

# CENG 218 Programlama Dilleri

## Bölüm 5: Nesne Yönelimli Programlama

Öğr.Gör. Şevket Umut Çakır

Pamukkale Üniversitesi

Hafta 7

# Hedefler

- Yazılımın yeniden kullanımı ve bağımsızlık kavramlarını anlamak
- Smalltalk diline aşina olmak
- Java diline aşina olmak
- C++ diline aşina olmak
- Nesne yönelimli dillerdeki tasarım sorunlarını anlamak
- Nesne yönelimli dillerdeki uygulama sorunlarını anlamak



# Giriş

- 1960'lı yıllarda Simula ile nesne odaklı programlama dilleri başladı
  - ▶ Hedefler, bir nesnenin nosyonunu, önceden tanımlanmış şekillerde olaylara tepki verme yeteneğini kontrol eden özelliklerle dahil etmekti.
  - ▶ Soyut veri tipi mekanizmalarının geliştirilmesindeki faktör
  - ▶ Nesne paradigmasının gelişmesine çok önemli



# Giriş

- 1980'lerin ortalarında, **nesne yönelimli programlamaya(object oriented programming)** olan ilgi patladı
  - ▶ Bugün hemen hemen her dilin bir tür yapılandırılmış benzer yapıları vardır



# Yazılımın Yeniden Kullanımı ve Bağımsızlık

- Nesneye yönelik programlama dilleri, yazılım tasarımında üç önemli ihtiyacı karşılar:
  - ▶ Yazılım bileşenlerinin olabildiğince yeniden kullanılması gerekir
  - ▶ Mevcut kodda minimum değişiklikle program davranışını değiştirmek gerekir
  - ▶ Farklı bileşenlerin bağımsızlığını sürdürme ihtiyacı
- Soyut veri türü mekanizmaları, arayüzleri uygulamalardan ayırarak yazılım bileşenlerinin bağımsızlığını artırabilir



# Yazılımın Yeniden Kullanımı ve Bağımsızlık

- Bir yazılım bileşeninin yeniden kullanım için değiştirilebileceği dört temel yol:
  - ▶ Verilerin veya işlemlerin genişletilmesi(extension)
  - ▶ Bir veya daha fazla işlemin yeniden tanımlanması(redefinition)
  - ▶ Soyutlama(Abstraction)
  - ▶ Çok biçimlilik(Polymorphism)
- Veri veya işlemlerin genişletilmesi:
  - ▶ Örnek: çift uçlu bir kuyruk veya kuyruk(deque) oluşturmak için öğelerin arkadan kaldırılmasına ve öne eklenmesine izin vermek için bir kuyruğa yeni metotlar eklemek



# Yazılımın Yeniden Kullanımı ve Bağımsızlık

- Bir veya daha fazla işlemin yeniden tanımlanması:
  - ▶ Örnek: Bir dikdörtgenden bir kare elde edilirse, alan veya çevre fonksiyonları, ihtiyaç duyulan azaltılmış veriyi hesaba katmak için yeniden tanımlanabilir.
- Soyutlama veya benzer işlemlerin iki farklı bileşenden yeni bir bileşene toplanması:
  - ▶ Örnek: bir daire ve dikdörtgeni, konum ve hareket gibi her ikisinin de ortak özelliklerini içermek için şekil adı verilen soyut bir nesnede birleştirebilir



# Yazılımın Yeniden Kullanımı ve Bağımsızlık

- Çok biçimlilik veya işlemlerin uygulanabileceği veri türünün uzantısı:
  - ▶ Örnekler: aşırı yükleme ve parametrelili türler

**Uygulama çatısı(Application framework):** geliştiricinin kullanımı için ilgili yazılım kaynakları koleksiyonu (genellikle nesneye yönelik biçimde) Örnekler: C++ 'da **Microsoft Foundation Classes** ve Java'da **Swing** pencere oluşturma araç takımı





# Yazılımın Yeniden Kullanımı ve Bağımsızlık

- Nesne yönelimli dillerin başka bir amacı vardır:
  - ▶ Yazılım bileşenlerinin dahili ayrıntılarına erişimi kısıtlama
- Dahili ayrıntılara erişimi kısıtlama mekanizmalarının birkaç adı vardır:
  - ▶ Kapsülleme(Encapsulation) mekanizmaları
  - ▶ Bilgi saklama(Information hiding) mekanizmaları



# Smalltalk

- Smalltalk, 1970'lerin başında Xerox Corp.'un Palo Alto Araştırma Merkezi'ndeki Dynabook Projesinden doğmuştur.
  - ▶ Dynabook, günümüzün dizüstü ve tablet bilgisayarlarının bir prototipi olarak tasarlandı
- Smalltalk, Simula ve Lisp'ten etkilendi
- ANSI standardı 1998'de elde edildi
- Smalltalk, nesne yönelimli paradigmaya en tutarlı yaklaşıma sahiptir
  - ▶ Her şey bir nesnedir, sabitler ve sınıfların kendileri dahil



# Smalltalk

- Tamamen/**saf(purely)** nesne odaklı olduğu söylenebilir
- Çöp toplama ve dinamik tip kontrolünü içerir
- PC'ler için yaygın hale gelmeden çok önce, menüler ve bir fare içeren bir pencere sistemi içerir
- Etkileşimli ve dinamik odaklı bir dildir
  - ▶ Sınıflar ve nesneler, bir dizi tarayıcı penceresi kullanılarak sistemle etkileşim yoluyla oluşturulur.
  - ▶ Önceden var olan sınıfların geniş bir hiyerarşisini içerir



# Smalltalk'ın Temel Öğeleri: Sınıflar, Nesneler, Mesajlar ve Kontrol

- Smalltalk'taki her nesnenin özellikleri ve davranışları vardır
- **Mesaj(Message)**: servis talebi
- **Alıcı(Receiver)**: mesaj alan nesne
- **Metot(Method)**: Smalltalk bir hizmeti nasıl gerçekleştirir
- **Gönderen(Sender)**: mesajın kaynağı
  - ▶ Verileri parametreler veya argümanlar şeklinde sağlayabilir
- **Mutatörler(Mutators)**: alıcı nesnede bir durum değişikliğine neden olan mesajlar



# Smalltalk'ın Temel Öğeleri

- **Mesaj geçme(Message passing)**: mesaj gönderme ve alma süreci
- **Arayüz(Interface)**: Bir nesnenin tanıdığı mesaj dizisi
- **Seçici(Selector)**: mesaj adı
- **Sözdizimi(Syntax)**: mesajı alan nesne önce yazılır, ardından mesaj adı ve herhangi bir argüman yazılır.
- Örnek: yeni bir set nesnesi oluşturmak:

`Set new "Yeni bir set nesnesi döndürür"`



# Smalltalk'ın Temel Öğeleri

- Yorumlar çift tırnak içine alınır
- **Show it** seçeneği: Smalltalk'ın girdiğiniz kodu değerlendirmesine neden olur **size** mesajı: bir kümedeki öğelerin sayısını döndürür  
Birden çok mesaj gönderebilir
  - ▶ Örnek: **Set new size "0 döndürür"**
  - ▶ **Set** sınıfı, **new** mesajını alarak bir **Set** örneğini döndürür ve örnek boyut mesajını alarak ve bir tamsayı döndürür .




# Smalltalk'ın Temel Öğeleri

- **Sınıf mesajı(Class message)**: sınıfa gönderilen mesaj
- **Örnek mesajı(Instance message)**: bir sınıfın bir örneğine gönderilen mesaj
- Önceki örnekte, `new` bir sınıf mesajıdır, `size` ise bir örnek mesajdır.
- **Tekli mesaj(Unary messages)**: argümentsiz
- **Anahtar kelime mesajları(Keyword messages)**: bağımsız değişkenler bekleyen mesajlar; mesaj ismi iki nokta üst üste ile biter
  - ▶ Örnek:  
`Set new includes: 'Hello' "false döndürür"`



# Smalltalk'ın Temel Öğeleri

- Birden fazla argüman varsa, her argümandan önce başka bir anahtar kelime gelmelidir
  - ▶ Anahtar kelime **at:** \_\_\_ **put:** \_\_\_ iki argüman bekler 
- Tekli mesajlar, anahtar kelime mesajlarından daha yüksek önceliğe sahiptir
  - ▶ Önceliği geçersiz kılmak için parantezler kullanılabilir
- **İkili mesajlar(Binary messages):** infix gösterimi ile aritmetik ve karşılaştırma ifadeleri yazılmasına izin verir





# Smalltalk'ın Temel Öğeleri

- Örnekler:

3 + 4                      *"7 döndürür"*

3 < 7                      *"true döndürür"*

- Nesnelere başvurmak için değişkenler kullanabilir

- Örnek:

```
| array |
array := Array new: 5.    "5 elemanlı bir dizi oluştur"
array at: 1 put: 10.      "5 değişiklik içeren ardışık
↪ işlemler"
array at: 2 put: 20.
array at: 3 put: 30.
array at: 4 put: 40.
array at: 5 put: 50.
Transcript show: array.   "Transcript ekranında gösterme"
```



# Smalltalk'ın Temel Öğeleri

- Geçici değişkenler dikey çubuklar arasında bildirilir ve büyük harfle yazılmaz
- İfadeler noktalara göre ayrılır
- Smalltalk değişkenlerinin atanmış veri türü yoktur
  - ▶ Herhangi bir değişken herhangi bir şeyi adlandırabilir
- Atama operatörü :=
  - ▶ Pascal ve Ada ile aynı



# Smalltalk'ın Temel Öğeleri

- Aynı nesneye yönelik bir dizi mesaj noktalı virgülle ayrılır:

```
| array |
array := Array new: 5.    "5 elemanlı bir dizi oluştur"
array at: 1 put: 10;      "5 değişiklik içeren ardışık
↪ işlemler"
    at: 2 put: 20;
    at: 3 put: 30;
    at: 4 put: 40;
    at: 5 put: 50.
Transcript show: array.  "Transcript ekranında gösterme"
```

- Smalltalk'ın değişkenleri değer semantiğini değil, referans semantiği kullanır
  - Bir değişken, bir nesneye atıfta bulunur; bir nesne içermiyor



# Smalltalk'ın Temel Öğeleri

- Eşitlik(Equality) operatörü `=`
- Nesne kimlik(identity) operatörü `==`

```
| array alias clone |
array := #(0 0).      "İki tamsayı bulunan dizi oluştur"
alias := array.       "Aynı nesneye ikinci bir referans oluştur"
clone := #(0 0).      "array nesnesinin bir kopyasını oluştur"
array == alias.       "true, aynı nesneye karşılık gelir"
array = alias.        "true, çünkü içerikleri aynı, == "
array = clone.        "true, aynı özelliklere sahip farklı nesneleri
↳ gösterirler"
array == clone.       "false, farklı nesneleri gösterirler"
clone at: 1 put: 1.   "clone dizisi artık 1 ve 0 değerlerini içerir"
array = clone.        "false, artık aynı özelliklere sahip değil"
```



# Smalltalk'ın Temel Öğeleri

- `to: do:` döngü oluşturur

```
| array |
array := Array new: 100.      "100 elemanlı bir dizi oluştur"
1 to: array size do: [:i |  "İndeks tabanlı döngü i
    ↪ değerini her adımda değiştirir"
    array at: i put: i * 10].
Transcript show: array.
```

- Kod **bloğu** köşeli parantez [] içine alınır
  - ▶ Blok, Scheme'deki lambda formuna benzer
  - ▶ **Blok değişkenleri(block variables)** olarak argümanlar içerebilir
- Smalltalk'ta, kontrol ifadeleri bile mesaj geçme(message passing) şeklinde ifade edilir.



# Smalltalk'ın Temel Öğeleri

- `ifTrue: ifFalse:` mesajları alternatif eylem tarzlarını ifade eder

```
| array |
array := Array new: 100.
1 to: array size do: [:i |
    i odd
        ifTrue: [array at: i put: 1]
        ifFalse: [array at: i put: 2]].
Transcript show: array.
```



- Bir dizinin içeriğini yazdırmak için:

```
array do: [:element| "Her elemanı transcript ekranına satır
    ↪ sonu karakterleri ile yazdır"
```

```
Transcript nextPutAll: element printString; cr]
```



Smalltalk, yinelemeleri gerçekleştirmek için birçok tür koleksiyon sınıfı ve mesaj içerir.

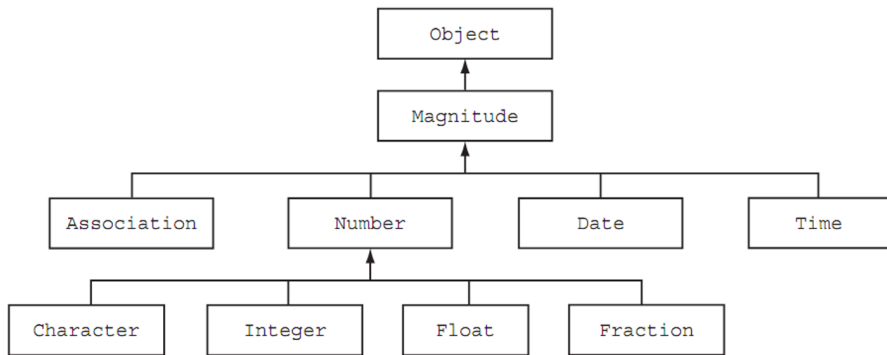


# Magnitude Hiyerarşisi

- Yerleşik sınıflar ağaç benzeri bir hiyerarşi içinde düzenlenir
  - ▶ Kök sınıfa `Object` denir
  - ▶ Sınıflar daha genelden daha özele doğru iner
- **Kalıtım(Inheritance)**: yapının ve davranışın yeniden kullanımını destekler
- **Çok biçimlilik(Polymorphism)**: farklı sınıflardan benzer hizmetler isteyen mesajlar için aynı isimlerin kullanılması
  - ▶ Başka bir kod yeniden kullanım biçimi



# Magnitude Hiyerarşisi



**Figure 5.1** Smalltalk's magnitude classes





# Magnitude Hiyerarşisi

- **Somut sınıflar(Concrete classes)**: nesneleri normalde programlar tarafından oluşturulan ve değiştirilen sınıflar
  - ▶ Örnekler: [Time](#) ve [Date](#)
- **Soyut sınıflar(Abstract classes)**: hiyerarşide altlarında bulunan sınıflar için ortak özelliklerin ve davranışların deposu görevi görür.
  - ▶ Örnekler: [Magnitude](#) ve [Number](#)
- Sınıfların ve metotların nasıl tanımlandığını görmek için Smalltalk sınıf tarayıcısını(class browser) kullanılabilir



# Magnitude Hiyerarşisi

The screenshot shows the Smalltalk IDE interface for the **Kernel-Numbers: Magnitude** class. The left pane displays a class hierarchy starting from **ProtoObject** down to **Magnitude**, which is highlighted. The middle pane shows the class's superclass (**Object**) and its subclasses (**Character**, **DateAndTime**, **TimeStamp**, **Duration**). The right pane lists the methods defined in the class, including **<**, **<=**, **<=>**, **=**, **>**, **>=**, **between:and:**, and **clampHigh:**. The bottom pane shows the source code for the **Magnitude** class, which is an abstract class that provides common protocols for objects that have instance variables, class variables, pool dictionaries, and a category.

```
Object subclass: #Magnitude
  instanceVariableNames: "
  classVariableNames: "
  poolDictionaries: "
  category: 'Kernel-Numbers'
```

I'm the abstract class Magnitude that provides common protocol for objects that have



# Magnitude Hiyerarşisi

The screenshot shows the Smalltalk IDE interface for the **Kernel-Numbers: Magnitude** class. The left pane displays the class hierarchy, with **Magnitude** selected under **Object**. The middle pane lists methods: **-- all --**, **\*Etoys-Squeakland-comparing**, **comparing**, **sorting**, **streaming**, and **testing**. The right pane shows a list of methods including **clampLow:high:**, **hash**, **hashMappedBy:**, **inRangeOf:and:**, **max:** (highlighted), **min:**, **min:max:**, and **putOn:**. Below the panes are tabs for **instance**, **class**, and **?**, followed by a row of navigation tabs: **browse**, **senders**, **implementor**, **versions**, **inheritance**, **hierarchy**, **vars**, and **source**. The main area displays the source code for the **max:** method:

```

max: aMagnitude
    "Answer the receiver or the argument, whichever has the greater
    magnitude."

    self > aMagnitude
        ifTrue: [ ^self ]
        ifFalse: [ ^aMagnitude ]
  
```

At the bottom, a status bar indicates: **no timeStamp · testing · 4 implementors · in no change set ·**



# Magnitude Hiyerarşisi

- Mesaj listesinde `>` operatörünü seçersek:

```
> aMagnitude
    "Answer whether the receiver is greater than the argument."

    ^aMagnitude < self
```

```
< aMagnitude
    "Answer whether the receiver is less than the argument."

    ^self subclassResponsibility
```

- Bir nesneye bir mesaj gönderildiğinde, Smalltalk mesaj adını uygun metoda bağlar.
- Dinamik veya çalışma zamanı bağlama(Dynamic or runtime binding):** nesneye yönelik sistemlerde yeniden kullanım için kodu organize etmenin önemli anahtarı



# Magnitude Hiyerarşisi

- Kalıtımı kullanmak için mevcut bir sınıftan yeni bir sınıf oluşturulur
- Kalıtım yoluyla yeni bir sınıf tanımlamak için **Classes** menüsünde **Add Subclass** seçeneğini kullanılır

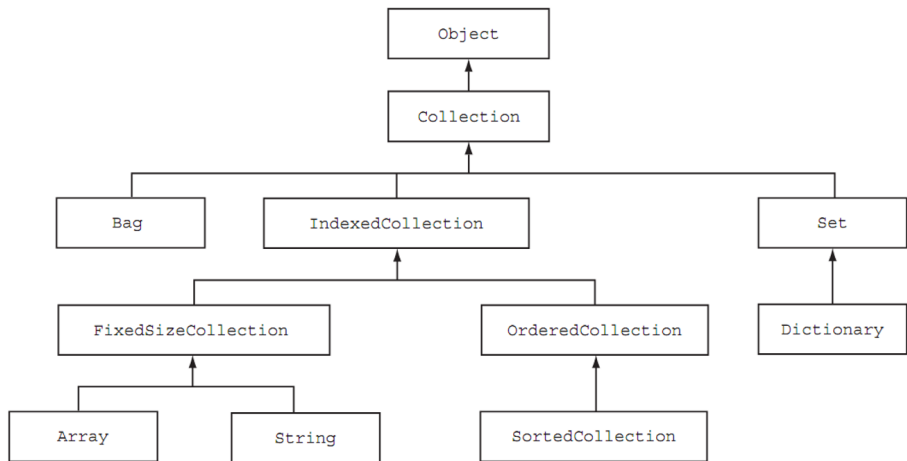


# Collection Hiyerarşisi

- Koleksiyonlar(Collections), öğeleri belirli bir şekilde düzenlenmiş kaplardır
  - ▶ Organizasyon türleri arasında doğrusal(linear), sıralı(sorted), hiyerarşik(hierarchical), çizge(graph) ve sırasız(unordered) türler yer alır
- Zorunlu dillerdeki yerleşik koleksiyonlar, tarihsel olarak diziler(array) ve dizelerle(string) sınırlı kalmıştır.
- Smalltalk, bir sınıf hiyerarşisinde organize edilmiş geniş bir koleksiyon türü kümesi sağlar
- Temel yineleyici(iterator) **do:**
  - ▶ Koleksiyon türüne göre değişen alt sınıflar tarafından uygulanır.



# Collection Hiyerarşisi



**Figure 5.4** The Smalltalk collection class hierarchy

# Collection Hiyerarşisi

**Tablo:** Smalltalk Collection sınıfındaki temel iterator türleri

Mesaj	Iterator Türü	Yaptığı iş
<code>do:aBlock</code>	Eleman tabanlı dolaşma	Alıcı koleksiyonundaki her bir öğeyi ziyaret eder ve argümanı her bir öğe olan bir bloğu değerlendirir
<code>collect:aBlock</code>	Map	Alıcı koleksiyonundaki her öğeyle bir bloğun değerlendirilmesinin sonuçlarının bir koleksiyonunu döndürür
<code>select:aBlock</code>	Filter in	Alıcı koleksiyonundaki bir bloğun true döndürmesine neden olan nesnelerin bir koleksiyonunu döndürür
<code>reject:aBlock</code>	Filter out	Alıcı koleksiyonundaki bir bloğun true döndürmesine neden olmayan nesnelerin bir koleksiyonunu döndürür
<code>inject:anObject into:aBlock</code>	Reduce or fold	Nesne çiftlerine anObject ve koleksiyondaki ilk öğeden başlayarak iki argümanlı bir blok uygular. Bloğu, önceki değerlendirmesinin sonucu ve koleksiyondaki bir sonraki öğe ile tekrar tekrar değerlendirir. Son değerlendirmenin sonucunu verir





# Collection Hiyerarşisi

- Smalltalk yineleyiciler oldukça polimorfiktir
  - ▶ Her tür koleksiyonla çalışabilir

```
'Hi there' collect: [ :ch |
    ch asUppercase].           "'HI THERE' döndürür"
'Hi there' select: [ :ch |
    ch isVowel].              "'iee' döndürür"
'Hi there' reject: [ :ch |
    ch=space].                "'Hithere' döndürür"
'Hi there' inject: space into: [ :ch1 :ch2 |
    ch1, ch2]                 "' H i   t h e r e ' döndürür"
```



# Collection Hiyerarşisi

- Smalltalk yineleyicileri `do:` ve `add:` metotlarına güvenir
  - ▶ Örnek: `collect:` yineleyicisi, fonksiyonel dillerdeki bir `map`'in polimorfik eşdeğeri

```
collect: aBlock
  "Evaluate aBlock with each of the receiver's elements as the
  ↪ argument. Collect the resulting values into a collection
  ↪ like the receiver. Answer the new collection."

  | newCollection |
  newCollection := self species new.
  self do: [:each | newCollection add: (aBlock value: each)].
  ^ newCollection
```



# Collection Hiyerarşisi

- Her tür koleksiyonu yerleşik koleksiyon türlerinin çoğuna dönüştürebilir

**asBag**

```
"Answer a Bag whose elements are the elements of the receiver."
^ Bag withAll: self
```

**withAll: aCollection**

```
"Create a new collection containing all the elements from aCollection."
^ (self new: aCollection size)
  addAll: aCollection;
  yourself
```

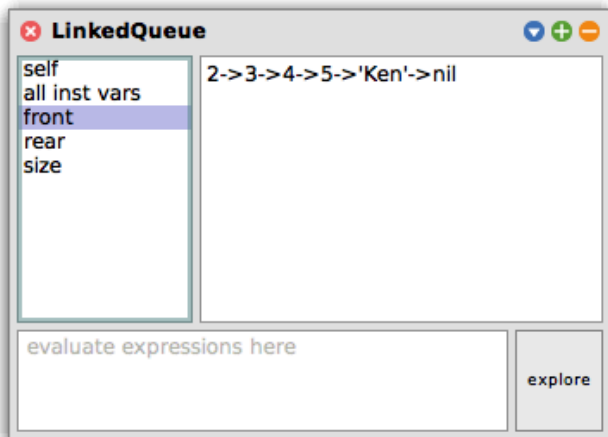
**addAll: aCollection**

```
"Include all the elements of aCollection as the receiver's elements."
→ Answer aCollection. Actually, any object responding to #do: can be
→ used as argument."
aCollection do: [:each | self add: each].
^ aCollection
```



# Collection Hiyerarşisi

- **inspect** mesajı, bir nesnenin örnek değişkenlerinin değerlerine göz atmak için alıcı nesnede bir denetçi penceresi açar.



# Collection Hiyerarşisi

- `LinkedList` sınıfı `Collection` sınıfından türetilen bir sınıftır
  - ▶ Smalltalk Collection hiyerarşisi içinde yer almaz
- Kitap içinde kodları mevcuttur
- Kullanımı aşağıdaki gibidir

```
| q |  
q := LinkedList new.  
q addAll: #(1 2 3 4 5).  
q dequeue.  
q enqueue: 'Ken'.  
q inspect.
```



# Smalltalk Sürümleri

- Squeak
- Pharo
- GNU Smalltalk



# Smalltalk Örnekleri

- Aşağıdaki işlevleri gerçekleştiren metotları yazınız:
  - ▶ "1 ile n arasındaki tek sayıların toplamını veren metot"
  - ▶ "Parametre olarak verilen koleksiyondaki elemanların toplamını veren metot"
  - ▶ "Parametre olarak verilen koleksiyondaki en küçük elemanı veren metot"
  - ▶ "Parametre olarak verilen koleksiyondaki tek elemanları veren metot"
  - ▶ "Parametre olarak verilen koleksiyondaki çift elemanları Transcript ekranına alt alta yazdıran metot"



# Java

- Başlangıçta cihazlara gömülü sistemler için bir programlama dili olarak tasarlanmıştır
  - ▶ Vurgu taşınabilirlik ve küçük ayak izi üzerineydi: "bir kez yazın, her yerde çalıştırın"
- Programlar makineden bağımsız bayt koduna derlenir
- Smalltalk'ta mevcut olmayan geleneksel sözdizimi, geniş bir kitaplık kümesi ve derleme zamanı tür denetimi desteği sağlar
- Bir istisna dışında tamamen nesne odaklıdır:
  - ▶ Skaler veri türleri (ilkel türler, primitive types) nesne değildir





# Java'nın Temel Öğeleri: Sınıflar, Nesneler ve Metotlar

- Bir Java programı, nesnelerin bir şeyler yapmasını sağlamak için sınıfları başlatır ve metotları çağırır
- Standart paketlerde birçok sınıf mevcuttur
  - ▶ Programcı tanımlı sınıflar, içe aktarılmak üzere kendi paketlerine yerleştirilebilir
- Değişken tanımı C'dekine benzer:

```
<class name> <variable name> = new <class name>(argument-1, ..., argument-n)
```

- Metot çağırısı Smalltalk'a benzer:

```
<object>.<method name>(argument-1, ..., argument-n)
```



# Java'nın Temel Öğeleri

- Örnek: program içe aktarılmış bir sınıf kullanıyor

```
import numbers.Complex;

public class TestComplex{
    public static void main(String[] args){
        Complex c1 = new Complex(3.5, 4.6);
        Complex c2 = new Complex(2.0, 2.0);
        Complex sum = c1.add(c2);
        System.out.println(sum);
    }
}
```



# Java'nın Temel Öğeleri

- **Sınıf metodu(Class method):** statik bir metot Java Sanal makinesi(Virtual Machine), programı TextComplex.main (<string dizisi>) olarak çalıştırır
  - ▶ Başlatma sırasında mevcut olan komut satırı argümanları args dizisine yerleştirilir
- Varsayılan olarak tüm sınıflar Object sınıfından miras alır
- Veri kapsülleme(encapsulation), private erişime sahip örnek değişkenleri bildirilerek uygulanır
  - ▶ Yalnızca sınıf tanımı içinde görülebilirler
- **Erişim metotları(Accessor methods):** programların bir sınıfın iç durumunu görüntülemesine izin verir ancak değiştirmesine izin vermez



# Java'nın Temel Öğeleri

- **Yapıcılar(Constructors)**: metotlar gibi, örnek değişkenler için başlangıç değerlerini belirtirler ve diğer başlatma eylemlerini gerçekleştirirler
  - ▶ **Varsayılan yapıcı(Default constructor)**: parametre içermez
  - ▶ **Yapıcı zincirleme(Constructor chaining)**: bir yapıcı diğerini çağırdığında
- Örnek değişkenlerine ve erişimci metotlarına **private** erişimin kullanılması, diğer kodları bozmadan veri temsilini değiştirmemize olanak tanır
- Java, referans semantiğini kullanır
  - ▶ Sınıflar ayrıca **referans türleri(reference types)** olarak da adlandırılır



# Java'nın Temel Öğeleri

- `==`, ilkel türler için eşitlik operatörüdür ve ayrıca referans türleri için nesne kimliği anlamına gelir
  - ▶ Object sınıfı, iki farklı nesnenin karşılaştırmasını uygulamak için alt sınıflarda geçersiz kılınabilen bir `equals` metodu içerir
- Metotlar, nokta gösterimi kullanılarak somutlaştırıldıktan (instantiation) sonra çağrılır: `z=z.add(w)` ;
- İşlemleri iç içe yerleştirebilir: `z=z.add(w).multiply(z)` ;
- Java, C++ gibi operatörün aşırı yüklenmesine izin vermez
- Java, birden fazla nesnenin bir metot çağrısının hedefi olabileceği **çoklu metotlara (multimethods)** izin vermez

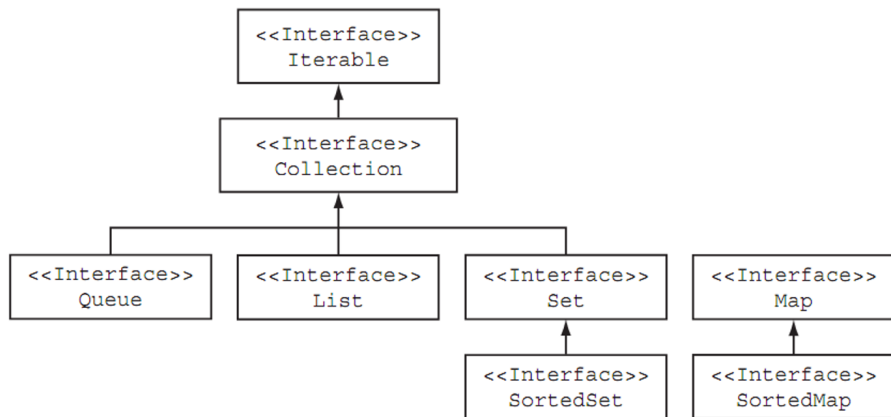


# Java Collection Çatısı: Arayüzler, Kalıtım ve Çok Biçimlilik

- **Çatı(Framework)**: ilgili sınıfların bir koleksiyonu
- `java.util` paketi: Java koleksiyon çatısını içerir
- **Arayüz(Interface)**: belirli bir türdeki nesneler üzerinde bir dizi işlem
  - ▶ Sistemlerdeki bileşenleri birbirine bağlayan yapılandırıcı görevi görür
  - ▶ Yalnızca tür adı ve bir dizi genel metot üstbilgisi içerir;  
Gerçekleştirici(implementer), işlemleri gerçekleştirmek için kodu içermelidir



# Java Collection Çatısı



**Figure 5.6** Some Java collection interfaces



# Java Collection Çatısı

```
public interface Collection<E> extends Iterable<E>{  
    boolean add(E element);  
    boolean addAll(Collection<E> c);  
    void clear();  
    boolean contains(E element);  
    boolean containsAll(Collection<E> c);  
    boolean equals(Object o);  
    boolean isEmpty();  
    boolean remove(Element E);  
    boolean removeAll(Collection<E> c);  
    boolean retainAll(Collection<E> c);  
    int size();  
}
```





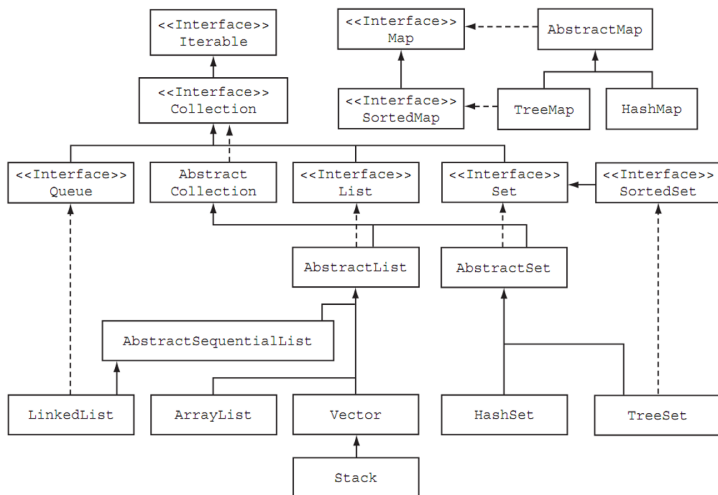
# Java Collection Çatısı

- Arayüzdeki ve metot başlıklarındaki `<E>` ve `E` **tür parametreleridir(type parameters)**
- Java statik tiplidir: tüm veri türleri derleme zamanında açıkça belirtilmelidir
- **Genel koleksiyonlar(Generic collections)**: parametrik polimorfizmden yararlanır
  - ▶ Java'nın ilk sürümlerindeki **ham koleksiyonlar(raw collections)**, parametrik polimorfizm yok
- Örnekler:

```
List<String> listOfStrings;  
Set<Integer> setOfIntegers;
```



# Java Collection Çatısı



**Figure 5.7** Some Java collection classes and their interfaces

# Java Collection Çatısı

- LinkedList gibi bazı sınıflar birden fazla arayüz uygular
  - ▶ Bir bağlı liste, bir liste veya bir kuyruk gibi davranabilir

```
List<String> listOfStrings = new LinkedList<String>();  
Queue<Float> queueOfFloats = new LinkedList<Float>();  
List<Character> listOfChars = new ArrayList<Character>();
```

- Farklı uygulamalara ve farklı öge türlerine sahip olsalar bile, iki List değişkeninde aynı metotlar çağrılabilir
- QueueOfFloats değişkeninin türü Queue olduğu için yalnızca Queue arabirim metotları çağrılabilir

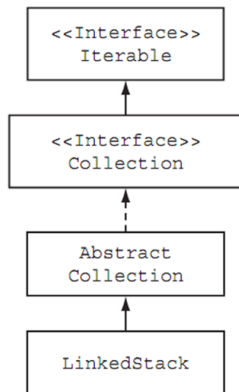


# Java Collection Çatısı

- Örnek: `AbstractCollection` sınıfının bir alt sınıfı olarak `LinkedList` adında yeni bir yığıt(stack) sınıfı türü geliştirilsin
  - ▶ Bize maliyetsiz olarak çok sayıda ek davranış verir
- **Özel iç sınıf(Private inner class):** başka bir sınıf içinde tanımlanan bir sınıf
  - ▶ İçeren sınıfın dışındaki hiçbir sınıfın onu kullanmasına gerek yoktur
- Java `Iterable` arayüzü yalnızca `iterator` metodunu içerir



# Java Collection Çatısı



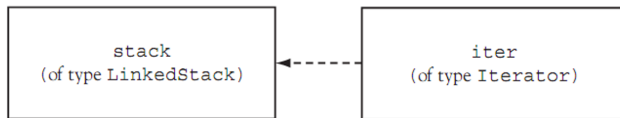
**Figure 5.8** Adding `LinkedStack` to the collection framework



# Java Collection Çatısı

- **Yedekleme deposu(Backing store):** iterator nesnesinin üzerinde çalıştığı koleksiyon nesnesi
- Örnek:

```
LinkedStack<String> stack = new LinkedStack<String>();  
Iterator<String> iter = stack.iterator();
```



**Figure 5.9** An iterator opened on a stack object



# Java Collection Çatısı

- Iterator türü, yedekleme deposuyla aynı öge türü için parametreleştirilir
  - ▶ Üç metodu belirten bir arayüzdür

```
public interface Iterator<E>{  
    public boolean hasNext(); //Daha fazla eleman varsa  
        ↪ true, aksi halde false  
    public E next(); //Mevcut elemanı döndürür ve  
        ↪ bir sonrakine geçer  
    public void remove(); //Mevcut elemanı siler  
}
```

- Yedekleme deposundaki her bir ögeyi ziyaret etmek için hasNext ve next metotlarını kullanılır

```
while (iter.hasNext())  
    System.out.println(iter.next());
```



# Java Collection Çatısı

- Java'nın geliştirilmiş **for** döngüsü, iterator-tabanlı while döngüsü için sözdizimsel şekerdir
  - ▶ Yalnızca bunu kullanmak için önkoşullar: koleksiyon sınıfı `Iterable` arayüzünü ve bir `iterator` metodu gerçekleştirmelidir

- Örnek:

```
for(String s : stack)
    System.out.println(s);
```





# Dinamik ve Statik Bağlama

- **Statik bağlama(Static binding)**: nesnenin gerçek sınıfını belirleyerek bir metodun hangi gerçeklemesinin kullanılacağını derleme zamanında belirleme süreci
  - ▶ Metot **final** veya **static** olarak bildirilmediği sürece gerçek kod derleyici tarafından üretilmez
- Java, derleme zamanında nesnenin metodunu belirleyemediğinde, **dinamik bağlama(dynamic binding)** kullanılır
- Java, dinamik bağlamayı gerçekleştirmek için bir miras ağacı aramasından daha verimli olan bir atlama tablosu(jump table) kullanır.



# Java'da Map, Filter ve Reduce Tanımlama

- Java 9 öncesinde, fonksiyonel programlama öğeleri yokken
- map, filter ve reduce, fonksiyonel bir dilde yüksek mertebeden fonksiyonlardır
  - ▶ Smalltalk'ta yerleşik koleksiyon metotları
- Temel iterator kullanarak Java'da map, filter ve reduce metotları tanımlamak mümkündür.
  - ▶ Iterators adlı özel bir sınıfta statik metotlar olarak tanımlanırlar
- map ve filter metotları bir girdi koleksiyonu bekler ve bir değer olarak bir çıktı koleksiyonu döndürür
  - ▶ Döndürülen asıl nesne, girdi koleksiyonuyla aynı somut sınıfta olacaktır.



# Java'da Map, Filter ve Reduce Tanımlama

- Bu metotlara kalan argüman olarak aktarılan işlemi nasıl temsil ederiz?
- Java'da, bu işlemi, yüksek mertebeden metodun gerçekleştirmesinde çağrılacak bir metodu tanıyan özel bir nesne türü olarak tanımlayabiliriz.

```
public static<E, R> Collection<R> map(<an operation>, Collection<E> c)
```

```
public static<E> Collection<E> filter(<an operation>, Collection<E> c)
```

```
public static<E> E reduce(E baseElement, <an operation>, Collection<E> c)
```



# Java'da Map, Filter ve Reduce Tanımlama

- Üç işlem gereklidir:
  - ▶ map metodunda, bir koleksiyon öğesini başka bir değere (belki farklı bir türden) dönüştüren tek argümanlı metot
  - ▶ filter metodunda, bir Boolean değeri döndüren tek argümanlı metot
  - ▶ reduce metodunda, aynı türden bir nesneyi döndüren iki argümandan oluşan bir metot
- Sonraki slaytta örnek:



# Java'da Map, Filter ve Reduce Tanımlama

```
package iterators;

public interface MapStrategy<E, R>{
    //E tipindeki bir elemanı, R sonuç türüne dönüştürür
    public R transform(E element);
}

public interface FilterStrategy<E>{
    //Eleman kabul edilirse true, aksi takdirde false verir
    public boolean accept(E element);
}

public interface ReduceStrategy<E>{
    //E tipindeki iki elemanı, E sonuç tipindeki bir elemana
    ↪ birleştirir
    public E combine(E first, E second);
}
```



# Java'da Map, Filter ve Reduce Tanımlama

- Bu strateji arayüzleri, uygulayıcıya uygun metodu tanıyan bir nesne beklemesini söyler.
  - ▶ Kullanıcıya, yalnızca bu arayüzlerden birini uygulayan bir sınıfın nesnesini sağlaması gerektiği söylenir.

```
public static<E, R> Collection<R> map(MapStrategy<E, R> strategy, Collection<E>  
↳ c)
```

```
public static<E> Collection<E> filter(FilterStrategy<E> strategy, Collection<E>  
↳ c)
```

```
public static<E> E reduce(E baseElement, ReduceStrategy<E> strategy,  
↳ Collection<E> c)
```



# Java'da Map, Filter ve Reduce Tanımlama

- MapStrategy arayüzü ve örnek gerçekleştirme

```
public interface MapStrategy<E, R>{  
    public R transform(E element);  
}
```

```
new MapStrategy<Integer, Double>(){  
    public Double transform(Integer i){  
        return Math.sqrt(i);  
    }  
}
```



# Java'da Map, Filter ve Reduce Tanımlama

- map, filter ve reduce fonksiyonlarının Java ve diğer dillerdeki sürümlerini karşılaştırma:
  - ▶ Fonksiyonel sürümler basittir: diğer fonksiyonları argüman olarak kabul ederler, ancak listeler ile sınırlıdır
  - ▶ Smalltalk sürümleri, herhangi bir koleksiyon üzerinde polimorfiktir ve bir lambda formundan daha karmaşık değildir.
  - ▶ Java sözdizimi biraz daha karmaşıktır
- Java'nın gerçek faydası, metotları neredeyse kusursuz hale getiren statik tür denetlemesidir.





# C++

- C++ ilk olarak Bjarne Stroustrup tarafından AT&T Bell Labs'ta geliştirilmiştir.
- Bir alt küme olarak C dilinin çoğunu, artı diğer özellikleri içeren, bazıları nesne yönelimli, bazıları olmayan bir uzlaşma dilidir.
- Artık çoklu kalıtım, şablonlar(templates), operatör aşırı yüklemesi(operator overload) ve istisnalar(exceptions) içerir



# C++ Temel Öğeleri: Sınıflar, Veri Üyeleri ve Üye İşlevleri

- C++, Java'ya benzer sınıf ve nesne bildirimlerini içerir
- **Veri üyeleri(Data members)**: örnek değişkenleri(instance variables)
- **Üye fonksiyonları(Member functions)**: metotlar
- **Türetilmiş sınıflar(Derived classes)**: alt sınıflar(subclasses)
- **Temel sınıflar(Base classes)**: üst sınıflar(superclasses)
- C++'daki nesneler otomatik olarak işaretçi veya referans değildir
  - ▶ C++'daki sınıf veri türü, C'deki **struct** veya kayıt(record) veri türü ile aynıdır



# C++ Temel Öğeleri

- Sınıf üyeleri için üç koruma seviyesi:
  - ▶ **Public** üyeler istemci kodu ve türetilmiş sınıflar tarafından erişilebilir
  - ▶ **Protected** üyeler istemci kodu tarafından erişilemez ancak türetilmiş sınıflar tarafından erişilebilir
  - ▶ **Private** üyeler istemci ve türetilmiş sınıflar tarafından erişilemez
- private, public ve protected anahtar sözcükler, yalnızca bireysel üye bildirimleri için geçerli olmak yerine (Java'da olduğu gibi) sınıf bildirimlerinde bloklar oluşturur



# C++ Temel Öğeleri

```
class A{  
    //Bir C++ sınıfı  
    public:  
    //Tüm public üyeler buraya  
    protected:  
    //Tüm protected üyeler buraya  
    private:  
    //Tüm private üyeler buraya  
};
```



# C++ Temel Öğeleri

- **Yapıcılar(Constructors)**: nesneleri Java'da olduğu gibi başlatır
  - ▶ Bir **new** ifadesinin yanı sıra bir bildirimin parçası olarak otomatik olarak çağrılabilir
- **Yıkıcılar(Destructors)**: bir nesne serbest bırakıldığında çağrılır
  - ▶ İsim tilde sembolünden önce gelir (~)
  - ▶ C++ 'da yerleşik çöp toplama olmadığı için gereklidir
- Üye fonksiyonları, bir sınıf adından sonra kapsam çözümleme operatörü :: kullanılarak bildirimin dışında uygulanabilir



# C++ Temel Öğeleri

- Bir sınıftaki uygulamalara sahip üye işlevlerinin **satır içi(inline)** olduğu varsayılır
  - ▶ Derleyici, fonksiyon çağrısını fonksiyonun gerçek koduyla değiştirebilir
- Örnek değişkenleri(instance variables), yapıcı(constructor) bildirimi ve gövde arasındaki virgülle ayrılmış bir listede iki nokta üst üste işaretinden sonra, başlangıç değerleri parantez içinde olacak şekilde başlatılır

```
Complex(double r=0, double i=0) : re(r), im(i) { }
```



# C++ Temel Öğeleri

- Yapıcı, parametreleri için varsayılan değerlerle tanımlanır

```
Complex(double r=0, double i=0)
```

- Bu, nesnelerin 0 ile bildirilen tüm parametreler arasında oluşturulmasına izin verir ve aşırı yüklenmiş oluşturucular oluşturma ihtiyacını ortadan kaldırır.

- Örnek:

```
Complex i(0,1); // i, (0, 1) olarak başlatıldı  
Complex y;      // y, (0, 0) olarak başlatıldı  
Complex x(1);   // x, (1, 0) olarak başlatıldı
```



# Genel Bir Koleksiyon Gerçekleştirmek için Şablon Sınıfı Kullanma

- **Şablon sınıfları (Template classes):** C++ 'da genel koleksiyonları tanımlamak için kullanılır
- C++ standart şablon kitaplığı (STL) birkaç yerleşik koleksiyon sınıfını içerir
- Örnek: Bölümde daha önce sunulan Java sürümüyle aynı temel arayüzle oluşturulmuş bir C++ `LinkedList` sınıfı kullanmak
  - ▶ Yeni `LinkedList` nesnesi otomatik olarak oluşturulur ve bu değişkenin bildirimde kullanılması üzerine `stack` değişkenine atanır.





# Genel Bir Koleksiyon Gerçekleştirmek için Şablon Sınıfı Kullanma

```
#include <iostream>
using std::cout;
using std::endl;
#include "LinkedStack.c"

int main(){
    int number;
    LinkedStack<int> stack;
    for(number = 1; number <= 5; number++)
        stack.push(number)
    cout << "The size is" << stack.size() << endl;
    while (stack.size() != 0)
        cout << stack.pop() << " ";
    cout << endl;
}
```



# Genel Bir Koleksiyon Gerçekleştirmek için Şablon Sınıfı Kullanma

- Şablon sınıfı, sınıf başlığında `template` anahtar kelimesi oluşturulur  
Örnek: `template <class E>`
- Bu sınıf, düğümleri için dinamik depolamayı kullanacağından, bir yıkıcı içermelidir



# Statik Bağlama, Dinamik Bağlama ve Sanal(Virtual) Fonksiyonlar

- Üye fonksiyonların dinamik bağlanması, C++ 'da bir seçenektir, ancak varsayılan değildir
  - ▶ Yalnızca **virtual** anahtar sözcüğüyle tanımlanan işlevler dinamik bağlama için adaydır
- **Saf sanal bildirim(Pure virtual declaration)**: 0 ve **virtual** anahtar sözcüğü ile bildirilen bir fonksiyon
  - ▶ Örnek:  
`virtual double area() = 0; //pure virtual`
  - ▶ Fonksiyon soyuttur ve çağrılmaz
  - ▶ İçeren sınıfı soyut hale getirir
  - ▶ Türetilmiş bir sınıfta geçersiz kılınmalıdır(overriden)



# Statik Bağlama, Dinamik Bağlama ve Sanal(Virtual) Fonksiyonlar

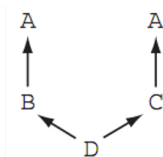
- Bir fonksiyon sanal olarak bildirildiğinde, C++ 'daki tüm türetilmiş sınıflarda bu şekilde kalır.
- Bir metodu sanal olarak bildirmek, dinamik bağlamayı etkinleştirmek için yeterli değildir
  - ▶ Nesne ya dinamik olarak tahsis edilmeli ya da başka bir şekilde bir referans aracılığıyla erişilmelidir
- C++, virgülle ayrılmış temel sınıflar listesi kullanarak çoklu miras sunar
  - ▶ Örnek: `class C : public A, private B {...};`



# Statik Bağlama, Dinamik Bağlama ve Sanal(Virtual) Fonksiyonlar

- Çoklu kalıtım, genellikle bir kalıtım yolundaki her bir sınıfın ayrı kopyalarını oluşturur
  - ▶ Örnek: D sınıfının bir nesnesinde A sınıfının iki kopyası vardır

```
class A {...};  
class B : public A {...};  
class C : public A {...};  
class D : public B, public C {...};
```



**Figure 5.10:** Inheritance graph showing multiple inheritance

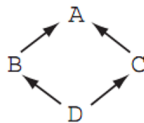
- Buna **tekrarlanan miras(repeated inheritance)** denir



# Statik Bağlama, Dinamik Bağlama ve Sanal(Virtual) Fonksiyonlar

- D sınıfında A'nın tek bir kopyasını almak için **virtual** anahtar sözcüğü kullanılır ve **paylaşılan kalıtım(shared inheritance)** neden olur

```
class A {...};  
class B : virtual public A {...};  
class C : virtual public A {...};  
class D : public B, public C {...};
```



**Figure 5.11:** Inheritance graph showing shared inheritance



# Nesne Tabanlı Dillerde Tasarım Sorunları

- Nesneye yönelik özellikler, statik yetenekleri değil dinamik yetenekleri temsil eder
  - ▶ Ekstra esnekliğin çalışma süresi cezasını azaltacak şekilde özellikler sunulmalıdır
- Satır içi (inline) fonksiyonlar, C++ 'da bir verimliliklerdir
- Nesne yönelimli diller için diğer sorunlar, çalışma zamanı ortamının doğru organizasyonu ve bir çevirmenin optimizasyonları keşfetme yeteneğidir.
- Programın kendisinin tasarımı, nesne yönelimli bir dilden maksimum fayda sağlamak için önemlidir.



# Sınıflar ve Türler

- Sınıflar bir şekilde tür sistemine dahil edilmelidir
- Üç olasılık:
  - ▶ Sınıflar tür denetiminden özellikle hariç tutulur: nesneler tipsiz varlıklar olur
  - ▶ Sınıflar tür yapıcılar yapılr: sınıflar dil türü sisteminin parçası olur (C++ tarafından benimsenmiştir)
  - ▶ Sınıfların tür sistemi haline gelmesine izin verilir: tüm diğer yapılandırılmış türler daha sonra sistemden çıkarılır





# Sınıflar ve Modüller

- Sınıflar, kodu düzenlemek için çok yönlü bir mekanizma sağlar
  - ▶ Java dışında, sınıflar uygulamanın arayüzden temiz bir şekilde ayrılmasına izin vermez ve uygulamayı istemci koduna maruz kalmaktan korumaz.
- Sınıflar, adların ayrıntılı bir şekilde içe ve dışa aktarımını kontrol etmek için yalnızca marjinal olarak uygundur.
  - ▶ C++ bir namespace mekanizması kullanır
  - ▶ Java bir paket(package) mekanizması kullanır



# Kalıtım ve Çok Biçimlilik

- Dört temel polimorfizm türü:
  - ▶ Parametrik polimorfizm: tür parametreleri bildirimlerde belirtilmeden kalır
  - ▶ Aşırı yükleme (ad hoc polimorfizm): farklı fonksiyon veya metot bildirimleri aynı adı paylaşır ancak her birinde farklı türde parametrelere sahiptir.
  - ▶ Alt tip polimorfizmi: bir türdeki tüm işlemler başka bir türe uygulanabilir
  - ▶ Kalıtım: bir tür alt tip polimorfizmi



# Kalıtım ve Çok Biçimlilik

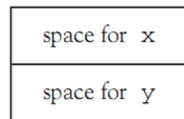
- **Çift dağıtım(Double-dispatch)** (veya **çoklu dağıtım(multi-dispatch)**) sorunu: kalıtım ve aşırı yükleme, iki veya daha fazla parametrede sınıf üyeliğine bağlı olarak aşırı yükleme gerektirebilecek ikili (veya n-ary) metotları hesaba katmaz
- C++ 'da, serbest (aşırı yüklenmiş) bir operatör işlevinin tanımlanması gerekebilir
- Bu sorunu çözme girişimleri **çoklu metotlar(multimethods)** kullanır: birden fazla sınıfa ait olabilen veya aşırı yüklenmiş dağıtımı birkaç parametrenin sınıf üyeliğine dayalı olabilen metotlar



# Nesnelerin ve Metotların Gerçekleştirilmesi

- Nesneler tipik olarak tam olarak C veya Ada'da kayıt yapıları gibi uygulanır.
- Örnek değişkenler, yapıdaki veri alanlarını temsil eder
  - ▶ Örnek:

```
class Point{  
    ...  
    public void moveTo(double dx, double dy){  
        x += dx;  
        y += dy;  
    }  
    ...  
    private double x,y;  
};
```



**Figure 5.12:** Space allocated for a C struct



# Nesnelerin ve Metotların Gerçekleştirilmesi

- Bir alt sınıfın bir nesnesi, önceki veri nesnesinin bir uzantısı olarak tahsis edilebilir, yeni örnek değişkenleri için kaydın sonunda alan tahsis edilir.

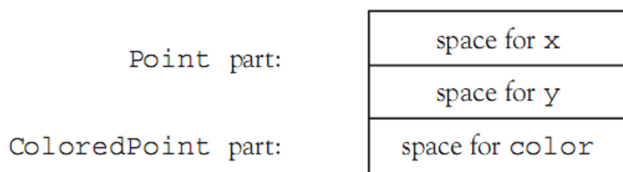
► Örnek:

```
class ColoredPoint extends Point{  
    ...  
    private Color color;  
}
```



# Nesnelerin ve Metotların Gerçekleştirilmesi

- Sonunda ayırma yaparak, temel sınıfın örnek değişkenleri, temel sınıfın herhangi bir nesnesi için ayrılan alanın başlangıcından itibaren aynı ofsette bulunabilir.



**Figure 5.13:** Allocation of space for an object of class ColoredPoint



# Kalıtım ve Dinamik Bağlama

- Her nesne için yalnızca örnek değişkenler için alan tahsis edilir
  - ▶ Metotlar için tahsis sağlanmaz
- Metotlar için dinamik bağlama kullanıldığında bu bir sorun haline gelir
  - ▶ Bir çağrı için kullanılacak kesin metot, yürütme dışında bilinmemektedir.
- Olası çözüm, dinamik olarak bağlı tüm metotları, her nesne için ayrılmış yapılarda doğrudan ekstra alanlar olarak tutmaktır.



# Tahsis(Allocation) ve Başlatma(Initialization)

- C++ gibi nesne yönelimli diller, C'nin geleneksel stack/heap biçiminde bir çalışma zamanı ortamını sürdürür
- Bu, nesneleri stack veya heap üzerinde tahsis etmeyi mümkün kılar
  - ▶ Java ve Smalltalk nesneleri heap alanında tahsis eder
  - ▶ C++, bir nesnenin doğrudan stack üzerinde veya bir işaretçi olarak tahsis edilmesine izin verir
- Smalltalk ve Java'nın açık serbest bırakma(deallocation) yordamları yoktur, ancak bir çöp toplayıcı kullanırlar
  - ▶ C++ yıkıcıları kullanır

