Java ile Nesne Merkezli ve Fonksiyonel Programlama

4. Bölüm
Arayüzler
(Interfaces)

Akın Kaldıroğlu

www.javaturk.org

Ocak 2017

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.

İçerik

- Bu bölümde şu konular ele alınacaktır:
  - Arayüz (interface) tipi,
  - Arayüzlerin gerçeklenmesi,
  - Yazılım tasarımında arayüzlerin kullanımı,
  - Java SE 8 ile gelen yenilikler ve
  - Geri çağırma metotları (callback methods) ve isimsiz sınıflar (anonymous classes)

Tanımlar

Interface - Arayüz Nedir?

- İki şeyin birbirleriyle iletişimde bulundukları ortama arayüz (ya da arabirim) (interface) denir.
- Örneğin dil (language), yüz, mimikler, hareketler ve tavırlar insanlar arası iletişimin ortamıdır, yani insanların arayüzüdür.
- Yazılımın nesneleri de birbirleriyle arayüzleri üzerinden iletişimde bulunurlar.

İmza ve Arayüz (Tekrar)

- Bir metotun, isim ve parametre listesinden oluşan bilgisine **imza** (**signature**) denir.
  - Dönüş değeri ve fırlatılan sıra dışı durumlar imzaya dahil değildir.
- Bir metotun, isim, parametre listesi, dönüş değeri tipi ve fırlattığı sıra dışı durumlardan oluşan bilgisine **arayüz** (**interface**) denir.

```
<niteleyici>* <dönüş tipi> <isim>(<Parametre>*) throws <exception>*

public double squared(double arg) throws IllegalArgumentException

public double squared(double arg) throws IllegalArgumentException
```

imza (signature)

arayüz (interface)

Metot Arayüzü

- Metot arayüzü, o metodun ne yaptığını ifade eden bilgisidir.
  - Arayüz, metodun görevini nasıl yaptığını gösteren gerçekleştirmesi (implementation) ile birlikte metodun iki parçasından birisidir.
  - Gerekirse arayüz bilgisi dokümantasyon ile desteklenir.

```
/**
* Method that calculates the square of a double parameter.
* @param Parameter whose square to be calculated.
* @return Square of the parameter passed.
* @throws IllegalArgumentException Thrown when a negative argument supplied.
*/
public double squared(double arg) throws IllegalArgumentException{
    if(arg < 0)
        throw new IllegalArgumentException("Negative argument.");
    else
        return Math.sqrt(arg);
}
```

Nesnenin Arayüzü (Tekrar)

- Bir nesnenin sahip olduğu metot arayüzlerinin tamamına, o nesnenin arayüzü denir.

- Dolayısıyla nesne arayüzü, nesnenin sınıfında tanımlanan metot arayüzlerinin toplamıdır.

Arayüz (Interface)

Interface – Arayüz - I

- **Arayüz** (**interface**), en kısa tanımıyla, tüm metotları soyut olan sınıftır.
- Arayüzler tanımlanırken **interface** anahtar kelimesi kullanılır.
- Arayüzlerin metotları otomatik olarak hem public hem de abstract olur.
- Aşağıdaki tanımlar birbirlerine eşittirler.

```
public interface Worker{

    public abstract void work();
}
```

```
public interface Worker{

    void work();
}
```

Interface – Arayüz - II

- Arayüz (interface) sadece arayüz sağlar, ne durum ne de gerçekleştirme sağlamaz.
  - Bu bir beyaz yalandır, dolayısıyla en azından bunu şimdilik böyle kabul edelim.
  - Çünkü Java SE 1.8 ile artık arayüzlerde gerçekleştirme verilebilir!
- Dolayısıyla bundan sonra aksi söylenene kadar arayüzleri, Java SE 1.8 ile gelen yenilikleri göz önüne almadan, hala Java SE 1.7 sürümünde olduğu gibi işleyeceğiz.
- Yani arayüzlerde herhangi bir gerçekleştirme olamayacağını farzedeceğiz.

Interface – Arayüz - III

- Arayüzdeki tüm metotlar abstract olmak zorundadırlar,
  - Arayüzün metotlarına gerçekleştirme verilemez.

- Bu yüzden bu yapılara interface (arayüz) denilmektedir.
  - Arayüzler bu anlamda sadece form – şekil yani "ne'lik" sağlayan yapılardır, içerik yani "nasıl'lık" sağlamazlar.

- Arayüzlerin kurucusu (constructor) yoktur ve nesnesi de oluşturulamaz.

Implementing An Interface

- Bundan dolayı bir sınıf, bir arayüzün alt tipi olurken "**implements**" anahtar kelimesini kullanır.
  - Yani bir sınıf, bir arayüzü implement eder (yerine getirir ya da gerçekleştirir).
  - Bir arayüzü gerçekleştirmek demek, metotlarını override ederek onlara gerçekleştirme (implementation) vermek demektir.

```
public interface Worker{
    void work();
}
```

```
public class Employee implements Worker{
    void work(){ ... }
}
public class Boss implements Worker{
    void work(){ ... }
}
```

Worker.java ve Employee.java

- *factory1* paketi.

Arayüz ve Miras

- Arayüzler, sorumluluklara gerçekleştirme sağlamadıklarından, kendilerinden bu arayüzleri devralan alt sınıfların, arayüzdeki tüm metotlara gerçekleştirme vermeleri gereklidir.
  - Çünkü arayüzdeki tüm metotlar soyutturlar.

- Aksi taktirde, yani alt sınıf, arayüzden devraldığı bir metoda bile gerçekleştirme vermezse, alt sınıfın soyut (abstract) tanımlanması gereklidir.
  - Çünkü sınıf hala soyut metoda sahiptir.

Sözleşme ve Yüklenici

- Arayüz, sorumlulukların sıralandığı bir soyut yapıdır.
  - Bir arayüzü gerçekleştiren bir sınıf, arayüzdeki metotlara birer gerçekleştirme sağlamakla yükümlüdür.
- Bu, arayüzün bir sözleşme (contract) ya da protokol (protocol), gerçekleştiren sınıfların ise birer yüklenici (implementor) olarak nitelendirilmesine de izin verir.
- Aynı arayüzü gerçekleştiren sınıflar, aynı sözleşmeyi yerine getiren farklı yüklenicilerdir.

Arayüz ve Is-A

- Arayüz ile onu gerçekleştiren sınıflar arasında is-a ilişkisi tabi olarak vardır.
- Bundan dolayı arayüzün, polymorphism, upcasting ve downcasting açısından, sınıflardan hiç bir farklı yoktur.
- Bir arayüz, kendisini gerçekleştiren sınıfların üst tipidir.
- Aralarındaki temel fark, soyutluk-somutluk spektrumundaki yerleridir.

Test.java

- *factory1* paketi.

Arayüz ve Yetkinlikler

- Arayüzlerin “is-a” hiyerarşisinde bir üst tip olarak görülmesi yanında, yetkinlik kazandırması da söz konusudur.

- Arayüzler, tanımladıkları soyut davranışlar ile alt sınıflarına yetkinlik kazandırırlar.
  - Bunda, bir sınıfın pek çok arayüzü yerine getirmesi de rol oynar.

- Sınıfların, yerine getirdikleri pek çok arayüz ile pek çok yetkinliği (ability) kazandığı düşünülür.

Arayüz ve Can-Do - I

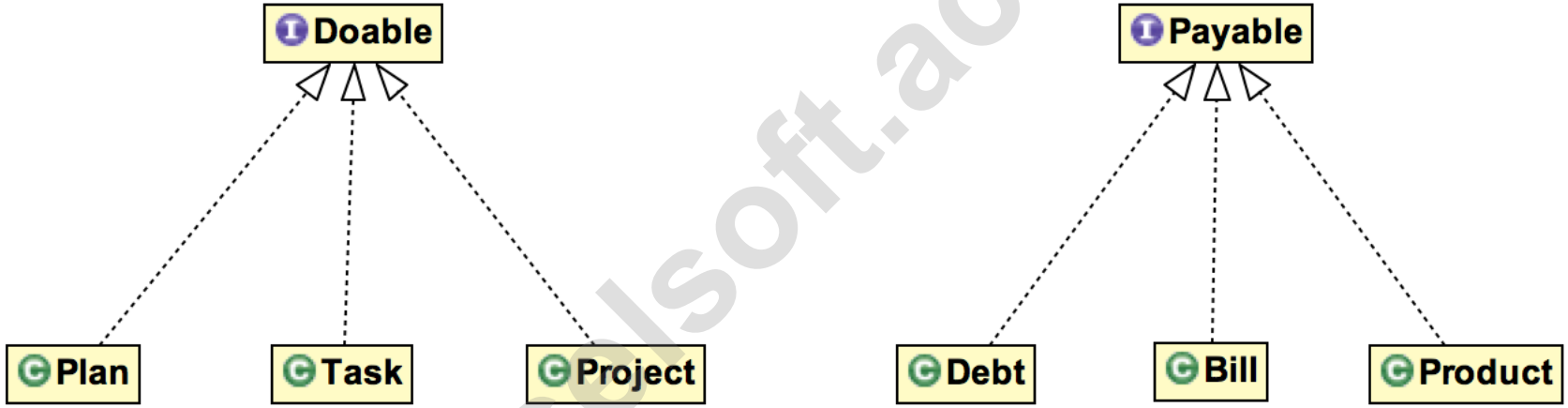
- Bu sebeple zaman zaman arayüzler, “able” son takısıyla isimlendirilirler:
- Bu durumda alt sınıflar ile arayüzleri arasındaki ilişki “can-do” olarak da okunabilir.

```
public interface Doable{
    void doIt();
}
```

```
public interface Payable {

   double calculatePrice();
   double calculateTax();
   void pay(double price);
}
```

Arayüz ve Can-Do - II



Doable.java ve Payable.java

- *canDo* paketi.

Arayüzlerin Kullanımı

- Arayüzler daha önce ifade ettiğimiz "program to an interface, not an implementation" prensibinin uygulanması için en ideal yapıdırlar.
  - Çünkü arayüzler sadece metot arayüzü sağlarlar.

Calculator.java

- *math* paketi.
- İhtiyaca göre farklı türde tek argümanlı matematiksel fonksiyonlara sahip olabilen bir Calculator düşünün.

- Matematiksel fonksiyonların, Calculator'e eklenebilmesi (pluggable) nasıl sağlanabilir?

- "Program to interface, not an implementation" prensibinin uygulamasını gözlemleyin.

Uygulama

- Celcius, Fahrenheit ve Kelvin sıcaklık derecelendirme sistemleri arasında dönüşüm yapacak bir yapıyı arayüz kullanarak tasarlayın.

Arayüz ve Miras

Pek Çok Arayüz Gerçekleştirme

- Bir sınıf aynı anda birden fazla arayüzü gerçekleştirebilir.
- Bunun için sınıf tanımlanırken, implements'ten sonra arayüzler virgül ile listelenir.
- Bu durumda sınıfın, yerine getirdiği arayüzlerden devraldığı tüm metotlara gerçekleştirme vermesi gerekir.
  - Aksi taktirde o sınıf soyut olmak zorundadır.

```
public class Employee implements Worker, Schedulable{

    void work(){ ... }

    public schedule(){ ... }
}
```

Pek Çok Arayüz Gerçekleştirme

- Bir sınıf hem bir sınıftan miras devir alırken bir ya da daha çok arayüzü yerine getirebilir.

```
public class Employee extends Person
                      implements Worker, Schedulable{

   public void live(){ ... }

   public void work(){ ... }

   public schedule(){ ... }
}
```

Employee.java

- *factory2* paketi.

Mirasa Yeniden Bakış

- Soyut sınıf ve arayüz mekanizmalarından sonra miras ile ilgili olarak şu ayrım yapılabilir:
  - Gerçekleştirme devralma (implementation inheritance): Bir sınıftan "extends" ile miras devralınmasıdır.
    - Bu durumda hem arayüz hem de gerçekleştirme devralınır:
      - Üye değişkenler ve metotlar.
  - Arayüz devralma (interface inheritance): Bir arayüzden "implements" ile miras devralınmasıdır.
    - Bu durumda sadece arayüz devralınır, hiç bir gerçekleştirme devralınmaz, (en azından şimdilik böyle kabul edelim).
      - Üye metotların arayüzleri.

Java APIsindeki Arayüzler - I

- Java API'sinde pek çok arayüz vardır.

- Bazıları:
  - java.lang.Comparable
  - java.lang.Runnable
  - java.util.Collection

Java APIsindeki Arayüzler - II

- Java API'sindeki arayüzlerden bazılarında hiç metot yoktur.
- Örneğin
  - java.io.Serializable
  - java.lang.Cloneable
  - java.util.RandomAccess
- Bir arayüzde hiç metot olmaması nasıl açıklanabilir?

Arayüz Metotlarına Erişim

- Arayüz metotları public erişime sahiptirler.
- Dolayısıyla, bu metotlar alt sınıflarda gerçekleştirilirken de public olmak zorundadırlar.
- Çünkü metotlar override edilirken, daha kısıtlayıcı bir erişim belirteci ile tanımlanamazlar.
- Yerine geçebilme (substitutability) özelliği arayüz ve onun alt sınıfları için de geçerlidir.

Arayüz - Durum

- Arayüzler durum (state) da sağlamazlar.

- İlk bakışta bir arayüzde durum tanımlanıyormuş sanılır:

```
public interface Worker{

    String name = "Ahmet";
    int  year = 10;

}
```

- Ama bu algı geçicidir,

Arayüz – Sabiteler

- Arayüzde tanımlanacak tüm alanlar otomatik olarak public, statik ve final olurlar.
  - Bu yüzden alanlar için public, static ve final dışında belirteç (modifier) kullanılamaz.
  - Ve tüm alanlara bir ilk değer vermek zorunludur.
  - Bu yüzden de büyük harflerle yazılıp, varsa kelimeler arası "_" ile ayrılır: NEXT_YEAR
- Bu durum, ironik bir şekilde, arayüzleri sadece sabiteleri toplayan bir yapı halinde kullanılmasına sebep olur.
- `java.io.ObjectStreamConstants` böyle bir arayüzdür.
- Arayüzlerin bu şekilde kullanımının alternatifi **enum** tipidir.

Months.java

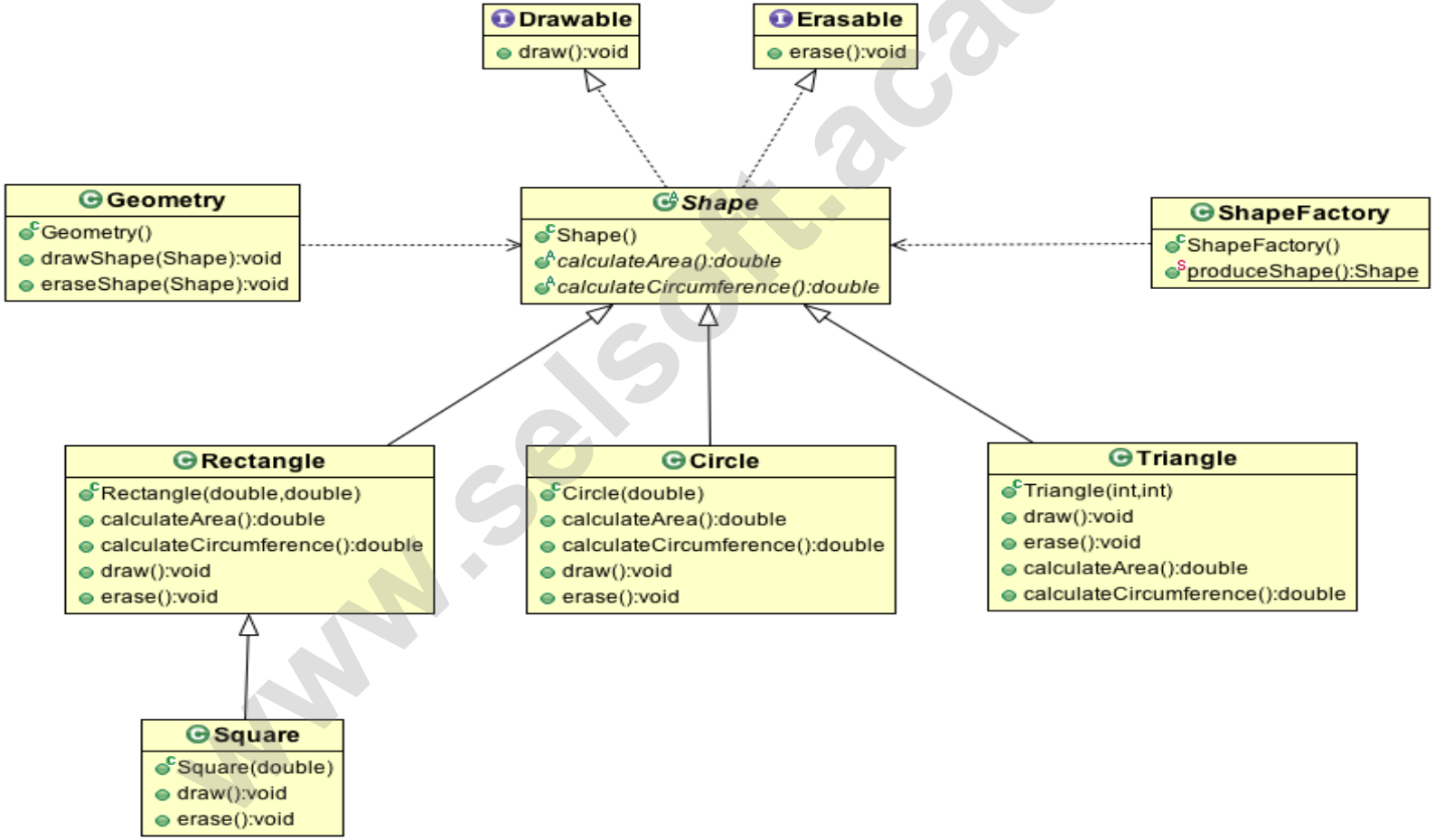
Arayüz – Soyut Sınıf

- Arayüzlerin bir duruma sahip olmamalarından dolayı, soyut sınıflar sıklıkla arayüzlerin alt tipleri olur.

- Bu durumda arayüz hiyerarşinin ortak davranışlarını içeriken, soyut sınıf hiyerarşi için gerekli durumu sağlar.
  - Soyut sınıf genelde devir aldığı soyut metotlara gerçekleştirme vermez sadece set/get metotlarını sağlar,
  - Bazen soyut sınıfların arayüzden devraldıkları metotlara genel-geçer bir davranış sağladığı da görülür.

Shape.java

- Bu örnekte bir Shape hiyerarşisi ele alınmaktadır.



Arayüz ve Kurucular

- Arayüzlerin kurucuları yoktur.
  - Bu, arayüzlerin hiç durum sağlamıyor olmalarından dolayı anlaşılabilir.

- Bu yüzden arayüzü gerçekleştiren sınıfların kurucularında, arayüzden kaynaklanan super() çağrıları bulunmaz.

- Alt sınıfların kurucularındaki super() çağrıları, miras devraldığı sınıflar içindir.

Arayüzler Arasında Miras - I

- Arayüzler de birbirlerinden miras devralabilirler.
  - Bu durumda bir arayüz, kendisinden miras devraldığı diğer arayüzü genişletir (extends).
- Arayüzler arasındaki miras ilişkisinde **extends** anahtar kelimesi kullanılır.
- Bu durumda alt arayüzü gerçekleştiren sınıfın, arayüzün devraldığı metotlara da gerçekleştirme vermesi gerekir.

```
public interface Worker{
    void work();
}

public interface HardWorker extends Worker{
    void workHard();
}
```

Worker.java ve HardWorker.java

- *org.javaturk.oofp.ch03.interfaces.factory.extension* paketi

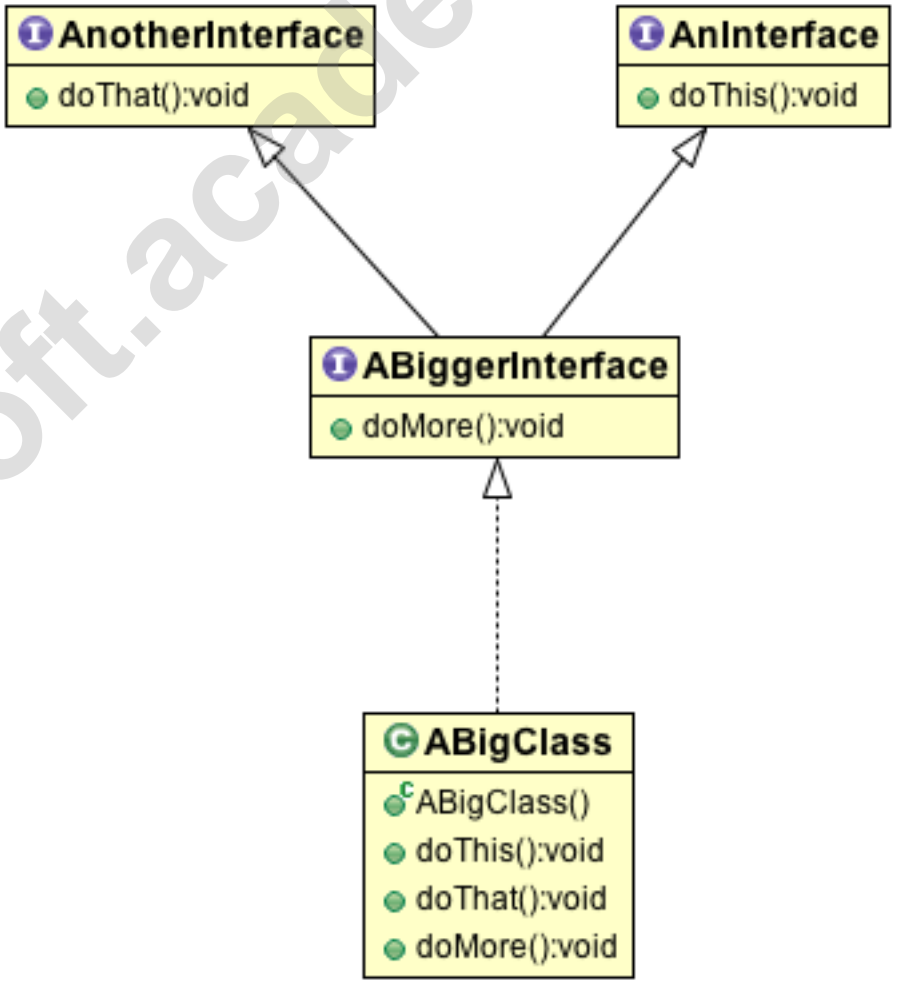
Arayüzler Arasında Miras - II

- Bir arayüz, aynı andan birden fazla arayüzden miras devir alabilir.

- Bu durumda miras alınan arayüzler, ard-arda virgül ile ayrılarak sıralanır.

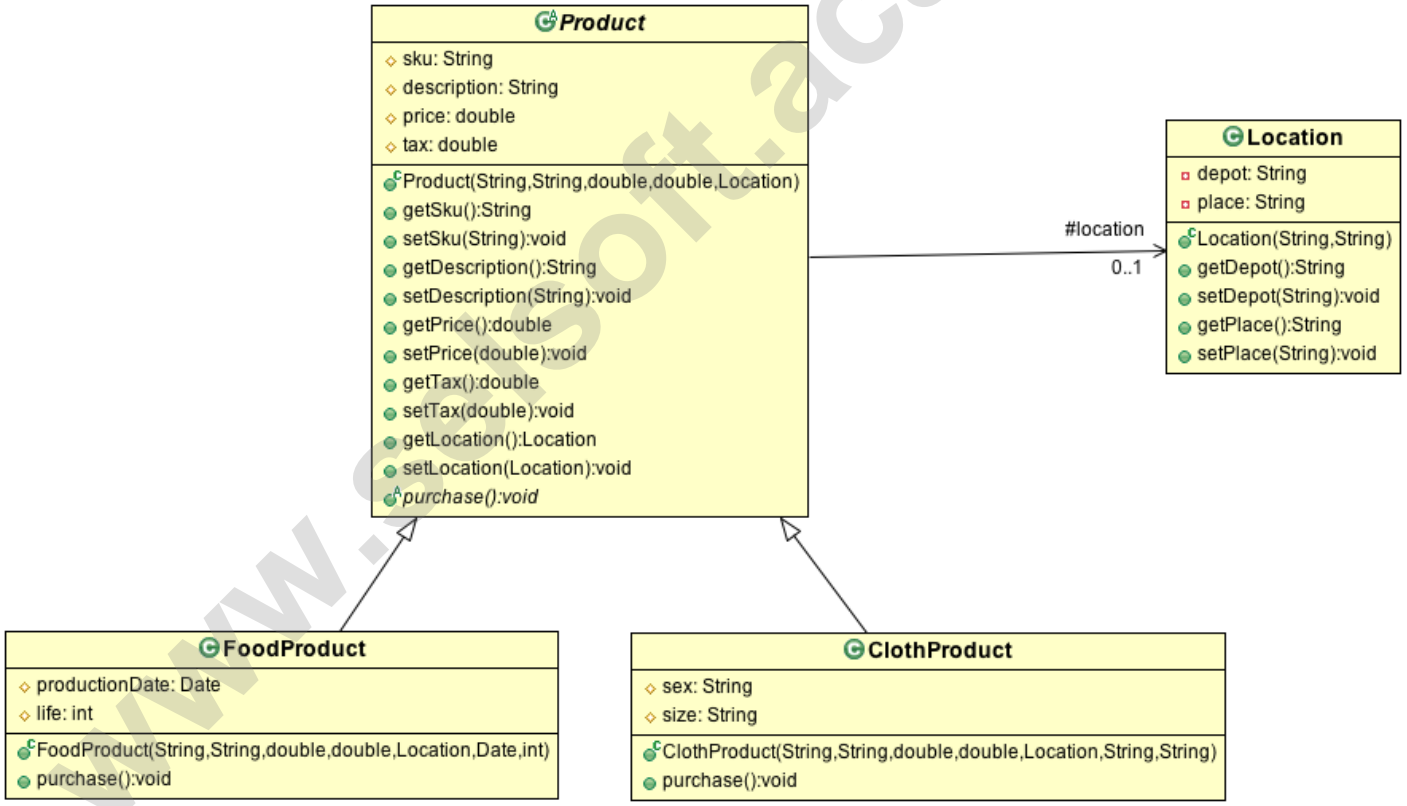
ABiggerInterface.java

- *org.javaturk.oofp.ch04.extending* paketi.



Uygulama - I

- Bir kurum ticaretini yaptığı ürünlerle ilgili olarak yazılım sisteminde şöyle bir yapı kurgulamıştır:



Uygulama - II

- Sonrasında bu kurumun iş yapışında aşağıdaki değişiklikler olmuş ama var olan Product yapısının bunlara izin vermediği gözlemlenmiştir.
  - Danışmanlık hizmeti sağlama (Location'u yok),
  - Kurumdaki eskimiş bilgisayarları satma (Product değil),
  - Ürettikleri ürünlerden özel günlerde hediye verme (Product'ın fiyatı var).
- "Kurum, bu tür değişiklikleri daha sistemi tasarlarken öngörerek baştan daha esnek bir sistem kurgulanmalıydı" diyorsanız, bu sistemi baştan bu şekilde tasarlayın.
- Çözümünüzün sınıf diyagramını çizin ve kodunu yazın.

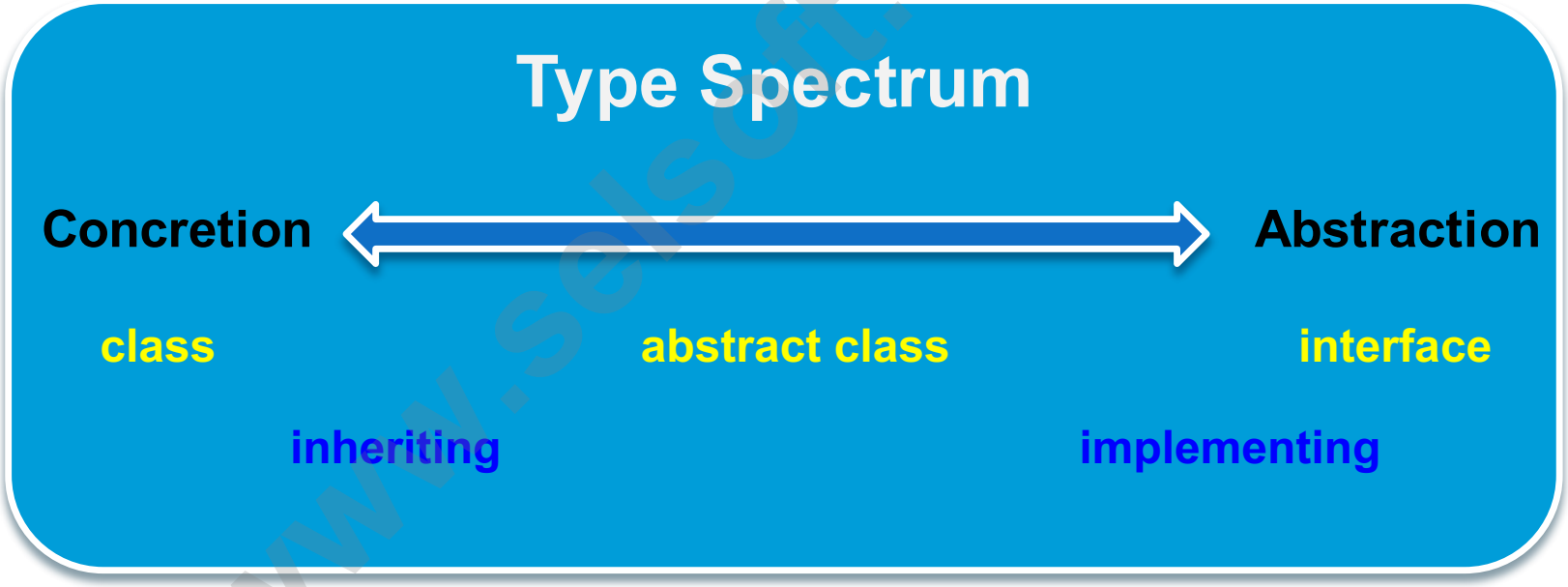
Tip Kavramı (Concept of Type)

Tip Kavramı

- Programlama dillerinde **tip** (**type**), alınabilecek değerler kümesidir (set of possible values).
- Java'da ilkel (primitive) ve karmaşık ya da referans (complex ya da reference) tipler vardır.
- Karmaşık tip ile başında bu yana kavramıyla hep somut sınıfları kastettik.
- Artık, tip olarak somut sınıflar yanında soyut sınıflar ve arayüzlere de sahibiz.

Tip Spektrumu

- Soyut ve somut olarak sınıflar ile arayüzler düşünüldüğünde, bir tip spektrumdan bahsedilebilir.
- Bu spektrumda farklı tipler farklı yerlere sahiptirler.

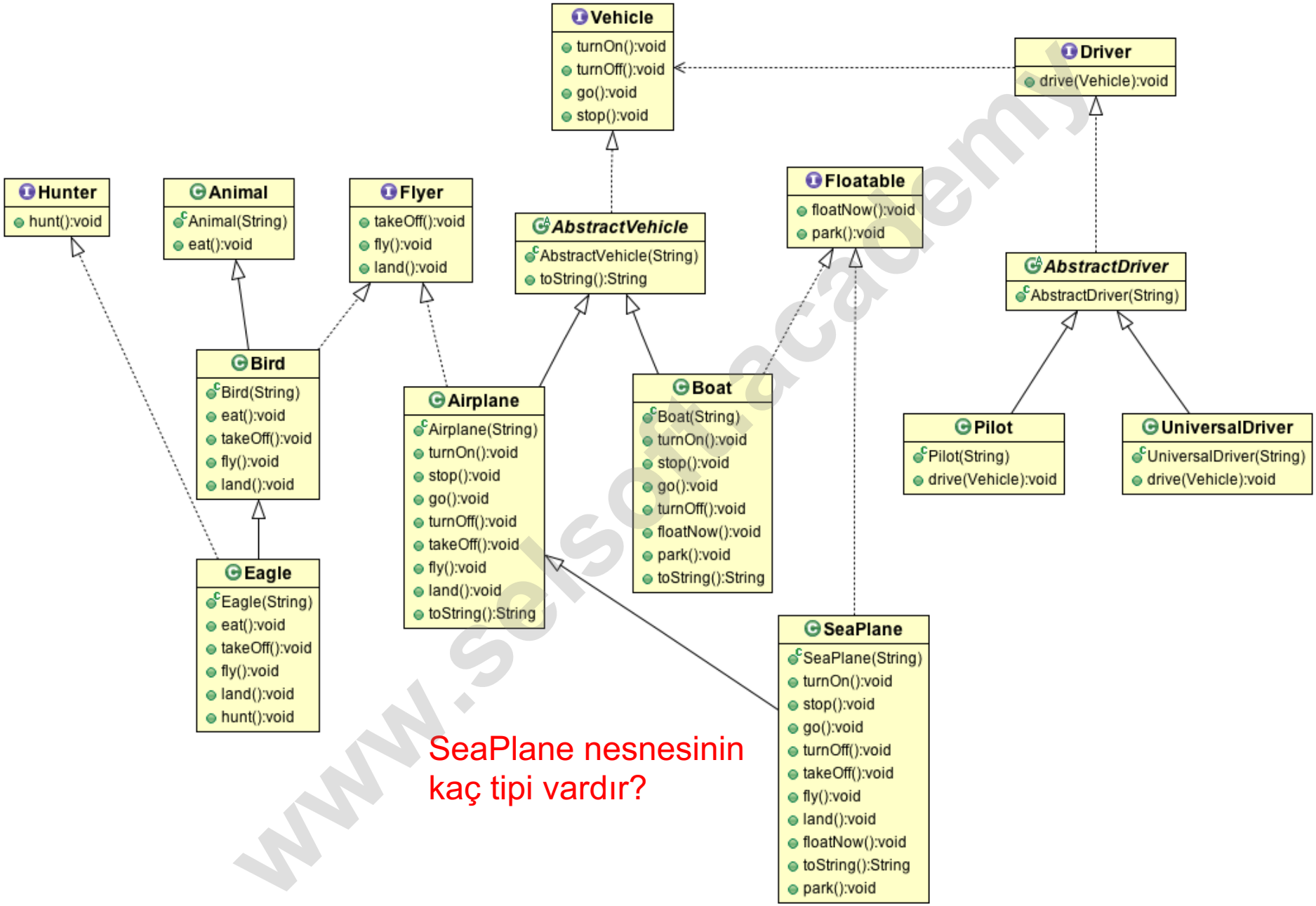


Çok Tipli Yapılar

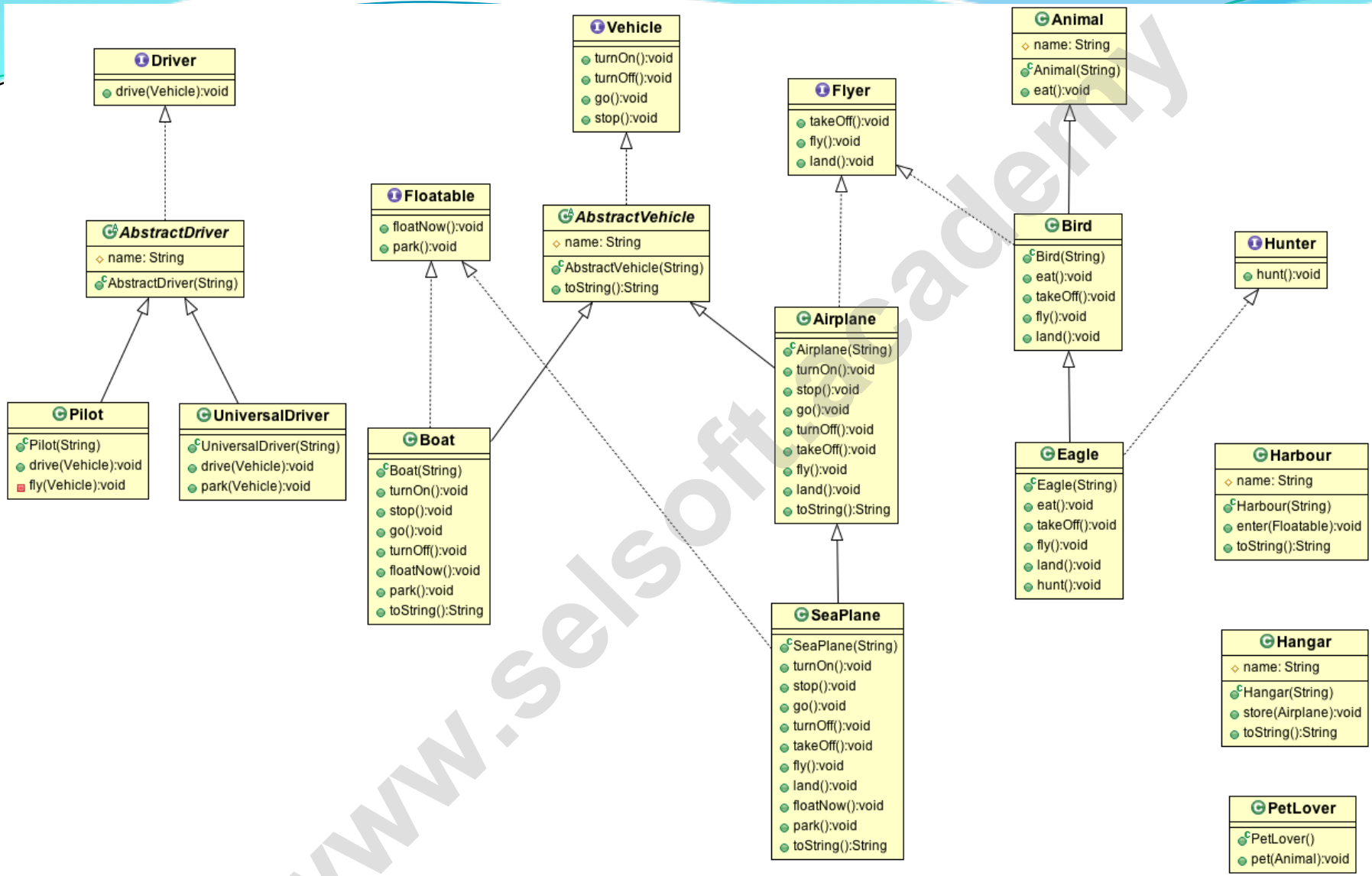
- Bir hiyerarşide, somut-soyut sınıf ve arayüz olarak birden fazla üst tipe sahip olan bir nesnenin birden fazla tipi var demektir.

- Ve nesne, o tiplerin her birisinden olan referanslar ile temsil edilebilir.

- Aynı nesnenin hangi bağlamda hangi tipini ortaya koyduğunu belirleyen şey de o nesneyi o bağlamda temsil eden referanstır.



SeaPlane nesnesinin kaç tipi vardır?



TestFlyer.java

- *org.javaturk.oofp.ch04.flyer* paketi.

Polymorphism

- Arayüzlerin de miras hiyerarşisine katılması sayesinde, çok tipli nesneler oluşturulabilir.

- Nesnelerin gerçekleştirmesi bir tanedir, o da sınıfıdır. Ama nesnelerin pek çok tipi vardır.
  - Bu sebeple bir sınıfın nesnesi, sahip olduğu her tipe dönüştürülebilir (upcasting).
  - Aynı nesne bu şekilde farklı bağlamlarda farklı tipte görünür.

- Bu ise çok şekilliliğin (polymorphism) tanımıdır.
- Bu durum, gerçekliği daha iyi resmetmeye yarar.

Geri Çağırma Metotları (Callback Methods)

Geri Çağırma (Call Back) - I

- Yazılım sistemlerinde sıklıkla, bir buttonun tıklanması ya da bir kullanıcının sistemi kullanmaya başlaması (login) gibi bazı olayların takibi gereklidir.
- Bu amaçla genel olarak, olayın kaynağı olan nesneye, olayın olduğunu bildirmesi için bir fonksiyon geçilir.
- İlgilenilen durum oluştuğunda da nesne, kendisine geçilen fonksiyonu çağırır.
- Bu mekanizmaya geri çağırma (callback), geri çağrılan metoda da geri çağırma metodu (callback method) denir.
- **Observer** (event-notification ya da publisher-subscriber) tasarım kalıbı bu problemi ve çözümünü tarif eder.

Geri Çağırma (Call Back) - II

- Geri çağırmayı Java'da kurgulamak için olayın kaynağına, fonksiyon değil, üzerinde belirli bir metot olan nesne geçilir.
  - Çünkü Java nesne merkezlidir ve nesne geçilmesi durumu çok daha geniş bir hareket alanı sağlar.

- Bu durumda olayın kaynağı olan nesnenin, olayın olması durumunda hangi metodu çağıracağını bilmesi gerekir.

- Bu sebeple Java'da geri çağırma nesneleri, üzerinde genelde bir tane geri çağırma metodu bulunduran arayüzlerden türetilir.

TimerExample.java

- *org.javaturk.oofp.ch04.callBack* paketi.

İsimsiz Sınıflar (Anonymous Classes)

İsimsiz Sınıflar

- Genelde geri çağırma metotlarının üzerinde bulunduğu sınıfların tek kullanımlık bir nesnesine ihtiyaç duyulur.
  - Yani arayüzü gerçekleştiren sınıfın bir tek nesnesine ihtiyaç vardır ve bu nesne sadece bir yerde kullanılır.

- Bu durumda Java, arayüzü yerine getiren sınıfın isimsiz bir şekilde, hızlıca oluşturulmasına ve bunun yapıldığı yerde tek bir nesnesinin yaratılıp kullanılmasına izin verir.

- Bu şekilde oluşturulan sınıflara isimsiz sınıf (anonymous class) denir.

İsimsiz Sınıflar - I

- İsimsiz sınıflar sıklıkla olayları (event) yakalamada kullanılırlar.
  - Çünkü çoğu zaman özel bir duruma işaret eden olay nesnesi sadece bir yerde yakalanır ve gereği yapılır.

```
Timer t = new Timer(1_000, new ActionListener() {

    @Override
    public void actionPerformed(ActionEvent e) {
        ...
    }
});
```

- Örnekteki **`ActionListener`**, sadece **`actionPerformed()`** metoduna sahip bir arayüzdür.

TimerExample.java

- *org.javaturk.oofp.ch04.anonymous.timer* paketi.

İsimsiz Sınıflar - I

- Yakalanan sıklıkla GUI olaylarıdır..
- Benzer şekilde çoğu zaman bir GUI bileşeninin durumundaki bir değişikliğe işaret eden olay nesnesi sadece bir yerde yakalanır ve gereği yapılır.

```
button.setOnAction(new EventHandler<ActionEvent>() {

        @Override
        public void handle(ActionEvent event) {
                ...
        }
});
```

- Örnekteki **`EventHandler`**, sadece **`handle()`** metoduna sahip bir arayüzdür.

MyApplication.java

- *org.javaturk.oofp.ch04.anonymous.event* paketi.

İsimsiz Sınıflar - II

- Sınıflar, tanımlamaya (class declaration) sahip oldukları halde isimsiz sınıflar ifadedirler (expression).
- İsimsiz sınıf ifadesi, bir kurucu çağrısına benzer ama içinde tekrar tanımlanan (override) metot ya da metotlar vardır.
- İsimsiz sınıflar hem arayüzleri gerçekleştirmede hem de sınıfları genişletmede kullanılabilirler.
- İsimsiz sınıflar genelde sadece bir metota sahip arayüzleri gerçekleştirmede kullanılmalarına rağmen birden fazla metodu yeniden tanımlayacak şekilde kullanılabilirler.
  - Olay yapılarında çağrılacak metot bir tane olduğundan, genelde tek metodu tekrar tanımlamada kullanılırlar.

İsimsiz Sınıflar - III

- İsimsiz sınıf ifadesi şöyledir:
  - new operatörü,
  - Gerçekleştirilecek arayüzün ya da genişletilecek sınıfın ismi,
    - new operatöründen sonra gelen tipin sınıf olması durumunda, kurucuya geçilecek parametreler de sıralanabilir.
    - Eğer tip arayüz ise, arayüzlerin kurucuları olmadığından, sanki varsayılan kurucu çağrılıyormuş gibi içi boş iki parantez bulunur.
  - Sınıf bloğu.
- İsimsiz sınıflar birer ifade olduklarından, bloklarında başka ifadeler olamaz, sadece metot gibi başka bloklar olabilir.
- İsimsiz sınıf ifadesi, arayüz gerçekleştirmesinde new operatöründen sonra arayüzün varsayılan kurucusunu çağırıyor bir görüntüye sahip olduğundan tuhaf görünür.

```
public interface DoerInterface {

        void doIt();

        void doThat();
}
```

```
new DoerInterface(){
    {
        System.out.println("Instance initializer block.");
    }

    @Override
    public void doIt(){
        System.out.println("I!ll always do it :)");
    }

    @Override
    public void doThat(){
        System.out.println("I'll always do that :)");
    }
}).doIt();
```

AnonymousDoesClassTest.java

- *org.javaturk.oofp.ch04.anonymous.doer* paketi.

İsimsiz Sınıflar - IV

- İsimsiz sınıflar, içinde bulundukları sınıfın üyelerine erişebilir.
- İsimsiz sınıflar, içinde bulundukları bloğun yerel değişkenlerine final ya da değeri değişmediği (effectively final) hallerde ulaşabilir.
  - Bu durumda da yerel değişkeni değiştiremez.
- İsimsiz sınıflar, sabite olmaları şartıyla statik alanlar tanımlayabilirler.

İsimsiz Sınıflar - V

- İsimsiz sınıflar ayrıca şunları tanımlayabilirler:
  - Alanlar,
  - Yerel sınıflar (local classes),
  - Üst tipinde olmayan metotlar,
  - Nesne ilk değer blokları
- İsimsiz sınıflar, statik ilk değer atama blokları ile üye arayüzler tanımlayamazlar.

WeirdAnonymousDoesClassTest.java

- *org.javaturk.oofp.ch04.anonymous.doer* paketi.

Java SE 1.8 Yenilikleri

Java SE 1.8 Yenilikleri

- Java SE 1.8 ile arayüzlere (interface) iki ciddi değişiklik yapıldı.
- Bu değişiklikler şunlardır:
  - default (varsayılan) metotlar,
  - static metotlar

- Bu iki tür metot da, şu ana kadar söylenen "arayüzler sadece şekil/form sağlar, gerçekleştirme sağlamaz" prensibini delen yeniliklerdir.

- Bu iki durumun da önemli sebepleri vardır.

Varsayılan Metot (Default Method)

Arayüz ve Gerçekleştirme

- Daha önce “Arayüzler sadece form – şekil sağlayan yapılardır, içerik sağlamazlar.” dendi.
- Java SE 8 ile birlikte arayüzlerde ciddi değişiklikler yapıldı.
- Bu değişikliklerden ikisi arayüzlerin sadece şekil yani metot arayüzü değil aynı zamanda içerik yani gerçekleştirme de sağlamasına izin verdi.

Arayüzü Güncellemek

- Yazılan kodlarda bulunan arayüzlerin, tabi olarak, sistemdeki sınıflar gibi zaman zaman değişmesi gerekir.
- Bir arayüze yeni bir metot eklendiğinde, o arayüzden miras devralan bütün sınıfların o metodu gerçekleştirmeleri gerekir.
- Bu durum özellikle kütüphaneler (library) için problemlidir:
  - Dünyaya dağılmış olan bir kütüphanedeki bir arayüze metot eklemek, o kütüphaneyi kullanan bütün kodları etkiler.
  - Kullanıcıların kütüphaneyi yeni sürüme çekmeleri, o arayüzü kullanan sınıflara, yeni metotları eklemelerini gerektirir.

Default Method–Varsayılan Metot - I

- Java SE 8 ile birlikte arayüzlere default method (varsayılan metot) ekleme imkanı geldi.

- Default metot, "**default**" kelimesiyle tanımlanır ve arayüzde bir gerçekleştirmeye sahip olur.

```
public interface Worker{
    void work();
    double calculateSalary();
}
```

```
public interface Worker{
    void work();
    double calculateSalary();

    default void newMethod(){
        ...
    }
}
```

Default Method–Varsayılan Metot - II

- Dolayısıyla, eklenen ve default olarak tanımlanan yeni metodun, arayüzün alt sınıflarında gerçekleştirilmesine gerek kalmaz.

- Diğer tüm metotlar gibi varsayılan metotlar da daima public olarak tanımlıdırlar.

Binary Compatibility - I

- Varsayılan metot, bir arayüzünün eski sürümünü kullanan bir yapıyla, yeni sürümü arasındaki uyumluluğu bozmamak üzere çıkarılmış bir özelliktir.

- **Binary compatibility** (ikili? uyumluluk), eskiden birlikte çalışan iki yapıdan birisinin değişmesine rağmen, yeni sürümünün diğer yapıyla hala uyumlu bir şekilde çalışabilmesine denir.
  - Yani uyumluluk, derlenmiş yapılar, .class dosyaları seviyesindedir.

Binary Compatibility - II

- Eğer bir projedeki bir arayüze yeni bir metot ekledikten sonra derleyip projede eklerseniz, var olan yapılar, arayüzün yeni sürümüyle çalışmaya devam edeceklerdir.

- Ama o yapılar, arayüze eklenen yeni metodu kullanamayacaklardır.

- Buna binary compatibility denir.

TestCompatibility.java - I

- FootballPlayer arayüzünün 1. sürümü ile çalışan AverageFootballPlayer ve TestCompatibility, FootballPlayer'a yeni bir metot eklenmesiyle oluşan 2. sürümü ile de çalışmaya devam eder.

1. Sürüm

```
public interface FootballPlayer{
    void play();
}
```

2. Sürüm

```
public interface FootballPlayer{
    void play();
    void behaveEthically();
}
```

```
public class TestCompatibility {
    public static void main(String[] args) {
      FootballPlayer player = new AverageFootballPlayer();
      player.play();
    }
}
```

Source Compatibility

- Bir önceki örnek **source compatibility** (**kaynak uyumluluğu**) sağlamaz, çünkü yenilenen arayüzü kullanan yapılar tekrar derlenirse, var olan metodu implement etmeleri gerekir.

- Varsayılan metotlar, bu durum için geliştirilmişlerdir.

- Eğer yeni eklenen metot varsayılan metot ise, bu yeni metodun, alt sınıflar tarafından gerçekleştirilmesine gerek kalmaz.

TestCompatibility.java - II

- Ama TestCompatibility, FootballPlayer'a yeni eklenen bir metodu AverageFootballPlayer nesnesi üzerinde çağıramaz, çünkü AverageFootballPlayer tekrardan derlenmelidir.

- AverageFootballPlayer sınıfı, yeni eklenen metodu gerçekleştirmeden tekrar derlenebilir çünkü yeni eklenen metot varsayılan metottur.

Neden Varsayılan Metot?

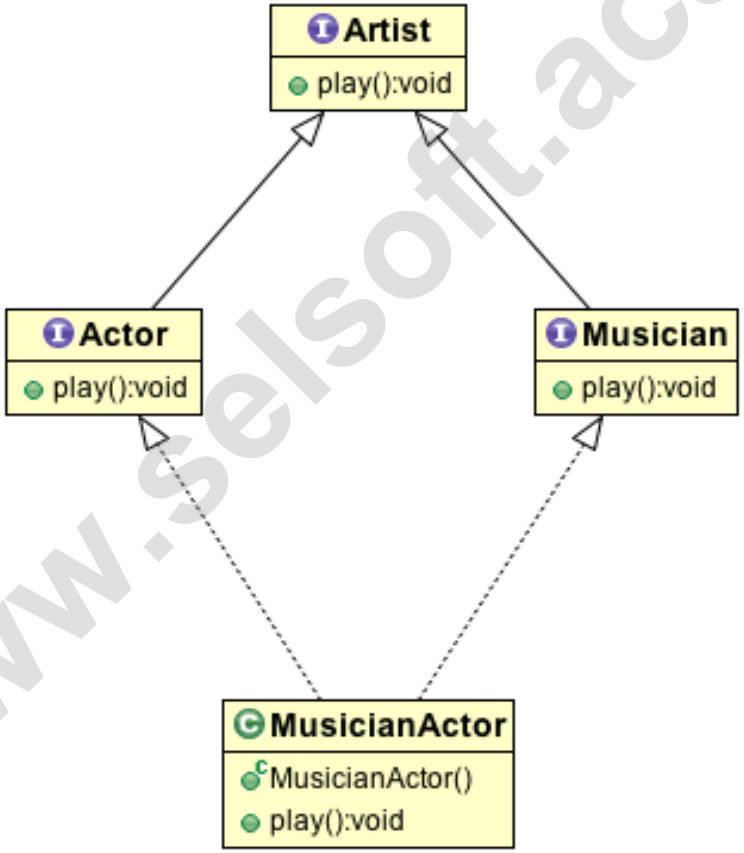
- Varsayılan metot, bir arayüzün eski sürümünü kullanan bir yapıyla, yeni sürümü arasında kaynak uyumluluğu sağlamak üzere çıkarılmış bir özelliktir.

- Arayüzün yeni sürümünde gelen yeni metotların varsayılan metot olarak tanımlanması halinde, o arayüzleri gerçekleştiren sınıfların eklenen yeni metodu gerçekleştirmelerine gerek kalmaz.

- Bu durum arayüzü ile gerçekleştiren sınıflar arasında "geriye dönük uyumluluk" (backward compatibility) sağlar.

Çoklu Miras - I

- Eğer bir sınıf birden fazla arayüzden miras devralır ve o arayüzler aynı default metoda sahip olurlarsa ne olur?



Çoklu Miras - II

- Bu durum Java'da derleme hatası verir.
- Bu duruma da "diamond problem" ya da "deadly diamond of death" denir.
- Java'da bu hatayı çözmenin tek yolu vardır:
  - Alt sınıfın devraldığı default metodu implement etmesi

multipleInheritance Paketi

Seçenekler

- Dolayısıyla varsayılan metoda sahip bir arayüzden miras devralmak isteyen bir sınıfın önünde şu üç durum vardır:
  - Varsayılan metodu olduğu gibi devralıp, override etmemek, dolayısıyla varsayılan gerçekleştirmeyi kullanmak,
  - Varsayılan metodu abstract olarak tekrar tanımlayıp, abstract bir sınıf olmak,
  - Varsayılan metodu override etmek.

Soyut Metot Default Gerçekleştirme

- Bir arayüz, diğer arayüzden extends ile miras yoluyla devraldığı bir metoda gerçekleştirme verip onu default olarak tanımlayabilir.
- Bu şekilde soyut olarak devralınan metot, devralan arayüzde gerçekleştirilmiş olur.

```
public interface Artist {

    void play();
}
```

```
public interface Actor extends Artist{

    default void play(){
        System.out.println("Actor plays!");
    }
}
```

Default Gerçekleştirme Soyut Metot

- Bir arayüz, diğer arayüzden extends ile miras yoluyla devraldığı bir default metodu tekrar default ya abstract olarak tanımlayabilir.

```
public interface Artist {

    default void play(){"Artist plays!"}
}
```

```
public interface Musician extends Artist{

        void play();
}
```

```
public abstract class Pianist implements Musician{

}
```

multipleInheritance.v2 Paketi

Garip Bir Durum!

- Davranışların tam olarak tanımlandığı (definition) yer değildir, davranışlar arayüzlerde tanıtılır, sınıflarda tanımlanır.
- Bu yüzden bu yapılara, sadece arayüz sağladığı için "arayüz" ya da "interface" denir.
- Ama yukarıdaki durumda tanımlamayı yapan arayüzdür, tanıtımı yapan ise sınıf!
- "`play()`" metodu Artist arayüzünde tam olarak tanımlanıyor ama hiyerarşinin en altındaki Pianist sınıfı ise aynı davranışı tanıtıyor, yani üst tipi olan Mucisian arayüzünden abstract olarak devralıyor ama bir gerçekleştirme vermiyor.

Bir Nokta

- `java.lang.Object` sınıfındaki (`hashCode()`, `toString()` vb.) metotlar asla bir arayüz üzerinde varsayılan olan tanımlanamaz.

- Eğer böyle olsaydı, o arayüzün tüm alt sınıflarında bu gerçekleştirme kullanılırdı.

- Ama bu mümkün değildir, mümkün olan bu metotları hiyerarşinin üstündeki sınıfta override etmek ve alt sınıfların aynen kullanmasına izin vermektir.

Java API Arayüz Varsayılan Metotları

- Java API'sinde varsayılan metotlara sahip arayüzlerden bazıları şunlardır:
  - `java.util.Collections`
  - `java.sql.Statement`

Statik Metot (Static Method)

Arayüzde Statik Metot - I

- Java SE 8 ile birlikte arayüz üzerinde statik metot tanımlanabilir.
  - Bunun için yine "**static**" anahtar kelimesi kullanılır.

```
public interface Printable {

    void print();

    default void format(){
        System.out.println("Printable is being formatted.");
    }

    static void startPrinting(){
        System.out.println("Printing has been started.");
    }
}
```

Arayüzde Statik Metot - II

- Diğer tüm arayüz metotları gibi statik metotlar da daima public olarak tanımlıdırlar.

- Arayüz üzerindeki statik metotlar arayüzü genişleten arayüzler ya da gerçekleştiren sınıflar tarafından devralınmazlar, bu yüzden statik metotlar sadece ve sadece tanımlandığı arayüz üzerinde çağrılırlar.

staticMethods Paketi

Neden Statik Metot? - I

- Java SE 8'e kadar statik metotlar sadece sınıflarda tanımlanabilirdi.

- Bu yüzden, arayüzlerde ifade edilen soyut nesne metotlarına yardımcı mahiyetteki araçsal metotlar (utility methods), ancak araçsal sınıflarda (utility classes) statik olarak tanımlanırdı.

  - Projelerde aşırı miktarda statik metoda sahip olan araçsal sınıflar çok yaygındır.
  - Bu tür araçsal sınıfların birliktelikleri (cohesion) düşük, bağımlılıkları (coupling) ise yüksek olmaktadır.

Neden Statik Metot? - II

- Örneğin
  - **`java.util.Collection`** arayüzü ve **`Collections`** araçsal sınıfı.
  - **`java.nio.file.Path`** arayüzü ve **`Paths`** araçsal sınıfı.
  - **`java.util.Arrays`** araçsal sınıfı.
- Arayüzlerde tanımlanabilen statik metotlarla bu durum ortadan kalkacaktır.
- Statik metotlar artık arayüzler üzerinde bulunabilecektir.
- Bu şekilde statik metotların az sayıdaki araçsal sınıf üzerinde toplanması engellenecek ve moduler yapılar kurgulanabilecektir.

Java API Arayüz Statik Metotları

- Java API'sindeki statik metotlara sahip arayüzlere örnek olarak şunlar verilebilir:
  - `java.util.Comparator`

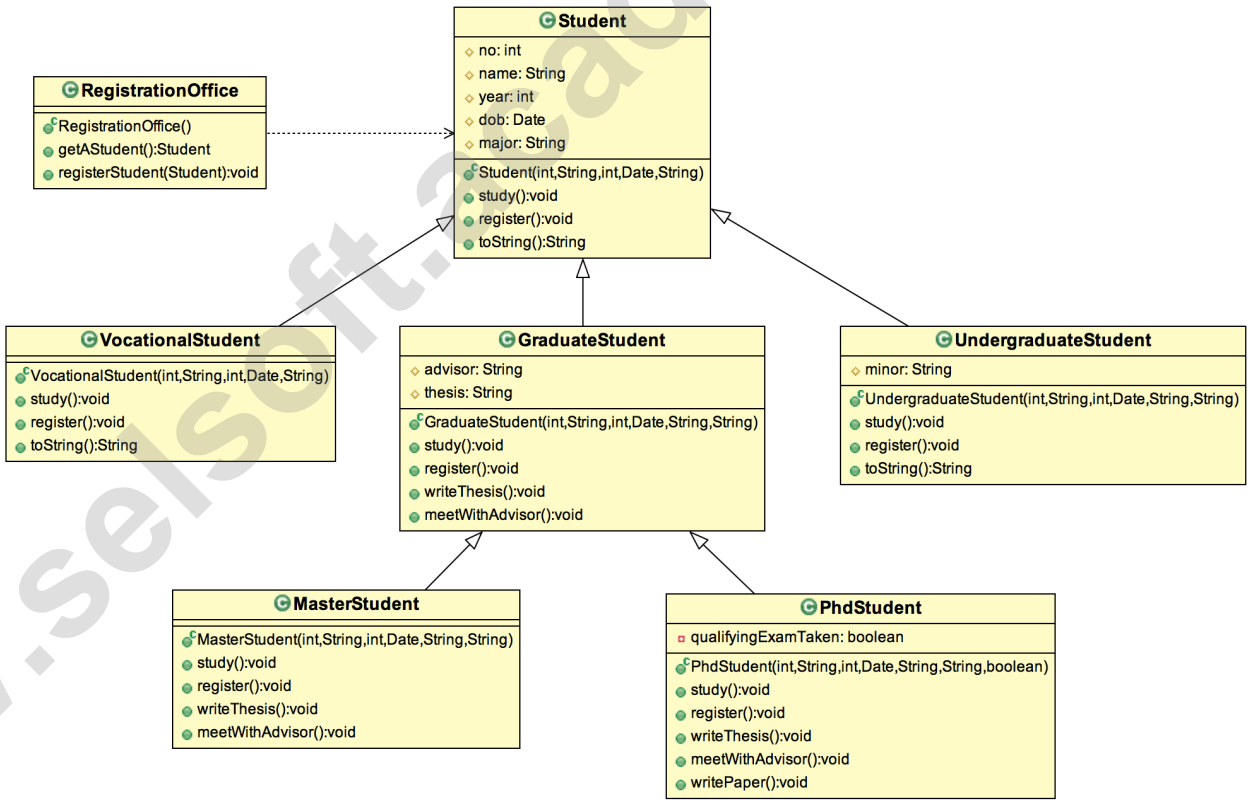
Özet

- Bu bölümde, bir arayüz (interface) yapısı ele alındı.
  - Arayüzlerin yazılımı tasarımında nasıl kullanılacağı incelendi.
  - Çok tipli yapılar ele alındı.
  - Java SE 8 ile gelen yenilikler incelendi.
    - Varsayılan (default) metotlar
    - Statik metotlar

Ödevler

Ödevler I

- 3. Bölüm uygulamalarında kurguladığınız yandaki hiyerarşiyi, Student bir arayüz olacak şekilde değiştirin.
- Bu durumda AbstractStudent isimli yeni bir soyut sınıfa ihtiyaç duyulacağını gözlemleyin.



Ödevler II

- Aşağıda sıralanmış farklı yetkinliklere sahip mühendisleri (engineers) düşünün.
  - Blogger
  - Parent
  - TeamFan
  - InstrumentPlayer
  - JavaEnthusiast
- Bu tür yetkinliklere farklı kombinasyonlarla sahip olan nesneleri nasıl oluşturacağınızı tartışın.
  - Çözümünüzün sınıf diyagramını çizin ve kodunu yazın.