Rethinking Swing Threading

Rethinking Swing Threading

http://today.java.net/pub/a/today/2003/10/24/swing.html

IntelliJ projesi: thread-deney-1: file:///C:\java\thread-deney-1\thread-deney-1.ipr

Yavaş ve istikrarsız Swing uygulamalarının temelinden yatan en önemli sebep, swing zincirlerinin uygunsuzluğudur.

Bu yüzden en başta swingin tek zincirli modelini anlamalıyıcız.

Swingin temelinde, tek zincirli bir model yatıyor. Yani bütün swing komponentleriyle her zaman aynı zincir yoluyla etkileşime girilebilir. Bunun altında yatan sebep, çok zincirli bir mimarinin karmaşıklığı ve sebep olacağı yavaşlıktır.

Tek zincir modelini sağlamak için, swing komponentleriyle etkileşimi sağlayacak bir zincir tahsis edilmiştir. Bu zincire, swing zinciri, awt zinciri veya event-dispatch (olay dağıtım) zinciri denir.

Swing zinciri çok fazla iş yapıyor. Bütün boyama, event kontrolleri bu zincirde gerçekleşiyor. Dolayısıyla, eğer zaten mevcut işlerinin dışında yük oluşturan başka işler de verilirse, problemler oluşuyor. Genellikle bu tarz problemler, swing dışı ağır işlerin bir olay dinleyicisi metodunda yürütülmesi (actionPerformed gibi) sebebiyle oluşuyor. Mesela veritabanı işlemlerinin buralarda yürütülmesi gibi. Bu durumda, swing zinciri veritabanı işiyle meşgul olduğu için, kullanıcılar, uygulamanın donduğunu düşünür. Halbuki aslında böyle değil. Dolayısıyla, her iş uygun zincirinde yürütülmeli.

Şimdi sorunlara ve bunları nasıl çözeceğimize bakalım. Sorunların çoğunun temelinde, senkronize kod modelini, asenkron swing zincir medoliyle yürütmek yatıyor. Bizim çözümümüz, bütün kod modelini, asenkron modele uygun tasarlamak olacak.

Standart swing zincir çözümü

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\44.gif

Bir arama uygulaması.

Düğmeye basınca aşağıdaki kod çalışıyor:

```
private void searchButton_actionPerformed() {
    outputTA.setText("Searching for: " +
                     searchTF.getText());
    //Broken!! Too much work in the Swing thread
    String[] results = lookup(searchTF.getText());
    outputTA.setText("");
    for (int i = 0; i < results.length; i++) {
        String result = results[i];
        outputTA.setText(outputTA.getText() +
                         '\n' + result);
    }
}
```

lookup metodu, veritabanı sorgusu yerine geçiyor. Burada içerikle ilgilenmiyoruz, sadece zamanı test ediyoruz. Bu yüzden lookup içinde bir koçan (stub) yürütüyoruz. Sadece thread.sleep() işlemini beş saniye boyunca çağırıyoruz.

Düğmeye basıldıktan sonra ekran görüntüsü şu şekilde:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\45.gif

Dikkat ederseniz, Go düğmesi, hala basılı görünüyor. Bunun sebebi, düğmenin basılı olmadığı (non-pressed) görüntünün oluşması için, actionPerformed metodunun dönmesi (return) gerekiyor.

Bir başka hata da, biz ilk olarak metin alanına "abcde" yazılmasını belirttiğimiz halde, bunu yapmıyor. Niye? Çünkü Swing yeniden boyamaları (repaint) hemen gerçekleşmez. Boyama isteği, swing olay kuyruğunda, swing zinciri tarafından işlenmeyi bekler. Ancak burada biz swing zincirini, lookup ile meşgul ediyoruz. Bu yüzden boyamayı yapamıyor.

Bu sorunları düzeltmek için, lookup metodunu swing dışı bir zincire taşıyoruz. İlk eğilimimiz, bütün metodu yeni bir zincirde yürütmek. Ancak bu çalışmaz, çünkü swing komponentleriyle etkileşim sadece swing zincirinden kurulabilir.

```
private void searchButton_actionPerformed() {
    new Thread(){
        public void run() {
           outputTA.setText("Searching for: " +
                     searchTF.getText());
           //Broken!! Too much work in the Swing thread
           String[] results = lookup(searchTF.getText());
           outputTA.setText("");
           for (int i = 0; i < results.length; i++) {
         String result = results[i];
               outputTA.setText(outputTA.getText() +
                        '\n' + result);
        }
    }.start();
}
```

Bu kod, hiçbir şey yapmaz.

Bunun yerine sadece lookup metoduna, yeni zincirden erişelim:

```
private void searchButton_actionPerformed() {
    outputTA.setText("Searching for: " +
                     searchTF.getText());
    //the String[][] is used to allow access to
    // setting the results from an inner class
    final String[][] results = new String[1][1];
    new Thread(){
        public void run() {
           results[0] = lookup(searchTF.getText());
        }
    }.start();
    outputTA.setText("");
    for (int i = 0; i < results[0].length; i++) {
        String result = results[0][i];
        outputTA.setText(outputTA.getText() +
                        '\n' + result);
    }
}
```

Burada results anonim iç sınıf tarafından erişildiği için, final olmalı. Bu durumda içeriği değiştiremeyiz. Biz de problemi düzeltmek için, results değişkenini göndermedik, onu bir array yapıp, ilk elemanını gönderdik.

Bu durumda uygulamayı çalıştırdığımızda:

file:///F:\Belgelerim\Notlarim\Java%20Notlarim\Resimler\Makaleler\46.gif

Dikkat ederseniz, metin alanında, sadece null görünüyor.

Bunun sebebi şu: Görüntü kodu, yeni zincir işlemi bitirdiğinde çalışmaya başlamıyor. Bunun yerine zincir işleme başladığında, çalışmaya başlıyor.

Bu sorunu çözmek için, SwingUtilities sınıfının iki metodunu kullanacağız: invokeLater() ve invokeAndWait().

Her iki metot da bir Runnable alıyor ve bunu swing zinciri içinde çalıştırıyor.

invokeAndWait metodu, Runnable işlemi bitirinceye kadar, swing zincirini durduruyor. invokeLater Runnable nesnesini, asenkronize çalıştırıyor. Yani swing zinciriyle paralel çalıştırıyor.

Şimdi kodumuz:

```
private void searchButton_actionPerformed() {
    outputTA.setText("Searching for: " +
                     searchTF.getText());
    final String[][] results = new String[1][1];
    new Thread() {
        public void run() {
            //get results.
            results[0] = lookup(searchTF.getText());
```

```
            // send runnable to the Swing thread
            // the runnable is queued after the
            // results are returned
            SwingUtilities.invokeLater(
                new Runnable() {
                    public void run() {
                        // Now we're in the Swing thread
                        outputTA.setText("");
                        for (int i = 0;
                             i < results[0].length;
                             i++) {
                            String result = results[0][i];
                            outputTA.setText(
                                outputTA.getText() +
                                '\n' + result);
                        }
                    }
                }
            );
        }
    }.start();
}
```

İki tane zincir içiçe. Önce lookup zinciri. Bu swing dışında gerçekleşiyor. Sonra, metin alanını güncellediğimiz yeni bir zincir. (Runnable için: file:///?Kod%20Parcalari|Runnable|416|9)

Ancak bu kolay okunur bir metot değil. Değişken kapsamlarına ve zincirlerinin işlem sıralarına çok özen göstermemiz gerekti. Eğer içiçe katmanlar, paylaşılmış referanslar ve belirli bir işlem sırası olsaydı, bunu kontrol etmemiz çok zor olurdu.

Temel sorun: Senkron-asenkron içiçeliği

Temel Sorun:

Temel sorun, senkron modeli, asenkron modelin içine giydirmekten kaynaklanıyor.

Bu çok yaygın bir sorun. Buna yönelik bir çözüm, hazır çerçeveler kullanmak. Ancak bu tarz çerçevelerin birçoğu yine belirli bir boyutun ötesinde sorunun tekrar ortaya çıkmasına engel olamıyor.

Temel çözüm, yaklaşım tarzımızı değiştirmek:

Çözüm: Olay Temelli Programlama

İki tane sorunu çözeceğiz:

1. Kodu, uygun zincirde işletmek.
2. SwingUtilities.invokeLater metodunu kullanarak asenkron işlem

Ancak asenkron işlemler de şu problemlere sebep oluyor:

1. Bağımlı komponentler
2. Değişken aktarma zorluğu
3. İşlem sıralaması.

Mesaj temelli sistemlerden, Java Messaging Servicei (JMS) düşünelim. Bunlar asenkron bir ortamda komponontlerin arasındaki bağları gevşetiyor. Mesajlaşma sistemi, sisteme asenkron olaylar fırlatıyor. İlgili taraflar bu olayları dinliyor ve onlara tepki veriyor. Sonuç, gevşek ve modüler bir sistem. Modüller sisteme eklenip, sistemden çıkartılabiliyor. Sistemin kalan kısmı bundan etkilenmiyor. Bağımlılıklar asgariye indirilmiş oluyor.

Şimdi şu bağımsızlaştırma meselesine bakalım ve buradan asenkron ortama gidelim. Asenkron ortamların sorunlarını çözdükten sonra, zincirleme meselesine geri dönelim.

Önce veritabanı kodunu, ayrı bir sınıfa taşıyalım. Böylece arayüz sınıfıyla, veritabanı mantığını birbirinden ayıralım:

```
class LookupManager {
   private String[] lookup(String text) {
      String[] results = ...
      // database lookup code
      return results
```

```
    }
}
```

Şimdi asenkron modele doğru gidelim. Bu çağrıyı asenkron yapmak için, çağrıyı dönen değerden soyutlamalıyız. Yani çağrı hiçbir şey döndürmemeli. Diğer sınıfların haberdar olmak istediği eylem nedir, diye sormalıyız. Aramanın bitmesi. Bu eylemi yansıtan bir dinleyici interfacei tanımlayalım. Bu interfacein lookupCompleted adlı metodu olacak:

```
interface LookupListener {
    public void lookupCompleted(Iterator results);
}
```

Daha iyisi, bir string arrayi yollamak yerine, LookupEvent sınıfı tanımlayıp, bunların nesnelerini göndermek olur. Bu ayrıca ileride başka bilgiler de göndermemiz durumunda, LookupListener interfaceini değiştirmeden bunu yapmamızı sağlar.

```
public class LookupEvent {
    String searchText;
    String[] results;

    public LookupEvent(String searchText) {
        this.searchText = searchText;
    }

    public LookupEvent(String searchText,
                       String[] results) {
        this.searchText = searchText;
        this.results = results;
    }

    public String getSearchText() {
        return searchText;
    }

    public String[] getResults() {
        return results;
    }

}
```

Dikkat ederseniz, bu sınıf değiştirilemez. Bu önemli, çünkü yol boyunca bu olayı kimlerin işleyeceğini bilmiyoruz.

LookupManager´a gerekli metotları ekleyelim:

```
List listeners = new ArrayList();

public void addLookupListener(LookupListener listener){
    listeners.add(listener);
}

public void removeLookupListener(LookupListener listener){
    listeners.remove(listener);
}
```

Olay fırlatmak için:

```
private void fireLookupCompleted(String searchText,
                                 String[] results){
    LookupEvent event =
        new LookupEvent(searchText, results);
    Iterator iter =
        new ArrayList(listeners).iterator();
    while (iter.hasNext()) {
        LookupListener listener =
            (LookupListener) iter.next();
        listener.lookupCompleted(event);
    }
}
```

Dikkat ederseniz, dinleyiciler koleksiyonunun bir kopyasını oluşturuyoruz. Bunun sebebi, olayın sonucunda, dinleyicilerden birinin kendisini çıkartmak isteyebileceğinden kaynaklanıyor. Bu durumda, sorunlar çıkardı.

Şimdi olayı fırlatalım:

```
public void lookup(String text) {
    //mimic the server call delay...
    try {
        Thread.sleep(5000);
    } catch (Exception e){
        e.printStackTrace();
    }
    //imagine we got this from a server
    String[] results =
        new String[]{"Book one",
                     "Book two",
                     "Book three"};
    fireLookupCompleted(text, results);
}
```

Şimdi arayüz sınıfının bu olayı dinleyebilmesi gerekiyor:

```
public class FixedFrame implements LookupListener

public void lookupCompleted(final LookupEvent e) {
    outputTA.setText("");
    String[] results = e.getResults();
    for (int i = 0; i < results.length; i++) {
        String result = results[i];
                outputTA.setText(outputTA.getText() +
                                 "\n" + result);
    }
}
```

Son olarak kayıt yapıyoruz:

```
public FixedFrame() {
    lookupManager = new LookupManager();
    //here we register the listener
    lookupManager.addListener(this);
    initComponents();
    layoutComponents();
}
```

Aynı şeyi lookup´ın başlangıcı için de yapalım:

```
public void lookup(String text) {
    fireLookupStarted(text);

    //mimic the server call delay...
    try {
        Thread.sleep(5000);
    } catch (Exception e){
        e.printStackTrace();
    }
    //imagine we got this from a server
    String[] results =
        new String[]{"Book one",
                     "Book two",
                     "Book three"};

    fireLookupCompleted(text, results);
}

private void fireLookupStarted(String searchText){
    LookupEvent event =
        new LookupEvent(searchText);
    Iterator iter =
        new ArrayList(listeners).iterator();
    while (iter.hasNext()) {
        LookupListener listener =
            (LookupListener) iter.next();
        listener.lookupStarted(event);
    }
}
```

Ancak bu durumda henüz bir sonuç kümesi yok, sadece arama metnini fırlatıyoruz.

```
public void lookupStarted(final LookupEvent e) {
    outputTA.setText("Searching for: " +
                     e.getSearchText());
}
```

Şimdi bağımsızlaştırmanın sağladığı kolaylığı görelim. Mesela hangi olay fırlatıldı, bunları takip edip, loglamak istiyoruz. Bir log sınıfı tanımlıyoruz ve olayları dinliyoruz:

```
public class Logger implements LookupListener {

    public void lookupStarted(LookupEvent e) {
        System.out.println("Lookup started: " +
                           e.getSearchText());
    }

    public void lookupCompleted(LookupEvent e) {
        System.out.println("Lookup completed: " +
                           e.getSearchText() +
                           " " +
                           e.getResults());
    }

}
```

Kayıt yapıyoruz:

```
public FixedFrame() {
    lookupManager = new LookupManager();
    lookupManager.addListener(this);
    lookupManager.addListener(new Logger());
    initComponents();
    layoutComponents();
}
```

Buradaki tek sorun, en başta harcadığımız kurulum maliyeti. Yeni bir olay tanımlamak, biraz zaman alıyor.

Zincirleme

Zincirleme

Şimdi, asenkron ortamla ilgili sorunları çözdük: Komponentleri, dinleyiciler vasıtasıyla bağımsızlaştırdık. Değişken aktarmayı, olay nesneleriyle çözdük. İşlem sırasını, olay oluşturma ve kaydetme sayesinde çözdük. Şimdi zincir meselesine geri dönelim.

```
public void lookupCompleted(final LookupEvent e) {
  //notice the threading
  SwingUtilities.invokeLater(
        new Runnable() {
            public void run() {
                outputTA.setText("");
                String[] results = e.getResults();
                for (int i = 0;
                     i < results.length;
                     i++) {
                    String result = results[i];
                    outputTA.setText(outputTA.getText() +
                                     "\n" + result);
                }
            }
        }
    );
}
```

Niçin bu kodu SwingUtilities içinden çalıştırıyoruz?

Çünkü biliyoruz ki, bu kod, swing dışı bir zincirden çağrılacak. Halbuki swing komponentleriyle etkileşime girmek için, swing zincirinden işlem yapmalıyız. Bu yüzden, bu kodu, swing zinciri içine yönlendiriyoruz (redirect).

Eğer dinleyicilerden birinin işlemi uzun sürecekse, bunun için de yeni bir zincir oluşturmalı. Swing harici dinleyicilerin, çağıran zincirde işlem yapması da mümkün, eğer hızlıysalar.

Şimdi LookupManager´ın zincirini oluşturalım:

```
private void searchButton_actionPerformed() {
    new Thread(){
        public void run() {
            lookupManager.lookup(searchTF.getText());
        }
    }.start();
}
```

Burada actionPerformed içinde yeni zinciri oluşturduk. Bunun yerine lookup içinde de oluşturabilirdik. Ancak biz bunu tercih etmedik, çünkü bu zincire, swing içinde işlem yapmamızdan dolayı ihtiyaç duyuyoruz. Kendine ait olmayan bir yerde bulunması, ek bir bağımlılık oluştururdu. Ayrıca bu LookupManager, swing harici bir bağlamdan da çağrılabilir. Mesela Logger gibi.

Gördüğünüz gibi bir kere olay dinleme yapısını kurduktan sonra, zincirleme oldukça kolay.

Easily Find Swing Threading Mistakes

http://www.clientjava.com/blog/2004/08/20/1093059428000.html

Swing komponentlerini swing dışı threadlerde değiştirip değiştirmediğini denetleyen bir kod.

RepaintManager bütün swing operasyonlarında ortak sınıftır. Bu çağrıldığı vakit, swing threadinin doğru yerde olduğunu kontrol eden bir kod yazıyoruz:

```
RepaintManager.setCurrentManager(MyRepaintManager());

...

public class ThreadCheckingRepaintManager extends RepaintManager {
    public synchronized void addInvalidComponent(JComponent jComponent) {
        checkThread();
        super.addInvalidComponent(jComponent);
    }

    private void checkThread() {
        if (!SwingUtilities.isEventDispatchThread()) {
            System.out.println("Wrong Thread");
            Thread.dumpStack();
        }
    }

    public synchronized void addDirtyRegion(JComponent jComponent, int i,
                                            int i1, int i2, int i3) {
        checkThread();
        super.addDirtyRegion(jComponent, i, i1, i2, i3);
    }
}
```

Daha sonra uygulamayı normal bir şekilde çağırıyoruz. Eğer bir swing manipülasyonunu swing dışı bir threadde yaptıysak, bize Wrong thread hatası üretiyor:

Mesela aşağıda:

```
public void actionPerformed(ActionEvent event) {
            new Thread(new Runnable() {
                public void run() {
                    try {
                        Thread.sleep(1000);
```

```
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                for (int i = 0; i < 100; i++) {
                    val++;
                    listModel.addElement(new Integer(val));
                }
            }
        }).start();
    }
}
```

listModele element eklemek, swing dışı bir threadde gerçekleştirilmiş.