Java ile Nesne Merkezli ve Fonksiyonel Programlama

2. Bölüm
Çok Şekillilik (Polymorphism)

Akın Kaldıroğlu

www.javaturk.org

Kasım 2015

Küçük Ama Önemli Bir Konu

- Bu dosya ve beraberindeki tüm, dosya, kod, vb. eğitim malzemelerinin tüm hakları Akın Kaldıroğlu'na aittir.
- Bu eğitim malzemelerini kişisel bilgilenme ve gelişiminiz amacıyla kullanabilirsiniz ve isteyenleri http://www.javaturk.org 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@javaturk.org 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 bölüm, nesne-merkezli programlamanın en temel kavramların olan çok şekilliliği (polymorphism) ele alacaktır.

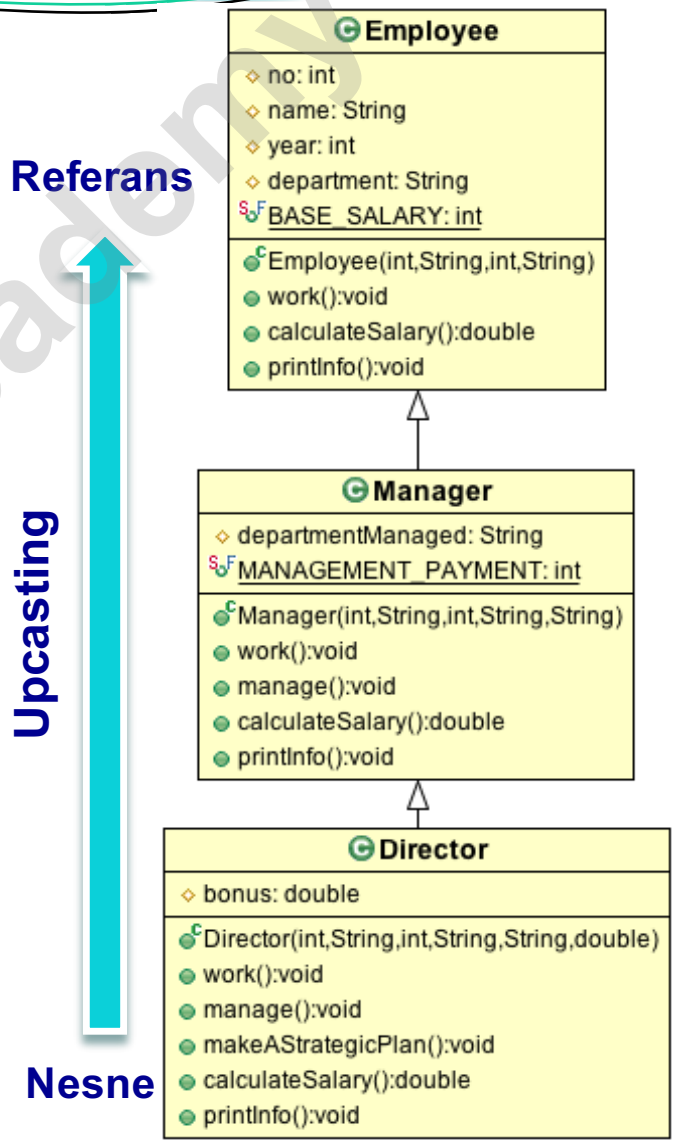
- Java'nın tipleri arasındaki mümkün dönüşümler (cast) incelenecektir.

Upcasting (Yükseltme)

Upcasting - I

- Bir referansı ya da nesneyi alıp onu üst tipinden bir referansa atamaya **upcasting** (**yükseltme**) denir.
- Upcasting, yerine geçebilme özelliğinden dolayı her zaman güvenlidir.
  - Dolayısıyla cast operatörü "()" kullanmaya gerek yoktur.

```
Manager m = new Manager();
Employee e = m;
ya da doğrudan
Employee e = new Manager();
Manager m = new Director();
Emlpoyee e = new Director();
```



Upcasting - II

- Hatırlayın, kalıtım bir genelleştirme-özelleştirme ilişkisi kurgular.
- Hiyerarşide aşağıda yer alan yani daha özel tiplerden olan nesneler, yerine geçebilme (substitutability) özelliğinden dolayı, yukarıda yer alan yani daha genel olan tiplerin referanslarına atanabilir.
- Bu durumda her özel tipin nesnesi, aynı hiyerarşideki daha genel tipten olan referanslara atanabilir.

```
Employee e = new Employee();
e = new Manager();
e = new Director();
Manager m = new Director();
```

Neden Upcasting?

- Dil, tabiatı itibariyle genel ifadeler ile daha çok şey anlatma eğilimindedir.
- Upcasting de program içerisinde daha genel referanslarla, her türlü alt tipten olan nesneyi gösterme yeteneği sağlar.
  - Employee e ifadesi "herhangi bir çalışan" anlamına gelir.
  - Manager m ifadesi de "herhangi bir yönetici" anlamına gelir.
- Bu ise programlarımızı basitleştirir.

```
Employee e = new Employee();
Manager m = new Manager();
Director d = new Director();
e = m;
e = d;
m = d;
```

Metot Parametrelerinde Upcasting

- Upcasting, sıklıkla metot parametrelerinde de görülür.
- Bu durum, bir metoda, daha genel tipten parametre almasına rağmen, o tipin tüm alt tiplerinden parametre geçilerek çağrılmasıylaa oluşur.
- paySalary() metodunun, parametre olarak Employee alması demek, kendisine her tür Employee nesnesinin geçilebilmesi demektir.

```
public class PayrollOffice {
  public void paySalary(Employee e){
    double salary = e.calculateSalary();
    System.out.println("Paying a salary of " + salary +
                       " to " + e.getName());
  }
}
```

TestPayrollOffice.java

Çok Şekillilik (Polymorphism)

Polymorphism - I

- Eski Yunanca'da poly çok, morph ise şekil demektir.
- **Polymorphism** de **çok şekillilik** demektir.
- Çok şekilli olan ise referanslardır.
- Polymorphism, bir referansın, zamanın farklı anlarında, kendi ya da alt tiplerinden olan farklı nesneleri gösterebilmesine denir.
  - Örnekteki e ve m referansları polymorphictir.

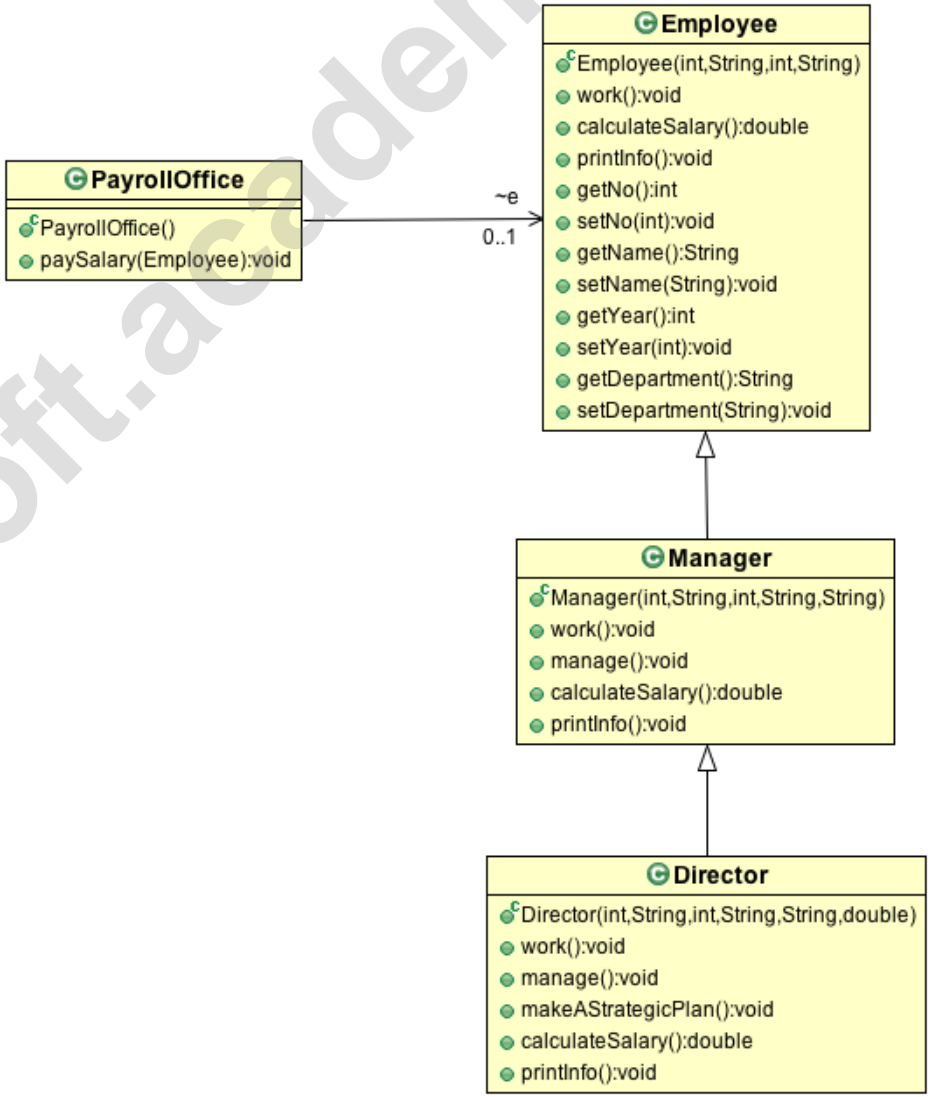
```
Employee e = new Employee();
Manager m = new Manager();
Director d = new Director();
e = m;
e = d;
m = d;
```

Polymorphism - II

- Polymorphism bir yaklaşımdır, upcasting ise onu gerçekleştiren mekanizmadır.
- Polymorphism sayesinde arayüz ile gerçekleştirmeyi ayırabiliriz.
- Referans, üst tipten olduğu için arayüzü, ona atanan nesneler ise, alt tiplerden olabildiğinden, gerçekleştirmeyi ifade ederler.
- Dolayısıyla, aynı arayüze sahip nesneler arasında, nesnenin gerçek tipini bilmeden, değişmeler yapabilirsiniz.
- Unutmayın, bir kalıtım hiyerarşisindeki nesneler, en azından en yukarıdaki nesnenin arayüzüne sahiptirler.

Polymorphism - III

- Bu yüzden polymorphism daha güzel bir tanımla, iki referansın birbirleriyle haberleşip, birbirlerinin gerçek tiplerini bilmemeleri demektir.
- PayrollOffice nesnesi sadece Employee nesnesini bilmekte, alt tiplerini (Manager ve Director) ise bilmemektedir.



Interface–Implementation Ayırımı - I

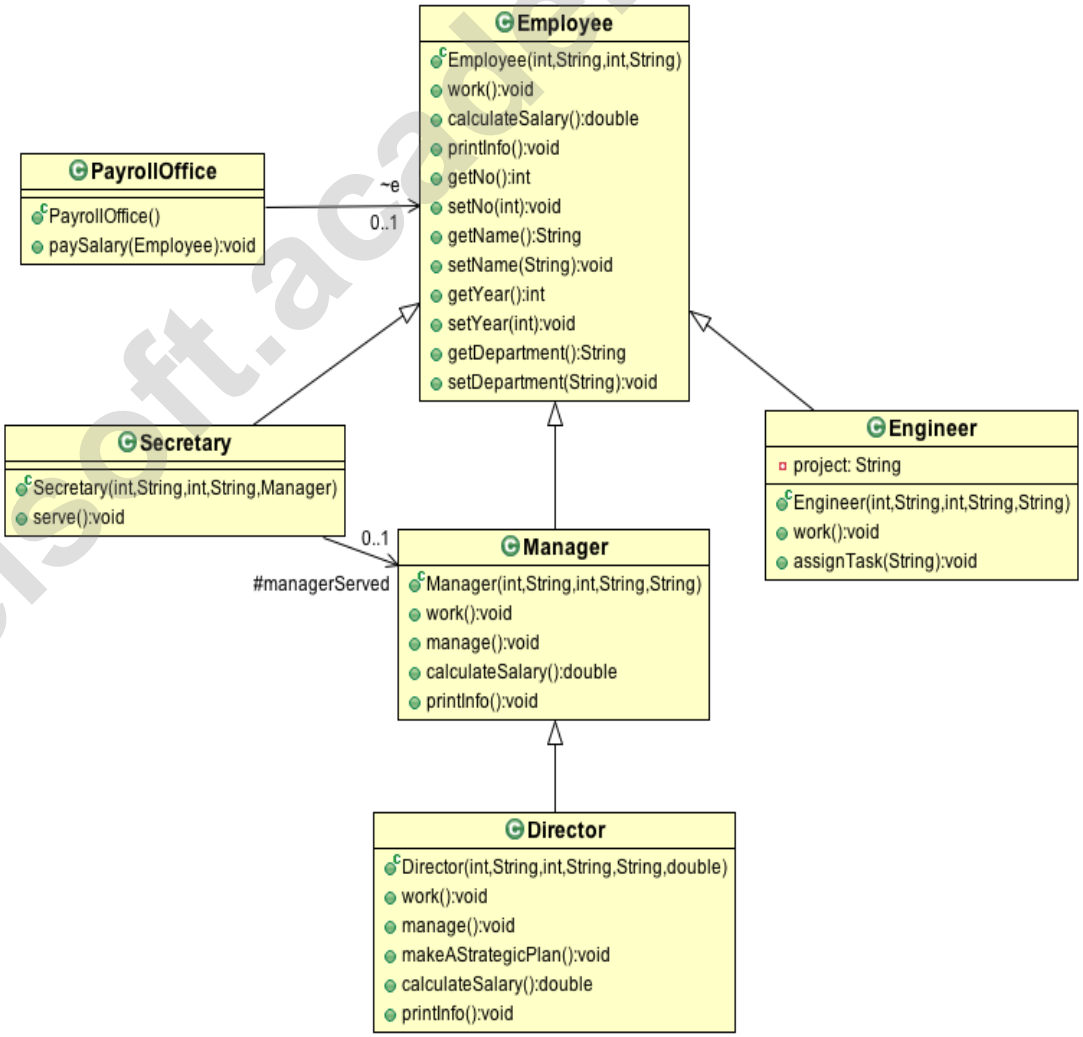
- Polymorphism sayesinde arayüz ile gerçekleştirmeyi ayırabiliriz.
- Üst tipten olan referans, tipinin arayüzünü, ona atanan nesneler ise gerçekleştirmeyi ifade ederler.
- Bu şekilde gerçekte hangi nesnenin kullanıldığını bilmeden, sadece üst tipe yani arayüze bağlı sınıflar yazılabilir.

Program to an interface not an implementation.

- Yani, elinizde bir kalıtım hiyerarşisi varsa, kodunuzu o hiyerarşinin arayüzünü belirleyen en üst tipine göre yazın, alttaki sınıfları, gerçekleştirmeleri, göz önüne almayın.
- Bu şekilde hiyerarşideki ekleme ve çıkarmalardan etkilenmezsiniz.

Interface–Implementation Ayırımı - II

- PayrollOffice, Employee hiyerarşisindeki değişimlerden etkilenmez,
- Çünkü PayrollOffice, sadece Employee'nin arayüzünü bilmektedir.



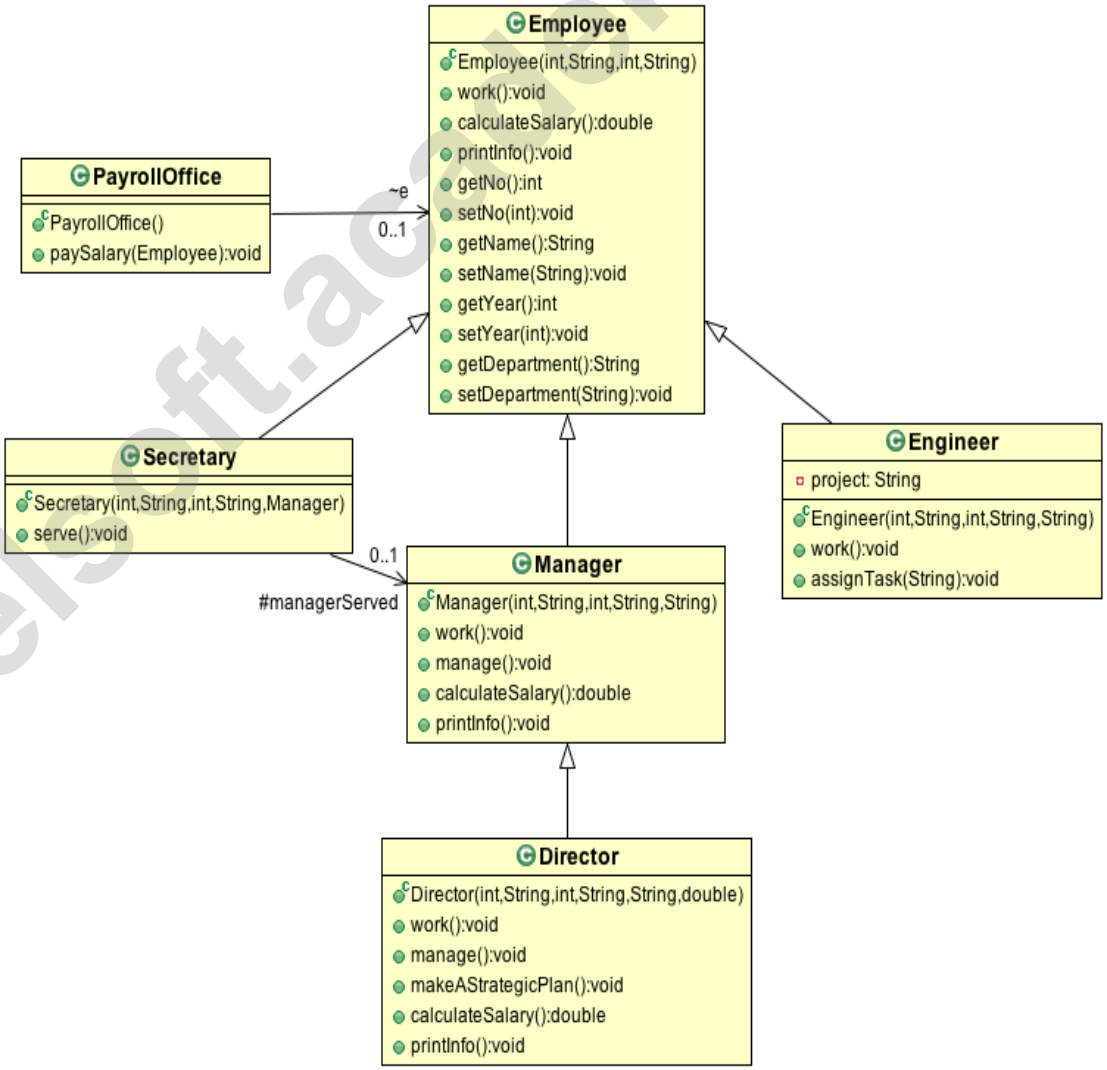
Polymorphic Metotlar

Polymorphic Metotlar - I

- Polymorphism, metotlar için de farklı bir anlama sahiptir.
- Bir metodun polymorphic yani çok şekilli olması, arayüzünün sabit olmasına rağmen, overriding sayesinde, pek çok gerçekleştirmeye sahip olması anlamına gelir.
- Dolayısıyla farklı gerçekleştirmeler, aynı arayüzün arkasında saklanabilir, birbirleri yerine geçecek şekilde kullanılabilir.

Polymorphic Metotlar - II

- Bu hiyerarşideki work() metodunu ele alalım.
- Bu metodun bir arayüzü olmasına karşın 5 tane gerçekleştirmesi vardır.
- Peki bir polymorphic metodun arayüzü ile gerçekleştirmeleri arasındaki ilişki nasıldır?



Polymorphic Metotlar - III

- Bu hiyerarşideki work() metodu farklı referanslar üzerinde çağrılırsa, hangi gerçekleştirmeler çalışır?
- Bir metodun arayüzü, o metodun kendisi üzerinde çağrıldığı referansın tipi tarafından belirlenir.
- O metodun hangi gerçekleştirmesinin çalışacağı ise referansın gösterdiği nesne tarafından belirlenir.

```
Employee e = new Employee();
e.work();
Manager m = new Manager();
m.work();
Director d = new Director();
d.work();
```

```
Employee e = new Employee();
e.work();
e = new Manager();
e.work();
e = new Director();
e.work();
```

Arayüz - Gerçekleştirme

- Employee tipinde e referansı üzerinde çağrılabilecek olan metotları belirleyen e'nin tipi olan Employee sınıfının arayüzüdür.
- Ama çalışma zamanında (run-time) hangi work() metodunun çağrılacağını belirleyen ise e'nin gösterdiği nesnenin tipidir.

```
Employee e = new Employee();
e.work();      => Employee's work()
e = new Manager();
e.work()       => Manager's work()
e = new Director();
e.work();      => Director's work()
```

TestPolymorphism.java

Başka Bir Açıdan Polymorphism - I

- Daha önce "bir metodun arayüzü, o metodun kendisi üzerinde çağrıldığı referansın tipi tarafından belirlenir. O metodun hangi gerçekleştirmesinin çağrılacağı ise referansın gösterdiği nesne tarafından belirlenir." dedik.
- Bu durumu şöyle de ifade edebiliriz:
  - Bir metodun bir referans üzerinde çağrılıp çağrılamayacağı, derleme zamanında (compile-time) belirlenir. Çünkü bu karar, referansın tipine bakılarak alınır.
    - Referansın tipinde o metot varsa çağrılabilir yoksa çağrılamaz.
  - Ama gerçekte hangi metodun çağrılacağı, referansın gösterdiği nesneye bağlı olduğundan ve bu da ancak çalışma zamanında belli olacağından, derleme zamanında bilinemez.

Başka Bir Açıdan Polymorphism - II

- Aşağıdaki kodda hangi work() çağrılır?

```
public class HR{
    public Employee getAnEmployee(){...}
}
```

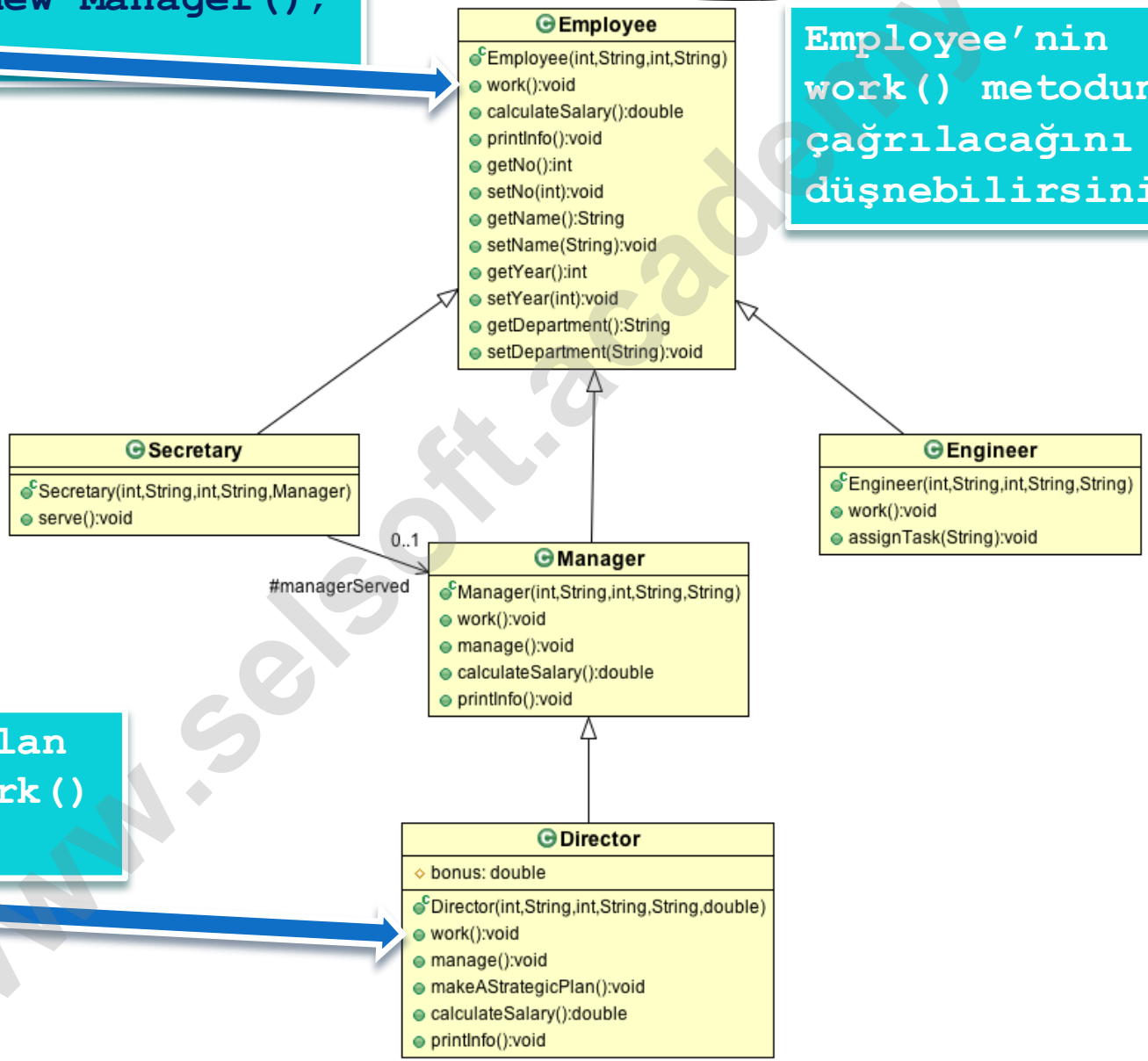
```
HR hr = new HR();
Employee e = hr.getAnEmployee();
e.work();      => Hangi work() çağrılır?
```

Başka Bir Açıdan Polymorphism - III

- Bu sorunun cevabı “bilinemez”dir.
- Koda bakarak, derleme zamanında Employee tipinden bir nesne döndürdüğünü gördüğümüz bir metodun, gerçekte Employee’nin hangi alt tipini döndürdüğünü koda bakarak bilmemiz mümkün değildir.
- Bu bilgi ancak çalışma zamanında, dönen nesnenin gerçek tipi bilinince ortaya çıkar.

TestHR.java

```
Employee e = new Manager();
e.work();
```



Employee'nin work() metodunun çağrılacağını düşnebilirsiniz!

Aslında çağrılan Manager'in work() metodudur!!!

Binding

Bağlama (Binding)

- Programlama dillerinde **bağlama** (**binding**), değişken, metot, vb. özelliklerin dilin elemanlarıyla ilişkilendirmesidir.
  - Örneğin Java'da "*****" sembolünün matematiksel çarpma işlemine bağlanması dilin tasarım zamanında yapılmıştır.
- Dillerde temelde iki bağlama zamanı önemlidir:
  - **Statik bağlama** (**static binding**): Statik bağlamada özelliklerin, dilin elemanlarıyla ilişkilendirmesi çalışma zamanından önce yapılır ve programın çalışması sırasında da değişmez.
  - **Dinamic bağlama** (**dynamic binding**): Dinamik bağlamada ise özelliklerin, dilin elemanlarıyla ilişkilendirmesi çalışma zamanında yapılır ve programın çalışması sırasında değişebilir.

Method Binding

- Programlama dillerinde, operationların, metotlara bağlanmalarına da **method binding** (**metot bağlama**) denir.

- Nesne merkezli dillerde de metotların bağlanmasında da statik ve dinamik olmak üzere iki farklı bağlama söz konusudur

- Metot polymorphismi, dynamic binding (dinamik bağlama) denen bir teknikle başarılır.

Operation ve Method Ayırımı - I

- Önce operation (message) ve method arasındaki ayırımı açıklayalım.
- Bu ayırım nesne-merkezli dillerde söz konusudur.
  - Operation (ya da message) ile soyut olarak yani arayüz seviyesinde bir referansın üzerinde çağrılabilecek olan davranışlar kastedilir.
  - Method ise bir nesnenin üzerinde çağrılan davranışın kendisidir.
- Operation arayüz, metot ise gerçekleştirme demektir.

Operation ve Method Ayırımı - II

- Operation (ya da message) daha çok bir kalıtım hiyerarşisinde birden fazla gerçekleştirmesi olan ve hiyerarşinin en tepesinde tanımlanan metodun arayüzünü, metot ise o hiyerarşideki gerçekleştirmeleri temsil eder.

- Bir hiyerarşide bir operation ama o operationun birden fazla gerçekleştirmesi yani metodu bulunur.

- Operation daha çok tasarım ve derleme zamanı, method ise derleme ve çalışma zamanı yapısıdır.

Dinamik Bağlama - I

- Dinamik bağlamada her şey çalışma zamanı tarafından belirlenir.

- Dinamik bağlamada, derleyici sadece çağrılan metodun arayüzü yani operation seviyesinde kontroller yapabilir.
  - Metodun arayüzü, üzerinde çağrı yapılan referansın tipinde var mı?
  - Metodun arayüzü ile çağrısı arasında bir uyumsuzluk var mı?
    - Örneğin, isim, parametre sayı ve tip kontrolleri, dönüş tipi kontrolü vs.

Dinamik Bağlama - II

- Ama derleyici operationun gerçekte hangi nesne üzerindeki gerçekleştirmesinin çağrılacağını bilemez.
  - Çünkü nesne bir çalışma-zamanı yapısıdır.

- Bu bilgi, çalışma zamanında üzerinde metot çağrısı yapılan referansın gösterdiği nesnenin gerçek tipi ortaya çıkana kadar bilinemez.
- Ne zaman nesne belli olur, o zaman o nesnenin üzerindeki metodun çağrılacağı belli olur.
- Bu yüzden bu tür bağlamaya late binding (geç bağlama) denir.

Uygulama - 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.
- Canvas diye bir başka sınıf oluşturun ve üzerine Shape alan, drawShape(Shape s) ve eraseShape(Shape s) metotlarını koyun.

Uygulama - II

- ShapeFactory isimli bir başka sınıfın üzerindeki createShape() isimli metodun da random olarak bir Shape nesnesi yaratıp döndürmesini sağlayın.

- Test sınıfında da random Shape nesneleri üretip, Canvas'ın metotlarına geçin ve hangi metotların çağrıldığını gözleyin.

Statik Bağlama

- Statik bağlamada her şey çalışma zamanından önce belirlenir ve programın çalışması sırasında da değişmez.
  - Muhtemelen derleyici tarafından belirlenir.
- Statik bağlanan metotlar ise, bellekteki kodlarına derleyici tarafından derleme zamanında bağlanır.
- Bu yüzden bu tür bağlanmaya early binding (erken bağlama) da denir.
- Genel olarak prosedürel dillerdeki metotların bağlanmaları statiktir.

Java'da Metotları Statik Bağlama - I

- Java'da static, private veya final olan metotlar statik olarak bağlanırlar.
  - static metotlar, nesne üzerinde çağrılmazlar, sınıf üzerinde çağrılırlar.
    - Bu yüzden override edilemezler, polymorphic değildirler ve bağlanmaları da statiktir.
  - private metotlar zaten devralınmadığından override da edilemezler ve bağlanmaları statiktir.
  - final metotlar devralınırlar ama override edilemezler ve bağlanmaları statiktir.
- Java'da sadece override edilen operationların birden fazla metodu olacağından, dinamik bağlanmaları söz konusudur.

Test.java

- Test.java in binding package.

Java'da Metotları Statik Bağlama - II

- Java'da static, private veya final olan metotlar statik olarak bağlanırlar.
- Statik bağlanan metotlar, dinamik bağlanan metotlara göre daha hızlı çalışma eğilimindedirler.
  - Çünkü statik bağlama çalışma zamanında yapılacak işleri derleme zamanına çeker.
- Bundan dolayı, override edilmeyen metotları "final" olarak işaretleyerek çalışma zamanı performansını arttıran araçlar vardır.

BindingTest.java

- Kullandığınız IDE'nin yardımıyla, BindingTest.java'nın main metodunda yapılan "e.work()" ve "boss.youWorkToo()" metodlarının "implementation"larını bulun.
  - Eclipse'te fare ile metodun üstüne gelip CTRL (Win) ya da CMD (Mac) tuşlarına basıp "Open Implementation"u seçin.
- Hangi metodun statik – dinamik bağlandığını belirleyin.

```
public class BindingTest {
   public static void main(String[] args) {
        HR hr = new HR();
        Employee e = hr.getAnEmployee();
        e.work();

        Boss boss = new Boss();
        boss.youWorkToo();
   }
}
```

Java'da Metotları Statik Bağlama - III

- Java'da static olan metotlar statik olarak bağlanırlar.

- Eğer, ebeveynde olan statik bir metodu alt sınıfta tekrar tanımlarsanız bu overriding olmaz.
  - Buna **gölgeleme** (**shadowing**) denir.
  - Yani, alt sınıftaki metot, ebeveynden devralınan metodu gölgeler.

Değişken Bağlama

- Java'da nesne ve sınıf değişkenleri de nesnelere statik olarak bağlanırlar.
- Overriding, sadece nesne metotları için geçerlidir, değişkenler için söz konusu değildir.
- Eğer, ebeveynde olan bir değişkeni (nesne ya da statik olsun) alt sınıfta tekrar tanımlarsanız bu overriding olmaz, **gölgeleme** (**shadowing**) olur.

Sonuçlar

Neden Polymorphism?

- Polymorphism, programların parçaları arasındaki bağımlılıkları arayüz seviyesine çekerek daha kolay değiştirebilme imkanı sağlar.
- Polymorphism sayesinde arayüz ile gerçekleştirmeyi ayırabiliriz.

Program to an interface not an implementation.

- Var olan hiyerarşiye yapılacak değişiklikler, o hiyerarşiyi en tepedeki sınıf (yani arayüz) düzeyinde bilen istemci sınıfları etkilemez.
- Bu polymorphismin en temel artısıdır.

Polymorphismin Bir Sonucu - I

- Polymorphismin negatif tarafı ise, bahsedilen en temel artısının bir sonucudur.
- Polymorphismde alt sınıfların kendilerine has özelliklerini kullanamayız.
- Çünkü polymorphism, bir hiyerarşideki nesneleri, hiyerarşinin en tepesindeki nesne cinsinden ifade ettiğinden, alt sınıfların nesneleri de tepe sınıfın nesnesi olarak görülür.
- Bu kalıtımın sağladığı genelleştirme- özelleştirme ilişkisi üzerine bina edilen polymorphic davranışın bir sonucudur.

Polymorphismin Bir Sonucu - II

- Bir metodun arayüzü, o metodun kendisi üzerinde çağrıldığı referansın tipi tarafından belirlenir.
  - Bundan dolayı Employee tipindeki referansın üzerinde sadece Employee'nin arayüzündeki metotlar çağrılabilir.
- O metodun hangi gerçekleştirmesinin alınacağı ise referansın gösterdiği nesne tarafından belirlenir.

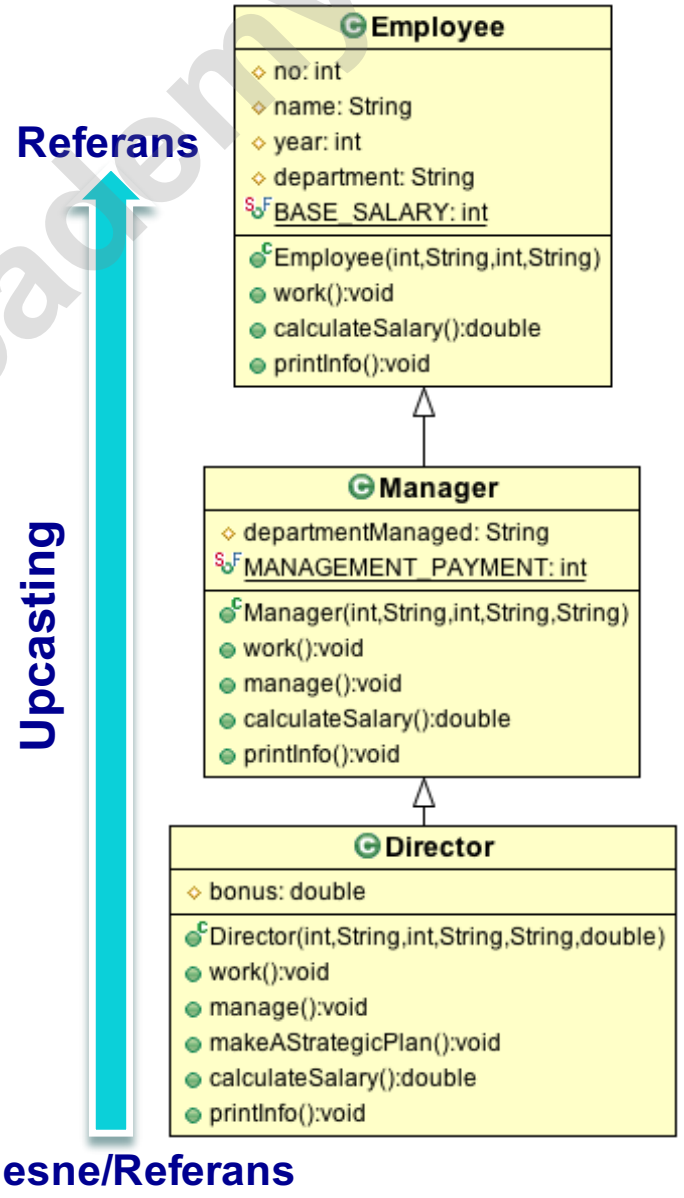
```
Employee e = new Employee();
e.work();
e = new Manager();
e.work();
e.manage();                    => Derleme hatası
e = new Director();
e.work();
e.makeAStrategicPlan() => Derleme hatası
```

Downcasting (Alçaltma)

Upcasting (Tekrar)

- Bir referansı ya da nesneyi alıp onu üst tipinden bir referansa atamaya **upcasting** (**yükseltme**) denir.
- Upcasting, yerine geçebilme özelliğinden dolayı her zaman güvenlidir.
  - Dolayısıyla cast operatörü "()" kullanmaya gerek yoktur.

```
Manager m = new Manager();
Employee e = m;
ya da doğrudan
Employee e = new Manager();
Manager m = new Director();
Emlpoyee e = new Director();
```



Downcasting - I

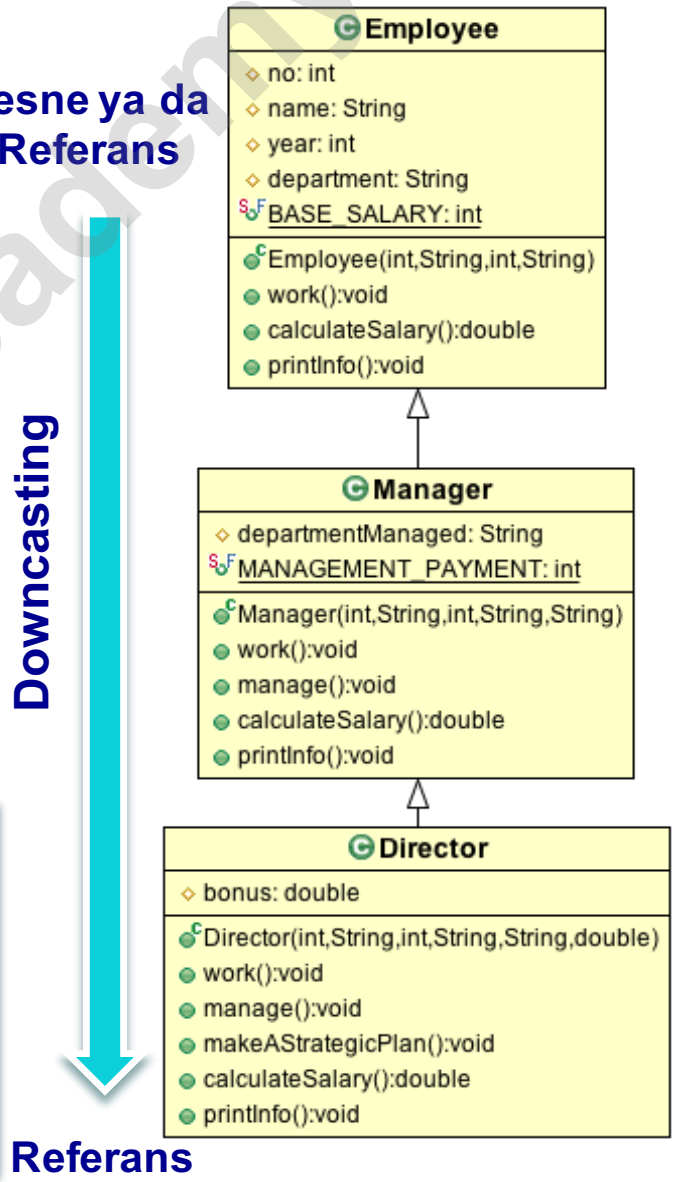
- Peki, üst bir tipten olan bir referansı ya da nesneyi, alt tiplerinden olan bir referansa atayabilir miyiz?
  - Ve bu ne işe yarar?

- Üst tipten olan bir referansı ya da nesneyi alıp onu alt tipinden bir referansa atamaya **downcasting** (**alçaltma**) denir.

Downcasting - II

- Java, üst tipten olan bir referansı ya da nesneyi alıp onu alt tipinden bir referansa atamaya izin vermez, derleme hatası verir.
  - Bu türden atamaya yani downcastinge ancak cast operatörü "()" ile izin verir.
  - Cast operatörü içinde hedef tip bulunur.

```
Employee e = new Employee();
Manager m = e;     // Derleyici hatası
Manager m = (Manager) e; // Problem!
m.manage();
Director d = (Director)new Manager();
Director d = (Director)new Employee();
```



Downcasting - III

- Cast operatörü kullanarak çevrime zorlamak, derleyici hatasını giderir ama çalışma zamanında hala gerçek nesnenin çevrilen tipe uygun olmamama riski vardır.
- Eğer cast edilen nesne ya da referansın gösterdiği nesne, hedef tipten ya da onun alt tipinden değilse, bu durumda çalışma zamanında java.lang.ClassCastException sıra dışı durumu oluşur.

```
Employee e = new Manager();
Manager m = (Manager) e;                    // Gerçek tipe geri dönüş

Employee e = new Employee();
Manager m = (Manager) e;                      // ClassCastException!
Director d = (Director) new Manager(); // ClassCastException!
Director d = (Director) new Employee();// ClassCastException!
```

DowncastingExample.java

Downcasting - IV

- Downcast işlemine çoğunlukla, üst tipten bir referans döndüren metot çağrılarından sonra ihtiyaç duyulur.
- Cast operatörü kullanarak çevrim yaparken, çevrimin uygun bir tipe yapılmaması ihtimalinden dolayı ClassCastException sıra dışı durumu fırlatılabilir.

```
HR hr = new HR();
Employee e = hr.getAnEmployee();
// Exact type of the returned object is not known!

Director d = (Director) e; // Risk of ClassCastException!
d.makeAStrategicPlan();
```

ClassCastException

- ClassCastException, java.lang paketindeki sıra dışı durum sınıflarından birisidir.
- Bir nesne, nesnesi olmadığı, kendi tipinin alt tiplerinden birine çevrilmeye çalışıldığında fırlatılır.

```
public class ClassCastException
extends RuntimeException
```

Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.

```
Object x = new Integer(0);
System.out.println((String)x);
```

instanceof Operatörü

- Cast operatörü kullanarak çevrim yaparken, çevrimin uygun bir tipe yapılmamasından doğacak ClassCastException sıra dışı durumunu önlemenin yöntemi, **instanceof** operatörüdür.
- instanceof operatörü sağa ve sola birer tane olmak üzere iki tane operand alır ve boolean döndürür:
  - Soldaki test edilecek referans, sağdaki ise hedef tiptir.

```
Employee e = new Employee();
if(e instanceof Manager){
      Manager m = (Manager) e;
}
Director d = (Director)new Manager();
Director d = (Director)new Employee();
```

InstanceofExample.java

Upcasting vs. Downcasting - I

- Upcasting, tüm nesneleri, ebeveynleri tipinden görmemizi ve onlara sanki ebeveynmiş gibi davranmamızı sağlar.

- Upcastingin negatif tarafı ise, üst tiplerden referanslarla gösterilen nesnelerin, ebeveynlerinde olmayan, kendilerine has olan özelliklerini kaybetmeleridir.

- Çünkü bir nesnenin üzerinde erişilebilecek olan özellikleri, o nesnenin referansının tipi belirler.
  - Referans üst tipten olduğu için referansın arayüzü, gerçek nesnenin arayüzünden daha dardır (extends anahtar kelimesi!)

Upcasting vs. Downcasting - II

- Dolayısıyla upcasting, nesneleri tek tipleştirir, farklılıklarını ortadan kaldırır.
  - Farklı özelliklere sahip olan nesneler, aynı referansa atandığında, arayüzleri aynileşir ve referansın arayüzüne iner.
- Downcasting ise bu şekilde kendine has olan özelliklerini, üst tipten referansa atanmasından dolayı kaybetmiş olan nesnelere, var olan özelliklerini geri kazandırır.
- Upcasting ile tek tipleşmiş olan nesneler, downcasting ile tabiri caizse kendilerine gelirler, tüm özelliklerini gösterebilirler.

TestPayrollOffice.java

- TestPayrollOffice.java'yı çalıştırmadan önce PayrollOfice.java'daki paySalary(Employee e) metodunu değiştirin.

Özet

- Bu bölümde, çok şekillilik (polymorphism) ele alındır.
  - Referanslar ve metotların polymorphic davranışları incelendi.
- Tipler arasındaki upcasting ve downcasting çevrimleri (conversion) ele alındı.
- **instanceof** operatörü ile RunTime Type Identification (RTTI) işlendi.

Ödevler

Ödevler I

- Daha önce oluşturduğunuz Shape hiyerarşisini ele alın.

- Canvas sınıfının üzerinde var olan drawShape() ve eraseShape() metotlarında çizilen ya da silinen nesnenin gerçek tipine göre yarı çapını, kenarlarını ve yüksekliğini konsola yazın.

Ödevler II

- Yandaki yapıyı oluşturun.
- Upcasting ve downcasting ile instanceof kullanacak şekilde RegistrationOffice metotlarını kurgulayın.