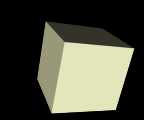
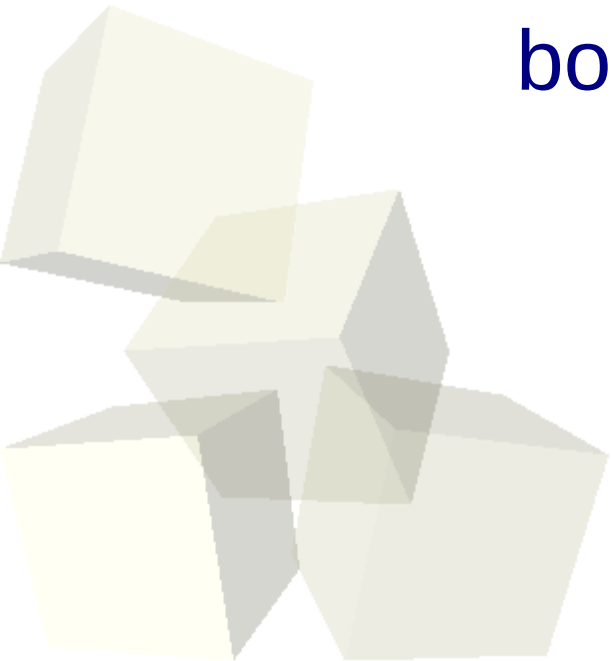


Genel Ağ Programlama Kavramları

Bora Güngören
Portakal Teknoloji
bora@portakalteknoloji.com



Genel Sunum Bilgisi ve Lisans

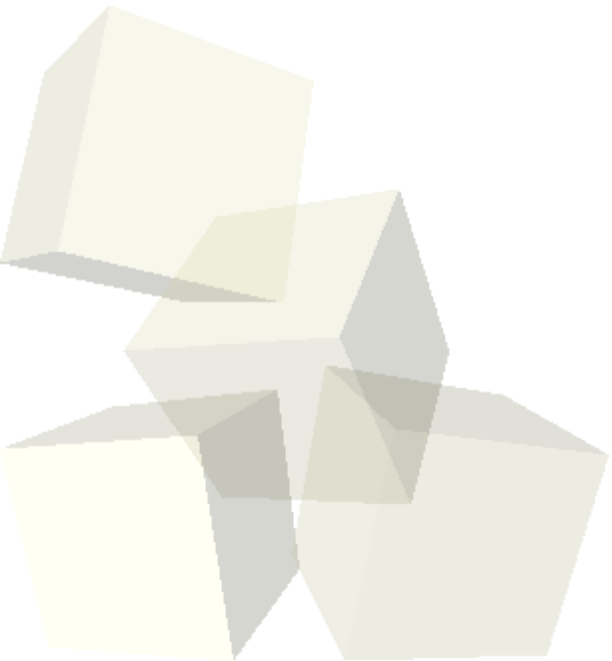
- Sunum 15 Aralık 2004 günü Erciyes Üniversitesi'nde LKD adına verilmiştir.
- Sunuma ait notlar GPL belgeleme lisansı ile dağıtılmaktadır ve LKD'den yada sunumu hazırlayan kişiden temin edilebilir.
  - GPL hakkında detaylı bilgi LKD İnternet sitesi olan http://linux.org.tr/ den yada GNU İnternet sitesi olan http://www.gnu.org/ dan alınabilir.
  - Kısaca özetlemek gerekirse, bu sunum notlarından yararlanmakta özgürsünüz. İçinden parçalar alıp kendi materyalinize eklemek isterseniz, kendi materyalinizi de GPL olarak sunduğunuz sürece, bunda da özgürsünüz.

Sunum İçeriği

- Yazılım mimarisi
  - Yazılım mimarisi tanımı
  - Mimari tasarım
  - İstemci sunucu uygulamaları
- Ağ programcılığı
  - Soket kavramı
  - Java'da soketler
  - Uygulama sözleşmesi (protokol) tasarımı

Sunum İçeriği

- Java RMI
  - Uzaktan Yordam Çağırma (RPC)
  - Java RMI Mimarisi
  - JRMP ve CORBA/IIOP
  - RMI Uygulaması Yazmak
  - Java 5.0 RMI değişiklikleri
  - Java RMI Güvenliği



- Bilgisayarların kullanımı yaygınlaştıkça, yazılımlardan beklenenler artmıştır. Bunun sonucu olarak giderek daha karmaşık yazılımlar üretiliyor.
  - Ancak bir yazılımının “iyi” olması yalnızca kendisinden bekleneni yapmasına değil, bir çok başka şeye de dayalıdır. Örneğin kodlarının yeniden kullanılabilmesi yada değiştirilmek için ele alındığı zaman kolayca anlaşılması gereklidir.
  - Bu gereksinimler, yazılımları tasarlarken daha farklı kriterleri de dikkate almamıza neden olmaktadır.

Yazılım Mimarisi Tanımı

- Yazılım mimarisi (software architecture), aşağıdaki sorular ile uğraşan bir yazılım mühendisliği alt başlığıdır.
  - Yazılımların bileşenlerinin nasıl bir araya getirileceği
  - Bileşenlerin basıl birlikte çalışacağı
  - Bileşenlerin ne şekilde iletişim kuracağı
  - Her bir bileşenin sistemin hangi amaçlarını sağlayacağı
- Yazılım mimarisi parçalardan oluşan bütüne odaklanmıştır.

Yazılım Mimarisi Tanımı

- Bu soruların yanıtlarını spesifik bir yazılım için aramak, o yazılımın mimarisini belirlemektir. Bu nedenle aşağıdaki tanımı tercih ediyoruz.
  - Bir **yazılım mimarisi**, bir yazılım sisteminin organizasyonu, yapısal elemanların ve sistemin içinde ve dışındaki bağlantı arabirimlerinin, davranışları ve elemanlar arasındaki etkileşimlerle birlikte seçimi, bu elemanların ve arabirimlerin sürekli olarak bir araya getirilerek daha büyük yapısal elemanların kurulması ve bu sürecin sistemin bütününe ulaşana kadar sürmesi boyunca alınan **bir dizi karar**dır.

Yazılım Mimarisi Tanımı

- Bu tanım UML'in yaratıcıları olan Booch, Rumbaugh ve Jacobson üçlüsünün kendi tanımlarıdır ve son 1991 yılında yapılmıştır.
  - Tanımın 10 yıldan uzun süredir değişikliğe gerek duymaması genelliğini göstermektedir.
  - Değişik teknolojiler, değişik mimariler ve diğer çok çeşitli tasarım seçenekleri ortaya çıkmış ve çıkmaktadır ancak bu tanım için bir farklılık yoktur.

Yazılım Mimarisi Tanımı

- Yazılım mimarisini, bir tasarım süreci olarak gören bu tanım, süreci de değişik ölçeklerdeki tasarım kararları (design decisions) olarak tanımlar.
  - Bu yaklaşım, hiyerarşiler kurma çabasından uzak olduğu için saf bilimsel yaklaşımdan uzak olsa da pratikte çok işe yarar.

- Günümüzde daha önceden geliştirilen yazılımların incelenmesi sonucunda zaten ortaya konmuş ve belgelenmiş çok çeşitli yazılım mimarileri vardır.
  - Mimari tasarım sırasında öncelikle bu mimarilerin eldeki projeye uygun olup olmadığı incelenir.
  - Eğer eldeki standart mimariler işe yaramıyorsa, sıfırdan bir mimari tasarlanır.
  - Belli başlı yazılım mimarilerini iyi tanımak, ve avantajları ile dezavantajlarını sıralayabilmek bu nedenle önemli bir bilgidir.

- Peki bir yazılım mimarisini tasarlayan kişi, yani **yazılım mimarı** (software architect) yada **sistem mimarı** (systems architect) ne yapar?
  - Bir **yazılım mimarı** (software architect), yazılımı oluşturan algoritma ve veri yapılarının detaylarının dışındaki **yapısal problemleri çözer**.
  - Bu problemler, bileşenlerin bir araya gelmesinde ortaya çıkar ve çözümleri tasarım kararlarına neden olur.

- Bu problemleri örneklemek gerekirse ilk akla gelenler:
  - İletişim sözleşmeleri
  - Eş zamanlı çalışma
  - Veri erişimi
  - İşlevsel gereksinimlerin bileşenlere dağıtılması
  - Sistemin mantıksal ve fiziksel olarak dağılımı
  - Ölçeklenebilirlik
  - Performans
  - Yüksek bulunurluk

- Yazılım mimarisi alanında önemli çalışmaları bulunan Mary Shaw, yazılım mimarisinde kullanılan modelleri sınıflandırarak bu alanda önemli bir adım atmıştır.
  - Yapısal modeller (structural models)
  - Çatı modelleri (framework models)
  - Dinamik modeller (dynamical models)
  - Süreç modelleri (process models)

İstemci Sunucu Mimarisi

- **İstemci sunucu** (client server) mimarisi, 1980'li yıllarla birlikte popülerleşen ve günümüzde de yoğun biçimde kullandığımız bir mimaridir.
  - Mimarinin tipik özeti, bir bileşenin (sunucu) diğer bileşenlerin (istemciler) **çeşitli isteklerini yerine getirmek için** (hizmet) **hazır beklemesi**dir.
  - İstemciler söz konusu hizmeti almak istedikleri zaman sunucu ile bir **bağlantı** kurarlar.
    - Zorunlu olmasa da tipik olarak her istemci için ayrı bir bağlantı açılır.
    - Bağlantı soyutlamasının hangi seviyede sağlanacağı değişken olabilir.

- Bir sunucunun tipik yaşam çevrimi aşağıdaki gibi olur:
  - Sunucu başlatılır. Ayarları yüklenir.
  - Sunucu bağlantılara hizmet vermeye hazır olduğu zaman **bağlantı kabul etmeye** başlar.
  - Sunucu sürekli olarak yeni istemcilerin açtığı bağlantılar üzerinden istemcilere yanıt vermeye başlar.
  - Sunucu kapatılmadan önce **yeni bağlantı kabul etmeyi bırakır**. Belli bir zaman aşımı süresi içine o anda aktif olan bağlantıların hizmetlerini verir yada veremez. Hizmet veremediği bağlantıları tek taraflı olarak kapatır.
  - Sunucu kapanır.

- Görüldüğü gibi sunucunun tasarımında ve gerçeklenmesinde değişik alt problemler bulunmaktadır.
  - Sunucu ayarlarının yapılması.
  - Sunucunun başlatılması ve ayarlarının yüklenmesi.
  - İstemcilerle olan iletişimin sağlanması.
  - Verilen hizmetin kendi özel detayları.
  - Sunucunun kapatılması süreci.
  - Performans.
  - Güvenlik.

- Bir istemcinin tipik yaşam çevrimi ise aşağıdaki gibidir.
  - İstemci başlatılır. Gerekli gördüğü hizmet için sunucuya bağlanması gerekene kadar diğer işlemleri yapar.
  - Sunucuya bir bağlantı istediğini belirtir.
  - Sunucu bağlantı için onay yollarsa bağlantıyı açar. Genelde onay, bağlantı ile ilgili bazı detaylar içerir.
  - Hizmet bağlantı üzerinden alınır.
  - Hizmet alındıktan sonra sunucuya bağlantıyı kapatma mesajı yollanır. Sunucudan onay gelsin gelmesin, zaman aşımı sonrası bağlantı kapatılır.
  - İstemci çalışmaya devam eder.

İstemci Sunucu Mimarisi

- İstemciler genelde aynı anda tek bir sunucudan hizmet alacak şekilde tasarlanır. Bu nedenle tasarımları daha sade ve üretilmeleri daha kolaydır.
  - İstemci sunucu mimarisinde bir amaç da bu avantajın üzerine gitmek ve işin akışı üzerindeki detayları ya sunucuya yada istemci ile sunucu arasındaki iletişim tekniğine yıkmaktır.
  - Bu sayede çok çeşitli istemciler çok değişik biçimlerde kolayca yazılabilir.

İstemci Sunucu Mimarisi

- Bazı uygulamalarda aynı yazılımda bir istemci bir de sunucu becerisi bulunabilir.
- Örneğin bir vekil sunucusu (proxy)
  - İnternet tarayıcıları tarafından bakınca bir HTTP sunucusu,
  - Diğer HTTP sunucuları ve vekil sunucuları tarafından bakınca bir İnternet tarayıcısı olarak görülür.
- Bunun dışında eşler arası (peer-to-peer / P2P) ağ mimarileri de hem istemci hem sunucu görevi gören yazılımlardan oluşan yapılar olarak düşünülebilir.

- Günümüzde 3-katmanlı mimari, 4-katmanlı mimari, dağıtık mimari gibi farklı mimari yapılar öne sürülmektedir.
  - Bu yapıların hepsi de belli problemler için son derece yararlı mimarilerdir.
  - Ancak hepsi istemci sunucu mimarisine dayalı yapılardır. Örneğin,
    - Dağıtık bir mimarideki nesne isteği aracısı (object request broker) aslında özel bir sunucudur.
    - Java RMI mimarisi bir HTTP sunucusuna gereksinim duyar.
- Bu nedenle istemci sunucu mimarisinde asgari de olsa deneyim kazanmak, diğer mimarileri anlamak için gereklidir.

- İnternet'in yoğun biçimde kullanıldığı günümüzde bir çok uygulama bazı işlevlerini ağ üzerinden gerçekleştirmektedir.
  - Bir virüs tarama uygulaması, yeni çıkan virüsleri tanıtan imza bilgilerini ağ üzerinden üreticisine ait bir sunucuya bağlanarak alır.
  - Herhangi bir yazılım kendisine ait bileşenlerin güncellenip güncellenmediğini İnternet üzerinden test edip güncellemeleri çekecek ve kuracak şekilde yapılandırılabilir.
  - Eposta okuma yazılımı, web tarayıcısı, FTP istemcisi, anında mesajlaşma gibi uygulamalar zaten doğaları gereği ağa ihtiyaç duyarlar.

- **Ağ programcılığı**, genelde düşük seviye (yani daha temel) işlemleri de bizim tasarladığımız biçimde ağı kullanan uygulamalar geliştirmektir.
  - Pratikte, temel ağ işlemlerini sağlarken yazdığımız kodlar, soyutlamaları kullanarak çok daha büyük işleri yazdığımız kodlardan daha uzun ve karmaşık olur.
- Takip eden sürede bazı temel kavramları kullanarak standart istemci/sunucu modelinde uygulamalar yazmayı göreceğiz.

- Yazacağımız uygulamalar çok spesifik ağ iletişim sözleşmelerini kullanan uygulamalar olmayacak.
  - Bu sözleşmeleri kullanmak için birinci yol önce ilgili sözleşmeyi öğrenmek (RFC'sini okumak) ve ardından eldeki daha düşük seviye ağ kitaplıklarını kullanarak oldukça uzun ve karmaşık kodlar yazmaktır.
  - İkinci yol ise popüler sözleşmeler için hazırda bulunan, önceden yazılmış kitaplıkları kullanmaktır.
  - Kitaplıkların yaygın kullanımı nedeni ile kavramları özetlerken daha sade uygulamalar yazacağız.

- Internet ilkolarak UNIX işletim sistemi ile birlikte ortaya çıkmıştır.
  - Bu nedenle diğer tüm işletim sistemleri ve ağ programlamaya dayalı sistemler de UNIX ağ modelini örnek alarak geliştirilmiştir. Bu nedenle önncelikle UNIX'in sunduğu ağ modeli ile başlayacak ve soket kavramını açıklayacağız.

- UNIX ile ilgili olarak bilmeniz gereken öncelikli şey, UNIX'de her şeyin bir dosya olduğudur.
  - UNIX'de aygıtlar da dahil olmak üzere her şey bir dosya olarak soyutlanır ve dolayısı ile dosyaya yazmakta ve dosyadan okumakta kullanılan mekanizmalar ile her türlü aygıta erişmek mümkündür.
  - Bu basitlik sayesinde çok temel ağ işlemlerini de kolayca yaparız.

Soket Kavramı

- Ancak bir ağın içerisinde olduğumuz zaman, iki bilgisayar arasında sayıları sürekli değişen çok sayıda **bağlantı** (connection) sahibi olacağız.
  - Bu bağlantılar UNIX mantığında yerelde yani kendi bilgisayarımızda birer dosya olarak görülecektir.
  - Bu dosyaların yaratılması ve programcıya sağlanması için bir mekanizma gereklidir. Bu mekanizmayı **soket** (socket) olarak adlandırıyoruz.

- Bir soket, UNIX mantığında, **programda ağ bağlantısını simgeleyen bir dosya bağlantısı**dır.

- Soketler sağladıkları ağ bağlantılarının özellikleri göre türlü türlüdür. Ancak özellikle üzerinde durulması gereken iki tür soket vardır.
  - **Akış** (stream) soketleri
  - **Datagram** soketleri

- Akış soketleri (stream sockets), iki taraflı iletişim için kullanılırlar.
  - Java'daki yada C++'daki girdi çıktı modelindeki akışları düşünürseniz, verilerin gönderildikleri sıra ile iletildiklerini hatırlayacaksınız.
  - Bir akış soketi bunu garantiler.

- Aynı bilgisayar içinde bu normal bir sonuç gibi gelse de ağ söz konusu olduğu zaman iletilen verilerin gecikmeler yaşaması veya farklı yollardan hedefe ulaşması mümkündür.
  - Bir akış soketinin bunu garantilemesi önemli bir şeydir. Ayrıca, akış soketleri, belli bir ölçüde hata denetimi ve düzeltilmesini sağlayan teknikleri kullanırlar.
  - Ancak iletişim boyunca açık kalacak bir bağlantıya ihtiyaç duyarlar.

- **Datagram** soketleri İse, iletişim boyunca açık kalacak bağlantılara gerek duymazlar.
  - Bu nedenle **bağlantısız** (connectionless) soketlerdir.
  - Bu avantajlarına karşılık olarak bazı zayıf yönleri vardır. Bir datagram soketi ile yolladığınız verinin karşı tarafa ulaşacağı yada verilerin doğru sıra ile ulaşacağı **garanti edilmez**.
  - Ancak tıpkı akış soketleri gibi, ulaşan verinin doğru ulaşması için gerekli hata denetimi ve düzeltilmesi teknikleri kullanılır.

- Bu iki tekniğin de kullanım alanları vardır. Akış soketleri karşılıklı iletişimin olduğu iistemci/sunucu sistemlerinde kullanılır.
  - Örneğin bir sohbet programı yada bir Web sunucusu akış soketleri ile çalışacaktır.
  - Ancak sadece bir kez, bir parça verinin iletileceği ve yanıtın da sadece “aldım” (acknowledge – ACK) biçiminde olacağı durumlarda da datagram soketlerini kullanmak verimli olacaktır.
  - Ayrıca ağ performansının aşırı yüksek olması istenen ancak arada bir veri kaybının çok önemli olmadığı durumlarda da datagram soketleri kullanılabilir.

- Akış soketleri TCP soketleri, datagram soketleri de UDP soketleri olarak da adlandırılır.
- Bunun nedeni bu iletişim protokollerini kullanmalarıdır. TCP için RFC 793, UDP için de RFC 791'i okuyabilirsiniz.

- Bir kez elinizde bir soket olduğu zaman bu soketi kullanarak bir **uygulama seviyesi sözleşmesi** (application level protocol) çerçevesinde başka bir uygulama ile haberleşebilirsiniz.
  - Soketler **sadece iki uygulama arasındaki haberleşme kanalını açar**lar.
  - O kanal üzerinden nasıl haberleşileceğini uygulama seviyesinde belirleyen bir **sözleşme** (protocol) olmalıdır.

- Bu durumda hangi dilde soket kullanırsak kullanalım elimizde iki aşamalı bir programlama modeli olacaktır.
  - Soketleri yaratmak iççin bir mekanizma.
  - Yaratılan soketleri kullanarak bir ağ sözleşmesi çerçevesinde veri alıp göndermek için bir mekanizma.

- Farketmiş olabilirsiniz, üst üste yerleşen bu katmanlar kendiliğinden bir veri saklama ilkesi oluşturmakta ve nesne tabanlı programlama için uygun bir ortam oluşturmaktadır.
  - C++ ve Java'nın bu alandaki popülerliği boşuna değildir.

Soket Kavramı

- Ağ programcılığı ile ilgili olarak öğrenmeniz gereken son temel kavram da **kapı** (port) kavramıdır.
  - Bir bilgisayarın ağa bağlanırken genelde tek bir fiziksel bağlantısı olur.
  - Bu modem ile bağlandığınız bir telefon hattı yada bir yerel ağ bağlantısı olabilir.
  - Ancak bu tek fiizksel bağlantı üzerinden gerekirse binlerce sanal bağlantının aynı anda yönetilmesi gerekecektir.

- Bu nedenle her birinin bir tamsayı numarası olan sanal **kapılar** (port) kullanılır.
  - Gelen ve giden veriler yanlarında hangi numaralı kapıyı kullanacaklarını da taşırlar.
  - Her bilgisayarda belli bir kapıyı dinleyen sadece bir uygulama olduğu sürece bir sorun olmadan tek fiziksel bağlantı ile binlerce ayrı ağ uygulaması çalışabilir.

- Tipik bir işletim sisteminde 0'dan 65535'e kadar numaralandırılmış kapılar olabilir.
  - UNIX ve UNIX benzeri sistemlerde (örneğin Linux) 0'dan 1023'e kadar numaralandırılmış kapılar sistem yöneticisi yada özel kullanıcılara ayrılır.
  - Bu nedenle, sistem yöneticisi yada özel bir kullanıcı kullanmayacaksa kendi uygulamalarımızda 1023'den büyük numaralı kapıları kulanmalıyız.
  - Örneğin Apache web sunucusu, apache adındaki özel kullanıcısı ile 80 numaralı kapıyı, Mozilla istemcisi ise 1160 numaralı kapıyı kullanacaktır.

- Bir istemci sunucu sistemi bir sunucu soketinden ve bir istemci soketinden oluşur.
  - Sunucu soketi, istemci soketlerinden gelen bağlantıları kabul eder.
  - Bu durumda aşağıdaki basit sunucu uygulaması en temel sunucu etkinliklerini sağlayacaktır.

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.*;


public class Sunucu{

    protected static final int KAPI = 6000;
    protected static ServerSocket s = null;
    protected static Socket soket = null;
```

```
protected static void init(){
    try {
        s = new ServerSocket (KAPI);
        soket = s.accept();
    }
    catch(IOException ioe){
        ioe.printStackTrace();
        System.exit(0);
    }
}
```

```
protected static void destroy(){
        try{
                soket.close();
                s.close();
         }
         catch(IOException ioe){
                ioe.printStackTrace();
                System.exit(0);
         }
      }
```

```
public static void main (String args[]){

        System.out.println (
                "Sunucu basliyor..."
                        );

        init();

        BufferedReader okuyucu = null;
```

```
        try{
okuyucu = new BufferedReader (
                    new InputStreamReader(
                        soket.getInputStream ()
                        )
                    );

                System.out.println
    (okuyucu.readLine ());
        }
```

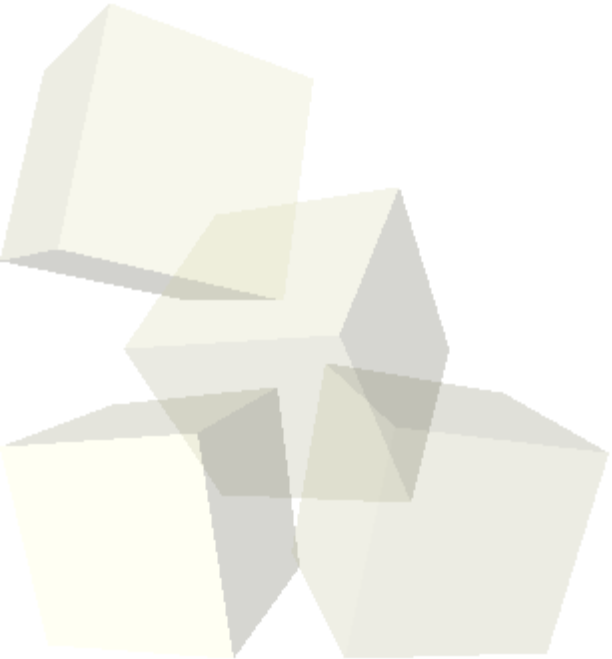
```
        catch(IOException ioe){
                ioe.printStackTrace();
    }

        destroy();
      }
}
```

- Dikkat ederseniz burada **IOException** hata durumu ile karşılaşabiliyoruz. Bunun nedeni soketlerin arka planda dosya sistemleri ile çalışmasıdır. Java'da bir soketin çeşitli durumları olabilir.
  - Yaratılmış (created)
  - Bağlanmış (bound)
  - Bağlantı kurmuş (connected)
  - Kapatılmış (closed)
- **ServerSocket** sınıfının **accept**() yöntemi bize geçerli ve bağlantı kurmuş (connected) bir soket nesnesine referans sağlamaktadır.

- Artık bu referans ile eriştiğimiz soket nesnesini kullnarak ağ bağlantıları kurabiliriz. Bu yöntemle ilgili olarak dikkat edilmesi gereken şeyler vardır.
  - Öncelikle yöntem bağlantı kurulana kadar çalışır. Bazı durumlarda bu uzun sürebilir. Bu nedenle bu yöntemi kullanacağımız kodları ayrı bir kanalda bulundurmak isteyebiliriz.
  - Yöntem çeşitli hata durumları yaratabilir. Bağlantı kurulması sırasında girdi / çıktı hataları oluşursa **IOException** hata durumu alabiliriz. Ayrıca bir zaman aşımı tanımlanmış ise **SocketTimeOutException** hata durumu da alabiliriz.

- Buna ek olarak güvenlik ayarlarımız varsa **accept()** yöntemi bize **SecurityException** hata durumu da verebilir.
  - Çünkü yaratılan soketin bağlanacağı sunucu ve kapı denetlenir.
- **ServerSocket** sınfı Java 2 SDK 1.4 için yeniden tasarlanmıştır.
  - Bu nedenle 1.4 öncesi derleyicilerle üretilen kodlar şu anda sorunsuz çalışsa bile, güvenlik ve performans amaçlarıyla, 1.4 yada 5.0 derleyici ile yeniden derlenmeleri önerilir.

- Şimdi bu sunucu yazılımına bağlanacak bir istemci yazılımı yazalım.



```
import java.net.*;
import java.io.*;

public class Istemci{

protected static final int KAPI = 6000;
protected static Socket soket = null;
```

```
protected static void init(){
    try{
InetAddress hedef =
InetAddress.getLocalHost();
soket = new Socket(hedef,KAPI);
    }
    catch(IOException ioe){
ioe.printStackTrace();
System.exit(0);
    }
}
```

```
protected static void destroy(){
    try{
soket.close();
    }
    catch(IOException ioe){
ioe.printStackTrace();
System.exit(0);
    }
}
```

```
public static void main (String args[]){

    init();

    try {
                BufferedWriter cikti = null;
                String mesaj = "Mesaj Metni";

                System.out.println ("Sunucuya
yollaniyor:"
                        + mesaj);
```

```
            cikti = new BufferedWriter (
                    new OutputStreamWriter (
                      soket.getOutputStream ()
                    )
                );

            cikti.write (mesaj, 0, mesaj.length());
            cikti.flush ();
        }
```

```
    catch(IOException ioe){
ioe.printStackTrace();
    }

    destroy();

     }
   }
```

- İstemci sınıfı da sunucu sınıfını andırmakla birlikte temel kavramlardan gelen farklılıkları var.
  - Bir bilgisayarda aynı anda birden fazla istemci aynı kapıyı kullanabilir. Bu nedenle o kapıya bağlanmak (bind) gibi bir etkinliğe gerek yoktur.
  - İstemcilerin soketlerini hedeflenen bir Internet adresi ve kullanılacak kapı numarası ile yaratırız.

- Uygulamayı test etmek istersek
  - Önce sunucuyu başlatacağız
  - Sonra istemciyi çalıştıracağız.
- Sunucuyu çalıştırmak için

```
$ java Sunucu
Sunucu basliyor...
```

- Burada **netstat** komutu ile TCP soketlerini bakmanızı öneririm. **6000 numaralı kapının dinlendiğini** göreceksiniz.

- İstemciyi çalıştırınca

```
$ java Istemci
Sunucuya yollaniyor:Mesaj Metni
$
```

- Bu arada sunucumuzun olduğu konsolda değişiklik oldu:

```
$ java Sunucu
Sunucu basliyor...
Mesaj Metni
```

- Buradaki sunucumuz bir tek mesaj aldıktan sonra sonlanmaktadır.
  - İstersek bu temel tasarımı geliştirebiliriz.

- Daha önce bağlantısız iletişimin de yararlı olabileceği örnekler vermiştik.
  - Datagram soketleri, yada diğer adıyla UDP soketleri, bu türdeki iletişimde kullanılır.
  - Java'da datagram haberleşmesi için yine java.net paketi içinde bir çift sınıf bulunmaktadır.

- Örnek uygulamamızda bir istemci, sunucudan o andaki döviz kurunu alacaktır.
  - Sunucu bunu istemcilere döviz kurlarının olduğu bir veri yapısını aktararak yapacaktır.
  - İstemciler de ekrana bu döviz kurlarını basacaktır.

```
import java.net.*;
import java.io.*;
import java.util.*;

class DatagramSunucu extends Thread{

protected static final int BOYUT = 10000;
protected static Map<String,Integer> kurlar =
   null;
protected DatagramSocket dgSoket = null;
```

```
DatagramSunucu(){
    this("sunucu");
}

DatagramSunucu(String ad){
    super(ad);
    kurlar = new HashMap<String,Integer>(10);
    kurlar.put("USD", 1500000);
    kurlar.put("EUR", 1800000);
    kurlar.put("YTL", 1000000);
```

```
    try{
dgSoket =
new DatagramSocket(6000);
    }
    catch(SocketException se){
        se.printStackTrace();
    }
}
```

```
public void run(){
    System.out.println("Sunucu calisti...");

    while(true){
try{
    byte[] tampon = new byte[BOYUT];
    DatagramPacket paket =
new DatagramPacket(tampon,
tampon.length);
    dgSoket.receive(paket);
```

```
    System.out.println("Paket alindi..");

    InetAddress istemciAdresi =
paket.getAddress();
    int istemciKapisi =
paket.getPort();

    Set<String> kur =
kurlar.keySet();
    Iterator<String> it =
kur.iterator();
```

```
    int i;
    for(i=0; i<tampon.length
&& it.hasNext(); ){

String s1 = it.next();
Integer in =
(Integer)kurlar.get(s1);

s1 = "<"+s1+">";
```

```
String s2 = "<"+in+">";
byte[] a = s1.getBytes();
byte[] b = s2.getBytes();

System.arraycopy(a,0,
tampon,i,a.length);
i = i + a.length;

System.arraycopy(b,0,
tampon,i,b.length);
i = i + b.length;
    }
```

```
if (i < tampon.length){
byte[] b =
new byte[i];

System.arraycopy(tampon,0,
b,0,i);
tampon = b;
    }
```

```
paket =
new DatagramPacket(tampon,
tampon.length,
istemciAdresi,
istemciKapisi);
dgSoket.send(paket);

System.out.println("Paket yollandi..");
}
```

```
paket =
new DatagramPacket(tampon,
tampon.length,
istemciAdresi,
istemciKapisi);
dgSoket.send(paket);

System.out.println("Paket yollandi..");
}
```

```
catch(IOException ioe){
    ioe.printStackTrace();
}
    }
}

public static void main(String[] args)
                    throws IOException {
    new DatagramSunucu().start();
}
}
```

```
import java.io.*;
import java.net.*;
import java.util.*;

public class DatagramIstemci {
public static void main(String[] args)
throws IOException {
    DatagramSocket soket =
new DatagramSocket();
```

```
    byte[] tampon = new b[256];

    InetAddress adres =
InetAddress.getLocalHost();
    DatagramPacket paket =
new DatagramPacket(tampon,
tampon.length,
adres, 6000);
    soket.send(paket);
```

```
 paket = new DatagramPacket(tampon,
tampon.length);
        soket.receive(paket);


 StringTokenizer strtok =
new StringTokenizer(
new String( paket.getData()),
"<> ");
```

```
Map<String, Integer> kurlar =
new HashMap<String,Integer>();

while(strtok.hasMoreTokens()){
String kur = strtok.nextToken();
if(strtok.hasMoreTokens()){
    String degeri =
strtok.nextToken();
    Integer deger =
Integer.decode(degeri);
                kurlar.put(kur,deger);
            }
    }
```

```
    Set<String> anahtarlar =
kurlar.keySet();
            Iterator<String> it =
anahtarlar.iterator();
            while(it.hasNext()){
              String s = it.next();
              System.out.println(s+" : "
+ kurlar.get(s) +" TL");
            }
    soket.close();
}
}
```

- Sunucumuzun ekran görüntüsü aşağıdaki gibi:

Sunucu calisti...
Paket alindi..
Paket yollandi..

- İstemcimizin de aşağıdaki gibi:

EUR : 1800000 TL
USD : 1500000 TL
YTL : 1000000 TL

- Datagram soketi kullanan uygulamamız için netstat çıktısına bakarsanız bu kez 6000 numaralı kapının, UDP'den dinlendiğini görebilirsiniz.

Uygulama Sözleşmesi Tasarımı

- İstemci sunucu modelinde yapılabilecek işlerin sınırı yoktur. Ancak istemci ile sunucu arasındaki iletişim yollanan paketlerde aktarılan bilgi ile sınırlıdır.
  - Bu nedenle paketlerin tasarlanması ve istemcinin sunucuya, sunucunun istemciye veriler aktarması gereklidir.
- Bu problem **uygulama sözleşmesinin tasarımı**dır.

Uygulama Sözleşmesi Tasarımı

- Bir uygulama sözleşmesi, istemci ve sunucu arasında gidip gelen paketleri tanımlar.
- Tipik bir pakette üç çeşit bilgi bulunabilir.
  - Kullanılan sözleşmeyi tanıtan bir kaç baytlık bir veri.
  - Paketi yollayanı ve becerilerini tanıtan bir kaç baytlık (kodlanmış) bir veri.
  - Paketin esas içeriği

Uygulama Sözleşmesi Tasarımı

- Paketin esas içeriği olarak nitelendirdiğimiz iletişim bilgisi üç biçimde olabilir.
  - Komut
  - Veri
  - Komut ve Veri
- Örneğin büyük bir veri birden fazla pakete bölünerek yollanacaksa komutlar (çok kabaca) aşağıdaki gibi tasarlanabilir.
  - AKTAR PAKETNO VERİ
  - DEVAM PAKETNO VERİ
- Bu konuda en iyi referans belli başlı sözleşmelerin tasarımlarıdır. Örneğin FTP, HTTP ve SMTP değerli örneklerdir.

- Java RMI (Remote Method Invocation) soketlerden bir üst seviyede sayabileceğimiz, ancak hala düşük seviyede bir iletişim biçimidir.
  - Saf Java ortamlarında yani hem istemcinin hem de sunucunun Java uygulaması olacağı durumlarda ve dağıtık mimarilerde kullanılır.
  - RMI'ın doğru anlaşılması için önce bazı temel kavramların görülmesi gerekir.

- 1980'lerde PC'lerin yaygınlık kazanması kurumların bilgi işlem mimarilerinde önemli değişiklikler yarattı.
  - Uzak konumlardaki şubeler kendi minik bilgi işlem merkezlerini kurmaya ve bazı işleri kendileri yapmaya başladı.
  - Şubeler ve merkez arasındaki eş zamanlılık istemci/sunucu modeli çerçevesinde geceleri yada belirlenmiş zamanlarda çalıştırılan görevler (task) ile sağlanıyordu.

Uzak Yordam Çağrısı

- Ancak bu modelin gerçek zamanlı çalışması gereken sistemlere uygun olmadığı açıktı.
  - Bankalar, sigorta şirketleri, önemli kamu kurumları, silahlı kuvvetler gibi organizasyonlar için daha farklı bir mimari gerekliydi.
  - Dağıtık mimari (distributed architecture) bu gereksinime yanıt verebilir mi?
- Dağıtık mimari, özetlersek, uygulamanın kullandığı bileşenlerin değişik konumlara dağıtıldığı bir yapıdır.

Uzak Yordam Çağrısı

- 1980'lerde dağıtık mimariyi gerçeklemek için aşağıdaki mekanizmalar kuruldu.
  - Bir şubedeki sistem kendi işlemlerini yaparken, arada bir bazı kritik işler için uzaktaki bir sunucuda bulunan bir işlevi (yordamı) sanki kendisinde kuruluymuş gibi kullanacaktır.
  - Ağ üzerinde iyi tanımlanmış bir iletişim sözleşmesi ile yordam çağrısı (procedure call), uzaktaki (remote) sunucuya yönlendirilecektir.
  - Uzak sunucu yordamı sanki kendisi içinden çağrılmış gibi işletecektir.

- Örneklersek:

```
// istemcide:
y = f (x, y);
// ağ  üzerinden f(),  x ve y sunucuya iletilir.
// sunucuda:
f(x,y);
// ağ üzerinden f()'nin dönüş değeri istemciye
   iletilir
// istemcide:
y = <değer> ;
```

Uzak Yordam Çağrısı

- Uzak yordam çağrısının olabilmesi için şunlar gerekir
  - İstemcinin sunucuya hangi yordamı çağırdığını anlatabilmesi ve aktarabilmesi
  - İstemcinin yordama geçilen parametreleri sunucya aktarabilmesi
  - Sunucunun kendisine gelen isteğe uygun yordamı seçebilmesi ve yordamı çağırabilmesi
  - Sunucunun yordamın dönüş değerini aktarabilmesi.

Uzak Yordam Çağrısı

- Uzak Yordam Çağrısı (RPC) ağ tabanlı bir teknik olduğu için genelde özel RPC paketlerinin tasarlanmasını gerektirir.
- Bu tasarım zordur
  - Teknik sorunlar (aktarılan verinin bağlantıları)
  - Verimlilik
  - Güvenlik
- Buna ek olarak iki farklı RPC uygulaması uyumlu olmak zorunda değildir.

Uzak Yordam Çağrısı

- Çeşitli sektörler yıllar içerisinde kendilerine özel RPC standartlarını oluşturdular.
- EDI (Electronic Data Interchange) aslında bir uzak yordam çağrısı mekanizmasıdır.

Uzak Yordam Çağrısı

- Java ortaya çıktığı zaman ağ üzerinden yürütülen tüm işleri yapmak için kuvvetli bir aday olmak zorundaydı.
  - Bu nedenle Java'da bir çeşit RPC desteği olması kaçınılmazdır.
  - Java soketler aracılığı ile eski RPC sistemlerine bağlanmak için altyapı sağlar.
- Buna ek olarak her iki uçta da Java uygulamaları olacaksa, kendi özel RPC mekanizması olan RMI'ı kullanabiliriz.

- RMI Java 1.1 ile gelmiş, her Java sürümü ile değişiklikler yapılmış ve sürekli gelişmiştir.
  - Java RMI (Remote Method Invocation – Uzak Yöntemin Çağrılması) mekanizması geleneksel RPC mekanizmasının geliştirilmiş bir biçimidir.
  - RPC mekanizması nesne tabanlı bir sistemde zorluklar çekebilir. (Referanslar, çok biçimlilik, vs)
  - Her iki taraftaki sistemlerde aynı sınıfın nasıl kullanılacağı belirsizdir.

- RMI, J2EE'de önemli bir yere sahiptir. EJB teknolojisinin arkasında yer alır.
  - Ancak uygulama geliştiriciler için EJB kullanırken RMI'a dair çok fazla bir bilgi gerekmez.
  - RMI detayları arka planda çözülür.
  - Bu nedenle RMI, J2EE kapsamında çok işlenmeyen, daha çok ağ programcılığı başlığında işlenen bir konudur.

- RMI istemcileri RMI sunucuları ile 5 katmanlı bir iletişim protolü üzerinden haberleşir.
  - RMI arabirimi
  - RMI iskelesi (stub)
  - Uzak referans katmanı
  - RMI iletim katmanı (transport layer)
  - Internet Protokolü (IP)
- İstemci, sunucu tarafından dışarıya sunulan hizmetleri (nesneleri) kendi sanal makinesinde bulunuyormuş gibi kullanır.

- RMI sunucusu tarafından bakarsak, sistemde 4 katmanlı bir iletişim protolü bulunur:
  - RMI iskeleti (skeleton)
  - Uzak referans katmanı
  - RMI iletim katmanı (transport layer)
  - Internet Protokolü (IP)
- Sunucu kendisi üzerinden yöntem çağrıldığı zaman yönteme geçilen nesne parametrelerini kullanmalıdır.

- RMI istemcisi bir yöntem çağırdığı zaman
  - Çağrı bir RMI arabirimi üzerinden yapılır
  - Arabirimi uygulayan iskele (stub) bu çağrıyı işler. İskelenin yaptığı, serileştirme kullanarak bu çağrıyı bir ağ paketine eklenecek hale getirmektir.
  - Uzak referans katmanı, iskeleden aldığı serileştirilmiş veriyi iletmek için iletim protokolünün detayları ile uğraşır. Örneğin uzak sunucunun adresinin belirlenmesi, ağ bağlantısının açılması bunlar arasındadır.
  - İletim katmanı RMI iletim protokolleri (JRMP yada IIOP) ile ilgili işlemler halleder.

- Normalde RMI sunucusuna olan bağlantının doğrudan IP protokolü üzerinden yapılması beklenir.
  - Ancak artık HTTP tabanlı (HTTP->IP) bağlantılar da ayarlanabilmektedir. Farklı kurumların sistemleri arasındaki bağlantı çok sayıda güvenlik duvarından geçeceği için bu teknik daha kullanışlı olmaktadır.

- RMI mimarisi ile ilgili java paketleri J2SE parçası olarak gelmektedir. Bunların bir kısmı şunlardır:
  - java.rmi paketi genel RMI bileşenlerini ve RMI arabirimini içerir.
  - java.rmi.server paketi sunucu arabirimlerini ve sınıflarını içerir
  - java.rmi.registry paketi RMI sunucu arama hizmeti için gerekli arabirim ve sınıfları içerir
  - javax.rmi.CORBA RMI'ın iletim protokolü olarak CORBA IIOP kullanmasını sağlayan sınıfları içerir.

- RMI uygulamaları geliştirirken java, javac gibi standart araçlara ek olarak bazı başka araçlar da kullanırız. Bunların başlıcaları şunlardır:
  - RMI derleyicisi (rmic) sunucuda bulunan derlenmiş (.class dosyası bulunan) bir sınıfın RMI mekanizmasını oluşturmak için kullanılır.
  - RMI kaydı (rmiregistry) RMI sunucularında bulunan nesnelerin kayıtlarını saklayan bir süreci çalıştırır.
  - HTTP sunucusu Java sınıflarının dosyalarını sunacaktır. Minimal bir sunucu yeterlidir.

- Bu araçlara ek olarak, daha ileri düzey uygulamalar için aşağıdaki araçlar kullanılır
  - IDL Java derleyicisi (idlj) verien bir IDL tanımından uygun Java iskele ve iskeletlerini inşa eder.
  - IIOP Geçici Adlandırma Servisi (tnameserv) CORBA istemcilerinin RMI ile sunulan nesnelere ulaşmasını sağlar. Servis sunucu ömrü ile sınırlıdır.
  - IIOP Kalıcı Adlandırma Servisi (ordb) ise geçici olmayan bir adlandırma servisi sunar.
  - RMI Etkinleştirme Demon'ı (rmid), istek üzerine etkinleştirme desteği sağlar.

- RMI sisteminin çalışması için aşağıdaki temel altyapı ayarlarının yapılması gereklidir.
  - Dağıtık sistemde karşı tarafta çalışması gereken her türlü kodun indirilmesi için bir HTTP sunucusu kurulmalıdır. Bu sunucu yalnızca GET becerisi olan minimalist bir sunucu olabilir.
  - RMI kaydı ayarlanmalı ve başlatılmalıdır.
- CORBA desteği için ek olarak aşağıdakiler başlatılmalıdır
  - RMI adlandırma servisleri
  - RMI etkinleştirme demon'ı

- RMI geliştirme süreci aşağıdaki adımlardan oluşur
  - Uzak arabirim tanımlanır
  - RMI sunucusu geliştirilir.
  - RMI iskeletleri (sunucu için) ve iskeleleri (stub) yaratılır.
  - Gerekli görülürse RMI sunucusu kaydı için gerekli işlemler yapılır.
  - RMI istemcisi geliştirilir.

- Java uzak yöntem protokolü (Java Remote Method Protocol), TCP/IP üzerinden işleyen oldukça basit bir iletim protokolüdür.
- Aşağıdaki paket biçimini kullanır
  - Paketlerin başında 4 bayt ASCII kodu olarak JRMI
  - Ardından 2 bayt protokol sürümü
  - Ardından 1 bayt tanımlayıcı
  - Ardından veriler

JRMP ve Corba/IIOP

- JRMP'de altı değişik mesaj türü vardır.
  - CALL VERİLER
  - PING
  - DGC TANIMLAYICI
  - RETURNDATA VERİLER
  - HTTPRETURN
  - PINGACK
- JRMP tek bir soket bağlantısı üzerinden çok sayıda (65K) istemcinin bağlanmasına izin verir.

JRMP ve Corba/IIOP

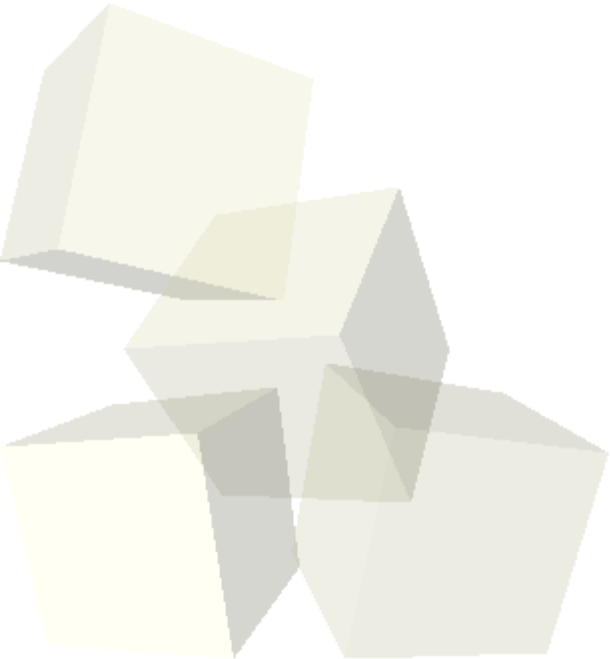
- IIOP (İnternet inter-ORB Protocol), CORBA standartı ile birlikte gelen genel amaçlı bir sözleşmedir.
  - CORBA'nın GIOP sistemini TCP/IP üzerinden işletir.
  - Daha genel amaçlı yazıldığı için çok daha karmaşık bir paket formatı vardır.
  - Java dışı sistemlerde de kullanılır.
- RMI sunucusu yada istemcisi Java dışı dağıtık sistemlere bağlanacaksa JRMP yerine IIOP kullanılmalıdır.

- RMI iletişiminde IIOP kullanılması sayesinde RMI mesajları CORBA'nın kullandığı IDL mesajlarına çevrilebilir.
  - Buna ek olarak RMI derleyicisi de IDL dönüşümü yapabilir.
  - Burada önemli detay standart Java sınıfları ve veri tipleri ile doğru IDL tipleri arasında dönüşüm yapmaktır.
  - CORBA 2.3 standartı bu konuda gerekli bilgiyi sağlamaktadır.

- Şimdi çok küçük bir RMI uygulaması yazacağız.
  - İki diziyi parametre olarak alan ve iki sayıdan oluşan basit bir sınıfın nesnesini parametre olarak dönen bir yöntemi çağırmayı amaçlıyoruz.
  - Yöntemin içi boş sayılır ancak normalde bu iki sayı dizisini x ve y koordinatları olarak görüp, bunlardan yada yakınlarından geçen doğruyu hesaplamasını bekleyebilirsiniz.

```
import java.rmi.*;

public interface Dogru extends Remote {
public Object dogruHesapla (double[] x,
   double[] y)
throws RemoteException;
}
```

```
import java.rmi.*;
import java.rmi.server.*;

public class DogruHesaplar
extends UnicastRemoteObject
implements Dogru {
```



```
protected static ab enson = null;
protected static final String isim =
"//localhost/DogruHesaplar";

DogruHesaplar()
throws RemoteException{
    super();
}
```

```
public Object dogruHesapla(double[] x,
   double[] y)
throws RemoteException {

    double a=0.5,b=0; // y = 0.5x
    // normalde hesaplama yapilacak...

    enson = new ab(a,b);
    return enson ;
}
```

```
public static void main(String args[]){

    if(System.getSecurityManager() == null){
System.setSecurityManager(
    new RMISecurityManager()
);

// yukarıdaki işlem çok önemli
```

```
// main'den devam
try{
    DogruHesaplar dh =
new DogruHesaplar();
// RemoteException
    System.out.println("Sunucu yaratildi...");

    Naming.rebind(isim,dh);
    // java.net.MalformedURLException
    // java.security.AccessControlException
```

```
// main'den devam
    System.out.println("Sunucu baglandi...");
}
catch(RemoteException re){
    re.printStackTrace();
}
catch(java.net.MalformedURLException mue)
   {
    mue.printStackTrace();
}
```

```
// main'den devam
catch(java.security.AccessControlException
   ace){
    ace.printStackTrace();
}
   }
}
}
```

- Burada yöntemin dönüş tipi olarak ab adında küçük bir sınıf kullanıyoruz.

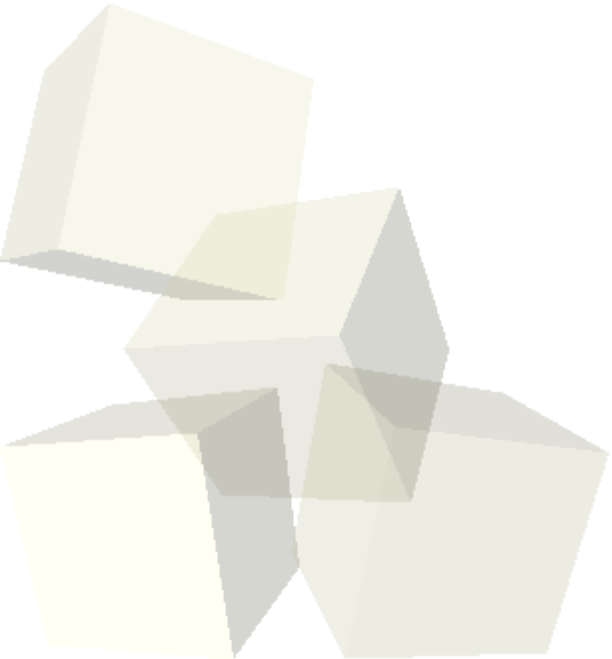
```
public class ab  implements Serializable{
// y = ax+b
public double a;
public double b;
ab(double a, double b){
    this.a=a;
    this.b=b;
}
}
```

```
import java.rmi.*;

public class RMIIstemci {
    public static void main(String args[]) {

if (System.getSecurityManager() == null) {
    System.setSecurityManager(
new RMISecurityManager()
);
```

```
}
try {
    String isim =
"//localhost/Dogru";
    Dogru dh =
(Dogru)Naming.lookup(isim);
double x[] = {0.0, 1.0, 2.0};
double y[] = {0.0, 0.45, 1.05};
// y = 0.5x + hata cikmali
```

```
ab denklem =
(ab)dh.dogruHesapla(x,y);

System.out.println("y= "
+denklem.a+"x + "
+denklem.b);

}
```

```
catch(Exception e){
    e.printStackTrace();
}
    }
}
```



- Şimdi uzak arabirim ve gerekli Java sınıfları için JAR dosyaları yaratacağız.

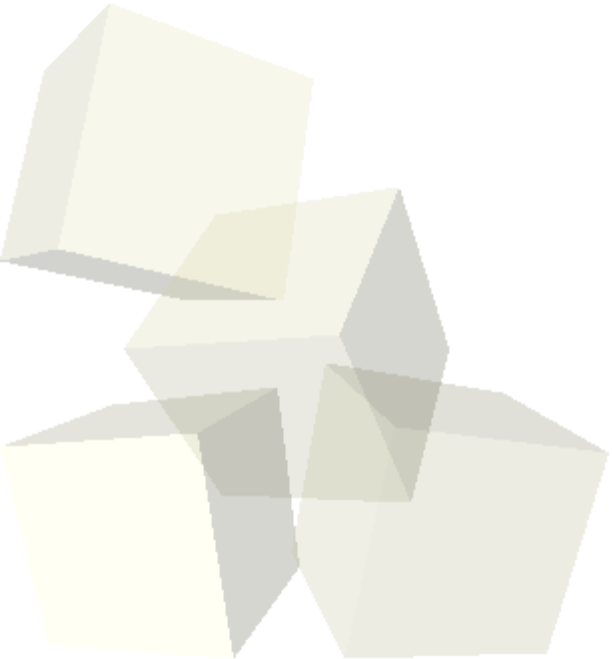
```
$ jar cvf dogru.jar Dogru.class
added manifest
adding: Dogru.class(in = 217) (out= 163)
   (deflated 24%)
```

- Şimdi uzak arabirim ve gerekli Java sınıfları için JAR dosyaları yaratacağız.

```
$ jar cvf dogru.jar Dogru.class
added manifest
adding: Dogru.class(in = 217) (out= 163)
   (deflated 24%)
```

```
$ jar cvf hesapla.jar DogruHesaplar.class
   ab.class
added manifest
adding: DogruHesaplar.class(in = 1757) (out=
   977)(deflated 44%)
adding: ab.class(in = 325) (out= 242)(deflated
   25%)
```

```
$ jar cvf istemci.jar RMIIstemci.class ab.class
added manifest
adding: RMIIstemci.class(in = 1503) (out=
   876)(deflated 41%)
adding: ab.class(in = 325) (out= 242)(deflated
   25%)
```



- Bu noktadan sonra JAR dosyalarını sistemdeki bilgisayarlara dağıtabiliriz. Dağıtık da olsa işlemler yapılabilecektir.
- Uzak nesneler için iskele yaratmamız gereklidir.
  - Şimdilik 1.2 sürümüne geri uyumlu iskele yaratalım

```
$ rmic -d . -v1.2 DogruHesaplar
```

- Bu komutla birlikte **DogruHesaplar_Stub.class** adında bir dosya oluşması gerekli.
  - RMI derleyicisine -v1.2 parametresi ile Java 1.2 uyumlu bir iskelet yaratmasını belirttik.
  - Java sürümleri ilerlerken 1.1, 1.2, 1.3 ve 5.0 sürümlerinde RMI iskeletleri konusunda ufak değişiklikler olmuştur.
  - İşi garantiye almak için biz 1.2 sürümüne kadar geri uyum sağladık. Ancak pratikte bunu yapmak durumunda değiliz.

- Şimdi uzak arabirimin JAR dosyasını, web sunucumuzun (örneğin Apache) sunduğu dosyaların olduğu yere (/www yada c:\www olabilir) taşıyalım.
- Ardından web sunucumuzu başlatmamız gerekli.
- Böylece RMI sisteminin kullanacağı sistemleri kurmuş olduk.

- Artık RMI kaydını başlatabiliriz:

$ unsetenv CLASSPATH
$ rmiregistry

  - UNIX'ler ve Linux gibi kabuk sağlayan işletim sistemlerinde **rmiregistry** komutu sonrasında & koyarak arka planda çalıştırmak isteyebilirsiniz.
  - Windows'larda **unsetenv** yerine **unset** kullanmalısınız ve ayrıca programın adı **rmiregistry.exe** olacaktır.

- Ardından Web üzerinden erişimleri düzenlemek için bir java.policy dosyası oluşturacağız. Bu dosya sunucumuzun kullanacağı güvenlik politikasını içeriyor.

```
grant {
    permission java.net.SocketPermission
   "*:1024-65535",
        "connect,accept";
    permission java.net.SocketPermission
   "*:80", "connect";
};
```

- Dosyayı şimdilik web sunucumuzun dizininde tutalım. Aslında bunun kamuya açık olmayan bir yerde durması gerekli. Bunu da yaptıktan sonra sunucumuzu başlatabiliriz.

```
$ java -Djava.rmi.server.codebase =
   "http://localhost/"
   -Djava.rmi.server.hostname = localhost
   -Djava.security.policy =
   "http://localhost/java.policy"
DogruHesaplar
```

- Ardından istemciyi başlatacağız:

```
$ java
   -Djava.rmi.server.codebase="http://localhos
   t/"
-
   Djava.security.policy="http://localhost/java.p
   olicy"
RMIIstemci localhost 1099
y= 0.5x + 0
```

- Elbette bu işlemler sırasında çok sayıda hata almamız mümkün. Bunları genel olarak beş başlıkta sayabiliriz.
  - CLASSPATH sorunları
  - Java güvenlik politikası sorunları
  - Sanal Makine sorunları
  - Genel ağ sorunları
  - Genel dosya erişim hakkı sorunları

Java 5.0 RMI Değişiklikleri

- Java 5.0'da RMI kullanımı ciddi oranda kolaylaşmıştır.
  - Yalnızca Java 5.0 sistemler ile kullanacaksanız, RMI iskeleleri (stub) yaratılması (rmic kullanımı) gerekli değildir. Doğrudan doğruya yereldeki .class ve .jar dosyaları ile çalışılabilir.
  - Solaris ve Linux işletim sistemlerinde RMI sunucularının inetd/xinetd üzerinden başlatılması için ek sınıflar getirilmiştir.

Java 5.0 RMI Değişiklikleri

- RMI / IIOP bağlantılarında da güncellemeler olmuştur.
  - J2SE ORB'u artık 1.4 kayıt tutma (logging) özelliklerini kullanabilecektir.
  - J2SE ORB'u yük dengeleme, dinamik köprüleme gibi özellikler kazanmıştır.

Java RMI Güvenliği

- RMI güvenliğinde öncelikli adım bir güvenlik yöneticisi kurmak ve politika dosyası belirlemektir.
  - Bunu örnek uygulamamızda yaptık.
  - Yazdığımız java.policy dosyası da oldukça detaylı olmalıydı.
- RMI diğer servislerle ortak çalışıyorsa bu servislerle olan iletişiminde JCA yada JCE API'leri kullanılabilir.
- http://servlet.java.sun.com/javaone/javaone2000/pdfs/TS-1002.pdf

Teşekkürler