

OCPJP YAZILARI  
LEVENT ERGÜDER

6

MAKE THE FUTURE JAVA



| injavawetrust  
azizfarmer.wordpress.com

Yazar : Levent ERGÜDER

Hazırlayan : Aziz ÇİFTÇİ

Kaynak: injavawetrust ocp yazıları

NOT: Orijinal yazılar için [injavawetrust.com](http://injavawetrust.com) a bakınız. Lütfen bu pdf hakkındaki yorumlarınızı [azizfarmer.wordpress.com](http://azizfarmer.wordpress.com) 'a iletiniz.

ÖNSÖZ .....	6
Bölüm-1 .....	7
Pure Java – 01 Legal Identifiers ve Naming Standards .....	8
Pure Java – 02 Class Declaration and Access Modifier .....	11
Pure Java – 03 Interface Declaration .....	14
Java 8 Interface New Features .....	17
Pure Java – 04 Declare Class Members .....	22
Pure Java – 05 Declare Class Members – 02 .....	28
Pure Java – 06 Declare Class Members – 03 Variable .....	32
Pure Java – 07 Declare Enums .....	38
Pure Java – 08 Declare Var-Args .....	43
Pure Java – 09 Java Tricks .....	45
Bölüm-2 .....	48
Pure Java – 10 Object Orientation – 01 Encapsulation IS-A HAS-A .....	48
Pure Java – 11 Object Orientation – 02 Polymorphism (Polimorfizm) .....	52
Pure Java – 12 Object Orientation – 03 Overridden .....	53
Pure Java – 13 Object Orientation – 04 Overloaded .....	58
Pure Java – 14 Java Tricks – Overloaded vs Overridden .....	62
Pure Java – 15 Reference Variable Casting .....	63
Pure Java – 16 Implementing an Interface .....	66
Pure Java – 17 Return Type .....	69
Pure Java – 18 Constructor – 01 .....	73
Pure Java – 19 Constructor – 02 .....	75
Pure Java – 20 Static Variables and Methods .....	81
Pure Java – 21 Coupling & Cohesion .....	87
BÖLÜM – 3 .....	90
Pure Java – 22 Literals .....	90
Pure Java – 23 Assignment Operators .....	94
Pure Java – 24 Variable Scope .....	97
Pure Java – 25 Initialize Instance Variables .....	100
Pure Java – 26 Initialize Local Variables .....	105
Pure Java – 27 Passing Variables into Methods .....	110
Pure Java – 28 Array Declaration .....	114
Pure Java – 29 Array Construction .....	115

Pure Java – 30 Array Initialization .....	118
Pure Java – 31 Legal Array Element Assignments .....	120
Pure Java – 32 Initialization Blocks.....	124
Pure Java – 33 Wrapper Classes – 01 .....	132
Pure Java – 34 Wrapper Classes – 02 .....	135
Pure Java – 35 Wrapper Classes – 03 .....	139
Pure Java – 36 More Overloading .....	144
Pure Java – 37 Garbage Collection .....	151
Pure Java – 38 the finalize().....	157
<b>BÖLÜM -4 .....</b>	<b>160</b>
Pure Java – 39 Operators – 01.....	160
Pure Java – 40 Operators – 02.....	164
Pure Java – 41 Operators – 03.....	168
Pure Java – 42 Operators – 04.....	171
<b>BÖLÜM- 5 .....</b>	<b>175</b>
Pure Java – 43 Flow Control – 01 .....	175
Pure Java – 44 Flow Control – 02 .....	179
Pure Java – 45 Loops .....	190
Pure Java – 46 break & continue.....	196
Pure Java – 47 Handling Exception – 01.....	200
Pure Java – 48 Handling Exception – 02.....	207
Pure Java – 49 Handling Exception – 03.....	209
Pure Java – 50 Handling Exception – 04 .....	212
Pure Java – 51 Handling Exception – 05.....	215
Pure Java – 52 Assertion Mechanism.....	217
<b>BÖLÜM-6 .....</b>	<b>221</b>
Pure Java – 53 String – 01.....	221
.....	222
.....	223
Pure Java – 54 String – 02 Methods .....	225
Pure Java – 55 StringBuffer & StringBuilder.....	229
Pure Java – 56 File and I/O – 01 .....	232
Pure Java – 57 File and I/O – 02 .....	237
Pure Java – 58 File and I/O – 03 .....	242

Pure Java – 59 File and I/O – 04 .....	245
Pure Java – 60 File and I/O – 05 – Serialization.....	249
Pure Java – 61 Dates, Numbers & Currency – Date & Calendar .....	259
Pure Java – 62 Dates, Numbers & Currency – DateFormat & SimpleDateFormat.....	266
Pure Java – 63 Dates, Numbers & Currency – Locale.....	277
Pure Java – 64 Dates, Numbers & Currency – NumberFormat & Currency & DecimalFormat .....	281
Pure Java – 65 Searching – Pattern & Matcher.....	288
Pure Java – 66 Tokenizing & Formatting .....	298
<b>BÖLÜM-7 .....</b>	<b>305</b>
Pure Java – 67 Generics & Collections – hashCode & equals & toString .....	305
Pure Java – 68 Generics & Collections – Collection API .....	315
.....	316
Pure Java – 69 Generics & Collections – ArrayList – 01.....	319
Pure Java – 70 Generics & Collections – ArrayList – 02.....	325
Pure Java – 71 Generics & Collections – Sort & Searching.....	329
Pure Java – 72 Generics & Collections – Set.....	346
Pure Java – 73 Generics & Collections – Map – 01 .....	357
Pure Java – 74 Generics & Collections – Map – 02 .....	366
Pure Java – 75 Generics & Collections – PriorityQueue .....	381
Pure Java – 76 Generics & Collections – Generics – 01.....	385
Pure Java – 77 Generics & Collections – Generics – 02.....	390
Pure Java – 78 Generics & Collections – Generics – 03.....	401
<b>BÖLÜM-8 .....</b>	<b>408</b>
Pure Java – 79 Nested Class – Inner Class .....	408
Pure Java – 80 Nested Class – Nested Interface .....	414
Pure Java – 81 Nested Class – Local Inner Class.....	417
Pure Java – 82 Nested Class – Anonymous Inner Class.....	427
Pure Java – 83 Nested Class – Static Nested Class .....	434
<b>BÖLÜM-9 .....</b>	<b>438</b>
Pure Java – 84 Thread – 01.....	438
Pure Java – 85 Thread – 02.....	443
Pure Java – 86 Thread – 03.....	448
.....	451
Pure Java – 87 Thread – 04.....	451

Pure Java – 88 Thread – 05.....	460
Pure Java – 89 Thread – 06 – synchronized .....	466
Pure Java – 90 Thread – 07 – Atomic & volatile.....	475
Pure Java – 91 Thread – 08 – Deadlock.....	477
Pure Java – 92 Thread – 09 – Thread Interaction.....	479
Bölüm-10 .....	486
Pure Java – 93 – Development – java & javac.....	486
.....	488
.....	488
Pure Java – 94 – Development – Using Classpaths .....	493
Pure Java – 95 – Development – Jar Files & Searching .....	498
Pure Java – 96 – Development – Static Imports.....	501
Online Java Egitim .....	504
Oracle Java SE sertifikayona yonetik egitim .....	504
Servlet&JSP / Oracle Web Component sertifikasyonuna yonetik egitim .....	505
More Java EE egitimi .....	506
AZİZ ÇİFTÇİ.....	507
https://azizfarmer.wordpress.com/about/.....	507

# ÖNSÖZ

Merhaba ben Aziz ÇİFTÇİ. Yaklaşık bir buçuk yıldır java aşığıyım. Javayı hiçbir dille aldatmadım 😊

Java aşkımlı belgelemek için çıktığım yolda tek Türkçe kaynak olan [injavawetrust.com](http://injavawetrust.com)'un yazılarını bir pdf altında toplamak istedim. Bunun başlıca iki sebebi var: birincisi bulduğum her ortamda internetin olmayışı. İkincisi bütün yazıları tek pakette toplamak .

Bunu hazırlamam yaklaşık 5 saat sürdü.(copy-paste ve birkaç düzenleme).aslında benim burda yaptığım pek bir şey yok. Asıl herşeyi yapmış kişi olan Levent ERGÜDER abime sonsuz teşekkürlerimi sunuyorum.

AZİZ ÇİFTÇİ

Yüzüncü Yıl Üniversitesi

Yazılım Topluluğu Başkanı

# Bölüm-1

Merhaba Arkadaslar,

Mart 2013 te basladigim ve yaklasik 2 yil suren [Oracle Certified Professional, Java SE 6 Programmer](#) notlarimin sonuna geldim.

Bu buyuk Java okyanusundan bir damla da olsa anlatmaya calistim. Java dunyasini gorunce insan gercekten cok az sey bildigini fark ediyor.

Bazi yazılarda ilgili sınavın disinda olan bilgiler de eklemeye calistim. Bundan sonraki surecte insallah yazıları gözden geçirip daha de genişletmek istiyorum.

Hedefimde güzel bir Java SE kitabı oluşturmak var.

Yazılarda olan herhangi bir hata anlasılamayan nokta gibi konular için geri bildirimde bulunabilirsiniz. Ortaya daha güzel bir kaynak çıkışına katkı olacaktır.

# Pure Java – 01 Legal Identifiers ve Naming Standards

Levent Erguder 24 March 2013 [Java SE](#)

Merhaba Arkadaslar,

Bu dersler **OCP Java SE 6 Programmer** sinavina dair notlarimdan olusacaktır. Bu sinava hazirlanirken bir kac kitaptan yararlandim. Turkce kaynaklarda olmayan detayları ve sinava yonelik dikkat edilecek noktalari da bu yazilarim boyunca paylasacagim.

Burada JDK kurulumu ve IDE kurulumundan bahsetmeyecegim, belirli seviyede Java bilgisi veya programlama



bilgisi oldugunu varsayıyorum

Javanin o buyulu dunyasina giris yapalim oyleyse,

## Class

**Class** yani **sinif** bir sablondur, peki neyin sablonudur bu ? **Class**, bir **object** ‘ in yani **objenin** sablonudur, peki ne is yapar **bireclass** ? Objenin durum [**state**] ve davranislarini [**behaviors(methods)**] larini tanimlamamizi saglar.

## Object

Calisma zamaninda Java Virtual Machine (JVM) , *new* anahtar kelimesi ile karsilastiginda, ilgili *class* tan bir *instance* olusturur (instance of a class) Bu objenin kendi durum[state] ve davranislari [behaviors(methods)]vardir.

## Legal Identifiers (Belirleyici)

Class, degisken (variable) , metot ve diger Java ogelerinin birer isme ihtiyaci vardir. Bu isimlere Javada *Identifier* denir. Java’da bir seylere isim verirken , ciddi anlamda problem yasayabiliriz, Java bu konuda



son derece hassastir

- Belirleyiciler (identifier), harf , \$ (currecy character) veya alt\_cizgi \_ (underscore) ile baslayabilir.
- Ilk karakterden sonra harf , \$ ve alt cizgiye ek olarak tabi ki rakam da kullanabiliriz.
- Belirleyicilerin uzunluk acisindan bir siniri yoktur.
- Java’nin anahtar kelimelerini (keyword) , belirleyicilere verilemez.

- Belirleyiciler büyük-küçük harfe duyarlıdır (case sensitive). Yani sayı ve Sayı farklı iki belirleyicidir.  
Geçerli belirleyicilere örnek;

```
int _sayi;  
int $sayi;  
int _____hebe_hube;  
int _$;  
int $10000;
```

Gecersiz belirleyicilere örnek;

```
int :a;  
int -d;  
int e#;  
int .f;  
int 7g;
```

: – # . gibi karakterler kullanılamaz ve rakam ile baslayamaz.

### Naming Standard

Burada Java'daki isimlendirme standartlarından bahsedeceğim, bunlar zorunlu olmamakla beraber defacto



kurallarıdır

#### Class ve Interface için ;

Class ve Interface'lerin ilk harfi büyük olmalıdır. Birden fazla kelime olduğunda her kelimenin ilk harfi büyük yazılır buna camelCase denir.

Class isimlerinin "isim" olması önerilir.

Dog  
Account  
PrintWriter

Interface isimlerinin "sifat" olması önerilir.

Runnable  
Serializable

#### Metot için;

Metot isimleri kucuk harfle baslamlı ve camelCase yapısına uygun olmalıdır ve fiil-isim çifti halinde tanımlanması uygundur.

```
getBalance  
doCalculation  
setCustomerName
```

#### Degiskenler icin;

Metotlarda olduğu gibi camelCase yapısına uygun, harfle başlayan mantıklı bir isimlendirme olmalıdır.

```
int hebele  
int hubele  
int a
```



gibi mantıksız anlaşılır isimlendirme olmasın

#### Sabit(Constant) icin;

Javada, sabitler static ve final anahtar kelimeleri kullanılarak oluşturulur. Tüm harfler büyük ve kelime aralarında alt\_cizgi olması önerilir.

```
MIN_HEIGHT
```

#### JavaBean Standards

Sınıfımızda *private* değişken olduğunu ve bunların getter/setterlarını düşünelim.

Getter için;

Eğer değişken boolean değilse, on ek *get* olmalı.

Eğer değişken boolean ise, *get* veya *is* olmalı.

public olmalı, arguman almamalı ve dönüs tipi olmalı.

Setter için;

On ek *set* olmalı.

public void olmalı, arguman almali.

#### JavaBean Listener

On ek olarak add veya remove olmalı.

Listener metodlarının son eki Listener ile bitmeli.

Örnek olarak ;

```
public void setMyValue(int v)
```

```
public int getMyValue()  
public boolean isMyStatus()  
public void addMyListener(MyListener m)  
public void removeMyListener(MyListener m )
```

Uygun olmayan isimlendirme ;

```
void setCustomerName(String s) public olmali  
public void modifyMyvalue(int v) modify olmamali  
public void addXListener (MyListener m) listener tipi eslesmeli
```

Burada uygun olmamaktan kasit hata vermesi degil, önerilenin disinda isimlendirmektir.

## Pure Java – 02 Class Declaration and Access Modifier

Levent Erguder 27 March 2013 Java SE

Merhaba Arkadaslar,

Bu dersimizde class declaration ve access modifier konusuna giris yapacagiz , bir kac ders boyunca devam edecegiz.

### Kaynak Dosya Tanimlamasi

- Javada , her kaynak dosya icin yalnızca bir adet *public* sinif tanimlanabilir.
- *public* tanimlanan sinif , dosya ismiyle mutlaka ayni olmak zorundadir. Ornegin ,  
*public class Employee* tanimlanmissa kaynak dosyasi mutlaka Employee.java olmalıdır.
- *package* tanimlamasi mutlaka en basta yer almalıdır.
- *import* tanimlamasi *package* tanimlamasi ile *class* tanimlamasi arasında yer almalıdır.
- *import* ve *package* tanimlamasi , bir kaynak dosyada tum sinif tanimlamalari icin gecerlidir.
- Bir kaynak dosya tabi ki birden fazla *public* olmayan sinif tanimlamasi icerebilir.

SourceFile.java

```
package chapter1.classDecAndModifier;
```

```
import java.io.*;
```

```
public class SourceFile {
```

```
}
```

```
class NonPublicClass {
```

```
}
```

## Modifiers ( Belirtec)

Javada modifier lar 2 ye ayrılır.

access modifiers : *public , protected , private*

non – access modifier : *strictfp, final , abstract*

Ilerleyen derslerde tüm bu belirtecleri derinlemesine inceleyecegiz.

Java ‘da 3 tane erişim belirteci vardır (*access modifier*) buna rağmen 4 tane erişim seviyesi (access level) vardır .

( default / package level )

Eğer bir değişkeni, metodu, sınıfı tanımlarken belirteç kullanmazsa default level olarak tanımlanırlar.

### Sınıf Erisimi (Class Access)

Java da sınıflar ya *public* olarak tanımlanır ya da default level (package level) tanımlanır, yani boş bırakılır.

Sınıflar için *protected* veya *private* tanımlaması söz konusu değildir !

### Default Access

class A , default level tanımlanmışsa ve class B den farklı pakette tanımlanmışsa, class B class A ya ulaşamaz .

#### A.java

```
package chapter1.package1;
```

```
class A {
```

```
}
```

#### B.java

```
package chapter1.package2;
```

```
public class B {
```

```
    public static void main(String[] args) {
```

```
        // A a= new A(); compile hatalı
```

```
}
```

```
}
```

## Public Access

Bir sinif public olarak tanimlanmissa , tum paketlerdeki siniflardan public tanimlanan bu sinifa erisim saglanabilir. Yukarıdaki ornekte A sinifini public olarak degistirirsek, farkli paketlerde olmalarina ragmen B sinifinden A sinifina ulasilabildigimizi gorebiliriz.

## Non-Acces Class Modifiers

**final , abstract ve strictfp** Javada anahtar kelimelerdir. Bu anahtar kelimeler siniflar icin kullanilabilir. Bu anahtar kelimelerden **strictfp** ile **final** veya **strictfp** ile **abstract** kullanilabilir hic bir sekilde **final** ve **abstract** yan yana gelemez. Bunların dogasi birbirine tamamen zittir.

### strictfp

siniflar ve metotlar icin kullanilabilir fakat asla degiskenler icin kullanilamazlar. Bir sinifi veya metodu strictfp olarak tanimlamak float sayilari IEEE 754 standartina uygun olmasini saglamak demektir.

### final

**final** anahtar kelimesinin siniflar ile kullanimi , final tanimlanan sinifin kalitilamayacagi yani bu sinifin bir alt sinifi olamayacagi anlamina gelmektedir. Ilerleyen konularda inheritance konularini isleyecegiz  
Bir sinifi final tanimlamak ve kalitilmamasina engel olmak genel olarak Object Oriented ruhuna ters gibi gozukmektedir. Java core kutuphanesindeki siniflarin cogu final olarak tanimlanmistir. Bu guvenlik icindir.  
Bir sinifi final olarak tanimlamak istiyorsak , bu siniftaki metotların override edilmesine gerek duymayacagimizi garanti etmemiz gerekmektedir.

Son olarak bir sinifin final tanimlanması performansa + etki saglamaktadir.

```
final class B {
```

```
}
```

```
public class A extends B {
```

```
}
```

Ilerleyen derslerde final anahtar kelimesi üzerinde duracağınız.

### abstract

abstract sınıfın amacı final sınıfların aksine, kalitilmaktır. Ilerleyen derslerde abstract classları derinlemesine inceleyeceğiz.

```
abstract public class A {
```

```

private int var1;
private String var2;
public abstract void method1();
public abstract void method2();
public void method3() {
    System.out.println("Method3");
}
}

```

abstract sinifimizi inceleyecek olursak *public abstract* veya *abstract public* tanimlasi yazilabilir.

- Degisken tanimlanabilir.
- *abstract* metot tanimlanabilir, abstract metodlarin govdesi yoktur yani { } yoktur. ve ; ile biterler.
- *abstract* class ta normal metot da tanimlayabiliriz.
- *abstract* class lardan nesne olusturulamaz. A = new A(); compile hatasi verir.
- Bir sinifta sadece 1 tane abstract metot bile tanimlansa tum sinif abstract olmak zorundadir.
- Bunun tersine hic abstract metot olmasa da bir sinif abstract olarak tanimlanabilir.
- *final* ve *abstract* hic bir sekilde yan yana gelemezler.

Illerleyen derslerde *abstract*, ve *final* konusuna tekrar donecegiz ve cok daha detayli olarak isleyecegiz.



## Pure Java – 03 Interface Declaration

[Levent Erguder](#) 05 April 2013 [Java SE](#)

Merhaba Arkadaslar,

Bu dersimiz, Java ‘da *Interface* (arabirim) tanimlanması ile ilgili olacaktır.

Bir interface (arabirim) tanimladigimizda , metodlar govdesiz olarak tanimlanır. yani { } icermezler. Interface bir sablon, kontrattır yani bir interface i uygulayan sinif bu metodları mutlaka uygulamak, (*override*) zorundadır.  
*abstract* bir sinif , bir interface i uygulayabilir ve metodlarını uygulamak zorunda degildir, abstract olmayan bir sinif mutlaka metodları uygulamak ,(*override*), zorundadır.

Interface (arabirim) , herhangi bir sinif tarafından uygulanabilir (*implements*) . Bu sayede kalitim hiyerarsisinde birbiriyle tamamen iliskisi olmayan siniflar dahi ayni arabirimini uygulayabilir. Bu Interface yapisinin en onemli ozelligidir.

*interface* kelimesi Javada bir anahtar kelimedir.

Basit bir interface ornegi,

```
interface NewInterface {  
    void method1();  
    void method2();  
}
```

Yukarıdaki şekilde tanımladığımız interface ve metodlar compiler tarafından şu şekilde algılanır.

```
interface NewInterface {  
    public abstract void method1();  
    public abstract void method2();  
}
```

Interface 'ler %100 abstract tir. Dikkat ederseniz tanımlanan metodlar abstracttır. Interfaceler de normal metodlar tanımlanamaz.

Interface'i uygulayan bir sınıfı bakalım simdi de.

```
class A implements NewInterface{  
  
    public void method1() {  
        //kodlar  
    }  
  
    public void method2() {  
        //kodlar  
    }  
}
```

- Abstract class' larda abstract ve non-abstract metodlar tanımlanabilirken Interface 'lerde sadece abstract metodlar tanımlanabilir.
- Tüm Interface metodları belirtelim ya da belirtmeyelim *public abstract* tir. *protected* veya *private* olarak tanımlanamazlar.
- Interface 'lerde degişken tanımlanabilir ve bu degişkenler belirtilmese bile mutlaka *public static* ve *final* dir. Dolayısıyla degiskenden ziyade bir constant tir.

- Interface metotlari static , final , strictfp veya native olamazlar . final ve abstract asla yanyana gelemez.
- Bir interface bir ya da daha fazla interface i extend edebilir. Java da siniflar sadece 1 sinifi extend edebilirken bu durum interfaceler icin gecerli degildir.
- Bir interface diger bir interface i implement edemez.
- interface taniminda yazalim ya da yazmayalim abstract anahtar kelimesi vardir yani ;  
*public interface NewInterface*  
*public abstract interface NewInterface* ayni anlamda gelmektedir.
- interface ler default veya public olarak tanimlabilir. private interface olamaz.

Su tanimlanan metotların tanımlanma şekli farklı olsa da , hepsi aynıdır. Cunku interface metotlarının hepsi abstract ve public tir ve tanımlarken abstract – public yer degistirebilir.

```
void method1();
public void method1();
abstract void method1();
public abstract void method1();
```

interface ve çoklu kalitim a bir örnek ,

```
interface A {
}

interface B {
}

interface C extends A , B {
}
```

interface değişkenleri biz belirtelim ya da belirtmeyelim public static final dir demistik , yani sabittir. Interface de tanımlanan değişkenler implements eden sınıfta değiştiremezler.

Bir interface içerisinde tanımlanabilecek değişken tanımlarından hepsi birbirinin aynısıdır.

```
public int x=1;
int x=1;
static int x=1;
```

```
final int x=1;  
public static int x=1;  
public final int x=1;  
public static final int x=1;
```

## Java 8 Interface New Features

Java 8 oncesinde interface'lerde sadece abstract (govdesiz) metotlar tanimlanabiliyordu.  
Java 8 ile birlikte artik interface 'ler default ve static metotlara sahip olabilmektedir!

### **interface default methods**

default anahtar kelimesini ekleyerek artik govdeli metotlari Interface'erde tanimlayabiliriz !

```
public interface Java8Interface {  
  
    default void defaultMethod() {  
        System.out.println("default method java 8 new feature!");  
    }  
  
    default void defaultMethod2() {  
        System.out.println("default method2 java 8 new feature!");  
    }  
}
```

default metotlar , govdesiz metotlar gibi override edilmek zorunda degildir , dilersek override edebiliriz.

default metotlar icin access level govdesiz metotlar gibi public ' tir!

```
public class Java8InterfaceImpl implements Java8Interface {  
  
}
```

Yukarıdaki kodumuz herhangi bir derleme hatasi vermeyecektir, cunku default metotlari override etmek zorunda degiliz!

Ufak bir test sinifi yazalim ;

```
public class Java8InterfaceTest {  
  
    public static void main(String[] args) {  
        Java8InterfaceImpl java8 = new Java8InterfaceImpl();  
        java8.defaultMethod();  
        java8.defaultMethod2();  
    }  
}
```

Ornegimizi calistirdigimizda Console'da

```
default method java 8 new feature!  
default method2 java 8 new feature!
```

Simdi de default metodu override edelim ;

```
public class Java8InterfaceImpl implements Java8Interface {  
  
    @Override  
    public void defaultMethod() {  
        System.out.println("default method override !!");  
    }  
}
```

Test sinifimizi tekrar calistirdigimizda Console'da

```
default method override !!  
default method2 java 8 new feature!
```

Java da bir class sadece bir class 'i kalitabilir (extends) fakat bir class birden fazla interface'i uygulayabilir (implements)

Multiple extends mekanizmasında Diamond Problemi ortaya cikabilir , bu nedenle Javada multiple extends mekanizmasi class'lar icin yoktur. Diamond Problemi default metotlar icin de ortaya cikabilir ;

2 tane interface tanimlayalim ve ayni isimde metot tanimlayalim. Bu durudmda bu 2 interface i uygulayan (implements) class hangi super metodu kullanacak ? Eger bu default metot override edilmezse bu durumda derleme hatasi olacaktir.

```
public interface DiamondInterface1 {  
  
    public default void diamondProblem(){  
        System.out.println("Diamond");  
    }  
}  
  
public interface DiamondInterface2 {  
  
    public default void diamondProblem() {  
        System.out.println("Diamond");  
    }  
}
```

Dikkat edecek olursak 2 tane Interface tanimladik ve ayni isimle default metot tanimladik.

```
public class DiamondClass implements DiamondInterface1, DiamondInterface2 {  
  
    // eger override edilmezse derleme hatasi verecektir.  
  
    // Duplicate default methods named diamondProblem  
  
    // public void diamondProblem(){  
    //     System.out.println("Diamond override");  
    // }  
}
```

Kafamizda kalmasi icin bir baska ornek ile aciklamaya calisalim ;  
3 tane interface olsun.

- Developer
- JavaDeveloper
- CSharpDeveloper

```
public interface Developer {  
  
    public void code();  
}  
  
public interface JavaDeveloper extends Developer {
```

```

@Override
public default void code() {
    System.out.println("Code Java");
}

}

public interface CSharpDeveloper extends Developer{
    @Override
    public default void code() {
        System.out.println("Code C#");
    }
}

```

Developer interface’inde code adında abstract bir metot tanımladık ve JavaDeveloper , CSharpDeveloper interface’leri bu metodu override etti.

JuniorDeveloper adında bir class tanımlarsak ve hem JavaDeveloper hem de CSharpDeveloper interface’lerini uygularsa (implements) bu durumda derleme hatalı verecektir.

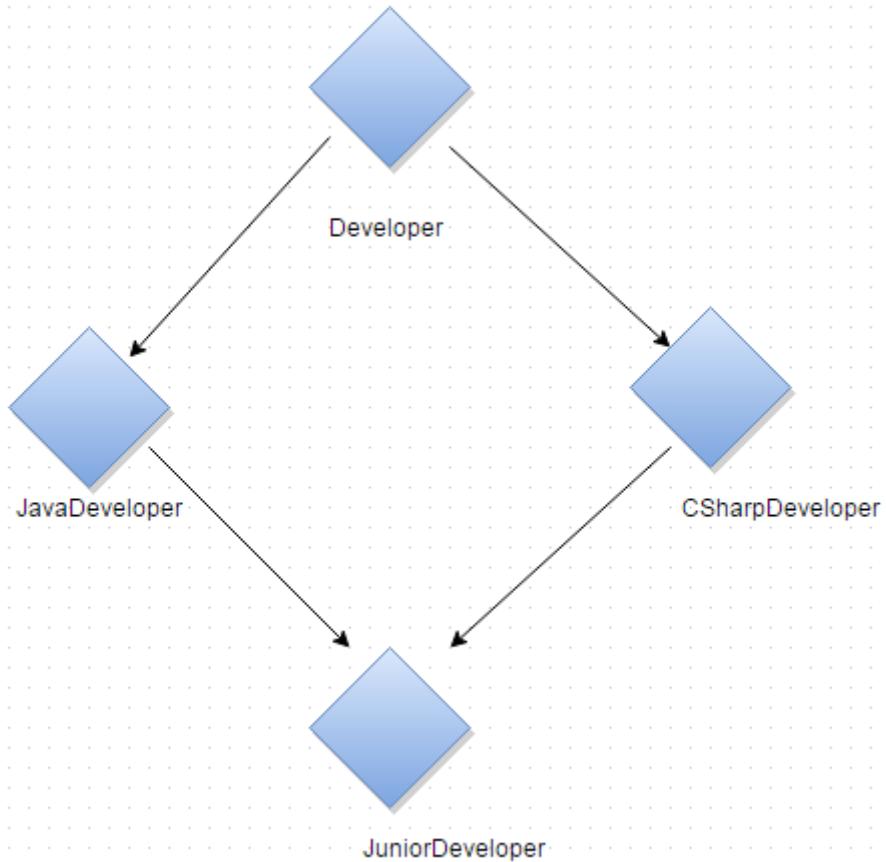
Bu durumda code metodu JuniorDeveloper sınıfında override edilmelidir.

```

public class JuniorDeveloper implements JavaDeveloper, CSharpDeveloper {
    @Override
    public void code() {
        System.out.println("Override Code");
    }
    // eger code metodu override edilmezse derleme hatalı verir.
}

```

Bu durumun aklimızda kalması için su diamond(karo) şeklini hatırlayabiliriz.



### **interface static methods**

Bir diger ozellik olarak Interface’te static metotlar tanimlayabiliriz. Interface’te tanimladigimiz default ve abstract metotlar gibi static metotlar da public metotlardir.

static metotlar override edilmez. Interface’te tanimlanan static metotların bir kisiti daha bulunmaktadir ; sadece InterfaceName.staticMethodName olarak bu static metodu cagirabiliriz. Diger durumlarda derleme hatasi verecektir.

```

public interface Java8InterfaceStaticMethod {

    public static void staticMethod() {
        System.out.println("This is Java8 interface static method.");
    }
}

public class Java8InterfaceStaticMethodImpl implements Java8InterfaceStaticMethod{
}
  
```

Ufak bir test sınıfı yazalım;

```
public class Java8InterfaceStaticMethodTest {  
  
    public static void main(String[] args) {  
  
        Java8InterfaceStaticMethod java8 = new Java8InterfaceStaticMethodImpl();  
  
        Java8InterfaceStaticMethod.staticMethod();  
  
        // derleme hatalı verir!  
        // Java8InterfaceStaticMethodImpl.staticMethod();  
        // derleme hatalı verir!  
        // java8.staticMethod();  
    }  
}
```

Dikkat edecek olursak sadece Java8InterfaceStaticMethod.staticMethod(); olarak bu static metoda ulaşım sağlayabildik , diğer durumlarda derleme hatalı verecektir.

Utility metodları static tanımlanır , dolayısıyla interface’lerde utility metodlarına uygun olacak şekilde static metodları kullanabiliriz.

## Pure Java – 04 Declare Class Members

Levent Erguder 07 April 2013 Java SE

Merhaba Arkadaşlar

Bu yazımızda Java’da Class Members (methods , instance (non-local) variable) tanımlanması hakkında bilgi vereceğim.

### Access Modifiers

Methodlar ve instance variable’lara 3 access modifier da uygulanır.

public

private

protected

ve tabii ki bunlardan biri kullanılmazsa varsayılan olarak package level (default) uygulanır.

Bunlardan default ve protected birbirine neredeyse aynıdır. İlerleyen kısımlarda farkından bahsedeceğim.

public üyeleri



Halka acik

*public* metotlara ve değişkenlere tüm sınıflardan erişim mümkündür. Tabiki ulaşacağımız metodun veya değişkenin sınıfına ait bir referans değişkenimiz aracılığıyla ulaşabiliriz. ikinci olarak sınıflar farklı paketlerde ise mutlaka ulaşılan sınıfı import etmek gereklidir. Dolayısıyla Java aynı paket altında aynı isimde iki sınıf tanımlanmasına izin vermez.

### A.java

```
package purejava;

import purejava2.B;

public class A {

    public static void main(String[] args) {

        B b = new B();
        System.out.println("Class A - " + b.var2 + " " + b.var1);
        b.method1();

    }
}
```

### B.java

```
package purejava2;

public class B {
    public int var1=1995;
    public String var2="Java";

    public void method1() {
        System.out.println(var2+" "+var1);
    }
}
```

A ve B siniflari farkli paketlerde olmasina ragmen, A sinifi B sinifinin public sinif degiskenlerine ve metotlarina ulasabilir ulasabilir.

### private uyeler

*private* metotlara ve sinif degiskenleri sadece ayni sinifta kullanabiliriz. Disaridan herhangi bir ulasim soz konusu degildir. Bir sinif diger bir sinifin ozelliklerini kalitim ile alsa private degiskenlere ve metotlara ulasim soz konusu olamaz. Yan *private* degiskenler ve metotlar kalitilamazlar.

B sinifindaki degiskenleri ve metodu *private* yapalim,

### B.java

```
package purejava2;

public class B {

    private int var1=1995;
    private String var2="Java";

    private void method1() {
        System.out.println(var2+" "+var1);
    }
}
```

A sinifi ile B sinifi ayni pakette olmalarina ve A sinifi B sinifini *extends* etmesine ragmen yine de *private* uye'lere ulasim saglanamayacaktir bununla birlikte *private* metotu *override* etmemize izin vermeyecektir.

### A.java

```
package purejava;

import purejava2.B;

public class A extends B{

    public static void main(String[] args) {

        B b = new B();
        System.out.println("Class A - " + b.var2 + " " + b.var1);
        b.method1();
    }
}
```

```

        private void method1() {
            System.out.println(var2+" "+var1);
        }

    }

}

```

### protected ve default uyeler

*protected* ve *default* erisim seviyeleri genellikle aynidir. Bir *default* uyeye sadece ayni paket içerisindeki bir sınıfın ulaşılabilir aynı şekilde bir *protected* uyeye de. Peki fark nerede ortaya çıkmaktadır , kalitimın söz konusu olduğu yerlerde .

Ornek sınıflarımız üzerinde inceleyelim...

B sınıfımızda *protected* ve *default* uyeler var.

#### B.java

```

package purejava2;

public class B {
    protected int var1 = 1995;
    protected String var2 = "Java";
    int var3=15;

    protected void method1() {
    }
    void method2() {
    }
}

```

#### A.java

```

package purejava;

import purejava2.B;

```

```

public class A extends B {

    public void test() {
        B b = new B();
        System.out.println(var1 + " " + var2);
        method1();
        // System.out.println(var3);
        // method2(); default metoda ulaslamaz - compile error

        // System.out.println(b.var1+ " " +b.var2); compile error
        // b.method1(); referans degisken ile ulaslamaz
        // b.method2(); referans degisken ile default metoda ulaslamaz

    }
}

class C extends A {

    public void getProtectedC() {
        method1();
        var1 = 20;
        // var3=10; // compile error
        // method2(); // compile error
    }
}

class D {

    public void getProtectedD() {
        // method1();
        // method2();
        // var1=20;
        // var3 = 10;
    }
}

```

}

A sinifindaki kodlari inceleyim..

- A ve B siniflari farkli paketlerdedir ve A sinifi B sinifini kalitmaktadir.
- A sinifinda kalitim aracılıgiyla B sinifinda protected olarak tanımlanan var1 ve var2 degiskenlerine erişim sağlanabilir.
- Aynı şekilde *protected* metoda kalitim aracılıgiyla erişim sağlanabilir.
- Buna rağmen *default* degiskenlere ve *default* metodlara kalitim aracılıgiyla erişim sağlanamaz.
- Referans degisken aracılıgiyla ( b , referans degiskeni ) *protected* degiskenlere ve metodlara erişim sağlanamaz.
- Referans degisken aracılıgiyla *default* degiskenlere ve metodlara erişim sağlanamaz.
- C sinifi A sinifini *extends* ettiği için , B sinifında bulunan protected degerlere kalitim vasitasiyla ulaşabilir. ( C sinifi A sinifini extends ettiği surece, farklı paketten dahi bu degerlere ulaşabilir ) . Buna rağmen *default* degiskenlere ve metodlara erişim sağlanamaz.
- D sinifi A sinifiyla aynı pakette olmasına rağmen , A sinifinin B sinifinden kalitim aracılıgiyla kalittiği , eristiği degisken ve metodlara erişim sağlayamaz. Normal şartlarda *protected* degiskenlerlere aynı pakette ulaşım sağlanabilirken burada neden ulaşım sağlanamadı ? Cunku A sinifi kalitim ile aldığı bu *protected* degisken ve metodlar, A sinifi disinda( D sinifi için) *private* duruma gelir.



Durumu özetleyen tablomuz

Visibility	Public	Protected	Default	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, through inheritance	No	No
From any non-subclass class outside the package	Yes	No	No	No

## Pure Java – 05 Declare Class Members – 02

[Levent Erguder](#) 16 April 2013 [Java SE](#)

Merhaba Arkadaslar,

Bu dersimizde Class Member (instance variable, methods) icin kullanabilecek *nonaccess member modifier* konusunu isleyecegiz. Bir onceki derste *Access Modifier* konusunu islemistik.

### final Methods

*final* anahtar kelimesi metotlar icin kullanildiginda , alt sinif tarafindan bu metodun ezilmesini (*override*) engeller.

**A.java**

```
package purejava5;

public class A {

    void method1() {

    }

    final void method2() {

    }
}
```

**B.java**

```
package purejava5;

public class B extends A{

    @Override
    void method1(){

    }

    /*     void method2(){
    }
```

```
 }*/
```

```
}
```

B sınıfımız A sınıfını *extends* etmektedir, *final* olarak tanımlanmayan method1 ezilebilirken, *final* olarak tanımlanan method2 ezilememektedir.

#### abstract Methods

*abstract* metodlar, *abstract* anahtar kelimesi ile tanımlanan, suslu parantezin olmadığı { } gariban



metotlardır sadece tanımlanmıştır içerisinde herhangi bir fonksiyonellik yoktur.

*abstract* metodlar noktalı virgül ile son bulurlar,

```
public abstract void method1();
```

Bir önceki derste belirttiğimiz gibi, bir sınıf sadece tek bir adet *abstract* metoda sahip olsa bile mutlaka bu sınıf *abstract* olarak tanımlanmalıdır.

```
public class C {  
    public abstract void method1();  
}
```

Yukarıdaki kod parçası derleme hatalı verecektir, çünkü sınıfımız *abstract* bir metoda sahiptir ve bu sınıf *abstract* olarak tanımlanmamıştır.

*abstract* sınıfı *extends* eden bir sınıf, *abstract* metodları ezmek (*override*) zorundadır. Eğer *extends* eden sınıf *abstract* ise ezmek zorunda değildir.

#### A.java

```
package purejava5;  
  
abstract public class A {  
  
    final void method1() {  
  
    }  
  
    void method2() {
```

```
        System.out.println("Hello Java");
    }

    abstract void method3();

    abstract int method4();

}

class B extends A {

    @Override
    void method3() {

    }

    @Override
    int method4() {
        return 0;
    }

}

abstract class C extends A {

    @Override
    void method3(){
        System.out.println("Override");
    }

    abstract void method5();

}

class D extends C {

    @Override
}
```

```

void method5() {
    // TODO Auto-generated method stub
}

@Override
int method4() {
    // TODO Auto-generated method stub
    return 0;
}

}

```

A sinifi *abstract* olarak tanimlanmistir ve 2 adet *abstract* metodu vardir.

B sinifi, A sinifini *extends* ettigi icin ve *abstract* olarak tanimlanmadigi icin bu 2 *abstract* metodu *override* etmek zorundadir.

C sinifi , *abstract* oldugu icin A sinifini *extends* etmesine ragmen A sinifinda tanimlanan *abstract* metotlari *override* etmek zorunda degildir, fakat dilersek *override* edebiliriz.

C sinifininda bir adet *abstract* metot tanimlanmistir.

D sinifi C sinifini *extends* ettiginde, C sinifinin *abstract* metodunu *override* etmek zorundadir bununla birlikte C sinifinin A sinifinden *kalittigi* ve *override* etmedigi method4 ,D sinifinda *override* edilmek zorundadir.

Genel kural sudur, *abstract* olarak tanimlanan bir metot , *abstract* olarak tanimlanmayan ilk sinifta mutlaka *override* edilmelidir.

#### **synchronized Methods**

*synchronized* olarak tanimlanan bir metoda ayni anda sadece bir tek adet *thread* erisebilir. Ilerleyen derslerde *synchronized* konusunu isleyecegim , burada konumuz belirtecler oldugu icin detaya girmeyecegim. *synchronized* anahtar kelimesi sadece metodlar icin kullanilabilir, degiskenler veya siniflar icin tanimlanamazlar.

#### **native Methods**

*native* , Java ‘ da anahtar kelimedir ve sadece metodlar icin kullanilabilir. Degiskenler ve siniflar icin kullanilamazlar. Javada , C kodlarini cagirip kullanabiliriz.

*native* metodlar *abstract* metodlar gibi govdesiz olarak tanimlanir.

#### **strictfp Methods**

*strictfp* belirtecinin siniflar icin kullanmistir, bu belirtec metodlar icin de kullanilabilir. Bir sinifi *strictfp* olarak tanimlayabildigimiz gibi metodlari da bireysel olarak tanimlayabiliriz. Floating islemlerin IEEE 754 standardina uygun olmasini saglar.

Burada Java da kullanılan *final*, *abstract*, *native* , *synchronized*, *native*, *strictfp* anahtar kelimelerini metodlar icin kullanimi inceledik. Burad incelemedigimiz sadece *static* metodlar kaldi , static metodlari daha detaylica isleyecegiz.

## Pure Java – 06 Declare Class Members – 03 Variable

Levent Erguder 05 May 2013 Java SE

Merhaba Arkadaşlar,

Bu dersimizde Java'da , variable (degisken) konusunu isleyecegiz.

Javada 2 tip degisken vardir.

*primitive*

*reference variable (referans degiskeni)*

**Primitive Degiskenler (primitive variables)**

primitive degiskenler su 8 tipden biri olabilir ;

*char, boolean, byte , short , int , long , double , float*

primitive bir degisken tanimlandiginda turu degistirilemez yani int a ,bir alt satirda short a olarak degismez. Java



ciddi bir dildir bu tarz kontrollere onem verir

primitive degiskenler su durumlarda tanimlanabilir ,

- bir class variable (yani static degisken)
- instance variable (objeye ait degisken)
- method parametresi

Burada bir dip not vermek istiyorum bir cok insan bu kavrami karistirmakta veya ayni sanmaktadır.

metot tanimlamasinda kullanılan degiskenler parametredir, methodu cagirirken gönderilen degiskenler ise argumandır .

Bu 8 primitive degiskenden 6 tanesi ( char ve boolean haric ) signed (isaretli) yani negatif ve pozitif degerler alabilir.

**Referans Degiskenler (reference variables)**

- Bir referans degisken bir objeyi isaret eder.
- Bir referans degisken bir kez tanimlanır ve turu degismez.
- Bir referans degisken tanimlandığı turde veya alt turde (subtype) objeyi isaret edebilir.

Illerleyen derslerde detaylıca bahsedeceğim burada sadece degisken tanimlama ile ilgileniyoruz.

**Primitive Degiskenlerin Tanımlanması**

primitive degiskenler class variable(sınıf degiskeni/static), instance variable, method parametresi, local degisken olarak tanımlanabilir.

Asagidaki tabloda 6 tane numerik primitive degiskenen bilgileri yer almaktadir. float ve double degiskenlerin



sinirlari diger 4 degisken gibi belirli degildir. ( it is complicated )

Type	Bits	Bytes	Minimum Range	Maximum Range
byte	8	1	-2 <sup>7</sup>	2 <sup>7</sup> -1
short	16	2	-2 <sup>15</sup>	2 <sup>15</sup> -1
int	32	4	-2 <sup>31</sup>	2 <sup>31</sup> -1
long	64	8	-2 <sup>63</sup>	2 <sup>63</sup> -1
float	32	4	n/a	n/a
double	64	8	n/a	n/a

boolean degisken , true veya false olabilir. Kac bit olacagi JVM e baglidir.

char degisken ise 16-bittir. char degisken isretsiz oldugu icin yani negatif deger olmayacagi icin 0 dan 65535 ( $2^{16}$ ) ya kadar siniri vardir.

### Referans Degiskenlerin Tanimlanmasi

referans degiskenler ; sinif degiskeni, instance variable, method parametre degiskeni ve local degisken olarak tanimlanabilir.

#### Instance Variable

instance variable , objenin kendine ait olan degiskenidir. instance degiskenler metot icerisinde tanimlanamaz.

- 4 erisim seviyesini kullanabilir. ( public, protected, private)
- final , transient olabilir.
- abstract, synchronized, strictfp, native, static olarak tanimlanamaz.
- static tanimlanan degiskenler instance variable degil class variable olur.

```
class A {  
    private String insVar1;  
    protected int insVar2;  
    public boolean insVar3;  
}
```

### Local (Automatic/Stack/Method) Degiskenler

- Local degiskenler bir metotla birlikte tanimlanan degiskenlerdir, yani instance variable objeye aitken local degiskenler metotlara aittir.
- Local degiskenlerin yasam alani ilgili metotla sinirlidir. Metot tamamlanincaya dek kendi



ucurulur

- Local degiskenler her zaman stackte tutulur, heap'te tutulmaz.
- Degiskenimiz referans degiskeni de olsa , stackte tutulur fakat referans oldugu obje heapedir. Local Obj kavramindan kasit local tanimli referans degiskenidir. Objelerin hepsi heapedir.
- Local degiskenler private , public,protected, transient, abstract, volatile,static gibi anahtar kelimeleri alamaz, sadece final anahtar kelimesini alabilirler.
- Genel olarak bir local degiskene tanimlandigi satirda deger verilir. Bu bir zorunluluk degildir fakat bir local degiskeni kullanmadan once mutlaka deger atanmalidir.

```
public class LocalVarTest {
    public static void main(String[] args) {
        int a;
        int b;
        String s;
        System.out.println(a);
    }
}
```

Yukarıdaki ornekte a,b,s local degiskenlerini tanimladik ne zaman bir degiskeni kullanmamız gerekti (System.out.println(a);) bize bu satırda derleme hatası verecektir.

*The local variable a may not have been initialized*

- Bir local degisken tanimlandigi metodun disindan kullanilamaz.

```
public class LocalVarTest {
    public void method1(){
        int var1;
    }
    public void method2(){
        var1=10;
    }
}
```

var1 cannot be resolved to a variable.

- Local degiskenenin adıyla instance degiskenen adı aynı olabilir. Buna Shadowing denir.

```
public class ShadowingTest {
    int shadow = 10;

    public void method1() {
        int shadow = 15;
        System.out.println("Local Variable:" + shadow);
    }

    public void method2() {
        System.out.println("Instance Variable:" + shadow);
    }

    public static void main(String[] args) {
        new ShadowingTest().method1();
        new ShadowingTest().method2();
    }
}
```

shadow instance degiskenimize 10 degerini atadik method1 de shadow ismindeki local degiskenimize 15 degerini atadik. Su ciktigisi aliriz.

*Local Variable:15*

*Instance Variable:10*

#### Dizi Tanimlanmasi( Array Declaration)

Burada dizilerin tanimlanmasini anlatacagim , ilerleyen derslerde dizilere detaylica bakacagiz



insallah

- Bir dizi (array) , ayni turde degiskenleri tutan bir yapidir.
- Bir dizi primitive veya obje referansi tutabilir (refarans degisken) fakat dizinin kendisi mutlaka ama mutlaka objedir dolayisiyla heapte tutulur. Yani primitive array diye bir sey soz konusu degildir.
- int [] myArray ve int myArray[] seklinde tanimlanabilir fakat koseli parantezin degisken isminden once gelmesi onerilir.Ilk tanimlama uygundur.

```
public class ArrayTest {

    private int[] myArray;
    protected int myArray2[];
}
```

```

String[] myArray3;
String[][] myArray4;
private boolean[] myArray5[];
long[] myarray6 ;

public void method1() {
    final int [] myArray7;
    char [] myArray8;
}

}

```

### final Degiskenler

Bir degiskeni final olarak tanimlamak , bu degiskene bir kez deger atandiktan sonra deger atanmasina engel olacaktir. Bir final primitive degiskene 100 degerini atadiysak 100 olarak kalacaktir. Bir final referans degiskenin de referans ettigi objeyi degistiremeyiz. final obje diye bir sey yoktur, final referans degiskeni olabilir.

final anahtar kelimesinden daha onceki derslerde de bahsetmistim , simdi 3 maddede toparlayalim ;

- final degiskeni siniflar icin kullanildiginda bu sinifin kalitilamayacagi anlamina gelmektedir.

```

final class A {

}

class B extends A{

}

```

derleme hatasi aliriz.

- final anahtar kelimesi metotlar icin kullanildiginda o metodun override(ezmek) edilemeyecegi anlamina gelmektedir.

```

class A {

    final void method1(){

    }
}

class B extends A{

```

```
void method10{  
}  
}
```

- final anahtar keliesi degiskenler icin kullanildiginda degiskenen degerinin degistirilemeyecegi anlamina gelmektedir.Derleme zamani hatasi aliriz.

```
public void method10{  
    final int var1=100;  
    var1=200;  
}
```

final olmayan instance degiskenlere degisken atamak zorunlulugumuz yoktur fakat final olarak tanimlanisra mutlaka deger atamamiz gerekmektedir.

```
class A {  
    final int var1;  
}
```

### transient Degiskenler

Bir degiskeni transient (gecici) olarak belirlersek JVM (Java Virtual Machine) serilasyon islemi sirasinda bu degiskeni atlayacaktır. Serilestirme konusuna cooook sonraki derslerde gelicem insallah omrum



yeterse

transient anahtar kelimesi sadece instance degiskenler icin kullanilabilir.

### volatile Degiskenler

volatile anahtar kelimesi sadece instance degiskenler icin kullanilabilir. Threadlerle calisirken senkronizasyon icin kullanilir , bunun yerine synchronized metotlari kullaniriz. Threadler konusuna serilestirme konusundan sonra gelecegim insallah.

### static Degiskenler ve Metotlar

- static degiskenler ve static metotlar sinifa aittir, herhangi bir objeye ait degildir.
- Butun static degisken ve metotlar yeni bir obje olusturmadan once hazir ve nazirdir.



- Bir static uyenin bir kopyasi vardir. Yani ortak maldir
- method, instance degisken,initialization blocks lar static olarak tanimlanabilir

- constructor(yapilandiric) ,interfaces, local degiskenler,siniflar ( nested olmadigi surece) , inner class method ve instance degiskenleri static olarak tanimlanamazlar.

```
package purejava6;

public class StaticTest {
    static int var1 = 10;

    static {
        var1++;
    }

    public static void main(String[] args) {
        System.out.println(var1++);
        method1();
    }

    public static void method1(){
        System.out.println(++var1);
    }
}
```



Bu ornek ile yazima son veriyorum

Ornegi anlamayi size bırakıyorum.

## Pure Java – 07 Declare Enums

[Levent Erguder](#) 13 May 2013 [Java SE](#)

Merhaba Arkadaslar,



% 100 Pure

Java derslerimizde Interface, Class ,Class Members tanimlanmasini gorduk son olarak



Enum tanimlanması ile bu tanimlama konusunu bitirmis olacacagiz insallah

Enum yapisini anlayabilmek icin su ornegi vermek istiyorum, bir kahve dukkaninda short, tall, grande ,venti boyutlarinda secenekler mevcuttur yani bizim ihtiyacimiz olan degisen sadece bu degerleri almalidir. Boyle bir yapiyi Java bize enum aracigiliyla saglayabilir.

```
enum CoffeeSize { SHORT , TALL , GRANDE ,VENTI };
```

enum Javada bir anahtar kelimedir. Burada verdigim SHORT, TALL gibi isimler buyuk harf olmak zorunda degil fakat bir Code Convention dir.

Enumlar kendilerine ozel sinif yapısında tanimlanabildi gibi bir sinifin uyeside olabilirler. Fakat bir metot içerisinde tanımlanamazlar.

Onçelikle bir sinif disarısında tanımlayalim. enumlar sinif uyesi olarak tanımlanmadığında, private veya protected olarak tanımlanamazlar.

#### JavaCoffeeTest.java

```
enum CoffeeSize {  
    SHORT, TALL, GRANDE, VENTI  
};  
  
class JavaCoffee {  
    CoffeeSize size;  
}  
  
public class JavaCoffeeTest {  
    public static void main(String[] args) {  
        JavaCoffee coffee = new JavaCoffee();  
        coffee.size = CoffeeSize.VENTI;  
        System.out.println(coffee.size);  
    }  
}
```

Ornegimizi inceleyecek olursak bir enum tanımladık, JavaCofee isimli sınıfımızın bu enum turundaki bir üyesi(member) var (size).

main metodumuz içerisinde bir JavaCoffee nesnesi oluşturuyoruz ve size degisenine VENTI degerini atıyoruz.

Ikinci ornegimizde ise enum yapisini bir sinif uyesi olarak tanımlayalim.Burada sınıf uyesi olarak tanımlanıkları için private ve protected tanımlanabilirler.

### JavaCoffeeTest2.java

```
class JavaCoffee {  
  
    protected enum CoffeeSize {  
        SHORT, TALL, GRANDE, VENTI  
    };  
  
    CoffeeSize size;  
  
}  
  
public class JavaCoffeeTest2 {  
    public static void main(String[] args) {  
        JavaCoffee coffee = new JavaCoffee();  
        coffee.size = JavaCoffee.CoffeeSize.SHORT;  
        System.out.println(coffee.size);  
    }  
}
```

Ek bir not olarak enum tanimlamasinin sonuna koyulan noktali virgul (bu iki ornek icin) zorunlu degildir. Ilk basta belirtigim gibi enumlari metodlar içerisinde tanimlayamayiz !

enum lar String degildir, integer degildir. enum i bir tur ozel sinif olarak dusunebiliriz. Listelenen her bir uye (short,tall, grande, venti ) bir CoffeeSize instancedir(ornek).

Enum yapisi su ornek yapiya benzemektedir.

### JavaCoffeeSize.java

```
public class JavaCoffeeSize {  
  
    public static final JavaCoffeeSize SHORT = new JavaCoffeeSize("SHORT", 0);  
    public static final JavaCoffeeSize TALL = new JavaCoffeeSize("TALL", 1);  
    public static final JavaCoffeeSize GRANDE = new JavaCoffeeSize("GRANDE", 2);  
    public static final JavaCoffeeSize VENTI = new JavaCoffeeSize("VENTI", 3);  
  
    public JavaCoffeeSize(String enumName, int index) {  
  
    }  
}
```

```

public static void main(String[] args) {
    System.out.println(JavaCoffeeSize.GRANDE);
}
}

```

### Enumlar ve Yapilandirici (Constructor) , Metot, Degisken

Enumlar ozel bir sinif yapisi gibidir. Bu nedenle yapilandirici, metot degisken tanimlamamiz mumkundur.

Ornegimizi inceleyelim.

**JavaCoffee.java**

```

enum CoffeeSize {
    SHORT(2), TALL(3), GRANDE(4), VENTI(5);

    CoffeeSize(int $){
        this.$ = $;
    }

    private int $;
    public int get$() {
        return $;
    }
};

public class JavaCoffee {

    CoffeeSize size;

    public static void main(String[] args) {
        JavaCoffee coffee1 = new JavaCoffee();
        coffee1.size = CoffeeSize.SHORT;

        JavaCoffee coffee2 = new JavaCoffee();
        coffee2.size = CoffeeSize.TALL;

        System.out.println(coffee1.size.get$());
        System.out.println(coffee2.size.get$());

        for(CoffeeSize cs:CoffeeSize.values()){
            System.out.println(cs);
        }
    }
}

```

```

        System.out.println(cs + " " + cs.get$());
    }

}

}

```

Ornegimizde integer bir deger alan yapılandirici tanimladik ve enum degelerini tanimlarken degerlerini atadik. Dikkat ederseniz \$ isminde bir degisen tanimladim, ilk dersten hatirlayacagini giber bu gecerli bir karakter ve isimlendirmeydi. Bu private degisenimizin getter metodunu yazdik. Boylece hem yapılandirici hem degisen hem de metot kullanmis olduk.

JavaCoffee sinifimizda CoffeeSize uyemiz vardir. 2 adet obje olusturduk ve bu objelerimizin size degisenine (CoffeeSize turunde) SHORT ve TALL degerlerini atadik. Bunları ekrana yazdirdik.  
Son olarak for dongumuzde her enum yapısında olan static values () metodunu cagirdik.

```

2
3
SHORT 2
TALL 3
GRANDE 4
VENTI 5

```

Son ornegimizde ise , metot override islemi gerceklestirecegiz. Dusunelim ki SHORT, TALL , GRANDE icin kodumuz B fakat VENTI icin kodumuz A bunun icin soyle bir yapı kurabiliriz.  
Noktali virgulu kullanimi burada zorunludur ! Unutmayalim.

### [JavaCoffee.java](#)

```

enum CoffeeSize {
    SHORT, TALL, GRANDE, VENTI {

        public String getCode() {
            return "A";
        }

    }; // Burada ; zorunludur !

        public String getCode() {
            return "B";
        }
    };
}

```

```
public class JavaCoffee {  
  
    CoffeeSize size;  
  
    public static void main(String[] args) {  
  
        for (CoffeeSize cs : CoffeeSize.values()) {  
            System.out.println(cs + " " + " Code:" + cs.getCode());  
        }  
  
    }  
}
```

*SHORT Code:B*

*TALL Code:B*

*GRANDE Code:B*

*VENTI Code:A*

Son olarak bir kac detayi daha belirtmek istiyorum.

## Pure Java – 08 Declare Var-Args

Levent Erguder 18 May 2013 Java SE

Merhaba Arkadaslar,

Bu yazimda Java da var-arg yapisindan bahsedecegim, bu yapi pek de bilinen bir yapi degildir ama kullanisli olabilecegi durumlar olabilir.

Oncelikle arguman ve parametre kavramindan bahsetmek istiyorum.

arguman (argument) ; Bir metodu cagirirken parantezler arasında yazdigimiz ifadelerdir.

```
method1("levent",23);
```

Burada levent ve 23 ifadeleri birer argumandir.

parametre(parameter) ; Bir metodu tanimlarken , nasil cagirlacagini belirttigimiz tanimladigimiz ifadeler ise parametredir.

```
void method1(String name, int age) { }
```

Bu metot String ve int turunde 2 adet parametre beklemektedir.

Bir metot dusunelim, bu metodumuzu hem bir “int” argumanla hem 2 hem 3 ... “int” argumanla cagirmamız o sekilde kullanmamız gerekmekte. Boyle bir yapıyi kurmak icin Java da var-arg yapısından yararlanırız. Metot tanımlamasında bu yapı uc nokta ... ile kurulur.

### VarArgTest.java

```
public class VarArgTest {  
    public static void main(String[] args) {  
        VarArgTest vt =new VarArgTest();  
  
        vt.method1();  
        vt.method1(10);  
        vt.method1(5,20);  
        vt.method1(10,25,50);  
  
        vt.method2('a');  
        vt.method2('a',10);  
        vt.method2('a',5,20);  
        vt.method2('a',10,25,50);  
  
    }  
  
    void method1(int... x) {  
        System.out.println(x.length);  
    }  
}
```

```

void method2(char c,int... x) {
    System.out.println(x.length);
}

/*
void method3(int...x , char c) {
    // Var-arg parametresi en sonda olmalıdır
}*/


/*
void method4(int ...z,int... x) {
    // Sadece bir tane var-arg parametresi kullanılabilir.
}*/


/*
void method5(int x...) {
    // 3 nokta parametre isminden önce gelmelidir.
}*
}

```

Ornegimizi inceleyecek olursak ,method1 'i  $\geq 0$  arguman ile cagirdik.

- Sadece bir tek var-arg parametresi kullanılabilir.
- Var-arg yapısında olmayan diger parametrelerle birlikte kullanılabilir.
- Var-arg yapısı metotda tanimlanirken en sonda olmalıdır.
- Var-arg yapisini tanimlarken 3 nokta parametre isminden önce gelmelidir.

Yazimi burada noktaliyorum.

## Pure Java – 09 Java Tricks

[Levent Erguder](#) 23 May 2013 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda onceki 8 yazimin ozeti seklinde olacaktir. Akilda tutulmasi gereken trickleri maddeler halinde paylasacagim.

[Identifier\(Belirtec\)](#)

- Javada belirtecler harf, underscore (\_) ve \$ isareti ile baslayabilir. Rakam ile baslayamaz.
- Ilk karakterden sonra rakam da kullanilabilir.
- Isimlendirmede uzunluk sinirlaması yoktur.
- Java da camelCase yapısına uygun olarak isimlendirme kullanılır.

#### Deklarasyon Kuralları

- Bir kaynak dosya (source file) sadece bir tek public class içerebilir.
- public class ile kaynak dosyanın adı aynı olmak zorundadır.
- Bir sınıf sadece tek bir package ifadesine sahip olabilir, bir den fazla import kullanılabilir.
- Bir kaynak dosya bir den fazla public olmayan sınıf içerebilir.

#### Class Access Modifiers ( Sınıf Erisim Duzenleyicileri)

- Java da 3 adet access modifier vardır ; public , protected, private
- Java da 4 adet access level vardır. public, protected, private ve default
- Sınıflar public veya default access e sahip olabilir.
- Default access e sahip sınıfa aynı paket altındaki tüm sınıflardan erişim sağlanabilir.
- public sınıfa diğer paketlerde bulunan sınıflardan da ulaşım sağlanabilir.

#### Class NonAccess Modifier

- Sınıflar sunları da alabilir ; final , abstract , strictfp
- Bir sınıf final ve abstract olamaz. Bu iki anahtar kelime hic bir şekilde yan yana gelemez.
- final sınıf kalıtlamaz
- abstract sınıfın obje oluşturulamaz.
- abstract sınıf hem abstract hem normal (concrete) metoda sahip olabilir.

#### Interface (Arabirim)

- interface (arabirim) bir sınıfın ne yapacağını söyleyen fakat nasıl yapacağını söylemeyen bir kontrattır.
- interface , herhangi bir sınıf tarafından implements edilebilir. Burada kalitim ağacı yapısı onemsizdir, birbiriyile alakasız iki sınıf aynı interface’ı implement edebilir. Arabirimlerin gücü buradadır.
- Bir interface (arabirim) sadece abstract metodlara sahip olabilir.
- interface metodları varsayılan olarak public ve abstracttir .
- interface degişkenleri public, static ve finaldir. Yani degiskenden ziyyade sabittir (constant)
- Bir sınıf sadece bir tek sınıfı kalıtabilir (extends) fakat birden fazla interface’ı implement edebilir.
- interface(arabirim) birden fazla interface’ı kalıtabilir (extends)
- interface başka bir interface’ı implements edemez.
- abstract bir sınıf bir interface’ı implements edebilir fakat interface’ın metodlarını override etmek zorunda değildir.

#### Member Access Modifier

- Metotlar ve instance degişkenler “member” olarak bilinirler.
- Sınıf üyeleri public, protected, default ve private erişim seviyelerinde olabilirler.
- Eğer bir sınıf bir sınıfın erişemiyorsa, bu sınıfın üyelerine de erişemez.
- public üyelerde farklı bir paketten de ulaşılabilir.

- this anahtar kelimesi , hali hazirda calisan nesneyi temsil eder.
- private uyelere sadece sinif icerisinde ulasabilir.
- private uyeler alt siniflardan gorulemezler , dolayisiyla private uyelere kalitilmazlar.
- default ve protected uyeler arasında fark kalitim durumunda ortaya cikmaktadır.
- default uyelere sadece ayni pakette bulunan siniflardan ulasim soz konusudur.
- protected uyelere hem ayni pakette bulunan siniflardan hem de kalitim aracigiliyla farkli pakette bulunan alt siniftan erisim mumkundur.

#### [Local Variables \( Yerel Degiskenler\)](#)

- Local (method,automatic veya stack olarak da isimlendirilir) degiskenler access modifier almazlar.( public, protected, private )
- Local degiskenler sadece final anahtar kelimesini kullanabilirler.
- Local degiskenlere varsayılan olarak deger atanmaz bu nedenle kullanılmadan once mutlaka ama mutlaka deger atamasi yapılmalıdır.

#### [Non Access Modifiers – Members](#)

- final metotlar, alt sinif tarafindan override edilemez.
- abstract metotlar noktali virgul ile biterler
- Bir sinif bir adet abstract metoda sahipse abstract olarak tanimlanmalıdır.
- Bir abstract sinif hem abstract hem abstract olmayan metot tanimlayabilir.
- abstract metotlar private ve final olamazlar. Cunku abstract bir sinif kalitilmak icin vardir, private ve final alanlar kalitilamazlar.
- synchronized anahtar kelimesi sadece metotlar ve kod bloklari icin kullanılabilir.
- synchronized metotlar public, protected, private, static, final olarak da tanımlanabilir.
- native , sadece metotlar icin kullanılabilir.
- strictfp , sinif ve metotlar icin kullanılabilir.

#### [Var-Args](#)

- Bir metot tanimlamasında parametre olarak tanımlanan degisken, 0 veya 1 veya 2 ... 3...5 arguman alarak çağırılabilir.Bu yapı Javada Var-Args ile kurulabilir.
- Bir var-args yapısı sadece tek bir var-args kabul eder. method1(int ...x)
- Hem var-arg hem normal degisken de kullanılabilir fakat var-args sonda olmalıdır.

#### [Variable Declaration \(Degisken Tanimlamasi\)](#)

- Instance variable , access modifier alabilirler. final veya transient olarak tanımlanabilirler.
- Instance variable static, abstract, syncronized, native, strictfp olarak tanımlanızlar.
- Local degiskenle instance variable ayni isme sahip olabilir buna golgeleme (shadowing) denir.
- final degiskenlere tekrar bir deger atamasi yapılamaz.
- Java da final object diye bir sey yoktur. Bir object referans degiskeni final olarak tanımlanabilir.
- transient , sadece instance variable icin kullanılabilir.
- volatile, sadece instance variable icin kullanılabilir.

#### [Array Declaration \(Dizi Tanimlamasi\)](#)

- Array (dizi) primitive veya obje tutabilirler. Fakat arrayin kendisi bir objedir.
- Array tanimlarken sol tarafta boyut belirleme derleme hatasina neden olur.

#### Static Degiskenler ve Metotlar

- static degiskenler ve metotlar ilgili sinifin bir objesi ile baglantili degildir.
- static degiskenen sadece bir tek kopyasi vardir.
- static metotlar , static olmayan uye'lere direkt olarak erisemezler.

#### Enum

- Bir enum, liste seklinde tanimlanan , sabit degerlerden olusan, bir yapidir.
- Bir enum, String veya int turunde degildir. Her bir enum , sabit degeri enum tipindedir.
- Bir enum , bir sinif icerisinde veya disarisinda tanimlanabilir. Metot icerisinde tanimlanamaz.
- Sinif disarisinda tanimlanan enum , static , final abstract protected veya private olamaz.
- Enumlar, yapılandirici metot ve degisken tanimlayabilir.
- Her enum values() isminde static metoda sahiptir ve enum in sabit degerlerini dondurur.

## Bölüm-2

### Pure Java – 10 Object Orientation – 01 Encapsulation IS-A HAS-A

Levent Erguder 01 June 2013 Java SE

Merhaba Arkadaslar,

Bu yazimda Java da Encapsulation ,Inheritance , IS-A ve HAS-A kavramlarindan bahsedecegim.

#### Encapsulation

Encapsulation , sarmalama demektir, yani koruma demektir. Bu korumayı saglamak icin;

- instance variable larimizi private olarak tanimlayalim.
- public getter/setter metotlar ile ulasimi saglayalim.
- getter/setter metotlar icin Java Standartlarinda isim kullanalim.

Bu uc maddeye dikkat edersek kodumuz esnek (flexibility) , bakimi kolay (maintainability) ve genislemeye acik (extensibiliyt) olacaktir.

Encapsulation olmayan durum;

```
class B {
    public int var1;
}

public class A {
    public static void main(String[] args) {
        B b = new B();
        b.var1 = -1;
    }
}
```

Encapsulation olan durum ;

```
package chapter1.test1;

class B {
    private int var1;

    public int getVar1() {
        return var1;
    }

    public void setVar1(int var1) {
        this.var1 = var1;
    }
}

public class A {
```

```

public static void main(String[] args) {
    B b = new B();
    b.setVar1(10);
    System.out.println(b.getVar1());
}

```

## Inheritance

Java da inheritance yani kalitim olmazsa olmazlardandır. Java da tüm sınıflar Object sınıfının alt sınıfıdır (subclass). Örneğimizi inceleyelim;

```

public class Test {

    public static void main(String[] args) {
        Test t1= new Test();
        String s="levent";
        int[] array = new int[5];
        int[] array2 = null;

        System.out.println(t1 instanceof Object); //true
        System.out.println(s instanceof Object); // true
        System.out.println(array instanceof Object); // true
        System.out.println(array2 instanceof Object); // false

        System.out.println(null instanceof Object); // false

    }
}

```

Javada String sınıfı da Object sınıfının alt sınıfıdır. Diziler primitive değerler tutsada kendileri objedir !

Kalıtımın 2 amacı vardır ;

- Kod tekrarını önlemek
- Polimorfizm (polymorphism)

```

class Animal {

    public void eat() {
}

```

```

        System.out.println("eating...");
    }

}

public class Bird extends Animal{
    public void fly() {
        System.out.println("flying...");
    }

    public static void main(String[] args) {
        Bird b = new Bird();
        b.fly();
        b.eat();
    }
}

```

Animal sinifimizin 1 metodu var, Bird sinifimizda 1 metot daha tanimliyoruz ve Bird sinifimiz Animal sinifi kalitiyor (extends) bu durumda artık Animal sinifindaki metoda da erisebilir.



Kod tekrarini azalttık

## IS-A

Object Oriented yapısında , IS-A yapısı kalıtma ( inheritance) ve arabirim uygulamasına (interface implementation) dayanmaktadır.

IS-A bu xxsinif/xxarabirim bu yysinif/yyarabirim tipindedir demektir.

```

class Vehicle {}

class Car extends Vehicle {}

interface Speedy{

}

public class Subaru extends Car implements Speedy{

}

```

Subaru IS-A Car

Subaru IS-A Vehicle

Subaru IS-A Speedy

Car IS-A Vehicle

Gordugunuz gibi bir alt sinif (subclass) ile ust sinif (super class)/arabirim arasında IS-A yaklasimi vardir.

### **HAS-A**

Bir sinif icerisinde , bir baska sinifin turunde degiskeni (instance variable) oldugu durumdur.

```
class A {  
    private B b;  
}  
  
class B{  
}
```

A HAS-A B yani A sinifi B sinifinin turunde bir degiskene sahiptir.

## **Pure Java - 11 Object Orientation - 02 Polymorphism (Polimorfizm)**

[Levent Erguder](#) 03 June 2013 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda Java'nin can alici konusu olan Polymorphism /polimorfizm yani çok biçimlilik kavramından bahsedecegim. Bir onceki yazimda bahsettigim üzere bir Java objesi birden fazla IS-A testini gecebilir.

Bir referans degiskeni sadece bir tek tipte olabilir ve tipi asla degismez.

Bir referans degiskeni, bir baska objeye atanabilir.

Bir referans degiskeni, class veya interface turunde olabilir.

Javada hatırlayacagimiz gibi bir sinif sadece tek bir sinifi kalitabilir (extends) , birden fazla arabirim (interface) uygulayabilir (implements).

```
class B {  
}  
  
interface C {  
}  
  
public class A extends B implements C {  
    void method1() {
```

```

A a = new A();
Object o = a;
B b = a;
C c = a;
}
}

```

Ornegimizi inceleyecek olursak , A sinifi B sinifi kalitmakta ve C arabirimini uygulamaktadir.

```

A a = new A();
Object o = a;
B b = a;
C c = a;

```

Burada sadece A turunde bir tek obje yaratilmistir. Fakat 4 adet farkli turde (Object,B,C,A turlerinde) referans degiskeni vardir.(a,b,c,o) Bu referans degiskenleri ayni objeyi gostermektedir/referansta bulunmaktadır.

Yukaridaki ornekte oldugu gibi, bir alt sinifin referans degiskeni her zaman bir ust sinifin ya da arabirimin referans degiskenine atanabilir. Java da bu yapiya *upcasting* denir.

Bununla birlikte metodlarin cagrilmasi dinamik olarak gerceklesmektedir. Java da buna *dynamic method dispatch* denir.

A sinifi B sinifini kalitti ve C arabirimini uyguladigi icin asagidaki IS-A testi dogrudur.

*A IS-A B*

*A IS-A C*

*A IS-A Object*

Bu yazim bir giris oldu, daha sonraki yazilarimda *Overriding/Overloading* konularindan bahsedecegim.

## Pure Java – 12 Object Orientation – 03 Overridden

Levent Erguder 08 June 2013 Java SE

Merhaba Arkadaslar,

Bu yazimda Java da *Overriding* kavramindan bahsedecegim .Javada bir sinifi kalittigimizda (extends) ust sinifta bulunan bir metodu alt sinifta ezelbiliriz/override. Hatirlayacagimiz gibi final olarak tanimlanan metodlar override edilemezler.

Test sinifimiz üzerinde devam edelim..

TestAnimals.java

```
class Animal {
```

```

int height=15;
static int size=10;

public void eat() {
    System.out.println("Animal-Eat");
}

static void breathe() {
    System.out.println("Animal- Breath");
}

}

class Bird extends Animal {

int height=3;
static int size=5;

@Override
public void eat() {
    System.out.println("Bird-Eat");
}

static void breathe() {
    System.out.println("Bird- Breath");
}

public void fly() {
    System.out.println("Bird-fly");
}

}

public class TestAnimals {
    public static void main(String[] args) {
        Animal a = new Animal();
        Animal b = new Bird();
        Bird c = new Bird();
    }
}

```

```

//Bird d = new Animal();

    a.eat();
    b.eat();
    c.eat();

    a.breathe();
    b.breathe();
    c.breathe();

//a.fly();
//b.fly();
c.fly();

System.out.println(a.height + " " + a.size);
System.out.println(b.height + " " + b.size);
System.out.println(c.height + " " + c.size);

}

}

```

- Bird sınıfı Animal sınıfını kalıtmaktadır ve eat() metodunu ezmektedir. @Override notasyonunu kullanabiliyoruz.
- static metodlar override edilmezler, tekrar tanımlanırlar /redefined.

Örnekimizi çalıştırıldığımızda şu çıktıyi alırız :

*Animal-Eat*

*Bird-Eat*

*Bird-Eat*

*Animal- Breath*

*Animal- Breath*

*Bird- Breath*

*Bird-fly*

*15 10*

*15 10*

*3 5*

Cıktıları tek tek inceleyelim,

```
Animal a = new Animal();
```

```
a.eat();
```

a referans degiskeninin Animal sinifinda bulunan eat() metodunu cagirmasi gayet dogal.

```
Animal b = new Bird();
```

```
b.eat();
```

b referans degiskeni Animal turundedir neden Bird sinifina ait eat() metodu cagrildi ?

Cunku Obje Bird objesidir ( new Bird(); ) . Java ‘da override edilen metotlar icin obje turune bakilir. Burada Animal sinifinda yer alan eat() metodu degil Bird sinifinda override edilen eat() metodu cagrilar.

```
Bird c = new Bird();
```

```
c.eat();
```

c referans degiskeni Bird turundedir ve yaratilan obje de Bird turundendir. Bu nedenle Bird sinifinda yer alan eat() metodu cagrilar.

```
Bird d = new Animal();
```

Type mismatch hatasi aliriz. Cunku Animal IS-A Bird testi saglanmaz.

Javada hiyerarsik olarak alta olan bir sinifin referans degiskeni ( burada d ) hic bir sekilde hiyerarsik olarak ustte olan (super class) bir sinifin objesini referans edemez. Yani sag tarafa new ile yazilan hic bir sinif soldaki sinifin ust sinifi olamaz.

static metotlarin cagrilmasina bakacak olursak ;

```
a.breathe();
```

a referans degiskeni ile cagrilarn breathe() metodunun Animal sinifinda yer alan breathe() metodu olmasini bekleriz. Burada bir problem yok.

```
b.breathe();
```

peki b referans degiskeni neden Animal sinifindaki breathe() metodunu cagirdi. Biraz once ayni b referans degiskeni Animal sinifindaki degil Bird sinifindaki metodu eat() metodunu cagirmisti. static metotlar override edilemezler ! Bu nedenle objenin turune degil referans degiskeninin turune bakilir. Referans degiskenimiz Animal turunde oldugu icin Animal sinifinda yer alan breathe() metodu cagrilar !

```
c.breathe();
```

c referans degiskeni ile cagrilan breathe() metodu Bird sinifina ait olacaktir. Burada farkli bir durum yoktur.

```
//a.fly();
//b.fly();
```

a referans degiskeni zaten Animal turunde bir objeyi referans ettiyi icin , Animal sinifinda olmayan fly() metodunu cagiramaz. Burada ilginc bir durum yok.

b referans degiskeni Animal turunde olmasina ragmen Bird turunde bir objeye referansta bulunur. Bird sinifi fly() metoduna sahiptir. Fakat buna ragmen fly() metodu b referans degiskeni ile cagrilamaz. Cunku b referans degiskeni Animal turundedir ve Animal sinifinda boyle bir metot yoktur.

```
c.fly();
```

c referans degiskeni hem Bird turundedir hem de bir Bird objesine referansta bulunur. Bu nedenle sorunsuzca fly() metodu cagrilabilir.

Degiskenlerin cagrilmasini inceleyecekl olursak ;

```
System.out.println(a.height + " " + a.size);
```

a referans degiskeni ile height ve size degiskenlerine ulasiyoruz. Tahmin ettigimiz gibi Animal sinifinda verilen degerler geliyor ,15 ve 10

```
System.out.println(b.height + " " + b.size);
```

b referans degiskeni static metotda oldugu gibi yine Animal sinifinda atanan degerleri dikkate alacaktir. Referans degiskenimiz Animal turundedir , dolayisiyla 15 ve 10 sonuclari atanacaktir.

```
System.out.println(c.height + " " + c.size);
```

c referans degiskeni Bird turunde ve yaratilan objede Bird turunde oldugu ici zaten Bird sinifinda yer alan atama islemleri dikkate alinacaktir.

Sonuc olarak söyleyebiliriz ki ;

- Override edilen metotlar icin Javada calisma zamaninda hangi metodun cagrilacagini objenin turu belirler

- static metotlar override edilemezler, tekrar tanimlanirlar /redefined. Bu nedenle objenin turu degil referans degiskenenin turu dikkate alinir.
- Degiskenlerin cagrilmasi/atanmasi da referans degiskenenin turune goredir.
- Sadece ama sadece override edilen metotlar icin obje turune bakilir !

Java da Override ile ilgili ilk kismi burada bitiriyorum.

## Pure Java – 13 Object Orientation – 04 Overloaded

Levent Erguder 15 June 2013 Java SE

Merhaba Arkadaslar,

Bu yazimda Java'da Overloaded(asiri yükleme) kavramindan bahsedecegim. Bir onceki yazimda OVERRIDING kavramindan bahsetmestim.

Java'da , metot parametre deklarasyonu/ifadesi/tanimlamasi farkli olmasi kosuluyla , ayni sinif veya altsinifta ayni isimde birden fazla metot tanimlanabilir. Bu metotlara Java' da asiri yuklenmis/ Overloaded metot denir.

Asiri yuklenmis/overloaded metot cagrildiginda hangi metodun cagrılacagina karar vermek icin, arguman tipi ve/veya sayisi kullanilir. Bu nedenle asiri yuklenmis/overloaded metotların parametreleri sayisi ve/veya turu *farkli* olmak zorundadir.

Basit bir ornek ile baslayalim ;

### Sum.java

```
public class Sum {
    public static void main(String[] args) {
        Sum s= new Sum();
        System.out.println(s.getSum(10, 20)); // int
        System.out.println(s.getSum(5.5, 20)); //double
        System.out.println(s.getSum(10, 2.7)); //double
        System.out.println(s.getSum(1.4, 2.2)); //double
    }
    public int getSum(int a, int b) {
        return a + b;
    }
    public double getSum(double a, double b) {
        return a + b;
    }
}
```

Gordugunuz gibi c̄esitli dēerlerle metotlari cagiriyoruz. Java bizim icin gerekli ayarlamalari yapiyor yani 2.metot icin 5.5 dēeri ve 20.0 olarak metoda uyacak sekilde ayarliyor. Bu kucuk ornekten sonra devam edelim ve 2. ornegimize bakalim;

### AnimalTest.java

```
class Animal {  
    public void eat() {  
        System.out.println("Animal-Eat");  
    }  
  
    public void eat(String s) {  
        System.out.println("Animal-Eat Overloaded");  
    }  
    public void eat(Animal a) {  
        System.out.println("Animal-Eat Animal");  
    }  
}  
  
class Bird extends Animal {  
  
    @Override  
    public void eat() {  
        System.out.println("Bird-Eat Override");  
    }  
  
    public void eat(int a) {  
        System.out.println("Bird-Eat Overloaded");  
    }  
  
    public void eat(Bird a) {  
        System.out.println("Bird-Eat Animal");  
    }  
}  
  
public class AnimalTest {  
    public static void main(String[] args) {
```

```

Animal a = new Animal();
Bird b= new Bird();
Animal ab = new Bird();

a.eat();
a.eat("overloaded");
a.eat(a);

b.eat();
b.eat(1);
b.eat(b);

ab.eat();
ab.eat("overloaded");
ab.eat(ab);

}

}

```

*Animal-Eat*

*Animal-Eat Overloaded*

*Animal-Eat Animal*

*Bird-Eat Override*

*Bird-Eat Overloaded*

*Bird-Eat Animal*

*Bird-Eat Override*

*Animal-Eat Overloaded*

*Animal-Eat Animal*

Ornegimizi inceledigimizde ozellikle dikkat etmemiz gereken nokta

```
ab.eat(ab);
```

Neden Animal-Eat Animal metodunu cagirdi ? Neden Bird-Eat Animal sinifini cagirmadi ?

*Hangi Overloaded metodun cagrilacagagi overridden metotlarin aksine , dinamik olarak Runtime 'da degil compile(derleme) zamaninda belirlenir. Overloaded metotlarda Overridden metotlarin aksine Obje turu degil referans turu onemlidir ve referans turune gore ilgili metot cagrilir.*

Simdi de overloaded metotlarin ozelliklerini incelemeye devam edelim ;

- Overloaded metotların argumanı listesi değiştirmek/farklı olmak ZORUNDADIR!
- Overloaded metotların donus tipleri değiştirebilir.
- Overloaded metotların access modifier /erisim belirtecleri değiştirebilir. (public private..)
- Overloaded metotlarda yeni exception tanımlanabilir veya tanımlanan exception genişletilebilir.
- Overloaded metotlar aynı sınıfta veya alt sınıfta tanımlanabilir.
- Aynı anda hem overloading hem overriding metotlar kullanılır.

Ne demek istediğimizi daha iyi anlayabilmek adına bu özellikleri kod üzerinde görelim ;

### OverloadedTest.java

```
public class OverloadedTest {
    public void OverloadedMethod() {

    }

    public void OverloadedMethod(int a) {

    }

    String OverloadedMethod(String s) {
        return s;
    }

    private int OverloadedMethod(char a) throws IllegalArgumentException{
        return 1;
    }

    protected Integer OverloadedMethod(boolean b, double d) {
        return 1;
    }
}

class SubOverloadedTest extends OverloadedTest {
    @Override
    public void OverloadedMethod(int a) {
```

```
public void OverloadedMethod(int a,int b) {}  
}
```

Ornegimizi dikkatlice inceledigimizde yazdigim ozellikleri uyguladigimizi gorebilirsiniz.

- 1.metodumuz parametre almiyor ve public olarak tanimlandi.
  - 2.metodumuz int bir parametre aliyor ve public olarak tanimlandi.
  - 3.metodumuzda donus tipini degistirdik , sadece donus tipini degistirmek yeterli olmaz, mutlaka ama mutlaka parametre listesi farkli olmalidir. Ayrica metodumuz icin varsayılan erisim belirteci kullanildi.
  - 4.metotta protected erisim belirtecini kullandik ve metodumuzda bir exception(istisna) tanimladik.
  - 5.metotta private erisim belirtecini kullandik.
  - 6.metotta overloading (asiri yükleme) islemi degil overriding (gecersiz kilma) islemi var.
  - 7.metotta yine overloading islemi gerceklestirdik, alt siniflarda da overloaded metod tanimlayabiliyoruz.
- Overloaded metodların biraz daha detayına ilerleyen yazılarım da gireceğim. Simdilik bu kadar detay yeterli olacaktır

## Pure Java - 14 Java Tricks - Overloaded vs Overridden

Levent Erguder 21 June 2013 Java SE

Merhaba Arkadaşlar,

Bu yazimda Overloaded ve Overridden kavramları arasındaki farklardan ve triklerden bahsedeceğim. Son 2 yazida Overridden ve Overloaded konusunu incelemistik, henuz incelememiz gereken bir kaç detay durum daha var bunlara sırası gelince degecegiz.

### Arguman tipi

- Overloaded metodlarda mutlaka değiştirmek zorundadır.
- Overridden metodlarda mutlaka aynı olmalıdır.

### Donus tipi

- Overloaded metodlarda değiştirebilir, sorun çıkartmaz.
- Overridden metodlarda , covariant return ( ilerleyen yazılıarda değişecek ) haric , değisemeyiz.

### Exceptions(Istisnalar)

- Overloaded metodlarda değiştirebilir, eklenebilir sorun çıkartmaz.
- Overridden metodlarda checked exception eklenemez fakat çıkartılabilir.
- Unchecked exception eklenebilir/cıkartılabilir. (Exception konusuna ilerleyen zamanlarda gelecegiz )

### Erisim belirteci

- Overloaded metotlarda degisebilir.
- Overridden metotlarda daha az kısıtlamalı olacak şekilde degisebilir. public metodu private/protected/varsayılan erişim belirteci olacak şekilde override edemeyiz.

### Metot Cagrilmasi

- Overloaded metotlarda referans tipi /reference type önemlidir ve argumanın tipine göre seçim yapılmaktadır. ( Argumanın tipine göre seçimle ilgili detaylı bir overloaded yazısı ekleyeceğim )
- Overloaded metotlar compile/derleme zamanında hangi metodun çağrılacağı belirlenir.
- Overridden metotlarda objenin turu/ object type önemlidir.
- Overridden metotlarda çalışma/runtime zamanında dinamik olarak hangi metodun çağrılacağı objenin turune göre belirlenir. Bu Dynamic Method Dispatch denilir.

Buradaki her bir maddeyi özümseyerek örnekler yaparak %100 anlamak gerekmektedir

## Pure Java - 15 Reference Variable Casting

Levent Erguder 04 July 2013 Java SE

Merhaba Arkadaşlar,

Bu yazımında Reference Variable Casting / referans değişken dönüşümü konusundan bahsedeceğim.

Daha önce de belirttiğim gibi, Java'da 2 tür değişken vardır

- primitive / ilkel( char, boolean, byte, short, int, long, double ve float)
- reference/referans

Bir reference variable/referans değişken bir objeye erişmek/access için kullanılır. Reference Variable belirli bir tipte tanımlanır ve tanımlanan bu tür değişkeni değiştiremez.

Bir reference variable/referans değişkeni, tanımladığı tipin objeye veya tanımladığı tipin alttipi/subtype turundeki objelere ulaşmak için kullanılabilir. Fakat üst tipin bir objeye erişmek için kullanılamazlar.

### CastTest.java

```
class Animal {  
    void eat() {  
        System.out.println("Animal Eat");  
    }  
}  
  
class Dog extends Animal {
```

```

@Override
void eat() {
    System.out.println("Dog Eat");
}

void bark() {
    System.out.println("Dog Bark");
}

}

public class CastTest {
    public static void main(String[] args) {
        Animal[] animal = { new Animal(), new Dog(), new Animal() };

        for(Animal a: animal){
            a.eat();
            if(a instanceof Dog) {
                //a.bark();
                ((Dog) a).bark();
            }
        }
    }
}

```

Ornegimizi inceleyecek olursak , Animal sinimizda eat() metodu bulunmaktadır, Dog sinifinda ise eat() metodunu override ediyoruz ve bark() metodunu ekliyoruz.

CastTest sinifimizda Animal turnude bir dizimiz var ve 3 tane eleman ekliyoruz. Burada dikkat ederseniz Animal dizimize Dog objesi ekleyebiliyoruz, cunku Dog sinifi Animal sinifini kalitmaktadir. Boylece, Dog IS-A Animal sarti saglanmaktadır.

if kontrolu içerisinde a'nin Dog turunde olup olmadigini kontrol ediyoruz, bu durumda ( new Dog() olarak ekledigimiz ikinci eleman sarti saglayacaktir ).

Burada Dog sinifinda yer alan bark() metodunu cagirabilmek icin reference variable/ referans degiskenimizi *cast* islemine tabi tutuyoruz.

Burada oldugu gibi , kalitim agacinda ust siniftaki bir sinifi alt sinifa cast etme islemine *downcast* denilir.

Simdi de ilginc bir noktayı inceleyelim ;

### CastTest2.java

```
class Animal {  
}  
  
class Dog extends Animal {  
}  
  
public class CastTest2 {  
    public static void main(String[] args) {  
        Animal animal = new Animal();  
        Dog d = (Dog) animal;  
    }  
}
```

Bu kod sorunsuzca compile edilir /derlenir. Fakat calisma zamaninda soyle bir hata ile karsilasilar ;  
*java.lang.ClassCastException*

Compiler burada derleme zamaninda hata vermemektedir, *downcast* in sorunsuz bir sekilde calisabilme ihtimaline karsi calisma zamanina kadar sorun cikartmaz.

```
Animal animal = new Dog();
```

seklinde olursa *downcast* sorunsuz bir sekilde gerceklesir.

Fakat soyle bir casting islemine derleme zamaninda dahi izin vermez ;

```
String s= (String)animal;
```

Animal ve String kalitim hiyerarsisinde farkli yerlerde bulunmaktadır ve burada casting islemi yapilmasi mumkun degildir.

downcasting oldugu gibi bir de *upcasting* islemi vardir ;

```
class Animal {  
}  
  
class Dog extends Animal {  
}
```

```

public class CastTest2 {

    public static void main(String[] args) {
        Animal animal = new Dog();
        Dog d = (Dog) animal;
        // String s= (String)animal;

        Dog d2 = new Dog();
        Animal a1 = d2;
        Animal a2 = (Animal) d2;
    }
}

```

Dog turunde bir obje olusturduk ve d2 reference variable / referans degiskenimizi Animal turundeki reference variable / referans degikenine atiyoruz. Burada oldugu gibi kalitim hiyerarsisinde ust sinifa dogru cast etme islemine upcasting denilmektedir.

*Animal a1=d2;* de oldugu gibi *imlicit* /kapali/ortulu veya *Animal a2=(Animal)d2;* de oldugu gibi *explicit* /acik olarak cast etme islemi yapilabilir.

Dog IS-A Animal ifadesi gecerlidir , cunku Dog , Animal sinifini kalittiği için sorunsuz bir sekilde casting yapılabilir.

## Pure Java – 16 Implementing an Interface

Levent Erguder 10 July 2013 Java SE

Merhaba Arkadaslar,

Bu yazımızda Java'da interface /arabirim konusuna biraz daha detaylıca bakacağız. Interface/arabirim tanımlamasını su yazında incelemistik <http://www.injavawetrust.com/pure-java-03-interface-declaration-3/> Daha önce degindigim gibi Java 'da interface/arabirim bir kontrattır ve uygulayan sınıflar (abstract olmadığı surece) bu kontrata uymak zorundadır yani metodları uygulamak zorundadır.

interface ile abstract sınıfların en önemli farkı sudur ; bir arabirim kalitim hiyerarşisi önemsiyor herhangi bir sınıfta uygulanabilir . Orneğin Runnable arabirimini birbirile hic alakası olmayan sınıflar ihtiyacı doğrultusunda uygulayıp kullanabilir.

Bu arabirimin en önemli ve en guclu noktasıdır.

Simdi bir arabirim/interface tanımlayalım ve incelemeye devam edelim ;

Speed.java

```
interface Speedy {
```

```
void speedUp();  
void beCool();  
}
```

Yukarıda linkini verdigim yazida belirtigim gibi , interface/arabirimlerde sadece govdesiz metot yani abstract metot tanimlanabilir ve bu metotlar mutlaka publictir. Ee hani nerede abstract anahtar kelimesi Java bizim icin bunu halleder.

abstract ve public anahtar kelimelerinin yeri degisebilir, yukarıdaki tanimlamayla su tanimlama ayndir.

```
interface Speedy {  
    public abstract void speedUp();  
    abstract public void beCool();  
}
```

### Vehicle.java

```
abstract public class Vehicle implements Speedy {  
  
    abstract int goDirection();  
  
    boolean haveEngine() {  
  
        return true;  
    }  
}
```

Vehicle sinifimizda Speedy arabirimizi uyguladik / implements . Fakat Vehicle sinifimiz abstract oldugu icin Speedy interface/arabirimine ait olan metotlari override etmedik. Bir abstract sinif bir arabimiri uyguladiginda/implements , interface/arabirime ait metotlari override etmek zorunda degildir.

### Car.java

```
public class Car extends Vehicle{  
  
    @Override  
    public void speedUp() {
```

```

    }

    @Override
    public void beCool() {

    }

    @Override
    int goDirection() {
        return 0;
    }

}

```

Abstract olmayan / concrete olan Car sınıfımız, Vehicle sınıfını extends ettiginde override edilmeyen tüm metotları override etmek zorundadır. Yani Vehicle abstract sınıfında bulunan abstract metotları ve Speedy interface/arabirimde bulunan ve Vehicle abstract sınıfı tarafından override edilmeyen metotları override etmek zorundadır.

Yani ilk abstract olmayan/ concrete sınıf tüm abstract metotları mutlaka override etmelidir.

Car sınıfı Vehicle sınıfını kalıtmakta ve Vehicle sınıfı da Speedy arabirimini uygulamaktadır. Bu nedenle su tanımlamalar doğrudur ;

Car IS-A Vehicle

Car IS-A Speedy

Vehicle IS-A Speedy

Car sınıfı aslında Speedy sınıfını implements etmektedir yani yukarıdaki tanımladığımız Car sınıfı tanımlaması/declare su ifade ile aynıdır.

```

public class Car extends Vehicle implements Speedy{
    //
}

```

- Bir sınıf birden fazla interface/arabirimini uygulayabilir /implements.
- Bir sınıf sadece tek bir sınıfı kalıtılabilir/extends.
- Bir arabirim birden fazla arabirimini kalıtılabilir /extends.

- Bir arabirim ne bir sınıfı ne bir arabirimini uygulayabilir /implements.
- Önce extends anahtar kelimesi sonra implements anahtar kelimesi kullanılmalıdır.

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

```
abstract class Ford implements Speedy extends Car{  
    // yanlis  
}  
  
abstract class Ford extends Car implements Speedy {  
    //dogru  
}
```

## Pure Java – 17 Return Type

Levent Erguder 13 July 2013 Java SE

Merhaba Arkadaşlar,

Bu Java yazımında metodların dönüş tipi ile ilgili durumları inceleyeceğiz. Daha önce bahsi geçen *covariant return type* ile başlayalım.

Overrided metodlarda dönüş tipi değişmeyeceğini, fakat covariant /esdeger dönüş tipi olursa değiştireceğini belirtmeliyiz. Örnek üzerinde inceleyelim ;

```
public class Car {  
    Car speedUp() {  
        return new Car();
```

```

    }

}

class Ford extends Car {

    @Override
    Ford speedUp() {
        return new Ford();
    }
}

```

Car sinifinda bulunan speedUp metodu Car donus tipindedir, Ford sinifi Car sinifini kalitmakta ve speedUp metodunu override etmektedir. Burada dikkat ederseniz donus tipimiz Car degil Ford oldu. Iste burada oldugu gibi override edilen metodun donus tipinin altsinif donus tipinde oldugu durumlarda covariant return type gecerli olacaktir.

Cunku Ford IS-A kurali gecerli olacaktir.

Daha once belirttigim gibi overrided metotlarda covariant return type disinda donus tipi degisemeyecegi gibi overloaded metotlarda bu durumda bir kisitlama yoktur donus tipi degisebilir.

```

void beCool() {

}

int beCool(int cool) {
    return 1;
}

```

Donus tipiyle ilgili olarak su kurallari aklimizda bulunduralim;

- Bir object referans tipinde donus tipi soz konusu oldugunda null kullanilabilir;

```

public JButton createButton() {
    return null;
}

public Integer giveMeInt(){
}

```

```

        return null;
    }

    public String giveMeString() {
        return null;
    }

    public String[] giveMeArray() {
        return null;
    }

    public Double[][] giveMeMatrix() {
        return null;
    }

    public Boolean giveMeBool() {
        return null;
    }
}

```

- primitive bir donus tipi icin bu durum gecerli degildir.

```

int beCool(int cool) {
    return null; //compile error
}

```

- primitive bir donus tipli bir metot icin ; tanimlanan donus tipine implicit/dolayli olarak donusturulebilecek bir donus tipi return ifadesinde kullanilabilir.

```

public int giveMeInt(){
    //return 'c';
    //return new Byte((byte)10);
    return new Short((short)10);
}

```

```

public float giveMeFloat() {
    //return 'c';
    //return new Byte((byte)10);
    //return new Short((short)10);
    return 10;
}

```

- primitive bir donus tipi metot icin ; tanimlanan donus tipine explicity/acik bir sekilde cast ederek kullanilabilir.

```
public int castFloat() {
    float f = 32.5f;
    return (int)f;
}
```

- Bir metodun donus tipi olmak zorunda degildir. void donus tipi belirtildiginde return ile bir deger dondurmeye calismak compile error / derleme hatasina yol acacaktir.

```
void iVoid(){
    //return 10; compile error
    return;
}
```

- Obje referans donus tipine sahip metot icin ; tanimlanan donus tipine implicity /dolayli olarak cast edilebilecek bir donus tipi kullanilabilir.

```
public Car getCar() {
    return new Ford();
}
```

```
public Object getObject() {
    int [] number = {1,2,3};
    return number;
}
```

Burada IS-A kurali gecerlidir, Ford IS-A Car, Array IS-A Object.  
IS-A kuralinin interface/arabirimler icin de gecerli oldugunu unutmayalim ;

```
interface Speedy {}

class Subaru implements Speedy {
    public Speedy getSpeedy() {
        return new Subaru();
    }
}
```

## Pure Java - 18 Constructor - 01

Levent Erguder 25 July 2013 Java SE

Merhaba Arkadaslar,

Bu yazimda Java'da constructor/ yapılandirici kavramından bahsedecegim.

Javada , yeni bir obje olusturuldugunda constructor/yapilandirici calistirilir/invoke , sinifin kendi yapılandiricisinin yaninda superclass'in yapılandiricisi da calisir detaylar biraz asagida olacak.

Peki yapılandirici ne zaman calisacaktir ? new anahtar kelimesini kullandigimizda yapılandirici calisacaktır. Bir de serilestirme yapısında calismaktadir fakat bu konuya bir hayli ilerde gelecegiz. Biz basit temeller ile baslayalim ;

### Basit Temeller

- Abstract/soyut siniflar dahil tum sinifların yapılandiricisi/constructor olmak zorundadir.
- Yapılandırı/constructor olması zorunluluğu tanımlanmak zorunda olduğu anlamına gmez.
- Eger bir sinif icin yapılandırıcı/constructor tanımlamazsa otomatik olarak varsayılan yapılandırıcı gizli olarak tanımlanır.
- Varsayılan yapılandırıcı herhangi bir parametre almaz. Varsayılan yapılandırıcıyı biz kendimiz de tanımlayabiliriz.

```
class A {  
    // A(){} A sinifina ait varsayılan yapılandırıcı  
  
}
```

- Eger kendimiz bir yapılandırıcı tanımlarsak , varsayılan yapılandırıcı otomatik olarak tanımlanmaz.

```
public class A {  
  
    int x;  
    int y;  
  
    public A(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public static void main(String[] args) {  
        A a = new A(10,20);  
        //A a2 = new A(); derleme hatası  
    }  
}
```

```
}
```

```
}
```

Yukarıdaki basit ornekte varsayılan yapılandıcı ile bir A objesi oluşturulmak istenmektedir buna rağmen derleme hatası olmaktadır, cunku ilgili yapılandırıcı mevcut değildir. Bunun nedeni biz kendimiz bir yapılandırıcı tanımladığımız için varsayılan yapılandırıcı otomatik olarak oluşmaz.

Derleme hatasını cozmek için varsayılan yapılandırıcıyı da tanımlamamız gerekmektedir.

### **Yapilandırıcı Zincir Yapısı / Constructor Chaining**

Horse h = new Horse();’ı inceleyelim, burada Horse sınıfının Animal sınıfını kalıttığını varsayıyalım. Şu adımlar gerçekleştirilecektir ;

- Horse sınıfının yapılandırıcısı çağrılmaktadır. Her yapılandırıcı , ait olduğu sınıfın superclass’ına ait yapılandırıcıyı gizli/implicit olarak çağrıır. super() yapısı burada devreye girer. Her yapılandırıcının basında (yani genel olarak) super() çağrısimi bulunmaktadır. Farklı durumları sonraki yazınlarda inceleyeceğiz.
- Animal sınıfına ait yapılandırıcı çağrılrı.
- Object sınıfına ait yapılandırıcı çağrılrı, cunku Object sınıfı tüm sınıfların superclass’ıdır. Animal sınıfı Object sınıfını gizli/implicit olarak kalıtmaktadır.
- Stack/yigin mantığında olan bu yapıda Object sınıfının yapılandırıcı tamamlanır sonrasında Animal sınıfının yapılandırıcı ve en son olarak da Horse sınıfı yapılandırıcı sonlanır.

orneği inceleyelim;

```
class B {  
    int t;  
  
    public B(int t) {  
        super();  
        this.t = t;  
    }  
  
}  
  
public class A extends B {  
  
    int x;  
    int y;
```

```

public A(int x, int y) { //derleme hatasi
    this.x = x;
    this.y = y;
}

public static void main(String[] args) {
    A a = new A(10,20);
    //A a2 = new A();
}
}

```

A sinifi , B sinifini kalitmaktadır, A sinifina ait yapılandıricıda hata meydana gelmektedir. Peki neden ? Cunku bu yapılandırinin basında gizli olarak super(); anahtar kelimesi vardır ve B sınıfına ait yapılandıriciyi çağırmaktadır. Fakat biz int bir parametre alan yapılandıricı tanımladığımız için parametresiz olan varsayılan yapılandıricı tanımlanmadığından derleme hatası alırız. B sınıfına varsayılan yapılandıriciyi eklemek sorunu çözülecektir.

Yapılandırıcı ile ilgili ilk yazımда işin en kolay temel örnekleri üzerinde durmaya çalıştım. Sonraki yazılarım da enince detayına kadar incelemeye devam edeceğiz.

## Pure Java – 19 Constructor – 02

[Levent Erguder](#) 26 July 2013 [Java SE](#)

Merhaba Arkadaşlar,

Bu yazımda Java' da Constructor/yapılandırıcı konusuna devam edeceğim. İlk yazida temel yapıdan bahsetmemistim burada biraz daha detaya inceleyeceğiz.

Yapılandırıcı/ Constructor için kurallar

- Yapılandırıcılar private dahil tüm access modifier / erişim belirteçlerini kullanabilirler.

```

class B {
    private B() {} //private yapılandırıcı
}

```

- Yapılandırıcıların ismiyle sınıfın ismi aynı olmak zorundadır.

- Yapilandiricilarin donus tipi olamaz , void de olamaz.
- Bir sinifta sinif ismiyle ayni olacak sekilde metot tanimlanabilir. Bu durumda yapilandirici tanimlanmis olmaz.

```
public class A {
    void A() {
        // metot
    }

    A() {
        //yapilandirici
    }
}
```

- Eger sinifimiz icin bir yapilandirici tanimlamazsa varsayılan yapilandirici olusturulur.

```
class C {
    // varsayılan yapilandirici otomatik olarak olusur
}
```

- Varsayılan yapilandirici her zaman no-args/ arguman almayan yapilandiricidir.
- Yapilandiricinin access modifier/erisim belirteci ile sinifin erisim belirteci aynidir. Yani default ise default public ise publictir.

```
//bizim tanimladigimiz
class C1 {

}

// compiler in gordugu
class C1 {
    C() {
        super();
    }
}

// bizim tanimladigimiz
public class C1{
```

```

}

//compilerin gordugu

public class C1{
    public C1() {
        super();
    }
}

```

- Eger no-args yapılandırıcıyı da kullanmak istiyorsak ve kendimiz başka yapılandırılar tanımladıysak bu sefer otomatik olarak no-args yapılandırıcı olusmayacaktır. Bu yapılandırıcı da tanımlamamız gerekmektedir.

```

class C {

    C(int var) { //varsayılan yapılandırıcı otomatik olusmaz

    }
}

```

- Her yapılandırıcının ilk statement/ifadesi , this() veya super() olmak zorundadır.

```

class D {

    D() {
        System.out.println("Derleme Hatası , super() her zaman ilk ifade olmalıdır");
        super();
    }
}

```

- this anahtar kelimesi ile overloaded/asırı yüklenmiş yapılandırıcılar çağrılabılır.

```

public class A {

    A() {
        this(10);
    }
}

```

```

A(int x) {
    System.out.println(x);
}

public static void main(String[] args) {
    A a = new A();
}

```

- Bir yapılandırıcı yukarıdaki örnekte olduğu gibi sadece bir diğer yapılandırıcı tarafından çağrılabılır/invoke. Bir metod içerisinde çağrılamaz.

```

public class A {

    A(int x) {
        System.out.println(x);
    }

    void method() {
        A(10); // derleme hatası
    }
}

```

- Bir yapılandırıcı hem `this` ile overloaded başka bir yapılandırıcıyı hem `super` ile superclass'ın yapılandırıcısını çağrıramaz.

```

class D {

    D() {
        this(10);
        super(); // derleme hatası
        System.out.println("Derleme Hatası");
    }

    D(int i) {

```

```
    }

}
```

- Birbirini rekursif olarak cagiran yapılandiricilar derleme hatasina neden olur.

```
• public class A {  
•  
•     A0 {  
•         this(10); //derleme hatasi  
•     }  
•  
•     A(int x) {  
•         this(); //derleme hatasi  
•     }  
}  
}
```

- Abstract siniflar dahil tum sinifların yapılandirici olmak zorundadir. Abstract sinifların yapılandiricisi, abstract/soyut sınıfı kalitan sınıfın bir objesi yaratıldığında calistirılır. Onceki yazida belirttigim gibi zincirleme bir çağrıma yapısı söz konusudur.

```
abstract class Abs {  
    Abs() {  
        System.out.println("Abstract/soyut sınıf yapılandırıcı");  
    }  
}
```

```
public class A extends Abs {  
  
    public static void main(String[] args) {  
        A a = new A0;  
    }  
}
```

- Arabirimler/interface yapılandırıcısı yoktur.

```
interface Iface {  
    Iface() { //derleme hatasi  
    }  
}
```

- Yapilandiricilar kalitilmazlar, override edilmezler. Asiri yuklenebilirler /overloaded.

```
class Constructor {  
  
    Constructor() {  
  
    }  
  
    Constructor(int i) {  
  
    }  
  
    Constructor(int i, int j) {  
  
    }  
  
    Constructor(String s, int j) {  
  
    }  
}
```

Son olarak su kodu inceleyelim ;

```
class Animal {  
    Animal(String s) {  
  
    }  
}
```

```
class Horse extends Animal{  
    //derleme hatasi  
}
```

Derleme hatasi verecektir , cunku Horse sinifi Animal sinifini kalitmaktadir ve Horse sinifinin varsayılan yapılandırıcısi super() anahtar kelimesini içermektedir, yani Animal sinifinin varsayılan yapılandırıcını çağırmaktadır, fakat Animal sınıfına ait varsayılan bir yapılandırıcı tanımlamadık.

## Pure Java - 20 Static Variables and Methods

[Levent Erguder](#) 30 July 2013 [Java SE](#)

Merhaba Arkadaşlar,

Bu yazimda Javada static anahtar kelimesinden bahsedecegim ve metotlar ve degisken icin kullanimi inceleyecegiz.

Oncelikle belirtmem gerekir ki static metot degiskeni bol miktarda tanimlamak object oriented yapisinin



temeline dinamit koymaktir

Gerekigi ozel zamanlarda kullanmak gereklidir. Dusunelim bize tum objelerden bagimsiz olarak gerekli bir sayac gerekmektedir bu sayaci nasil tutabiliyoruz. Boyle durumlarda static bir degisken cozum saglayabilir.

static degiskenler ve metotlar sinifa aittirler. Ayri bir instance/ornek/nesne yaratmadan kullanabiliriz. static bir degiskenin tek bir kopyasi bulunur ve tum instance/ornek/nesneler icin bu tek deger paylasilir.

Su basit ornegimizi inceleyelim , static bir int degisken tanimladik ve 0 a atadik , yapılandırıcı içerisinde degerini 1 arttirdik.

```
public class CounterEx {  
  
    static int count =0;  
  
    CounterEx() {  
        count++;  
    }  
  
    public static void main(String[] args) {  
        new CounterEx();  
    }  
}
```

```

        new CounterEx();
        new CounterEx();
        System.out.println("counter value:" + count); //3
    }
}

```

count degerine 0 atamasi islemi CounterEx sinifi JVM ( Java Virtual Machine) ‘e herhangi bir CounterEx instance/nesnesi yaratilmadan once gereklesir. Ek olarak static degiskenler de instance degiskenler gibi deger atanmadigi durumda otomatik deger alirlar. Bildigimiz gibi int bir degisen icin otomatik deger 0 dir.

Peki static degisen degilde instance/ornek/obje degiskeni kullansaydik

```

public class CounterEx {

    int count = 0;

    CounterEx() {
        count++;
    }

    public static void main(String[] args) {
        new CounterEx();
        new CounterEx();
        new CounterEx();
        System.out.println("counter value:" + count); // compile error
    }
}

```

bu durumda derleme hatasi aliriz. Bu Javaya yeni baslayan arkadaslarin basina cok fazla gelen bir hata turudur.

*Cannot make a static reference to the non-static field count*

Peki neden boyle bir sorun oluyor ? JVM hangi CounterEx objesinin , count degiskenine ulasacagini bilmiyor. main() metodunun kendisi static metot ve bu nedenle static bir metot icerisinden static olmayan degiskene ulasim saglayamayayiz.

*A static method cant access a non static(instance) variable or method , because there is no instance! static bir metottan static olmayan bir metoda da ulasamayiz ;*

```

public class CounterEx {

    int count = 0;

    CounterEx() {
        count++;
    }

    void method() {
        System.out.println(count); // OK
    }

    public static void main(String[] args) {
        new CounterEx();
        new CounterEx();
        new CounterEx();
        System.out.println("counter value:" + count); // 3
        method(); // compile error
    }
}

```

Su kodu inceleyelim ;

```

class Counter {

    private int count;

    public int getCount() {
        return count;
    }

    public void setCount(int count) {
        this.count = count;
    }
}

```

```

        static int staticVar;

    }

public class Test {
    public static void main(String[] args) {
        Counter c = new Counter();
        c.setCount(10);

        System.out.println(c.getCount());

        System.out.println(Counter.staticVar);
    }
}

```

Burada ise c Counter turunde bir referans degiskenidir ve bir Counter objesini referans etmektedir. Bu referans ettigi objenin counter degiskene 10 atiyoruz ve sonrasında bu degeri getCount metodunu cagirarak aliyoruz. Burada count degiskeni obje ile baglantilidir.

static degiskene , basina sinif ismini yazarak ulasabiliriz. *Counter.staticVar* da oldugu gibi.  
Burada inceledigimiz ana kurallari tekrar soylecek olursak ;

```

int size =10;

static void doMore(){
    int x =size; //static bir metottan non-static bir degiskene erismek
    // derleme hatasina yol acar.

    // static method cannot access an instance (non-static) variable
}

void go();

static void doMore(){
    go(); // static metottan non-static bir metoda erismek derleme
    //hatasina yol acar

    // static method cannot access a non-static method
}

```

```

static int count;

static void foo(){ }

static void bar() {
    foo();
    int x= count;

    // static metottan static degiskenlere ve metotlara erisim saglanabilir.
    // static method can access a static method or variable.
}

```

Bir kac trick bilgi daha verelim, static metotlar override edilemezler, redefined / tekrar tanimlanirlar.

Oncelikle override isleminin oldugu ornegi inceleyelim , yani metotlarimizin static olmadigi

```

class SuperClass {
    void foo() {
        System.out.println("I am SuperClass");
    }
}

class SubClass extends SuperClass {
    void foo() {
        System.out.println("I am SubClass");
    }
}

public class Test {
    public static void main(String[] args) {
        SuperClass sc1 = new SuperClass();
        SubClass sc2 = new SubClass();
        SuperClass sc3 = new SubClass();

        sc1.foo(); // I am SuperClass
    }
}

```

```

        sc2.foo(); // I am SubClass
        sc3.foo(); // I am SubClass

    }

}

```

Ciktiyi inceleyerek olursak supriz bir durum olmadı bildigimiz override kuralları isledi. Peki ya foo metodumuz static olursa ?

```

class SuperClass {
    static void foo() {
        System.out.println("I am SuperClass");
    }
}

class SubClass extends SuperClass {
    static void foo() {
        System.out.println("I am SubClass");
    }
}

public class Test {
    public static void main(String[] args) {
        SuperClass sc1 = new SuperClass();
        SubClass sc2 = new SubClass();
        SuperClass sc3 = new SubClass();

        sc1.foo(); // I am SuperClass
        sc2.foo(); // I am SubClass
        sc3.foo(); // I am SuperClass
    }
}

```

Ciktiyi inceleyerek olursak , override islemi gerceklesmedi ve sc3 referans degiskeni SuperClass a ait olan foo metodunu cagirdi, oysa override oldugunda SubClass sinifina ait olan foo metodu cagrilir. Ek bir

detay olarak override islemi olmasa da SubClass sinifinda tanimladigimiz foo metodu icin override kurallari *gecerlidir*. Yani SuperClassta public yapip SubClassta protected yapamayiz. Bu kisimlarda sorun varsa override yazilarimi okumanizi öneririm. static ile ilgili farkli durumları iceren yazılarla paylaşmaya çalışacağim.

## Pure Java - 21 Coupling & Cohesion

[Levent Erguder](#) 31 July 2013 [Java SE](#)

Merhaba Arkadaşlar,

Bu yazimda Java' da *Coupling* / baglasim/baglanti yani “nesnelerin birbirine baglanması” ve *Cohesion*/baglilik/yapisiklik kavramlarından bahsedecegim. Ilk bakista Turkce kelime karsiliklari benzer



gozukebilir fakat yazı sonunda kavramları anlamış olacağımızı umuyorum

Oncelikle sunu belirtmem gerekip ki *Coupling* ve *Cohesion* kavramları Object Oriented yaklaşımında olmazsa olmazlardandır, Object Oriented yaklaşımının kalitesiyle doğrudan ilgilidir.

Iyi bir Object Oriented dizaynında *loose coupling* / gevsek baglanti ve *high cohesion* istenen bir durumdur. Bir programda cesitli gorevleri yerine getirecek bir çok sınıf bulunur. Bu sınıflar birbirleriyle etkileşim haline girerek ortak bir şekilde çalışırlar. Bu durumda nesneler arasında bağımlılıklar olusur. Bir sınıf diğer bir sınıf



hakkında ne kadar fazla bilgiye sahipse o sınıfın bağımlılığı o derece artar ( bir nevi ask gibi ) Düşünuldüğünde bu bağımlılıkları ortadan kaldırmak mümkün degildir cunku mutlaka sınıflar birbirleriyle ortak çalışmaya zorundadır. Bu durumda yapacağımız şey esnek bağımlılıklar oluşturmaktır yani *loose coupling*.

### Coupling

Bir sınıfın diğer sınıfları ne kadar bildiği ve sınıf üyelerine nasıl eriştiği ile ilgilidir.

A sınıfı B sınıfının değişkenlerine direkt olarak ulaşmamalıdır, getter/setter metodları üzerinden ulaşmalıdır. B sınıfına ait instance değişkenler public tanımlanmamalıdır.

*Loose Coupling*, A ve B sınıfı örneğinde olduğu gibi sınıfların iyi bir *encapsulation* kuralına uygun olarak tanımlanması, sınıfların birbirlerine olan referans değişkenlerini minimize etme yaklaşımıdır. *Tight Coupling* bu



kurallara uymamak, Object Oriented in divine dinamit koymaktır

```
class A {
```

```
    void doMore() {
        B b = new B();
        b.badBoy = "Levent";
        System.out.println(b.badBoy);
    }
}
```

```
}
```

```
class B {
```

```
    public String badBoy;
```

```
}
```

yerine ;

```
class A {
```

```
    void doMore() {
```

```
        B b = new B();
```

```
        b.setBadBoy("Levent");
```

```
        String goodBoy = b.getBadBoy();
```

```
        System.out.println(goodBoy);
```

```
    }
```

```
}
```

```
class B {
```

```
    private String badBoy;
```

```
    public String getBadBoy() {
```

```
        return badBoy;
```

```
    }
```

```
    public void setBadBoy(String badBoy) {
```

```
        this.badBoy = badBoy;
```

```
    }
```

```
}
```

Su Spring yazimda Coupling , De-Coupling konusunda farkli bilgiler yer almaktadir.

<http://www.injavawetrust.com/spring-ders-1-baslangic-ve-kurulum/>

#### Cohesion

*Coupling* ,siniflarin birbirleriyle etkilesim konusu ile ilgilidir. *Cohesion* ise tekil olarak her bir sinifin nasil dizayn edildigi ile ilgilidir. Her bir sinifin iyi tanimlanmis bir amaci bulunmalidir. Yani bir sinif icerisinde

birbirinden farksız işleri yapmak uygun bir davranış olmayacağındır. Her sınıfın sorumluluğu ve rolü net ve iyi tanımlı olmalıdır.

Aşağıdaki kodu incelediğimizde bir sınıf içerisinde bir çok iş yapılmaktadır ve bu sınıfın bir çok rolü bulunmaktadır.

```
class BudgetReport {  
    void connectToDB() {  
    }  
    void generateReport() {  
    }  
    void saveFile(){  
    }  
    void print(){  
    }  
}
```

Bunun yerine işleri ilgili sınıflara bolmek ve her sınıfın iyi tanımlanmış birbiriyle ilgili görevler yapmak istenilen bir durumdur ve bu yaklaşım *high cohesion* olarak adlandırılır.

```
class BudgetReport{  
    void generateBudgeReport() {  
    }  
}
```

```
class ConnectDB {  
    void getConnection() {  
    }  
}
```

```
class PrintStuff {  
    void printdoc() {  
    }  
}
```

```
class FileSaver {  
    void saveFile() {  
    }  
}
```

}

Yazimi burada sonlandiriyorum. Bu yazı ile Object Orientation konusunu bitirdik.

## BÖLÜM – 3

### Pure Java - 22 Literals

[Levent Erguder](#) 07 August 2013 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda Java da Literal konusuna deginecegiz. Java da literaller ;

- Integer Literals
- Floating-Point Literals
- Boolean Literals
- Character Literals
- Literal Values For Strings

### **Integer Literals**

Java da integer/tamsayı değerlerinin 3 farklı gösterimi vardır. decimal, octal ve hexadecimal literaller. Sırasıyla inceleyelim.

#### Decimal Literal

Decimal Literal her zaman kullandığımız varsayılan gösterimdir,

```
int size=100;
```

```
int length=50;
```

#### Octal Literal

Octal (Sekizli) Literallerde 0 ve 7ye kadar olan rakamlar kullanılabilir. Octal literalleri gösterirken basına 0 eklemeyi unutmayalım. Ornegimizi inceleyelim

#### OctalTest.java

```
public class OctalTest {
    public static void main(String[] args) {
        int zero = 0;
        int one = 01;
        int six = 06;
        int seven = 07;
        int eight = 010;
        int nine = 011;
        //int compileError = 08;

        System.out.println(zero + " " + one + " " + six + " " + seven + " " + eight + " " + nine);
    }
}
```

Ornegimizi incelersek ,eight yani 8 değerini yazarken Octal/8lik sistemi kullandığımızın farkına varabiliriz. 08 şeklinde bir gösterim derleme hatasına neden olur. Cunku Octal sistemde 0 ve 7 kapalı aralığında değerleri kullanabiliriz.

#### Hexadecimal Literal

Octal Literal de olduğu gibi burada 16lik sayı sisteme göre sayılar ifade edilmektedir.

Bildigimiz üzere 16lik sistemde 10-15 arası sayılar şu şekilde ifade edilmektedir.

10 a 11 b 12 c 13 d 14 e 15 f

### HexTest.java

```
public class HexTest {  
    public static void main(String[] args) {  
  
        int a = 0x01;  
        int b = 0X8abc;  
        int c = 0xDEad;  
        System.out.println(a + " " + b + " " + c);  
    }  
}
```

Hex Decimal Literalleri ifade ederken sayimizin basina 0x veya 0X ekliyoruz. Burada hem x hem diger a,b,c karakterlerinde buyuk kucuk harf farkliligi yoktur.a veya A ayni anlami ifade eder.

Bu uc literal de int tipinde tanimlanir. Java da varsayılan tam sayı ; int tir. long turunde bir degisen tanımladığımızda bunu derleyiciye açıkça belirtmemiz gerekmektedir. Bunu l veya L karakteri ile belirtebiliriz.

```
long d = 100L;  
long e = 0234L;  
long f = 0xFabL;
```

### Floating-Point Literals

Kayan Noktalı Literaller, onluk sayıları kesirli bilesenleriyle gösterir. Java da , kayan noktalı literaller varsayılan olarak double dir. float bir literal tanımlamak için biraz önce long literal için yaptığımız gibi bu durumu derleyiciye özel olarak bildirmemiz gerekmektedir.

Bu bildirimini f veya F karakteri ile yapabiliriz. Ek olarak hatırlayacağımız gibi double bir degisen için 64 bit yerine gerekirken float bir degisen için bu durum 32 bittir.

Bir diğer detay olarak dilersek double literaller için D veya d ekini kullanabiliriz. Kullanmasak da varsayılan olarak double literal olacaktır.

```
// float g = 100.343; // derleme zamani hatasi alir  
float h = 100.343f;  
float i = 100.343F;
```

```
double j = 13231.4334D; // D opsiyoneldir  
double k = 43243.254523d; // d opsiyoneldir  
double l = 2013;  
double m = 24.5;
```

Not; Araya virgul degil nokta koydugumuza dikkat edelim.

### Boolean Literals

Bir mantiksal literal true veya false degerini alır. Java da true ve false degerlerinin sayisal bir karsılığı bulunmaz. C de 0 false diger sayilar true anlamına gelmekteydi , Java da böyle bir durum yoktur.

```
boolean b = true;  
boolean c = false;  
//boolean d = 1; // derleme zamani hatasi
```

### Character Literals

Karakter Literaller Java da tek tırnak içerisinde ifade edilirler. Ayrıca karakterin Unicode değerini yazıp da ifade edebiliriz. Hatırlayacağımız gibi Javada karakterler 16 bit unsigned/isretsiz integer değerlerdir.

```
public class CharTest {  
    public static void main(String[] args) {  
  
        char a = 'a';  
        char b = 100;  
        char c = (char) 90000;  
        char d = (char) -20;  
        char e = '\\";  
        char f = '\";  
  
        System.out.println(a + " " + b + " " + c + " " + d + " " + e + " " + f);  
    }  
}
```

a degiskene tek tırnak ile bir karakter atamasında bulunduk.

b degiskene direkt olarak bir integer atamasında bulunduk.

c ve d degiskenlerine bakacak olursak, burada bir cast işlemi söz konusu neden peki ? Yukarıda belirttiğim gibi karakterler 16 bittir (maximum 65535) ve unsigned / isretsiz yani negatif değer alamaz. Bu nedenle 90000 değeri siniri astığı için cast işlemine tabi tutulmalıdır. Aynı şekilde -20 değeri de sinirin altında kaldığı için kast

islemine tabi tutulmalıdır.

e ve f degisenlerinde \ kacis karakter yardımı ile tek tırnak ve çift tırnak karakterlerinin atamalarında bulunduk.

### Literal Values For Strings

Karakter Katları literalleri çift tırnak arasında gösterilir. Bir string literalı, String objesinin değerini temsil etmektedir.

```
String s = " Levent";
```

Burada belirtmeden gecmemiz gerekiyor ki stringler primitive degildir, burada sadece literal kısmı basitçe anlatıldı. İlerleyen konularda String konusuna detaylıca göz atacağız.

## Pure Java - 23 Assignment Operators

[Levent Erguder](#) 12 August 2013 [Java SE](#)

Merhaba Arkadaşlar,

Bu yazımda Java da Assignment/atama operatorunu (= yani eşittir) ve doğal olarak da değişkenler üzerinde incelemelerde bulunacağız.

Oncelikle *degisen* nedir ? Değişkenler Java da birer bit tutucularıdır/ bit holders.

int , double, String, Button, String[] turunda değişken yani bit tutucular tanımlayabiliriz.

*primitive* değişkenleri inceleyeceğiz olursak. Örneğin ; byte bir değişken düşünelim ve değeri 6 olsun , Java da 1 byte 8 bittir. Bu durumda 6 değeri ,bit pattern/yapı 00000110 olarak temsil edilir.

```
Button b = new Button();
```

b hakkında ne söylebiliriz peki ?

b bir reference variable / referans değişkendir ve Button objesine referansta bulunmaktadır. Bir referans değişken de bit holder/tutucudur. Yani tuttuğu bu bitler ilgili objeye nasıl ulaşacağına bilgisini taşır. Bit formatını bilememekteyiz

```
Button b = null;
```

b, hiç bir objeye referansta bulunmuyor ,anlamına gelmektedir.

Incelemeye devam edelim ve bir kaç örnek verelim, bunlara zaten yabancı değiliz.

```
int x=10; //literal assignment  
int y=x+2; //assignment with an expression  
int z=x*y; //assignment with an expression
```

Burada dikkat cekmek istedigim konu tamsayi literallerinde varsayılan tur int tir. Bir önceki konuda literal konusuna deginmistik.

```
byte b = 10; //automatically narrows literal value to a byte  
byte b = (byte) 10; // explicitly cast the int literal to a byte
```

Ilk satirdaki atama islemi gecerlidir. Burada Compiler/derleyici otomatik olarak literal int degeri byte degere cevirir. Ikinci satirda oldugu gibi explicit/acik bir sekilde byte degere cast islemi yapilmaktadir. Bu ornegimiz short degisken icin de gecerlidir. Simdi de su ornegimizi inceleyelim;

```
byte a = 5;  
byte b = 10;  
byte c = b+c; //compiler error
```

Burada 1 ve 2.satirda sorunsuzca atama islemi gerceklesecektir. Fakat 3.satirda derleme hatasi alacagiz. Peki boyle basit bir islemde  $5+10$  degerinin 15 , yani byte degerinin sinirinda olmasinda bile neden hata aldik ? Java ciddi bir programlama dilidir ve son derece titiz bir denetim soz konusudur. Bu hatanin nedeni sudur ; possible loss of precision // yani muhtemel veri kaybi.

Javada , expression/ifade tamsayi degerleri icin int e yukseltirilir. yani  $b+c$  bir expression/ifadedir dolayisiyla 15 degeri int e yukseltirilir. Daha sonrasinda bu int 15 degerini byte degere atamaya calisiyoruz, Java buna izin vermeyecektir.

Bu sorunu cozmek icin, explicitly/ acik bir sekilde cast islemi yapmamiz gereklidir;

```
byte c = (byte)(a+b);
```

### Primitive Casting

Casting; bir primitive/ilkel turden baska bir ilkel ture donusturme islemidir. Onceki yazılarda bahsettim burada biraz daha detaylica inceleyecegiz.

Casting islemi explicit/acik veya implicit/kapali olabilir.

*implicit cast* ornegi;

```
int a=100;  
long b=a; // implicit // kapali cast islemi
```

burada a degiskenini b degiskenine direkt olarak atadik. Bir int turunde degiskeni long turunde degiskene direkt olarak atayabiliyoruz. Benzer sekilde bir byte degiskeni short, int, long degiskene , bir short degiskeni int ve long degiskene direkt olarak atayabiliyoruz.

*Explicit cast ornegi;*

```
float f = 124.232f;  
long b = (int) f;
```

burada float turunde bir degisken acik bir sekilde / explicit cast islemine tabi tutuluyor. Eger parantez icerisinde (int) ifadesini yazmazsa Java bize kizacak ve derleme zamani hatasi verecektir. Bu ifade ile derleyiciye sunu diyoruz bu cast islemi gerceklestir ve veri kaybini onayliyorum, yani b degerimiz 124 olacaktir, virgulden sonraki kisim atilacaktir.

tamsayi primitive/ilkel tipleri (byte, short, int, long) degiskenler explicit/kapali bir sekilde double veya float degiskene atanabilirler.

```
double d=100L;  
int a=10;  
float f=a;  
long c=1000;  
float g=c;
```

tamsayi primivite degiskenlere noktali/float sayilar atayamayiz.

```
int a=10.5;  
short s=20.2;  
byte b=30.0;
```

### Assigning Float Numbers

Daha once belirtigim gibi Java da noktali kayan sayilar / floating-point implicity/kapali sekilde double dir, float degildir ! Yani 30.5 literali float degil double dir.

```
//float f=30.5; //compiler error.  
float f2 = (float) 30.5;  
float f3 = 30.5f;  
float f4 = 30.5F;  
float f5=30;
```

Bir kac ek bilgi daha verelim ;

```
byte b = 128; //compiler error
```

Bu atama islemi hata verecektir. Neden peki ? Hatirlayacagimiz gibi Java da bir byte degiskeni 8 bittir ve 256 deger tutabilir. Bir byte degiskeni -128 ve 127 arasında bir deger tutabilir. 128 degeri bu sinirin disindadir ve Java bu duruma kizacaktir, derleme zamaninda bize hata verecektir.

Su ornegimizi inceleyelim;  
Java da bu atama ifadesi sorunsuzca calisacaktir.

```
byte b = 3;  
b += 5;
```

Su ifade de cast islemi yapmamiz gerekir;

```
byte b = 3;  
//b=b+5; compiler error  
b =(byte)(b+5);
```

Degiskenler ve cesitli atamalarla ilgili biraz kod yazmak noluyor diye gormek faydalı olacaktir.

## Pure Java – 24 Variable Scope

[Levent Erguder](#) 17 August 2013 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimizda Java'da degiskenlerin kapsam alanlarından(variable scope) yani bir degisken Java kodumuz icerisinde nerede biliniyor nerede gecerli ya da nerede bilinmiyor ve gecersiz(compile error/derleme hatasi) olur bunu inceleyecegiz. Bir onceki yazida Variable/degisken nedir sorusuna da cevap aramistik.

Ornek sinifimiz üzerinde inceleyelim ;

CoolClass.java

```
public class CoolClass {  
  
    static int count = 20; // static variable  
    String s; // instance variable  
    int i; // instance variable
```

```

{
    s = "Initialization block";
    int block = 250;
}

public CoolClass() { // constructor // yapilandirici
    i += 10;
    int j = 100;
}

void coolMethod() {
    int k = 25; // local variable

    for (int t = 0; t < 10; t++)
        k += k + t;

}
}

```

- count , static degiskendir.
- i , instance degiskendir.
- k, local degiskendir.
- t, block degisenidir yani local degiskendir.
- block, init block degisenidir.
- j, constructor/yapilandirici degisenidir yani local degiskendir.

Simdi de bu degisenlerin yasam alanlarini inceleyelim.

- static degisenler en genis yasam alanina sahiptirler(variable scope). static degisenler , sinif JVM( Java Virtual Machine) tarafindan yuklendiginde (load) olusturulur (create) ve ilgili class JVM de kaldigi surece static degisenler de yasayacaktir.
- static degisenlerden sonra yasam alani genisligi acisindan instance degisenler gelmektedir. Bir instance variable , instance olusturuldugunda (new) yaratilir ve instance kaldirilina kadar gecerli olacaktir.
- Daha sonrasinda ise local degisenler gelmektedir. local degisenler ilgili metotta gecerli olacaktir ve bu metot stackte oldugu surece hayatta olacaktir. Aslinda yine de hayatta olabilirler fakat kapsam disi olacaklardir (out of scope)

- block degiskenleri sadece ilgili blockta gecerli olacaktir. { } disinda bu degiskenlerin bir yasam alani yoktur.

Simdi de daha iyi anlayabilmek icin bir kac orne uzerinde incelemelerde bulunalim ve genel hatalari gozden gecirelim;

- static bir metottan non-static degiskenlere (instance) ulasmak.

```
public class ScopeError {
    int a;
    public static void main(String[] args) {
        System.out.println(a);
    }
}
```

Onceki yazilarda da belirtigimiz gibi, static bir metottan non-static bir degisken/metota direkt olarak ulasamayiz.Cunku a, bir instance variabledir ve biz instance kullanmadan ulasmaya calisiyoruz.

- local bir degiskene farkli bir metottan erismek

```
public class ScopeError {
    public static void main(String[] args) {
        ScopeError se = new ScopeError();
        se.method1();
    }

    void method1() {
        int y=10;
        System.out.println(y);
        y++;
        method2();
    }

    void method2(){
        //y++; // derleme hatasi
    }
}
```

- Bir block degiskenine , block sonlantiktan sonra erismeye calismak.

```

void method3() {
    for (int j = 0; j < 5; j++) {
        System.out.println(j);
    }

    // System.out.println(i); //compile error //derleme hatasi

    for (int j = 0; j < 5; j++)
        System.out.println(j);
    // System.out.println(j); //compile error //derleme hatasi

    int k=0;
    for (int j = 0; j < 5; j++)
        k++;

    for (int j = 0; j < 5; j++)
        int z=10; //compile error

}

```

1. for dongusunde j degiskenini for block'u disinda kullanmaya calisirken derleme zaman hatasi aldik. Peki 2.for dogunsunde neden hata aldik ? Dikkat ederseniz { } kullanmadik , bu nedenle for dongusunden sonra sadece bir ifade gecerli olacaktir. (statements)

3.for dongusu sorunsuz calisirken 4.for donsunde yine hata aldik ! Dikkat ederseniz bir for dongusunden sonra {} kullanilmazsa tek satir degil tek bir statement gecerli olacaktir. int z=10; bir expressiondir. Daha fazla detay icin ;

<http://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html>

## Pure Java – 25 Initialize Instance Variables

[Levent Erguder](#) 30 August 2013 [Java SE](#)

Merhaba Arkadaslar, bu yazimiz Java'da instance variable 'a initialize islemi/deger atamasi uzerine olacaktir.

Java , bir degisken(instance variable) tanimladigimizda 2 secenek sunar ya oldugu gibi birakiriz/uninitialized ya da deger atamasinda bulunuruz.

Degisken kapsam alaninina gore (instance variable , local variable ... ) farkli durumlar ortaya cikmaktadır. Bir onceki yazida Degisken kapsam alani ( Variable Scope) konusunu incelemistik.

### Primitive && Object Type Instance Variable

Instance Variable(member variables) , hatirlayacagimiz gibi sinif seviyesindeki degiskenlerdi. Yani bu instance variable'lar bir metot, yapislandirici, kod blogu içerisinde tanimlanamaz.

Instance Variable'lara varsayılan olarak deger atamasi yapilir. Asagidaki tabloda varsayılan olarak atanen bu degerleri gorebiliriz.

#### Primitive Instance Variable

Yukarıda primitive tiplerin listesi yer almaktadır. Daha oncesinde de primitive/ilkel tipler nelerde incelemistik.Burada ornek üzerinde deger atamayan primitive instance variable/degiskenleri garemos.

#### Primitive.java

```
public class Primitive {  
  
    int insVariable;  
    byte insVariable2;  
    boolean insVariable3;  
    double insVariable4;  
    float insVariable5;  
    char insVariable6;  
  
    @Override  
    public String toString() {  
        return "Primitive [insVariable=" + insVariable + ", insVariable2=" +  
               insVariable2 + ", insVariable3=" + insVariable3 + ",  
               insVariable4=" + insVariable4 + ", insVariable5=" +  
               insVariable5 + ", insVariable6=" + insVariable6 + "]";  
    }  
}
```

```

    }

    public static void main(String[] args) {
        Primitive pri = new Primitive();

        System.out.println(pri);
    }

//Primitive [insVariable=0, insVariable2=0, insVariable3=false, insVariable4=0.0, insVariable5=0.0, insVariable6=
}

```

### Object Reference Instance Variable

Referans degiskenler varsayılan olarak null olarak atanacaktır.

[ObjectReference.java](#)

```

import java.awt.Button;

public class ObjectReference {

    String s;
    Button b;
    Thread t;
    Runnable r;
    ObjectReference o;
    Object obj;

    @Override
    public String toString() {
        return "ObjectReference [s=" + s + ", b=" + b + ", t=" + t + ", r=" + r
               + ", o=" + o + ", obj=" + obj + "]";
    }

    public static void main(String[] args) {
        ObjectReference obj = new ObjectReference();
        System.out.println(obj);
    }
}

```

```
    }

//ObjectReference [s=null, b=null, t=null, r=null, o=null, obj=null]

}
```

Simdi de su ornegi inceleyelim;

#### NPE.java

```
public class NPE {

    String str;

    public static void main(String[] args) {
        NPE npe = new NPE();

        System.out.println(npe.str);
        System.out.println(npe.str.toLowerCase());
    }
}
```

String degiskenimize deger atamadigimiz icin null olacaktir, fakat toLowerCase() metodunu cagirdigimizda meshur NullPointerException hatasini calisma zamaninda/runtime alacagiz.

#### Array Instance Variable

Illerleyen yazilarda array konusuna tekrar gelecegiz ama burada ilgili kismi inceleyelim. Java da bir array objedir. Bu nedenle sadece declare/tanimlanan ve acik bir sekilde deger atanmasi yapilmayan /explicitly initialize diziler/array null degerine sahip olacaklardir.

#### CoolArray.java

```
import java.util.Arrays;

public class CoolArray {

    int[] coolVar;
    double[] coolVar2;
    String[] coolVar3;
```

```

@Override
public String toString() {
    return "CoolArray [coolVar=" + Arrays.toString(coolVar) + ", coolVar2=" +
        Arrays.toString(coolVar2) + ", coolVar3=" +
        + Arrays.toString(coolVar3) + "]";
}

public static void main(String[] args) {

    CoolArray ca = new CoolArray();
    System.out.println(ca);
}

//CoolArray [coolVar=null, coolVar2=null, coolVar3=null]
}

```

Tüm diziler null değere sahip olacaklardır. Peki initialize işlemini gerçekleştirirsek dizi içerisindeki elemanlar ne olacak ?

### CoolArray.java

```

import java.util.Arrays;

public class CoolArray {

    int[] coolVar = new int[2];
    double[] coolVar2 = new double[3];
    String[] coolVar3 = new String[4];

    @Override
    public String toString() {
        return "CoolArray [coolVar=" + Arrays.toString(coolVar) + ", coolVar2=" +
            Arrays.toString(coolVar2) + ", coolVar3=" +
            + Arrays.toString(coolVar3) + "]";
    }

    public static void main(String[] args) {
}

```

```

        CoolArray ca = new CoolArray();
        System.out.println(ca);
    }
//CoolArray [coolVar=[0, 0], coolVar2=[0.0, 0.0, 0.0], coolVar3=[null, null, null, null]]
}

```

Ciktiyi inceledigimizde , dizimiz hangi turdense (int , double, String) elemanlarida otomatik olarak ilgili turun sahip oldugu degere sahip olmaktadır (int 0 , double 0.0 , String null )

## Pure Java – 26 Initialize Local Variables

Levent Erguder 31 August 2013 Java SE

Merhaba Arkadaslar,

Bu yazimda Java da Local diger adlariyla Stack veya Automatic Variable olarak adlandirilan degiskennelere deger atanmasi/initialize islemi ile ilgili bilgiler aktaracagim.

### Local Primitive

Test sinifimizda main metodumuzda 2 adet degisken tanimlayalim. Asagidaki kod herhangi bir derleme hatasi/compile error icermez.

```

public class Test {

    public static void main(String[] args) {

        int number1;

        int number2;

    }
}

```

```
}
```

Peki bu degiskenlerin degerini ekrana yazdirmaya calisalim ;

```
public class Test {  
  
    public static void main(String[] args) {  
  
        int number1;  
  
        int number2;  
  
        System.out.println(number1); //derleme hatasi  
  
        System.out.println(number2); //derleme hatasi  
  
        System.out.println(number1+number2); //derleme hatasi  
  
    }  
  
}
```

Gordugunuz gibi derleme hatasi ile karsilastik. Peki neden ? Bir onceki yazidan hatırlayacagımız gibi “int” turunde bir instance variable/degisken icin varsayılan deger 0 di, fakat number1 ve number2 instance variable degil, local/yerel degiskendir.

Local/yerel degiskenleri kullanmadan once MUTLAKA deger atamak/initialize gerekmektedir.

```
public class Test {  
  
    public static void main(String[] args) {  
  
        int number1=5;  
  
    }  
}
```

```
int number2=10;

int number3;

System.out.println(number1);

System.out.println(number2);

System.out.println(number1+number2);

}

}
```

number1 ve number2 local/yerel degiskenlerini kullandik bu nedenle deger atamasinda bulunduk. number3 degiskenini kullanmadik bu nedenle deger atamak zorunda degiliz. Tabi kullanmadigimiz degiskeni tanimlamak gereksiz bir hareket olacaktir :)

#### Local Object Reference

Local Primitive / ilkel tipte degisken tanimlayabildigimiz gibi referans tipte degiskenlerde tanimlayabiliriz. String turunde bir local/yerel degisken tanimlayalim , unutmayalim ki String bir primitive degisken degildir !

Asagidaki kodu inceleyelim;

```
public class Test {

    public static void main(String[] args) {

        String str;
```

```
if(str==null) {  
  
    System.out.println("HelloWorld");  
  
}  
  
}  
  
}
```

Bu kod masum gozukebilir ama str==null satirinda derleme hatasi olacaktir. Neden peki ? str degiskeni instance variable/degiskeni degil local/yerel degiskendir ve varsayılan olarak bir deger atanmaz.

str degiskeni bir degeri yoktur (uninitialized) , null degildir !

null ile uninitialze ayni anlamda gelmemektedir!

str degiskenenine null degeri atadigimizda derleme hatasi kaybolacaktır, cunku str degiskenenine bir initialize islemi yapilmistir.

```
public class Test {  
  
    public static void main(String[] args) {  
  
        String str=null;  
  
        if(str==null) {  
  
            System.out.println("HelloWorld");  
  
        }  
  
    }  
  
}
```

## Local Array

Diger local/yerel degiskenler gibi local/yerel dizileride kullanmadan once deger atama islemi yapilmalidir.  
(initialize). Dizilerin elemanlari ise , dizi ister instance ister local olsun uygun varsayılan degere sahip olacaktır. (0, 0.0, false,null .. )

```
public class Test {  
  
    public static void main(String[] args) {  
  
        int [] array1;  
  
        int [] array2=new int[10];  
  
        double [] array3= new double[5];  
  
        String [] array4 = new String[5];  
  
        for(int i: array2) {  
  
            System.out.print(i + " ");  
  
        }  
  
        System.out.println();  
  
        for(double i: array3) {  
  
            System.out.print(i + " ");  
  
        }  
    }  
}
```

```
System.out.println();

for(String i: array4) {

    System.out.print(i + " ");

}

//0 0 0 0 0 0 0 0 0

//0.0 0.0 0.0 0.0 0.0

//null null null null null

}

}
```

## Pure Java – 27 Passing Variables into Methods

Levent Erguder 09 September 2013 Java SE

Merhaba Arkadaslar,

Java da *pass-by value* ve *pass-by-referance* kavramlarini bol miktarda duymusudur. Bir metodun, taniminda aldiği degiskenler(parametreler) hem primitive/ilkel hem de referans tipte olabilir.

Degisken nedir sorusuna onceki yazilarimda cevap aramistik, hatirlayacagimiz gibi degisken; bit tutucularidir.<http://www.injavawetrust.com/pure-java-23-assignment-operators/>

Primitive/ilkel tipte degiskenler icin bir degiskeni diger degiskene atamak ; icerigini/content yani *bit pattern*/bit yapisini bir degiskenden diger bir degiskene kopyalamak anlamina gelmektedir. Obje referans degiskenlerini icin de bu durum gecerlidir ! Referans degiskenen icerigi/content bir *bit pattern*/yapi/ornek dir. Yani a referans degiskenini b referans degiskene atamak , a ‘nin bit pattern’ini , b’nin bit pattern’ine kopyalamak anlamina gelmektedir.

### Obje Referans Degiskeni ile Metot Cagirma

Bir metoda , bir obje referans degiskeni gonderdigimizde , objenin kendisini degil o objenin referans degiskenini gonderiyoruz. Hatirlayacagimiz gibi bir referans degisken de bit holder/tutucudur.Yani tuttugu bu bitler ilgili objeye nasil ulasacagini bilgisini tasir.

Aslinda, ilgili objenin referans degiskenini gonderiyoruz desek de aslinda referans degiskeninin bir kopyasini gonderiyoruz ! Yani degiskenin bit pattern/kalibinin kopyasini alip gondermekteyiz. Boylece 2 tane identical/ozdes referans degiskeni olacaktir. Bu iki referans degisken de Heap'teki ayni objeye referansta bulunacaktır. Burada yeni bir obje olusmadigini belirtmek gerekir .

### Test.java

```
package passbyreference;

class Numbers {
    int num1;
    int num2;

    public Numbers(int num1, int num2) {
        super();
        this.num1 = num1;
        this.num2 = num2;
    }

    protected void addValue(Numbers n) {
        System.out.println("main method");
        n.num1=n.num1+10;
        n.num2=n.num2+20;

        System.out.println("numb1:" + n.num1);
        System.out.println("numb2:" + n.num2);
    }
}

public class Test {

    public static void main(String[] args) {

        Numbers number = new Numbers(10,20);
        System.out.println("Before Method Call");
        System.out.println("numb1:" + number.num1);
        System.out.println("numb2:" + number.num2);
    }
}
```

```

        number.addValue(number);

        System.out.println("After Method Call");
        System.out.println("numb1:" + number.num1);
        System.out.println("numb2:" + number.num2);
    }

}

```

Ornegimizi inceleyecek olursak , Numbers sinifimizda 2 tane int turunde instance degiskeni tanimladik ve Test sinifimizda bir referans degiskeni ve obje olusturduk. “number” referans degiskenimizi addValue metoduna gonderdik, burada num1 ve num2 alanlarini degistirdik (20 ve 40 yaptik) sonrasinda ise main metodunda “number” referans degiskenimizin tuttugu objenin num1 ve num2 alanlarini yazmak istedik ve biraz once degistirdigimiz 20 ve 40 degerleri olarak degistigini gorduk. Cunku ortada 1 obje var ve 2 referans degiskeni de bu objeye referansta bulunmaktadir boylece ikisi de degisiklik yapabilmektedir.

Aslinda Java ister primitive/ilkel ister obje referans tipinde degiskeni olsun *pass-by-value* kuralina gore calismaktadir. *Pass-by-value* demek *Pass-by-variable-value* demektir aslinda *pass-by-copy-of-the variable* demektir !

Ornegin int turunde bir degiskenen degeri 5 olsun, aslinda 5 degerini ifade eden bitlerin kopyasini metoda gondermekteyiz. Eger obje referans turunde bir degiskeni gonderiyorsak bu referans degiskeni ifade eden bitlerin kopyasini metoda gondermekteyiz.

*(pass-by-copy-of-the-bits-in-the-variable)*

Bir diger detay noktayı daha inceleyelim ; yukaridaki ornekte degiskenen degerini degistirdik fakat cagrilan metot icerisinde orijinal referans degiskenen baska bir objeye referansta bulunmasini ya da null olmasini saglayamayiz.

## Test2.java

```

package passbyreference;

class Foo {
    String name = "levent";

    void bar() {
        Foo f = new Foo();
        System.out.println(f.name);
        doStuff(f);
    }
}

```

```

        System.out.println(f.name);
    }

    void doStuff(Foo g) {
        System.out.println(g.name);
        g.name = "erguder";
        System.out.println(g.name);
        g = new Foo();
        g = null;

    }
}

public class Test2 {
    public static void main(String[] args) {

        Foo f = new Foo();
        f.bar();

    }
/*
 * levent
 * levent
 * erguder
 * erguder
 */
}

```

g degiskeneninin yeni bir objeye referans olmasini saglamak (g = new Foo(); ) , f degiskeneninin ayni objeye referans olmasini saglamaz ! Benzer sekilde g degiskenini null yapmak , yani hic bir objeye referansta bulunma demek f degiskenini etkilemez.

Bunun nedeni doStuff metodu referans degiskeninin kopyasina sahip , aslina degil ! doStuff metodu Foo objesine giden yolu bilmekte fakat f referans degiskenine giden yolin bilgsini bilmemektedir. Bu nedenle doStuff metodu objenin durumunu/state degistirebilirken , f degiskeninin farkli bir objeye referans olmasini saglayamaz.

### Primitive/Illkel Tipte Degisken Ile Metot Cagirma

Bir primitive/illkel tipte degiskeni metoda gonderdigimizde, herkesin bildigi anladam *pass-by-value* kurali islemektedir yanip *pass-by-copy-of-the-bits-in-the-variable*.

Ornegimizi inceleyecekl olursak , num1 degiskenimize 10 degerini atiyoruz ve modify metodunu cagirip degiskenin degerini degistiriyoruz (15) fakat main metoduna tekrar geri geldigimizde ve num1 degerimizi yazdirdigimizda aslinda degiskenimizin degismediğini gorururuz.

#### Test3.java

```
public class Test3 {  
    public static void main(String[] args) {  
  
        int num1=10;  
        System.out.println(num1);  
        modify(num1);  
        System.out.println(num1);  
    }  
    static void modify(int num) {  
        num = num+5;  
        System.out.println(num);  
    }  
    /*  
     *  
     * 10  
     * 15  
     * 10  
     */  
}
```

## Pure Java – 28 Array Declaration

[Levent Erguder](#) 16 September 2013 [Java SE](#)

Merhaba Arkadaslar,

Bir kac yazi boyunca Java'da Array/Dizi konusundan bahsedecegim. Aslinda programlama ile biraz ugrasan herkes diziler konusunda az cok fikir sahibi olmustur. Zaten buraya kadar da dizileri ara ara kullandik. Burada biraz daha detaylica inceleyip once noktalarini gorecegiz.

Genel olarak 3 nokta uzerinde duracagim;

- Bir array/dizi referans degiskeni nasil tanimlariz (declare)

- Bir array/dizi objesi nasıl oluştururuz (construct)
- Bir array/diziye eleman eklemek, doldurmak (initialize)

Burada bir “declare” kısmını inceleyeceğiz

Java'da tüm Array/Diziler her seyden önce bir objedir ! Diziler, aynı tipte primitive/ilkel veya referans tipte değişken tutabilir fakat dizinin kendisi her zaman objedir.

Dizileri iki şekilde tanımlarız (declare) ;

```
int[] coolArray; // önerilen ! recommended  
int coolArray[]; // derleme hatası vermez , önerilmez
```

İki tanım sekilde derleme hatası vermez , ilk tanım önerilen yöntemdir. Yani koseli parantezler değişken isminden önce koymalıdır.

```
String[][][] cool; //onerilen  
String[] notCool[]; //onerilmez
```

Tanımlama (declare) ifadesinde ,dizinin boyutunu belirtmek derleme zamanı hatasına yol acar. (compile error) ;

```
int [10] errorDeclaration; //compile error
```

Bunun nedeni , JVM , bir dizi objesi yaratmadan önce bellekte yer ayırma işlemi yapmayacaktır (allocate)

## Pure Java – 29 Array Construction

[Levent Erguder](#) 22 September 2013 [Java SE](#)

Merhaba Arkadaşlar,

Bir önceki yazida Array Declaration konusunu inceledik. Bu yazida Array Construction konusunu inceleyeceğiz.

“Array Construction” ifadesinden kasıt , bir array/dizi objesini Heap’te yaratmak/olusturmak/create anlamına gelmektedir. Bir array/dizi objesi yaratılmadan önce Java, Heap’te bu obje için ne kadar yer ayrılmazı gerektiğini (allocate) bilmelidir. Bu nedenle dizinin boyutu, oluşturulma zamanında(creation time) belirli olmalıdır.

### Construction One-Dimensional Array ( Tek Boyutlu Dizi Oluşturma)

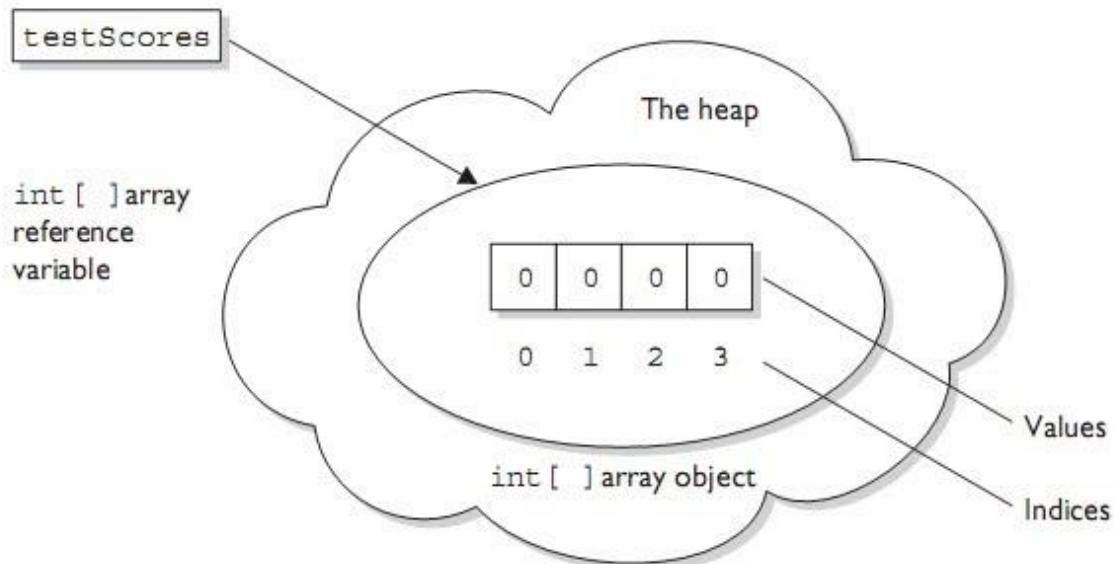
Yeni bir dizi oluşturmak için new anahtar kelimesini kullanırız.

```
int[] testScores; //declare bildiri/tanitim islemi  
testScores = new int[4]; //construct array / dizi olusturma
```

ya da

```
int [] testScores = new int[4];
```

Bu basit dizi olusturma islemini hemen hemen herkes bilmektedir. Bu kodların anlamını biraz daha detaylıca inceleyeceğiz; Heap'te yeni bir dizi objesi oluşturulur ve bu dizi objesi 4 tane elaman tutmaktadır. Onceki yazınlarda incelediğimiz gibi bu elemanlar için varsayılan değer 0 olacaktır. Hatırlayacağımız gibi tüm diziler objedir. Bu nedenle referans tipindeki testScores değişkeni Heap'teki bu objeyi göstermektedir.(refer)



### A one-dimensional array on the Heap

Yukarıda belirttiğim gibi, JVM, Heap'te yeterli/uygun alanı ayırmak(allocate) için dizinin boyutunu oluşturulma zamanında bilmelidir. Bu nedenle aşağıdaki kod derleme hatası verecektir.

```
int[] testScores = new int []; //compile error /derleme hatası
```

Constructing Multidimensional Array ( Çok Boyutlu Dizi Oluşturma)

Cok Boyutlu diziler, dizilerin dizisidir. (arrays of array). Ne demek istedik dikkatlice inceleyelim;

```
int [][] myArray = new int[3][];
myArray[0] = new int[2];
myArray[0][0] = 6;
```

```
myArray[0][1] = 7;  
myArray[1] = new int[3];  
myArray[1][0] = 9;  
myArray[1][1] = 8;  
myArray[1][2] = 5;
```

Cok boyutlu diziler de objedir. myArray referans degiskeni Heap'te 2 boyutlu bir dizi objesine referansta bulunmaktadır/tutmaktadır. (2-D int [] [] array object). Burada dikkat ederseniz [3][] ifadesinde sadece ilk koseli parantezde sayı belirttiğin bu gecerlidir cunku JVM'in myArray referans degiskeninin boyutunu bilmesi yeterli olacaktır.

```
myArray[0] = new int[2];
```

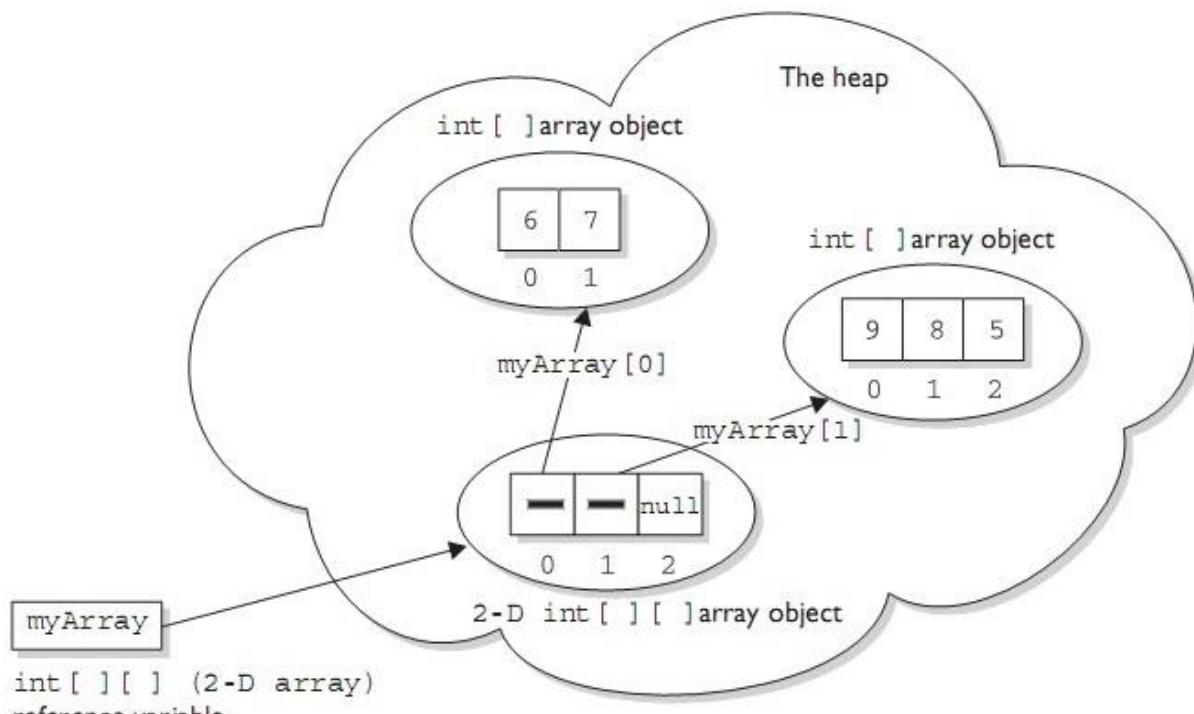
ifadesi ile yeni bir tek boyutlu dizi objesi oluşturuyoruz. (int [] array object)

```
myArray[0][0] = 6;  
myArray[0][1] = 7;
```

Oluşturduğumuz bu diziye değer atamasında bulunuyoruz, değer atamasında bulunmasaydı int tipinde bir dizinin elemanları varsayılan olarak 0 olacaktır. Aynı işlemi diğer tek boyutlu dizi için de yapıyoruz ;

```
myArray[1] = new int[3];  
myArray[1][0] = 9;  
myArray[1][1] = 8;  
myArray[1][2] = 5;
```

2 Boyutlu, myArray dizimiz için 3. elemanı (myArray[2]) herhangi bir diziye referansta bulunmadığı için bu elemanın değeri, varsayılan değer olarak null olacaktır. Bunun nedeni myArray 2 boyutlu bir dizi referans değişkenidir ve her elemanı bir int tipinde diziye referansta bulunur. Her dizi bir obje olduğu için, myArray için varsayılan değer olarak 0 değil null olacaktır.



## A two-dimensional array on the Heap

# Pure Java - 30 Array Initialization

[Levent Erguder](#) 11 October 2013 [Java SE](#)

Merhaba Arkadaşlar,

Onceki iki yazida array declaration ve array construction konularından bahsetmistik. Bu yazida array initialization ve hem construction hem initialization işlemini beraber yaptığımız durumlari inceleyecegiz.

Array Initialization ‘ dan kasit dizinin icerisine “bir seyler” eklemek, doldurmaktir. Bu “bir seyler” hem primitive degerler olacagi gibi , objeleri tutan referans degiskenleri de olabilir.

```
Animal [] pets = new Animal[3];
```

3 tane Animal referans tipinde ve null degerli elemanimiz ve Heap’te 1 tane array objemiz olustu. Burada Animal objesi yok ! Burasi onemli bir nokta.

```
pets[0]= new Animal();
pets[1]= new Animal();
pets[2]= new Animal();
```

Bu kod parcasinda 3 tane Animal objesi olusturuyoruz ve ilgili indekslere atama islemini gerceklestiriyoruz.

```
pets[3]= new Animal(); //ArrayIndexOutOfBoundsException
```

```
int[] x = new int [5];  
x[5] =10; //ArrayIndexOutOfBoundsException  
  
int[] z = new int [2];  
int y=-3;  
z[y]= 4; //ArrayIndexOutOfBoundsException
```

Yukarıdaki örnekler çalışma zamanında (runtime) , `ArrayIndexOutOfBoundsException` hatası verecektir. Dizi indekslerinin Java'da 0 dan başladığını ve negatif değer alamayacağını belirmekte fayda var.

Donguler ile ilk değer atama işlemini (Initialization) gerçekleştirebiliriz.

Array objeleri `length` adında public bir değişkene sahiptir. `length` değişkeni , array'in kaç tane eleman tuttuğunu bize söyleyebilir. Bu elemanların initialize edildiğinden bilgisini bize vermez.

```
int[] number = new int[5];  
  
System.out.println(number.length);  
for (int i = 0; i < number.length; i++) {  
    number[i] = i * 2;  
}  
System.out.println(number.length);
```

Declare, construct ve initialize işlemini tek statement(ifade) ile de yapabiliriz.

```
int [] ages = {10,15,20};  
  
// ages adında "int" tipinde bir array referans değişkeni tanımladık  
// 3 elemana sahip bir array oluşturduk  
// array'in elemanlarını 5,10,15 ile doldurduk  
// Heap'te oluşan array objesini ages değişkenine atadık.  
  
int ages;  
ages = {10,15,20}; //compile error
```

Benzer şekilde çok boyutlu diziler için de geçerlidir ,

```
int[][] scores = { { 2, 4, 5 }, { 10, 20, 3, 9 }, { 4 }, {} };
```

Yukarıdaki kod Heapse 5 tane obje olusturur. Bunlardan ilki ,scores degiskeneninin tuttugu cokboyutlu dizinin objesidir. (array of int arrays) . Her bir { } yeni bir array'i temsil eder, her array bir obje oldugu icin dolayisiyla 4 adet objede burada olusur.

Bir diger yontem olarak *anonymous array* initialization(ilk deger atama) yapisi kullanilabilir ;

```
int [] ages2 ;
ages2 = new int[]{20,30,40,50};
```

*anonymous array* olustururken , dizi boyutu belirmek derleme hatasina yol acar.

```
int [] ages2 ;
ages2 = new int[4]{20,30,40,50}; //compile error
```

## Pure Java – 31 Legal Array Element Assignments

[Levent Erguder](#) 29 October 2013 [Java SE](#)

Merhaba Arkadaslar ,

Bu yazimizda , array/diziler konusuna devam edecegiz ve diziler konusunu burada sonlandiracagiz ( en azindan



simdilik ). Burada dizi elamanlarina deger atama ve dizilerin birine atanmasini konusuna biraz daha yakindan bakacagiz.

Bir dizinin farkli tipte elemanlari olsa bile, sadece tek bir tipte tanimlanabilir. ( int [] ,String[] , Animal[] )

int tipinde bir dizi elemani, byte, short veya char turunde olabilir , long , float ya da double turunde olamaz.

```
public class Test1 {

    public static void main(String[] args) {
        int[] coolArray = new int[6];

        byte b = 10;
        char c = 'c';
        short s = 20;
    }
}
```

```

long l = 50;
float f = 10;
double d = 100;

coolArray[0] = b;
coolArray[1] = c;
coolArray[2] = s;
// coolArray[3] = l; // compiler error
// coolArray[4] = f //compiler error
//coolArray[5] = d; //compiler error

for (int i : coolArray) {
    System.out.print(i + " ");
}
}
}

```

Referans tipte diziler icin, IS-A kuralina uygun olacak sekilde eleman atama islemi gerceklestirilebilir.

```

interface Vehicle {

}

class Car implements Vehicle {

}

class Ford extends Car {

}

class Fiat extends Car {

}

public class Test2 {

    public static void main(String[] args) {

```

```

Vehicle[] vh = { new Car(), new Ford(), new Fiat() };

// Car IS-A Vehicle

// Ford IS-A Vehicle

// Fiat IS-A Vehicle


Car[] car = { new Car(), new Ford(), new Fiat() };

// Ford IS-A Car

// Fiat IS-A Car


Ford[] ford = { new Ford() };

// Bu diziye new Fiat() veya new Car() eklenemez , cunku IS-A kuralina
// uygun degil.

}

}

```

Dizi referanslarının birbirine atanmasını inceleyeceğiz, bir int tipinde diziye byte, short, byte turundaki değişken atanabilirken, int tipinde diziye short ya da byte tipinde bir dizi atanamaz.

```

public class Test3 {

    public static void main(String[] args) {

        int[] intArray;

        int[] intArray2 = new int[5];

        byte[] byteArray = new byte[10];

        short[] shortArray = new short[10];


        intArray = intArray2;

        // intArray = byteArray; //compile error
        //byteArray = shortArray; //compile error
    }
}

```

Referans tipte diziler için IS-A kuralına uygun olarak değişken atama işlemi gerçekleştirilebilir;

```

Vehicle[] vehicle;
Car[] cars;
Fiat[] fiatArray = new Fiat[5];
Ford[] fordArray = new Ford[10];

vehicle = fiatArray; //Fiat IS-A Vehicle
vehicle = fordArray; // Ford IS-A Vehicle

cars = fiatArray; // Fiat IS-A Car
cars = fordArray; // Ford IS-A Car

// fiatArray = fordArray; // Ford IS-A Car kuralina uygun degil

```

Cok boyutlu diziler ile tek boyutlu diziler arasindaki ornek atama islemlerine bakalim ;

```

public class Test5 {

    public static void main(String[] args) {
        int[] intArray;
        int[][] intMultiArray = new int[5][];
        // intArray = intMultiArray; // compile error
        // 2 boyutlu dizi tek boyutlu diziye atanamaz

        // intMultiArray =intArray;
        // Tek boyutlu dizi 2 boyutlu diziye atanamaz.

        intArray = intMultiArray[0];
        //intMultiArray[0]; tek boyutlu bir int dizidir, ve intArray'e atanabilir.

        int[] intArray2 = intMultiArray[1];
    }
}

```

# Pure Java - 32 Initialization Blocks

[Levent Erguder](#) 30 October 2013 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda Java'da bulunan initialization(ilk kullanima hazırlama) blocks konusundan bahsedecegim. Java da 2 turlu initialization block vardır.

- static initialization block
- instance initialization block

Bir sinif yuklediginde ilk olarak ve *sadece* 1 kez static initialization block calisir.

Instance olusturuldu *her defasinda* instance initialization block calisir.

Basit bir ornek yapalim :

```
public class InititialTest {  
  
    static {  
        System.out.println("Static Initialization Block");  
    }  
  
    {  
        System.out.println("Instance Initialization Block");  
    }  
  
    public static void main(String[] args) {  
  
        System.out.println("Main -1 ");  
        InititialTest initial = new InititialTest();  
        InititialTest initial2 = new InititialTest();  
  
    }  
}
```

Bu ornekte ilk olarak ne calisir ? Main-1 den once static initialization block calisacaktir ! Daha sonrasinda 2 instance olusturdugumuz icin 2 defa instance initialization block calisacaktir. static initialization block ise sadece bir kez calisir.

## Static Initialization Block

Main -1

Instance Initialization Block

Instance Initialization Block

Burada sorun yoksa eger isi bir adim daha karmasiklastiralim;

Peki birden fazla static initialization block veya instance initialization block ve yapislandirici varsa hangi sirada calisacaklar ?

```
public class InitializationTest2 {  
  
    InitializationTest2() {  
        System.out.println("Default Constructor");  
    }  
  
    static {  
        System.out.println("1 - Static Block");  
    }  
  
    {  
        System.out.println("1 - Instance Block");  
    }  
  
    static {  
        System.out.println("2 - Static Block");  
    }  
  
    {  
        System.out.println("2 - Instance Block");  
    }  
  
    public static void main(String[] args) {  
        System.out.println("Main - 1");  
  
        InitializationTest2 init = new InitializationTest2();  
    }  
}
```

```
        InitializationTest2 init2 = new InitializationTest2();  
    }  
}
```

Cıktımız su sekilde olacaktır ;

```
1 - Static Block  
2 - Static Block  
Main - 1  
1 - Instance Block  
2 - Instance Block  
Default Constructor  
1 - Instance Block  
2 - Instance Block  
Default Constructor
```

İlk önce static initialization block'lar çalışacaktır. Dikkat ederseniz ilk olarak 1. static block çalışacaktır. Sınıf yüklenikten sonra main metodu çalışacaktır. Daha sonrasında 2 tane instance oluşturmak istiyoruz. Dikkat ederseniz instance oluştururken yapılandırıcıdan önce instance initialization block'lar çalışacaktır. Yine static initialization block'ta olduğu gibi ilk sırada olan block ilk olarak çalışacaktır.

Block'larda tanımlayabileceğimiz degiskenlere dair bir karar örnek ;

```
public class InitializationTest3 {  
  
    final static int var0;  
  
    // final static int var5; //compiler error  
  
    // final static degisken tanimladigin yerde ya da static blockta ilk deger  
    // atanmasi yapilmalidir  
  
    static int var7;  
    static {  
        int var1 = 20;  
        // static int var2=30; //static degisken tanimlanamaz.  
        final int var3 = 40;  
        var0 = 40;
```

```

        var7 = 10;
    }

    {
        int var6 = 20;
        var7 = 30;

    }

public static void main(String[] args) {
    System.out.println(var0);
    System.out.println(InitializationTest3.var0);

    // System.out.println(var1); //compile error
    // System.out.println(var3); /compile error
    // System.out.println(var6); //compile error
}
}

```

Buraya kadar her sey tamamsa olumcul vurusu yapalim isin icine kalitimi da katalim;

```

class A {

    {
        System.out.println("1.) A Instance Init Block");
    }

    static {
        System.out.println("1.) A static init block");
    }

    A0 {
        System.out.println("A - No-Arg Constructor");
    }
}

```

```
{  
    System.out.println("2.) A Instance Init Block");  
}  
  
}  
  
class B extends A {  
  
    B() {  
        System.out.println("B - No-Arg Constructor");  
    }  
  
    static {  
        System.out.println("1.) B static init block");  
    }  
  
    {  
        System.out.println("1.) B Instance Init Block");  
    }  
  
}  
  
class C extends B {  
  
    static {  
        System.out.println("1.) C static init block");  
    }  
  
    C() {  
        this(10);  
        System.out.println("C - No-Arg Constructor");  
    }  
  
    {
```

```

        System.out.println("1.) C Instance Init Block");

    }

    C(int c) {
        System.out.println(c);
    }

    {

        System.out.println("2.) C Instance Init Block");
    }

    static {
        System.out.println("2.) C static init block");
    }

    public static void main(String[] args) {
        System.out.println("1)C main method");
        C c = new C();
        System.out.println("2)C main method");

        D d = new D();
        D d2 = new D();
    }
}

class D {
    static {
        System.out.println("1.) D static init block");
    }

    {
        System.out.println("2.) D Instance Init Block");
    }
}

```

```
}
```

1.) A static init block

1.) B static init block

1.) C static init block

2.) C static init block

1)C main method

1.) A Instance Init Block

2.) A Instance Init Block

A - No-Arg Constructor

1.) B Instance Init Block

B - No-Arg Constructor

1.) C Instance Init Block

2.) C Instance Init Block

10

C - No-Arg Constructor

2)C main method

1.) D static init block

2.) D Instance Init Block

2.) D Instance Init Block

C sinifi B sinifini,B sinifi da A sinifini kalitmistir/extends.

Unutmayalim , static initialization block'lar bir kez ve sinif ilk yuklendiginde calisirlar. main metodunda "1)C main method" ciktisinden once C sinifinda bulunan static init block'lar calisacaktir! Bundan da once , C sinifi B sinifini , B sinifi A sinifini kalittigi icin once hiyerarsik olarak en tepedeki sinifin static init block'u calisacaktir. Burada unutulmamasi gereken bir diger nokta da bir sinif birden fazla static init block varsa daha once yazilmis olan static init block once calisacaktir!

1.) A static init block

1.) B static init block

1.) C static init block

2.) C static init block

Sinifların yüklenme işlemi tamamlandı sırasında main metodunun çalışmasında !

### 1) C main method

C turunden bir instance olusturdugumuzda sirasi ile su islemler gereklesir ;

Hiyerarsinin tepesindeki A sinifina ait yapılandiricidan once instance initialization block calisacaktir.Daha sonrasinda A sinifina ait yapılandirici/constructor calisacaktir. Burada da static init block' ta oldugu gibi instance init blockta da once yazilan block once calisacaktir. Daha sonrasinda A sinifinin yapılandiricisi calisacaktir.

#### 1.) A Instance Init Block

#### 2.) A Instance Init Block

#### A - No-Arg Constructor

Ayni mantikla B sinifi da calisacaktir.

#### 1.) B Instance Init Block

#### B - No-Arg Constructor

C sinifinda da yine once instance initialization blocklar yazildigi sira ile calisacaktir, C sinifinin varsayılan/default yapılandiricinda this(10) ile diger yapılandirici cagrilmaktadir.

Dolayisiyla once 10 sonrasinda C – No-Arg Constructor yazilacaktir.

#### 1.) C Instance Init Block

#### 2.) C Instance Init Block

10

#### C - No-Arg Constructor

Daha sonrasinda main metodu devam edecektir ;

### 2)C main method calisacaktir.

Diger bir ornek olarak D sinifindan 2 tane instance olusturdugumuzda static initialization block 1 kez calisacaktir. Cunku D sinifi ilk olarak kullanildiginda yuklenecektir.

Fakat instance initialization block instance olusturuldugunda calisacaktir.

#### 1.) D static init block

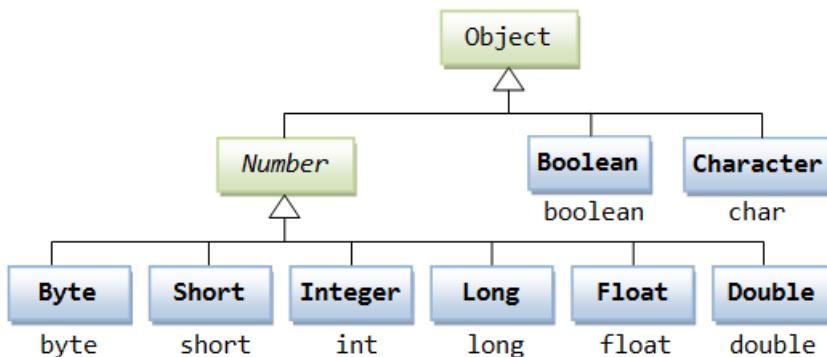
#### 2.) D Instance Init Block

## 2.) D Instance Init Block

# Pure Java - 33 Wrapper Classes - 01

[Levent Erguder](#) 30 November 2013 [Java SE](#)

Merhaba Arkadaslar, bu yazimda Java'da yer alan Wrapper siniflardan bahsedecegim. Wrapper'in turkce karsiliği sargı/sarıcı gibi anlama gelmektedir. Bu karsılığı aklimızda tutmakta fayda var. Oncelikle asagidaki tabloda Wrapper siniflar neler bunu gorebiliriz ;



Dikkat edecek olursak Java'da yer alan tum primitive/ilkel tipler icin bir Wrapper sinif yer almaktadir. Bunlardan char haric digerlerinde sadece bas harfi buyuk char da ise Character sinifi olarak karsimiza cikmaktadır. Number abstract sinifi Byte, Short, Integer gibi Wrapper siniflarin super sinifidir. Tum Wrapper siniflar Object sinifini kalitmaktadir. Dolayisiyla Wrapper turundan bir degisen primitive/ilkel tipte degildir.

Wrapper siniflarin bazi ozelliklerine bakacak olursak ;

```
public final class Integer extends Number implements Comparable<Integer> {  
    ...  
}  
  
public final class Double extends Number implements Comparable<Double> {  
    ...  
}
```

Ornegin ; Integer ve Double sinifinin tanimlanmasina goz attigimizda, final oldugunu ve Number sinifini kalittigini gormekteyiz. Ayrica Comparable arabirimini uygulamaktadir, bu arabirime ilerleyen donemlerde gelecegiz.

Boolean ve Character wrapper siniflarinin tanimlanmasina goz atalim ;

```

public final class Boolean implements java.io.Serializable, Comparable<Boolean>
{
    ...
}

public final class Character implements java.io.Serializable, Comparable<Character> {
    ...
}

```

Wrapper sınıflar final olarak tanımlanmıştır ve Comparable arabirimini uygulamaktadır.

Wrapper objelerin en önemli özelliği ise, *immutable*(degismez) olmalarıdır. Bir kere değer verildikten sonra bu değer degistirilmez yeni bir obje oluşturulur.

Wrapper sınıflarının yapılandırıcılarını inceleyeceğiz;

Primitive	Wrapper Class	Constructor Arguments
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
double	Double	double or String
float	Float	float, double, or String
int	Integer	int or String
long	Long	long or String
short	Short	short or String

Örnek bir test sınıfı yazalım ;

```

public class Test {

    public static void main(String[] args) {

        Boolean b = new Boolean("true");
        Boolean b2 = new Boolean("levent");
        Boolean b3 = new Boolean(true);
        Boolean b4 = new Boolean(false);

        System.out.println("Boolean Test");
    }
}

```

```
System.out.println(b + " " + b2 + " " + b3 + " " + b4);

// Byte bt0 = new Byte(10);

// derleme hatasi olur cunku Byte Wrapper sinifinin yapılandırıcısı

// byte veya String parametre almaktadır. Hatırlayacağımız gibi Java da

// tamsayı literalleri

// varsayılan olarak int tipindedir tir. Bu nedenle cast işlemi yapılmak

// zorundadır.

Byte bt = new Byte((byte) 10);

Byte bt2 = new Byte("10");

System.out.println("Byte Test");

System.out.println(bt + " " + bt2);

Character c = new Character('c');

System.out.println("Character Test");

System.out.println(c);

Integer i1 = new Integer(10);

Integer i2 = new Integer("10");

System.out.println("Integer Test");

System.out.println(i1 + " " + i2);

Float f1 = new Float(10);

Float f2 = new Float(10.5);

Float f3 = new Float(10.5);

Float f4 = new Float("10.5");

Float f5 = new Float("10.5f");

System.out.println("Float Test");

System.out.println(f1 + " " + f2 + " " + f3 + " " + f4 + " " + f5);

Long lo1 = new Long(10);

Long lo2 = new Long("10");
```

```

        System.out.println("Long Test");

        System.out.println(lo1 + " " + lo2);

    }

}

```



Siz burada yazilanlardan cok daha fazlasini yazin ve deneyin diger turlu akilda kalmaz Byte , Short wrapper siniflarinda karsimiza cikabilecek durumu kod icerisinde aciklama olarak verdim. Boolean sinifinin ,String yapılandiricisinda “tRuE” , “truE” gibi true kelimesinin buyuk-kucuk harfini onemsemeden yazdigimizda sonuc true donecektir, bunun disinda her String literal bize false sonucunu dondurecektir.

Wrapper siniflar ile ilgili ilk bolumu burada bitiriyorum.

## Pure Java - 34 Wrapper Classes - 02

[Levent Erguder](#) 01 December 2013 [Java SE](#)

Merhaba Arkadaslar,

Bir onceki yazida Wrapper siniflari incelemistik. Bu yazida Wrapper siniflara devam edecegiz ve wrapper siniflarda bulunan su metotlari inceleyecegiz.

- valueof()
- xxxValue()
- parseXXX()

### valueOf()

Wrapper siniflarda bulunan , static valueOf() metodunun bir kactane overloaded versiyonu vardir. valueOf() metodu String degeri veya primitive degeri Wrapper degere donusturur.

```

public class Test {

    public static void main(String[] args) {
        Integer i1 = Integer.valueOf("100"); // String literali Integer a donusturduk
        Integer i2 = Integer.valueOf(100); // primitive degeri Integer a donusturduk.
        Integer i3 = Integer.valueOf("100", 2);
        // bu overloaded versiyonda ikinci parametre radix(taban) degeridir.

        // Integer i4 = Integer.valueOf("100.5"); // j
        // java.lang.NumberFormatException firlatir.
    }
}

```

```

        System.out.println("Integer Test");

        System.out.println(i1 + " " + i2 + " " + i3);

        Float f1 = Float.valueOf("10.5");
        Float f2 = Float.valueOf("10.5f");
        Float f3 = Float.valueOf(10);
        Float f4 = Float.valueOf("10");

        System.out.println("Float Test");
        System.out.println(f1 + " " + f2 + " " + f3 + " " + f4 + " " + f4);

        Boolean b1 = Boolean.valueOf("true");
        Boolean b2 = Boolean.valueOf(true);
        Boolean b3 = Boolean.valueOf("injavawetrust");

        System.out.println("Boolean Test");
        System.out.println(b1 + " " + b2 + " " + b3);

        Long lo1 = Long.valueOf(10);
        Long lo2 = Long.valueOf("10");
        Long lo3 = Long.valueOf("100", 3);

        System.out.println("Long Test");
        System.out.println(lo1 + " " + lo2 + " " + lo3);
    }
}

```

Bazi aciklamalari kod icerisinde aciklamaya calistim. Bu konuda bol miktar ornek kod yazarak incelemelerde bulunmanizi oneririm.

#### **xxxValue()**

Wrapper tiplerden primitive tiplere donus bu metot ile yapilabilir.

```

public class Test2 {

    public static void main(String[] args) {
        Integer i1 = new Integer(100);
    }
}

```

```

byte b = i1.byteValue();

// byte b2 = i1.shortValue(); //compile error

// byte b3= i1.intValue(); //compile error


System.out.println("byte Test");

System.out.println(b);

System.out.println("short test");

short s = i1.byteValue();

short s2 = i1.shortValue();

// short s3 = i1.intValue(); // compiler error

// short s4= i1.longValue(); //compiler error

System.out.println(s + " " + s2);

System.out.println("int test");

int i2 = i1.byteValue();

int i3 = i1.shortValue();

int i4 = i1.intValue();

// int i5 = i1.longValue();

System.out.println(i2 + " " + i3 + " " + i4);

System.out.println("long test");

long lo1 = i1.byteValue();

long lo2 = i1.shortValue();

long lo3 = i1.intValue();

long lo4 = i1.longValue();

System.out.println(lo1 + " " + lo2 + " " + lo3 + " " + lo4);

System.out.println("float test");

float f1 = i1.byteValue();

float f2 = i1.shortValue();

float f3 = i1.intValue();

float f4 = i1.longValue();

// float f5=i1.doubleValue();

```

```

        System.out.println(f1+" "+f2+" "+f3+" "+f4);

    }

}

```

Benzer sekilde bol miktar test kodu yazarak incelemenizi öneririm.

### [parseXXX\(\)](#)

parseXXX metodu String degeri primitive degere donusturur.

```

public class Test3 {

    public static void main(String[] args) {

        byte b1 = Byte.parseByte("10");

        // byte b2 = Short.parseShort("10"); //compiler error

        System.out.println("byte test");
        System.out.println(b1);

        System.out.println("short test");
        short s1 = Byte.parseByte("10");
        short s2 = Short.parseShort("10");
        // short s3= Integer.parseInt("10"); //compiler error
        System.out.println(s1 + " " + s2);

        System.out.println("int test");
        int i1 = Byte.parseByte("10");
        int i2 = Short.parseShort("10");
        int i3 = Integer.parseInt("10");
        // int i4 = Long.parseLong("10");

        System.out.println(i1 + " " + i2 + " " + i3);

        System.out.println("float test");
        float f1 = Byte.parseByte("10");

```

```

        float f2 = Short.parseShort("10");
        float f3 = Integer.parseInt("10");
        float f4 = Long.parseLong("10");
        //float f5 = Double.parseDouble("10");

        System.out.println(f1 + " " + f2 + " " + f3 + " " + f4);

    }

}

```

Ozetleyecek olursak ;

```

xxxValue() Wrapper ---> primitive
parseXXX() String ---> primitive
valueOf() String ---> Wrapper

```

Oneri: Bu konuyu daha iyi anlayabilmek icin bol miktar kod yazmak faydalı olacaktır.

## Pure Java – 35 Wrapper Classes – 03

[Levent Erguder](#) 06 December 2013 [Java SE](#)

Merhaba Arkadaşlar,

Bu yazımında *Boxing* , *AutoBoxing* , *Unboxing* , *AutoUnboxing* gibi konulardan ve giriş olacak şekilde Wrapper sınıflarda `==`(equality operatoru) ve `equals` metodundan bahsedeceğim.

Boxing ve Unboxing işlemleri , Wrapper sınıfların daha elverişli bir şekilde (convenient) kullanılmasını sağlar. Java 5 öncesinde bu özellikler bulunmamaktaydı;

```

// Java 5 oncesi // Pre java 5

Integer y = new Integer(1000); // olustur

int x = y.intValue();      //ac

x++;                      // kullan

y = new Integer(x);       // tekrar olustur

System.out.println("y=" + y); // ekrana bas

```

Java 5 sonrasında ise su şekilde kullanılır ;

```

Integer y = new Integer(1000); // olustur

```

```
y++; // ac , kullan,tekrar olustur  
System.out.println("y=" + y); //ekrana bas
```

Bu kodlara zaten bircogumuz zaten asinayiz, ama kavramlari ogrenmekte ve detaylica incelemekte fayda var, Java ciddi bir dildir gereken ozeni gostermek gerekir.

primitive/ilkel tipteki (int, float,long) degiskenlerin , atama islemlerinde (assignment), ilgili Wrapper sinif tipine otomatik olarak donusturulmesi islemine **autoboxing** denir.

```
Integer i = Integer.valueOf(10); //valueOf metodunu bir onceki yazida incelemistik  
//ya da  
Integer i2 = new Integer(10);  
//ifadeleri ile Wrapper tipe donusturmek (boxing) yerine  
Integer i3 = 10;  
//seklinde kullanim ile autoboxing islemi yapilarak Wrapper tipe  
//otomatik olarak donusum yapilabilir (autoboxing)  
int primitive = 10;  
Integer autoboxing = primitive;  
Integer autoboxing2 = 100;  
  
Double autoboxing3 = 100d;  
// Double autoboxing4=100; // int Double 'a donusturulemez.
```

```
Long autoboxing5 = 100L;  
// Long autoboxing6 =100; //int Long'a donusturulemez.
```



//not bol miktarda kod yazip deneyin

Wrapper sinif tiplerindeki degiskenlerin , otomatik olarak , ilgili primitive/ilkel tipe donusturulmesine **auto-unboxing** denilir.

```
Integer i = new Integer(10); //boxing  
int autoboxing = i.intValue(); //unboxing  
// seklinde boxing ve unboxing islemleri yapilabilir  
//
```

```

Integer i2 = 10; // seklinde auto-boxing islemi gerceklestirilir

int i3=i2;    //auto-unboxing islemi

int auto_unboxing = 0;

Integer i = new Integer(10);

auto_unboxing = i;

long auto_unboxing2 = new Long(10);

double auto_unboxing3 = new Double(10);

//not: diger tipler icin de bol miktar ornek yazin

```

`==` ve `equals` kullanimi

Test sinifimiz inceleyelim;

```

public class Test {

    public static void main(String[] args) {

        Integer i = 100;
        Integer i2 = 100;

        System.out.println(i == i2); // true
        System.out.println(i == 100); // true
        System.out.println(i.equals(i2)); // true
        System.out.println(i.equals(100)); // true

    }
}

```

Muhtemelen tum sonuclar beklediginiz sonuclardi. Peki 100 yerine 1000 olarak degistirelim ;

```

public class Test {

    public static void main(String[] args) {

        Integer i = 1000;
    }
}

```

```

Integer i2 = 1000;

System.out.println(i == i2); // false
System.out.println(i == 1000); // true
System.out.println(i.equals(i2)); // true
System.out.println(i.equals(1000)); // true

}

}

```

Beklemedigimiz sekilde `i==i2` ifadesi false dondu. Farkli wrapper instance'lari icin memory/bellek kazanmak amaci Java ;

*Boolean*

*Byte*

*Character \u0000 dan \u0007f*

*Short ve Integer -128 den 127 ye kadar esitlik varsa true degeri dondurecektir.*

```

Boolean b = true;
Boolean b2 = true;

System.out.println("Boolean Test");
System.out.println(b==b2); //true
System.out.println(b.equals(b2)); //true

Short s = 100;
Short s2 = 100;
System.out.println("Short Test");
System.out.println(s==s2); //true
System.out.println(s.equals(s2)); //true

```

new ile farkli 2 instance olusturdugumuzda ise asagidaki durumlar ortaya cikmaktadir. Object sinifinda equals metodu bulunmaktadır. Wrapper siniflar equals() metodunu override etmektedir. Bu nedenle new anahtar kelimesiyle olusturulan farkli 2 instance equals() metodunda true dondurecektir. Ilerleyen yazilarda equals, hashCode gibi konulari inceleyecegiz. Simdilik bu kadar bilgi veriyorum, fazla kafa karisikligina yol



acmayalim

Object sinifindaki equals metodu

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

Integer sinifindaki override edilmis equals metodu ;

```
public boolean equals(Object obj) {  
    if (obj instanceof Integer) {  
        return value == ((Integer)obj).intValue();  
    }  
    return false;  
}
```

Test sinifimiz ;

```
public class Test2 {  
  
    public static void main(String[] args) {  
  
        Integer i = new Integer(10);  
        Integer i2 = new Integer(10);  
  
        System.out.println("Integer Test");  
        System.out.println(i==i2); //false  
        System.out.println(i.equals(i2)); //true  
        System.out.println(i==10); //true  
        System.out.println(i.equals(10)); //true  
  
        Long lo1 = new Long(10);  
        Long lo2 = new Long(10);  
  
        System.out.println("Long Test");  
        System.out.println(lo1==lo2); //false
```

```

        System.out.println(lo1.equals(lo2)); //true
        System.out.println(lo1==10); //true
        System.out.println(lo1.equals(10)); //false

        System.out.println("Long - Integer Test");
        System.out.println(lo1.equals(i)); //false
        //System.out.println(lo1==i); //compiler error

    }

}

```

Bu konuda bol miktar kod yazarak incelemeler yapmanizi öneririm.

Son olarak , Wrapper referans degiskenleri null degerde olabilir , bu nedenle meshur NullPointerException'a neden olabilir.

```

public class Test2 {

    static Integer x; // null

    public static void main(String[] args) {
        int t = 10;
        System.out.println(x + t); // null +10 ? NPE !
    }
}

```

Ilterleyen yazilarda bu konulari tekrar farkli yonlerinde inceleyecegiz.

## Pure Java - 36 More Overloading

[Levent Erguder](#) 11 December 2013 [Java SE](#)

Merhaba Arkadaslar,

Bir kac yazi boyunca Java'da Wrapper siniflari inceledik. Hatirlayacagimiz gibi Overloading konusunu onceki yazilarda incelemistik fakat daha derinlemesine inceleyecegimizi belirtmistim , bu yazida Overloading konusunu isin icine Wrapper siniflari ve Var-args degiskenlerini de katarak inceleyecegiz.

Overloading konusunun trickly olmasina neden olacak bir kac maddemiz sunlar olacak ;

- Widening
- Autoboxing
- Var-args

Oncelikle kucuk bir hatirlatma icin basit bir kod ornegimizi gorelim ;

```
public class CoolClass {  
  
    static void coolMethod(int x) {  
        System.out.print("int ");  
    }  
  
    static void coolMethod(long x) {  
        System.out.print("long ");  
    }  
  
    static void coolMethod(double x) {  
        System.out.print("double ");  
    }  
  
    public static void main(String[] args) {  
  
        byte b = 10;  
        short s = 10;  
        long l = 10;  
        float f = 5.0f;  
  
        coolMethod(b);  
        coolMethod(s);  
        coolMethod(l);  
        coolMethod(f);  
        // int int long double  
    }  
}
```

dikkat edecek olursak byte tipindeki b degiskeni ile short tipindeki s degiskeni int tipinde parametre alan metoda sorunsuzca gittiler. long tipindeki l degiskeni ise int tipinde parametre alan metoda gidemez , long tipinde degisken alan metoda gider, ayrica long tipinde bir degisken alan metot olmasaydi double turunde degisken alan metoda da gidebilir. benzer sekilde f degiskeni float turundedir ve double turunde parametre alan metoda sorunsuzca gidebilir. Tam bir eslesme durumu olmadiginda, JVM , en kucuk argumanli uygun metod ile esletirme yapar ve cagirir. not: bu noktayı anlamak acisından bol miktar deneme kod yazılmalıdır. Simdi bir baska test sinifi yazalim ve su sekilde olsun;

```
public class Test {  
    static void go(Integer x) {  
        System.out.println("Integer");  
    }  
  
    static void go(long x) {  
        System.out.println("long");  
    }  
  
    public static void main(String[] args) {  
        int i = 10;  
        go(10);  
    }  
}
```

int tipinde bir degiskenimiz var, go metodunu cagirdigimizda Integer degil long parametre alan metodumuz cagrılır. Java, boxing (int ten Integer'a donusum) yerine widening (int —>long ) islemi tercih eder. Simdi de var-arg metod iceren basit bir ornek inceleyelim ;

```
public class Test2 {  
  
    static void go(int x, int y) {  
        System.out.println("int,int");  
    }  
  
    static void go(byte... x) {  
        System.out.println("byte ...");  
    }  
}
```

```

public static void main(String[] args) {
    byte b = 10;
    go(b, b);
}

```

go(b,b) metodumuz int,int paremetresi alan metodu cagiracaktir. Bu iki ornekten sunlari cikartabiliriz;

- Widening , boxing den daha onceliklidir.
- Widening , var-args tan daha onceliklidir.

Bir diger durum olarak Boxing ve Var-args yapisini inceleyelim ;

```

public class Test3 {

    static void go(Byte x, Byte y) {
        System.out.println("Byte,Byte");
    }

    static void go(byte... x) {
        System.out.println("byte ...");
    }

    public static void main(String[] args) {
        byte b = 10;
        go(b, b);
        //Byte,Byte
    }
}

```

go(b,b) metodumuz Byte, Byte parametre alan metodu cagiracaktir. Buradan da anlayabilecegimiz gibi ;

- Boxing , Var-args tan daha onceliklidir.

#### **Widening Reference Variable**

Simdi de referans tipinde degiskenlerin widening durumlarini inceleyelim;

```

class Animal {
}

```

```

class Dog extends Animal {
}

public class Test5 {
    public static void main(String[] args) {

        Dog d = new Dog();
        go(d);

    }

    static void go(Animal a) {
        System.out.println("Animal");
    }
}

```

Buna benzer ornek kodlari daha once incelemistik, Dog IS-A Animal oldugu icin Animal parametre alan metoda sorunsuzca gidebilir.

Bu konuya ilgili bir ornek daha inceleyelim ;

```

public class Test4 {

    static void go(Number n) {
        System.out.println("Number");
    }

    static void go(short s) {
        System.out.println("short");
    }

    public static void main(String[] args) {

        Short s = 10;
    }
}

```

```

        go(s);
    }
}

```

Short tipinde degiskenimiz var, short parametreli metot mu Number parametreli mi metot cagrılır ? Widening islemi mantigi burada da uygulanacaktır ve Number parametreli metot cagrilacaktır. ( Short IS-A Number)

not : Asagidaki ornek yapisi dikkalice inceleyin ve bu konuda bol miktarda ornek kod yazin

```

Short s=10;
Short>Number>Object>short>int>long>short...>int...

```

Onceki yazilarda Wrapper siniflarin hiyerarsik bir tablosunu incelemistik, ornegin Short IS-A Integer diyemeyiz !

```

public class Test6 {
    public static void main(String[] args) {

        Short s = 10;
        // go(s); // compiler error
    }

    static void go(Integer x) {
        System.out.println("Integer");
    }
}

```

Byte , Short'a ya da Integer'a ya da Short Integer'a vb genisletilemez.(widening)

### Combining Widening and Boxing

Simdide widening ve boxing isleminin ayni anda gerektigi durumları inceleyelim ;

```

public class Test7 {

    static void go(Short s) {
        System.out.println("Short");
    }
}

```

```

public static void main(String[] args) {
    byte b = 10;
    // go(b); // once byte tipinden short tipine widening islemi uygulanmali
    // sonrasinda Short tipine boxing uygulanmali, fakat bu durum illegaldir
    // derleme hatasina yol acar
}
}

```

once byte tipinden short tipine widening islemi uygulanmali sonrasinda Short tipine boxing uygulanmali, fakat bu durum illegaldir derleme hatasina yol acar.

Pei Short tipinde parametre yerine Object tipinde parametre olarak degistirecek olrusak ;

```

public class Test8 {

    static void go(Object o) {
        Byte b2 = (Byte) o;
        System.out.println(b2);
    }

    public static void main(String[] args) {
        byte b = 10;
        go(b);
    }
}

```

Bu sefer sorunsuzca calisacaktir. byte tipinden Byte tipine Boxing islemi olacaktir daha sonrasinda Byte tipinden Object tipine widening islemi olacaktir. ( Byte Is -A Object)

not: asagidaki sekilde parametreler alan metodlari birbiriyle karsilastirip bol miktar kod yazmayi ihmali etmeyin.

```

byte b=10;
byte>short>int>long>Short>Number>Object

```

Combination with Var-args

Son olarak Var-args ile ilgili kucuk bir ornek kod daha yazalim ;

```

public class Test9 {

    static void wide_varargs(long... x) {
        System.out.println("long....");
    }

    static void box_varargs(Integer... x) {
        System.out.println("Integer...x");
    }

    public static void main(String[] args) {
        int i=10;
        wide_varargs(i);
        box_varargs(i);
    }
}

```

i int tipinde degiskenimiz , 1.metotda widening islemine 2.metotda ise boxing islemine tabi tutulur ve sorunsuzca calisir.

- Primitive widening mantik olarak en kucuk parametreli metodu kullanir.
- Bir Wrapper siniftan diger Wrapper sinifa widening islemi yapilamaz ( IS-A sartini saglamaz ! )
- Bir primitive tipteki degisken widening ten sonra boxing islemine tabi tutulamaz.( int , Long olamaz)
- Once boxing islemi sonrasinda widening islemi yapilabilir. (int , Integer, Object)
- Var-arg degiskenleri widening veya boxing islemlerine tabi tutabiliyoruz.

Yazimi burada sonlandiriyorum. Bu yaziyi bir kac defa okumak ve burada yazilanlarin bir kac kati kadar ornek kod yazmak faydalı olacaktır.

## Pure Java - 37 Garbage Collection

[Levent Erguder](#) 14 December 2013 [Java SE](#)

Merhaba Arkadaslar bu yazimda Java da Garbage Collection mekanizmasindan bahsedecegim.

Oncelikle Garbage Collection nedir sorusuna yanit arayalim , Garbage Collection memory/kaynak yonetiminin adidir. Bir bilgisayar programi calistiginda ister Java ister C, C++ veya farkli bir dil olsun memory(hafiza) kullanimi soz konusudur. C gibi dillerde pointer kullanimi, calloc() , malloc() ve free() gibi fonksiyonlarla memory yonetimi yapilirken Java da bu memory yonetimi Garbage Collector tarafindan saglanmaktadır.

Garbage Collector, otomatik Garbage Collection mekanizmasi ile memory-leak(bellek aciklari/sizintilari)'lere engel olarak Java'nin **Robust (dayanikli, guclu,direncli)** ozelligini saglamaya yardimci olur.

Ayrıca C gibi dillerde olan free() fonksiyonu yerine, Javada Garbage Collector bu isi uslendigi icin bir nevi pointer mekanizmasından kurtulmamiza destek saglar. Bu da Java'nin **Simple(Basit)** ozelligini saglamaktadir. Java'nin ozelliklerine dair Akin Kaldiroglu hocamizin yazisini ve ingilizce kaynak olarak ilgili kaynagi inceleyebilirsiniz;

<http://www.javaturk.org/?p=48>

<http://web.cs.wpi.edu/~cs525e/s01/java/features.html>

Bir Java programi calistiginda JVM(Java Virtual Machine) , Isletim Sisteminden, memory(hafiza) temin eder. Bu alana Java Heap Memory ya da kisaca heap deriz. Hatirlayacagimiz gibi Java da objeler ve instance variable'lar Heap'te mutlu mutlu yasarlar.

Garbage Collector calistiginda amaci Heap'te bulunan ve ulasilanmayan(unreachable bir nevi death) objeleri Heapten silmektiir ve bu alani yeni objeler icin hazirlamaktir.

Yazimin devaminda Garbage Collector yerine GC kisaltmasini kullanacagim. Peki aklimiza su soru gelmektedir GC'ler ne zaman nasil cagrilar ?

GC , JVM (Java Virtual Machine)'in kontrolu altindadir. Dolayisiyla GC'nin ne zaman calisacagi JVM'in kontrolundedir. Bir Java programindan , JVM'e GC'yi calistirmasi icin istekte bulunabiliriz fakat istegimizin kabul edilecegine dair bir **garanti yoktur!**

Java'da basit bir HelloWorld programi bile yaysak, yani bir main metodu kullansak en azindan bir Thread'in calismasina neden oluruz.Ilerleyen yazilarda Thread konusunu isleyecegiz , simdilik burada Thread'in alive veya dead oldugu duruma gore konusalim. Her Thread'in kendi bir yasam dongusu vardir(lifecycle) ve kendi Stack alani vardir.

Bir objeye, yasayan (alive) hic bir Thread ulasmiyorsa , bu obje GC tarafindan yok edilmeye uygundur. Bununla birlikte bir objeye hic bir canli(alive) thread ulasmamasina ragmen ve bu objenin silinmesi isleminin uygun olmasina ragmen bir nedenle bu silme islemi gerceklesmeyebilir. Bu nedenle burada da bir **garanti yoktur!** Simdi de ornek kodlar yazalim ve objelerin hangi durumlarda garbage collection icin uygun oldugunu incelemeye calisalim;

#### Nulling a Reference(Referans degiskene null atama islemi)

Bir objeye ulasabilen bir referans degiskeni yoksa, bu objeye ne olacagini pek onemi yoktur. Yani bu durumda garbage collection icin uygun hale gelir.

```
public class GarbageTest {  
  
    public static void main(String[] args) {  
        StringBuffer sb = new StringBuffer("Levent");  
        System.out.println(sb);  
        // StringBuffer objesi garbage collection icin uygun degil!
```

```

        sb = null;
        // Simdi ise garbage collection icin uygun hale gelir.
    }
}

```

#### Reassigning a Reference Variable( Referans degiskenine tekrar atama islemi)

Bir referans degiskeninin referansta bulundugu obje degistirilirse yani tekrar atama islemi yapilrsa ve bu objeye baska bir degisken tarafindan referansta bulunulmuyorsa , ilgili obje garbage collection icin uygun hale gelecektir.

```

public class GarbageTest2 {

    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("levent");
        StringBuffer sb2 = new StringBuffer("erguder");

        System.out.println(sb);
        // bu noktada sb referans degiskeninin referans ettigi obje
        // garbage collection icin uygun degildir.

        sb = sb2;
        // bu noktada sb referans degiskeninin referans ettigi obje
        // garbage collection icin uygun hale gelmistir.
        // sb referans degiskenin tuttugu "levent" StringBuffer objesine baska
        // bir referans degiskeni referansta bulunmamaktadir.

    }
}

```

Su ornegimizi inceleyelim;

```

public class GarbageTest3 {

    public static void main(String[] args) {
        Date d = getDate();
        //bu adimda sb referans degiskeninin referansta bulundugu StringBuffer "levent" objesi
    }
}

```

```

//GC tarafindan silinmek icin uygun hale gelecektir. getDate() metodu calismis ve bitmistir.

System.out.println(d);

//bu adimda

//d2 referans degiskeninin yasam alani getDate metodu sonlaninca bitmesine ragmen ilgili objeyi
//d referans degiskeni gosterecegi icin Date objesi GC tarafindan silinmek icin uygun durumda
degildir.

}

public static Date getDate() {
    Date d2 = new Date();

    StringBuffer sb = new StringBuffer("levent");

    return d2;

    //metot tamamlandiginda sb referans degiskeninin referansta bulundugu
    // StringBuffer "levent" objesi garbage collection icin uygun hale gelecektir
    // cunku sb degiskeninin yasam alani(scope) getDate metodu ile sinirlidir.
    // bu metot sonlaninca sb referans degiskenin scope alani sonlanacaktir ve referansta bulundugu
obje

    // GC tarafindan silinmeye uygun hale gelecektir.

    //bununla birlikte bu metot d2 referans degiskenini dondurmektedir,
    //main metodundaki d referans degiskeni ile d2 referans degiskeni ayni Date objesini gosterecektir.

}

}

```

Açıklamaları kod üzerinde yapmaya çalıştım, kısacası önemli olan ilgili adımda yasayan bir thread'ten bir referans degiskenin ilgili objeye ulaşıp ulaşmadığıdır.

### Isolating a Reference

Bir başka durum ise “islands of isolation” durumudur. Kod üzerinde incelemeye devam edelim;

```

public class Island {

    Island i;

    public static void main(String[] args) {
        Island i2 = new Island();
        Island i3 = new Island();
        Island i4 = new Island();

        //3 tane Island turunde obje olusturduk

        i2.i = i3; // i2.i , i3 e referansta bulunur
        i3.i = i4; // i3.i , i4 e referansta bulunur
        i4.i = i2; // i4.i , i2 ye referansta bulunur

        i2=null;
        i3=null;
        i4=null;

        // i2, i3,i4 referans degiskenlerine null degeri atadik.

        //i2.i referans degiskeni yine i3 un referansta bulundugu Island objesine
        //i3.i referans degiskeni yine i4 un referansta bulundugu Island objesine
        //i4.i referans degiskeni yine i2 nin referansta bulundugu Island objesine referansta bulunmaktadır.
        //Fakat bunlar kendi aralarında bir ada(Island) olustudukları için ve disariyla baglantılı olmadığı
        //Garbage Collector calistiginda bunun farkina varacak ve olusturulan bu 3 obje de silinmek için
        uygun hale gelecektir.

    }
}

```

### Forcing Garbage Collection

Aslında Garbage Collection mekanizmasını “zorlayamayız” yukarıda belirttiğim gibi program içerisinde JVM’e istekte bulunabiliriz ve mekanizmayı tetiklemeye çalışabiliriz. JVM bizim isteğimizi dikkate almak veya tüm silinmek için uygun olan objeleri silmek gibi bir **garantisi yoktur!**

Java'da bulunan Runtime sınıfı aracılığı ile (Runtime instance ile) JVM ile iletişim kurabiliyoruz ve Garbage Collection mekanizmasını çalıştırması için istekte bulunabiliyoruz. Küçük bir örnek yapalım bu örneğimizde 100bin Date objesi oluşturuyoruz;

```
import java.util.Date;

public class CheckMemory {

    public static void main(String[] args) {

        Runtime rt = Runtime.getRuntime();

        System.out.println("Total Memory:" + rt.totalMemory());
        System.out.println("Free Memory:" + rt.freeMemory());

        Date d=null;

        //Bu adımda cilginca Date objesi oluşturuyoruz

        for(int i=0; i<100000; i++){
            d= new Date();
            d=null;
        }

        //Deli gibi Date objesi oluşturduk şimdi freeMemory metodunu tekrar çağırıyoruz
        System.out.println("After Free Memory:" + rt.freeMemory());

        //rt referans değişkeni ile JVM e garbage collector u çalıştırma isteğinde bulunuyoruz.
        rt.gc(); // ya da System.gc();

        System.out.println("After GC Free Memory:" + rt.freeMemory());
    }

    // Benzeri örnek çıktı şu şekilde, çalıştırıldığınızda sizde farklı olacaktır
    // GC tüm oluşturulan objeleri silmeyi garanti etmediği için After GC Free Memory farklılıklar gösterebilir.

    //Total Memory:62390272
    //Free Memory:61717816
    //After Free Memory:59750560
}
```

```
//After GC Free Memory:61803160
```

```
}
```

Aciklamalari kod icerisinde yapmaya calistim. Kisacagi gc() metodu ile JVM e istekte bulunuruz ve istegimizin dikkate alınmasının veya sonuclarının **garantisi yoktur!**

Yazimi burada sonlandiriyorum. Bu yazidan cikaracagimiz en onemli sonuc sudur ;

*In Java , you cant trust Thread and GC , in life ; women.*

## Pure Java - 38 the finalize()

[Levent Erguder](#) 19 December 2013 [Java SE](#)

Merhaba Arkadaslar,

Bir onceki yazimizda Java'daki **Garbage Collection** mekanizmasından bahsetmistik. Bu yazimda ise bu mekanizma ile iliskili olan ve *Object* sinifinda bulunan *finalize()* metodundan bahsedecegim. *Object* sinifi Java'da sinif hiyerarsisinin tepesinde bulunmaktadır ve tum siniflar Object sinifini kalittigi icin her sinifin ornegi/instance'i bu *finalize()* metoduna sahip olacaktir.

Object sinifinda finalize metodunu inceledigimizde ;

```
protected void finalize() throws Throwable {  
}
```

seklinde tanimlandigini gorururuz. Ici bos oldugu icin ilgili Sınıfta bu metodu override etmemiz gerekir.

- Garbage Collection mekanizması , **Garbage Collector(GC)** ile Heap'ten ilgili objeyi silmeden once, finalize() metodunu cagirir. Bunun bize ne gibi bir faydası olabilir, obje silinmeden once yapmamız gereken bir işlem olabilir. Dosya kapatma işlemi gibi bir işlem olabilir veya silinmek üzere olan (eligible/uygun) objeyi silinmekten kurtarabilirim (uneligible duruma getirebilirim)
- finalize() metodu GC tarafından sadece bir kez çağrılabılır.
- Eğer GC bir objeyi silmeye karar verdiyse ve finalize() metodu kod icerisinden çağrılmadiysa, GC , finalize() metodunu çağrımayı **garanti eder**.
- Hatırlaycagimiz gibi GC her objeyi silmek gibi bir **garanti saglamaz**. Dolayisiyla finalize() metodu hic **calismayabilir**.

Ornek bir uygulama yazalim;

```
public class FinalizeTest {  
  
    public static void main(String[] args) throws Throwable {
```

```

        for (int i = 0; i < 4500; i++) {
            new FinalizeTest();
        }

        Runtime.getRuntime().gc();

    }

@Override
protected void finalize() throws Throwable {
    System.out.println("hello finalize");
}

```

Ornek bir ciktigı ;

```

hello finalize
hello finalize
hello finalize
hello finalize

```

FinalizeTest sinifimizda finalize() metodunu override ettik. for dongusunde ise 4500 tane FinalizeTest objesi olusturduk. Daha sonrasinda bir onceki yazida inceledigimiz gc() metodunu cagirdik. Kodu tekrar cagirdiginizda farkli bir sonuc cikabilir hatta gc() metodu cagrılması ile hic bir nesne temizlenmemis olabilir. 4500'u arttirarak ekrana daha fazla ciktigı yazmasini saglayabiliyoruz.

```

public class FinalizeTest {

    public static void main(String[] args) throws Throwable {

        for (int i = 0; i < 300000; i++) {
            new FinalizeTest();
        }

        //Runtime.getRuntime().gc();
    }
}

```

```
}

@Override
protected void finalize() throws Throwable {
    System.out.println("hello finalize");
}
}
```

gc() metodunu cagirmasak bile bol miktar obje olusturalim ve finalize() metodunun GC tarafindan cagrildigini gorelim.

```
hello finalize
hello finalize
hello finalize
.....
hello finalize
hello finalize
.....
..... bol miktar hello finalize
hello finalize
```

Ciktilar ornegi her calistirdiginizda farkli olabilir.

Bu yazi ile birlikte Bolum 3'un sonuna geldik.Yazimi burada sonlandiriyorum.  
Son 2 yazi icin unutmamiz gereken nokta ;

*In Java , you cant trust Thread and GC , in life ; women.*

# BÖLÜM -4

## Pure Java - 39 Operators - 01

[Levent Erguder](#) 04 January 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazı ile birlikte Pure Java yazılarimin 4.bolume giriş yapacagiz. Bir kac yazida Javadaki Operatorleri inceleyecegiz. Bir cogunu kullandik fakat burada toplu olarak inceleyecegiz.  
4.bolum diger bolumlerden çok daha kisa olacak.

Konu basliklarimiz ;

- Assignment Operator(Atama Operatoru)
- Compound Assignment Operators(Birlesik Atama Operatorleri)
- Equality & Relational Operators(Illiskisel Operatorler)
- Type Comparison Operator ( Tip Karsilastirma Operatoru)
- Arithmetic Operators(Aritmetik Operatorler)
- Conditional Operators(Kosullu Operatorler)
- Bitwise Operators(Bitsel Operatorler)
- Logical Operators(Mantiksal Operatorler)

## Assignment Operators(Atama Operatoru)

Java 'da Atama Operatoru bildigiz gibi esittir ( $=$ ) operatorudur. Atama Operatorlerinde dikkat etmemiz gereken konulari onceki yazilarda islemistik, *explicit(acik) casting* isleminde deger kaybi gibi.

```
public class OperatorTest {  
    public static void main(String[] args) {  
        int i=257;  
        byte b=(byte)i;  
        System.out.println(b); //1  
    }  
}
```

Onceki yazilarda degisken kavrami ve degiskenlerin birbirine atama isleminin ve deger atama isleminin ne anlamla geldiginden bahsetmistik.

## Compound Assignment Operators(Birlesik Atama Operatoru)

Java'da 11 tane Compound Assignment Operator bulunmaktadır.

```
+=  
-=  
*=  
/=  
%=  
&=  
|=  
^=  
<<=  
>>=  
>>>=
```

Compound(Birlesik) Operatorlere dari bir kac ornek kod ;

```
public class OperatorTest {  
    public static void main(String[] args) {  
  
        int y = 10;  
  
        y -= 5; // y = y - 5;  
        y += 20; // y=y+20;  
        y *= 4; // y=y*4;  
        y /=25; //y=y/25;  
  
        System.out.println(y); //4  
  
        int x = 3;  
  
        x *=2+5;  
        //      x = x + 2 * 5; yanlis  
        // x=x *(2+5); dogru
```

```

    // Compound Operatoru kullanirken dikkat etmemiz gereken bir nokta.

}

}

```

## Equality & Relational Operators(İliskisel Operatorler)

Java da 6 tane relational(iliskisel) operator bulunur. Bunları bol miktarda kullanmaktayız.

```

<<=>==!=

class RelationalOperatorTest{

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2)
            System.out.println("value1 == value2");
        if(value1 != value2)
            System.out.println("value1 != value2");
        if(value1 > value2)
            System.out.println("value1 > value2");
        if(value1 < value2)
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}

```

### Equality Operator

Java ‘da 2 relational operatore “equality operator” de denilir. Bunlar ;

```

==
!=

```

Her compare/karsilastirma islemi birbirile uyumlu olan tipler olmalıdır. Incompatible(farklı/uyumsuz) tiplerde karsilastirma yapılamaz. Ilkel tipleri karsilastirabildigimiz gibi referans tipleri de karsilastirabiliriz.

Ilkel tipler icin karsilastirma ornegimiz ;

```

class ComparePrimitives {

    public static void main(String[] args) {
        System.out.println("char 'a' == 'a'? " + ('a' == 'a'));
        System.out.println("char 'a' == 'b'? " + ('a' == 'b'));
        System.out.println("5 != 6? " + (5 != 6));
        System.out.println("5.0 == 5L? " + (5.0 == 5L));
        System.out.println("true == false? " + (true == false));
    }
}

```

`==` operatorunu kullanirken dikkat edelim , yanlislikla `=` operatorunu kullanırsak yanlis sonuc ortaya cikacaktir. Bazi durumlarda ise compile/derleme hatasina neden olur.

```

public class Test {

    public static void main(String[] args) {
        boolean b=false;
        if(b=true){ // == yerine = yanlis kullanildiginda
            System.out.println("b is false");
        }

        int x=1;
        // if(x=1) {} // compile error

        if(x==1) {} // boolean olmali
    }
}

```

Primitive(ilkel) tipler arasında `==` operatorunu kullanabildigimiz gibi referans tipler icin de `==` operatorunu kullanabiliriz. Referans tipler icin karsilastirma ornegimiz;

```

public class CompareReference {

    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("levent");
    }
}

```

```

        StringBuilder sb2 = new StringBuilder("levent");
        StringBuilder sb3=sb;

        System.out.println("sb==sb2 ?"+(sb==sb2));
        System.out.println("sb==sb3 ?"+(sb==sb3));
        System.out.println("sb2==sb3 ?"+(sb2==sb3));
        System.out.println("sb2!=sb3 ?"+(sb2!=sb3));

    }

}

```

StringBuilder i ilerleyen yuzelarda inceleyecegiz. Burada onemli olan nokta sb, sb2,sb3 primitive degil referans tipinde degisenlerdir. Onceki yuzelarda degisen nedir ,primitive ve referans tipteki degisenleri bir degiseni bir degiskene atamanin ne anlama geldigini incelemistik.

`==` operatoru 2 objeyi “meaningfully equivalent/anlamsal olarak esitlik/icerik olarak” test etme islemini yapmaz. Bu testi equals metodu yapar. Bu metodu ve `==` operatorunu ilerleyen yuzelarda detaylica tekrar inceleyecegiz.

`==` operatorunu Enum’lar icin de kullanabiliriz. Ornek kodumuz ;

```

class EnumEqual {
    enum Color {
        RED, BLUE
    };

    public static void main(String[] args) {
        Color c1 = Color.RED;
        Color c2 = Color.RED;
        if (c1 == c2) {
            System.out.println("==");
        }
    }
}

```

## Pure Java – 40 Operators – 02

[Levent Erguder](#) 10 January 2014 [Java SE](#)

Merhaba Arkadaslar,

Bir onceki yazida Java'da Operatorler konusu ile 4.bolume baslamistik. Bu bolumde de bir onceki yazida belirtigimiz **Type Comparison Operator** yani *instanceof* operatorunu inceleyecegiz.

### Type Comparison Operator(Tip Karsilastirma Operatoru)

Java da tip karsilastirma operatoru olarak *instanceof* anahtar kelimesi kullanilarak yapilir. *instanceof* operatoru sadece referans tipte degiskenler icin kullanilabilir. *instanceof* operatoru IS-A kontrolu yapmaktadır.

Java'da 8 tane primitive tipte degisken bulunmaktadır. Bildigimiz gibi **String** bu ilkel tiplerden degildir. Dolayisiyla *instanceof* operatorunu kullanabiliyoruz.

```
public class Test {  
    public static void main(String[] args) {  
  
        String name = "levent";  
        if (name instanceof String) {  
            System.out.println("name is a String"); // instanceof kontrolu // IS-A  
        }  
  
        int age = 24;  
        // primitive tipler icin instanceof kontrolu yapilamaz.  
        // if(age instanceof Integer){ } //derleme hatasi //  
        // if(age instanceof int) { } //derleme hatasi  
  
        if(name instanceof Object) {  
            System.out.println("name is an Object"); // String IS-A Objects  
        }  
    }  
}
```

*instanceof* operatoru IS-A kontrolu yapmaktadır demistik, simdi sinif ve arabirim kullanarak biraz kod daha yazalim ;

```
interface Z {  
}  
  
class A implements Z {  
}
```

```

class B extends A {
}

public class C extends B {

    public static void main(String[] args) {

        Z z1 = new A();
        Z z2 = new B();
        Z z3 = new C();

        A a0 = new A();
        A a1 = new B();
        A a2 = new C();

        B b0 = new B();
        B b1 = new C();

        C c1 = new C();

        // z1 referans degiskeni A sinifi turunde bir objeye refarans
        // etmektedir.
        // IS-A kuralini dikkate alarak sonucları dusunelim.

        System.out.println("z1 instanceof test");
        System.out.println(z1 instanceof A); // true
        System.out.println(z1 instanceof B); // false
        System.out.println(z1 instanceof C); // false

        System.out.println("ao instanceof test");
        System.out.println(a0 instanceof Z);
        System.out.println(a0 instanceof A);
        System.out.println(a0 instanceof B);
        System.out.println(a0 instanceof C);

        System.out.println("c1 instanceof test");
    }
}

```

```

        System.out.println(c1 instanceof Z);
        System.out.println(c1 instanceof A);
        System.out.println(c1 instanceof B);
        System.out.println(c1 instanceof C);

    }

}

```

Sonuclar sasirtici gelirse onceki yuzlarda bahsettim IS-A kuralini incelemekte fayda olacaktir. Bu konuda bol miktar kod yazmanizi oneririm.

*instanceof* kontrolunu **null** ile de yapabiliriz, sonuc olarak her zaman *false* donecektir.

```

public class TestNull {

    public static void main(String[] args) {
        String a=null;
        boolean b = null instanceof String; //false
        boolean c = a instanceof String; //false
        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
    }
}

```

*instanceof* operatorunu farkli sinif hiyerarsileri icin **kullanamayiz**.

```

class Cat {

}

interface IFace {

}

public class Dog {
    public static void main(String[] args) {
        Dog dog = new Dog();
}

```

```

        System.out.println(dog instanceof Dog);

        // farkli sinif hiyerasisinde bulunan siniflar arasında instanceof
        // operatoru derleme hatasina neden olur.

        //System.out.println(dog instanceof Cat); // derleme hatasi
        //Bu durum interface icin soz konusu degildir.

        System.out.println(dog instanceof IFace);

    }

}

```

Hatirlayacagimiz gibi Javada bir dizi icin her zaman IS-A Object kontrolu dogrudur.

```

public class ArrayTest {

    public static void main(String[] args) {
        int [] number = new int[5];
        //array tanimlamasini incelemistik. Array tipindeki bir degisken icin her zaman IS-A Object kurali
        dogrudur.
        //Bu dizimizin elemanlari "int" tipinde oldugunu hatirlayalim
        System.out.println(number instanceof Object);
    }
}

```

## Pure Java - 41 Operators - 03

[Levent Erguder](#) 12 January 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazida inceleyecegimiz operatorler ;

- Arithmetic Operators
- Increment & Decrements Operators
- Conditional Operator

### Arithmetic Operators

Bu operatorlerimizi zaten daha oncesinden biliyoruz bol miktarda kullandik,

+ - \* / %

```

public class ArithmeticOperators {

    public static void main(String[] args) {
        int x = 5*3;
        int y = x-4;
        System.out.println(x);
        System.out.println(y);

        int z = 13%5;
        System.out.println(z);

    }
}

```

Bu noktada dikkat etmemiz gereken bir konu String Concatenation ile ilgili olabilir.

```

public class StringConcatenation {

    public static void main(String[] args) {
        String a = "java";
        int b = 10;
        int c = 5;
        System.out.println(a + b + c);
    }
}

```

Ornegimizi calistirdigimizda java15 mi yanzacak ? Hayir , burada a degiskenimiz String tipinde oldugu icin a+b islemi String olacaktir (“java10”) sonrasinda c degiskenimizi ekledigimizde “java105” olacaktir.

```

public class StringConcatenation {

    public static void main(String[] args) {
        int b=5;
        System.out.println(""+b+3);
        System.out.println(b+3);
    }
}

```

```
    }  
}
```

### Increment(arttirma) & Decrement(azaltma) Operators

Java da bu operatorler ;

```
++ --
```

Bu operatorlerde dikkat etmemiz gereken nokta prefix(onek) veya postfix(sonex) olmasi durumuna gore sonucun degisecegidir.

```
public class Test {  
    public static void main(String[] args) {  
        int i = 5;  
        int k = 5;  
        System.out.println(i++); // postfix  
        System.out.println(++k); // prefix  
  
        int t = 0;  
        int[][] a = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 7, 9 } };  
        System.out.println(a[t++][++t]);  
    }  
}
```

### Conditional Operator

Conditonal Operator'e Ternary Operator de denilir.

x = (boolean expression) ? assign if true : assing if false

```
public class Ternary {  
  
    public static void main(String[] args) {  
        int point = 75;  
        String result = point < 50 ? "Fail" : "Success";  
        System.out.println(result);  
  
        String result2 = point < 50 ? "F" : point > 90 ? "A" : "B";  
        System.out.println(result2);  
    }  
}
```

```
    }  
}
```

## Pure Java - 42 Operators - 04

[Levent Erguder](#) 19 January 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazim ile OCP notlarimin 4.bolumunu bitirecegim. Bu yazida Java'da su operatorleri isleyecegiz;

- Bitwise(bitsel) ve Bit Shift Operators
- Logical(Mantiksal) Operators

### Bitwise ve BitShifting Operators

Bitwise veya Bit Shift operatorleri ozel durumlarda, kriptolama gibi durumlarda kullanilmaktadir.

```
& (ve) | (veya) ^ (xor)
```

```
public class Bitwise {
```

```
    public static void main(String[] args) {
```

```
        byte b1=6&8;
```

```
        //0110 --->6
```

```
        //1000 --->8
```

```
        //----- & ve islemi
```

```
        //0000
```

```
        System.out.println(b1);
```

```
        byte b2=5|9;
```

```
        //0101 --->5
```

```
        //1001 --->9
```

```
        //----- | veya islemi
```

```
        //1101
```

```
        System.out.println(b2);
```

```
        byte b3=7^5;
```

```

        //0111 --->7
        //0101 --->5
        //----- ^ xor islemi
        //0010

        System.out.println(b3);

    }

}

```

```

public class BitShifting {

    public static void main(String[] args) {

        byte b = 8>>1;
        //00001000 --->8
        // bir bit saga kaydirirsak;
        //00000100 --->4
        System.out.println(b);

        b=7>>2;
        //00000111 --->7
        // 2 bit saga kaydirirsak
        //00000001 --->1
        System.out.println(b);

        b=4<<3;
        //00000100 -->4
        // 3 bit sola kaydirirsak
        //00100000 -->32
        System.out.println(b);

    }
}

```

```
}
```

## Logical(Mantiksal) Operators

*Logical(Mantiksal)* Operatorleri fazla miktarda kullanırız. Java'da *Logical(Mantiksal)* operatorler bulunur;

&& shortcut logical **and**

|| shortcut logical **or**

& boolean logical **and**

| boolean logical **or**

^ boolean logical **xor**

! logical **not**

**Shortcut** operatorlerinde şartın sağlanıldığı veya sağlanmadığı kesin olduğunda işlem bitirilir sonraki **operand** (islemci/islenen) calistirılmaz.

```
public class ShortcutOR {  
    public static void main(String[] args) {  
  
        int i=5;  
        if(i>4 || i/0>1){  
            System.out.println(i);  
        }  
  
        if(i>4 | i/0>1){  
            System.out.println(i);  
        }  
    }  
}
```

1. if kodunda  $i > 4$  şartı sağlanır , veya mantığında sadece bir operandın true olması yeterli olacaktır. Dolayısıyla  $i/0$  çalışmayaçak ve hata fırlatmayacağındır.

2.if kodunda ise şart sağlanmasına rağmen | operatoru kullanıldığı için , ikinci operand da çalışacak ve  $i/0$  işlemi sonucu hata fırlatılacaktır.

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

```

int i=5;
if(i<4 && i>1){
    System.out.println(i);
}
System.out.println("Exception ikinci if te firlatilacaktir");

if(i<4 & i>1){
    System.out.println(i);
}
}
}

```

ShortcutAnd mantigi da shortcutOr mantigi seklinde calisacaktir.

- 1.if durumunda i<4 şart saglanmaz shortcut && kullanildigi icin , ve mantiginda iki operandin da true oldugu durumda islem true olacagi icin, burada ikinci operand calistirilmaz.
- 2.if durumunda i<4 şart saglanmaz shortcut && kullanilmadigi icin , şartin saglanmasi ihtimali olmamasina ragmen ikinci operand da calistirilir ve hata firlatilir.

&& ve || operatorleri sadece boolean operand'lar ile calisir. & ve | operatorlerinin aksine su sekilde boolean olmayan literaller icin kullanimi hatali olacaktir

**10&&5**

**5||7**

$\wedge$  (XOR) operatoru de sadece boolean operand'lar ile calisir.

**public class Logical {**

```

public static void main(String[] args) {

    boolean bool = false;
    System.out.println(!true);
    System.out.println(!bool);

    System.out.println(true ^ false);
    System.out.println(true ^ true);
}

```

```
        System.out.println(false ^ false);

    }

}
```

# BÖLÜM- 5

## Pure Java - 43 Flow Control - 01

[Levent Erguder](#) 29 January 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazi ile birlikte OCP notlarimin 5.bolumune gecis yapmis olacagiz. 5.bolumde su konu basliklarindan bahsedecegim

- Flow Control
- Exceptions
- Assertions

### **if , switch Statement**

Evet biliyorum bu konu basligi programlama ile ilk karsilastigimizda karsimiza gelmekte ,burada yeri gelmisenken cok kisa sekilde inceleyecegiz ve bir cok Java'cinin bilmedigi **switch** lerde bir kac trick gorecegiz. Bunun disinda ufac oneriler ve ince detaylardan bahsedecegim.

### **if-else branching/dallanmasi**

En temel if durumumuz(statement);

```
if(booleanExpression){

    System.out.println("Inside if statement")
```

```
}
```

Parantezler icersindeki ifade(expression) boolean degerde olmalıdır yani **true** ya da **false**.

Burada kucuk bir not vermek istiyorum, **if-else** yapısında tek satır dahi yazsanız code standard'ı açısından mutlaka suslu parantez { } kullanılmalıdır.

```
if(x>3)  
y=2;  
z+=8;  
a=y+x;
```

Sinavda buna benzer kodları görebilirsiniz fakat sınav sizin dikkatinizi ve detay bilginizi sorgular. Normalde kod yazarken karışıklığa neden olmaması için ilgili yere suslu parantezleri unutmayalım.

if-else branching/dallanması yapısında dikkat etmemiz gereken bir kaç nokta;

- else statement'i 0 yada 1 tane olabilir.
- else if statement'i ise 0 ya da 1'den fazla olabilir.
- eğer ilk else if statement'i başarılı olursa sonrasında else if'ler veya else çalışmaz.

Yukarıdaki duruma özellikle Java ve programlamada yeni olan arkadaşlar dikkat etmelidir.

```
public class Test {  
  
    public static void main(String[] args) {  
        int x = 1;  
        if (x == 3) {  
        } else if (x < 4) {  
            System.out.println("<4");  
        } else if (x < 2) {  
            System.out.println("<2");  
        }  
    }  
}
```

Yukarıdaki kodda birinci else-if şartı sağlandığı için ikinci else-if şartı çalışmamaktadır.

Fakat kodumuzda else-if yerine sadece if olursa sağlanan her şartın çalıştığını görebiliriz.

```
public class Test2 {  
    public static void main(String[] args) {  
        int x = 1;  
        if (x == 3) {  
        }  
        if (x < 4) {  
            System.out.println("<4");  
        }  
        if (x < 2) {  
            System.out.println("<2");  
        }  
    }  
}
```

Su kodu inceleyelim ;

```
public class Test3 {  
    public static void main(String[] args) {  
  
        int x = 5;  
        int y = 10;  
  
        if (x > 3)  
        if (y > 10)  
            System.out.println("y>5");  
        else  
            System.out.println("else");  
    }  
}
```

Yukarıdaki koda benzer durumlarda else hangi if'e ait olacak ? Birinci if şartının else şartımı mı yoksa ikinci if şartının else şartı mı ? else kendinden bir önceki if şartına ait olacaktır.

Sinavda bu tarz gibi sorular gelecektir. Normal şartlarda mutlaka suslu parantez koymamız kafa karışıklığına engel olacaktır.

```
if durumumuz(statement) ,
```

```
if(booleanExpression){  
    System.out.println("Inside if statement")  
}
```

Burada en onemli noktamiz booleanExpression olmasiydi. Bu sart saglandigi surece her expression yazilabilir.

```
if( (x>3 && y<2) | doStuff() {  
}
```

Yine burada dikkat edilmesi gereken konu calisma onceliklerine dair olacaktir. Sinavda sizi sasirtmak icin ellerinden geleni yapacaklardir. Gercek hayatta kod yazarken ise bol miktarda parantez kullanimi ile temiz kod yazmaya ozen gösterin. Aksi halde kafa karisikligina neden olacaktir.

```
int trueInt=1;  
int falseInt=0;  
  
if(trueInt) //illegal  
if(trueInt==true) //illegal  
if(1) //illegal  
if(1) //illegal  
if(falseInt==false) //illegal  
if(trueInt==1) //legal  
if(falseInt==0) //legal
```

En çok yapılan **hatalardan** biri == yerine = kullanmak olacaktir. Bu durum gerçek hayatta karsimiza cikabilecegi gibi sinavda da karsimiza cikacaktır. Dikkat etmemiz gerek. boolean degerler disinda compile/derleme hatasi verecegi icin IDE bizi uyarır ama eger asagidaki gibi bir boolean ifade kontrolu yapiyorsak, kodumuz istemedigimiz sekilde calisir ve “nasıl ya” diye sorularini kendi kendimize sormaya baslayabiliriz.

```
boolean b=false;
```

```
if(b=true) {
```

```
        System.out.println("== yerine = kullanildi")  
    }
```

## Pure Java - 44 Flow Control - 02

[Levent Erguder](#) 05 February 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde Java'da *switch statement/ifadesini* inceleyecegiz. *switch* statement/ifadesini coklu *if-else* yapisi yerine kullanabiliriz. Oncelikle bildigimiz tarzda basit bir ornek kod gorelim sonrasında detayli noktalarini inceleyelim.

### **SwitchTest.java**

```
package purejava44;  
  
public class SwitchTest {  
  
    public static void main(String[] args) {  
  
        int x = 3;  
  
        if (x == 1) {  
  
            System.out.println("x esittir 1");  
  
        } else if (x == 2) {  
  
            System.out.println("x esittir 2");  
  
        } else if (x == 3) {  
  
            System.out.println("x esittir 3");  
        }  
    }  
}
```

```
    } else {

        System.out.println("x hakkında bilgim yok!");

    }

switch (x) {

    case 1:

        System.out.println("x esittir 1");

        break;

    case 2:

        System.out.println("x esittir 2");

        break;

    case 3:

        System.out.println("x esittir 3");

        break;

    default:

        System.out.println("x hakkında bilgim yok!");

    }

}
```

```
}
```

**switch** statement/ifadesinin genel formu

```
switch (expression) {  
  
    case constant1: code block  
  
    case constant2: code block  
  
    default: code block  
  
}
```

Java 6 da, switch ifadesinin expression kismi **char**, **byte**, **short**, **int** tipinde ya da bu primitive tiplerin wrapper siniflari olan **Character**, **Byte**, **Short**, **Integer** tipinde ya da **enum** olabilir ,fakat **long**, **float**, **double** tipinde ya da bu primitive tiplerin wrapper siniflari olan **Long**, **Float**, **Double** tipinde olamaz. Java 7 de ek olarak **String** tipinde de olabilir.

Simdi biraz daha detaya inelim ve ozellikle sinav icin bilmemiz gereken noktalara deginelim.

1) case constant/sabiti , compile time/derleme zamani sabiti olmalıdır. Ne demek istedigimi kod uzerinde gorelim ;

### SwitchTest2.java

```
public class SwitchTest2 {  
  
    public static void main(String[] args) {  
  
        final int a = 1;  
  
        final int b;  
  
        b = 2;
```

```

int c=3;

int x = 0;

switch (x) {

    case a:

        // ok

    case b: // compiler error //derleme hatasi

    case c: // compile error //derleme hatasi

}

}

}

```

dikkat ederseniz b ve c degiskenenin **case** ifadesinde kullanimi derleme hatasina neden olacaktir, degiskenen final olmasi yeterli olmamakta ancak ve ancak ilk tanimlandigi yerde/declare deger atanmasi **gerekmektedir**.

2) **case** constant/sabiti ,**switch** expression'da kullandigimiz degiskenen tipine uygun deger araliginda olmalidir.

### **SwitchTest3.java**

```

public class SwitchTest3 {

    public static void main(String[] args) {

        byte b=10;
    }
}

```

```
switch(b){  
  
    case 25:  
  
    case 100:  
  
        //case 128: // 128 degeri byte tipinin degeri disindadir.  
  
        //bu nedenle cast etmek gerekir ya da derleme hatasina neden olur  
  
    case (byte)128:  
  
}  
  
}  
}
```

3) *case* ifadesinde birden fazla ayni deger kullanilamaz.

```
byte b=10;  
  
switch(b){  
  
    case 25:  
  
    case 100: //derleme hatasi  
  
    case 100; //derleme hatasi  
  
}
```

4) Wrapper sınıf tipinde expression kullanabiliriz.

```
switch(new Integer(4)) {  
  
    case 4: System.out.println("boxing is OK");  
  
}
```

5) *switch-case* yapısı için dikkat etmemiz bir kaç yazım kuralı bulunmaktadır. Şu kod örneklerini inceleyelim ;

```
switch(x) {  
  
    case 0 { // case expression'dan sonra iki nokta(:) gereklid.  
  
        int y = 7;  
  
    }  
  
}  
  
  
switch(x) {  
  
    0: {} //case ifadesi eksik !  
  
    1: {} //case ifadesi eksik !  
  
}
```

### **break & fall-through**

*switch-case* yapısında **break** statement/ifadesi seçimlidir.

*case* constant/sabitleri kontrolü yukarıdan aşağı doğru olmaktadır, eğer **break** kullanmazsa ilk eslesen *case* constant/sabitinden itibaren aşağıya doğru devam etmektedir.

### **SwitchTest4.java**

```
enum Color {  
    red, green, blue  
}  
  
public class SwitchTest4 {  
  
    public static void main(String[] args) {  
  
        Color c = Color.green;  
  
        switch (c) {  
  
            case red:  
  
                System.out.print("red ");  
  
            case green:  
  
                System.out.print("green ");  
  
            case blue:  
  
                System.out.print("blue ");  
  
            default:  
  
                System.out.println("done");  
        }  
    }  
}
```

```

        int x = 1;

        switch(x) {

            case 1: System.out.println("x is one");

            case 2: System.out.println("x is two");

            case 3: System.out.println("x is three");

        }

    }

}

```

green degeri icin eslesme oldugu icin green blue done olarak cikti olacaktir. Benzer sekilde x=1 degeri icin 3 *case* ifadesi de calisacaktir. Bu yapiya *fall-through* denilmektedir.

*break* kullandigimiz da ise ilgili şart saglandiktan sonra *switch-case* yapisindan cikilacaktir.

```

int x = 1;

switch(x) {

    case 1: {

        System.out.println("x is one"); break;

    }

    case 2: {

        System.out.println("x is two"); break;
    }
}

```

```
}

case 3: {

    System.out.println("x is two"); break;

}

}
```

### default keyword

Java'da **default** anahtar kelimedir/keyword ve **switch-case** yapısında istege bağlı olarak kullanılabilir.

#### SwitchTest5.java

```
public class SwitchTest5 {

    public static void main(String[] args) {

        int x = 7;

        switch (x) {

            case 2:

            case 4:

            case 6:

            case 8:

            case 10: {

                System.out.println("x cift sayidir");

                break;
            }
        }
    }
}
```

```

    }

    default:

        System.out.println("x tek sayidir");

    }

}

```

**default** anahtar kelimesi **switch-case** yapısında en sona gelmek zorunda degildir. Birinci **switch** ifadesinde x=2 icin ilk şart saglandiktan sonra sirasiyla case'ler calisacaktir. Ikinci **switch** ifadesinde x=7 oldugu icin **default** calisacaktir ve yukaridan asagiya dogru **fall-through** gerceklesecektir. Yani **fall-through** kuralimiz **default** icin de gecerlidir.

```

public class SwitchTest6 {

    public static void main(String[] args) {

        int x = 2;

        switch (x) {

            case 2:

                System.out.println("2");

            default:

                System.out.println("default");

        }

        case 3:

            System.out.println("3");
    }
}

```

```
case 4:  
  
        System.out.println("4");  
  
    }  
  
  
  
  
System.out.println("Test 2");  
  
x = 7;  
  
  
  
  
switch (x) {  
  
    case 2:  
  
        System.out.println("2");  
  
    default:  
  
        System.out.println("default");  
  
    case 3:  
  
        System.out.println("3");  
  
    case 4:  
  
        System.out.println("4");  
  
    }  
  
  
  

```

```
}
```

## Pure Java - 45 Loops

[Levent Erguder](#) 01 March 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda Java da dongu(loop) yapılarından bahsedecegim. Evet bir çok dilde bu yapı zaten benzer yapida calismaktadir. Konu akisi geregi bu bolumu de isleyecegim, donguler konusundan sonra daha keyifli konulara yavas yavas gececegiz.

Burada anlatacagim ek detaylar, öneriler ve sinavda dikkat edilmesi gereken noktalar yine de faydalı olacaktır.

### **while dongusu**

while dongu yapisi genel olarak , dongunun kaç defa dondugunu bilmedigimiz durumlarda faydalı olabilir.

**while** dongusu genel formati ;

```
while(expression){  
    // do something  
}
```

Burada **expression** boolean bir sonuc olmalıdır. **while** dongusu bu **expression** sonucunun **true** olduğu surece çalışacaktır.

**expression** kısmında kullanılacak değişkenimiz , **while** dongusunden önce tanımlanmış olması gereklidir. Yani ;

```
while(int x=2; x<3) { } //derleme hatası verecektir.
```

**while** dongusune hiç girilmeyebilir. Eğer expression ilk durumda **false** değere sahipse , donguye girilmeden atlanacaktır.

```
public class LoopTest1 {  
    public static void main(String[] args) {  
        int x = 5;  
        while (x > 5) {  
            System.out.println("dongu icerisi");  
        }  
        System.out.println("dongu disi");  
  
    }  
}
```

## **do-while dongusu**

**do-while** dongusu **while** dongusune benzer , fark olarak **expression** kontrolu dongunun sonunda yapilir.

Genel format ;

```
do {  
    //do something  
} while(expression)
```

**do-while** dongusunde ,**expression** kontrolu **do-while** dongusunun sonunda yapilacagi icin **do-while** dongusu en azindan 1 kez calisacaktir.

```
do{  
    System.out.println("dongu ici");  
} while(false);  
  
//sinavda noktali virgul(;) un olup olmadigina  
//dikkat etmek gerek
```

Sinavda dikkat edilmesi gereken bir kac ifade;

```
int x=1;  
  
while(x){ } //derleme hatasi , x boolean tipinde degil  
while(x=5) { } //derleme hatasi , atama operatoru  
while(x==5){ } //gecerli bir expression  
while(true){ } //gecerli bir expression
```

## **for dongusu**

Java 6 ile itibaren for dongusune yeni bir yapi geldi. Bunlar ;

- basic for loop(temel for dongusu)
- enhanced for loop(gelistirilmis for dongusu)

### **temel for dongusu**

**for** dongusu en cok kullanilan dongu yapisidir. Ozellikle dongunun kac defa calisacagi biliniyorsa bu dongu yapisi kullanilmaktadir. **for** dongusunun 3 bolumden olusur ;

- degiskenlerin tanimlanmasi(declaration) ve deger atanmasi(initialization)
- kontrol ifadesi(boolean expression)
- tekrarlama ifadesi(iteration expression)

for dongusu genel formati;

```
for(Initialization ; Condition ; Iteration) {  
//  
}
```

for dongusunun , **initialization** kisminda 0 veya daha fazla ayni tipte degiskeni tanimlayip deger atamasinda bulunabiliriz.

```
for (int x = 10, y = 5, z=20; x < 15; x++) {  
}
```

Degisken tanimlama ve deger atama kismi **for** dongusune girmeden once calisacaktir. Bu kisim **for** dongusunde sadece bir kez en basta calisacaktir.

*Sinavda dikkat edilmesi gereken noktalardan birisi, degiskenlerin scope/kapsam alanina dikkat etmek gerekir.*

```
for(int x=1; x<5; x++){  
    System.out.println(x);  
}  
  
System.out.println(x); //derleme hatasi, scope/kapsam disi
```

**initialization** kisminden sonra , **condition** kismi calisacaktir.

Bu kisim **boolean** deger olmalidir. Burada dikkat edilmesi gereken nokta ; en fazla 1(bir) **logical expression**(mantiksal ifade) olabilir fakat bu **logical expression** karisik/complex olabilir.

```
for (int x = 0, y = 5; ((x < 5 && y > 3) | x == 1); x++) {  
  
}
```

Yukarıdaki **for** dongusu derleme hatasi vermez fakat bu **for** dongusunde derleme hatasi olacaktır;

```
for(int x=1,y=5; (x>5),(y<3); x++ ) {  
  
}
```

*Sinavda dikkat edilmesi gereken nokta: sadece bir tane test expression (ifadesi) olabilir. Diger bir deyisle birden fazla test kontrolunu virgul ile ayirip for dongusunde kullanamayiz.*

for dongusunun govdesi calisip sonlandiktan sonra iteration kismi calisacaktir. Yani **iteration** kismi **for** dongusunde en son olarak calisacaktir.

for dongusunde bu uc kisimda zorunlu degildir. Yani bu **for** dongusu sorunsuzca sonsuza kadar calisir.

```
for (;;) {  
    System.out.println("Sonsuz Dongu");
```

```
}
```

*for dongusu initialization ve iteration kismi olmadan while dongusu gibi kullanilabilir. Burada ; işaretlerine dikkat etmek gerekir.*

```
int i = 0;  
for (; i < 10;) {  
    i++;  
    //  
}
```

Tekrar etmekte fayda var. Sinavda variable scope/degisken yasam alani/kapsam alani onemli.

```
int x = 3;  
for (x = 12; x < 20; x++) {  
    System.out.println(x);  
}  
System.out.println(x);  
//x for dongusunden once tanimlandigi icin kapsam alani  
//burada da gecerli olacaktir.
```

```
for (int x = 12; x < 20; x++) {  
    System.out.println(x);  
}  
System.out.println(x);  
// x for dongusunde tanimlandigi icin bu noktada  
//x degiskenenin kapsam alani gecerli olmayacaktir  
//ve derleme hatasi verecektir.
```

Son bir ek olarak , **for** dongusundeki bu 3 kisim birbirinden bagimsizdir. Yani bu uc kisimda ayni degiskenlerin kullanilmamasina gerek yoktur. Bununla birlikte Iteratin kisminda her zaman increment operatoru gormeye aliskiniz , fakat boyle olmaz zorunda degildir. Yani;

```
int b=3;  
for(int a=1; b!=1; System.out.println("iterate")) {  
    b=b-a;
```

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

### gelistirilmis for dongusu

Gelistirilmis for dongusu(enhanced for loop) Java 6 ile birlikte geldi. Gelistirilmis for dongusu , array ve collection yapilarinda kolaylik saglayan bir yapıya sahiptir.

Gelistirilmis for dongusu genel formati;

```
for(declaration : expression)
```

ornek olarak;

```
// temel for dongusu
int a[] = { 1, 2, 3, 4 };
for (int x = 0; x < a.length; x++) {
    System.out.println(a[x]);
}

//gelistirilmis for dongusu
for(int n:a){
    System.out.println(n);
}

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

        int[] intArray = { 1, 2, 3, 4 };
        Integer[] wrapperIntArray = { 1, 2, 3, 4 };

        for (int x : intArray)
            System.out.println(x);

        for (Integer x2 : intArray)
            System.out.println(x2);

        for (int x3 : wrapperIntArray)
            System.out.println(x3);
    }
}
```

```

        System.out.println(x3);

        for (Integer x4 : wrapperIntArray)

            System.out.println(x4);

    }

}

```

Gelistirilmis for dongusunde kullandigimiz dizinin tipi ile, “declaration” kisminda kullandigimiz degiskenen tipi uyumlu olmalıdır.

```

long[] longArray = { 1, 2, 3, 4 };

Long[] wrapperLongArray = { 1L, 2L, 3L, 4L };

for (Long x4 : longArray)

    System.out.println(x4);

for (int x5 : longArray) //derleme hatasi ,
    System.out.println(x5);

//long veya Long tipindeki dizi icin "declaration" kisminda
//int tipinde degisken kullanamayiz.

```

Bir diger dikkat edilmesi gereken nokta ,”declaration” kisminda kullanilan degiskenen daha once tanimlanmis olmamasi gerekmektedir.

```

int x5=0;

for(x5:intArray) //derleme hatasi

System.out.println(x5);

```

Gelistirilmis for dongusunde primitive tipleri kullanabilecegimiz gibi referans tipte elemanlar iceren dizileri de kullanabiliriz.

```

class Animal {

}

class Dog extends Animal {
}

```

```

class Cat extends Animal {

}

public class TestAnimal {
    public static void main(String[] args) {
        Animal[] animalArray = { new Dog(), new Cat(), new Cat() };

        Dog[] dogArray = { new Dog(), new Dog(), new Dog() };

        for (Animal d : animalArray) {

        }

        // for (Dog d2 : animalArray) { // derleme hatasi
        // }

        for (Dog d3 : dogArray) {

        }
    }
}

```

Daha fazla kod ornegini size birakiyorum , bol miktar kod yazmak faydalı olacaktır.

## Pure Java - 46 break & continue

[Levent Erguder](#) 07 March 2014 [Java SE](#)

Merhaba arkadaşlar,

Bu yazımında unlabeled/labeled(etiketsiz/etiketli) **break** ve **continue** keyword(anahtar kelime)lerini inceleyeceğiz.

**continue** ifadesi(statement) bir dongu içerisinde olmak zorundadır fakat hatırlayacağımız gibi **break** ifadesi(statement) hem dongu içerisinde hem de **switch** ifadesi (statement) içerisinde kullanılabilir. **break** ifadesi(statement) en icteki dongunun durmasını/kırılmasını saglar ve bir sonraki kod satırından devam eder.

**continue** ifadesi(statement) en icteki dongunun bir sonraki iterasyona geçmesini saglar.

```

public class Test {

    public static void main(String[] args) {

```

```
for (int i = 0; i < 10; i++) {  
  
    if (i % 2 == 0) {  
  
        continue;  
  
    }  
  
    if (i == 7) {  
  
        break;  
  
    }  
  
    System.out.println("i:" + i);  
  
}  
  
}
```

i%2==0 ile , cift sayilar icin şart saglandigi icin continue calisacak ve dongu bir sonraki iterasyona gececektir.  
i=7 oldugunda break şarti saglanacak ve dongu kirlacaktir.

Sinavda su tarz sorular gelebilir, iyi analiz etmek gereklidir.

```
public class Test2 {

    public static void main(String[] args) {

        int[] ia = { 1, 3, 5, 7, 9 };

        for (int x : ia) {

            for (int j = 0; j < 3; j++) {

                if (x > 4 && x < 8)

                    continue;

                System.out.print(" " + x);

                if (j == 1)

                    break;

                continue;

            }

            continue;

        }

    }

}
```

```
}
```

break ve continue ifadelerini etiketli(label) olarak da kullanabiliriz.

Etiketsiz break ve continue ifadesin en icteki dongu icin calismaktadir, ic ice dongu yapilarinda etiket kullanarak break ve continue nun calisma kapsamini en icteki donguden disardaki dongulere cikartabiliriz.

```
public class Test3 {  
  
    public static void main(String[] args) {  
  
        outer: for (int i = 0; i < 5; i++) {  
  
            for (int j = 0; j < 5; j++) {  
  
                System.out.println("Hello");  
  
                break outer;  
  
            }  
  
            System.out.println("outer"); // Never prints  
  
        }  
  
        System.out.println("Good-Bye");  
  
    }  
}
```

icerdeki dongu calistiginda icerdeki break etiketli oldugu icin iki dongu de kirlacaktir.  
ayni kodu continue ifadesi icin de deneyebiliriz.

```
public class Test4 {
```

```

public static void main(String[] args) {

    outer: for (int i = 0; i < 5; i++) {

        for (int j = 0; j < 5; j++) {

            System.out.println("Hello");

            continue outer;

        }

        System.out.println("outer"); // Never prints

    }

    System.out.println("Good-Bye");

}

}

```

Sinavda dikkat edilmesi gereken noktalardan biri ; dongu disindaki etiket ismi ile break/continue sonrasındaki etiket isminin aynın olduguna dikkat etmek gerek.

Bir de donguden onceki etiket ile dongu arasında kod olmamalıdır. etiket:dongu(for/while/do)

## Pure Java - 47 Handling Exception - 01

[Levent Erguder](#) 30 March 2014 [Java SE](#)

Merhaba arkadaslar,

Bu bolumden itibaren Java da Exception mekanizmasini incelemeye baslayacagiz. OCP 6 notlari olmasina ragmen Java 7 ile birlikte gelen ozelligi de inceleyecegiz.

Error/hatayı bulma ve handle/yonetme islemi robust/güclü uygulamalar için çok önemli bir konudur. Exception, ortaya çıkan istisnai durumdur. Türkçe karşılık olarak istisna olarak da adlandırılır. Exception/hata/error kodun bir kısmında ortaya çıkan/cıkabilecek exceptional condition yani istisnai durumdur. Java da kavram olarak Exception ve Error farklı anlama gelmektedir. İlerleyen yazınlarda inceleyeceğiz.

Java da exception mekanizmasi icin su anahtar kelimeler kullanilir;

**try , catch , throw , throws , finally.**

#### **try – catch**

**try** , exceptionlara karsi korunmak ve onlari yonetmek icin, izlemek istedigimiz kodu, bir try blogu icine aliriz. **try** blogunun anlami “caution/dikkat” hata cikabilir.

**try** ve **catch** beraber calisir ortaklasa calisir. **catch** cumleciginin kapsamı oncesinde gelen **try** cumlecigindeki ifadelerle sinirlidir. **try** veya **catch** i tek basina kullanamayiz.

Genel format ;

```
try {  
  
} catch (FirstException ex) {  
  
} catch (SecondException ex) {  
  
} catch(ThirdException ex) {  
  
}
```

Ilk olarak basit bir ornek yapalim;

#### **Test1.java**

```
package purejava47;  
  
public class Test1 {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 0;  
  
        try {  
            System.out.println(a / b);  
            System.out.println("Bu kisim calismayacak.");  
        } catch (ArithmeticsException ex) {  
            System.out.println("0 a bolme hatasi.");  
            System.out.println(ex);  
        }  
        System.out.println("catch blogundan sonra");  
    }  
}
```

```
    }  
}
```

**try** blogu icerisinde a/b islemi **ArithmaticException** ‘ a neden olacaktir. Dikkat ederseniz bundan sonra **try** blogu calismaya devam etmeyecektir **catch** blogu calismaya devam edecektr. **catch** cumleciginin amaci , exceptionlari cozumlemek ve hic hata olmamis gibi programin devam etmesini saglamaktir. Giriste temel formatta belirtigim gibi , birden fazla **catch** kullanabiliriz.

### Test2.java

```
import java.io.*;  
  
public class Test2 {  
    public void readFromFile(String fileName) {  
        try {  
            FileReader fis = new FileReader(fileName);  
            System.out.println(fileName + " dosya bulundu");  
            char data = (char) fis.read();  
            System.out.println("read: " + data);  
        } catch (FileNotFoundException e) {  
            System.out.println("Dosya bulunamadi. " + e.getMessage());  
        } catch (IOException e) {  
            System.out.println("IOException meydana geldi.");  
            e.printStackTrace();  
        }  
        System.out.println("catch sonrasi");  
    }  
  
    public static void main(String[] args) {  
        Test2 reader = new Test2();  
        reader.readFile("mydata.txt");  
        System.out.println("main sonu");  
    }  
}
```

Sonraki yazılarda farkli acilardan tekrar try-catch’e deginecegim. Simdilik **finally** ile devam edelim.

## **finally**

**try-catch** mekanizmasini handling exception / hatalari yonetmek icin kullaniyoruz. Dusunelim, bir metotta dosya okuma islemi yapiyoruz ya da veritabanina baglanti yapiyoruz bir sorun olsa bile dosyanin veya veritabanini baglantisinin kapatilmasini isteriz. Bu durumda kapatma kodunu hem normal kod akisi içerisinde hem de catch içerisinde koymamiz gereklidir, bunun yerine **finally** anahtar kelimesini kullanarak bu sorunu cozabiliriz. **finally**, try/catch blogundan sonra kullanilabilir ya da catch olmadan try ile birlikte kullanilabilir fakat tek basina kullanilamaz. finally den sonra da catch kullanilamaz.

```
try {  
  
} catch(MyFirstException) {  
  
}  
catch(MySecondException) {  
  
}  
finally {  
  
}
```

finally blogu bir exception/hata fırlatilsa da fırlatilmasada ilgili catchte yakalansa da yakalanmasada(yani farklı turkde bir exception olabilir) calisacaktir. finally her zaman calisir kurali gecerlidir, tabi bir kac istisnasi mevcuttur. Bunlardan bahsedecegim.

Sinavda try-catch-finally yapisina dikkat etmek gereklidir. Hangi durumlarda derleme hatasi verecegini bilmek onemli.

```
// try-finally birlikte kullanilabilir.  
  
try {  
    // do stuff  
} finally {  
    //clean up  
}  
  
//finally , catchten sonra olamaz. Derleme hatasi  
try {  
    // do stuff
```

```

} catch (SomeException ex) {
    // do exception handling
} finally {
    // clean up
}

//try tek basina kullanilamaz
try {
    // do stuff
}
// catch veya finally gerekli
System.out.println("out of try block");

// derleme hatasi, try ile catch arasinda kod olamaz.
try {
    // do stuff
}
System.out.print("below the try");
catch(Exception ex) { }

```

### Test3.java

```

package purejava47;

public class Test3 {
    public static void main(String[] args) {

        System.out.println("#####test 1#####");

        try {
            System.out.println("try 1");
            throw new NullPointerException();
        } catch (NullPointerException e) {
            System.out.println("catch 1");
        } finally {
            System.out.println("finally 1");
        }
    }
}

```

```

        }

System.out.println("#####test 2#####");

try {
    System.out.println("try 2");
} catch (NullPointerException e) {
    System.out.println("catch 2"); // bu kisim calismayacak
} finally {
    System.out.println("finally 2"); // finally catch calissa da

}

System.out.println("#####test 3#####");

try {
    System.out.println("try 3");
    System.out.println(10 / 0);
} catch (NullPointerException e) {

} finally {
    System.out.println("finally 3"); // 10/0 islemi ArihtmeticException

}

}
}

```

Aciklamalari kod uzerinde yapmayacalistim. Simdi de su ornege bakalim ;

#### **Test4.java**

```
package purejava47;
```

```
public class Test4 {
```

```

private int getValue() {
    int value = 5;
    try {
        value = 20 / 0;
        return value;
    } catch (Exception e) {
        value = 10;
        System.out.println("catch");
        return value; // exception yakalanacaktir return olmasina ragmen finally calisacaktir.
    } finally {
        value = 50;
        System.out.println("finally");
        return value; //finally blogu calistigi icin en son olarak 50 degerini dondurecektir.
    }
}

public static void main(String[] args) {
    Test4 test = new Test4();
    System.out.println(test.getValue());
}

```

**finally** blogunun calismayacagi bir kac durum mevcuttur. Bunlarin bir tanesi exit() metodunun kullanimidir.

### Test5.java

```

package purejava47;

public class Test5 {
    public static void main(String[] args) {
        try {
            System.out.println("try");
            System.exit(0);
        }finally{
            System.out.println("calismayacak");
        }
    }
}

```

```

        System.out.println("calismayacak - 2");

    }

}

```

Bu bolumde try, catch ve finally anahtar kelimelerinin inceledik. Bu ilk bolumde Java da exception mekanizmasina hizli bir goz attik. Bu konudaki bilginize gore bol miktar ornek kod yazmaniz faydalı olur.

## Pure Java - 48 Handling Exception - 02

[Levent Erguder](#) 09 April 2014 [Java SE](#)

Merhaba Arkadaslar,

Bir onceki bolumde Java'da exception mekanizmasina giris yapmistik. Bu yazimda **ducked exception** kavramindan dolayisiyla Propagating Uncaught Exceptions kavramindan bahsedecegim.( yakalanamayan hatalarin yayilimi diye cevirmeye calisalim. )

### Propagating Uncaught Exceptions

try blogunda bir exception fırlatildiginda ve bu exceptiona karsilik uygun bir catch olmadigi durumda ne olacak ? Eger bir metot fırlatilan bu exception'i desteklemiyorsa (yani yakalayamiyor) bu duruma **ducked exception** denilir.

**Ducked exception** durumundan once stack mantigini anlamamız gereklidir.

Programımız *main()* metodunda baslar bu main metodu *a()* metodunu *a()* metodu *b()* metodunu ve *b()* metodu da *c()* metodunu çağırırsa **stack** yapısı şu şekilde olacaktır;

```

c
b
a
main

```

**Ducked exception** mekanizması da aynı yapıda çalışacaktır.

4 katlı bir bina düşünelim, bina yukarıdan aşağıya coktugunde 4.kat 3.katin, 3.kat 2.katin üzerine 2.kat da 1.katin üzerine dusecektir.

Benzer şekilde *c()* metodundan bir exception fırlatıldığında bu exception sırası ile *b*, *a* ve *main* metoduna ulaşacaktır sonuc olarak uygulamamız binanın tuz buz olması gibi patlayacaktır.

### DuckedException.java

```

public class DuckedException {
    public static void main(String[] args) {
        doStuff();
        System.out.println("Uygulamamız patladığı için bu çıktı ekrana basılmaz.");
    }

    static void doStuff() {
        doMoreStuff();
    }
}

```

```

static void doMoreStuff() {
    int x = 5 / 0; //  

    // ArithmeticException hatasi firlatilacaktir.  

}  

}

```

*Exception in thread “main” java.lang.ArithmaticException: / by zero  
at purejava48.DuckedException.doMoreStuff(DuckedException.java:14)  
at purejava48.DuckedException.doStuff(DuckedException.java:10)  
at purejava48.DuckedException.main(DuckedException.java:5)*

main metodu doStuff() metodunu , doStuff() da doMoreStuff() metodunu cagirmakta.Yani stack yapisi;

*doMoreStuff()  
doStuff()  
main()  
doMoreStuff() metodunda 5/0 islemi ArithmaticException'a neden olur ve bu exception doMoreStuff() metodundan doStuff() metoduna oradan da main() metoduna firlatilir.Bu metodlarin hic birinde gonderilen hatayı yakalayacak bir catch yapısı olmadığı için uygulama dibe vuracak ve patlayacaktır.*System.out.println* satırımız çalışmayaçaktır.*  
4.kat 3.katin uzerine dustugunde eger 3.katimiz saglamsa binamiz cokmeyecektir. Benzer sekilde ducked exception mekanizmasında firlatilan hatayı yakalarsak uygulamamizi patlamaktan kurtarabiliriz.

```

public class DuckedException {  

    public static void main(String[] args) {  

        doStuff();  

        System.out.println("Hata yakalandi, uygulama calismaya devam eder.");  

    }  

    static void doStuff() {  

        try{  

            doMoreStuff();  

        }catch(Exception e) {  

        }  

    }  

}

```

```
static void doMoreStuff() {  
    int x = 5 / 0; //  
    // ArithmeticException hatasi fırlatılacaktır.  
}  
}
```

## Pure Java - 49 Handling Exception - 03

[Levent Erguder](#) 19 April 2014 [Java SE](#)

Merhaba Arkadaşlar,

Bu yazimda Java da exception mekanizmasina devam edecegim.

### Exception Hiyerarsisi

Java 'da primitif turler disinda her sey obje olmalıdır. Dolayisiyla Java'daki tüm exceptionlar da objedir. Java'da tüm exceptionlar `java.lang.Exception` sınıfının alt sınıfıdır.

// resim sitede yok.

Resme dikkat edecek olursak `Throwable` sınıfı iki ana kola ayrılıyor

; **Error** ve **Exception**. **Unchecked** ve **Checked exception** kavramını ilerleyen yazılarla yazacağım.

**Error'lar** beklenmedik hata durumlarında fırlatılır.

Ornegin meshur;

`java.lang.OutOfMemoryError: PermGen space`

Genel olarak uygulamamız **Error** durumlarını kurtarmak/duzeltmek gibi bir eğilim göstermez. Bu nedenle **Error'ları** handle/yakalamamız yönetmemiz gerekmektedir.

**Error'lar** hiyerarsi geregi **Exception** değildir. Bu detay aklımızda olsun.

**Throwable** sınıfının bize sağladığı `printStackTrace` metodunu catch içerisinde kullanabiliriz. Bu sayede try blogu içerisinde olusan hatalının(exception/error turu) bilgisini elde edebiliriz.

### **PrintStackTraceTest.java**

```
package purejava49;  
  
public class PrintStackTraceTest {  
    public static void main(String[] args) {  
        try {
```

```

        throw new ArrayIndexOutOfBoundsException();

    } catch (Exception e) {
        e.printStackTrace();
    }

    try {
        throw new OutOfMemoryError();
    } catch (Error e) {
        e.printStackTrace();
    }

    try {
        throw new RuntimeException();
    } catch (Throwable e) {
        e.printStackTrace();
    }
}

```

Eclipse'te , Console da ;

```

java.lang.ArrayIndexOutOfBoundsException
    at purejava49.PrintStackTraceTest.main(PrintStackTraceTest.java:6)

java.lang.OutOfMemoryError
    at purejava49.PrintStackTraceTest.main(PrintStackTraceTest.java:11)

java.lang.RuntimeException
    at purejava49.PrintStackTraceTest.main(PrintStackTraceTest.java:17)

```

tek bir catch ifadesinde birden fazla exception turu yakalayabiliriz.

catch(Exception e)

seklinde yazdigimda firlatilan tum exception turlerini yakalayabiliriz, fakat firlatilabilecek exceptionlari ozel olarak belirtmek isteyebiliriz. Bu durumda birden fazla catch ifadesi kullanabiliriz.

**ExceptionMatching.java**

```

package purejava49;

import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

public class ExceptionMatching {

    public static void main(String args[]) {
        try {
            RandomAccessFile raf = new RandomAccessFile("myfile.txt", "r");
            byte b[] = new byte[1000];
            raf.readFully(b, 0, 1000);

        } catch (FileNotFoundException e) {
            System.err.println("File not found");
            System.err.println(e.getMessage());
            e.printStackTrace();
        } catch (IOException e) {
            System.err.println("IO Error");
            System.err.println(e.toString());
            e.printStackTrace();
        }
    }
}

```

Ornegimizi calistirdigimizda ,

```

File not found

myfile.txt (No such file or directory)

java.io.FileNotFoundException: myfile.txt (No such file or directory)

at java.io.RandomAccessFile.open(Native Method)
at java.io.RandomAccessFile.<init>(RandomAccessFile.java:241)
at java.io.RandomAccessFile.<init>(RandomAccessFile.java:122)

```

```
at purejava49.ExceptionMatching.main(ExceptionMatching.java:11)
```

Burada dikkat etmemiz gereken nokta ; **IOException** , **FileNotFoundException** 'in super sınıfıdır. catch mekanizmasında önce **FileNotFoundException** i yazmamız gereklidir.

```
public class FileNotFoundException extends IOException {  
    ...  
}
```

Daha genel olak exception turu yani super sınıf daha aşağıdaki catch içerisinde yazılmalıdır.

```
} catch (FileNotFoundException e) {  
    ...  
}  
} catch (IOException e) {  
    ...  
}
```

Yerine şu şekilde yazarsak derleme hatalı olur.

```
} catch (IOException e) {  
    ..  
}  
} catch (FileNotFoundException e) {  
    ..  
}
```

## Pure Java – 50 Handling Exception – 04

[Levent Erguder](#) 20 September 2014 [Java SE](#)

Merhaba Arkadaşlar,

Bu bölümde Java'nın handle or declare kuralını inceleyeceğiz. Java'nın exception mekanizmasında **checked** ve **unchecked** exception olmak üzere 2 tür exception mevcuttur. Onceki bölümde **Exception** ve **Error** kavramının aslında birbirinden farklı olduğunu ve bu sınıfların **Throwable** sınıfını kalıttığını gördük. **Runtime exception(calisma zamani)**'lar ve tüm **Error**'lar(**java.lang.Error** sınıfını kalanan tüm error tipleri) **unchecked exception**dir.

**Checked** exceptionlar isin isine girdiginde bu durumda handle or declare kuralına uymamız gerekmektedir.

## MyException.java

```
import java.io.FileNotFoundException;
import java.io.IOException;

public class MyException {

    void giveMeException() throws IOException {
        try {
            throw new FileNotFoundException();
        } catch (FileNotFoundException e) {
            System.out.println("Exception catch");
        }
        throw new IOException();
    }
}
```

MyException sınıfında giveMeException adında bir metot tanımladık.

FileNotFoundException , IOException sınıfının bir alt sınıfıdır. IOExpception ve alt sınıf tipindeki exception türlerinin hepsi checked exceptiondir. Dolayısıyla bu exception tiplerinin fırlatılacağı durumlarda Java bizi bu hata üzerinde zorlar.

Yani ya bu tehlikeli hata kodunu try-catch blogu arasında almamızı ya da metot tanımında **throws** anahtar kelimesini kullanarak bu riskli durumu(exception fırlatılma ihtimalını) bildirmemiz gereklidir.

Iste burada olduğu gibi try-catch blogu arasına **handle** ,**throws** anahtar kelimesini kullanarak metotta tanıtlamaya **declare** denilir. Kuralımız basit söz konusu checked exception ise ya **handle** ya **declare** !

**Checked** exception fırlatan bir metodu çağırduğumuzda **handle-daclere** kuralına uymak zorundayız. Bu durumda bu kurala uygun şekilde ya throws anahtar kelimesini kullanmalı ya da try-catch blogunda yakalamamız gereklidir.

```
void callExceptionMethod() {
    //checked exception fırlatan bir metodu çağırduğumuzda handle-daclere kuralına uymak zorundayız.
    //burada derleme hatası verecektir.
    //giveMeException();

}
```

Handle – Declare kuralı unchecked exceptionlar için zorunlu değildir !

## MyException2.java

```
public class MyException2 {  
  
    public static void main(String[] args) {  
        giveMeUncheckedException3();  
    }  
  
    void giveMeUncheckedException() {  
        throw new RuntimeException();  
    }  
  
    void giveMeUncheckedException2() {  
        throw new ArithmeticException();  
    }  
  
    static void giveMeUncheckedException3() {  
        // java.lang.ArithmaticException:  
        int a = 5 / 0;  
    }  
}
```

ArithmaticException , RuntimeException sinifinin bir alt sinifidir. RuntimeException tipinde bir exception unchecked exceptiondir. Dolayiyisla bu kodlari try-catch blogu arasina almamiza ya da throws metoduyla metod taniminda yazmamiz icin Java bizi zorunlu tutmaz.

Hatirlayacagimiz gibi **java.lang.Error** ve alt sinifi tipinde tum errorlar **unchecked exception**'tir. Dolayisiyla handle – declare kuralina uymak zorunda degiliz.

## MyError.java

```
public class MyError {  
  
    void giveMeError() {  
        throw new StackOverflowError();  
    }  
  
    void giveMeAnotherError() {  
        throw new OutOfMemoryError();  
    }  
}
```

```

    }

    void callErrors() {
        giveMeError();
        giveMeAnotherError();
    }
}

```

Son olarak catch blogunda yakaladigimiz bir exception'i tekrar firlatabiliriz;

### Rethrow.java

```

public class Rethrow {

    void throwMeAgain() throws IOException {
        try {
            throw new IOException();
        } catch (IOException e) {
            throw e;
        }
    }
}

```

*throwMeAgain* metodunda yeni bir **IOException** exception'i fırlattık ve yakaladık. Aynı exception'i **catch** blogunda tekrar fırlatıyoruz. Dolayısıyla burada **IOException** , **checked exception** olduğu için metodumuzda **throws** anahtar kelimesini kullanarak **IOException**'i declare etmemiz gereklidir.

## Pure Java – 51 Handling Exception – 05

[Levent Erguder](#) 28 September 2014 [Java SE](#)

Merhaba Arkadaşlar,

Bu yazıyla birlikte Java Exception konusunu bitirecegiz.

Onceki yazılardan hatırladığımız gibi Java Exception mekanizmasında 2 tur exception vardı.  
 Bunlar **checked** exception ve **unchecked** exception'dı. Hatırlayacağımız gibi Java'da error ve exception teknik acıdan farklı kavramlardır. **java.lang.Error** sınıfı tipinde tüm exceptionlar/hatalar , error tipindedir. **java.lang.Exception** sınıfı tipinde tüm exception/hatalar , exception tipindedir.  
 Java'da **java.lang.RuntimeException** sınıfı tipinde ve alt sınıfı tipinde tüm exception'lar unchecked

exception'dir. Bununla birlikte tum errorlar yani **java.lang.Error** tipinde veya alt sinif tipinde tum errorlar da **unchecked** exception'dir.

Hatirlayacagimiz gibi checked exception'lar, *hande or declare* kuralina uymak zorundaydi. Unchecked exceptionlar( Error ve RuntimeException) ise bu kurala uymak zorunda degildi , fakat dilersek bu unchecked exceptionlari da yakalayabiliriz. Bununla birlikte bir error'u yakalamak yararsiz/gereksiz olacaktir. Bir error'un sonucu duzeltmek/kurtarmak cogu zaman imkansizdir. Benzer sekilde RuntimeException tipinde uncheked exception'lari da yakalabiliriz fakat bu durum da kotu bir programlama dizayni olacaktir.

Unchecked exception'lar icin tercih edilen yakalanmasi degildir ! Birakin programinizi crash etsin (kisrin/exception firlatilsin). Unchecked exception'i yakalamak yerine hataya neden olan kismi cozmek kodu modifiye etmek daha dogru bir yaklasimdir.

Ornegin, NullPointerException 'i yakalamak yerine ilgili yerde null kontrolu yapmak dogru bir yaklasimdir.

### Test.java

```
public class Test {  
  
    public static void main(String[] args) {  
  
        String str = getValue();  
  
        // NullPointerException icin try/catch kontrolu dogru bir yaklasim degildir.  
        try {  
            if (str.contains("test")) {  
                //  
            }  
        } catch (NullPointerException npe) {  
            System.out.println("NullPointerException is catched");  
        }  
  
        // Bu sekilde kullanım daha dogru bir yaklasimdir!  
        if (str != null && str.contains("test")) {  
            //  
        }  
  
        static String getValue() {  
    }
```

```
        return null;  
    }  
}
```

## Pure Java – 52 Assertion Mechanism

[Levent Erguder](#) 04 October 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda Java'da yer alan *Assertion* mekanizmasından bahsedecegim.

Java 1.4 ile gelen “**assert**” bir anahtar kelimedir.

Bir metoda arguman olarak sayı gecirdigimizi dusunelim, (methodA()) fakat bu sayı negatif bir sayı olmamalidir. Bu kontrolu normal şartlarda su sekilde yapabiliyoruz ;

```
private void methodA(int num) {  
    if (num >= 0) {  
        useNum(num + x);  
    } else { // num must be < 0  
        // This code should never be reached!  
        System.out.println("Yikes! num is a negative number! "  
            + num);  
    }  
}
```

methodA() içi validate işlemini assertion mekanizması ile yapabiliyoruz ;

```
private void methodA(int num) {  
    assert(num > 0);  
    useNum(num + x);  
}
```

Assertion mekanizması gayet basit olarak çalışır. İlgili assertion şartı true olmazsa exception(error) fırlatılacaktır(**AssertionError**).

Eğer true donecekse her şey yolunda olacak ve kod akışı devam edecektir

Assertion mekanizması 2 formatta yazılabilir ; simple , really simple

Really simple ;

```
private void doStuff() {  
    assert (y > x);  
    // more code assuming y is greater than x  
}
```

Simple;

```
private void doStuff() {  
    assert (y > x); "y is " + y + " x is " + x;  
    // more code assuming y is greater than x  
}
```

### Assertion Expression Rules/Kurallari

Assertion mekanizmasi bir ya da iki expression'a sahip olabilir. **Simple** ya da **Really Simple** olmasina gore. Birinci expression'in sonucu mutlaka boolean olmalıdır! Ikinci expression sonuc donduren(value) herhangi bir expression olabilir. Simple assertion ornegimizde ikinci expression String ; bu bir nevi System.out.println() gibi calisacaktir.

```
void noReturn() {}  
  
int aReturn() { return 1; }  
  
void go() {  
    int x = 1;  
  
    boolean b = true;  
  
    // bu assert statement'lari legaldir.  
  
    assert(x == 1);  
    assert(b);  
    assert(true);  
    assert(x == 1) :x;  
    assert(x == 1) :aReturn();  
    assert(x == 1) :new ValidAssert();  
  
    // ILLEGAL assert statement'lari  
    assert(x = 1); // boolean degil,  
    assert(x); // boolean degil  
    assert 0; // boolean degil  
    assert(x == 1) :; // bonus degeri(value) yok
```

```
assert(x == 1) : noReturn(); //donus degeri(value) yok  
assert(x == 1) : ValidAssert va; // donus degeri(value) yok  
}
```

### Enabling Assertions

Assertion mekanizmasini kullanabilmek icin oncelikle enable/aktif hale nasil getirebilecegimizi bilmemiz gerekmektedir.

Hatirlayacagimiz gibi assert anahtar kelimesi Java 1.4 ile birlikte gelmistir. Java 1.4 ten once bu kelime bir identifier olarak kullanilabilir.

```
// Java 1.4 oncesi  
  
int assert = getInitialValue();  
  
if (assert == getActualResult()) {  
  
// do something  
}
```

### Test.class

```
public class TestClass {  
  
    public static void main(String[] args) {  
  
        int x=1;  
  
        assert(x==2);  
  
        System.out.println("Assert Mekanizmasi aktif degil!");  
    }  
}
```

TestClass’imizi komut satirinda derleyelim ve Assert mekanizmasinin enabled/disabled oldugu durumlara gore bakalim;

```

levent@ekmekTeknesi: ~/Desktop
levent@ekmekTeknesi:~/Desktop$ javac TestClass.java
levent@ekmekTeknesi:~/Desktop$ java TestClass
Assert Mekanizmasi aktif degil!
levent@ekmekTeknesi:~/Desktop$ java -ea TestClass
Exception in thread "main" java.lang.AssertionError
        at TestClass.main(TestClass.java:6)
levent@ekmekTeknesi:~/Desktop$ java -enableassertions TestClass
Exception in thread "main" java.lang.AssertionError
        at TestClass.main(TestClass.java:6)
levent@ekmekTeknesi:~/Desktop$ 

```

Assertion mekanizmasi varsayılan/default olarak kapalıdır(disabled) , aktif hale getirmek için java komutu ile birlikte -ea ya da -enableassertions ifadesini kullanmalıyız. Assertion mekanizmasının kapalı olduğu durumda(disabled) , assert kontrolü pas geçildi ve false assert kontrolü nedeniyle error(AssertError) fırlatılmadı fakat assert kontrolünün pas geçilmemiş durumlarda yani assertion mekanizmasının aktif edildiği durumda false sonuc AssertionError fırlatılmasına neden oldu.

Assertion mekanizmasını paket seviyesinde de enabled/disabled yapabiliriz.

Command-Line Example	What It Means
java -ea java -enableassertions	Enable assertions.
java -da java -disableassertions	Disable assertions (the default behavior of Java 6).
java -ea:com.foo.Bar	Enable assertions in class com.foo.Bar.
java -ea:com.foo...	Enable assertions in package com.foo and any of its subpackages.
java -ea -dsa	Enable assertions in general, but disable assertions in system classes.
java -ea -da:com.foo...	Enable assertions in general, but disable assertions in package com.foo and any of its subpackages.

### Using Assertions Appropriately

Bu konu başlığında , Assertions mekanizmasının uygun yerlerde kullanımı inceliyeceğiz.

*Assertion mekanizmasını public metodlarla kullanmak uygun değildir.(inappropriate)*

```

public void doStuff(int x) {
    assert (x > 0); // inappropriate
}

```

Assertion mekanizmasını private metodlarla kullanmak uygundur.(appropriate)

```
private void doStuff(int x) {  
    assert (x > 0); // appropriate  
}
```

*Assertion mekanizmasini Command-line argumanlari icin kullanmak uygun degildir( inappropriate)  
Assertion mekanizmasini , metodumuz public bile olsa kodumuzun kontrol edilmeyen(check) senaryolari icin kullanmak uygundur(appropriate) .  
Ornek switch kod yapisinda , default kismina hic ulasilmamali. Bu konuda bir assert statement'i yazilabilir.*

```
switch(x) {  
    case 1: y = 3; break;  
    case 2: y = 9; break;  
    case 3: y = 27; break;  
  
    default: assert false; // we're never supposed to get here!  
}
```

*Assertion mekanizmasini side effect(yan tesir) gosterecek sekilde kullanmak uygun degildir.*

```
public void doStuff() {  
    assert (modifyThings());  
    // continues on  
}  
  
public boolean modifyThings() {  
    y = x++;  
    return true;  
}
```

*assert mekanizmasinin her zaman calisacagini bir garantisи yoktur. Bu nedenle burada oldugu gibi icerisinden bir metot cagirmak ve side effect yapacak sekilde kullanmak uygun degildir. Assertion mekanizmasinin yan etkisi (side effect) olmamalidir.*

## BÖLÜM-6

### Pure Java - 53 String - 01

[Levent Erguder](#) 11 October 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazı ile birlikte bolum 6'ya basliyoruz. Bundan sonra konular daha da keyifli hale gelecek.

String'ler hakkında hep soylenen bir kavram vardır ; **immutable** peki nedir bu String'lerin immutable olması ?

### String'in immutable olma özelliği

Öncelikle String'ler hakkında temel bilgileri öğrenelim. String mantığı Java'ya özgü değildir. Diğer programlama dillerinde de String kavramı söz konusudur.

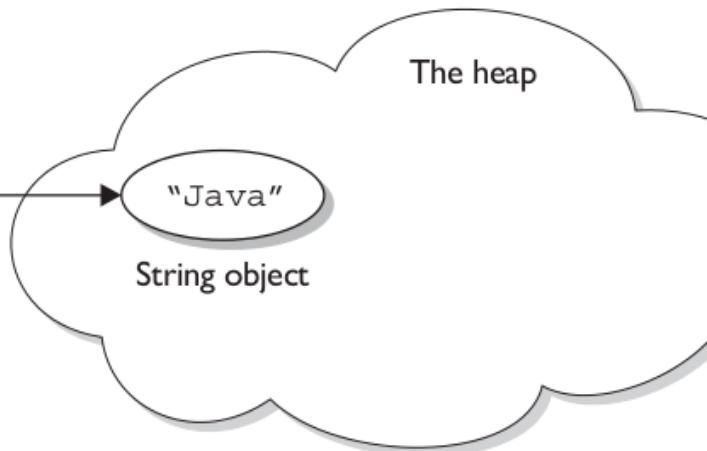
- Java'da String'in her karakteri 16-bit Unicode bir karakterdir. Bu sayede farklı karakterin gösterimi kolaylıkla yapılabilir.
- Java'da , String'ler objedir ! Unutmayalım bizim sadece 8 tane primitive tipimiz var. Bunlar ; byte, short , int , long , float , double , boolean , char dir. String primitive type değildir !

String'lerin Immutable özelliğini anlayabilmek adına örneklerimizi inceleyelim.

```
String x = "Java";
```

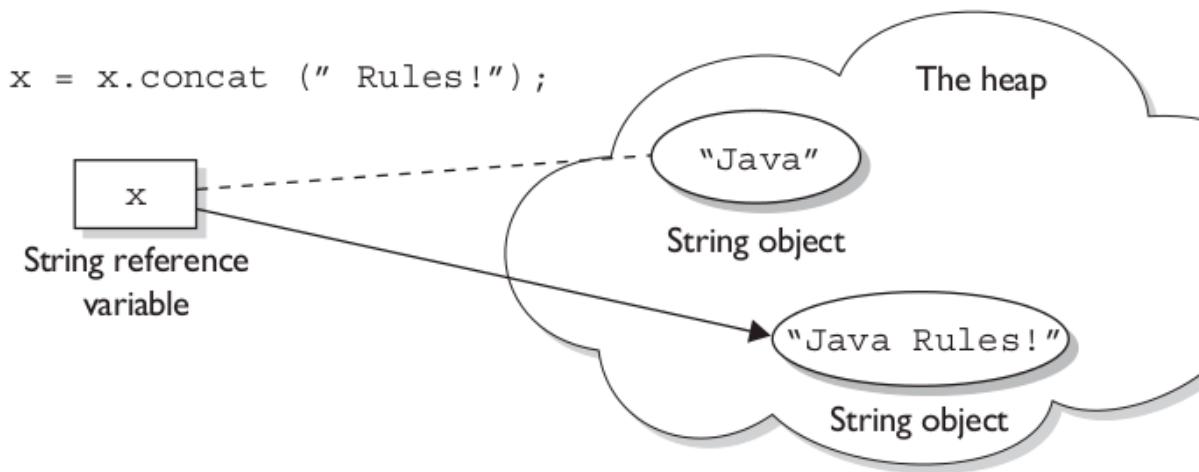
```
String x = "Java";
```

**String reference variable**



**Bu noktaya dikkat edelim !**

```
x = x.concat(" Rules");
```



`x` referans degiskeni , String objesini referans gostermektedir. Bu String objesinin degeri "Java" idi.

```
x = x.concat(" Rules");
```

İlgili "Java" objesinin degerini "Java Rules" olarak degistirmedi ! Bunun yerine "Java Rules" degerine sahip yeni bir String objesi olusturuldu. Iste, Strinlerin immutable yani degismez olma ozelligi bu anlamda gelmektedir.

String'in **immutable** olmasi demek , String objelerinin degistirilemez olmasidir. Tabi ki String referans degiskenenin referansta bulundugu obje degisebilir.

### Test.java

```
public class Test {
    public static void main(String[] args) {
        String x = "Java"; // "Java" degerine sahip String objesi olusur.
        x = x.concat(" Rules"); // "Java Rules" degerine sahip yeni bir String objesi olusur.
        // "Java" objesi hala heapde yasamaktadir.

        System.out.println("x = " + x); // Java Rules
        x.toUpperCase(); //Yeni bir obje olusur fakat "JAVA RULES"
        // Fakat x referans degiskeni "Java Rules" objesini referans gostermektedir.
        System.out.println("x = " + x);

    }
}
```

```
}
```

## Test2.java

```
public class Test2 {  
    public static void main(String[] args) {  
  
        String s1 = "spring ";  
        // 1)"spring" degerine sahip bir String objesi olusur.  
  
        String s2 = s1 + "summer ";  
        // 2)"summer" objesi olusur  
        // 3)"spring summer" objesi olusur  
        // s2 referans degiskeni "spring summer" objesine referansta bulunur  
  
        s1.concat("fall ");  
        //4) "fall" degerine sahip String objesi olusur.  
        //5) "spring fall" degerine sahip String objesi olusur. s1 in referans gosterdigi obje degismez !  
  
        s2.concat(s1);  
        //6) "spring summer spring" objesi olusur.  
  
        s1 += "winter ";  
        //7) "winter" degerine sahip obje olusur.  
        //8)"spring winter" degerine sahip obje olusur.  
  
        System.out.println(s1 + " " + s2);  
  
    }  
}
```

## String & Memory

Bu bolumde String objelerinin Memory'deki(hafiza) durumuna inceleyecegiz. Programlama dillerinin amaclarindan birtanesi memory/hafizayi verimli(efficient) sekilde kullanmaktir.

Uygulamalar(application/proje) buyudukce String literalleri memoryde buyuk bir yer kaplayacak duruma gelir. Java bu konuda memory verimliliği(efficient) saglamak adina ozel bir memory kullanir. Bu ozel alana “String Constant Pool” denilir !

Compiler , String literali ile karsilastiginda, identical/ayni/ozdes String literali String Pool'da olup olmadigi kontrol edilir. Eger identical/ozdes String literali , String Pool'da bulunursa yeni bir String literal objesi String Pool'da olusmaz !

Ornegimizi inceleyelim ; String Poolda 2 tane obje olusur bunlar ; "levent" ve "erguder" objesidir. str ve str2 referans degiskeni "levent" objesini referans etmektedir. str3 referans degiskeni ise "erguder" referans degiskenenine referans etmektedir.

Unutmayalim , str2 'nin referans ettigi String objesinin degeri "levent" oldugu icin ve String Pool'da bu String degeri mevcut oldugu icin ( str ="levent" degerini ilk olarak tanimladik ) yeni bir obje olusmaz!

```
String str = "levent";
String str2 ="levent";
String str3 = "erguder";
String s ="one";
// 1 obje olusur

String s2 = new String("two");
// new anahtar kelimesi ile String objesi olusturdugumuz zaman 2 tane
// obje olusur.
// 1) Non-pool memory (normal heap alani) de yeni bir obje olusur.
// 2) String Pool'da two objesi olusur.
```

## Pure Java - 54 String - 02 Methods

[Levent Erguder](#) 14 October 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazida Java'da String sinifina ait metotlari inceleyecegiz.String sinifina ait olan bu metotlari bol miktar kullanma ihtiyaci hissederiz bu nedenle bunlari iyi ogrenmek son derece faydalı olacaktır.

### length()

```
public int length() {...}
```

length() metodu String'in kac karakterden olustugu bilgisini doner. Basit bir ornek yapalim;

```
public class StringTest {
```

```
public static void main(String[] args) {  
    String name = "levent";  
    int len = name.length();  
    System.out.println("karakter sayisi:" + len);  
}  
}
```

length() metodu bize 6 degerini donecektir.

### isEmpty()

```
public boolean isEmpty()
```

isEmpty metodu bize String uzunlugu 0 ise true degeri donecektir. Yani String'in bos oldugu durumda. Bu metot bazen kontrol amacli cok faydalı olabilmektedir.

```
public class IsEmptyTest {  
  
    public static void main(String[] args) {  
        String str = "";  
        boolean isEmptyString = str.isEmpty();  
        System.out.println(isEmptyString);  
    }  
}
```

### charAt

```
public char charAt(int index)
```

charAt metodu belirtilmis indeks(specified index) degerindeki karakteri dondurur.

```
public class CharAtTest {  
  
    public static void main(String[] args) {  
        String name = "levent";  
        char c = name.charAt(2);  
    }  
}
```

```

        System.out.println(c);

        // char c2 = name.charAt(10);

        // java.lang.StringIndexOutOfBoundsException

    }

}

```

Java'da index degerlerinin 0 dan basladigini unutmayalim 2.index → 3.karakter olacaktir bu da v harfidir. Index degeri olarak uygun bir deger vermezsek bu durumda *StringIndexOutOfBoundsException* hatasi aliriz.

#### **codePointAt**

```
public int codePointAt(int index) { ... }
```

codePointAt metodu , String literali icerisinde, index'i verilen karakterin Unicode degerini bize dondurur.

```
public class CodePointAt {
```

```

    public static void main(String[] args) {
        System.out.println("a".codePointAt(0));
    }
}
```

#### **getChars**

```
public void getChars(int srcBegin, int srcEnd, char dst[], int dstBegin) { ... }
```

srcBegin , kopyalanacak String literalin baslangic indexi (dahil)

srcEnd , kopyalanacak String literalin son karakteri(haric)

dst , kopyalanacak array

dstBegin , kopyalanacak array icin, kopyalamaya baslanacak index

```
public class GetCharsTest {
```

```
    public static void main(String args[]) {
```

```
        String str = new String("injavawetrust");
```

```
        char[] charArray = new char[4];
```

```
//injavawetrust icin, 2.index baslangic -->j
```

```

        // 6.karakter dahil degil --> bu durumda ilgili string java olacaktir.

        //charArray e kopyalanacaktir, 0.indexten baslanacaktir.

        str.getChars(2, 6, charArray, 0);

        System.out.println(charArray);

    }

}

```

### getBytes

```

public byte[] getBytes() {..}

public byte[] getBytes(String charsetName)

throws UnsupportedEncodingException { ... }

public byte[] getBytes(Charset charset) {...}

```

getBytes metodunun overloaded versiyonları bulunmaktadır. String literaldeki karakterleri byte dizisine çevirir. Overloaded versiyonlarında bu encoding işlemi için ilgili Charset'i kullanır.

```

public class GetBytesTest {

    public static void main(String[] args) {
        String name = "levent erguder";

        byte bytes[] = name.getBytes();

        for (byte b : bytes) {
            System.out.println(b);
        }
    }
}

```

### compareTo & compareToIgnoreCase

```

public int compareToIgnoreCase(String str) {...}

public int compareTo(String anotherString) {...}

```

compareTo metodu lexicographically(sozluksel bicimde) karsilastirma yapmamizi saglar. Karsilastirma islemi Unicode degerine gore olmaktadır.

Sonuc negatif sayi ise soldaki String sagdaki Stringten lexicographically(sozluksel bicimde) olarak daha onceadir.

Sonuc pozitif sayi ise soldaki String sagdaki Strinden daha sonradir.

Sonuc 0 ise ayni String degere sahiptir.(equal)

```
public class CompareToTest {  
  
    public static void main(String[] args) {  
  
        System.out.println("a".compareTo("d"));  
  
        System.out.println("b".compareTo("d"));  
  
        System.out.println("e".compareTo("a"));  
  
        System.out.println("ABa".compareTo("ABC"));  
  
        System.out.println("a".compareToIgnoreCase("A"));  
  
  
        System.out.println("I".compareTo("e"));  
  
        System.out.println("levent".compareTo("erguder"));  
  
  
        // ilk olarak l ve e arasinda farklilik oldugu icin unicode olarak aradaki fark donecektir  
    }  
}
```

**NOT : String sinifinda yer alan bir cok metodu burada aciklamaya calisicam. Bu yazi metotlar bitene kadar guncellenecektir.**

**Sinavda sorumlu olunan metotlar cok daha azdir. Fakat gercek hayatta daha cok metoda ihtiyacimiz var bu nedele mumkun oldugunda hepsini paylasicam insallah.**

## Pure Java – 55 StringBuffer & StringBuilder

[Levent Erguder](#) 17 October 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda **java.lang.StringBuffer** ve **java.lang.StringBuilder** siniflarinda bahsedecegim. Onceki yazilardan hatirlayacagimiz gibi **String** objeleri **immutable** ozellige sahipti. String objeleri ile cok fazla degisiklik yapmamiz(manipulation) sonucunda yeni objeler olusacaktir ve bu olusan objelerin buyuk bir kismi **String Pool**da doldurulmis objeler olarak kalacaktir.(yani referans gosteren bir degiskeni olmayacaktir). String objeleri uzerinden cok fazla degisiklik yapilacak zaman String sinifini kullanmak yerine StringBuffer veya StringBuilder siniflarini kullanmak verimlilik saglayacaktir. StringBuffer ve StringBuilder objeleri immutable degildir dolayisiyla bu objeler uzerinde degisiklik yapilabilir.

## **StringBuffer vs StringBuilder**

StrinBuffer ve StringBuilder ayni metotlara sahiptir . Bu siniflar arasindaki fark sudur ; StringBuilder thread-safe degildir yani **StringBuilder** sinifinin metotlari **synchronized** degildir. Dolayisiyla StringBuilder , StringBuffer sinifa gore daha hizli calisacaktir.

## **StringBuffer & StringBuilder vs String**

String'in immutable ozelligini incelemistik , ornegin ;

```
String x = "abc";
x.concat("def");
System.out.println("x = " + x); // cikti x = abc olacaktir
```

Hatirlayacagimiz gibi bu kod orneginde ;

- 1) "abc" objesi olustu ve x referans degiskeni bu objeyi gostermeyecektir.
- 2) "def" objesi olustu fakat terkedilmis durumda (abandoned)
- 3) concat metodu sonucu "abcdef" objesi olustu fakat burada x degiskene assign(atama) yapilmadigi icin bu "abcdef" objesi de terkedilmis durumdadır(abandoned)

Biz sadece 1 tek objeyi referans gostermektedir fakat terkedilmis olarak 2 tane obje var bu verimli bir durum degildir.(wasting memory)

Simdi benzer durumu StringBuffer icin yapalim. (StringBuilder da olur)

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb); //cikti sb = abcdef olacaktir
```

Burada "abc" degerine sahip **StringBuffer** objesi olustu , append metodu sonrasında ilgili metodumuz "abcdef" degerine sahip oldu. Burada String orneginde oldugu gibi ne "def" degerine sahip obje olustu ne de yeni bir "abcdef" objesi olustu. Sadece 1 objemiz var bu objemizi modifiye ettik !

## **StringBuffer & StringBuilder sinifi metotlari**

### **append metodu**

append metodunun bir cok overloaded hali bulunmaktadır. Yukarida bahsettigim gibi StringBuffer sinifi thread-safe'tir dolayisiyla metotlari synchronized'dir. Ornek bir append metodu;

```
public synchronized StringBuffer append(String str) { ... }
```

append metodu, ilgili StringBuffer ya da StringBuilder objesinin degerine ekleme yapacaktır.

```
public class AppendTest {
```

```

public static void main(String[] args) {
    StringBuffer sb = new StringBuffer("levent");
    sb.append(" erguder ");
    sb.append(1989);

    System.out.println(sb);
}

}

```

### **delete metodu**

```
public synchronized StringBuffer delete(int start, int end) { ... }
```

delete metodu, StringBuffer veya StringBuilder'in degerini (start-end] araliginda silecektir.

```

public class DeleteTest {

    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("0123456789");
        sb.delete(4, 6);
        // index 0 dan baslayacaktir!
        // baslangic indexi dahil,(4)
        // bitis indexi dahil degildir (6)

        System.out.println(sb);
    }
}

```

### **insert metodu**

insert metodunun da append metodu gibi bir cok overloaded hali mevcuttur. append metodu ilgili  
StringBuilder/StringBuffer'in sonuna ekliyordu, insert metodu yardimi ile diledigimiz index alanindan itibaren  
bu ekleme islemiini yapabiliyoruz.

```

public StringBuffer insert(int offset, int i) { ... }

public class InsertTest {

```

```

public static void main(String[] args) {
    StringBuilder sb = new StringBuilder("0123456789");
    sb.insert(4, "----"); // 4.indexten sonra ekleme yap!(insert)
    System.out.println(sb);

}
}

```

### **reverse metodu**

reverse metodu , StringBuffer/StringBuilder objesinin degerini ters cevirir. Belki bir yerde isimize yarar =)

```

public synchronized StringBuffer reverse() {...}

public class ReverseTest {

    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("levent erguder");
        sb.reverse();
        System.out.println(sb);
    }
}

```

### **toString metodu**

toString metodu ile StringBuffer ve StringBufer objesinden String obje olustururuz.

```

public synchronized String toString() {..}

public class ToStringTest {

    public static void main(String[] args) {
        StringBuffer sb = new StringBuffer("test string");
        System.out.println(sb.toString());
    }
}

```

StringBuffer/StringBuilder siniflari , String sinifindaki tanimli metodlarla ayni metodlara da sahiptir.

## **Pure Java – 56 File and I/O – 01**

[Levent Erguder](#) 22 October 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde dosya sistemi ve Java da IO (input/output) konusunu inceleyecegiz. OCP 6 sinavinda bilinmesi gereken java.io kutuphanesinde yer alan siniflar ;

*File*

*FileReader*

*FileWriter*

*PrintWriter*

*BufferedReader*

*BufferedWriter*

*Console*

Serilestirme(Serialization) bolumunde ise **InputStream** ve **OutputStream** ( ve bu siniflarin altsiniflari) inceleyecegiz.

**File**

java.io.File sinifi , bir dosyayı (file) ya da dizini(folder) göstermeye yarayan bir sınıfıdır. java.io.File sınıfı dosyadan veri okumak ya da dosyaya veri yazmak için kullanılmaz. java.io.File sınıfı yeni boş bir dosya oluşturmak, dosya(file) aramak , directory(dizin) oluşturmak ve dosya silmek gibi işlemler için kullanılır.

```
import java.io.File;

import java.io.IOException;

public class FileTest {

    public static void main(String[] args) throws IOException {
        File file = new File("test.txt");

        // java.io.File sınıfı tipinde obje , harddiskte yeni bir obje

        // olusturmaz!

        System.out.println(file.exists()); // dosya var mı ?
    }
}
```

```
        System.out.println(file.createNewFile()); // yeni dosya olusturma  
createNewFile ile yapilir.  
  
        System.out.println(file.exists()); // tekrar kontrol et !  
  
    }  
  
}
```

Ornegi ilk calistirdigimizda ciktimiz ;

*false*

*true*

*true*

seklinde olacaktir. File sinifindan yeni bir obje olusturmak harddiskte ilgili dosyayı olusturmaz! Bu nedenle ilk file.exists() kontrolü false donecektir. File sinifinda yer alan createNewFile metodu ile yeni bir dosya olusturabiliyoruz, eger sorunsuzca yeni bir dosya olusursa true degeri donecektir. Dosya olustuktan sonra file.exists() kontrolu yaparsak bu sefer true degeri donecektir.

ikinci kez calistirdigimizda ciktimiz ;

*true*

*false*

*true*

seklinde olacaktir. Bunun nedeni artık ilk file.exists() kontrolunde test.txt dosyamizin mevcut olmasidir. (true) Ikinci olarak createNewFile() metodu eger dosya yoksa yeni dosya olusturur varsa yeni dosya olutmaz.

ozetle ;

```
public boolean exists() { }
```

ilgili dosya, harddiskte ilgili dizinde yer aliyorsa true donecektir aksi durumda false donecektir.

```
public boolean createNewFile() throws IOException { ...}
```

Ilgili dosya, harddiskte ilgili dizinde yoksa olusturur varsa olusturmaz!

not0: burada createNewFile() metodu IOException fırlatmaktadır. Hatırlayacağımız gibi IOException bir checked exceptiondir. Dolayısıyla handle or declare kuralına uymamız gerekmektedir.

not1: test.txt dosyası için bir dizin belirtmedik varsayılan olarak projemizin içerisinde src klasoru ile aynı dizinde yer olacaktır.

not2: eclipse dosyayı görebilmek için projeye sağ tıklayıp refresh yapın.

#### FileWriter & FileReader

Pratikte FileWriter ve FileReader sınıfları tek başına kullanılmaz, bu sınıfların wrapper sınıfları ile birlikte kullanılır.

#### FileIOTest.java

```
import java.io.File;  
  
import java.io.FileReader;  
  
import java.io.FileWriter;  
  
import java.io.IOException;  
  
public class FileIOTest {  
  
    public static void main(String[] args) throws IOException {  
  
        File file = new File("file.txt");
```

```
FileWriter fw = new FileWriter(file);

fw.write("levent erguder");

fw.write(" 25");

// write metodu ile String veya char array yazabiliyoruz. int alan

// yapılandırıcı ilgili sayının unicode değeridir.

fw.flush(); // dosyaya yazması için flush metodu çağrılmalıdır.

fw.close(); // IO işlemlerinde , DB bağlanma işlemlerinde close metodu

// çağrılmalıdır. Bu bağlantının kapatılması

anlamına

// gelmektedir.

// Açık bağlantı bırakmamak için close metodunu çağrımayı unutmayalım !

char[] input = new char[50];

int size = 0;

FileReader fr = new FileReader(file); // FileReader objesi oluşturduk

size = fr.read(input); // dosya içeriğini oku

System.out.println("size:" + size); // kaç byte okundu ?
```

```

        for (char c : input) {

            System.out.print(c);

        }

        fr.close();

    }

}

```

## Pure Java – 57 File and I/O – 02

[Levent Erguder](#) 24 October 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde File ve IO konusuna devam edecegiz. Hatirlayacagimiz gibi java.io.File sinifi hem dosya(file) hem dizin(directory) temsil etmeye (representation) yarar.

**java.io.File** sinifi yardimi ile dosya olusturma, dosyayı silme, dosyaya yeni isim verme gibi isler yapabiliriz  
Bununla birlikte yeni bir dizin de olusturabiliriz.

```
File file = new File("test.txt");
```

- 1)Bu kod File tipinde bir obje olusturur.
- 2)Eger "test.txt" adinda bir dosya yoksa olusturulmaz.
- 3)Eger "test.txt" adinda bir dosya varsa bu dosyaya referansta bulunulur.

Unutmayalim bu kod hic bir zaman yeni bir "test.txt" dosyasi harddiskte olusturmaz. Bunun icin

```
File file = new File("test.txt");
file.createNewFile();
```

createNewFile metodunu kullanmamız gerekdir.

Dizin(directory) olusturmak, dosya olusturmakla benzer sekilde yapilir. Ornegin ;

```
File mydir = new File("mydirectory");
mydir.mkdir();
```

mkdir fonksiyonu ile yeni bir directory(dizin) olusturabiliriz. java.io.File sinifinin yapislandiricisini kullanarak olusturdugumuz bu dizin içerisinde yeni bir dosya olusturabiliriz.

```
File mydir = new File("mydirectory");
mydir.mkdir();
File file = new File(mydir, "test.txt");
file.createNewFile();
```

eger dizin olusturulmadan , dizin içerisinde dosya olusturulmak istenirse calisma zamaninda hata alinir. Yani ;

```
File mydir = new File("mydirectory");
//mydir.mkdir(); // dizin olusturma islemi atlandi.
File file = new File(mydir, "test.txt");
file.createNewFile();
```

java.io.File objesi hem file(dosya) hem dizne(directory) referansta bulunabilir.

```
System.out.println(mydir.isDirectory());
System.out.println(file.isFile());
```

Dosya ve dizin olusturdugumuz gibi isim degisikligi yapma veya silme islemi de yapabiliriz.  
Dizin/directory bos degilse bu dizni silemeyiz. **delete** metodu ile silme islemini yapabiliriz.

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class FileIOTest {

    public static void main(String[] args) throws IOException {
        File mydir = new File("mydirectory");
```

```

mydir.mkdir();

System.out.println(mydir.isDirectory());

File file = new File(mydir, "test.txt");

file.createNewFile();

System.out.println("mydir is directory: " + mydir.isDirectory());

System.out.println("file is file :" + file.isFile());

// dizin bos olmadiginda silme islemi yapilamaz.

// false deger doneciktir

System.out.println(mydir.delete());

// dosya silinir , true deger doneciktir

System.out.println(file.delete());

// dizin icerisindeki dosyayi sildikten sonra tekrar deneyelim.

System.out.println(mydir.delete());

}

}

```

Dosyaya yeni isim vermek icin renameTo metodunu kullaniriz;

```

import java.io.File;

import java.io.IOException;

public class RenameToFile {

    public static void main(String[] args) {

        File file = new File("name1.txt");

        try {

            file.createNewFile();

        } catch (IOException e) {

            e.printStackTrace();

        }
    }
}

```

```
File file2 = new File("newName.txt");

file.renameTo(file2); // dosyaya yeni isim verdik

}

}
```

Dizne/directory yeni isim vermek icin de **renameTo** metodunu kullaniriz ;

```
import java.io.File;

public class RenameToDirectory {

    public static void main(String[] args) {
        File myDir = new File("myDir");
        myDir.mkdir();

        File myDir2 = new File("newMyDir");

        myDir.renameTo(myDir2); // dizne/directory yeni isim verdik.
    }
}
```

Biz dizin içerisindeki dosya veya alt dizinleri listeleyebiliriz. Bunun icin **list** veya **listFiles** metodunu kullanabiliriz.

```
import java.io.File;
import java.io.IOException;

public class ListDirectory {
    public static void main(String[] args) throws IOException {
        File mydir = new File("testdir");
        mydir.mkdir();

        File subdir = new File(mydir, "subdir");
        subdir.mkdir();
```

```

File file1 = new File(mydir, "test1.txt");
file1.createNewFile();

File file2 = new File(mydir, "test2.txt");
file2.createNewFile();

File file3 = new File(mydir, "test3.txt");
file3.createNewFile();

File file4 = new File(mydir, "test4.txt");
file4.createNewFile();

String[] listName = mydir.list();

for (String name : listName) {
    System.out.println("found:" + name);
}

File[] fileListName = mydir.listFiles();

for (File f : fileListName) {
    if (f.isFile()) {
        System.out.println("file name :" + f.getName() + " " +
f.getAbsolutePath());
    } else if(f.isDirectory()){
        System.out.println("directory name :" + f.getName() + " " +
f.getAbsolutePath());
    }
}
}

isFile metodu , File objesi bir dosyaya referansta bulunuyorsa true donecektir.
isDirectory metodu, File objesi bir dizne referansta bulunuyorsa true donecektir.

```

## Pure Java - 58 File and I/O - 03

[Levent Erguder](#) 24 October 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazida **BufferedWriter** , **PrintWriter** , **BufferedReader** , **Console** siniflarini inceleyecegiz.

Onceki yazilarda dosyaya veri yazmak icin **FileWriter** ve dosyadan veri okumak icin **FileReader** siniflarini kullanmistik. Bu siniflar biraz daha basit ve ilkel siniflardir. Genel olarak tek basina kullanilmazlar bunun yerine **BufferedWriter** ,**PrintWriter**, **BufferedReader** gibi siniflarla birlikte kullanilirlar.

Ornegimizi inceleyelim ;

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

public class BufferedTest {

    public static void main(String[] args) throws IOException {
        File file = new File("buffered.txt");
        FileWriter fw = new FileWriter(file);
        BufferedWriter bw = new BufferedWriter(fw);

        bw.write("www.injavawetrust.com");
        bw.write("\n"); // alt satira gecmek icin \n karakterini kullandik
        bw.write("Levent Erguder");

        bw.flush();
        bw.close();

        FileReader fr = new FileReader(file);
        BufferedReader br = new BufferedReader(fr);

        // BufferedReader readLine metodu saglar
        // bu metod dosyayı satır satır okur.
    }
}
```

```

        String lineContent = null;
        int lineCount = 1;
        while ((lineContent = br.readLine()) != null) {
            System.out.println("line " + lineCount + " : " + lineContent);
            lineCount++;
        }
    }
}

```

BufferedWriter sinifi yapılandiricisi Writer sinifi tipinde bir parametre almaktadir.

*(FileWriter IS – A Writer)*

BufferedReader sinifi yapılandiricisi Reader sinifi tipinde bir parametre almaktadir.

*(FileReader IS – A Reader)*

**BufferedWriter** ile dosyaya veri yazarken yeni satir icin \n karakterini kullandik.

**BufferedReader** sinifi bize **FileReader** sinifinda olmayan dosya icerigini satir satir okuyan **readLine** metodunu saglar.

**PrintWriter** sinifinin String parametre alan yapılandiricisi vardir , **File** ve **FileWriter** objeleri olusturmadan da yeni bir dosya olusturabiliriz. Dilersek **FileWriter** tipinde parametre alan yapılandiriciyi da kullanabiliriz.

```

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class PrintWriterTest {

    public static void main(String[] args) throws IOException {

        PrintWriter pw = new PrintWriter("printwriter.txt");
        pw.println("www.injavawetrust.com");
        pw.print("Levent Erguder");
        pw.flush();
        pw.close();

        File file = new File("printwriter2.txt");
        FileWriter fw = new FileWriter(file);
    }
}

```

```

PrintWriter pw2 = new PrintWriter(fw);

pw2.println("be an oracle man , import java.*");
pw2.println("injavawetrust!");

pw2.flush();
pw2.close();

System.out.println("Islem tamamlandi.");
}

}

```

**FileWriter** sinifininin **boolean** alan yapılandırıcıları vardır. Bu yapılandırıcıları kullanarak ve arguman olarak **true** değeri vererek dosya içeriğine ekleme yapabiliriz.

```

public FileWriter(File file, boolean append) throws IOException {...}

public FileWriter(String fileName, boolean append) throws IOException {...}

import java.io.FileWriter;
import java.io.IOException;

public class AppendContentFile {

    public static void main(String[] args) throws IOException {

        FileWriter fw = new FileWriter("mytext.txt",true);
        fw.write("test ");
        fw.flush();
        fw.close();

    }
}

```

**java.io.Console** sınıfı , konsol(console) dan klavye aracılığıyla veri okumayı sağlar. **java.io.Console** sınıfı Java 6 ile birlikte gelmiştir.

```
import java.io.Console;
```

```

public class ConsoleTest {

    public static void main(String[] args) {
        Console c = System.console();
        // Console sinifinin yapılandırıcısı private tanımlıdır. Bu nedenle
        // System.console() ile Console objesine sahip olabiliriz.

        System.out.print("UserName:");
        String username = c.readLine();

        char[] password = c.readPassword("%s", "Password:");
        //readPassWord metodu String değil char[] dizisi döner !

        System.out.println("####");
        System.out.print(username + " ");
        System.out.print(password);

    }
}

```

## Pure Java – 59 File and I/O – 04

[Levent Erguder](#) 24 October 2014 [Java SE](#)

Merhaba Arkadaşlar,

Bu yazida java dosya IO işlemlerinde stream konusunu inceleyecegiz ;

- **InputStream**
- **FileInputStream**
- **OutputStream**
- **FileOutputStream**
- **BufferedInputStream**
- **BufferedOutputStream**
- **OutputStreamWriter**
- **InputStreamReader**

Onceki bolumlerde isledigimiz Reader/Writer siniflari karaktere yoneliktir(Character Oriented) InputStream/OutputStream ise byte'a yoneliktir(byte oriented)  
InputStream ve OutputStream abstract siniflardir.Dosya IO islemlerinde FileInputStream ve FileOutputStream kullanilir.  
Onceki orneklerde String veya char dizisini dosyaya yazmistik, bu ornekte byte dizisini dosyaya yazacagiz.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class StreamTest1 {

    public static void main(String[] args) throws IOException {

        byte[] b = { 108, 101, 118, 101, 110, 116 };
        // levent ---> karakterlerine karsilik gelen unicode degerleridir.

        // OutputStream sinifi abstract siniftir.
        // FileOutputStream IS-A OutputStream dir.

        OutputStream os = new FileOutputStream("test.txt");
        os.write(b);

        os.flush();
        os.close();

        InputStream is = new FileInputStream("test.txt");
        for (int i = 0; i < b.length; i++) {
            System.out.print(" " + (char) is.read());
        }

        is.close();
    }
}
```

```
    }  
}
```

Benzer ornegi BufferedOutputStream ve BufferedInputStream kullanarak da yapabiliyoruz.

```
import java.io.BufferedInputStream;  
import java.io.BufferedOutputStream;  
import java.io.File;  
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.InputStream;  
import java.io.OutputStream;  
  
public class BufferedStreamTest {  
  
    public static void main(String[] args) throws IOException {  
        File file = new File("bufferedstream.txt");  
        OutputStream os = new FileOutputStream(file);  
        BufferedOutputStream bos = new BufferedOutputStream(os);  
  
        byte[] b = { 108, 101, 118, 101, 110, 116 };  
  
        bos.write(b);  
        bos.flush();  
        bos.close();  
  
        InputStream is = new FileInputStream(file);  
        BufferedInputStream bis = new BufferedInputStream(is);  
  
        while(bis.available()>0){  
            char c =(char) bis.read();  
            System.out.print(c);  
        }  
    }  
}
```

```
        bis.close();
    }

}
```

OutputStreamWriter sınıfı ile karakterleri stream şekilde yazarken Charset kullanabiliriz.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.OutputStreamWriter;

public class OutputStreamWriterTest {

    public static void main(String[] args) throws IOException {

        OutputStream os = new FileOutputStream("test.txt");
        OutputStreamWriter writer = new OutputStreamWriter(os, "UTF-8");

        writer.write(305);
        writer.write(351);
        writer.write(287);
        writer.write(252);
        writer.write(220);
        writer.write(246);

        writer.flush();
        writer.close();

        InputStream is = new FileInputStream("test.txt");
        // InputStreamReader isr = new InputStreamReader(is, "ISO-8859-1");
        InputStreamReader isr = new InputStreamReader(is, "UTF-8");
    }
}
```

```

        int i = 0;

        char c;

        while ((i = isr.read()) != -1) {

            c = (char) i;

            System.out.println(c);

        }

    }

}

```

Bu siniflar disinda bir cok IO sinifi mevcuttur. Ihтиyaca yonelik olarak kullanılabilir.

## Pure Java - 60 File and I/O - 05 - Serialization

[Levent Erguder](#) 25 October 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde Javada **Serialization**(Serilestirme) konusundan bahsedecegim. Bu sayede 2 yeni File IO sinifi gorecegiz ;

**java.io.ObjectOutputStream**

**java.io.ObjectInputStream**

**Serialization**(Serilestirme) , “bu objeyi instance variable’lariyla birlikte kaydet(save)” anlamina gelmektedir. Bu durumda ilgili sinifin **Serializable** olmasi gereklidir. *java.io.Serializable* bir arabirimdir(interface).

```

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Dog implements Serializable {
    String name = "Karabas";
}

public class SerializeDog {

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Dog dog = new Dog();
    }
}

```

```

        FileOutputStream fos = new FileOutputStream("dog.serial");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(dog);
        oos.flush();
        oos.close();

        FileInputStream fis = new FileInputStream("dog.serial");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Dog serializedDog = (Dog)ois.readObject();
        System.out.println(serializedDog.name);

        ois.close();

    }
}

```

Adim adim inceleyelim ;  
 Dog isminde bir sinif tanimi yaptik,Dog sinifimiz Serializable interface/arabirimini uygulamaktadir.  
 (implements)

Daha sonrasonda **FileOutputStream** sinifindan yararlaniyoruz burada dosya uzantisi .serial olmak zorunda degil. Siz istediginiz degeri verebilirsiniz.  
 Burada dikkat etmemiz gereken yeni gordugumuz **ObjectOutputStream** sinifidir. **ObjectOutputStream** sinifi bizewriteObject metodunu saglamaktadir. Bu metoda arguman olarak **Serializable** bir sinif objesi verebiliriz. Bu islem sonrasi artik bizim dog objemiz instance degiskenleri ile birlikte "dog.serial" dosyasina kayit edildi.  
 Dosyamiz vasitasiyla objemizi geri olusturmak icin(deserialized) icin **FileInputStream** ve **ObjectInputStream** siniflarindan yararlaniyoruz. **ObjectInputStream** sinifinda yer alan **readObject** metodu Object tipinde bir donuse sahiptir. Burada (Dog) tipine reference downcasting yapiyoruz.Burada objemiz Dog tipine cast edilebildigi icin calisma zamaninda herhangi bir sorun cikmayacaktir , cunku serialized/serilestirdigimiz objemizin tipi Dog serialized etmeye calistigimiz objenin tipi de Dog.  
 Unutmayalim soz konusu Serialization ilgili sinif bu arabimi uygulamalidir. Eger uygulamazsa derleme hatasi vermez fakat calisma zamaninda su hatayi(exception) verecektir.

```
java.io.NotSerializableException
```

Eger HAS-A iliskisi varsa bu durumda iki sinifi da Serializable yapmamiz gereklidir;

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Keyboard implements Serializable {
    int size;
    String color;

    public Keyboard(int size, String color) {
        super();
        this.size = size;
        this.color = color;
    }
}

class Laptop implements Serializable {
    Keyboard key = new Keyboard(3, "black");
    int price = 5000;
}

public class LaptopTest {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Laptop laptop = new Laptop();

        FileOutputStream fos = new FileOutputStream("laptop.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(laptop);
        oos.flush();
    }
}
```

```

        oos.close();

        FileInputStream fis = new FileInputStream("laptop.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Laptop serializedLaptop = (Laptop) ois.readObject();
        System.out.println(serializedLaptop.price);
        System.out.println(serializedLaptop.key.color);
        System.out.println(serializedLaptop.key.size);

        ois.close();
    }
}

```

Burada Laptop HAS-A Keyboard ilişkisi vardır. Hatırlayacağımız gibi bir sınıf bir başka sınıf tipinde referans değişkeni sahipse bu ilişkiye HAS-A ilişkisi diyorduk.

Bu örneğimizde Laptop sınıfına ait bir objeyi serileştirmek istiyoruz. Eğer Keyboard sınıfı tipindeki değişkenimiz null olarak bırakılırsa Keyboard sınıfını Serializable yapmaya gerek yok fakat burada olduğu gibi kullanılabıkça bu durumda Serializable olmak zorundadır. Aksi durumda çalışma zamanında *NotSerializableException* hatası/exception verecektir.

Simdi ise farklı bir durumdan bahsedelim Keyboard sınıfımız Serializable değil ve Laptop HAS-A Keyboard ilişkisi söz konusu. Biz Laptop objesini serileştirmek istiyoruz bu durumda ne yapmalıyız ?

Keyboard referans değişkenimiz için transient anahtar kelimesini kullanırız . Serialization işleminde **transient** olarak tanımlanan instance variable işleme tabi tutulmaz atlanır.  
Biraz önceki örneğimizdeki ilgili kısmı şu şekilde değiştirelim ve tekrar çalışıralım ;

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Keyboard {
    int size;

```

```

String color;

public Keyboard(int size, String color) {
    super();
    this.size = size;
    this.color = color;
}

}

class Laptop implements Serializable {
    transient Keyboard key = new Keyboard(3, "black");
    int price = 5000;
}

public class LaptopTest {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Laptop laptop = new Laptop();

        FileOutputStream fos = new FileOutputStream("laptop.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(laptop);
        oos.flush();
        oos.close();

        FileInputStream fis = new FileInputStream("laptop.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Laptop serializedLaptop = (Laptop) ois.readObject();
        System.out.println(serializedLaptop.price);
        // System.out.println(serializedLaptop.key.color); // key null
        // olacaktır
        // System.out.println(serializedLaptop.key.size); // key null
        // olacaktır

        ois.close();
    }
}

```

```
    }  
}
```

Laptop objemizi serileştiriyoruz fakat Keyboard referans değişkenimiz transient olduğu için serileştirme işleminde atlanyor. Deserialized işleminde ise null değerine sahip olur. Peki null değerden başka bir değere sahip olmasını nasıl sağlarız ? Bunun için kullandığımız **writeObject** ve **readObject** metodlarını kendimiz yazmamız gerekmektedir ;

```
private void writeObject(ObjectOutputStream os) {  
  
}  
  
private void readObject(ObjectInputStream is) {  
  
}
```

Bu metodlar Serialization ve Deserialization işlemlerinde otomatik olarak çalışacaktır.

```
import java.io.FileInputStream;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.io.Serializable;  
  
class Keyboard {  
    int size;  
    String color;  
  
    public Keyboard(int size, String color) {  
        super();  
        this.size = size;  
        this.color = color;  
    }  
}
```

```

class Laptop implements Serializable {

    transient Keyboard key = null;
    int price = 5000;

    private void writeObject(ObjectOutputStream os) throws IOException {
        os.defaultWriteObject();
        os.writeInt(3);
        os.writeUTF("black");

        // burada eklenme sirasi onemlidir.
    }

    private void readObject(ObjectInputStream is) throws ClassNotFoundException, IOException {
        is.defaultReadObject();
        key = new Keyboard(is.readInt(), is.readUTF());
    }
}

public class LaptopTest {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        Laptop laptop = new Laptop();

        FileOutputStream fos = new FileOutputStream("laptop.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(laptop);
        oos.flush();
        oos.close();

        FileInputStream fis = new FileInputStream("laptop.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Laptop serializedLaptop = (Laptop) ois.readObject();
        System.out.println(serializedLaptop.price);
        System.out.println(serializedLaptop.key.color);
        System.out.println(serializedLaptop.key.size);
    }
}

```

```
        ois.close();
    }
}
```

Laptop HAS-A Keyboard durumu var. Burada dikkat edecek olursak Keyboard sınıfı Serializable olarak tanımlı degildir. Bu durumda Keyboard referans değişkeni transient olarak tanımlanmalıdır böylelikle serileştirme işleminde atlanacaktır ve null değere sahip olacaktır.

Burada Laptop sınıfı içerisinde **writeObject** ve **readObject** metodlarını kullanarak **defaultWriteObject** ve **defaultReadObject** metodları yardımı ile diledigimiz değerleri uygun şekilde atayabiliriz ve okuyabiliriz(read). (Burada ekleme sırası ve yapılandırıcılarla dikkat etmek gereklidir)

Serialization işlemi sırasında otomatik olarak **writeObject** çalışır, deserialization işlemi sırasında otomatik olarak **readObject** metodu çağrılır.

Kısaçısı manuel olarak objemizin state/durumu/instance değişkenlerini yazmak(okumak) için **writeObject** ve **readObject** metodlarını kullanırız.

Hatırlayacağımız gibi bir obje oluşturduğumuzda yapılandırıcı çalışır. Fakat deserialized işleminde yapılandırıcı çalışmaz. Instance değişkenler için initialze işlemi tekrar yapılmaz. Serileştirme işlemi sırasında hangi değerleri verdiysek o değerler geri getirilecektir. Tabii transient olan instance variablelar null, 0 gibi default değerlere sahip olacaktır.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Foo implements Serializable {
    int num = 3;

    Foo() {
        num = 20;
    }

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        FileOutputStream fos = new FileOutputStream("foo.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
    }
}
```

```

        Foo foo = new Foo();
        foo.num = 50;

        oos.writeObject(foo);
        oos.flush();
        oos.close();

        FileInputStream fis = new FileInputStream("foo.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Foo deserializedFoo = (Foo) ois.readObject();
        System.out.println(deserializedFoo.num);
        // deserialize islemi sirasinda yapislandiri calismaz !
        // deserialize islemi sirasinda instance degiskenler icin initialize
        // islemi tekrar calismaz !
        // num degiskeni 50 degerine sahipti bu sekilde serilestirilmisti.

        ois.close();
    }

}

```

Simdi bir diger ornegi girelim ;

```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Animal {
    int height = 30;
}

```

```

public class Dog extends Animal implements Serializable {
    String name;

    public Dog(int height, String name) {
        this.height = height;
        this.name = name;
    }

    public static void main(String[] args) throws IOException, ClassNotFoundException {
        FileOutputStream fos = new FileOutputStream("dog.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        Dog dog = new Dog(50, "karabas");

        System.out.println(dog.height);
        System.out.println(dog.name);

        oos.writeObject(dog);
        oos.close();

        FileInputStream fis = new FileInputStream("dog.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        Dog serializedDog = (Dog) ois.readObject();

        System.out.println(serializedDog.height);
        System.out.println(serializedDog.name);

        ois.close();
    }
}

```

Dog sinifimiz Animal sinifini kalitmaktadir. Dog sinifi Serializable olmasina ragmen Animal sinifi Serializable degildir.

Hatirlayacagimiz gibi Deserialized islemi sirasinda yapılandiricilar ve instance degiskenlere deger atanmasi yapılmayacaktir. Fakat burada Animal sinifi Serializable degildir. Deserialized islemi sirasinda , Serializable olmayan sinifların yapılandiricilari ve instance degiskenlere deger atanmasi yapilir.

Yani burada Dog sinifinin yapılandiricisi deserialized isleminde calismaz, Animal sinifindaki instance degiskenenin initialize islemi ise deserialized islemi sirasinda caliscaktir.

Collection veya array tipinde instance degiskenleri serilestirmeye calistigimizda , her eleman Serializable ozellige sahip olmalidir.

Static degiskenlerin Serialization islemleri sirasinda kullanilmasi önerilmez. Hatirlayacagimiz gibi static degiskenler sinifa aittir, instance degiskenler objeye aittir.

## Pure Java - 61 Dates, Numbers & Currency - Date & Calendar

[Levent Erguder](#) 29 October 2014 [Java SE](#)

Merhaba Arkadaslar, Java API'si bizim icin date, number , currency gibi konular icin çok cesitli siniflar ve metotlar tanimlamaktadir. OCP 6 sinavinda bu siniflarin ve metotlarin bir kismi sinava dahildir;

- **java.util.Date**
- **java.util.Calendar**
- **java.text.DateFormat**
- **java.text.NumberFormat**
- **java.util.Locale**
  - Burada farkli siniflari da yeri geldikce kullanacagiz. Ornegin DateFormat sinifinin alt sinifi olan **:java.text.SimpleDateFormat**

### Date

java.util.Date sinifinda yer alan bir çok metot deprecated(eskimis/modasi gecmis) durumdadır fakat **Calendar** ve **DateFormat** sinifları ile birlikte kullanilmaya devam etmektedir ve hala kullanılan onemli metottlara sahiptir.

```
• import java.util.Date;  
•  
• public class DateTest {  
•  
•     public static void main(String[] args) {  
•         Date date = new Date();  
•  
•         // Returns the number of milliseconds since January 1, 1970, 00:00:00
```

```

•           // GMT

•           long time = date.getTime();

•           System.out.println(time);

•           /* @return the difference, measured in milliseconds, between

•           // the current time and midnight, January 1, 1970 UTC.

•           long now = System.currentTimeMillis();

•           System.out.println(now);

•           Date date2 = new Date(now);

•           System.out.println(date2);

•           }

•       }

```

- getTime() metodu 1 Ocak 1970 00.00.00 tarihinden suana kadar gecenki surenin milisaniye cinsinden degerini dondurur. Benzer bir metot olarak currentTimeMillis() metodudur.

```

• import java.util.Date;

• 

• public class CompareDates {

• 

•     public static void main(String[] args) {
• 
•         Date date1 = new Date();
• 
•         Date date2 = new Date();
• 
• 
•         int comparison = date1.compareTo(date2);
• 
•         System.out.println(comparison);
• 
•         // 0 ise iki date/tarih esittir.
• 
•         // negatif deger ise 1.date daha eski tarihtir.
• 
•         // pozitif deger ise 1.date daha yendir.
• 
•     }
• 
• }

```

- getDay, getHour gibi metodlar deprecated olmustur kullanilmasi onerilmez.

- **Calendar**

`java.util.Calendar` sinifi tarih/date ile ilgili islemler icin kullanilir. `java.util.Date` sinifinda deprecated metotlar yerine `java.util.Calendar` sinifinda yer alan metotlar kullanilir.

- Calendar sinifi abstract siniftir, dolayisiyla new anahtar kelimesi ile Calendar sinifindan yeni bir instance/ornek/obje olusturamayiz. Calendar sinifinda overloaded getInstance() metotlari yer almaktadir.

- `public static Calendar getInstance()`
- `{`
- `Calendar cal = createCalendar(TimeZone.getDefaultRef(), Locale.getDefault(Locale.Category.FORMAT));`
- `cal.sharedZone = true;`
- `return cal;`
- `}`
- `public static Calendar getInstance(Locale aLocale)`
- `{`
- `Calendar cal = createCalendar(TimeZone.getDefaultRef(), aLocale);`
- `cal.sharedZone = true;`
- `return cal;`
- `}`
- `public static Calendar getInstance(TimeZone zone)`
- `{`
- `return createCalendar(zone, Locale.getDefault(Locale.Category.FORMAT));`
- `}`
- `public static Calendar getInstance(TimeZone zone,`
- `Locale aLocale)`
- `{`
- `return createCalendar(zone, aLocale);`
- `}`

- Dikkat edecek olursak bu metotların hepsi `createCalendar` metodunu çağırmaktadır.

- `private static Calendar createCalendar(TimeZone zone,`
- `Locale aLocale)`
- `{`
- `Calendar cal = null;`
- 
- `String caltype = aLocale.getUnicodeLocaleType("ca");`
- `if (caltype == null) {`

```

•          // Calendar type is not specified.

•          // If the specified locale is a Thai locale,
•          // returns a BuddhistCalendar instance.

•          if ("th".equals(aLocale.getLanguage()))

•              && ("TH".equals(aLocale.getCountry())))

•                  cal = new BuddhistCalendar(zone, aLocale);

•          } else {

•              cal = new GregorianCalendar(zone, aLocale);

•          }

•          } else if (caltype.equals("japanese")) {

•              cal = new JapaneseImperialCalendar(zone, aLocale);

•          } else if (caltype.equals("buddhist")) {

•              cal = new BuddhistCalendar(zone, aLocale);

•          } else {

•              // Unsupported calendar type.

•              // Use Gregorian calendar as a fallback.

•              cal = new GregorianCalendar(zone, aLocale);

•          }

•      }

•      return cal;

•  }

```

- Genel olarak bu overloaded metodlardan parametre almayan overloaded `getInstance()` metodu kullanırız. Bu metod bize `GregorianCalendar` tipinde bir obje verecektir. Dikkat edecek olursak Java farklı tipteki takvimlere/calendar destek sağlamaktadır. `GregorianCalendar` , `Calendar` sınıfını kalıtmaktadır dolayısıyla `GregorianCalendar IS A Calendar` ilişkisi söz konusudur.

```

•  import java.util.Calendar;

•  import java.util.Date;

•  import java.util.GregorianCalendar;

•  public class CalendarUtil {

•      public static void main(String[] args) {

•          Calendar calendar = Calendar.getInstance();

•          System.out.println(calendar instanceof GregorianCalendar);

```

- `System.out.println(calendar);`
- 
- `Date date = calendar.getTime();`
- `System.out.println(date);`
- `// Date objesi olusturabiliriz.`
- `}`
- `}`

- `getInstance()` metodu ile GregorianCalendar tipinde bir objeye sahip oluruz. IS-A kurali geregi Calendar tipindeki bir referans degisken GregorianCalendar tipindeki bir objeyi referans edebilir.
- `instanceof` kontrolunu GregorianCalendar icin yaptigimizda true donecektir. elde ettigimiz calendar objesini yazdirdigimiz zaman uzun bir cikti elde ederiz;

- `java.util.GregorianCalendar[time=1414573471563,areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="Europe/Istanbul",offset=7200000,dstSavings=3600000,useDaylight=true,transitions=171,lastRule=java.util.SimpleTimeZone[id=Europe/Istanbul,offset=7200000,dstSavings=3600000,useDaylight=true,startYear=0,startMode=2,startMonth=2,startDay=-1,startDayOfWeek=1,startTime=3600000,startTimeMode=2,endMode=2,endMonth=9,endDay=-1,endDayOfWeek=1,endTime=3600000,endTimeMode=2]],firstDayOfWeek=1,minimalDaysInFirstWeek=1,ERA=1,YEAR=2014,MONTH=9,WEEK_OF_YEAR=44,WEEK_OF_MONTH=5,DAY_OF_MONTH=29,DAY_OF_YEAR=302,DAY_OF_WEEK=4,DAY_OF_WEEK_IN_MONTH=5,AM_PM=0,HOUR=11,HOUR_OF_DAY=1,MINUTE=4,SECOND=31,MILLISECOND=563,ZONE_OFFSET=7200000,DST_OFFSET=0]`

- Calendar sinif get metodu ile yukaridaki bilgilere ulasabiliriz ;  
Ornegin calendar objesinden YEAR degerini almak icin get metodunu su sekilde kullanabiliriz;

- `int year = calendar.get(Calendar.YEAR);`

- Calendar sinifina gidip YEAR degiskenenin degerine baktigimiz zaman ;

- `public final static int YEAR = 1;`

- Burada dikkat edersek YEAR degiskenenin degeri “1” yani yil/year degerine su sekilde de ulasabiliriz ;

- `int year = calendar.get(1);`

- Fakat bu sekilde ulasmak yanlis bilgileri almaya neden olabilir. Bunun yerine `Calendar.ILGILIDEGER` seklinde ulasmak dogru olacaktir. Yoksa 1 degeri YEAR 2 degeri MONTH a karsilik geldigini bir sure sonra unuturuz ve karistiririz. Calendar sinifini incelerseniz

bu tarz alanlardan bol miktar yer almaktadir. Kisacasi 1.sekilde kullanmak daha dogru olacaktir.

- `int month = calendar.get(Calendar.MONTH);`
- `Calendar.MONTH ile de ay bilgisine ulasabiliriz. Burada unutmayalim ocak/january ->0dan baslar. Bu mantigi anladiktan sonra kod icerisindeki aciklamalardan da yararlanip ornegimizi anlayabiliyoruz ;`
- `import java.util.Calendar;`
- 
- `public class CalendarTest {`
- 
- `public static void main(String[] args) {`
- 
- `Calendar calendar = Calendar.getInstance();`
- 
- `int year = calendar.get(Calendar.YEAR);`
- `// int year = calendar.get(1);`
- `// public final static int YEAR = 1;`
- `// YEAR degiskeni 1 degerine sahiptir .`
- `// get metoduna 1 veya Calendar.YEAR argumani vermek ayni seydir.`
- 
- `int month = calendar.get(Calendar.MONTH);`
- `// hangi ay ? OCAK/JANUARY = 0 baslar ! 1'den baslamaz !`
- 
- `int daysOfWeek = calendar.get(Calendar.DAY_OF_WEEK);`
- `// haftanin hangi gunu ? SUNDAY -> 1 dir , burada 0 dan baslamaz !`
- `// MONDAY-->2 dir ... SATURDAY --> 7 dir`
- 
- `int dayOfMonth = calendar.get(Calendar.DAY_OF_MONTH);`
- `// ayin kacinci gunu (1 den baslar)`
- 
- `int daysOfYear = calendar.get(Calendar.DAY_OF_YEAR);`
- `// yilin kacinci gunu (1 den baslar)`
- 
- `System.out.println("year:" + year + " month :" + month);`

- ```
System.out.println("daysOfWeek:" + daysOfWeek + " daysOfMonth:" + dayOfMonth + "
daysOfYear:" + daysOfYear);
```
- 
- ```
int weekOfMonth = calendar.get(Calendar.WEEK_OF_MONTH);
// ayin kacinci haftasi ? (1den baslar)
```
- 
- ```
int weekOfYear = calendar.get(Calendar.WEEK_OF_YEAR);
// yilin kacinci haftasi ? (1den baslar)
```
- ```
System.out.println("weekOfMonth: " + weekOfMonth + " weekOfYear :" + weekOfYear);
```
- 
- ```
int hour = calendar.get(Calendar.HOUR); // 12 lik saat
int hour_of_day = calendar.get(Calendar.HOUR_OF_DAY); //24luk saat
int minute = calendar.get(Calendar.MINUTE);
int millisecond = calendar.get(Calendar.MILLISECOND);
```
- 
- ```
System.out.println("hour :" +hour + " hours of day : "+ hour_of_day+" minute:" +minute +
millisecond :" +millisecond);
```
- 
- ```
int zoneOffset = calendar.get(Calendar.ZONE_OFFSET) ;
System.out.println(zoneOffset/(1000*3600));
// +2 GMT , 7200000 milisaniyedir.
// 1000 mili saniye 1 saniyedir.
// 3600 saniye 1 saattir.
```
- }
- }

- **getMaximum , getMinimum , getActualMaximum , getActualMinimum** gibi metodları kullanabiliriz. Özellikle ilgili ayın son gününü bulmak için faydalı olacaktır. (28 29 30 31 ? ) Açıklamalar kod içerisinde yer almaktadır.

- ```
import java.util.Calendar;
```
- 
- ```
public class CalendarTest2 {
```
- 
- ```
public static void main(String[] args) {
```
-

- Calendar calendar = Calendar.getInstance();
- int maxDayOfWeek = calendar.getActualMaximum(Calendar.DAY\_OF\_WEEK);
- // haftanin gunlerinin max degeri 7dir.
- int minDayOfWeek = calendar.getActualMinimum(Calendar.DAY\_OF\_WEEK);
- // haftanin gunlerinin min degeri 1 dir.
- int maxWeekOfYear = calendar.getActualMaximum(Calendar.WEEK\_OF\_YEAR);
- int maxWeekOfYear2 = calendar.getMaximum(Calendar.WEEK\_OF\_YEAR);
- int lastDayOfMonth = calendar.getActualMaximum(Calendar.DAY\_OF\_MONTH);
- // ayin son gununu bu sekilde bulabiliriz.
- 
- System.out.println(maxDayOfWeek);
- System.out.println(minDayOfWeek);
- System.out.println(maxWeekOfYear);
- System.out.println(maxWeekOfYear2);
- // 52.hafta 28 Aralik ta bitmektedir.
- // 1.hafta 29 30 31 aralık seklinde devam etmektedir. Bu nedenle
- // getMaximum 53 donecektir.
- System.out.println(lastDayOfMonth);
- 
- }
- }

- NOT: Bu yaziyi insallah tekrar guncelleyecegim diger konularin devam etmesi icin simdilik burada sonlandiriyorum. Soyleceklerim henuz bitmedi.

## Pure Java - 62 Dates, Numbers & Currency – DateFormat & SimpleDateFormat

[Levent Erguder](#) 31 October 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda **java.text.DateFormat** ve **java.text.SimpleDateFormat** siniflarindan bahsedecegim.

### DateFormat

**java.text.DateFormat** sinifi da **java.util.Calendar** sinifi gibi **abstract** bir siniftir. Benzer sekilde **getInstance()** veya overloaded **getDateInstance()** metodundan birisi kullanilir. **getInstance()** metodu da **getDateTimeInstance()** metodunu cagirmaktadir.

```
public final static DateFormat getInstance() {
```

```

        return getDateTieInstance(SHORT, SHORT);

    }

.....



public final static DateFormat getDateTieInstance(int dateStyle,
                                                int timeStyle)

{

    return get(timeStyle, dateStyle, 3, Locale.getDefault(Locale.Category.FORMAT));

}

```

**getDateInstance()** metodu da **getDateTieInstance(..)** metodu da get metodunu cagirmaktadir. **get** metodunu inceledigimiz de geriye **SimpleDateFormat** tipinde obje dondurdugunu gorebiliriz. *SimpleDateFormat IS-A DateFormat* tir.

```

private static DateFormat get(int timeStyle, int dateStyle,
                            int flags, Locale loc) {

    if ((flags & 1) != 0) {

        if (timeStyle < 0 || timeStyle > 3) {

            throw new IllegalArgumentException("Illegal time style " + timeStyle);

        }
    } else {

        timeStyle = -1;
    }
}

```

```
    }

    if ((flags & 2) != 0) {

        if (dateStyle < 0 || dateStyle > 3) {

            throw new IllegalArgumentException("Illegal date style " + dateStyle);

        }
    } else {

        dateStyle = -1;

    }

    try {

        // Check whether a provider can provide an implementation that's closer

        // to the requested locale than what the Java runtime itself can provide.

        LocaleServiceProviderPool pool =

            LocaleServiceProviderPool.getPool(DateTimeProvider.class);

        if (pool.hasProviders()) {

            DateTime providersInstance = pool.getLocalizedObject(
                DateTimeGetter.INSTANCE,

                loc,

                timeStyle,
```

```

        dateStyle,
        flags);

    if (providersInstance != null) {

        return providersInstance;

    }

}

return new SimpleDateFormat(timeStyle, dateStyle, loc);

} catch (MissingResourceException e) {

    return new SimpleDateFormat("M/d/yy h:mm a");

}

}

```

Ihtiyaca gore ilgili metodlardan **getInstance()** ya da **getDateInstance()** in overloaded metodlarindan birisi kullanilabilir. Aralarindaki fark format ile ilgilidir.

```

import java.text.DateFormat;

import java.util.Date;

public class DateFormatTest {

    public static void main(String[] args) {

```

```
Date date = new Date();

System.out.println("Date :" + date);

DateFormat dateFormat = DateFormat.getInstance();

System.out.println("getInstance() : " + dateFormat.format(date));

dateFormat = DateFormat.getDateInstance();

System.out.println("getDateInstance() : " + dateFormat.format(date));

dateFormat = DateFormat.getDateInstance(DateFormat.SHORT);

System.out.println("getDateInstance(DateFormat.SHORT):" +
dateFormat.format(date));

dateFormat = DateFormat.getDateInstance(DateFormat.MEDIUM);

System.out.println("getDateInstance(DateFormat.MEDIUM):" +
dateFormat.format(date));

dateFormat = DateFormat.getDateInstance(DateFormat.LONG);
```

```

        System.out.println("getDateInstance(DateFormat.LONG):" +
dateFormat.format(date));

        dateFormat = DateFormat.getDateInstance(DateFormat.FULL);

        System.out.println("getDateInstance(DateFormat.LONG):" +
dateFormat.format(date));

    }

}

```

DateFormat sınıfında **format** metodu yer almaktadır bu metod Date değeri uygun formatta String'e cevirmektedir.

```

public final String format(Date date)

{
    return format(date, new StringBuffer(),
        DontCareFieldPosition.INSTANCE).toString();

}

```

DateFormat sınıfında parse metodu yer almaktadır bu metod String değeri uygun şekilde Date e cevirmektedir. Bu metod ParseException hatası fırlatabilir bu exception bir checked exceptiondir.

```

public Date parse(String source) throws ParseException

{
    ParsePosition pos = new ParsePosition(0);
}

```

```
        Date result = parse(source, pos);

        if (pos.index == 0)

            throw new ParseException("Unparseable date: \"" + source + "\" ,

            pos.errorIndex);

        return result;

    }

import java.text.DateFormat;

import java.text.ParseException;

import java.util.Date;

public class DateFormatTest2 {

    public static void main(String[] args) throws ParseException {

        Date date = new Date();

        DateFormat dateFormat = DateFormat.getInstance();

```

```
System.out.println(date);

// System.out.println(dateFormat.parse(date)));

// parse metodu String deger alir Date e cevirir.

String toStringDate = date.toString();

String formattedDate = dateFormat.format(date);

// System.out.println(dateFormat.parse(toStringDate)); //

// java.text.ParseException: Unparseable date hatasi verir. Ilgili

// String in formati

// bu format icin parse etmeye uygun degil.

System.out.println(dateFormat.parse(formattedDate));

dateFormat = DateFormat.getDateInstance(DateFormat.SHORT);

formattedDate = dateFormat.format(date);
```

```
        System.out.println(formattedDate);

        System.out.println(dateFormat.parse(formattedDate));

        // parse metodu DateFormat a gore saniye kismini ya da saat dakika

        // saniye kismini 00

        // olarak parse eder.

        // ihtiyaca gore kullanilabilir.

    /*
     * Fri Oct 31 21:51:36 EET 2014
     *
     * Fri Oct 31 21:51:00 EET 2014
     *
     * 10/31/14
     *
     * Fri Oct 31 00:00:00 EET 2014
     */
}

}
```

## **SimpleDateFormat**

Ozellesmis bir date/tarih formati icin **SimpleDateFormat** sinifini kullanabiliriz. [SimpleDateFormat](#) linkinden yararlanip ilgili harflerin anlamini anlayabiliriz. Diledigimiz formata uygun sekilde duzenleme yapabiliriz.

```
import java.text.SimpleDateFormat;

import java.util.Date;

public class SimpleDateFormatTest {

    public static void main(String[] args) {

        Date date = new Date();

        SimpleDateFormat sdf = new SimpleDateFormat();

        System.out.println(sdf.format(date));

        sdf = new SimpleDateFormat("MM-dd-yy");

        System.out.println(sdf.format(date));

        sdf = new SimpleDateFormat("MM/dd/yyyy");

        System.out.println(sdf.format(date));
    }
}
```

```
sdf = new SimpleDateFormat("MM/dd/yyyy HH:mm");

System.out.println(sdf.format(date));

sdf = new SimpleDateFormat("MM/dd/yyyy E HH:mm");

System.out.println(sdf.format(date));

sdf = new SimpleDateFormat("MM/dd/yyyy E HH:mm:ss:SS a ");

System.out.println(sdf.format(date));

sdf = new SimpleDateFormat("MM/dd/yyyy E HH:mm:ss:SS a ");

System.out.println(sdf.format(date));

sdf = new SimpleDateFormat("dd.MM.yy E hh:mm:ss:SS a ");

System.out.println(sdf.format(date));

sdf = new SimpleDateFormat("dd.MMM.yy ");

System.out.println(sdf.format(date));

//
```

<http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html>

```
    }  
  
}
```

Java DateFormat ve SimpleDateFormat siniflariyla tarih/date formati icin esnek yapıya sahiptir. Diledigimiz sekilde kullanabiliriz.

## Pure Java – 63 Dates, Numbers & Currency – Locale

[Levent Erguder](#) 01 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda **java.util.Locale** sinifindan bahsedecegim. Locale sinifi

, **DateFormat** , **SimpleDateFormat** , **NumberFormat**siniflari ile kullanilabilir. Bununla birlikte bir çok yerde Locale objesini olusturup kod yazmamiz gerekebilir. Java API'sine gore Locale '*spesifik cografi, politik veya kulturel bolgedir*'.

**Locale** sinifi abstract degildir dolayisiyla new anahtar kelimesi ile yeni bir obje olusturabiliriz. **Locale** sinifinin yapılandiricilari overloaded'tir. Ornegin ;

```
Locale(String language)
```

```
Locale(String language, String country)
```

Kullanicabilecegimiz localleri **getAvailableLocales()** metodu ile gorebiliriz;

```
import java.util.Locale;  
  
public class GetAvailableLocalesTest {  
  
    public static void main(String[] args) {  
        Locale[] locales = Locale.getAvailableLocales();  
  
        int i = 1;  
        for (Locale loc : locales) {  
            System.out.println(i + " " + loc);  
            i++;  
        }  
    }  
}
```

```
    }  
}
```

Locale objesi olusturalim ve bir kac metodu kullanalim ;

```
import java.util.Locale;  
  
public class LocaleTest {  
  
    public static void main(String[] args) {  
        Locale locale = new Locale("TURKISH", "tr");  
  
        System.out.println(locale);  
        System.out.println("Country Code: " + locale.getCountry());  
        System.out.println("Country Name" + locale.getDisplayCountry());  
        System.out.println("Language :" + locale.getLanguage());  
        System.out.println("Language:" + locale.getDisplayLanguage());  
        // A three-letter abbreviation of this locale's country.  
        System.out.println("3 harfli " + locale.getISO3Country());  
  
    }  
}
```

Language ve Country alanları birbiriyle uyusmak zorunda degil, yani country icin Italyanca dilini kullanabiliriz.

```
import java.util.Locale;  
  
/*  
 * public Locale(String language) {  
 *     this(language, "", "");  
 * }  
 */  
  
/*
```

```

public Locale(String language, String country) {
    this(language, country, "");
}

*/
public class LocaleTest2 {

    public static void main(String[] args) {

        System.out.println("Language");
        Locale locTR = new Locale("tr");
        System.out.println(locTR.getDisplayCountry());// bos doner
        // country degeri verilmedi this ile cagrilan yapılandirici country
        // alanina"" degerini atamaktadir
        System.out.println(locTR.getDisplayLanguage());
        System.out.println(locTR.getDisplayLanguage(locTR));

        System.out.println("Language & Country");
        Locale locTR2 = new Locale("tr", "TR");
        System.out.println(locTR2.getDisplayCountry());
        System.out.println(locTR2.getDisplayCountry(locTR2));
        System.out.println(locTR2.getDisplayLanguage());
        System.out.println(locTR2.getDisplayLanguage(locTR2));

        System.out.println("it & TR");
        Locale locIT = new Locale("it", "TR");
        //Country karsiliği varsayılan dilde yani Ingilizce olarak göster.
        System.out.println(locIT.getDisplayCountry());

        //Country karsılığı, locIT deki dili kullanarak göster. Italyanca
        System.out.println(locIT.getDisplayCountry(locIT));

        //Language karsılığı ingilizce olarak göster.
        System.out.println(locIT.getDisplayLanguage());
    }
}

```

```

        // Language karsiligi italyanca olarak goster.

        System.out.println(locIT.getDisplayLanguage(locIT));

    }

}

```

**DateFormat** ve **SimpleDateFormat** siniflariyla **Locale** sinifini beraber kullanabiliyoruz. **SimpleDateFormat** sinifinin **Locale** tipinde parametre alan yapislandiricisi vardir. Benzer sekilde **DateFormat** sinifinda **getDateInstance** metodunun **Locale** tipinde parametre alan overloaded hali vardir.

```

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.Locale;

public class LocaleTest3 {

    public static void main(String[] args) {

        Calendar calendar = Calendar.getInstance();
        Date date = calendar.getTime();

        Locale locIT = new Locale("it", "IT");

        DateFormat defaultUSDateFormat = DateFormat.getDateInstance(DateFormat.FULL);
        System.out.println(defaultUSDateFormat.format(date));

        DateFormat italianDateFormat = DateFormat.getDateInstance(DateFormat.FULL, locIT);
        System.out.println(italianDateFormat.format(date));

        Locale locTR = new Locale("tr", "TR");
        DateFormat turkishDateFormat = DateFormat.getDateInstance(DateFormat.FULL, locTR);
        System.out.println(turkishDateFormat.format(date));

        SimpleDateFormat sdf = new SimpleDateFormat("dd.MMM.yyy", locTR);

```

```
        System.out.println(sdf.format(date));

    }

}

ciktimiz ;
```

Saturday, November 1, 2014

sabato 1 novembre 2014

01 Kası̄m 2014 Cumartesi

01.Kas.2014

## Pure Java - 64 Dates, Numbers & Currency - NumberFormat & Currency & DecimalFormat

[Levent Erguder](#) 04 November 2014 [Java SE](#)

Merhaba Arkadaslar, Bu

yazimda **java.text.NumberFormat** , **java.text.DecimalFormat** ve **java.util.Currencies** siniflarini inceleyecegiz.  
**java.text.NumberFormat** sinifi da **java.text.DateFormat** Sınıfı gibi abstract bir sınıfır.

```
public abstract class NumberFormat extends Format { ... }
```

DateFormat sınıfında olduğu gibi benzer şekilde getInstance veya getNumberInstance metodundan yararlanırız. getInstance ve getNumberInstance aynı ise yaramaktadır. Bu metodların overloaded versiyonları da vardır. Örneğimizde kullanacağız.

```
/**  
 * Returns a general-purpose number format for the current default locale.  
 * This is the same as calling  
 * {@link #getNumberInstance() getNumberInstance()}.  
 */  
  
public final static NumberFormat getInstance()  
{  
    return getInstance(Locale.getDefault(), NUMBERSTYLE);  
}  
  
/**  
 * Returns a general-purpose number format for the current default locale.
```

```
*/  
  
public final static NumberFormat getInstance() {  
    return getInstance(Locale.getDefault(), NUMBERSTYLE);  
}
```

Cagrılan ilgili overloaded **getInstance** metodunu incelersek varsayılan olarak geriye `java.text.DecimalFormat` tipinde bir obje dondugunu gorebiliriz.

```
public class DecimalFormat extends NumberFormat {...}
```

`DecimalFormat` IS-A `NumberFormat` ilişkisi söz konusudur. Varsayılan/default **Locale** `en_US` tir ve sayı formatı farklı Local değerine göre değişkenlik gösterebilir. Kod içerisinde açıklamaları inceleyelim.

```
import java.text.DecimalFormat;  
  
import java.text.NumberFormat;  
  
import java.util.Locale;  
  
  
public class NumberFormatTest {  
  
    public static void main(String[] args) {  
        NumberFormat numberFormat = NumberFormat.getInstance();  
        // getInstance() metodunda varsayılan olarak Locale.getDefault()  
        // kullanılır.  
        System.out.println(Locale.getDefault()); // en_US default locale dir.  
  
        NumberFormat numberFormatTR = NumberFormat  
            .getInstance(new Locale("TR"));  
        // overloaded getInstance metodunu kullanarak kendimiz bir Locale  
        // değerini verebiliriz.  
  
        System.out.println(numberFormat instanceof DecimalFormat);  
        // NOT : DecimalFormat IS-A NumberFormat tir !  
  
        double salary = 123.456;  
        System.out.println(numberFormat.format(salary));  
        // Varsayılan locale için yani en_Us için kusuratlı/floating-point  
        // sayılar için arada nokta(.) vardır.
```

```

        System.out.println(numberFormatTR.format(salary));

        // TR icin kusuratli/floating point sayilar icin arada virgul (,)
        // vardir.

        // ###Dikkat edecek olursak farkli Locale gore nokta veya virgul
        // olabilir.

    }

}

```

Kusuratlı sayıları formatladığımız gibi currency/para birimi için de formatlama yapabiliyoruz. Bunun içi `getCurrencyInstance` metodunu kullanabiliyoruz. `java.util.Currency` sınıfında yer alan `getCurrencyCode` ve `getSymbol` metodları ile de parabirim kodu ve sembol bilgisine ulaşabiliyoruz.

```

import java.text.NumberFormat;
import java.util.Currency;
import java.util.Locale;

public class NumberFormatCurrencyTest {

    public static void main(String[] args) {
        NumberFormat numberCurrencyFormat = NumberFormat.getInstance();
        // Locale.getDefault() en_us olduğu için $ olarak formatlayacaktır.

        NumberFormat numberCurrencyFormatGE = NumberFormat
                .getInstance(Locale.GERMANY);
        NumberFormat numberCurrencyFormatIT = NumberFormat
                .getInstance(Locale.ITALY);
        NumberFormat numberCurrencyFormatUK = NumberFormat
                .getInstance(Locale.UK);
        NumberFormat numberCurrencyFormatJP = NumberFormat
                .getInstance(Locale.JAPAN);
        NumberFormat numberCurrencyFormatTR = NumberFormat
                .getInstance(new Locale("tr", "TR"));
    }
}

```

```

        double salary = 123.456;

        System.out.println(numberCurrencyFormat.format(salary));

        System.out.println(numberCurrencyFormatGE.format(salary));

        System.out.println(numberCurrencyFormatIT.format(salary));

        System.out.println(numberCurrencyFormatUK.format(salary));

        System.out.println(numberCurrencyFormatJP.format(salary));

        System.out.println(numberCurrencyFormatTR.format(salary));


        String USDcode = Currency.getInstance(Locale.US).getCurrencyCode();

        String USDSymbol = Currency.getInstance(Locale.US).getSymbol();


        String GBPcode = Currency.getInstance(Locale.UK).getCurrencyCode();

        String GBPSymbol = Currency.getInstance(Locale.UK).getSymbol(Locale.UK);




        String EURcode = Currency.getInstance(Locale.GERMANY).getCurrencyCode();

        String EURSymbol = Currency.getInstance(Locale.GERMANY).getSymbol(
            Locale.GERMANY);





        System.out.println(USDcode + " " + USDSymbol);

        System.out.println(GBPcode + " " + GBPSymbol);

        System.out.println(EURcode + " " + EURSymbol);

    }

}

```

Simdi de **NumberFormat** sinifinda yer alan diger bazi metodlari inceleyelim.

**getMaximumFractionDigits** metodu virgulden sonra en fazla/maximum kactane deger olabilecegi bilgisini dondurur. Bu deger varsayılan olarak 3 tur.

**getMinimumFractionDigits** ise benzer durum icin minimum degeri dondurur bu deger 0(sifir) dir.

Ornegin ; 123.456789 sayisi virgulden sonra 6 rakamlidir fakat varsayılan maximum fraction 3tur. Bu durumda 123.457 olacaktir .Virgulden sonraki rakam 7 oldugu icin yukari yuvarlayacaktır. 5 ve uzeri icin yukari yuvarlama soz konusudur.

**setMaximumFractionDigits** metodunu kullanarak virgulden sonraki kullanilabilecek maximum rakam sayisini arttirabiliriz ya da **setMinimumFractionDigits** metodunu kullanarak kullanilmasi gereken minimum rakam sayisini ayarlayabiliriz.

Benzer sekilde **setMaximumIntegerDigits** ve **setMinimumIntegerDigits** metodlarini kullanarak virgulden once kullanilabilecek rakam sayisini duzenleyebiliriz.

```
import java.text.NumberFormat;

public class NumberFormatFractionTest {

    public static void main(String[] args) {
        NumberFormat numberFormat = NumberFormat.getInstance();

        int maxFractionDigits = numberFormat.getMaximumFractionDigits();
        // varsayılan olarak 3 tür

        int minFractionDigits = numberFormat.getMinimumFractionDigits();
        // varsayılan olarak 0 dir

        System.out.println("minimum fraction digits :" + minFractionDigits);
        System.out.println("maximum fraction digits :" + maxFractionDigits);

        double salary = 123.456789;
        System.out.println(salary);
        System.out.println(numberFormat.format(salary));

        numberFormat.setMaximumFractionDigits(5);
        // virgulden sonraki kullanılan rakam sayısı maksimum 5 olsun.
        System.out.println(numberFormat.format(salary));

        double salary2 = 123;
        System.out.println(salary2); // double sayı için tek 0 koymaktır.

        numberFormat.setMinimumFractionDigits(2);
        // virgulden sonraki kullanılan rakam sayısı minimum 2 olsun.
        // eğer yoksa 00 olarak ekleyecektir.
        System.out.println(numberFormat.format(salary2));

        numberFormat.setMaximumIntegerDigits(5);
        // virgulden önce maksimum 5 karakter olsun
        numberFormat.setMinimumIntegerDigits(3);
```

```

// virgulden once minimum 3 karakter olsun.

System.out.println(numberFormat.format(123456789));
System.out.println(numberFormat.format(89));
}

}

```

NumberFormat sınıfında yer alan **format** metodu integer/tamsayı ya da kayan noktalı sayı literallerini String e dönüştürür. NumberFormat sınıfında yer alan **parse** metodu String değeri Number a dönüştürür. **parse** metodu ParseException fırlatabilir.

```

import java.text.NumberFormat;
import java.text.ParseException;

public class NumberFormatParseTest {

    public static void main(String[] args) {
        NumberFormat numberFormat = NumberFormat.getInstance();

        // NumberFormat sınıfında yer alan parse metodu String değeri Number a
        // dönüştürür.

        // NumberFormat sınıfında yer alan format metodu integer/tamsayı ya da
        // kayan noktalı sayı literallerini String e dönüştürür.

        // parse metodu ParseException fırlatabilir.

        try {
            System.out.println(numberFormat.parse("123.456789"));

            numberFormat.setParseIntegerOnly(true);
            System.out.println(numberFormat.parse("123.456789"));

        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
}

```

**java.text.DecimalFormat** sınıfını kendi belirledigimiz özelleşmiş pattern'e göre sayıları formatlamak için kullanabiliriz.

```
import java.text.DecimalFormat;
import java.text.DecimalFormatSymbols;

public class DecimalFormatTest {

    public static void main(String[] args) {

        DecimalFormat decimalFormat = new DecimalFormat("#,###.###");
        // 3 basamak arasında virgül koyar. 123,456
        // kusuratlı sayılar başlangıcı için nokta.
        // kususatlı 3 basamak olmalı.

        DecimalFormat decimalFormat2 = new DecimalFormat("#,##.##");
        // 2 basamak arasına virgül koyar. 12,34,56

        DecimalFormat decimalFormat3 = new DecimalFormat("000000000.0000");
        // tam bu patterndeki rakam adedine ulaşana kadar 0 ekler
        // 000123456.7890

        DecimalFormat decimalFormat4 = new DecimalFormat("$###,###.##");
        // dolar işaretini koyduk.

        DecimalFormat decimalFormat5 = new DecimalFormat("\u00A5###,###.##");
        // yen işaretini koyar.

        double salary = 123456.789;

        System.out.println(decimalFormat.format(salary));
        System.out.println(decimalFormat2.format(salary));
        System.out.println(decimalFormat3.format(salary));
        System.out.println(decimalFormat4.format(salary));
        System.out.println(decimalFormat5.format(salary));

        DecimalFormatSymbols symbols = new DecimalFormatSymbols();
    }
}
```

```

        symbols.setDecimalSeparator(':');
        symbols.setGroupingSeparator(';');
        // dilersek kendimiz nokta ve virgul karekterlerini baska karakter
        // olacak sekilde degistirip formatlayabiliriz.

        DecimalFormat decimalFormat6 = new DecimalFormat("#,###.##", symbols);

        System.out.println(decimalFormat6.format(salary));

    }
}

```

## Pure Java - 65 Searching - Pattern & Matcher

[Levent Erguder](#) 06 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde **java.util.regex.Matcher** , **java.util.regex.Pattern** siniflari yardimi ile searching/finding konusunu inceleyecegiz.

Regex konusu Java'ya ozgu bir konu degildir. Diger programlama dillerinde de bulunur. Daha iyi anlamak icin kucuk kucuk orneklerle baslayalim.

Pattern sinifinda yer alan compile metoduna arayacagimiz regex'i yaziyoruz.

Matcher sinifinda yer alan matcher metoduna icerisinde arama yapilacak String literali/degeri veriyoruz.(content)

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class JavaRegexTest {

    public static void main(String[] args) {
        Pattern p = Pattern.compile("t");
        // ilgili icerikte/content "injavawetrust" t harfini arayacagiz.
        Matcher m = p.matcher("www.injavawetrust.com");
    }
}

```

```

while (m.find()) {
    System.out.print(m.start() + " ");
}

System.out.println();
Pattern p2 = Pattern.compile("ja");
// arama tek karakter olarak degil birden fazla karakter kalibi/patterni
// seklinde olabilir/
// burada igiliicerikte "ja" literalini arayacaktır.

Matcher m2 = p2.matcher("www.injavawetrust.com");

while (m2.find()) {
    System.out.print(m2.start() + " ");
}

System.out.println();
Pattern p3 = Pattern.compile("j|t|w");
// arama yaparken or/veya yapisini kullanabiliriz. | karakteri ile
// yapabiliriz.

Matcher m3 = p3.matcher("www.injavawetrust.com");

while (m3.find()) {
    System.out.print(m3.start() + " ");
}

System.out.println();
Pattern p4 = Pattern.compile("[jwt]");
// Veya/or yapisi icin | karakterini kullanabildigimiz gibi [ ] seklinde
// de kullanabiliriz.
// [jwt] ---> [j|w|t] ile aynidir.

Matcher m4 = p4.matcher("www.injavawetrust.com");

while (m4.find()) {
    System.out.print(m4.start() + " ");
}

```

```

}

System.out.println();
Pattern p5 = Pattern.compile("[jwt][er]");
// aradigimiz pattern yapisinda ilk karakterden sonra(j veya w veya t)
// e veya r gelmesini istiyoruz.

Matcher m5 = p5.matcher("www.injavawetrust.com");

while (m5.find()) {
    System.out.print(m5.start() + " ");
}

System.out.println();
Pattern p6 = Pattern.compile("[a-i]");
// Araya tire (-) koyarak da or | yapabiliriz.
// bunun anlami a|b|c|d..|i seklindedir.

Matcher m6 = p6.matcher("www.injavawetrust.com");

while (m6.find()) {
    System.out.print(m6.start() + " ");
}

System.out.println();
Pattern p7 = Pattern.compile("[1-5]");
// benzer sekilde sayilar icin de yapabiliriz.

Matcher m7 = p7.matcher("56782");

while (m7.find()) {
    System.out.print(m7.start() + " ");
}

System.out.println();
Pattern p8 = Pattern.compile("[^ajw]");
// caret karakterini kullanabiliriz. bunun anlami aj|w disindaki
// karakterleri bul demektir.

```

```

Matcher m8 = p8.matcher("www.injavawetrust.com");

while (m8.find()) {
    System.out.print(m8.start() + " ");
}

}

```

Simdi de su ornegi inceleyelim;

```

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class JavaRegexTest2 {

    public static void main(String[] args) {

        Pattern p = Pattern.compile("ab");
        Matcher m = p.matcher("abababa");

        while (m.find()) {
            System.out.print(m.start() + " ");
        }

        // bu islem bir kere calisir. Buradaki dongu calismayacaktir !
        while (m.find()) {
            System.out.print(m.start() + " ");
        }

        System.out.println();
        Pattern p2 = Pattern.compile("aba");
        Matcher m2 = p2.matcher("abababa");

        while (m2.find()) {

```

```

        System.out.print(m2.start() + " ");

        // burada 2 tane sonuc dondurecektir. baslangic indexi 0 ve 4 olacak

        // sekilde.

        // baslangic indexi 2 olan dahil edilmez. Cunku idexi 0 olan aba

        // degerine 2.index de dahil.

        // Regex islemesi soldan saga olur ve genel olarak kullanilan bir

        // index bir daha kullanilmaz.

        // aba 0 3

        // aba 4 7 seklinde olur

        // aba 2 5 kullanilmaz!

    }

}

}

```

### Searches Using Metacharacters

Arama islemlerinde Metacharacter kullanabiliyoruz bunlar ;

- \d digit
- \s whitespace(bosluk)
- \w harf,rakam ya da alt çizgi
- \D non-digit
- \S non-whitespace
- \W non-word
- . the dot/nokta metacharacter

```

import java.util.regex.Matcher;

import java.util.regex.Pattern;

public class JavaRegexMetaCharTest {

    public static void main(String[] args) {

        Pattern p = Pattern.compile("\d");
        // \d --> [0-9] anlamına gelmektedir.

        Matcher m = p.matcher("java1 reg2ex!");

        while (m.find()) {

```

```

        System.out.print(m.start() + " ");

    }

Pattern p2 = Pattern.compile("\\D");
// \D non-digit anlamina gelmektedir --> [^0-9]
// bosluk karakterini almaz!

Matcher m2 = p2.matcher("java1 reg2ex!");

System.out.println();
while (m2.find()) {
    System.out.print(m2.start() + " ");
}

Pattern p3 = Pattern.compile("\\s");
// \s --> space bosluk karakterini bulur.

Matcher m3 = p3.matcher("java1 reg2ex!");

System.out.println();
while (m3.find()) {
    System.out.print(m3.start() + " ");
}

Pattern p4 = Pattern.compile("\\S");
// \S --> non-space anlamina gelir --> [^\\s]

Matcher m4 = p4.matcher("java1 reg2ex!");

System.out.println();
while (m4.find()) {
    System.out.print(m4.start() + " ");
}

Pattern p5 = Pattern.compile("\\w");
// \w --> harf veya rakam

Matcher m5 = p5.matcher("java1 reg2ex!");

System.out.println();

```

```

        while (m5.find()) {
            System.out.print(m5.start() + " ");
        }

        Pattern p6 = Pattern.compile("\\W");
        // \W non-word --> ozel karakterler bosluk vs
        Matcher m6 = p6.matcher("java1 reg2ex!");
        System.out.println();
        while (m6.find()) {
            System.out.print(m6.start() + " ");
        }

        Pattern p7 = Pattern.compile("a.c");
        // . nokta metakarakteri
        // "buraya herhangi bir karakter gelebilir demektir"

        // nokta metacharacteri kullanirken \. seklinde kullanmiyoruz.
        // \. metacharacter degildir nokta karakterini temsil eder.
        Matcher m7 = p7.matcher("ac abc a c a d");
        // ac yi almaz cunku arada bir karakter yok.
        // abc
        // a c ise alir.
        // a d almaz arada 2 karakte bosluk var.
        System.out.println();
        while (m7.find()) {
            System.out.print(m7.start() + " ");
        }
    }
}

```

## Quantifiers

Quatifier, aranan elemanların/karakterlerin kaç defa geçeceğini bilgisini belir.

\* 0 veya daha fazla → {0,}

+ 1 veya daha fazla → {1,}

**? 0 veya 1 tane → {0,1}**  
**{X} X sayisi kadar**  
**{X,} X veya daha fazla**  
**{X,Y} minimum X maximum Y kadar**

```
import java.util.regex.Matcher;  
import java.util.regex.Pattern;  
  
public class JavaRegexQuantifierTest {  
    public static void main(String[] args) {  
  
        Pattern p = Pattern.compile("\\d");  
        Matcher m = p.matcher("1 a12 234b");  
  
        while (m.find()) {  
            System.out.println("start : " + m.start() + " end :" + m.end() + " content:" + m.group());  
            // start metodu match/eslesen regex icin baslangic indexi verir.  
            // end metodu match/eslesen regex icin baslangic indexi verir.  
            // group metodu match/eslesen regexi verir.  
        }  
  
        Pattern p2 = Pattern.compile("\\d+");  
        // + karakteri 1 veya daha fazla anlamina gelir.  
        Matcher m2 = p2.matcher("1 a12 234b");  
        System.out.println("one or more");  
        while (m2.find()) {  
            System.out.println("start : " + m2.start() + " end :" + m2.end() + " content:" +  
m2.group());  
            // Burada dikkat edilecek olursa \d+ olarak kullanildi.  
            // 12 veya 234 kisminda yanyaya birden fazla rakam geldigi icin  
            // bunlari birlikte degerlendirir.  
            // araya bosluk geldigi zaman ayirir.  
        }  
  
        // Pattern p3 = Pattern.compile("a\\d?");  
        // ? karakteri 0 veya 1 anlamina gelir.
```

```

Pattern p3 = Pattern.compile("a\\d+");
System.out.println("one or more 2");
while (m3.find()) {
    System.out.println("start : " + m3.start() + " end :" + m3.end() + " content:" +
m3.group());
    // a\\d? anlami a dan sonra 0 veya 1 tane digit/sayı varsa
    // a\\d+ anlami a dan sonra 1 veya daha fazla digit/sayı varsa
}

Pattern p4 = Pattern.compile("\\d{3}");
Matcher m4 = p4.matcher("1a123 ax");
System.out.println("3digits");
while (m4.find()) {
    System.out.println("start : " + m4.start() + " end :" + m4.end() + " content:" +
m4.group());
}

Pattern p5 = Pattern.compile("0[xX][0-9a-fA-F]");
// 0 ile baslasin 2.karakter x veya X olsun sonrasinda rakam(0-9) ya da a-f ya da A-F arasinda harf
olsun.

Matcher m5 = p5.matcher("12 0x 0x12 0Xf 0xg");
System.out.println("more regex");
while (m5.find()) {
    System.out.println("start : " + m5.start() + " end :" + m5.end() + " content:" +
m5.group());
}

}
}

```

### Greedy Quantifiers

\* ? + quatifier'ları acıgovlu/greedy davranışlıdır. Reluctant/isteksiz özelliği için ? karakterini ekleriz.

```
import java.util.regex.Matcher;
```

```

import java.util.regex.Pattern;

public class GreedyQuantifiersTest {

    public static void main(String[] args) {

        Pattern p = Pattern.compile(".*xx");

        // regexi inceledigimizde xx ile bitmesini istiyoruz ve oncesinde
        // herhangi bir karakter olabilir diyoruz.

        // Daha once belirtigimiz gibi regex mekanizmasi soldan saga dogru
        // calisir.

        Matcher m = p.matcher("yyxxxxxx");

        while (m.find()) {
            System.out.println(m.start() + " " + m.group());
            // burada beklentimiz
            // 0 yyxx
            // 4 xyxx seklinde fakat boyle calismayacaktir !
            // bunun yerine 0 yyxxxxxx seklinde bir cikti verecektir.
            // Bunun nedeni regex mekanizmasi greedy/acgozlu davranmaktadır. tum
            // icerige bakar sagdan itibaren en çok esleserek sekilde alır.
        }

        // * greedy/acgozlu *? reluctant/isteksiz
        // ? greedy/acgozlu ?? reluctant/isteksiz
        // + greedy/acgozlu +? reluctant/isteksiz

        Pattern p2 = Pattern.compile(".*?xx");
        //burada reluctant/isteksiz calisactir.

        Matcher m2 = p2.matcher("yyxxxxxx");

        while (m2.find()) {
            System.out.println(m2.start() + " " + m2.group());
        }
    }
}

```

```
    }  
}
```

**Not:** Bu yaziya ilerleyen donemlerde eklemeler yapilacaktir. Simdilik bu kadar !

## Pure Java - 66 Tokenizing & Formatting

[Levent Erguder](#) 07 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde **tokenizing** ve **formatting** konusundan bahsedecegim. Bu yazi ile birlikte bolum 6 sonu olacak insallah.

### Tokenizing

**Token** , simge/sembol/jeton gibi anlamlara gelmektedir. Tokenizing tan kastimiz , buyuk bir icerigi/contenti ufk ufk parcalara bolmektir.Peki bu parcalara neye gore ayiracagiz ? Genelde virgul(,) nokta(.) gibi karakterlere gore ayrim yapilir bunlara delimiter(ayirici/sinirlayici) denilir.

Tokenizing islemini **java.lang.String** , **java.util.Scanner**, **java.lang.StringTokenizer** siniflari yardimi ile yapabiliriz.

`java.lang.String` sinifinda yer alan `split` metodunu kullanabiliriz.

```
public class StringSplitTest {  
  
    public static void main(String[] args) {  
  
        String content = "name1,name2,name3,name4";  
  
        String[] names = content.split(",");  
        for (String s : names) {  
            System.out.println(s);  
        }  
  
        content = "name1.name2:name3:name4.name5@name6";  
        System.out.println("virgul - iki nokta-nokta- @");  
        names = content.split(",|:|@|.");  
        // onceki yazida isledigimiz gibi ,birden fazla delimiter kullanmak icin  
        // veya karakterini kullaniriz.  
        // benzer sekilde noktayı kullanmak icin kacis karakteriyle kullanmamız gereklidir. \\.  
        for (String s : names) {  
            System.out.println(s);  
        }  
    }  
}
```

```
    }  
}
```

**Tokenizing** mantigini onceki bolumle karistirmayalim. Yukarıdaki ornegi dusunecek olursak virgul, nokta gibi karakterleri **birayrac/delimiter** olarak kabul ediyor ve bu kisimlardan ayirma islemi yapiyoruz. Searching mantiginda ise ilgili regex'i bir icerikte/content arama islemi yapiyoruz.

Tokenizing islemi java.util.Scanner sinifi da benzer sekilde yapar.

Ilgili delimiter patternine uygun sekilde icerigi/content token'lara/parcalara ayirir. Varsayılan olarak bosluga gore ayirma islemi yapar.

Dilersek **useDelimiter** metodunu kullanabiliriz.

```
import java.util.Scanner;  
  
public class ScannerTest {  
  
    public static void main(String[] args) {  
  
        String strToken;  
        Scanner s1 = new Scanner("1 true 34 hi");  
        System.out.println("###Test 1 #####");  
        while (s1.hasNext()) {  
            strToken = s1.next();  
            System.out.println(strToken);  
        }  
  
        System.out.println("###Test 2 #####");  
        while (s1.hasNext()) {  
            strToken = s1.next();  
            System.out.println(strToken);  
            // burada cikti vermeyecek. bu islem bir kere yapilir.  
        }  
        s1.close();  
  
        System.out.println("###Test 3 #####");  
        Scanner s2 = new Scanner("1 true 34 hi");  
        int intToken;
```

```

boolean boolToken;

while (s2.hasNext()) {
    if (s2.nextInt()) {
        intToken = s2.nextInt();
        System.out.println("int token:" + intToken);

    } else if (s2.hasNextBoolean()) {
        boolToken = s2.nextBoolean();
        System.out.println("boolean token:" + boolToken);
    } else {
        strToken = s2.next();
        System.out.println("string token:" + strToken);
    }
}

// nextInt nextBoolean gibi metodları kullanabiliriz.
// birada next metodunu kullanmayı unutmayalım yoksa dongu sonlanmaz.

s2.close();

Scanner s3 = new Scanner("1,true,34,hi");

System.out.println("###Test 3 #####");
s3.useDelimiter(",");
while (s3.hasNext()) {
    strToken = s3.next();
    System.out.println(strToken);
}
s3.close();

System.out.println("###Test 4 #####");
String content = "1,true,34,hi";
String[] tokens = content.split(",");
for (String s : tokens ) {
    System.out.println(s);
}

}

```

```
}
```

Searching ve Tokenizing farkini anlamak icin su ornegi inceleyelim ;

```
public class SearchingVSTokenizing {  
  
    public static void main(String[] args) {  
  
        Pattern p = Pattern.compile("\\d");  
        // \d --> [0-9] anlamina gelmektedir.  
  
        String content = "java1 reg2ex!";  
        System.out.println("Searching Test");  
        Matcher m = p.matcher(content);  
  
        while (m.find()) {  
            System.out.print(m.start() + " ");  
            // Searching islemi ilgili regexi bulma islemidir.  
        }  
        System.out.println();  
  
        String[] tokens = content.split("\\d");  
        System.out.println("Tokenizing Test");  
        for (String token : tokens) {  
            System.out.println(token);  
            // Split/tokenizing islemi ise ilgili regex(delimiter) a gore  
            // parcalara ayirma islemidir.  
        }  
    }  
}
```

**java.util.StringTokenizer** sinifini da kullanabiliyoruz.

```
import java.util.StringTokenizer;
```

```

public class StringTokenizerTest {

    public static void main(String[] args) {
        String content = "test1 test2 test3";
        StringTokenizer st = new StringTokenizer(content);
        System.out.println("Bosluga gore token ayirma islemi");
        while (st.hasMoreElements()) {
            System.out.println(st.nextElement());
        }

        String content2 = "test1,test2:test3.test4";
        StringTokenizer st2 = new StringTokenizer(content2, ",|:\\\\.");
        // StringTokenizer icin delimiter arasinda | karakterine gerek yok.
        // StringTokenizer(content2, ", :\\\\.");

        System.out.println("virgule gore token ayirma islemi");
        System.out.println("token count :" + st2.countTokens());

        while (st2.hasMoreElements()) {
            /*
             * hasMoreTokens da kullanilabilir. hasMoreElements, hasMoreTokens
             * metodunu cagirmaktadir.
             * public boolean hasMoreElements() {
             * return hasMoreTokens();
             * }
             */
            System.out.println(st2.nextElement());
            /*
             * nextElement metodu da kullanilabilir. nextElement metodu,
             * nextToken metodunu cagirmaktadir.
             * public Object nextElement() {
             * return nextToken();
             * }
             */
        }
    }
}

```

```
    }  
}
```

## Formatting

C dilinden hatitlayacagimiz printf fonksiyonu vardi. %d karakterini kullanarak cikti elde ederdik. Benzer sekilde ozelligi Javada kullanabiliriz. Formatting konusunu anlayabilmek icin ornegi inceleyelim ;

```
import java.util.Formatter;  
  
public class FormattingTest {  
  
    public static void main(String[] args) {  
  
        System.out.printf("%d", 10);  
        System.out.println();  
  
        // System.out.printf("%f",10);  
        // java.util.IllegalFormatConversionException hatasi verecektir.  
        // %f kayan noktalı sayılar/kusuratlı sayılar icindir.  
  
        /*  
         * b boolean  
         * c char  
         * d integer  
         * f floating point  
         * s string  
         */  
  
        System.out.format("%d", 10);  
        //format metodu da kullanabiliriz.  
        System.out.println();  
  
        System.out.printf("%2$d %1$d %3$f", 123, 456, 10.4);  
        // index$ ifadesi ile arguman olarak verilen degerlerin sirasini  
        // duzenleyebiliriz.
```

```

System.out.println();

System.out.printf("%07d", 123);
// 0 , metoda verilen argumana padding uygular. Sayimiz 7 haneli olana
// kadar soluna 0 ekleyecektir.

System.out.println();

System.out.printf("%+d %+d", 123, -123);
// Varsayılan olarak pozitif sayılarda + işaretini eklenmez. + karakterini
// ekleyip formatlayabiliriz.

System.out.println();

System.out.printf("%(d %(d", 123, -123);
// ( parantez işaretini negatif sayıları parantez içeresine alır.
// negatif sayıyı pozitif olarak formattar , mutlak değerini alır.

System.out.println();

System.out.printf(">%d< \n", 123);
System.out.printf(">%7d< \n", 123);
System.out.printf(">%-7d< \n", 123);
// - karakteri formattamayı sola dayalı yapar.

// Bu formatlama işlemlerini Formatter sınıfı yardımı ile de
// yapabiliriz.

Formatter formatter = new Formatter();
Formatter newFormat = formatter.format("%2$d %1$d", 123, 456);
System.out.println(newFormat);
formatter.close();
}
}

```

# BÖLÜM-7

## Pure Java - 67 Generics & Collections – hashCode & equals & toString

[Levent Erguder](#) 11 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde java.lang.Object sinifinda yer alan **toString** , **equals** ve **hashCode** metotlarini inceleyecegiz. Hatirlayacagimiz gibi **java.lang.Object** sinifi , Java'da sinif hiyerarsisinin en tepesindedir ve varsayılan olarak her sinif icin IS-A Object önermesi doğrudur. Bununla birlikte java.lang kutuphanesi varsayılan olarak import edilir.

### **toString**

toString metodu objenin temsili(representation) string degerini doner.

Bu metodun override edilmesi önerilir neden mi su kucuk ornegimizi inceleyelim.

Basit bir ornek yazalim ;

```
public class Animal {  
  
    int weight = 20;  
    int height = 10;  
  
    public static void main(String[] args) {  
        Animal animal = new Animal();  
        System.out.println(animal);  
    }  
}
```

```
}
```

System.out.println metodunu kullandığımızda bu metod dolaylı olarak toString metodunu çağırmaktadır.  
paketismi+sinifismi+@Sayı şeklinde bir çıktı verecektir. Bu okunabilirlik açısından pek hoş durmamaktadır.  
toString metodunu incelediğimizde neden böyle bir çıktı verdigini görebiliriz.

```
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

Eclipse’te alt+shift+s yardımı ile Generate toString()... sekmesini kullanabiliriz ve otomatik olarak override edebiliriz.

```
public class Animal {  
  
    int weight = 20;  
    int height = 10;  
  
    public static void main(String[] args) {  
        Animal animal = new Animal();  
        System.out.println(animal);  
    }  
  
    @Override  
    public String toString() {  
        return "Animal [weight=" + weight + ", height=" + height + "]";  
    }  
}
```

Override ettiğimiz toString metodunda return ifadesi cıktımız olacaktır. Burayı dilediğimiz gibi değiştirebiliriz.

## equals

Onceki bölümde equals metodunu ve == kontrolunu inceledik. Hatırlayacağımız gibi == kontrolü referans tipteki değişkenler için ilgili referans değişkenler aynı objeyi mi referans ediyor kontrolu yapmaktadır. Eğer bu iki referans değişken aynı objeyi referans ediyorsa true aksi takdirde false değeri donmaktadır. Wrapper ve String sınıfları için özel durumlar söz konusuydu. Immutable özelliğini ve Pool konusunu unutmayalım !  
Eğer iki referans değişkeninin aynı objeyi gösterip göstermediği kontrolü gerekiyorsa == , anlamalı olarak/meaningfully eşitlik kontrolü gerekiyorsa **equals** kullanılmalıdır.

```

public class EqualsTest {

    public static void main(String[] args) {
        Integer i1 = new Integer(10);
        Integer i2 = new Integer(10);

        System.out.println(i1 == i2);
        // i1 ve i2 referans degiskenleri farkli objeleri gostermektedir. Bu
        // nedenle false deger doner.

        System.out.println(i1.equals(i2));
        // equals metodu referans kontrolu yapmaz. Anlamsal olarak/meaningfully
        // esitlik kontrolu yapar. true deger donecektir.
    }
}

```

Integer sınıfında equals metodu override edilmistir ;

```

public boolean equals(Object obj) {
    if (obj instanceof Integer) {
        return value == ((Integer)obj).intValue();
    }
    return false;
}

```

Object sınıfındaki equals metodu ;

```

public boolean equals(Object obj) {
    return (this == obj);
}

```

Integer sınıfı gibi diğer Wrapper sınıflarında ve String sınıfında da equals metodu override edilmiş durumdadır. Peki override edilmeyen durumda durum nasıl olmaktadır ;

```

public class Animal {

```

```

public static void main(String[] args) {
    Animal animal = new Animal();
    Animal animal2 = new Animal();
    Animal animal3 = animal;

    System.out.println(animal == animal2);
    // == kontrolu false donecektir. == kontrolu bu iki referans degiskenin
    // ayni objeyi gosterip gostermediginin kontrolunu yapar.

    System.out.println(animal == animal3);
    // == kontrolu animal ve animal3 referans degiskenleri icin true
    // donecektir.
    // animal referans degiskeni animal3 referans degiskenine atandiktan
    // sonra animal3 ve animal referans degiskeni ayni objeyi referans
    // etmektedir.

    System.out.println(animal.equals(animal2));
    // java.lang.Object sinifinda bulunan equals metodu varsayılan olarak ==
    // de oldugu gibi referans kontrolu yapar.
    // Burada false donecektir

    System.out.println(animal.equals(animal3));
    // == kontrolu true dondugu icin yani animal ve animal3 ayni objeyi
    // gosterdikleri icin
    // equals da true donecektir.
}

}

```

**equals** metodunu override etmedigimiz surece varsayılan durum icin yani `java.lang.Object` sinifinda yer alan haliyle kullanabiliriz. `java.lang.Object` sinifinda `equals` metodu `==` kontrolu gibi calismaktadir yani kontrol edilen 2 referans degisken ayni objeyi gosteriyorsa true aksi durumda false deger donecektir.

```
public class Animal {
```

```
    int weight;
```

```

int height;

public Animal(int weight, int height) {
    this.weight = weight;
    this.height = height;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Animal other = (Animal) obj;
    if (height != other.height)
        return false;
    if (weight != other.weight)
        return false;
    return true;
}

public static void main(String[] args) {
    Animal animal = new Animal(10, 20);
    Animal animal2 = new Animal(10, 20);

    System.out.println(animal == animal2);
    // == kontrolu false doner. animal ve animal2 ayni objeye referansta
    // bulunmuyor.

    System.out.println(animal.equals(animal2));
    // true deger doneciktir. Override edilen equals metodunda height ve weight degerlerinin
    // ayni olmasi 2 Animal objesinin anlamsal olarak/meaningfully esit
    // olmasini saglar.
    // Eclipse ile otomatik olarak olusturduk. Dilersek kendimiz degisiklik
}

```

```

// yapabiliriz.

Animal animal3 = new Animal(20, 40);
Animal animal4 = new Animal(40, 70);

System.out.println(animal3 == animal4);
System.out.println(animal3.equals(animal4));
}

}

```

Unutmayalim equals metodu parametre olarak java.lang.Object sınıfı tipinde bir değişken almaktadır. Sonrasında cast etmek gerekmektedir.

#### equals Contract

equals metodunun sahip olması gereken bazı özellikler mevcuttur;

##### reflexive(yansımlı)

x.equals(x); //true olmalıdır.

##### symmetric(simetriksiz)

x.equals(y); //true ise

y.equals(x) //true olmalıdır.

##### transitive(gecisli)

x.equals(y); //true

y.equals(z); //true ise

x.equals(z); //true olmalıdır

##### consistent(tutarlı)

equals metodunun bir çok defa kullanılmamasında sonuc değişmemelidir. Tutarlı olmalıdır.

x.equals(y); // true ise her zaman true donmelidir.

##### null reference

x.equals(null); // false değer donmelidir.

## hashCode

java.lang.Object sınıfında yer alan **hashCode** metodu bir native metottur. Bu metot JVM tarafından uygulanır/implements.

Her sınıf için varsayılan olarak IS-A Object önermesi doğrudur. Dolayısıyla her sınıf varsayılan olarak hashCode metodunu kullanabilir.

```
public native int hashCode();
```

hashCode 32 bit signed/isaretli tekil bir değerdir/sayıdır. hashCode değerini objeler için ID değeri olarak düşünübiliriz.

HashMap ,Hashtable HashSet gibi Collection yapıları ilgili objelerin store/ saklanması/ doldurulması konusunda hashCode değerini kullanır.

Yazının basında `toString` metodundan bahsetmistiğim. `println` metodu dolaylı olarak `toString` metodunu çağrılmaktaydı ve obje referanslarının çıktısı olarak paketismi+sinifismi+@+hashCode(hex formatta) değerini vermektedir. Burada hashCode değerimiz hexadecimal formattadır. Bu değerin decimal/onluk karşılığını hashCode metodunu çağırarak görebiliriz;

```
public class Dog {  
  
    public static void main(String[] args) {  
        Dog dog = new Dog();  
        Dog dog2 = new Dog();  
  
        System.out.println(dog.hashCode());  
        System.out.println(dog2.hashCode());  
  
        System.out.println(dog);  
        System.out.println(dog2);  
    }  
  
}
```

hashCode metodunun override edilmesi Collection yapılarında özellikle gerekmektedir. hashCode değeri ilgili objeyi collection/koleksiyon/yığın arasında yer tespitinin yapılip(locate) bulabilmemizi sağlar.

hashCode metodunun uygun/appropriate ve dogru bir sekilde/correct override edilmesi onemlidir. Eger uygun bir sekilde override edilmezse verimsiz bir durum soz konusu olacaktir/inefficient. hashCode metodu ilgili tum objeler icin ayni degeri donebilir bu legaldir fakat uygun/appropriate degildir.

```
public int hashCode() { return 2014; }
```

Peki hashCode degerinin ayni olmasi neden verimsiz olmaktadır ? Collection yapısında bir objenin bulunması 2 asamalıdır ;

- Dogru bucket/sepeti bul.(hashCode)
- Aranan dogru elamani bul>equals)

Eger yukarida oldugu gibi tum objeler icin tek bir hashCode degeri donmesi butun objelerin ayni bucket/sepet icerisinde olmasi demektir. Elimizde 1000 tane obje oldugunu dusunursek hashCode metodu bu durumda pek yardimci olmayacaktir cunku butun objeler tek bir bucket/sepet icerisinde yer almaktadir. Bunun yerine hashCode metodunun uygun bir sekilde override edilmesi sonucunda cok daha fazla bucket/sepet olacaktir. Ilgili bucket/sepeti bulduktan sonra icerisinde daha az obje yer alacaktir bu durumda aranan objeyi bulmak cok daha kolay olacaktir.

Kisacasi tum objeler icin ayni hashCode degeri verilebilir bu durum legal olmasina ragmen Collection yapisi icin yavas dolayisiyla verimsiz olacaktir.

Eger iki obje referansi icin equals kontrolu true donuyorsa hashCode kontrolu de true olmalıdır.

```
public class EqualsTest {  
  
    public static void main(String[] args) {  
        Integer i1 = new Integer(10);  
        Integer i2 = new Integer(10);  
  
        System.out.println(i1 == i2);  
        // i1 ve i2 referans degiskenleri farkli objeleri gostermektedir. Bu  
        // nedenle false deger doner.  
  
        System.out.println(i1.equals(i2));  
        // equals metodu referans kontrolu yapmaz. Anlamsal olarak/meaningfully  
        // esitlik kontrolu yapar. true deger donecektir.
```

```

        System.out.println(i1.hashCode());
        System.out.println(i2.hashCode());
        System.out.println(i1.hashCode() == i2.hashCode());
        // Eger iki obje referansi icin equals kontrolu true donuyorsa hashCode
        // kontrolu de true olmalidir.

    }
}

```

equals metodu override edilince hashCode metodunu da override etmek uygun olacaktir. Animal sinifi orneginde sadece equals metodunu override etmistik simdi de Eclipse yardimi ile 2 metodu da override edelim.

```

public class Animal {

    int weight;
    int height;

    public Animal(int weight, int height) {
        this.weight = weight;
        this.height = height;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + height;
        result = prime * result + weight;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)

```

```

        return true;

    if (obj == null)
        return false;

    if (getClass() != obj.getClass())
        return false;

    Animal other = (Animal) obj;

    if (height != other.height)
        return false;

    if (weight != other.weight)
        return false;

    return true;
}

public static void main(String[] args) {
    Animal animal = new Animal(10, 20);

    Animal animal2 = new Animal(10, 20);

    System.out.println(animal.equals(animal2));
    System.out.println(animal.hashCode());
    System.out.println(animal2.hashCode());
}
}

```

Eger iki obje referansi icin equals kontrolu false donuyorsa bu durumda hashCode degeri ayni da olabilir farkli da olabilir.

Serialization soz konusu oldugunda transient olan instance degiskenler default olarak deserialization islemine tabi tutuluyordu. Bu nedenle equals ve hashCode metodlarini override ederken transient instance degiskenleri kullanmamak gerekir.

equals ve hashCode metodlari icin su onermelere uygun sekilde override islemi yapilmalidir.

x.equals(y)==true ise  
x.hashCode()==y.hashCode() true olmali !

`x.equals(y)== false` ise

`x.hashCode() == y.hashCode()` `true` ya da `false` olabilir.

`x.hashCode() != y.hashCode()` `true` ise

`x.equals(y) == false` olmalıdır

`x.hashCode() == y.hashCode()` `true` ise

`x.equals(y) == true` veya `false` olabilir.

Override the equals correctly , we are same instance of humanity.

## Pure Java - 68 Generics & Collections - Collection API

[Levent Erguder](#) 14 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Onceki bolumlerde array konusunu gorduk. Hatirlayacagimiz gibi arraylerin belirli bir size/boyutu vardi. Cogu zaman daha esnek veri yapılarina/data structure ihtiyac olmaktadır. Java bu konuda son derece esnek olan **Collection API** sini sunmaktadır. Collection API , array/diziler uzerine uygulanmaktadır(implements) OCP 6 sinavi bunyesinde olan collection interface'leri sunlardır;

- **Collection**
- **List**
- **Set**
- **SortedSet**
- **NavigableSet**
- **Map**
- **SortedMap**
- **NavigableMap**
- **Queue**

OCP 6 sinavi bunyesinde olan collection sınıfları

**Map'ler**

- **HashMap**
- **Hashtable**
- **TreeMap**

- **LinkedHashMap**

#### Set'ler

- **HashSet**
- **LinkedHashSet**
- **TreeSet**

#### List'ler

- **ArrayList**
- **LinkedList**
- **Vector**

#### Queue

- **PriorityQueue**

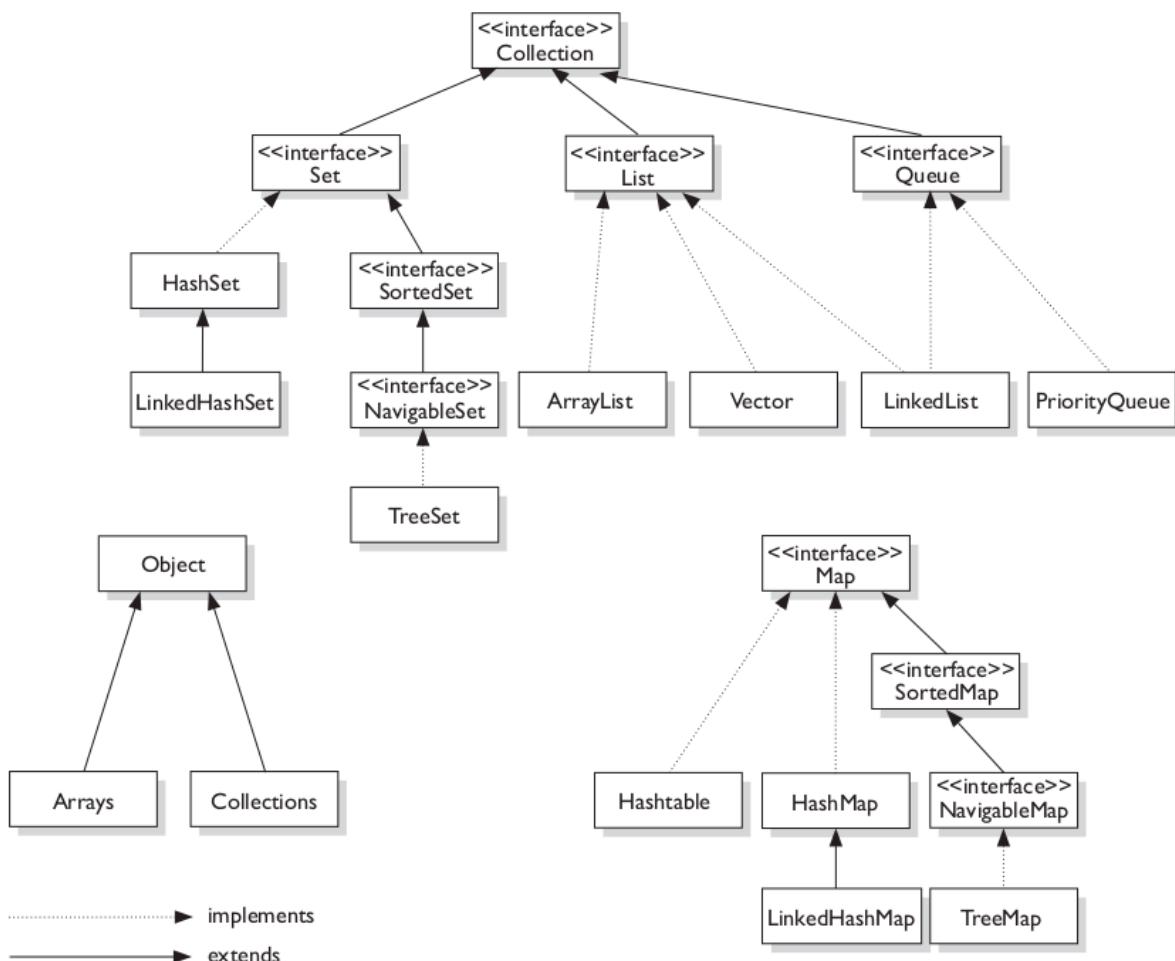
#### Utility

- **Collections**
- **Arrays**

Bu interface ve classlar **java.util** paketinde yer almaktadir. Listede dikkat edelim **java.util.Collection** bir interface ,**java.util.Collections** ise bir sinifdir.

List , Set , Queue arabirimleri Collection arabirimini kalitmaktadir.

Map arabiri ise Collection arabirimini kalitmaz



Collection'lar yapıları geregi 4'e ayrılır ;

### **Ordered & Unordered**

Bir Collection'in ordered/sıralı olması elemanlarının belirli/specific bir sıradan eklenmesi anlamına gelmektedir. Örneğin java.util.ArrayList ordered bir yapıya sahiptir, eklendiği sıradan(index based) elemanlar saklanacaktır/store.

Bir diğer örnek olarak java.util.HashMap 'e eklenen elemanlar eklendiği sıradan tutulmaz , dolayısıyla unordered bir yapıya sahiptir. java.util.LinkedHashMap ya da java.util.LinkedHashSet ise eklendiği sıradan(index based) elemanlar saklanır/store.

### **Sorted & Unsorted**

Sorted/sıralanmış collection yapısından kasıt belirli kurallara göre elemanların sıralı olması demektir. Varsayılan olarak natural order mantığı kullanılmaktadır. Yani alfabeyi düşünduğumuz zaman A harfi B harfinden önce gelmektedir ya da tamsayılar için 1 , 2 den önce gelmektedir. Bu şekildeki sıralama mantığına natural order denilmektedir.

Peki Car gibi bir sınıf objelerini nasıl sıralayacağımız ? String veya sayılar için natural order kavramından bahsetmek mümkün değildir. Bu durumda java.util.Comparable ya da java.util.Comparator arabirimlerini kullanmamız gerekmektedir. İlerleyen bölümlerde bu 2 interface yapısının kullanımı inceleyeceğiz.

Sorted yapıya sahip Collection'lar aynı zamanda Ordered yapıya sahiptir fakat Ordered yapıya sahip Collection'lar Sorted değildir. java.util.TreeSet ve java.util.TreeMap , Sorted bir yapıya sahiptir.

Simdi de List, Set , Map ve Queue interfacelerini ve bu sınıfları uygulayan sınıfları kısaca inceleyelim

### **java.util.List**

Bir List için önemli olan index'tir. Metotları index based bir yapıya sahiptir. List arabirimini uygulayan/implements java.util.ArrayList , java.util.Vector , java.util.LinkedList index based ordered bir yapıya sahiptir.

Bir List elemanları duplicate (aynı elemandan birden fazla) yapısında olabilir.

### **java.util.ArrayList**

java.util.ArrayList , java.util.List arabirimini uygulamaktadır. Ordered bir yapıya java.util.LinkedList sahiptir fakat Sorted değildir. ArrayList'i dinamik olarak genişleyebilen array gibi düşünübiliriz. ArrayList fast iteration ve fast random access söz konusu olduğunda java.util.LinkedList'ten daha verimlidir.

### **java.util.LinkedList**

LinkedList sınıfı da benzer şekilde index based Ordered bir yapıya sahiptir. Fast Insertion/deletion işlemi kullanılmaktansa bu durumda LinkedList daha verimli olacaktır. LinkedList sınıfı Deque arabirimini aracılığıyla Queue arabirimini uygulamaktadır. Bu nedenle LinkedList IS-A Queue önermesi doğrudur.

### **java.util.Vector**

Vector sınıfı Java'nın 1.2 versiyonundan itibaren vardır. Vector temel olarak ArrayList'e benzemektedir fakat Vector'un metodları thread-safe yani synchronized 'dır. Vector sınıfı yerine ArrayList tercih

edilmektedir. Thread safe yapisi gerektigi durumda java.util.Collections sinifinda yer alan metotlar yardimi ile yapilmaktadir. Ilerleyen bolumlerde inceleyecegiz.

#### **java.util.Set**

Set /kume icerisinde elemanlar matematikteki kume kavraminda oldugu gibi tekil olmak zorundadir. Duplicate elemanlara izin vermez.

#### **java.util.HashSet**

HashSet unsorted ve unordered yapiya sahiptir. Elemanlari eklerken/insert hashCode degerini kullanir. Verimli/Efficient hashCode() metodu performans acisinden fayda saglayacaktir. Ordered in onemli olmadigi duplicate izin verilmeyen bir veri yapisina ihtiyac duyuldugunda HashSet kullanilabilir.

#### **java.util.LinkedHashSet**

LinkedHashSet , HashSet in ordered versiyonudur. Eklendirme sirasina gore elemanlar siralanacaktır. Duplicate yapisina izin verilmeyen ve ordered bir veri yapisina ihtiyac duyuldugunda LinkedHashSet kullanilabilir.

#### **java.util.TreeSet**

TreeSet ordered ve sorted bir yapiya sahiptir. TreeSet elemanlari natural order yapida siralanir bu siralanma ascending/kucukten buyuge dogru yapida olacaktir.

#### **java.util.Map**

Map yapisi unique/essiz/tekil bir identifier/key yapisi ve bu key'e karsilik gelen value yapisi uzerine kurulmustur. Key unique olmak zorundadir fakat value duplicate olabilir.

#### **java.util.HashMap**

HashMap unsorted ve unordered yapiya sahiptir. HashSet 'te oldugu gibi elemanların eklenmesi hashCode degerine gore edilir. Verimli/Efficient bir hashCode metodu verimlilik saglayacaktir. HashMap 1 tane null key ve birden fazla null value/deger eklememize izin verir. (key , null degerde bile olsa unique olmalidir ve value null bile olsa duplicate olabilir)

#### **java.util.Hashtable**

java.util.Vector sinifi gibi java.util.Hashtable sinifi da Javanin eski versiyonlarından beri vardir. Hashtable ismine dikkat edelim HashTable degil Hashtable ismine sahiptir. Vector sinifi gibi Hashtable sinifi da thread-safe yapiya sahiptir.  
Hashtable null key ve null value/deger eklenmesine izin vermez.

#### **java.util.TreeMap**

TreeMap , TreeSet gibi sorted ve ordered yapiya sahiptir. Benzer sekilde natural ordered yapisina uygun sekilde elemanlar siralanir. Natural ordered yapisina uygun olmayan durum icin java.util.Comparable veya java.util.Comparator arabirimlerinden yararlanilir.

#### **java.util.Queue**

Queue/kuyruk yapisi FIFO(first in first out) kuralina gore calisir. Ordered bir yapiya sahiptir. Elemanlar sona eklenir , bastan cikartilir. (First in first out)

#### **java.util.PriorityQueue**

Java 5 te eklenen PriorityQueue priority-in , priority out yapisini uygular. PriorityQueue ordered yapiya sahiptir.

Burada Collection API'sinde yer alan arabirimleri/interface ve siniflari kisaca inceledik. Ilerleyen bolumdelerde bu siniflarla ilgili ornekleri yaparken tekrar deginecegiz.

# Pure Java - 69 Generics & Collections – ArrayList – 01

[Levent Erguder](#) 18 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Onceki bolumlerde Collection API hakkında on bilgiler verdik. Bu bolumde elimizi koda bulayacagiz. Ilk olarak java.util.ArrayList sinifini kod ornekleri ile anlamaya calisacagiz. Bu yazida Java **Generics** konusuna da deginecegiz fakat asil**Generics** konusunu ilterleyen bolumerde isleyecegiz.

**java.util.ArrayList** sinifi **java.util.List** arabirimini uygulamaktadir. ArrayList'ler dinamik olarak buyuyebilirler ve eleman ekleme(insert) ve arama(search) islemlerinde array/dizilere gore cok daha verimlidir.

java.util.ArrayList sinifinin tanimlanmasina baktigimizda ;

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{...}
```

Illerleyen yazılarda buradaki E'nin ne anlamına geldiginden bahsedecegiz. Simdi en basit orneklerle baslayalim ;

```
import java.util.ArrayList;

class Test {

}

public class HelloArrayList {

    public static void main(String[] args) {
        ArrayList myArrayList = new ArrayList();
        // ArrayList'e elemani add metodu ile ekleyebiliriz.
        myArrayList.add(10);
        myArrayList.add(20);
        myArrayList.add("test");
        myArrayList.set(2, "new set text");
        // set metodu ile ilgili indexteki elemani override edebiliriz.
        myArrayList.add(true);
        myArrayList.add(new Test());
        // overloaded add metodunu kullanarak ilgili indexe eleman
        // ekleyebiliriz.
        myArrayList.add(1, '#');
    }
}
```

```

        for (int i = 0; i < myArrayList.size(); i++) {
            System.out.println(myArrayList.get(i));
        }
    }
}

```

Hatırlayacağımız gibi diziler aynı tipte elemanlar alabiliyordu. ArrayList'e dikkat edecek olursak int, String, boolean , char hatta kendi oluşturduğumuz Test sınıfı tipinde de elemanlar almaktadır. ArrayList'e diledigimiz elemanları ekleyebiliriz.

ArrayList'e eleman eklemek için overloaded **add** metodunu kullanırız. Index değerini arguman olarak verdigimiz **add** metodunu kullanarak diledigimiz index değerine ekleme yapabiliriz. **get** metodunu kullanarak index değerini vererek elemanlara ulaşabiliriz. **set** metodu ile de ilgili indexteki elemani override edebiliriz. Bir örnek daha yazalım ve *size* ,*isEmpty* , *contains* , *indexOf* , *remove* ve *clear* metodlarını inceleyelim.

```

import java.util.ArrayList;
import java.util.List;

public class ArrayListMethodsTest {

    public static void main(String[] args) {
        List myArrayList = new ArrayList();

        boolean isEmpty = myArrayList.isEmpty();
        int size = myArrayList.size();

        System.out.println("isEmpty:" + isEmpty);
        System.out.println("size:" + size);

        myArrayList.add(20);
        myArrayList.add(20);
        myArrayList.add("10");

        // eleman ekledikten sonra tekrar test edelim.
        isEmpty = myArrayList.isEmpty();
        size = myArrayList.size();
    }
}

```

```
System.out.println("isEmpty:" + isEmpty);
System.out.println("size:" + size);

boolean contains = myArrayList.contains(20);
boolean contains2 = myArrayList.contains("10");

System.out.println("contains:" + contains);
System.out.println("contains:" + contains2);

// eger yoksa -1 doner
int indexOf = myArrayList.indexOf(50);

// eger 1den fazla varsa ilk index degeri doner.
int indexOf2 = myArrayList.indexOf(20);

System.out.println("indexOf:" + indexOf);
System.out.println("indexOf:" + indexOf2);

// elemanlari yazdiralim.
for (Object o : myArrayList) {
    System.out.print(o + " ");
}
System.out.println();

// elemani silelim.
myArrayList.remove(0);

// elemanlari yazdiralim.
for (Object o : myArrayList) {
    System.out.print(o + " ");
}

// clear metodu tum elemanlari siler/temizler.
myArrayList.clear();
```

```

size = myArrayList.size();

System.out.println("\nsize:" + size);

}
}

```

Bir baska ornek olarak *addAll* , *removeAll* ve *subList* metodlarini da inceleyelim ;

```

import java.util.ArrayList;
import java.util.List;

public class MoreArrayListMethodstest {

    public static void main(String[] args) {
        ArrayList myList = new ArrayList();
        myList.add(10);
        myList.add(20);
        myList.add(30);
        myList.add(40);

        ArrayList myList2 = new ArrayList();
        myList2.add(5);
        // addAll metodu ile tum collection icerigini ekleyebiliriz.
        myList2.addAll(myList);

        for (Object o : myList) {
            System.out.print(o + " ");
        }

        System.out.println();
        for (Object o : myList2) {
            System.out.print(o + " ");
        }

        List mySubList = myList.subList(2, 4);
    }
}

```

```

        System.out.println();
        for (Object o : mySubList) {
            System.out.print(o + " ");
        }

myArrayList2.removeAll(myArrayList);
// removeAll metodu ile ekledigimiz collectioni cikartabiliriz.
System.out.println();
for (Object o : myArrayList2) {
    System.out.print(o + " ");
}

myArrayList.removeAll(myArrayList);
System.out.println();
for (Object o : myArrayList) {
    System.out.print(o + " ");
}

}
}

```

Genelde ArrayList’erde yukaridaki orneklerde oldugu gibi farkli farkli turlerde elemanlar olmaz bunun yerine array/dizilerde oldugu gibi ayni tipten elemanlar olmasi istenir. Java 1.5 ile **Generics** yapisi eklenedi. **Generics** yapisi ile Collectionlara eklenebilecek elemanları kısıtlayabiliriz.

```

import java.util.ArrayList;

public class GenericArrayListTest {

    public static void main(String[] args) {
        ArrayList <String> stringArrayList = new ArrayList <String>();
        stringArrayList.add("test");
        stringArrayList.add("test2");
        // Generic ifadesi kullanildigi icin sadece String
        // ekleyebiliriz.
    }
}

```

```

        // stringArrayList.add(10); //derleme hatasi verir

    // Generic <String> yapisi sadece

        // String elemanlara izin vermektedir.

    }

}

```

- Collection tanimlamalari/declaration genelde **Coding to an interface** kuralina uygun sekilde yapilir.
- Generic yapisinda < > primitive tipler kullanamayiz.Sadece reference type kullanabiliriz.
- Collectionlar'in elemanlari objedir , primitive elemanlar tutmazlar. Ornegin int tipindeki elemanlar icin autoboxing uygulanir.

```

import java.util.ArrayList;
import java.util.List;

public class CodingToAnInterfaceTest {

    public static void main(String[] args) {

        List <Integer> myList = new ArrayList <Integer>();

        // referencia degiskenen tipi List, arabirim.

        // objemizin tipi ArrayList. Coding to an interface yapisina uygun.

        // #madde1

        // List myList = new ArrayList(); // int kullanamayiz !

        // #madde2

        List beforeJava5 = new ArrayList();
        beforeJava5.add(new Integer(1989));
        // Java5 ten once autoboxing olmadigi icin , kendimiz boxing yapmamiz
        // gereklidir.

        List <Integer> afterJava5 = new ArrayList <Integer>();
        afterJava5.add(1989);
        // Java5ten sonra autoboxing islemi yapilir.
        // Her iki durumda da elemanlar primitive degil!
    }
}

```

```
// #madde 3  
}  
}
```

## Pure Java – 70 Generics & Collections – ArrayList – 02

[Levent Erguder](#) 18 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Onceki yazida java.util.ArrayList sinifini incelemeye baslamistik. Bu yazida ArrayList ile ilgili konulara devam edecegiz.

ArrayList'ler arraylerle benzerlik gosterirler. Java su donusumlere izin verir.

- java.util.List → array
- array → java.util.List

java.util.Arrays sinifinda yer alan asList metodunu kullanarak bir diziyi/array List'e cevirebiliriz.

```
import java.util.Arrays;  
import java.util.List;  
  
public class ArraysAsListMethod {  
  
    public static void main(String[] args) {  
        String[] namesArray = { "names1", "names2", "names3", "names4" };  
  
        List namesList = Arrays.asList(namesArray);  
        // java.util.Arrays sinifinda bulunan asList metodu ile diziyi List'e  
        // donusturabiliriz.  
  
        int size = namesList.size();  
        System.out.println("size:" + size);  
  
        for (String str : namesArray) {  
            System.out.println(str);  
        }  
    }  
}
```

```

// List'teki elemani guncellersek otomatik olarak dizi/arraydeki eleman
// da güncellenecektir.

namesList.set(0, "set new names1");

for (String str : namesArray) {
    System.out.println(str);
}

// Arrays.asList metodunu kullanarak olarak elde ettigimiz List'e yeni
// eleman ekleyemeyiz!
// namesList.add("java.lang.UnsupportedOperationException");

}

}

```

- List'teki elemani guncellersek otomatik olarak dizi/arraydeki eleman da güncellenecektir.
- Arrays.asList metodunu kullanarak olarak elde ettigimiz List'e yeni eleman ekleyemeyiz!

Bir diger ornek olarak , toArray metodunu kullanabiliyoruz. Overloaded toArray metodu yardimi ile List'leri array'e cevirebiliriz.

```

import java.util.ArrayList;
import java.util.List;

public class ToArrayMethodTest {

    public static void main(String[] args) {

        List namesList = new ArrayList();

        namesList.add("name1");
        namesList.add("names");
        namesList.add("name3");
        namesList.add("name4");
    }
}

```

```

Object[] namesObjectArray = namesList.toArray();
// Object array

for (Object name : namesObjectArray) {
    System.out.print(name + " ");
}

System.out.println();

String[] namesStringArray = new String[namesList.size()];
namesStringArray = namesList.toArray(namesStringArray);
// String array

for (String name : namesStringArray) {
    System.out.print(name + " ");
}

System.out.println();

namesStringArray[0] = "new name1";
System.out.println(namesStringArray[0]);
System.out.println(namesObjectArray[0]);
System.out.println(namesList.get(0));
// asList metodunun aksine degisiklikten etkilenmez.

namesList.add("add new name");
// List'e yeni eleman ekleyebiliriz.

}

```

Suana kadar List elemanlarında dolasmak icin gelistirilmis for dongusu kullandik.Bu islemi java.util.Iterator ve java.util.ListIterator interface/arabirimleri yardimi ile de yapabiliriz.

```

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

```

```

import java.util.ListIterator;

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

        List names = new ArrayList();
        names.add("name1");
        names.add("name2");
        names.add("name3");
        names.add("name4");

        System.out.println("For Dongusu");
        for (int i = 0; i < names.size(); i++) {
            System.out.print(names.get(i) + " ");
        }
        System.out.println("\nGelismis For Dongusu");
        for (String name : names) {
            System.out.print(name + " ");
        }

        System.out.println("\nIterator");
        Iterator iterator = names.iterator();
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
        System.out.println("\nListIterator");
        ListIterator listIterator = names.listIterator();
        while (listIterator.hasNext()) {
            System.out.print(listIterator.next() + " ");
        }

        System.out.println("\nReverse");
        while (listIterator.hasPrevious()) {
            System.out.println(listIterator.previous());
        }
    }
}

```

```
    }  
}
```

## Pure Java - 71 Generics & Collections - Sort & Searching

[Levent Erguder](#) 20 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazida **java.util.Collections** ve **java.util.Arrays** utility siniflari yardimi ile sorting/siralama konusuna deginecegiz. Hatirlayacagimiz gibi Collection bir arabirimdir , Collections ise bir siniftir. Bunlari birbirine karistirmayalim.

Daha sonrasinda **java.util.Comparable** ve **java.util.Comparator** arabirimlerini inceleyecegiz.

Son olarak searching **Arrays** ve **Collections** konusunu inceleyeyiz.

Son olarak binarySearch metodunu inceleyecegiz.

Ilk olarak **java.util.Arrays** sinifini kullanarak dizileri siralayalim. Dizinin elemanlari sayi ise kucukten buyuge siralama yapacaktir. Dizinin elemanlari String ise unicode degerine gore siralama yapacaktir. Tabi bizim icin faydalı olacak konu sozluk mantiginda harf/kelime siralamasi yapacagidir. Burada dikkat etmemiz gereken nokta buyuk harflerin unicode degerleri kucuk harflerin unicode degerlerinden daha kucuktur.

```
import java.util.Arrays;  
  
public class ArraysSortTest {  
  
    public static void main(String[] args) {  
        Integer[] numbers = { 10, 20, 50, 25, 35, 80, -20, -30, 0, 10 };  
  
        for (int n : numbers) {  
            System.out.print(n + " ");  
        }  
        System.out.println();  
  
        System.out.println("Sorting Numbers");  
        Arrays.sort(numbers);  
  
        for (int n : numbers) {  
            System.out.print(n + " ");  
        }  
    }  
}
```

```

System.out.println("\n");

String[] stringArray = { "de", "", "da", "De", "DE", "a", "A", "1", "Z" };

for (String str : stringArray) {
    System.out.print(str + " ");
}

System.out.println();

System.out.println("Sorting Strings");

Arrays.sort(stringArray);

for (String str : stringArray) {
    System.out.print(str + " ");
}
}

```

Collections sınıfını kullanalım , benzer şekilde int veya String elemanına sahip List'leri sıralayabiliriz.

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CollectionsSortTest {

    public static void main(String[] args) {
        List <Integer> numberList = new ArrayList<Integer>();

        numberList.add(20);
        numberList.add(10);
        numberList.add(50);
        numberList.add(40);
        numberList.add(-20);
    }
}

```

```

        numberList.add(-5);

        for (int i : numberList) {
            System.out.print(i + " ");
        }
        System.out.println();

        Collections.sort(numberList);

        System.out.println("Sorting Numbers");
        for (int i : numberList) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
}

```

Sayı veya String elemanlara sahip dizi ve List'leri natural ordering/dogal sıralama kuralan göre sıraladık. Peki elemanlarımız Car sınıfında olsa bu durumda ne yapmaliyiz ?

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Car {

    public Car(String hp, String brand, Integer price) {
        super();
        this.hp = hp;
        this.brand = brand;
        this.price = price;
    }
}

```

```

private String hp;
private String brand;
private Integer price;

public String getHp() {
    return hp;
}

public void setHp(String hp) {
    this.hp = hp;
}

public String getBrand() {
    return brand;
}

public void setBrand(String brand) {
    this.brand = brand;
}

public Integer getPrice() {
    return price;
}

public void setPrice(Integer price) {
    this.price = price;
}

@Override
public String toString() {
    return "Car [hp=" + hp + ", brand=" + brand + ", price=" + price + "]";
}
}

```

```

public class CollectionCarSort {

    public static void main(String[] args) {
        Car ford = new Car("50", "ford", 100);
        Car fiat = new Car("20", "fiat", 50);
        Car subaru = new Car("500", "subaru", 5000);
        Car bmw = new Car("200", "bmw", 1000);

        List <Car> carList = new ArrayList <Car> ();
        carList.add(ford);
        carList.add(fiat);
        carList.add(subaru);
        carList.add(bmw);

        // Collections.sort(carList);
        // Derleme hatasi verecektir! Siralama islemini neye gore kime gore
        // yapacak ?
    }
}

```

Siralama islemini neye gore kime gore nasil yapacaginin bilgisi olmadigi icin Collections.sort metodu derleme hatasi verecektir. Ornegin price(fiyat) gore siralama yapalim. Bu durumda Comparable arabirimini su sekilde kullanabiliriz.

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Car implements Comparable <Car>{

    public Car(String hp, String brand, Integer price) {
        super();
        this.hp = hp;
        this.brand = brand;
        this.price = price;
    }
}

```

```
}

private String hp;
private String brand;
private Integer price;

public String getHp() {
    return hp;
}

public void setHp(String hp) {
    this.hp = hp;
}

public String getBrand() {
    return brand;
}

public void setBrand(String brand) {
    this.brand = brand;
}

public Integer getPrice() {
    return price;
}

public void setPrice(Integer price) {
    this.price = price;
}

@Override
public int compareTo(Car car) {
    return price.compareTo(car.getPrice());
}
```

```

@Override
public String toString() {
    return "Car [hp=" + hp + ", brand=" + brand + ", price=" + price + "]";
}

}

public class ComparableTest {

    public static void main(String[] args) {
        Car ford = new Car("50", "ford", 100);
        Car fiat = new Car("20", "fiat", 50);
        Car subaru = new Car("500", "subaru", 5000);
        Car bmw = new Car("200", "bmw", 1000);

        List <Car> carList = new ArrayList <Car>();
        carList.add(ford);
        carList.add(fiat);
        carList.add(subaru);
        carList.add(bmw);

        for (Car c : carList) {
            System.out.println(c);
        }
        System.out.println();

        Collections.sort(carList);

        System.out.println("Comparable Sorting");
        for (Car c : carList) {
            System.out.println(c);
        }
    }
}

```

```
}
```

**java.util.Comparable** arabirimini implement ettigimizde **compareTo** metodunu override etmemiz gerekir. Sonrasında return ifadesi içerisinde Integer.compareTo metodunu cagiriyoruz. Hatirlayacagimiz gibi compareTo metodu degerleri dondurebilir;

```
thisObject.compareTo(anotherObject);
```

negative If thisObject < anotherObject  
zero If thisObject == anotherObject  
positive If thisObject > anotherObject

Comparable arabirimini kullanarak bu sekilde diledigimiz state(instance variable)e gore siralama yapabiliyoruz. Ornegin brand'a gore siralama yapmak istersek **compareTo** metodunu su sekilde yazabilirim ;

```
public int compareTo(Car car) {  
    return brand.compareTo(car.getBrand());  
}
```

Eger buyukten-kucuge olacak sekilde siralamak istersek **compareTo** metodunda yer degisikligi yapmamiz yeterli olacaktir;

```
@Override  
public int compareTo(Car car) {  
    return car.getPrice().compareTo(price);  
}
```

Eger **Comparable** arabirimini implement edilirken **Generics** yapisi kullanilmazsa bu durumda **compareTo** metodumuz parametre olarak Car degil Object sinifi tipinde olacaktir.

```
class Car implements Comparable {  
  
    // ....  
  
    @Override  
    public int compareTo(Object o) {  
        Car car = (Car) o;  
        return price.compareTo(car.getPrice());  
    }  
}
```

```
    }  
//....  
}
```

**Comparable** arabirimini inceledik simdi de **Comparator** arabirimini inceleyelim. **Comparator** arabirimde **Comparable** arabirimine benzer mantikta calismaktadir. Aralarindaki temel fark sudur ; Ornegin yukaridaki ornekte elemanlari Car olan bir List'i siralamak istedik bu nedenle Car sinifi Comparable arabimini uyguladi yani List'te hangi eleman varsa o sinif Comparable arabirimini uygulamalidir. Comparator arabirimini ise baska bir sinif uygulayabilir ve bu ayri sinifta ilgili sinifa ait siralama kurali yazilabilir.

Ornegimizi inceleyelim , aciklamalari kod icerisinde bulabilirsiniz;

```
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;  
  
class Car {  
  
    public Car() {  
    }  
  
    public Car(String hp, String brand, Integer price) {  
        super();  
        this.hp = hp;  
        this.brand = brand;  
        this.price = price;  
    }  
  
    private String hp;  
    private String brand;  
    private Integer price;  
  
    public String getHp() {  
        return hp;  
    }
```

```

public void setHp(String hp) {
    this.hp = hp;
}

public String getBrand() {
    return brand;
}

public void setBrand(String brand) {
    this.brand = brand;
}

public Integer getPrice() {
    return price;
}

public void setPrice(Integer price) {
    this.price = price;
}

@Override
public String toString() {
    return "Car [hp=" + hp + ", brand=" + brand + ", price=" + price + "]";
}

}

public class ComparatorTest implements Comparator<Car> {
    // Comparable arabirimini kullanmak icin eger Car sinifini tipinde
    // elemanlardan olusan bir List'i siralama istiyorsak bu durumda mutlaka Car
    // sinifi Comparable arabirimini uygulamalidir.
    // Fakat Comparator arabirimini bu sekilde baska bir sinif ile kullanilabilir.
    // Dilersek Car sinifinda Comparator seklinde uygulayabiliyoruz.

    public static void main(String[] args) {
        Car ford = new Car("50", "ford", 100);

```

```

Car fiat = new Car("20", "fiat", 50);
Car subaru = new Car("500", "subaru", 5000);
Car bmw = new Car("200", "bmw", 1000);

List<Car> carList = new ArrayList <Car>();
carList.add(ford);
carList.add(fiat);
carList.add(subaru);
carList.add(bmw);

for (Car c : carList) {
    System.out.println(c);
}

System.out.println();

ComparatorTest comparatorTest = new ComparatorTest();
// overloaded sort metodunu kullanabiliriz.

// burada 1.arguman siralamak istedigimiz List
// 2.arguman ise Comparator arabirimini uygulayan sinifimiz olacaktir.

Collections.sort(carList, comparatorTest);

System.out.println("Comparator Sorting");
for (Car c : carList) {
    System.out.println(c);
}

}

// Comparator arabirimini uyguladigimiz zaman compare metodunu override
// etmemiz gerekir!
// Comparable arabiriminde ise bu metot compareTo metoduydu.

@Override
public int compare(Car car1, Car car2) {
    return car1.getPrice().compareTo(car2.getPrice());
}

```

```
}
```

Eger buyukken-kucuge siralamak istersek **compare** metodunda su sekilde yer degistirmemiz yeterli olacaktir ;

```
public int compare(Car car1, Car car2) {  
    return car2.getPrice().compareTo(car1.getPrice());  
}
```

Arrays.sort metodu ile natural order'a uygun dizileri siralama yaptik. Benzer sekilde java.util.Comparator arabirimini kullanabiliriz. **Arrays.sort** metodunun overloaded versiyonu mevcuttur.

```
import java.util.Arrays;  
  
import java.util.Comparator;  
  
class Car {  
    private String hp;  
    private String brand;  
    private Integer price;  
  
    public Car(String hp, String brand, Integer price) {  
        super();  
        this.hp = hp;  
        this.brand = brand;  
        this.price = price;  
    }  
  
    public String getHp() {  
        return hp;  
    }  
  
    public void setHp(String hp) {  
        this.hp = hp;  
    }  
  
    public String getBrand() {  
        return brand;  
    }
```

```

    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public Integer getPrice() {
        return price;
    }

    public void setPrice(Integer price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "Car [hp=" + hp + ", brand=" + brand + ", price=" + price + "]";
    }
}

public class ArraysComparator implements Comparator<Car> {

    public static void main(String[] args) {
        Car ford = new Car("50", "ford", 100);
        Car fiat = new Car("20", "fiat", 50);
        Car subaru = new Car("500", "subaru", 5000);
        Car bmw = new Car("200", "bmw", 1000);

        Car[] carArray = { ford, fiat, subaru, bmw };

        // Arrays.sort(carArray);
        // Car cannot be cast to java.lang.Comparable
        ArraysComparator arrayComparator = new ArraysComparator();
        Arrays.sort(carArray, arrayComparator);
    }
}

```

```

        for (Car c : carArray) {
            System.out.println(c);
        }
    }

    @Override
    public int compare(Car car1, Car car2) {
        return car1.getPrice().compareTo(car2.getPrice());
    }
}

```

**Arrays.binarySearch** metodu ile search/arama yapabiliyoruz. **binarySearch** metodunun doğru çalışabilmesi için önceki **Arrays.sort** metodu ile sort işlemi yapılmalıdır.

```

import java.util.Arrays;

public class ArraysBinarySearch {

    public static void main(String[] args) {
        String[] names = { "name2", "name1", "name6", "name4", "name3", "name5" };

        Arrays.sort(names);

        for (String s : names) {
            System.out.println(s);
        }

        int index = Arrays.binarySearch(names, "name3");

        System.out.println("name3 index:" + index);

    }
}

```

**binarySearch** metodunun overloaded hali bulunmaktadır. Onceki ornegimize **binarySearch** metodunu ekleyelim. Unutmayalim oncesinde **Arrays.sort** kullanilmazsa , **binarySearch** metodu dogru calismayacaktir.

...

```
public class ArraysComparator implements Comparator<Car> {

    public static void main(String[] args) {
        Car ford = new Car("50", "ford", 100);
        Car fiat = new Car("20", "fiat", 50);
        Car subaru = new Car("500", "subaru", 5000);
        Car bmw = new Car("200", "bmw", 1000);

        Car[] carArray = { ford, fiat, subaru, bmw };

        // Arrays.sort(carArray);
        // Car cannot be cast to java.lang.Comparable
        ArraysComparator arrayComparator = new ArraysComparator();
        Arrays.sort(carArray, arrayComparator);

        for (Car c : carArray) {
            System.out.println(c);
        }

        int index = Arrays.binarySearch(carArray, bmw, arrayComparator);
        System.out.println("index:" + index);

    }

    @Override
    public int compare(Car car1, Car car2) {
        return car1.getPrice().compareTo(car2.getPrice());
    }
}
```

**binarySearch** metodunu Collections sinifinda da mevcuttur.

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class CollectionBinarySearch {

    public static void main(String[] args) {
        List names = new ArrayList();
        names.add("name2");
        names.add("name1");
        names.add("name6");
        names.add("name4");
        names.add("name3");
        names.add("name5");

        Collections.sort(names);

        for (String name : names) {
            System.out.println(name);
        }

        int index = Collections.binarySearch(names, "name3");
        System.out.println("index name3:" + index);
    }
}

```

**binarySearch** metodunun benzer sekilde Comparator tipinde parametre de alan overloaded hali vardir.

```

...
public class ComparatorTest implements Comparator<Car>{
    // Comparable arabirimini kullanmak icin eger Car sinifini tipinde
    // elemanlardan olusan bir List'i siralama istiyorsak bu durumda mutlaka Car
    // sinifi Comparable arabirimini uygulamalidir.
    // Fakat Comparator arabirimini bu sekilde baska bir sinif ile kullanilabilir.
    // Dilersek Car sinifinda Comparator seklinde uygulayabiliriz.
}

```

```

public static void main(String[] args) {

    Car ford = new Car("50", "ford", 100);
    Car fiat = new Car("20", "fiat", 50);
    Car subaru = new Car("500", "subaru", 5000);
    Car bmw = new Car("200", "bmw", 1000);

    List carList = new ArrayList();
    carList.add(ford);
    carList.add(fiat);
    carList.add(subaru);
    carList.add(bmw);

    for (Car c : carList) {
        System.out.println(c);
    }
    System.out.println();

    ComparatorTest comparatorTest = new ComparatorTest();
    // overloaded sort metodunu kullanabiliriz.
    // burada 1.arguman siralamak istedigimiz List
    // 2.arguman ise Comparator arabirimini uygulayan sinifimiz olacaktir.
    Collections.sort(carList, comparatorTest);

    System.out.println("Comparator Sorting");
    for (Car c : carList) {
        System.out.println(c);
    }

    int index = Collections.binarySearch(carList, bmw, comparatorTest);

    System.out.println("index:" + index);

}

// Comparator arabirimini uyguladigimiz zaman compare metodunu override

```

```

// etmemiz gerekir!

// Comparable arabiriminde ise bu metot compareTo metoduydu.

@Override

public int compare(Car car1, Car car2) {

    return car2.getPrice().compareTo(car1.getPrice());
}

}

```

## Pure Java – 72 Generics & Collections – Set

[Levent Erguder](#) 20 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazida Java Collection API'de bulunan java.util.Set konusunu dolayisiyla java.util.HashSet , java.util.LinkedHashSet ve java.util.TreeSet siniflarini inceleyecegiz. Hatirlayacagimiz gibi java.util.Set bir arabirimdir ve java.util.Collection arabirimini kalitmaktadir.

```
public interface Set<E> extends Collection<E> { .. }
```

Matematikte bir kumede(set) ayni elamanдан sadece birtane olabilir. Benzer sekilde Java'da da Set mantiginda duplicate elamanlara izin verilmez. Set'e eleman eklemek icin List'te oldugu gibi add metodunu kullanabiliriz. List orneklerinde kullandigimiz metodlar **isEmpty** , **size** , **contains** , **remove** , **clear** gibi metodlari kullanabiliyoruz.

```

import java.util.HashSet;

import java.util.Set;

public class HashSetExample {

    public static void main(String[] args) {
        Set<Integer> numbers = new HashSet<Integer>();

        boolean isEmpty = numbers.isEmpty();
        int size = numbers.size();

        System.out.println("isEmpty:" + isEmpty);
        System.out.println("size:" + size);
    }
}

```

```

numbers.add(5);
numbers.add(5);
// Ayni int degerini 2 kere eklemeye izin vermez.
numbers.add(10);
numbers.add(25);
numbers.add(100);
numbers.add(500);
numbers.add(1000);

isEmpty = numbers.isEmpty();
size = numbers.size();
boolean contains = numbers.contains(5);
boolean remove = numbers.remove(25);
System.out.println("isEmpty:" + isEmpty);
System.out.println("size:" + size);
System.out.println("contains:" + contains);
System.out.println("remove:" + remove);

for (Integer number : numbers) {
    System.out.println(number);
}

Set<String> names = new HashSet<String>();
names.add("names1");
names.add("names2");
names.add("names3");
names.add("names4");
names.add("names4");
names.add(new String("names4"));
// ayni String degeri eklememize izin vermez.

for (String name : names) {
    System.out.println(name);
}

```

```
    }  
}
```

Set'in elemanlarını doldurmak için gelismis for dongusu veya Iterator sınıfını kullanabiliriz.

```
import java.util.HashSet;  
import java.util.Iterator;  
import java.util.Set;  
  
public class IteratorSet {  
  
    public static void main(String[] args) {  
  
        Set<Integer> numbers = new HashSet<Integer>();  
  
        numbers.add(10);  
  
        numbers.add(25);  
  
        numbers.add(100);  
  
        numbers.add(500);  
  
        numbers.add(1000);  
  
  
        for (int i : numbers) {  
  
            System.out.println(i);  
        }  
  
        System.out.println();  
  
        Iterator iterator = numbers.iterator();  
  
        while (iterator.hasNext()) {  
  
            System.out.println(iterator.next());  
        }  
  
    }  
}
```

HashSet' icin dikkat edeceklerimiz 2 nokta ;

- Duplicate eleman eklemeye izin vermez.

- Eleman eklenmesi List'te oldugu gibi index based degildir. Elemanlari ekledigimiz sirada HashSet'e eklenmez.(Unordered)

Eger ordered bir Set veri yapisina ihtiyacimiz varsa **LinkedHashSet** kullanabiliriz.

```
import java.util.LinkedHashSet;
import java.util.Set;

public class LinkedHashSetExample {

    public static void main(String[] args) {
        Set<Integer> numbers = new LinkedHashSet<Integer>();

        numbers.add(5);
        numbers.add(5);
        // Ayni int degerini 2 kere eklemeye izin vermez.
        numbers.add(10);
        numbers.add(25);
        numbers.add(100);
        numbers.add(500);
        numbers.add(1000);

        for (Integer number : numbers) {
            System.out.println(number);
        }

        Set<String> names = new LinkedHashSet<String>();
        names.add("names1");
        names.add("names2");
        names.add("names3");
        names.add("names4");
        names.add("names4");
        names.add(new String("names4"));
        // ayni String degeri eklememize izin vermez.

        for (String name : names) {
            System.out.println(name);
        }
    }
}
```

```
        }  
    }  
}
```

List örneklerimizde **toArray** metodu ile Set'i array/diziye dönüştürebiliriz.

```
import java.util.LinkedHashSet;  
import java.util.Set;  
  
public class ToArrayMethod {  
  
    public static void main(String[] args) {  
  
        Set<String> names = new LinkedHashSet<String>();  
  
        names.add("names1");  
        names.add("names2");  
        names.add("names3");  
        names.add("names4");  
        names.add("names4");  
  
        Object[] objectArray = new Object[names.size()];  
        objectArray = names.toArray();  
  
        for (Object o : objectArray) {  
            System.out.println(o);  
        }  
        System.out.println();  
  
        String[] myArray = new String[names.size()];  
        names.toArray(myArray);  
  
        for (String name : myArray) {  
            System.out.println(name);  
        }  
    }  
}
```

```
}
```

Arrays.asList metodunu onceki bolumlerde kullandik. Arrays.asSet metodu yok, bir diziyi Set'e cevirmek icin overloaded yapılandirici yardimi ile kullanabiliriz.

```
import java.util.Arrays;  
import java.util.HashSet;  
import java.util.Set;  
  
public class ArrayToSet {  
    public static void main(String[] args) {  
        String[] names = { "name1", "name2", "name3", "name4", "name5" };  
  
        Set<String> mySet = new HashSet<String>(Arrays.asList(names));  
        mySet.add("name6");  
  
        for (String name : mySet) {  
            System.out.println(name);  
        }  
    }  
}
```

Simdi de String veya Wrapper sinif disinda bir ornek inceleyelim ;

```
import java.util.HashSet;  
import java.util.Set;  
  
class Car {  
}  
  
public class HashSetTest {  
    public static void main(String[] args) {  
        Car car = new Car();  
        Car car2 = new Car();
```

```

Car car3 = new Car();

Set<Car> carSet = new HashSet<Car>();

carSet.add(car);
carSet.add(car2);
carSet.add(car3);

System.out.println(carSet.size());

}
}

```

Bu ornekte 3 elemani da **HashSet**'e ekleyecektir. Bu referans degiskenleri icin **equals** metodu false dondugu icin bunların hic biri duplicate eleman durumunda degildir. Dolayisiyla elemanlar diziye eklenir. **equals** metodunu override ettigimiz durumda ise **equals** true olan referans degiskenleri icin duplicate durumu soz konusudur.

```

import java.util.HashSet;
import java.util.Set;

class Car {

    String hp;
    String brand;
    Integer price;

    public Car(String hp, String brand, Integer price) {
        super();
        this.hp = hp;
        this.brand = brand;
        this.price = price;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((brand == null) ? 0 : brand.hashCode());
    }
}

```

```

        result = prime * result + ((hp == null) ? 0 : hp.hashCode());
        result = prime * result + ((price == null) ? 0 : price.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Car other = (Car) obj;
        if (brand == null) {
            if (other.brand != null)
                return false;
        } else if (!brand.equals(other.brand))
            return false;
        if (hp == null) {
            if (other.hp != null)
                return false;
        } else if (!hp.equals(other.hp))
            return false;
        if (price == null) {
            if (other.price != null)
                return false;
        } else if (!price.equals(other.price))
            return false;
        return true;
    }

    @Override
    public String toString() {
        return "Car [hp=" + hp + ", brand=" + brand + ", price=" + price + "]";
    }
}

```

```

    }

}

public class HashSetTest {

    public static void main(String[] args) {
        Car car = new Car("100HP", "Ford", 1000);
        Car car2 = new Car("100HP", "Ford", 1000);
        // car.equals(car2) true donecektir. Dolayisiyla duplicate olacaktir.

        // car car3 car4 arasinda equals false oldugu icin Set'e duplicate
        // olmadan eklenir.
        Car car3 = new Car("50HP", "Fiat", 1000);
        Car car4 = new Car("10HP", "Ford", 100);

        Set<Car> carSet = new HashSet<Car>();
        carSet.add(car);
        carSet.add(car2);
        carSet.add(car3);
        carSet.add(car4);

        System.out.println(carSet.size());

        for (Car c : carSet) {
            System.out.println(c);
        }
    }
}

```

Son olarak **ordered** ve **sorted** ozellige sahip olan `java.util.TreeSet`'i inceleyelim. Hatirlayacagimiz gibi Collection'lari Generics olmadan da kullanabiliriz. `TreeSet` eger **Generics** yapida tanimlanmadiysa ilk hangi tipte eleman eklendiye sonraki elemanlar da ayni tipte olmalıdır. Bu durum `HashMap`, `LinkedHashMap` , `ArrayList` vs icin gecerli degildir.

Bunun nedeni `TreeSet` sorted oldugu icin farkli tipteki elemanlari siralama yapamayacaktır.

```

import java.util.LinkedHashSet;
import java.util.Set;

```

```

import java.util.TreeSet;

public class TreeSetTest {

    public static void main(String[] args) {

        Set linkedHashSet = new LinkedHashSet();
        linkedHashSet.add(10);
        linkedHashSet.add(20);
        linkedHashSet.add("non-generic durumda string de ekleyebiliriz");

        Set treeSet = new TreeSet();

        treeSet.add(10);
        treeSet.add(20);
        // TreeSet e Integer elemanlar ekledikten sonra String ekleyemeyiz.
        treeSet.add("ClassCastException");
        // java.lang.ClassCastException: java.lang.Integer cannot be cast to
        // java.lang.String
    }
}

```

Bir diger ornegimizi inceleyelim , TreeSet'e Car tipinde eleman ekleyelim ;

```

import java.util.Set;
import java.util.TreeSet;

class Car {

}

public class TreeSetTest2 {
    public static void main(String[] args) {
        Set treeSetCar = new TreeSet();
        Car car = new Car();
    }
}

```

```

        treeSetCar.add(car);

        // Car sinifi Comparable olmadigi icin calisma zamaninda

        // java.lang.ClassCastException hatasi verecektir.

    }

}

```

Car sinifi Comparable olmadigi icin calisma zamaninda java.lang.ClassCastException hatasi verecektir. Peki neden Integer ya da String tipinde eleman ekledigimizde calisma zamaninda hata almiyoruz ?

Bunun nedeni Wrapper siniflar ve String Comparable arabirimini uygulamaktadir.

```

public final class Integer extends Number implements Comparable { ..}

public final class String

    implements java.io.Serializable, Comparable, CharSequence {

```

Car sinifimiz icin **java.util.Comparable** arabirimini uygulayalim.Bu durumda TreeSet'e Car tipinde elemanlar ekleyebiliriz.

```

import java.util.Set;

import java.util.TreeSet;

class Car implements Comparable {

    public Car(String hp, String brand, Integer price) {

        super();
        this.hp = hp;
        this.brand = brand;
        this.price = price;
    }

    private String hp;
    private String brand;
    private Integer price;

    @Override
    public int compareTo(Car car) {
        return price.compareTo(car.price);
    }
}

```

```

    }

    @Override
    public String toString() {
        return "Car [hp=" + hp + ", brand=" + brand + ", price=" + price + "]";
    }
}

public class TreeSetTest3 {

    public static void main(String[] args) {
        Car ford = new Car("50", "ford", 100);
        Car fiat = new Car("20", "fiat", 50);
        Car subaru = new Car("500", "subaru", 5000);
        Car bmw = new Car("200", "bmw", 1000);

        Set treeSetCar = new TreeSet();
        treeSetCar.add(ford);
        treeSetCar.add(fiat);
        treeSetCar.add(subaru);
        treeSetCar.add(bmw);

        for (Car c : treeSetCar) {
            System.out.println(c);
        }
    }
}

```

## Pure Java - 73 Generics & Collections - Map - 01

[Levent Erguder](#) 20 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde **java.util.Map** arabirini ve bu arabimini uygulayan **java.util.HashMap** , **java.util.Hashtable** ,**java.util.LinkedHashMap** siniflarini inceleyecegiz.

Hatirlayacagimiz gibi Map , key/value mantigina gore calismaktadir. Key unique/essiz olmalidir value ise duplicate olabilir.

Basit bir ornek ile baslayalim. **HashMap**'e **put** metodunu kullanarak eleman ekleyebiliriz. **put** metodunda key/value parametreleri alir.

**HashMap** unordered ve unsorted bir yapiya sahiptir , **HashMap** index based yapida calismaz.

**Map**'in elemanlarini dondurmek icin **keySet** metodunu kullanabiliyoruz. Diger kullanimlari da inceleyecegiz.

```
import java.util.HashMap;
import java.util.Map;

public class HashMapTest {

    public static void main(String[] args) {
        Map map = new HashMap();

        map.put(1, "one");
        //Map'lere eleman eklemek icin put metodunu kullaniriz.

        map.put(2, "two");
        map.put(3, "three");
        map.put(4, "four");
        map.put("mykey", "myvalue");

        for (Object key : map.keySet()) {
            System.out.println("key:" + key + " value:" + map.get(key));
        }
    }
}
```

HashMap'in diger metotlarini inceleyelim.

HashMap'te **isEmpty** , **size** , **containsKey** , **containsValue** , **remove** , **clear** metotlarini kullanabiliyoruz. **get** metodu List'teki gibi calismaz. **get** metoduna ilgili key verilir ve geriye value dondurur. **HashMap** index based bir yapiya sahip degildir.

```
import java.util.HashMap;
import java.util.Map;

public class HashMapMethodTest {

    public static void main(String[] args) {
```

```
Map<Integer, String> myMap = new HashMap<Integer, String>();  
  
boolean isEmpty = myMap.isEmpty();  
int size = myMap.size();  
  
System.out.println("isEmpty:" + isEmpty);  
System.out.println("size:" + size);  
  
myMap.put(10, "test1");  
myMap.put(20, "test2");  
myMap.put(30, "test3");  
myMap.put(40, "test4");  
  
String value = myMap.get(20);  
// get metodune key verip value aliyoruz.  
System.out.println("value:" + value);  
  
boolean containsKey = myMap.containsKey(10);  
// key'i 10 olan eleman var mi ?  
System.out.println("containsKey 10 : " + containsKey);  
  
boolean containsValue = myMap.containsValue("test1");  
// value test1 olan elaman var mi ?  
  
System.out.println("containsValue test1 : " + containsValue);  
  
myMap.remove(10);  
// key'i 10 olan elamani sil.  
  
containsKey = myMap.containsKey(10);  
System.out.println("containsKey 10 : " + containsKey);  
  
containsValue = myMap.containsValue("test1");  
System.out.println("containsValue test1 : " + containsValue);
```

```

        isEmpty = myMap.isEmpty();
        size = myMap.size();

        System.out.println("isEmpty:" + isEmpty);
        System.out.println("size:" + size);

        myMap.clear();
        // clear metodu tum Map'teki elemanlari siler.

        isEmpty = myMap.isEmpty();
        size = myMap.size();

        System.out.println("isEmpty:" + isEmpty);
        System.out.println("size:" + size);

    }
}

```

Map yapısında , key unique/essiz/tek olmalıdır , duplicate izin verilmez. Aynı key'e sahip eleman eklendiginde eski elemani override eder. Key farklı olduktan sonra value duplicate olabilir.

```

import java.util.HashMap;
import java.util.Map;

public class HashMapDuplicate {

    public static void main(String[] args) {
        Map<String, String> map = new HashMap<String, String>();

        map.put("key1", "value1");
        map.put("key1", "new value1");
        // key unique olmalıdır. Aynı key'e sahip eleman eklendiginde eski
        // elemani override eder.

        map.put("key2", "value1");
    }
}

```

```

        map.put("key2", "new value1");

        // key farkli olduktan sonra value ayni olabilir.

        for (String key : map.keySet())
        {
            System.out.println(key + " " + map.get(key));
        }

    }
}

```

**java.util.LinkedHashMap , ordered** yapıya sahiptir;

```

import java.util.LinkedHashMap;
import java.util.Map;

public class LinkedHashMapTest {

    public static void main(String[] args) {
        Map<String, String> linkedMap = new LinkedHashMap<String, String>();
        // LinkedHashMap , ekleme sirasina gore calisir. LinkedHashMap ordered
        // yapidadir.

        linkedMap.put("key1", "value1");
        linkedMap.put("key2", "value2");
        linkedMap.put("key3", "value3");
        linkedMap.put("key4", "value4");

        for (String key : linkedMap.keySet()) {
            System.out.println(key + " " + linkedMap.get(key));
            // ordered
        }
    }
}

```

**HashMap** null key degerine ve null value degerlere sahip olabilir. **Hashtable** ise null key veya null degerlere sahip olamaz.

```

import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;

public class NullKeyAndValues {

    public static void main(String[] args) {
        Map hashMap = new HashMap();

        hashMap.put(null, null);
        // HashMap e null key veya null value olabilir.

        Map hashtable = new Hashtable();
        hashtable.put(null, null);
        // Hashtable null key veya null value eklenemez.
        // calisma zamaninda hata verecektir.
        // java.lang.NullPointerException

    }
}

```

String ya da Wrapper tipteki elemanlar ile calistigimiz durumda isin icine equals ve hashCode metotlari girmektedir. Aciklamalari kod içerisinde bulabilirsiniz. Ornegi anlayamiyorsaniz oncesinde equals ve hashCode'u tekrar etmeye fayda var. Unutmayalim Map içerisinde elemanın bulunması 2 asamalidir

- once ilgili bucket bulunur(hashCode'a bakılır)
- sonrasında equals a bakılır.

```

import java.util.HashMap;
import java.util.Map;

```

```

class Animal {

}

```

```

class Dog {
    String name;
}

```

```

int age;

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Dog other = (Dog) obj;
    if (age != other.age)
        return false;
    if (name == null) {
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name))
        return false;
    return true;
}

public Dog(String name, int age) {
    super();
    this.name = name;
    this.age = age;
}

class Car {
    public Car(String hp, String brand, Integer price) {
        super();
        this.hp = hp;
        this.brand = brand;
        this.price = price;
    }
}

```

```

    }

    String hp;
    String brand;
    Integer price;

    @Override
    public int hashCode() {
        return brand.length();
    }

    // hashCode olarak bu metot kullanildiginda brand degisirse ilgili elemana
    // ulasamayiz.

    // @Override
    // public int hashCode() {
    // return 10;
    // }
    // hashCode olarak bu metot kullanildiginda brand degisse de ilgili elemana
    // ulasilabilir.Cunku hashCode degeri degismemekte.

    // Map mantiginda elemanların bulunmasi icin once hashCode degerine bakilir.
    // Sonrasinda equals a bakilir.

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Car other = (Car) obj;
        if (brand == null) {
            if (other.brand != null)

```

```

        return false;

    } else if (!brand.equals(other.brand))
        return false;

    if (hp == null) {
        if (other.hp != null)
            return false;

    } else if (!hp.equals(other.hp))
        return false;

    if (price == null) {
        if (other.price != null)
            return false;

    } else if (!price.equals(other.price))
        return false;

    return true;
}

}

public class HashMapReferenceType {

    public static void main(String[] args) {

        Animal animal = new Animal();
        Animal animal2 = new Animal();

        Map animalMap = new HashMap();
        animalMap.put(animal, "animal value");

        System.out.println("animal : " + animalMap.get(animal2));
        // animal2.hashCode() == animal.hashCode() false oldugu icin ve
        // animal2.equals(animal) false oldugu icin null deger donecektir.

        Dog dog = new Dog("dog1", 5);
        Dog dog2 = new Dog("dog1", 5);
    }
}

```

```

Map dogMap = new HashMap();
dogMap.put(dog, "dog value1");

// dog.equals(dog2); true
// dog.hashCode() == dog2.hashCode() false oldugu durumda elemani
// bulamaz
System.out.println("dog :" + dogMap.get(dog2));

Car car = new Car("50", "ford", 100);

Map hashMap = new HashMap();
hashMap.put(car, "value1");

System.out.println(hashMap.get(car));

car.brand = "bmw";
System.out.println(hashMap.get(car));

}

}

```

## Pure Java - 74 Generics & Collections - Map - 02

[Levent Erguder](#) 23 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde Map konusuna devam edecegiz. **java.util.TreeMap** ve **Backed Collection** konusunu inceleyecegiz. Hatirlayacagimiz gibi **java.util.TreeMap** **sorted** ve **ordered** ozellige sahiptir. **java.util.Treemap** sinifi **java.util.Navigable** arabirimini uygulamaktadir.

```

public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
{...}

```

Ornek kod ile baslayalim , natural order'a gore key'e gore siralama yapacaktir. String key'ekledikten sonra , Integer tipte bir key ekleyemeyiz. Bu durumda String ile Integer arasında siralama yapamayacaktir.llk hangi tip eklersek o tipte key'leri eklemeye devam etmeliyiz.

```
import java.util.Map;
import java.util.TreeMap;

public class TreeMapTest {

    public static void main(String[] args) {
        Map treeMap = new TreeMap();

        treeMap.put("key1", "value1");
        treeMap.put("key5", "value5");
        treeMap.put("key2", "value2");
        treeMap.put("key3", "value3");

        // treeMap.put(1, "java.lang.ClassCastException");

        for (Object key : treeMap.keySet()) {
            System.out.println(key + " " + treeMap.get(key));
        }

    }
}
```

Hatirlayacagimiz gibi Wrapper siniflar ve String sinifi Comparable arabirimini uygulamaktadir. Bu nedenle TreeMap'te kullanabiliyoruz. Eger kendi olusturdugumuz bir sinifi TreeMap'te key olarak kullanmak istersek bu durumda Comparable arabirimini uygulamalidir. Aksi takdirde *java.lang.ClassCastException* hatasi verecektir.

```
import java.util.TreeMap;

class Animal {
```

```

class Car implements Comparable {

    public Car(String hp, String brand, Integer price) {
        super();
        this.hp = hp;
        this.brand = brand;
        this.price = price;
    }

    String hp;

    @Override
    public String toString() {
        return "Car [hp=" + hp + ", brand=" + brand + ", price=" + price + "]";
    }

    String brand;
    Integer price;

    @Override
    public int compareTo(Car car) {
        return price.compareTo(car.price);
    }
}

public class TreeMapCarTest {

    public static void main(String[] args) {
        Animal animal = new Animal();
        TreeMap<Animal, String> animalTreeMap = new TreeMap<Animal, String>();
        // animalTreeMap.put(animal, "java.lang.ClassCastException");
        // key olarak Comparable olan siniflari kullanabiliriz. Yani bu durumda
        // Animal sinifi Comparable olmali aksi durumda calisma zamaninda hata
        // verecektir.
    }
}

```

```

Car ford = new Car("50", "ford", 100);
Car ford2 = new Car("150", "ford", 500);
Car ford3 = new Car("250", "ford", 1000);

TreeMap<Car, String> carTreeMap = new TreeMap<Car, String>();

carTreeMap.put(ford, "cheap car");
carTreeMap.put(ford3, "fast car");
carTreeMap.put(ford2, "nice car");

for (Car key : carTreeMap.keySet()) {
    System.out.println(key + " --> " + carTreeMap.get(key));
}

}

```

TreeMap sinifi ceilingEntry , ceilingKey , higherEntry , higherKey , floorEntry , floorKey , lowerEntry , lowerKey metodlarini inceleyelim.

```

import java.util.Map.Entry;
import java.util.TreeMap;

public class TreeMapNavigateTest {

    public static void main(String[] args) {
        TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();

        treeMap.put(1, "one");
        treeMap.put(2, "two");

        treeMap.put(30, "thirty");
        treeMap.put(40, "forty");
        treeMap.put(50, "fifty");

        treeMap.put(70, "seventy");
    }
}

```

```

treeMap.put(100, "hundred");

// ceilingEntry(key) minimum >=key olan Entry dondurur.

// ##### ceilingEntry

Entry<Integer, String> ceilingEntry = treeMap.ceilingEntry(40);

Integer ceilingEntryKey = ceilingEntry.getKey();

String ceilingEntryValue = ceilingEntry.getValue();

// ceilingKey minimum>=key olan key degerini dondurur.

// #####ceilingKey

Integer ceilingKey = treeMap.ceilingKey(40);

// ceiling metotlari icin >= soz konusudur. 40 key degerine sahip bir

// eleman oldugu icin ilgili key/value alacaktir.

System.out.println("ceilingEntryKey: " + ceilingEntryKey);

System.out.println("ceilingEntryValue: " + ceilingEntryValue);

System.out.println("ceilingKey: " + ceilingKey);

// higherEntry(key) minimum>key olan Entry dondurur.

// ##### higherEntry

Entry<Integer, String> higherEntry = treeMap.higherEntry(40);

Integer higherEntryKey = higherEntry.getKey();

String higherEntryValue = higherEntry.getValue();

// higherKey minimum>key olan key degerini dondurur.

// Burada buyuk esit yok!

// ### higherKey

Integer higherKey = treeMap.higherKey(40);

// higher metotlari > soz konusudur. 40 degerini almaz 40 degerinden

// buyuk olan ilk 50 oldugu icin bu elemani alir.

System.out.println("higherEntryKey: " + higherEntryKey);

System.out.println("higherEntryKey: " + higherEntryValue);

System.out.println("higherKey: " + higherKey);

```

```

// floorEntry(key) maximum<=key olan Entry dondurur.

Entry<Integer, String> floorEntry = treeMap.floorEntry(40);

Integer floorEntryKey = floorEntry.getKey();

String floorEntryValue = floorEntry.getValue();

// floorKey maximum<=key olan key degerini dondurur.

Integer floorKey = treeMap.floorKey(40);

System.out.println("floorEntryKey: " + floorEntryKey);

System.out.println("floorEntryValue: " + floorEntryValue);

System.out.println("floorKey: " + floorKey);

// lowerEntry(key) maximum lowerEntry = treeMap.lowerEntry(40);

Integer lowerEntryKey = lowerEntry.getKey();

String lowerEntryValue = lowerEntry.getValue();

Integer lowerKey = treeMap.lowerKey(40);

System.out.println("lowerEntryKey: " + lowerEntryKey);

System.out.println("lowerEntryValue: " + lowerEntryValue);

System.out.println("lowerKey: " + lowerKey);

}

}

```

Simdi de TreeMap'te bulunan diger metotlari inceleyelim; **firstEntry** , **lastEntry** , **pollFirstEntry** , **pollLastEntry** , **descendingMap** , **descendingKeySet**

```

import java.util.NavigableMap;

import java.util.NavigableSet;

import java.util.TreeMap;

import java.util.Map.Entry;

public class TreeMapMoreNavigateTest {

    public static void main(String[] args) {
        TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();

```

```

treeMap.put(1, "one");
treeMap.put(2, "two");

treeMap.put(30, "thirty");
treeMap.put(40, "forty");
treeMap.put(50, "fifty");

treeMap.put(70, "seventy");
treeMap.put(100, "hundred");

Entry<Integer, String> firstEntry = treeMap.firstEntry();
Integer firstKey = treeMap.firstKey();

Entry<Integer, String> lastEntry = treeMap.lastEntry();
Integer lastKey = treeMap.lastKey();

System.out.println("firstKey:" + firstKey);
System.out.println("lastKey:" + lastKey);

// pollFirstEntry , ilk elemani getirir ve siler.
Entry<Integer, String> pollFirstEntry = treeMap.pollFirstEntry();
// pollLastEntry , son elemani getirir ve siler.
Entry<Integer, String> pollLastEntry = treeMap.pollLastEntry();

firstKey = treeMap.firstKey();
lastKey = treeMap.lastKey();

System.out.println("after poll firstKey:" + firstKey);
System.out.println("after poll lastKey:" + lastKey);

for (Integer key : treeMap.keySet()) {
    System.out.println("key : " + key + " value : " + treeMap.get(key));
}

```

```

//descendingMap geriye NavigableMap dondurur.NavigableMap bir arabirimdir.

//SortedMap arabirimini kalitir , SortedMap de Map arabirimini kalitir.

// Bu metot siiralamayı tersten yapar ve geriye DescendingSubMap tipinde obje dondurur.

NavigableMap<Integer, String> descendingMap = treeMap.descendingMap();

System.out.println("descendingMap");

for (Integer key : descendingMap.keySet()) {

    System.out.println("key : " + key + " value : " + treeMap.get(key));

}

NavigableSet navigableDescendingKeySet = treeMap.descendingKeySet();

System.out.println("navigable descending keyset");

for(Integer key: navigableDescendingKeySet){

    System.out.println(key);

}

}

```

java.util.TreeMap icin kullandigimiz bu metodlari benzer sekilde java.util.TreeSet sinifi icin de kullanabiliriz.

```

import java.util.Iterator;

import java.util.NavigableSet;

import java.util.TreeSet;

public class TreeSetTest {

    public static void main(String[] args) {

        TreeSet treeSet = new TreeSet();

        treeSet.add(10);

        treeSet.add(20);

        treeSet.add(30);

        treeSet.add(40);
    }
}

```

```
Integer first = treeSet.first();
Integer last = treeSet.last();

Integer ceiling = treeSet.ceiling(30);
Integer higher = treeSet.higher(30);

Integer floor = treeSet.floor(30);
Integer lower = treeSet.lower(30);

NavigableSet navigableDescendingSet = treeSet.descendingSet();
Iterator descendingIterator = treeSet.descendingIterator();

System.out.println("first:" + first);
System.out.println("last:" + last);

System.out.println("ceiling:" + ceiling);
System.out.println("higher:" + higher);

System.out.println("floor:" + floor);
System.out.println("lower:" + lower);

System.out.println("treeset");
for (Integer i : treeSet) {
    System.out.println(i);
}

System.out.println("navigable descendingset");
for (Integer i : navigableDescendingSet) {
    System.out.println(i);
}
System.out.println("descending iterator");
while (descendingIterator.hasNext()) {
    System.out.println(descendingIterator.next());
}
```

```
    }  
}
```

Son olarak backed collection yapisini inceleyelim. **headMap** , **tailMap** ,**subMap** gibi metodlari inceleyelim.

```
import java.util.NavigableMap;  
import java.util.SortedMap;  
import java.util.TreeMap;  
  
public class BackedMap {  
  
    public static void main(String[] args) {  
        TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();  
  
        treeMap.put(1, "one");  
        treeMap.put(2, "two");  
  
        treeMap.put(30, "thirty");  
        treeMap.put(40, "forty");  
        treeMap.put(50, "fifty");  
  
        treeMap.put(70, "seventy");  
        treeMap.put(100, "hundred");  
  
        SortedMap<Integer, String> headMapExclusive = treeMap.headMap(30);  
        // headMap(30) varsayılan olarak exclusive çalışır.  
  
        /*  
         * public SortedMap<K,V> headMap(K toKey) {  
         * return headMap(toKey, false);  
         * }  
         */  
  
        // headMap exclusive  
        System.out.println("headMap exclusive");  
    }  
}
```

```

        for (Integer key : headMapExclusive.keySet()) {
            System.out.println(key + " " + headMapExclusive.get(key));
        }

        System.out.println("headMap inclusive");
        // boolean arguman alan overloaded metodlar NavigableMap donmektedir.
        // public interface NavigableMap<K,V> extends SortedMap<K,V> { }
        // NavigableMap yerine SortedMap de kullanabiliriz.
        // NavigableMap IS-A SortedMap

        NavigableMap<Integer, String> headMapInclusive = treeMap.headMap(30, true);
        for (Integer key : headMapInclusive.keySet()) {
            System.out.println(key + " " + headMapInclusive.get(key));
        }

        /*
         * public NavigableMap<K,V> headMap(K toKey, boolean inclusive) {
         * return new AscendingSubMap(this,
         * true, null, true,
         * false, toKey, inclusive);
         * }
         */
    }

    System.out.println();

    SortedMap<Integer, String> tailMapInclusive = treeMap.tailMap(30);
    // tailMap(30) varsayılan olarak inclusive çalışır.

    /*
     * public SortedMap<K,V> tailMap(K fromKey) {
     * return tailMap(fromKey, true);
     * }
     */
}

System.out.println("tailMapInclusive ");
for (Integer key : tailMapInclusive.keySet()) {

```

```

        System.out.println(key + " " + tailMapInclusive.get(key));
    }

    System.out.println("tailMap exclusive ");
    NavigableMap<Integer, String> tailMapExclusive = treeMap.tailMap(30, false);
    for (Integer key : tailMapExclusive.keySet()) {
        System.out.println(key + " " + tailMapExclusive.get(key));
    }

    System.out.println();

    System.out.println("subMap");
    SortedMap<Integer, String> subMap = treeMap.subMap(30, 70);
    // varsayılan olarak [dahil-haric) şeklindedir. inclusive-exclusive
    for (Integer key : subMap.keySet()) {
        System.out.println(key + " " + subMap.get(key));
    }

    /*
     * public SortedMap<K,V> subMap(K fromKey, K toKey) {
     * return subMap(fromKey, true, toKey, false);
     * }
     */

    System.out.println("subMap2");
    NavigableMap<Integer, String> subMap2 = treeMap.subMap(30, true, 70, true);

    for (Integer key : subMap2.keySet()) {
        System.out.println(key + " " + subMap2.get(key));
    }
}

```

Benzer şekilde java.util.TreeSet için **headSet** , **tailSet** ve **subSet** metodları kullanılabilir.

```
import java.util.NavigableSet;
```

```

import java.util.SortedSet;
import java.util.TreeSet;

public class BackedSet {

    public static void main(String[] args) {
        TreeSet treeSet = new TreeSet();

        treeSet.add(1);
        treeSet.add(2);

        treeSet.add(30);
        treeSet.add(40);
        treeSet.add(50);

        treeSet.add(70);
        treeSet.add(100);

        SortedSet headSetExclusive = treeSet.headSet(30);
        /*
         * public SortedSet headSet(E toElement) {
         * return headSet(toElement, false);
         * }
         */
        System.out.println("headSet exclusive");
        for (Integer i : headSetExclusive) {
            System.out.println(i);
        }

        NavigableSet headSetInclusive = treeSet.headSet(30, true);

        System.out.println("headSet Inclusive");
        for (Integer i : headSetInclusive) {
            System.out.println(i);
        }
    }
}

```

```

    }

SortedSet tailSetInclusive = treeSet.tailSet(30);
/*
 * public SortedSet tailSet(E fromElement) {
 * return tailSet(fromElement, true);
 * }
 */

System.out.println("tailSet Inclusive");
for (Integer i : tailSetInclusive) {
    System.out.println(i);
}

NavigableSet tailSetExclusive = treeSet.tailSet(30, false);
System.out.println("tailSe tExclusive");
for (Integer i : tailSetExclusive) {
    System.out.println(i);
}

SortedSet subSet = treeSet.subSet(30, 70);
/*
 * public SortedSet subSet(E fromElement, E toElement) {
 * return subSet(fromElement, true, toElement, false);
 * }
 */

System.out.println("subSet inclusive-exlusive");
for (Integer i : subSet) {
    System.out.println(i);
}

NavigableSet subSet2 = treeSet.subSet(30, true, 70, true);
System.out.println("subset2 inclusive-inclusive");
for (Integer i : subSet2) {
    System.out.println(i);
}

```

```
    }  
}  
}
```

Backed Collection'a eleman eklerken ya da cikartirken dikkat edilmesi gereken bazi noktalar yer almaktadir.

- Orijinal TreeMap/TreeSet e eleman ekledigimizde/sildigimizde eger backed collection'in sinirlari icerisindeyse backed collection a da ekler/siler.
- Backed Collection a eleman ekledigimiz/sildigimiz elemanlar orijinal TreeMap/TreeSet ten de silinir/eklenir.

```
package treemap;  
  
import java.util.NavigableMap;  
import java.util.TreeMap;  
  
public class AddBackedMap {  
  
    public static void main(String[] args) {  
        TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();  
  
        treeMap.put(1, "one");  
        treeMap.put(2, "two");  
  
        treeMap.put(30, "thirty");  
        treeMap.put(40, "forty");  
        treeMap.put(50, "fifty");  
  
        treeMap.put(70, "seventy");  
        treeMap.put(100, "hundred");  
  
        NavigableMap<Integer, String> subMap = treeMap.subMap(30, true, 70, true);  
  
        System.out.println("subMap");  
        for (Integer key : subMap.keySet()) {  
            System.out.println(key + " " + subMap.get(key));  
        }  
    }  
}
```

```

    }

    treeMap.put(60, "sixty");
    // treeMap e 60 degerini ekledik bu deger subMap in sinirlari icinde
    // ayni zamanda subMap'e de ekler.

    treeMap.put(1000, "thousand");
    // treeMap e eklenen 1000 degeri subMap e eklenmez. subMap in sinirlari
    // disindadir.

    subMap.put(65, "sixty five");
    // subMap'e ekledigimiz bu deger treeMap e de eklenir.

    // subMap.put(80, "exception");
    // subMap e sinir disinda key ekleyemeyiz. [30,70] arasında eklenebilir.
    // java.lang.IllegalArgumentException: key out of range

    System.out.println("after subMap");
    for (Integer key : subMap.keySet()) {
        System.out.println(key + " " + subMap.get(key));
    }

    System.out.println("after treeMap");
    for (Integer key : treeMap.keySet()) {
        System.out.println(key + " " + treeMap.get(key));
    }

}

```

## Pure Java - 75 Generics & Collections - PriorityQueue

[Levent Erguder](#) 24 November 2014 [Java SE](#)

Merhaba Arkadaslar ,

Bu yazimda `java.util.PriorityQueue` sinifindan bahsedecegim. **PriorityQueue** , `AbstractQueue` i kalitmaktadir.

AbstractQueue ise **Queue** arabirimini uygulamaktadir/implements.Queue arabirimini de Collection arabirimini kalitmaktadir.

```
public class PriorityQueue extends AbstractQueue  
    implements java.io.Serializable { ... }
```

```
public abstract class AbstractQueue  
    extends AbstractCollection  
    implements Queue { ... }
```

```
public interface Queue extends Collection { .. }
```

LinkedList IS-A Queue ve PriorityQueue IS-A Queue onermeleri dogrudur.

```
Queue queueA = new LinkedList();  
Queue queueB = new PriorityQueue();
```

**PriorityQueue** ile ilgili orneklerimize baslayalim;

```
import java.util.PriorityQueue;  
import java.util.Queue;  
  
public class PriorityQueueTest {  
  
    public static void main(String[] args) {  
        Queue pq = new PriorityQueue();  
        //pq.add(null);  
        //PriorityQueue e null eleman eklenemez.  
        pq.add(10);  
        pq.add("10");  
        // java.lang.ClassCastException  
        // java.lang.Integer cannot be cast to java.lang.String  
  
    }  
}
```

PriorityQueue'e null eleman eklenemez , calisma zamaninda NullPointerException hatasi verecektir.Non-Generics bir PriorityQueue yapısında ilk eleman Integer'sa sonrasında farkli tipte bir eleman eklenemez elemanların hepsi Integer tipinde olmalıdır. Aksi durumda ClassCastException hatasi verecektir.

PriorityQueue'e **add** veya **offer** metodu ile eleman eklenebilir. add metodu **offer** metodunu çağırmaktadır.**remove** metodu ile ilgili elaman silinebilir. **peek** metodu PriorityQueue'teki ilk elemani dondurur , element metodu **peek** metodunu çağırır.**poll** metodu PriorityQueue'teki ilk elemani dondurur fakat bu elemani siler , **remove** metodu poll metodunu çağırır.

```
import java.util.PriorityQueue;

public class PriorityQueueTest2 {

    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>();

        pq.add(1);
        pq.add(100);
        pq.add(10);
        pq.add(10);
        pq.add(50);
        pq.add(12);
        pq.add(150);

        /*
         * add metodu offer metodunu çağırmaktadır.
         * public boolean add(E e) {
         *     return offer(e);
         * }
         */
        pq.offer(-20);
        pq.offer(-10);

        pq.remove(12);

        Integer firstElement = pq.peek();
        // peek metodu PriorityQueue'teki ilk elemani dondurur fakat bu elemani
        // silmez.
    }
}
```

```

System.out.println(firstElement);

Integer firstElementPoll = pq.poll();
// poll metodu PriorityQueue'teki ilk elemani dondurur fakat bu elemani
// siler.

System.out.println(firstElementPoll);

Integer secondElementRemove = pq.remove();
// remove metodu poll metodunu cagirir.

System.out.println(secondElementRemove);

firstElement = pq.peek();
firstElement = pq.peek();
firstElement = pq.peek();
firstElement = pq.element();
/*
 * element metodu peek metodunu cagirir.
 *
 * public E element() {
 *     E x = peek();
 *     if (x != null)
 *         return x;
 *     else
 *         throw new NoSuchElementException();
 * }
 */

// -20 silindi icin peek metodu -10 donecektir ve -10 degerini silmez.
// bu metodu defalarca cagirabiliriz sonuc degismez.

System.out.println(firstElement);

System.out.println(pq);
// Arka planda Iterator calisir ve Ordered olarak yazma garantisi
// yoktur.

```

```

        int size = pq.size();

        // poll metodu elemani sildigi icin for dongusu disinda size metodu
        // cagrilmalidir.

        for (int i = 0; i < size; i++) {
            System.out.print(pq.poll() + " ");
        }

        System.out.println();
        System.out.println(pq.peek());
        // artik elaman kalmadigi icin null donecektir.

    }
}

```

## Pure Java – 76 Generics & Collections – Generics – 01

[Levent Erguder](#) 29 November 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde **Generics** konusundan bahsedecegim. Onceki bolumlerde **Generics** yapisini kullandik fakat burada isin biraz daha detayina girmeye calisacagiz.

Generics yapisi, type kavramini class ve interfaceler icin aktif eder. Class, interface veya method taniminda(define/declare) parametre ile bu tip bilgisini kullanabilmemizi saglar. Peki bunun en onemli faydası nedir ? type parametreleri kodunuza tekrar kullanilabilirlik ozelligi kazandirir. Bu kismi ilerleyen yuzelarda inceleyecegiz.

Generics yapisinin non-generics yapisina karsi su avantajlari vardir;

- Compile time/derleme zamaninda guclu bir type check/tip kontrolu.
  - Cast etme islemini ortadan kaldirmasi
- Ornegin non-generics olarak tanimli bir AraryList'ten elemani get ile aldigimizda geriye Object doner. Non-generics Collection'larda elemanlar java.lang.Object tipindedir.

```

List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);

List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0); // no cast

```

- Generic/genelleyici bir implementation

Generics yapisini kullanarak farkli type degerine sahip Collection'lar icin kullanilabilecek, customized ozellige sahip , type safe ve kolay okunabilir bir kod yazilabilir.

### Mix Generics & Non-Generics Collection

Java'da array/diziler her zaman type safe'tir. String olarak tanimlanmis bir dizi bir baska tipte eleman eklenemez. Java 5 ten once type safe Collection tanimlanması icin bir yapı mevcut degildi. Java 5 ten sonra Collection yapıları icin **Generics** yapısı eklendi.

Bir non-generic collection herhangi bir tipte elaman tutabilir. Unutmayalim bir collection'in elemanları primitive olamaz her zaman objedir.

Non-generics ve generics tanimli oldugu durumu hatırlamak icin bir ornek yapalim ;

```
import java.util.ArrayList;
import java.util.List;

class Dog {

}

public class Test {

    public static void main(String[] args) {
        List myList = new ArrayList();

        myList.add("test");
        myList.add(10);
        myList.add(new Dog());

        String str = (String) myList.get(0);
        // non-generics bir collection yapısında elemani aldiktan sonra cast
        // etmemiz gereklidir.

        List <String> myGenericsList = new ArrayList <String>();
        myGenericsList.add("test");
        // myGenericsList.add(10);
        // olaraka generics yapida tanımladığımız için farklı tipte bir
        // eleman ekleyemeyiz.
```

```
    }  
}  
}
```

Non-generics ve generics yapisini beraber kullandigimizda risk soz konusudur. Generics yapida tanimli bir List'i non-generics tanimli bir parametre alan metoda arguman olarak verebiliriz. Eger ilgili metotta farkli tipte bir eleman eklersek bu durumda ekledigimiz noktada hata almayiz. Sonrasinda elemanlara ulasirken hata aliriz.

```
import java.util.ArrayList;  
import java.util.List;  
  
public class TestLegacy {  
  
    public static void main(String[] args) {  
        List myList = new ArrayList();  
        myList.add(10);  
        myList.add(20);  
  
        addSomeElements(myList);  
  
        // myList generics yapisinda tanimlandigi icin get metodu Integer  
        // tipinde doner.  
        // Fakat bu eleman "danger" olarak diger metotta eklendi. 3.indexteki  
        // elemana ulasirsak exception aliriz ;  
        // java.lang.ClassCastException:  
        Integer exception = myList.get(3);  
  
        // for (Integer i : myList) {  
        // System.out.println(i);  
        // // 3.indexe geldiginde calisma zamaninda hata olacaktir.  
        // // java.lang.ClassCastException:  
        // }  
    }  
}
```

```

// Generics tanimli bir List i non-generics tanimli bir parametre alan
// metoda arguman olarak verebiliriz.

public static void addSomeElements(List nonGenericList) {
    nonGenericList.add(30);

    nonGenericList.add("danger!");
    // List'e yeni bir eleman ekledik. Bu noktada calisma zamani hatasi
    // vermeyecektir!
}

}

```

Burada aklimize su soru gelebilir , neden generics yapısında bir List'e farklı bir tipte eleman eklenebiliyor ve eklentiği noktada problem çıkarmıyor. Bunun nedeni type bilgisi çalışma zamanında yoktur. Bunun anlamı tüm generics tanımlamalar çalışma zamanında compiler tarafından silinir.

```

List myList = new ArrayList();
// List myList = new ArrayList();
//calisma zamaninda generics tanimlari silinir bu sekle donusturulur.

```

Generics yapısı compile-time koruma sağlar. Bu sayede derleme zamanında generics tanımından farklı bir tipte eleman eklemenize izin vermez.

Dizileri/array düşündüğümüzde , diziler hem derleme zamanında/compile time hem de çalışma zamanında /runtime koruma sağlarlar. Peki neden generics yapısı da bu şekilde hem çalışma zamanında/runtime hem derleme zamanında/compile time koruma sağlamaz. Bunun nedeni legacy/eski kodlara destek sağlamaktır.

Bununla birlikte eğer non-generics ve generics tanımlarını birbiriyile mix etmezsek runtime için korumaya gerek yoktur, generics ifadesinin bizim için compile time da koruma sağlaması yeterli olacaktır.

### **Polymorphism & Generics**

Onceki bolumlerde polymorphism konusunu incelemistik. Hatırlayacağımız gibi IS-A kuralına uygun şekilde bir referans değişken kendi tipinde ya da subclass/altsınıf tipinde bir objeye referansta bulunabilir. Peki generics ifadeleri için de bu şekilde kullanılabilir mi ?

```

import java.util.ArrayList;
import java.util.List;

```

```

class Animal {

}

class Dog extends Animal {

}

public class PolymorphismTest {

    public static void main(String[] args) {
        List animalList = new ArrayList();
        List dogList = new ArrayList();
        // ArrayList IS-A List bu nedenle List tipinde bir referans degisken
        // ArrayList tipinde bir objeyi referansta bulunabilir.

        // List <Animal> animalDogList = new ArrayList <Dog>();
        // derleme hatasi verecektir.

        // Referans degiskenenin taniminda kullandigimiz generics tanimi ile ,
        // obje olustururken kullandigimiz generics ifadesi ayni olmalıdır.
        // Subclass dahi olacak sekilde farkli olamaz.
    }
}

```

Collection taniminda, referans degiskenenin taniminda kullandigimiz generics ifadesi ile new anahtar kelimesi ile kullandigimiz generics ifadesi ayni olmalıdır. Buraya polymorphismde oldugu gibi subclass tipi bile yazamayiz!

Peki array tanimlarinda durum nasil olmaktadır ?

```

class Animal {

}

class Dog extends Animal {

```

```

}

public class PolymorphismTestArray {

    public static void main(String[] args) {
        Animal[] animalArray = new Animal[10];
        Dog[] dogArray = new Dog[10];
        Animal[] animalDogArray = new Dog[10];
    }
}

```

Array/diziler icin bu durum problem cikarmayacaktir. IS-A kuralina uygun sekilde dizi tanimlamasi yapisilabilir.

```

List <Animal> animalDogList = new ArrayList <Dog>();
//derleme hatasi verir
Animal[] animalDogArray = new Dog[10];
//derleme hatasi vermez!

```

## Pure Java - 77 Generics & Collections - Generics - 02

[Levent Erguder](#) 30 November 2014 [Java SE](#)

Merhaba Arkadaslar

Bu bolumde **Generics** konusuna devam edecegiz.

### Generics Methods

Polymorphism kavraminin bize sagladigi en onemli faydalardan birtanesi, bir metot parametresi calisma zamaninda/runtime kendi tipinde ya da subtype tipinde bir objeye referansta bulunabilir. Polymorphism ve override konusunu kisaca hatirlayalim

```

class Animal {
    public void eat() {
        System.out.println("Animal eat");
    }
}

class Dog extends Animal {

```

```

public void eat() {
    System.out.println("Dog eat");
}

}

class Cat extends Animal {
    public void eat() {
        System.out.println("Cat eat");
    }
}

public class Test {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Animal animal = new Animal();
        Animal animalCat = new Cat();

        polymorphismTest(dog);
        polymorphismTest(animal);
        polymorphismTest(animalCat);

    }

    public static void polymorphismTest(Animal a) {
        a.eat();
    }
}

```

polymorphismTest metodu parametre olarak Animal tipinde bir degisen almaktadir. Hatirlayacagimiz gibi override metotlarda objenin tipi onemlidir. Objenin tipine gore ilgili sinifa ait metot calisacaktir.

Bu durum isin icine Collection girdiginde nasil olacaktir peki ?

```
import java.util.ArrayList;
```

```

import java.util.List;

class Animal {
    public void eat() {
        System.out.println("Animal eat");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("Dog eat");
    }
}

public class CollectionGenericsMethodTest {

    public static void main(String[] args) {
        List<Animal> animalList = new ArrayList<Animal>();
        List<Dog> dogList = new ArrayList<Dog>();

        // List animalDogList = new ArrayList();
        // Derleme hatasi verir! Referans degisen olarak tanimliysa ,
        // new anahtar kelimesinden sonra olmalidir.

        genericsRules(animalList);
        // genericsRules(dogList);
        // dogList'i arguman olarak veremeyiz. Ilgili metot List tipinde
        // bir parametre almaktadir.
        // Dog IS-A Animal olsa da buraya List tipine bir degisen
        // gonderemeyiz !
    }

    public static void genericsRules(List<Animal> myAnimals) {
        System.out.println("test");
    }
}

```

```
}
```

Peki benzer durumu array/dizi ile yaparsak ne olur ?

```
import java.util.ArrayList;
import java.util.List;

class Animal {
    public void eat() {
        System.out.println("Animal eat");
    }
}

class Dog extends Animal {
    public void eat() {
        System.out.println("Dog eat");
    }
}

public class ArrayTestMethod {

    public static void main(String[] args) {
        Animal[] animalArray = new Animal[5];
        Dog[] dogArray = new Dog[5];

        Animal[] animalDogArray = new Dog[5];

        arrayRules(animalArray);
        arrayRules(dogArray);
        arrayRules(animalDogArray);
    }

    public static void arrayRules(Animal[] animal) {
        System.out.println("test");
    }
}
```

```
}
```

Burada kodumuz sorunsuz calisacaktir. Animal[] tipinde parametre alan bir metoda Dog[] tipinde bir referans degisken gonderebiliriz. (Dog IS-A Animal)

Peki neden array icin calisan bu durum collection icin calismamaktadir;

```
import java.util.List;

class Animal {

}

class Dog extends Animal {

}

class Cat extends Animal {

}

public class Test {

    public static void main(String[] args) {
        Cat[] catArray = { new Cat(), new Cat() };

        addAnimal(catArray);
        // Cat[] tipindeki referans degiskenimiz Animal[] tipinde parametre alan
        // metoda gonderebiliriz.

    }

    public static void addAnimal(Animal[] animalArray) {
        animalArray[0] = new Dog();
        //bu metoda arguman olarak Cat[] tipinde bir arguman gonderdik.
        // java.lang.ArrayStoreException: hatasi verecektir!
    }
}
```

Eger metoda Animal sinifin alt sinif tipinde bir dizi argumani gecersek sorunsuzca caliscaktir , fakat Cat[] tipindeki bir diziye/array'e Dog tipinde bir obje ekledigimizde calisma zamaninda hata verecektir.

Bu senaryo onune gecmek istedigimiz bir durumdur. Iste bu nedenle List<Dog> tipindeki bir degiskeni List<Animal> tipinde bir parametre alan bir metoda gonderemeyiz.

```
import java.util.ArrayList;
import java.util.List;

class Animal {

}

class Dog extends Animal {

}

class Cat extends Animal {

}

public class Test {

    public static void main(String[] args) {
        Cat[] catArray = { new Cat(), new Cat() };

        addAnimal(catArray);
        // Cat[] tipindeki referans degiskenimiz Animal[] tipinde parametre alan
        // metoda gonderebiliriz.

        List animalList = new ArrayList();
        addAnimal(animalList);
    }

    public static void addAnimal(Animal[] animalArray) {
        animalArray[0] = new Dog();
        // bu metoda arguman olarak Cat[] tipinde bir arguman gonderdik.
        // java.lang.ArrayStoreException: hatasi verecektir!
    }
}
```

```

// Bu metoda sadece List tipinde bir arguman gelebiliriz.

public static void addAnimal(List animalList) {

    animalList.add(new Dog());
    animalList.add(new Cat());

    // Burada subclass tipinde objeler ekleyebiliriz. Bu metot sadece
    // List tipinde parametre alabilir. Dolayisiyla burada
    // guvendeyiz. Calisma zamaninda problem olmayacaktir.

}

}

```

### Generics ? wildcard

Generics yapısında ? wildcard olarak kullanılmaktadır. `List<? extends Animal>` şeklinde kullanabiliriz.

```

import java.util.ArrayList;
import java.util.List;

class Animal {

}

class Dog extends Animal {

}

class Cat extends Animal {

}

public class GenericsWildcard {

    public static void main(String[] args) {
        List<Animal> animalList = new ArrayList<Animal>();

        List<Dog> dogList = new ArrayList<Dog>();

        addAnimal(animalList);
        addAnimal(dogList);
    }
}

```

```

    }

    public static void addAnimal(List<? extends Animal> animalList) {
        // <? extends Animal> olarak tanimli oldugu zaman yeni bir elaman
        // ekleyemeyiz. sadece null ekleyebiliriz.

        // animalList.add(new Animal());
        // animalList.add(new Dog());

        animalList.add(null);
    }
}

```

List tipinde parametre alan bir metoda sadece tipinde bir degiskeni arguman olarak verebiliriz, fakat **List<? extends Animal>** tipinde parametre alan bir metoda sinifin altsinifi tipinde de bir arguman verebiliriz fakat bu metot icerisinde yeni bir eleman ekleyemeyiz. Sadece null eleman ekleyebiliriz. ? wildcard ile extends anahtar kelimesini kullanabiliyoruz. Burada ? karakterini bir arabirimle kullansak da extends anahtar kelimesi ile kullanabiliyoruz ? implements seklinde bir generics tanimi yoktur.

```

import java.util.List;

interface Speedy {
}

public class WildcardInterface {

    public void wildcard(List<? extends Speedy> test) {

    }
}

```

Bir diger kullanim sekli olarak super anahtar kelimesi kullanilabilir. <? super Dog> ifadesi , Dog veya ust sinif tipini kabul eder , alt sinifi kabul etmez. <? super Dog> generics yapisi eleman eklemeye izin verir , Dog tipinde ya da alt sinif tipinde obje ekleyebiliriz fakat ust sinif tipinde bir eleman ekleyemeyiz.

```

import java.util.ArrayList;
import java.util.List;

class Animal {

}

class Dog extends Animal {

}

class Kangal extends Dog {

}

public class GenericsWildcardSuper {

    public static void main(String[] args) {

        List<Animal> animalList = new ArrayList<Animal>();
        List<Dog> dogList = new ArrayList<Dog>();
        List<Kangal> kangalList = new ArrayList<Kangal>();

        addAnimal(animalList);
        addAnimal(dogList);

        // addAnimal(kangalList);
        // derleme hatası verir.

    }

    // ? super Dog , buraya Dog ve super/ust sınıf tipinde arguman kabul eder.

    public static void addAnimal(List<? super Dog> dogList) {
        dogList.add(new Kangal());
        dogList.add(new Dog());
        //Burada Dog veya alt sınıf tipinde obje ekleyebiliriz.

        // dogList.add(new Animal());
    }
}

```

```
    }  
}
```

List ve List tipinde degiskeni metoda gectigimizde sorun teskil etmez. Ekledegimiz elemanlar calisma zamaninda exceptiona neden olmaz.

Sadece ? wildcard'i kullandigimizda buraya herhangi bir tip kabul edilebilir.

```
import java.util.ArrayList;  
import java.util.List;  
  
class Car {  
  
}  
  
public class GenericsWildCardTest {  
  
    public static void main(String[] args) {  
        List<Integer> integerList = new ArrayList <Integer>();  
        List<String> stringList = new ArrayList <String>();  
        List<Car> carList = new ArrayList <Car>();  
        List<Object> objectList = new ArrayList<Object>();  
  
        wildcard(integerList);  
        wildcard(stringList);  
        wildcard(carList);  
        wildcard(objectList);  
    }  
  
    // Sadece ? wildcard'i kullandigimizda buraya herhangi bir tipte tip kabul  
    // edilebilir anlamina gelmektedir.  
    public static void wildcard(List<?> myList) {  
        System.out.println("wildcard ?");  
        // myList.add("Compile Error");  
        // ? tek basina kullanildiginda burada eleman eklenemez.
```

```

    // sadece null eklenir

    myList.add(null);

}

}

```

Metot paramertesi **List<Object>** ise sadece **<Object>** tipinde arguman alabilir.

```

import java.util.ArrayList;
import java.util.List;

class Car {

}

public class GenericsObjectTest {

    public static void main(String[] args) {
        List<Integer> integerList = new ArrayList<Integer>();
        List<String> stringList = new ArrayList<String>();
        List<Car> carList = new ArrayList<Car>();
        List<Object> objectList = new ArrayList<Object>();

        // wildcard(integerList);
        // wildcard(stringList);
        // wildcard(carList);
        wildcard(objectList);
    }

    public static void wildcard(List <Object> myList) {
        System.out.println("Object");
        myList.add(10);
        myList.add("test");
        myList.add(new Car());
    }
}

```

```
}
```

<? extends Object> ile <?> aynidir.

## Pure Java – 78 Generics & Collections – Generics – 03

[Levent Erguder](#) 03 December 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde Generics konusuna devam edecegiz. Bu bolume kadar type-safe collection yapisini gorduk. Burada Generics konusunun basinda bahsettigimiz su ozelligi inceleyegiz;

Generics yapisi, type kavramini class ve interfaceler icin aktif eder. Class, interface veya method taniminda(define/declare) parametre ile bu tip bilgisini kullanabilmemizi saglar. Peki bunun en onemli faydası nedir ? type parametreleri kodunuza tekrar kullanilabilirlik ozelligi kazandirir.

Generics yapisini kullanarak farkli type degerine sahip Collection'lar icin kullanabilecek, customized ozellige sahip , type safe ve kolay okunabilir bir kod yazilabilir.

Aslinda cok uzaga gitmeye gerek yok , Collection API yapısında bir cok class ve interface bu sekilde yazilmistir, ornegin ;

```
public interface List<E> extends Collection<E> {...}  
public interface Set<E> extends Collection<E> {...}  
public interface Map<K,V> {...}  
public class TreeSet<E> extends AbstractSet  
    implements NavigableSet<E>, Cloneable, java.io.Serializable  
{...}
```

Class ve interface'lerde oldugu gibi yine bir cok metot Collection API'de bu sekildedir.

```
boolean add(E e);  
E remove(int index);  
V put(K key, V value);
```

Peki burada class , interface ya da metotlarda kullanılan E , K, V gibi harflerin anlamı nedir neden kullanılmaktadır ?

<E> ifadesi bizim için placeholder/yer tutucu görevi üstlenmektedir. List arabirimini bizim için generic/genel bir “template/taslak” özelliği taşıır.

Biz kod yazdığımızda dilersek List<Dog> , List<Car> , List<String> vb gibi bu List'in tipini değiştirebiliriz.

Burada isim olarak E harfi kullanılması zorunlu değildir fakat convention gereği E , “Element” anlamında kullanılmaktadır.

Diger kullanılan ya da bizim uygun yerlerde (conventiona uyacak şekilde) kullanmamız gereken harfler ve anımları söyledir ;

E – Element

K – Key

N – Number

T – Type

V – Value

Bu durumda ArrayList'teki eleman ekleme metoduna bakalım.Bu ne anlama geliyor olabilir ?

```
public boolean add(E e) {  
    ensureCapacity(size + 1); // Increments modCount!!  
    elementData[size++] = e;  
    return true;  
}
```

Bunun anlamı E yi hangi tipte tanımladıysak , o tipte eleman ekleme yapabiliriz. List<String> tipinde bir generics ifadeye Integer tipinde eleman ekleyemeyiz.

### Generic Class

Generic Class tanımı , non-generic class tanımından farklı olarak deklarasyonda type parametresi eklenerek yapılır.

Generic Class tanımına örnek Comparable arabirimini kısa ve güzel bir örnek olacaktır.

```
public interface Comparable<T> {  
    public int compareTo(T o);  
}
```

Daha iyi anlamak için kendimiz bir örnek yapalım;

```
class MyGenericClass <T>{
```

```

// T , type paremetresi

//instance degisen T tipi olabilir.

T genericReference;

//Metot donus tipi T olabilir.

public T getGenericReference() {

    return genericReference;
}

public MyGenericClass(T genericReference) {

    this.genericReference = genericReference;
}

}

public class GenericTest {

    public static void main(String[] args) {

        MyGenericClass <Integer> intOb = new MyGenericClass<Integer>(100);

        MyGenericClass <String> strOb = new MyGenericClass<String>("Test");

        MyGenericClass raw = new MyGenericClass(10.5);

        Integer intValue = intOb.getGenericReference();

        String strValue = strOb.getGenericReference();

        double doubleValue = (double) raw.getGenericReference();

        System.out.println("int :" + intValue);

        System.out.println("String :" + strValue);

        System.out.println("double :" + doubleValue);
    }
}

```

```
}
```

Generic Class tanimında birden fazla type parametresi kullanabiliriz.

```
class UseTwo<T, X> {

    T one;
    X two;

    public T getOne() {
        return one;
    }

    public X getTwo() {
        return two;
    }

    public UseTwo(T one, X two) {
        this.one = one;
        this.two = two;
    }
}

public class TwoTypeParameterTest {

    public static void main(String[] args) {
        UseTwo<String, Integer> twos = new UseTwo<String, Integer>("Java", 2014);

        String one = twos.getOne();
        int two = twos.getTwo();

    }
}
```

## Generics Method & Constructor

Süada kadar olan örneklerde parameter type bilgisini class tanımında yaptıktan fakat class tanımında yapmayıip metot tanımında da yapabiliriz.

```
public class GenericMethodTest {  
  
    public static<E> void genericMethod(E[] myArray) {  
        for (E element : myArray) {  
            System.out.println(element);  
        }  
    }  
  
    public static void main(String[] args) {  
        Integer[] intArray = { 1, 2, 3, 4, 5 };  
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };  
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };  
  
        System.out.println( "Array integerArray contains:" );  
        genericMethod( intArray ); // pass an Integer array  
  
        System.out.println( "\nArray doubleArray contains:" );  
        genericMethod( doubleArray ); // pass a Double array  
  
        System.out.println( "\nArray characterArray contains:" );  
        genericMethod( charArray ); // pass a Character array  
    }  
}
```

Donus tipimiz de T olabilir ;

```
public class Test {  
  
    public static <T> testMethod(T t) {  
  
        // instanceof kontrolü yapabiliriz.  
        if (t instanceof Integer) {  
            // Sonrasında cast edebiliriz.  
        }  
    }  
}
```

```

        Integer number = (Integer) t;
        number = number + 20;
        t = (T) number;
    }

    return t;
}

public static void main(String[] args) {
    System.out.println(testMethod(10));
}
}

```

Type parametresi sınıf tanımda yapılmışsa bu type parametresini non-static metodlarda sorunsuzca kullanabiliriz fakat static metodlarda bu type parametresini **kullanamayız**. Bu nedenle static metodlar için type parametresini tekrar tanımlamak gereklidir.

```

public class GenericStaticTest {

    public void nonStaticMethod(T t) {

    }

    //    public static void staticMethodWithoutType(T t) {
    //
    //    }

    public static void staticMethodWithType(T t) {

    }
}

```

#### Bounded Type Parameters

Onceki bolumlerde ? wildcard kullanımını incelemistik. Kendimiz bir type belirlediğimizde bunu sınırlamak isteyebiliriz. Ornegin sayısal işlemler yaptığımız bir generic metod düşünelim bunu Number sınıfı ve altsınıfları için sınırlayabiliriz.

```

public class BoundedTypeParameter {

    public static void boundedTypeMethod(T t) {
        System.out.println("T: " + t.getClass().getName());
    }

    public static void main(String[] args) {
        boundedTypeMethod(100);
        boundedTypeMethod(100.5);
        boundedTypeMethod(100.5f);

        // boundedTypeMethod("compile error");
    }
}

```

Upper Bounded wildcard kullanimi ile , unbounded wildcard kullanimini karistirmayalim.

```

public class NumberHolder<? extends Number> {}

//unbounded wildcard

public class NumberHolder<?> { ? aNum; }

//unbounded wildcard

//derleme hatasi

public class NumberHolder { T aNum; }

// Yes

```

Not: Generics konusu ciddi anlamda karisik ve kapsamlı bir konudur. Sinav icin bu kadar bilgi yeterli olacaktir.  
Ilterleyen donemlerde bu konu daha da detayli hale gelecektir.

# BÖLÜM-8

## Pure Java - 79 Nested Class - Inner Class

[Levent Erguder](#) 05 December 2014 [Java SE](#)

Merhaba Arkadaslar,

8.bolumde Java'da bulunan **Nested Class** konusunu incelecegiz.

Java dili bir sinif içerisinde bir baska sinif tanimlamana izin verir. Bu kavrama genel olarak Nested Class denilir.

Nested Class'lar 2 ye ayrilir ; **static** ve **non-static** . static tanimli nested class'lara **static nested class** denilir. non-static nested classlar ise **inner class** olarak isimlendirilir.

### 1. Non-Static Nested Class(Inner Class)

- Regular Inner Class
- Local Inner Class
- Anonymous Inner Class

### 2. Static Nested Class

Ilk olarak **Regular Inner Class** ile baslayalim, yazi boyunca Regular Inner Class'lara **Inner Class** diyecegiz.

```
public class OuterClass {  
  
    class InnerClass {  
  
    }  
  
    static class StaticNestedClass {  
  
    }  
}
```

Bir nested class , outer sinifin uyesidir(member) . Non-static nested class(inner class) , outer sinifin uyelerine(instance variable ve metotlar) ulasabilir. Bu uyeler private tanimli bile olsa bu ulasim soz konusudur. Bunun tersi de gecerlidir yani Outer class'tan Inner class in instance degiskenlerine erisim saglanabilir.

Nested Class'lar , outer/dis siniflarin bir uyesidir. Bir sinifin uyeleri 3 access modifier'a da sahip olabilir(Dolayisiyla 4 access level'e sahip olabilir) . Bu nedenle Nested Class'lar private, protected veya public olarak tanimlanabilir.

Unutmayalim ki outer siniflar ise sadece public anahtar kelimesini kullanabilir ya da package level olarak tanimlanabilir fakat private veya protected anahtar kelimesini sınıf tanımlamasında kullanamayız. Class Declaration konusunu 1.bolumde islemistik.

Nested Class nedir ve nasıl tanımlanır sorusuna yanıt aradık, peki nested class'lar neden kullanılır ?

### **Mantıksal Gruplama(Logically Grouping)**

Eğer bir nested class sadece bir sınıf için kullanışlı/useful durumda ise mantıksal olarak bu 2 sınıfı (outer ve inner) bir arada tanımlamak ve tutmak yerinde olacaktır. Nested Class'lar bir nevi yardımcı sınıflardır , bunları package level olarak tanımlamak yerine ilgili sınıfın üyesi olarak tanımlamak daha uygun olacaktır.

### **Encapsulation arttirimi**

Outer ve Inner sınıflarını düşünelim. Inner sınıfı Outer sınıfının private üyesine erişebilir.

Inner sınıfın kendisi de diğer sınıflardan gizli durumda olacaktır.

### **Readable(Okunabilirlik) and maintainable(surdurulebilirlik) kod**

Outer sınıf ile Nested Class yapısı içice birbirine yakın olacağı için kodun okunabilirlik ve surdurulebilirlik özelliği artacaktır.

Nested Class içeren kaynak dosyayı(.java) , derlediğimiz zaman hem outer class hem nested class için derlenmiş (.class uzantılı ) dosya olusur. Nested Class'ların isimlendirilmesi sekli **Outer\_Class\_Ismi\$Nested\_Class\_Ismi.class** şeklinde olacaktır. Örneğin yukarıdaki kodumuzu derlediğimiz zaman su dosyaları olusur ;

*OuterClass.class*

*OuterClass\$InnerClass.class*

*OuterClass\$StaticNestedClass.class*

.class uzantılı dosyaları , projenizde **bin** klasörü altında görebilirsiniz.

### **Non-Static Nested Class(Inner Class)**

Non-Static Nested Class'lara , inner class denilir. Bir inner class'in örneği(instance , outer class'in örneğinin hali hazırda mevcut olması ile mümkündür. Yani outer class'tan bir obje oluşturmadığımız surece , inner class'in objesi olusamaz.

Hatırlayacağımız gibi bir sınıfın üyeleri(instance variable ve instance method) ancak ilgili sınıfın objesi(instance/örneği olduğu zaman anlam kazanır. Çünkü instance değişkenler ve instance method'ları objeye(instance aittir. Benzer şekilde inner class da outer class in üyesidir , dolayısıyla outer class in objesi(instance/örneği gereklidir.

```
public class OuterClass {  
  
    public static void main(String[] args) {  
        OuterClass outerRef = new OuterClass();  
        // InnerClass'a ait bir obje(instance oluşturmak için oncelikle  
        // OuterClass tipinde bir obje(instance oluşturuyoruz.  
        // InnerClass a ait örneği(instance/objeyi şu şekillerde
```

```

// olusturabiliriz.

InnerClass innerRef = outerRef.new InnerClass();
OuterClass.InnerClass innerRef2 = outerRef.new InnerClass();

InnerClass innerRef3 = new OuterClass().new InnerClass();
OuterClass.InnerClass innerRef34 = new OuterClass().new InnerClass();

}

void createInnerClassInstance() {
    InnerClass innerClass = new InnerClass();
    // Instance Metot'tan bahsedebilmek icin OuterClass tipinde objenin
    // varligi gerekmektedir.
    // Dolayisiyla non-static metot'da InnerClass tipinde objeyi bu sekilde
    // olusturabiliriz.

}

class InnerClass {

}

```

- Inner Class , Outer Class'ta bulunan degiskenlere(instance variable veya static variable) erisim saglayabilir.Bu degiskenler private bile olsa erisim saglanır.
- Inner Class , Outer Class'ta bulunan metodlara(instance metodlara veya static metodlara) erisim saglayabilir. Bu metodlar private bile olsa erisim saglanır
- Benzer sekilde Outer Class, Inner Class'in degiskenlerine ve metodlarina erisim saglayabilir. Bu degiskenler ve metodlar private olsa bile erisim saglanır.

```

public class OuterClass2 {

    private String privateVariable = "i am private instance variable";
    private static String privateStatic = "I am private static variable";

    public static void main(String[] args) {

```

```

        OuterClass2.InnerClass2 innerRef = new OuterClass2().new InnerClass2();

        innerRef.innerTestMethod();

        System.out.println(innerRef.privateInnerVariable);

    }

    private void outerTestMethod() {

        System.out.println("Outer Test Method");

    }

    private void outerStaticTestMethod() {

        System.out.println("Outer Static Test Method");

    }

    class InnerClass2 {

        private String privateInnerVariable = "I am private inner variable";

        private void innerTestMethod() {

            System.out.println(privateVariable);

            System.out.println(privateStatic);

            outerTestMethod();

            outerStaticTestMethod();

        }

    }

}

```

Inner classlarda static degiskenler sadece final olarak tanımlanabilir. Buna rağmen final olacak şekilde bir static metot tanımlanamaz.

```

public class OuterClass3 {

    class InnerClass3 {

        private static final String staticVariable = "Static Inner Class Variable";
        // inner class içerisinde static degisken final olmalıdır.
        // inner class içerisinde static metot tanımlayamayız.
    }
}

```

```
// private final static void innerStaticMethod(){} //derleme hatasi  
}  
  
}  
  
Icice birden fazla inner class tanimlanabilir.
```

```
public class OuterClass4 {  
  
    class InnerClass {  
        class InnerClass2 {  
  
        }  
    }  
}
```

### Shadowing

Hatirlayacagımız gibi bir sinifta instance variable ve local variable aynı isme sahipse bu durumda shadowing durumu ortaya cıkmaktaydı. Bu durumda instance variable'a ulaşmak için this anahtar kelimesinden yararlanmaktaydık. Hatirlayacagımız gibi this anahtar kelimesi hali hazırda ilgili objeyi referans almaktadır.

Benzer shadowing durumu Outer class ve Inner class için de söz konusudur. Yani outer class'ta tanımlı bir instance degisken ile inner class'ta bulunan bir degiskenin ismi aynı olursa ortaya shadowing durumu çıkacaktır.

Oncelikle shadowing konusunu bir örnek ile hatırlayalım

```
public class ShadowingTest {  
  
    String shadowing = "I am instance variable";  
  
    public static void main(String[] args) {  
        ShadowingTest st = new ShadowingTest();  
        st.shadowingTest();  
    }  
  
    void shadowingTest() {
```

```

String shadowing = "I am local variable";

System.out.println(shadowing);

System.out.println(this.shadowing);

}

}

```

**this** anahtar kelimesi inner class icerisinde oldugu icin inner class a ait olan objeye referans edecektir. Bunun icin outer classta yer alan shadowinge neden olan degiskene *OuterClassName.this.InnerClassName.shadowingVariable* seklinde ulasim saglanabilir.

```

public class ShadowingOuter {

    private String shadowing = "Outer Test";

    public static void main(String[] args) {
        InnerClass innerClass = new ShadowingOuter().new InnerClass();
        innerClass.printTest();
    }
}

public class InnerClass {
    private String shadowing = "Inner Test";

    public void printTest() {
        System.out.println(shadowing);
        System.out.println(this.shadowing);
        // this anahtar kelimesi inner class icerisinde oldugu icin inner
        // class a ait olan objeye referans edecektir.
        // Bunun icin outer classta yer alan shadowinge neden olan degiskene
        // OuterClassName.this.InnerClassName.shadowingVariable
        // seklinde ulasim saglanabilir.
        System.out.println(ShadowingOuter.this.shadowing);
    }
}

```

inner class sınıf üyeleridir bu nedenle private, protected , public gibi access modifierları alabilirler.  
Benzer şekilde abstract , final ,static olarak tanımlanabilir(static tanımlandığında artık inner class olarak isimlendirilmez , static nested class olarak isimlendirilir)

```
public class Outer {  
    private class Inner {  
  
    }  
  
    class SubInner extends Inner {  
  
    }  
  
    final class FinalInner {  
  
    }  
  
    // class CanNotExtendsFinalClass extends FinalInner {}  
  
    class SubInner2 extends Outer {  
  
    }  
  
    abstract class AbstractInnerClass {  
    }  
}
```

## Pure Java – 80 Nested Class – Nested Interface

[Levent Erguder](#) 05 December 2014 [Java SE](#)

Merhaba Arkadaşlar,

Onceki bolumde **Inner Class** konusunu incelemistik. Bu bolumde **Nested Interface(Nested Interface)** konusuunu inceleyecegiz. Hatırlayacagımız gibi Inner Class’lar , Nested Class’ların bir alt koludur. Non-static Nested Class’lara , Inner Class denilmektedir. Nested Interface’lerin sadece **static** versiyonu olduğu için Nested Interface ve Inner Interface aynı anlama gelmektedir.

### Nested Interface

Java’da **Nested Class** yapısı olduğu gibi **Nested Interface** yapısı da mevcuttur. **Inner interface , Nested**

**Interface** ile aynı anlama gelmektedir. Bir sınıf veya bir interface içerisinde tanımlı interface' e , **Nested Interface(Inner Interface)** denilir.

- Nested Interface'ler private ya da protected anahtar kelimesini alamazlar. Nested Interface'ler varsayılan olarak **public**tir.
- Hatırlayacağımız gibi Interface'lerin metodları varsayılan olarak **public abstract** olarak tanımlıdır. Benzer şekilde nested interface'lerin metodlarında varsayılan olarak public ve abstract tir. private ya da protected tanımlanamaz.
- Benzer şekilde değişkenler interface ve Nested Interface'lerin değişkenleri public , final ve static'tir yanı Constant(sabittir)
- Nested Interface'ler varsayılan olarak **static** tir.

```
public interface OuterNestedInterface {  
  
    // private interface PrivateNestedInterface { } //derleme hatası.  
  
    interface PublicNestedInterface {  
        // private void privateNestedMethod(); //derleme hatası  
        public abstract void testMethod();  
    }  
  
    // nested interface varsayılan olarak public ve static'tir  
    public static interface PublicStaticNestedInterface {  
        // private int test = 2014; // private tanımlanamaz.  
        public static final int fps = 2014;  
  
    }  
}
```

Nested Interface'si uyguladığımız(implements) bir örnek yapalım.

- Dikkat ederseniz Nested Interface'si uyguladığımızda sadece Nested Interface'te bulunan metodu override etmemiz gereklidir. Outer interface'te bulunan metodu override etmemiz gerekmeyez.
- Benzer şekilde Outer interface'si uyguladığımızda(implements) sadece Outer interface'te bulunan metodları override etmemiz gereklidir. Nested Interface'te bulunan metodu override etmemiz gerekmeyez.
- Unutmayalım Java da birden fazla arabirimini/interface uygulayabiliriz.

```
interface OuterNestedInterface {  
  
    public abstract void outerMethod();
```

```
public interface PublicNestedInterface {  
    public abstract void innerMethod();  
}  
  
public class OuterNestedTest implements OuterNestedInterface.PublicNestedInterface {  
  
    @Override  
    public void innerMethod() {  
        System.out.println("test");  
    }  
  
}  
  
class Outer2 implements OuterNestedInterface {  
  
    @Override  
    public void outerMethod() {  
        // TODO Auto-generated method stub  
    }  
  
}  
  
class Outer3 implements OuterNestedInterface, OuterNestedInterface.PublicNestedInterface {  
  
    @Override  
    public void outerMethod() {  
    }  
  
    @Override  
    public void innerMethod() {  
    }  
}
```

Outer class içerisinde Nested Interface tanımlayabiliriz :

```
import chap2.OuterClassNestedInterface.NestedInterface;

public class OuterClassNestedInterface {
    interface NestedInterface {
        void nestedMethod();
    }
}

class OuterClass implements NestedInterface {
    @Override
    public void nestedMethod() {
    }
}

}

class OuterClass2 implements OuterClassNestedInterface.NestedInterface {
    @Override
    public void nestedMethod() {
    }
}

interface OuterInterface extends NestedInterface {
}
```

## Pure Java - 81 Nested Class - Local Inner Class

[Levent Erguder](#) 05 December 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yazimda Local Inner Class'tan bahsedecegim. Onceki yazilarda **Regular Inner Class** ve **Nested Interface** konusunu incelemistik.

### Local Inner Class

Local Inner Class'lar bir block içerisinde tanimlanır( { } ), bu tanimlama tipik olarak bir method içerisinde yapilir. Bu nedenle Method Local Inner class da denilmektedir.

- Local Inner Class'in objesi/ornegi/instance sadece tanimlandigi (block)metot içerisinde yapilabilir. Outer sinifin diger metotlarında ya da bir baska sinif Local Inner Class objesi olusturamaz.
- Local Inner Class'a ait obje olusturma kodu(new) ilgili metot içerisinde Local inner class tanimindan(declaration) sonra olmalıdır. Oncesinde yapilamaz , derleme hatası verir.

```
public class OuterClass {  
  
    public static void main(String[] args) {  
  
        OuterClass outer = new OuterClass();  
  
        outer.createALocalInner();  
  
    }  
  
    void createALocalInner() {  
  
        // LocalInner localInner = new LocalInner();  
  
        // derleme hatasi verir. Local Inner instance'i olusturmak icin bu kod  
  
        // local inner class tanimindan sonra olmalıdır.  
  
        class LocalInner {  
  
        }  
  
        LocalInner localInner = new LocalInner();  
  
    }  
}
```

- Regular Inner Class'larda oldugu gibi Local Inner Class'lar da , Outer sinifların private uyeleri dahil tüm uyelerine ulasım saglayabilir.

```
public class OuterClass2 {  
  
    private String privateVar = "I am Outer variable";
```

```

private void outerPrivateOuter() {
    System.out.println("I am Outer Method");
}

public static void main(String[] args) {
    OuterClass2 outer = new OuterClass2();
    outer.createALocalInner();
}

void createALocalInner() {

    class LocalInner {
        private void localInnerMethod() {
            System.out.println(privateVar);
            outerPrivateOuter();
        }
    }

    LocalInner localInner = new LocalInner();
    localInner.localInnerMethod();
}
}

```

- Local Inner Class'lar , sadece tanimlandigi metotta final local degiskenlere ulasabilir. final olmayan local degiskenlere ulasim saglanamaz.

```

public class OuterClass3 {

    public static void main(String[] args) {
        OuterClass3 outer = new OuterClass3();
        outer.createALocalInner();
    }

    void createALocalInner() {
        // String outerLocalVariable="I am outer local variable";
    }
}

```

```

final String finalOuterLocalVariable = "I am final outer local variable";
// final local degiskenlere erisim saglanabilir.

class LocalInner {
    private void localInnerMethod() {
        // System.out.println(outerLocalVariable);
        System.out.println(finalOuterLocalVariable);

    }
}

LocalInner localInner = new LocalInner();
localInner.localInnerMethod();
}
}

```

- Non-static metot içerisinde tanımlanan local inner classlarda static metot tanımlanamaz.
- Non-static metot içerisinde tanımlanan local inner classlarda static degisken sadece final olarak tanımlanabilir.

```

public class OuterClass4 {

    public static void main(String[] args) {
        OuterClass4 outer = new OuterClass4();
        outer.createALocalInner();

    }

    void createALocalInner() {

        class LocalInner {

            final static String finalStaticLocalVariable = "I am static final local variable";
            // static String staticLocalVariable="I am static local variable";
            // //derleme hatasi

            // private static void localInnerMethod() {

```

```

// //derleme hatasi

}

// private static final void localInnerMethod() {

// //derleme hatasi

}

}

LocalInner localInner = new LocalInner();

}

}

```

- Local Inner Class'lar private, protected,public tanimlanamaz. Zaten metot içerisinde tanımlı olduğu için scope'u tanımlandığı metotla sınırlıdır.
- Local Inner Class'lar abstract veya final olabilir.

```

public class OuterClass5 {

    void createALocalInner() {

        abstract class AbstractLocalInner {

        }

        class SubLocalInner extends AbstractLocalInner {

        }

        final class FinalClass {

        }

        // class SubLocalInner2 extends FinalClass {
        //
    }
}

```

```
// }  
  
}  
}
```

- static metot icerisinde de local inner class tanimlayabiliriz. benzer sekilde yine sadece final static degisken tanimlayabiliriz. static metot tanimlayamayiz.

```
public class OuterClass6 {  
  
    public static void main(String[] args) {  
  
        class LocalInner {  
  
            private String localInstanceVariable = "I am local inner variable";  
            private final static String finalStaticLocalVariable = "I am static final local variable";  
  
            // static String staticLocalVariable="I am static local variable";  
  
            void localInnerMethod() {  
                System.out.println("local inner method");  
            }  
  
            // static void staticLocalInnerMethod() {  
            // System.out.println("derleme hatasi");  
            // }  
        }  
  
        LocalInner localInner = new LocalInner();  
        System.out.println(localInner.localInstanceVariable);  
        System.out.println(localInner.finalStaticLocalVariable);  
    }  
}
```

```
public class OuterClass {
```

```

public static void main(String[] args) {
    OuterClass outer = new OuterClass();
    outer.createALocalInner();
}

void createALocalInner() {

    // LocalInner localInner = new LocalInner();
    // derleme hatasi verir. Local Inner instance'i olusturmak icin bu kod
    // local inner class tanimindan sonra olmalidir.

    class LocalInner {
    }

    LocalInner localInner = new LocalInner();
}

}

```

- Regular Inner Class'larda oldugu gibi Local Inner Class'lar da , Outer siniflarin private uyeleri dahil tum uyelerine ulasim saglayabilir.

```

public class OuterClass2 {

    private String privateVar = "I am Outer variable";

    private void outerPrivateOuter() {
        System.out.println("I am Outer Method");
    }

    public static void main(String[] args) {
        OuterClass2 outer = new OuterClass2();
        outer.createALocalInner();
    }

    void createALocalInner() {

```

```

class LocalInner {
    private void localInnerMethod() {
        System.out.println(privateVar);
        outerPrivateOuter();
    }
}

LocalInner localInner = new LocalInner();
localInner.localInnerMethod();

}

```

- Local Inner Class'lar , sadece tanimlandigi metotta final local degiskenlere ulasabilir. final olmayan local degiskenlere ulasim saglanamaz.

```

public class OuterClass3 {

    public static void main(String[] args) {
        OuterClass3 outer = new OuterClass3();
        outer.createALocalInner();
    }
}

void createALocalInner() {
    // String outerLocalVariable="I am outer local variable";
    final String finalOuterLocalVariable = "I am final outer local variable";
    // final local degiskenlere erisim saglanabilir.

    class LocalInner {
        private void localInnerMethod() {
            // System.out.println(outerLocalVariable);
            System.out.println(finalOuterLocalVariable);
        }
    }
}

LocalInner localInner = new LocalInner();

```

```
        localInner.localInnerMethod();  
    }  
}
```

- Non-static metot içerisinde tanımlanan local inner classlarda static metot tanımlanamaz.
- Non-static metot içerisinde tanımlanan local inner classlarda static degişken sadece final olarak tanımlanabilir.

```
public class OuterClass4 {  
  
    public static void main(String[] args) {  
        OuterClass4 outer = new OuterClass4();  
        outer.createALocalInner();  
  
    }  
  
    void createALocalInner() {  
  
        class LocalInner {  
  
            final static String finalStaticLocalVariable = "I am static final local variable";  
            // static String staticLocalVariable="I am static local variable";  
            // //derleme hatası  
  
            // private static void localInnerMethod() {  
            // //derleme hatası  
  
            //        // }  
  
            // private static final void localInnerMethod() {  
            // //derleme hatası  
            //        // }  
        }  
  
        LocalInner localInner = new LocalInner();
```

```
    }  
}
```

- Local Inner Class'lar private, protected,public tanimlanamaz. Zaten metot içerisinde tanımlı olduğu için scope'u tanımlandığı metotla sınırlıdır.
- Local Inner Class'lar abstract veya final olabilir.

```
public class OuterClass5 {  
  
    void createALocalInner() {  
  
        abstract class AbstractLocalInner {  
  
        }  
  
        class SubLocalInner extends AbstractLocalInner {  
  
        }  
  
        final class FinalClass {  
  
        }  
  
        // class SubLocalInner2 extends FinalClass {  
        //  
        // }  
  
    }  
}
```

- static metot içerisinde de local inner class tanımlayabiliriz. benzer şekilde yine sadece final static degişken tanımlayabiliriz. static metot tanımlayamayız.

```
public class OuterClass6 {  
  
    public static void main(String[] args) {
```

```

        class LocalInner {
            private String localInstanceVariable = "I am local inner variable";
            private final static String finalStaticLocalVariable = "I am static final local variable";
            // static String staticLocalVariable="I am static local variable";

            void localInnerMethod() {
                System.out.println("local inner method");
            }

            // static void staticLocalInnerMethod() {
            // System.out.println("derleme hatasi");
            // }
        }

        LocalInner localInner = new LocalInner();
        System.out.println(localInner.localInstanceVariable);
        System.out.println(localInner.finalStaticLocalVariable);
    }

}

```

## Pure Java - 82 Nested Class - Anonymous Inner Class

[Levent Erguder](#) 06 December 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu yaziya kadar **regular inner class** ve **method local inner** class yapilarini inceledik.Burada **Anonymous Inner Class** konusunu inceleyecegiz.

**Anonymous Class**'lar kodunuzun daha kisa olmasini saglar. Sınıf tanımlamanizi ve ilgili sınıfın bir örnek/instance oluşturmanızı aynı anda yapabileceğiniz saglar. **Anonymous Inner Class**'ların isimleri yoktur. Örneğimiz ile başlayalım ;

```

class Keyboard {
    public void write() {
        System.out.println("Keyboard Write");
    }
}

```

```

    }

}

public class Computer {

    Keyboard keyboard = new Keyboard() {

        @Override

        public void write() {

            System.out.println("Anonymous Write");

        }

    }; // noktali virgulu unutmayalim!

    Keyboard keyboard2 = new Keyboard();

    public static void main(String[] args) {

        Computer com = new Computer();

        com.keyboard.write();

        com.keyboard2.write();

    }

}

```

Anonymous inner class yapısında yeni bir obje olusturmaktaiz. Burada objenin tipi Keyboard degildir , objenin tipi Keyboard sinifinin anonymous subclass/altsinif tipindedir.

```

Keyboard keyboard = new Keyboard() {

    @Override

    public void write() {

        System.out.println("Anonymous Write");

    }

};

```

*Unutmayalim ; Java'da anonymous inner class ,subclass/alt siniftir !*

```
class Engine {
```

```

public void speedUp() {
    System.out.println("Engine speedUp");
}

}

public class Car {

    Engine engine = new Engine() {

        @Override
        public void speedUp() {
            System.out.println("Anonymous speedUp");
        }

        public void drift() {

        }
    };

    public static void main(String[] args) {
        Car car = new Car();
        car.engine.speedUp();

        // car.engine.drift();
        // derleme hatasi
    }
}

```

Burada "engine" referans degiskeni Engine sinifi tipindedir. Objenin tipi ise Engine sinifinin , Anonymous alt sinifi tipindedir.

Hatirlayacagimiz gibi , Java'da bir referans degisken kendisi veya alt sinif tipinde objeye referansta bulunabilir. Referans degiskenin ait oldugu Engine sinifinda drift isimli metot bulunmamaktadir Bu nedenle drift metodunu cagiramayiz !

speedUp metodunun override hali cagrılacaktır , cunku objenin tipi alt sinif tipindedir.

Anonymous yapısında class/sinif kullanabildigimiz gibi abstract class/sinif veya interface/arabirim de kullanabiliriz.

```
interface MyInterface {  
    public void test();  
}  
  
abstract class AbstractClass {  
    public abstract void test2();  
}  
  
public class AnonymousTest {  
  
    public static void main(String[] args) {  
        MyInterface myInterface = new MyInterface() {  
            @Override  
            public void test() {  
                System.out.println("Anonymous test");  
            }  
        };  
  
        myInterface.test();  
  
        AbstractClass abs = new AbstractClass() {  
  
            @Override  
            public void test2() {  
                System.out.println("Anonymous test2");  
            }  
        };  
  
        abs.test2();  
    }  
}
```

Burada interface tipinde bir obje olusturmuyoruz! MyInterface arabirimini uygulayan(implements) anonymous sinif tipinde bir obje olusturuyoruz. Burada anonymous implementer class sadece bir arabirimi uygulayabilir ve anonymous subclass ya bir ilgili sinifi kalitabilir(extends) ya da ilgili arabirimi uygulayabilir(extends).

- Bir anonymous class , outer class in uyelerine erisebilir.(methods & variable)

```
class Anonymous {  
    void test() {  
    }  
}  
  
public class AnonymousTest2 {  
  
    private String privateVariable = "private variable";  
  
    private void outerMethod() {  
        System.out.println("Outer method");  
    }  
  
    Anonymous anonymous = new Anonymous() {  
        void test() {  
            System.out.println(privateVariable);  
            outerMethod();  
        }  
    };  
  
    public static void main(String[] args) {  
        AnonymousTest2 anonymousTest2 = new AnonymousTest2();  
        anonymousTest2.anonymous.test();  
    }  
}
```

- Bir anonymous class eger metot icerisinde tanimlanmissa, metot icerisindeki local degiskenlere ulasamaz. Eger local degisken final tanimliysa bu durumda ulasim saglanabilir.

```
class Anonymous {  
    void test() {
```

```

    }

}

public class AnonymousTest3 {

    private void outerMethod() {
        String localVariable="I am local variable";
        final String finalLocalVariable="I am final local variable";
        Anonymous anonymous = new Anonymous() {
            void test() {
                //System.out.println(localVariable); //derleme hatası
                System.out.println(finalLocalVariable);
            }
        };
    }
}

```

- **Shadowing** durumu Anonymous class'lar için de ortaya çıkabilir.

```

class Anonymous {
    void test() {
    }
}

public class AnonymousTest4 {

    private String shadowing = "Shadowing Outer";

    private void outerMethod() {

        Anonymous anonymous = new Anonymous() {
            void test() {
                String shadowing = "Shadowing Anonymous";
                System.out.println(shadowing);
                System.out.println(AnonymousTest4.this.shadowing);
            }
        };
    }
}

```

```
        }
    };
}

}
```

Anonymous class içerisinde static degisken sadece final olarak tanımlanabilir. static metot ise final olsa da tanımlanamaz.

```
class Anonymous {
    void test() {
    }
}

public class AnonymousTest5 {

    private String shadowing = "Shadowing Outer";

    Anonymous anonymous = new Anonymous() {
        // static String staticVariable = "compile error";

        final static String finalStaticVariable = "i am final static variable";

        void test() {
            String shadowing = "Shadowing Anonymous";
            System.out.println(shadowing);
            System.out.println(AnonymousTest5.this.shadowing);
        }
    };

    // static void compileError() {}
    // final static void compileErrorToo() {}

};

}
```

Buraya kadar inceledigimiz Anonymous Inner Class'lar, **Plain-Old Anonymous Inner Class** olarak isimlendirilir. Bir diger kullanım sekli olarak Argument-Defined Anonymous Inner Class olarak adlandirilan yapı vardir.

```
interface MyInterface {  
    void test();  
}  
  
class MyClass {  
    void doStuff(MyInterface iface) {  
        System.out.println("MyClass doStuff");  
    }  
}  
  
public class ArgumentDefinedAnonymous {  
  
    public static void main(String[] args) {  
        MyClass myClass = new MyClass();  
  
        myClass.doStuff(new MyInterface() {  
  
            @Override  
            public void test() {  
                // test metodu  
                // anonymous inner class  
            } //doStuff metodu  
        });  
    }  
}
```

doStuff metodu MyInterface tipinde bir parametre almaktadir. new anahtar kelimesi ile interface objesi olusturamayiz.

Bu durumda **Argument Defined Anonymous Class** kullanabiliriz. doStuff metodunu cagirirken bu yapıyi kullanabiliriz.

## Pure Java – 83 Nested Class – Static Nested Class

[Levent Erguder](#) 07 December 2014 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde Static Nested Class'lari incelecegiz. Bu yazi ile Nested Class konusunu bitirecegiz.

Static Nested Class'lara, Static Inner Class ifadesi kullanilmaktadir fakat bu ifade dogru degildir . Static Nested Classlar , Top-level nested class'tir.

```
public class Outer {  
  
    static class StaticNested {  
    }  
  
}
```

StaticNested sinifinin kendisi static degildir, static sinif diye bir sey yoktur. static anahtar kelimesinin anlamı Outer sinifinin static bir uyesi anlamında kullanilmaktadir.Static Nested Class bir inner class degildir.

- Bu nedenle, diger static uyeler icin oldugu gibi Outer sinifinin instance/objesi olmadan da bu static uyeye ulasilabilir. Inner classlara ulasim icin Outer objesi gereklidir.

```
public class Outer {  
  
    static class StaticNested {  
    }  
  
    class InnerClass {  
  
    }  
  
    public static void main(String[] args) {  
  
        Outer.StaticNested staticNested = new Outer.StaticNested();  
        StaticNested staticNested2 = new StaticNested();  
  
        Outer.InnerClass innerClass = new Outer().new InnerClass();  
        InnerClass innerClass2 = new Outer().new InnerClass();  
    }  
}
```

- Static Nested Class'lar static veya non-static metotlara sahip olabilir. Inner Class sadece non-static metoda sahip olabilir.

```

public class Outer {

    static class StaticNested {
        void method() {
            System.out.println("StaticNested method");
        }

        static void staticMethod() {
            System.out.println("StaticNested static method");
        }
    }

    class InnerClass {
        void method() {
            System.out.println("InnerClass method");
        }

        // static void staticMethod() {
        //     System.out.println("compile error");
        // }
    }

    public static void main(String[] args) {

        Outer.StaticNested staticNested = new Outer.StaticNested();
        StaticNested staticNested2 = new StaticNested();

        Outer.InnerClass innerClass = new Outer().new InnerClass();
        InnerClass innerClass2 = new Outer().new InnerClass();
    }
}

```

- Static Nested Class'lar static veya instance degisken tanimlayabilir. Static tanimli metotdan non-static instance degiskene ulasim saglanamaz.

```

public class Outer {

    static class StaticNested {
        private String instanceVariable = "I am instance variable";
        private static String staticVariable = "i am static variable";

        void method() {
            System.out.println(staticVariable);
            System.out.println("StaticNested method");
        }

        static void staticMethod() {
            //System.out.println(instanceVariable);
            //derleme hatası
            System.out.println(staticVariable);
            System.out.println("StaticNested static method");
        }
    }

    public static void main(String[] args) {

        Outer.StaticNested staticNested = new Outer.StaticNested();
        StaticNested staticNested2 = new StaticNested();

    }
}

```

- Static Nested sınıf, Outer sınıfının static değişkenlerine ve static metodlarına erişim sağlanabilir. Instance değişkenlere ve non-static değişkenlere erişim sağlanamaz.

```

public class Outer {

    private String outerInstanceVariable = "i am outer instance variable";
    private static String outerStaticVariable = "i am static outer instance variable";
}

```

```

private void outerTest() {

}

private static void outerStaticTest(){

}

static class StaticNested {
    void method() {
        // System.out.println(outerInstanceVariable);
        System.out.println(outerStaticVariable);

        //outerTest();
        //derleme hatasi

        outerStaticTest();
    }
}

```

## BÖLÜM-9

### Pure Java - 84 Thread - 01

[Levent Erguder](#) 11 December 2014 [Java SE](#)

Merhaba Arkadaslar

9.bolum Java'ta Thread mekanizmasi ile ilgili olacak. 9.bolumde ;

- Thread nedir? Nasil olustururuz ?
- Thread i nasil calistiririz ?
- Thread'in durumlari nelerdir ?
- Senkronizasyon
- Thread Interaction/Etkilesimi gibi konulari inceleycegiz.

Java'da “**thread**” 2 anlama gelmektedir;

- java.lang.Thread sinifinin bir ornegi/instance/objesi
- calisan bir is parcacigi(lightweight process)

java.lang.Thread sinifinin bir orgnegi/instance Java'da bildigimiz diger objeler gibidir. Benzer sekilde degiskenleri, metotlari vardir , Heap'te yasarlar. Java'da sinif hiyerarsisinin tepesinde java.lang.Object sinifi bulunmaktadır. Her sinif varsayılan olarak Object sinifini kalitmaktadir dolayisiyla Thread IS-A Object onemesi doğrudur.

“calisan is parcacigi” olarak anlami ; kendi “**Stack**” alanina sahip individual(bireysel/ayri/ozgun) lightweigh process anlamina gelmektedir.

Process kavramı , Isletim Sitemi acisindan genel olarak calisan program ve uygulamalar anlamina gelir. Word, paint, Eclipse gibi. Bir process birden fazla thread/is parcacigi icerebilir.

Java'da her thread'e karsilik bir stack calisir. Eger siz yeni thread'ler olusturmasaniz bile bir thread calisacaktir. main() metodu “main” ismine sahip bir thread'i calistirir. Eger kendimiz yeni thread'ler olusturursak bu durumda bu tread'ler farkli farkli Stacklerde calisacaktir.

JVM(Java Virtual Machine), mini bir Isletim Sistemi (OS) gibi calisir ve thread'lerin program/zaman yonetimi(schedule) JVM in kontrolu altindadir.Soz konusu Thread oldugunda “garanti” , “kesin” bir durum soz konusu degildir. Farkli JVM'ler farkli thread schedule mekanizmasi kullanabilirler.

java.lang.Thread sinifini inceldigimizde onemli olan(sinavda sorumlu oldugumuz) su metotlari gorebiliriz. Bunlari ilerleyen yazılarda inceleyecegiz.

**start()**  
**yield()**  
**sleep()**  
**run()**

Thread konusunda olayın action kismi **run()** metodudur.Yeni bir thread'te bir is(job) yapmak istiyorsanız bunu **run()** metodu içerisinde tanımlayabilirsiniz. Yeni bir stack cagrilmasina **run()** metodu baslatır.

Yeni bir “is parcacigi/thread” olusturmak icin 2 yaklasim vardir ;

- java.lang.Thread sinifini kalitmak
- java.lang.Runnable arabirimini uygulamak

**java.lang.Thread** sinifini kalitmak en kolay yoludur fakat genel olarak guzel bir Object Oriented yaklasimi degildir.

Bir Car sinifi Thread sinifini kalittiği zaman Car IS-A Thread onemesi doğru olur. Subclass/alt sinif , super class/ust sinifin daha ozellesmis hali olmalıdır. Eger daha ozellesmis bir Thread yapisi alt sinifta yapılmayacaksa kalitmak iyi bir OO tercihi olmayacaktır.

Ikinci olarak, Java'da bir sinif sadece bir sinifi kalitabilir. Thread sinifini kalittigimiz zaman kotayı doldurmus oluruz.

Bunun yerine genel olarak **java.lang.Runnable** arabirimini kullanmak daha doğru bir yaklasim olacaktır.

- Bir sınıf birden fazla arabirimini uygulayabilir dolayısıyla herhangi bir kısıta neden olmaz.
- Arabirimler sınıf hiyerarşisinde yer almazlar ve sınıf hiyerarşisinden bağımsız olarak herhangi bir sınıf tarafından uygulanabilir.

### **extending java.lang.Thread ( Thread sınıfını kalitmak)**

Bir thread(is paracığı) tanımlamak/define için ;

- java.lang.Thread sınıfını kalıt.
- run() metodunu override et

```
public class CoolThread extends Thread {  
  
    @Override  
    public void run() {  
        System.out.println("The job is starting!");  
    }  
  
}
```

Override edilmiş run metoduna dikkat edelim. Bununla birlikte Overloaded run metodları da tanımlayabiliriz fakat overloaded run metodları kendimiz çağrımadıktan sonra Thread sınıfı tarafından bu metodlar görmezden gelinir(ignore). Thread sınıfının dikkate aldığı metod parametre almayan run() metodudur ve bu metodun çağrıması yeni bir stack oluşturmasını sağlar.

Kısacası ;

- Sadece parametre almayan run() metodunu Thread sınıfı tarafından otomatik çağrılrır.
- run metodunu yeni bir stack oluşturmasını sağlar.

### **implementing java.lang.Runnable (Runnable arabirimini uygulamak)**

java.lang.Runnable arabirimini incelediğimizde sadece run metodunun tanımı olduğunu görebiliriz.

```
public interface Runnable {  
  
    public abstract void run();  
}
```

Runnable arabirimini uygulayarak , “is paracığı”/thread tanımlama özelliğine sahip olabiliyoruz.

```
public class CoolRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println("The job is starting!");  
    }  
}
```

```
}
```

```
}
```

### Instantiating a Thread( Thread sinifinden bir ornek olutmak)

Her is parcaciginin(thread) baslangici icin Thread objesi/ornegi gereklidir.

Ister Thread sinifini kalitalim , ister Runnable arabirimini uygulayalim fark etmez yine de Thread sinifinin ornegine/objesine ihtiyacimiz vardır.

Eger Thread sinifini kalittiyasak bu durumda ilgili siniftan bir instance/ornek/obje olusturmamiz yeterli olacaktır.

```
CoolThread t = new CoolThread();
```

Eger Runnable arabirimini uyguladiysak , bu durumda hala Thread sinifi ornegine/instance ihtiyacimiz vardır.

Thread sinifiniisci(worker) .Runnable arabirimini is(job) olarak dusunebiliriz. Birisi bizim isimizi (job) yapmalidir.

Thread sinifinda Runnable parametre alan bir yapılandirici mevcuttur. Runnable arabirimini uygulayan siniflarin orneklerini/objelerini bu yapılandiriciya arguman olarak gecebiliriz.

```
CoolRunnable coolRunnable = new CoolRunnable();
```

```
Thread t = new Thread(coolRunnable);
```

```
Thread t2 = new Thread(coolRunnable);
```

```
Thread t3 = new Thread(coolRunnable);
```

Burada oldugu gibi birden fazla java.lang.Thread yapılandiricisina ayni CoolRunnable objesini vermemizin anlami ayni is uzerinde(job) üzerinde birden fazlaisci(worker) calisabilir anlamina gelmektedir.

Thread sinifinin kendisi Runnable'dir. Bunun anlami Runnable tipinde parametre alan yapılandiriciya ,Thread tipinde bir arguman gecebiliriz.

Bu legal bir durumdur fakat kotu bir yaklasimdir bir geregi yoktur.

```
Thread t2 = new Thread(new CoolThread());
```

Buraya kadar yeni bir Thread objesi olusturmaktan bahsettim. calisan bir “**is parcacigi**” kavramindan henuz bahsetmedik.

Thread'ler farkli state/durumlarda bulunabilir. Bunlardan ilerleyen bolumlerde bahsedecegiz.

Bir Thread objesi olusturuldugu , durumu/state “**new**” dir. Bu durumda thread/is parcacigi , “**alive/canli**” durumda degildir. Ne zaman ki **start()** metodu cagrilir bu durumda “**new**” durumundan cikar. **run()** metodu tamamlandigi zaman “**dead/olu**”duruma gecer.  
Bu durumların/state kontrolü **isAlive()** ve **getState()** metodları yardımı ile sağlanabilir.  
Siniflarımızın son hali;

```
public class CoolThread extends Thread {  
  
    @Override  
    public void run() {  
        System.out.println("The job is starting!");  
    }  
  
    public static void main(String[] args) {  
        CoolThread t = new CoolThread();  
        Thread t2 = new Thread(new CoolThread());  
        }  
    }  
  
public class CoolRunnable implements Runnable {  
  
    @Override  
    public void run() {  
        System.out.println("The job is starting!");  
    }  
  
    public static void main(String[] args) {  
        CoolRunnable coolRunnable = new CoolRunnable();  
        Thread t = new Thread(coolRunnable);  
        Thread t2 = new Thread(coolRunnable);  
        Thread t3 = new Thread(coolRunnable);  
        }  
    }
```

## Pure Java - 85 Thread - 02

[Levent Erguder](#) 16 December 2014 [Java SE](#)

Merhaba Arkadaslar,

Onceki yazida Java'da Thread konusuna giris yaptik, thread kavramindan bahsettik. Bu bolumde Thread'i calistirmayı inceleyecegiz.

Hatirlayacagimiz gibi yeni baslayan bir thread yeni bir Stack meydana getirecektir. Thread'i baslatmak icin **start()** metodunu kullaniriz.

```
t.start();
```

- t, java.lang.Thread sinifi tipinde referans degiskendir.
- Thread'in baslamasi yeni bir stack olusturur.
- Thread "new" durumundan "runnable" durumuna gecer.
- Runnable duruuna gectikte sonra run() metodunu calistirilabilir.

Unutmayalim, Thread 'i baslatabiliriz(**start**) , Runnable'ı baslatmak(**start**) kavrami soz konusu degildir. Bu nedenle start metodu java.lang.Thread sinifinin ornegi/instance uzerinden(Thread tipinde referans degisken aracılıgiyla ) olur.

Basit bir ornek ile baslayalim.

```
public class MyThread extends Thread {  
  
    @Override  
    public void run() {  
        System.out.println("This is run() method!");  
    }  
  
    public static void main(String[] args) {  
  
        MyThread thread = new MyThread();  
        thread.start();  
  
    }  
}
```

Hatirlayacagimiz gibi java.lang.Thread sinifi java.lang.Runnable arabirimini uygulamaktadir. Runnable arabirimini run() metoduna sahiptir. Burada run() metodunu override ettik ve yapilmasi gereken isi burada belirttik.

- start() metodu yeni bir thread ve stack baslatir daha sonrasinda run metodunu cagirir.
- run() metodunu direkt cagirmak yeni bir thread ve stack olusturmaz fakat legal bir durumdur. Birden fazla run metodunu cagirabiliriz.
- run() metodunu cagirmak diger metotlari cagirmak gibidir, dolayisiyla diger metotlarda oldugu gibi ayni mevcut/current stack icerisinde calisir/yasar.
- Unutmayalim instance metotlar Stack'te yasarlardır.
- start() metodunu bir referans degisen uzerinden cagirdiktan sonra , tekrar start metodunu cagirdigimizda java.lang.IllegalThreadStateException hatasi aliriz.

```
public class MyThread extends Thread {

    @Override
    public void run() {
        System.out.println("This is run() method!");
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();

        thread.run();
        thread.run();

        // thread.start();
        // java.lang.IllegalThreadStateException

    }
}
```

Benzer ornegimizi Runnable arabirimini kullanarak da yapabiliyoruz.

```
class MyRunnableThread implements Runnable {

    @Override
    public void run() {
        System.out.println("This is Runnable run metot");
    }
}
```

```

    }

}

public class RunnableTest {

    public static void main(String[] args) {

        MyRunnableThread myRunnable = new MyRunnableThread();
        Thread t = new Thread(myRunnable);
        t.start();

    }
}

```

Simdi de birden fazla Thread ile calisan kucuk bir ornek yazalim;

```

class MyRunnableMoreThread implements Runnable {

    @Override
    public void run() {
        System.out.println("run method()");
        for (int i = 0; i < 10; i++) {
            String name = Thread.currentThread().getName();
            long id = Thread.currentThread().getId();
            System.out.println("index:" + i + " " + name + " " + id);
        }
    }
}

public class RunnableMoreThreadTest {

    public static void main(String[] args) {

```

```

MyRunnableMoreThread myRunnable = new MyRunnableMoreThread();

Thread t1 = new Thread(myRunnable);

Thread t2 = new Thread(myRunnable);

Thread t3 = new Thread(myRunnable);

t1.setName("First");

t2.setName("Second");

t3.setName("Third");

t1.start();

t2.start();

t3.start();

}

}

```

Ornegimizi incelersek t1, t2, t3 isimlerine sahip 3 tane Thread referans degiskenleri olusturduk. Bu referans degiskenleri olustururken myRunnable referans degiskenini Thread sinifinin yapislandiricisina arguman olarak verdik.

setName metodu ile threadlere isim verebiliriz. Sonrasinda start metodunu her bir referans degisken icin cagirdik.

Thread.currentThread() o an calisan Thread'i doner. Ornegimizi calistirdigimizda suna benzer bir cikti aliriz;

```

run method()
run method()
index:0 First 9
index:1 First 9
index:2 First 9
index:3 First 9
index:0 Second 10
index:4 First 9
run method()
index:0 Third 11
....
```

Bu ciktinin bir GARANTISI yoktur! Burada hangi thread'in once calisacagi, hangisinin ne kadar calisacagini(duration) bir garantisi yoktur. Bir thread baslatiktan sonra isi bitmeden(run metodu

tamamlanmadan) digeri baslayabilir/calisabilir. Burada onemli nokta her bir thread icin run() metodu tamamlanacaktır.

Bir thread calismasini sonlandirdiginda(run metodu tamamlandiginda) olu/dead duruma gecer. Teknik olarak API de karsiliği TERMINATED durumudur. Bununla birlikte Thread objesi hala yasamaktadir. Thread in TERMINATED durumda olması objenin silinmesine neden olmaz.

Bir thread TERMINATED duruma geldiginde artık yeniden baslatilamaz. Bir thread referans degikseni uzerinden 2.kez start() metodunu cagiramayiz bu durumda IllegalThreadStateException hatasi aliriz. Bir thread yeniden baslatilamaz.

run metodunu cagirmak yeni bir thread baslatmaz, diger metot cagrilmalari gibi ayni stack uzerinde cagrilir. start metotlarini run metodu olarak degistirirsek;

```
t1.run();  
t2.run();  
t3.run();
```

```
run method()  
index:0 main 1  
index:1 main 1  
index:2 main 1  
index:3 main 1  
index:4 main 1  
index:5 main 1  
index:6 main 1  
index:7 main 1  
index:8 main 1  
index:9 main 1  
run method()  
index:0 main 1  
index:1 main 1  
index:2 main 1  
index:3 main 1  
....
```

Ciktiyi inceledigimiz zaman sadece “main” threadi calismaktadir. Bununla birlikte burada sonuc/cikti GARANTIDIR. Diger metot cagrilmalarinda oldugu gibi calisacaktir. run metodu ilk kez cagrildiktan sonra bittikten sonra 2 ve 3.kez cagrilacaktir.

### Thread Scheduler

Thread Scheduler, JVM'in bir parcasidir.

Thread Scheduler hangi eligible/uygun/secilebilir “thread” in calistirilacagina karar verir. Eligible/uygun thread’ten kasit runnable state/durumdaki threadlerdir.

Runnable State’e sahip threadler , JVM tarafindan calistirilabilir/aktif halde is yapar. Runnable durumda hali hazinda calisan thread(currently running thread) durumuna gecemezler.

Hangi runnable state’e sahip thread’in secilecegine dair bir garanti yoktur. Burada queue/kuyruk mantigi da gecerli degildir, garanti yoktur. Yani 1.thread calisti sonra 2.thread calisti 3.thread calismadan tekrar 1.thread o an calisabilir.

Thread’leri runnable queue yerine runnable pool olarak dusunebiliriz.

Thread Scheduler bizim kontrolümüzde degildir, JVM tarafindan kontrol edilir. Sadece bazi metotlar yardimi ile bu schedule islemini etkileyebiliriz(influence)

**java.lang.Thread**

```
public static void sleep(long millis) throws InterruptedException  
public static void yield()  
public final void join() throws InterruptedException  
public final void setPriority(int newPriority)
```

**java.lang.Object**

```
public static void sleep(long millis) throws InterruptedException  
public static void yield()  
public final void join() throws InterruptedException  
public final void setPriority(int newPriority)
```

Bu metotlari ilerleyen yazılarda inceleyecegiz.

## Pure Java – 86 Thread – 03

[Levent Erguder](#) 18 December 2014 [Java SE](#)

Merhaba Arkadaslar

Bu yazida bir Thread’in durumundan/state bahsedecegiz. **java.lang.Thread** sinifinda **State** isminde bir enum yer almaktadir. Bu enumda su degerler mevcuttur.

```
public class Thread implements Runnable {  
    public enum State {
```

NEW,

```
RUNNABLE,  
  
BLOCKED,  
  
WAITING,  
  
TIMED_WAITING,  
  
TERMINATED;  
}  
  
}
```

### **NEW**

Thread objesi/instance olusturuldugu fakat **start** metodunun henuz cagrilmadigi durumda thread'in durumu **NEW** 'dir.

Bu durumda Thread objesi mevcuttur ve **live** durumundadir fakat thread calismaya baslamadigi icin **not-alive** durumdadır.

### **RUNNABLE**

Thread , RUNNABLE durumuna ilk olarak **start()** metodu cagrildigi zaman girer, bununla birlikte blocked/waiting/sleeping durumlarindan tekrar RUNNABLE duruma donebilir.

Thread , RUNNABLE durumunda **alive** durumdadır.

RUNNABLE durumu , thread'in calismaya(**running** duruma gecme) uygun/eligible oldugu durumdur.

### **RUNNING**

Thread Scheduler ,thread pool/havuzundan runnable durumdaki(state) bir thread'i secer ve bu thread calismaya baslar(running).

RUNNING durumu asil action'in oldugu durumdur. RUNNING durumundan tekrar RUNNABLE duruma donus ya da waiting/blocking/sleeping durumlarina donus soz konusudur.

Burada onemli nokta Thread.State enum'inda **RUNNING** degeri yoktur. Burada RUNNING durumundan kasit hali hazilda Thread Scheduler tarafindan secilen RUNNABLE thread'tir.

### **WAITING**

**WAITING** state , thread'in calismaya uygun/eligible olmadigi bir durumdur. Thread bu durumda **alive** durumdadır fakat Thread Scheduler tarafindan secilmek icin uygun degildir(not eligible). Sadece runnable durumunda olan bir thread running durumu icin uygun olabilir (eligible)

**WAITING** durumunun anlami bir thread diger bir thread'in belirli bir isi yapmasini bekliyor demektir. Bir Thread'in durumu su durumlarda WAITING olabilir.

## **java.lang.Object**

```
public final native void wait() throws InterruptedException;
```

Bir thread , bir obje uzerinden wait metodunu cagirdiginda WAITING state'e gecer. Bir thread WAITING duruma gectiginde bir baska thread'in ilgili obje uzerinden **notify()** ya da **notifyAll()** metodunu cagirmasi ile WAITING durumu sonlanir ve RUNNABLE duruma gecilir.

#### **java.lang.Thread**

```
public final void join() throws InterruptedException
```

join metodu , bir thread bitene kadar(run metodu sonlana kadar) diger thread lerin beklemesini saglar.

join metodu cagrildiginda, hali hazilda calisan thread(currenct thread) , **thread1** thread'i islemini bitirene kadar bekleyecektir(wait).

```
thread1.join();
```

#### **TIMED\_WAITING**

TIMED\_WAITING durumu da thread'in calismaya uygun/eligible olmadigi durumlardir. Thread bu durumda alive durumdadır fakat Thread Scheduler tarafindan secilmek icin uygun degildir.

**join** ve **wait** metodlarinin overloaded halleri mevcuttur. Overloaded metodlarda belirli bir timeout suresi kullanilabilir. Bu durumda TIMED\_WAITING durumu soz konusu olacaktir.

#### **java.lang.Object**

```
public final native void wait(long timeout) throws InterruptedException;
```

Bir thread , bir obje uzerinden wait(long timeout) metodunu cagirdiginda TIMED\_WAITING state'e gecer. Bir thread TIMED\_WAITING duruma gectiginde bir baska thread'in ilgili obje uzerinden **notify()** ya da **notifyAll()** metodunu cagirmasi ya da timeout ile WAITING durumu sonlanir ve RUNNABLE duruma gecilir.

#### **java.lang.Thread**

```
public final synchronized void join(long millis) throws InterruptedException
```

**join** metodu , bir thread'in diger thread islemini bitirene kadar beklemesini saglar fakat burada oldugu gibi parametre alan overloaded join metodu thread'in belirtilen timeout suresi kadar beklemesini saglar.

#### **java.lang.Thread**

```
public static native void sleep(long millis) throws InterruptedException;
```

Bir thread , SLEEPING durumunda olabilir. sleep metodu static bir metottur , cagrildigi noktada hali hazilda calisan thread'i milisaniye cinsinden sleeping durumuna girmesini saglar.

Burada onemli nokta uyuma suresi bittiginde tekrar calismaya devam edeceginin bir garantisi yoktur !

Ilgili Thread sleeping durumundan sonra tekrar RUNNABLE duruma gecer.Bu durumda ne zaman Thread Scheduler tarafindan secilirse tekrar RUNNING durumuna gececektir.

SLEEPING durumu da ayri bir enum degeri yoktur. Daha ozel isim olarak SLEEPING kavrami kullanilmaktadir. Yoksa sleep metodu , ilgili thread icin TIMED\_WAITING durumuna sahip olmasina neden olur.

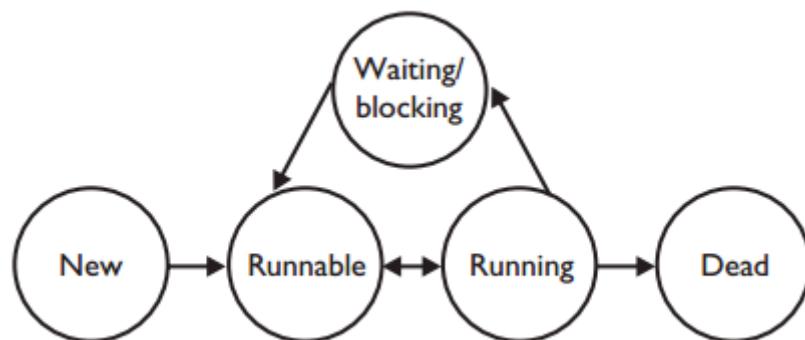
### **BLOCKED**

synchronized bir block'a ya da metoda ayni anda sadece bir thread girebilir. Ilgili objenin kilidine (object lock) hangi thread sahipse o thread isini bitirene kadar bir baska thread synchronized metoda ya da block'a giris yapamaz. Burada oldugu gibi diger threadlerin obje kildi icin bekledigi duruma **BLOCKED** durumu denilir.

### **TERMINATED**

Thread calismasini sonlandirdiginda yani run metodu sonlandiginda DEAD yani TERMINATED duruma gecer.

TERMINATED duruma gecen bir thread tekrar RUNNABLE duruma gecemez.



## Pure Java - 87 Thread - 04

[Levent Erguder](#) 20 December 2014 [Java SE](#)

Merhaba Arkadaslar,

Onceki bolumde Thread State'lerini inceledik. Bu bolumde java.lang.Thread sinifinda yer alan su metodlari inceleyecegiz;

- **sleep**
- **yield**
- **join**

### **sleep**

*java.lang.Thread* sinifinda overloaded 2 tane **sleep** metodu yer almaktadir.

```
public static native void sleep(long millis) throws InterruptedException;  
public static void sleep(long millis, int nanos)
```

Burada dikkat edilecek nokta;

- sleep metodu **static** bir metottur.
- sleep metodu checked bir exception olan **InterruptedException** fırlatabilir. Dolayisiyla bu metodu cagirdigimiz yerde **handle or declare** kuralina uymamiz gereklidir.
- Birinci metot **native** bir metottur ve long tipinde parametre almaktadir. **sleep** native metodu JVM tarafindan uygulanir.
- 1 saniye 1000 milisaniyedir.
- Ikinci metot long tipinde ve int tipinde parametre almaktadir. Bu metotta milisaniye+nanosaniye olarak uyuma/sleep suresi belirlenir. 1 milisaniye 1000000( $10^6$ ) nanosaniyedir. Ikinci metot birinci sleep metodunu cagirir.

```
public static void sleep(long millis, int nanos)  
throws InterruptedException {  
    if (millis < 0) {  
        throw new IllegalArgumentException("timeout value is negative");  
    }  
  
    if (nanos < 0 || nanos > 999999) {  
        throw new IllegalArgumentException(  
            "nanosecond timeout value out of range");  
    }  
  
    if (nanos >= 500000 || (nanos != 0 && millis == 0)) {  
        millis++;  
    }  
  
    sleep(millis);  
}
```

**Thread.sleep** metodu , hali hazinda calisan Thread'i, arguman olarak verilen belirtilmis sure kadar askiya alir(suspend)/uyutur.

sleep metodunda belirtilmis sure bittiginde , tekrar calismaya devam edeceginin bir **garantisi yoktur**. Bu sure bittiginde uyutulmus thread, timed\_waiting durumundan runnable durumuna gecer. Thread Scheduler tarafindan calistirilmak icin tekrar uygun hale gelir(eligible).

Dolayisiyla “minimum” , arguman olarak verilen sure kadar ilgili thread calismayacaktir.

**Unutmayalım; sleep metodunu static metottur. t , Thread tipinde bir referans degisen olsun. t referans degiseni üzerinden sleep metodunu çağırınak bu thread'i sleep durumuna geçirmez ! sleep metodunu çağrıldığı noktada hali hazırda çalışan thread'i sleep durumuna geçirir!**

```
Thread.sleep(1000);  
t.sleep(1000);
```

Simdi basit bir örnek yapalım, 3 tane thread oluşturalım , run metodunda 500 milisaniye(0.5 saniye) olacak şekilde sleep metodunu kullanalım.

```
public class ThreadSleepTest implements Runnable {  
  
    public static void main(String[] args) {  
        ThreadSleepTest sleepTest = new ThreadSleepTest();  
  
        Thread thread1 = new Thread(sleepTest);  
        Thread thread2 = new Thread(sleepTest);  
        Thread thread3 = new Thread(sleepTest);  
  
        thread1.setName("myThread1");  
        thread2.setName("myThread2");  
        thread3.setName("myThread3");  
  
        thread1.start();  
        thread2.start();  
        thread3.start();  
    }  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println("Thread name :" + Thread.currentThread().getName() + " index:" +  
i);  
            try {  
                Thread.sleep(500);  
                // sleep metodunu 0.5 saniye hali hazırda çalışan thread'in  
                // durumunu sleeping(timed_waiting) yapacaktır.  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

```

        } catch (InterruptedException e) {
            System.out.println("exception handled");
        }
    }
}

```

Burada ciktinin/output (oncelik acisindan) bir garantisi yoktur.

Thread.sleep metodunu kodumuzda her yerde cagirabiliriz. main metodu calistiginda “main” ismine sahip thread calisacaktir. Burada calisan main threadini 3 saniye sleep durumuna gecirebiliriz.

```

public class ThreadMainTest {

    public static void main(String[] args) {
        System.out.println("Before calling sleep");

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("After calling sleep");
    }
}

```

### **priority & yield**

Thread'lerin sahip oldugu bir **priority**(oncelik) degeri vardir. Bu deger genel olarak 1-10 arasında degismektedir, fakat farkli Thread Schedulerlar icin bu deger 10dan kucuk de olabilir.

JVM , preemptive(sonsuz oncelikli), priority-based bir schedule algoritmaya sahiptir. Bu JVMlerde time-slicing(zaman dilimi) scheduler yaklasimini uygulanir.

Bununla birlikte tum JVM'lerde time-slicing yaklasimi uygulanmak zorunda degildir. JVM spesifikasyonu time-slicing scheduler uygulanmasini şart kosmaz.

Bir çok JVM için threadlerin priority değeri önemlidir. Birden fazla thread runnable durumundaysa yüksek oncelige(priority) sahip thread running durumuna geçirmek için tercih edilebilir.

Bununla birlikte, bu durumun bir garantisı yoktur.

Priority değerleri daha etkili/verimli program olması açısından kullanılabilir fakat oncelik sıralaması garanti bir şekilde çalışmamayacaktır.

MAX\_PRIORITY, MIN\_PRIORITY, NORM\_PRIORITY değerlerine ve “main” threadinin priority değerlerine göz atalım;

```
public class PriorityTest {  
  
    public static void main(String[] args) {  
  
        System.out.println(Thread.MAX_PRIORITY);  
  
        System.out.println(Thread.MIN_PRIORITY);  
  
        System.out.println(Thread.NORM_PRIORITY);  
  
        System.out.println(Thread.currentThread().getPriority());  
  
    }  
}
```

setPriority metodunu kullanarak threadlere priority(oncelik) değeri verebiliriz.

```
public class ThreadPriorityTest implements Runnable {  
  
    @Override  
    public void run() {  
  
        for (int i = 0; i < 10000; i++) {  
  
            System.out.println(Thread.currentThread().getName() + " index:" + i);  
        }  
    }  
  
    public static void main(String[] args) {  
  
        ThreadPriorityTest sleepTest = new ThreadPriorityTest();  
  
        Thread thread1 = new Thread(sleepTest);  
        Thread thread2 = new Thread(sleepTest);  
    }  
}
```

```

Thread thread3 = new Thread(sleepTest);

thread1.setName("myThread1");
thread2.setName("myThread2");
thread3.setName("myThread3");

thread1.setPriority(10);
thread2.setPriority(1);
thread3.setPriority(5);

thread1.start();
thread2.start();
thread3.start();

}

}

```

Simdi de yield metodunu inceleyelim ;

```
public static native void yield();
```

yield kelimesinin bir çok turkçe anlamı vardır buraya en uygun olanı yerini bırakmak/geri çekilmek olacaktır.

yield metodu çağrılmışında , hali hazırda çalışan metot geri çekiliyor(runnable duruma) geçer ve yerini aynı priority değeri sahip bir thread'e bırakır.

yield metodu sleeping/waiting/blocking gibi durumlara neden olmaz. Bununla birlikte tabii yield metodunun da bir garantisı yoktur. Schedule bu isteği gormezden gelebilir(ignore).

yield metodunu kullanmak çoğu zaman pek kullanışlı ve uygun olmayacağından (appropriate). Test amaçlı ve debug amaçlı kullanılabilir(race condition test).

### **join**

join metodu , bir thread bitene kadar(run metodunu sonlana kadar) diğer thread'lerin beklemesini sağlar.

- sleep ve yield metodlarının aksine join metodu non-static bir metodudur.
- join metodu InterruptedException fırlatabilir.
- 3 tane overloaded versiyonu vardır.

```
public final void join() throws InterruptedException  
public final synchronized void join(long millis)  
public final synchronized void join(long millis, int nanos)
```

1.metot islem bitene kadar bekleyecektir. 2.metot verilen sure(milisaniye) kadar bekleyecektir. 3.metot da verilen sure(milisaniye+nanosaniye) kadar bekleyecektir.

join() metodu cagildiginda, hali hazilda calisan thread(current thread) , thread1 thread'i islemini bitirene kadar bekleyecektir(wait).

```
thread1.join();
```

Simdi de join olmadan ve join kullanarak basit bir ornek yapalim.

```
public class ThreadWithoutJoinExample implements Runnable {  
  
    @Override  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println("Thread name:" + Thread.currentThread().getName() + " index:" +  
i);  
        }  
    }  
  
    public static void main(String[] args) {  
  
        ThreadWithoutJoinExample runnable = new ThreadWithoutJoinExample();  
  
        Thread thread1 = new Thread(runnable);  
        Thread thread2 = new Thread(runnable);  
        Thread thread3 = new Thread(runnable);  
  
        thread1.setName("myThread1");  
        thread2.setName("myThread2");  
        thread3.setName("myThread3");  
    }  
}
```

```

        thread1.start();
        thread2.start();
        thread3.start();
    }

}

```

Burada 3 tane thread kullandık , çıktı/output tahmin edilemez.

```

Thread name:myThread1 index:0
Thread name:myThread2 index:0
Thread name:myThread1 index:1
Thread name:myThread2 index:1
Thread name:myThread1 index:2
Thread name:myThread2 index:2
Thread name:myThread1 index:3
Thread name:myThread2 index:3
Thread name:myThread1 index:4
Thread name:myThread2 index:4
Thread name:myThread3 index:0
.....

```

Simdi de join kullanalım;

```

public class ThreadJoinExample implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println("Thread name:" + Thread.currentThread().getName() + " index:" +
i);
        }
    }

    public static void main(String[] args) throws InterruptedException {
        ThreadJoinExample runnable = new ThreadJoinExample();

```

```

        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        Thread thread3 = new Thread(runnable);

        thread1.setName("myThread1");
        thread2.setName("myThread2");
        thread3.setName("myThread3");

        thread1.start();
        thread1.join();

        // thread1 bitene kadar baska thread calismaz.

        thread2.start();

        thread2.join();

        // thread2 bitene kadar baska thread calismaz.

        thread3.start();

    }

}

```

Bu ornegimizde thread1.join diyerek bu thread bitene kadar diger thread'lerin calismasina engel oluyoruz. Bittikten sonra thread2.join digerek diger threadlerin calismasina engel oluyoruz. Bu durumda ciktimiz her defasinda ayni olacaktir ve sirali bir sekilde yazacaktir.

```

Thread name:myThread1 index:0
Thread name:myThread1 index:1
Thread name:myThread1 index:2
Thread name:myThread1 index:3
Thread name:myThread1 index:4
Thread name:myThread1 index:5
Thread name:myThread1 index:6
Thread name:myThread1 index:7
Thread name:myThread1 index:8
Thread name:myThread1 index:9
Thread name:myThread2 index:0

```

```
Thread name:myThread2 index:1  
Thread name:myThread2 index:2  
Thread name:myThread2 index:3  
Thread name:myThread2 index:4  
...
```

sleep , yield ve join metodunu kısaca özetleyecek olursak ;

**sleep** metodu , hali hazırda çalışan threadin(current thread) ,**InterruptedException** olmadıgı surece, “en az” belirtilen sure kadar(milisaniye ya da milisaniye+nanosaniye) kadar uyumasını sağlayacaktır(timed\_waiting). “en az” olmasının nedeni sure biter bitmez tekrar çalışacagının bir garantisı yoktur. Sure bittiginde thread , timed\_waiting durumundan den runnable duruma gecer. Dolayısıla Thread Scheduler tarafından seçilmeye uygun hale gelir.  
**yield** metodunun bir garantisı yoktur. Hali hazırda çalışan threadin geri çekimmesini(yield) ve yerini aynı priority degerine sahip başka bir threadin running duruma geçmesi için Thread Scheduler'a öneride bulunur. Thread Scheduler bu metodu görmezden gelebilir(ignore).  
**join** metodu , bir thread bitene kadar(run metodu sonlana kadar) diğer thread'lerin beklemesini saglar.

## Pure Java - 88 Thread - 05

[Levent Erguder](#) 29 December 2014 [Java SE](#)

Merhaba Arkadaşlar

Bu bölümde **Thread Interference** , **Memory Consistency Error** ve **Thread Safe & Shared**

**Resources** konusunu inceleyeceğiz

Simdi aşağıdaki küçük kod örneğimiz üzerinden kötü bir senaryo düşünelim ;

```
public class Counter {  
    private long count = 0;  
    public void add(long value){  
        this.count = this.count + value;  
    }  
}
```

A ve B olmak üzere 2 tane thread olduğunu düşünelim. Burada value değeri 2 ve 3 olarak değerini varsayıyalım;

*this.count = 0* değerine sahiptir.  
A threadi register(hafıza)'dan değeri okur. (*this.count = 0*)  
B threadi register(hafıza)'dan değeri okur. (*this.count = 0*)

A thread'i 2 degerini ekler.

A thread'i 2 degerini register'a yazar. this.count = 2 dir.

B thread'i zaten degeri okumustu bu noktada 3 degeri ekler.

B thread'i register'a 3 degerini yazar. this.count = 3 oldu.

this.count degerini 5 beklerken 3 oldu ! Peki neden boyle oldu ?

Birden fazla thread , “ayni kaynaga” ulastiginda (read) problem olmaz.

Problem teskil edecek nokta ; birden fazla thread “ayni kaynaga”(instance variable), “ayni obje referansi” uzerinden ulasip degistirmeye calistiginda(write) ortaya cikabilir. Bu probleme **Race Condition** adi verilir. Local degiskenerin durumu ve instance degiskenler yazinin sonunda tekrar ele alinacaktir.

**Running** durumunda bir thread , Thread Scheduler tarafindan heran calismasi runnable duruma gecebilir bir baska thread running duruma gecebilir.

Bu nedenle hangi thread'in “paylasilan kaynaga” ulasmaya calistigini bilemeyeiz.

**Thread Interference**(girisim/karisma/karisim) , ayni ‘data’ uzerinde farkli threadler araliklarla(interleave) calistigi durumda ortaya cikabilir.

Benzer bir ornek olarak :

c++ ifadesi(statement) icin su adimlar yapilir;

c degerinin mevcut degeri alinir

c degeri bir(1) artirilir. (c- , icin 1 azaltılır)

c degeri kaydedilir(store)

Thread A : c degiskenenin degerini alir.

Thread B : c degiskenenin degerini alir.

Thread A : c degiskeninin degerini arttirir. c = 1

Thread B : c degiskeninin degerini azaltir. c = -1

Thread A : c degiskeninin degerini kaydeder(store) c = 1

Thread B : c degiskeninin degerini kaydeder(store) c = -1

Burada increment ve decrement metodlarindan sonra beklenen deger 0 olmalidir , fakat Thread A'nin c degiskenenin degerini 1 yapmasi Thread B tarafindan overwrite edildi. Farkli durumlarda da tam tersi bir durum olabilir.

Bu ornek cogu zaman dogru calisacaktir, fakat bunun bir garantisi yoktur.

```
class Counter implements Runnable {
```

```
    @Override
```

```
    public void run() {
```

```
        increment();
```

```
        decrement();
```

```
    }

    int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}

class CounterTest {

    public static void main(String[] args) {
        Counter counter = new Counter();

        Thread t1 = new Thread(counter);
        Thread t2 = new Thread(counter);

        t1.start();
        t2.start();

        System.out.println(counter.c);
    }
}
```

```
}
```

Bir baska problem olarak **Memory Consistency Error** karsimiza cikmaktadır.

```
int counter = 0;  
  
public void increment() {  
  
    counter++;  
  
    System.out.println(counter);  
  
}
```

counter degiskeni bir artirildiktan sonra consola degeri basilmakta. Eger bu iki ifadeyi(statement) ayni thread calistiracaksa problem yok fakat counter++ ifadesini ayri thread console basilmasini ayri thread yapacak olursa bu durumda ekran'a 0 basma ihtimali vardır.

Thread A'daki bu degisimin Thread B tarafindan gorunur/visible olmasinin bir garantisi yoktur.

Farkli threadler ayni 'data' üzerinde tutarsız olduğu durumda **Memory Consistency Error** ortaya çıkar.

#### Thread Safe & Shared Resources

Thread safe ; birden fazla thread **ayni anda/es zamanlı olarak/simutaneously** metoda erismeye calistiginda problem olmaması anlamını tasır. Burada problemden kasit ; race condition , deadlock , memory consistency error gibi problemlerdir.

#### Local Variables

Local Variables , **thread-safe** ozellige sahiptir. Her thread'in kendi stack alanında local degiskenler yasarlar. Bunun anlamı local degiskenler thread'ler arasında paylasılmazlar.

```
public void test(){  
  
    int threadSafeLocalVariable = 0;  
  
    threadSafeLocalVariable++;  
  
}
```

#### Local Object References

Local degiskenler primitive ya da reference type olsun **Stack'te** yasarlar. Objeler ise **Heap'te** yasar. Local Object Reference'in kendisi **Stack'te** yasar. Bu reference degiskenin gösterdiği obje ise **Heap'te** yasar.

Bir obje local olarak kullanılıyorsa yani metod dışarısında çıkmazsa bu durumda thread safe ozellige sahiptir.

```
public void testMethod() {  
  
    LocalObject localObject = new LocalObject();
```

```
        localObject.callMethod();  
  
        testMethod2(localObject);  
  
    }  

```

**testMethod’u** inceledigimizde , calisan her thread kendi LocalObject tipinde objesini olusturacaktir ve localObject reference degiskeni bu objeyi gosterecektir. Burada oldugu gibi kullanim sekli LocalObject objesini thread-safe yapacaktir. Yani objenin method içerisinde olusmasi ve geri dondurulmemesi(void) , localObject referans degiskeni uzerinden metod cagrilmasi problem teskil etmez.

Reference type degiskenen metot parametresi olarak kullanildigi durumda metodun **thread-safe** ozelligi ortadan kalkar.

Birden fazla thread testMethod2 “ayni” obje referansi uzerinden testMethod2’ye ulasirsa bu durumda ayni objeyi birden fazla thread paylasacaktır. Burada her thread icin bir obje **olusmaz!!**.

Burada birden fazla thread calissa bile sadece bir tane LocalObject tipinde obje olusacaktır. Her thread icin bir obje olusmadigi icin yani obje threadler arasında paylasildigi icin thread-safe durumu ortadan kalkar.

```
public void testMethod2(LocalObject localObject) {  
  
    localObject.setValue("value");  
  
    ....  
  
}  

```

### Object Members

Yazinin basinda inceledigimiz ornekler Object Member yani instance degiskenlerin thread safe olmamasindan kaynaklanan problemlerle ilgiliydi.

Instance degiskenler obje ile birlikte heap’te yasrarlar. Bu nedenle eger birden fazla thread “ayni obje” refansi uzerinden bir metot cagirrsa ve bu metot içerisinde instance degiskenin degeri degistirilirse bu durumda bu metot **thread-safe** degildir.

```
public class NotThreadSafe {  
  
    StringBuilder builder = new StringBuilder();  
  
    public void add(String text) {  
  
        this.builder.append(text);  
  
    }  
  
    private long count = 0;  

```

```
public void add(long value) {  
    this.count = this.count + value;  
}  
  
}
```

Birden fazla thread add metodunu “ayni NotThreadSafe objesi” uzerinden ayni anda/es zamanli olarak/simultaneously cagirirsa burada race condition durumu ortaya cikar.

```
public class NotThreadSafe {  
  
    StringBuilder builder = new StringBuilder();  
  
    public static void main(String[] args) {  
        NotThreadSafe sharedInstance = new NotThreadSafe();  
  
        new Thread(new MyRunnable(sharedInstance)).start();  
        new Thread(new MyRunnable(sharedInstance)).start();  
  
    }  
  
    public void add(String text) {  
        this.builder.append(text);  
    }  
  
}  
  
class MyRunnable implements Runnable {  
    NotThreadSafe instance = null;  
  
    public MyRunnable(NotThreadSafe instance) {  
        this.instance = instance;  
    }  
  
    public void run() {  
        this.instance.add("some text");  
    }  
}
```

```
    }  
}
```

Bununla birlikte birden fazla thread add metodunu “farkli NotThreadSafe“ objesi uzerinden **ayni anda/es zamanli olarak/simultaneously** cagirirsa burada race condition durumu ortaya cikmaz.

....

```
public static void main(String[] args) {  
    NotThreadSafe sharedInstance1 = new NotThreadSafe();  
    NotThreadSafe sharedInstance2 = new NotThreadSafe();  
  
    new Thread(new MyRunnable(sharedInstance1)).start();  
    new Thread(new MyRunnable(sharedInstance2)).start();  
}
```

Burada her thread kendi NotThreadSafe objesine sahiptir. NotThreadSafe objesi paylasilmadigi icin thread'lerin birbirine mudahale etmesi/karismasi(interfere) mumkun degildir. Burada race condition durumu ortaya cikmayacaktir.

## Pure Java – 89 Thread – 06 – synchronized

[Levent Erguder](#) 01 January 2015 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde Thread Safe & Immutability ve Synchronizing konusunu inceleyecegiz. Daha sonrasinda synchronized anahtar kelimesini inceleyecegiz.

Onceki yazida **race condition** ve **thread safe** konusunu incelemistik. Java'da immutable objeler thread safe ozellige sahiptir.

Immutable sinifimizi inceleyelim, value isminde private bir instance degisen tanimladik ve yapılandiricida bu degeri assign ediyoruz.

Bu degiskeni degistirmek icin bir setter metodu tanimlamadik sadece getter metodu tanimladik. Boylece Immutable sinifimiz immutable ozellige sahip oldu.

```
public class Immutable {  
    private int value = 0;  
  
    public Immutable(int value) {  
        this.value = value;  
    }
```

```

    }

    public int getValue() {
        return this.value;
    }
}

```

Simdi de bir add metodu tanımlayalım , eger yine immutable ozelligini korumak istersek su sekilde yapabiliriz ;

```

public class Immutable {
    private int value = 0;

    public Immutable(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }

    public Immutable add(int valueToAdd) {
        return new Immutable(this.value + valueToAdd);
    }
}

```

Bir obje **thread safe** olsa da , bu objenin reference degiskeni **thread safe** olmayabilir.

```

public class Calculator {
    private Immutable currentValue = null;

    public Immutable getValue() {
        return currentValue;
    }

    public void setValue(Immutable newValue) {

```

```
this.currentValue = newValue;  
}  
  
public void add(int newValue) {  
    this.currentValue = this.currentValue.add(newValue);  
}  
}
```

Calculator sınıfı Immutable sınıfı tipinde referans degiskene sahiptir.

Calculator HAS-A Immutable.

setValue metodu ve add metodu üzerinden currentValue değeri degistirilebilir.  
currentValue instance degiskendir dolayısıyla thread safe ozellige sahip degildir. Birden fazla thread  
ayni Calculator objesi üzerinden currentValue değerini degistirmeye calisirsa race condition ortaya  
cikacaktır.

Immutable sınıfının thread safe olması , bu Immutable sınıfın kullanılmasını (HAS-A) thread safe  
yapmaz. Peki thread safe ozelligini nasıl saglayacagiz ? **synchronized** anahtar kelimesi ile...

### **synchronized**

**synchronized** block ya da metot , race condition durumundan kacitmamizi saglar. Java'da synchronized  
anahtar kelimesi metotlar ve block'larla birlikte kullanilabilir.

Java'da synchronization mekanizması Lock(kilit) kavramı ile calisir. Java'da her objenin bir  
yerlesik/dahili/built-in kilidi(**Lock**) vardir. Bu kilide **intrinsic(esas/asıl) lock** veya **monitor lock** denilir.  
synchronized instance metota giren bir Thread, ilgili sınıfın hali hazırda objesinin kilidini(lock) alır. Ilgili  
sınıfın hali hazırda objesinden kasıt ; “**this**” tir. Hatırlayacağımız gibi instance metotlar objeye aittir  
ve **this** anahtar kelimesi hali hazırda mevcut objeye referansta bulunur.

Her obje icin sadece bir tek kilit/lock(**intrinsic Lock**) vardir. Bir thread ilgili objenin kilidini ele gecirir ve  
birakana kadar(release) baska bir thread bu kilidi ele geciremez. Bunun anlamı **synchronized** bir metota  
giren bir thread ilgili objenin kilidini ele gecirir. Bir baska thread bu **synchronized** metota , kilidi elinde  
bulunduran thread'in synchronized metotta calismasi sonlana kadar giremez.

Sadece metotlar ve blocklar **synchronized** olabilir degiskenler ya da siniflar **synchronized** olamazlar.

**Her objenin bir tek kilidi(Lock) vardir.**

**Bir thread birden fazla kilide(Lock) sahip olabilir.** Ornegin; bir thread synchronized bir metoda girer ve  
ilgili this objesinin kilidine sahip olur.

Bu synchronized metottan farkli bir obje referansi üzerinden synchronized bir metot cagirir bu durumda  
yne ilgili this objesinin kilidine sahip olur.

Bir siniftaki tum metotlar synchronized tanimlanmak zorunda degildir. Bir sinif hem synchronized hem  
de non-synchronized metotlara sahip olabilir.

Onceki yazida yazdigimiz race condition'a neden olan ornegimizi tekrar inceleyelim ;

```
class Counter implements Runnable {  
  
    @Override  
    public void run() {  
        increment();  
        decrement();  
    }  
  
    int c = 0;  
  
    public void increment() {  
        c++;  
    }  
  
    public void decrement() {  
        c--;  
    }  
  
    public int value() {  
        return c;  
    }  
}  
  
class CounterTest {  
  
    public static void main(String[] args) {  
  
        Counter counter = new Counter();  
  
        Thread t1 = new Thread(counter);  
        Thread t2 = new Thread(counter);  
  
        t1.start();  
        t2.start();  
    }  
}
```

```
        System.out.println(counter.c);

    }

}
```

Onceki yazida race condition'a neden kaynaklandigini incelemistik. Race Condition durumunu cozmek icin increment ve decrement metodlarini **synchronized** yapabiliriz.

```
public synchronized void increment() {
    c++;
}

public synchronized void decrement() {
    c--;
}
```

Hatirlayacagimiz gibi c++ veya c- ifadesi(statement) race conditiona neden oluyordu. increment ve decrement metodlari synchronized tanimlandiginda bu metodlara ayni anda 2 tane thread giremez. Dolayisiyla 2 thread arasında race condition durumu olamaz. synchronized tanimlanan metodlar thread-safe ozellige sahiptir.

Burada bir tek Counter objesi kullanildigi icin synchronized metodlar threadlerin birbirini blocklamasina neden olur. Eger her thread farkli objeyi kullanirsa bu durumda synchronized metodlar threadlerin birbirini blocklamasina neden olmaz. Bu durumda ornegin increment metodlari 2 thread icin de ayni anda calisabilir , bir thread izerinden calisirken diger thread bu thread'i beklemez. Bu threadler birbirlerini etkilemez. Bununla birlikte zaten bu durumda synchronized olmasa bile race condition durumu olmaz.

```
Counter counter1 = new Counter();
Counter counter2 = new Counter();

Thread t1 = new Thread(counter1);
Thread t2 = new Thread(counter2);

t1.start();
```

```
t2.start();
```

**static** metotlar da synchronized olabilir. Hatırlayacagımız gibi static metotlar ve static degiskenler sinifa aittir.

non-static metotlar ise objeye aittir. Bu nedenle objenin kilidini ele geciren thread ayni obje uzerinden calisan threadleri bloke ediyordu. Peki static metotlar icin durum nasil olacak ?

static methodlarda “**this**” uzerinden bir kilit yaklasimi yoktur.

static metotlar uzerinden **synchronized** yapildigindan kilit ele gecirme islmemi “**java.lang.Class objesi**” uzerinden olacaktir. Her sinif icin sadece bir tek “Class Objesi” vardir. Bu kilidi ele geciren thread static synchronized metot'a girecektir. Bu kilit birtane oldugu icin ayni anda sadece bir tek thread static **synchronized** metoda girebilir.

Java da **java.lang.Class** isminde bir sinifimiz yer almaktadir. Class literali sinif/class veya arabirimini temsil etmek/represent icin kullanilir.

**not:** bu konu Java reflection konularina girmektedir. Sinav kapsaminda Java reflection yer almamaktadir. Bu yazilar bittikten sonra bu konuda yazi yazacagim. Simdilik basit bir ornek inceleyelim.

```
public class ClassLiteral {
```

```
    ClassLiteral() {
        System.out.println("Constructor run!");
    }
```

```
    public static void main(String[] args) throws ClassNotFoundException, InstantiationException,
```

```
        IllegalAccessException {

```

```
            Class class1 = ClassLiteral.class;
            Class class2 = Class.forName("ClassLiteral");
            //Class.forName("packageName.ClassLiteral"); olmalidir!
```

```
            class1.newInstance();
            class2.newInstance();
        }
    }
```

*synchronized non-static method calisirken “this”(hali hazirda uzerinde calisilan ) uzerinden kilit ele gecirilir.  
synchronized static method calisirken CLASSNAME.class uzerinden kilit ele gecirilir.CLASSNAME.class , ilgili sinifin “java.lang.Class Objesi” ni temsil eder.*

```
public class MyClass {
```

```

public static int staticCount;
public int instanceCount;

public static synchronized int getCount() {
    staticCount++;
    return staticCount;
}

public synchronized int getCount2() {
    instanceCount++;
    return instanceCount;
}

}

```

Counter örneğimizi metodları static olacak şekilde tekrar inceleyelim;

```

class Counter implements Runnable {

    @Override
    public void run() {
        increment();
        decrement();
    }

    static int c = 0;

    public static synchronized void increment() {
        c++;
    }

    public static synchronized void decrement() {
        c--;
    }
}

```

```

public static int value() {
    return c;
}

}

class CounterTest {

    public static void main(String[] args) {

        Counter counter = new Counter();

        Thread t1 = new Thread(counter);
        Thread t2 = new Thread(counter);

        t1.start();
        t2.start();

        System.out.println(counter.c);

    }
}

```

Burada tek bir Counter objesi kullandik, eger her thread icin yeni Counter objesi kullansaydik non-static metodlarda oldugu gibi calismayacaktir.  
non-static metodlar icin farkli objeler üzerinde calisildiginda bu durumda threadler birbini blocklamazlar, race condition durumu ortaya cikmaz.  
static synchronized metodlar icin paylasilan tek bir kilit oldugu icin , bu durumda threadler farkli objeler üzerinden de calissa bu durumda birbirlerini blocklayacaklardir.

```

Counter counter1 = new Counter();
Counter counter2 = new Counter();

Thread t1 = new Thread(counter1);

```

```
Thread t2 = new Thread(counter2);
```

- static synchronized metotlar ve non-static synchronized metotlar arasında bir engelleme/blocklama söz konusu olmaz.
- static synchronized metotlar **java.lang.Class** instance/objesi üzerinde kilit , non-static synchronized metotlarda **this** üzerinde kilit söz konusudur.

metotları synchronized yapabildiğimiz gibi blockları da synchronized yapabiliriz.

tüm metodu synchronized yapmamıza gerek yoktur, race condition'a neden olabilecek kod blogunu synchronized yapmak yeterli olacaktır.

Bir metot içerisinde sadece bir block'u synchronized yaptıgımızda bu metoda aynı anda 2 thread girebilir fakat synchronized yapılan block'a aynı anda sadece bir thread girebilir.

**“Farklı objeler”** üzerinde çalışan threadler birbirini blocklamaması durumu synchronized block yapısı için de geçerlidir.

```
public class SynchronizedBlock {  
  
    int count = 0;  
  
    public void add(int value) {  
        // not synchronized  
        synchronized (this) {  
            this.count += value;  
        }  
        // not synchronized  
    }  
}
```

Onceki örneğimizi genişletelim ;

```
public class MyClass {  
  
    public static int staticCount;  
    public int instanceCount;  
  
    public static synchronized int getCount() {  
        staticCount++;  
    }  
}
```

```

        return staticCount;
    }

    public static int getCount2() {
        synchronized (MyClass.class) {
            staticCount++;
            return staticCount;
        }
    }

    public synchronized int getCount3() {
        instanceCount++;
        return instanceCount;
    }

    public int getCount4() {
        synchronized (this) {
            instanceCount++;
            return instanceCount;
        }
    }
}

```

## Pure Java - 90 Thread - 07 - Atomic & volatile

[Levent Erguder](#) 03 January 2015 [Java SE](#)

Merhaba Arkadaslar,

Bu yazida **Atomic** kavramini ve **volatile** anahtar kelimesini inceleyecegiz.

**Atomic** action, bolunemeyen en kucuk islem/adim olarak dusunebiliriz.

Ornek ile aciklayacak olursak ;

c++ ifadesi(statement) icin su adimlar yapilir;

*c degerinin mevcut degeri alinir*

*c degeri bir(1) artirilir.*

*c degeri kaydedilir(store)*

Yani burada 3 tane **atomic** action soz konusudur. Programlama acisindan kucuk bir kod parçası gibi gozuken c++ ifadesi **atomic** bir yapıya sahip degildir.

Atomic action'lar **interleaved**(aralikli) degildir bunun anlami atomic actionlar **thread interference**(karisma) tehlikesi arz etmez.

Bununla birlikte hala tum tehlikelerden korunmus demek degildir. **Memory consistency error** hala tehlike arz etmektedir.

Memory consistency error problemini tekrar hatirlayacak olursak ;

```
int counter = 0;  
public void increment() {  
    counter++;  
    System.out.println(counter);  
}
```

counter degiskeni bir artirildiktan sonra consola degeri basilmakta. Eger bu iki ifadeyi(statement) ayni thread calistiracaksa problem yok fakat counter++ ifadesini ayri thread console basilmasini ayri thread yapacak olursa bu durumda ekran'a 0 basma ihtimali vardir.

Thread A ve Thread B olsun , Thread A'daki bu degisimin(c++ isleminin sonucunun) Thread B tarafindan gorunur/visible olmasinin bir garantisi yoktur.

Farkli threadler ayni 'data/variable' uzerinde tutarsiz oldugu durumda **memory consistency error** ortaya cikar.

- **volatile** anahtar kelimesi **memory consistency error** problemine karsi bir cozum saglar.
- **volatile** bir degiskenen degerinin degismesi diger thread'ler tarafindan her zaman gorunur hale gelir.
- Memory access'ler atomic'tir. primitive ya da referans variable'larin okunup/yazilmasi atomic'tir. long ve double icin bu garanti yoktur.
- **volatile** olarak tanimlan long ve double degiskenler icin de Memory access atomic hale gelir.

**volatile** bir degisken CPU cache yerine bilgisayarın main memory'sinden okunur ve CPU cache yerine main memory'sine yazilir. non-volatile degiskenler icin boyle bir garanti yoktur.

**volatile** anahtar kelimesi degisken uzerinde bir degisiklik olduguna diger threadler tarafindan gorunmesini/visibility **garanti eder**.

Birden fazla thread kullanicilar bir uygulamada , non-volatile olan degiskenler performans nedeniyle CPU cache'inde tutulabilir. Bilgisayarın birden fazla CPU'su oldugunu dusunusek her thread farkli CPU uzerinde calisabilir bu nedenle her thread non-volatile olan degiskenlerin degerini CPU cache'ine farkli degerde yazabilir.

Yukarıdaki memory consistency error problemine neden olabilecek ornegi tekrar inceleyecekl olursak ;

```
int counter = 0;  
public void increment() {  
    counter++;  
    System.out.println(counter);
```

}

Bu metot içerisinde birden fazla thread çalışığında bir thread counter++ yaparak değerini 1 yaptığında bu değer CPU cache yazılırsa problem ortaya çıkabilir. counter değişkenin non-volatile olduğu için main memory'e yazılmasını bir garantisı yoktur.

counter değişkenini volatile olarak tanımladığımızda JVM bu değişkenin her zaman main memory'e yazılacağını ve main memory'den okunacığının garantisini verir.

```
volatile int counter = 0;
```

Birden fazla thread değişkene ulaşıp son(latest) değerini almak istediklerinde(read) volatile anahtar kelimesi yeterli olacaktır.

Thread A ve Thread B volatile bir değişkenin değerini main memory'den aynı anda alır ve ikisi de bir artırmış ve tekrar main memory yazarsa beklenen değer 2 olması gerekiyorken 1 değer yazılır.

Thread A **volatile** bir değişkeni main memory yazarsa , Thread B'nin daha sonraki okumaları(read) problem teskil etmez. Cunku **volatile** bir değişken main memory yazılır ve böylelikle diğer thread'ler için visibility/okunabilirlik problemi olusmaz.

Birden fazla thread paylaşılan bir değişken üzerinde read/write işlemi yapıyorsa volatile anahtar kelimesi yeterli olmayacağıdır. Bu durumda **synchronized** anahtar kelimesi kullanılmalıdır.

Bir thread **volatile** değişkeni main memory yazar ve diğer thread'ler için son değeri(latest) main memory'den okuma garantisı vardır. non-volatile değişkenler için böyle bir garanti yoktur.

**volatile** bir değişkeni main memory'e yazmak CPU cache yazmaktan daha maliyetli bir istir , gerekmedikçe kullanılmamaya ozen gösterelim

## Pure Java - 91 Thread - 08 - Deadlock

[Levent Erguder](#) 03 January 2015 [Java SE](#)

Merhaba Arkadaşlar,

Bu yazımız deadlock konusunu inceleyeceğiz. Multithread uygulamada threadlerin zamanında çalışmasına **Liveness** denilir. Bu çalışmaya engel olan bazı problemler vardır; bunlardan bir tanesi **Deadlock'lardır**

**Deadlock** ; 2 veya daha fazla thread'in birbirlerini blocklama durumudur. Her objenin built-in bir kilidi(lock) vardır.

Thread'ler , birbirlerinin elinde bulunan obje kilidini beklediği durumda bu blocklama durumu söz konusu olacaktır.

Thread 1 synchronized bir methoda girer ve A kilidini ele gecirir. Daha sonrasinda yeni bir synchronized metoda girmeye calisir fakat bu metoda girmek icin gerekli olan obje kilidi Thread 2 nin elindedir(B kilidi). Bu durumda Thread 1 , Thread 2 nin isini bitirmesini ve obje kilidini birakmasini bekler(B kilidi).

Bu sirada Thread 2 , synchronized bir methoda girer ve B kilidini ele gecirir. Bu metod icerisinde synchronized bir method yer almaktadir ve A kilidi gerekmektedir, fakat A kilidi Thread 1'in elinde bulunmaktadır. Bu durumda Thread 2 , Thread 1 'in isini bitirmesini ve obje kilidini birakmasini bekler(A kilidi).

Thread 1 , B kilidini beklemektedir.Thread 2 ise A kilidini beklemektedir. Threadlerin birbirlerinin kilitlerine ihtiyac duyduyu ve birakmasini/release bekledigi bu durumda deadlock durumu ortaya cikacaktir.

```
public class TestThread {  
    public static Object LockA = new Object();  
    public static Object LockB = new Object();  
  
    public static void main(String args[]) {  
  
        ThreadDemo1 thread1 = new ThreadDemo1();  
        ThreadDemo2 thread2 = new ThreadDemo2();  
  
        thread1.start();  
        thread2.start();  
    }  
  
    private static class ThreadDemo1 extends Thread {  
        public void run() {  
            synchronized (LockA) {  
                System.out.println("Thread 1: Holding lock A...");  
                try {  
                    Thread.sleep(10);  
                } catch (InterruptedException e) {  
                }  
                System.out.println("Thread 1: Waiting for lock B...");  
                synchronized (LockB) {  
                    System.out.println("Thread 1: Holding LockA & LockB...");  
                }  
            }  
        }  
    }  
}
```

```

    }

private static class ThreadDemo2 extends Thread {
    public void run() {
        synchronized (LockB) {
            System.out.println("Thread 2: Holding LockB...");
            try {
                Thread.sleep(10);
            } catch (InterruptedException e) {
            }
            System.out.println("Thread 2: Waiting for LockA...");
            synchronized (LockA) {
                System.out.println("Thread 2: Holding LockA & LockB...");
            }
        }
    }
}

```

Bu problemi cozmek icin kilitlerin yerini degistirebiliriz ya da bu tarz bir lock mekanizmasi yerine java.util.concurrent.locks paketinde yer alan Lock arabirimini kullanilarak timeout suresi verilebilir. Bu konu sinav kapsaminda yer almamaktadir. Bu yazilar bittiginde ilerleyen donemlerde Thread konusunu guncelledigimde bu konu eklenecektir.

## Pure Java - 92 Thread - 09 - Thread Interaction

[Levent Erguder](#) 04 January 2015 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde java.lang.Object sinifinda yer alan **wait** , **notify** ve **notifyAll** metodlarini inceleyecegiz.

Bir thread mail-delivery threadi , diger threa ise mail-processor threadi olarak dusunelim.

wait-notify mekanizmasi yardimi ile, mail-processor threadi maili ornegin 2 saniyede bir surekli check edebilir ya da bunun yerine mail geldiginde beni uyar/bilgilendir diyebilir(notify).

Bir baska degisle , wait notify mekanizmasi , threadi bir nevi bekleme odasina alir ve diger threadin bilgilendirmesi/ilani ile(notify) bekleme odasindan cikmasini saglar.

**wait** , **notify** ve **notifyAll** metodlari mutlaka synchronized metot veya block icerisinden cagrilmalidir. Bir thread ilgili objenin kilidini ele gecirmeden wait ya da notify metodlarini cagiramaz.

wait ve notify olmadan bir ornek yapalim;

```

public class WithoutWaitNotifyTest {

    public static void main(String[] args) {
        Calculate calculateThread = new Calculate();
        calculateThread.start();

        System.out.println("After calculate...");
        System.out.println("total is: " + calculateThread.total);
        System.out.println("counter is: " + calculateThread.counter);

    }
}

class Calculate extends Thread {

    int total;
    int counter;

    @Override
    public void run() {

        for (int i = 0; i < 100000; i++) {
            total += i;
            counter = i;
        }
    }
}

```

Calculate sınıfında , dongu içerisinde basit bir işlem yaplıyoruz , örneği bir kere çalıştırırsak farklı çıktılar elde ederiz, örneğin ;

*After calculate...*

*total is: 0*

*counter is: 0*

*After calculate...*

*total is: 81836821*

*counter is: 14062*

*After calculate...*

*total is: 557997121*

*counter is: 38800*

Burada amacımız bu toplama işlemi bittikten sonra yazdırma , bu durumda wait notify mekanizmasını such şekilde kullanabiliriz.

```
public class WithWaitNotifyTest {  
  
    public static void main(String[] args) {  
  
        Calculate calculateThread = new Calculate();  
  
        calculateThread.start();  
  
        synchronized (calculateThread) {  
  
            try {  
  
                System.out.println("Waiting for calculateThread to complete...");  
  
                calculateThread.wait();  
  
            } catch (InterruptedException e) {  
  
                e.printStackTrace();  
  
            }  
  
            System.out.println("total is: " + calculateThread.total);  
  
            System.out.println("counter is: " + calculateThread.counter);  
  
        }  
  
    }  
  
    class Calculate extends Thread {  
  
        int total;  
  
        int counter;  
  
        @Override  
        public void run() {  
  
            synchronized (this) {  
  
                for (int i = 0; i < 100000; i++) {  
  
                    total += i;  
  
                }  
  
            }  
  
        }  
    }  
}
```

```

        counter = i;
    }
    notify();
}
}
}

```

calculateThread i start metodu ile calistirdikdan sonra , islemini tamamlamasini bekliyoruz. Bunun icin Calculate objesi uzerinden (calculateThrea referans degiskeni) synchronized kod blogu yaziyoruz. wait metodunu calculateThread referans degiskeni uzerinden cagiriyoruz boylelikle bu thread sonlana kadar diger threadlerin calismasini engelliyoruz.(block)

```

synchronized (calculateThread) {
    try {
        System.out.println("Waiting for calculateThread to complete...");
        calculateThread.wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("total is: " + calculateThread.total);
    System.out.println("counter is: " + calculateThread.counter);
}

```

notify metodunu cagirarak , isimizin bittigini bildiriyoruz(notification) . Boylelikle blockladigimiz thread(main threadi) tekrar calismaya devam ediyor.

calculateThread bitene kadar main threadi blocklandigi icin ornegimizi calistirirsak her defasinda ayni ciktiyi elde ederiz.

*Waiting for calculateThread to complete...*

*total is: 704982704*

*counter is: 99999*

notifyAll metodu , tum bekleyen threadler icin bildirim yapacaktir(notification).

Tum threadler uyarilacak ve WAITING durumunda olan threadler RUNNABLE duruma gecektir.

```
class Message {
```

```

private String text;

public Message(String text) {
    this.text = text;
}

public String getText() {
    return text;
}

public void setText(String text) {
    this.text = text;
}

}

class Waiter implements Runnable {

    Message message;

    public Waiter(Message message) {
        this.message = message;
    }

    @Override
    public void run() {
        synchronized (message) {
            try {
                System.out.println("Waiter is waiting for the notifier at " +
Thread.currentThread().getName());
                message.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

        }

        System.out.println("Waiter is done waiting at : " + Thread.currentThread().getName());

        System.out.println("Waiter got the message: " + message.getText());

    }

}

class Notifier implements Runnable {

    Message message;

    public Notifier(Message message) {

        this.message = message;

    }

    @Override

    public void run() {

        System.out.println("Notifier is sleeping for 3 seconds at ");

        try {

            Thread.sleep(3000);

        } catch (InterruptedException e1) {

            e1.printStackTrace();

        }

        synchronized (message) {

            message.setText("Notifier took a nap for 3 seconds");

            System.out.println("Notifier is notifying waiting thread to wake up at ");

            message.notifyAll();

        }

    }

}

public class WaitNotifyExample {

```

```

public static void main(String[] args) {

    Message message = new Message("Hello notify test");

    Waiter waiter = new Waiter(message);

    Thread waiterThread = new Thread(waiter, "waiterThread");

    waiterThread.start();

    Thread waiterThread2 = new Thread(waiter, "waiterThread2");

    waiterThread2.start();

    Thread waiterThread3 = new Thread(waiter, "waiterThread3");

    waiterThread3.start();

    Notifier notifier = new Notifier(message);

    Thread notifierThread = new Thread(notifier, "notifierThread");

    notifierThread.start();

}

}

```

Buradada 3 tane Waiter Threadi olusturduk , 1 tane Notifier Threadi olusturduk. Message sinifi tipindeki objemizi synchronized blogunda kilit olarak kullandik. 3 tane Thread'i beklemeye aldig/block. Notifier sinifinin isi bittiginde Message objesinin kilidini bekleyen bu 3 Thread'e bildirim yaptik.

*Waiter is waiting for the notifier at waiterThread  
 Waiter is waiting for the notifier at waiterThread2  
 Waiter is waiting for the notifier at waiterThread3  
 Notifier is sleeping for 3 seconds at  
 Notifier is notifying waiting thread to wake up at  
 Waiter is done waiting at : waiterThread3  
 Waiter is done waiting at : waiterThread2  
 Waiter is done waiting at : waiterThread  
 Waiter got the message: Notifier took a nap for 3 seconds  
 Waiter got the message: Notifier took a nap for 3 seconds  
 Waiter got the message: Notifier took a nap for 3 seconds*

Burada notifyAll metodu yerine notify metodu kullanırsak , bu 3 Thread'in sadece birtanesine bildirimde bulunur , diger 2 thread waiting durumunda kalır.

*Waiter is waiting for the notifier at waiterThread2  
Waiter is waiting for the notifier at waiterThread3  
Waiter is waiting for the notifier at waiterThread  
Notifier is sleeping for 3 seconds at  
Notifier is notifying waiting thread to wake up at  
Waiter is done waiting at : waiterThread2  
Waiter got the message: Notifier took a nap for 3 seconds*

## Bölüm-10

### Pure Java - 93 - Development - java & javac

[Levent Erguder](#) 06 January 2015 [Java SE](#)

Merhaba Arkadaşlar,

Bu bölümde komut satırında **java** ve **javac** komutlarının terminal/command line kullanımını inceleyeceğiz.

#### **javac komutu ile derleme(compiling with javac)**

**javac** komutu , Compiler/derleyiciyi calistirmak/invoke icin kullanılır. javac komutunun bir çok option/secenegi vardır. Sinavda önemli olan option'lar **-classpath** ve **-d**

#### **javac [options] [source files]**

Bu option'ların neler olduğunu görmek için **-help** option kullanılabilir.

#### **javac –help**

**Usage:** javac

**where** possible options include:

- g**              **Generate** all debugging info
- g:none**        **Generate no** debugging info
- g:{lines,vars,source}**    **Generate** only some debugging info
- nowarn**        **Generate no** warnings
- verbose**       **Output** messages about what the compiler **is** doing
- deprecation**    **Output** source locations **where** deprecated **APIs** are used

```

-classpath      Specify where to find user class files and annotation processors
-cp            Specify where to find user class files and annotation processors
-sourcepath     Specify where to find input source files
-bootclasspath  Override location of bootstrap class files
-extdirs        Override location of installed extensions
-endorseddirs   Override location of endorsed standards path
-proc:{none,only}  Control whether annotation processing and/or compilation is done.
-processor [...] Names of the annotation processors to run; bypasses default discovery process
-processorpath  Specify where to find annotation processors
-d              Specify where to place generated class files
-s              Specify where to place generated source files
-implicit:{none,class}  Specify whether or not to generate class files for implicitly referenced files
-encoding       Specify character encoding used by source files
-source         Provide source compatibility with specified release
-target         Generate class files for specific VM version
-version        Version information
-help           Print a synopsis of standard options
-Akey[=value]    Options to pass to annotation processors
-X              Print a synopsis of nonstandard options
-J              Pass directly to the runtime system
-Werror        Terminate compilation if warnings occur
@              Read options and filenames from file

```

### Compiling with -d

*myProject/source/* dizininde *MyClass.java* adında bir dosya olusturalim. Icerigi su sekilde olsun;

```
public class MyClass { }
```

```
myProject
```

```
|
```

```
|--source
```

```
||
```

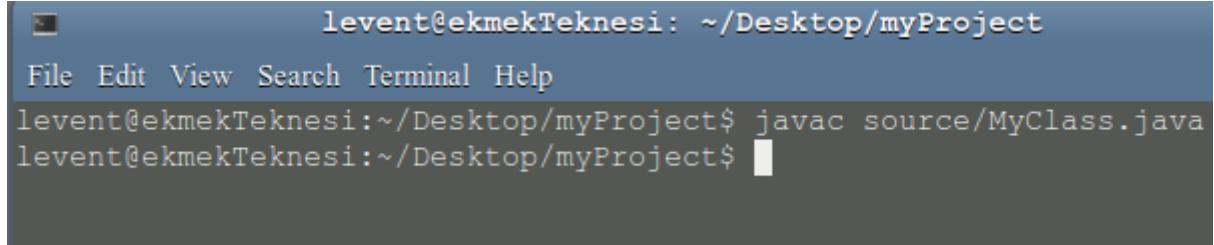
```
||-- MyClass.java
```

```
|
```

```
-- classes
```

```
|  
|--  
myProject dizininde olalim sonrasında komut satirinda su komutu calistiralim;
```

```
javac source/MyClass.java
```

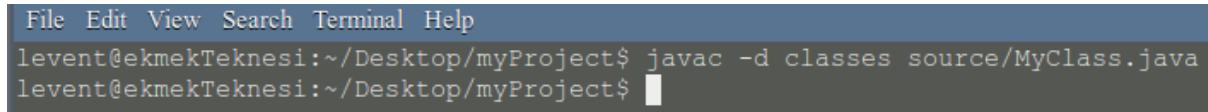


```
levent@ekmekTeknesi: ~/Desktop/myProject  
File Edit View Search Terminal Help  
levent@ekmekTeknesi:~/Desktop/myProject$ javac source/MyClass.java  
levent@ekmekTeknesi:~/Desktop/myProject$
```

*MyClass.java* dosyamizi derledigimizde *MyClass.class* dosyamız varsayılan/default olarak **.java** dosyası ile aynı dizinde olusacaktır. Peki biz farklı bir dizinde **.class** dosyasını oluşturmak istersek ne yapmalıyız? Bu durumda **-d** option’ini kullanabiliriz.

Ornegin *MyClass.java* dosyamızın derlenmiş halini *myProject/classes* dizini altında olmasını istiyoruz. Bu durumda classes dizinini *myProject* dizini altına oluşturralım.

```
javac -d classes source/MyClass.java
```



```
File Edit View Search Terminal Help  
levent@ekmekTeknesi:~/Desktop/myProject$ javac -d classes source/MyClass.java  
levent@ekmekTeknesi:~/Desktop/myProject$
```

*MyClass.class* dosyamız *myProject/classes* dizini altında olustu.

*MyClass.java* dosyamızda **package** tanımı yer almiyordu. Bu dosyamiza *com.injavawetrust* package tanimini ekleyelim ve paket tanimina uygun olarak *com/injavawetrust* dizini altına koyalim.

```
package com.injavawetrust;  
  
public class MyClass { }  
  
myProject  
|  
|--source  
| |  
| |--com  
| | |  
| | |--injavawetrust
```

```
| |  
| | - MyClass.java  
| |  
| --classes
```

*MyClass.java dosyamizin derlenmis halinin(.class) , myProject/classes altında olmasini istiyoruz.  
myProject dizininden su komutu yazabiliriz;*

```
javac -d classes source/com/injavawetrust/MyClass.java
```

*source dizininden su komutu yazabiliriz;*

```
javac -d ..//classes com/injavawetrust/MyClass.java
```

*..(iki nokta) bir ust dizini temsil eder. source dizininin ust dizinine git oradaki classes klasorunu bul anlamindadir.*

*Bu komutulari calistirdigimizda classes dizini altında com/injavawetrust alt dizinleri olusacaktir ve injavawetrust klasorumuz altında MyClass.class dosyamiz yer alacaktir.*

*Burada alt klasorleri kendisi olusturacaktir ,fakat ornegin olmayan bir ana dizin kullanmaya calisirsak kendisi olusturmayacaktir.*

```
javac -d ..//classes2 com/injavawetrust/MyClass.java
```

*javac: directory not found: classes2*

### **the java command**

*javac komutu ile .java uzantili dosyamizi derleyerek .class uzantili dosya haline getiririz. java komutu ile de bu derledigimiz .class uzantili dosyamizi calistirabiliriz.*

*Genel format ;*

**java [options] class [args]**

```
public class HelloJava{  
  
    public static void main(String[] args) {  
        System.out.println("Hello Java World.");  
    }  
}
```

Bu basit ornegimizi javac komutu ile derleyelim ;

```
javac HelloJava.java
```

Derledikten sonra HelloJava.class dosyamizi calistiralim;  
Burada dikkat edelim calistirmak icin javac komutu degil java komutunu kullaniyoruz. Ikinci olarak  
derlerken dosya ismini .java olarak belirtmemiz gerekirken , calistirirken .class uzantisi olmamali.

```
java HelloJava
```

Sinav icin bilmemiz gereken options ; **classpath(-cp)** ve **-D** dir . -D yardimi ile yeni bir system  
property'si olusturup kullanabiliriz.

```
import java.util.*;  
  
public class TestProps {  
    public static void main(String[] args) {  
        Properties p = System.getProperties();  
        String propertyValue = p.getProperty("myProp");  
        System.out.println("propertyValue:" + propertyValue);  
    }  
}
```

TestProps sinifimizi derleyelim.

```
javac TestProps.java
```

TestProps sinifina dikkat edecek olursak myProp isminde bir property'i degerini getProperty methodu  
ile almakta daha sonrasinde console basmaktadır. -D yardimi ile bu property degerini verebiliriz.

```
java -DmyProp=Levent TestProps
```

Ekran ciktisi :

*propertyValue:Levent*

**-D** kullanmazsa bu durumda ;

```
java TestProps
```

*propertyValue:null*

Eger bosluk kullanilacaksa bu durumda cift tırnak arasında olmalıdır

```
java -DmyProp="be an oracle man, import java.*" TestProps
```

Property ismi -D'den hemen sonra olmalıdır, arada bosluk olmamalıdır.

```
java -D myProp=Levent TestProps
```

= (esittir) den önce ve sonra bosluk olmamalıdır.

```
java -DmyProp= Levent TestProps
```

```
java -DmyProp =Levent TestProps
```

```
java -DmyProp = Levent TestProps
```

## Command-Line Arguments

```
public class CommandArgs {  
    public static void main(String[] args) {  
  
        for (String arg : args) {  
            System.out.println(arg);  
        }  
    }  
}
```

Dosyamizi derleyelim ;

```
javac CommandArgs.java
```

Kodumuzu calistirirken command-line argument verelim;

```
java CommandArgs be an oracle man , import java.*;
```

args dizine 0.indexten baslayarak  
args[0] = be  
args[1] = an  
args[2] = oracle  
args[3] = man  
args[4] = ,  
args[4] = import  
args[4] = java.\*;  
seklinde atanacaktir. Dolayisiyla dongu ile yazdirdigimizda ;

*be  
an  
oracle  
man  
,  
import  
java.\*;*

Bosluga gore tek tek arguman olarak degerlendirecek , bir butun olarak yapmak icin yine cift tırnak kullanabiliriz.

```
java CommandArgs "be an oracle man , import java.*;"
```

*be an oracle man , import java.\*;*

main metodunda String [] args kullanabildigimiz gibi var-args yapisini da kullanabiliriz.

```
public class CommandArgs {  
    public static void main(String ...args) {  
  
        for (String arg : args) {  
            System.out.println(arg);  
        }  
    }  
}
```

}

## Pure Java - 94 - Development - Using Classpaths

[Levent Erguder](#) 07 January 2015 [Java SE](#)

Merhaba Arkadaslar

Bu bolumde **classpath** kavramini ve package yapisi ile kullanimini inceleyecegiz.

Bir onceki yazida javac ve java komutlarinin komut satirinda kullandik. Orneklerimizde java.lang kutuphanesini ve java.util kutuphanesini kullandik. Aslinda Java arka planda bizim icin bu paketleri buldu ve biz ilgili sinifların yerini belirtmeden kodumuz sorunsuzca derlendi ve calisti.

javac ve java komutlarinin searching konusunda ayni algoritmlari ve yaklasimlari vardir.

- Ilk olarak Core Java classlarinin bulundugu dizine/directory bakarlar.
- Ilgili sinif bulundugu zaman aramayı sonlandırırlar.
- Ikinci olarak baktıkları yer classpath'lerle belirtilmiş dizinlerdir.
- classpath , ‘sinifları bulmak için arama yapılacak dizin’ olarak tarif edilebilir. classpath'ler 2 şekilde tanımlanabilir.
  1. Operating System Environment variable
  2. Command-line option'i olarak

Command-line da tanımlanan classpath bilgisi Operating System Environment olarak tanımlanan classpath bilgisini override eder , fakat Command line sadece ilgili calistirma/cagrisim (invocation) icin gecerlidir.

### classpath declaration

classpath directory/dizin bilgisini icerir, bu bilgiyi delimiter(ayirici) ile yazabilirim. Unix-based isletim sistemlerinde dizinler icin / , delimiter olarak : kullanilir.

-classpath /com/foo/acct:/com/foo

Burada 2 tane dizin/directory belirtiyoruz.

/com/foo/acct ve /com/foo

Bu dizinler/directory / ile basladigi icin **absolute path** olarak isimlendirilir.

Burada onemli nokta; bir subdirectory/alt dizin belirttigimizde ust dizini belirtmis olmayiz.

Yani /com/foo/acct ve /com/foo dizinleri arama yapılacak dizin olarak kullanılırken /com dizininde arama yapılmayacaktır.

Sinavda dikkat edilmesi gereken bir nokta Unix convention'a uyulmasıdır. Yani Windows'ta path \ şeklinde olurken Unix – based olarak / kullanılmaktadır. Bununla birlikte delimiter da windows'ta ; olurken unix te : şeklindedir.

Arama/Searching yapıldığında **java** ve **javac** komutları current directory varsayılan olarak aranmaz.

Bunun icin ozellikle belirtmemiz gereklidir. nokta(.) karakterini kullanarak mevcut/current directory arama yapabiliyoruz.

`-classpath /com/foo/acct:/com/foo:`

Bu classpath bilgisi bir onceki bilgi ile benzerdir fakat tek farki en sona eklenen .(nokta) path bilgisidir. Boylelikle arama isleminde current directory de kullanilacak.

Bir diger onemli nokta olarak arama islemi **soldan saga** dogru olur.

`-classpath /com:/foo:`

`-classpath .:/foo:/com`

Bu iki classpath ayni anlamda gelmez. Birincisinde ilk olarak /com dizininden baslanarak arama yapilir. Ikincide ise once . (nokta) yani mevcut directoryden aranmaya baslanir.

Ornegin Test.class sinifimizin /com ve .(nokta) da oldugunu varsayalim. Birinci classpath /com da olan sinifi bulacak ve aramayı sonlandiracaktir. Ikinci classpath mevcut dizindeki Test.class dosyasini bulacak ve aramayı sonlandiracaktir.

Son olarak -classpath yerine -cp de kullanilabilir fakat -cp tutarsizliklara neden olabilir(inconsistent). Bir cok JVM destekler fakat -cp nin bir garantisi yoktur.

### Packages & Searching

Isin isine package tanimlari ve classpath girdigi zaman sinavda dikkat edilmesi gereken noktalar ortaya cikmatakadir.

Simdi `com/foo` dizininde MyClass.java dosyasi olusturalim ;

```
package com.foo;

public class MyClass {
    public void hi() {
    }
}
```

Bu basit kod orneginde, MyClass com.foo package/paketinin bir uyesidir diyoruz. Bunun anlamı sinifimizin full qualified ismi com.foo.MyClass seklindedir.

Bir sinif package icerisindeyse package ismi full qualified isminin bir parcasidir ve bu isim bolunemez atomic bir ozellige sahiptir.

Bir baska sinifta bu sinifi import edelim;

```
import com.foo.MyClass;
import com.foo.*;

public class Another {
```

```

void go() {
    MyClass m1 = new MyClass(); // alias name
    com.foo.MyClass m2 = new com.foo.MyClass(); // full name
    m1.hi();
    m2.hi();
}

```

### Relative & Absolute Paths

**classpath**, bir yada daha fazla path bilgisinden olusur. classpath'te yer alan her bir path **absolute** ya da **relative patholabilir**.

Unix'te absolute path / (forward slash) ile baslar. (root) Windows'ta ise c:\ olarak baslar. Sinavda unix based baz alınmaktadır.

/ (forward slash) sistemin root dizinini gösterir. Soz konusu absolute path olduğunda current directory onemli degildir , absolute path her zaman root'u baz aldiği için hangi dizinde olduğumuzun bir onemi yoktur.

**relative path** ise / (forward slash) ile baslamaz.

```

/ (root)
|
|-- dirA
|
|-- dirB
|
|--dirC

```

-cp dirB:dirB/dirC

dirB ve dirB/dirC pathleri / ile baslamadığı için relative path'dır. Bu 2 relative path sadece current directory dirA olduğunda bir anlam kazanacaktır.

current directory dirA olduğunda dirB ve dirC dizinleri search edilir. dirA search edilmez, cunku classpath de .(nokta) karakteri yer almamaktadır.

current directory root olsaydı dirB ya da dirC search edilmezdi. Cunku dirB root'un direkt olarak alt dizini degildir.

current directory dirB olsaydı yine dirB ya da dirC search edilmezdi. cunku dirB dizini dirB isimli bir alt dizine sahip degil !

dirB dizini altında dirB ve dirB/dirC şeklinde klasör olsaydı bu durumda current directory üstteki dirB olsaydı

```
-cp dirB:dirB/dirC  
alt dirB ve dirC yi search edecekti.
```

Bir diger ornek olarak

```
/ (root)  
|  
|--dirA  
|  
|-- dirB  
|  
|--dirC
```

```
-cp /dirB:/dirA/dirB/dirC  
current directory dirA olsaydi hangi dizinler search edilecekti ? ya da root olsaydi ya da dirB ?  
dikkat edelim classpath'te verilen pathler relative path degildir ! bu pathler / ile basladigi icin absolute path'tir.  
Soz konusu absolute path oldugunda hangi dizinde oldugumuzun bir onemi yoktur. absolute path'i bir nevi root'a gore relative olarak dusunebiliriz. Cunku mevcut dizin onemsiz olmakta ve root'u baz almaktadir.  
/dirB gecersiz bir dizin olacaktir ve dirB dizini search edilmeyecektir. Cunku dirB root'un direkt olarak alt dizini degildir.
```

/dirA/dirB/dirC gecerli bir dizin olacaktir ve dirC dizini search edilecektir.  
Burada nokta(.) da kullanilmadigi icin mevcut dizin de search edilmeyecektir.

myProject/src dizininde

A sinifi

```
public class A {
```

B sinifi

```
public class B extends A {
```

src dizininde

```
javac B.java
```

myProject/src/test dizininde

A sinifi

```
package test;
```

```
public class A { }
```

B sinifi

```
package test;
```

```
public class B extends A { }
```

src dizininde

```
javac B.java
```

myProject dizininde

```
javac src/test/B.java
```

hata verir!

```
src\test\B.java:2: cannot find symbol
```

symbol: class A

```
public class B extends A {} →
```

test.A.java sinifini bulmasi icin classpath kullanmaliyiz

```
javac -cp src src/test/B.java
```

src/test2/pack1

A sinifi

```
package test2.pack1;
```

```
public class A {}
```

```
src/test2/pack2
```

```
B sinifi
```

```
package test2.pack2;
```

```
import test2.pack1.A;
```

```
public class B extends A {}
```

```
src dizininde;
```

```
javac test2/pack2/B.java
```

```
myProject dizininde
```

```
javac -cp src src/test2/pack2/B.java
```

```
myProject/foo dizininde
```

```
javac -cp ..../src ..../src/test2/pack2/B.java
```

## Pure Java - 95 - Development - Jar Files & Searching

[Levent Erguder](#) 07 January 2015 [Java SE](#)

Merhaba Arkadaslar,

Bu bolumde **jar** konusunu inceleyecegiz.

Java developer'lar **jar** dosyalarina yabanci degildir. Ihtiyacimiz olan siniflari projemize eklemek icin **jar** dosyalarini kullaniriz. En basitinden bir veritabanina baglanmamiz gerektiginde ornegin mysql icin com.mysql.jdbc\_XXX.jar i projemize ekleriz , boylelikle bu jar içerisinde yer alan class dosyalarini projemizde kullanabiliriz.

**JAR , Java Archive** anlamina gelmektedir. **Jar** dosyalari sikistirilmis/compress data'dir ve zip formati uzerine kurulmustur. Bir jar dosyasini winrar gibi bir programla extract edebiliriz.

Asagidaki dizin sekline uygun class dosyalarimiz olsun. Bunun icin ilgili .java uzantili dosyalari compile edip .class dosyasini olusturabiliriz. Ilgili dizinde ek olarak txt dosyasi yer almaktadir. Jar dosyalarinda sadece .class dosyasi olmak zorunda degildir.

```
package com.injavawetrust;
```

```
public class HelloJava {}
```

```
package com.util;
```

```
public class Utility{}
```

```
jarTest
```

```
|
```

```
|---src
```

```
|
```

```
|---com
```

```
| |
```

```
| |---injavawetrust
```

```
| | |
```

```
| | |---HelloJava.class
```

```
|
```

```
|---util
```

```
|
```

```
|---Utility.class
```

```
|
```

```
|---readMe.txt
```

command line da jar komutu yardimi ile jar dosyasi olusturabiliriz.

command line da ;

```
jar -h
```

jarTest/src dizininde ;

```
jar -cf myjar.jar com
```

komutu ile jar dosyamizi olusturabiliriz.

-c create a new archive / yeni bir jar dosyasi olusturmak icin kullanilir.

-f specify archive file name / jar dosyasina isim vermek icin kullanilir.

jarTest/src dizinimizde myjar.jar dosyamiz olustu.

myjar.jar dosyamizi winrar gibi bir programla extract edersek.

```
myjar
|
+--com
|
|   +--injavawetrust
|   |
|   |   +--HelloJava.class
|   |
|   +--util
|   |
|   |   +--Utility.class
|   |
|   |--readMe.txt
|   |
|   |
|   +--META-INF
|   |
|   |
|   +--MANIFEST.MF
```

jar dosyamizin extract hali , jar olmadan oncekine benzemektedir. Ek olarak META-INF klasoru ve bu klasor altında MANIFEST.MF dosyasi olustu. jar komutu otomatik olarak bu klasoru ve dosyayı oluşturur.

jar dosyalarini classpath kullanarak aramak ayni class dosyalarini bulmak gibidir. classpath'e dizin bilgisi veriyorduk burada ise jar dosyamizin ismini veriyoruz.

UseJar.java dosyamizi olusturalim bu sinifimiz com.util.Utility sinifina ihtiyac duymaktadir.

```
import com.util.Utility;
public class UseJar extends Utility {
}
```

jarTest dizini altında bu UseJar sinifimizi olusturalim.

```
javac -classpath src/myjar.jar UseJar.java
```

UseJar.java dosyamizi compile edebilmek icin ,classpath'te myjar.jar dosyamizi kullandik cunku ihtiyac duyduğumuz com.util.Utility class dosyasi myjar.jar dosyasi içerisinde mevcuttur.

jarTest/src dizini altında myjar.jar dosyamizla birlikte com klasoru de yer almaktadir.

jarTest dizini altında classpath'e ilgili dizin bilgisini de verebiliriz;

```
javac -classpath src UseJar.java
```

## Pure Java – 96 – Development – Static Imports

[Levent Erguder](#) 07 January 2015 [Java SE](#)

Merhaba Arkadaslar,

Bu yazı OCP notlarimin son yazisi olacak. Bu boludme **static import** konusunu inceleyecegiz.

**static** anahtar kelimesinin suana kadar 4 yerde kullandik ;

- **static variable**
- **static methods**
- **static initialize block**
- **static nested class**

Bununla birlikte **static** anahtar kelimesini **import** anahtar kelimesi ile birlikte de kullanabiliriz. Static import'lar bir sınıfın static member üyelerini import etmek için kullanılabilir.

**static import** olmadan ;

```
public class TestStatic {  
    public static void main(String[] args) {  
        System.out.println(Integer.MAX_VALUE);  
        System.out.println(Integer.toHexString(42));  
    }  
}
```

**static import** ile ;

```
import static java.lang.System.out;  
import static java.lang.Integer.*;
```

```
public class TestStaticImport {
```

```

public static void main(String[] args) {
    out.println(MAX_VALUE);
    out.println(MIN_VALUE);
    out.print(Integer.MAX_VALUE);
    out.println(toHexString(42));
}
}

```

Bu ozellige “**static import**” denilmesine ragmen syntax olarak **import static** seklinde olmalidir. Bu anahtar kelimelerin yer degistirmesi compile error/derleme hatasina neden olur.

Burada *java.lang.System* sinifinda yer alan *out* static variable’i import ettik ;

```
public final static PrintStream out = nullPrintStream();
```

Eger birden fazla static member kullanacaksak bu durumda \* **wilcard’ini** kullanabiliriz.

*java.lang.Integer* sinifinda yer alan tum static member’lari import ettik dolayisiyla *java.lang.Integer* sinifinda yer alan static degiken olan *MAX\_VALUE* ve *MIN\_VALUE* degiskenlerini direkt olarak kullanabiliriz.

```

public static final int MAX_VALUE = 0x7fffffff;
public static final int MIN_VALUE = 0x80000000;

```

static variable kullanabildigimiz gibi direkt olarak static method da kullanabiliriz.

```

public static String toHexString(int i) {
    return toUnsignedString(i, 4);
}

```

Sinavda dikkat etmemiz gereken ;

- Unutmayalim syntax olarak “**import static**” seklinde olacak “**static import**” derleme hatasi verir.
- ambiguous/belirsiz/kararsiz kulanimlara dikkat edelim.

Ornegin *java.lang.Integer* ve *java.lang.Long* siniflarinda *MAX\_VALUE* adinda static degiskenler mevcuttur.

```

public static final int MAX_VALUE = 0x7fffffff;
public static final long MAX_VALUE = 0x7fffffffffffffL;

```

Bu durumda hem Integer hem Long sinifini **static import** yapisinda kullandigimizda ortaya ambiguous/belirsiz/kararsiz durum cikacaktir. Bu durumda kodumuz derleme hatasi verir.

```
import static java.lang.Integer.*;
```

```

import static java.lang.Long.*;
import static java.lang.System.*;

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

        // out.print(MAX_VALUE);
        // The field MAX_VALUE is ambiguous
        // compile error/derleme hatasi
    }
}

```

Eger sadece MAX\_VALUE degiskenini static import edecek olursak bu durumda import isleminde ambiguous problemi ortaya cikacak ve compile error/derleme hatasi verecektir;

```

import static java.lang.Integer.MAX_VALUE;
import static java.lang.Long.MAX_VALUE;
//The import java.lang.Long.MAX_VALUE collides with another import statement

```

- static import ile ; static variable , constant(static final variable) ya da static method'lari kullanabiliriz.
- static import ile instance degiskenlere ve instance methodlara erisim saglanamaz.

```

import static java.lang.Integer.*;

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

        //System.out.println(byteValue());
        //compile error
    }
}

```

Yazimi burada sonlandiriyorum.  
 Herkese Bol Javali Gunler dilerim.  
 Be an oracle man , import java.\*;  
 Levent Erguder  
 OCP, Java SE 6 Programmer  
 OCE, Java EE 6 Web Component Developer

# Online Java Egitim

Merhaba arkadaslar

Bu yazida Online Java Egitimleri hakkında bilgilere yer verecegim. Bu egitimleri 2014 Nisan ayindan beri vermekteyim.

Suan icin 3 farkli kapsamda Java egitimi mevcuttur.

1. Oracle Java SE sertifikayona yonelik egitim
2. Servlet&JSP / Oracle Web Component sertifikasyona yonelik egitim
3. More Java EE egitimi

## Oracle Java SE sertifikayona yonelik egitim

Amac : Bu egitim Java dunyasina giris yapabilmemiz icin gerekli olan Java SE konularini icermektedir.

Yeterli Java SE ogrendikten sonra Mobil tarafina ya da Java EE uzerine ilerleyebilirsiniz.

Egitim kapsamı [Oracle Certified Professional, Java SE 6 Programmer](#) sinavini baz almaktadir.

Ilgili Oracle sinavi hakkında bilgi icin ; [Oracle Certified Professional, Java SE 6 Programmer Sinavi Hakkında](#)

- **Bu egitime katilmak icin on şart nedir?**

Hic programlama bilgisi olmayan kisilere bu egitimi onermiyorum. Onceinde bir miktar C C# java gibi dillerde biraz deneyim arıyorum. Buna ragmen ciddi bir kararlilik ve istek varsa egitime dahil olunabilir.

- **Hangi detayda ve derinlikte oluyor, bastan mi aliyoruz ?**

Oracle'in Java SE sertifikasını baz alarak anlatıyorum. Konular temelden fakat derin bir sekilde olacaktır. Ilgili kitabı inceleyebilirsiniz. Yeri geldikce Java 7 ve Java 8 deki konulara da deginiyorum. Bu kitabı bitiriyoruz.

<http://www.amazon.com/SCJP-Certified-Programme.../.../0071591060>

- **Egitim online mi olmaktadır, egitimlerin kaydi alinıyor mu ?**

Egitimler online olmaktadır.

Egitimlerin kaydi alınmaktadır.

- **Egitimin suresi ve gunu nedir ? Kac hafta surmektedir ?**

Egitimler 14-15 hafta kadar surmektedir.

Her egitim grubu icin farkli egitim gunleri olmaktadır.

Pazartesi aksamları 21.00 – 23.45 arasında 45-50 dakkalik 3 blok ders

- **Egitimler nasil isleniyor , hangi araclar kullanıyoruz?**

Egitimler icin Teamviewer ve/veya Hangout kullanıyoruz.

- **Ornek bir Java SE egitim videosu kaynagi mevcut mu ?**

Ornek bir Java SE dersinden 1.hafta 1.ders

<https://drive.google.com/folderview...>

- **6) Yetkinlik**

Java SE egitimini bir çok arkadaşla beraber bitirme sansimiz oldu. Dileyen olursa geri bildirimler icin bu

egitimlere katilan arkadaslara yonlendirme yapabilmirim.

<http://www.injavawetrust.com/oracle-certified-professional.../>

- **ucret ?**

ogrenci kisi basi 500TL toplam

calisan 600TL toplam

4 taksit olarak aliyorum.

Iban paylasiyorum.

Eft/havale olarak aliyorum.

Maddi olarak problem varsa daha fazla taksit yapabilmirim.

## Servlet&JSP / Oracle Web Component sertifikasyonuna yonetik egitim

**Amac : Bu egitim ile Java EE dunyasina giris yapabilsiniz. Sektorel olarak Java SE tek basina yeterli olmayacaktir. Java kariyeri isteyen arkadaslar yeterli bir Java SE bilgisinden sonra bu egitime dahil olabilir.**

- **Katilabilmek icin on şart nedir?**

Kendini rahatca ifade edebilecek seviyede yeterli Java SE bilgisi

- **Egitim online mi olmaktadır, egitimlerin kaydi alınıyor mu ?**

Egitimler online olmaktadır.

Egitimlerin kaydi alınmaktadır.

- **Egitimin suresi ve gunu nedir ? Kac hafta surmektedir ?**

Egitimler Cumartesi ya da Pazar gunleri 10.00 – 13.00 arasında olmaktadır.

Haftalık egitim 45-55 dakkalik 3 dersten olusmaktadır.

Egitim 8 hafta surmektedir.

- **Egitimin konusu nedir ?**

temel kavramlar

servlet&servlet container

tomcat

servlet lifecycle

request response

attribute

listener

session

jsp

jsp implicit objects

jsp attribute

jsp action

expression language

inclusion

jstl

taglar

- **Egitimler nasil isleniyor , hangi araci kullaniyoruz?**  
Egitimler icin Teamviewer ve/veya Hangout kullaniyoruz.  
Eclipse ve Apache Tomcat kullaniyoruz.
- **Hangi kaynaklardan yararlaniyoruz ?**  
Kendi notlarim uzerinden anlatiyorum.  
<http://www.amazon.com/Head-First-Servlets-JSP-.../.../0596516681>  
<http://www.amazon.com/Scwcd-Exam-Study-Kit-Cer.../.../1932394389>  
<http://download.oracle.com/otnd.../.../servlet-3.0-fr-oth-JSpec/>
- **Ornek bir egitim videosu kaynagi mevcut mu ?**  
1.haftanin ilk bolumunu izleyebilirsiniz.  
<https://drive.google.com/.../0ByTgFvVnNySNnVNRTJtYTN2T2M/view>
- **Yetkinlik**  
<http://www.injavawetrust.com/oracle-certified-expert-java-.../>
- **ucret ?**  
ogrenci kisi basi 200TL toplam  
calisan 300TL toplam  
2 taksit olarak alıyorum.  
Iban paylasıyorum.Eft/havale olarak alıyorum.  
Maddi olarak problem varsa daha fazla taksit yapabilirim.

## More Java EE egitimi

Bu egitim henuz hic yapilmadi. Servlet&JSP egitimine katilan arkadasların talebi doğrultusunda bu egitim planlandi. 10 Ekim 2015 te baslayacak sekilde planliyorum.

**Amac : Java kariyeri isteyen arkadaslara yönelik olan bu egitimde sektörde olarak kullanılan Maven JSF Hibernate/Eclipselink Spring Spring MVC konularına giriş yapılacak ve baslangic-orta duzeyde olacak sekilde ilerlenecektir.**

- **Katilmek icin on şart nedir?**  
Kendini rahatca ifade edebilecek seviyede yeterli Java SE bilgisi  
Servlet&JSP , Apache Tomcat bilgisi.
- **Egitim online mi olmaktadır, egitimlerin kaydi alınıyor mu ?**  
Egitimler online olmaktadır.  
Egitimlerin kaydi alınmaktadır.
- **Egitimin suresi ve gunu nedir ? Kac hafta surmektedir ?**  
Egitimler Cumartesi ya da Pazar gunleri 10.00 – 13.00 arasında olmaktadır.  
Haftalık egitim 45-55 dakikalik 3 dersten olusmaktadır.  
Egitim 8-16 hafta olarak planliyorum.
- **Egitimin konusu nedir ?**  
Egitim konuları baslangic orta duzeyde ortaya karisik ;  
Maven  
JSF  
JPA/Hibernate  
Spring Core  
Spring MVC

- **ucret ?**

ogrenci kisi basi aylik 125TL

calisan arkadaslar kisi basi 150TL

Iban paylasiyorum.Eft/havale olarak aliyorum.

Maddi olarak problem varsa daha fazla taksit yapabilirim.

Yukaridaki aciklamalar yeterli olmadigi , kafaniza takilan durumlarda iletisim icin ;

erguder.levent@gmail.com

0555 588 48 65

**-SON-**

---

**AZİZ ÇİFTÇİ**

**<https://azizfarmer.wordpress.com/about/>**

