

# Bölüm 11

## Kalıtım ve Çok Biçimlilik (Inheritance and Polymorphism)



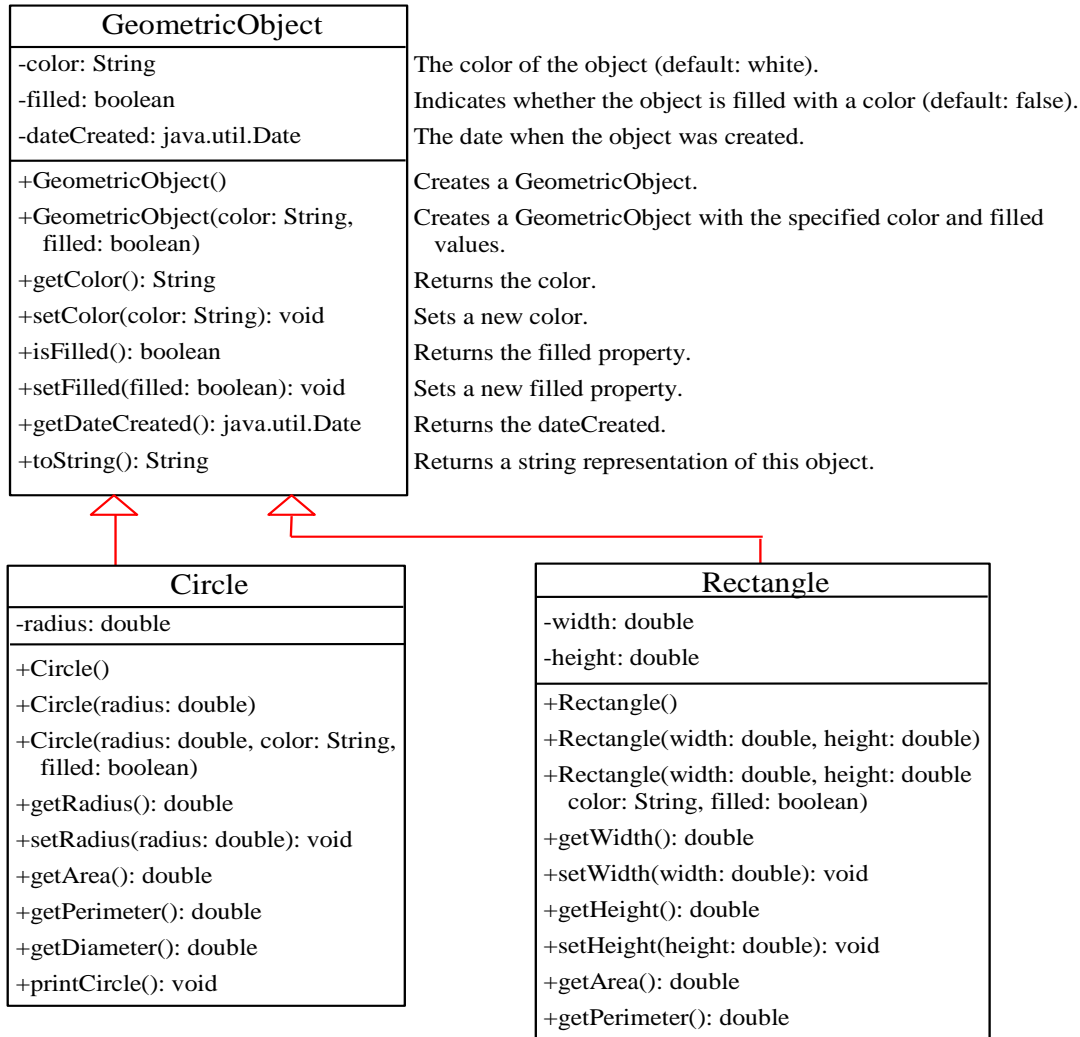
# Motivations

Daireleri, dikdörtgenleri ve üçgenleri modellemek için sınıfları tanımlayacağımızı varsayalım. Bu sınıfların birçok ortak özelliği vardır. Kod fazlalığını ve tekrarını önlemek için bu sınıfları tasarlamamanın en iyi yolu nedir?

Cevap, kalıtım (inheritance) kullanmaktır.



# Süper (Üst) Sınıflar ve Alt Sınıflar (Superclasses and Subclasses)



GeometricObject1

Circle4

Rectangle1

TestCircleRectangle

Run

# Superclass' ın Constructor' ı Miras Alınır Mı?

Hayır. Miras alınmaz.

Açık veya dolaylı olarak çağırılırlar.

Açık çağrım super anahtar kelimesini kullanarak gerçekleştirilir.

Bir yapıcı, bir sınıfın örneğini oluşturmak için kullanılır. Özelliklerin ve yöntemlerin aksine, bir üst sınıfın yapıcıları alt sınıf tarafından miras alınamazlar. Bunlar yalnızca super anahtar sözcüğünü kullanarak, alt sınıfların yapıcılarından çağırılabilirler. Super anahtar sözcüğü açıkça kullanılmazsa, superclass ögesinin argümansız yapıcısı otomatik olarak çağrılır.

# Superclass'ın Yapıcısı Her Zaman Çağrılır

Bir yapıcı, aşırı yüklenmiş bir yapıcı veya üst sınıfının yapıcısını çağırabilir. Bunlardan hiçbiri açıkça çağrılmazsa, derleyici, yapıcıdaki ilk ifade olarak `super ()` komutunu koyar. Örneğin,

```
public A() {  
}
```

is equivalent to

```
public A() {  
    super();  
}
```

```
public A(double d) {  
    // some statements  
}
```

is equivalent to

```
public A(double d) {  
    super();  
    // some statements  
}
```

# Super Anahtar Kelimesini Kullanma

Super anahtar sözcüğü, super öğesinin görüldüğü sınıfın üst sınıfına işaret eder. Bu anahtar kelime iki şekilde kullanılabilir:

- Bir üst sınıf yapıcısını çağırma
- Bir üst sınıf metodunu çağırma



# Dikkat

Süper sınıf kurucusunu çağırmak için super anahtar sözcüğünü kullanmanız gerekir. Bir üst sınıf yapıcısının adını bir alt sınıfta çağırmak, bir sözdizimi hatasına neden olur. Java, super anahtar sözcüğünü kullanan ifadenin ilk kurucuda görünmesini gerektirir.



# Constructor Chaining (Yapıcı Zincirleme)

Bir sınıf örneği oluşturmak, tüm üst sınıfların yapıcılarını miras zinciri boyunca çağırır. Buna yapıcı zincirleme denir.

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```



# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

1. Main metottan  
başla

# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

2. Faculty  
constructor' ının  
çağırılması

# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

3. Employee' inin  
argümentsiz constructor'  
ının çağırılması

# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

4. Employee 'nin(String) constructor' ının çağırılması

# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

5. Person() constructor' ının  
çağırılması

# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```

6. println' nin yürütülmesi

# Trace Execution

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

7. println' nin yürütülmesi

# Trace Execution

```
public class Faculty extends Employee {  
    public static void main(String[] args) {  
        new Faculty();  
    }  
  
    public Faculty() {  
        System.out.println("(4) Faculty's no-arg constructor is invoked");  
    }  
}  
  
class Employee extends Person {  
    public Employee() {  
        this("(2) Invoke Employee's overloaded constructor");  
        System.out.println("(3) Employee's no-arg constructor is invoked");  
    }  
  
    public Employee(String s) {  
        System.out.println(s);  
    }  
}  
  
class Person {  
    public Person() {  
        System.out.println("(1) Person's no-arg constructor is invoked");  
    }  
}
```



8. println' nin yürütülmesi



# Trace Execution

```
public class Faculty extends Employee {
    public static void main(String[] args) {
        new Faculty();
    }

    public Faculty() {
        System.out.println("(4) Faculty's no-arg constructor is invoked");
    }
}

class Employee extends Person {
    public Employee() {
        this("(2) Invoke Employee's overloaded constructor");
        System.out.println("(3) Employee's no-arg constructor is invoked");
    }

    public Employee(String s) {
        System.out.println(s);
    }
}

class Person {
    public Person() {
        System.out.println("(1) Person's no-arg constructor is invoked");
    }
}
```

9. println' nin yürütülmesi

# Example on the Impact of a Superclass without no-arg Constructor

Programdaki hataları bulunuz:

```
public class Apple extends Fruit {  
}  
  
class Fruit {  
    public Fruit(String name) {  
        System.out.println("Fruit's constructor is invoked");  
    }  
}
```



# Alt Sınıfın Bildirimi (Declaring a Subclass)

Bir alt sınıf, üst sınıftaki özellikleri ve yöntemleri extend eder.

Ayrıca:

- ❑ Yeni özellikler (properties) ekleyebilir.
- ❑ Yeni metotlar (methods) ekleyebilir.
- ❑ Üst sınıfın yöntemlerini geçersiz kılabilir. (override edebilir)



# Superclass Metotlarını Çağırma

Circle sınıfındaki printCircle () metodunu aşağıdaki gibi yeniden yazabilirsiniz:

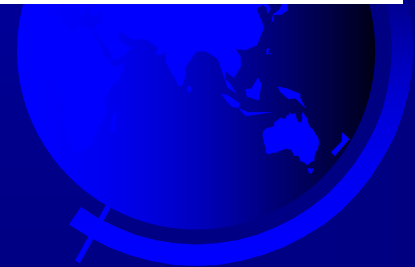
```
public void printCircle() {  
    System.out.println("The circle is created " +  
        super.getDateCreated() + " and the radius is " + radius);  
}
```



# Superclass' da Metotları Override Etme

Bir alt sınıf, bir üst sınıftan metotlarını miras alır. Bazen alt sınıfın, üst sınıfta tanımlanan bir metodun uygulamasını değiştirmesi gerekir. Buna yöntem geçersiz kılma (override) denir.

```
public class Circle extends GeometricObject {  
    // Other methods are omitted  
  
    /** Override the toString method defined in GeometricObject */  
    public String toString() {  
        return super.toString() + "\nradius is " + radius;  
    }  
}
```



# NOT

Bir örnek yöntemi yalnızca erişilebilir olması durumunda geçersiz kılınabilir. Dolayısıyla, private bir yöntem geçersiz kılınamaz, çünkü kendi sınıfı dışında erişilebilir değildir. Bir alt sınıfta tanımlanan bir yöntem, üst sınıfta private ise, iki yöntem birbiriyle tamamen ilişkili değildir.



# NOT

Örnek bir yöntem gibi, statik bir yöntem de miras alınabilir. Ancak, statik bir yöntem geçersiz kılınamaz. Üst sınıfta tanımlanan statik bir yöntem bir alt sınıfta yeniden tanımlanırsa, üst sınıfta tanımlanan yöntem gizlenir.



# Overriding vs. Overloading

*Aşırı yükleme aynı ada sahip birden çok yöntemi farklı imzalar ile tanımlamak anlamına gelir. Geçersiz kılma, alt sınıftaki bir yöntem için yeni bir uygulama sağlamak anlamına gelir.*

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overrides the method in B  
    public void p(double i) {  
        System.out.println(i);  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        A a = new A();  
        a.p(10);  
        a.p(10.0);  
    }  
}  
  
class B {  
    public void p(double i) {  
        System.out.println(i * 2);  
    }  
}  
  
class A extends B {  
    // This method overloads the method in B  
    public void p(int i) {  
        System.out.println(i);  
    }  
}
```



Method Overloading	Method Overriding
Provides functionality to reuse a method name with different parameters.	Used to override a behavior which the class has inherited from the parent class.
Usually in a single class but may also be used in a child class.	<b>Always in two classes</b> that have a child-parent or IS-A relationship.
<b>Must have</b> different parameters.	<b>Must have</b> the same parameters and same name.
May have different return types.	Must have the same return type or covariant return type (child class).
May have different access modifiers(private, protected, public).	<b>Must NOT</b> have a lower modifier but may have a higher modifier.
May throw different exceptions.	<b>Must NOT</b> throw a new or broader checked exception.

## OVERRIDING

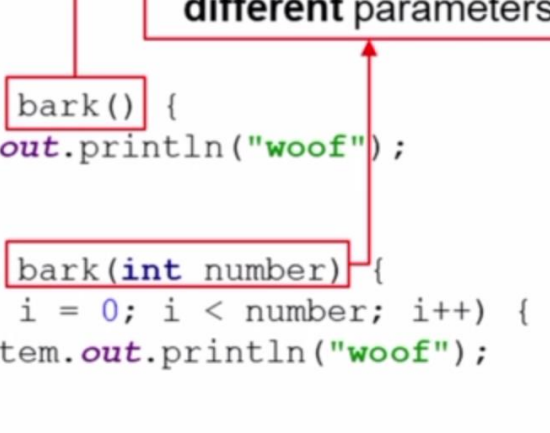
```
class Dog {  
    public void bark() {  
        System.out.println("woof");  
    }  
}  
  
class GermanShepherd extends Dog {  
    @Override  
    public void bark() {  
        System.out.println("woof woof  
woof");  
    }  
}
```



same name  
same parameters

## OVERLOADING

```
class Dog {  
    public void bark() {  
        System.out.println("woof");  
    }  
  
    public void bark(int number) {  
        for(int i = 0; i < number; i++) {  
            System.out.println("woof");  
        }  
    }  
}
```



same name  
different parameters

# Nesne Sınıfı ve Metodları

Java'daki her sınıf `java.lang.Object` sınıfından türetilmiştir. Bir sınıf tanımlandığında miras belirtilmezse, sınıfın üst sınıfı `Object` sınıfıdır.

```
public class Circle {  
    ...  
}
```

Equivalent

```
public class Circle extends Object {  
    ...  
}
```

# Object Sınıfında toString() Metodu

ToString () yöntemi, nesnenin bir string temsilini döndürür. Varsayılan uygulama, nesnenin bir örneği olduğu bir sınıf adından, (@) işaretinden ve bu nesneyi temsil eden bir sayıdan oluşan bir string döndürür.

```
Loan loan = new Loan();  
System.out.println(loan.toString());
```

Kod, Loan@15037e5 gibi bir şey görüntüler. Bu mesaj çok yararlı veya bilgilendirici değil. Genellikle toString yöntemini geçersiz kılmalısınız, böylece nesnenin mantıklı bir string gösterimini döndürebilirsiniz.



# Çok Biçimlilik (Polymorphism)

*Polimorfizm, bir üst tip değişkeninin bir alt tip nesneyi gösterebileceği anlamına gelir.*

Kalıtım ilişkisi, bir alt sınıfın, üst sınıfından özellikleri ek yeni özellikler ile devralmasını sağlar. Bir alt sınıf, üst sınıfının özelleşmiş bir halidir; Bir alt sınıfın her örneği aynı zamanda onun üst sınıfının bir örneğidir, ancak bunun tersi de değildir. Örneğin, her daire geometrik bir nesnedir, ancak her geometrik nesne bir daire değildir. Bu nedenle, bir alt sınıfın örneğini her zaman onun üst sınıf türünün bir parametresine geçirebilirsiniz.



# Polimorfizm, Dinamik Bağlama ve Genel Programlama (Polymorphism, Dynamic Binding and Generic Programming)

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

m yöntemi, Object türünün bir parametresini alır. Herhangi bir nesneyle onu çağırabilirsiniz.

Alt tipin bir nesnesi, üst tip değerinin gerekli olduğu her yerde kullanılabilir. Bu özellik polimorfizm olarak bilinir.

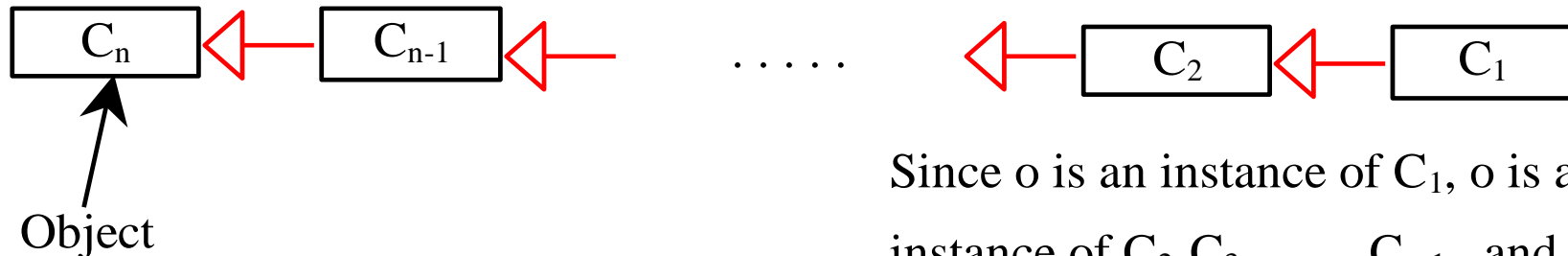
M yöntemi (Object x) yürütüldüğünde, x'in toString yöntemi çağrılır. x GraduateStudent, Student, Person veya Object sınıfının bir örneği olabilir. GraduateStudent, Student, Person ve Object sınıfları, toString yöntemi için kendi uygulamalarına sahiptir. Hangi uygulamanın kullanıldığı, çalışma zamanında Java Sanal Makinesi tarafından dinamik olarak belirlenir. Bu yetenek dinamik bağlama olarak bilinir.

PolymorphismDemo

Run

# Dinamik Bağlama (Dynamic Binding)

Dinamik bağlama aşağıdaki gibi çalışır: Bir  $c$  nesnesinin,  $C_1, C_2, \dots, C_{n-1}$  ve  $C_n$  sınıflarının bir örneği olduğunu, burada  $C_1$ 'in  $C_2$ 'nin bir alt sınıfı olduğu,  $C_2$ 'nin  $C_3$ 'ün bir alt sınıfı olduğu ...  $C_{n-1}, C_n$ 'nin bir alt sınıfıdır. Yani,  $C_n$  en genel sınıftır ve  $C_1$  en spesifik sınıftır. Java'da,  $C_n$ , Object sınıfıdır. O,  $p$  yöntemini çağırırsa, JVM, bulunana kadar bu sırada,  $C_1, C_2, \dots, C_{n-1}$  ve  $C_n$ 'deki  $p$  yönteminin uygulamasını arar. Bir uygulama bulunduğunda, arama durur ve ilk bulunan uygulama çağrılır.



Since  $o$  is an instance of  $C_1$ ,  $o$  is also an instance of  $C_2, C_3, \dots, C_{n-1}$ , and  $C_n$

# Metot Eşleştirme ve Bağlama

Bir yöntem imzasını eşleştirmek ve bir yöntem uygulamasını bağlamak iki konudur. Derleyici, parametre türüne, parametre sayısına ve derleme sırasındaki parametrelerin sırasına göre eşleşen bir yöntem bulur. Birkaç alt sınıfta bir yöntem uygulanabilir. Java Sanal Makinesi, yöntemin uygulanmasını çalışma zamanında dinamik olarak bağlar.





# Genel Programlama (Generic Programming)

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

- Polimorfizm, yöntemlerin çok çeşitli nesne argümanları için genel olarak kullanılmasına izin verir. Bu genel programlama olarak bilinir. Bir yöntemin parametre türü bir üst sınıfsa (örneğin, Nesne), bir nesnenin parametresinin alt sınıflarından herhangi birinin bu yöntemine (örneğin, Öğrenci veya Dize) geçebilirsiniz. Yöntemde bir nesne (örneğin bir Öğrenci nesnesi veya bir String nesnesi) kullanıldığında, çağrılan nesnenin yönteminin belirli bir uygulaması (örneğin, toString) dinamik olarak belirlenir.



# Casting Objects

Bir ilkel türdeki değişkenleri diğerine dönüştürmek için cast operatörünü kullanırız. Casting, kalıtım hiyerarşisinde bir sınıf türündeki bir nesneyi diğerine dönüştürmek için de kullanılabilir.

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting  
m(o);
```

Object o = new Student() ifadesi, implicit casting olarak bilinir, Student sınıfının örneği otomatik olarak Object sınıfının da bir örneği olduğu için bu kullanım doğrudur.



# Casting Neden Gereklidir?

Aşağıdaki ifadeyi kullanarak o nesne referansını o Öğrenci türünün bir değişkenine atamak istediğinizi varsayalım:

```
Student b = o;
```

Bir derleme hatası meydana gelir. `Object o = new Student ()` ifadesi çalışıyor da neden `Student b = o` ifadesi çalışmıyor? Bunun nedeni, bir Öğrenci nesnesinin daima bir `Object` örneği olmasıdır, ancak bir `Object` mutlaka bir Öğrenci örneği değildir. Bunun gerçekten bir Öğrenci nesnesi olduğunu görebilseniz de, derleyici bunu bilecek kadar akıllı değildir. Derleyiciye Öğrenci nesnesi olduğunu söylemek için açık bir casting kullanın. Sözdizimi, ilkel veri türleri arasında casting yapmak için kullanılabenzer. Hedef nesne türünü parantez içine alın ve kopyalanacak nesnenin önüne aşağıdaki şekilde yerleştirin:

```
Student b = (Student)o; // Explicit casting
```

# Superclass' dan Subclass' a Casting

- Bir nesneyi bir üst sınıftan bir alt sınıfa casting ederken açık (explicit) casting kullanılmalıdır. Bu tip casting her zaman başarılı olmayabilir.

```
Apple x = (Apple)fruit;
```

```
Orange x = (Orange)fruit;
```



# instanceof Operatoru

instanceof operatörü bir nesnenin bir sınıfın örneği olup olmadığını test eder.

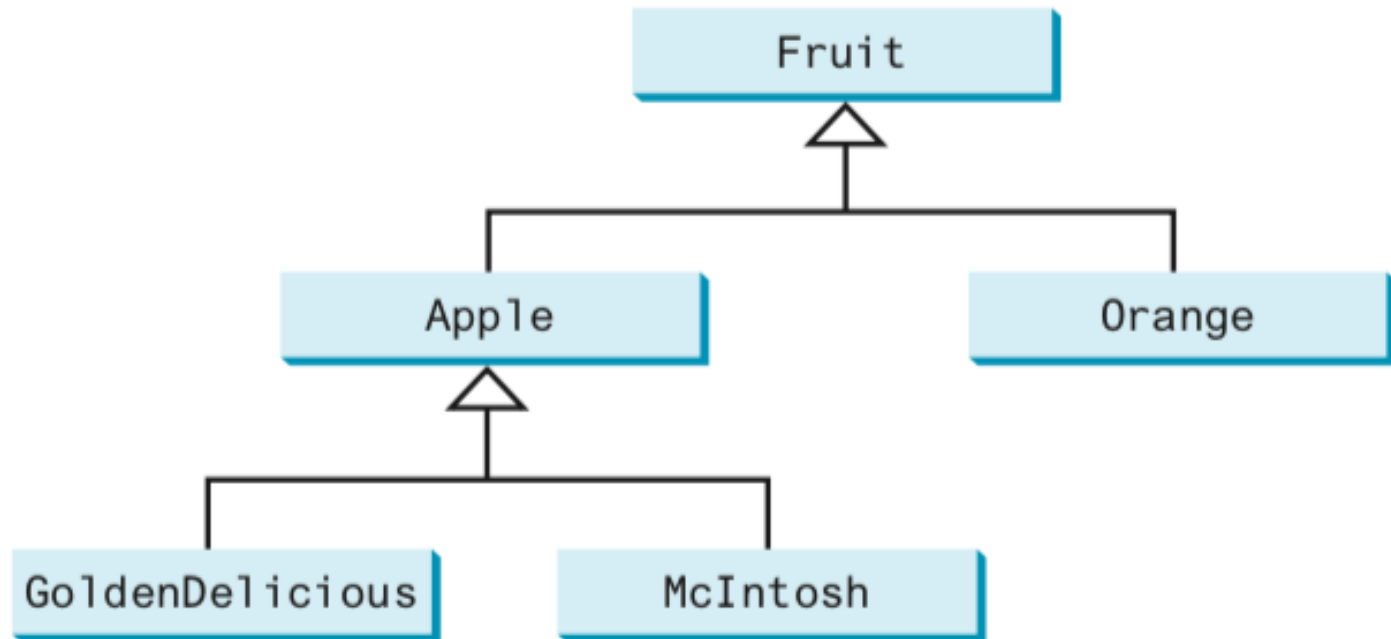
```
Object myObject = new Circle();  
... // Some lines of code  
/** Perform casting if myObject is an instance of  
    Circle */  
if (myObject instanceof Circle) {  
    System.out.println("The circle diameter is " +  
        ((Circle)myObject).getDiameter());  
    ...  
}
```



# İpucu (TIP)

Casting işleminin anlaşılmasına yardımcı olmak için, Fruit apple ve orange sınıflarının analojisini bakmak gerekir. Fruit sınıfını, Apple ve orange için bir üst sınıf olarak düşünebilirsiniz. Elma bir meyvedir, bu yüzden her zaman güvenle bir Elma örneğini Meyve değişkenine atayabilirsiniz. Bununla birlikte, bir meyve mutlaka bir elma değildir, bu nedenle bir Meyve örneğini bir elma değişkenine atamak için açık bir döküm kullanmanız gerekir.





Assume the following code is given:

```
Fruit fruit = new GoldenDelicious();  
Orange orange = new Orange();
```

# Örnek: Polimorfizm ve Casting Gösterme

Bu örnek iki geometrik nesne oluşturur: bir daire ve bir dikdörtgen, nesneleri görüntülemek için `displayGeometricObject` yöntemini çağırır. `DisplayGeometricObject`, nesne bir daire ise alanı ve çapı gösterir ve nesne bir dikdörtgen ise alanı gösterir.

TestPolymorphismCasting

Run





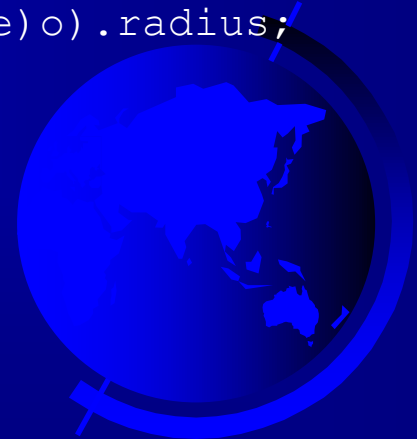
# equals Metodu

Equals () yöntemi iki nesnenin içeriğini karşılaştırır. Object sınıfında equals() yönteminin varsayılan uygulaması aşağıdaki gibidir:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {  
    if (o instanceof Circle) {  
        return radius == ((Circle)o).radius;  
    }  
    else  
        return false;  
}
```



# NOT

== Karşılaştırma operatörü, iki ilkel veri tipi değerini karşılaştırmak veya iki nesnenin aynı referanslara sahip olup olmadığını belirlemek için kullanılır. Eşit yöntemi, yöntemin nesnelerin tanımlayıcı sınıfında değiştirilmesi şartıyla, iki nesnenin aynı içeriğe sahip olup olmadığını sınaama amaçlıdır. == operatörü, equals yönteminden daha güçlüdür, çünkü == operatörü, iki referans değişkeninin aynı nesneye başvurup başvurmadığını kontrol eder.

# ArrayList ce Vector Sınıfı

Nesneleri saklamak için bir dizi oluşturabilirsiniz. Ancak dizi oluşturulduktan sonra dizinin boyutu sabittir. Java, sınırsız sayıda nesneyi saklamak için kullanılabilecek ArrayList sınıfını sağlar.

## java.util.ArrayList

+ArrayList()

Creates an empty list.

+add(o: Object) : void

Appends a new element o at the end of this list.

+add(index: int, o: Object) : void

Adds a new element o at the specified index in this list.

+clear(): void

Removes all the elements from this list.

+contains(o: Object): boolean

Returns true if this list contains the element o.

+get(index: int) : Object

Returns the element from this list at the specified index.

+indexOf(o: Object) : int

Returns the index of the first matching element in this list.

+isEmpty(): boolean

Returns true if this list contains no elements.

+lastIndexOf(o: Object) : int

Returns the index of the last matching element in this list.

+remove(o: Object): boolean

Removes the element o from this list.

+size(): int

Returns the number of elements in this list.

+remove(index: int) : Object

Removes the element at the specified index.

+set(index: int, o: Object) : Object

Sets the element at the specified index.

- Aşağıdaki deyim bir ArrayList oluşturur ve **cities** değişkenine referansını atar. Bu ArrayList nesnesi dizeleri saklamak için kullanılabilir.
- *ArrayList<String> cities = new ArrayList<String>();*
- Aşağıdaki ifade bir ArrayList oluşturur ve başvurusunu **dates** değişkenine atar. Bu ArrayList nesnesi tarihleri depolamak için kullanılabilir.
- *ArrayList<java.util.Date> dates = new ArrayList<java.util.Date>();*



**TABLE 11.1** Differences and Similarities between Arrays and **ArrayList**

Operation	Array	ArrayList
Creating an array/ArrayList	<code>String[] a = new String[10]</code>	<code>ArrayList&lt;String&gt; list = new ArrayList&lt;&gt;();</code>
Accessing an element	<code>a[index]</code>	<code>list.get(index);</code>
Updating an element	<code>a[index] = "London";</code>	<code>list.set(index, "London");</code>
Returning size	<code>a.length</code>	<code>list.size();</code>
Adding a new element		<code>list.add("London");</code>
Inserting a new element		<code>list.add(index, "London");</code>
Removing an element		<code>list.remove(index);</code>
Removing an element		<code>list.remove(Object);</code>
Removing all elements		<code>list.clear();</code>

Bir diziden dizi listesi oluşturma örneği:

```
String[] array = {"red", "green", "blue"};  
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```



## LISTING 11.8 TestArrayList.java

```
1  import java.util.ArrayList;
2
3  public class TestArrayList {
4      public static void main(String[] args) {
5          // Create a list to store cities
6          ArrayList<String> cityList = new ArrayList<>();
7
8          // Add some cities in the list
9          cityList.add("London");
10         // cityList now contains [London]
11         cityList.add("Denver");
12         // cityList now contains [London, Denver]
13         cityList.add("Paris");
14         // cityList now contains [London, Denver, Paris]
15         cityList.add("Miami");
16         // cityList now contains [London, Denver, Paris, Miami]
17         cityList.add("Seoul");
18         // Contains [London, Denver, Paris, Miami, Seoul]
19         cityList.add("Tokyo");
20         // Contains [London, Denver, Paris, Miami, Seoul, Tokyo]
```

```
22 System.out.println("List size? " + cityList.size());
23 System.out.println("Is Miami in the list? " +
24     cityList.contains("Miami"));
25 System.out.println("The location of Denver in the list? "
26     + cityList.indexOf("Denver"));
27 System.out.println("Is the list empty? " +
28     cityList.isEmpty()); // Print false
29
30 // Insert a new city at index 2
31 cityList.add(2, "Xian");
32 // Contains [London, Denver, Xian, Paris, Miami, Seoul, Tokyo]
33
34 // Remove a city from the list
35 cityList.remove("Miami");
36 // Contains [London, Denver, Xian, Paris, Seoul, Tokyo]
37
38 // Remove a city at index 1
39 cityList.remove(1);
40 // Contains [London, Xian, Paris, Seoul, Tokyo]
41
42 // Display the contents in the list
43 System.out.println(cityList.toString());
```

```
45      // Display the contents in the list in reverse order
46      for (int i = cityList.size() - 1; i >= 0; i--)
47          System.out.print(cityList.get(i) + " ");
48      System.out.println();
49
50      // Create a list to store two circles
51      ArrayList<Circle> list = new ArrayList<>();
52
53      // Add two circles
54      list.add(new Circle(2));
55      list.add(new Circle(3));
56
57      // Display the area of the first circle in the list
58      System.out.println("The area of the circle? " +
59          list.get(0).getArea());
60  }
61 }
```

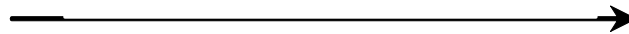




# protected Düzenleyici (Modifier)

- protected düzenleyici, bir sınıftaki veri ve yöntemlere uygulanabilir. public bir sınıftaki protected bir veriye veya protected bir metoda, aynı paketdeki herhangi bir sınıftan ya da alt sınıflarından (alt sınıflar farklı pakette olsalar dahi) erişilebilir.
- private, default, protected, public

Visibility increases



private, none (if no modifier is used), protected, public

# Erişilebilirlik Özeti

## (Accessibility Summary)

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

# Görünürlük Değiştiriciler (Visibility Modifiers)

package p1;

```
public class C1 {  
    public int x;  
    protected int y;  
    int z;  
    private int u;  
  
    protected void m() {  
    }  
}
```

```
public class C2 {  
    C1 o = new C1();  
    can access o.x;  
    can access o.y;  
    can access o.z;  
    cannot access o.u;  
  
    can invoke o.m();  
}
```



```
public class C3  
    extends C1 {  
    can access x;  
    can access y;  
    can access z;  
    cannot access u;  
  
    can invoke m();  
}
```

package p2;

```
public class C4  
    extends C1 {  
    can access x;  
    can access y;  
    cannot access z;  
    cannot access u;  
  
    can invoke m();  
}
```

```
public class C5 {  
    C1 o = new C1();  
    can access o.x;  
    cannot access o.y;  
    cannot access o.z;  
    cannot access o.u;  
  
    cannot invoke o.m();  
}
```

- Sınıf üyelerine, herhangi bir paketteki alt sınıflar veya aynı paketteki sınıflar tarafından erişilebilmesi için protected erişim düzenleyiciyi kullanın.
- Sınıf üyelerine herhangi bir sınıfın erişebilmesini sağlamak için public erişim düzenleyiciyi kullanın.



- Your class can be used in two ways: (1) for creating instances of the class and (2) for defining subclasses by extending the class. Make the members **private** if they are not intended for use from outside the class. Make the members **public** if they are intended for the users of the class. Make the fields or methods **protected** if they are intended for the extenders of the class but not for the users of the class.
- The **private** and **protected** modifiers can be used only for members of the class. The **public** modifier and the default modifier (i.e., no modifier) can be used on members of the class as well as on the class. A class with no modifier (i.e., not a public class) is not accessible by classes from other packages.



# A Subclass Cannot Weaken the Accessibility

Bir alt sınıf, üst sınıftaki protected bir yöntemi geçersiz kılabilir ve genel olarak görünürlüğünü değiştirebilir. Bununla birlikte, bir alt sınıf, üst sınıfta tanımlanan bir yöntemin erişilebilirliğini zayıflatamaz. Örneğin, bir yöntem üst sınıfta public olarak tanımlanırsa, alt sınıfta public olarak tanımlanmalıdır.



# NOT

The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method. A final local variable is a constant inside a method.



# The final Modifier

- The final class cannot be extended:

```
final class Math {  
    ...  
}
```

- The final variable is a constant:

```
final static double PI = 3.14159;
```

- The final method cannot be overridden by its subclasses.





# Özet

- Mevcut bir sınıftan yeni bir sınıf tanımlayabilirsiniz. Bu sınıf mirası olarak bilinir. Yeni sınıfa alt sınıf, child sınıf veya genişletilmiş sınıf denir. Mevcut sınıfa üst sınıf, parent sınıf veya temel sınıf denir.
- Bir yapıcı, bir sınıfın örneğini oluşturmak için kullanılır. Özelliklerin ve yöntemlerin aksine, bir üst sınıfın yapıcıları alt sınıfta miras alınmaz. Bunlar sadece super anahtar sözcüğünü kullanarak, alt sınıfların yapıcılarından çağrılabilir.

- ❑ Bir yapıcı, üst sınıfının yapıcısını aşırı yüklenmiş bir yapıcıyı çağırabilir. Çağrı, yapıcıdaki ilk ifade olmalıdır. Bunlardan hiçbiri açıkça çağrılmazsa, derleyici `super ()` öğesini superclass'ın argümanını yapıcıyı çağıran, yapıcıdaki ilk ifade olarak koyar.
- ❑ Bir yöntemi geçersiz kılmak için, yöntemin, alt sınıfında, üst sınıfındakiyle aynı imza ve aynı veya uyumlu dönüş türünü kullanarak tanımlanması gerekir.



- Bir örnek yöntemi yalnızca erişilebilir olması durumunda geçersiz kılınabilir. Dolayısıyla, **private** bir yöntem geçersiz kılınamaz, çünkü kendi sınıfı dışında erişilebilir değildir. Bir alt sınıfta tanımlanan bir yöntem, üst sınıfta **private** ise, iki yöntem birbiriyle tamamen ilişkili değildir.
- Örnek bir yöntem gibi, statik bir yöntem de alınabilir. Ancak, statik bir yöntem geçersiz kılınamaz. Üst sınıfta tanımlanan statik bir yöntem bir alt sınıfta yeniden tanımlanırsa, üst sınıfta tanımlanan yöntem gizlenir.



- Java'daki her sınıf `java.lang.Object` sınıfından türetilmiştir. Bir sınıf tanımlandığında herhangi bir üst sınıf belirtilmezse, üst sınıfı `Object`s sınıfıdır.
- Bir yöntemin parametre türü bir üst sınıfsa (örneğin, Nesne), bir nesnenin parametresinin alt sınıflarının herhangi birinin bu yöntemine (örneğin, Daire veya Dize) geçebilirsiniz. Bu polimorfizm olarak bilinir.
- Bir alt sınıfın bir örneğini bir üst sınıfın değişkenine atmak her zaman mümkündür, çünkü bir alt sınıfın bir örneği her zaman bir üst sınıfın bir örneğidir. Bir üst sınıfın bir örneğini alt sınıfının bir değişkenine dönüştürürken, derleyiciye niyetinizi (`Alt SınıfAdı`) döküm notasyonu ile doğrulamak için açık döküm kullanılmalıdır.

- ❑ Bir sınıf bir tip tanımlar. Bir alt sınıf tarafından tanımlanan bir tür, alt tür olarak adlandırılır ve onun üst sınıf tarafından tanımlanan bir tür, bir alt tür olarak adlandırılır
- ❑ Bir başvuru değişkeninden bir örnek yöntemi çağırırken, değişkenin gerçek türü, çalışma zamanında hangi yöntemin uygulanacağına karar verir.
- ❑ Bu dinamik bağlama olarak bilinir.
- ❑ Bir nesnenin bir sınıf örneği olup olmadığını sınamak için `AClass` nesnesini kullanabilirsiniz. Verilerin ve yöntemlerin, sınıf dışı kişilerin farklı bir paketten erişmesini önlemek için `protected` değiştiriciyi kullanabilirsiniz.
- ❑ Bir sınıfın final olduğunu ve genişletilemediğini ve bir yöntemin final olduğunu ve geçersiz kılınmadığını belirtmek için final değiştiriciyi kullanabilirsiniz.

