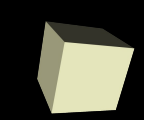
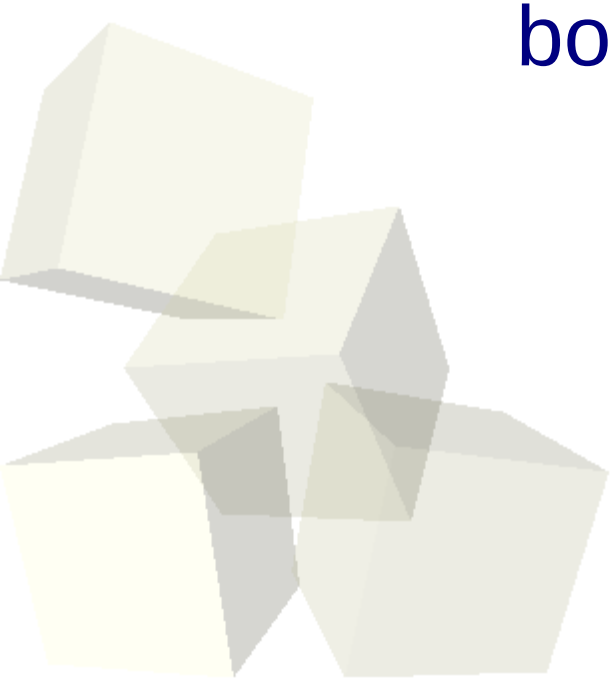


Ağ Sunucusu Programlaması

Bora Güngören
Portakal Teknoloji
bora@portakalteknoloji.com

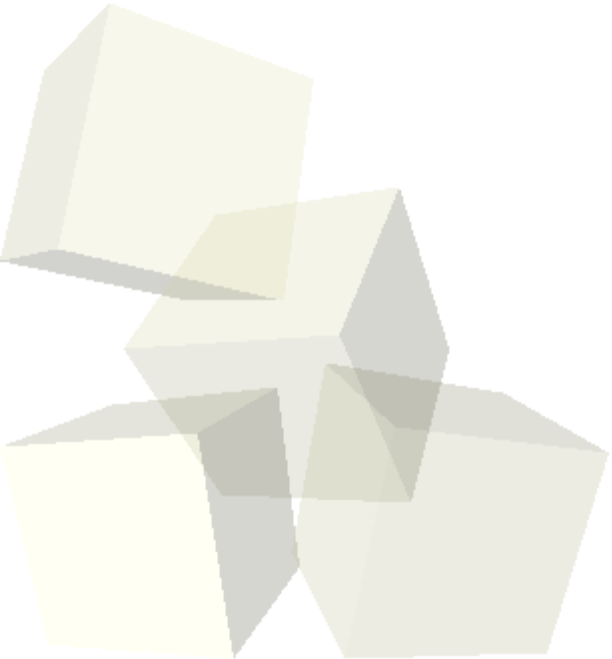


Genel Sunum Bilgisi ve Lisans

- Sunum 16 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

- Çok kanallı mimarinin gerekleri
- Java'da kanallar
- Komut (Command) Tasarım Biçimi
- İleri kanal teknikleri
- Kanal grupları
- Çok kanallı bir ağ sunucusu



Çok Kanallı Mimarinin Gerekleri

- Bir uygulamanın aynı anda birden fazla işi yapması aslında mümkün değildir.
  - Ancak bilgisayarın işletim sistemi sistem kaynaklarını, özellikle de işlemci gücünü, değişik uygulamalar arasında nöbetleşe paylaştırabilir.
  - Bu durumda kullanıcı için aynı anda birden fazla uygulama çalışıyor gibi gözükür.
- **Çok kanallı** (multi-threaded) bir uygulama, aynı anda birden fazla küçük uygulama başlatarak bunları yöneten bir uygulamadır.
  - Çok kanallı uygulamaları onlarca yıldır kullanıyoruz.

Çok Kanallı Mimarinin Gerekleri

- Tipik bir çok kanallı uygulamanın karşılaştığı sorunlar nelerdir?
- Tüm sorunlar bazı ortak kaynaklara aynı anda ulaşmaya çalışan birden fazla kanal olmasından doğar.
  - Aynı anda bir veri tabanına yazan ve bir veritabanından okuyan iki kanal (küçük uygulama) bulunduğunu düşünün. Veritabanından okuyan uygulama işinin ortasındayken işletim sistemi tarafından kenara alınsa ve yazan uygulama başlatılsa ne olur?
  - Yazıcıya veri yollayan iki uygulamadan birisi ötekinin işini bitirmesini beklemeden kendi verilerini yollamaya başlarsa ne olur?
  - Bir uygulama diğeri için gereken dosyayı silerse ne olur?

Çok Kanallı Mimarinin Gerekleri

- Çok kanallı uygulamaların karşılaştığı bu problemlere **eşzamanlılık** (synchronization) problemleri adını veriyoruz.
  - Tüm eşzamanlılık problemleri temelde benzerdir. Ancak erişilen kaynakların özellikleri nedeni ile değişik şekillerde çözülürler.
- Bunun dışında kanallarla çalışırken işletim sisteminin sağladığı modelin kendine has yapısından kaynaklanan problemler de yaşanabilir.
  - Windows NT ile gelen ve bugün de kullanılan dinamik öncelik sistemi buna örnektir. (Detayları daha sonra)

Çok Kanallı Mimarinin Gerekleri

- Herhangi bir programlama dilinde çok kanallı uygulamalar (kabaca) aşağıdaki biçimde tasarlanır ve geliştirilir.
  - Her işi yapacak olan kanallar belirlenir.
  - Her kanalık erişeceği kaynaklar belli olduktan sonra eşzamanlılık sorunları kestirilir. Bu sorunlar için eldeki tekniklerle önlem alınır.
  - Sistemdeki kanalları yaratacak ve çalıştıracak ayrı bir uygulama yazılır. Bu uygulamanın kendisi kısa ömürlü olabilir yada sürekli olarak diğer kanalları yönetmek için çalışmaya devam edebilir.
  - Sistem test edilince önceden farkedilmeyen eşzamanlılık sorunları çıkar. Bu sorunlar giderilir.
  - Sorunsuz olan yazılım sistemi yüklenme testine girer. Yeni problemler bulunur ve giderilir.

Java'da Kanallar

- Java dili 1996 yılında tasarlandığı zaman en baştan çok kanallı olarak tasarlanmıştır.
  - En ilkel **main(String args[])** uygulamaları haricinde tüm Java uygulamaları kendiliğinden çok kanallıdır.
  - Ancak bu uygulamaların çoğunda eşzamanlılık sorunları sınıf kitaplıkları içinde çözülmüştür. Bu nedenle programcı çok kanallı sistemin farkına varmadan işini yapar.
    - Bu yaklaşım nedeni ile zaman zaman **çok kanallı sistemde güvenli** (thread safe) olmayan bazı sınıf kitaplıklarını sanki güvenliymişçesine kullanıyoruz. Bu en sık rastlanan kanal hatalarından birisidir.

Java'da Kanallar

- Java 1.4 kanal mimarisi POSIX standartında belirtilen kanal mimarisine oldukça yakındır. Ancak her işletim sisteminin bu teknikleri desteklemeyeceği varsayımı ile sadeleştirilmiştir.
  - Daha önceden POSIX kanalları (pthread) ile çalışmış birisi için Java kanallarını kullanmak çok kolaydır.
- Java sanal makinası kendi üzerindeki her kanalı işletim sistemindeki bir kanala (yada sürece) eşlemeye çalışır.
- Görsel bileşenlerin olduğu uygulamalarda ise durum biraz farklıdır.
  - Kanal özelliği olan görsel bileşenleri yönetecek ayrı bir kanal bulunmaktadır. Ancak uygulamalar yönetimi bu kanala bırakmadan da yazılabilir.

Java'da Kanallar

- Java kanal mimarisinde tüm kanallar **Runnable** (çalışabilir) adındaki bir arabirimi uygulamaktadır.
  - Runnable arabiriminde tanımlanan tek yöntem, kanal çalıştırıldığında yapılacak olan işlerin yazıldığı **run()** yani **çalış** yöntemidir
- Bu arabirimi uygulayan ve kanallar için ciddi avantajlar sağlayan bir sınıf ise **Thread** sınıfıdır.
  - **Thread** sınıfı kendi içerisinde bir **Runnable** nesnesi (yani bu arabirimi uygulayan herhangi bir sınıfın nesnesi) barındıracak şekilde tasarlanmıştır.
  - Bu nedenle eğer **Runnable** arabirimini uygulayan herhangi bir sınıfı yazarsak, o sınıfın nesnelerini birer **Thread** nesnesi içine saklayarak çalıştırabiliriz.

```
public
interface Runnable {
    public abstract void run();
}

public
class Thread implements Runnable {
    //...
}
```

- Bir **Thread** nesnesi yaratılırken (yapıcı çağrısı olurken) **init()** yöntemi kullanılarak içindeki veriler yapılandırılır.

private void init(**ThreadGroup** g, **Runnable** target, **String** name, **long** stackSize)

- Buradaki parametreler
  - **g**, kanalın ait olduğu kanal grubuna referans
  - **target**, kanalın çalıştıracağı **Runnable** nesnesi
  - **name**, kanalın adı
  - **stackSize**, kanalın kullanacağı yığının maksimum boyutu (bayt olarak). Ancak sıfır girilirse bir sınıf olmaz.

- Şimdi bu sınıfın en çok kullanılan dört yapıcısını tanıyalım:
  - Varsayılan yapıcı kanalı herhangi bir gruba dahil etmeden, çalıştıracağı bir **Runnable** olmadan, otomatik yaratılan bir isim ile ve sınırsız yığın boyutu ile yaratır.

```
public Thread() {
init(null, null, "Thread-" + nextThreadNum(), 0);
    }
```

  - Tek parametre olarak **Runnable** referansı alan yapıcı en yaygın kullanılan yapıcıdır. Bu yapıcı ile kanala kendisinin çalıştıracağı bir nesne veririz.

```
public Thread(Runnable target) {
init(null, target, "Thread-" + nextThreadNum(), 0);
    }
```

- Yapıcılara devam edelim
  - **Runnable** ve **String** parametreleri alan yapıcı, kanalın hedefini ve adını belirler.

```
public Thread(Runnable target, String name) {
init(null, target, name, 0);
    }
```

  - Kanal grubu, hedef ve isim alan yapıcı ise kanalın bu üç parametresini ayarlar.

```
public Thread(ThreadGroup group, Runnable target,
   String name) {
init(group, target, name, 0);
    }
```

Java'da Kanallar

- Kanallarla çalışırken ilk öğrenilen yöntem **currentThread()** yöntemidir.

public static native Thread currentThread();

- Bu yöntem sınıfın durağan bir yöntemi olarak **o sırada çalışmakta olan kanala bir referans** döner.
  - Dikkat ederseniz bu yöntemi çağırdığımız zaman çalışan kanal içinde bulunduğumuz kanal olacaktır. Bu durumda **kendi kanalımıza referans** elde etmiş ouruz.
  - Bu yöntemin döndüğü değeri **this** yada **super** referanslarını kullandığımız gibi kullanmamız gerekir.

Java'da Kanallar

- Kanallarla çalışırken ilk öğrenilen yöntem **currentThread()** yöntemidir.

public static native Thread currentThread();

- Bu yöntem sınıfın durağan bir yöntemi olarak **o sırada çalışmakta olan kanala bir referans** döner.
  - Dikkat ederseniz bu yöntemi çağırdığımız zaman çalışan kanal içinde bulunduğumuz kanal olacaktır. Bu durumda **kendi kanalımıza referans** elde etmiş ouruz.
  - Bu yöntemin döndüğü değeri **this** yada **super** referanslarını kullandığımız gibi kullanmamız gerekir.

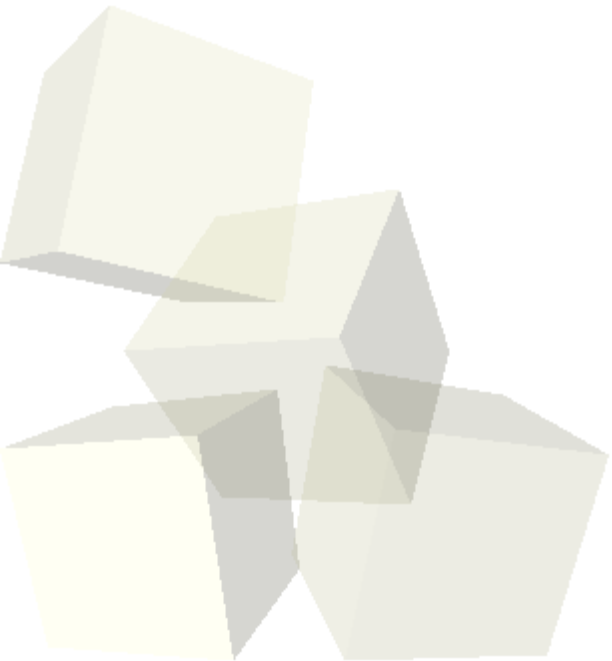
- İlk kanal uygulamamız:

```
class Kanal1 {
    static java.lang.Runnable r;
    static java.lang.Thread t;

    public static void main(String args[]){

            r = t = Thread.currentThread();

          System.out.println(t.toString());
    }
}
```

- Bu uygulamanın tipik çıktısı aşağıdaki gibidir:

Thread[main,5,main]

- Burada elde edilen bilgiler sırası ile
  - Kanalın sınıfı
  - Kanalın adı
  - Önceliği
  - Kanal grubunun adı



- Burada **toString()** yönteminin kodu açıklayıcı olacaktır.

```
public String toString() {
        ThreadGroup group = getThreadGroup();
         if (group != null) {
                return "Thread[" + getName() + "," +
                                getPriority() + ","
                                + group.getName() + "]";
         } else {
                return "Thread[" + getName() + ","
                        + getPriority() + "," + "" + "]";
         }
}
```

- Kendi kanal sınıfımızı yazarken aşağıdakileri yazmamız gerekir.
  - Yapıcı(lar)
  - **run()** yöntemi
- Aslında eğer sınıfımız içinde veri barındırmıyorsa yapıcı yazılmasına gerek olmaz ancak genelde kanal sınıflarının eriştikleri belli kaynaklar olacaktır ve bu kaynaklara referanslara sahip olmaları gerekir.

```
class Kanalim1 extends Thread {

    private static int sayac = 0;

    private static int sayac() {
        return Kanalim1.sayac++;
    }

    Kanalim1(){
      super("Kanalim1-" + Kanalim1.sayac());
    }
```

```
    public void run(){
            System.out.println("Su anda "
                            + this.getName()
                            + " calisiyor");
    }

}
```

- Kanallar her ne kadar çalıştırıldıklarında **run()** yöntemini kullansalar da, çalıştırılmaları için işletim sisteminin kanal kuyruğuna girmeleri gerekir.
  - Bu kuyruğa girmeleri için önce onların **start()** yöntemi çalıştırılır.
  - Kuyruğa giren kanal, bir diğer kanal işini bitirene kadar bekleyecektir.

```
class Kanal2 {

    public static void main(String args[]){
        Thread t1 = new Thread(
                            new Kanalim1()
                                ),
                    t2 = new Kanalim1();
        System.out.println("Baslatiyorum.");
        t1.start();
        t2.start();
        System.out.println("Baslattim.");
    }
}
```

- Bu uygulamanın çıktısı aşağıdaki gibi:

```
Baslatiyorum.
Baslattim.
Su anda Kanalim1-0 calisiyor
Su anda Kanalim1-1 calisiyor
```

- Görüldüğü gibi kanalları yaratan esas kanalın işi bitmeden diğerleri başlayamadı.
- Bu durumda, pratik uygulamada her kanalın **gönüllü olarak çalışmayı bırakması** ve diğer kanallara fırsat vermesi gereklidir.

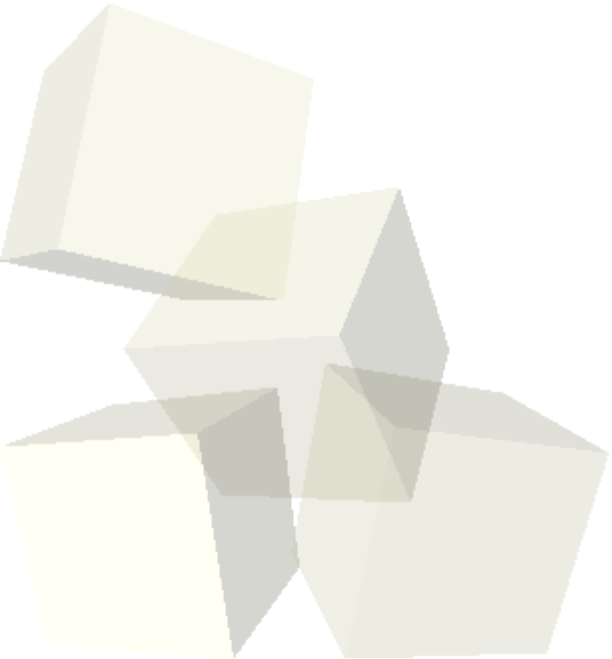
- Bir kanalın gönüllü olarak çalışmaya ara vermesi genellikle **uykuya yatması** (sleep) mekanizması ile olur.

public static native void sleep(**long** millis) **throws InterruptedException**;

- Burada hata durumu oluşabileceğine dikkat edin. Eğer bu kanal uykuya yatmak isterken yada uykudayken durumu bir diğer kanal tarafından bölünürse bu hata durumu oluşur.
  - Bu nedenle bir kanalın uykuya yattığı kodlar try-catch bloğunda olmak zorundadır.

- Şimdi uyuma/uyanma mekanizmasını kullanan bir örnek yapalım.
  - İki kanalımız olacak.
  - Birincisi dosyaya yazacak ve uykuya yatacak.
  - İkincisi dosyadan okuyacak ve uykuya yatacak.

```
import java.io.*;

class Kanalim2 extends Thread {

private FileInputStream fis;
private boolean hazir;
```



```
import java.io.*;

class Kanalim2 extends Thread {

private FileInputStream fis;
private boolean hazir;

Kanalim2 (String dosya){
hazir = false;
```

```
try {
fis = new FileInputStream(dosya);
hazir = true;
}
catch(FileNotFoundException fnfe){
System.err.println("Dosya bulunamadi.");
}
catch(IOException ioe){
ioe.printStackTrace(System.err);
}
}
```

```
public void run(){

byte baytlar[] = new byte[6];
int offset = 0;
int n = 5;
while(true){
int okundu = 0;
try{
if (hazir == true)
okundu =
fis.read(baytlar);
}
```

```
catch(IOException ioe){
ioe.printStackTrace(System.err);
okundu = 0;
}

if (okundu == 0){
System.out.println("Okunacak yazi kalmadi.");
break;
}
```

```
String s = null;
try {
s = new String (baytlar,"ISO-8859-9");
}
catch(UnsupportedEncodingException uee){
uee.printStackTrace(System.err);
s = null;
}

System.out.println(s+" okundu.");
```

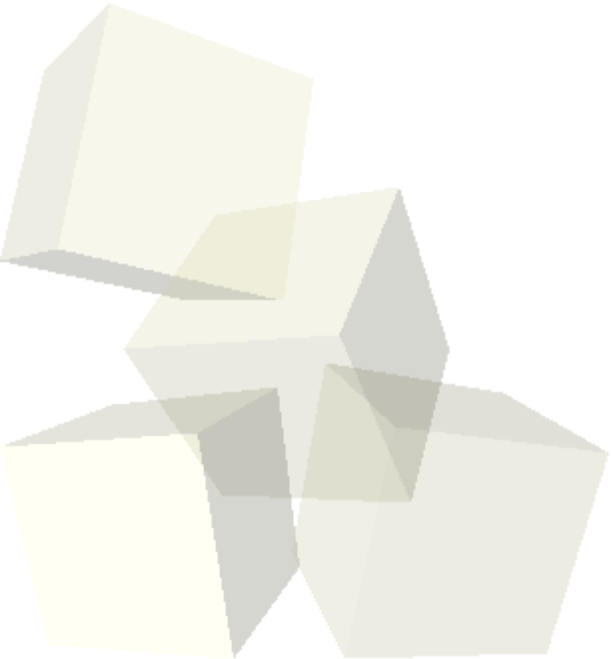
```
int saniye = 2;
try{
Thread.currentThread().sleep(saniye * 1000);
}
catch(InterruptedException ie){
ie.printStackTrace(System.err);
}


}
```

```
try{
fis.close();
}
catch(IOException ioe){
ioe.printStackTrace(System.err);
}

}

}
```

```
class Kanalim3 extends Thread {

private FileOutputStream fos;
private  boolean hazir;

Kanalim3(String dosya){
hazir = false;
try{
boolean ekle = true;
fos = new FileOutputStream(dosya,ekle);
hazir = true;
}
```

```
catch (FileNotFoundException fnfe){
fnfe.printStackTrace(System.err);

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

```
public void run(){

String s = null;
int n = 5;
while (n > 0){
s = "Dene-" + n;
n--;
```



```
try {
if (hazir == true){
byte[] baytlar = s.getBytes();
fos.write(baytlar);
System.out.println(
"Dosyaya " + s+ " yazdim.");
}
}
```

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

try {
int saniye = 2;
Thread.currentThread().sleep(saniye * 1000);
}
catch(InterruptedException ie){
ie.printStackTrace(System.err);
}
}
```

```
try {
fos.close();
}
catch(IOException ioe){
ioe.printStackTrace(System.err);
}

}
}
```

```
class Kanal3 {

public static void main(String args[]){

java.util.Random r =
new java.util.Random(
System.currentTimeMillis()
);

int karakter = Math.abs (r.nextInt())%8 + 1;
```

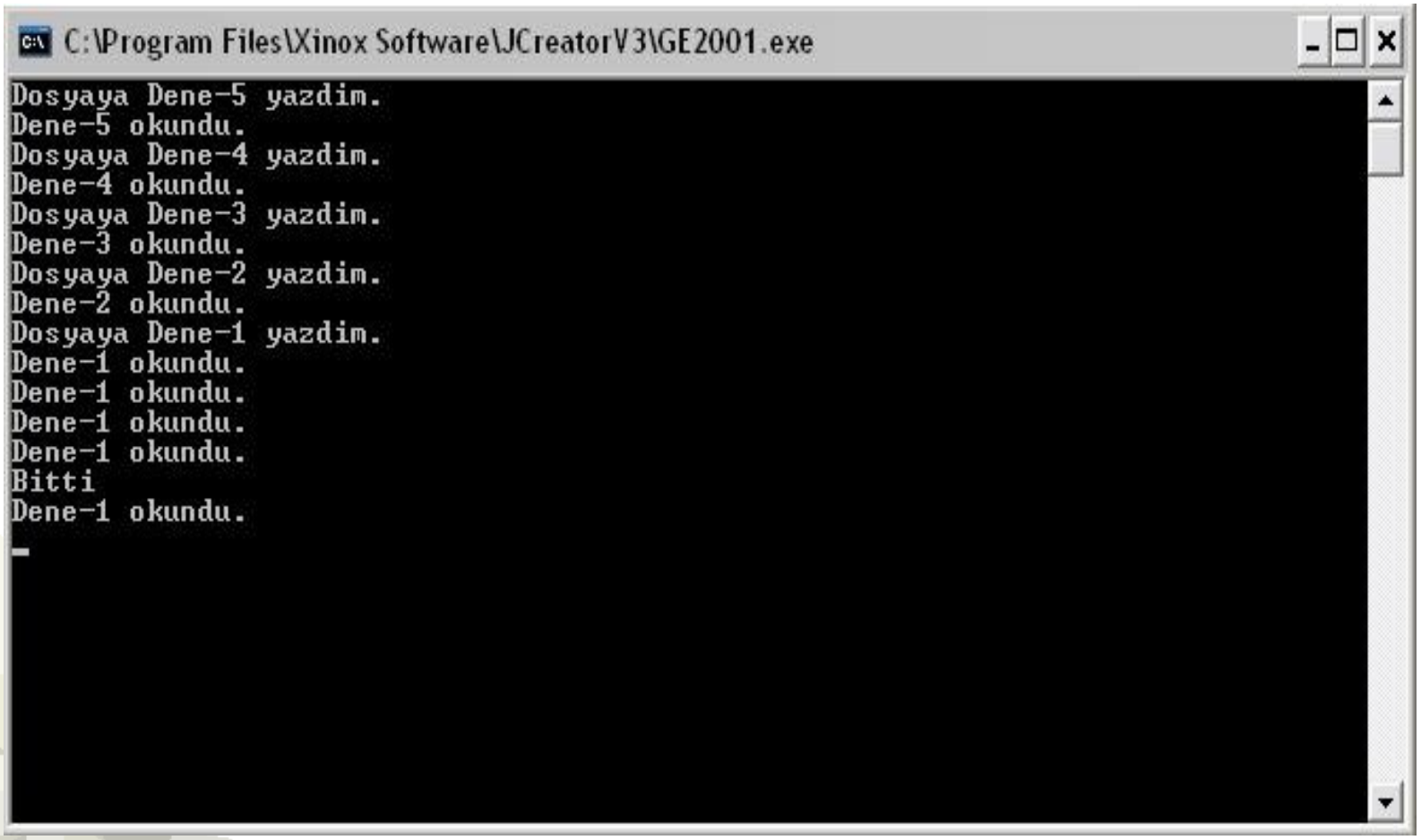
```
// rastgele dosya adı yaratalım

String s = "";
while(karakter-- > 0){
s = s + (char)( 'a' + Math.abs (r.nextInt())% 26 );
}
s = s + ".txt";
```

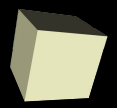
```
File f = new File (s);
boolean yaratildi = false;
if (f.exists() == false){
try {
yaratildi = f.createNewFile();
}
catch(IOException ioe){
ioe.printStackTrace(System.err);
}
}
```

```
if (yaratildi == false)
return;

String ad = f.getPath();

Thread t1 = new Kanalim3(ad); // yazici
Thread t2 = new Kanalim2(ad); // okuyucu
```

```
t1.start();
t2.start();

int saniye = 15;
try {
Thread.currentThread().sleep(saniye * 1000);
}
catch(InterruptedException ie){
ie.printStackTrace(System.err);
}
System.out.println("Bitti");
}
}
```

- Bu uygulamayı çalıştırdığımızda ne oluyor?





Java'da Kanallar

- Bir problem var. Bu okuyucu kanalda oluyor.
  - Dosyanın son satırını defalarca okuyoruz.
  - Okuyucu kanalın dosyaya her zaman yazılacağını varsaymak yerine daha başka bir biçimde yazılması gerekli.
- Belki de yazıcı kanal, okuyucu kanala kendisinin bir şey yazdığını belirtmeli.
  - Bu durumda iki kanal uygulama uzayında bir değişkeni paylaşabilir.
  - Yazıcı yazdıkça bu değişkene müdahale eder. Örneğin artırır.
  - Okuyucu uyandığında değişkene bakar, pozitif ise okur; okudukça azaltır.
  - Peki aynı anda yazıcı da okuyucu da uyanık olursa ne olur?

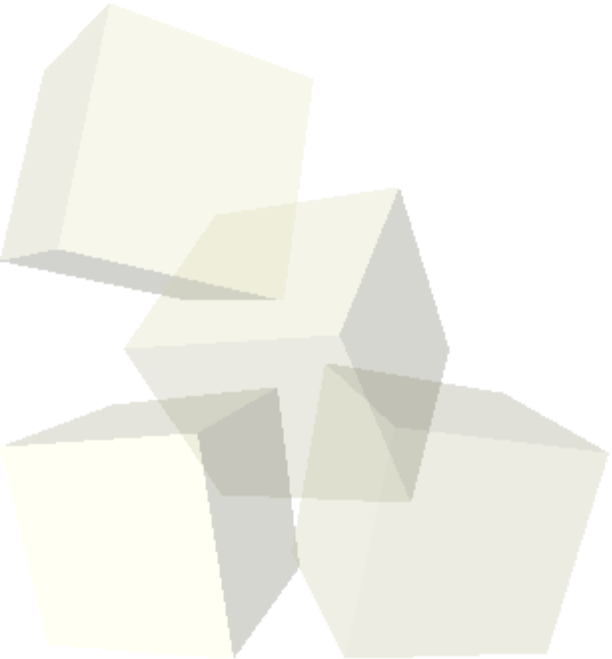
- Okuyucu/Yazıcı problemi eşzamanlılık problemlerinin en klasiğidir.
- Şu andaki hali ile bu problem üretici/tüketici sınıfı problemlere girer.
  - Tüketiciler üretilmeden tüketemezler.
  - Bunu sağlamak için **karşılıklı ayrım** (mutual exclusion) sağlanması gerekir.
- Karşılıklı ayrımı sağlamanın klasik bir yolu bir mutex değişkeni yada bir semafor kullanmaktır.

```
class VeriYapisi {

public void ekle(int i){
System.out.println("ekle");
try{
Thread.currentThread().sleep(Paylasilan.SURE);
}
catch(java.lang.InterruptedException ie){

}
}
```

```
public void cikar(int i){
System.out.println("cikar");
try{
Thread.currentThread().sleep(Paylasilan.SURE);
}
catch(java.lang.InterruptedException ie){

}
}

}
```

```
class Paylasilan {

public static final int N = 10;
public static final int SURE = 1000;

// 3 semafor
public static boolean mutex= true; //acik/kapali tipi
public static int bos = 1; // sayac tipi
public static int dolu = 9; //sayac tipi

public static VeriYapisi vy = new VeriYapisi();

}
```

```
class Uretici implements Runnable {

public void run(){

while(true){

// uret...
```

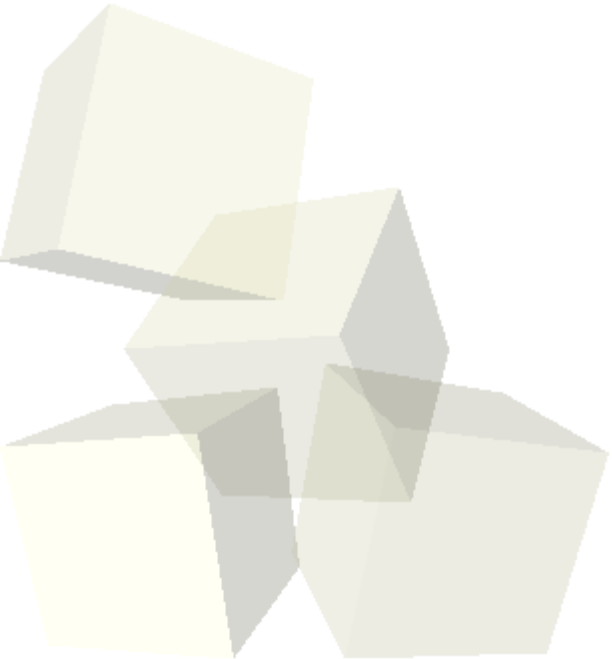


```
if(Paylasilan.mutex == true){

int urun=1;
Paylasilan.bos--;

Paylasilan.mutex = false; // girdim
System.out.println("Uretici girdi.");
Paylasilan.vy.ekle(urun);
System.out.println("Uretici cikti");
Paylasilan.mutex = true; // ciktim
```

```
Paylasilan.dolu++;
if (Paylasilan.dolu == Paylasilan.N)
try{
Thread.currentThread().sleep(Paylasilan.SURE);
}
catch(java.lang.InterruptedException ie){

}
}
```

```
else {
System.out.println("Uretici bekliyor.");
try{
Thread.currentThread().sleep(Paylasilan.SURE/2);
}
catch(java.lang.InterruptedException ie){
}
}
}
}
}
```

```
class Tuketici implements Runnable {

public void run(){

while(true){

// tuket...
```



```
if(Paylasilan.mutex == true){

int urun=1;
Paylasilan.dolu--;

Paylasilan.mutex = false; // girdim
System.out.println("Tuketici girdi");
Paylasilan.vy.cikar(urun);
System.out.println("Tuketici cikti");
Paylasilan.mutex = true; // ciktim
```

```
Paylasilan.bos++;
if (Paylasilan.bos == Paylasilan.N)
try{
Thread.currentThread().sleep(Paylasilan.SURE);
}
catch(java.lang.InterruptedException ie){

}
}
```

```
else {
System.out.println("Tuketici bekliyor.");
try{
Thread.currentThread().sleep(Paylasilan.SURE/2);
}
catch(java.lang.InterruptedException ie){

}
}
}

}
}
```

```
class UreticiTuketici {

public  static void main(String args[]){

Thread t1 = new Thread(new Uretici(),"Uretici");
Thread t2 = new Thread(new Tuketici(),"Tuketici");

t1.start(); t2.start();

}

}
```

Java'da Kanallar

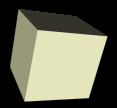
- Kendi semaforlarımız ile yazdığımız çözüm çok hoş oldu ama giderek artan sayıda paylaşılan kaynak için daha merkezi bir çözüm olmalıdır.
- Java'da her nesne için bir izleme durumu (monitor state) bulunur. Nesnenin bu durumunda ona aynı anda yalnızca bir kullanıcının erişmesi sağlanır.
  - İzleme durumunu seçimlerde oy sandığının olduğu kabine benzetebilirsiniz. Bir kişi kabine girdiğinde, o çıkana kadar kimse kabine giremez. Burada monitör durumu kabin, kullanıcılar kişiler ve erişilen nesne de oy sandığı olacaktır.

- Ancak izleme durumu tekniği pahalı bir tekniktir.
  - Bu nedenle Java'da her nesnenin izleme durumu aktif olmaz. Bizim o nesnenin durumunu açıkça aktif hale getirmemiz gerekir.
  - Bir nesnenin izleme durumunu aktif hale getirmek için **synchronized** (eşzamanlı) anahtar sözcüğünü kullanırız.

```
synchronized (nesne) {

    // burada nesneye olan erişim izleme durumunda

}
```

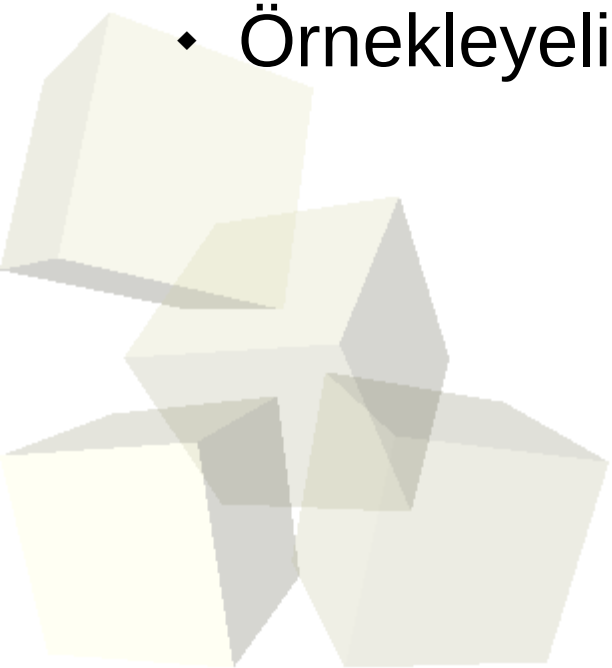
- Zaman zaman bir sınıfın belli bir yöntemi çağrıldığında yöntem bitene kadar izleme durumunda kalınması da gerekli olabilir.
  - Bu durumda yöntemi nitelediğimiz sözcüklere (public, static, vs.) **synchronized** sözcüğünü ekleriz.

```
public synchronized void f() {

    // izleme durumundayız

}
```



Java'da Kanallar

- Kanallar söz konusu olunca verilen bir örnek bir yazıcıya yazı yazdırmak isteyen iki uygulamadır.
  - Tipik olarak, işletim sistemi tarafından sağlanan basit bir servisi kullanan iki ayrı kanal olacaktır.
  - Bu kanalların eşzamanlı çalışması için yazıcıyı simgeleyen nesnenin izleme durumuna sokularak çalışılması yada servisin kendisinin izleme durumunda olması gereklidir.
  - Örnekleyelim.

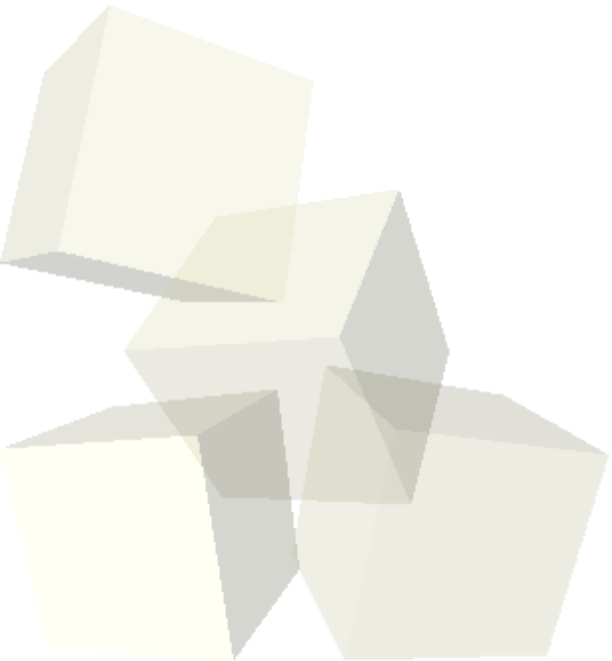


```
class Yazici {

public static final int SURE = 1000;

// synchronized // birazdan açacağız
public void yazdir(String metin){

System.out.print("Ust bilgi :: ");
try{
// metni cekiyor.
Thread.sleep(Yazici.SURE);
}
```

```
catch (java.lang.InterruptedException ie){

}

System.out.print(metin);
System.out.println(" :: Alt bilgi");
}
}
```

```
class Ofis implements Runnable {

String s;
Yazici y;

Ofis(String s, Yazici y){
this.s = s;
this.y = y;
}
```

```
public void run(){
y.yazdir(s);
}
}
```



```
class Synch1 {

public static void main(String args[]){

Yazici y = new Yazici();

Thread bir = new Thread (new Ofis("bir",y));
Thread iki = new Thread (new Ofis("iki",y));

bir.start(); iki.start();

}
}
```

- Kodda şu anda eşzamanlılık desteği yok

// synchronized // birazdan açacağız
public void yazdir(**String** metin){

  - Sonuç aşağıdaki gibi oluyor. Hataya dikkat edin.

  Ust bilgi :: Ust bilgi :: bir :: Alt bilgi
  iki :: Alt bilgi

- Eşzamanlılık desteğini açalım

synchronized // birazdan açacağız
public void yazdir(**String** metin){

  - Sonuç aşağıdaki gibi oluyor.

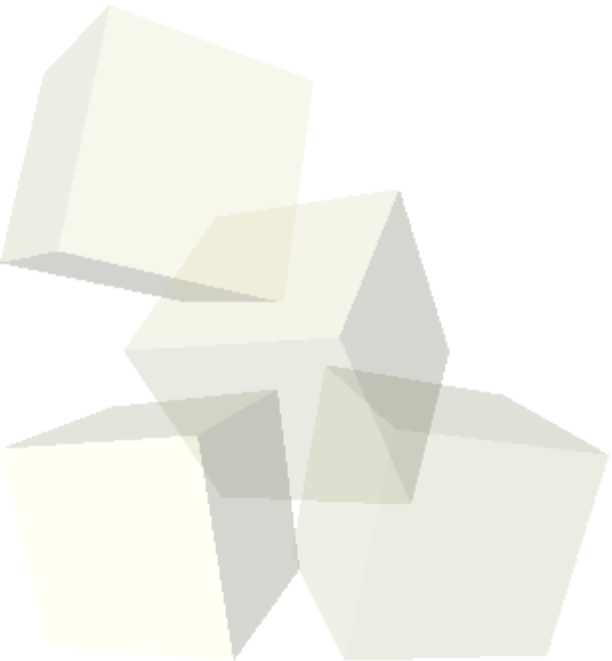
  Ust bilgi :: bir :: Alt bilgi
  Ust bilgi :: iki :: Alt bilgi

Java'da Kanallar

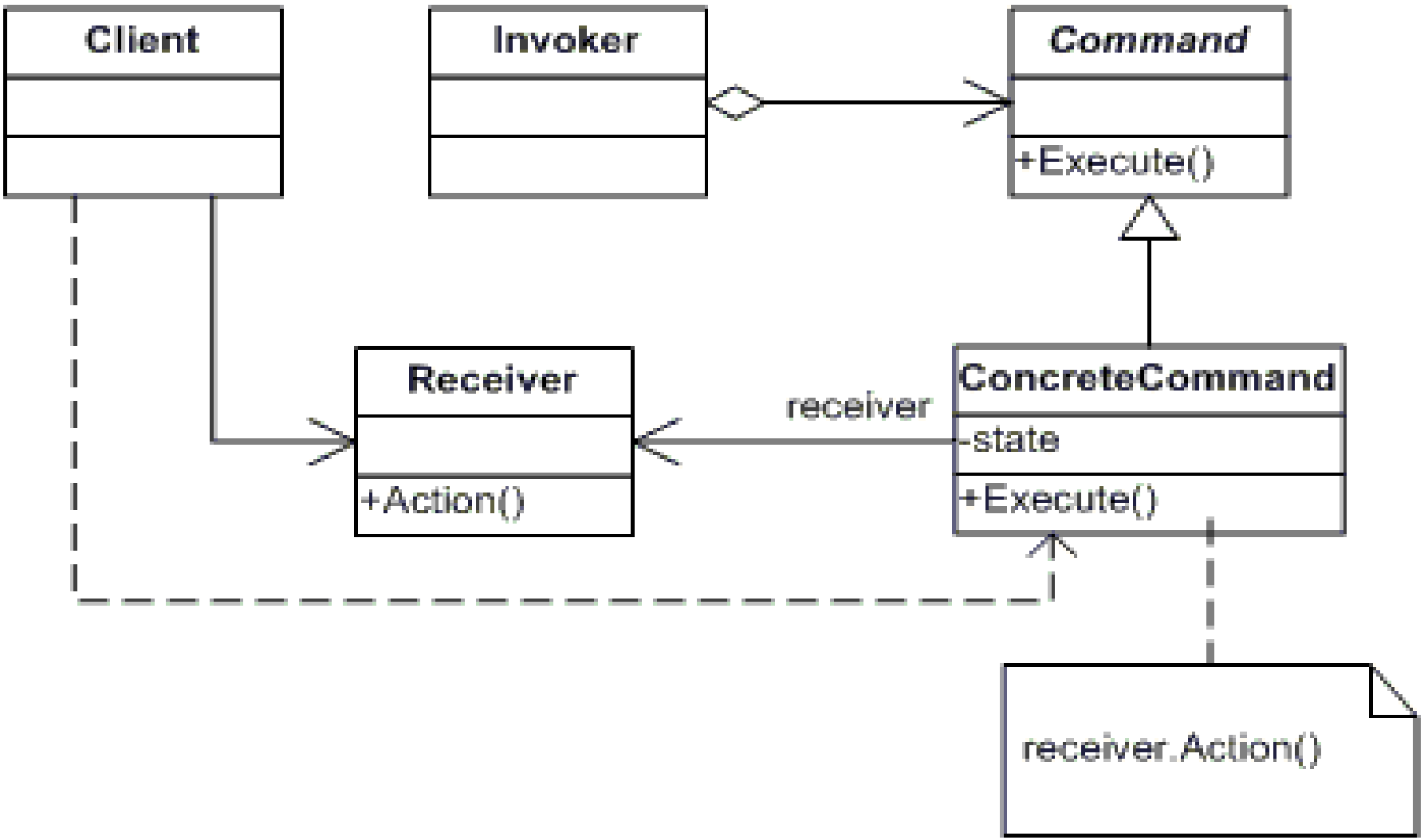
- Aynı örneği bu kez de yazıcıyı simgeleyen nesneyi eşzamanlı olarak tutarak yapalım.
- **Yazici** sınıfında eşzamanlılıkla ilgili bir şey yapmıyoruz.
  - Bu mantıklı bir durum olabilir. Çünkü çoklukla üçüncü kişilerin hazırladığı ve kanal güvenliği (thread safety) akılda tutulmadan yazılmış sınıfları kullanırız.
  - Kanal güvenliği performansı bir miktar düşürdüğü için bazen bilinçli olarak göz ardı edilir.

```
class Ofis implements Runnable {

// aynı veriler ve yapıcı

public void run(){
synchronized(y){
y.yazdir(s);
}
}

}
```

- Bu durumda çıktı yine olması gerektiği gibi doğru olacaktır.
- Java'daki izleme durumu tekniği masrafına rağmen çok iyi çalışmaktadır. Nadiren kendi semaforlarımızı yazmamız gerekir.



Komut Tasarım Biçimi



Komut Tasarım Biçimi

- Bir **Komut** nesnesinin tek becerisi, istendiği zaman işletilebilmesidir.
  - **Komut** ne yapacağını kendisi bilir.
  - Alıcı sınıf, **Komut** nesnesi üzerinde çeşitli ayarlar yapabilse de komutun ne yapacağı yine kendisine kalmıştır.
- Bu bize bir **Runnable** nesnesini ve onun **run()** yöntemini hatırlatıyor.
  - Ayar gerektirmeyen basit bir komut **Runnable** arabirimini uygulayan bir **anonim iç sınıf** (anonymous inner class) nesnesi olarak yazılabilir.
  - Bu şekildeki komutları serileştirme ile saklayabilir ve RMI ile kolayca uzak bilgisayarlara da aktarabiliriz.

```
import java.util.List;
import java.util.ArrayList;

class KomutVerici extends Thread {

private Alici hedef;

public KomutVerici(Alici hedef){
this.hedef = hedef;
}
```

```
public void run(){

Runnable komut = new Runnable(){
                    public void run(){
                     System.out.println("komut isletildi.");
                     }
                };
// komut hazır alıcıya  geçilecek
hedef.komutAl(komut);
System.out.println("KomutVerici kapaniyor.");
}

}
```

```
class Alici extends Thread{

List<Runnable> liste;

Alici(){
liste = new ArrayList<Runnable>(5);
}

public void komutAl(Runnable komut){
liste.add(komut);
}
```

```
public void run(){

System.out.println("Komutlar isletiliyor");

for (Runnable komut : liste){
Thread t = new Thread (komut);
t.start();
}
}
}
```

```
class Komut{

public static void main(String args[]){

System.out.println("Kanallar yaratiliyor");

Alici al = new Alici();
KomutVerici kv = new KomutVerici(al);
```

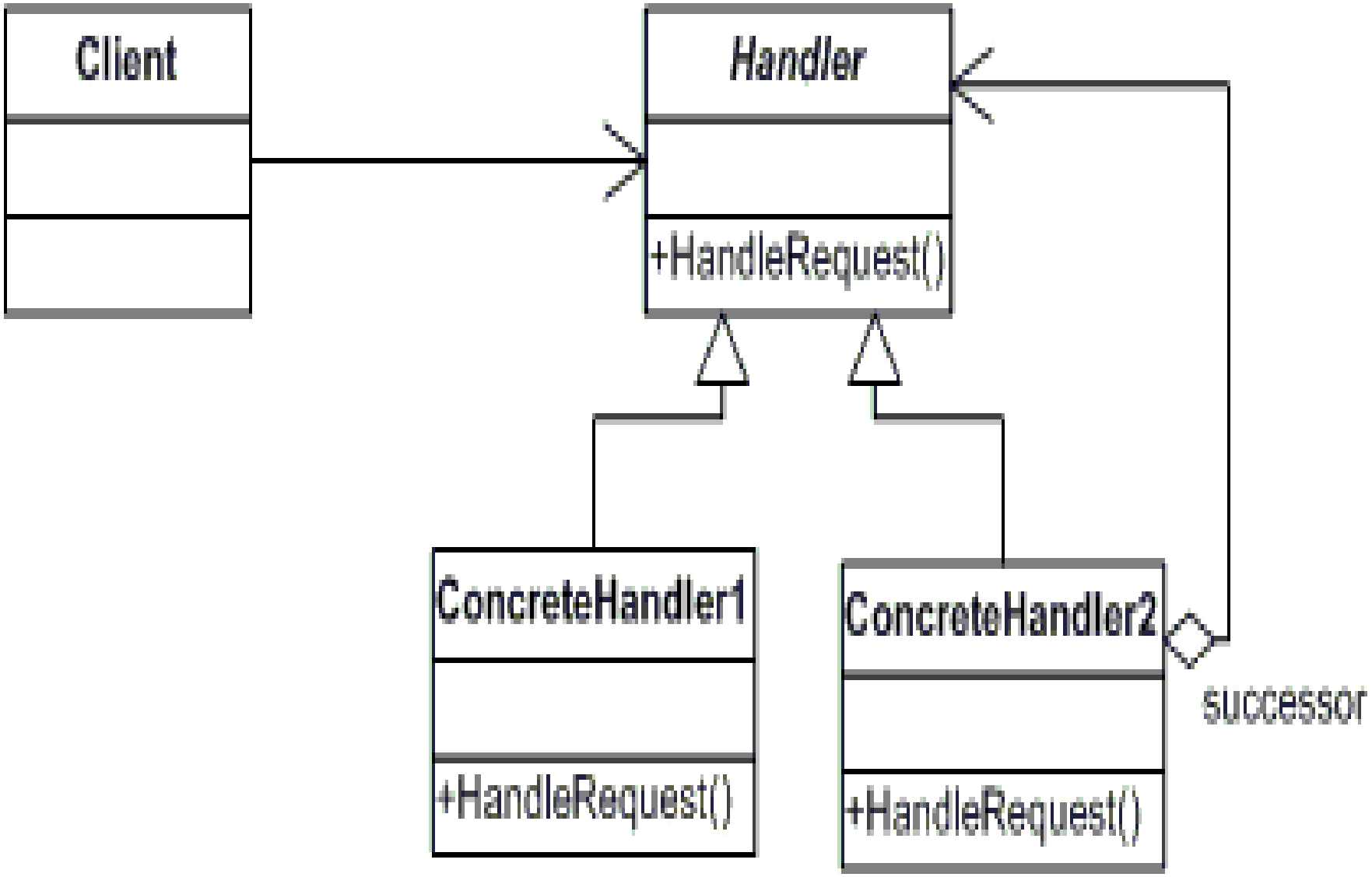
```
kv.start(); al.start();

try {
Thread.currentThread().sleep(5000);
}
catch(InterruptedException ie){
ie.printStackTrace(System.err);
}

System.out.println("Hersey bitti.");
}
}
```

Komut Tasarım Biçimi

- Uygulama çalıştırıldığı zaman çıktı aşağıdaki gibi

  Kanallar yaratiliyor
  KomutVerici kapaniyor.
  Komutlar isletiliyor
  komut isletildi.
  Hersey bitti.
- Komutu veren kanal kapatıldıktan sonra bile komut işletilebildi.
- Aslında bir komut elden ele gezebilir.
  - Bu şekildeki bir yapıya **Sorumluluk Zinciri** (Chain of Responsibility) adı verilir.
  - C++ ve Java'daki hata durumu işleme mekanizması tipik bir sorumluluk zinciridir.

Komut Tasarım Biçimi



- Kanallar arası haberleşme için uyuma/uyanma tekniğini kullanarak işlemlerin belli bir sıra ile olmasını garantiye alıyorduk.
  - Ancak işletim sisteminde çalışan başka kanallar da vardır. Bu kanallara ayrılan zamanın kaç milisaniye olacağını öngöremeyiz.
  - Bu nedenle hassas eşgüdüm için uyuma uyanmaya güvenemeyiz.
- Özellikle bir kanalın bir diğer kanalın uyumasını ve uyanmasını beklemesi, bir diğerine haber vermesi gibi konularda uyuma/uyanma yetersiz kalır.

- **Problem:** Bir kanalın uyuma/uyanma mekanizması kullanmak yerine **çok kısa bir süre için kenara çekilmesi** ve böylece diğer kanallardan birisinin devreye girebilmesi.
- **Çözüm: Thread** sınıfındaki yield() yöntemi şu anda çalışan kanalın anlık olarak durdurulmasını sağlar.

public static native void yield();

- Örnekleyelim.

```
class GonulluCekilen extends Thread{

public void run(){

for(int i=0; i<5;i++){
System.out.println("Uzun suren is");
System.out.println("GonulluCekilen cekildi");
Thread.yield();
}
}
}
```

```
class FirsatKollayan extends Thread {

public void run(){
System.out.println("minik is");
}

}
```

```
class Yield {

public static void main(String args[]){

Thread kanallarim[] = new Thread[5];
kanallarim[0] = new GonulluCekilen();
kanallarim[0].start();
for(int i = 1; i< kanallarim.length; i++){
kanallarim[i] = new FirsatKollayan();
kanallarim[i].start();
}
}
}
```

- Uygulama çalışınca çıktısı aşağıdaki gibi oluyor:

```
Uzun suren is
GonulluCekilen cekildi
Uzun suren is
GonulluCekilen cekildi
minik is
minik is
minik is
minik is
Uzun suren is
GonulluCekilen cekildi
Uzun suren is
GonulluCekilen cekildi
Uzun suren is
GonulluCekilen cekildi
```

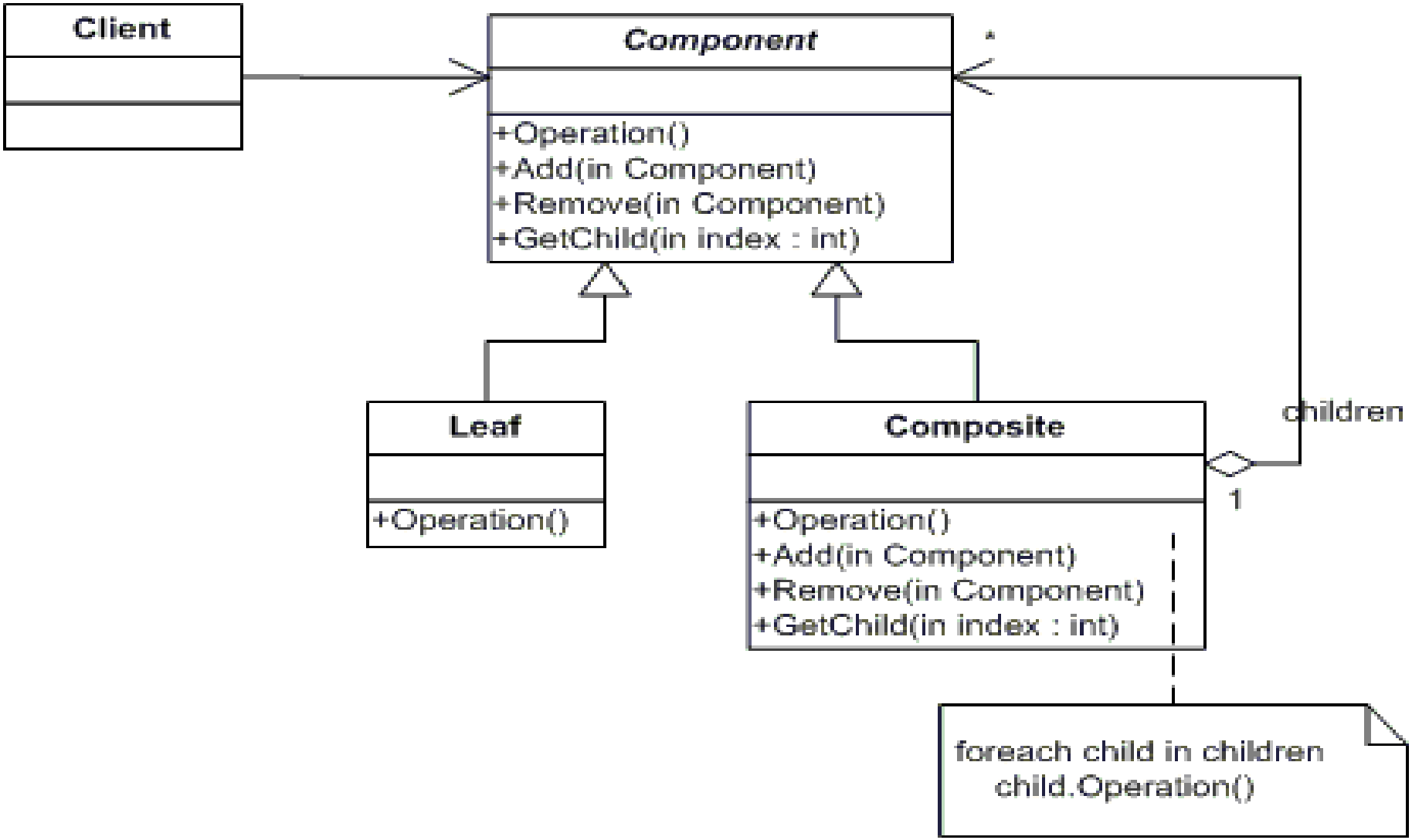
- Java'da herhangi bir nesnenin izleme durumunu kullanmak için **Object** sınıfında yazılan iki temel yöntem vardır.
  - **wait()** yöntemi, bu nesneye erişen kanalın başka bir kanal nesnenin **notify()** yöntemini (yada **notifyAll()** yöntemini) çağırana kadar beklemesini sağlar. Bu yöntem **izleme durumunun** (monitor state) **sahip değiştirmesine** dayanır.

```
synchronized (nesne) {
        while (<koşul sağlanmıyor>)
                nesne.wait();
            // artık işlerini yapabilir.
}
```

  - **notify()** yöntemi ise nesneyi bekleyen rastgele bir kanalı uyandıracaktır.

Kanal Grupları

- Java'da her kanalın bir **Kanal Grubu** (Thread Group) içinde bulunması önemli bir prensiptir.
- Kanal grupları kanalları ve başka kanal gruplarını içerecek biçimde organize edilmiş bir ağaç yapısıdır. Bu yapı bir **bileşke** (composite) örneğidir.
  - Yapının tepesindeki (kökündeki) kanal grubu hariç tüm kanal gruplarının bir ebeveyni (parent) bulunacaktır.
  - Birinci temel prensip, her seviyedeki kanal gruplarının doğrudan içinde bulunan kanalın sadece kendi kanal grubu hakkında bilgi edinmesi ve kanal grubu üzerinde toplu işlem yapmasıdır.
  - İkinci temel prensip, bir kanal grubunun ağacın sadece bir seviyesindeki kanalları kilitlemeye hakkı olmasıdır.


Client
Component
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
*
children
1
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
foreach child in children
child.Operation()

Kanal Grupları

- Kanal gruplarını uygulamak için **ThreadGroup** sınıfını kullanırız.
  - **add**(**Thread** t) yöntemi ile gruba kanal eklenebilir
  - **remove**(**Thread** t) yöntemi ile de belli bir kanal gruptan çıkabilir.
  - **list**() yöntemi ile kanal grubunu listeleyebiliriz.
  - **activeCount()** yöntemi ile kanal grubu ve altındaki kanal gruplarında etkin durumdaki toplam kanal sayısı hakkında yaklaşık bir sayı alabiliriz.
  - **enumerate**(**Thread**[] list) yöntemi ile kanal grubundaki aktif kanallara referansları aktarılan **Thread** dizisine yazabiliriz. Eğer dizi yeterince uzun değilse bazı kanallar için olan referanslar yazılmayacaktır.

```
class Kanalim extends Thread {

private static int seriNo = 0;
private int benimNumaram;

Kanalim(){
benimNumaram = ++seriNo;
}
```

```
Kanalim(ThreadGroup grup){
super(grup,"Kanalim");
benimNumaram = ++seriNo;
}

public String toString(){
return benimNumaram + "numarali Kanalim.";
}
```

```
public void run(){

System.out.println(this.toString() + "basladi.");
for (int i = 0; i<2; i++)
try  {
System.out.println(this.toString() + "uykuya yatiyor");
Thread.sleep(1000);
System.out.println(this.toString() + "uyandi");
}
```

```
catch(InterruptedException ie){

}
System.out.println(this.toString() + "kapaniyor.");
}

}
```

```
class KanalGrubu {

public static void main(String args[]){

ThreadGroup grup = new ThreadGroup("grubum");

for(int i=0; i<2; i++){
(new Kanalim(grup)).start();
}
```

```
System.out.println("main cekiliyor.");
Thread.yield();
System.out.println("main geri geldi.");

System.out.println();
int aktif  =  grup.activeCount();
System.out.println("Grupta "  + aktif + " aktif kanal
   var.");
grup.list();
System.out.println();
```

```
System.out.println("main cekiliyor.");
Thread.yield();
System.out.println("main geri geldi.");

System.out.println("Grubu donduruyorum.");
grup.suspend();
aktif  =  grup.activeCount();
System.out.println("Grupta "  + aktif + " aktif kanal
   var.");
grup.list();
System.out.println();
```

```
System.out.println("main cekiliyor.");
Thread.yield();
System.out.println("main geri geldi.");

System.out.println("Grup geri gelsin.");
grup.resume();

System.out.println("main bitti...");
}

}
```

main cekiliyor.
1numarali Kanalim.basladi.
1numarali Kanalim.uykuya yatiyor
2numarali Kanalim.basladi.
2numarali Kanalim.uykuya yatiyor
main geri geldi.

Grupta 2 aktif kanal var.
java.lang.ThreadGroup[name=grubum,maxpri=10]
    1numarali Kanalim.
    2numarali Kanalim.

```
main cekiliyor.
main geri geldi.
Grubu donduruyorum.
Grupta 2 aktif kanal var.
java.lang.ThreadGroup[name=grubum,maxpri=10
   ]
     1numarali Kanalim.
     2numarali Kanalim.

main cekiliyor.
main geri geldi.
Grup geri gelsin.
main bitti...
```

Kanal Grupları

1numarali Kanalim.uyandi
1numarali Kanalim.uykuya yatiyor
2numarali Kanalim.uyandi
2numarali Kanalim.uykuya yatiyor
1numarali Kanalim.uyandi
1numarali Kanalim.kapaniyor.
2numarali Kanalim.uyandi
2numarali Kanalim.kapaniyor.

Çok Kanallı Bir Ağ Sunucusu

- Çok kanallı bir ağ sunucusu aşağıdaki biçimde oluşacaktır.
  - Yönetici kanal diğer kanalları yaratmaktan ve istemcilerden gelen bağlantıları bu kanallara bağlamaktan sorumludur.
  - Yönetici kanalın bir daemon kanal olması yararlıdır.
  - Diğer kanallar ise sadece gelen istemci bağlantılarını işler.
  - Yönetici kanal her kanala belli bir zaman aşımı yada belli bir sayıda istemci isteği atar. Bu ikisinden birisi geçildikten sonra ilk fırsatta kanalı kapatır.

Çok Kanallı Bir Ağ Sunucusu