

JAVA FİNAL

İSTİSNA DURUMU VE HATA YAKALAMA

Exception programın çalışma zamanı sırasında olağan dışı meydana gelen durumlardır. Programın çalışma zamanı sırasında olağan dışı bir durum gerçekleşirse compiler bir obje yaratır. Bu objeye exception object diyebiliriz.

Exception object runtime sırasında oluşturulan hata hakkında bilgileri tutar.

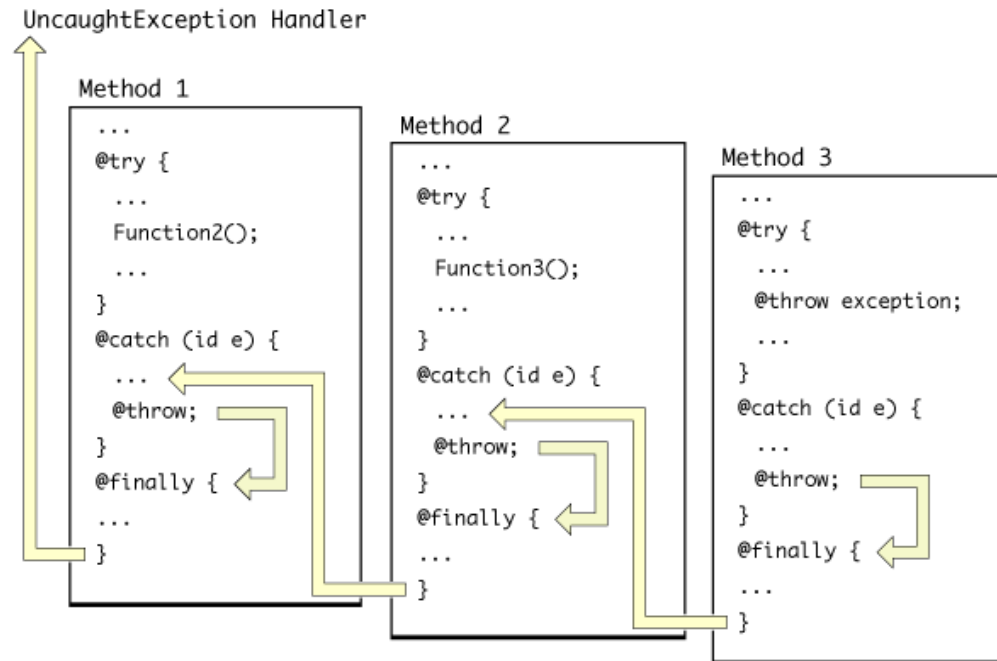
Bir method içerisinde oluşturulan exception metodu çağırarak exception a gönderilirse buna throwing an exception (hata fırlatma) denir.

Hata fırlatıldığı zaman kod bloğu içerisinde exception handle edilir. Buna exception handling denir.

Altındaki örneklerde 2,3,4. Satırlarda kod bloğu işletilir. Hata alınan durumda ise catch bloğu kadar olan kodlar execute edilmez. Exception objesi yaratılır ve catch bloğunun içerisine girilir. Burada tekrar kod bloğu işletilir. Exception objesi bu metodu çağırarak objeye gönderilir.

```
1 try {
2
3     Kod bloğu
4
5 } catch (Exception e) {
6     Kod Bloğu
7     throw ne Exception("");
8 }
```

Altındaki şekilde 3 method için Exception hiyerarşisi görülmektedir.



Yazdığımız kodlarda hatayı yakalamak için try bloğu yazmamız gerekir. Eğer try bloğunda hata yakalanıp Exception objesi fırlatılıyorsa, metod bildirim yapılr throws anahtar kelimesi ile Exception listesi belirtilmelidir.

Alttaki örnekte dosya oluşturulması sırasında bir hata alınırsa Exception fırlatılır. Method bildiriminde ise fırlatılacak exception objelerinin listesi belirtilmelidir.

```
public void fileCreate() throws Exception{  
    try{  
        File file= new File("my file");  
        file.createNewFile();  
    }  
    catch(Exception e){  
        throw new Exception("File not created");  
    }  
}
```

Exceptionlar 3' e ayrılır.

Checked Exception :

Bazı kod bloklarının hata fırlatabilme durumu ele alınarak handle etmemiz gerekir.

Örnek olarak java.io.FileReader objesini oluşturduğumuz zaman compiler bizde exception' ı handle etmemizi isteyecektir. Aksi durumda projesi compile edemeyiz.

```
public void fileRead() throws FileNotFoundException {  
    FileReader fileReader = new FileReader(new File("c:\\test.txt"));  
}
```

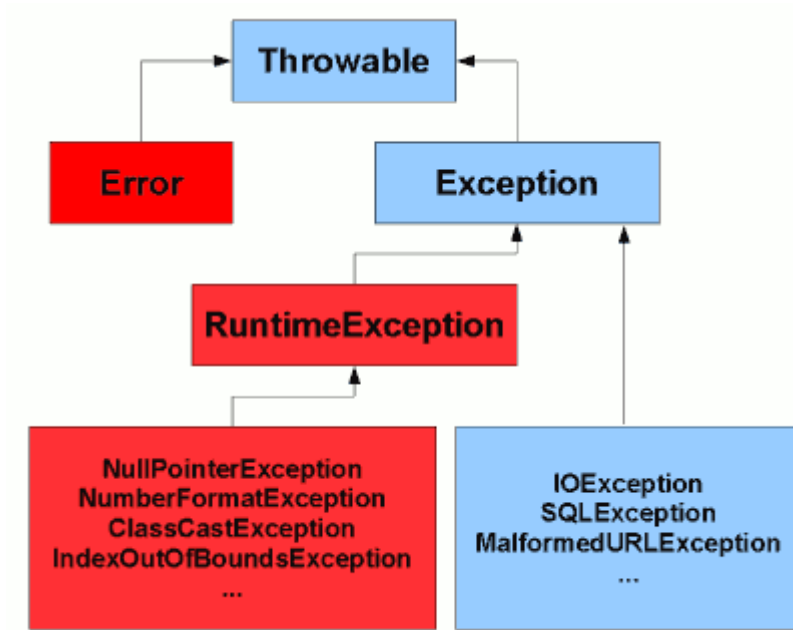
Runtime Exception (unchecked exception) :

Compiler tarafından, handle etme zorunluluğu olmayan hata tipleridir. Fakat Runtime sırasında hata aldığımız durumda handle etmemişsek programı recover etme şansımız yoktur.

Alttaki örnekte getDate metodu gelen Calender objesini date objesine çevirir. Fakat Calender objesi boş ise (null) NullPointerException hatası fırlatır. Fakat compiler bize bu durumu handle etme zorunluluğu getirmemiştir.

```
public Date getDate(Calendar cal) {  
  
    return cal.getTime();  
}
```

Error: Programda ciddi hatalar meydana geldiğinde oluşur. Kesinlikle recover edemeyiz.



Try- Catch Finally Bloğu

- Bir hatayı yakalamak için kod bloğunu try kod bloğu içine almayız.
- Try bloğu içerisinde bir hata alınırsa Exception objesi oluşur. Exception objelerini catch blokları içerisinde handle edebiliriz.

```

try{
}
catch(Exception type e){
}
catch(Exception type e){
}

```

- Catch içerisinde farklı istisnaları yönetebiliriz. Alttaki örneği inceleyebilirsiniz.

```

try{
}
}catch(FileNotFoundException e){
    System.out.println("Dosya Bulunamadı")
}catch(IOException e){
    throw new IOException();
}

```

- Java 7 ile gelen özellik ile Exception' ları OR' layarak handle edebilirsiniz.

```

catch(IOException | FileNotFoundException e){
}

```

Finally Bloğu :

Uygulama sırasında catch bloğu execute edildikten sonra exception fırlatılmadan bu bloğun execute edilmesi garanti edilir.

Alttaki örneği inceleyelim :

```
PrintWriter printWriter = null;
try {
    printWriter = new PrintWriter(new File("C:\\Users\\ssakinmaz\\Test.txt"));
    printWriter.print("Serkan");
} catch (FileNotFoundException e) {
    e.printStackTrace();
}finally{
    printWriter.close();
}
```

Hata alınsa da alınmasa da printWriter.close(); ile stream ve bu stream ile ilişkili kaynaklarla bağlantı kesilir.

Your Code ...

```
1 public class MyClass {
2     static void throwOne() throws IllegalStateException{
3         System.out.println("Inside throwOne.");
4         throw new IllegalStateException("demo");
5     }
6     public static void main(String args[]) {
7         try{
8             throwOne();
9         }
10        catch(IllegalStateException e){
11            System.out.println("Caught: "+e);
12        }
13    }
14 }
15
```

Exception Metodları :

getMessage(); Exception ile ilgili detay mesajı gösterilir.

toString(); Exception tipi ile beraber hata mesajını gösterir.

e.printStackTrace(); Hata mesajı ile birlikte stack trace gösterir.

EXCEPTION SINIFI OLUŞTURMA

Kodumuzun yapısına göre kendi exception sınıfımızı oluşturabiliriz.

Alttaki örnekte exception sınıfından extends edilen yeni bir exception sınıfı yazılmıştır.

```

public class MyException extends Exception{

    public MyException(String message){
        super(message);
    }

}

```

Kod içinde ise ;

throw new MyException("Hata Mesajı"); şeklinde kullanılabilir.

İŞ PARÇACIKLARI

Thread sınıfı iş parçacığını başlatan sınıftır.

İşlemcinin boş zamanına göre işlem yapar. Her iş parçacığının öncelik sırası vardır. Öncelik sırası ilerleyen rütbelerde da ayrıntılı alınır. Tek CPU' ya sahip bilgisayarlarda aynı anda bir parçacığı çalıştırır. Bu işlemleri yapan JVM (Java sanal Makinesi) ' dir.

```

package javaapplication34;
class Isg extends Thread{
    public void run() {
        for (int i=0; i<5 ; i++){
            System.out.println(this.getName()+"-->" +i);
        }
    }
}
public class JavaApplication34 {

    public static void main(String[] args) {
        Isg i1=new Isg();
        Isg i2=new Isg();
        i1.start();
        i2.start();
    }
}

```

Çıktısı ;

```

Output - JavaApplication34 (run) X
run:
Thread-0-->0
Thread-0-->1
Thread-0-->2
Thread-0-->3
Thread-0-->4
Thread-1-->0
Thread-1-->1
Thread-1-->2
Thread-1-->3
Thread-1-->4
BUILD SUCCESSFUL (total time: 0 seconds)

```

Run() metodu :

Thread işlemlerini yapan ana metottur.

Start(); Thread' leri başlatan metottur.

Sleep(); Thread' leri belli zaman aralıklarında uyutur.

Robot.java sınıfı aşağıda gösterilmiştir.

```
package javaapplication35;

public class Robot extends Thread {
    private int donguSayisi;
    public Robot(String isim, int donguSayisi){
        super(isim);
        this.donguSayisi=donguSayisi;
    }
    public void run(){
        try{
            if(donguSayisi==0){
                return;
            }
            for(int i=0; i<donguSayisi; i++){
                System.out.println(this.getName()+"-->"+i);
            }
        }
        catch(Exception e){
            System.err.println("404 NOT FOUND");
        }
    }
}
```

Main;

```
package javaapplication35;

public class JavaApplication35 {

    public static void main(String[] args) {
        Robot a=new Robot("A" ,3);
        Robot b=new Robot("B",4);
        Robot c=new Robot("C",5);
        Robot d=new Robot("D",6);

        a.start();
        b.start();
        c.start();
        d.start();
    }
}
```

Stop (); metodu Thread' leri durdurur.

Sleep metodunun kullanıldığı Thread örneği;

```
package javaapplication36;

public class UyuUyan extends Thread {
    public void run() {
        int sayac=0;
        long d=1000;
        try{
            while (true) {
                System.out.println("Uyuyor..");
                Thread.sleep(d);
                sayac=sayac+1;
                if(sayac>=5) {
                    return ;
                }
            }
        } catch (Exception ex) {
            System.err.println("Thread Uyumuyor..");
        }
    }
}
```

Main;

```
package javaapplication36;

public class JavaApplication36 {
    public static void main(String[] args) {
        UyuUyan rl=new UyuUyan();
        rl.start();
    }
}
```

Çıktısı; 5 saniye boyunca 1 saniye aralıklarda ekrana Uyuyor.. yazar.

Thread' in aktif olup olmadığını gösteren yol isAlive(); metodudur.

```
package javaapplication36;

public class JavaApplication36 {
    public static void main(String[] args) {
        UyuUyan rl=new UyuUyan();
        System.out.println(""+rl.isAlive());
        rl.start();
    }
}
```

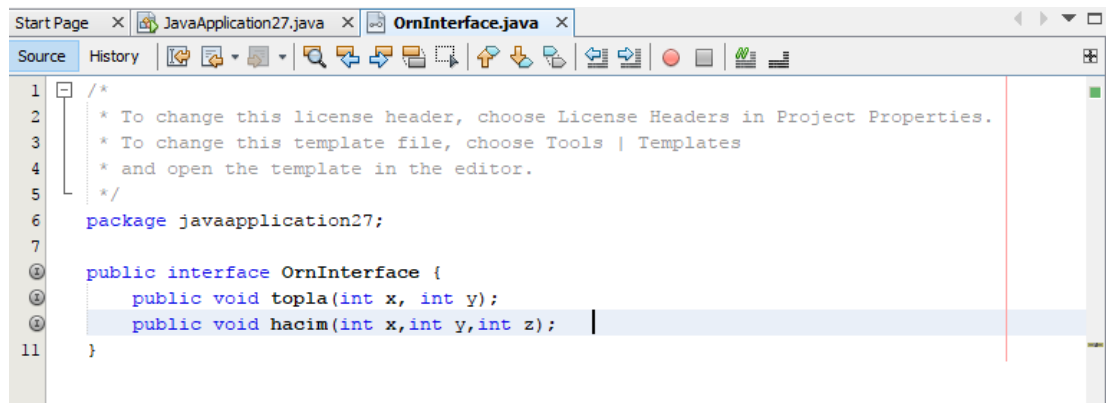
Main yordamında kontrol edilen Thread;

Çıktı olarak ilk false yazar. Yani Thread uyumuyor. Sonra Thread' i başlatır.

INTERFACE (ARAYÜZLER)

Java' da arayüz soyut sınıfların yerine kullanılır. Ama soyut sınıftan farklı ve daha kullanışlıdır. Arayüz kullanarak, bir sınıfın neler yapacağını belirlerken, onları nasıl yapacağını gizleyebiliriz. Arayüzün yapısı sınıfın yapısına benzese de aralarında önemli farklar vardır.

- Arayüz, interface anahtar sözcüğü ile tanımlanır. Arayüz abstract metotlar içerir.
- Arayüz, anlık(instance) değişkenler içeremez. Ancak belirtkeleri konmamış olsalar bile, arayüz içindeki değişkenler final ve static ve olur. Bu demektir ki arayüzde tanımlanan değişkenler, onu çağıran sınıflar tarafından değiştirilemez.
- Arayüz, yalnızca public ve ön tanımlı (default) erişim belirtkisi alabilir başka erişim belirtkisi alamaz.
- Public damgalı arayüz public damgalı class gibidir. Her kod ona erişebilir.
- Erişim damgasız arayüz, erişim damgasız class gibidir. Bu durumda, arayüze, ait olduğu paket içindeki bütün kodlar ona erişebilir. Paket dışındaki kodlar erişemez.
- Arayüzler, public erişim belirtkisi ile nitelenmişse, içindeki bütün metotlar ve değişkenler otomatik olarak public olur.
- Bir sınıf birden çok arayüze çağrılabilir.
- Aynı arayüzü birden çok sınıf çağırabilir.
- Sınıftaki metotlar tam olarak tanımlıdır, ama arayüzde metotların gövdesi yoktur. Onlar abstract metotlardır. Metodun tipi, adı, parametreleri vardır, ama gövde tanımı yoktur; yani yaptığı iş belirli değildir. Metotların gövdesi, o arayüze çağıran sınıf içinde yapılır. Böylece bir metot, farklı sınıflarda farklı tanımlanabilir. Bu özellik, Java' da polimorfizmi olanaklı kılan önemli bir niteliktir.

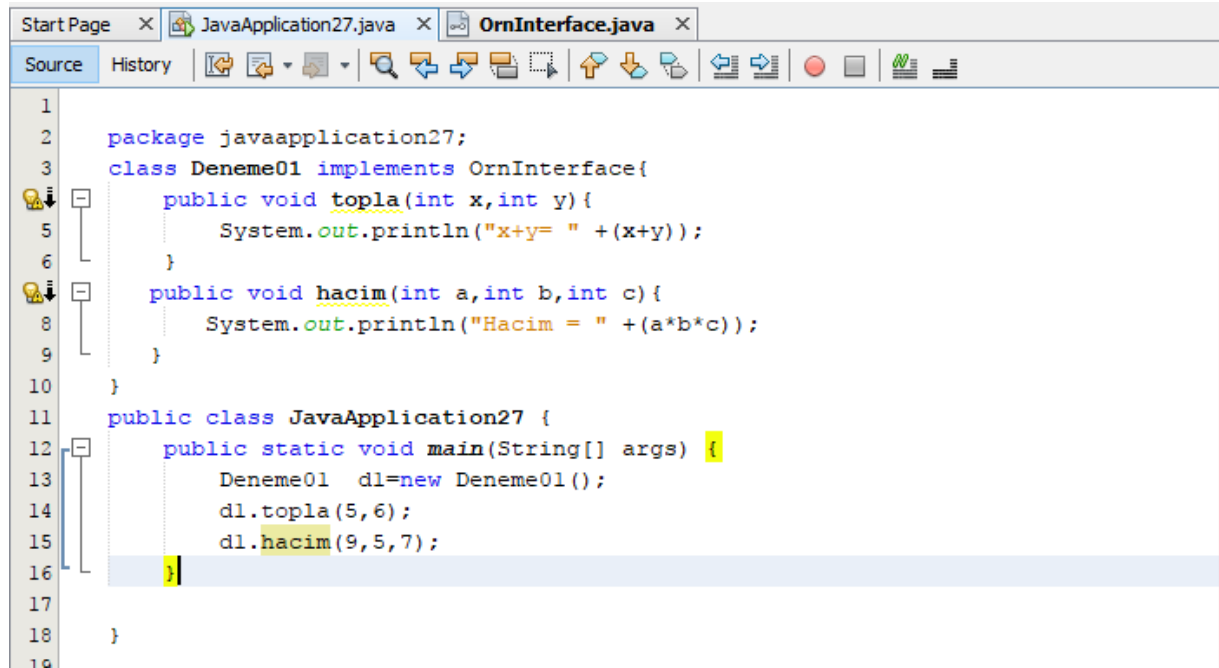


```
1  /*
2  * To change this license header, choose License Headers in Project Properties.
3  * To change this template file, choose Tools | Templates
4  * and open the template in the editor.
5  */
6  package javaapplication27;
7
8  public interface OrnInterface {
9      public void toplu(int x, int y);
10     public void hacim(int x,int y,int z);
11 }
```

İmplements: Bir arayüzü sınıfa miras almak için kullanılır.

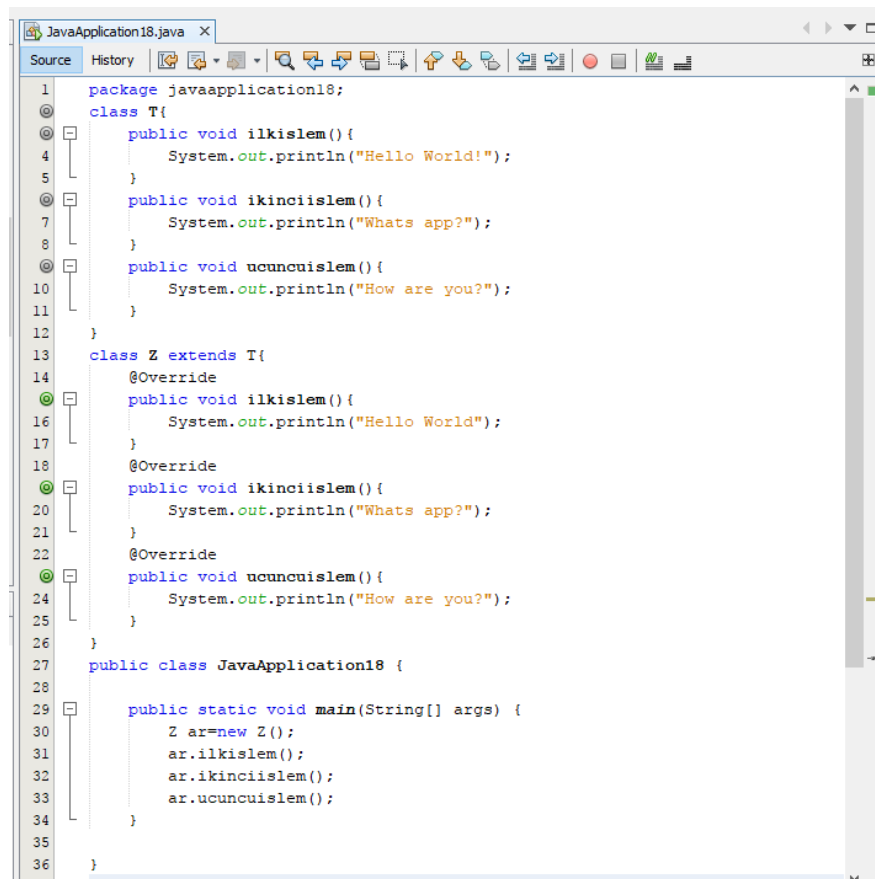
Herhangi bir Java sınıfı implements direktifini kullanarak, bir interface sınıfının sahip olduğu metotları implemente edebilir. Interface sınıflarında sadece methodlar deklare edilebilir ve implemente edilemez. İmplementasyonu alt sınıflar üstlenir.

Arayüzün main metodunda kullanılan hali ;



```
1 package javaapplication27;
2 class Deneme01 implements OrnInterface{
3     public void toplama(int x,int y){
4         System.out.println("x+y= " +(x+y));
5     }
6     public void hacim(int a,int b,int c){
7         System.out.println("Hacim = " +(a*b*c));
8     }
9 }
10
11 public class JavaApplication27 {
12     public static void main(String[] args) {
13         Deneme01 d1=new Deneme01 ();
14         d1.toplama(5,6);
15         d1.hacim(9,5,7);
16     }
17 }
18 }
19 }
```

POLIMORFİZM (ÇOK BİÇİMLİLİK)



```
1 package javaapplication18;
2 class T{
3     public void ilkislem(){
4         System.out.println("Hello World!");
5     }
6     public void ikincislem(){
7         System.out.println("Whats app?");
8     }
9     public void ucuncuslem(){
10        System.out.println("How are you?");
11    }
12 }
13 class Z extends T{
14     @Override
15     public void ilkislem(){
16         System.out.println("Hello World");
17     }
18     @Override
19     public void ikincislem(){
20         System.out.println("Whats app?");
21     }
22     @Override
23     public void ucuncuslem(){
24         System.out.println("How are you?");
25     }
26 }
27 public class JavaApplication18 {
28     public static void main(String[] args) {
29         Z ar=new Z();
30         ar.ilkislem();
31         ar.ikincislem();
32         ar.ucuncuslem();
33     }
34 }
35 }
36 }
```

Türemiş sınıflardan oluşan aynı işlemleri farklı sınıflar altında yapan yöntemdir.

Ana sınıf belirlendikten sonra türemiş bir sınıf oluşturmak için extends metodunu kullanırız.

```
package javaapplication37;
class AnaSınıf{
}
class Turemissınıf extends AnaSınıf{
}
public class JavaApplication37 {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }
}
```

KLAVYEDEN DEĞER ALMA (SCANNER SINIFI)

Kullanıcıdan veri almak için scanner sınıfına ihtiyaç duyarız. Scanner sınıfının paketlerini kullanacağımız ortamda tanımlamamız gerekmektedir.

```
package javaapplication37;
import java.util.Scanner;
public class JavaApplication37 {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Scanner k=new Scanner(System.in);
        int ar=k.nextInt();
        System.out.println(ar);
    }
}
```

İmport metodu ile Scanner sınıfını fonksiyona tanımlarız. Ardından Scanner sınıfının içinde bir nesne oluştururuz. Daha sonra nextInt (); ile bu nesneyi integer bir nesneye dönüştürürüz. Eğer string, double, float' a dönüştürmek istiyorsak next' ten sonra int yerine string, double veya float yazabiliriz.

KALITIM

Overriding Metodu;

Herhangi bir alt sınıfta süper sınıfa ait bir metot aynı isim, aynı parametre ve dönüş tipi ile kullanılırsa overriding yapmış oluruz. Alt sınıfta yeni bir nesne ürettiğimizde süper sınıfa göre değil yeni sınıfa uygun çalışmaktadır.

SOYUT KAVRAM, THIS, FİNAL

Soyut Kavram (Abstract)

Nesne yönelimli programlama yaklaşımına göre bir sınıfın bütün metotları belirli olmak zorunda değildir. Kısaca bir mevhumun mücerret olması durumunda nesne yönelimli programlamadaki ifade biçimidir. Örneğin bir çalışan sınıfını düşünelim. Her çalışanın bir maaş aldığını biliriz ama her çalışanın maaş hesaplaması farklı şekillerde yapılır. İşte bu durumda çalışan sınıfının bir maaş fonksiyonu bulunur ve bu maaş hesaplama fonksiyonunu her çalışan sınıfı kendisine göre yeniden düzenler.

```
public abstract class Calisan {  
    public abstract int maas();  
}
```

Yukarıdaki örnek sınıfın ve içerdiği örnek metodun ikisi de soyuttur. Basitçe soyut bir method içeren bir sınıf soyuttur. Bir methodun soyut olması ise içeriğinin tanımlanmamasıdır. Örneğin yukarıdaki maaş fonksiyonunun içeriği yazılmamış sadece prototipi yazılmıştır.

Şimdi yukarıdaki bu sınıftan miras alan yeni bir sınıf tanımlayalım.

```
public class Sekreter extends Calisan{  
    public int maas() {  
        return 1000;  
    }  
}
```

Yukarıda da gösterildiği gibi sekreter sınıfı çalışan sınıfını miras almıştır (inheritance) ve dolay(abstact method, mücerret fiil) olduğu için sekreter sınıfının iki seçeneği bulunmaktadır. Ya bu sınıf fonksiyonunun içeriğini dolduracak ve soyut olmaktan kurtulacak(overriding) ya da bu sınıfın içerisinde bu tanımlama yapılmayarak bu sınıf da soyut olacaktır.

```
public abstract class Sekreter extends Calisan{  
    public int maas() {  
        return 1000;  
    }  
}
```

Yukarıda yazılmış olan sekreter örneği ise maaş fonksiyonunun somutlaştırılmış halde bırakıldığı örnektir. Bu durumda sekreter sınıfında miras kalan bir soyut fonksiyon bulunmakta(maaş) ve dolayısıyla bu sınıfta soyut olmaktadır.

Abstract Metotlar;

Abstract sınıflar, yan soyut sınıflardır. Soyut sınıfların ortak özelliklerini kullanabilmekteyiz. Soyut sınıflar kendisinden türeyen sınıflardır. Soyut sınıflardan nesne oluşturulmaz. Bunun yerine extends edilerek yeni sınıflara o özelliği kullanarak soyut metotlar üretilir.

Bazı durumlarda, yapılacak işlere uyan somut bir üst sınıf ve ona ait somut metotlar tanımlamak mümkün olmayabilir. Böyle durumlarda, soyut bir üst-sınıf ve ona ait soyut metotlar tanımlamak sorunu kolayca çözebilir.

Soyut metot (Abstract class) adı ve parametreleri olduğu halde gövdesi olmayan bir metottur. Dolayısıyla belirli bir iş yapmaz. O, alt-sınıflarda örtülür (overriding).

This Metodu

Java' da bir metodun içinde o metodun ait olduğu sınıftan yaratılacak nesneyi veya o nesnenin bir alt değişkenini tanımlamamız gerektiğinde kullanılan deyim **this** denir. Bulduğumuz sınıfta nesne daha tanımlanmadığı için bu nesneyi direkt olarak kullanamayız. İşte this ile ait olduğu class içinde yaratılan methodlar o class' ın nesnesini kullanabilmektedir. Basit olarak söylemek gerekirse this anahtar kelimesi o anda hangi nesne üzerinde işlem yapılıyorsa o nesnenin referansını döndürür.

This anahtar kelimesi kullanarak aynı sınıfın içinde diğer yapıcı methodları da çağırabilme yeteneğine sahibiz. Diğer bir husus ise this anahtar kelimesini static methodlar içinde kullanamayız. Çünkü this kelimesi sınıfa ait nesnelerin oluşturulmasıyla bellekte yer tutar.

Aşağıdaki örnekte this kullanılmamıştır. Class Ornek içindeki urunadi, marka ve modeli constructor methodumuz olan örnek içinde nasıl kullanabiliriz.

```
1 public class Ornek{
2
3 private String urunadi;
4 private String marka;
5 private String model;
6
7 public Ornek(String urunadi, String marka, String model) {
8 urunadi= urunadi;
9 marka = marka;
10 model = model;
11 System.out.println(urunadi + " " + marka + " " + model);
12 }
13 }
```

This' li kullanım ile farkı göreceksiniz.

```
1 public class Ornek{
2
3 private String urunadi;
4 private String marka;
5 private String model;
6
7 public Ornek(String urunadi, String marka, String model) {
8 this.urunadi= urunadi;
9 this.marka = marka;
10 this.model = model;
11 System.out.println(urunadi + " " + marka + " " + model);
12 }
13 }
```

Ayrıca this ile aynı sınıf içerisindeki diğer yapıcı methodları da çağırabiliriz. Aşağıdaki kodu da inceleyiniz.

```
1 public class Ornek{
2
3 private String urunadi;
4 private String marka;
5 private String model;
6
7 public Ornek(){
8 this("a","b","c");
9 }
10 public Ornek(String urunadi, String marka, String model) {
11 this.urunadi= urunadi;
12 this.marka = marka;
13 this.model = model;
14 System.out.println(urunadi + " " + marka + " " + model);
15 }
16 }
```

Final ve Static

Static Değişkenler:

Java' da class içerisinde tanımladığımız static değişkenlere class variable denir. Static değişkenler tüm objeler için ortaktır. Aynı tipten tüm objeler için ortak olan static alanları yapabiliriz.

Mesela Matematik' te sıkça kullanılan π sayısını static olarak tanımlayabiliriz.

```
package javaapplication38;  
class Math{  
    static double PI=3.14;  
}
```

Static Methodlar:

Static methodlar ise static değişkenler gibi tüm objelerin ortak olarak kullanıldığı methodlardır. Static methodlar class ismiyle çağırılır.

Final Değişkenler:

Final değişkenlerde ilk değer verildikten sonra güncellenemezler. Genellikle final ve static bir arada kullanılarak Constant' ları oluşturur.

Final Sınıf ve Methodlar:

- Final sınıflar extends edilemez.
- Final methodlar ise override edilemez.

Java' da sıkça kullanılan String sınıfı finaldir.

LİST TİPLERİ İLE ÇALIŞMA

ArrayList Sınıfı

ArrayList () → Başlangıç olarak sığası 10 terim olan boş bir dizi oluşturur.

ArrayList (Collection c) →Parametrede belirtilen koleksiyona ait öğeler içeren bir liste oluşturur. Öğeler, iterator' un belirlediği sırayla dizilirdirler.

ArrayList (int initialCapacity) →Sığası (capacity) parametrenin belirlediği sayıda olan bir dizi oluşturur.

ArrayList <E>

Java.util.ArrayList <E> sınıfı kendiliğinden büyüeyebilen dizi (array) kurar ve temelde Collections Vektor sınıfı ile aynı sayılır. ArrayList sınıfının özellikleri şunlardır.

- Veri eklenip silindikçe ArrayList kendi uzunluğunu otomatik olarak ayarlar.
- ArrayList listesine erişim işlemi $O(1)$, sokuşturma (insertion) işlemi $O(n)$ ve silme işlemi (diletion) işlemi $O(n)$ zaman karmaşasına sahiptir.
- ArrayList sokuşturma, silme ve arama eylemlerini yapan methodlara sahiptir.
- ArrayList üzerinde foreach döngüsü, iteratörler ve indexler yardımıyla gezinilebilir.

ArrayList ve Array Arasında Seçim

Programcı, ne zaman ArrayList ne zaman Array kullanılması gerektiği konusunda ikileme düşebilir. Eğer, depoya konulacak öge sayısı belirli ve o sayı sık sık değişmiyorsa array seçimi uygun olur. Ama öge sayısı baştan bilinmiyorsa ya da sık sık değişiyorsa ArrayList doğru bir seçimdir. Tabii, buna ek olarak şunu söylemeliyiz: ArrayList<E> nesnelerin depolanması içindir. İlkel veri tiplerini depolamak için array seçilmesi daha uygun olur. Bütün bunların ötesinde ArrayList sınıfı List arayüzünün methodlarını kullanma yeteneğine sahiptir dolayısıyla array yapısına oranla programcıya daha çok kolaylık sağlar.

Başlıca ArrayList Methodları

Sun, generic tipler için <E> simgesinin kullanılmasını öneriyor. E simgesi koleksiyon içindeki öğelerin veri tipi yerine geçer. Aşağıdaki methodlarda şu bildirimlerin yapıldığını varsayacağız.

```
int i;  
ArrayList<E> a;  
E e;  
Iterator<E> iter;  
ListIterator<E> liter;  
E[] earray;  
Object[]
```

void add(int index, Object element) → Listede indisi belirtilen yere öge sokuşturur(insert). O indisten sonraki öğelerin konumları birer geriye kayar.

boolean add(Object o) → Parametrede verilen nesneyi listenin sonuna ekler.

boolean addAll(Collection c) → Parametrede verilen koleksiyonun bütün öğelerini listenin sonuna ekler. Ekleme sırası koleksiyonun iteretörü' nün belirlediği sıradadır.

boolean addAll(int index, Collection c) → Belirtilen index' ten başlayarak verilen koleksiyonu listeye yerleştirir.

void clear () → Listedeki bütün öğeleri siler; boş liste haline getirir.

Object clone () → ArrayList kılığının (instance) bir kopyasının yapar.

boolean contains (Object o) → Parametrede belirtilen nesne listede varsa true değerini alır.

void ensureCapacity (int minCapacity) → Gerekliyorsa ArrayList nesnesinin sığasını artırarak, parametrenin belirlediği minimum sığa kadar öğeyi depo edebilmesini sağlar.

Object get (int index) → İndeksi belirtilen öğeyi verir.

int indexOf (Object o) → Parametrede verilen nesnenin listedeki indeksini verir. Nesne listede yoksa -1 değerini alır.

boolean isEmpty () → Listenin boş olup olmadığını kontrol eder.

int lastIndexOf (Object elem) → Belirtilen öğenin listedeki son indeksini gösterir.

Object remove (int index) → İndeksi verilen öğeyi siler.

KOMPOZİYON

Başka bir sınıfı sınıfın içinde deęişken olarak kullanmaya kompozisyon denir. C++’ da bir sınıf birden fazla sınıftan miras alabilir ama Java’ da bunu yapamayız. Java’ da bir sınıf yalnızca bir sınıftan türeyebilir. Eğer sınıfın başına final yazarsak o sınıftan yeni bir sınıf türetmeyiz. Başkası bizim sınıfımızı kalıtım yoluyla kullanmaması için final kullanabiliriz. Ama kompozisyon ile bizim sınıfımızı başkası kullanabilir.

Nokta.java dosyası;

```
Source History | [Icons] | [Red Circle]
1 package javaapplication38;
2 class Nokta {
3     int x,y;
4     public void yaz () {
5         System.out.println("[X="+x+":Y="+y+"] ");
6     }
7     public void nokta () {
8         x=0;
9         y=0;
10    }
11    public void nokta(int x,int y) {
12        this.x=x;
13        this.y=y;
14    }
15 }
16
```

RenkliNokta.java dosyası;

```
Source History | [Icons] | [Red Circle]
1 package javaapplication38;
2 class RenkliNokta extends Nokta{
3     String renk;
4     public void yaz() {
5         System.out.println("[X="+x+":Y="+y+"] ");
6         System.out.println("Renk= "+renk);
7     }
8 }
9
```

Main yordamı;

```
Source History | [Icons] | [Red Circle]
1 package javaapplication38;
2 public class JavaApplication38 {
3
4     public static void main(String[] args) {
5         RenkliNokta r1= new RenkliNokta();
6         r1.yaz();
7     }
8 }
9
```

YAPILANDIRICI METHODLAR (CONSTRUCTOR)

Methodlar belli görevleri verdiğimiz, işlemler sonucunda bize bir geri dönüş değeri verebilen ya da geri dönüşü olmadan belirli işlemleri tamamlayan yapılardır. Programımızda ihtiyaç olduğu yerde methodları çağırarak işlemleri gerçekleştirebiliriz.

Fakat bazı durumlarda bizim methodu çağırılmamıza gerek kalmadan, bir sınıftan nesne oluşturduğumuz anda bazı işlemlerin yerine getirilmiş halde olmasını isteyebiliriz. Neden böyle bir şeye ihtiyaç duyarız, çünkü ileride yazacağımız karmaşık kodlarda ve büyük çaplı bir projede bu tür durumlar kaçınılmazdır. Bazen fazla koddan kurtulmak, bazen de mecbur kalmaktan dolayı bu şekilde methodlara ihtiyaç duyarız. Bu durumda Yapılandırıcı Methodlar devreye girer.

Yapılandırıcı methodlar nesne oluşturduğumuz anda çalıştırılan methodlardır. Herhangi bir geri dönüş tipi yoktur. Evet, ilginç ama void tipinde dahi bir geri dönüşleri yoktur. Bir yapılandırıcının yaptığı iş, bir nesneyi ilk kullanıma hazırlamaktır.

Yapılandırıcı methodları şu şekilde özetleyebiliriz.

- Yapılandırıcıların erişim belirteci mutlaka public olmalıdır.
- Yapılandırıcıların adları buldukları sınıfın adıyla aynı olmalıdır.
- Yapılandırıcı method çağırılırken new anahtar sözcüğü kullanılır.
- Yapılandırıcılar bellekte nesneye bir yer ayrılmasını sağlar.
- Yapılandırıcılar her çağırılışlarında yeni bir nesne oluştururlar.

```
1 // Created by MahmutMUTLU 1 2014
2
3 class YapilandiriciMetot {
4
5     int ycap;
6     double cevre;
7     double alan;
8     final static double pi = 3.14;
9
10    void bilgileriYaz(YapilandiriciMetot d) {
11        System.out.println("Dairenin Yarıçapı : " + d.ycap);
12        System.out.println("Dairenin Alanı : " + d.alan);
13        System.out.println("Dairenin Cevresi : " + d.cevre);
14        System.out.println();
15    }
16
17    public YapilandiriciMetot(int r) {
18        ycap = r;
19        alan = pi * r * r;
20        cevre = 2 * pi * r;
21    }
22
23    public static void main(String args[]) {
24
25        YapilandiriciMetot d1 = new YapilandiriciMetot(3);
26        d1.bilgileriYaz(d1);
27        YapilandiriciMetot d2 = new YapilandiriciMetot(11);
28        d1.bilgileriYaz(d2);
29    }
30 }
```


Yukarıdaki örnekte görüldüğü üzere yapılandırıcı methodumuzu daha nesneyi oluştururken kullanmış olduk. New anahtar sözcüğüyle birlikte verdiğimiz değerler, gerekli işlemler yapılarak yapılandırıcı method içerisindeki değişkenlere atandı. Bu değişkenleri de ekrana bastık.

Dikkat ettiyseniz aynı değişkenleri ekrana yazıyoruz fakat farklı sonuçlar alıyoruz. Çünkü yapılandırıcı methodlar her nesne için ayrı ayrı sonuçları saklar. Yukarıdaki maddelerde belirtildiği gibi her çağırıldıklarında yeni bir nesne oluştururlar. Bu sebeple değişkenlerin değerlerinde herhangi bir çakışma olmaz.

OVERLOADİNG (AŞIRI YÜKLEME)

Java, aynı isme sahip birden fazla methodlar tanımlamamıza izin vermektedir. Bu şekilde birden fazla aynı methodun yazılmasına **overloading** yani aşırı yükleme denir.

Overloading yapabilmek için aynı isimdeki methodlarımıza farklı parametreler göndermemiz gerekiyor. Burada parametrelerimizin türü veya sayısının farklı olması yeterli olacaktır.

```
1 // Created by MuhammedTutar | 2014
2
3 package org.kod5;
4
5 public class MethodOverloading {
6     void kareKok(int x){
7         x = x*x;
8         System.out.println("Integer X: " + x);
9     }
10
11     void kareKok(double x){
12         x = x*x;
13         System.out.println("Double X: " + x);
14     }
15
16     public static void main(String[] args) {
17         MethodOverloading over = new MethodOverloading();
18         over.kareKok(5);
19         over.kareKok(2.87);
20     }
21 }
22
23
24
25
26
27
28 }
```

Örneğin karekök() metodunu aşırı yüklemiş oluyoruz. Tek parametre alıyor ancak türleri farklı. Aynı türde farklı sayıda parametre göndererek de aşırı yükleme işlemini gerçekleştirmiş oluyoruz. Bu program parçası aşırı yüklenmiş methoda gelen değer türüne göre değişmektedir.

OVERRIDE (METHOD EZME)

Override Türkçede ezme, geçersiz kılmak anlamındadır. Java' da override ile yapılan şey üst sınıftan alınan bir özelliği alt sınıfta değiştirmektir.

Üst sınıftan override edilen method alt sınıfta aynı isimle tanımlanmalıdır. Dönüş tipi aynı olmalıdır.

Private, static, final methodları override yapılamaz. Ancak tekrar tanımlanarak kullanılabilir.

Override edilen methodla argüman listesi aynı olmalıdır. Override edildiği methodtan daha kısıtlı erişime sahip olamamalıdır.

JavaApplication39.java Dosyası;

```
6 package javaapplication39;
7
8 import java.util.Date;
9
10 public class JavaApplication39 {
11     protected double salary;
12     protected String name;
13     protected Date birthDate;
14     public String getDetails() {
15         return "Name:" +name+ "\n"+"Salary: "+salary;
16     }
17     public static void main(String[] args) {
18         // TODO code application logic here
19     }
20 }
21 }
```

Manager.java Dosyası ve override edilmiş method;

```
6 package javaapplication39;
7
8 public class Manager extends JavaApplication39 {
9     protected String department;
10     @Override
11     public String getDetails() {
12         return "Name: " + name + "\n"
13             + "Salary: " + salary + "\n"
14             + "Manager of: " + department;
15     }
16 }
```

Manager sınıfı JavaApplication sınıfından farklı olarak String tipinde “department” adında bir değişkene sahip ve getDetails() methodu da override edilmiş durumda. Böylece Manager sınıfının kendisine ait getDetails() methodu olmuş oluyor ve override edildiği methoda ek olarak Manager’ ın department değişkenini de geri döndürüyor.

Yukarıdaki örnekte JavaApplication39 sınıfındaki getDetails() methodu public olarak tanımlanmış Bu demek oluyor ki Manager sınıfındaki override edilmiş getDetails() methodu protected, private veya default olamaz. Çünkü bu 3 erişim belirleyici methodun erişimini daha kısıtlı hale getirir.

Ayrıca bu noktada **super** anahtar sözcüğünden biraz bahsetmek faydalı olacaktır. Eğer override ettiğimiz methodu tamamen değiştirmek istemiyorsak, sadece geliştirmek ve ekleme yapmak istiyorsak o zaman super anahtar kelimesini kullanabiliriz. Bu sayede parent class’ ın methodlarını ya da dataalarını çekebiliyoruz. Bu küçük ayrıntı olarak şunu da eklemekte fayda var extends edilen sınıf,

başka bir sınıftan inherit edilmişse o en tepedeki class' a da super anahtar sözcüğü ile ulaşabiliriz.

```
1 public class Manager extends Employee {  
2     protected String department;  
3  
4     public String getDetails(){  
5         return super.getDetails() +  
6             "\nManager of: " + department;  
7     }  
8  
9 }
```