Java ile Nesne Merkezli Programlamaya Giriş

9. Bölüm
Tekrar Kullanım ve Miras

Akın Kaldıroğlu

www.javaturk.org

Aralık 2016

Küçük Ama Önemli Bir Konu

- Bu dosya ve beraberindeki tüm, dosya, kod, vb. eğitim malzemelerinin tüm hakları **Selsoft Yazılım, Danışmanlık, Eğitim ve Tic. Ltd. Şti**.'ne aittir.
- Bu eğitim malzemelerini kişisel bilgilenme ve gelişiminiz amacıyla kullanabilirsiniz ve isteyenleri **http://www.selsoft.academy** adresine yönlendirip, bu malzemelerin en güncel hallerini almalarını sağlayabilirsiniz.
- Yukarıda bahsedilen amaç dışında, bu eğitim malzemelerinin, ticari olsun/olmasın herhangi bir şekilde, toplu bir eğitim faaliyetinde kullanılması, bu amaca yönelik olsun/olmasın basılması, dağıtılması, gerçek ya da sanal/Internet ortamlarında yayınlanması yasaktır. Böyle bir ihtiyaç halinde lütfen benimle, **akin.kaldiroglu@selsoft.academy** adresinden iletişime geçin.
- Bu ve benzeri eğitim malzemelerine katkıda bulunmak ya da düzeltme ve eleştirilerinizi bana iletmek isterseniz çok sevinirim.
- Bol Java'lı günler dilerim.

Gündem

- Bu bölümde, nesne-merkezli programlamada tekrar kullanım (reusability) ele alınacaktır.
- Tekrar kullanımın en temel iki hali olan bileşim (composition) ve kalıtım ya da miras (inheritance) işlenecektir.
- Overriding mekanizması işlenecektir.
- Object sınıfı ve metotları ele alınacaktır.

Tekrar Kullanım (Reusability)

Tekrar Kullanım (Reusability)

- **Tekrar kullanım** (**reusability**), var olan yazılım yapılarından yararlanarak, lego bloklarını kullanır gibi, yeni yazılım sistemleri geliştirmektir.
  - Tekrar kullanım, Yazılım Mühendisliği'nin nirvanasıdır.
- Ama yazılımların, soyut, aşırı karmaşık ve değişime zorunlu doğası, bir yazılım yapısının, kendisi için geliştirildiği sistemden başka bir yerde kullanılabilmesini son derece zorlaştırmaktadır.
- Yine de nesne merkezli diller, en temel seviyede tekrar kullanımı amaçlayan mekanizmalara sahiptirler.

Farklı Seviyelerde Tekrar Kullanım

- Yazılımda, çok farklı şeyler tekrar kullanıma konu olabilir:
  - Metotlar, tekrar kullanımın en basit ve sık uygulandığı yapılardır.
  - Sınıfların tekrar kullanımı daha geniş olmakla birlikte daha zordur
- Daha karmaşık olan bileşenler (components) ve çerçeveler (frameworks) ile tekrar kullanım çok daha yüksek seviyede elde edilir ama başarılması bir o kadar da zordur.
- Ayrıca iş süreçleri analizi, mimari yaklaşımlar, tasarım, test yapıları vs. hep tekrar kullanıma konudurlar.
  - Tasarım kalıpları, çok tipik tasarım tekrar kullanımına örnektir.

Yeni Bir Sınıf

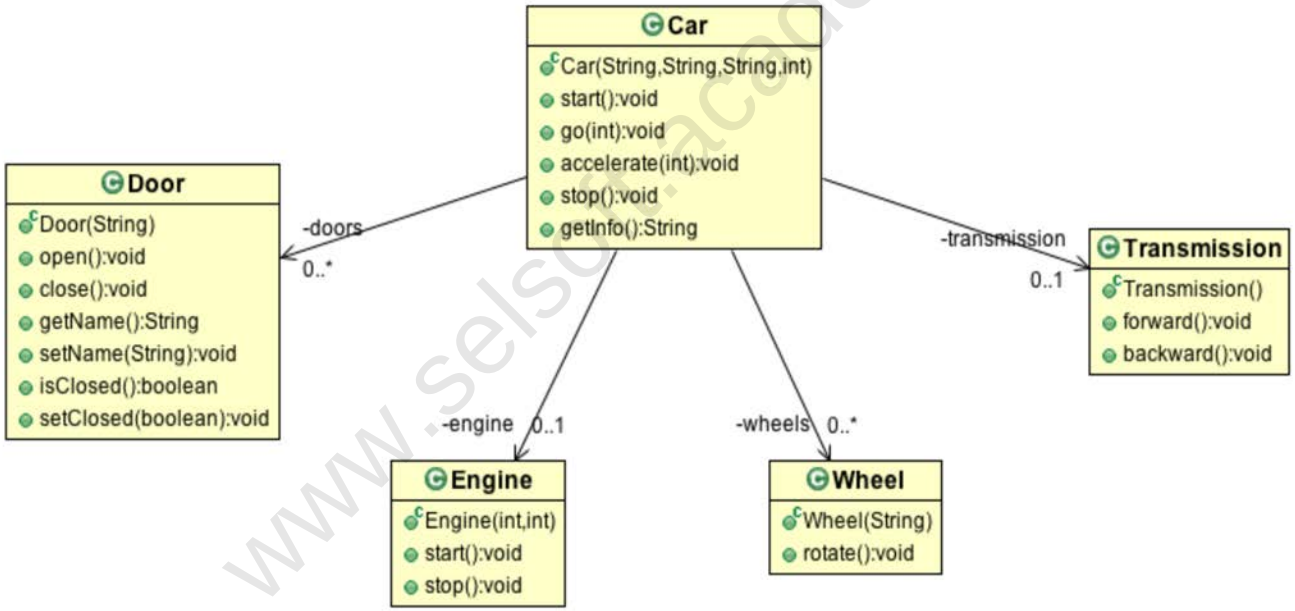
- Yeni bir sınıfa ihtiyaç duyulduğunda alternatifler şunlardır:
  - Pazardan bir tane satın almak,
  - Sıfırdan yazmak,
  - Var olan sınıflardan yararlanarak bileşik (composite) bir sınıf oluşturmak,
  - Var olan bir sınıftan devralarak bir alt sınıf (sub-class) oluşturmak.
- İdeal ve aynı zamanda en az muhtemel olan ilk seçenektir.
- İlk başta kolay gözüküp de uzun vadede en sıkıntılı olan ikinci seçenektir.
- 3. ve 4. seçenekler ise sırasıyla tekrar kullanımın, bileşik nesne oluşturma (composition) ve kalıtım (inheritance) şekilleridir.

Bileşim (Composition)

Bileşik Nesneler

- **Nesne birleştirme** (**object composition**), birden fazla nesneyi bir araya getirilerek daha karmaşık nesneler oluşturmaktır.
  - **Bileşik nesneler** (**composite object**) genel olarak, başka sınıfların nesnelerini, nesne değişkeni olarak kendinde barındıran yapılardır.
- Bu ilişki **sahip olma** (**has-a**) ilişkisi olarak ifade edilir.
  - Unutulmamalıdır ki sahip olma, referanslar üzerinden gerçekleşmektedir.
- Nesne birleştirme ile nesneler arasında bir **ilişki** (**association**) ve aynı zamanda bir **bağımlılık** (**coupling**) oluşturulur.
  - Bu durum, nesneler arasındaki en yaygın ilişki kalıbıdır.

Car as A Composite Object



```
public class Car {
   private String make;
   private String model;
   private String year;
   private int distance;
   private int speed;
   private Engine engine;
   private Transmission tx;
   private Door[] doors;
   private Wheel[] wheels;

   public Car(..., int doorCount,...){
    engine = new Engine(...);
    tx = new Transmission(...);
    doors = new Door[doorCount];
    wheels = new Wheel[4];
    ...
   }

   public void start(){
     engine.start();
   }
  ...
}
```

```
public class Engine{
   private String make;
   private int cc;
   private int horsePower;
   private int rpm;
   ...
}
```

```
public class Wheel{
   private int size;
   ...
}
```

```
public class Door {
   private boolean closed;
   ...
}
```

```
public class Transmission {
   private boolean manual;
   ...
}
```

CarTest.java

Bileşik Nesne ve Bileşenler

- Bileşik nesneler, bileşenlerinden hizmet alırlar:
  - Bileşik nesne, kendisinden isteneni, bileşenlerinden hizmet alara yerine getirir.
  - Buna, **yönlendirme** (**delegation**) denir.

- Bileşik nesnelerin arayüzleri, bileşenlerinden bağımsızdır.
  - Bileşik nesne, bileşenlerinin arayüzlerini toplayıp, farklı bir arayüzle kulanıma açar.

Bileşenlerin Yaratılması

- Bileşik nesnelerde, bileşenlerin ne zaman oluşturulacağı karar verilmesi gereken bir konudur:
  - Tanıtıldığı yerde ya da kuruculardа oluşturulabilir.
  - Bir başka yerde oluşturulup bileşik nesneye geçilebilir.
- İlk durum genel olarak daha sıkı (composition), ikinci durum ise daha gevşek bir ilişkiyi (aggregation) ifade edebilir.

Uygulama

- Selesi, ön ve arka tekerleği ile pedal takımı olan bir bisiklet sınıfı oluşturun.
  - Bisiklet sınıfı, belirtilen sınıflardan nesne değişkenlerine sahip olmalı ve kendisinden istenen hizmetlere, nesne değişkenleri yardımıyla cevap vermelidir.

Miras - Kalıtım (Inheritance)

Miras I

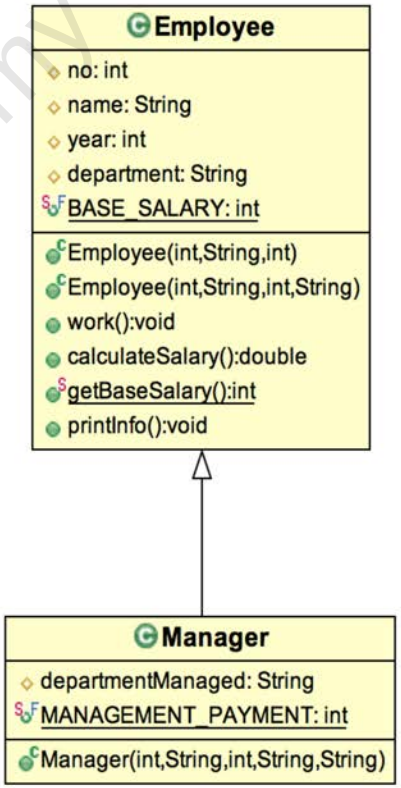
- **Miras** ya da **kalıtım** (**inheritance**), aralarında yapısal benzerlik bulunan nesneleri ifade etmekte kullanılan, en yaygın ikinci tekrar kullanım kalıbıdır.
- Miras, **is-a** (**olma**) ya da **is-like-a** (**gibi olma**) ilişkisidir.
- Kendisinden miras alınan sınıfa **ebeveyn** (**parent/base/super**), miras alan sınıfa ise **çocuk** (**child/derived/sub-class**) sınıf denir.
- Türetilen ya da çocuk sınıf, ebeveyninden, miras olarak alınabilecek üye değişkenler ile üye metotları devralır.
- Dolayısıyla, çocuk sınıflar, ebeveynlerine, durum ve davranış açısından benzerler.

Miras II

- Miras yapısını kurmak için Java'da **extends** anahtar kelimesi kullanılır:

```
public class Employee{
   protected int no;
   protected String name;
   protected int year;
   protected String department;
   ...
}

public class Manager extends Employee{
   protected String departmentManaged;
   ...
}
```



Miras İlişkisi - I

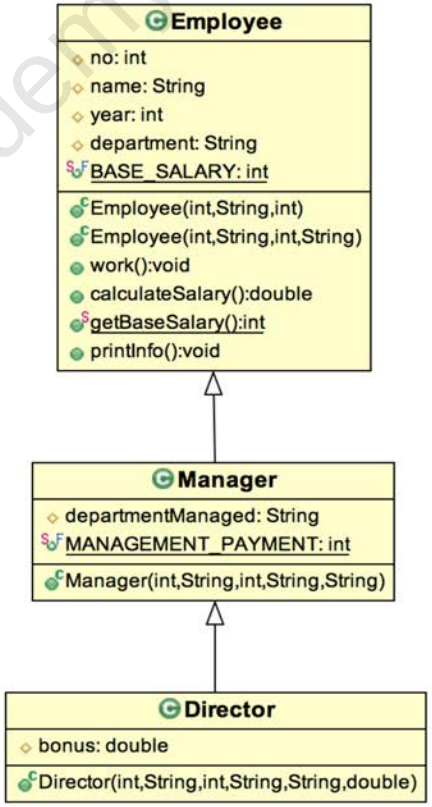
- Bu ilişki aşağıdaki okuma şekillerine imkan verir:
  - Her Manager/Bütün Managerlar aynı zamanda bir Employee'dir.
  - Her Manager/Bütün Managerlar bir Employee gibidirler.
- Çocuk sınıf, ebeveyninde **private** olan yapıları devralamaz, ancak **protected** , **public** ya da aynı pakette ise varsayılan olanları devralır.
- **protected** olan üye değişkenler halen dış dünyaya kapalıdırlar ama genelde **public** olan metotlar her halükarda devralınırlar.

Miras İlişkisi - II

- Miras ilişkisi ile tüm devralınabilecek olan ebeveyn üyeleri, çocuk sınıflar tarafından devralınır:
  - Nesne değişkenleri ve metotları
  - Sınıf değişkenleri ve metotları
- Ebeveynin kurucuları ise çocuklar tarafından devralınmaz.

Employee, Manager ve Director. java

- Employee, Manager ve Director sınıflarının kurucularına dikkat edin.
- org.javaturk.oop.ch09.factories.factory1.Test
- Test sınıfında, Manager ve Director nesneleri üzerinde, devralınan değişken ve metotlara ulaşımı gözlemleyin.



```
Employee e1 = new Employee(1, "Ali", 8, "Production");
e1.printInfo();
System.out.println("Maaşı: " + e1.calculateSalary());
e1.work();

Manager m1 = new Manager(2, "Fatma", 3, "Production",
                         "Production");
m1.printInfo();
System.out.println("Maaşı: " + m1.calculateSalary());
m1.work();

Director d1 = new Director(4, "Mehmet", 2, "Management",
                           "Management", 3000);
d1.printInfo();
System.out.println("Maaşı: " + d1.calculateSalary());
d1.work();

Employee.getBaseSalary();
Manager.getBaseSalary();
Director.getBaseSalary();
```

Üye Erişim Niteleyicileri I (Tekrar)

- Üyelere erişim için 4 seviye vardır:
  - **public** olan üyelere her yerden erişilir ve devralınır.
  - **private** olan üyelere sadece içinde bulunduğu sınıftan erişilir, dışarıdan erişime tamamen kapalıdır ve devralınamaz.
  - Varsayılan (default) olan üyelere erişim sadece paket içindeki sınıflara açıktır ve sadece aynı paketteki çocuk sınıflar devralabilir.
  - **protected**, hem aynı paketteki sınıflara açıktır ve hem de herhang bir paketteki alt sınıflarca devralınır.
    - Mirası düşünerek, üyelerin private yerine protected yapılması genel bir uygulamadır.
    - Devir/miras amacıyla protected olan üyeler de API'ye dahildirler.

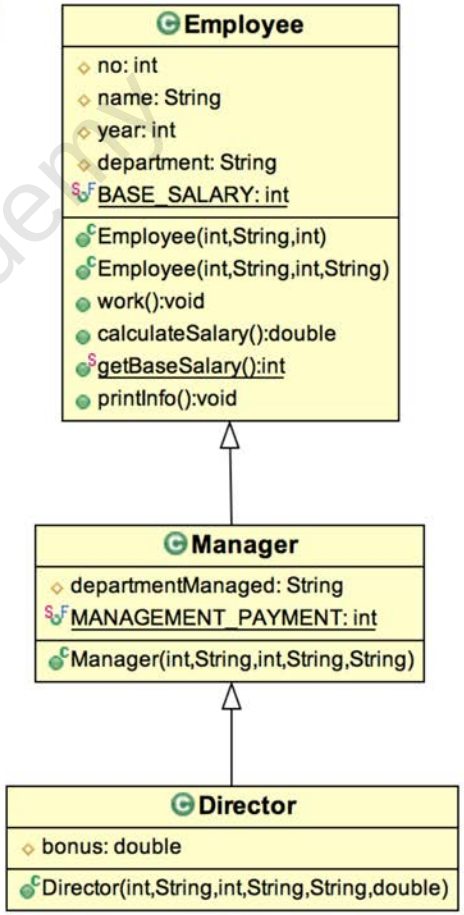
Üye Erişim Niteleyicileri II (Tekrar)

InheritanceExample.java

- Önce, aynı paketteki ParentClass1 sınıfından devralmayı sonra da otherPakage paketindeki ParentClass2 sınıfından devralmayı deneyin, protected olan üyelerin davranışını gözlemleyin.

Statik Üyeler

- Statik üyeler de devralınırlar ve hem alt sınıf hem de alt sınıfın nesnelerin üzerinden erişilebilirler.



InheritingStaticMembersExample.java

Kurucular (Constructors)

Kurucular (Constructors)

- Miras söz konusu olduğunda kurucularla ilgili iki önemli nokta söz konusudur:
  - Kurucular devralınmazlar.
    - Dolayısıyla her türetilen sınıf kendi kurucusunu tanımlamak zorundadır.
  - Hiyerarşide altta bulunan her sınıf, ebeveynindeki bir kurucuyu çağırmak zorundadır.
    - Bu da "her çocuk sınıfın nesnesinin içinde, gizli de olsa bir ebeveyn nesne var" anlamına gelmektedir.

super() Çağrısı - I

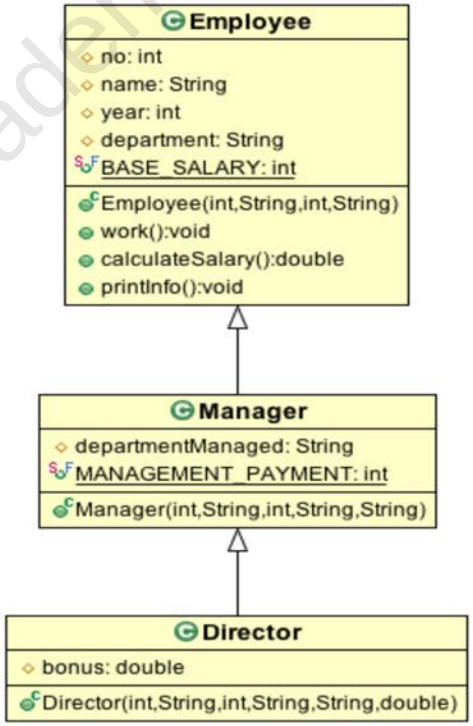
- Bir sınıfın, ebeveynindeki bir kurucuyu çağırması **super()** ile olur.
- super() çağrısı parametre geçmezse, ebeveyndeki argümansız kurucu çağrılmış olur.
- super() çağrısı tabi olarak parametre geçebilir, bu durumda ebeveyndeki bir akıllı kurucu çağrılmış olur.
  - Tipik olarak, ebeveynin tanımladığı durum bilgisi çocuk nesne oluşturulurken kurucusuna geçilir ve bu kurucu da bu durum bilgisini super() ile ebeveynindeki akıllı bir kurucuya geçer ki atamalar ebeveynin kurucusunda yapılsın.
  - Çocuk nesnenin kurucusuna geçilen ve ona has olan durum bilgis ise ebeveyne geçilmez.

super() Çağrısı - II

- super() çağrısı içinde bulunduğu kurucuda ilk çalışan kod olmalıdır.
  - Dolayısıyla, çocuk nesne oluşmadan önce, içindeki gizli olan ebeveyn nesne oluşmalıdır.

Employee, Manager ve Director. java

- org.javaturk.oop.ch09.factories.factory1.Test
- Employee, Manager ve Director sınıflarının kurucularına dikkat edin.
- Manager ve Director sınıflarının kurucularındaki super() çağrılarını gözlemleyin.



Başlatma (Initialization)

Mirasta Başlatma

- Hiyerarşide altta bulunan her sınıf, ebeveynindeki bir kurucuyu çağırmak zorundadır.
  - Bu da "her çocuk sınıfın nesnesinin içinde, gizli de olsa bir ebevey nesnesi var" anlamına gelmektedir.
- Bir sınıfın, ebeveynindeki bir kurucuyu çağırması **super()** ile olur.
- super() yoluyla yapılan kurucu çağrıları hiyerarşinin en tepesindeki sınıfa kadar devam eder.
- Dolayısıyla en önce hiyerarşinin en tepesindeki sınıfın kurucusu çağrılır ve nesnesi oluşur.
- Bunun dışında başlama sırasında değişen bir şey yoktur.

Başlama Sırası

- Dolayısıyla başlama sırası, sınıf hiyerarşisindeki en yukarıdaki sınıftan aşağıya doğru olur. Her sınıftaki başlama sırası ise
  - Sınıf değişkenleri (statik başlatma blokları dahil)
  - Nesne oluşturuluyorsa
    - Nesne değişkenleri (nesne başlatma blokları dahil)
    - Kurucu çağrısı

  şeklindedir.
- Birden fazla sınıf ve nesne değişkeni olduğu durumda başlatma sırası, fiziksel sırayla belirlenir, önce gelen önce başlatılır.

InitializationOrder.java

Genişletme ve Yerine Geçebilme

Genişletme

- Çocuk sınıflar, ebeveynlerinden miras olarak devraldıkları yapılara ekleme yapabilirler:
  - Çocuk sınıflar, genel olarak, ebeveynlerinde olmayan, yeni üye değişkenlere ve yeni metotlara sahip olurlar.
  - **extends** anahtar kelimesi zaten bu genişletmeyi ifade etmektedir
- Bu durumda türetilen çocuk sınıf, yeni üye değişkenlerle daha zengin bir yapıya, yeni metotlarla da daha geniş bir arayüze sahip olur.

Yerine Geçebilme I

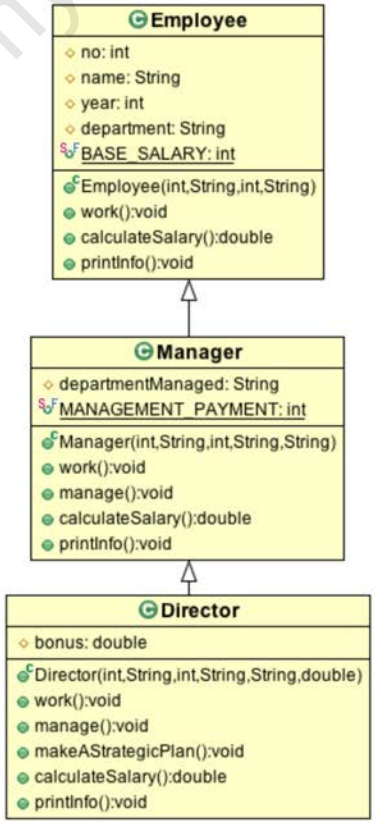
- Miras ilişkisinde çocuk sınıfın arayüzü, en azından ebeveyninin ara yüzüdür.
  - Çocuk sınıflar, ebeveynlerinden devraldıkları arayüze eklemeler yaparak daha geniş bir ara yüze sahip olup, daha çok iş yapar hale gelseler bile, ebeveynlerinin arayüzünü desteklemeye devam ederler.
  - Bu durum, üye değişkenler için de böyledir, yani ebeveynde erişilen he değişkene, çocuklarda da erişilir.
    - Ama prensip olarak değişkenlerin **protected** olduğunu ve dışarıdan ulaşılamadığını varsayıyoruz.
- Dolayısıyla, çocuk sınıflar, ebeveynlerinin sağladığı her özelliği, değişken ya da arayüz, sağlamak zorundadırlar.

Yerine Geçebilme II

- Bu durum, yerine geçebilme (**substitutability**) olarak ifade edilir ve hiyerarşide aşağıdan yukarıya doğru çalışır.
  - Yani, ebeveynin olduğu her yerde, ebeveynin çocuklarından biris olabilir.
    - Her Manager aynı zamanda bir Employee'dir.
    - Yani, patron, "bana bir çalışan çağırın" dediğinde, ona bir Manager gelirs patronun isteği yerine gelmiş olur.
    - Ya da patron, tüm çalışanlar toplansın dediğinde, Manager "beni çağırm diyemez.

Genelleştirme-Özelleştirme

- Miras ilişkisi, bir **genelleştirme-özelleştirme** (**generalization-specialization**) ya da **genel-özel** (**generic-specific**) ilişkisidir.
- Yani hiyerarşide yukarı çıkıldıkça daha genel nesneler, aşağı inildikçe, o nesnelerin daha özel halleri bulunur.
- Ama yerine geçebilme özelliği her zaman geçerlidir:
  - Her Director aynı zamanda hem bir Manager hem de bir Employee'dir.

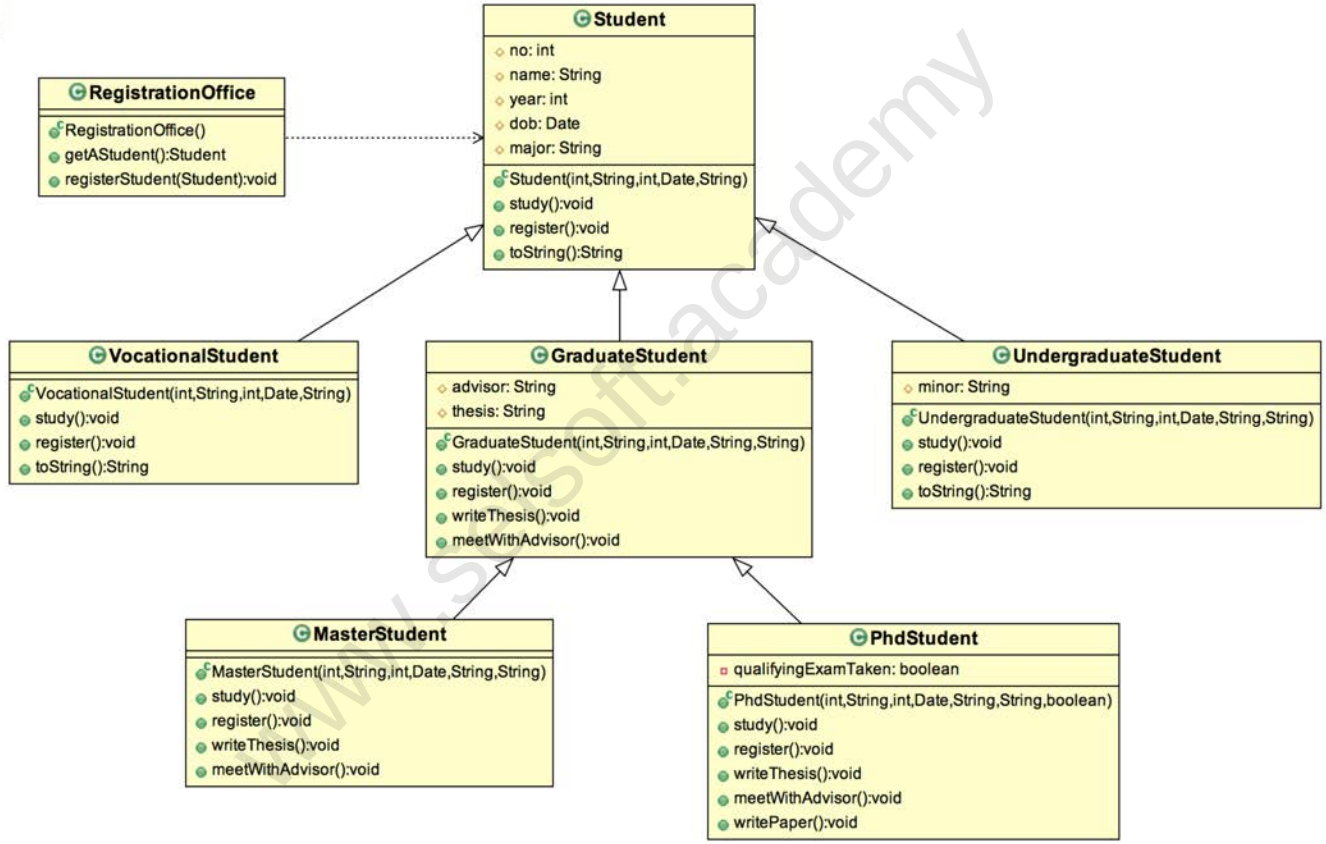


Employee, Manager ve Director. java

- org.javaturk.oop.ch09.factories.factory2.Test

Uygulama

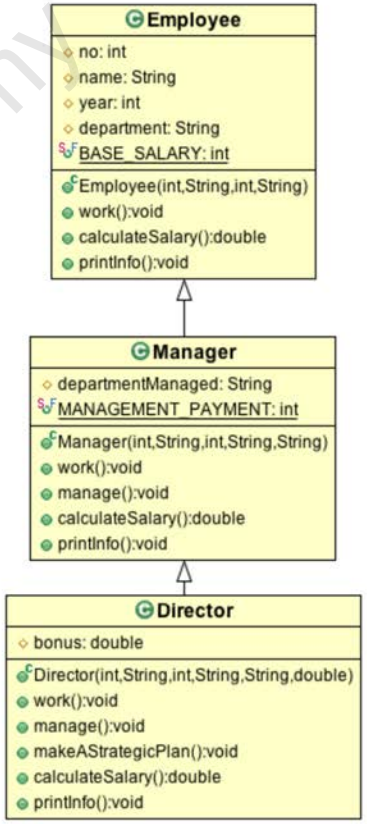
- Bir üniversitedeki öğrencileri, aralarındaki miras ilişkisini göz önüne alarak şekildeki gibi modelleyin.
  - Kurucu çağrılarını super() kullanarak kurgulayın.
- Hangi durumlarda genişletme söz konusudur tartışın.



Overriding (Ezme)

Davranış Devralma - I

- Bir hiyerarşide yer alan nesnelerinden ebeveynde tanımlanan bir metodun, **private** olmadığı müddetçe, altındaki nesneler tarafından da devralındığını biliyoruz.
  - Hiyerarşideki bütün nesneler, hangi tür olursa olsun, ebeveynindeki sorumluluklara sahiptir.
- Fakat çocukların aynı sorumluluğu farklı davranarak yerine getirmeleri mümkün müdür?



Davranış Devralma - II

- Hiyerarşinin altındaki nesneler tabi olarak kendileri için gerekli yeni metotlar tanımlayabilirler.
- Fakat istenen şey, aynı metodun tüm hiyerarşi için geçerli olurken hiyerarşide altta bulunan çocukların metodun davranışını değiştirebilmeleridir.
- Bu ise metodun arayüzünün aynen bırakılıp, gerçekleştirmesinin yani kodunun değiştirilmesiyle söz konusu olur.

Overriding – Ezme - I

- Nesneler, ebeveynlerinden devraldıkları metotların arayüzlerini değiştirmeden, kodunu değiştirebilirler.
- Buna **overriding** ya da **ezme** denir.
- Yani çocuk nesneler, ebeveynlerindeki sorumluluğu, farklı bir şekilde yerine getirmeyi tercih edebilirler.

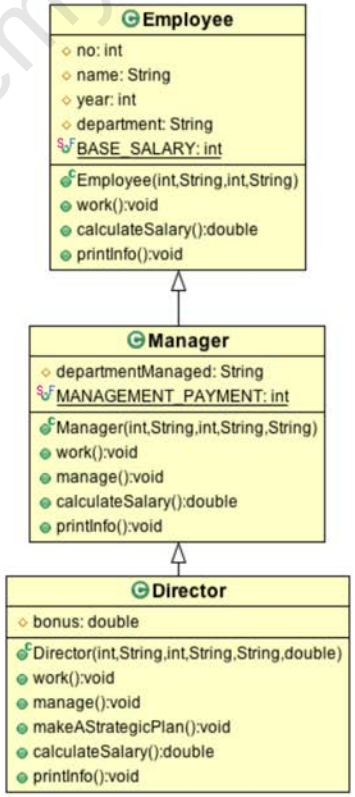
- Overriding ile aynı sorumluluk farklı şekillerde yerine getirilir:
  - Sorumluluk aynıdır, çünkü arayüz (interface) aynıdır, ama sorumluluğu yerine getirme şekli (implementation) farklıdır.

Overriding – Ezme - II

- Bu şekilde override edilen metotlara **polymorphic** (**çok şekilli**) metotlar denir.
- Çünkü sorumluluk bir tanedir çünkü arayüz bir tanedir ve ebeveynde tanımlanır.
- Ama sorumluluğu yerine getirme yani metot birden fazladır.
- Bu yüzden override edilebilen metotlara polymorphic denir.

Employee, Manager ve Director. java

- org.javaturk.oop.ch09.factories.factory2.Test
- *Employee* üzerinde tanımlanan *work()* , *printInfo()* ve *calculateSalary()* metotlarının *Manager* ve *Director* için override edildiğini gözlemleyin.



Overriding – Ezme - III

- Override, sadece nesne metotları için geçerlidir.
  - Nesne metotlarını aynı arayüzle alt sınıflarda tekrar tanımlarsanız, onları override etmiş (ezmiş) olursunuz.
  - Üye değişkenleri aynı isimle alt sınıflarda tekrar tanımlarsanız, ebeveyndekileri devralmamış, sadece saklamış olursunuz.
    - Çünkü overriding değişkenler için tanımlı değildir.
- Statik metotlar da override edilemezler.
- Polymorphic davranış sadece nesne metotları için geçerlidir, üye değişkenler ve statik metotlar polymorphic değildirler.
- Polymorphismi ileride ele alacağız.

ShadowingExample.java

StaticOverridingExample.java

Bir Nokta!

```
public class ParentClass{
    public int i;
    public void f(){}
}

public class SubClass extends ParentClass{
    private int i;
    void f(){}
}
```

- Yukarıdaki kod derleme hatası verecektir.
- Neden?

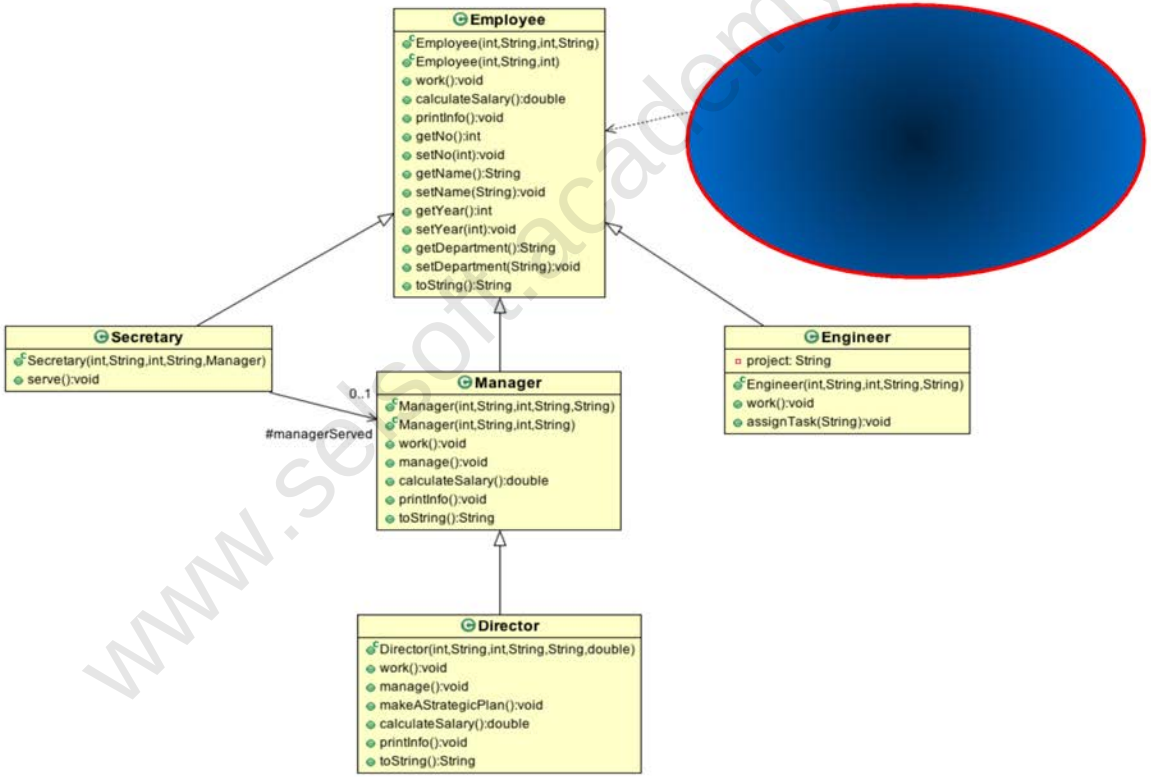
Daha Kısıtlayıcı Olarak Override

- Override ederken, devralınan metodu daha kısıtlayıcı bir erişim belirteciyle tanımlayamazsınız.
- Aksi taktirde, ebeveyn üzerinden ulaşılan bir metodun, çocuk nesneler üzerinden ulaşılamaması söz konusu olurdu!

Override Ederken Alt Tip Return - I

- Java SE'nin 1.5 sürümünden itibaren, override ederken, devralınan metodun dönüş tipi, alt tipleriyle yer değiştirebilir.
  - Buna covariant return type denir.
- Yani alt tipteki bir metot, ebeveynindeki metodun döndürdüğü tipin daha özel tipini döndürebilir, yerine geçebilme özelliği hala korunur.
- Çünkü daha geniş bir tip bekleyen istemci sınıfa daha özel bir tip döndürmek problemli değildir.
  - Yerine geçebilme özelliği!

Override Ederken Alt Tip Return - II



HR.java & HRForManagers.java

- org.javaturk.oop.ch09.covariant package

Bir Nokta: Overloading

- Ebeveyn sınıfta tanımlanan bir metodu, farklı arayüz ama aynı isimle bir alt sınıfta tekrar tanımlarsak, tabi olarak bu **overloading** olur.

@Override Notu

- Override edilen metotlar "**@Override**" ile notlandırabilir (annotating).
- Java'da "@" ile başlayan ve tip, metot, değişken, parametre vb. yapıları niteleyen öğelere **not** (**annotation**) denir.
- Notların kullanımı bazen isteğe bağlıdır "@Override"da olduğu gibi, bazen daha fonksiyonel amaçlar için kullanılır.
- Notlar, Java API'sinin bir parçasıdır.
- "@Override" da java.lang paketindeki notlardan birisidir ve nitelediği metodun override edildiğini ifade eder.
  - Bu not kullanıldığı halde override yapılmazsa derleyici hata verir!

Uygulama

- Modellediğiniz üniversite öğrencilerinde uygun metotları override edin.

super

super Anahtar Kelimesi

- Zaman zaman bir hiyerarşide bulunan bir alt sınıftan, onun ebeveyn sınıfındaki yapılara ulaşmak isteyebilirsiniz.
- Bu durumda **super** anahtar kelimesi kullanılır.
- super ebeveyn nesnesine bir referanstır ve ebeveyn nesnesinin özelliklerine ve davranışlarına ulaşmakta kullanılır.
- super sadece hemen bir üstteki ebeveyne ulaşmak için kullanılır,
  - "super.super" gibi bir kullanım söz konusu değildir.

SuperExample.java

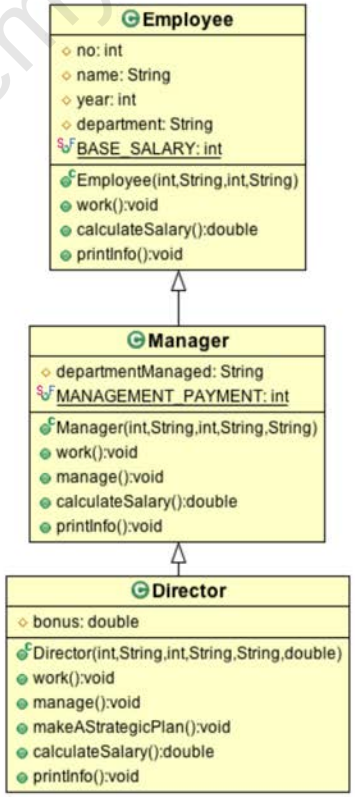
- Bu örnek dışında factory paketindeki Manager ve Director' sınıflarında da super kullanımı vardır.

super ve Tekrar Kullanım

- super, tekrar kullanımı arttırır, kod tekrarını azaltır.
- Özellikle bir alt tipin davranışının, ebeveyninin davranışını aynen alıp, öncesine ve/veya sonrasına eklemeler yaptığı durumlarda ebeveyndeki davranışı copy-paste ile tekrarlamak yerine super ile tekrar kullanılması çok daha kaliteli bir kod ortaya çıkaracaktır.

Employee, Manager ve Director. java

- org.javaturk.oop.ch09.factories.factory3.Test
- *Employee* üzerinde tanımlanan *work()* , *printInfo()* ve *calculateSalary()* metotlarının *Manager* ve *Director* için override edilirken super() çağrısından nasıl yararlanıldığını gözlemleyin.



Uygulama

- Modellediğiniz üniversite öğrencilerinde override ettiğiniz metotlarda mümkün olan yerlerde "super" kullanarak kod tekrarından kaçının.

Bileşim, Miras ve Sarmalama

Bileşim ve Sarmalama

- Bileşim'de bileşik nesne ile parça nesneler arasında tek yönlü (uni-direction) ya da iki yönlü (bi-direction) ilişkiler söz konusudur.
- Bileşim'de bileşik nesne ile iyi sarmalanmış parça nesneler arasındaki bağımlılık ise sadece arayüz bağımlılığıdır.
- Bileşik nesne, parça nesnelerin gerçekleştirme detaylarını (implementation details) bilmez, sadece arayüzleri üzerinden hizmet alır.

Miras ve Sarmalama

- Mirasın, sarmalamayı bozduğu, kırdığı iddia edilir.
- Inheritance violates encapsulation.
- Çünkü, çocuk sınıflardan ebeveyne doğru ciddi bir bağımlılık vardır.
- Miras, çocuk sınıf ile ebeveyni arasında sadece arayüz düzeyinde değil ama gerçekleştirme düzeyinde de bir bağımlılık oluşturur.
- Bu da ebeveyne yapılacak gerçekleştirme değişikliklerinin, arayüzde hiç bir şey değişmese bile, çocuk sınıflarda da değişikliğe yol açması anlamına gelecektir.

Bileşim ve Miras

- Bileşim, daha sağlıklı bir bağımlılık oluşturmasından dolayı mümkün olan her durumda mirasa tercih edilmelidir.

Favor object composition over class inheritance.

- is-a ilişkisinin bir ya da daha fazla has-a ilişkisine dönüşebildiği durumlarda bileşim tercih edilmelidir.
- Derin, 3 ve daha fazla katmana sahip ve geniş miras hiyarşileri de kaçınılması gereken durumlardandır.

final Sınıflar ve Metotlar

Final Sınıflar

- Eğer bir sınıf **final** olarak tanımlanırsa mirasa konu olamaz, yani hiç bir sınıf tarafından ebeveyn olarak görülemez.
  - Final tanımlanan sınıflar, başka sınıflardan miras devralabilir.
- Java API'sinde final olan sınıflara örnek:
  - java.lang.String, java.lang.StringBuffer, java.lang.StringBuilder
  - java.lang.Boolean, java.lang.Integer gibi bütün wrapperlar
  - java.lang.System, java.lang.Math, java.lang.Class
  - java.util.Objects, java.util.Scanner

FinalClassExample.java

Neden Final Sınıf?

- Bir sınıfı final yapmanın en temel sebebi, hiyerarşiden kaçınmak dolayısıyla o sınıfın sağladığı davranışların tamamının o haliyle kullanıldığından emin olmaktır.
- Bunun sebebi de şunlar olabilir:
  - Sınıfın sağladığından daha iyisini yapılamayacağını düşünmek,
  - Güvenlik ve güvenirlik (security, reliability-safety) gibi sebepler,
  - Mirasın oluşturduğu yüksek bağımlılıktan kaçınmak.
- final sınıfların tüm durum değişkenlerinin de private olmaları beklenir, protected yapmaya gerek yoktur.

Final Metotlar

- Eğer bir metot **final** olursa, o metot ezilemez.
- Bir metodu final yapmanın en temel sebebi, o metodun sağladığı davranışın o haliyle kullanıldığından emin olmaktır.
- final olan metot, devralınır ama ezilemediği için, çocuk sınıfın nesneleri üzerinden ulaşıldığında, ebeveynde tanımlanan davranışı gösterir.
- final olan bir sınıfın tüm metotları doğrudan final olur, ayrıca final olarak tanımlanmalarına gerek yoktur.

FinalMethodExample.java

Neden Final Metot?

- Bir sınıfı final yapmanın en temel sebebi, o sınıfın sağladığı davranışların tamamının o haliyle kullanıldığından emin olmaktır.
- Bunun sebebi de şunlar olabilir:
  - Genel olarak sınıfın final olmasındaki sebepler burada da geçerlidir.
  - Ayrıca tüm hiyerarşide tek bir gerçekleştirmenin olmasını istemek

private mi final mı?

- **private** bir metot devralınmaz, bu yüzden override ezilmesi söz konusu değildir.
- Bu yüzden eğer çocuk sınıf private metodu aynen tekrar tanımlarsa bu ne overriding olur ne de overloading olur!

- **final** bir metot devralınır ama ezilemez.
- Bu yüzden çocuk sınıf final bir metodu tekrar tanımlayamaz.
- Bir metodu hem private hem de final yapmak ne anlama gelir?
  - private varken metodu final da yapmak gereksizdir.

java.lang.Object Sınıfı

java.lang.Object Sınıfı - I

- **java.lang** paketindeki **Object** sınıfı, bütün Java sınıflarının doğrudan ve üstü kapalı olarak kendisinden miras devraldıkları sınıftır.
- Dolayısıyla Object sınıfı, her Java sınıfının ebeveynidir.
- Java derleyicileri, sınıfınızın .class dosyasını üretirken, "extends Object" ifadesini koda eklerler.

java.lang

Class Object

java.lang.Object

`public class Object`

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

`JDK1.0`

java.lang.Object Sınıfı - II

- Java SE'nin 1.8 sürümü itibariyle Object sınıfının 11 tane metodu vardır.
- Bu metotların 6 tanesi finaldır dolayısıyla override edilemez.
- Geri kalan 5 metot, tüm Java sınıflarında override edilebilir, hatta edilmelidir.
- Eğer bu metotlar override edilmezlerse Object sınıfındaki varsayılan davranış geçerli olur.

Object Sınıfı – final Metotları

- Object sınıfındaki şu metotlar override edilemez:
  - Class getClass(): Nesnenin sınıfının Class nesnesini döndürür.
  - void notify() ve void notifyAll(): Thread davranışı ile ilgilidir, ileride ele alıancaktır.
  - void wait() metotları: 3 tane overloaded hali vardır ve Thread davranışı ile ilgilidir, ileride ele alınacaktır.
- Bu metotların davranışları, override edilmesine izin verilmeyecek kadar temel ve önemlidir.

java.lang.Object Sınıfı - III

- Object sınıfındaki şu metotlar override edilebilir:
  - String toString(): Nesnenin String formunu oluşturur ve geri döndürür.
  - boolean equals(Object o): Referansı, kendisine geçilen bir başka nesne referansı ile karşılaştırır.
  - int hashCode(): Nesnenin hash kodunu döndürür.
  - Object clone(): Nesnenin kopyasını oluşturup geri döndürür.
  - void finalize(): Garbage collector tarafından toplanmadan önce çağrılır. Temizlik yaparak sistem kaynaklarını salıvermek için kullanılır.

toString() Metodu

- toString() metodu nesnenin String sunumunu geri döndürür.
- Object sınıfı üzerindeki asıl hali ise, tabi olarak en temel bilgilerden yola çıkarak gerçekleştirilmiştir ve nesnenin tam ismi ile hash kodunu döndürür.

```
getClass().getName() + '@' + Integer.toHexString
```

durumunu String olarak döndürecek şekilde override edilmelidir.

ToStringExample.java

equals() ve ==

- "==" ile eşitlik kontrolü sadece basit tipler üzerinde yapılmalıdır.
- Nesnelerin eşitliğini anlamak için "==" kullanılmaz, equals() kullanılır.
- Eğer karmaşık tipleri kıyaslamakta "==" kullanılırsa, bu durumda nesneler yerine nesnelerin referansları kıyaslanır.
  - Eğer referanslar aynı nesneyi gösteriyorsa "==" true döndürür, ak taktirde false döndürür.
- Bu yüzden nesnelerin, durumları açısından aynı olup olmadıklarını anlamak için equals() metotları daima override edilmelidir.

EqualsExample.java

Hash Code - I

- Hash code, JVM'in her nesne için ürettiği bir int değerdir.
- Hash code, işaretli 2^32 değerden birisidir.
- java.lang.Object üzerindeki hashCode() metodu native olarak gerçekleştiren ve nesnenin bellekteki adresini kullanarak int bir değer üreten bir hash fonksiyonu kullanır.
- Hash code, özellikle Java API'sindeki HashSet vb. torba (collection) nesneleri tarafından, kendisine eklenen nesneleri yönetmekte kullanılır.

Hash Code - II

- Java'nın bazı sınıflarında hash code şöyle hesaplanır:
  - Integer sınıfı int değerini hash code olarak çevirir,
  - Long sınıfı **(int)(value ^ (value >>> 32))** değerini,
  - Double sınıfı **(int)(bits ^ (bits >>> 32))** değerini,
  - Character sınıfı **(int)value** değerini,
  - Boolean sınıfı, true ve false için 1231 ve 1237 değerlerini,
  - String sınıfı s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1] değerini döndürür.
- Siz de oluşturduğunuz her sınıfta hashCode() metodunu ezerek tekil (unique) bir int değer üretmelisiniz.

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

a hash code value for this object.

See Also:

`equals(java.lang.Object)`, `System.identityHashCode(java.lang.Object)`

equals() ve hashCode() - I

- hashCode() ve equals() metodları, bir arada, tutarlı bir şekilde gerçekleştirilmelidir.
- Tutarlılık açısından equals() metodunun true döndürdüğü nesneler için hashCode() da aynı int değeri hash olarak döndürmelidir.
  - Yani durumu aynı olan aynı tipten nesneler için aynı hash code sö konusudur.
  - Yani durumu aynı olmayan aynı tipten nesneler için farklı hash code söz konusudur.
  - Aynı tipten olmayan nesneler için tabii olarak farklı hash code söz konusudur.

HashCodeExample.java

equals() ve hashCode() - II

- hashCode() metodunu ezmek, özellikle iş nesneleri için çok zor değildir.
- Örneğin nesnenin tekil olan alanlarını kullanarak hash üretmek sağlıklı bir yoldur.
- Ama her halükarda tutarlı sonuç için hashCode() ve equals() metodunda aynı alanlar kullanılmalıdır.

Tekli ve Çoklu Miras

Tekli Miras (Single Inheritance) - I

- Java'da, bir sınıfın diğer bir sınıftan devralması anlamında tekli miras vardır.
  - Yani bir sınıf, aynı anda birden fazla sınıftan miras devralamaz.
  - Dolayısıyla extends anahtar kelimesinen sonra sadece bir tane sınıf gelebilir.
- Ama bir sınıfın, hiyerarşinin farklı katmanlarından gelen birden fazla ebeveyni olabilir.
  - Her Director aynı zamanda hem bir Manager hem de bir Employ dir, yani iki tane ebeveyni vardır.

Tekli Miras (Single Inheritance) - II

- Sınıftan devralma anlamında çoklu mirasa izin veren C++'ın bu özelliğinin getirdiği sıkıntıları aşmak üzere Java'da tekli miras vardır.
- İleride çoklu mirasın Java'da nasıl olabileceğini öğreneceğiz.
- Ayrıca arayüzü (interface) devralmak ile gerçekleştirmeyi ya da kodu (implementation) devralmayı da ayıracağız.

Özet

- Bu bölümde, nesne merkezli dillerin bir diğer temel özelliği olan miras ya da kalıtım ele alındı.
- Kalıtımın yapısı, bu yolla devralınan özellikler, başlatma sırası, overriding mekanizmaları detaylı olarak işlendi.
- Bütün Java sınıflarının bir üst sınıfı olarak java.lang.Object sınıfı ve metotları işlendi.

Ödevler

Ödevler I

- Shape sınıfının en tepede olduğu bir hiyerarşi düşünün.
  - Shape'in üzerinde draw(), erase(), calculateArea() ve calculateCircumference() metotları vardır.
  - Circle, Rectangle, Square ve Triangle ise Shape'in alt sınıflarıdır ve bu metotları override ederler.
  - Metotları override ederken mümkünse "super"i kullanın.
  - Sınıflardaki equals(), hashCode() ve toString() metotlarını override edin.
  - Test sınıfında da random Shape nesneleri üretip üzerinde metot çağrıları yapın.
- Daha önce yaptığınız üniversite örneğindeki sınıflar için equals(), hashCode() ve toString() metotlarını override edin.