarıdaki kodda, aday bilgisi tutuluyor. Programımızda olması gereken bilgi, adayın numarası ve oy sayısıdır. Biz de bu bilgileri tutan iki adet değişken tanımlıyor ve bu değişkenlerin getter/Setter fonksiyonlarını kodluyoruz. Ayrıca sınıfın [inşa fonksiyonunda \(constructor\)](#), başlangıç olarak oy sayısını 0 yapıyoruz. Son olarak adayların oy sayısını arttıran bir metot yukarıdaki kodun sonunda yer almaktadır. Basit bir arttırma işlemidir.

Oy pusulasını tutan kod ise aşağıdaki şekilde kodlanmıştır:

```
1  #include <iostream>
2  using namespace std;
3  class oypusulasi{
4  private:
5      int oy;
6  public:
7      int getOy(){
8          return oy;
9      }
10     void setOy(int o){
11         oy = o;
12     }
13 };
```

Yukarıdaki kodda, görüldüğü üzere, oy pusulasında sadece kime oy verildiği bilgisi tutulmuştur. Bu bilgiyi [kapsülleme gereği getter ve setter fonksiyonları](#) ile erişilebilir halde sınıfta kodluyoruz.

Son olarak kodumuzu test etmek için bir main fonksiyonu yazalım:

```

3  #include "aday.h"
4  #include "oypusulasi.h"
5  #include "hatalioy.h"
6  //www.bilgisayarkavramlari.com
7  #include <iostream>
8  #include <conio.h>
9  using namespace std;
10
11 int oyoku(){
12     int oy;
13     cin >> oy;
14     if(oy>=5||oy<=1)
15         throw hatalioy();
16     return oy;
17 }
18
19 int main(){
20     aday adaylar[6];
21     oypusulasi oylar[10];
22     for(int i = 0;i<10;i++){
23         try{
24             oylar[i].setOy(oyoku());
25             adaylar[oylar[i].getOy()].oysayisiniArttir();
26         }
27         catch(hatalioy h){
28             cout << "girilen oy hatalidir" <<endl;
29         }
30     }
31     for(int i = 1;i<6;i++){
32         cout << i << ". adayın oy sayisi: " << adaylar[i].getOysayisi() << endl;
33     }
34     getch();
35 }

```

Oy okuma işlemi için “oyoku” isimli fonksiyon yazılmıştır. Bu fonksiyonun özelliği, oyları okurken yanlış bir oy girilmesi durumunda, (oyların 1 ile 5 arasında olacağını hatırlayınız) bir istisna fırlatmasıdır. Bu istisna, try bloğu içerisinde çağrıldığı yerde yakalanır ve kodun 27nci satırında bulunan catch bloğu ile ekrana bir hata mesajı basılır. Şayet oy başarılı bir şekilde okunabilirse, bu durumda ilgili oy pusulasına (ki toplam 10 adet pusula bulunuyor) oy değeri kaydedilir. Şayet bir istisna burada oluşmamışsa kodun 25. Satırına geçebiliriz. İstisna oluşması durumunda kodun 25. Satırı hiç çalışmadan catch bloğuna gidilir.

Kodumuzun 25. satırında, adayların tutulduğu nesne dizisinin (object array) ilgili elemanının (hangi elemana oy verildiyse) oy miktarını 1 arttıran oysayisiniArttir() fonksiyonu çağrılır.

Son olarak kodun 31-33 satırları arasında, adayların oy durumları ekrana basılır.

Yukarıdaki kodda bulunan ve kendi istisnamız olan hatalioy sınıfını aşağıdaki şekilde kodlayabiliriz:

```

1 #include <iostream>
2
3 class hataliOy{
4 private:
5     int oy;
6 public:
7     hataliOy(){
8     }
9     hataliOy(int oy){
10         this->oy=oy;
11     }
12     int getOy(){
13         return oy;
14     }
15 };

```

1. Bir banka yazılımı için aşağıdaki özelliklerle program yazmanız isteniyor. Sistemde tutulacak veriler:

1. Hesap sahibinin ismi
2. Hesap numarası
3. Bakiye
4. Hesap tipi (vadeli / vadesiz)

Yukarıdaki hesap bilgileri için aşağıdaki işlemleri yapmanız isteniyor:

1. Hesapların ilk değer olarak bakiyesinin 0 atanması gerekir.
2. Hesaba para yatırılabilir
3. Hesaptan para çekilebilir
4. Hesap bilgisi görüntülenebilir (sahibinin ismi, numarası, tipi ve bakiyesi)

1. 2. Sorudaki programınızı, 10 müşteriye hizmet verebilecek şekilde genişletin.
2. 2. Sorudaki programınızı, hesap hareketleri görüntülenecek şekilde genişletin. Buna göre bir hesap hareketinin, tarihi, saati, miktarı ve hesap numarası bulunur.

Çözüm:

Yukarıdaki sorunun çözümü için basitçe bir hesap nesnesi oluşturmamız yeterli olacaktır. Yukarıdaki 4. soru için hesap hareketlerini tutan ilave bir sınıfa daha ihtiyacımız olacak. Öncelikle, hesapların tutulduğu bir hesap sınıfını programlayarak başlayalım.

```

7 class hesap{
8 private:
9     int hesapno;
10    float bakiye;
11    string hesapSahibi;
12    int tipi; // 0 vadeli, 1 vadesiz
13    hesaphareketi h[100];
14    int hhsayisi;

```

Soruda tanımlandığı üzere, her hesap için hesapno, bakiye , hesap tipi ve hesap sahibi bilgilerini tutuyoruz. Ayrıca hesap hareketlerinden oluşan ilave bir dizi, her hareketi ayrıca kayıt altına alıyor.

Hesap sınıfımızın constructor fonksiyonunda, hesap bakiyesini 0 olarak atıyoruz.

```

16 hesap(){
17     bakiye=0;
18     hhsayisi = 0;
19 }

```

Ayrıca sınıfımızdaki hesap hareketlerini tutan dizinin kaçınıcı elemanında olduğumuz bilgisi, hhsayisi isimli değişkende durmaktadır. Bu değişken de henüz hesap hareketi olmadığı için 0 olarak atanıyor.

Sırasıyla hesap sınıfındaki metotların kodlamasını inceleyelim:

```

20 void hesapDetay(){
21     cout << "Hesap No:" << hesapno << " bakiye : " << bakiye << " hesapsahibi : " << hesap
22
23     if(tipi == 0)
24         cout << " vadeli";
25     else
26         cout << " vadesiz";
27     cout << endl << "Hesap Hareketleri" << endl << "-----"<<endl;
28     for(int i = 0;i<hhsayisi;i++){
29         cout << h[i].getmiktar() << endl;
30     }
31
32 }

```

Yukarıda, hesap detaylarını ekrana basan fonksiyon verilmiştir. Fonksiyondaki 21nci satırda bulunan cout komutu ile ekrana hesabın temel 3 bilgisi bastırılmıştır. Ardında hesabın tipi yazıya çevrilerek bastırılmıştır. Hesap tipini tutmak için kullanılan int tipi, kullanıcı açısından anlamsız olabilir o yüzden yazıya çevirme işlemi yapılır.

Son olarak 27-30 satırları arasında, hesap hareketlerini bastırmak için, mevcut hesap hareketi sayısını tutan değişken kadar dönen döngü içerisinde her hesap hareketinin miktarı basılmaktadır.

Bu çıktının örneği aşağıda verilmiştir.

```

Hesap No:123 bakiye : 700 hesapsahibi : Ali Demir vadesiz
Hesap Hareketleri
-----
1000
-250
-100
50

```

Yukarıdaki ekran görüntüsünde, örnek olarak bir hesabın detayları basılmış ve bu hesapta yapılan hesap hareketleri altında gösterilmiştir.

Hesap hareketi olarak kabul edilebilecek iki temel işlem para çekmek ve para yatırmak işlemleridir. Bunların kodlamasını aşağıda anlatalım:

```

60 void paracek(float miktar){
61     if(bakiye-miktar<0)
62         throw yetersizBakiye(bakiye-miktar);
63     h[hhsayisi].sethesapno(hesapno);
64     h[hhsayisi].setmiktar(miktar*-1);
65     hhsayisi++;
66     bakiye -=miktar;
67 }
68 void parayatir(float miktar){
69     h[hhsayisi].sethesapno(hesapno);
70     h[hhsayisi].setmiktar(miktar);
71     hhsayisi++;
72     bakiye += miktar;
73 }

```

Para çekme işlemi, belirtilen miktar kadar bakiyeyi azaltmamaktadır. Bu işlem sırasında öncelikle miktarın yeterli olup olmadığı kontrol edilir. Şayet miktar yeterli bulunursa bakiye azaltılır, şayet yeterli bulunmazsa bu durumda bir istisna (exception) fırlatılır. Burada fırlatılan istisna, proje kapsamında bizim tanımladığımız yetersizBakiye istisnasıdır.

Yetersi bakiye istisnası, basit bir şekilde bakiye bilgisini de içinde saklayan bir sınıf (class) olarak tasarlanabilir:

```

1  class yetersizBakiye{
2  private:
3      float miktar;
4  public:
5      yetersizBakiye(){
6      }
7      yetersizBakiye(float miktar){
8          this->miktar = miktar;
9      }
10     float getMiktar(){
11         return miktar;
12     }
13 };

```

Hesap hareketlerinin içerisine, ayrıca miktar konulurken -1 değeri ile çarpılmaktadır. Bunun anlamı, hesap hareketinin eksi değer olarak tutulmasıdır.

Benzer kodlama, para yatırma fonksiyonunda da kullanılmaktadır. Yatırılan miktar, hesap hareketlerine işlendikten sonra, bakiye değeri parametre olarak verilen miktar kadar arttırılmaktadır.

Hesap hareketlerinin detayı için sınıf kodlamasını aşağıdaki şekilde yazabiliriz:

```

1 #include <iostream>
2 #include <string.h>
3
4 using namespace std;
5
6 class hesaphareketi{
7 private:
8     int hesapno;
9     float miktar;
10    string tarih;
11    int saat;

```

Bu sınıf, basit bir şekilde hesapta yapılan işlemleri tutmak için tasarlanmıştır. İlgili üyelerin getter / setter fonksiyonlarının yazılması yeterlidir.

SORU 10: C++ dilinde üzerine bindirme

Bu yazının amacı, C++ dili için fonksiyon üzerine bindirme (function overriding veya method overriding) konusunu açıklamaktır.

Üzerine bindirme işlemi, basitçe [aralarında miras \(inheritance\) ilişkisi bulunan iki sınıf](#) için kullanılabilir. C++ dilindeki, nesneler arasında kurulan bu ilişki modeli hakkında daha detaylı bilgi için, bu bağlantıya tıklayabilirsiniz.

Öncelikle iki sınıf tanımlayıp bu iki sınıf arasında miras ilişkisi kurulmalıdır. Bu miras ilişkisinden sonra, sınıflardan birisi, diğerine bazı fonksiyonlarını miras bırakacaktır. İşte fonksiyon üzerine bindirme de tam olarak burada devreye girer. Şayet miras alınan fonksiyonlar alt sınıfta yeniden tanımlanmak isteniyorsa, bu duruma üzerine bindirme (overriding) ismi verilir.

Örneğin, dikdörtgen olarak tanımlı bir sınıfımız bulunsun.

```

class dikdortgen{
public:
    int x,y;
    int boy,en;
public:
    int alan();
};
int dikdortgen::alan(){
    return en *boy;
}

```

Yukarıdaki bu tanıma uyan diğer bir iki boyutlu şekil ise, kare'dir. Bilindiği üzere kare kenarları birbirine eşit bir dikdörtgendir. O halde kare sınıfını aşağıdaki şekilde tanımlayabiliriz.

```

class kare:public dikdörtgen{ // kare sınıfı, dikdörtgen sınıfından miras alır.
public:
    int alan();
};
kare::alan(){
    return en * en ;
}

```

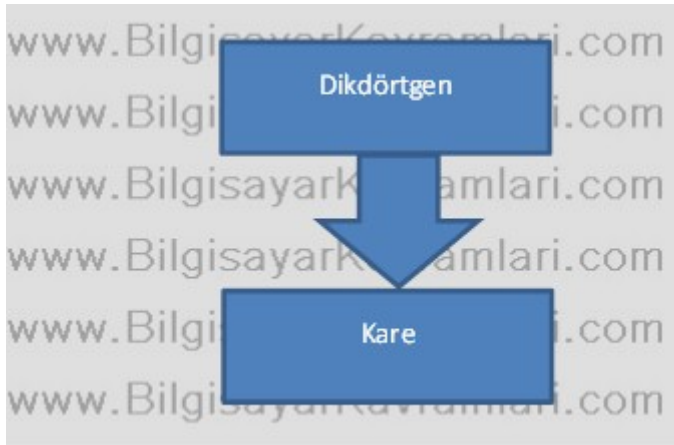

};

Yukarıdaki kare sınıfında görüldüğü üzere, her kare'nin zaten sahip olduğu x,y koordinatları ve en, boy bilgisi yeniden kodlanmamış bu bilgiler miras ile alınmıştır. Alan hesaplamak için yazılan alan() fonksiyonu ise, yeniden yazılmıştır. Bunun sebebi kare sınıfında sadece tek kenar bilgisinin yeterli olması ve bu durumda alan hesabının değişmesidir.

Şayet kare sınıfı için, yeniden alan fonksiyonu yazılmasaydı, dikdörtgen sınıfının alan fonksiyonunu kullanabilirdi. Ancak buradaki yeniden yazılan aynı isme ve parametrelere sahip olan alan fonksiyonu, üzerine yükleme işlemi yapmıştır (overriding).

Yukarı Atamak (Upcasting)

Miras ilişkisi içerisinde bulunan iki sınıftan tanımlanan iki nesne arasında atama yapılabilir. Örneğin yukarıdaki kodu ele alalım ve bu kodla oluşan sınıf diyagramını çizmeye çalışalım:



Yukarıdaki bu çizimden anlaşılabileceği üzere, her kare bir dikdörtgendir ifadesi kullanılabilir. Bu ifade aynı zamanda dikdörtgen sınıfının, kare sınıfının atası olduğu, veya kare sınıfının, dikdörtgen sınıfının bir çocuğu olduğu şeklinde de yorumlanabilir.

Yukarıdaki bu ilişki şeklinde, bizim için dikdörtgen sınıfı üst, kare sınıfı ise alt sınıf olarak kabul edilebilir. Bu durumda kare sınıfından bir nesnenin, dikdörtgen sınıfından bir nesneye atanmasına upcasting ismi verilir. Burada nesne olduğu gibi kalmakta ancak [nesne göstericisi \(object referer\)](#) farklı bir sınıftan olmaktadır. Bu işlemin kod olarak karşılığı aşağıdaki şekildedir:

```
dikdörtgen *d = new kare();
```

Görüldüğü üzere kare sınıfından tanımlanan bir nesne, dikdörtgen sınıfından bir nesne göstericisi tarafından gösterilmiştir. Bu kullanım, tip belirsizliği yaşanan durumlarda sıklıkla konu olur. Örneğin kodlamak istediğimiz fonksiyon, bir parametre olarak dikdörtgen alıyor ve bu dikdörtgenin orjine (merkez 0,0 noktasına) uzaklığını ölçüyor olsun. Bu durumda fonksiyonumuz aşağıdaki şekilde olabilir:

```
int mesafe(dikdörtgen *d){
    return sqrt(x*x+y*y);
}
```

Yukarıdaki bu fonksiyon, her türlü dikdörtgeni kabul etmektedir. O halde bu fonksiyona bir dikdörtgen nesnesi veya bir kare nesnesi, parametre olarak verilebilir.

Upcasting işlemi, sanal fonksiyonların kullanımı açısından önem taşır.

Sanal Fonksiyonlar (Virtual Functions)

Yukarı atama işlemi sırasında yaşanan bir problem, yukarı yükleme işlemi sonrasında, nesnenin bir fonksiyonu çağırıldığında, ata sınıfından mı çocuk sınıfından mı fonksiyonun çalışacağıdır.

Örneğin, yukarıdaki miras konusunu ele alalım. Dikdörtgen sınıfından bir nesne göstericisinin içerisine, kare sınıfından bir nesne atanmıştır. Şimdi bu atamadan sonra

```
d->alan();
```

şeklinde bir fonksiyon çağırımı yapılırsa, acaba kare sınıfının mı, dikdörtgen sınıfının mı alan fonksiyonu çalışacaktır?

Herhangi özel bir durum olmaksızın, yukarıda yazdığımız kodlar için, d->alan() çağırımı, ata sınıfında (base class, ancestor, parent) olan alan fonksiyonunu çalıştıracaktır.

Peki acaba çocuk sınıftaki alan fonksiyonun çalışmasını isteseydik kodu nasıl değiştirmemiz gerekirdi?

Sınıf tanımı sırasında, ata sınıfta bulunan fonksiyonu tanımlarken başına virtual eklenmesi, bu sınıfın ileride bir şekilde üzerine yükleme yapıldığında (override), ve ayrıca yukarı atama yapıldığında (upcasting), üzerine yükleme yapılan fonksiyon tarafından çalıştırılacaktır.

Örnek olarak yukarıdaki sınıf tanımını aşağıdaki şekilde değiştirirsek,

```
class dikdortgen{
public:
    int x,y;
    int boy,en;
public:
    virtual int alan(){
        return en*boy;
    }
};
```

Bu yeni tanımdan sonra yukarı atama yapılırsa,

```
dikdortgen *d = new kare();
```

artık d göstericisinin alan fonksiyonu çağırıldığında,

```
d->alan();
```

çalışacak olan alan fonksiyonu, kare sınıfındaki alan fonksiyonudur. Çünkü dikdörtgen sınıfında bulunan alan fonksiyonu sanal bir fonksiyondur (virtual function).

SORU 11: C++ üzerinde çok şekillilik

C++ programlama dili, nesne yönelimli olmasından dolayı çok şekillilik (polymorphism) özelliğini destekler. Çok şekillilik (polymorphism) bir işin farklı şekillerle yapılabilmesidir. Örneğin bir metodun farklı şekillerde çağrılabilmesi, çok şekillilik özelliği ile mümkündür. Bu yazıda, metodların ve işlemlerin (operators) üzerine [yüklenmesi \(overloading\)](#) anlatılacaktır.

Amacımız bir fonksiyonu farklı şekillerde çalıştırabilmektir. Nesne yönelimli özelliği olmayan dillerde, bir fonksiyon tek bir isimle bir kere tanımlanabilir. Örneğin aşağıdaki dikdörtgenin alanını hesaplayan fonksiyonu ele alalım:

```
int alan(int a, int b){
    return a*b;
}
```

Yukarıdaki fonksiyon gayet başarılı bir şekilde a ve b kenarları verilen dikdörtgenin alanını hesaplar. Ancak aynı alan hesabını örneğin float sayılar için düzenlemek istesek ve verilen değere göre çalışmasını istesek:

```
float alan (float a, float b){
    return a*b;
}
```

Şeklinde yazılan fonksiyon, aldığı parametreler itibarıyla, farklı bir fonksiyon iken, nesne yönelimli özelliği bulunmayan dillerde, bu iki fonksiyonu aynı anda kodlamamız mümkün olmaz.

Benzer şekilde, her kare bir dikdörtgendir mantığıyla, tek kenar uzunluğu verildiğinde alan hesaplamak istersek

```
int alan(int a){
    return a*a;
}
```

Fonksiyonu, bir karenin alanını başarılı bir şekilde hesaplayabilirken, daha önceden tanımlı olan fonksiyonlarla çakışmaktadır.

C++, JAVA veya CSharp gibi diller için yukarıdaki bütün bu fonksiyon tanımları farklı anlama gelmektedir ve tek bir [sınıf \(class\)](#) içerisinde tanımlanabilir. Örneğin aşağıdaki C++ sınıfını ele alalım

```
class dikdörtgen{
public:
    int a,b;
public:
    int alan(int a,int b);
    float alan(float a,float b);
    int alan(int a);
};
```

Yukarıdaki bu tanımlama, C++ dili açısından problemsiz bir tanımdır ve hangi alan fonksiyonunun çalıştırılacağı, verilen parametrelere göre algılanır.

Örneğin yukarıdaki tanımlardan sonra aşağıdaki main fonksiyonunu yazarsak:

```
int main(){
    dikdörtgen *d = new dikdörtgen();
    cout << d->alan(3,4) << endl << d->alan(2.2,7.8) << endl << alan(5) <<
endl;
}
```

Yukarıdaki kod, sorunsuz bir şekilde üç ayrı fonksiyonu çağırarak çalıştırır.

Yukarıdaki örnekte yapılan işlem, alan isimli [fonksiyonun üzerine yüklenmesidir \(overloading\)](#). Yani aynı isme sahip birden fazla fonksiyon tanımlanmış ve bu fonksiyonların parametre tipleri veya parametre sayılarına göre farklı durumlar için yeniden kodlanmıştır. Bu anlamda üzerine yükleme işlemi bir çok şekillilik işlemidir (polymorphism) ve aynı işi yapan birden fazla şekilde çalışmaya izin verir.

Üzerine yükleme işlemi, [fonksiyonların yapıcıları \(constructors\)](#) için de kullanılabilir. Örneğin dikdörtgen tanımı sırasında farklı bilgiler ile bir dikdörtgenin tanımlanacağını düşünelim.

- Dikdörtgen hakkında hiçbir bilgi vermeyebiliriz
- Dikdörtgenin kenarlarını verebiliriz
- Dikdörtgenimiz özel olarak bir kare olabilir ve tek kenarını verebiliriz.

Yukarıdaki bu üç farklı durum için, dikdörtgen tanımını değiştirmemiz gerekir. Bu durumda olası, [yapıcı fonksiyonlarımız \(constructors\)](#) aşağıdaki şekilde olabilir :

public:

```
dikdörtgen(){
    a=0;b=0;
}
dikdörtgen(int a){
    this->a = a; b= 0;
}
dikdörtgen(int a,int b){
    this->a=a;
    this->b=b;
}
```

Benzer bir üzerine yükleme işlemi de, işlemler (operators) için yapılabilir.

SORU 12: C++ Nesne Yönelimli Programlama İlişki Türleri

Bu yazının amacı, C++ dili için, nesneler arasında kurulabilecek olan ilişki tiplerini açıklamaktır.

Temel olarak bir nesne kendi özellikleri ve metotları bulunan bir varlıktır. Nesne yönelimli programlama modelinde, nesnelerin özelliklerini paylaşmak veya diğer nesnelerin metotlarına erişmek için birbirleri ile iletişime girmeleri gerekir.

Bu anlamda inceleyeceğimiz 4 temel ilişkiden bahsedebiliriz. Bu ilişki tipleri nesnelerin özlük yapılarını bozmadan, dışarıdaki bir nesne ile kurdukları ilişki türüdür.

- Sınıflar (Class) arasında kurulan “bir çeşit” (a-kind-of) ilişkisi
- Nesneler (object) arasında kurulan “bir” (is-a) ilişkisi
- Sınıflar (Class) arasında kurulan “parçası” (part-of) ilişkisi veya “sahiplik” (has-a) ilişkisi
- Miras (Inheritance) ilişkisi

Yukarıdaki bu 4 tip ilişki için aşağıdaki örnekleri verebiliriz.

Öncelikle iki adet sınıf tanımını aşağıdaki şekilde yapalım :

Nokta sınıfı aşağıdaki şekilde kodlanmış olsun:

```
class Nokta {  
    özellikler:  
    public :  
        int x, y  
  
    metotlar:  
        setX(int yeniX)  
        getX()  
        setY(int yeniY)  
        getY()  
}
```

Ayrıca çember için aşağıdaki kodu yazalım:

```
class Çember {  
    özellikler:  
        int x, y,  
        Çap  
  
    metotlar:  
        setX(int yeniX)  
        getX()  
        setY(int yeniY)  
        getY()  
        setÇap(yeniÇap)  
        getÇap()  
}
```

Yukarıdaki bu sınıf tanımlarından sonra artık ilişki tiplerimizi belirleyebiliriz.

“Bir Çeşit” ilişkisi

Bu ilişki tipinde, sınıflar arasında birinin diğer çeşitten olması vurgulanır. Örneğin nokta ile çember arasında “bir çeşit” ilişkisi bulunuyorsa ve biz “çember bir çeşit noktadır” diyebiliyorsak. Bu tanımları iki sınıf üzerinde yapabiliriz.



Bu çıkarım, yukarıdaki örnek için doğrudur. Bunun sebeplerini aşağıdaki şekilde sayabiliriz:

- İki sınıfta da x ve y özellikleri bulunur. Nokta sınıfı için x ve y değeri, noktanın 2 boyutlu uzaydaki konumunu belirlerken, Çember sınıfı için, çemberin merkezinin konumunu belirler. Dolayısıyla, x ve y iki sınıf için de aynı anlamdadır.
- İki sınıfta da x ve y özellikleri için getter / setter (alıcı verici) metotları tanımlanmıştır ve iki sınıf için de bu metotların anlamı aynıdır.
- Çember sınıfında, ilave olarak bir çap bilgisi bulunur.

Yukarıdaki bu listeye göre, çember sınıfının bir çeşit nokta olduğunu ve noktanın bütün özelliklerini taşıdığını söylemek mümkündür.

“Bir” ilişkisi

Bu ilişki tipi, yukarıda anlatılan “Bir Çeşit” ilişkisinin tamamen aynısıdır. Tek farkı, sınıflar arasında değil de nesneler arasında tanımlı bir ilişki modeli olmasıdır. Yani yukarıdaki tanımlı sınıflardan üretilen nesneler üzerinde kurulabilir. Bu bağlantı türü, literatürde ISA veya is-a şeklinde yazılan ve Türkçeye “bir” şeklinde çevrilen ilişkidir.

Örneğin, çember nesnesi, özellikleri itibariyle bir noktanın çapı olan halidir. Dolayısıyla çember **bir** noktadır. Şeklinde kurulan cümledeki “bir” kelimesi bu bağlantı tipini ifade eder.



Yukarıdaki yeni şekilde dikkat edileceği üzere çember ve nokta kutuları, köşeleri yuvarlak şekilde çizilmiştir. Bu UML standardı olarak, nesneleri ifade eden gösterim biçimidir. Bir önceki şekilde olduğu gibi köşeli kutlar, sınıf ifade eder.

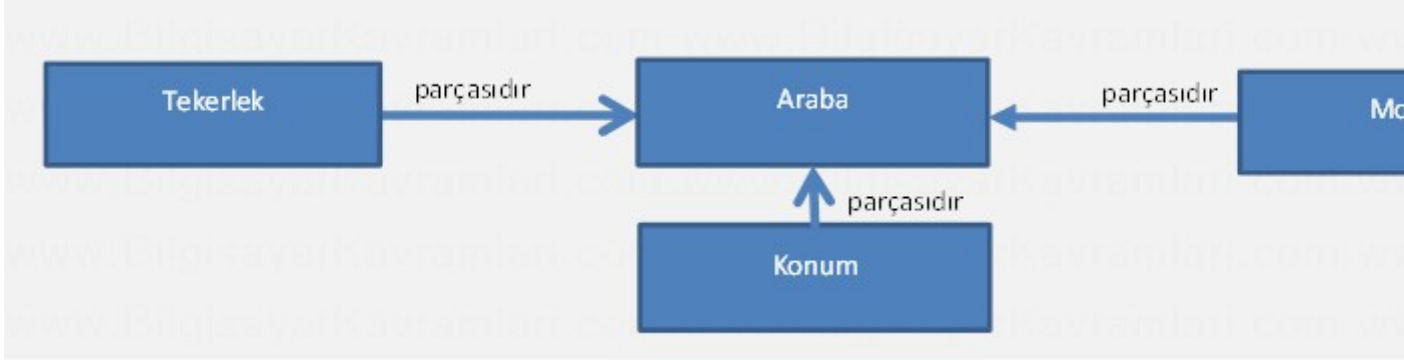
“parçası” ilişkisi

Bu ilişki türünde, bir nesne, diğer bir nesneyi bir özelliği olarak barındırır. Aslında bir nesnenin alt nesnesi olması durumudur. Parçası ilişkisinde bu ilişki türü, sınıflar arasında tanımlıdır. Örneğin bir araba sınıfının, tekerlek sınıfını bulundurması düşünülebilir. Yani her arabada tekerlek vardır.

```
class Araba {  
    özellikler:  
        Tekerlek tekerlek  
        Motor motor  
}
```

```
Nokta konum
metotlar:
  set(Nokta konum)
}
```

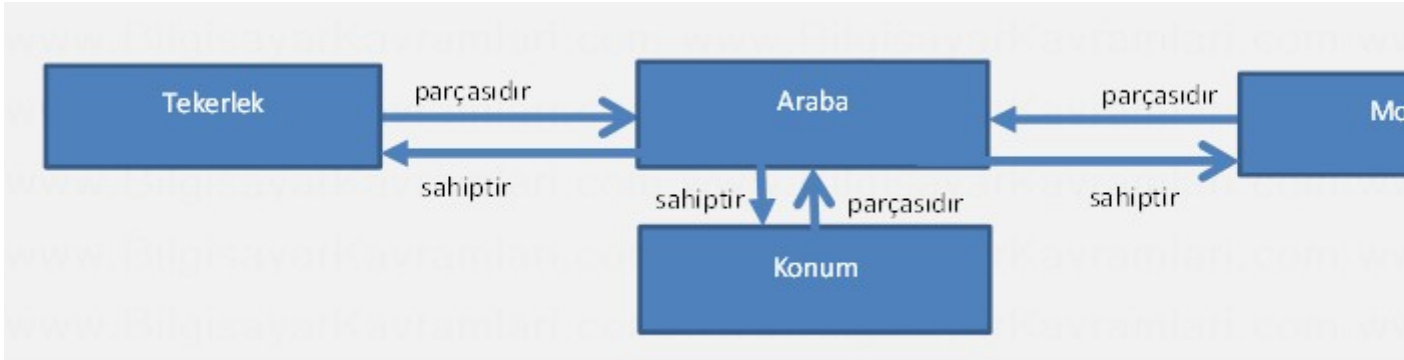
Yukarıdaki örnekte, Araba sınıfının, konum, tekerlek ve motor özellikleri bulunmaktadır. Buna göre bu sınıftan üretilen her araba bu özellikleri taşır ve farklı değerlere sahiptir. Örneğin her arabanın farklı motor özellikleri ve farklı tekerlek bilgileri olabilir. Benzer şekilde her arabanın konumu da farklı olabilir. Örneğimizde, bu dış sınıfların her biri, Araba sınıfının bir parçasıdır.



Yukarıdaki şekilde görüldüğü gibi, sınıflar arasında, parçasıdır şeklinde bir ilişki kurulmuştur.

“Sahiplik” ilişkisi

Bu ilişki türü, bir önceki “parçası” ilişkisinin tam tersidir.



Bu ilişki türüne göre, Araba, tekerleğe, konuma ve motora **sahiptir**.

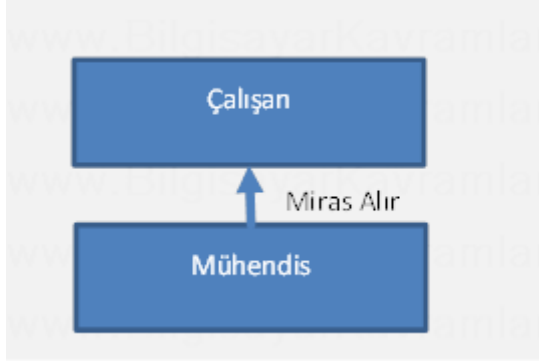
Kalıtım İlişkisi

Bu ilişki türü, “bir çeşit” ve “bir” ilişki türlerinin birleşimi olarak düşünülebilir.

Literatürde miras ilişkisi şeklinde de kullanılan bu ilişkiye göre bir sınıftaki özelliklerin diğer sınıfa doğrudan geçmesi mümkündür. Örneğin bir noktanın taşıdığı özelliklerin hepsi, çember tarafından taşınmaktadır. Öyleyse bu özelliklerin yeniden çemberde tanımlanmasına gerek duyulmaz. Çember, nokta sınıfından kalıtım yoluyla özellikleri alırken, kendisine özgü, farklı olan özellik ve metotları tanımlar.

Yukarıdaki şekilde görüldüğü üzere, iki nesne arasındaki kalıtım ilişkisi, miras almak şeklinde ifade edilebilir. Miras ilişkisi için genelde ata / torun (ancestor / decestor) veya ebeveyn / çocuk (parent / offspring) ifadeleri kullanılabilir. İlişkinin, ata olan kısmını gösteren bir üçgen konulması ve nesnelerin alt / üst şeklinde yerleştirilmesi, genelde miras ilişkisini ifade eder. Bu kullanım ayrıca UML standardında da geçmektedir.

Örnek olarak çalışan ve bu çalışandan türetilen bir mühendis sınıfını düşünelim.



Yukarıdaki bu şeklin C++ dilindeki kodlaması aşağıda verilmiştir.

```
class Çalışan {
public:
    Çalışan(string isim, float ödemeOrani);
    string getIsim() const;
    float getOdemeOrani() const;
    float odeme(float calismaSaati) const;
protected:
    string Isim;
    float OdemeOrani;
};

Çalışan::Çalışan(string isim, float ödemeOrani)
{
    Isim = isim;
    OdemeOrani = ödemeOrani;
}

string Çalışan::getIsim() const
{
    return Isim;
}

float Çalışan::getOdemeOrani() const
{
    return OdemeOrani;
}

float Çalışan::odeme(float calismaSaati) const
{
    return calismaSaati * OdemeOrani;
}
```


Yukarıdaki kodda, bir çalışan sınıfının özellikleri ve metotları verilmiştir. Bu kodu kullanarak aynı özellikleri taşıyan mühendis sınıfı da aşağıdaki şekilde verilsin.

```
class Muhendis {
public:
    Muhendis(string isim,
              float odemeOrani,
              bool maasliMi);

    string getIsim() const;
    float getOdemeOrani() const;
    bool getMaasli() const;

    float odeme(float calismaSaati) const;

protected:
    string Isim;
    float OdemeOrani;
    bool maasli;
};
Muhendis::Muhendis(string isim,
                    float odemeOrani,
                    bool maasliMi)
{
    Isim = isim;
    OdemeOrani = odemeOrani;
    maasli = maasliMi;
}

string Muhendis::getIsim() const
{
    return Isim;
}

float Muhendis::getOdemeOrani() const
{
    return OdemeOrani;
}

bool Muhendis::getMaasli() const
{
    return maasli;
}

float Muhendis::odeme(float calismaSaati) const
{
    if (maasli)
        return OdemeOrani;
    /* else */
    return calismaSaati * OdemeOrani;
}
```

Yukarıdaki bu iki sınıfta, görüldüğü üzere, ortak olan özellikler bulunmaktadır. Bu ortaklıktan faydalanarak kurulacak miras ilişkisi için aşağıdaki şekilde C++ kodu yazılmalıdır.

```
class Muhendis: public Çalışan {
```

Yukarıdaki bu yazımda, iki sınıf ismi arasında kullanılan : (iki nokta üst üste) işareti, sağdaki sınıftan, soldaki sınıfı miras yoluyla bilgi aldığını gösterir.

Burada Çalışan sınıfının başında, ayrıca erişim belirleyici (Access modifier) çeşitlerinden “public” kelimesi kullanılmıştır. Burada 3 ihtimal bulunur:

- public kalıtım
- protected kalıtım
- private kalıtım

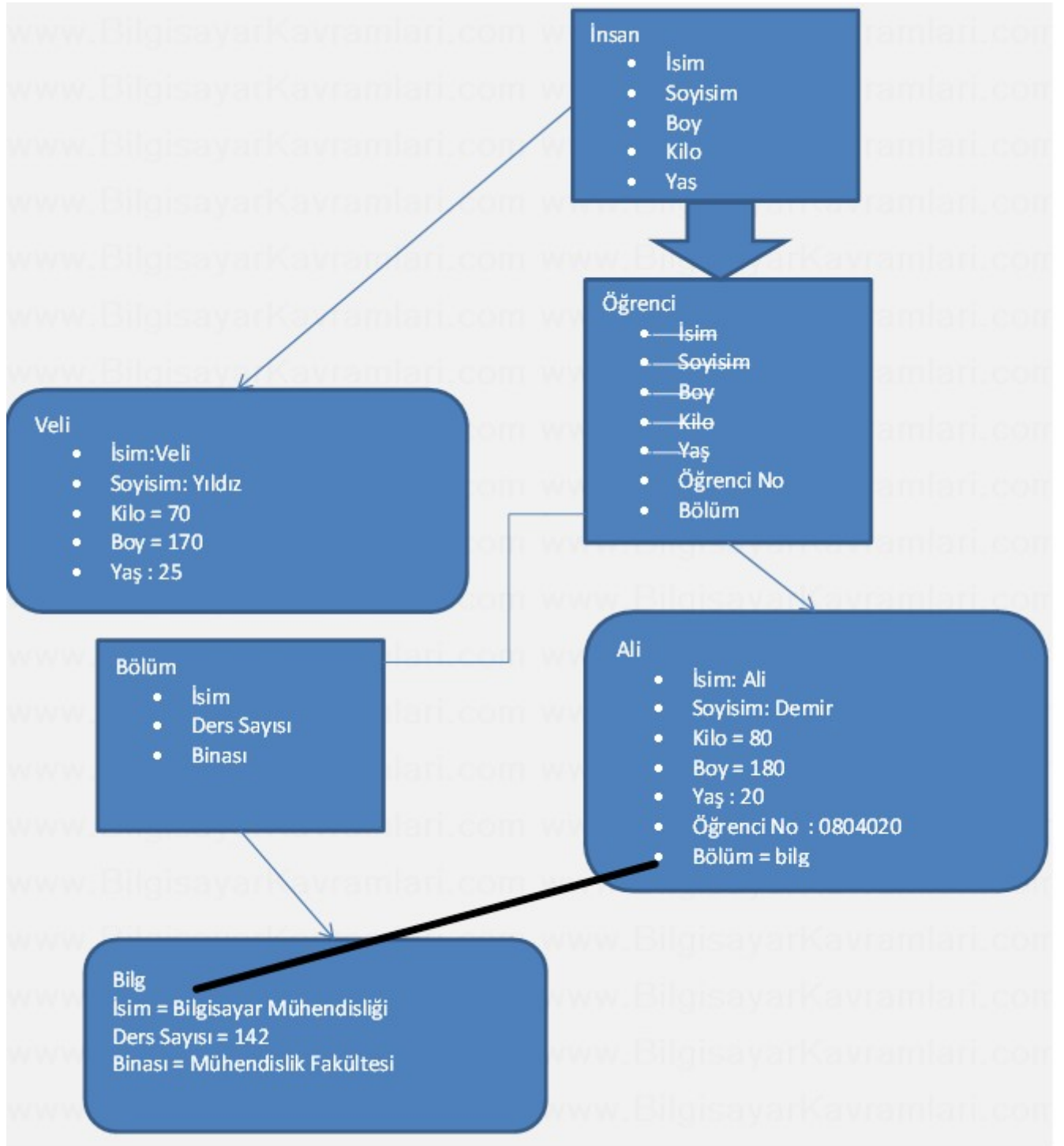
Kalıtım modelinin public olması, yani yukarıdaki örnek için Çalışan sınıfının başında public yazılması durumunda, Mühendis sınıfı, Çalışan sınıfının bütün özelliklerine erişir. Yani Çalışan sınıfında bulunan özellikler, public veya private olması farketmeksizin, Mühendis sınıfının, private özellikleri olarak miras alınır.

Protected kalıtım modelinde ise, Çalışan sınıfında bulunan private hariç bütün özelliklerini, kendisinin protected özelliği gibi miras alır.

SORU 13: Örnek C++ Sınıf İlişkileri

Bu yazının amacı, C++ dili üzerinden, bazı sınıf ve nesne ilişkilerini anlatmaktır.

Örnek olarak, aşağıda, şekli verilen diyagramı ele alalım:



Yukarıdaki bu şekilde, köşeli kutular içerisinde yazılan bilgiler sınıf (class) bilgileridir. Yuvarlak köşeli kutular içerisinde ise, bu sınıflardan türetilmiş nesneler (objects) görülmektedir.

Son olarak kalın ok, ISA ilişkisini, yani miras (kalıtım, inheritance) gösterirken, kalın çizgi, HASA ilişkisini ifade etmektedir.

Bu ilişkilerin modellendiği kod aşağıda anlatılmıştır.

```

1  #include <iostream>
2  #include <string>
3  #include <conio.h>
4  using namespace std;
5
6  class insan{
7  public:
8      int boy;
9      int kilo;
10     int yas;
11     char * isim;
12     char * soyisim;
13 };
14
15 class bolum {
16 public :
17     char isim[20];
18     int dersSayisi;
19     char bina [50];
20 };
21
22 class ogrenci : public insan {
23 public:
24     bolum* Bolum;
25     char ono[20];
26 };
27
28

```

Yukarıdaki kodun 1-5. Satırlarında, gerekli olan include işlemleri yapılmış ve namespace tanımlanmıştır.

Sonraki 6-27 satırları arasında sınıf tanımlamaları yapılmıştır. Bu sınıf tanımlamaları, diyagramda verildiği üzere her sınıfın ve her sınıftaki özelliklerin kodlanması şeklindedir.

Dikkat edilecek bir husus, 24. Satırda, öğrenci sınıfı için tanımlanan özelliklerden birisi olan Bölüm değişkeninin, bolum tipinde olmasıdır. Bunun anlamı, öğrenci sınıfının, bölüm tipinde bir sınıfı bulunmaktadır (HASA).

Ayrıca kodun 22. Satırında bulunan “class ogrenci : public insan” tanımından anlaşılaacağı üzere, öğrenci sınıfı, insan sınıfından miras almıştır. Dolayısıyla, insan sınıfında bulunan bütün özellikler, öğrenci sınıfına da geçebilmektedir. Ayrıca bu satırda bulunan ve insan sınıfı, miras alınmadan önce yazılan “public” kelimesi, bu miras ilişkisi üzerinden, insan sınıfında bulunan özelliklerin, öğrenci sınıfına açık olduğudur.

```

31 int main(){
32     ogrenci * ali = new ogrenci();
33     insan * veli = new insan();
34     ali->boy = 180;
35     ali->kilo = 80;
36     ali->yas = 20;
37
38
39     strcpy(ali->ono , "0804020");
40     veli->boy = 170;
41     veli->kilo = 70;
42     veli->yas = 25;
43     bolum * bilg = new bolum();
44     strcpy(bilg->isim , "Bilgisayar Müh.");
45     bilg->dersSayisi = 142;
46     strcpy(bilg->bina , "Mühendislik Binası");
47     ali->Bolum = bilg;
48     cout << "alinin bolumunun binasi : " << ((bolum*)ali->Bolum)->bina << endl ;
49
50
51     getch();
52 }

```

Yukarıda, kodumuzdan nesnelerin üretildiği, main fonksiyonu alıntılanmıştır. Bu kodun, 32 , 33 ve 43. Satırlarında, daha önceden tanımlanmış olan sınıflardan birer nesne üretilmiştir. Burada dikkat edilecek bir husus, üretilen nesnelerin ” new ” operatörü ile tanıtılmıştır. Bu durumda tanıtımı bir değişkene atayabilmemiz için, [nesne atfına \(object referrer\)](#) ihtiyaç duyulur. Bu nesne atıfları tanımlanırken, * operatörü ile pointer olarak tanımlanmasının sebebidir.

SORU 14: DOM (DNM)

DOM yani İngilizcedeki Document Object Modelling veya Türkçe karşılığı ile Doküman Nesne Modellemesi basitçe bir işaretleme dili (genellikle [HTML](#) veya [XML](#) gibi bir dil) için etiketlere (tags) erişmeyi sağlayan bir erişim yöntemidir.

Günümüzde SAX ve DOM en yaygın olarak kullanılan erişim yöntemleridir. SAX daha çok tek seferde işlenen ve ardışık olarak etiketlere erişileceği durumlarda kullanılır. DOM ise bir dokümanı ağaç yapısına benzer şekilde ele alır ve bir ağacın çocuklarına (children) erişir gibi dokümandaki etiketlere erişim sağlar.

Öncelikle DSO ile bir xml dosyasında yapabileceğimiz işlemleri inceleyelim:

DSO: Data Shared Object

xml dokümanları aslında birer DSO'dur. Yani ufak birer [databasedirler](#). [HTML](#) dokümanlarının ise bu DSO'ları kullanması mümkündür. Bir html dokümanının içine:

```
<XML ID="dsoBook" SRC="http://www.shedai.net/documents/Book.xml">
```

```
</XML>
```

şeklinde bir satır eklerseniz, bu [xml](#) dokümanına [html](#) komutlarıyla erişiminiz mümkün olur.

[Database konularından](#) bilindigi uzere: Her tablonun alanlari vardır. Ve her kayıt bu tablodaki alanlari doldurur. Mesela: calisan tablosu:

ISIM	SOYISIM	BOLUM
Ali	Gelir	14
Selami	Gider	-
Veli	Bakar	10

Seklindeki bir tabloda, ISIM, SOYISIM, BOLUM gibi bilgiler birer alandır. [ali,gelir,14], [selami,gider,-] gibi bilgilerde kayittir.

[XML dokumaninin](#) database gibi kullanilmasina gelince, [her xml dokumani](#) bir tablo, her yeni komut bir kayıt, ve her komutun alt komutu da birer alandır. Yani yukaridaki tabloyu xml'e uyarlarsak:

<calisan>

<eleman>

<isim>ali</isim>

<soyisim>gelir</soyisim>

<bolum>14</bolum>

</eleman>

<eleman>

<isim>Selami</isim>

<soyisim>Gider</soyisim>

</eleman>

<eleman>

<isim>Veli</isim>

<soyisim>Bakar</soyisim>

<bolum>10</bolum>

</eleman>

</calisan>

sekinde bir xml dokumani elde ederiz.

Peki bir html komutunu, bir xml komutuna nasıl bağlarız?

İki türlü bağlamamız mümkün.

- Tablo bazlı bağlama
- Kayıt bazlı bağlama

Önce tablo bağlamaya bakalım. Şimdi bizim mesur inventory.xml dosyası için bir html oluşturalım.

```
<!-- File Name: Inventory Table.htm -->

<HTML>

<HEAD>

<TITLE>Book Inventory</TITLE>

</HEAD>

<BODY>

<XML ID="dsoInventory" SRC="Inventory.xml"></XML>

<H2>Book Inventory</H2>

<TABLE          DATASRC="#dsoInventory"          BORDER="1"
CELLPADDING="5">

<THEAD>

<TH>Title</TH>

<TH>Author</TH>

<TH>Binding</TH>

<TH>Pages</TH>

<TH>Price</TH>
```

```

</THEAD>

<TR ALIGN="center">

<TD><SPAN DATAFLD="TITLE"
STYLE="font-style:italic"></SPAN></TD>

<TD><SPAN DATAFLD="AUTHOR"></SPAN></TD>

<TD><SPAN DATAFLD="BINDING"></SPAN></TD>

<TD><SPAN DATAFLD="PAGES"></SPAN></TD>

<TD><SPAN DATAFLD="PRICE"></SPAN></TD>

</TR>

</TABLE>

</BODY>

</HTML>

```

Bu kodun calisan hali asagidadir:

Book Inventory

Title	Author	Binding	Pages	Price
<i>The Adventures of Huckleberry Finn</i>	Mark Twain	mass market paperback	298	\$5.49
<i>Leaves of Grass</i>	Walt Whitman	hardcover	462	\$7.75
<i>The Legend of Sleepy Hollow</i>	Washington Irving	mass market paperback	98	\$2.95
<i>The Marble Faun</i>	Nathaniel Hawthorne	trade paperback	473	\$10.95
<i>Moby-Dick</i>	Herman Melville	hardcover	724	\$9.95
<i>The Portrait of a Lady</i>	Henry James	mass market paperback	256	\$4.95
<i>The Scarlet Letter</i>	Nathaniel Hawthorne	trade paperback	253	\$4.25
<i>The Turn of the Screw</i>	Henry James	trade paperback	184	\$3.35

Buradaki en onemli satir:

```
<TABLE DATASRC="#dsInventory" ...
```


satiridir. Bu satirla, table komutumuzun bilgileri #dsoInventory'den okumasi gerektigini soyluyoruz. (ki #dsoInventory daha oncede belirttigimiz gibi <XML ID="dsoInventory" SRC="Inventory.xml"></XML> satiriyla tanimlanmis bir xml kaynagidir.)

Peki [html](#) bu alanlardan hangisini table'in hangi bolmesine yazacagini nasil anliyor?

Iste bunu yapanda <TD></TD> seklindeki satirlar.

Biraz daha HTML baglantisi

Simdi bu xml, html baglantisini bir adim daha ileri goturerek, asagidaki goruntuyu elde etmeye calisalim:

Book Inventory

|< First Page | < Previous Page | Next Page > | Last Page >|

Title	Author	Binding	Pages	Price
<i>The Adventures of Huckleberry Finn</i>	Mark Twain	mass market paperback	298	\$5.49
<i>The Adventures of Tom Sawyer</i>	Mark Twain	mass market paperback	205	\$4.75
<i>The Ambassadors</i>	Henry James	mass market paperback	305	\$5.55
<i>The Awakening</i>	Kate Chopin	mass market paperback	195	\$4.55
<i>Billy Budd</i>	Henry Melville	mass market paperback	195	\$4.49

Yukaridaki goruntuyu elde etmek icin, ONCLICK ozellikleri ve DATAPAGESIZE="5" gibi yeni komutlari iceren asagidaki kodu yazmamiz gerekiyor.

```
<HTML>

<BODY>

<XML ID="dsoInventory" SRC="Inventory Big.xml"></XML>

<H2>Book Inventory</H2>

<BUTTON ONCLICK="InventoryTable.firstPage()">

|&lt; First Page

</BUTTON>
```



```
STYLE="font-style:italic"></SPAN></TD>

<TD><SPAN DATAFLD="AUTHOR"></SPAN></TD>

<TD><SPAN DATAFLD="BINDING"></SPAN></TD>

<TD><SPAN DATAFLD="PAGES"></SPAN></TD>

<TD><SPAN DATAFLD="PRICE"></SPAN></TD>

</TR>

</TABLE></BODY></HTML>
```

JavaScript kullanarak XML üzerinde arama yapmak:

```
<XML ID="dsoInventory" SRC="Inventory Big.xml"></XML>
```

```
<H2>Find a Book</H2>
```

```
Title text: <INPUT TYPE="TEXT" ID="SearchText">&nbsp;
```

```
<BUTTON ONCLICK='FindBooks()'>Search</BUTTON>
```

```
<HR>
```

```
Results:<P>
```

```
<DIV ID=ResultDiv></DIV>
```

```
<SCRIPT LANGUAGE="JavaScript">
```

```
function FindBooks ()
```

```
{
```

```
SearchString = SearchText.value.toUpperCase();
```

```
if (SearchString == "")
```

```
{
```

```
ResultDiv.innerHTML = "&lt;ltYou must enter text into "
```

```

+ “‘Title text’ box.&gt;”;

return;

}

dsoInventory.recordset.moveFirst();

ResultHTML = “”;

while (!dsoInventory.recordset.EOF)

{

TitleString = dsoInventory.recordset(“TITLE”).value;

if (TitleString.toUpperCase().indexOf(SearchString)

>=0)

ResultHTML += “<I>”

+ dsoInventory.recordset(“TITLE”)

+ “</I>, ”

+ “<B>”

+ dsoInventory.recordset(“AUTHOR”)

+ “</B>, ”

+ dsoInventory.recordset(“BINDING”)

+ “, ”

+ dsoInventory.recordset(“PAGES”)

+ ” pages, ”

+ dsoInventory.recordset(“PRICE”)

+ “<P>”;

dsoInventory.recordset.moveNext();

}

```

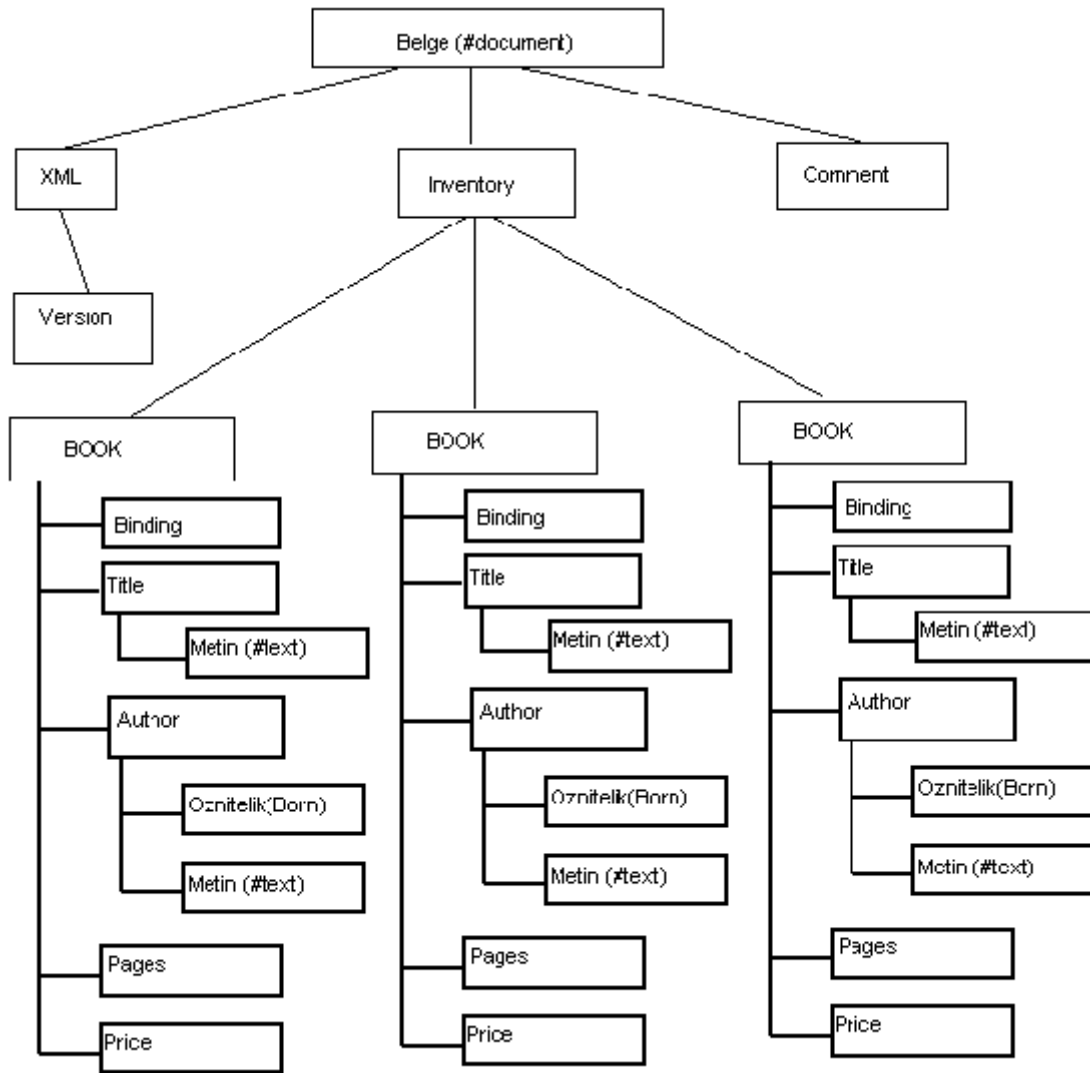
```
if (ResultHTML == "")  
  
ResultDiv.innerHTML = "&ltno books found&gt;";  
  
else  
  
ResultDiv.innerHTML = ResultHTML;  
  
}  
  
</SCRIPT>
```

[XML dokumanlarini](#) nasıl JavaScript ve dom yardimiyla kullaniriz bir bakalim.

Bir html sayfasina DSO eklemeyi yukaridaki orneklerde gosterdik. Simdi asagidaki ornek html kodunu inceleyelim.

```
<SCRIPT LANGUAGE="JavaScript" FOR="window" EVENT="ONLOAD">  
  
Document = dsoBook.XMLDocument;  
  
title.innerText=  
  
Document.documentElement.childNodes(0).text;  
  
author.innerText=  
  
Document.documentElement.childNodes(1).text;  
  
binding.innerText=  
  
Document.documentElement.childNodes(2).text;  
  
pages.innerText=  
  
Document.documentElement.childNodes(3).text;  
  
price.innerText=  
  
Document.documentElement.childNodes(4).text;  
  
</SCRIPT>
```

Yukaridaki kod, JavaScript ile DOM parser'ini kullanarak, XML dokumanina erisiyor. DOM, xml dokumanlarini parcalarken agac yapısına benzer bir yapı kullanır. Ve her yeni xml komutu ([database mantigina gore](#) kayidi) birer childNode olur.



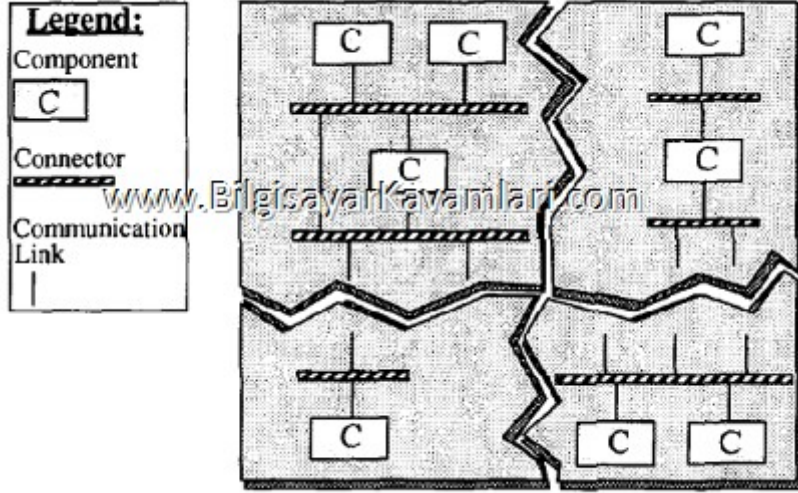
Örneğin yukarıdaki şekilde her bir book ögesi documentElement'in birer childNode'udur. Ve her book ögesinde beşer adet childNode'u vardır.

SORU 15: C2 Üslûbu (C2 Style)

Bilgisayar bilimlerinde özellikle yazılım mühendisliği (Software engineering) konusunda kullanılan ve bileşen (component) ve mesaj (message) temelli yazılım geliştirmeyi amaçlayan bir üsluptur.

C2 tasarım kültüründe yazılım bileşenler şeklinde ele alınır ve yazılımı oluşturan bu bileşenler üzerinden bir ağ (network) çizilir. Bu ağda bileşenler arası haberleşme gösterilir. Bu sayede tasarlanan bir bileşenin daha verimli ve tekrar tekrar kullanılması mümkün olmuş olur.

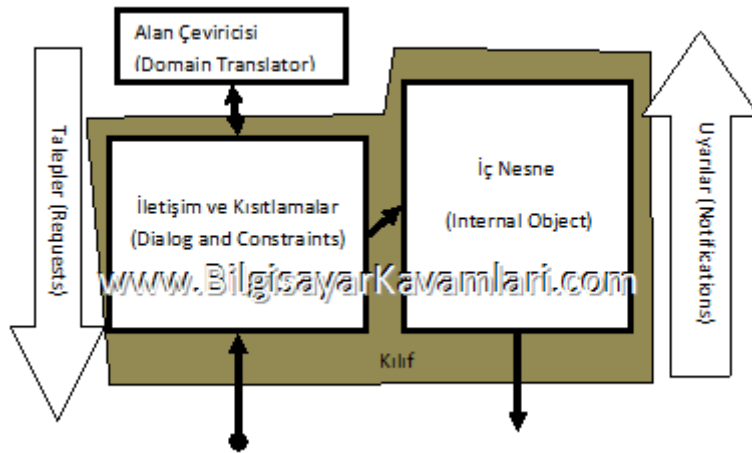
1996 yılında bu yaklaşımı ilk kez ortaya koyan Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, ve Richard N. Taylor, "Using Object-Oriented Typing to Support Architectural Design in the C2 Style (C2 stili ile [nesne yönelimli](#) ve destek mimarili tasarım)" isimli makalelerinde bu nesneler arası ilişkiyi aşağıdaki şekilde temsil etmişlerdir:



Yukarıdaki şekilde görüleceği üzere her bileşen C harfi ile temsil edilmiş ve bağlar birer çizgiyle gösterilmiştir. Ayrıca kalın çizgiler ana bağlantı noktalarını göstermektedir.

Bu yazı şadi evren şeker tarafından yazılmış ve bilgisayarkavamlari.com sitesinde yayınlanmıştır. Bu içeriğin kopyalanması veya farklı bir sitede yayınlanması hırsızlıktır ve telif hakları yasası gereği suçtur.

Yine aynı makalede bulunan bir C2 bileşeninin iç yapısını aşağıdaki şekilde Türkçe terminoloji ile gösterebiliriz:



Yukarıdaki şekilde anlatılmak istenen bir nesnenin bir kılıf içerisinde ele alındığıdır. Yani C2 tasarımında bileşen olarak görülen herşey aslında nesne yönelimli programlamada bir [nesne\(object\)](#) ile karşılaşılır ancak bu nesneye bir kılıf giydirilmiş olarak kabul edilebilir. Nesne ile iletişim ayrıca bir arayüz ile yapılırken nesnenin dışarıya doğrudan erişimi bulunmaktadır. Bu durum kılıftan çıkan ve talepler (request) yönünde olan bir ok ile temsil edilmiştir.

Doğrudan kılıf (wrapper) dışına erişim yapabilen nesnenin yanında bütün uyarılar (notifications) iletişim ve kısıtlama nesnesi üzerinden iç nesneye erişebilmektedir. Aslında

tasarım olarak nesne yönelimli programlamanın [kapsülleme \(encapsulation\)](#) mantığına oldukça yakın olan bu durumda, ayrıca nesnelerin hepsinin kendi adreslerinde çalıştıklarını düşünebiliriz. Kendi adres alanlarında (own address space) çalışan nesneler değişken paylaşımı (Shared variable) korunmuş olacağı için, çok kullanıcı (multiuser) , çok işlemli (multiprocessing/[threading](#)) ve eşzamanlı (concurrent) modellemeleri desteklemektedir.

SORU 16: Nesne serileme ve dizme (Object Serialization , Marshalling)

Bilgisayar bilimlerinde kullanılan bir yaklaşım olan nesne yönelimli programlama (object oriented programming) sayesinde gelişmiş olan bir kavramdır. Basitçe bir [nesnenin \(object\)](#) hafızada (RAM) olan bilgilerinin saklanmak veya ağ üzerinden yollanmak gibi amaçlarla bir [dizgiye \(string\)](#) dönüştürülmesi işlemidir.

Bu dizginin yapısı olarak çoğunlukla XML dili kullanılır. bu sayede verinin içeriğini belirleyen değişken değerleri kolayca oluşturulup parçalanabilir (parse).

Nesne sıralama (object serialisation) işlemi farklı dillerde farklı şekillerde yapılmakla beraber genelde nesnede bulunan private (husus) değişken değerleri oluşturulan sonucun içinde yer almaz.

JAVA dilinde nesne serileme

Java dilinde bir nesnenin serilenmesini aşağıdaki örnek kod üzerinden inceleyelim:

```
1  import java.io.*;
2
3  /**
4   * serilestirilebilir bir sınıf
5   */
6  class serileme implements Serializable {
7      static private final long serialVersionUID = 42L;
8
9      private String ozellik1;
10     private int ozellik2;
11
12     public serileme(String ozellik1, int ozellik2) {
13         this.ozellik1 = ozellik1;
14         this.ozellik2 = ozellik2;
15     }
16
17     @Override
18     public String toString() {
19         return ozellik1 + ", " + ozellik2;
20     }
21 }
```

bilgisayar kavramları

Yukarıdaki kodda “serileme” isminde bir [sınıf \(class\)](#) oluşturulmuş ve bu sınıfın Serializable arayüzünü (interface) implement etmesi sağlanmıştır. Buradaki amaç bu arayüzde tanımlı olan ve her serileştirilecek nesnede üzerinebindirilmesi (override) mecburi olan toString() fonksiyonunu bu sınıfın yazarına zorlamaktır.

Kodda görüleceği üzere 17. Satırda bulunan toString fonksiyonu bir gerekliliktir. Bu fonksiyon yazılmazsa serileştirme işlemi olamaz. Zaten bildiğimiz üzere serileştirme işlemi de aslında bir [dizgiye \(string\)](#) dönüştürme işlemidir. Dolayısıyla toString fonksiyonu burada devreye girer.

Bu yazı şadi evren şeker tarafından yazılmış ve bilgisayar kavramları.com sitesinde yayınlanmıştır. Bu içeriğin kopyalanması veya farklı bir sitede yayınlanması hırsızlıktır ve telif hakları yasası gereği suçtur.

Yukarıdaki örnek sınıfımızda basitçe “ozellik1” ve “ozellik2” isminde iki özellik (attribute) bulunmaktadır. Sınıfımızın [yapıcısında \(constructor\)](#) bu değerlere atama yapılmakta ve toString fonksiyonunda ise bu değerler bir virgül (,) sembolü ile ayrılarak [bir dizgi \(string\)](#) oluşturulmaktadır.

Artık yukarıdaki sınıfı kullanan aşağıdaki şekilde bir sınıf yazabiliriz.

```
23 public class Main {
24     /**
25      * nesne kaydi
26      */
27     private static void nesneKaydet(Serializable object, String filename) throws
28         ObjectOutputStream objstream = new ObjectOutputStream(new FileOutputStream(
29         objstream.writeObject(object);
30         objstream.close();
31     }
32
33     /**
34      * nesne yükleme
35      */
36     private static Object nesneYukle(String filename) throws ClassNotFoundException
37         ObjectInputStream objstream = new ObjectInputStream(new FileInputStream(
38         Object object = objstream.readObject();
39         objstream.close();
40         return object;
41     }
42
43     public static void main(String[] args) {
44         serileme ilkHali = new serileme("sadi evren seker", 123);
45         System.out.println(ilkHali);
46         try {
47             nesneKaydet(ilkHali, "nesne.ser");
48             serileme geriYuklenmis = (serileme) nesneYukle("nesne.ser");
49             System.out.println(geriYuklenmis);
50         } catch (Exception e) {
51             e.printStackTrace();
52         }
53     }
54 }
55
```

Yukarıdaki yeni kodda Main isminde bir sınıf tanımlanmakta bu sınıfın içerisinde artık serileştirilebilir (serializable) olan “serileme” isimli sınıfımızdan bir nesne dosyaya kaydedilmekte ve tekrar okunarak ekrana basılmaktadır.

Kodda kabaca 43. Satırdan itibaren başlayan ve ilk çalışan main fonksiyonu içerisinde ilkHali isimli bir [dizgiye \(string\)](#) “serileme” ismindeki nesnenin serileştirilmiş hali atanmakta ve ekrana basılmaktadır.

46. satırdan itibaren ise yine Main [sınıfında](#) bulunan nesneKaydet ve nesneYukle isimli fonksiyonlara “serileme” isimli sınıftan ürettiğimiz bu nesne verilmekte ve nesne dosyaya kaydedilip tekrar dosyadan okunarak nesne haline dönüştürülmektedir.

[Yukarıdaki kodun tam ve çalışır haline buradan erişebilirsiniz.](#)

Yukarıdaki kodun ekran çıktısı aşağıdaki şekildedir:

```
C:\Users\shedai\Desktop\bilgisayarkavramlari>javac Main.java
```

```
C:\Users\shedai\Desktop\bilgisayarkavramlari>java Main
deneme yazisi, 123
deneme yazisi, 123
```

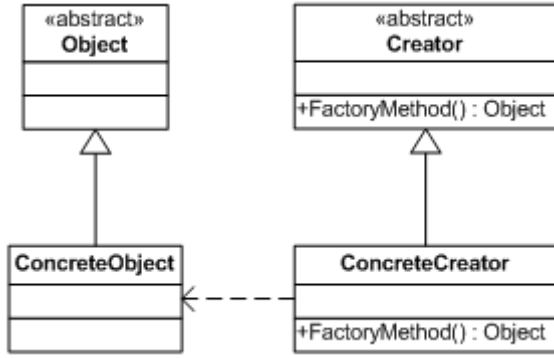
SORU 17: Fabrika Metotları (Factory Methods)

Fabrika metotları tanım itibariyle sabit [metotlardır \(static methods\)](#). Yani bir fabrika metodu (factory method) çağrılmak için bir nesneye ihtiyaç duymaz, doğrudan [sınıfa \(Class\)](#) erişilerek çağrılabilir. Fabrika metotları bunun yanında değer olarak bir nesne döndüren metotlardır. Yani bir fabrika metodu çağrıldığında dönüş değeri olarak bir sınıftan türetilmiş bir nesne beklenir.

Bu anlamda zaten isminden de anlaşılacağı üzere fabrika metotları, nesne üreten metotlardır. Bir sınıfın tanımı içerisinde yer alırlar ve bu sınıfa doğrudan erişilerek nesne üretirler.

Örneğin [kartezyen uzaydaki \(cartesian space\)](#) bir koordinatın, [kutupsal uzaydaki bir koordinata \(polar coordinate\)](#) çevrilmesi işlemini ele alalım. Kartezyen uzay değerlerini tutan bir sınıfımız olacak ve çevrim işlemini yapan bir metodumuzu bulunacaktır. Bu metot yapısı itibariyle kutupsal koordinatta nesneler üretmek zorundadır. İşte bu üretim işlemi dolayısıyla çevrim yapan bu metoda fabrika metodu ismi verilir.

Fabrika metotları ayrıca bir [tasarım kalıbıdır \(design pattern\)](#). Yani nesne yönelimli programlama da sıkça kullanıldığı için hazır bir kalıp haline getirilmiştir ve örneğin UML [sınıf diyagramlarında \(class diagrams\)](#) kullanılan bir kalıp olarak hazırlanmıştır. Yazılım projelerinin tasarımı aşamasında bu kalıplardan faydalanılarak büyük ölçekteki projelerin geliştirilmesi hızlandırılmakta ve hata ihtimali azaltılmaktadır. Bir fabrika metodu aşağıdaki sınıf diyagramı kalıbı ile gösterilebilir:



Yukarıda görüldüğü üzere bir üreteç [soyut sınıftan \(abstract class\)](#) [miras ilişkisindeki \(inheritance\)](#) bir nesne tarafından bir nesne üretilmektedir.

SORU 18: Sabit Metotlar (Static Methods)

[Nesne yönelimli programlamada](#) kullanılan bir terimdir. Basitçe bir [nesnenin \(object\)](#) çalışabilmesi , yaşayabilmesi için bu nesnenin tanımlı olduğu bir [sınıfa \(class\)](#) ihtiyaç vardır. Ayrıca bu [sınıftan \(class\)](#) tanımlanan nesnenin programlama dili ve işletim sistemi marifetiyle hafızada bir alana ihtiyacı vardır.

Bilindiği üzere [nesnelerin](#) tanımlı oldukları [sınıflar \(class\)](#) içerisinde özellikleri (properties) ve metotları (methods) bulunur. Yani her nesnenin tanımında bir miktar değişken (variable) ve [fonksiyon \(function\)](#) bulunabilmektedir. Bu değişken (özellik, property) ve fonksiyonlara (metotlara, methods) erişmek için bu nesneye erişilmesi söz konusudur.

Ancak bazı durumlarda sabit metotlar (static methods) tanımlanabilir. Bu metotlar bir nesne ile var olmayan ancak sınıfın kendisinde tanımlı metotlardır. Yani bu metodun çalışması için o sınıftan bir nesne üretilmesi gerekmemektedir. Bu anlamda sabit metotlara, sınıf metotları (class methods) ismi de verilmektedir.

Örneğin çok meşhur sabit metotlardan birisi Thread sınıfında buluna sleep() metodudur. Bu metodu çalıştırmak için Thread.sleep(int) yazmak yeterlidir. Görüldüğü üzere metodun çalışması için önce [Thread sınıfından](#) bir nesne üretmeye sonra bu nesnenin sleep() fonksiyonunu çağırmaya gerek yoktur. Doğrudan sınıf ismi ile metoda erişmek mümkündür.

Bu yazı şadi evren şeker tarafından yazılmış ve bilgisayar kavramları.com sitesinde yayınlanmaktadır.

Diğer meşhur bir sabit metot main fonksiyonudur. Bu metodun çalışması için nesne üretilmesine ihtiyaç yoktur. main metodunu içeren bir sınıf doğrudan çalıştırılabilir.

Benzer şekilde biz de kendi sabit metotlarımızı yazabiliriz. Bunu aşağıdaki JAVA kodu üzerinden görmeye çalışalım. Önce basit bir sınıf ve bu sınıftan bir [nesne](#) üreterek kullanan bir ana sınıf (main class) yazmaya çalışalım:

```
public class deneme{  
  
public int kare(int x){
```

```

return x*x;

}

}

public class ana{

public static void main(String args[]){

deneme d = new deneme();

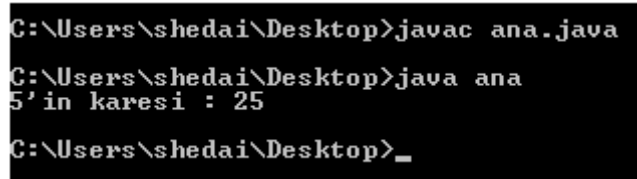
System.out.println("5'in karesi : " + d.kare(5));

}

}

```

Yukarıda iki sınıf verilmiştir. ana sınıfındaki main fonksiyonu içerisinde deneme sınıfından d isminde bir nesne oluşturulmuş ve bu nesnenin kare metoduna d.kare(5) şeklinde erişilmiştir. Yukarıdaki kodun çalışması sonucu ekran görüntüsü aşağıda verilmiştir:



```

C:\Users\shedai\Desktop>javac ana.java
C:\Users\shedai\Desktop>java ana
5'in karesi : 25
C:\Users\shedai\Desktop>_

```

Görüldüğü üzere 5'in karesi olarak 25 sonucu başarılı bir şekilde bulunmuştur. Ancak bu erişim sırasında bir nesne oluşturmaya ihtiyaç yoktur. Yani her kare alma işlemi sırasında hafızada (RAM) bir deneme sınıfından nesne oluşturmak ve bu nesneyi çağırmak gerekmez. Bunun yerine sabit metot (static method) kullanılabilirdi. Bu durumda kodumuz aşağıdaki şekilde olacaktır:

```

public class deneme{

public static int kare(int x){

return x*x;

}

}

public class ana{

public static void main(String args[]){

System.out.println("5'in karesi : " + deneme.kare(5));

```

```
}  
  
}
```

Yukarıdaki yeni kodun çalışması ve ekran çıktısı bir önceki kod ile bire bir aynıdır:

```
C:\Users\shedai\Desktop>javac ana.java  
C:\Users\shedai\Desktop>java ana  
5'in karesi : 25  
C:\Users\shedai\Desktop>_
```

Bu yeni kodda yapılan değişiklik kare metodunun static hale getirilmesi ve bu metodu çağıran ana sınıfındaki main metodunun doğrudan bu fonksiyona erişmesidir.

Sabit metotlar (static methods) nerelerde kullanılırlar?

Sabit metotlar yapı olarak [nesnelerin](#) oluşturulması ve çalışmasından bağımsız metotlardır. Bu yapıları itibariyle hafızanın verimli kullanılması ve dolayısıyla bir nesneye ihtiyaç duymadan bazı fonksiyonların çalıştırılması istendiğinde oldukça kullanışlıdırlar.

Yukarıdaki örnek incelenirse sabit metodun kullanıldığı ikinci örnekte bir nesne üretilmesi ve bu nesneye erişilmesi gerekmemiştir. Bu hafızadan verim sağlayacağı gibi çalışma süresini de belirli bir oranda kısaltmaktadır.

Performans kaygısının yanında sabit metotların sağladığı bir avantaj da tasarım aşamasındadır. Her metot yapı olarak bir nesneye bağlı olmayabilir. Örneğin insan [sınıfını](#) ve insan sınıfından türetilen nesneleri düşünelim. İnsan sınıfını yazarken bir insanın boyu, kilosu, yaşı gibi özelliklerinden ve yemek yemek, uyumak, yürümek veya konuşmak gibi metotlarından bahsedebiliriz. Buradaki örneklerin hepsi her insanda farklılık gösterecek özelliklerdir. Yani her insan farklı şekillerde uyuyabilir, yürüyebilir veya yemek yiyebilir. Diğer bir değişle her insanın yaptığı bu eylemler aslında tasarım itibariyle aynı mantığa dayanmakla birlikte her insanda farklı parametrik yapılara sahip ve her insanın taşıdığı boy, yaş, kilo gibi değerlerden etkilenmektedir.

Örneğin insan sınıfında herhangi bir özellikten etkilenmeyen çarpma fonksiyonu olduğunu düşünelim. Basitçe her insanın çarpma yapma özelliği olduğunu ve iki sayıyı çarpabileceğini düşünürsek bu fonksiyon herhangi bir özellikten etkilenmeyecektir. Yani kilosu farklı olan iki kişi ya da yaşı ya da boyu farklı olan iki kişinin sayıları çarptığında aynı değeri bulmasını bekleriz. Öyleyse bu fonksiyonu nesnelere değil doğrudan sınıfa koymamız isabetli olur çünkü bu sınıftan ne kadar nesne üretilirse üretilsin sonuçta çarpma işlemi farklılık göstermeyecektir.

Görüldüğü üzere sabit metotlar tasarım özelliği veya performans kaygıları gibi amaçlarla kullanılırlar.

SORU 19: Birleşim Noktaları (JoinPoints)

Bağlam yönelimli (aspect oriented) programlamada kullanılan terimlerden biridir. Anlam olarak programda bulunan bir kontrol noktasının bir veya daha fazla bağlamın etkisinde olduğu nokta demektir.

Bu anlamda bir kodda bulunan bütün satırlar birleşim noktası (joinpoint) olabilir ancak bu yaklaşım bağlam yönelimli programlama açısından doğru değildir. Daha doğrusu bütün olası birleşim noktalarından en mantıklı ve optimum noktaları almaktır. Ayrıca bilinmelidir ki aynı sonucu veren iki farklı birleşim noktası kümesi almak mümkündür.

Birleşim noktalarının çeşitleri

Bağlam yönelimli programlama yaklaşımı yapı olarak nesne yönelimli programlamanın üzerine kurulu olduğu için birleşim noktaları da nesne yönelimli programlama öğelerinden oluşacaktır. Örneğin nesne yönelimli programlamada kullanılan metotlar, yapıcılar (constructors), istisnalar (exceptions) veya özellikler adayı birer birleşim noktasıdır.

Örneğin metotların birer birleşim noktası olarak görülmesi aslında metotların bir programın çalışma akışının parçası olmasından kaynaklanmaktadır. Yani bir programı belirli sayıda metotun belirli bir sırayla çağrılması olarak düşünmek mümkündür. Bu çalışma sırasında bir metot başlar çalışır ve biter. Dolayısıyla metotların başlaması, çalışması veya bitmesi birer birleşim noktası (join point) olabileceği gibi metotun tamamı da bir birleşim noktası olabilir.

Benzer şekilde bir istisna (exception), üretilerek fırlatılan (throw) bir yapıdadır. Dolayısıyla bir istisnanın fırlatıldığı veya yakalandığı (catch) noktaların ikisi de sık kullanılan birleşim noktalarıdır.

SORU 20: Bağlam Örücüler (Aspect Weavers)

Bilgisayar bilimlerinde yazılım geliştirme sürecinde kullanılan yaklaşımlardan birisi de bağlam yönelimli programlamadır (Aspect oriented programming). Bu yaklaşım, yazılım geliştirme yaklaşımları sıralasında nesne yönelimli programlamadan sonra gelmektedir. Yani bir anlamda her bağlam yönelimli programlama yaklaşımı aslında bir nesne yönelimli programlama yaklaşımını kapsar.

Nesne yönelimli programlama yaklaşımından farklı olarak bağlam yönelimli programlama yaklaşımında ilave bağlamlar tutulur. Yani yazılımın modellenmesinde mevcut nesne yönelimli yaklaşım izlenir ancak bunun üzerine yazılımın kod yayılımı (code scattering) ve kestirme fonksiyonellikler (crosscut functions) gibi özelliklerden dolayı ilave olarak bağlamlar (aspects) tutulur.

Bu bağlamların yazılım ile bütünleştirilmesine bağlam örmek (aspect weaving) ismi verilir ve bağlam ile veriler modellemede farklı dururken çalışma sırasında bağlam örülmü (aspect woven) olarak dururlar.

Derleme zamanı örgü (Compile time weaving)

Yazılım modellemesinde kullanılan sınıflar (class) ile bağlamların (aspects) birleştirilemsi işleminin derleme sırasında olması durumudur. Örneğin java üzerinde çalışan aspectj bu gruba girmektedir.

Derleme zamanı örgü de kendi içinde iki gruba ayrılır. Genelde C# ve JAVA gibi nesne yönelimli programlamanın öncü dilleri ara bir kod üretirler (byte code) bu kod kullanılan çerçeve (framework) üzerinde çalışır ve platform bağımsızlığını (platform independence) sağlayan da bu ara kodun farklı ortamlarda çalışmasını sağlayacak ara katmanlar bulunmasıdır. Örneğin .net veya jvm (java virtual machine, java sanal makinesi) gibi.

Dolayısıyla derleme zamanı örgü seçenekleri bu ara kodun (byte code) üretilmesinden önce kaynak kod (source code) üzerinden çalışanlar ve byte kod üzerinden çalışanlar olarak ikiye ayrılabilirler.

Byte kod üzerinde işlem yapmak genelde daha yaygındır. Bunun birinci sebebi kolay olmasıdır. Örneğin java dili, üretilen byte koddan çok daha karmaşıktır. Diğer bir sebep ise bağlam örücülerin (aspect weavers) kaynak koda erişmesinin engellenmesidir. Yani örneğin java dilinde kodlanan bir uygulamanın kaynak kodlarına üçüncü parti bir bağlam örücünün erişmesi genelde istenmez. Son olarak byte kod işlemek kaynak kod işlemeye göre daha hızlıdır (basit olması da bir etken olduğundan) dolayısıyla performans olarak byte kod üzerinden bağlam örmek daha uygundur.

Derleme zamanı örgü sonucunda üretilen kod sabit bir şekilde bağlamları içermektedir. Yani herhangi bir bağlamanın değişme şansı yoktur veya değişmesi istenirse yeniden derleme süreci gerekmektedir.

Çalışma zamanı örgü (Run-time weaving)

Derleme zamanından farklı olarak derlenmiş ve çalıştırılmaya başlamış uygulamalar üzerinde örülen bağlamlardır. Bu tip bağlam örme işlemlerinde uygulamanın üretilmiş olan byte kodu ortamda çalıştırılırken uygulama geliştirme sürecinde üretilen bağlamlar uygulamaya örülür.

Çalışma zamanı örgünün en önemli avantajı, uygulamaya müteharrik (dinamik) bir şekilde bağlam eklenebilmesidir. Örneğin web sunucuları veya çoklu ortam sunucuları (ses, video gibi) gibi sürekli erişim altında olan suncular için bu özellik oldukça önemlidir.