

QT İle Uygulama Geliştirme Kurs Notları

Kaan ASLAN

C ve Sistem Programcıları Derneği

Güncelleme Tarihi: 03/07/2010

Bu kurs notları Kaan ASLAN tarafından yazılmıştır. Kaynak belirtilmek koşuluyla her türlü alıntı yapılabilir.

1. Qt Ortamı Hakkında Temel Bilgiler

Bu bölümde olarak Qt ortamı hakkında temel bilgiler verilecektir.

1.1. Qt Nedir?

Qt (genellikle “kyu:t” biçiminde okunuyor fakat “kyu ti” biçiminde de okunabiliyor) C++ tabanlı bir GUI ortamıdır (GUI framework). Bu ortamın ana amacı C++ kullanarak pencereci (yani GUI) uygulamalar geliştirmektir. Her ne kadar Qt’nin ana amacı GUI uygulamalar geliştirmek olsa da bu ortam zamanla pek çok gereksinimi karşılayacak kütüphanelere ve araçlara sahip duruma gelmiştir.

Anahtar Notlar: İngilizce “framework” sözcüğünü Türkçe “ortam” olarak kullanacağız. Örneğin “Qt Framework” yerine “Qt Ortamı” gibi. GUI ise “Graphical User Interface” sözcüklerinden kısaltılmıştır. “Grafik Kullanıcı Arayüzü” anlamına gelmektedir.

1.2. Qt’nin Kısa Tarihi

Qt ortamı bir kütüphane biçiminde 1990’lı yılların başlarında geliştirilmeye başlanmıştır. Geliştiriciler daha sonra Troll Tech isimli firmayı kurdular ve Qt bu firma tarafından geliştirilmeye devam etti. Qt’nin ilk versiyonları oldukça sade idi. Versiyonlar ilerledikçe kütüphaneye yeni özellikler eklendi. Qt’nin ilk versiyonları yalnızca UNIX/Linux sistemlerinde (yani X11 üzerinde) çalışıyordu. Daha sonra Qt diğer platformlarda da kullanılabilir yani “cross platform” hele getirildi. Böylece önce Windows sistemlerinde sonra da Mac OS X sistemlerinde Qt kullanılmaya başlandı.

Troll Tech Qt’yi 2008 yılında Nokia’ya sattı. O zamanlar Nokia’nın kafası karışıkta. Symbian sistemlerinin artık modern gereksinimleri karşılayamayacağını Nokia anlamıştı fakat kendi telefonları için hangi işletim sisteminin kullanılması gerektiği konusunda sağlam bir karar verememişti. Maemo, Mee Go gibi Linux tabanlı işletim sistemlerini denediyse de başarılı olamadı. İşte Nokia bu sıralarda Qt’yi telefonları için kullanacağı işletim sisteminde faydalanmak amacıyla satın aldı. Nokia daha sonra Microsoft ile anlaştı ve akıllı telefonlarında Microsoft’un “Windows Mobile” işletim sistemini kullanmaya karar verdi. Böylece Qt’de şirket için atıl duruma düşmüş oldu. Nokia’da zaten maddi sıkıntı da baş göstermeye başlamıştı. Bunların sonucu olarak Qt’yi Digia isimli firmaya 2012’de sattı. Zaten daha sonra Nokia’nın da büyük bölümünü Microsoft satın aldı. Digia Qt işini 2014 yılında kendisine bağlı “Qt Company” isimli bir firmaya devretti. (Tabii “Qt Company” ile “Digia” aynı firmalar olarak değerlendirilebilir.)

Nokia Qt’yi satın aldığıında “Qt Project” isimli bir topluluk oluşturmuştu. Bu grup açık kaynak kodlu olarak Qt projesinin geliştirilmesine yardımcı oluyordu. Bugün hala Digia’nın yanı sıra “Qt Project” topluluğu da önemli katkılar sağlamaktadır. “Qt Project” topluluğu bir IDE yaratma girişiminde de bulundu. Böylece “Qt Creator” isimli IDE doğdu. Bugün Qt Creator IDE’si oldukça tatmin edici bir düzeye gelmiştir. Artık pek çok programcı Visual Studio, Eclipse, Netbeans IDE’leri yerine geliştirmeyi doğrudan Qt Creator üzerinde geliştirme yapmaktadır. “Qt Creator” da Qt’nin

kendisi gibi Linux, Windows ve Mac OS X gibi platformlarda çalışabilmektedir. Yani "cross platform" özelliğine sahiptir.

Qt 4.0'la ve 5.0'la birlikte bazı önemli değişiklikler ve yenilikleri de bünyesine katmıştır. Bugün için Qt'nin son stabil versiyonu 5.6'dır. Qt'nin son zamanlardaki en önemli yenilikleri "Qt Quick" denilen ve QML sentaksıyla arayüz oluşturmaya izin veren platformudur. Aynı zamanda Qt Android ve IOS sistemlerinde de kullanılabilir hale getirilmiştir.

1.3. Qt'nin Kullanım Lisansları

Eskiden Troll Tech firması Qt için iki lisans sunuyordu. Birincisi katı bir "açık kaynaklı (open source)" lisanstı. Buna göre biz eğer Qt'ye para ödememişsek yazdığımız programların kaynak kodlarını açmak zorundaydık. Diğeri ticari (commercial) lisanstı. Eğer biz Troll Tech'e para ödersek ürünlerimizi açmak zorunda kalmıyorduk. Daha sonra Troll Tech açık kaynaklı lisansı gevşetmeye başlamıştır. Nokia Qt'yi satın alınca açık kaynaklı lisans daha da gevşetmiştir.

Bugün Qt ticari (commercial), GPL, LGPL ve diğer bazı open source lisanslarla sunulmaktadır. Yani özetle bugün için hukuki durum şöyledir: Biz Qt'yi indirip kendi projelerimizde istediğimiz gibi kullanabiliriz. Projemizi açmak ya da bedava dağıtmak zorunda değiliz. Ancak eğer Qt'nin kaynak kodları üzerinde değişiklik yapıp onu da açmak istemiyorsak Digia'ya para ödeyip ticari lisansa sahip olmamız gerekir.

1.4. Library, ToolKit ve Framework Kavramları

İngilizce'de kullanılan "library", "toolkit" ve "framework" terimleri birbirlerine benzer kavramları belirtmektedir. "Library" yani kütüphane içerisinde derlenmiş kodların bulunduğu dosyalardır. Biz "library" içerisindeki fonksiyonları sınıfları istediğimiz yerde kullanabiliriz. "Toolkit" terimi çoğu kez belli bir amaca yönelik "library" topluluğu olarak kullanılmaktadır. "Framework" kavramının pek çoklarına göre ayırıcı özelliği akış kontrolünün terslenmiş olması (inversion of control) durumudur. Yani "framework"lerde yalnızca biz fonksiyonu çağırırız. Framework içerisindeki kodlar akış kontrolünü ele alır bazı olaylarda bizim kodlarımızı çağırırlar. Ayrıca "framework"ler pek çok yardımcı araca da sahip olabilmektedir. Genel olartak "framework" kavramı daha büyük ve geniş kapsamlı bir organizasyon belirtmektedir. Tabii ne olursa olsun bu üç kavramın herkes tarafından kabul edilen kesin sınırları çizilmiş tanımlarının olmadığını söyleyelim.

Qt için "library" "toolkit" ve "framework" terimlerinin hepsi bir dönem kullanılmıştır. Bugün itibarıyla Qt'ye "framework" demek daha uygun gözükmektedir.

1.5. İşletim Sistemlerinin GUI Alt Sistemlerine Genel Bakış

Bilgisayar sistemlerine terminal bağlanması 1957-58 yıllarına denk gelmektedir. O yıllara kadar programcılar bilgisayarlarla doğrudan etkileşemiyorlardı. Ancak 1957-58 yıllarından itibaren programcılar bilgisayarlarla klavye ve ekran yoluyla doğrudan etkileşim içerisine girmeye başlamıştır. Bu tür terminal ekranlarına konsol (console) ekranları da denilmektedir.

80'li yılların başında IBM ilk kişisel bilgisayarları çıkarttığında DOS işletim sistemi grafik bir arayüze sahip değildi. (Ancak Machintosh sistemleri ilk grafik arayüz kullanan sistemlerdi.) Daha Microsoft Windows sistemleriyle birlikte grafik arayüz kullanımına geçmiştir. UNIX dünyası da 80'li yılların sonlarına doğru yavaş yavaş grafik arayüzlerle tanıştı. Fakat bugün UNIX/Linux sistemleri ağırlıklı olarak server sistemlerinde kullanılmaktadır. Fakat server sistemlerinde de grafik arayüz genellikle -yavaşlatıcı bir etken oluşturduğu için- bulundurulmamaktadır. Ancak kişisel bilgisayara yüklenmiş olan UNIX/Linux sistemleri (bunlara "client" makineler de diyebiliriz) ağırlıklı olaeak grafik arayüzlerle kullanılmaktadır.

Eskiden konsol ekranlarında karakterler ekrana kalıp olarak ekrana basılıyordu. Zaten bu devirlerde ekranın kontrolünü sağlayan elektronik birimler şimdakilere göre ilkel düzeydeydi. Sonra ekran nokta temelinde kontrol edilebilmeye başlandı. Bugün ekrandaki görüntüyü oluşturan en küçük noktasal birime pixel ("picture element"

sözcüklerinden kısaltma) denilmektedir. Modern grafik kontrol kartları bu pixel'leri depolayıp ekrana gönderme konusunda çok yetenek kazanmışlardır.

Windows sistemlerinin çekirdek (kernel) ile entegre edilmiş bir grafik sistemi vardır. Bu grafik sistem geliştirici düzeyinde GDI denilen bir API kütüphanesi ile kullanılıyordu. Hala bu kullanım devam etmektedir. Sonra Microsoft yine GDI üzerine GDI+ isimli bir kütüphane daha tasarlamıştır. .NET Form uygulamaları bu kütüphaneyi kullanmaktadır. Fakat Microsoft zaman içerisinde GDI'nin yavaş olduğunu gördüğü için pek çok katmandan geçmeden grafik işlemlerini daha çabuk yapan ve ismine "Direct-X" denilen bir kütüphane daha geliştirmiştir. Direct-X önceleri yalnızca oyun ve animasyon programcıları tarafından kullanılıyordu. Daha sonra Direct-X'i kullanarak GUI elemanları oluşturan kütüphaneler GUI kütüphaneleri de ortaya çıkmıştır. Bununların en bilineni .NET dünyasında kullanılan WPF (Windows Presentation Foundation) ortamıdır.

UNIX/Linux sistemlerinde grafik çalışma çekirdek ile entegre edilmemiştir. Örneğin Linux çekirdeğinde grafik çalışmayla ilgili hiçbir kod yoktur. UNIX/Linux sistemlerinde temel grafik işlemler ayrı bir katman tarafından yapılmaktadır. Bu katmana X11 ya da halk arasındaki ismiyle "X Window" denilmektedir. X Window sistemi client-server mantığıyla çalışan aşağı seviyeli bir katmandır. X Window çok temel pencere ve grafik işlemlerini yapmak için düşünülmüştür. Bu nedenle modern GUI uygulamaları için yetersizdir. X Window sistemini doğrudan kullanabilmek için XLib denilen kütüphaneden faydalanılmaktadır. XLib'in daha modern XRC isimli bir versiyonu da vardır.

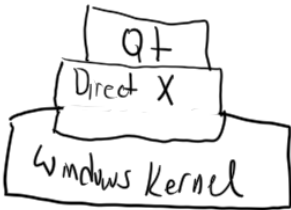
Tıpkı Microsoft'un Direct-X alt kütüphanesi gibi UNIX/Linux dünyasında da daha hızlı grafik işlemler yapmak için "Open GL" denilen bir kütüphane de geliştirilmiştir. Open GL belli bir zamandan sonra "cross platform" olmuştur. Bu kütüphane Windows sistemlerinde de kullanılabilir. Bazı "cross platform" GUI kütüphaneleri arka planda Open GL'den faydalanmaktadır.

Machintosh sistemleri 10 versiyonuyla birlikte (Mac OS X'i kastediyoruz) UNIX türevi bir çekirdeğe geçmiştir. Mac OS X'in kernel kodları açıktır ve buna Darwin denilmektedir. (Darwin geniş ölçüde Free BSD ve Mach çekirdeğinin kodlarını bulunduruyor.) Mac OS X sistemlerinde grafik işlemler Cocoa denilen bir kütüphaneyle yürütülmektedir. Bu kütüphane Objective-C ve Swift dillerinden doğrudan kullanılabilir. Cocoa'nın C'den kullanımı için Carbon isminde bir API grubu bulunmaktadır. Özetle bugünkü Mac sistemlerinde grafik işlemler çekirdeğin üzerine kurulmuş olan ve ismine "Cocoa" denilen bir katman kullanılarak yapılmaktadır.

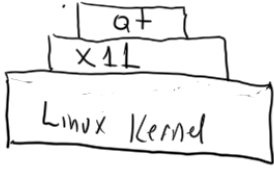
1.6. Qt'nin Cross Platform Özelliği

Cross Platform kavramı "birden fazla işletim sisteminde aynı biçimde kullanılabilirliği" ifade etmektedir. Qt Linux, BSD gibi UNIX türevi sistemlerde, Windows sistemlerinde ve Mac OS X sistemlerinde aynı biçimde kullanılan bir ortamdır. Ancak Qt yalnızca kaynak kod temelinde bir taşınabilirliğe sahiptir, çalıştırılabilen (executable) kod düzeyinde bir taşınabilirliğe sahip değildir. Yani biz bir platformda oluşturduğumuz çalıştırılabilir (executable) bir Qt dosyasını başka bir platforma götürerek çalıştıramayız. Fakat biz bir Qt projesini başka bir platforma götürüp orada derlersek uygulama aynı işlevsellikle çalışacaktır. Bu da geliştiricilerin her platform için başka bir ortam ya da kütüphane kullanma zorunluluğunu ortadan kaldırmaktadır. (Halbuki Java ve .NET ortamları çalıştırılabilen kod düzeyinde taşınabilirliğe sahiptir. Buna İngilizce "binary portability" denilmektedir.

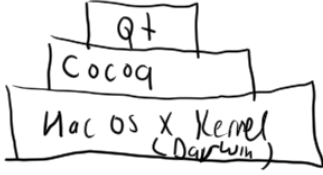
Qt "cross platform"luk özelliğini her sistemde o sisteme özgü altyapıyı kullanarak sağlamaktadır. Örneğin Qt'nin Windows sürümünde Qt Microsoft'un "Direct X" alt yapısını kullanmaktadır.



UNIX/Linux sistemlerinde ise Qt doğrudan X Windows sistemini kullanarak işlevselliğini sağlamaktadır.



Mac OS X sistemlerinde de Qt Cocoa alt yapısını kullanarak yazılmıştır.



1.7. C/C++ Programcıları İçin Qt'nin Alternatifleri

Qt'nin Windows'taki ilk alternatif MFC'dir. MFC Microsoft'un yalnızca Windows sistemleri için (Windows CE de dahil olmak üzere) tasarladığı aşağı seviyeli bir kütüphane ve ortamdır (framework). MFC hızlıdır, yeterlidir, ancak maalesef öğrenilmesi zordur. Üstelik "cross platform" özelliğinin olmaması da bir dezavantajdır. Microsoft uzunca bir süredir artık MFC'ye geliştirme anlamında bir yatırım yapmamaktadır.

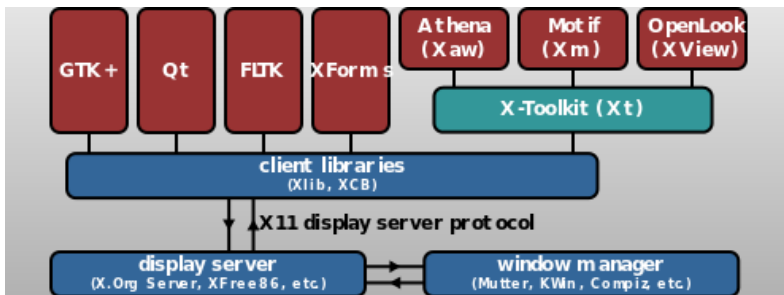
Windows için diğer bir alternatif Borland firmasının C++ Builder kütüphanesi olabilir. Fakat bu platformun da programcıları çok azalmıştır. Ayrıca bu platformun paralı olması da önemli bir dezavantajdır.

Tabii Windows'ta doğrudan USER32.DLL ve GDI32.DLL (buradaki 32 sizi yanılsın 64 bit sistemlerde 64 bit işlevsellik sunmaktadır) içerisindeki API fonksiyonları kullanılarak da pencere programları yazılabilir. Bunun için C++'a da gerek yoktur. Çünkü API fonksiyonları zaten C'de yazılmıştır ve doğrudan C'den çağrılabilir. Ancak C'de API fonksiyonlarıyla çok aşağı seviyeli olarak GUI programların yazılması oldukça zahmetlidir.

Linux sistemlerinde Qt'ye ilk alternatif GTK+ olabilir. GTK+ seviye olarak Windows'un API'lerine benzetilebilir. Yani Windows API programlamadaki uygulama geliştirme zorluğu burada da söz konusudur. GTK+ temelde C'de yazılmış fonksiyonlardan oluşan bir kütüphanedir. Ancak GTK+'ı kullanarak oluşturulmuş nesne yönelimli kütüphaneler de (örneğin Mono) vardır.

Linux sistemlerinde diğer bir alternatif Xt (X Toolkit) denilen kütüphanedir. Bu kütüphane de GTK+ seviyesindedir. Bu kütüphaneyi temel alan başka kütüphaneler (örneğin Motif) ve ortamlar bulunmaktadır.

Tabii Linux sistemlerinde programcı en aşağı seviyeden XLib ya da XRC ile doğrudan X Window işlemleri yapabilir. Ancak X Windows yukarıda da belirtildiği gibi çok aşağı seviyeli. Aşağıdaki şekil seviyeler konusunda bir fikir verebilir:



Mac OS X sistemlerinin GUI doğal ortamı "Cocoa"dır. Cocoa Objective-C ile yazılmıştır ve en doğal olarak Objective-C ile kullanılmaktadır. Objective-C dili C'nin üzerine kurulmuş nesne yönelimli bir dildir. Ancak C++ kadar ayrıntılı ve karışık değildir. Fakat şimdilerde Apple Objective-C dilini bırakıp onun yerine Swift isimli daha basit ve daha modern bir dili kullanma kararı vermiştir. Bu nedenle artık yeni Apple programcıları Objective-C yerine Swift öğrenmek istemektedirler.

Yukarıdakiler dışında C/C++ programcıları için aslında sözünü etmediğimiz pek çok GUI kütüphanesi vardır. Ancak bunlar çok sınırlı kullanılmaktadır.

1.8. Qt Ortamının Kurulumu

Qt'nin Windows'taki kurulumu oldukça basittir. Kursun verildiği tarihte Digia'nın temel indirme (download) sayfasının adresi şöyledir:

<https://www.qt.io/download-open-source/#section-2>

Buradaki seçeneklerden biri "online install"dur. Qt Windows altında Microsoft derleyicisiyle (cl.exe) ya da gcc derleyicisiyle (gcc'nin Windows portuna MinGW denilmektedir) kullanılabilir. "Offline install" için ise iki ayrı seçenek bulunmaktadır. Qt-Creator IDE'si ayrıca kurulabilir ya da yukarıdaki "offline install" paketlerinin içerisinde zaten bulunmaktadır. Windows'ta ayrıca Qt Visual Studio'ya entegre edilerek de kullanılabilir. Bunun için "Visual Studio Add-In" paketi indirilerek kurulmalıdır. Ancak Qt'nin Visual Studio Add-In paketlerinde bazı sorunlar oluşabilmektedir. Ayrıca bu Add-In paketleri Visual Studio versiyonlarıyla sıkı sıkıya bağlıdır.

Linux'ta Qt kurulumu oldukça kolaydır. Zaten pek çok dağıtımın yazılım yöneticisinde (örneğin Ubuntu, Mint gibi) Qt bir seçenek olarak indirilip kurulabilmektedir. Tabii kurulum yine yukarıdaki Web sayfasından ilgili paketler indirilerek de yapılabilir.

Mac OS X sistemlerinde kurulum da oldukça kolaydır. Yukarıdaki linkten "OS X Host" seçeneğiyle belirtilen paketler indirilerek kurulum yapılır.

Windows'ta Qt kurulumunu yaptıktan sonra komut satırından çalışabilmek için MinGW ya da Visual Studio IDE'lerinin "path" ayarlarının yapılması gerekir. Örneğin Windows'ta Qt'yi MinGW ile kullanmak isteyelim. Bunun için önce "qmake.exe" programının bulunduğu dizini belirlemeliyiz. (Örneğin kursun yürütüldüğü makinede qmake'in bulunduğu dizin "C:\Qt\Qt5.6.0\5.6\mingw49_32\bin" biçimindedir.) Bundan sonra benzer biçimde GNU make'in Windows versiyonu olan "mingw32-make.exe" için de yol ifadesi belirlenmelidir. (Örneğin bu yol ifadesi de kursun yürütüldüğü makinede "C:\Qt\Qt5.6.0\Tools\mingw492_32\bin" biçimindedir.) Nihayet bu iki dizin "Gelişmiş Sistem Ayarları/Ortam Değişkenleri/Sistem Değişkenleri/Path" çevre değişkenine eklenir. Artık komut satırına geçildiğinde oradan hem "qmake" hem de "mingw-make" programları kullanılabilir.

1.9. Build Araçlarına Kısa Bir Bakış

Yazılım projeleri çok fazla kaynak dosyadan ve yardımcı dosyalardan oluşabilmektedir. Projelerdeki bu çok sayıda kaynak dosyanın derlenerek çalıştırılabilir (executable) hale getirilmesi süreci yorucu olabilmektedir. Örneğin bir projede 10 kaynak dosya ile çalışıyor olalım. C++'ta önce bunların hepsinin derlenmesi ve sonra da birlikte bağlanması (link edilmesi) gerekir. Daha sonra bu dosyalardan yalnızca birinde değişiklik yaptığımızda yalnızca onu derlememiz fakat hep birlikte yine bağlama (link) işlemi yapmamız uygun olur. Ayrıca bazı projeler hedef olarak yalnızca tek bir çalıştırılabilir dosya da üretmeyebilirler. Örneğin bazı projelerde birkaç dinamik kütüphanenin ve birkaç da çalıştırılabilir dosyanın üretilmesi hedeflenebilmektedir. İşte bu sıkıcı işlemleri otomatize etmek için "build araçlarından (build tools)" faydalanılmaktadır. Bugün IDE'lerin hemen hepsi zaten kendi içerisinde bu tür "build" araçlarını barındırdıkları için IDE'lerde çalışırken programcılar bu araçların hiç farkında olmayabiliyorlar.

Build araçlarının en ünlüsü ve klasığı şüphesiz "make"tir. "Make" aracı ilk kez UNIX sistemlerinde geliştirilmiştir. Daha sonra GNU projesi kapsamında modernize edilmiştir. GNU'nun make aracına "GNU make" denilmektedir. Bugün UNIX/Linux dünyasından ağırlıklı olarak "GNU make" aracı kullanılmaktadır. Microsoft DOS zamanından başlayarak make'in çok benzeyen bir biçimini de kullanmıştı. Microsoft'un make programına "nmake" denilmektedir. Microsoft pek çok dili kapsayan daha genel amaçlı bir "build aracı" da geliştirmiştir. Visual Studio arka planda bunu kullanmaktadır. Bu araca "MSBuild" denilmektedir.

"Make" aracının kullanılmasını öğrenmek belli bir zaman alabilmektedir. Bu nedenle arka planda "make" aracını kullanan daha yüksek seviyeli araçlar da gerçekleştirilmiştir. Örneğin "cmake" daha basit bir kullanıma sahiptir. "cmake" bize make dosyası üretir, sonra o dosya yeniden make işlemine sokulmaktadır.

Troll Tech firması ilk kez Qt’yi çıkarttığında Qt’ye özgü bir build aracı da geliştirmişti. Bugün hala Qt’de “qmake” denilen bu özel araç kullanılmaktadır.

C/C++ dünyasının dışında da başka dillerden kullanılan pek çok "build aracı" vardır. Örneğin "An" Java dünyasında en çok tercih edilenlerden biridir.

1.9.1. QMake ile Çalışmak

"Qmake" Qt için tasarlanmış yüksek seviyeli bir "build" aracıdır. Qmake ile çalışırken programcı önce “.pro” uzantılı bir dosyaya (aslında bu dosyanın uzantısı ".pro" olmak zorunda değildir) proje içeriğini yazar. Sonra bu .pro dosyası “qmake” isimli programa sokulur. qmake bu ".pro" dosyasını alarak bize bir "make dosyası" (Microsoft derleyici sisteminde çalışıyorsa nmake dosyası) oluşturur. Sonra da biz bu "make" dosyasını girdi yaparak “make” programını çalıştırırız. "Make" programı da o dosyadaki yönergeleri izleyerek derleme ve bağlama işlemlerini yapıp çalıştırılabilen dosyayı oluşturacaktır. Aslında pek çok durumda bizim bu ".pro" dosyasını elle yazmamıza gerek bile kalmamaktadır. qmake programı proje dizinine geçilip “-project” seçeneği ile çalıştırıldığında dizini tarayarak bizim için “.pro” dosyasını da oluşturmaktadır. Örneğin:

```
qmake - project
```

Qt Creator IDE’sinde çalışırken bizim ".pro" dosyasını kendimizin oluşturmaya gerek yoktur. Zaten görsel olarak biz Qt Creator’da projeye dosyaları ekledikçe IDE arka planda “.pro” dosyasına gerekli eklemeleri yapmaktadır. Biz QtCreator’da build işlemi yaptığımızda Qt Creator yine yukarıdaki adımları uygulayarak önce "qmake" programını sonra da "make" programını çalıştırarak çalıştırılabilen dosyası oluşturmaktadır.

1.10. C++’ın Qt Ortamında Kullanılması

Qt’deki doğal çalışmada her sınıf iki kaynak dosya olarak organize edilmelidir. Gerçekten de Qt’nin kendi kaynak dosyalarında da bu yöntem izlenmiştir. Örneğin Sample isimli bir sınıf bildirecek olalım. Tipik organizasyon şöyle olmalıdır:

```
/* sample.hpp */

#ifndef SAMPLE_HPP_
#define SAMPLE_HPP_

class Sample {
public:
    Sample();
    //...
};

#endif

/* mainwindow.cpp */

#include "Sample.hpp"

Sample::Sample()
{
    //...
}
//...
```

Qt’nin kendi sınıfları Pascal yazım tarzıyla harflendirilmiştir. (Yani yalnızca tüm sözcüklerin ilk harfleri büyük). Fakat sınıfların elemanlarının harflendirilmesi “deve notasyonu”yla (camel casting) yapılmıştır. Deve notasyonunda ilk sözcüğün tamamı küçük harflerle sonraki sözcüklerin yalnızca ilk harfleri büyük harflerle isimlendirildiğini biliyorsunuz. Örneğin:

```
setTestCount  
numberOfStudents  
addItem
```

gibi.

Genel olarak Qt'de sınıfların "get" üye fonksiyonları (getter) veri elemanı xxx biçiminde, "set" üye fonksiyonları (setter) ise setXxx biçiminde isimlendirilmektedir. Örneğin:

```
result = x.number(); // getter  
x.setNumber(10);     // setter
```

Qt kütüphanesinde bulunan sınıfların hepsi 'Q' harfiyle başlayarak isimlendirilmiştir. (Örneğin QWidget, QPushButton, QApplication gibi.) Ancak programcı kendi sınıflarını 'Q' ile başlayarak isimlendirmemelidir. Böylece hangi sınıfların Qt'nin orijinal sınıfları olduğu hangisinin uygulama programcılar tarafından oluşturulduğu daha iyi anlaşılabilir. (Örneğin MFC'de de aynı yöntem kullanılmıştır. MFC'nin tüm sınıfları 'C' harfiyle başlayarak isimlendirilmiştir. Örneğin CWnd, CList gibi.)

Qt'ye yeni başlayanların dikkatini çekecek olan bir nokta da C++'ın standart kütüphanesinde bulunan bazı genel amaçlı "yararlı (utility)" sınıfların bir benzerinin Qt'de de bulunuyor olmasıdır. İyi de neden? Öncelikle Qt (MFC için de aynı şey söylenebilir) tarihsel olarak C++ standartlarından önce oluşturulmuştur. Yani bazı genel amaçlı Qt sınıfları C++'ta bunların eşdeğer standart karşılıkları daha yokken oluşturulmuştur. İkincisi bu genel amaçlı Qt sınıfları Qt kütüphanesinin bazı kısımlarıyla oldukça uyumludur. Dolayısıyla Qt'de çalışırken genel amaçlı birtakım işlemler için C++'ın standart kütüphanesinde bulunan sınıflar yerine Qt kütüphanesinde bulunan benzerlerini tercih etmeliyiz. Fakat buradan Qt'de C++'ın standart kütüphanesinin hiç kullanılmaması gerektiği sonucunu da çıkarmamalısınız.

Qt'nin kütüphanesindeki sınıfların bildiriminin bulunduğu başlık dosyalarına genel olarak sınıf isimleriyle aynı isimler verilmiştir. Troll Tech eskiden bunlara ".h" uzantısı vermişti. Ancak Qt-4 ile birlikte artık bunlardaki uzantılar kaldırılmıştır. Bu durumda örneğin "QApplication" isminde bir sınıf kullanacaksak onun bildiriminin bulunduğu başlık dosyasını aşağıdaki gibi include etmeliyiz:

```
#include <QApplication>
```

include işlemini genel olarak açısız parantezler içerisinde yapabilirsiniz. Çünkü proje "build" edilirken derleyiciye komut satırı argümanlarıyla bu dosyaların bulunduğu dizine de bakması gerektiği söylenmektedir.

2. Qt'de GUI Programlamanın Temelleri

Bu bölümde Qt ile GUI programlama uygulamaları hakkında temel bilgiler açıklanacaktır.

2.1. GUI Ortamlarında Mesaj Tabanlı Çalışma Modeli

Mesaj tabanlı programlama modelinde klavye ve fare gibi aygıtlarda oluşan girdileri programcı kendisi almaya çalışmaz. Fare gibi, klavye gibi girdi aygıtlarını işletim sisteminin (ya da GUI alt sistemin) kendisi izler. Oluşan girdi olayı hangi pencereye ilişkinse işletim sistemi ya da GUI alt sistem, bu girdi olayını "mesaj" adı altında bir yapıya dönüştürerek o pencerenin ilişkin olduğu (yani o pencereyi yaratan) programın "mesaj kuyruğu (message queue)" denilen bir kuyruk sistemine yerleştirir. Mesaj kuyruğu içerisinde mesajların bulunduğu FIFO prensibiyle çalışan bir kuyruk veri yapısıdır. Sistemin daha iyi anlaşılması için süreci maddeler halinde özetlemek istiyoruz:

1. Her programın (ya da thread'in) "mesaj kuyruğu" denilen bir kuyruk veri yapısı vardır. Mesaj kuyruğu mesajlardan oluşmaktadır.

2. İşletim sistemi ya da GUI alt sistem gerçekleşen girdi olaylarını "mesaj (message)" adı altında bir yapı formatına dönüştürmekte ve bunu pencerenin ilişkin olduğu programın (ya da thread'in) mesaj kuyruğuna eklemektedir.

3. Mesajlar ilgili olayı betimleyen ve ona ilişkin bazı bilgileri barındıran yapı (structure) nesleridir. Örneğin Windows'ta mesajlar MSG isimli bir yapıyla temsil edilmişlerdir. Bu yapının elemanlarında mesajın ne mesajı olduğu (yani neden gönderildiği) ve olaya ilişkin bazı bilgiler bulunur.

Görüldüğü gibi GUI programlama modelinde girdileri programcı elde etmeye çalışmamaktadır. Girdileri bizzat işletim sisteminin kendisi ya da GUI alt sistemi elde edip programcıya mesaj adı altında iletmektedir.

GUI programlama modelinde işletim sisteminin (ya da GUI alt sistemin) oluşan mesajı ilgili programın (ya da thread'in) mesaj kuyruğuna eklemenin dışında başka bir sorumluluğu yoktur. Mesajların kuyruktan alınarak işlenmesi ilgili programın sorumluluğundadır. Böylece GUI programcısının mesaj kuyruğuna bakarak sıradaki mesajı alması ve ne olmuşsa ona uygun işlemleri yapması gerekir. Bu modelde programcı kodunu şöyle düzenler: Bir döngü içerisinde sıradaki mesajı kuyruktan al, onun neden gönderildiğini belirle, uygun işlemleri yap, kuyrukta mesaj yoksa da bloke de bekle". İşte GUI programlarındaki mesaj kuyruğundan mesajı alıp işleyen döngüye mesaj döngüsü (message loop) denilmektedir.

Bir GUI programının işleyişini tipik akışı aşağıdaki gibi bir kodla temsil edebiliriz:

```
int main()
{
    <ana pencereyi yarat>
    for (;;) {
        <sıradaki mesajı al>
        <mesajı işle>
        <X tuşuna basılırsa
            döngüde gik>
    }
    return 0;
}
```

mesaj döngüsü

Bu temsili koddan da görüldüğü gibi tipik bir GUI programında programcı bir döngü içerisinde mesaj kuyruğundan sıradaki mesajı alır ve onu işler. Mesajın işlenmesi ise "ne olmuş ve ben buna karşı ne yapmalıyım?" biçiminde oluşturulmuş olan kodlarla yapılmaktadır.

Peki bir GUI programı nasıl sonlanmaktadır? İşte pencerenin sağındaki (bazı sistemlerde solundaki) X simgesine kullanıcı tıkladığında işletim sistemi ya da GUI alt sistem bunu da bir mesaj olarak o pencerenin ilişkin olduğu prosesin (ya da thread'in) mesaj kuyruğuna bırakır. Programcı da kuyruktan bu mesajı alarak mesaj döngüsünden çıkar ve program sonlanır.

GUI ortamımız ister .NET, ister Java, ister MFC olsun, isterse Qt olsun, işletim sisteminin ya da GUI alt sistemin çalışması hep burada ele açıklandığı gibidir. Yani örneğin biz .NET'te ya da Java'da işlemlerin sanki başka biçimlerde yapıldığını sanabiliriz. Aslında işlemler bu ortamlar tarafından aşağı seviyede yine burada anlatıldığı gibi yapılmaktadır. Bu ortamlar (frameworks) ya da kütüphaneler çeşitli yükleri üzerimizden alarak bize daha rahat bir çalışma modeli sunarlar. Ayrıca şunu da belirtmek istiyoruz: GUI programlama modeli özellikle nesne yönelimli programlama modeline çok uygun düşmektedir. Bu nedenle bu konuda kullanılan kütüphanelerin büyük bölümü sınıflar biçiminde nesne yönelimli diller için oluşturulmuş durumdadır.

Şimdi GUI programlama modelindeki mesaj kavramını biraz daha açalım. Yukarıda da belirttiğimiz gibi bu modelde programcıyı ilgilendiren çeşitli olaylara “mesaj” denilmektedir. Örneğin klavyeden bir tuşa basılması, pencere üzerinde fare ile tıklanması, pencere içerisinde farenin hareket ettirilmesi gibi olaylar hep birer mesaj oluşturmaktadır. İşletim sistemleri ya da GUI alt sistemler mesajları birbirinden ayırmak için onlara birer numara karşılık getirirler. Örneğin Windows’ta mesaj numaraları WM_XXX biçiminde sembolik sabitlerle kodlanmıştır. Programcılar da konuşurken ya da kod yazarken mesaj numaralarını değil, bu sembolik sabitleri kullanırlar. (Örneğin WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_KEYDOWN gibi) Mesajların numaraları yalnızca gerçekleşen olayın türünü belirtmektedir. Oysa bazı olaylarda gerçekleşen olaya ilişkin bazı bilgiler de söz konusudur. İşte bir mesaja ilişkin o mesaja özgü bazı parametrik bilgiler de işletim sistemi ya da GUI alt sistem tarafından mesajın bir parçası olarak mesajın içerisine kodlanmaktadır. Örneğin Windows’ta biz klavyeden bir tuşa bastığımızda Windows WM_KEYDOWN isimli mesajı programın mesaj kuyruğuna bırakır. Bu mesajı kuyruktan alan programcı mesaj numarasına bakarak klavyenin bir tuşuna basılmış olduğunu anlar. Fakat hangi tuşa basılmıştır? İşte Windows basılan tuşun bilgisini de ayrıca bu mesajın içerisine kodlamaktadır. Örneğin WM_LBUTTONDOWN mesajını Windows farenin sol tuşuna tıkladığında kuyruğa bırakır. Ancak ayrıca basım koordinatını da mesaja ekler. Yani bir mesaj oluştuğunda yalnızca o mesajın hangi tür bir olay yüzünden oluştuğu bilgisini değil aynı zamanda o olayla ilgili bazı bilgileri de kuyruktaki mesajın içerisinden alabilmekteyiz.

GUI programlama modelinde bir mesaj oluştuğunda o mesajın bir an evvel işlenmesi ve akışın çok bekletilmemesi gerekir. Aksi takdirde programcı kuyruktaki diğer mesajları işleyemez bu da “program donmuş etkisi” yaratmaktadır. Eğer bir mesaj alındığında uzun süren bir işlem yapılmak isteniyorsa bir thread oluşturulup o işi o thread’e devretmek ve böylece mesaj döngüsünün işlenmesini sağlamak gerekir.

GUI programlama modellerinde genel olarak mesaj kavramı pencere kavramıyla ilişkilendirilmiştir. Yani bir pencere yaratıldıktan sonra bir mesajın oluşma durumu da yoktur. Bu nedenle mesaj döngüsüne girmeden önce programcının en az bir pencere (tipik olarak programın ana penceresi) yaratmış olması gerekir.

Windows gibi bazı sistemlerde thread’lerle ilişkilendirilmiştir. Bu sistemlerde prosesin tek bir mesaj kuyruğu yoktur. Her thread’in ayrı bir mesaj kuyruğu vardır. Bu durumda işletim sistemi ya da GUI alt sistem bir pencereye ilişkin bir işlem gerçekleştiğinde o pencerenin hangi prosesin hangi thread’i tarafından yaratılmış olduğunu belirler ve mesajı o thread’in mesaj kuyruğuna bırakır. Böylece biz bir thread oluşturup o thread’te de bir pencere yaratmışsak artık bizim de o thread’te o pencerenin mesajlarını işlemek için mesaj döngüsü oluşturmamız gerekir. Tabii eğer thread’imizde biz hiçbir pencere oluşturmamışsak böyle bir mesaj döngüsünü oluşturmamıza da gerek yoktur. (Örneğin Microsoft eğer bir thread bir pencere yaratmışsa böyle thread’lere “GUI thread’ler” yaratmamışsa “worker thread’ler” demektedir).

2.2. Windows Sistemlerinde İskelet GUI Programı

Eğer biz Windows’ta Qt gibi bir ortam kullanmıyor olsaydık ekrana bir ana pencere çıkartan programı API düzeyinde C ile yazmak zorunda kalırdık. Bu Windows’taki en aşağı seviyeli GUI programlama biçimidir. Biz de gelişkin bir kütüphane ya da framework kullanmadan bir GUI programının ne kadar zor yazılabildiğini göstermek için aşağıdaki iskelet Windows programını vermek istiyoruz:

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    WNDCLASSEX wcex;
    MSG msg;

    wcex.cbSize = sizeof(WNDCLASSEX);
    wcex.style = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc = WndProc;
    wcex.cbClsExtra = 0;
    wcex.cbWndExtra = 0;
    wcex.hInstance = hInstance;
    wcex.hIcon = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_APPLICATION));
```

```

wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
wcex.lpszMenuName = NULL;
wcex.lpszClassName = "MyClass";
wcex.hIconSm = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_APPLICATION));

if (!RegisterClassEx(&wcex))
    exit(EXIT_FAILURE);

HWND hWnd = CreateWindow(
    "MyClass",
    "Generic GUI",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT,
    800, 600,
    NULL,
    NULL,
    hInstance,
    NULL
);

if (!hWnd)
    exit(EXIT_FAILURE);

ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

while (GetMessage(&msg, NULL, 0, 0)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message) {
        case WM_CLOSE:
            DestroyWindow(hWnd);
            break;
        case WM_DESTROY:
            PostQuitMessage(0);
            break;
        default:
            return DefWindowProc(hWnd, message, wParam, lParam);
            break;
    }

    return 0;
}

```

Programcı gözüyle bir GUI programı nasıl yazılır? Programcının mesaj döngüsü içerisinde sıradaki mesajı alması, o mesajın neden gönderildiğini (yani hangi olayın gerçekleştiğini) tespit etmesi ve uygun işlemi yapması gerekir. Yani programcı programını “ne olmuş ve ben buna karşı ne yapmalıyım?” biçiminde oluşturmalıdır.

GUI programlama modellerinde genel olarak mesaj kavramı pencere kavramıyla ilişkilendirilmiştir. Yani bir pencere yaratılmadıktan sonra bir mesajın oluşma durumu da yoktur. Bu nedenle mesaj döngüsüne girmeden önce hiç olmazsa programın ana penceresinin yaratılmış olması gerekir.

2.3. XWindow (X11) Sistemlerinde İskelet GUI Programı

XWindow (X11) sistemleri client-server biçimde çalışmaktadır. Burada biz bu sistemlerde boş bir pencere çıkartan örnek bir program vereceğiz. Amacımız yine gelişkin bir kütüphane ya da framework kullanmadan UNIX/Linux

sistemlerinde basit bir GUI programının bile ne kadar zor yazılabildiğini size göstermek. Örneğimizde XLIB kütüphanesi kullanılıyor. XLIB seviye olarak yukarıda gördüğümüz Windows API sistemine göre biraz daha aşağı seviyeli gibi durmaktadır. XLIB'i kullanan GTK+ kütüphanesinin seviye olarak Windows API fonksiyonlarına daha çok benzediğini söyleyebiliriz. Aşağıda ekrana bir ana pencere çıkartan örnek bir XLIB programı görüyorsunuz:

```
/* xlib-helloworld.c */

#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
    Display *disp;
    Window w;
    XEvent e;
    int scr;

    disp = XOpenDisplay(NULL);
    if (disp == NULL) {
        fprintf(stderr, "Cannot open display\n");
        exit(1);
    }

    scr = DefaultScreen(disp);
    w = XCreateSimpleWindow(disp, RootWindow(disp, scr), 10, 10, 100, 100, 1,
        BlackPixel(disp, scr), WhitePixel(disp, scr));
    XSelectInput(disp, w, ExposureMask | KeyPressMask);
    XMapWindow(disp, w);

    for (;;) {
        XNextEvent(disp, &e);
        if (e.type == KeyPress)
            break;
    }

    XCloseDisplay(disp);

    return 0;
}
```

2.4. Qt'de İskelet GUI Programı

Bu bölümde hiç IDE kullanmadan bir giriş niteliğinde iskelet bir GUI programının nasıl oluşturulacağı ve build edileceği gösterilecektir. İskelet GUI programında yalnızca içi boş bir ana pencere vardır. İskelet GUI programını oluşturmak için sırasıyla şunlar yapılmalıdır:

1) Öncelikle programın ana penceresine ilişkin bir sınıf QWidget isimli bir sınıftan türetilerek bildirilmelidir (tabii bu sınıfın bildirimi yukarıda da belirtildiği gibi iki dosya halinde yapılmalıdır):

```
/* mainwindow.hpp */

#ifndef MAINWINDOW_HPP_
#define MAINWINDOW_HPP_

#include <QWidget>

class MainWindow : public QWidget {
public:
    MainWindow();
};

#endif
```

```
/* mainwindow.cpp */

#include "mainwindow.hpp"

MainWindow::MainWindow() : QWidget(NULL)
{}
```

2) main fonksiyonu ayrı dosyada aşağıdaki gibi bildirilir:

```
/* app.cpp */

#include <QApplication>
#include "mainwindow.hpp"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    MainWindow mainWindow;
    mainWindow.show();

    app.exec();

    return 0;
}
```

3) Sıra qmake dosyasının oluşturulmasına gelmiştir. İskelet GUI programı için ".pro" dosyası şöyle oluşturulabilir:

```
/* genericgui.pro */

QT += widgets

HEADERS += mainwindow.hpp
SOURCES += mainwindow.cpp app.cpp
```

4) Artık komut satırına geçilip bu dosyaların bulunduğu dizinde qmake işlemi aşağıdaki gibi yapılır:

```
qmake genericgui.pro
```

Bu işlem sonucunda "Makefile" isimli bir make dosyası üretilcektir. Mmake programlarına komut satırı argümanı verilmezse bu programlar default olarak "Makefile" isimli dosyayı aradığını anımsayınız.

5) Artık make işlemi Windows'ta MinGW paketi kullanılıyorsa aşağıdaki gibi:

```
mingw32-make
```

Windows'ta Visual Studio paketi kullanılıyorsa aşağıdaki gibi:

```
nmake
```

Ve UNIX/Linux sistemlerinde de aşağıdaki gibi yapılabilir:

```
make
```

6) Artık çalıştırılabilen dosya oluşmuştur. Onu herhangi bir yolla çalıştırabiliriz.

Tabii her türlü yeniden derlemede bu işlemlerin hepsinin yeniden yapılmasına gerek yoktur. Örneğin eğer ".pro" dosyasında bir değişiklik söz konusu değilse "qmake" işleminin yapılması gerekmez. Doğrudan "make" işlemi yapılabilir. Eğer karışıklık olduğu düşünülüyorsa projedeki "amaç dosyaları (object files)" aşağıdaki silip sanki hiç daha önce make yapılmamış gibi işlemlere devam edilebilir:

```
mingw32-make clean
nmake clean
make clean
```

2.5. Qt'deki İskelet GUI Programının Açıklaması

Qt'deki iskelet GUI programın main fonksiyonu şöyledir:

```
#include <QApplication>
#include "mainwindow.hpp"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    MainWindow mainWindow;
    mainWindow.show();

    app.exec();

    return 0;
}
```

Qt'de her olgu bir sınıfla temsil edilmiştir. Örneğin QApplication sınıfı uygulamanın kendisini temsil etmektedir. exec fonksiyonu QApplication sınıfının static fonksiyonudur. (Dolayısıyla bu fonksiyon QApplication::exec() biçiminde de çağrılabilirdi.) Programın mesaj döngüsü exec fonksiyonunun içerisinde oluşturulmuştur. Programın ana penceresi kapatılınca exec fonksiyonu geri döner. Böylece main fonksiyonu sonlanır, program da bitmiş olur.

Qt'de pencere kavramı “widget (window gadget)” terimiyle temsil edilmektedir. İskelet programda programın ana penceresi için QWidget sınıfından faydalanılmıştır. Fakat programda birtakım eklemeler yapmaya olanak sağlamak için QWidget sınıfını doğrudan değil türeterek kullandık:

```
/* mainwindow.hpp */

#ifndef MAINWINDOW_HPP_
#define MAINWINDOW_HPP_

#include <QWidget>

class MainWindow : public QWidget {
public:
    MainWindow();
};

#endif

/* mainwindow.cpp */

#include "mainwindow.hpp"

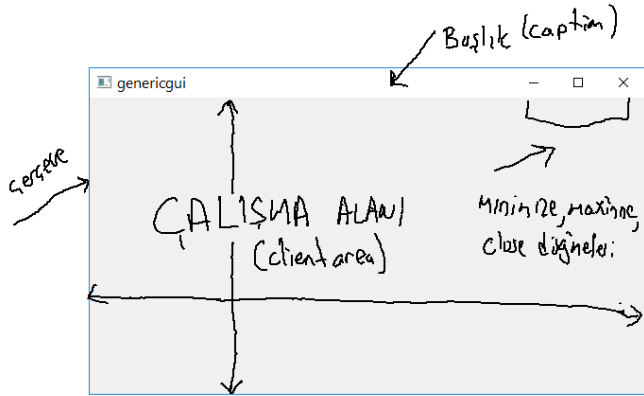
MainWindow::MainWindow() : QWidget(NULL)
{
}
```

QWidget sınıfı türünden bir nesne yaratıldığında bir pencere de yaratılmış olur. Ancak henüz görünür değildir. QWidget sınıfının show üye fonksiyonu pencereyi görünür hale getirmektedir. Ayrıca pencerenin mesaj döngüsüne girmeden önce ana pencerenin yaratıldığına dikkat ediniz.

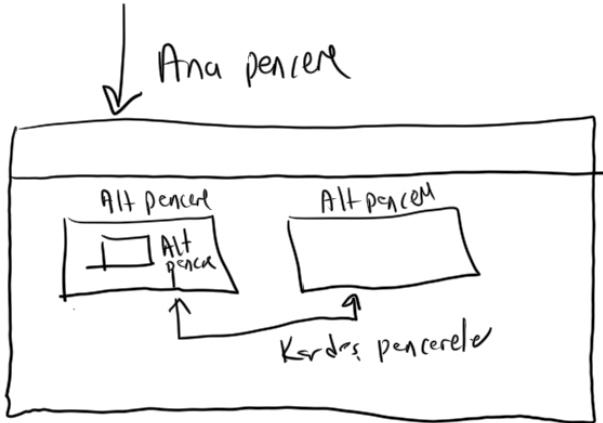
2.6. GUI Sistemlerinde Pencere Terminolojisi

GUI tabanlı sistemlerde ekranda bağımsız olarak kontrol edilebilen dikdörtgensel bölgelere pencere (window) denilmektedir. Konumlarına ve işlevselliklerine göre pencereler birkaç gruba ayrılmaktadır. Doğrudan masaüstüne açılan pencerelere “ana pencereler (top level windows)” denilmektedir. Pek çok GUI alt sisteminde görünür durumda olan ana pencereler bir çubukta gösterilmektedir (örneğin Windows’taki task bar). Her ne kadar zorunlu değilse de ana pencerelerin genellikle bir çerçevesi (frame), bir başlık kısmı (caption) ve başlık kısmının üzerinde bazı simgeleri bulunur.

Pencerenin sınır çizgilerinin ve başlığının altında kalan alana “çalışma alanı (client area)” denilmektedir. Çalışma alanı bizim aktif çizim yapabileceğimiz, başka alt pencereleri yerleştirebileceğimiz (yani bizim için ayrılmış) olan alandır. Tabii bir pencerenin toplam genişliği ve yüksekliği çalışma alanının genişliği ve yüksekliği ile aynı olmak zorunda değildir. Eğer pencerenin başlığı ve sınır çizgileri yoksa bu durumda pencerenin kendi genişliği ve yüksekliği çalışma alanının genişliği ve yüksekliği ile aynı olur.



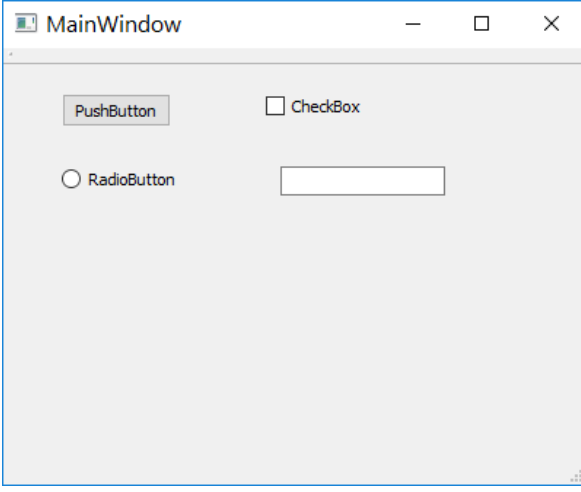
Bir pencerenin içerisinde görüntülenen, onun dışına çıkamayan pencerelere “alt pencereler (child windows)” denilmektedir. Alt pencerelerde genellikle başlık kısmının ve sınır çizgilerinin bulunması istenmez. (Ancak tabii alt pencereler de istenirse başlık kısmına ve sınır çizgilerine sahip olabilirler.) Alt pencerelerin alt pencereleri de söz konusu olabilir. Her alt pencerenin bir üst penceresi (parent window) vardır. Aynı üst pencereye sahip olan pencerelere kardeş pencereler (sibling windows) denilmektedir.



Ayrıca bir de ismine “sahiplenilmiş (owned)” pencere denilen bir pencere türü daha vardır. Tipik olarak diyalog pencereleri bu türdendir. Sahiplenilmiş pencereler hem bir çeşit alt pencere gibi hem de ana pencere gibi davranırlar. Bunlar her zaman üst pencerelerinin yukarısında görüntülenir. Bunların üst pencereleri minimize edildiğinde bunlar da görünmez olurlar.

2.6.1 GUI Elemanları ve Alt Pencereler

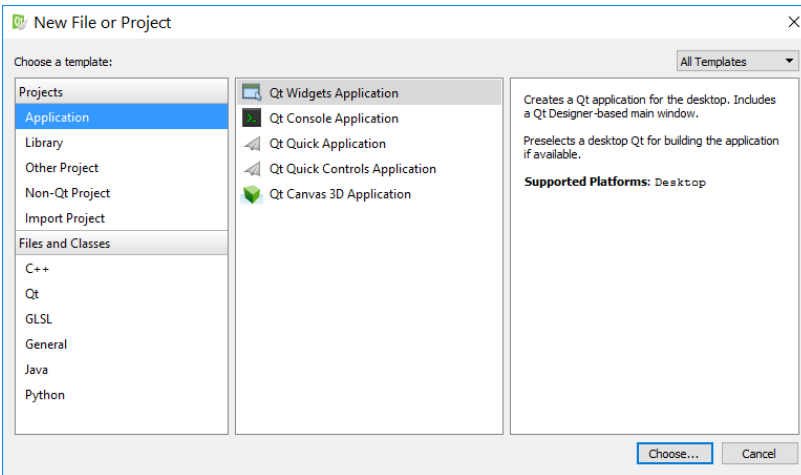
Programların GUI arayüzlerinde kullanılan düğmeler, edit alanları, listeleme kutuları, menüler vs. hep aslında birer alt penceredir. Örneğin:



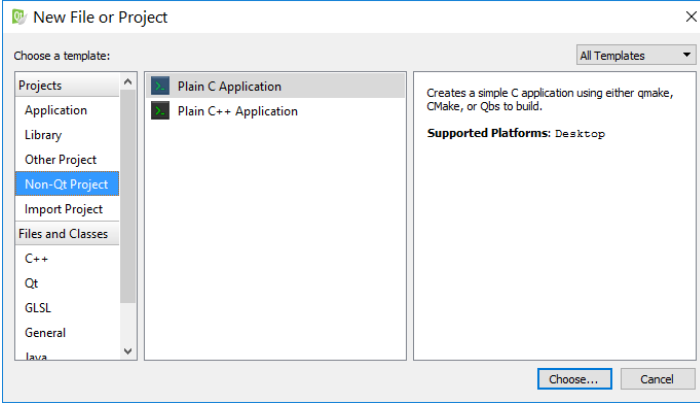
Aslında yukarıdaki ana pencerede gördüğünüz düğme gibi, seçenek kutusu gibi, radyo düğmesi gibi görsel öğeler içi boş pencereler üzerinde çizim işlemleri yapılarak oluşturulmuşlardır. Biz de sıfırdan içi boş bir pencereden hareketle kendi görsel öğelerimizi oluşturabiliriz. Ancak Qt gibi ortamlarda çeşitli görsel elemanlar alt pencere biçiminde zaten önceden oluşturulmuş durumdadır. Bu görsel öğeler birer sınıfla temsil edilmiştir. Böylece programcı hangi görsel öğeyi kullanacaksa o sınıf türünden bir nesne yaratır, sonra o nesnenin üye fonksiyonlarını çağırarak nesnenin tam olarak istediği biçimde gözükmelerini sağlar. GUI ortamlarında kullanıcı arayüzleri hep böyle oluşturulmaktadır. Ancak bir noktaya dikkat çekmek istiyoruz: Her ne kadar Qt’de -diğer ortamlarda olduğu gibi- pek çok görsel öğe alt pencere biçiminde hazır bulundurulmuş olsa da bunlar programcının isteklerini tam olarak karşılamıyor olabilir. Bu durumda programcı arzu ettiği görsel öğeleri sıfırdan içi boş bir alt pencerelerden hareketle oluşturmak ya da başkaları tarafından oluşturulmuş olanları satın alıp kullanmak isteyebilir. Neyse ki gereksinimlerin büyük çoğunluğu mevcut alt pencere sınıflarıyla karşılanabilmektedir.

2.7. Qt Creator IDE’sinin Kısa Tanıtımı

Diğer IDE’lerde olduğu gibi Qt Creator’da da çalışmalar proje temelinde (yani proje oluşturularak) gerçekleştirilmektedir. Bir proje uygulamayı temsil eden varlıklar toplamıdır. Dosya sisteminde içerisinde çeşitli dosyaların bulunduğu klasörler ile kendisini gösterir. Qt Creator IDE’si de diğer IDE’lerde olduğu gibi çeşitli şablon proje seçeneklerine sahiptir. Bir proje yaratmak için “File/New File or Project” seçilir. Karşımıza aşağıdaki gibi bir diyalog penceresi çıkacaktır:



Proje şablonları bazı iskelet kodları bizim için oluşturmaktadır. Tabii programcı hiçbir şablon kullanmadan tamamen boş bir proje de oluşturabilir. Ayrıca Qt Creator IDE’sinde hiç Qt’ye bulaşmadan düz C/C++ programları da yazabiliriz:



Qt Creator IDE'si (default durumda) proje dosyası olarak doğrudan "qmake" için oluşturulmuş ".pro" dosyasını kullanır. Yani biz bir projeyi açacaksak ".pro" dosyasını IDE'de yüklemeliyiz.

Programın "build" edilmesi ve çalıştırılması için (tabii IDE arka planda qmake ve make işlemleri yapmaktadır) Ctrl + R tuşu kısayol tuşu ya da ilgili araççubuğu elamanı kullanılmaktadır.

2.8. Qt'nin Yardımcı Sınıflarına Genel Bir Bakış

Her ne kadar bşalngıçta Qt bir GUI kütüphanesi olarak düşünölmüşse de zaman içerisinde pek çok yapılmış ve genel amaçlı bir ortam haline getirilmiştir. Qt'deki her sınıf ve fonksiyon GUI uygulamalarına yönelik değildir. Qt'de pek çok uygulamada kullanılabilecek çok sayıda genel amaçlı sınıf da bulunmaktadır. Genel amaçlı sınıfların bazıları hem GUI uygulamalarında hem de başka uygulamalarda dolaylı olarak kullanılabilir. Biz de bu bölümde bazı genel amaçlı sınıfları çok derine girmeden ele alacağız.

Daha önceden de belirttiğimiz gibi Qt'deki sınıfların bildirimleri genellikle aynı isimli başlık dosyalarında bulundurulmuştur. Şüphesiz bazı başlık dosyaları başka başlık dosyaları tarafından dolaylı olarak include edilmiş olabilir. Örneğin QString sınıfı QChar sınıfını kullandığı için <QString> başlık dosyasının içerisinde <QChar> başlık dosyasının include edilmiş olması çok muhtemeldir. Ancak programcı birbirlerini kullanan sınıflar arasındaki bu dolaylı ilişkiyi bilmek zorunda değildir. Dolayısıyla kullandığı her sınıf için onun başlık dosyasını açıkça include etmesi uygunb olur.

Qt kütüphanesinde bazı alt sistemlerin kullandığı tüm başlık dosyalarını include etmiş olan bazı include dosyaları da vardır. Örneğin <QtCore> pek çok temel başlık dosyasını include eden bir include dosyasıdır. <QtGui> GUI uygulamaları için gereken başlık dosyalarını include eden başlık dosyasıdır. Pek çok dosyayı tek tek include etmek yerine yalnızca bu dosyaları include etmek pratik bir yöntem olabilir. Ancak bu yöntemin derleme zamanınının uzamasına yol açabileceğini de belirtelim.

Anahtar Notlar: Qt Creator IDE'sinde Console tabanlı deneme projesi oluşturmak için (QT'nin temel sınıfları böyle bir projeyle denenebilir) şu işlemler yapılabilir:

- 1) File/New File or Project menüsü seçilir.
- 2) Uygulama türü olarak Qt Console Application seçilir.
- 3) Sonra main fonksiyonun içi tamamen silinebilir.

2.8.1. QChar ve QString Sınıfları

QChar tek bir karakteri temsil eden bir sınıftır. Bir QChar nesnesi bir UNICODE olarak tutar ve onu bize istediğimiz karakter kodlamasıyla (encoding) ile verebilir. QChar sınıfının başlangıç fonksiyonlarında (constructors) nesnenin tutacağı bir karakter verilebilir. Örneğin:

```
QChar ch('x');
```

```
QDebug() << ch;
```


Anahtar Notlar: qDebug sınıfı herhangi bir nesnenin içerisindeki bilgileri ekrana (stderr dosyasına) basmak için kullanılmaktadır. Bu fonksiyon ilerde ele alınmaktadır.

QChar sınıfının karakter test işlemi yapan pek çok static olmayan üye fonksiyonu vardır. (Örneğin isPunct, isNumber, isPrint gibi.) Yani <ctype.h> başlık dosyasında bulunan karakter test fonksiyonlarının bir benzeri bu sınıfın üye fonksiyonları biçiminde bulunmaktadır. Örneğin:

```
QChar ch('9');

QDebug() << ch.isNumber (); // true
```

Yine QChar sınıfının toUpper ve toLower üye fonksiyonları büyük harf küçük harf dönüştürmesini yapmaktadır. Örneğin:

```
QChar ch1('a');
QChar ch2;

ch2 = ch1.toUpper();
QDebug() << ch2;
```

QChar sınıfının karşılaştırma işlemleri yapan operatör fonksiyonları vardır. Örneğin:

```
QChar ch1('a');
QChar ch2('c');

if (ch1 > ch2)
    qDebug() << ch1 << " > " << ch2;
else if (ch1 < ch2)
    qDebug() << ch1 << " < " << ch2;
else
    qDebug() << ch1 << " == " << ch2;
```

QString sınıfı C++'ın standart string sınıfının pek çok işlevselliğine sahiptir. QString sınıfı karakterleri QChar dizisi gibi tutmaktadır. QString sınıfı tıpkı C++'ın string sınıfı gibi kapasite kullanmaktadır. Yani tıpkı vector sınıfında olduğu gibi QString sınıfı da gerektiğinde daha büyük alanı tahsis ederek yeniden tahsisat miktarını azaltmaya çalışmaktadır. QString sınıfının yazıyı çeşitli parametrelerle oluşturan başlangıç fonksiyonları (constructors) vardır:

```
QString();
QString(const QChar *unicode, int size = -1);
QString(QChar ch);
QString(int size, QChar ch);
QString(QLatin1String str);
QString(const QString &other);
QString(QString &&other);
QString(const char *str);
QString(const QByteArray &ba);
```

Örneğin:

```
#include <QString>
#include <QDebug>

int main()
{
    QString str1("This is a test");
    qDebug() << str1;                // "this is a test"

    QString str2(10, QChar('a'));
    qDebug() << str2;                // "aaaaaaaaaa"
```

```

    return 0;
}

```

QString sınıfının append ve prepend isimli üye fonksiyonları yazının sonuna ve başına ekleme yapmak için kullanılır. Bu fonksiyonların pek çok overload edilmiş biçimi vardır. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str("is a");

    str.prepend("This ");
    str.append(" test");

    qDebug() << str;           // "This is a test"

    return 0;
}

```

Sınıfın isEmpty üye fonksiyonu yazının boş olup olmadığını (yani yazı içerisinde karakter olup olmadığını), isNull fonksiyonu da nesnenin default başlangıç fonksiyonuyla yaratılıp yaratılmadığını (yani tamamen boş olup olmadığını) belirtir. (Aslında QString yazının sonunda '\0' karakteri yerleştirmektedir. Fakat default başlangıç fonksiyonuyla nesne yaratıldığında gerçekten yazının içerisinde '\0' bile yoktur). Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str1;
    QString str2("");
    QString str3("Ankara");

    qDebug() << str1.isNull();           // true
    qDebug() << str1.isEmpty();          // true

    qDebug() << str2.isNull();           // false
    qDebug() << str2.isEmpty();          // true

    qDebug() << str3.isNull();           // false
    qDebug() << str3.isEmpty();          // false

    return 0;
}

```

Sınıfın insert üye fonksiyonları indeks numarası olarak yazının arasına ekleme yapmak için kullanılmaktadır. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str = "Anra";

    str.insert(2, "ka");
    qDebug() << str;           // "Ankara"

    return 0;
}

```

Örneğin:

```
#include <QString>
#include <QDebug>

int main()
{
    QString str = "Anra";

    str.insert(2, 'k');
    str.insert(3, 'a');

    qDebug() << str;          // "Ankara"

    return 0;
}
```

QString sınıfının remove üye fonksiyonları yazıdan karakter silmek için kullanılır. Örneğin:

```
#include <QString>
#include <QDebug>

int main()
{
    QString str("Bugün hava çok güzel");

    str.remove(10, 4);
    qDebug() << str;          // "Bugün hava güzel"

    return 0;
}
```

Sınıfın replace üye fonksiyonları yazı içerisindeki bir karakter grubunu başka bir karakter grubuyla değiştirmektedir. Örneğin:

```
#include <QString>
#include <QDebug>

int main()
{
    QString str("İstanbul, Bursa, Ankara, Bursa");

    str.replace("Bursa", "Eskişehir");
    qDebug() << str;          // İstanbul, Eskişehir, Ankara, Eskişehir

    return 0;
}
```

Sınıfın indexOf üye fonksiyonları yazı içerisinde bir yazı ya da karakter ararlar. Eğer bulurlarsa buldukları yerin yazı içerisindeki indeks numarasıyla bulamazlarsa -1 değeri ile geri dönerler. indexOf fonksiyonların pek çok overload biçimleri vardır. Örneğin:

```
#include <QString>
#include <QDebug>

int main()
{
    QString str = "Bugün hava çok güzel";
    int index;

    index = str.indexOf("hava");
}
```

```

if (index == -1)
    qDebug() << "cannot find!";
else
    qDebug() << "Found at index " << index;

index = str.indexOf('a');
if (index == -1)
    qDebug() << "cannot find!";
else
    qDebug() << "Found at index " << index;

return 0;
}

```

indexOf fonksiyonları yazı içerisinde aranan kısmın ilk bulunduğu yerin indeks numarasını bize vermektedir. Bu fonksiyonların ayrıca lastIndexOf isminde son bulunan yerin indeks numarasını veren biçimleri de vardır.

Nesne yönelimli kütüphanelerin çoğunda yazının belli bir kısmını elde fonksiyonlar bulunmaktadır. Bunlar genellikle SubString biçiminde isimlendirilmiştir. Ancak Qt'nin QString sınıfında bu işi QString'te mid isimli üye fonksiyonları yapmaktadır. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str = "Bugün hava çok güzel";
    QString result;

    result = str.mid(6, 4);
    qDebug() << result;        // "hava"

    result = str.mid(6);
    qDebug() << result;        // "hava çok güzel"

    return 0;
}

```

mid fonksiyonunun ikinci parametresi default argüman almıştır. Bu parametre ihmal edilirse “geri kalan tüm karakterler” elde edilir.

QString sınıfının number isimli static fonksiyonları parametre olarak aldıkları sayısal değerlerden QString nesnesi oluşturmaktadır. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str;

    str = QString::number(1234);
    qDebug() << str;

    str = QString::number(1234.56);
    qDebug() << str;

    return 0;
}

```

Bu işlemin tam tersi static olmayan toXXX (toInt, toLong, ToFloat, toDouble gibi) üye fonksiyonlarıyla yapılmaktadır. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str("123.45");
    double result;

    result = str.toDouble();
    qDebug() << result;

    return 0;
}

```

QString nesnesinden C++'ın std::string nesnesi de elde edilebilir. Bunun için sınıfın toString üye fonksiyonu kullanılır. Örneğin:

```

#include <iostream>
#include <string>

#include <QString>
#include <QDebug>

int main()
{
    QString str("this is a test");
    std::string cppstr;

    cppstr = str.toString();
    std::cout << cppstr << std::endl;

    return 0;
}

```

Sınıfın toUpper ve toLower isimli static olmayan üye fonksiyonları bize büyük harfe ya da küçük harfe dönüştürülmüş yazı verir. Örnek:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str("ankara");
    QString result;

    result = str.toUpper();

    qDebug() << result;    // ANKARA

    return 0;
}

```

QString sınıfının <, >, <=, >=, == ve != olmak üzere altı karşılaştırma operatör fonksiyonu da vardır. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str1("ankara");
    QString str2("Ankara");
}

```

```

if (str1 > str2)
    qDebug() << str1 << " > " << str2;
else if (str1 < str2)
    qDebug() << str1 << " < " << str2;
else if (str1 == str2) // kasten yerleştirildi
    qDebug() << str1 << " == " << str2;

return 0;
}

```

Sınıfın + operatör fonksiyonu iki QString nesnesi içerisindeki yazıları yeni bir QString nesnesinde uç uca ekler ve o nesneyle geri döner. Benzer biçimde sınıfın += operatör fonksiyonu da vardır. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str1("ankara");
    QString str2("istanbul");
    QString result;

    result = str1 + str2;
    qDebug() << result; // "ankaraistanbul"

    result += QString("samsun");
    qDebug() << result; // "ankaraistanbulsamsun"

    result += "eskişehir";
    qDebug() << result; // "ankaraistanbulsamsuneskişehir"

    return 0;
}

```

QString sınıfının split fonksiyonları yazıyı belirlediğimiz karakterleri dikkate alarak parse eder. Fakat bunu bize QStringList türünden nesne tutan bir sınıf (container class) biçiminde verir. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str = "ankara, izmir, bursa, adana";
    QStringList strList;

    strList = str.split(", ");

    for (int i = 0; i < strList.size(); ++i)
        qDebug() << strList[i];

    return 0;
}

```

QString sınıfının arg isimli üye fonksiyonları yazı içerisindeki “%n” kalıplarını inceler. Bu “%n” kalıpları yer tutucudur. En düşük n’in bulunduğu “%n” kalıbı yerine argümanları ile belirtilen değeri yerleştirirler ve bu değer yerleştirilmiş yeni yazıyı yine QString olarak geri döndürürler. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    int a = 10, b = 20;

```

```

QString str1("a = %1, b = %2");
QString str2;
QString str3;

QDebug() << str1;           // "a = %1, b = %2"

str2 = str1.arg(a);
QDebug() << str2;           // "a = 10, b = %2"

str3 = str2.arg(b);
QDebug() << str3;           // "a = 10, b = 20"

return 0;
}

```

Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    int a = 10, b = 20;

    QString str1("b = %2, a = %1");
    QString str2;
    QString str3;

    qDebug() << str1;           // "b = %2, a = %1"

    str2 = str1.arg(a);
    qDebug() << str2;           // "b = %2, a = 10"

    str3 = str2.arg(b);
    qDebug() << str3;           // "b = 20, a = 10"

    return 0;
}

```

Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    int a = 10, b = 20;

    qDebug() << QString("a = %1, b = %2").arg(a).arg(b);

    return 0;
}

```

Eğer en düşük numaralı tutucu birden fazla kez yinelenirse arg onların hepsini değiştirir. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    int a = 10;

```

```

    qDebug() << QString("%1, %1").arg(a);    // "10, 10"

    return 0;
}

```

Klavyeden QString okumak için doğrudan bir fonksiyon yoktur. Bunun dolaylı birkaç yolu olabilir. Biz burada QTextStream kullanarak yazı okuyan bir örnek verelim:

```

#include <QString>
#include <QDebug>
#include <QTextStream>

int main()
{
    QTextStream qts(stdin);
    QString str;

    str = qts.readLine();
    qDebug() << str;

    return 0;
}

```

QTextStream sınıfı ileride ele alınacaktır.

QString sınıfının “capacity” ve “size” üye fonksiyonları kapasite ve size değerlerini almak için “reserve” ve “resize” üye fonksiyonları ise kapasite ve size değerini değiştirmek için kullanılabilir. Örneğin:

```

#include <QString>
#include <QDebug>

int main()
{
    QString str("this is a test");

    qDebug() << str.size();
    qDebug() << str.capacity();

    str.reserve(20);
    str.resize(6);

    qDebug() << str.size();
    qDebug() << str.capacity();

    return 0;
}

```

QString sınıfının başka faydalı üye fonksiyonları da vardır. Bunlar Qt dokümanlarından incelenebilir. Ancak kursumuzda gerektiğinde bu fonksiyonlar sanki görülmüş gibi kullanılabilir.

2.8.2. QDebug Sınıfı

Bu sınıf ekrana ya da başka bir aygıta debug amaçlı bilgiler yazmak için kullanılmaktadır. Aslında yukarıdaki örneklerde zaten QDebug sınıfını dolaylı olarak kullandık. Qt içerisindeki pek çok sınıfın QDebug parametrelili bir << operatör fonksiyonu vardır. Böylece çeşitli sınıfların içerisindeki bilgiler teşhis amaçlı QDebug sınıfı yoluyla görüntülenebilir. QDebug sınıfını işlevsel olarak Java ve C#’teki toString metotlarına benzetebiliriz. Örneğin Qt’nin QXXX biçiminde bir sınıfı olsun. Bu sınıfa ilişkin aşağıdaki gibi global << operatör fonksiyonu vardır:

```
QDebug & operator <<(const QDebug &, const QXXX &);
```

Böylece biz örneğin bir QPoint sınıf nesnesini QDebug sınıfı yardımıyla ekrana (stderr dosyasına) yazdırabiliriz:


```
QPoint pt(5, 3);

QDebug() << pt;
```

Tabii aynı şeyi cout ile yapamazdık:

```
cout << pt; // error! ostream sınıfının böyle bir operatör fonksiyonu yok
```

QDebug sınıfının başlangıç fonksiyonu parametre olarak bizden bir dosyanın yol ifadesini ya da QIODevice nesnesini ister. (QFile sınıfı da QIODevice sınıfından türeilmiştir.) QDebug bilgileri bu dosyaya yazdırmaktadır. Örneğin:

```
#include <iostream>
#include <cstdlib>
#include <QDebug>
#include <QFile>

using namespace std;

int main()
{
    QFile file("out.txt");

    if (!file.open(QIODevice::WriteOnly | QIODevice::Text)) {
        cout << "cannot open file" << endl;
        exit(EXIT_FAILURE);
    }

    QDebug debug(&file);

    debug << "this is a test" << 123;

    return 0;
}
```

Kütüphane içerisinde qDebug isimli global bir fonksiyon vardır. Bu fonksiyon QDebug sınıfı türünden bir nesneyi bize verir. qDebug fonksiyonunun geri dönüş değerine ilişkin QDebug nesnesi hedef olarak stderr dosyasına yazma yapmaktadır.

QDebug sınıfının << operatör fonksiyonları normal olarak kombine işleminde araya boşluk karakteri yerleştirmektedir. Ancak sınıfın nospace üye fonksiyonu bu boşlukların verilmesini engeller. Örneğin:

```
#include <QDebug>

int main()
{
    int a = 10, b = 20;

    qDebug() << a << b;           // "10 20"
    qDebug().nospace() << a << b; // "1020"

    return 0;
}
```

Sınıfın maybeSpace üye fonksiyonu yazdırma işleminin başına bir SPACE boşluk bırakır, maybeQuote ise bir tane “ işareti çıkartmaktadır. quote fonksiyonu QChar, QString, QByteArray sınıflarını yazdırırken bunları iki tırnak içerisinde yazdırır. Default durumda zaten yazdırma böyledir. Bu istenmiyorsa noquote fonksiyonu çağırılmalıdır.

2.8.3. QVector Sınıfı

QVector sınıfı adeta C++'ın standart std::vector sınıfının Qt karşılığı gibidir. Bu da std::vector sınıfı gibi şablon (template) bir sınıftır. Sınıfın şablon parametresi dinamik dizi içerisinde tutulacak türü alır.

QVector sınıfı dinamik büyütülen bir diziyi temsil eder. Bu sınıfta da size ve capacity kavramları std::vector sınıfıyla aynıdır. QVector nesnesi default başlangıç fonksiyonuyla ya da belli bir size değeri ile de yaratılabilir. Eleman ekleme işlemi için append fonksiyonları kullanılmaktadır. Örneğin:

```
#include <QDebug>
#include <QVector>

int main()
{
    QVector<int> vect;

    for (int i = 0; i < 10; ++i)
        vect.append(i);

    qDebug() << vect;

    return 0;
}
```

QVector nesnesinin elemanlarına at fonksiyonu ya da [] operatör fonksiyonu ile erişilebilir:

```
#include <QDebug>
#include <QVector>

int main()
{
    QVector<int> vect;

    for (int i = 0; i < 10; ++i)
        vect.append(i);

    for (int i = 0; i < vect.size(); ++i)
        qDebug() << vect[i];

    return 0;
}
```

at ya da [] operatör fonksiyonları ile vektörün sınırı dışına erişim yapılmak istenirse exception oluşur.

Sınıfın clear üye fonksiyonu vektör içerisindeki tüm elemanları siler. empty fonksiyonu vektörün içinin boş olup olmadığını bize verir. erase ile elemanlar silinebilir. first fonksiyonu ve front fonksiyonu ilk elemanı, last fonksiyonu da son elemanı vermektedir. indexOf ve lastIndexOf fonksiyonlarıyla vektör içerisinde arama yapılabilir. removeAt belli bir indeksteki elemanı silmek için kullanılır.

mid fonksiyonu vektörün belli bir parçasını bize yeni bir QVector olarak verir. Örneğin:

```
#include <QDebug>
#include <QVector>

int main()
{
    QVector<int> vect, result;

    for (int i = 0; i < 10; ++i)
        vect.append(i);

    result = vect.mid(3, 4);
}
```

```

    qDebug() << result;        // QVector(3, 4, 5, 6)

    return 0;
}

```

QVector sınıfının T şablon parametrelili << operatör fonksiyonları da eleman eklemesi için kullanılabilir. Örneğin:

```

#include <QDebug>
#include <QVector>

int main()
{
    QVector<int> vect;

    vect << 10 << 20 << 30 << 40 << 50;

    qDebug() << vect;        // QVector(10, 20, 30, 40, 50)

    return 0;
}

```

QVector C++'ın standart kütüphanesindeki “random access iterator” gereksinimleri karşılamaktadır. Yani biz QVector sınıfını C++'ın vector sınıfı gibi de kullanabiliriz. Böylece QVector C++'ın standart kütüphanesindeki algoritmalarla da çalışabilmektedir. Örneğin:

```

#include <iostream>
#include <QDebug>
#include <QVector>

using namespace std;

int main()
{
    QVector<int> vect;

    vect << 10 << 20 << 30 << 40 << 50;
    for (QVector<int>::iterator iter = vect.begin(); iter != vect.end(); ++iter)
        cout << *iter << " ";
    cout << endl;

    return 0;
}

```

C++11 ve C++14'ün bazı özelliklerini de Qt'de kullanabiliriz. Örneğin Qt'nin nesne tutan (container) sınıfları iterator uyumlu olduğu için bunlar aralık tabanlı for döngüleriyle kullanılabilir:

```

#include <iostream>
#include <QDebug>
#include <QVector>

using namespace std;

int main()
{
    QVector<int> vect;

    vect << 10 << 20 << 30 << 40 << 50;

    for (int x : vect)
        cout << x << " ";
    cout << endl;
}

```

```

    return 0;
}

```

Aslında Qt'nin kendisinde de başından beri bir foreach makrosu vardır.

```

#include <iostream>
#include <QDebug>
#include <QVector>

using namespace std;

int main()
{
    QVector<int> vect;
    int x;

    vect << 10 << 20 << 30 << 40 << 50;

    foreach (x, vect)
        cout << x << " ";
    cout << endl;

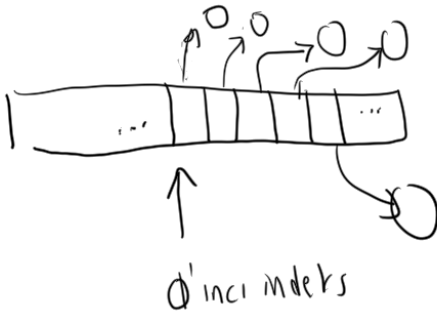
    return 0;
}

```

Qt'nin foreach makrosu iki parametreye sahiptir. Birinci parametre container'ın elemanını tutacak nesneyi alır. İkinci parametre ise container nesnesinin kendisini almaktadır. Yine bu makro her yinelemede container'ın sıradaki elemanını birinci parametreyle belirtilen nesnenin içerisine yerleştirir. Tabii artık bunun yerine C++11 ile gelen aralık tabanlı for döngüleri tercih edilebilir.

2.8.4. QList Sınıfı

QList sınıfı C++'ın standart kütüphanesindeki std::deque sınıfına benzemektedir. QList nesnesinin başına ve sonuna eleman ekleme sabit zamanlı ($O(1)$) karmaşıklığa sahiptir. Ancak QList std::deque sınıfından farklı olarak elemanların kendisini değil, bunların adreslerini tutmaktadır. Yani sınıfın ekleme yapan üye fonksiyonları önce heap üzerinde tahsisat yapıp nesneyi orada oluştururlar. Sonra onun adresini dinamik büyütülen çift yönlü dizide saklarlar.



QList sınıfı da şablon bir sınıftır. Sınıfın şablon parametresi tutulacak türü belirtir. Aslında kullanım QVector sınıfına oldukça benzemektedir. Örneğin:

```

#include <QDebug>
#include <QList>

int main()
{
    QList<int> list;

    list << 10 << 20 << 30 << 40 << 50;
}

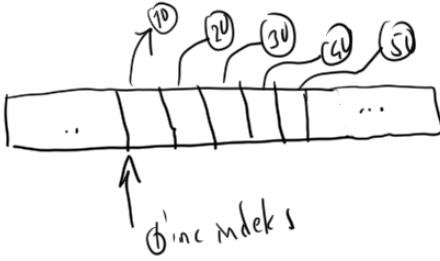
```

```

qDebug() << list;          // (10, 20, 30, 40, 50)

return 0;
}

```



QList sınıfının genel kullanımı QVector sınıfına çok benzemektedir. Yine eleman eklemek için “append” ve “prepend” fonksiyonları ya da “<< operatör fonksiyonları” kullanılır. Yine QList de C++’ın “random access iterator” özelliklerini sağlamaktadır. Eleman silme ve insert işlemleri benzer biçimde yapılmaktadır. Sınıfın “toStdList” ve “toVector” fonksiyonları bize container içerisindeki değerleri C++’ın “std::list” ve “std::vector” nesnesi biçiminde verir. Yine at fonksiyonu ve [] operatör fonksiyonu eleman erişimi için kullanılabilir.

QList sınıfında adres türünden açılımları için özelleştirme yapılmıştır. Biz QList içerisinde zaten adres tutacaksa artık bu adresler için de heap’te tahsisat yapılamaz. Doğrudan adreslerin kendileri tutulmaktadır. QList sınıfında programcının ayarlayabileceği bir kapasite değeri yoktur. (Tabii sınıf kendi içerisinde bir kapasite kullanmaktadır.)

2.8.5. QLinkedList Sınıfı

Bu sınıf C++’ın “std::list” sınıfına benzemektedir. QLinkedList sınıfı elemanları çift bağlı liste içerisinde tutar. Dolayısıyla çift yönlü (bidirectional) iteratör desteğine sahiptir. Elemana erişim doğrusal karmaşıklıkta, başa ve sona ekleme işlemleri sabit karmaşıklıkta. Tabii eklenecek yerin iteratör pozisyonu bilindiğinde araya ekleme çok hızlı yani sabit karmaşıklıkta yapılabilir.

```

#include <QDebug>
#include <QLinkedList>

int main()
{
    QLinkedList<int> llist;

    for (int i = 0; i < 10; ++i)
        llist.append(i);

    for (QLinkedList<int>::iterator iter = llist.begin();
         iter != llist.end(); ++iter)

        qDebug() << *iter;

    return 0;
}

```

QLinkedList sınıfının pek çok elemanı QVector ve QList sınıfına benzerdir. Örneğin yine append işlemi << operatör fonksiyonuyla yapılabilir. front ve back fonksiyonları baştaki ve sondaki elemanları alıp değiştirmek için kullanılabilir. Araya ekleme ve aradan silme yapmak için insert ve erase fonksiyonları vardır. Ancak bunlar indeks ile değil iteratör ile çalışmaktadır. Insert ve erase için iteratörler eleman eklenirken alınıp saklanabilir. Ya da arama yönetimiyle bu iteratörler elde edilebilir. Tabii eleman insert etme ve silme işlemi bağlı listelerde sabit zamanlı yani çok hızlıdır. Böylece araya eleman ekleme ve aradan eleman silme gibi işlemlerin çok yoğun yapıldığı durumlarda QLinkedList tercih edilebilir. Yine iki QLinkedList nesnesi toplanabilir.

Sınıfın diğer fonksiyonları ilgili dokümanlardan incelenebilir.

2.8.6. QSet ve QMap Sınıfları

Bu sınıflar C++'ın standart “std::set” ve “std::map” sınıflarına çok benzemektedir. “QSet” tek parametrelili bir sınıf şablonudur. Halbuki “QMap” iki şablon parametresine sahiptir. Bu iki sınıf da arka planda elemanları dengelenmiş ikili ağaçlarda tutar. Anahtar verildiğinde değeri hızlı bir biçimde elde etmek amacıyla kullanılmaktadır. QSet sınıfında anahtar ve değer bileşik olarak tek bir türün içerisinde bulunmaktadır. QMap’te ise bunlar ayrı ayrı verilmektedir. Eleman ekleme işlemi için insert fonksiyonları ya da [] operatör fonksiyonları kullanılabilir. Örneğin:

```
#include <QDebug>
#include <QString>
#include <QMap>

int main()
{
    QMap<QString, int> map;

    map.insert("Ali Serçe", 123);
    map.insert("Veli Şahin", 456);
    map.insert("Kaan Aslan", 876);

    qDebug() << map;

    return 0;
}
```

Tabii QMap ve QSet iterator yoluyla dolaşıldığında anahtara göre sıralı bir dizilim elde edilir. QMap içerisindeki bütün anahtarlar “keys” isimli fonksiyonla, bütün değerler “values” isimli fonksiyonla elde edilebilmektedir. Bu fonksiyonlar QList nesnesi geri döndürmektedir. Örneğin:

```
#include <QDebug>
#include <QString>
#include <QMap>

int main()
{
    QMap<QString, int> map;

    map.insert("Ali Serçe", 123);
    map.insert("Veli Şahin", 456);
    map.insert("Kaan Aslan", 876);

    int value;
    foreach (value, map.values())
        qDebug() << value;

    return 0;
}
```

Yine QSet ve QMap sınıflarının “toStdSet” ve “toStdMap” isimli üye fonksiyonları vardır.

QMap nesnesine aynı anahtarı birden fazla kez ekleyebiliriz (yani “std::multimap” gibi). Ancak ekleme için insert yerine “insertMulti” fonksiyonu kullanılır.

Qt’nin eskiden beri kendi iterator sınıfları vardır. Örneğin “Qlist” için “QlistIterator”, “Qmap” için “QmapIterator” gibi. İşte QMap nesnesini de tersten dolaşmak için “QmapIterator” sınıfından faydalanmak gerekir. Örneğin:

```
#include <QDebug>
#include <QString>
```

```

#include <QMap>
#include <QMapIterator>

int main()
{
    QMap<QString, int> map;

    map.insert("Ali Serçe", 123);
    map.insert("Veli Şahin", 456);
    map.insert("Kaan Aslan", 876);

    QMapIterator<QString, int> mapiter(map);

    mapiter.toBack();

    while (mapiter.hasPrevious()) {
        mapiter.previous();
        qDebug() << mapiter.key() << "," << mapiter.value();
    }

    return 0;
}

```

Anahtar Notlar: Qt’de daha pek çok nesne tutan sınıf vardır. Biz burada çok temel birkaçını üstün körü inceledik. Diğer sınıflar hakkında yeri geldikçe açıklamalar yapılacaktır.

2.8.7. QPoint Sınıfı

QPoint sınıfı GUI uygulamalarında ekranda bir noktayı temsil etmek için kullanılmaktadır. Sınıfın başlangıç fonksiyonu bizden noktanın x ve y değerlerini alır. Biz onları istersek “x” ve “y” fonksiyonlarıyla elde edebiliriz. Örnein:

```

#include <QDebug>
#include <QPoint>

int main()
{
    QPoint pt(10, 20);

    qDebug() << pt;
    qDebug() << pt.x() << "," << pt.y();

    return 0;
}

```

Sınıfın +, - gibi operatör fonksiyonları ve işlemli atama operatörleri de bulunmaktadır. Örneğin:

```

#include <QDebug>
#include <QPoint>

int main()
{
    QPoint pt1(10, 20), pt2(3, -4);

    pt1 += pt2;
    qDebug() << pt1;

    return 0;
}

```

2.8.8. QSize Sınıfı

QSize sınıfı genişlik-yükseklik kavramını temsil eden bir sınıftır. Sınıfın başlangıç fonksiyonu bizden genişlik ve yükseklik değerini alır. Bu değerler “width” ve “height” fonksiyonlarıyla geri alınabilir, “setWidth” ve “setHeight” fonksiyonları ile yeniden set edilebilirler. QSize sınıfının da +, - ve işlemli atama operatör fonksiyonları vardır. Örneğin:

```
#include <QDebug>
#include <QSize>

int main()
{
    QSize sz(10, 30);

    qDebug() << sz;
    qDebug() << sz.width() << ", " << sz.height();

    return 0;
}
```

2.8.9. QRect Sınıfı

QRect sınıfı dikdörtgensel bir alanı temsil etmekte kullanılır. Sınıf bizden dikdörtgen koordinatlarını üç biçimde ister:

```
QRect(const QPoint &topLeft, const QPoint &bottomRight);
QRect(const QPoint &topLeft, const QSize &size);
QRect(int x, int y, int width, int height);
```

Örneğin:

```
#include <QDebug>
#include <QRect>

int main()
{
    QRect rect(QPoint(10, 20), QSize(50, 50));

    qDebug() << rect;

    return 0;
}
```

Sınıfın “contains” isimli üye fonksiyonları bir noktanın ya da bir dikdörtgenin bu dikdörtgenin içinde olup olmadığını test etmek için kullanılır. Sınıfın “top”, “bottom”, “left” ve “right” fonksiyonları dikdörtgenin köşe koordinatlarının x ve y değerlerini bize verir. Ayrıca “x” ve “y” fonksiyonları da tamamen “left” ve “top” fonksiyonlarıyla aynı işlemi yapmaktadır. Sınıfın “width” ve “height” fonksiyonları da bize dikdörtgenin genişlik, yükseklik değerlerini vermektedir. Dikdörtgeni ötelemek için “moveRight”, “moveLeft”, “moveTop”, “moveBottom” ve “moveTo”, ve “translate” gibi fonksiyonlar da vardır. Örneğin:

```
#include <QDebug>
#include <QRect>

int main()
{
    QRect rect(QPoint(10, 20), QSize(50, 50));
    qDebug() << rect;           // QRect(10,20 50x50)

    rect.translate(5, 5);
    qDebug() << rect;           // QRect(15,25 50x50)

    rect.moveLeft(40);
    qDebug() << rect;           // QRect(40,20 50x50)
```



```
    return 0;
}
```

Sınıfın + operatör fonksiyonu & ve | operatör fonksiyonları, karşılaştırma operatör fonksiyonları vardır. & operatör fonksiyonu kesişim işlemini, | operatör fonksiyonu ise birleşim işlemini yapar. Bu işlemler ayrıca “united” ve “intersects” üye fonksiyonlarıyla da yapılabilmektedir.

QRect sınıfının diğer üye fonksiyonları Qt dokümanlarından incelenebilir.

2.9. İskelet GUI Programının Qt Creator IDE’sinde Sıfırdan Oluşturulması

Qt Creator IDE’si iskelet bir GUI programını bizim için oluşturabilmektedir. Ancak biz bütün bu işlemleri “Empty Project” seçeneğinden sıfırdan da oluşturabiliriz. Bunun için sırasıyla şu adımlar izlenir:

1) Menüden “File/New File or Project” seçilir. Sonra karşımıza gelen diyalog penceresinde template olarak “Other Project/Empty qmake Project” seçilir.

2) Fare ile (kalın fontla gösterilen) proje üzerine gelinerek bağlam menüsünden (sağ tuşa basılarak çıkartılan menüden) “Add New” seçilir. Karşımıza çıkan diyalog penceresinde “C++ Header File” seçilir ve dosyaya “mainwindow.hpp” ismi verilir. İçeriği aşağıdaki gibi oluşturulur:

```
#ifndef MAINWINDOW_HPP_
#define MAINWINDOW_HPP_

#include <QWidget>

class MainWindow : public QWidget {
public:
    MainWindow();
};

#endif
```

3) Fare ile (kalın fontla gösterilen) proje üzerine gelinerek bağlam menüsünden “Add New” seçilir. Karşımıza çıkan diyalog penceresinde “C++ Source File” seçilir ve dosyaya “mainwindow.cpp” ismi verilir. İçeriği aşağıdaki gibi oluşturulur:

```
#include "mainwindow.hpp"

MainWindow::MainWindow() : QWidget(NULL)
{
}
```

Aslında 2 ve 3 numaralı adımlar tek hamlede de gerçekleştirilebilmektedir. Bunun için fare ile (kalın fontla gösterilen) proje üzerine gelinerek bağlam menüsünden “Add New” seçilir. Karşımıza çıkan diyalog penceresinde “C++/C++ class” seçilir. Burada sınıf ismine “MainWindow” taban sınıf ismine de “QWidget” verilir. Bu durumda IDE bizim için zaten QWidget’tan türetilmiş içi boş bir sınıfı oluşturacaktır.

4) Fare ile koyu renkte gösterilen proje üzerine gelinerek bağlam menüsünden “Add New” seçilir. Karşımıza çıkan diyalog penceresinden “C++ Source File” seçilir. Burada dosyaya app.cpp gibi bir isim verilebilir. Dosyanın içerisine aşağıdaki içerik yazılır.

```
#include <QApplication>
#include "mainwindow.hpp"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
```

```
MainWindow mainWindow;  
mainWindow.show();  
  
app.exec();  
  
return 0;  
}
```

5) “.pro” dosyasının başına aşağıdaki satır eklenir:

```
QT += widgets
```

6) Program çalıştırılır.

2.10. İskelet GUI Programının Qt Creator IDE’sinde IDE’nin Kendisi Tarafından Oluşturulması

Aslında iskelet program Qt Creator IDE’sinin kendisi tarafından da oluşturulabilmektedir. Zaten programcılar genellikle projelerini bu biçimde başlatmayı tercih ederler. Bunun için sırasıyla şunlar yapılmalıdır:

1) Menüden “File/New File or Project” seçilir. Karşımıza çıkan diyalog penceresinden “Application/Qt Widget Application” seçilir. Projenin yeri ve ismi girilir.

2) Oluşturulacak sınıf ve dosya isimlerinin görüldüğü diyalog penceresinde “Base Class” olarak “QMainWindow” yerine QWidget seçilir ve “Generate Form” unchecked yapılır.

2.11. Qt’de Alt Pencerelelerin Oluşturulması

Anımsanacağı gibi bir pencerenin içinde yer alan ve onun dışına çıkamayan pencerelere “alt pencereler (child windows)” denilmektedir. İçi tamamen boş, hatta sınır çizgileri bile olmayan alt pencereler oluşturulabilir. Ancak Qt’de (aslında neredeyse tüm framework’lerde) zaten önceden oluşturulmuş ve çeşitli işlevsellikler kazandırılmış alt pencereler vardır. Bu alt pencereler önceki konularda da belirtildiği gibi çeşitli sınıflarla temsil edilmişlerdir. Örneğin “QPushButton” düğme biçiminde, “QLineEdit” edit alanı biçiminde bir alt pencereyi temsil etmektedir. İşte programcılar kendi alt pencerelerini sıfırdan oluşturmak yerine zaten işlevsellik kazandırılmış bu alt pencere sınıflarını kullanarak oluşturmayı tercih ederler.

Qt’deki tüm pencerelere ilişkin ortak özellikler (pencere ister ana pencere olsun isterse alt pencere olsun) “QWidget” isimli sınıfta toplanmıştır. QWidget sınıfındaki elemanlar her pencerenin sahip olduğu ortak özellikleri temsil etmektedir.

<ŞEKİL>

Bir alt pencere oluşturmak için o alt pencere sınıfı türünden bir nesne yaratılır. Ancak bu alt pencerenin hangi pencerenin alt penceresi olduğunu belirtmek için yaratım sırasında (ya da daha sonra) onun üst penceresi de belirtilmelidir. Bir alt pencere nesnesini yaratan programcının yapacağı ilk şey onu konumlandırmak ve istediği gibi konfigüre etmektir. Bu işlemler için o alt pencere sınıfının (ya da onun taban sınıflarının) ilgili üye fonksiyonları kullanılmaktadır.

Üst pencerelerle alt pencereler arasındaki ilişki nesne yönelimli programlama tekniğinde “içerme ilişkisi (composition)” olarak modellenilebilir. Anımsanacağı gibi içerme ilişkisinin iki önemli özelliği vardır:

1) İçeren nesne ile içerilen nesnenin ömürleri aynıdır.

2) İçerilen nesne tek bir nesne tarafından içerilir.

Örneğin “İnsan sınıfı” ile “Karaciğer Sınıfı” arasında içerme ilişkisi vardır. İnsan doğduğunda karaciğeri ile doğar, öldüğünde karaciğeri de ölür (organ bağışısı gibi uç durumları dikkate almayınız). Bir kaaraciğer tek bir insanın karaciğeridir. Benzer biçimde örneğin “Otomobil” sınıfıyla “Motor” sınıfı arasında da içerme ilişkisi vardır. Araba motoruyla birlikte satışa sunulur. Araba pert olunca motoru da pert olmuştur. Ancak “Doktor” sınıfıyla “Hastane” sınıfı arasında, “Oda” ile “Duvar” sınıfı arasında içerme ilişkisi değil “birleşme ilişkisi (aggretaion)” bulunmaktadır.

C++’ta içerme ilişkisi iki biçimde oluşturulabilir:

1) İçeren sınıfın içerilen sınıf türünden bir private veri elemanı vardır:

```
class A {  
    //...  
};
```

```
class B {  
    //...  
private:  
    A m_a;  
};
```

B b;

2) İçeren sınıfın içerilen sınıf türünden gösterici veri elemanı vardır. İçeren sınıfın başlangıç fonksiyonu (constructor) içerisinde bu veri elemanı için new operatörüyle tahsisat yapılır. Bitiş fonksiyonunda da silme işlemi yapılır:

```
class A {  
    //...  
};
```

```
class B {  
public:  
    B()  
    {  
        m_pa = new A();  
        //...  
    }  
    ~B()  
    {  
        delete m_pa;  
    }  
private:  
    A *m_pa;  
};
```

B b;

Görüldüğü gibi ikinci yöntem daha daha gibi görünsede Qt’de daha yaygın olarak tercih edilmektedir. Çünkü bu yöntemde biz daha rahat çalışma olanağına sahip oluruz. (Örneğin C++’ta veri elemanları için hangi başlangıç fonksiyonlarının çağrılacağı “mil sentaksında” belirtilmelidir ve bunlar için başlangıç fonksiyonları sınıf bildirimindeki sıraya göre çağrılmaktadır. Bu ise Qt bağlamında kimi zorluklara yol açabilmektedir. Ayrıca Qt’de üst pencere sınıfı delete edildiğinde onun bitiş fonksiyonları onun alt pencereler nesneleri için delete işlemi yapmaktadır. Böylece ikinci yöntemde biz üst pencere sınıfının bitiş fonksiyonu içerisinde alt pencere nesnelerini delete ile silmek zorunda kalmayız.) İkinci yöntemin önemli bir avantajı da başlık dosyalarında eksik olarak (incomplete) gösterici bildiriminin yapılabilmesine olanak sağlamasıdır. Böylece biz başlık dosyasında o alt pencere sınıfın başlık dosyasını include etmek zorunda kalmayız. Bunun sonucunda da projemiz önışlemci aşamasından daha hızlı geçer.

Birinci yöntem kullanılarak ana pencere içerisinde bir düğme oluşturma örneği şöyle verilebilir:

```

// mainwindow.hpp

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QWidget>
#include <QPushButton>

class MainWindow : public QWidget
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    QPushButton m_pushButtonOk;
};

#endif // MAINWINDOW_H

// mainwindow.cpp

#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QWidget(parent), m_pushButtonOk("Ok", this)
{
}

MainWindow::~MainWindow()
{}

// main.cpp

#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}

```

Şimdi de aynı şeyi ikinci yöntemle yapmaya çalışalım:

```

#ifndef MAINWINDOW_HPP
#define MAINWINDOW_HPP

// mainwindow.hpp

#include <QWidget>

class QPushButton;

class MainWindow : public QWidget
{
    Q_OBJECT

```

```

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    QPushButton *m_pushPushButtonOk;
};

#endif // MAINWINDOW_HPP

// mainwindow.cpp

#include "mainwindow.hpp"
#include <QPushButton>

MainWindow::MainWindow(QWidget *parent)
    : QWidget(parent)
{
    m_pushPushButtonOk = new QPushButton("Ok", this);
}

MainWindow::~MainWindow()
{
    // gerek yok
    // delete m_pushPushButtonOk;
}

// app.cpp

#include "mainwindow.hpp"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}

```

Peki ikinci yöntemde neden “mainwindow.hpp” dosyası içerisinde <QPushButton> başlık dosyasını include etmedik de QPushButton türünden göstericiyi eksik bildirimle bildirdik? Yanıt basit: “mainwindow.hpp” dosyası hem “app.cpp”den hem de “mainwindow.cpp”den include edilmektedir. Eğer biz “mainwindow.cpp” içerisinde <QPushButton> dosyasını include etmiş olsaydık bu durumda bu dosya her iki cpp dosyası derlenirken önileme sokulacaktı. Halbuki bu eksik bildirim sayesinde <QPushButton> dosyası yalnızca “mainwindow.cpp” dosyası derlenirken include işlemine sokulmaktadır. Tabii küçük çaplı projelerde önilemcideki bu zaman kaybı ciddi bir boyutta olmaz. Ancak bazı orta ve büyük çaplı projelerde bu kayıp önemli olabilmektedir.

Burada bir noktayı da vurgulamak istiyoruz: Aslında Qt programcıları çoğu kez görsel arayüzlerdeki alt pencereleri kod yazarak oluşturmamaktadır. Daha sonra da ele alınacağı gibi bu işlemler Qt’de kod yazmadan tamamen görsel düzeyde yapılabilmektedir.

2.12. Qt’de Sinyal (Signal) Slot (Slot) Mekanizması

Qt’deki sinyal slot temel olarak mesaj işlemleri için düşünülmüş bir mekanizmadır. Alt pencere sınıflarını incelemeden önce bu mekanizmanın bilinmesi gerekmektedir. Sinyal kavramı Qt’ye özgü bir kavramdır; bunun UNIX/Linux sistemlerindeki sinyal kavramıyla ya da sinyal işlemedeki sinyal kavramıyla bir ilgisi yoktur.

Sinyal ve slot birer fonksiyondur. Sinyal slot mekanizmasının kullanılabilmesi için bu mekanizmaları kullanacak sınıf bildiriminin başında Q_OBJECT makrosunun bulunması gerekir. (Zaten Qt Creator IDE’sinde doğrudan ya da dolaylı

olarak taban sınıfı QObject olan bir sınıf oluşturulmak istendiğinde IDE otomatik olarak o sınıfın bildirimine bu makroyu eklemektedir.) Ayrıca sınıfın sinyal slot mekanizmasına dahil edilebilmesi için o sınıfın QObject sınıfından türetilmiş olması da gerekmektedir. Örneğin sinyal slot mekanizmasının kullanılabileceği bir sınıf şöyle bildirilebilir:

```
// sample.hpp

#include <QObject>

class Sample : public QObject {
    Q_OBJECT
public:
    //...
};
```

Q_OBJECT makrosu bazı bildirimleri sınıfa dahil eder. Ancak Qt programcısının bu makronun hangi bildirimleri sınıfa dahil ettiğini bilmesine gerek yoktur. Fakat yine de kafanızda bir fikrin oluşması için biz bu makronun içeriğini vermek istiyoruz:

```
#define Q_OBJECT \
public: \
    Q_OBJECT_CHECK \
    QT_WARNING_PUSH \
    Q_OBJECT_NO_OVERRIDE_WARNING \
    static const QMetaObject staticMetaObject; \
    virtual const QMetaObject *metaObject() const; \
    virtual void *qt_metacast(const char *); \
    virtual int qt_metacall(QMetaObject::Call, int, void **); \
    QT_WARNING_POP \
    QT_TR_FUNCTIONS \
private: \
    Q_DECL_HIDDEN_STATIC_METACALL static void qt_static_metacall(QObject *, QMetaObject::Call, int, void **); \
    struct QPrivateSignal {};
```

Peki Q_OBJECT makrosuyla bildirimleri sınıfa eklenen bu fonksiyonların ve static veri elemanlarının tanımlamaları nerededir? İşte bu fonksiyonların tanımlamalarını programcı yapmamaktadır. İsmine “Meta Object Compiler (MOC)” denilen bir program yapmaktadır. “moc” isimli bu program bir başlık dosyasını komut satırı argümanı olarak alır ve çıktı olarak da Q_OBJECT makrosundaki fonksiyonların tanımlamalarını içeren bir C++ kaynak dosyası oluşturur. Sonra da bu dosya diğer modüllerle birlikte “build” işlemine sokulmaktadır. Qt programcısı “moc” programını manuel olarak kendisi çalıştırabilir. Ancak böyle bir gereksinimle çok seyrek karşılaşılmaktadır. Zaten “qmake” programı “.pro” dosyasından ürettiği “make” dosyasının içerisine “moc” programının çalıştırılacağı ve elde edilen dosyanın da bağlama işlemine katılacağı bilgisini yazmaktadır.

Sinyal slot mekanizmasını kullanabilmek için Q_OBJECT makrosunun dışında ayrıca programcının sinyal ve slot fonksiyonlarını sınıf bildirimine eklemesi gerekir. Sinyal fonksiyonunun yalnızca bildirimi sınıfa yerleştirilir. Tanımlaması programcı tarafından yapılmaz. Örneğin:

```
class Sample : public QObject {
    Q_OBJECT
signals:
    void onSomethingDone(int code);
    //...
public:
    //...
};
```

Sinyal fonksiyonlarının prototipleri “signals” isimli erişim belirleyici kullanılarak bildirilmelidir. “signals” belirleyicisi QT5’e kadar “protected” anlamına geliyordu:

```
#define signals protected
```

Qt-5'te "signals" belirleyicisi artık "public" anlamına gelmektedir:

```
#define signals public
```

Peki biz sinyal fonksiyonlarını sınıfta bildirirken neden doğrudan "protected" ya da "public" bölüm belirleyicilerini kullanmıyoruz da "signals" isimli makroyu kullanıyoruz? İşte başlık dosyasını ele alan "moc" programı "signals" sözcüğüne bakarak prototipi verilen fonksiyonun sinyal fonksiyonu olduğunu anlar. Yani "signals" sözcüğü "moc" programı için gerekmektedir. Bir kez daha belirtmek istiyoruz: Sinyal fonksiyonlarının yalnızca sınıf içerisindeki bildirimlerini (yani prototiplerini) programcı oluşturmaktadır. Bu fonksiyonların tanımlamalarını yine "moc" programı kendisinin ürettiği bir C++ dosyasının içerisine yerleştirilmektedir.

Slotlar bir sinyal oluştuğunda çağrılacak fonksiyonları belirtirler. Slotların hem bildirimini hem de tanımlamalarını programcı yapmak zorundadır. Tipik olarak slotlar sinyallere bağlanmaktadır. Bir sinyal oluştuğunda o sinyale bağlı olan slot fonksiyonları çağrılır. Slotları bildirmek için sınıf bildiriminde "slots" makrosu kullanılır. Slotlar "public", "protected" ya da "private" olabilirler. Örneğin:

```
class Sample : public QObject {
    Q_OBJECT
signals:
    void onSomethingDone(int code);
    //...
public slots:
    void somethingDoneHandler(int code);
public:
    //...
};
```

"slots" makrosu aşağıdaki gibi oluşturulmuş olan boş bir makrodur:

```
#define slots
```

Yani "slots" yerine boşluk karakteri yerleştirilerek bu sözcük silinmektedir. "moc" programı slotlarla ilgili bir şey yapmamaktadır. "slots" sözcüğü tamamen okunabilirliği sağlamak için kullanılmaktadır. Yani "slots" belirleyicisi kaldırılrsa da build işleminde bir sorun ortaya çıkmaz.

Slotların sinyallere bağlanması için QObject sınıfının "connect" isimli static fonksiyonlarıyla yapılmaktadır. En çok kullanılan "connect" fonksiyonu şöyledir:

```
connect(const QObject *sender, const char *signal, const QObject *receiver,
        const char *method, Qt::ConnectionType type = Qt::AutoConnection);
```

Son parametrenin default argüman aldığına dikkat ediniz. Sinyaller ve slotlar static olmayan üye fonksiyonlardır. Bir sinyalin oluşması (yani sinyal fonksiyonunun çağırılması) bir nesneyle yapılır. Bir slot da ancak bir nesneyle çağrılabilir. İşte sinyal slot bağlantısı sırasında bu durum belirtilir. Yani "connect" fonksiyonları "bu nesne ile falanca sinyal oluşursa şu nesne ile onun filanca üye fonksiyonunu (slotunu) çağır" demek için kullanılmaktadır.

QObject sınıfının static "connect" fonksiyonu bizden sırasıyla sinyal fonksiyonunun çağrılacağı nesnenin adresini, sinyal fonksiyonunu, slot fonksiyonunun çağrılacağı nesnenin adresini ve slot fonksiyonunu almaktadır. Qt5'e kadar sinyal ve slot fonksiyonları isim ve parametre türleriyle yazısal olarak bizden isteniyordu. (connect fonksiyonunun ilgili parametrelerinin const char * türünden olduğuna dikkat ediniz). Qt5'te artık sinyal ve slot fonksiyonları üye fonksiyon adresleriyle de girilebilmektedir. Eğer sinyal ve slot fonksiyonlarının isimleri yazısal olarak "connect" fonksiyonuna verilecekse bu yazının programcının kendisi tarafından oluşturulmasına gerek yoktur. Bu yazılar SIGNAL ve SLOT isimli makrolarla oluşturulmaktadır. Bu makrolara argüman olarak sinyal ve slot fonksiyonlarının isimleri ve parametre türleri verilir. Geri dönüş değerleri ve parametre değişkenlerinin isimleri verilmez.

Örneğin A sınıfının “foo” isimli bir sinyali olsun. Biz de bu sinyal tetiklendiğinde B sınıfının “bar” isimli bir slotunun çağrılmasını isteyelim. Bunun için connect işlemi şöyle yapılabilir:

```
A a;  
B b;  
//...  
QObject::connect(&a, SIGNAL(foo()), &b, SLOT(bar()));
```

Burada “a” nesnesi ile “foo” sinyali tetiklendiğinde (emit edildiğinde) “b” nesnesi ile “bar” slot fonksiyonu çağrılacaktır.

Örneğin bir pencere içerisinde “pushButtonOk” isimli bir düğme nesnesinin QPushButton sınıfı kullanılarak yaratıldığını varsayalım. QPushButton sınıfının “clicked” isimli bir sinyali vardır. Bu sinyal düğmenin üzerine tıklanıldığında QPushButton tarafından tetiklenmektedir (emit edilmektedir). Şimdi bu düğmeye tıklanıldığında biz MainWnd isimli sınıfın “clickHandler” isimli bir slot fonksiyonunun çağrılmasını isteyelim. Bağlantı işlemi şöyle yapılabilir:

```
QObject::connect(&pushButtonOk, SIGNAL(clicked()), &mainWnd, SLOT(clickHandler()));
```

Burada mainWnd MainWnd sınıfı türünden nesneyi belirtmektedir.

SIGNAL ve SLOT makroları aşağıdaki gibi bildirilmiştir:

```
#define SLOT(a)          "1"#a  
#define SIGNAL(a)       "2"#a
```

Bir sinyale bir slot bağlanırken sinyal ile slot fonksiyonlarının parametrik yapılarının ve geri dönüş değerlerinin aynı olması beklenir. Ancak slot fonksiyonu sinyal fonksiyonunun daha az parametresine sahip olabilir. (Örneğin sinyal fonksiyonunun 5 parametresi olsun. Slot fonksiyonu bunun ilk 3 parametresine sahip olabilir. Bu durumda sinyal tetiklendiği zaman yalnızca onun ilk 3 parametresi slota aktarılır.)

Bir nesne üzerinde bir sinyal oluştuğunda ona bağlı slot fonksiyonlarının çağrıldığını yukarıda belirtmiştik. Pekiyi sinyaller nasıl oluşturulmaktadır? Bir sinyalin oluşturulmasının en temel yolu doğrudan o sinyal fonksiyonun çağrılmasıdır. Örneğin:

```
emit m_pushButtonOk.clicked();
```

Sinyal fonksiyonları çağrılırken çağırma ifadesinin başına okunabilirliği artırmak için genellikle “emit” sözcüğü getirilir. “emit” sözcüğünün okunabilirlik dışında hiçbir işlevi yoktur dolayısıyla bu makro kaldırılrsa bile bir sorun oluşmaz. “emit” aşağıdaki gibi bildirilmiş boş bir makrodur:

```
#define emit
```

Yukarıda da belirtildiği gibi sinyal fonksiyonları Qt5’e kadar “protected” fonksiyonlar olarak bildirilebiliyorlardı. Dolayısıyla onları biz yukarıdaki gibi dışarıdan çağırabiliyorduk. Sinyal fonksiyonlarını ancak o sınıfın içerisinde ya da türemiş sınıftan çağırabiliyorduk. (Örneğin C#’taki event delege nesneleri de öyledir.) Ancak Qt5 ile birlikte yeni “connect” fonksiyonlarıyla sinyal fonksiyonları da mecburen public yapılmıştır. (Qt-5 ile birlikte “signals” makrosunun “public” yapıldığını anımsayınız.)

Bir sinyale birden fazla slot bağlanabilir. Bu durumda sinyal oluştuğunda (emit edildiğinde) sinyale bağlanmış olan slot fonksiyonlarının hepsi tek tek çağrılacaktır. Örneğin:

```
QObject::connect(&pushButtonOk, SIGNAL(clicked()), &a, SLOT(clickHandlerA()));  
QObject::connect(&pushButtonOk, SIGNAL(clicked()), &b, SLOT(clickHandlerB()));
```


Burada pushButtonOk isimli düğmeye tıklandığında “a” nesnesiyle “clickHandlerA” fonksiyonu “b” nesnesiyle de “clickHandlerB” fonksiyonu çağrılacaktır.

Bir sinyale birden fazla slotun bağlanması durumunda sinyal oluştuğunda tüm slot fonksiyonları çağrılır ancak son slotun geri dönüş değeri sinyal çağrısından elde edilen değer olmaktadır. Ayrıca bir sinyale birden fazla slotun bağlandığı durumda Qt dokümanları o sinyal emit edildiğinde slot fonksiyonlarının çağrılma sırası hakkında bir garanti vermemektedir. (Ancak uygulamada slot fonksiyonlarının sırasıyla çağrıldığı görülmektedir.)

Bir sinyale yalnızca slotlar değil başka sinyaller de bağlanabilmektedir. Böyle bir sinyal emit edilmesi ona bağlı olan sinyallerin emit edilmesine yol açmaktadır. Örneğin foo sinyaline bar sinyali bağlanmış olsun. Bu durumda foo sinyali emit edildiğinde bu da bar sinyalinin de emit edileceği anlamına gelir. Örneğin:

```
QObject::connect(&pushButtonOk, SIGNAL(clicked()), &pushButtonSave, SIGNAL(clicked()));
```

Burada pushButtonOk nesnesi ile clicked sinyali emit edildiğinde sinyali emit edildiğinde bu işlem pushButtonSave sinyalinin emit edilmesine yol açacak dolayısıyla da bu sinyale bağlı olan slot fonksiyonları çağrılacaktır.

Qt5 ile birlikte sinyal slot bağlantıları artık fonksiyon adresleriyle de yapılabilmektedir. Bunun için overload edilmiş başka connect fonksiyonu kullanılır:

```
connect(const QObject *sender, const QMetaMethod &signal, const QObject *receiver, const QMetaMethod &method, Qt::ConnectionType type = Qt::AutoConnection);
```

Bağlantı şöyle yapılabilir:

```
QObject::connect(&m_pushButtonOk, &QPushButton::clicked, &a, &Sample::clickhandler);
```

Tabii bu durumda sinyal ve slot fonksiyonlarının public düzeyde olması gerekmektedir. Çünkü C++’ta dışarıdan yalnızca public üye fonksiyonların adresleri alınabilir.

Şimdi bir sinyal slot uygulaması üzerinde duralım. Bu uygulamada sinyal ve slot içeren sınıflar farklı sınıflar olsun. Örneğin A sınıfındaki bir sinyal tetiklendiğinde B sınıfının bir slotunu çağırarak isteyelim:

```
// a.hpp
```

```
#ifndef A_HPP
#define A_HPP

#include <QObject>

class A : public QObject
{
    Q_OBJECT
public:
    A();
    void fire();
signals:
    int onSomethingDone(int a, int b);
};

#endif // A_HPP
```

```
// a.cpp
```

```
#include <QDebug>
#include "a.hpp"

A::A()
{}
```

```

void A::fire()
{
    int result;

    result = emit onSomethingDone(10, 20);
    qDebug() << result;
}

// b.hpp

#ifndef B_HPP
#define B_HPP

#include <QObject>

class B : public QObject
{
    Q_OBJECT
public:
    B();
public slots:
    int add(int a, int b);
    int mul(int a, int b);
};

#endif // B_HPP

// b.cpp

#include <QDebug>
#include "b.hpp"

B::B()
{}

int B::add(int a, int b)
{
    qDebug() << "B::add" ;

    return a + b;
}

int B::mul(int a, int b)
{
    qDebug() << "B::mul" ;

    return a * b;
}

// main.cpp

#include <QObject>
#include "a.hpp"
#include "b.hpp"

int main(int argc, char *argv[])
{
    A a;
    B b;

    QObject::connect(&a, SIGNAL(onSomethingDone(int,int)), &b, SLOT(add(int, int)));
    a.fire();
}

```

```

QObject::connect(&a, SIGNAL(onSomethingDone(int,int)), &b, SLOT(mul(int, int)));
a.fire();

return 0;
}

```

Burada önce sinyale tek bir slot bağlanmıştı. fire fonksiyonu sinyalin emit edilmesine yol açtığına dikkat ediniz. fire işlemi ile birlikte onSomethingDone sinyaline o zamana kadar bağlı olan bütün slot fonksiyonları çağrılmaktadır.

“connect” fonksiyonuyla sinyale bağlantı yapılıyorsa o bağlantı QObject sınıfının static “disconnect” fonksiyonlarıyla kaldırılabilir.

```

bool QObject::disconnect(const QObject *sender, const char *signal,
    const QObject *receiver, const char *method);

```

Örneğin:

```

QObject::disconnect(&m_pushButtonOk, SIGNAL(clicked()), &a, SLOT(clickHandler()));

```

Sinyal Slot Mekanizmasında sıkça sorulan sorular ve yanıtları şunlardır:

Soru: Sinyal fonksiyonu yazmak istersek bunu nasıl yazabiliriz?

Cevap: Bir kere sinyal fonksiyonun yerleştirileceği sınıfın QObject sınıfından türetilmesi ve sınıf bildiriminin başında da Q_ OBJECT makrosunun bulunması zorunludur. Bu koşullar sağlandıktan sonra sınıfta “signals” erişim belirleyicisi kullanılarak sinyal fonksiyonun yalnızca bildirimi (yani ptototipi) başlık dosyasına yazılır. Sinyal fonksiyonunun tanımlamasını moc (meta object compiler) aracı oluşturmaktadır.

Soru: Biz bir sınıf için slot fonksiyonunu yazmak istersek bunu nasıl yazabiliriz?

Cevap: Slot fonksiyonunun bulunacağı sınıfın da QObject sınıfından türetilmiş olması ve sınıf bildiriminin başına Q_ OBJECT makrosunun yerleştirilmiş olması gerekir. Bu koşullar sağlandıktan sonra sınıfın herhangi bir bölümünde bu fonksiyonun bildirimi ve sonra da tanımlaması yapılabilir. Bildirim sırasında “protected slots”, “public slots” örneklerinde olduğu gibi “slots” sözcüğü kullanılabilir. Fakat bu okunabilirliği artırmak içindir. “moc” slotlarla ilgili birşey yapmamaktadır.

Soru: Qt5’ten önce sinyal fonksiyonları “proteced” üye fonksiyon olmak zorundaydı. Bunun anlamı nedir?

Cevap: Sinyal fonksiyonu “protected” olursa onun çağırılması (emit edilmesi) o sınıf tarafından ya da o sınıftan türetilmiş bir sınıf tarafından yapılabilir. (Bu aslında daha güvenli ve istenen bir durumdur.) Ancak Qt5’teki adres olarak “connect” yapan yeni fonksiyonlar yüzünden sinyaller de “public” yapılmıştır.

Soru: Bir sinyale nasıl bir slot fonksiyonu bağlayabiliriz?

Cevap: Normal olarak sinyallere geri dönüş değeri ve parametrik yapısı sinyal fonksiyonuyla aynı olan slot fonksiyonlarını bağlayabiliriz. Ancak istisna olarak parametrik yapısı sinyalden daha az olan (ama ilk n tanesinin aynı yapıda olması gerekir) slot fonksiyonları bağlanabilmektedir.

Soru: Bir sinyale değişik zamanlarda çeşitli slotlar bağladığımız düşünelim. Sinyal fonksiyonu çağrılırsa ne olur?

Cevap: Çağırılma noktasına kadar o sinyale ne kadar slot ya da sinyal bağlanmışsa onların hepsi çağrılır. Ancak Qt dokümanları bu çağırımların sırası hakkında bir garanti vermemektedir.

Soru: Sinyal slot mekanizmasını kullanabilmek için mutlaka bir GUI uygulaması içerisinde bulunmak gerekir mi?

Cevap: Hayır bu mekanizma her durumda kullanılabilir.

Soru: Sinyallerin GUI’deki mesajlarla ilgili olması gerekiyor mu?

Cevap: Hayır gerekmiyor. Qt’nin kodları mesaj kuyruğundan pencere mesajlarını alıp bunlar için uygun sinyal fonksiyonlarını çağırılmaktadır. Yani sinyal slot mekanizması genel bir mekanizmadır ancak pencere mesajlarında da bu mekanizma kullanılmaktadır.

Soru: Sinyal fonksiyonun yalnızca bildirimini (prototipini) programcı yapıyor, tanımlamasını “moc” aracı oluşturuyor. Neden sinyallerin tanımlamasını da biz yapmıyoruz?

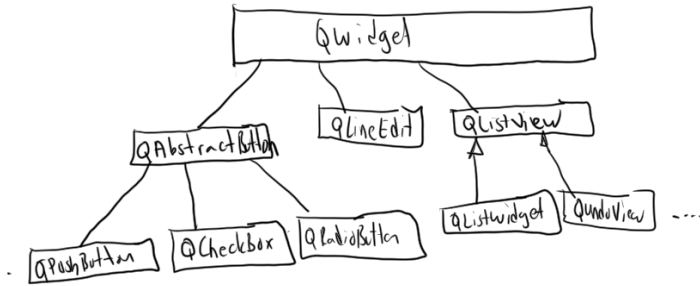
Cevap: Sinyal fonksiyonları o sinyale bağlı olan slotlara bakıp onları çağırılmaktadır. Qt tasarımcıları bu ayrıntıyla programcıların uğraşmasını istemedikleri için bu fonksiyonların “moc” aracı tarafından yazılmasını uygun görmüşlerdir.

2.13. Qt’de Sınıf Dokümanlarının İncelenmesi

Qt’nin yerel ya da web tabanlı dokümanlarında tüm sınıflar başlıklar altında açıklanmaktadır. Sınıf dokümanlarının başında önce bir içerik (contents) kısmı bulunmaktadır. Sonra sınıf hakkında özet bilgiyi içeren bir başlık kısmı gelir. Sonra “Public Types” kısmında o sınıf içerisinde (nested) bildirilmiş diğer türler listelenmiştir. Bunu “Properties” isimli bölüm izlemektedir. Burada sınıftaki önemli bazı elemanları “set” ve “get” eden üye fonksiyonlar listelenmektedir. “Property” sözcüğü diğer bazı dillerdeki (örneğin C#) property’lerle karıştırılmamalıdır. Qt’de sınıfın public veri elemanlarına ya da onları get ve set eden üye fonksiyonlarına “property” denilmektedir. Property’ler tipik olarak sınıfın bazı özellikleri hakkında bilgi almak ya da onu değiştirmek için kullanılırlar. Dokümantasyonda daha sonra “Public Functions” bölümü gelmektedir. Burada sınıfın dışarıdan çağırılacak public fonksiyonları açıklanır. Sonra da sırasıyla sınıfın “slot” fonksiyonları ve “sinyal” fonksiyonları listelenmektedir. Bunu da sınıfın “protected” elemanlarının açıklamaları izler.

3. Qt’de Pencere (Widget) Sınıflarının Kullanımı

Qt’de pencerelere “widget” denilmektedir. Widget sözcüğü “window gadget” sözcüklerinden kısaltılarak uydurulmuştur. Tüm pencereler ister ana pencere olsun, ister alt pencere olsun bazı ortak özelliklere sahiptir. İşte pencerelerin ortak elemanları tepedeki bir QWidget sınıfında toplanmıştır. Diğer bütün pencere sınıfları QWidget sınıfından türetilmiştir.



Tabii yukarıdaki türetme şeması Qt’nin tüm widget sınıflarını kapsamıyor. Biz bu şemayı kafanızda bir fikir oluşun diye çizdik.

Anımsanacağı gibi nesne yönelimli programlamada türetme kod tekrarını engellemek için de kullanılmaktadır. Bazı sınıfların ortak elemanları varsa o ortak elemanlar bir taban sınıfta toplanır ve oradan türetme yapılır. Örneğin yukarı şekle bakıldığında QPushButton, QCheckBox ve QRadioButton sınıflarının ortak elemanlarının QAbstractButton sınıfında toplandığı, tüm pencerelerin de ortak elemanlarının QWidget sınıfında toplandığı görülmektedir. Böyle bir sınıf sistemini öğrenirken tümünden gelim yöntemi daha uygun olabilmektedir. Bu nedenle biz önce QWidget sınıfının temel elemanlarını (fakat hepsini değil) ele alacağız.

3.1. QWidget Sınıfı

QWidget sınıfının bazı üye fonksiyonları pencerenin konumlandırılmasıyla ilgilidir. pos üye fonksiyonu bize pencerenin sol-üst köşe koordinatını verir:

```
QPoint pos() const;
```

Sınıfın x ve y üye fonksiyonları pencerenin sol üst köşe koordinatlarını bize int değerler olarak verir. Aslında yapılan şey pos fonksiyonuyla aynıdır.

```
int x() const;
int y() const;
```

move fonksiyonları ise pencereyi konumlandırmak için kullanılmaktadır. (Konumlandırma pencerenin sol üst köşesi belli bir koordinatta olacak biçimde yapılır)

```
void move(int x, int y);
void move(const QPoint &);
```

Ana pencereler için orijin noktası masaüstünün sol üst köşesi, alt pencereler için onların üst pencerelerinin çalışmanının sol üst köşesidir. Birim pixel cinsindendir.

rect üye fonksiyonu bize pencerenin konumunu genişlik-yükseklikle birlikte QRect olarak vermektedir:

```
QRect rect() const;
```

Ancak rect fonksiyonu her zaman pencerenin konumunu kendine göre yani çalışma alanı orijinli olarak verir. Bu nedenle rect ile alınan QRect nesnesinin sol üst köşe değeri her zaman 0 olur. Ayrıca QRect ile verilen genişlik ve yükseklik çalışma alanının genişlik ve yüksekliğidir. Yani bunun içerisine pencere sınır çizgileri ve başlık kısmı değil değildir.

size üye fonksiyonu bize pencerenin boyutunu verir, resize ise boyutu değiştirmek için kullanılmaktadır:

```
QSize size() const;
void resize(int w, int h);
void resize(const QSize &);
```

Buradaki genişlik ve yükseklik yine çalışma alanının genişlik ve yüksekliğidir. Yani buna sınır çizgileri ve başlık kısmı dahil değildir.

geometry fonksiyonu bize yine pencerenin konumunu QRect olarak verir. Ancak rect fonksiyonundan farklı olarak sol üst köşe koordinatı kendisine orijinli değildir. Üst pencerenin çalışma alanı orijinlidir. (Ana pencereler için masaüstü orijinli.) Yine geometry fonksiyonunda verilen genişlik ve yükseklik rect fonksiyonunda olduğu gibi çalışma alanının genişlik ve yüksekliğidir. geometry fonksiyonuyla biz pencerenin çalışma alanının sol-üst köşesini üst pencere çalışma alanına göre elde ederiz. geometry fonksiyonunun set işlemi yapan setGeometry isimli biçimi vardır:

```
void setGeometry(int x, int y, int w, int h);
void setGeometry(const QRect &);
```

QWidget sınıfının width ve height isimli üye fonksiyonları bize pencerenin çalışma alanının genişlik ve yüksekliğini verir. (Yani pencerenin sınır çizgileri ve başlık kısmı dahil değildir)

```
int width() const;
int height() const;
```

frameGeometry üye fonksiyonu bize pencerenin konumunu QRect olarak verir. Ancak geometry fonksiyonundan farklı olarak burada verilen genişlik ve yüksekliğe sınır çizgileri ve başlık kısmı dahildir. Ayrıca yine geometry fonksiyonundan farklı olarak frameGeometry ile elde edilen sol üst köşe değerleri çalışma alanının sol üst köşesine ilişkin değil, pencerenin tamamına ilişkindir.

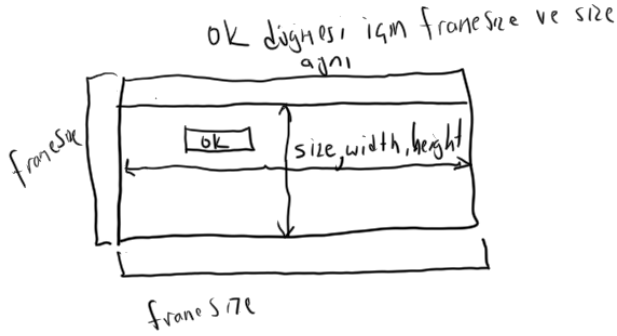
QWidget sınıfının frameSize isimli üye fonksiyonu pencerenin sınır çizgileri ve başlık kısmı dahil olmak üzere bize onun yüksekliğini ve genişliğini vermektedir.

Genellikle ana pencerelerin bir başlık kısmı ve sınır çizgileri vardır. Ancak alt pencerelerin yoktur. Yani alt pencereler için size ile frameSize aynı değerleri verir. Benzer biçimde alt pencereler için geometry değeri ile frameGeometry değeri istisna durumlar dışında aynı olacaktır.

Qt'de pencere konuma ilişkin üye fonksiyonların işlevleri yeni başlayanlara karışık gelebilmektedir. Bu konuda sıkça sorulan tipik sorular ve cevapları şunlardır:

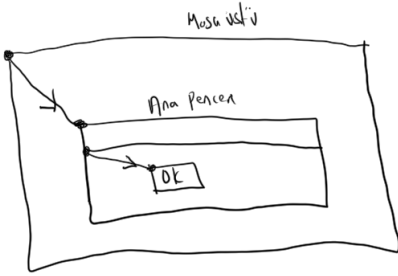
Soru: size fonksiyonu ile frameSize arasındaki farklılık nedir?

Cevap: size bize ilgili pencerenin çalışma alanının genişlik ve yüksekliğini, frameSize ise sınır çizgileri ve başlık kısmı dahil olmak üzere tüm pencerenin genişlik ve yüksekliğini vermektedir. size ile width ve height fonksiyonları her zaman aynı değeri verir. Fakat genellikle alt pencerelerin başlık kısımları ve sınır çizgileri olmadığı için alt pencereler söz konusu olduğunda size ile frameSize aynı olacaktır.



Soru: pos, x ve y ve move fonksiyonlarındaki sol üst köşe koordinatları neresidir ve orijini nereye göredir?

Cevap: Bu fonksiyonlardaki sol üst köşe her zaman pencerenin tamamının sol üst köşesidir. Buradaki sol üst köşe koordinatları alt pencere için üst pencerenin çalışma alanının sol üst köşesi orijinli, ana pencereler için masa üstünün sol üst köşesi orijinlidir.

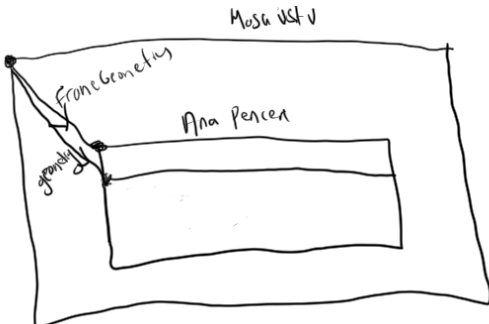


Soru: geometry ile rect fonksiyonları arasında ne fark vardır?

Cevap: Her iki fonksiyon da çalışma alanının genişlik ve yüksekliğini verir. Ancak rect sol üst köşeyi kendisi orijinli vermektedir. Halbuki geometry üst pencere orijinli olarak (ana pencere söz konusuysa masa üstü masa üstü orijinli olarak) verir.

Soru: geometry ile frameGeometry fonksiyonları arasındaki fark nedir?

Cevap: geometry bize çalışma alanının genişlik ve yüksekliğini verir. Oysa frameGeometry bize tüm pencerenin (sınır çizgileri ve başlık kısımları da dahil olmak üzere) genişlik ve yüksekliğini vermektedir. Ayrıca geometry bize sol üst köşe koordinatı olarak çalışma alanının koordinatını verir. Halbuki frameGeometry tüm pencerenin sol üst köşesinin koordinatını vermektedir. Yani frameGeometry ile verilen x ve y değerleri pos fonksiyonu ile verilenle aynıdır.



Soru: Mademki rect bize her zaman sol üst köşe koordinatı sıfır olan bir QRect veriyor, bunun uygulamada bir anlamı olabilir mi? Yani rect ile elde edilen bilgi ile size ile elde edilen bilgi aynı olmuyor mu?

Cevap: Evet teorik olarak böyle ancak bazen çalışma alanı konumunun QRect olarak ve çalışma alanı orijinal alınması gerekebilmektedir.

QWidget sınıfının windowTitle isimli QString türünden üye fonksiyonu pencere başlık yazısını almak için setTitle isimli üye fonksiyonu ise onu set etmek için kullanılır.

QWidget sınıfının bir grup bool geri dönüş değerine sahip isXXX üye fonksiyonu vardır. Bunlar ilgili özelliğin olup olmadığı konusunda biz bilgi verirler. isXXX fonksiyonlarının listesi şöyledir:

- isMaximized uygulamanın ana penceresinin maximize durumunda olup olmadığını belirtir.
- isMinimized uygulamanın ana penceresinin maximize durumunda olup olmadığını belirtir.
- isVisible herhangi bir pencerenin o anda görünür olup olmadığını anlamakta kullanılır (show fonksiyonuyla pencere görünür yapıp hide ile görünmez yapılabilir)
- isFullScreen uygulamanın ana penceresinin tam ekran durumunda olup olmadığını belirtir.
- isModal pencerenin "owned" bir pencere (yani dialog penceresi) olup olmadığını bilgisini verir.

Ayrıca pencereyi maximize duruma getirmek için showMaximized, minimize duruma getirmek için showMinimized ve normal (restore) boyuta getirmek için showNormal fonksiyonları bulunmaktadır. Bu fonksiyonlar aynı zamanda slot fonksiyonlardır.

QWidget sınıfının pencere boyutunun maksimum ve minimum değerlerini set eden üye fonksiyonları vardır. Bu fonksiyonlar çalışma alanını temel alırlar. Yani buradaki değerlere pencerenin başlık kısmı ve sınır çizgileri dahil değildir:

- setMaximumHeight fonksiyonu pencerenin getirilebileceği (fareyle ya da fonksiyonlarla) maksimum yüksekliği belirlemekte kullanılır.
- setMaximumWidth fonksiyonu pencerenin getirilebileceği (fareyle ya da fonksiyonlarla) maksimum genişliği belirlemekte kullanılır.
- setMinimumHeight fonksiyonu pencerenin getirilebileceği (fareyle ya da fonksiyonlarla) minimum yüksekliği belirlemekte kullanılır.
- setMinimumWidth fonksiyonu pencerenin getirilebileceği (fareyle ya da fonksiyonlarla) minimum genişliği belirlemekte kullanılır.
- setMaximumSize fonksiyonu ise pencerenin getirilebileceği (fareyle ya da fonksiyonlarla) maksimum genişlik ve yüksekliği beraber belirlemekte kullanılır.
- setMinimumSize fonksiyonu pencerenin getirilebileceği (fareyle ya da fonksiyonlarla) minimum genişlik ve yüksekliği beraber belirlemekte kullanılır.

Yukarı set fonksiyonlarının maxHeight, maxWidth, minHeight, minWidth ve maxSize ve minimumSize isimli get işlemi yapan biçimleri de vardır.

QWidget sınıfının setParent isimli fonksiyonu alt pencerelerde o alt pencerenin hangi pencerenin alt penceresini olduğunu belirlemkte kullanılır. Pek çok standart alt pencere sınıfının başlangıç metodu zaten bizden üst pencere nesnesini hemen almaktadır.

QWidget sınıfının diğer elemanları çeşitli konular içerisinde ele alınacaktır.

QWidget sınıfının üç sinyal fonksiyonu ve 20 civarında da slot fonksiyonu vardır. Bunlar ileride gerektiği durumlarda ele alınacaktır.

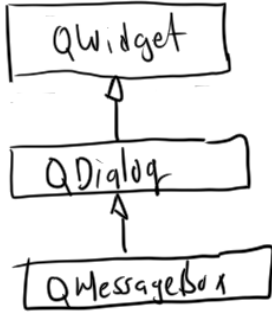
Qt'nin Standart Alt Pencere Sınıfları

Qt'de görsel arayüzü oluşturmak için pek çok standart alt pencere sınıfı bulunmaktadır. Bu sınıfların hepsi doğrudan ya da dolaylı olarak QWidget sınıfından türetilmiştir. Bu bölümde bu sınıfların önemli olanları ilk aşamada ele alınacaktır. Geri kalan standart alt pencere sınıfları daha ileride ele alınmaktadır.

Mesaj Pencerelelerinin Kullanımı

GUI uygulamalarında birtakım mesajlar diyalog pencereleriyle iletilmektedir. QDebug acil olarak teşhis amaçlı mesaj iletiminde kullanılan bir sınıftır. Release derlemesinde QDebug çağrıları koddan çıkartılmaktadır. Zaten QDebug sınıfı ile mesaj bildirimi output penceresinde yazısal olarak yapılmaktadır. İşte GUI uygulamalarında birincil mesaj iletimi için QMessageBox sınıfı kullanılmaktadır.

QMessageBox sınıfı QDialog sınıfından türetilmiştir:



Mesaj penceresi çıkartmak için önce QMessageBox sınıfı türünden yerel bir nesne yaratılır. Sonra sınıfın setText fonksiyonuyla pencere içerisindeki yazı, setWindowTitle fonksiyonuyla da başlık yazısı oluşturulur. Nihayet exec fonksiyonuyla dialog penceresi görüntülenir. Örneğin:

```
void MainWindow::pushButtonOkClickedHandler()
{
    QMessageBox mb(this);

    mb.setWindowTitle("Dikkat");
    mb.setText("Ok tuşuna basıldı");
    mb.exec();
}
```

Default durumda mesaj penceresinde yalnızca Ok tuşu vardır. Fakat biz istersek setDefaultButtons üye fonksiyonuyla birden fazla tuşun mesaj penceresinde bulunmasını sağlayabiliriz:

```
void MainWindow::pushButtonOkClickedHandler()
{
    QMessageBox mb(this);

    mb.setWindowTitle("Dikkat");
    mb.setText("Ok tuşuna basıldı");
}
```



```

        mb.setStandardButtons(QMessageBox::Yes | QMessageBox::No | QMessageBox::Cancel);

        mb.exec();
    }

```

Bu tuşların yanı sıra QMessageBox::Ok, QMessageBox::Save, QMessageBox::Abort, QMessageBox::Retry, QMessageBox::Ignore gibi seçenekler de vardır. Düğmelerin sıraları konusunda programcı belirleme yapamamaktadır. Ancak default düğme setDefaultButton fonksiyonuyla belirlenebilir. Örneğin:

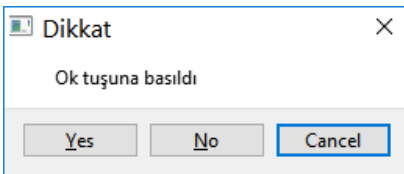
```

void MainWindow::pushButtonOkClickedHandler()
{
    QMessageBox mb(this);

    mb.setWindowTitle("Dikkat");
    mb.setText("Ok tuşuna basıldı");
    mb.setStandardButtons(QMessageBox::Yes | QMessageBox::No | QMessageBox::Cancel);
    mb.setDefaultButton(QMessageBox::Cancel);

    mb.exec();
}

```



Tabii programcının hangi tuşla dialog penceresini kapattığının da bilinmesi gerekir. İşte exec fonksiyonunun geri dönüş değeri bunu bize verir. Örneğin:

```

void MainWindow::pushButtonOkClickedHandler()
{
    QMessageBox mb(this);
    int result;

    mb.setWindowTitle("Dikkat");
    mb.setText("Ok tuşuna basıldı");
    mb.setStandardButtons(QMessageBox::Yes | QMessageBox::No | QMessageBox::Cancel);
    mb.setDefaultButton(QMessageBox::Cancel);

    result = mb.exec();

    switch (result) {
        case QMessageBox::Yes:
            qDebug() << "yes";
            break;
        case QMessageBox::No:
            qDebug() << "no";
            break;
        case QMessageBox::Cancel:
            qDebug() << "cancel";
            break;
    }
}

```

Mesaj penceresinde aynı zamanda tek bir simge de görüntülenebilir. Bunun için setIcon üye fonksiyonu kullanılmaktadır. Simgeler şunlardan biri olabilir:

```

QMessageBox::NoIcon
QMessageBox::Question
QMessageBox::Information

```

```
QMessageBox::Warning  
QMessageBox::Critical
```

Örneğin:

```
QMessageBox mb(this);  
int result;  
  
mb.setWindowTitle("Dikkat");  
mb.setText("Ok tuşuna basıldı");  
mb.setStandardButtons(QMessageBox::Yes|QMessageBox::No|QMessageBox::Cancel);  
mb.setDefaultButton(QMessageBox::Cancel);  
mb.setIcon(QMessageBox::Question);
```

Aslında mesaj pencereleri daha pratik olarak QMessageBox sınıfının static fonksiyonlarıyla da çıkartılabilir. Örneğin information fonksiyonu “information” simgesiyle mesaj penceresi çıkartır:

```
void MainWindow::pushButtonOkClickedHandler()  
{  
    int result;  
  
    result = QMessageBox::information(this, "Dikkat", "Save edecek misiniz?",  
                                     QMessageBox::Yes|QMessageBox::No|QMessageBox::Cancel);  
  
    switch (result) {  
        case QMessageBox::Yes:  
            qDebug() << "yes";  
            break;  
        case QMessageBox::No:  
            qDebug() << "no";  
            break;  
        case QMessageBox::Cancel:  
            qDebug() << "cancel";  
            break;  
    }  
}
```

Örneğin warning fonksiyonu da benzer biçimde uyarı simgesiyle mesaj penceresini açar.

```
result = QMessageBox::warning(this, "Dikkat", "Save edecek miniz?",  
                              QMessageBox::Yes|QMessageBox::No|QMessageBox::Cancel);
```

Bu fonksiyonlarda ayrıca son bir parametre girilerek default düğme de belirlenebilir:

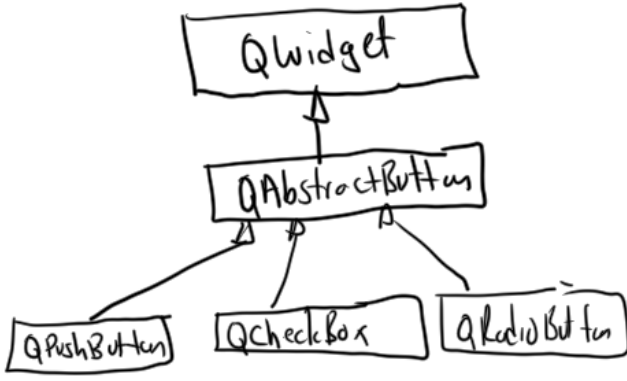
```
result = QMessageBox::warning(this, "Dikkat", "Save edecek miniz?",  
                              QMessageBox::Yes|QMessageBox::No|QMessageBox::Cancel, QMessageBox::No);
```

Bu static fonksiyonların da geri dönüş değerleri yine basılan tuşu belirtmektedir.

QPushButton Sınıfı

Düğmeler en çok kullanılan görsel arayüz elemanlarıdır. Bir işlemi başlatmak ya da bitirmek için kullanılırlar. Bunlara İngilizce “push button” denilmektedir. Çünkü düğmelerde fare ile tıklandığında değil el fareden çekilince başlatılmaktadır.

Düğmelerin (push buttons), seçenek kutularının (check boxes) ve radyo düğmelerinin (radio buttons) ortak elemanları olduğu için bu ortak elemanlar tepede QAbstractButton isimli bir sınıfta toplanmıştır. Dolayısıyla QPushButton da bu sınıftan türetilmiştir. Tabii QAbstractButton sınıfı da QWidget sınıfından türetilmiş durumdadır:



QPushButton nesnesi default başlangıç metoduyla yaratılabilir ya da iki parametrelili başlangıç metoduyla da yaratılabilir:

```

QPushButton(QWidget *parent = Q_NULLPTR)
QPushButton(const QString &text, QWidget *parent = Q_NULLPTR);
  
```

İki parametrelili başlangıç fonksiyonunun birinci parametresi düğme üzerinde görüntülenecek yazıyı belirtir. İkinci parametre bu düğmenin hangi pencerenin düğmesini olduğunu belirlemek için kullanılan üst pencere nesne adresidir.

Düğmeler default bir boyutla yaratılır. Fakat QWidget sınıfında gördüğümüz resize, setGeometry, move gibi fonksiyonlarla onların boyutlarını ve konumlarını ayarlayabiliriz.

QPushButton sınıfının QAbstractButton sınıfından gelen clicked isimli sinyali düğmeye basılıp çekildiği zaman tetiklenir.

```
void clicked(bool checked = false);
```

clicked sinyalinin parametresi QPushButton için pek anlamlı değildir. Bu nedenle genellikle slotta kullanılmaz.

QPushButton sınıfının setFlat fonksiyonu düğmeyi çıkıntısız düz bir biçimde görüntüler. Ancak bunun etkisi platformlara göre de değişmektedir.

QCheckBox Sınıfı

Küçük bir kutu ve yazıdan oluşan alt pencerelere seçenek kutuları (check box) denilmektedir. Seçenek kutuları belli bir özelliğin bütüne eklenmesi amacıyla kullanılmaktadır. Genellikle ekstra birtakım seçenekler için bu pencere tercih edilir.

QCheckBox nesnesi de QPushButton sınıfı ile benzer biçimde yaratılır:

```

m_checkBox1 = new QCheckBox("Yazıyı da gönderilsin", this);
m_checkBox2 = new QCheckBox("Kurulumdan sonra programı çalıştır", this);
m_checkBox1->move(10, 10);
m_checkBox2->move(10, 40);
  
```

Seçenek kutularından elde edilecek bilgi onun çarpılan (checked) çarpılmadığı (unchecked) bilgisidir. QAbstractButton sınıfından gelen isChecked fonksiyonu bize bu bilgiyi vermektedir. Örneğin:

```

if (m_checkBox->isChecked()) {
    //...
}

if (m_checkBox->isChecked()) {
    //...
}
  
```

```
}
```

Seenek kutularının üç konumlu biçimleri de vardır (yani arpılı, arpısız ve ekimser gibi). Ancak default durumda seenek kutuları iki konumludur. Onları üç konumlu hale getirmek için setTristate fonksiyonu kullanılır. Örneğın:

```
m_checkBox->setTristate(true);
```

Tabii üç konumlu seenek kutularında isChecked fonksiyonu yine yalnızca arpılı olup olmama durumunu bize verir. Ü konumdan hangisinin seilmiş olduėu checkState fonksiyonuyla elde edilebilir. Bu fonksiyonun geri dönüş değeri şunlardan biri olabilir:

```
Qt::Unchecked  
Qt::PartiallyChecked  
Qt::Checked
```

Tabii iki konumlu seenek kutularında yine checkState fonksiyonu kullanılabilir.

Programlama yoluyla seenke kutuları arpılabilir ya da arpıları kaldırılabilir. (Örneğın form açıldığında bir seenek kutusunun default olarak arpılı bir biçimde karşımıza gelmesini isteyebiliriz) Bunun için QAbstractButton sınıfından gelen setChecked ve QCheckBox sınıfından gelen setCheckState fonksiyonları kullanılabilir. Örneğın:

```
m_checkBox1->setCheckState(Qt::PartiallyChecked);  
m_checkBox2->setChecked(true);
```

QCheckBox sınıfının QAbstractButton sınıfından gelen clicked, pressed, released ve toggled isimli sinyalleri vardır. Bunlar sırasıyla seenek kutusuna tıklandığında, fare ile basıldığında, el fareden akıldığında ve iki konumlu düğmenin konumu değıştirildiğinde tetiklenir. Ancak ayrıca QCheckBox sınıfında stateChanged isimli sinyal de vardır:

```
void stateChanged(int state);
```

Buradaki parametre yeni konumu anlatır. 0 “unchecked”, 1 “partially checked” ve 2 de “checked” anlamına gelmektedir. Örneğın:

```
//..  
QObject::connect(m_checkBox, SIGNAL(stateChanged(int)), this,  
                 SLOT(checkBox1StateChangedHandler(int)));  
//...  
void MainWindow::checkBoxStateChangedHandler(int state)  
{  
    static char *states[] = {"Unchecked", "Undetermined", "Checked"};  
    qDebug() << states[state];  
}
```

Peki bu sinyallere (event'lere) neden gereksinim duyuluyor olabilir? Bazen birbirleriyle ilişkili seenek kutuları ya da radyo düğmeleri bulunuyor olabilir. Örneğın bir seenek kutusunun arpısının kaldırılması başka seenek kutularındaki arpıları anlamsız hale getiriyor olabilir. Onların otomatik olarak arpılarının kaldırılması bu sinyal yoluyla gerçekleştirilebilir.

QRadioButton Sınıfı

Radyo düğmeleri bir yuvarlak kutucuk ve yanında bir yazıdan oluşan alt pencerelerdir. Radyo düğmeleri birden çok seenek içerisinde onlardan yalnızca birini semek için kullanılmaktadır. Örneğın bir test sınavındaki A, B, C, D, E seenekleri radyo düğmeleriyle sunulabilir. Ya da örneğın birisinin mezuniyet durumu (İlkokul, Ortaokul, Lise vs.) radyo düğmeleriyle temsil edilebilir. Tek bir radyo düğmesinin bir anlamı yoktur. Birden fazla radyo düğmesi bir grup olarak kullanıldığında anlamlıdır. QRadioButton sınıfı da QPushButton ve QCheckBox sınıfı gibi QAbstractButton sınıfından türetilmiştir.

QRadioButton nesneleri de QPushButton ve QCheckBox sınıflarında olduğu gibi üst pencere ve bir yazı belirtilerek yaratılır.

Programcı bir grup radyo düğmesinden hangisinin seçili olduğu bilgisini almak ister. Maalesef bunun çok pratik bir yolu yoktur. Tek tek QRadioButton nesneleri ile isChecked fonksiyonun çağırılması gerekir. Örneğin:

```
void MainWindow::pushButtonOkClickedHandler()
{
    QString text = "nothing";

    if (m_radioButtonA->isChecked())
        text = m_radioButtonA->text();
    else if (m_radioButtonB->isChecked())
        text = m_radioButtonB->text();
    else if (m_radioButtonC->isChecked())
        text = m_radioButtonC->text();
    else if (m_radioButtonD->isChecked())
        text = m_radioButtonD->text();
    else if (m_radioButtonE->isChecked())
        text = m_radioButtonE->text();

    QMessageBox::information(this, "Message", text + " selected", QMessageBox::Ok);
}
```

Peki çok sayıda RadioButton nesnesi varsa else-if yapısı kodda uzun yer tutmaz mı, bunun alternatifi yok mudur? Aslında bir QObject nesneleri arasında üstlük-altlık (parent-child) ilişkisi vardır. QObject sınıfının children isimli üye fonksiyonu o QObject'ın tüm alt nesnelerini bize verir:

```
const QObjectList &children() const;
```

QObjectList sınıfı QObject nesnelerinin adreslerini tutan bir vektör tarzı sınıftır. Bu durumda biz bir QObject nesnesinin tüm alt nesnelerini şöyle elde edebiliriz:

```
for (QObject *obj : objList) {
    //...
}
```

Bu işlemin iteratör yoluyla da yapılabileceğini başlangıç konularında görmüştük. Peki QObject nesnesinin alt nesnelerinin hangi sınıf türünden olduğunu nasıl anlarız? İşte bunun için birkaç yol vardır. Örneğin QObject sınıfının inherits isimli üye fonksiyonu bizden bir sınıf ismini parametre olarak alır ve ilgili nesnenin o sınıftan türetilmiş olup olmadığını bize verir:

```
bool inherits(const char *className) const;
```

Örneğin:

```
QObjectList objList = children();
for (QObject *obj : objList) {
    if (obj->inherits("QRadioButton")) {
        QRadioButton *rb = reinterpret_cast<QRadioButton *>(obj);
        //...
    }
}
```

Yine programlama yoluyla QAbstractButton sınıfından gelen setCheck fonksiyonuyla istediğimiz bir radyo düğmesini seçebiliriz. Ancak bu fonksiyonla biz radyo düğmesindeki seçimi false parametresiyle kadiramayız.

```
QObjectList objList = children();
```

```

for (QObject *obj : objList) {
    if (obj->inherits("QRadioButton")) {
        QRadioButton *rb = reinterpret_cast<QRadioButton *>(obj);
        //...
    }
}

```

Aynı şey C++'ın `dynamic_cast` operatörüyle de yapılabilir:

```

QObjectList objList = children();
for (QObject *obj : objList) {
    if (QRadioButton *rb = dynamic_cast<QRadioButton *>(obj)) {
        //...
    }
}

```

Bu durumda biz bir pencere içerisindeki radyo düğmelerinden seçilmiş olanı şöyle de elde edebiliriz:

```

QString text = "nothing";
QObjectList objList = children();
for (QObject *obj : objList) {
    if (QRadioButton *rb = dynamic_cast<QRadioButton *>(obj)) {
        if (rb->isChecked())
            text = rb->text();
    }
}

```

Tabii else-if yapısı pek çok durumda çok daha pratiktir.

`QRadioButton` sınıfının da `QAbstractButton` sınıfından gelen `clicked`, `pressed`, `released` ve `toggled` isimli sinyalleri vardır. Bunlar sırasıyla seçenek kutusu ya da radyo düğmesine tıklandığında, fare ile basıldığında, el fareden çekildiğinde ve iki konumlu düğmenin konumu değiştirildiğinde tetiklenir.

Aynı üst pencerenin bütün `QRadioButton` alt pencereleri (yani kardeş `QRadioButton` pencereleri) aynı grubu oluşturur. Yani örneğin bir pencerenin içerisine yerleştirdiğimiz tüm radyo düğmeleri çoktan seçmenin bir elemanı olacaktır. Fakat bazı uygulamalarda birden fazla farklı konuya ilişkin grup oluşturulması istenebilir. Bunu sağlamanın tipik bir yolu şöyledir: `QGroupBox` isimli bu tür amaçlarla kullanılan bir alt pencere sınıfından faydalanmaktır. `QGroupBox` sınıfı bir widget sınıfıdır. Bu alt pencerenin amacı yalnızca birtakım pencerelere üst pencerelik yapmaktır. Dolayısıyla biz farklı radyo düğme gruplarını farklı `QGroupBox` pencerelerinin içerisine yerleştirirsek bunlar ayrı grup oluşturur.

QLineEdit Sınıfı

Qt'de edit alanlarının tek satırlı olanlarıyla çok satırlı olanları ayrı sınıflarla temsil edilmişlerdir. `QLineEdit` tek satırlı edit alanını temsil etmektedir. Çok satırlı edit alanı ise `QTextEdit` sınıfıyla temsil edilmektedir. Bir `QLineEdit` nesnesi yine diğer pencerelerde olduğu gibi üst pencere nesnesi belirtilerek yaratılır.

`QLineEdit` sınıfının `text` isimli fonksiyonu bize edit alanı içerisindeki yazıyı verir. `setText` fonksiyonu ise tam ters olarak bu yazıyı set etmemizi sağlar.

`QLineEdit` sınıfının `alignment` ve `setAlignment` fonksiyonları yazının hizalanması işlemini yapar. Hizalama için Qt isim alanındaki aşağıdaki `Alignment` isimli enum sabitleri kullanılmaktadır:

```

Qt::AlignLeft
Qt::AlignRight
Qt::AlignHCenter
Qt::AlignJustify

```

QLineEdit sınıfının maxLength ve setMaxLength fonksiyonları edit alanına girilebilecek karakteri sınırlamak için kullanılmaktadır.

QLineEdit sınıfının echoMode ve setEchoMode fonksiyonları klavyeden girilen karakterlerin görüntülenmesi ile ilgilidir. Örneğin tipik olarak parola girişlerinde kullanıcının yazdıkları edit alanında gösterilmez. Onun yerine o alanda “*”lar gösterilir. Bu fonksiyonlar QLineEdit::EchoMode isimli bir enum elemanlarını kullanırlar. Bu enum’un elemanları şöyledir:

```
QLineEdit::Normal  
QLineEdit::NoEcho  
QLineEdit::Password  
QLineEdit::PasswordEchoOnEdit
```

Sınıfın “displayText” isimli fonksiyonu bize görüntülenen default password yazısını (yani yıldızların kendilerini) verir. Tabii sınıfın text fonksiyonu bize password karakterlerini değil girilen gerçek yazıyı vermektedir.

QLineEdit sınıfının placeholderText ve setPlaceholderText fonksiyonları edit alanı boşken gösterilecek yazıyı get ve set etmekte kullanılır. Burada belirtilen yazı edit alanı boşken gösterilmektedir. Sınıfın “readOnly” ve setReadOnly” fonksiyonları edit alanını yalnızca okunabilir hale getirmektedir. Yalnızca okunabilir edit alanlarına klavye ile yazı girilemez. Yoksa setText fonksiyonuyla giriş yapılabilir.

QLineEdit sınıfının “selectedText” isimli fonksiyonu bize yazının seçili olan kısmını verir. Sınıfın setSelection isimli fonksiyonu yazının belli bölümünü programlama yoluyla seçmek için kullanılır. selectAll fonksiyonu tüm yazıyı seçer, deselect fonksiyonu ise seçimi kaldırır. Sınıfın cursorPosition ve setCursorPosition isimli fonksiyonları imlecin konumunu alıp onu set etmek için kullanılır. Edit alanı içerisindeki her karakterin ilk karakter sıfır olmak üzere bir pozisyon numarası vardır.

QLineEdit sınıfının setModified ve isModified fonksiyonları edit alanı içerisindeki karakterlerde değişiklik yapıp yapılmadığını belirlemek için kullanılmaktadır. Örneğin tipik olarak programcı setModified fonksiyonunu false parametresiyle bir kez çağırır. Sonra isModified fonksiyonu ile duruma bakar. Eğer isModified true ise demek ki bir kullanıcı edit alanı üzerinde değişiklik yapmıştır.

Sınıfın clipboard işlemi yapan “cut”, “copy” ve “paste” fonksiyonları vardır. Sınıfın frame ve setFrame fonksiyonları edit alanı etrafındaki çerçeve çizginin görünüp görünmemesini sağlar.

QLineEdit sınıfının inputMask ve setInputMask isimli fonksiyonları giriş formatını ayarlamakta kullanılır. Giriş formatı için bazı özel karakterler özel anlamlara gelmektedir. Örneğin:

```
m_lineEditName->setInputMask(“00/00/0000”);
```

Buradaki sıfır aslında “herhangi bir sayısal karakter” anlamına gelir. Format karakterleri için Qt’nin dokümanlarına başvurabilirsiniz.

QLineEdit sınıfının “clear”, “copy”, “cut”, “paste”, “redo”, “selectAll”, “setText” ve “undo” fonksiyonları aynı zamanda slot fonksiyonlarıdır.

QLineEdit sınıfının “textChanged” isimli sinyali edit alanı içerisindeki yazıda bir değişiklik olduğu zaman tetiklenir. Bu sinyalin parametresi const QString & türündendir. Edit alanında bir değişiklik olduğunda edit alanındaki değiştirilmiş yazı slot fonksiyonuna parametre olarak aktarılmaktadır. Ancak “textChanged” sinyali yazıda programla yoluyla (yani “setText” fonksiyonu yoluyla) değişiklik yapıldığında emit edilmemektedir. Eğer bu yolla da emit işleminin yapılması isteniyorsa “textEdited” sinyali kullanılmalıdır.

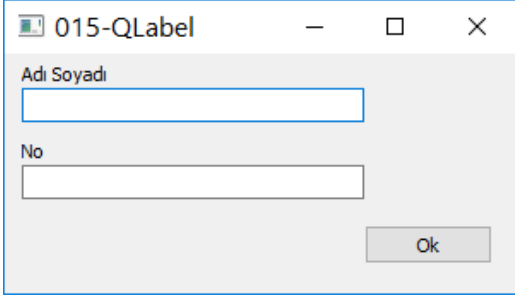
Edit alanı içerisindeki yazının fontu QWidget sınıfından gelen setFont fonksiyonuyla değiştirilebilir.

QLabel Sınıfı

QLabel sınıfı bir yazıyı görüntülemek için kullanılan alt pencereyi oluşturur. Yani QLabel içerisinde yalnızca yazı bulunan bir alt pencereyi temsil etmektedir. Örneğin:

```
m_label = new QLabel("This is a test", this);
m_label->setFont(QFont("Consolas", 20));
m_label->move(10, 10);
```

Böylece GUI ekranlarında birtakım yazılar hep QLabel kullanılarak oluşturulabilir.



Form Editörün Kullanılması

Form editörün kullanılması sırasıyla şu aşamalardan geçilip gerçekleştirilmektedir:

- 1) Kullanıcı form editörde birtakım görsel elemanları alet kutusundan sürükleyerek form'a bırakır. Aslında form editör bu yapılan işlemleri bir XML dosyasına kaydeder. Bu XML dosyasının uzantısı ".ui" biçimindedir.
- 2) ".ui" uzantılı XML dosyasını alarak bundan bir ".h" uzantılı başlık dosyası oluşturan "uic.exe (ui compiler)" isimli bir program vardır. Bu program girdi olarak ui dosyasını alır ve çıktı olarak ".h" dosyası verir. "uic" programı şöyle kullanılabilir:

```
uic -o mainwindow_ui.h mainwindow.ui
```

-o seçeneği oluşturulacak ".h" dosyasını belirlemek için kullanılmaktadır.

- 3) Programcı oluşturulan bu ".h" dosyasını include ederek kullanır. Peki bu ".h" dosyasının içerisinde ne vardır? Bu dosyanın içerisinde ui dosyasının ismi "xxx.ui" olmak üzere "Ui_xxx" biçiminde global isim alanında oluşturulmuş bir sınıf vardır. Bu sınıf aynı zamanda türetme yoluyla "ui" isimli bir isim alanının içerisinde de bildirilmiştir. Örneğin:

```
class Ui_MainWindow
{
    //...
};

namespace Ui {
    class MainWindow: public Ui_MainWindow {};
} // namespace Ui
```

Bu durumda biz doğrudan Ui_XXX sınıfını kullanabileceğimiz gibi ui::xxx biçiminde türetilmiş sınıfı da kullanabiliriz. İşte "Ui_xxx" sınıfının içerisinde public bölümde veri elemanı olarak bizim yerleştirdiğimiz widget nesneleri bulunmaktadır. Ayrıca bu sınıfta setupUi isimli static olmayan bir fonksiyon da bulunmaktadır. Örneğin:

```
class Ui_MainWindow
{
public:
    QPushButton *m_pushbuttonOk;
    QPushButton *m_pushbuttonCancel;

    void setupUi(QWidget *MainWindow)
    {
```



```

    if (MainWindow->objectName().isEmpty())
        MainWindow->setObjectName(QStringLiteral("MainWindow"));
    MainWindow->resize(400, 300);
    m_pushbuttonOk = new QPushButton(MainWindow);
    m_pushbuttonOk->setObjectName(QStringLiteral("m_pushbuttonOk"));
    m_pushbuttonOk->setGeometry(QRect(210, 250, 75, 23));
    m_pushbuttonCancel = new QPushButton(MainWindow);
    m_pushbuttonCancel->setObjectName(QStringLiteral("m_pushbuttonCancel"));
    m_pushbuttonCancel->setGeometry(QRect(300, 250, 75, 23));

    retranslateUi(MainWindow);

    QMetaObject::connectSlotsByName(MainWindow);
} // setupUi

void retranslateUi(QWidget *MainWindow)
{
    MainWindow->setWindowTitle(QApplication::translate("MainWindow", "MainWindow", 0));
    m_pushbuttonOk->setText(QApplication::translate("MainWindow", "Ok", 0));
    m_pushbuttonCancel->setText(QApplication::translate("MainWindow", "Cancel", 0));
} // retranslateUi

};

```

Görüldüğü gibi setupUi fonksiyonu bizden bir QWidget nesnesi istemektedir. Bu QWidget nesnesi yaratılacak widget'ların yerleştirileceği pencereyi belirtir. Böylece programcı yapacağı tek şey bu başlık dosyasını include edip bu sınıf türünden nesne yaratıp setupUi fonksiyonunu çağırmasıdır:

```

#include "mainwindow.hpp"
#include "ui_manualui.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    m_form = new Ui::Form;

    m_form->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete m_form;
}

```

Qt Creator IDE'sinde biz bir proje yaratırken “generate Form” seçeneği kutusu çarpılırsa (checked yapılırsa) zaten ana pencere için bir ui dosyası hazır biçimde oluşturulur.

Qt Widgets Application

Location
Kits
Details
Summary

Class Information

Specify basic information about the classes for which you want to generate skeleton source code files.

Class name:

Base class:

Header file:

Source file:

Generate form: ☒

Form file:

Next Cancel

Bu durumda Ui içeren bir şablon projede üretilen dosyalar şöyle olacaktır:

```
// mainwindow.hpp

#ifndef MAINWINDOW_HPP
#define MAINWINDOW_HPP

#include <QWidget>

namespace Ui {
class MainWindow;
}

class MainWindow : public QWidget
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
};

#endif // MAINWINDOW_HPP

// mainwindow.cpp

#include "mainwindow.hpp"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
}
```

```

        delete ui;
    }

// ui_mainwindow.h

/*****
** Form generated from reading UI file 'mainwindow.ui'
**
** Created by: Qt User Interface Compiler version 5.6.0
**
** WARNING! All changes made in this file will be lost when recompiling UI file!
*****/

#ifndef UI_MAINWINDOW_H
#define UI_MAINWINDOW_H

#include <QtCore/QVariant>
#include <QtWidgets/QAction>
#include <QtWidgets/QApplication>
#include <QtWidgets/QButtonGroup>
#include <QtWidgets/QHeaderView>
#include <QtWidgets/QWidget>

QT_BEGIN_NAMESPACE

class Ui_MainWindow
{
public:

    void setupUi(QWidget *MainWindow)
    {
        if (MainWindow->objectName().isEmpty())
            MainWindow->setObjectName(QStringLiteral("MainWindow"));
        MainWindow->resize(400, 300);

        retranslateUi(MainWindow);

        QObject::connectSlotsByName(MainWindow);
    } // setupUi

    void retranslateUi(QWidget *MainWindow)
    {
        MainWindow->setWindowTitle(QApplication::translate("MainWindow", "MainWindow", 0));
    } // retranslateUi

};

namespace Ui {
    class MainWindow: public Ui_MainWindow {};
} // namespace Ui

QT_END_NAMESPACE

#endif // UI_MAINWINDOW_H

// mainwindow.ui

<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
    <class>MainWindow</class>
    <widget class="QWidget" name="MainWindow">
        <property name="geometry">
            <rect>
                <x>0</x>
                <y>0</y>
                <width>400</width>
                <height>300</height>

```

```

    </rect>
</property>
<property name="windowTitle">
    <string>MainWindow</string>
</property>
</widget>
<layoutdefault spacing="6" margin="11"/>
<resources/>
<connections/>
</ui>

```

Form editörde bir widget seçildiğinde onun özellikleri (property'leri) değiştirilebilmektedir. Sinyal-Slot mekanizması da yine ui işlemlerine entegre edilmiştir. Biz form editörde bir widget üzerine gelip farenin sağ tuşu ile “goto slot” menüsünden sinyal için slotlar girebiliriz. Bu durumda form editör “.ui” dosyası içerisinde bu işlemi de XML olarak kodlar. Üretilen “.h” dosyasındaki setUi fonksiyonu içerisinde bu bağlantılar yapılmaktadır.

Widget nesnelerinin isimleri “özellikler penceresindeki” objectName property’si ile ayarlanabilir. Default durumda form editör widget nesnelerini onların türlerinden hareketle isimlendirmektedir.

QListWidget Sınıfı

Qt’de QListWidget diğer framework’lerdeki “listbox” kontrolüne karşılık gelmektedir. Bu widget birtakım elemanları listelemek için kullanılır.

QListWidget nesnesine eleman eklemek için sınıfın addItem ve addItemList fonksiyonları kullanılır:

```

void addItem(const QString &label);
void addItem(QListWidgetItem *item);
void addItemList(const QStringList &labels);

```

Aslında QListWidget sınıfı QListWidgetItem nesnelerini tutmaktadır. QString parametrelili addItem fonksiyonları kendi içerisinde QListWidgetItem nesnelerini yaratıp bunları eklemektedir.

QListWidgetItem sınıfında QListWidget elemanlarının özellikleri bulunur. Biz QListWidgetItem nesnesi yoluyla belli bir elemana “icon” atayabiliriz, onun fontunu, rengini vs. değiştirebiliriz.

Add fonksiyonlarının yanı sıra QListWidget sınıfının Insert fonksiyonları da vardır.

QListWidget sınıfının selectedItems isimli üye fonksiyonu bize seçilmiş olan elemanları QList<QListWidgetItem *> container nesnesi biçiminde verir. Benzer biçimde sınıfın currentItem fonksiyonu da bize o anda seçili olan elemanı vermektedir. Benzer biçimde setCurrentItem fonksiyonu da bize seçilmiş elemanı değiştirme olanağı sermektedir. selectedItems ile currentItem arasındaki fark ilkinin çoklu seçimde kullanılabilmesidir.

Benzer biçimde QListWidget sınıfının currentIndex fonksiyonu bize seçilmiş olan elemanın indeksini (satır numarasını) vermektedir.

Sınıfın currentItemChanged isimli sinyali her eleman seçildiğinde tetiklenmektedir. Sınıfın itemDoubleClicked isimli sinyali ise widget üzerinde bir elemana çift tıklanıldığında tetiklenir.

Qt'de Dizin ve Dosya İşlemleri

Qt'de dizin ve dosya işlemleri için çeşitli sınıflar kullanılmaktadır. Bu bölümde önce “yol ifadesi (path)” ve proseslerin çalışma dizini (current working directory) kavramları ele alınacak bu sınıfların en temel olanları tanıtılacaktır.

Yol İfadesi (Path Name) Nedir?

Bir dosyanın yerini belirten yazısal ifadeye yol ifadesi (path name) denilmektedir. Yol ifadeleri mutlak (absolute) ve görelî olmak üzere ikiye ayrılmaktadır. Mutlak yol ifadesi kök dizinden itibaren yer belirtirken, görelî yol ifadesi prosesin çalışma dizininden itibaren yer belirtir. Her prosesin bir çalışma dizini vardır. Bir yol ifadesinin ilk karakteri Windows'ta '\', UNIX/Linux sistemlerinde '/' ise böyle yol ifadeleri mutlak, değilse görelîdir. Örneğin:

```
"\a.dat"      ---> Windows, mutlak
"a\b\c.dat"   ---> Windows, görelî
"\a\b\c.dat"  ---> Windows, mutlak
"a.dat"       ---> UNIX/Linux ya da Windows görelî
"/a.dat"      ---> UNIX/Linux, mutlak
"a/b/c.dat"   ---> UNIX/Linux, görelî
"/a/b/c.dat"  ---> UNIX/Linux, mutlak
```

Windows UNIX/Linux uyumunu korumak için '/' karakterini de dizin geçişlerinde kabul etmektedir.

Windows sistemlerinde ayrıca sürücü (drive) kavramı da vardır. Her sürücünün ayrı bir kökü bulunur. UNIX/Linux sistemlerinde sürücü yoktur. Dolayısıyla tek bir kök vardır.

Peki Windows'ta mutlak ya da görelî yol ifadesi hangi sürücüye ilişkindir?

Windows'ta sürücü de içeren yol ifadelerine tam yol ifadesi (full path name) denilmektedir. Sürücü bir harf ve ':' karakterinden oluşur. Örneğin:

```
"c:\a\b\c.dat" /* tam yol ifadesi */
"e:\a\b\c.dat" /* tam yol ifadesi */
```

Windows'ta prosesin çalışma dizini "Proses Kontrol Bloğunda" tam yol ifadesi biçiminde tutulur. İşte eğer biz mutlak bir yol ifadesinde sürücü kullanmamışsak, Windows prosesin çalışma dizini hangi sürücüdeyse o mutlak yol ifadesinin o sürücüye ilişkin olduğunu kabul eder. Örneğin, prosesimizin çalışma dizini "e:\temp" olsun. Biz de "\a\b\c.dat" biçiminde bir mutlak yol ifadesi vermiş olalım. Burada kök "e" sürücüsünün köküdür.

Windows'ta ilginç bir durum daha vardır. Görelî bir yol ifadesi bir sürücü içerebilir. Örneğin prosesimizin çalışma dizini "d:\temp" olsun:

```
"c:a\b\c.dat"
"e:c.dat"
```

Buradaki görelî yol ifadeleri nereden itibaren yer belirtmektedir? İşte Windows burada bazı çevre değişkenlerine (environment variables) başvurur. Bu çevre değişkenleri tanımlı değilse Windows yine bu sürücülerin kök dizinlerinden itibaren yolu belirlemektedir. Yani bu çevre değişkenleri tanımlanmamışsa (pek çok sistemde tanımlanmamıştır) yukarıdaki yol ifadeleri aşağıdakilerle eşdeğer olur:

```
"c:\a\b\c.dat"
"e:\c.dat"
```

Proseslerin çalışma dizinleri için başında neresidir? Program çalıştırıldığında çalışma dizini onu çalıştıran program tarafından belirlenir. Fakat "cmd.exe" gibi "bash" gibi shell programları üzerinden program çalıştırılıyorsa bu shell programları çalıştırılabilen dosyanın bulunduğu yeri prosesin başlangıçtaki çalışma dizini yapmaktadır. Windows'un grafik arayüzü de ("explorer.exe") aynı biçimde çalıştırılabilen dosyanın bulunduğu yeri başlangıçtaki yol ifadesi olarak ele alır. Bir proses çalışırken sistem fonksiyonlarıyla ya da API fonksiyonlarıyla prosesin çalışma dizini değiştirilebilmektedir.

Genel olarak IDE’lerde program çalıştırıldığında onun çalışma dizini bu IDE’lerin ayarlarından değiştirilebilmektedir. QtCreator IDE’sinde uygulamanın çalışma dizinini değiştirmek için sol taraftaki ana araç çubuğundan “Projects” seçilir. Burada “Run” seçeneğine gelinir. “Working Directory” kısmına istenilen dizin girilir.

Anahtar Notlar: Sağ taraftaki ana araç çubuğundaki “Projects” seçeneği seçildiğinde “Build” sekmesinde “Shadow build” isimli bir seçenek kutusu vardır. Bu seçenek kutusu default olarak çarpılanmış biçimdedir. “Shadow build” object dosyaların ve “executable dosyaların” proje dizininin dışındaki ayrı bir dizinde tutulacağını belirtir. Eğer biz “Shadow build” seçenek kutusundaki çarpıyı kaldırırsak bu dosyalar proje dizini içerisinde oluşturulur.

QDir Sınıfı

QDir dizin işlemleri yapmakta kullanılan ana bir sınıftır. Sınıf nesnesi dizin ismi verilerek yaratılır. Bu dizinin mevcut olması zorunlu değildir:

```
QDir dir("test");
```

- Sınıfın static olmayan exists üye fonksiyonu ilgili dizinin var olup olmadığı bilgisini bize verir. Örneğin:

```
QDir dir("D:\\Dropbox\\Kurslar\\Qt-Mayis-2016");

QDebug() << (dir.exists() ? "Var" : "Yok");
```

- Sınıfın dirName isimli fonksiyonu bize dizinin yalnızca ismini verir (yol ifadesini değil). Örneğin:

```
QDir dir("D:\\Dropbox\\Kurslar\\Qt-Mayis-2016");

QDebug() << dir.dirName();
```

- Sınıfın path isimli static olmayan üye fonksiyonu dizinin tüm yol ifadesini bize verir.

- Sınıfın mkdir isimli static olmayan fonksiyonu ilgili dizinin altında yeni bir dizin yaratır.

```
QDir dir("D:\\Dropbox\\Kurslar\\Qt-Mayis-2016");

if (!dir.mkdir("test")) {
    cerr << "cannot create directory!..\n";
    exit(EXIT_FAILURE);
}

cout << "Ok\n";
```

Dizin zaten varsa fonksiyon başarısız olmaktadır.

- Sınıfın static olmayan rmdir isimli fonksiyonu dizini siler. Ancak dizinin silinmesi için dizinin boş olması gerekir. Örneğin:

```
QDir dir("D:\\Dropbox\\Kurslar\\Qt-Mayis-2016");

if (!dir.rmdir("test")) {
    cerr << "cannot delete directory!..\n";
    exit(EXIT_FAILURE);
}

cout << "Ok\n";
```

- Sınıfın entryList isimli üye fonksiyonları bize dizin içerisindeki dosyaları QStringList nesnesi olarak verir.

```
#include <QDir>
#include <QDebug>
```

```

int main(int argc, char *argv[])
{
    QDir dir("D:\\Dropbox\\Kurslar\\Qt-Mayis-2016");

    QStringList sl = dir.entryList();

    foreach(QString name, sl)
        qDebug() << name;

    // diğer alternatif
    for (QString name : dir.entryList())
        qDebug() << name;

    return 0;
}

```

Normal olarak entryList dizindeki tüm girişlerinin isimlerini (yani alt dizinlerin ve dosyaların) bize verir. Fakat bu fonksiyonun parametresi QDir::Filter türünden bir enum türündendir ve default argüman almıştır. Böylece biz istediğimize uygun biçimde olan girişleri elde edebiliriz:

```

#include <QDir>
#include <QDebug>

int main(int argc, char *argv[])
{
    QDir dir("c:\\windows");

    QStringList sl = dir.entryList(QDir::Files|QDir::Executable);

    foreach(QString name, sl)
        qDebug() << name;

    return 0;
}

```

QDir::Filter enum türünün elemanlarını Qt dokümanlarından inceleyiniz.

- Sınıfın entyrInfoList üye fonksiyonu yine bize benzer biçimde dizin içerisindeki dosyaların listesini verir. Ancak bu listeyi QFileInfo nesneleri biçiminde bize vermektedir. QFileInfo bir dosyanın yalnızca ismini değil diğer özelliklerini de barındıran bir sınıftır. Örneğin:

```

#include <iostream>
#include <QDir>
#include <QFileInfo>
#include <QDebug>

using namespace std;

int main(int argc, char *argv[])
{
    QDir dir("c:\\windows");

    for (QFileInfo qf : dir.entryInfoList(QDir::Files|QDir::Executable))
        cout << qf.fileName().toString() << ", " << qf.size() << endl;

    return 0;
}

```

- QDir sınıfının currentPath isimli static üye fonksiyonu bize çalışmakta olan programın çalışma dizinini (current working directory) verir. setCurrent fonksiyonu ise bunu değiştirmektedir. Örneğin:

```

#include <iostream>
#include <QDir>
#include <QFileInfo>
#include <QDebug>

using namespace std;

int main(int argc, char *argv[])
{
    cout << QDir::currentPath().toStdString() << endl;

    QDir::setCurrent("c:\\windows");

    QDir dir(".");
    for (QFileInfo qf : dir.entryInfoList())
        cout << qf.fileName().toStdString() << ", " << qf.size() << endl;

    return 0;
}

```

QByteArray Sınıfı

QByteArray sınıfı dosya işlemlerinde karşımıza çok sık çıkmaktadır. Bu sınıf byte'ları (yani unsigned char türünden nesneleri) tutan bir sınıftır. QByteArray C++'ın standart kütüphanesindeki "deque" veri yapısına benzetilebilir. Sınıfın başlangıç fonksiyonu bizden tutacağı byte bilgilerini alabilir. Ya da QByteArray nesnesi default başlangıç fonksiyonuyla içi boş olarak yaratılabilir:

```

QByteArray();
QByteArray(const char *data, int size = -1);
QByteArray(int size, char ch);
QByteArray(const QByteArray &other);
QByteArray(QByteArray &&other);

```

Örneğin:

```
QByteArray qba(100, 0);    // nesnenin içerisinde 100 byte var, hepsi 0
```

- Sınıftaki eleman sayısı size ya da length ya da count üye fonksiyonuyla elde edilebilir.

- QByteArray C++'ın iterator sistemini desteklemektedir. Kullanımı std::deque sınıfına benzetilebilir. Sona eleman eklemek için push_back ya da append fonksiyonları kullanılır. Örneğin:

```
qba.append(10);           // nesnenin sonuna 10 değerindeki byte ekleniyor
```

- clear üye fonksiyonu yine nesne içerisindeki tüm byte'ları silmek için kullanılır.

- insert fonksiyonlarıyla belli bir pozisyonel insert yapılabilir.

- indexOf fonksiyonları arama amacıyla kullanılır.

- Nesnenin içerisindeki belli bir kısım mid fonksiyonuyla çekilip alınabilir. Örneğin:

```

#include <iostream>
#include <QByteArray>

using namespace std;

int main(void)
{
    QByteArray qba;

```



```

    for (int i = 0; i < 100; ++i)
        qba.push_back(i);

    QByteArray result = qba.mid(10, 5);

    for (unsigned char ch : result)
        cout << (int)ch << " ";
    cout << endl;

    return 0;
}

```

- Nesnenin başına ekleme yapmak için prepend fonksiyonları kullanılır.

- Kapasite artımı için reserve, size artırımını için resize fonksiyonları kullanılmaktadır.

- Sınıfın setNum fonksiyonları nesnenin içerisine çeşitli türlerden değerleri yazısal biçimde byte byte kodlar. Fakat bu fonksiyonlar sona ekleme yapmazlar. Nesnenin içerisindeki bilgileri silerek onu yeniden oluştururlar. Örneğin:

```

#include <iostream>
#include <QByteArray>

using namespace std;

int main(void)
{
    QByteArray qba;
    int a = 123456;

    qba.setNum(a);    // yazısal eklama 6 byte

    cout << qba.size() << endl;    // 4

    return 0;
}

```

- Sınıfın toXXX fonksiyonları bize nesne içerisindeki yazısal biçimde bulunan değeri XXX türünden vermektedir. Başka bir deyişle toXXX fonksiyonları setNum fonksiyonlarının ters işlem yapan biçimi gibidir. Örneğin:

```

#include <iostream>
#include <QByteArray>

using namespace std;

int main(void)
{
    QByteArray qba;
    int a = 123456;
    int b;

    qba.setNum(a);    // yazısal olarak 6 byte kodlar

    b = qba.toInt(); // yazısal olan 6 byte'ı yeniden int türüne dönüştürür

    cout << b << endl;    // 123456

    return 0;
}

```

Ayrıca sınıfın toString fonksiyonu QByteArray içerisindeki byte'ları bize yazıya dönüştürerek C++ string sınıfı biçiminde verir.

- Sınıfın number isimli static fonksiyonları belli türdeki değerleri bize QByteArray olarak vermektedir. Bu fonksiyonlar da değerleri yine setNum fonksiyonlarında olduğu gibi karaktere dönüştürerek bu karakterleri QByteArray içerisinde tutmaktadır: Örneğin:

```
#include <iostream>
#include <QByteArray>

using namespace std;

int main(void)
{
    QByteArray qba;

    qba = QByteArray::number(12.34);

    for (QByteArray::iterator iter = qba.begin(); iter != qba.end(); ++iter)
        cout << *iter << endl;

    return 0;
}
```

Yukarıda da belirtildiği gibi QByteArray aslında std::deque sınıfına benzetilebilir. Nesne içerisindeki byte'lar yine [] operatör fonksiyonuyla char olarak sanki bir diziymiş gibi elde edilebilmektedir. Örneğin:

```
#include <iostream>
#include <QDebug>
#include <QByteArray>

using namespace std;

int main(void)
{
    QByteArray qba;

    qba.setNum(123456);

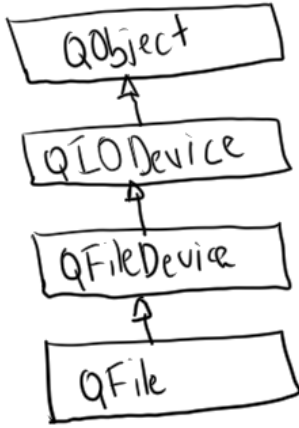
    for (int i = 0; i < qba.length(); ++i)
        qDebug() << qba[i];

    return 0;
}
```

QByteArray sınıfının başka önemli üye fonksiyonları da vardır. Bu fonksiyonlar Qt dokümanlarından incelenebilir.

QT'de Dosya İşlemleri ve QFile Sınıfı

Dosya işlemleri için kullanılan temel sınıf QFile sınıfıdır. Sınıfın türetme şeması şöyledir:



QFile sınıfı Qt'de disk tabanlı dosyaları temsil etmekte kullanılmaktadır. QFileDevice ve QIODevice sınıfları daha genel dosyası kavramları ifade etmektedir. Başka deyişle dosya gibi kullanılan başka kavramlar da bu taban sınıflardan türetilmiş durumdadır. Bir QFile nesnesi sınıfın yol ifadesi alan başlangıç fonksiyonuyla yaratılabilir. Örneğin:

```
QFile file("a.dat");
```

- Bir dosyayı açmak için QFile sınıfının QIODevice sınıfından gelen open fonksiyonu kullanılabilir:

```
bool QIODevice::open(OpenMode mode)
```

Fonksiyon parametre olarak açış modunu almaktadır. Dosya açış modları QIODevice sınıfın içerisindeki OpenMode isimli enum türüyle temsil edilmiştir. Açış modları şunlardan oluşturulabilir:

```
QIODevice::NotOpen  
QIODevice::ReadOnly  
QIODevice::WriteOnly  
QIODevice::ReadWrite  
QIODevice::Append  
QIODevice::Truncate  
QIODevice::Text  
QIODevice::Unbuffered
```

ReadOnly bayrağı yalnızca okunabilir dosya açmak için, WriteOnly bayrağı yalnızca yazılabilir dosya açmak için, ReadWrite bayrağı ise hem okunabilir hem de yazılabilir dosya açmak için kullanılır. Text bayrağı Windows için CR/LF dönüştürmesi yapmaktadır. Yani dosyayı text modda açar. Bu bayrağın UNIX/Linux sistemlerinde bir etkisi yoktur. Append bayrağı her yazma işleminde yazılacakları sona ekler. Truncate bayrağı eğer dosya zaten varsa onu sıfırlar ve açar. Normal durumda dosya yoksa yaratılmaktadır. Bu bayrağın kullanılmadığı durumda açılmak istenen dosya yoksa ve açış mode QIODevice::WriteOnly ya da QIODevice::ReadWrite ise dosya yaratılır. Varsa olan açılır. Unbuffered bayrağı dosya tamponlamasını ortadan kaldırmaktadır. NotOpen bayrağı bir açış bayrağı değildir. Bu default durumdur. Yani henüz dosyanın açılmamış olduğunu belirtir. Fonksiyon başarı durumunu belirten true değerleri ile başarısızlık durumunda false değeri ile geri dönmektedir.

- Sınıfın QIODevice sınıfından gelen close fonksiyonu dosyayı kapatır. Sınıfın bitiş fonksiyonu zaten dosya açıksa onu kapatmaktadır. (Kaldı ki işletim sistemi düzeyinde zaten bir proses sonlandığında o prosesin açmış olduğu dosyalar otomatik kapatılmaktadır.) Yani pek çok durumda close kullanmamıza gerek kalmaz.

Örneğin:

```
#include <iostream>  
#include <cstdlib>  
#include <QFile>  
#include <QDebug>
```

```

using namespace std;

int main(void)
{
    QFile file("a.dat");

    if (!file.open(QIODevice::WriteOnly)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }

    cout << "Ok\n";

    file.close();          // aslında bu programda

    return 0;
}

```

- Sınıfın static bazı fonksiyonları bir dosya üzerinde bütünsel klasik işlemleri yapmakta kullanılmaktadır: Örneğin copy fonksiyonu dosya kopyalamak için, remove fonksiyonu dosya silmek için, rename fonksiyonu dosyanın ismini değiştirmek için kullanılır. Örneğin:

```

#include <iostream>
#include <cstdlib>
#include <QFile>
#include <QDebug>

using namespace std;

int main(void)
{
    if (!QFile::rename("a.dat", "b.dat")) {
        cerr << "cannot rename file!..\n";
        exit(EXIT_FAILURE);
    }

    cout << "ok\n";

    return 0;
}

```

- Dosyanın açık olup olmadığı isOpen fonksiyonuyla elde edilebilir.

- exists fonksiyonu dosyanın olup olmadığını belirlemek için kullanılır.

- Dosyadan okuma yapmak için iki overload edilmiş iki static olmayan read fonksiyonu ve bir de readAll fonksiyonu vardır:

```

qint64 QIODevice::read(char *data, qint64 maxSize);
QByteArray QIODevice::read(qint64 maxSize);
QByteArray QIODevice::readAll();

```

Bu fonksiyonlar dosya göstericisinin gösterdiği yerden itibaren parametreleriyle belirtilen miktarda byte değerini okur. İki parametrelili read fonksiyonu okunan bilgileri bizim verdiğimiz char türden bir adresten itibaren yerleştirir. Tek parametrelili read fonksiyonu ise okunan bilgileri QByteArray olarak vermektedir. readAll fonksiyonu dosya göstericisinin gösterdiği yerden dosya sonuna kadar tüm byte'ları okumaktadır. İki parametrelili read fonksiyonu IO hatası durumunda -1 değerine geri döner. Ancak diğerleri boş bir QByteArray nesnesine geri dönmektedir.

```

#include <iostream>

```

```

#include <cstdlib>
#include <QFile>

using namespace std;

int main(void)
{
    QFile file("test.txt");

    if (!file.open(QIODevice::ReadOnly)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }

    char buf[100 + 1];
    qint64 n;

    n = file.read(buf, 100);
    buf[n] = '\0';

    cout << buf << endl;

    file.close();

    return 0;
}

```

Örneğin:

```

#include <iostream>
#include <cstdlib>
#include <QFile>

using namespace std;

int main(void)
{
    QFile file("test.txt");

    if (!file.open(QIODevice::ReadOnly)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }

    QByteArray qba;

    qba = file.read(100);

    cout << qba.toString() << endl;

    file.close();

    return 0;
}

```

Burada QByteArray sınıfının toString fonksiyonuyla okunan byte'lar yazıya dönüştürülmüştür. Ancak std::string sınıfının 1 byte karakterlerden oluşan yazıları tuttuğunu unutmayınız.

- QFile sınıfının yazma işlemini yapan üç write fonksiyonu vardır:

```

qint64 QIODevice::write(const char *data, qint64 maxSize);
qint64 QIODevice::write(const char *data);

```

```
qint64 QIODevice::write(const QByteArray &byteArray);
```

Birinci fonksiyon belli bir adresten itibaren belli sayıda byte'ı dosya göstericisinin gösterdiği yerden itibaren dosyaya yazar. İkinci fonksiyon bir yazıyı null karakter görene kadar byte byte dosyaya yazmaktadır. Üçüncü fonksiyon ise QByteArray içerisindekileri dosyaya yazmaktadır. Örneğin:

```
#include <cstdlib>
#include <iostream>
#include <QDebug>
#include <QFile>

int main(void)
{
    QFile file1("../Console-Test/main.cpp");
    QFile file2("a.dat");

    if (!file1.open(QIODevice::ReadOnly)) {
        qDebug() << "cannot open file!..";
        exit(EXIT_FAILURE);
    }

    if (!file2.open(QIODevice::WriteOnly|QIODevice::Truncate)) {
        qDebug() << "cannot open file!..";
        exit(EXIT_FAILURE);
    }

    QByteArray qba;

    qba = file1.readAll();
    file2.write(qba);

    return 0;
}
```

Örneğin:

```
#include <iostream>
#include <cstdlib>
#include <QFile>

using namespace std;

int main(void)
{
    QFile file("test.txt");

    if (!file.open(QIODevice::ReadWrite|QIODevice::Truncate)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < 100; ++i)
        file.write(reinterpret_cast<const char *>(&i), sizeof(int));

    file.reset();

    int val;
    for (int i = 0; i < 100; ++i) {
        file.read(reinterpret_cast<char *>(&val), sizeof(int));
        cout << val << " ";
    }
    cout << endl;
}
```

```

    file.close();

    return 0;
}

```

- Dosya göstericisini konumlandırmak için seek ve reset fonksiyonları kullanılır:

```

bool QFileDevice::seek(qint64 pos);
bool QIODevice::reset();

```

QFile sınıfının diğer üye fonksiyonları Qt dokümanlarından incelenebilir.

- Sınıfın pos üye fonksiyonu dosya göstericisinin konumunu verir, atAnd fonksiyonu ise dosya göstericisinin dosyanın sonunda olup olmadığı bilgisini verir.

Dosya İşlemlerinde Kullanılan Adaptör Sınıflar

Bir sınıfı alarak karmaşık işlemleri o sınıfı kullanarak yapan sınıflara adaptör sınıf (adapter class) denilmektedir. Dosya işlemlerinde kullanılan QTextStream ve QDataStream iki önemli adaptör sınıftır. Bu sınıflar sayesinde yalnızca QFile değil genel olarak QIODevice sınıfından türetilmiş olan sınıflar için text ve binary işlemler daha kolay yapılabilmektedir.

QTextStream Sınıfı

QTextStream nesnesi bizden bir QIODevice (QFile sınıfının bu sınıftan türetildiğini anımsayınız) ya da bir yazıyı alarak üzerinde text işlemleri yapar. Örneğin:

```

#include <iostream>
#include <cstdlib>
#include <QFile>
#include <QTextStream>

using namespace std;

int main(void)
{
    QFile file("test.txt");

    if (!file.open(QIODevice::ReadOnly)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }
    QTextStream ts(&file);

    QString str = ts.readAll();
    cout << str.toString() << endl;

    file.close();

    return 0;
}

```

Yukarıda da gördüğümüz gibi QTextStream sınıfının readAll isimli üye fonksiyonu dosyanın içerisindeki tüm bilgiyi okur bize onu yazı biçiminde QString nesnesi olarak verir.

Sınıfın readLine üye fonksiyonu dosya göstericisinin gösterdiği yerden bir satırı okur ve o satırı bize QString nesnesi olarak verir:

```
QString QTextStream::readLine(qint64 maxlen = 0);
```

Örneğin:

```
#include <iostream>
#include <cstdlib>
#include <QFile>
#include <QTextStream>

using namespace std;

int main(void)
{
    QFile file("test.txt");

    if (!file.open(QIODevice::ReadOnly)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }
    QTextStream ts(&file);

    QString str;

    while (!(str = ts.readLine()).isNull())
        cout << str.toStdString() << endl;

    file.close();

    return 0;
}
```

QTextStream sınıfının çeşitli türden sayıları yazıya dönüştürerek yazdıran << operatör fonksiyonları vardır. Örneğin:

```
#include <cstdlib>
#include <iostream>
#include <QDebug>
#include <QFile>
#include <QTextStream>

int main(void)
{
    QFile file("test.txt");

    if (!file.open(QIODevice::WriteOnly|QIODevice::Truncate)) {
        qDebug() << "cannot open file!..";
        exit(EXIT_FAILURE);
    }

    QTextStream ts(&file);

    for (int i = 0; i < 10; ++i)
        ts << "Sayi: " << i << ", Karesi: " << i * i << endl;

    return 0;
}
```

Örneğin Excel'den CSV olarak kaydedilmiş bir text dosyayı aşağıdaki gibi yazdırabiliriz:

```
#include <iostream>
#include <cstdlib>
#include <QFile>
#include <QTextStream>
```



```

using namespace std;

int main(void)
{
    QFile file("Excel.csv");

    if (!file.open(QIODevice::ReadOnly)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }
    QTextStream ts(&file);

    for (;;) {
        QString row = ts.readLine();
        if (row.isNull())
            break;
        QStringList sl = row.split(";");

        for (QString elem : sl)
            cout << elem.toStdString() << '|';
        cout << endl;

    }

    file.close();

    return 0;
}

```

Benzer biçimde sınıfın bir text dosyadaki karakterleri okuyarak onu bize sayı olarak veren >> operatör fonksiyonları da vardır. Örneğin test.txt dosyasının içeriği şöyle olsun:

```

1.2      2.2      3.3
4.4      5.5      6.6
7.7      8.8      9.9

```

Buradaki sayıları aşağıdaki gibi tek tek okuyabiliriz. Okuma işlemi ilk digit olmayan karakterde (örneğin boşluk karakteri sonlanmaktadır) sonlanmaktadır:

```

#include <iostream>
#include <cstdlib>
#include <QFile>
#include <QTextStream>

using namespace std;

int main(void)
{
    QFile file("test.txt");

    if (!file.open(QIODevice::ReadOnly)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }
    QTextStream ts(&file);
    double d;

    for (;;) {
        ts >> d;
        if (ts.atEnd())
            break;
    }
}

```

```

        cout << d << endl;
    }

    file.close();

    return 0;
}

```

QDataStream Sınıfı

QDataStream sınıfı da yine QIODevice nesnesini bizden alır. Bu nesneyi kullanarak bazı binary okuma yazma işlemlerini bizim için yapar. Yani QTextStream text işlemlerini kolaylaştırırken QBinaryStream binary işlemleri kolaylaştırmaktadır.

- Sınıfın << operatör fonksiyonları sayıları dosyaya yazdırmaktadır. Ancak bunlar QTextStream sınıfında olduğu gibi sayıyı yazıya dönüştürüp onun karakterlerini dosyaya yazdırmak yerine doğrudan sayıyı byte byte dosyaya yazdırmaktadır. Örneğin:

```

#include <iostream>
#include <cstdlib>
#include <QFile>
#include <QDataStream>

using namespace std;

int main(void)
{
    QFile file("test.dat");

    if (!file.open(QIODevice::WriteOnly|QIODevice::Truncate)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }
    QDataStream ds(&file);

    for (int i = 0; i < 10; ++i)
        ds << i;

    file.close();

    return 0;
}

```

- Benzer biçimde QDataStream sınıfının >> operatör fonksiyonları da dosya göstericisinin gösterdiği yerden binary okuma yaparak onu bir nesneye yerleştirir. Örneğin:

```

#include <iostream>
#include <cstdlib>
#include <QFile>
#include <QDataStream>

using namespace std;

int main(void)
{
    QFile file("test.dat");

    if (!file.open(QIODevice::ReadOnly)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }
}

```

```

    QDataStream ds(&file);

    int val;
    for (;;) {
        ds >> val;
        if (ds.atEnd())
            break;
        cout << val << endl;
    }

    file.close();

    return 0;
}

```

Sınıfın diğer üye fonksiyonları Qt Dokmanlarından elde edilebilir.

Şüphesiz biz kendi sınıflarımız için QTextStream ya da QDataStream sınıflarına yönelik << ve >> operatör fonksiyonlarını yazabiliriz. Örneğin:

```

#include <iostream>
#include <cstdlib>
#include <QFile>
#include <QTextStream>

using namespace std;

class Sample {
private:
    int m_a, m_b;
public:

    Sample(int a, int b) : m_a(a), m_b(b)
    {}
    int a() const {return m_a;}
    int b() const {return m_b;}
};

QTextStream &operator <<(QTextStream &ds, const Sample &s)
{
    ds << s.a() << ", " << s.b() << endl;

    return ds;
}

int main(void)
{
    QFile file("test.txt");

    if (!file.open(QIODevice::WriteOnly|QIODevice::Truncate)) {
        cerr << "cannot open file!..\n";
        exit(EXIT_FAILURE);
    }
    QTextStream ts(&file);

    Sample s(10, 20);

    ts << s;

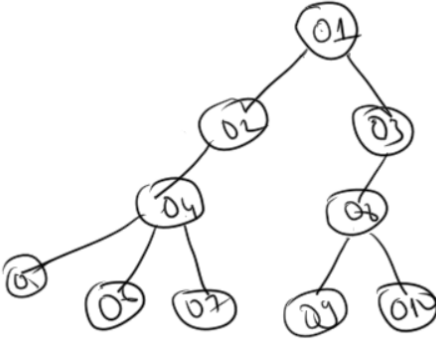
    file.close();
}

```

```
    return 0;  
}
```

Qt'de Dinamik Bellek Yönetimi

Normal olarak her new işlemi için bir biçimde bir delete işleminin yapılıyor olması gerekir. Tabii bilindiği gibi Windows gibi, Linux gibi, Mac OS X gibi sistemlerde heap prosese özgü olduğu için program sonlandığında prosesin kullandığı heap alanı boşaltılmaktadır. Qt'de QObject sınıfı temel alınarak otomatik bir boşaltım öngörülmüştür. Tasarım kalıplarında "Composite" Kalıbı denilen teknik uygulanmıştır. Bu tekniğe göre QObject türünden ya da QObject'ten türetilmiş her nesne için bir üst nesne (parent) set edilebilmektedir. Bu üst nesne delete edildiğinde ya da ömrünü bitirip yok edileceği zaman o üst nesnenin ağaç olarak tüm alt nesnelere delete uygulanmaktadır. Örneğin:



Burada O1 delete edildiğinde O1'in bitiş fonksiyonu O2 ve O3 için delete işlemi yapar. Onlar da kendi tuttukları nesneler için delete işlemlerini yapacak böylece tüm ağaç delete edilmiş olacaktır.

O halde Qt'de biz QObject türetilmiş bir nesne yaratırken onun üst nesnesini uygun bir biçimde ayarlarsak bu üst nesne delete edildiğinde bizim nesnemizin de delete edilmesi sağlanmış olur. Eğer bir QObject nesnesi için üst nesne NULL olarak geçilmişse bu durumda bu nesne otomatik olarak delete edilmez. Onu bizim delete etmemiz gerekir. Ayrıca QObject sınıfından türetilmemiş sınıflar için böyle bir otomatik delete mekanizması yoktur. Onların delete edilmesini bizim sağlamamız gerekir. Örneğin QVector gibi bir container sınıf QObject sınıfından türetilmemiştir. Dolayısıyla böyle bir nesneyi biz new operatörüyle dinamik olarak yaratmışsak onu yine bizim bir noktada delete etmemiz gerekir.

QObject sınıfının destroyed isimli sinyali nesne yok edilmeden hemen önce emit edilmektedir. Yani biz bir nesne yok edilmeden hemen önce başka sınıflarda birtakım şeyler yapacaksa bu sinyalden faydalanabiliriz.

Qt'de Kaynak (Resource) Kullanımı

Bir programda kullanılan icon gibi, bitmap gibi, ses gibi öğelerin çalıştırılabilen dosyaların ya da dinamik kütüphanelerin içerisine yerleştirilerek kullanılmasına "kaynak kullanımı" denilmektedir. Qt'de kaynak kullanımı şu aşamalardan geçilerek gerçekleştirilir:

1) QT'nin kaynakları bir XML dosyasında ifade edilmektedir. Ancak programcının bu XML sentaksını bilmesine gerek yoktur. Çünkü QtCreator gibi IDE'ler bu XML'i görsel işlemlerle oluşturabilmektedir. QtCreator'da proje seçeneklerinden "Add New / Qt / Qt Resource File" seçilirse uzantısı ".qrc" olan bir dosya oluşturulur.

2) QtCreator'da ".qrc" uzantılı dosyanın üzerine gelip bağlam menüsünden "Open With / Resource Editor" yapılırsa (zaten default durum budur) aşağıda Add isimli bir combobox görüntülenecektir. Buradan "Add Prefix" seçilerek bir örnek verilebilir. Örnekler "/" sembolü ile başlar. Ancak örnek hiç verilmeyebilir. Bu durumda örnek yalnızca "/" sembolünden oluşur. Bu bağlamdaki "örnek" adeta bir izin etkisi yaratmak için düşünülmüştür. Sonra "Add" combobox'ından "Add Files" seçilir. Buradan anlaşılabileceği gibi Qt'de her kaynak aslında bir dosyadır.

3) Oluşturulan “.qrc” uzantılı XML kaynak dosyası “rcc” denilen (resource compiler) bir programa sokulur. Bu program tüm bu kaynakları binary dosyada sıkıştırılmış olarak toplar. Bu binary dosya da çalıştırılabilen dosyanın ya da dinamik kütüphane dosyasının içerisine gömülmektedir. Ancak programcının bu rcc programını manuel olarak çalıştırması gerekmez. Zaten qmake programı “.pro” dosyası içerisindeki “RESOURCES” direktifini gördüğünde tüm bu işlemleri otomatik olarak yapmaktadır. Dolayısıyla QtCreator IDE’sinde de bizim ekstra bir işlem yapmamıza gerek kalmaz. Örneğin, kaynak kullanan bir qmake “.pro” dosyası şöyle olabilir:

```
QT += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = 022-ResourceUSage
TEMPLATE = app

SOURCES += main.cpp\
           mainwindow.cpp

HEADERS += mainwindow.hpp

FORMS += mainwindow.ui

RESOURCES += \
            myresources.qrc
```

4) Program qmake işlemine sokulduktan sonra artık kaynak dosyası rcc ile derlenip çalıştırılabilen dosyaya ya da dinamik kütüphane dosyasına yerleştirilmiş durumdadır. Şimdi sıra onu kullanmaya gelmiştir. Qt’de kaynaklar tamamen bir dosya gibi düşünülmüştür. Nasıl normal dosyaların bir yol ifadesi varsa kaynakların da bir yol ifadesi vardır. Kaynakları belirten yol ifadesi “:/” karakterleriyle başlamak zorundadır. Bu karakterlerle başlayan yol ifadesi QFile sınıfına verilirse QFile buradaki dosyayı gerçek dosya sisteminde aramaz, kaynakta arar. “:/” karakterlerinin kaynakta yol ifadesi belirtmesi şüphesiz işletim sistemi ya da C standartları genelinde bir durum değildir. Qt’nin kendi düzenlemesidir. Ancak Qt’nin dosya sınıfları bu özelliği destekleyecek biçimde yazılmışlardır. Qt’nin diğer sınıflarında dosya işlemleri hep QFile sınıfı ile yapıldığı için bu yol ifadesi Qt’nin diğer sınıflarında da kaynak belirtmek için kullanılabilir. Artık biz dosyanın içeriğini QFile nesnesi ile elde edebiliriz. Örneğin kaynağımızda “Message.txt” isimli bir dosya bulunuyor olsun. Biz bu dosyanın içindekileri şöyle elde edebiliriz:

```
QFile file(":/My/Message.txt");

if (!file.open(QIODevice::ReadOnly))
    return;

QTextStream ts(&file);
QString msg;

msg = ts.readAll();
QMessageBox::information(this, "Message", msg);

file.close();
```

İşte Qt’de bizden ne zaman bir dosyanın yol ifadesi istense biz normal bir disk dosyasının yol ifadesi yerine ‘:/’ ile başlatılmış olan kaynaktaki bir yol ifadesini verebiliriz. Benzer biçimde QDir sınıfı gibi sınıflar da tamamen kaynaktan çalışabilecek biçimde yazılmışlardır. Yani örneğin biz QDir nesnesini kaynaktaki bir yol ifadesiyle yaratıp onun içerisindeki dosyaların listesini alabiliriz.

QIcon Sınıfının Kullanımı

Qt’de küçük resimcikler QIcon sınıfıyla temsil edilmektedir. Her ne kadar sınıfın ismi QIcon olsa da aslında bu sınıf dosya formatı olarak “bmp”, “ico”, “png” gibi temel formatları da desteklemektedir. Bir QIcon nesnesi sınıfın yol ifadesini alan başlangıç fonksiyonuyla oluşturulabilir. Biz yol ifadesi olarak başı “:/” ile başlayan kaynakta bir dosyayı da verebiliriz. Qt kütüphanesinde pek çok sınıf bizden bir QIcon da istemektedir. QListWidget sınıfının tuttuğu

QListWidgetItem nesnelerinin birer QIcon elemanları da vardır. Böylece biz listeleme kutusunda bir yazının yanı sıra bir de küçük resimcik görüntüleyebiliriz.

QComboBox Sınıfı

QComboBox sınıfı klasik combobox penceresi oluşturmak için kullanılmaktadır. Genel kullanımı QListWidget sınıfına oldukça benzerdir. Nesne yine form editör kullanılarak ya da manuel biçimde yaratılır. Yine comboBox penceresine eleman eklemek için sınıfın addItem üye fonksiyonları kullanılır. Örneğin:

```
ui->m_comboBox->addItem(QIcon(":/Banana.png"), "Muz");
ui->m_comboBox->addItem(QIcon(":/Apple.png"), "Elma");
ui->m_comboBox->addItem(QIcon(":/Cherry.png"), "Vişne");
ui->m_comboBox->addItem(QIcon(":/Raspberry.png"), "Böğürtlen");
```

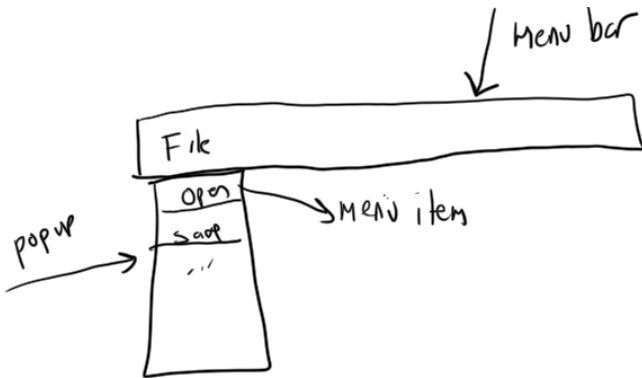
Yine seçili olan elemanın indeksi currentIndex fonksiyonuyla elde edilebilir. Seçili elemanın yazısı da currentText fonksiyonuyla elde edilebilir. ComboBox açılır açılmaz istenilen bir elemanın seçili durumda gözükmesi için setCurrentIndex fonksiyonu kullanılır.

QComboBox sınıfının yeni bir eleman seçildiğinde emit edilen "currentIndexChanged" isimli bir sinyali vardır.

Qt'de Menu Kullanımı

Menüler Windows gibi bazı işletim sistemlerinde ve pencere yöneticilerinde pencereye özgüdür ve pencere içerisinde görüntülenir. Mac OS X gibi bazılarında ise masaüstünde tek bir menü bulunmaktadır. Aktif uygulama değiştiğinde bu menü o uygulamanın menüsü olur.

Menülerdeki ana çubuğa menü çubuğu (menu bar) denilmektedir. Menü çubuğuna bağlı olan menü pencerelerine ise popup denir. Popup pencereler üzerinde menü elemanları vardır:



Menülerin ve araç çubuklarının QWidget penceresinde bulundurulması mümkün olsa da oldukça zahmetlidir. Bunun için ana pencere görevi yapacak QMainWindow sınıfı düşünülmüştür. Yani menü ve araç çubuğu içeren uygulamaların ana pencereleri için QWidget sınıfı değil, QMainWindow sınıfı kullanılmalıdır.

Menü çubuğu Qt'de QMenuBar sınıfıyla, Popup pencereleri, QMenu sınıfıyla, menü elemanları ise QAction sınıfıyla temsil edilmektedir. QAction nesneleri yalnızca menü elemanları olarak değil aynı zamanda araç çubuğu (toolbar) elemanları olarak da kullanılmaktadır. Menü elemanı seçildiğinde ya da araç çubuğu elemanına tıklandığında emit edilecek sinyaller QAction sınıfının içerisinde.

QMainWindow nesnesi yaratıldığında zaten bir QMenuBar nesnesi de yaratılmaktadır. Bu nesnenin adresini almak için QMainWindow sınıfının menuBar üye fonksiyonu kullanılır. Yani bizim ayrıca QMenuBar nesnesi yaratmamıza gerek yoktur.

Menüler tamamen Qt'nin form editörüyle oluşturulabilir. Ancak biz öncelikle manuel oluşturma üzerinde duracağız.

Bir QMenu nesnesi popup başlığında görüntülenecek yazı verilerek yaratılabilir. Bu nesnenin QMenuBar nesnesine eklenmesi için QMenuBar sınıfının addMenu fonksiyonu kullanılır.

```
QAction *QMenuBar::addMenu(QMenu *menu);
```

Örneğin:

```
QMenu *filePopup = new QMenu("File");
menuBar()->addMenu(filePopup);
```

Aslında ayrı bir QMenu nesnesi yaratıp addMenu yapmak yerine tek hamlede overload edilmiş aşağıdaki addMenu fonksiyonunu da kullanabiliriz. Bu fonksiyon zaten kendi içerisinde QMenu nesnesini yaratıp eklemektedir. Ayrıca yarattığı QMenu nesnesini bize de vermektedir:

```
QMenu *QMenuBar::addMenu(const QString &title);
QMenu *QMenuBar::addMenu(const QIcon &icon, const QString &title);
```

Örneğin:

```
QMenu *filePopup, *editPopup;

filePopup = menuBar()->addMenu("File");
editPopup = menuBar()->addMenu("Edit");
```

Menu elemanları için QAction nesneleri kullanılmaktadır. QMenu sınıfının addAction üye fonksiyonları QAction nesneleri yaratıp bunu QMenu nesnesine ekler. İki addAction overload fonksiyonu şöyledir:

```
QAction *addAction(const QString &text);
QAction *addAction(const QIcon &icon, const QString &text);
```

Örneğin:

```
QMenu *filePopup, *editPopup;
QAction *openItem;

filePopup = menuBar()->addMenu("&File");
editPopup = menuBar()->addMenu("&Edit");

openItem = filePopup->addAction("&Open");
```

QAction nesnesi icon da içerebilir. Örneğin:

```
openItem = filePopup->addAction(QIcon(":/MyIcons/Open.png"), "Open");
```

QAction sınıfının triggered isimli sinyali menü elemanı seçildiğinde emit edilir. Böylece bu sinyal yoluyla biz bir menü elemanı seçildiğinde uygun işlemleri yapabiliriz. Örneğin:

```
QObject::connect(openItem, SIGNAL(triggered(bool)), this, SLOT(onOpenItemTriggered()));
```

Menü elemanlarını checked ve unchecked yapabiliriz. Her seçildiğinde elemanın checked ya da unchecked olmasını sağlamak için sınıfın setCheckable fonksiyonunu true parametresiyle çağırmanız gerekir. Ayrıca sınıfın setChecked fonksiyonuyla biz elemanı checked ya da unchecked duruma getirebiliriz. Çarpılanacak menü elemanlarında icon bulunabilir. Ancak bu genellikle tercih edilen bir durum değildir.

Bir menü elemanı seçilmiş gibi etki gösteren tulara “kısa yol tuşları (shortcut keys)” denilmektedir. Qt’de kısayol tuşu oluşturmak için QAction sınıfının setShortcut fonksiyonu kullanılır. Bu fonksiyon parametre olarak QKeySequence

nesnesi almaktadır. QKeySequence sınıfının başlangıç fonksiyonu da tuşların isimlerinden hareketle nesneyi oluşturmaktadır. Örneğin:

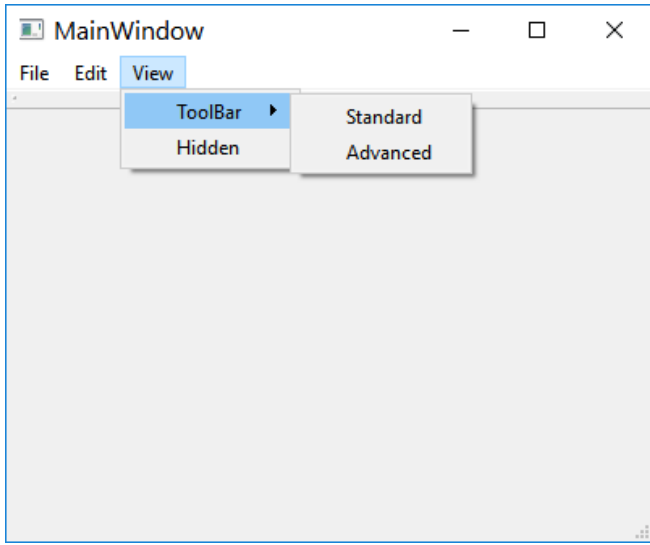
```
m_openItem->setShortcut(QKeySequence("Ctrl+O"));
```

Menü elemanlarındaki yazının belli bir karakterinin soluna & getirilirse o elemanın altı çizilir. Altı çizili eleman menü açıksa o tuşa basılarak seçilebilmektedir. Menü elemanlarının altı çizili görüntülenmesi için Windows sistemlerinde “Alt” tuşuna basmak gerekir.

Alt menüler (sub menus) nasıl oluşturulmaktadır? Eğer biz bir QMenu nesnesine onun AddMenu fonksiyonuyla popup menü eklersek eklenen popup artık o menünün alt menüsü olur. Örneğin:

```
m_viewPopup = menuBar()->addMenu("View");  
m_viewToolBarPopup = m_viewPopup->addMenu("ToolBar");  
m_toolBarStandardItem = m_viewToolBarPopup->addAction("Standard");  
m_toolBarAdvancedItem = m_viewToolBarPopup->addAction("Advanced");
```

Burada menü çubuğuna bir View popup eklenmiştir. View popup’ın içerisine de “ToolBar” isimli başka bir popup eklenmiştir. Görüntü şöyle olacaktır:



Bir menü elemanı QAction sınıfının setEnabled üye fonksiyonu ile pasif ya da aktif hale getirebiliriz. Pasif durumda olan menü elemanları seçilemez. Örneğin henüz bir dosya açık değilse Close seçeneğinin, açıksa da Open seçeneğinin pasif olması daha uygun olabilir.

Bir menü elemanını ya da popup pencereyi setVisible fonksiyonuyla tamamen yokmuş gibi görünmez yapabiliriz.

Menü elemanlarındaki yazının fontları da setFont fonksiyonuyla değiştirilebilir.

Menü İşlemlerinin Form Editör (Qt Designer) İle Yapılması

Aslında neredeyse hiç kod yazmadan form editör ile tüm menü sistemini oluşturabiliriz. Bunun için “designr” da menü girişleri fare ve klavye yoluyla yapılır. “Designer” her bir menü elemanı için bir QAction nesnesi oluşturur. Onu da aşağıdaki Action kısmında görüntüler. QAction nesneleri için slotlar benzer biçimde bağlam menüsünden girilebilmektedir. Ayrıca “QAction” nesnelerinin çeşitli özellikleri kolaylıkla “designer” dan değiştirilebilmektedir.

Araç Çubuklarının Kullanımı

Qt’de araç çubukları (toolbar) QToolBar sınıfıyla temsil edilmektedir. QMainWindow sınıfının addToolBar fonksiyonları yeni bir araç çubuğunu ana pencereye ekler. Bir pencerede birden fazla araç çubuğu bulunabilmektedir:

```
void addToolBar(QToolBar *toolbar);
QToolBar *addToolBar(const QString &title);
```

Birinci fonksiyonda QToolBar nesnesini biz yaratıp onun adresine addToolBar fonksiyonuna biz veririz. İkinci fonksiyonda biz araç çubuğuna bir isim veririz. QToolBar nesnesini bu fonksiyon yaratıp ekler. Bize de yaratılmış olan nesnenin adresini verir.

Eğer “designer” ile çalışılıyorsa “designer” zaten bir araç çubuğunu bizim için eklemiş durumdadır. “Designer” bu kendisinin eklediği içi boş araç çubuğu nesnesine “mainToolBar” simini vermiştir. Tabii biz bu ismi değiştirebiliriz.

Qt’de hem neülere hem de araç çubuklarına QAction nesneleri eklenir. Böylece bir işlem hem menü ile hem de araç çubuğuyla yapılacaksa aynı QAction nesnesi hem menüye hem de araç çubuğuna eklenmelidir.

Araç çubuğuna QAction nesnesini eklemek için QToolBar sınıflarının addAction fonksiyonları kullanılabilir. Ancak “designer”da “action editör”den bir action nesnesini sürükleyip araç çubuğuna bırakarak da ekleme işlemi görsel biçimde yapılabilmektedir.

Araç çubuğu elemanın üzerine gelinip bir süre beklendiğinde bir ipucu yazısı çıkmaktadır. Bu ipucu yazısı default durumda QAction nesnesinin kendi yazısıdır. Ancak bu yazı QAction sınıfında değiştirilebilmektedir.

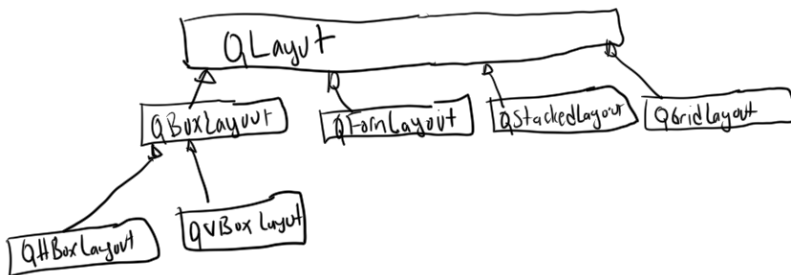
QMainWindow ve Central Widget Kavramı

QMainWindow sınıfı türünden bir nesne yaratıldığında oluşturulan ana pencerenin çalışma alanı başka bir pencereyle kaplanabilir. Çalışma alanını kaplayan bu pencereye “merkezi pencere (central widget)” denilmektedir. Merkezi pencere QMainWindow sınıfının centralWidget fonksiyonuyla alınıp setCentralWidget fonksiyonuyla set edilmektedir. Biz “designer”da QMainWindow ile bir uygulama oluşturduğumuzda “designer” zaten merkezi pencereyi QWidget sınıfı türünden bir pencereyle set etmiş durumda olur. Bu durumda biz QMainWindow yoluyla designer’da bir proje oluşturup oraya birtakım widget’lar yerleştirdiğimizde aslında bu widget’ların üst pencereleri bu merkezi pencere olmaktadır. Designer yarattığı bu merkezi pencereye ui->centralWidget ifadesiyle erişebiliriz. Tabii merkezi pencereyi temsil eden bu veri elemanın ismi de değiştirilebilmektedir.

Qt’de Layout İşlemleri

Qt’nin GUI tasarımlarında layout kavramı önemli yer tutmaktadır. Bu layout kavramı .NET’te WPF’de, Android’te de benzer biçimlerde kullanılmaktadır.

Qt’de QLayout sınıfından türetilmiş olan çeşitli layout nesneleri vardır. Layout nesneleri birer pencere belirtmez. Yani bunlar birer “widget” değildir. Bunlar “widget”ları ve diğer nesnelerini konumlandırmak için kullanılmaktadır. Biz burada bu layout nesnelerini tek tek inceleyeceğiz. Şüphesiz layout işlemleri yalnızca kodla ya da yapılabilir. “Qt Designer” kullanılarak da görsel olarak yapılabilir.



QHBoxLayout ve QVBoxLayout Kullanımı

QHBoxLayout yatay konumlandırma için QVBoxLayout ise dikey konumlandırma için kullanılmaktadır. Kullanım basittir. Bu layout sınıfları türünden bir nesne yaratılır. Sonra bu layout nesnelerinin QLayout sınıfından gelen addItem ve addLayout fonksiyonlarıyla bunlara layout ve widget nesneleri eklenir. Örneğin:

```
QHBoxLayout *hBoxLayout = new QHBoxLayout();

hBoxLayout->addWidget(ui->m_pushButtonOk);
hBoxLayout->addWidget(ui->m_pushButtonCancel);
hBoxLayout->addWidget(ui->m_pushButtonIgnore);
```

Qt’de her pencereye yalnızca bir tane layout nesnesi iliştilirilebilir. Bu işlem için QWidget sınıfının setLayout üye fonksiyonu kullanılmaktadır. Örneğin biz merkezi pencereye bu layout nesnesini şöyle iliştilirebiliriz:

```
ui->centralWidget->setLayout(hBoxLayout);
```

QHBoxLayout sınıfı kendisine eklenen widget ya da diğer layout nesnelerini yatay, QVBoxLayout sınıfı ise dikey olarak dizer.

Bir layout nesnesi başka bir layout nesnesini içerebilir. Ancak bir widget’a tek bir layout nesnesi iliştilirilebilmektedir. Bu durumda birden fazla layout nesnesini bir “widget”a iliştilirmek istiyorsak onları bir layout nesnesinde toplayıp topladığımız layout nesnesini “widget”a iliştiliriz. Örneğin:

```
QHBoxLayout *hBoxLayout1 = new QHBoxLayout();
QHBoxLayout *hBoxLayout2 = new QHBoxLayout();
VBoxLayout *vBoxLayout = new QVBoxLayout();

for (int i = 0; i < 5; ++i) {
    QPushButton *button = new QPushButton(QString::number(i));
    hBoxLayout1->addWidget(button);
}

for (int i = 5; i < 10; ++i) {
    QPushButton *button = new QPushButton(QString::number(i));
    hBoxLayout2->addWidget(button);
}

vBoxLayout->addItem(hBoxLayout1);
vBoxLayout->addItem(hBoxLayout2);

ui->centralWidget->setLayout(vBoxLayout);
```

Yukarıdaki bu işlemler “designer”da sürükleyip bırak işlemleriyle aynı biçimde yapılabilir.

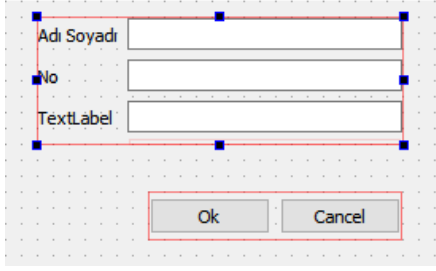
Layout nesnelerinin dört köşeye göre açıklıkları setLayoutMargins fonksiyonuyla ayarlanabilmektedir. “Designer”da bu ayarlama işlemi görsel olarak da yapılabilir. Ayrıca layout içerisindeki her eleman index numarası verilerek belli değerin katlarına çekilebilir. “Designer”da bu işlem layoutStretch girişiyle yapılabilir.

StackedLayout Kullanımı

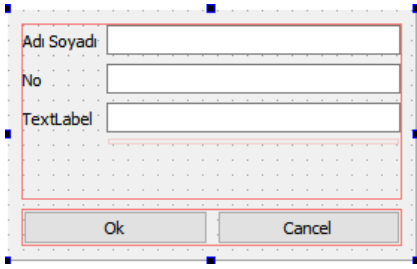
StackedLayout isimli layout sınıfı Qt’ye çok sonraları eklenmiştir. Hatta hala QtCreator IDE’si “designer”ına bu layout nesnesini eklememiştir. StackedLayout default durumda kendisine eklenen elemanları üst üste dizer. Bunlardan istenileni biz setCurrentWidget fonksiyonuyla öne alabiliriz. Yani aynı anda tek bir widget önde olacak biçimde gösterilmektedir.

FormLayout Kullanımı

FormLayout sınıfı ikili olarak widget'ları alt alta görüntülemektedir. Tipik olarak ikili satırların solundaki widget bir QLabel, sağdaki widget ise QLineEdit olur. Örneğin:



Burada bir FormLayout içerisine üç çift QLabel ve QLineEdit widget'ları yerleştirilmiştir. Aşağıdaki Ok ve Cancel tuşları da bir QHBoxLayout içerisine yerleştirilmiştir. Ancak ana pencerenin kendisine bir layout nesnesi iliştilmemiştir. Biz bu iki layout nesnesini ana pencereye bir QVBoxLayout yoluyla iliştirebiliriz:



Artık pencereyi genişletyip daralttığımızda widget'lar onlara göre konumlanırlar.

FormLayout kullanımında çiftin soldaki QLabel elemanın hizalaması değiştirilebilmektedir. Örneğin bu yazıları sola doğru ya da sağa doğru hizalayabiliriz.

GridLayout Kullanımı

GridLayout hücrelerden oluşan bir yapıdır. Biz de herhangi widget ya da başka layout nesnesini bu hücrelere yerleştirebiliriz. Kullanım için programcının önce formunu hücresel olarak ele alması gerekir. QGridLayout nesnesine widget ya da başka bir layout eklerken kullanılan addWidget ya da addLayout fonksiyonları bizden eklenecek elemanın yanı sıra onun hangi satır ve hangi sütundaki hücreye ekleneceğini de ister. Ayrıca bu fonksiyonlar yayılma (span) miktarını da bizden istemektedir. Örneğin 3'üncü satırın 2'inci hücresinden başlanarak bir widget'ın 2 hücre yer kaplaması sağlanabilir. Örneğin bir addWidget fonksiyonun parametrik yapısı tipik olarak şöyledir:

```
void addWidget(QWidget *widget, int fromRow, int fromColumn, int rowSpan, int columnSpan, Qt::Alignment alignment = Qt::Alignment());
```

Layout Nesnelerindeki Büyütme Küçültme Ayarları (Size Policy)

Layout nesnelerine birtakım widget'lar yerleştirildiğinde bu layout büyüdükçe ya da küçültüldükçe içerisindeki bu widget'ların genişleyen ya da daralan alanda ne kadar yer kaplayacağı QWidget sınıfından gelen sizePolicy fonksiyonuyla ayarlanmaktadır. Tüm widget'ların yine QWidget sınıfından gelen virtual bir sizeHint fonksiyonu vardır. sizeHint fonksiyonu bize QSize türünden genişlik, yükseklik içeren bir değer verir. Bir widget'ta sizeHint değeri bize o widget'ın mevcut durumuyla (örneğin içerisindeki yazının fontu vs. büyütülmüş olabilir) ne kadar genişlik ve yükseklik kaplaması gerektiğini belirtir. sizeHint değeri widget'ın size değeriyle aynı anlama gelmemektedir. size widget'ın o andaki gerçek genişlik ve yüksekliğidir. sizeHint ise onun iyi bir biçimde gözükmesi için tavsiye edilen ideal genişlik ve yüksekliğidir. İşte sizePolicy ayarlanırken hep sizeHint değeri dikkate alınarak ayarlamalar yapılmaktadır. Örneğin:

Ankara

İstanbul

Burada iki QPushButton nesnesi vardır. Soldakinin de sağdakinin de size değerleri (75, 23) biçiminde aynıdır. Soldakinin size değeri ile sizeHint değeri de aynıdır. Çünkü soldaki için ideal görüntü mevcut büyüklükle sağlanmaktadır. Halbuki sağdakinin sizeHint değeri size değerinden daha büyüktür (örneğimizde sağdakinin sizeHint değeri (104, 41)). Bunun nedeni düğmenin üzerindeki yazının fontunun daha büyük olmasıdır. Bu yazının hepsinin gözükmesi için sağdaki düğmenin olması gerektiği genişlik ve yükseklik onun sizeHint değeridir.

sizeHint değerinin yanı sıra widget'lar için bir de minimumSizeHint değerleri vardır. minimumSizeHint değeri widget'ın olabileceği en küçük değeri belirtir. Bazı widget'larda default durumda sizeHint ile minimumSizeHint aynı değerdedir. Maalesef sizeHint ve minimumSizeHint fonksiyonları sanal fonksiyonlardır. Bu değerleri değiştirebilmek için ilgili widget'tan türetme yapıp bu fonksiyonun override edilmesi gerekir.

widget'ların QWidget sınıfından gelen sizePolicy ve setSizePolicy fonksiyonlarının parametrik yapıları şöyledir:

```
QSizePolicy sizePolicy() const;  
void setSizePolicy(QSizePolicy::Policy horizontal, QSizePolicy::Policy vertical);
```

Görüldüğü gibi genişlik yükseklik ayarları QSizePolicy isimli bir sınıfla temsil edilmektedir. sizePolicy fonksiyonu bize mevcut genişlik yükseklik politikasını verir. setSizePolicy fonksiyonu ise bize yataydaki politikayla dikeydeki politikayı ayrı ayrı belirleme olanağı vermektedir. Tabii biz bu politika belirlemelerini “designer”da doğrudan görsel olarak yapabilmekteyiz. Yatay düşük politika türleri şunlardır:

Fixed: Bu durumda widget her zaman sizeHint ile belirtilen boyutta tutulur. Büyütülmez ya da küçültülmez.

Minimum: Bu seçenekte widget sizeHint'ten daha fazla küçültülemez, fakat büyütülebilir.

Maximum: Bu seçenekte widget sizeHint değerinden daha fazla büyütülemez ancak küçültülebilir. Fakat küçültme de ancak minimumSizeHint değerine kadar yapılabilir.

Preferred: Bu seçenekte widget minimumSizeHint değerine kadar küçültülebilir, ancak istenildiği kadar büyütülebilir.

Expanding: Bu seçenekte widget minimumSizeHint değerine kadar küçültülebilir, ancak istenildiği kadar büyütülebilir. Bu seçeneğin Preferred seçeneğinden farkı, bu widget'ların yerleştirildiği layout'lar büyütülünce eldeki boş alanı Preferred değil Expanding widget'ların paylaşmasıdır.

Minimum Expanding: Bu seçenekte widget sizeHint'ten daha fazla küçültülemez, fakat büyütülebilir. Bu özelliğe sahip widget'ların yerleştirildiği layout'lar büyütülünce eldeki boş alanı Preferred değil Expanding widget'lar paylaşmaktadır.

Ignored: Widget her zaman geri kalan alanı almaya çalışır.

Spacer Kullanımı

Layout işlemleri için iki tür spacer vardır: Horizontal ve vertical. Spacer'lar default olarak geri kalan alanı alan boş elemanlardır. Bunlar sayesinde biz birtakım öğeleri belirli yerlere yaslayabiliriz. Bu iki spacer nesnesinin de ayrıca sizePolicy özelliği vardır ve bunlar default olarak “Expanding” durumdadır. Böylece bunlar geri kalan alanı eğer başka Expanding olan widget yoksa her zaman alma eğiliminde olurlar

Diyalog Pencereleeri

Her zaman üst penceresinin yukarısında görüntülenen “sahiplenilmiş (owned)” pencerelere diyalog pencereleri denilmektedir. Diyalog pencereleri kendi aralarında “modal” ve “modeless” olmak üzere ikiye ayrılmaktadır. Modal diyalog pencereleri açıldığında artık kapanana kadar onun üst penceresi olan arka planla etkileşim kalmaz. Örneğin MessageBox pencereleri tipik olarak modal pencerelerdir. Halbuki modeless pencerelerde arka plan etkileşimi devam etmektedir. Örneğin “Find and Replace” pencereleri tipik olarak “modeless” diyalog pencereleridir.

Modal Diyalog Pencerelerinin Oluşturulması

Modal diyalog pencereleri şu aşamalardan geçilerek oluşturulur:

1) Diyalog pencerelerinin işlevsellikleri QDialog sınıfıyla sağlanmaktadır. Dolayısıyla öncelikle QDialog sınıfından bir sınıfın türetilmesi gerekir. Aslında bu türetmenin elle yapılmasına gerek yoktur. Qt Creator’da proje üzerine gelinip bağlam menüsünden “Add New” seçilip çıkan diyalog penceresinden “Qt/Qt Designer Form Class” ile bu işlem otomatik olarak yapılabilir. Add New diyalog penceresinde ilerlerken seçenek olarak “Dialog With Buttons” seçilmelidir.

2) Diyalog penceresi açılacağı zaman ilgili diyalog sınıfından bir nesne yaratılır ve bu nesneyle QDialog sınıfından gelen exec fonksiyonu çağrılır. Modal diyalog nesnesinin yerel olmasında bir sakınca yoktur.

```
void MainWindow::on_m_actionAddRecord_triggered()
{
    AddRecordDialog addRecordDialog;
    addRecordDialog.exec();
}
```

3) Diyalog penceresinin pencere başlığı QDialog sınıfının windowTitle fonksiyonuyla değiştirilebilir. Diyalog penceresini kapatmak için QDialog sınıfının done isimli fonksiyonu kullanılır. Tipik olarak bir modal diyalog penceresinde en azından Ok ve Cancel biçiminde iki düğme bulunur. İşte bu düğmelere tıklandığında done fonksiyonu çağrılmalıdır. done fonksiyonun parametresi QDialog penceresinin kapanma nedenini belirtir. Tipik olarak QDialog::Accepted ve QDialog::Rejected değerleri tercih edilmektedir. Bu değerler exec fonksiyonun geri dönüş değeri biçiminde elde edilmektedir. Tipik olarak diyalog penceresini açan programcı exec fonksiyonunun geri dönüş değerini kontrol etmektedir:

```
void MainWindow::on_m_actionAddRecord_triggered()
{
    AddRecordDialog addRecordDialog;

    if (addRecordDialog.exec() == QDialog::Accepted) {
        //...
    }
}
```

4) Pekiyi diyalog penceresinde oluşan bilgiler nasıl kullanılacaktır? Biz bu bilgileri doğrudan diyalog sınıfının içerisinden kullanabiliriz. Fakat daha çok bu bilgiler diyalog penceresini açan yerden kullanılmaktadır. done işlemi yapıldığında diyalog penceresinin içerisinde oluşan data’ların sınıfın veri elemanlarında saklanması uygun olur. Eğer bu veri elemanları private bölüme konulacaksa bunlar için get ve set fonksiyonlarının yazılması gerekir.

Modeless Diyalog Pencereleri

Aslında modeless diyalog pencerelerinin yaratılması modal pencerelere çok benzemektedir. Ancak yaratım exec fonksiyonuyla değil show fonksiyonuyla yapılır. Tabii bu durumda diyalog sınıf nesnesinin dinamik olarak yaratılması gerekir. Çünkü modeless pencereler yaratıldıktan sonra modal pencerelerde olduğu gibi mesaj döngüsünü kendi kontrolleri altına almazlar. Dolayısıyla yerel blok bittiğinde diyalog nesnesi için bitiş fonksiyonun çağrılmaması gerekir. Örneğin:

```
void MainWindow::on_m_actionModeless_triggered()
{
```

```

    m_mmd = new MyModelessDialog(this);
    m_mmd->show();
}

```

Modeless diyalog penceresi içerisinde arka plandaki ana pencere üzerinde işlemleri yapabilmemiz gerekir. Bunun için tipik olarak arkadaki ana pencerenin adresi modeless pencere sınıfına geçirerek o sınıfta saklanabilir:

```

MyModelessDialog::MyModelessDialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::MyModelessDialog)
{
    ui->setupUi(this);

    m_mainWindow = reinterpret_cast<MainWindow *>(parent);
}

```

Bundan sonra artık bu pencere sınıf nesnesiyle (örneğimizdeki m_mainWindow) işlemler yapılır. Modeless diyalog penceresi normal olarak close fonksiyonuyla kapatılmalıdır.

Qt'de Standart Diyalog Pencereleeri

Aslında dosya seçme, renk seçme, font seçme gibi bazı temel diyalog pencereleri zaten hazır olarak bulunmaktadır. Burada bu hazır diyalog pencerelerini ele alacağız. Bu pencerelere ilişkin sınıfların hepsi QDialog sınıfından türetilmiş durumdadır.

QFileDialog Sınıfının Kullanımı

QFileDialog sınıfının kullanımı çok basittir. Önce bu sınıf türünden bir nesne üst pencere adresi verilerek yaratılır. Sonra bu nesneyle sınıfın QDialog sınıfından gelen exec fonksiyonu çağrılır. Örneğin:

```

QFileDialog fileDialog(this);

if (fileDialog.exec() == QFileDialog::Accepted) {
    //...
}

```

Sınıfın selectedFiles isimli fonksiyonu bize seçili olan dosyaların yol ifadelerini QStringList biçiminde verir. Default durumda QFileDialog penceresi tek bir dosyanın seçimine izin vermektedir. O halde biz bu QStringList nesnesinin ilk elemanından seçilen dosyanın yol ifadesini alabiliriz. Örneğin:

```

QFileDialog fileDialog(this);

if (fileDialog.exec() == QFileDialog::Accepted) {
    qDebug() << fileDialog.selectedFiles()[0];
}

```

Diyalog penceresi alındığında başlık kısmındaki yazı istenildiği gibi set edilebilir. Bunun için QFileDialog sınıfının başlangıç fonksiyonundan faydalanılabilir. Örneğin:

```

QFileDialog fileDialog(this, "Bir dosya seçiniz");

```

Diyalog penceresi belli bir dizinle de açılabilir:

```

QFileDialog fileDialog(this, "Bir dosya seçiniz", "c:\\windows");

```

QFileDialog sınıfının setFilterNames fonksiyonu bizden filtreleme bilgilerini bir QStringList nesnesi olarak alır. Her filtreleme yazısı bir yazı ve parantez içerisinde joker karakterlerinden oluşmaktadır. Örneğin:

```

QFileDialog fileDialog(this, "Bir dosya seçiniz", "c:\\windows");

QStringList filters;
filters << "All File (*.*)" << "Text Files (*.txt)";
fileDialog.setNameFilters(filters);

if (fileDialog.exec() == QFileDialog::Accepted) {
    qDebug() << fileDialog.selectedFiles()[0];
}

```

Default olarak ilk filtreleme bilgisi seçili durumdadır.

Biz setFileMode fonksiyonuyla diyalog penceresinde yalnızca dizinlerin gösterilmesini sağlayabiliriz. Örneğin:

```

QFileDialog fileDialog(this, "Bir dosya seçiniz", "c:\\windows");
fileDialog.setFileMode(QFileDialog::DirectoryOnly);

```

Sınıfın diğer elemanları Qt dökümanlarından izlenebilir.

QFileDialog sınıfının çeşitli static fonksiyonları zaten kendi içlerinde QFileDialog nesnesi oluşturup exec işlemi yapmaktadır. Bu nedenle bunların kullanılması çok pratiktir:

```

QString getOpenFileName(QWidget *parent = Q_NULLPTR, const QString &caption = QString(), const
QString &dir = QString(), const QString &filter = QString(), QString *selectedFilter =
Q_NULLPTR, Options options = Options())

```

```

QStringList getOpenFileNames(QWidget *parent = Q_NULLPTR, const QString &caption = QString(),
const QString &dir = QString(), const QString &filter = QString(), QString *selectedFilter =
Q_NULLPTR, Options options = Options())

```

```

QString
getSaveFileName(QWidget *parent = Q_NULLPTR, const QString &caption = QString(), const QString
&dir = QString(), const QString &filter = QString(), QString *selectedFilter = Q_NULLPTR,
Options options = Options())

```

Bu fonksiyonlar başarı durumunda seçilen dosya ya da dosyaların yol ifadelerini geri dönüş değeri olarak verir. Eğer pencere iptal düğmesi ile kapatılırsa QString nesnesinin isNull fonksiyonu true ile geri döner. Örneğin:

```

QString path;

path = QFileDialog::getOpenFileName(this, "Bir dosya seçiniz", "c:\\windows", "All Files
(*.*)");

if (!path.isNull()) {
    //...
}

```

Renk Seçme diyalogPenceresi

Renk seçme işlemleri için QColorDialog sınıfı kullanılmaktadır. Bu sınıf da benzer biçimde kullanılır. Önce QColorDialog sınıfı türünden nesne tanımlanır. Daha sonra bu nesneyle exec fonksiyonu çağrılır. Örneğin:

```

QColorDialog cd(this);

if (cd.exec() == QColorDialog::Accepted) {
    //...
}

```

Seçilen renk sınıfın selectedColor fonksiyonuyla alınabilir. Bu fonksiyon seçilen rengi bize QColor olarak verir.

Renk seçme diyalog penceresi belli bir renk seçilmiş olarak açılabilir. Bu işlem sınıfın başlangıç fonksiyonunda yapılabilir ya da daha sonra setCurrentColor fonksiyonuyla yapılabilir. Örneğin:

```
QColorDialog cd(Qt::blue, this);

if (cd.exec() == QColorDialog::Accepted) {
    ui->m_textEdit->setTextColor(cd.selectedColor());
}
```

Qt isim alanında GlobalColor isimli enum türünün içerisinde bazı temel renkler vardır. Biz bunları Qt::blue, Qt::red gibi ifadelerle kullanabilmekteyiz.

QColorDialog sınıfının setWindowTitle isimli fonksiyonu diyalog penceresinin başlık yazısını oluşturmakta kullanılabilir.

QColorDialog sınıfının diğer elemanları Qt dokümanlarından incelenebilir.

Renk seçme diyalog penceresini tek bir fonksiyonla açmak için QColorDialog sınıfının içerisindeki getColor static fonksiyonu kullanılabilir:

```
QColor QColorDialog::getColor(const QColor &initial = Qt::white, QWidget *parent = Q_NULLPTR,
const QString &title = QString(), ColorDialogOptions options = ColorDialogOptions());
```

Font Seçme Diyalog Penceresinin Kullanımı

Font seçmek için QFontDialog sınıfı kullanılmaktadır. Yine diğer standart diyalog pencerelerinde olduğu gibi font seçme diyalog penceresinde de nesne yaratılıp exec fonksiyonu uygulanır. Örneğin:

```
QFontDialog fd(this);

if (fd.exec() == QDialog::Accepted) {
    //...
}
```

Seçilen font sınıfının selectedFont fonksiyonuyla alınabilir. QFontDialog sınıfının da QDialog sınıfından gelen setWindowTitle fonksiyonu ile pencere başlığı set edilebilir. Diğer standart diyalog pencere sınıflarında olduğu gibi QFontDialog sınıfında işlemi tek bir fonksiyonla yaptıran getFont isimli fonksiyonu vardır:

```
QFont getFont(bool *ok, const QFont &initial, QWidget *parent = Q_NULLPTR, const QString &title
= QString(), FontDialogOptions options = FontDialogOptions());
```

Örneğin:

```
QFont font;
bool resultOk;

font = QFontDialog::getFont(&resultOk, QFont("Segoe Print", 12), this, "Bir font seçiniz");
if (resultOk) {
    ui->m_textEdit->setFont(font);
}
```

Girdi Diyalog Penceresinin (Input Dialog) Kullanımı

Girdi diyalog penceresi bir “label” ve bir “line edit” pencerelerinden oluşmaktadır. Kullanım yine diğer diyalog pencerelerdeki gibidir. Nesne yaratılır ve exec fonksiyonu çağrılır. Örneğin:

```
QInputDialog id(this);
```



```
if (id.exec() == QDialog::Accepted) {
    //...
}
```

Dialog penceresindeki etiket yazısı setLabelText fonksiyonuyla değiştirilebilir. Örneğin:

```
QInputDialog id(this);
id.setLabelText("Pencere başlığını giriniz:");

if (id.exec() == QDialog::Accepted) {
    //...
}
```

Penceredeki “line edit” içerisindeki bilgi sayı olarak da yazı olarak da elde edilebilir. Girişi yazı olarak elde etmek için textValue fonksiyonu kullanılır. Girilen yazıyı sayısal olarak elde etmek için intValue ve doubleValue fonksiyonlarından faydalanılır. Girişin formatı sınıfın setInputMode fonksiyonuyla belirlenebilir. setIntMaximum, setIntMinimum gibi fonksiyonlar da girdi aralığını belirlemek için kullanılabilir. Örneğin:

```
QInputDialog id(this);
id.setLabelText("Pencere genişliğini giriniz:");
id.setInputMode(QInputDialog::IntInput);
id.setIntMaximum(700);

if (id.exec() == QDialog::Accepted) {
    resize(id.intValue(), height());
}
```

QInputDialog sınıfının da bir grup static fonksiyonu doğrudan dialog penceresini açmak için kullanılabilir:

```
double getDouble(QWidget *parent, const QString &title, const QString &label, double value = 0,
double min = -2147483647, double max = 2147483647, int decimals = 1, bool *ok = Q_NULLPTR,
Qt::WindowFlags flags = Qt::WindowFlags());
```

```
int getInt(QWidget *parent, const QString &title, const QString &label, int value = 0, int min
= -2147483647, int max = 2147483647, int step = 1, bool *ok = Q_NULLPTR, Qt::WindowFlags flags
= Qt::WindowFlags());
```

```
QString getItem(QWidget *parent, const QString &title, const QString &label, const QStringList
&items, int current = 0, bool editable = true, bool *ok = Q_NULLPTR, Qt::WindowFlags flags =
Qt::WindowFlags(), Qt::InputMethodHints inputMethodHints = Qt::ImhNone);
```

```
QString getMultiLineText(QWidget *parent, const QString &title, const QString &label, const
QString &text = QString(), bool *ok = Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags(),
Qt::InputMethodHints inputMethodHints = Qt::ImhNone);
```

```
QString getText(QWidget *parent, const QString &title, const QString &label,
QLineEdit::EchoMode mode = QLineEdit::Normal, const QString &text = QString(), bool *ok =
Q_NULLPTR, Qt::WindowFlags flags = Qt::WindowFlags(), Qt::InputMethodHints inputMethodHints =
Qt::ImhNone);
```

Slider Kullanımı

Slider miktar ayarlamak için sıklıkla tercih edilen widget’lardandır. Özellikle müzkte açmak gibi, bir rengi açmak gibi, bir kademeyi belirlemek gibi işlemlerde kullanılabilir. Slider QSlider sınıfıyla temsil edilmektedir. QSlider türünden bir nesne yaratılır. Pencerenin yatay mı, dikey mi olacağı sınıfın başlangıç fonksiyonunda belirtilir. Ancak “Qt designer”da sanki bu kontrol yatay ve dikey biçiminde iki tane varmış gibi temsil edilmektedir. Slider’ın maksimum ve minimum değerleri default olarak 0 ve 99 biçimindedir. Ancak sınıfın setMaximum ve setMinimum fonksiyonlarıyla ayarlanabilir. O anda yürütecini (thumb) konumu value fonksiyonuyla elde edilebilir. Klavye odağı

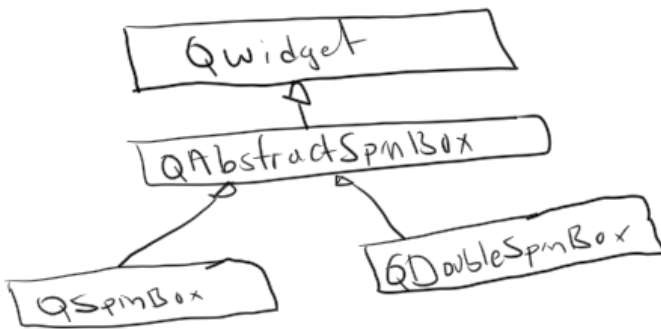
slider üzerindeyken ok tuşları ve “page up”, “page down” tuşlarıyla yürüteç hareket ettirilebilir. Ok tuşlarıyla hareketin kaç birim olacağı setSingleStep fonksiyonuyla “page up” ve “page down” tuşlarıyla hareketin kaç birim olacağı ise setPageStep fonksiyonuyla belirlenebilir. Yine tick’lerin sıklığı ve konumları belirlenebilmektedir.

Slider’da yürüteç konumlandırıldığında valueChanged isimli sinyal emit edilir.

Anahtar Notlar: Qt’nin “Signal Slot Ediörü” hiç kod yazmadan sinyal-slot bağlantısının yapılması için kullanılmaktadır. Bizeden bu editörde hangi nesnenin hangi sinyalinin hangi nesnenin hangi slotu ile bağlamacağı sorulmaktadır. Sinyal-Slot bağlantıları da “ui” kodları içerisinde XML olarak belirtilebilmektedir. setupUi fonksiyonu da bu bilgilerden hareketle connect fonksiyonlarını çağırıp bağlantıları yapmaktadır.

SpinBox Kullanımı

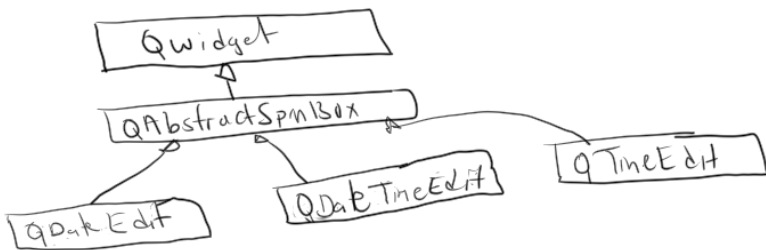
SpinBox (ya da spinner) artırmalı azaltmalı sayı oluşturmak için kullanılan bir penceredir. Genellikle bir şeyin miktarı konusunda belirleme yapılacağı zaman tercih edilmektedir. SpinBox Qt QSpinBox ve QDoubleSpinBox sınıflarıyla temsil edilmektedir. İki sınıfın ortak elemanları QAbstractSpinBox sınıfında toplanmıştır.



SpinBox pencerelerinde de o andaki sayısal değer value fonksiyonuyla alınır ve setValue fonksiyonuyla set edilir. Yine SpinBox nesnelerinin maximum ve minimum değerleri vardır. Default değerler [0,99] biçimindedir. Yine QSpinBox ve QSpinBoxDouble sınıflarının valueChanged isimli sinyalleri vardır. Bu sinyaller spinner üzerindeki değer değiştiğinde tetiklenmektedir. Bu nesneler read-only moda geçirilebilirler.

QDateEdit, QDateTimeEdit ve TimeEdit Widget’larının Kullanımı

QDateEdit, QDateTimeEdit ve QTimeEdit sınıfları QAbstractSpinBox sınıfların türetilmiştir. Yani bu sınıflar bir çeşit “spinner” davranışına sahiptir.



Sınıfların yine setMinimumDate ve setMaximumDate fonksiyonları edit alanında gösterilen tarihler için maximum ve minimum aralık belirlemesini yapar. Sınıfların date ve setDate fonksiyonları pencerede tutulan tarihi bize verip onun set edilmesini sağlar. QDateTimeEdit sınıfın ise dateTime ve setDateDateTime fonksiyonları hem tarih hem de zaman set etmek için kullanılmaktadır. Qt’de tarih bilgileri QDate sınıfı ile zaman bilgileri QTime sınıfı ile tarih ile zaman bilgileri ise QDateTime sınıfı ile temsil edilmektedir.

Sınıfların displayFormat fonksiyonları tarih ve zamanın nasıl görüntüleneceğini belirlemekte kullanılır. Sınıfların setCurrentSection fonksiyonları tarih ve zaman alanlarının hangi parçalarının aktif yapılacağıı belirlemekte kullanılmaktadır.

Progress Bar Kullanımı

ProgressBar bir işlemin ne kadarının geçip ne kadarının kaldığı konusunda bilgi vermek için kullanılmaktadır. Tipik olarak kurulum işlemlerinin GUI arayüzlerinde progress bar tercih edilmektedir. Progress Bar Qt’de QProgressBar sınıfıyla temsil edilmektedir.

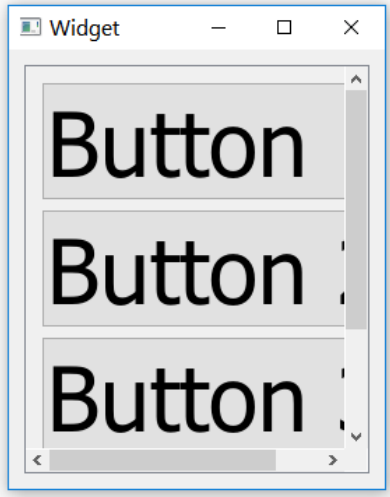
Sınıfın value ve setValue fonksiyonları doluluk değerini get ve set etmek için kullanılmaktadır. Sınıfın setMinimum ve setMaximum fonksiyonları en düşük ve en yüksek aralığı tespit etmek için kullanılır.

Progress bar’ın doldurulması belli olaylar gerçekleştiğinde yapılmaktadır. Doldurmanın bir mesaj sırasında yapılması programın donmasına yol açar. Bu nedenle genellikle progress kullanımında çoğu kez bir thread’de gerekmektedir.

QScrollArea Kullanımı

Bir horizontal ya da vertical layout nesnesi içerisine ya da bir widget’ın içerisine pek çok kontrol yerleştirdiğimizi düşünelim. Bu durumda bunların hepsi aynı anda ilgili alana sığmayacaktır. İlgili layout nesneleri bunları sığdırmak için otomatik olarak bunları küçültülecektir. Halbuki bazen biz bunu istemeyebiliriz. İşte bu tür işlemler Qt’de QScrollArea isimli bir widget yoluyla yapılmaktadır.

QScrollArea sınıfının içerisinde onun çalışma alanını kaplayan bir QWidget nesnesi vardır. Biz kendi widget’larımızı bu QScrollArea sınıfının içerisindeki QWidget nesnesinin içerisine yerleştiririz. Tabii bizim nesnelerimiz aynı zamanda bir layout nesnesinin içerisinde olmalıdır. İşte normal olarak layout nesneleri içerisindeki widget’ların sizePolicy’sine bağlı olarak onları büyütür ya da küçültür onları pencere içerisindeki tüm boş alanı kullanmasını sağlar. İşte anımsanacağı gibi pek çok sizePolicy’de küçültülecek minimum değer widget’ın minimumSize değeri kadardır. Pek çok standart widget bu minimumSize değerini içerisindeki bileşenlerden hareketle oluşturmaktadır. QScrollArea nesnesi içerisine yerleştirdiğimiz widget’lar eğer minimumSize değerine erişmişse kaydırma çubuklarının çıkmasına yol açmaktadır. Aksi durumda zaten widget daha fazla küçültülememektedir.



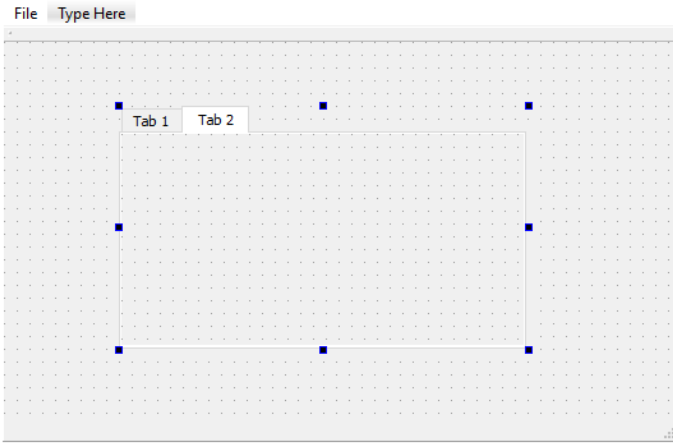
QTabWidget Kullanımı

Tab’lı arayüzler son zamanlarda çok yaygınlaşmıştır. Eskiden bu arayüzler yerine MDI (Multiple Document Interface) tarzı arayüzler tercih ediliyordu. Qt’nin QTabWidget sınıfı hem diyalog pencerelerindeki klasik “property sheet”leri oluşturmak için hem de tab’lı doküman arayüzlerini (tabbed document interface) oluşturmak için kullanılmaktadır.

QTabWidget kullanımı oldukça kolaydır. “Qt-Designer” bu kullanımı zaten kolaylaştırmaktadır. QTabWidget kullanımı şöyledir:

1) Bir QTabWidget nesnesi sınıfın başlangıç fonksiyonuyla yaratılıp ilgili pencereye eklenebilir. Bu işlem Qt-Designer’da sürükleyip bırak işlemi ile de yapılabilir. “Qt-Designer”da sürükleyip bırak ile QTabWidget oluşturulduğunda

başlangıçta kolaylık olsun diye “designer” iki tane tab’ı da bizim için yaratmış durumdadır. Biz onları istersek tab’ın üzerine gelip farenin sağ tuşuna basarak silebiliriz.



2) QTabWidget nesnesine yeni bir tab eklemek için sınıfın addTab fonksiyonları kullanılmaktadır:

```
int addTab(QWidget *page, const QString &label);
int addTab(QWidget *page, const QIcon &icon, const QString &label);
```

QTabWidget nesnesine dinamik olarak tab eklemek için bir QWidget nesnesine sahip olmamız gerekir. Biz içi boş yeni bir QWidget nesnesini new operatörüyle yaratıp onun içerisine birşeyler yeştirip onu tab olarak yukarıdaki fonksiyonlarla ekleyebiliriz. Ancak bu amaçla “designer”da kullanılabilir. Şöyle ki “Qt-Creator” da projeye yeni bir eleman eklenir. Eklenen eleman “Qt/Qt Designer Form Class” olarak seçilir. Bu durumda designer bize tıpkı diyalog pencerelerinde olduğu gibi bir “ui” dosyası olan bir sınıf oluşturacaktır.

3) O anda aktif olan tab’ın QWidget nesnesi currentWidget fonksiyonuyla elde edilebilir. O andaki aktif olan Tab2’nin indeks numarası ise currentIndex fonksiyonuyla alınabilir. İki tab’ın arasına da yeni bir tab eklenebilmektedir. Bu işlem de insertTab fonksiyonlarıyla yapılmaktadır:

```
int insertTab(int index, QWidget *page, const QString &label);
int insertTab(int index, QWidget *page, const QIcon &icon, const QString &label);
```

Belli bir tab setTabEnable fonksiyonuyla enable ya da disable yapılabilir.

Tab’a birer icon iliştirebiliriz. Bu işlem tab eklenirken de daha sonradan setTabIcon fonksiyonuyla da yapılabilir.

Tab şekilleri setTabShape fonksiyonuyla iki seçenektten biri olarak değiştirilebilir. QTabWidget::Rounded default durumdur. QTabWidget::Triangular ise üçgensel tab şeklini oluşturur.

setTabPosition fonksiyonu ile tab konumları değiştirilebilmektedir. Yine sınıfın setMovable fonksiyonu ile tab’ların fare ile sürüklenerek yer değiştirilmeleri sağlanabilir. Sınıfın setTabsClosable fonksiyonu ise tab’ların kapanabilir olmasını sağlar.

Sınıfın removeTab fonksiyonu belli bir indeks’teki tabı silmek için kullanılır. clear ise hepsini silmek için kullanılmaktadır.

4) QTabWidget sınıfının dört sinyal fonksiyonu vardır. Bunlardan en önemlileri tab’ın x tuşuna basıldığında tetiklenen CloseRequested fonksiyonu ve yenei bir tab’a geçildiğinde tetiklenen currentChanged fonksiyonudur.

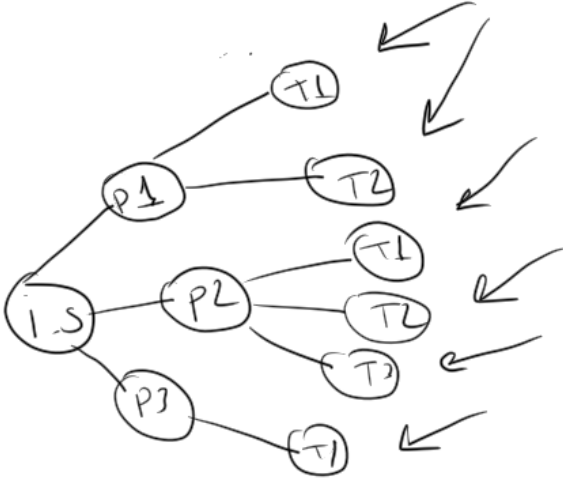
```
void currentChanged(int index);
void tabBarClicked(int index);
```

```
void tabBarDoubleClicked(int index);  
void tabCloseRequested(int index);
```

Qt'de Thread İşlemleri

Çalışmakta olan programlara process denilmektedir. Proses çalışmakta olan programın her şeyini temsil eder. (Örneğin çalışmakta olan programın açmış dosyalar, çevre değişkenleri, komut satır argümanları vs.) Thread ise yalnızca akış belirtmektedir. Bir proses normal olarak tek bir akışla çalışmaya başlar. Bu akış da bir thread belirtmektedir. Buna prosesin “ana thread’i (main thread)” denir. Prosesin ana thread proses yaratıldığında yaratılmaktadır. Diğer thread’ler proses çalışırken programcı tarafından çeşitli sistem fonksiyonları ya da bu fonksiyonları çağıran kütüphane fonksiyonları tarafından yaratılmaktadır. Örneğin Windows’ta “createThread API fonksiyonu, UNIX/Linux sistemlerinde pthread_create POSIX fonksiyonu thread yaratımında kullanılır. Thread’lerin yaratılması ve kullanılması sistemler arasında farklılıklar göstermektedir. Ancak Qt’de thread’ler platform bağımsız QThread sınıfıyla temsil edilmektedir. Tabii bu sınıfın fonksiyonları aslında hangi platformda çalışılıyorsa o platformun thread işlemleri yapan sistem fonksiyonlarını çağırır. Fakat biz bu sayede “cross platform” thread işlemleri yapabiliriz.

Normal olarak Windows gibi, Linux gibi, Mac OS X gibi sistemler çok prosesli çok thread’li sistemlerdir. Bu sistemlerde thread’ler işletim sistemi tarafından zaman paylaşım (time sharing) bir biçimde çalıştırılmaktadır. Bu sistemlerde bir thread belli bir süre çalıştırılır. Sonra çalışmasına ara verilir. Diğer thread’e geçilir. O thread belli süre çalıştırılır. Böyle “biraz ondan biraz bundan” biçiminde çalışma devam ettirilir.



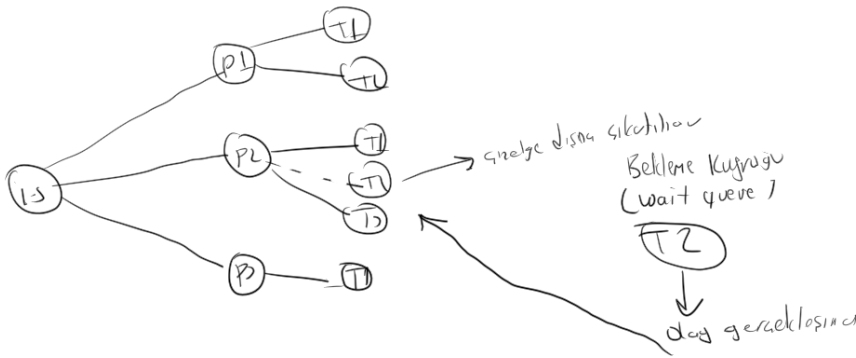
Bir thread bloke olup beklese bile diğer thread’lerin çalışması bundan etkilenmez. Bir thread’in parçalı çalışma süresine “quanta süresi” ya da “quantum” denilmektedir. Thread belli bir süre çalıştırılır, quanta süresi dolunca thread’in çalışması kesilir ve durumu saklanır. Sonra çalışma diğer thread’ten devam ettirilir. Dışarıda bakan kişi tüm thread’ler sanki aynı anda çalışıyormuş gibi bir izlenim edinir. Halbuki bunlar zaman paylaşım (time sharing) olarak çalıştırılmaktadır.

Birden çok işlemcinin ya da çekirdeğin bulunduğu durumda thread’lerin zaman paylaşım (time sharing) çalışması konusunda ciddi bir değişiklik olmamaktadır. Örneğin işletim sistemi yine her çekirdek için ayrı birer kuyruk oluşturup o çekirdeğe atanan thread’leri kendi aralarında yine zaman paylaşım (time sharing) çalıştırmaktadır. Tabii bu durumda şüphesiz birim zamanda yapılan iş miktarı artar ve herkesin programı toplamda daha hızlı çalışmış olur.

Bir thread’in quanta süresi ne kadar olmalıdır? Eğer bu süre çok uzun tutulursa karşılıklı etkileşim (interactivity) azalır. Eğer kısa tutulursa bu kez geçişler arasındaki zaman kaybı yapılan işe değmez hale gelir. Birim zamanda yapılan iş miktarı (terminolojide buna “throughput” denilmektedir) azalır. İşte işletim sistemleri makinenin de hızına bakarak quanta süresini baştan ayarlayabilmektedir. (Bazı sistemlerde bu ayarlama hiç yapılamaz, hep aynı quanta süreleri kullanılır.)

Thread’lerin Bloke Olması

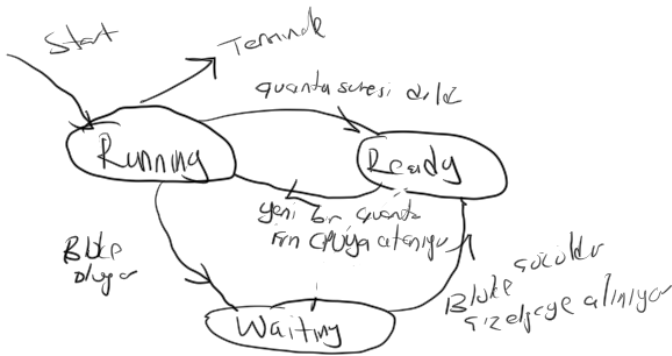
Bir thread kendisine ayrılan quanta süresini genellikle sonuna kadar kullanmaz. Bir thread göreceli olarak uzun süre beklemeye yol açacak bir işleme başladığında işletim sistemi o thread'i geçici olarak çizelge dışına çıkartır ve o olayı kendisi izler. Olay gerçekleştiğinde thread yeniden çizelgeye sokulmaktadır. İşte dışsal bir olayı başlatan bir thread'in geçici olarak çizelge dışına çıkartılması sürecine "thread'in bloke olması (blocking)" denilmektedir.



Pekiye blokeye yol açabilecek işlemler nelerdir? Bloke işlemi kernel modda işletim sisteminin sistem fonksiyonlarında ya da aygıt sürücüler tarafından gerçekleştirilmektedir. Blokeye yol açabilecek tipik işlemler şunlardır:

- Dosya işlemleri
- Soket işlemleri
- Klavye işlemleri
- Sleep gibi beklemeye yol açan işlemler
- Semaphore, Mutex gibi senkronizasyon işlemleri
- GUI programlamada mesaj kuyruğundan mesaj alan fonksiyonlar (örneğin mesaj kuyruğu boş olabilir, bu durumda yapılacak birşey yoktur ve bloke oluşabilir.)

Böylece bir thread yaşamını şöyle bir döngüde geçirmektedir:



Burada "Running" thread'in yeni bir quanta için CPU'ya atanmış olduğu anlamına gelir. "Ready" thread'in çizelgede olup yeni bir quanta için beklediğini belirtir. "Waiting" bloke durumunu belirtir. Yani artık thread çizelgede değildir. Bloke durumu çözülünce thread yeniden çizelgeye alınmaktadır.

Thread'ler quanta sürelerini kullanma yüzdelerine göre "IO yoğun (IO Bound)" ve "CPU yoğun (CPU bound)" olmak üzere ikiye ayrılmaktadır. IO yoğun thread'ler zamanlarının büyük bölümünü uykuda geçirirler. Bunların blokeleri çözüldüğünde kısa bir işlem yapıp yeniden uykuya dalarlar. Örneğin:

```
for (;;) {
    val = getKeyboardInt();
    process(val);
}
```

Halbuki CPU yoğun thread'ler quanta sürelerinin büyük bölümünü bloke olmadan CPU'da geçirirler.

Bir sistemde çok fazla thread'in olması çok ciddi yavaşlama olacağı anlamına gelmez. Bu thread'lerin çok büyük bölümü IO yoğun thread'lerdir ve aslında bunlar çok az CPU zamanı kullanmaktadırlar. Ancak çok sayıda CPU yoğun thread'ler ciddi anlamda sistem performansını düşürebilmektedir.

Şüphesiz zaman paylaşımli çalışmada programın iki noktası arasında geçen süre o andaki sistemin yoğunluğuyla doğrudan ilgilidir.

Thread'lere Neden Gereksinim Duyulmaktadır?

Thread'lere birkaç bakımdan gereksinim duyulmaktadır. Bunları maddeler haline ele alalım:

- 1) Thread'ler arka plan işlemlerin yapılmasına iyi derecede olanak sağlamaktadır. Örneğin bir akış devam ederken arka planda periyodik birtakım işlemlerin yapılması istenebilir. İşte bu işlemler bir thread'e havale edilirse programın organizasyonu çok daha kolay olur.
- 2) Thread'ler bir programın toplamda daha fazla CPU'dan pay alması için kullanılabilir. Tabii bu da programın daha hızlı çalışması anlamına gelir.
- 3) Thread'ler paralel programlama için mecburen kullanılmaktadır. Paralel programlama thread'lerin farklı CPU ya da çekirdeklere atanarak aynı anda çalıştırılmasını sağlamaya yönelik programlama modelidir.
- 4) Thread'ler GUI programlama modelinde bir mesajda uzun süre bir işlemin yapılması gerektiği durumda kullanılabilir. Şöyle ki: Biz bir mesajda uzun zaman alan bir işlem yaparsak bu durumda mesaj döngüsü işlemez, pencere sanki donmuş etkisi oluşur. İşte bu tür durumlarda uzun süren işlemi bir thread'e havale edip mesaj döngüsünün devam etmesinin sağlanması gerekir.

Thread'lerin Yaratılması

Qt'de thread işlemleri QThread sınıfıyla temsil edilmiştir. Qt'de bir thread akışını başlatma işlemi şöyle yapılmaktadır:

- 1) Programcı QThread sınıfından bir sınıf türetir ve bu sınıfta QThread sınıfındaki run isimli sanal fonksiyonu override eder. run fonksiyonunun geri dönüş değeri void türündendir, parametresi de yoktur:

```
virtual void run();
```

- 2) Programcı türettiği sınıf nesnesi kullanarak QThread sınıfının start isimli fonksiyonu çağırır. Bu start fonksiyonu thread akışını yaratarak run sanal fonksiyonu çağırılmaktadır. Böylece bu fonksiyon override edildiği için thread QThread sınıfından türetilmiş sınıfın run fonksiyonundan devam eder.

Bir thread nesnesine ilişkin thread akışının devam edip etmediği QThread sınıfının isRunning fonksiyonuyla anlaşılabilir. Benzer biçimde isFinished fonksiyonu da o anda ilgili nesneye ilişkin thread akışının sonlanıp sonlanmadığı bilgisini bize verir.

Thread'lerin Stack'lerinin Birbirlerinden Ayrılması

Bilindiği gibi yerel değişkenler ve parametre değişkenleri stack'te yaratılmaktadır. Thread'lerin de stack'leri birbirlerinden ayrılmıştır. Böylece iki thread akışı aynı fonksiyonda ilerlerken fonksiyonun yerel değişkenlerinin farklı kopyalarını kullanıyor durumda olur. Çünkü o yerel değişkenler hangi tarafından kullanılıyorsa o thread'in stack'inde yaratılmış durumdadır. Ancak global değişkenler, static yerel değişkenler ve heap alanı prosese özgüdür ve tüm thread'ler tarafından ortak kullanılmaktadır.

Thread'lerin Sonlandırılması

Bir thread start fonksiyonuyla çalışmaya başlatılınca run sanal fonksiyonu çağrılmaktadır. Bu fonksiyon sonlandığında thread akışı da sonlanmış olur. Thread'lerin sonlandırılmasının en doğal yolu run fonksiyonun bitmesidir. Event döngüsündeki thread'leri bu döngüden çıkartmak için quit ve exit fonksiyonları kullanılmaktadır. Bu konu ileride ele alınacaktır. Bir thread doğrudan thread nesnesi ile terminate fonksiyonu çağrılarak da sonlandırılabilir. Bu durumda aniden thread akışı sonlandırılmaktadır. Bu aslında tavsiye edilen bir durum değildir. Arka planda periyodik işlemler yapan thread'lerin hiç bloke olmadan meşgul bir döngü içerisinde beklemesi iyi bir teknik değildir. Böyle bir döngü oluşturulsa bile küçük bir miktar sleep işlemleriyle thread bloke edilmelidir. Qt'de bir thread akışının sonlandırılması thread nesnesinin yok edildiği anlamına gelmez. Thread akışı sonlandırıldıktan sonra aynı thread nesnesiyle start fonksiyonu çağrılarak yeniden bir thread akışı oluşturulabilir. Şüphesiz çalışmakta olan bir thread akışına sahip bir nesne üzerinde yeniden start işlemi yapmak anlamlı değildir. Gerçi bu durumda herhangi bir çalışma zamanı exception'ı oluşmaz. Bu durumda start fonksiyonu hiçbirşey yapmadan geri döner.

Thread nesnesinin thread akışı devam ettiği sürece yaşıyor durumda olması gerekmektedir. Eğer thread akışı devam ettiği halde thread nesnesi yok edilirse thread akışı yine de devam eder. Ancak bu akış içerisinde thread nesnesinin hiçbir elemanının kullanılmıyor olması gerekmektedir. (Bu tıpkı dinamik yaratılmış bir nesnesi ile bir onun bir fonksiyonu çağrıldığında onun içerisinde "delete this" ile nesneyi silmek gibi etki oluşturur.)

Bir thread sonlandığında nesne üzerinde finished sinyali tetiklenir. Benzer biçimde thread ilk çalıştırıldığında da started isimli sinyal tetiklenmektedir. Aslında start fonksiyonu da aynı zamanda bir slot fonksiyonudur. Örneğin biz bir thread bittiğinde otomatik olarak başka bir thread'i start edebiliriz.

QThread sınıfının wait fonksiyonu belli bir zaman aşımı geçene kadar ya da söz konusu thread akışı bitene kadar çağırılan thread'i bloke bekletmektedir.

```
bool QThread::wait(unsigned long time = ULONG_MAX);
```

Örneğin:

```
thread.wait();
```

Burada zaman aşımı vermediğimizden dolayı söz konusu thread bitene kadar akış bekletilmektedir.

Thread'lerin Senkronize Edilmesi

Birden fazla thread ortak bir kaynağı (bu kaynak herhangi bir biçimde olabilir. Örneğin bir sınıf nesnesi ya da bir container gibi) kullanırken kaynak kararsız durumda kalabilir. Örneğin bir thread bir vector'e eleman insert ediyor olsun. Insert işlemi için bir kaydırma gerekmektedir. Tam bu işlem yapılırken thread ona ayrılan quanta süresini doldup işletim sistemi tarafından "threadler arası geçiş (context switch)" yapıldığını varsayalım. Artık bu bağlı liste kararsız bir durumda kalacaktır. Diğer thread ona erişim yapmak istediğinde program çökebilecektir.

Kritik Kodların (Critical Sections) Oluşturulması

Başından sonuna kadar tek bir thread akışı tarafından işletilmesi gereken kodlara kritik kodlar denilmektedir. Bir thread akışı kritik koda girdiğinde başka bir thread akışı oraya girmek isterse CPU zamanı harcamadan bloke beklemelidir. Ta ki diğer thread kritik koddan çıkana kadar. Kritik kodlar ortak bir flag değişkeni ile oluşturulamazlar.


```

bool g-Flag;
//...
while ( g-Flag )
{
    //...
    g-Flag = true;
    //...
    g-Flag = false;
}

```

Kritik Kod

Bu kodun iki önemli kusuru vardır:

- 1) Akış döngüden geçtiğinde henüz flag true hale getirilmeden threadler arası geçiş oluşabilir. Bu durumda birden fazla thread kritik koda girebilir.
- 2) Burada bekleme meşgul bir döngüyle (busy loop) yapılmaktadır. Yani bekleyen thread CPU zamanı harcayarak bekler.

İşte bu nedenle kritik kodların oluşturulması işletim sisteminin sağladığı sistem fonksiyonlarıyla yapılmaktadır. Qt’de bu fonksiyonları kullanan çeşitli sınıflar bulundurulmuştur.

Mutex Nesnelerinin Kullanımı

Pek çok işletim sisteminde ve framework’lerde birbirine çok benzer kullanıma sahip mutex nesneleri bulunmaktadır. Mutex ismi “Mutual Exclusion” sözcüklerinde kısaltma yapılarak uydurulmuştur. Qt’de mutex kavramı “QMutex” sınıfı ile temsil edilmektedir. QMutex kullanımı şöyledir:

- 1) Senkronize edilecek iki thread’in görebileceği biçimde (örneğin global düzeyde ya da static veri elemanı olarak) bir QMutex nesnesi yaratılır.
- 2) Kritik kod şöyle oluşturulmaktadır:

```

QMutex g-mutex;
//...
g-mutex.lock();
//...
g-mutex.unlock();

```

Kritik Kod

Bir thread akışı lock fonksiyonundan geçmişse artık başka bir thread akışı lock fonksiyonunu çağırdığında bloke olur. Taki kilidi almış olan thread unlock fonksiyonuyla kilidi bırakana kadar. Böylece iki thread akışı aynı anda kritik koda girmemiş olurlar.

Mutex nesnesi özyinelemeli (recursive) olabilir ya da olmayabilir. Öztinelemeli mutex nesnesi lock edildiğinde yeniden aynı thread tarafından lock edilmek istenirse bloke oluşmaz. Ancak mutex’in lock edildiği kadar unlock edilmesi gerekir.

```

void foo()
{
    //...
    g_mutex_lock();
    // bar();
    g_mutex_unlock();
    //...
}

void bar()
{
    //...
    g_mutex_lock();
    //...
    g_mutex_unlock();
    //...
}

```

Mutex default olarak özyinelemeli değildir. Onu özyinelemeli yapmak için başlangıç fonksiyonunda buun belirtilmesi gerekir:

```
QMutex::QMutex(RecursionMode mode = NonRecursive);
```

Bazen biz kritik koda gireceksen eğer bloke olacaksak hiç girmemeyi yeğleyebiliriz. (Örneğin onun yerine başka bir iş yapmayı) İşte QMutex sınıfının trylock fonksiyonu bunu yapmaktadır:

```
bool QMutex::tryLock(int timeout = 0);
```

Kilitlenmiş bir mutex nesnesi kilidi almış thread tarafından unlock ile açılabilir. lock işlemini yapmamış bir thread unlock yapmaya çalışırsa unlock hiçbir etki göstermez.

Birden fazla thread aynı mutex nesnesi nedeniyle lock fonksiyonunda bloke olmuşsa kilit açıldığında hangi bekleyen thread'in kritik koda giriş yapacağı konusunda genel olarak işletim sistemleri bir garanti vermemektedir. Ancak mümkün olduğunca adil bir çizelgenin (yani pek çok koşul aynıysa ilk gelenin ilk hizmet alması) uygulandığı söylenebilir. Ancak bu konuda bir garanti verilmediği için programcı buna tasarımında dikkat etmelidir.

QMutexLocker Sınıfı

Bu sınıf yardımcı bir sınıftır. Sınıfın başlangıç fonksiyonu bizden bir QMutex nesnesini alır ve onu lock etmeye çalışır. Bitiş fonksiyonu da onu unlock eder. Böylece kritik kod kimi durumlarda aşağıdaki gibi oluşturulabilir:

```

{
    QMutexLocker locker(&g_mutex);
    // Kritik Kod
}

```

Bu sınıf özellikle try bloğu içerisinde bir mutex nesnesi ile lock işlemi yapıldığında otomatik unlock yapılması için kullanılmaktadır. Anımsanacağı gibi bir throw işlemi gerçekleştiğinde yerel değişkenler için de ters sırada bitiş fonksiyonları çağrılmaktadır.

Semafor Nesnelerinin Kullanımı

Semafor trenlerdeki dur-geç lambaları için kullanılmış olan bir sözcüktür. Edsger Dijkstra tarafından bulunmuştur. Semaforlar sayaçlı senkronizasyon nesneleridir. Tipik olarak bir kritik koda en fazla n tane akışın girebilmesini sağlamak için kullanılmaktadır. Örneğin tipik olarak programcının elinde n tane kaynak vardır. $k > n$ kadar thread bu

kaynağı bölüşmek ister. İlk n thread'e bu kaynaklara verilir. Fakat sonraki thread'ler kaynak kalmadığı için bloke bekletilir. Üretici-Tüketici problemlerinde “uyuyan berber (sleeping barbers)” gibi problemlerde, “yemek yiyen filozoflar (dining philosophers)” tarzı problemlerde semaforlar kullanılmaktadır.

Qt'de semaforlar QSemaphore sınıfıyla temsil edilmektedir. Semaphore kullanımı sırasıyla şu adımlardan geçilerek yapılmaktadır:

1) Thread'lerin ortak erişebileceği QSemaphore sınıfı cinsinden bir nesne yaratılır. (Örneğin tipik olarak global ya da bir sınıfın static veri elemanı biçiminde tanımlama yapılabilir.) Semafor sayacı sınıfın başlangıç fonksiyonunda verilir:

```
QSemaphore(int n = 0);
```

2) Kritik kod şöyle oluşturulmaktadır:

```
g_sem.acquire();  
[  
Kritik  
Kod  
]  
g_sem.release();
```

Semafor sayacı 0 ise semafor kilitlidir ve acquire fonksiyonu bu durumda giriş izni vermez, thread'i bloke bekletir. Semafor sayacı 0'dan büyükse kilit açıktır. acquire fonksiyonu geçiş izni verir ve otomatik olarak semafor sayacını 1 eksiltir. release fonksiyonu da semafor sayacını 1 artırmaktadır. Örneğin semafor sayacının başlangıçta 3 olduğunu düşünelim. Birinci thread kritik koda giriş yaptığında sayaç 2'ye düşecek, ikinci thread giriş yaptığında 1'e düşecek ve üçüncü thread giriş yaptığında da 0'a düşecektir. Artık 4'üncü thread acquire fonksiyonunda bloke edilerek bekletilecektir. Kritik koda girmiş thread'lerden biri kritik koddan release ile çıktığında sayaç 1 artırılıp yeniden 1 haline gelir. Bu durumda bekleyen thread'lerden biri kritik koda girebilir.

Semaforların thread temelinde sahipliği yoktur. Bir thread kritik koda girmiş olmasa bile bir semaforun sayacını release ile artırabilir. Aslında acquire ve release fonksiyonları default argüman almaktadır. Yani sayacı artırma ve eksiltme aslında birer birer değil, daha fazla fazla miktarlarda yapılabilmektedir. Tabii bu özel bir durumdur.

Semafor sayacı 1 olan semaforlara “ikili semaforlar (binary semaphores)” da denilmektedir. İkili semaforlar “mutex” nesnelere çok benzemektedir. Ancak tabii mutex nesnesinin sahipliği thread temelinde alınırken semaforların sahiplikleri thread temelinde değildir. (Yani bir mutex'i yalnızca lock etmiş thread unlock edebilir. Halbuki bir semaforun sayacını her thread istediği zaman artırabilir.) Yine QSemaphore sınıfında bir tryAcquire fonksiyonu da vardır. Bu fonksiyon QMutex sınıfındaki tryLock fonksiyonu gibi çalışmaktadır.

Üretici-Tüketici Problemi (Producer Consumer Problem)

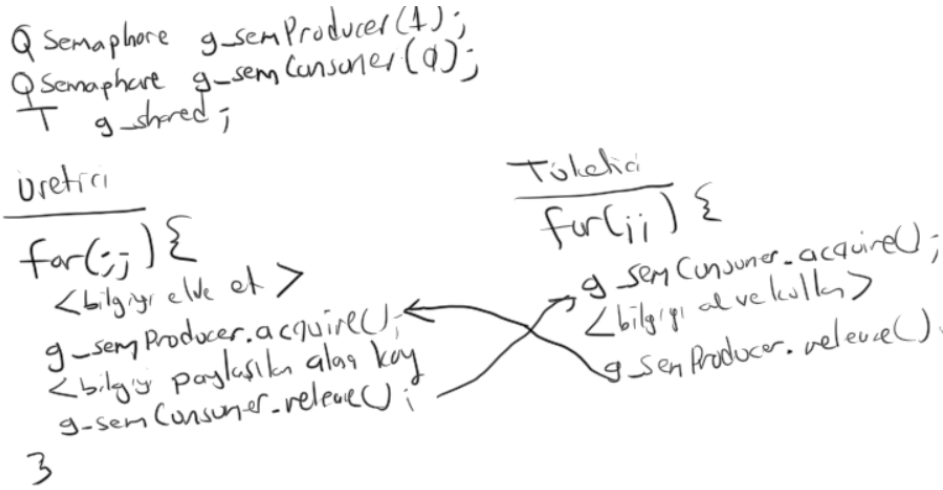
Üretici-Tüketici problemi gerçek hayatta en fazla karşılaşılan senkronizasyon problemlerinden biridir. Bu problemde bir thread'e üretici diğerine de tüketici thread denilmektedir. Üretici thread bir değeri elde eder, tüketici kullanabilir diye onu bir değişkene yazar. Tüketici de onu oradan alarak kullanır. Bu işlemler bir döngü içerisinde çok kez yapılmaktadır. Bu problemde üretici thread ve tüketici thread birbirlerinden bağımsız (asenkon) olarak çalıştığı için bazı anomaliler oluşabilmektedir. Örneğin üretici thread tüketici thread henüz eski değeri almadan yeni bir değeri paylaşılan değişkene yerleştirebilir. Bu durumda tüketici thread değer kaçırmış olur. Bazen biçimde tüketici thread de henüz üretici thread yeni bir değeri koymadan eski değeri yeniden alabilir. O halde öyle bir senkronizasyon mekanizması oluşturulmalıdır ki, üretici thread yeni değeri koymadan tüketici thread eski değeri yeniden almasın ve üretici thread de eski değeri tüketici thread almadan yeni bir koyarak eski değeri ezmesin

Bir işi parçalarına ayırırken bu tür senkronizasyon gereksinimleriyle çokça karşılaşılmaktadır. Örneğin bir thread bir dosya ismini bulur, diğerine verir, diğeri de o dosya içerisinde aramalar yapıyor olabilir. Bir thread satranç tahtasındaki mümkün hamleleri bulup diğeri bir thread'in pozisyon analizi yapması için ona verebilir.

Üretici-Tüketici probleminin çok üreticili ve çok tüketicili ve kuyruklu biçimleri de vardır.

Üretici-Tüketici Probleminin Semfor Nesneleriyle Çözümü

Üretici-Tüketici probleminin semafor nesneleriyle çözümünde iki semafor kullanılır. Üretici semaforun sayacı başlangıçta 1'e tüketici semaforun sayacı da 0'a konumlandırılır. Üretici yeni bilgiyi yerleştirirken tüketici de yeni bilgiyi alırken bu işlemleri semafor eşliğinde yaparlar. Üretici tüketicinin tüketici de üreticinin semafor sayacını artırmaktadır. Algoritma aşağıdaki şekilde özetlenebilir:



Üretici Tüketici Probleminin Tamponlu (Kuyruklu) Versiyonu

Üretici-Tüketici probleminde paylaşılan alanın bir elemanlık bir alan olması yerine çok elemanlı bir kuyruk sistemi olması işlemleri hızlandırır. Tabii kuyruk için de belli bir maksimum uzunluk önceden belirlenebilir. Böylece üretici thread yalnızca kuyruk tam dolu olduğunda tüketici thread de kuyruk tam boş olduğunda bekleyecektir.

Üretici Tüketici Probleminin Tamponlu (Kuyruklu) Versiyonunun Semafor Nesneleriyle Çözümü

Üretici-Tüketici probleminin tamponlu çözümünde üretici semafor, semafor sayacı tamponun büyüklük değeriyle ile yaratılır. Böylece tüketici hiç çalışmasa bile üretici en fazla tamponun büyüklüğü kadar değeri paylaşılan alana yerleştirecektir. Burada tampon sıranın bozulmaması için bir kuyruk sistemi biçiminde organize edilir. Eğer kuyruk sistemi için Qt'nin hazır bir sınıfı kullanılacaksa kuyruğa erişimin de bir mutex ile senkronize edilmesi gerekir. Eğer kuyruk manuel oluşturulacaksa buna gerek olmayabilir.

Okuma Yazma Kilitleri (Read/Write Locks)

Bazen bir grup thread ortak bir kaynağa (örneğin bir veri yapısına) yazma ve okuma yapmak amacıyla erişir. Tabii bu erişimin senkronize edilmesi gerekmektedir. Ancak birden fazla thread'in kaynağa okuma amaçlı erişmesi genel olarak bir soruna yol açmamaktadır. Fakat bir thread'in yazma amaçlı erişmesi kesinlikle kaynak bozulmasına yol açacak potansiyeldedir. Örneğin global bir bağlı listeye bir bazı thread'ler ekleme yaparken (yazma işlemi) bazıları da onda arama yapıyor (okuma işlemi) olsun. Bu durumda iki arama yapan thread'in bu işi yapmasında bir senkronizasyon sorunu oluşmaz. Ancak bir thread okuma yaparken diğeri herhangi biri yazma yapmamalıdır. Benzer biçimde bir thread yazma yaparken diğeri herhangi bir thread okuma ya da yazma yapmamalıdır. Eğer böylesi durumlarda mutex ya da semaforlarla senkronizasyon uygulanmaya çalışılırsa boşuna okuma yapan thread'ler de birbirlerini bekleyecektir. İşte bu özel durum için okuma yazma kilitleri oluşturulmuştur. Okuma yazma kilitleriyle kritik kod oluşturulurken bloke durumu aşağıdaki gibi ifade edilebilir. Burada kritik koda Birinci Thread'in girdiğini düşünelim. Bu tablo ikinci thread'in bloke durumunu vermektedir.

Thread 1	Thread 2	Thread 2'nin Bloke Durumu
Yazma	Yazma	Bloke
Yazma	Okuma	Bloke
Okuma	Yazma	Bloke
Okuma	Okuma	Bloke Yok

Okuma yazma kilitlerinde thread kritik koda hangi amaçla girdiğini belirtir. Eğer okuma amaçlı giriyorsa başka thread'lerin de yalnızca okuma amaçlı girmesine izin verilir. Ancak yazma amaçlı giriyorsa başka thread'leri okuma amacıyla yine de yazma amacıyla girişine izin verilmez.

Okuma yazma kilitleri Qt'de QReadWriteLock sınıfıyla gerçekleştirilmiştir. Sınıfın kullanımı şöyledir:

- 1) Thread'lerin erişebileceği bir global ya da sınıfın static elemanı biçiminde QReadWriteLock nesnesi tanımlanır.
- 2) Okuma amaçlı kritik kod şöyle oluşturulur:

```
g_readWriteLock.lockForRead();
[
// Kritik Kod
]
g_readWriteLock.unlock();
```

Yazma amaçlı kritik kod da şöyle oluşturulmaktadır:

QReadWriteLock sınıfının da yine tryLockForRead ve tryLockForWrite gibi blokeye yol açmayan fonksiyonları vardır. Ayrıca tıpkı QMutexLocker sınıfı gibi QReadWriteLock sınıfı için de QReadLocker ve QWriteLocker sınıfları vardır. Bunların başlangıç fonksiyonları bizden QReadWriteLock nesnesini alır, bitiş fonksiyonları da nesneyi unlock etmektedir.

```
g_readWriteLock.lockForWrite();
[
// Kritik Kod
]
g_readWriteLock.unlock();
```

Çok Thread'li Uygulamalarda volatile Belirleyicisinin Önemi

Derleyiciler optimizasyon seçenekleri kapatılsa bile nesneleri yazmaçlarda ya da geçici yerlerde saklayıp onları oradan alıp kullanabilmektedir. Örneğin:

```
x = a + b + 10;
y = a + b + 20;
```

Derleyiciler birinci deyimdeki a + b'nin toplamı zaten bir yazmaçtaydıysa ikinci, deyimde a ve b'ye yeniden erişmeden doğrudan bu toplamı kullanabilirler. Örneğin:

```
while (flag) {  
    //...  
}
```

Burada derleyici her defasında flag değişkenine bakmak yerine eğer o yazmaçtıysa onu oradan alıp kullanabilir. Yani flag nesnesine hiç erişmeyebilir. Tabii bunun için bu değişkenin değerinin değişmediğine emin olması gerekir.

Derleyicilerin optimizasyonları tek thread’li bir akış dikkate alınarak yapılmaktadır. Halbuki çok thread’li dünyada derleyicinin ürettiği optimize edilmiş kod bizim programımızın yanlış çalışmasına yol açabilir. Örneğin:

```
x = a + b + 10;  
----> Başka bir thread’in a ya da b’yi değiştirdiğini düşünün  
y = a + b + 20;
```

Burada iki deyim arasında başka bir thread’in a ya da b’yi değiştirmesi y’deki değeri etkilemektedir. Ancak derleyiciler bu durumu dikkate almadan optimizasyon yapabilmektedir. Bu durumda bu değişimden derleyiciinn ürettiği kod etkilenmeyebilecektir. Ya da örneğin:

```
while (g_flag) {  
    //...  
}
```

Burada eğer döngü içerisinde derleyici g_flag değişkeninin hiç değişmediğini görürse her defasında belleğe erişmek yerine yazmaç tuttuğu değeri kullanabilir. Yani kod derleyici tarafından aşağıdaki gibi derlenebilir:

```
reg = g_flag;  
while (reg) {  
    //...  
----> Burada g_flag başka bir thread tarafından 0’a set edilse bile döngüden çıkılamayabilecektir.  
}
```

volatile belirleyicisi C’de derleyiciye şunu emretmektedir: “Bak derleyici bu değişkeni ben her kullandığımda git onu belleğe başvurarak taze taze oradan al, yazmaçta onun değeri bulunuyor olsa bile onu oradan alma”.

İşte biz de ortak kullanılan nesneleri volatile bildirmeliyiz. volatile nesneler her defasında yeniden belleğe başvurularak oradan alınırlar.

```
volatile int g_flag;
```

Atomiklik Kavramı

Bir işlemin hiç kesilmeden (thread’ler arası geçiş oluşmadan) tek hamlede yapılması durumuna atomiklik denilmektedir. Makine komutlarının hepsi atomiktir. Yani bir makine komutu bitmeden onun ortasında thread’ler arası geçiş oluşmaz. Ancak bir grup makine komutu çalıştırılırken komutlar arasında kesilme olabilir. Aşağıdaki kod iki thread’in çalıştırdığını düşünelim:

```
for (int i = 0; i < 10000000; ++i) {  
    ++g_count;  
}
```

Normalde iki thread de g_count değişkenini on milyon kere artırdığına göre bunun sonucunda değişkenin değerinin yirmi milyon olması gerekir. Fakat muhtemelen böyle olmayacaktır. Peki neden?

Eğer buradaki artırım tek bir makine komutuyla yapılsaydı tek işlemcili ya da çekirdekli sistemlerde bir sorun oluşmayacaktı. Fakat derleyici işlem tek bir operatörle yapıldı diye onu tek bir makine komutuyla ifade etmek zorunda değildir. Örneğin bu artırım derleyici tarafından 32 bit Intel işlemcilerinde aşağıdaki gibi makine kodlarına dönüştürülüyor olabilir:

```
MOV     EAX, g_count
----> Tam bu noktada thread'ler arası geçiş oluyorsa ne olur?
INC     EAX
MOV     g_count, EAX
```

Burada thread'ler arası geçiş ters bir noktada oluyorsa değişkenin artırılmasında sorun ortaya çıkar.

Bu tür örneklerde değişkeni volatile yapmak bir çözüm oluşturmaz. Çünkü volatile “bu işlemi tek bir makine komutuyla atomik yap” anlamına gelmemektedir. O halde bu tür masum işlemleri bile senkronizasyon ile gerçekleştirmeliyiz. Örneğin bir mutex kullanılabilir:

```
for (int i = 0; i < 10000000; ++i) {
    g_mutex.lock();
    ++g_count;
    g_mutex.unlock();
}
```

Tabii mutex kullanımı önemli bir yavaşlamaya yol açabilmektedir. İşte bazı basit tek makine komutuyla yapılabilecek işlemler için bazı sınıflar bulundurulmuştur. Bu sınıfların üye fonksiyonları atomik bir biçimde o işlemi yaparlar. Bu sayede mutex kullanımı elimine edilmiş olur.

Çok İşlemcili ya da Çok Çekirdekli Sistemlerde Makine Komutlarının Atomikliği

Çokişlemcili ya da çok çekirdekli sistemlerde belleğe erişen makine komutları atomik olmak zorunda değildir. Örneğin Intel işlemcilerinin kullanıldığı SMP (Symmetric Multiprocessor) sistemlerinde iki çekirdek aynı anda aynı bellek bölgesine eriştiğinde (hizalamaya da bağlı olarak) bu işlemler atomik olmaktan çıkabilmektedir. Bu durumda örneğin iki ayrı çekirdekte çalışan program aynı anda aynı bellek bölgesine eriştiğinde bozuk bir verinin oluşabilme olasılığı çok düşük olsa da vardır. Örneğin:

```
g_a = 0;
/* Birinci Thread */
g_a = 123;

/* İkinci Thread */
x = g_a;
```

Burada ikinci g_a'yı okurken 0 ya da 123'ün dışında bozuk bir değer de okuyabilir. (Bu durum çok düşük bir olasılıkla gerçekleşir ancak böyle bir olasılık vardır). İşte bu tür durumlarda volatile anahtar sözcüğünün bir etkisi yoktur. mutex gibi bir senkronizasyon sorunu çözer ancak bu çözüm bir yavaşlamaya da yol açabilmektedir. Intel mimarisinde (ve başka mimarilerde de değişik biçimlerde) bu tür makine komutlarının atomikliğini sağlamak için komutun önüne “lock” öneki getirilmektedir. Bu öneki gören işlemci diğer işlemcileri ya da çekirdekleri durdurarak erişimi yapar. Böylece atomiklik sağlanmış olur. İşte Qt'de başına lock öneki getirilerek tek bir makine komutuyla çok işlemcili ya da çok çekirdekli sistemlerde temel işlemleri yapabilen sınıflar oluşturulmuştur.

Qt'de QAtomicInteger<T> sınıfı kendi içerisinde bir değer tutar ve bu değere “lock” önekiyle artırma eksiltme, ve'leme, veya'lama gibi işlemleri yapar. Sonra da istediğimizde bu güncellenmiş değeri bize geri verir. Örneğin:

```
QAtomicInteger<int> g_a(10);
g_a.fetchAndAddAcquire(1);
qDebug() << (int)g_a;
```

Ekleme için fetchAndAddXXX fonksiyonları kullanılabilir. Burada XXX yerine Acquire, Ordered ve Relaxed sonekleri gelebilmektedir. Bu ekler işlemcinin “komut yeniden düzenleme (instruction reordering)” özelliği ilgilidir. Bunlardan herhangi biri kullanılabilir. Yukarıda da açıklandığı gibi yalnızca değer atama işleminin bile çok işlemcili ya da çok çekirdekli ortamlarda atomik yapılması gerekir. Bunun için sınıfın store isimli fonksiyonu kullanılır. Değeri almak için load fonksiyonu ya da T türüne dönüştürme yapan tür dönüştürme operatör fonksiyonu kullanılabilir.

QAtomicInteger sınıfında şablon parametresinin tamsayı türlerine ilişkin olması zorunludur. Ayrıca QAtomicInteger sınıfından türetilmiş fakat şablon olmayan QAtomicInt sınıfı da vardır. Bu sınıfın kullanımı aynı biçimdedir. Yalnızca bu sınıfın tuttuğu nesnenin türü int'tir.

Qt'de bu iki sınıfın dışında ayrıca QAtomicPointer isimli bir sınıf da vardır. Bu sınıf göstericilerle benzer işlemleri yapar.

Fonksiyonların Thread Güvenliliği (Thread Safety) ve Yeniden Girişliliği (Reentrancy)

Bir fonksiyon aynı anda birden fazla thread tarafından çağrıldığında hiçbir sorun çıkmıyorsa o fonksiyona thread güvenli (thread safe) fonksiyon denilmektedir. Yalnızca yerel değişken kullanan fonksiyonlar farklı thread'lerden aynı anda çağrılırsa da soruna yol açmazlar. Onlar thread güvenlidirler. Ancak static yerel değişken ya da global değişken kullanan fonksiyonlar thread güvenli değildirler. Yani onların iki ayrı thread'ten iç içe çağrılmaları soruna yol açabilir. İşte Qt'deki fonksiyonların thread güvenli olup olmadığını bilmek gerekir. Eğer fonksiyon thread güvenli değilse ve birden fazla thread'ten aynı anda kullanılıyorsa bozulma oluşacağından bu çağrıların mutex bir nesneyle senkronize edilmesi gerekir.

Qt'de bir fonksiyonun thread güvenli olması ile yeniden girişli (reentrant) olması farklı anlamlara gelmektedir. Eğer sınıfın static olmayan bir üye fonksiyonu aynı nesne ile farklı thread'lerden çağrıldığında bile soruna yol açmıyorsa bu üye fonksiyon thread güvenlidir. Fakat aynı nesne ile farklı thread'lerden çağrıldığında soruna yol açıyor ancak farklı nesnelerle çağrıldığında soruna yol açmıyorsa bu fonksiyon yeniden girişlidir. Her thread güvenli static olmayan fonksiyon aynı zamanda yeniden girişlidir. Ancak her yeniden girişli fonksiyon thread güvenli değildir. Örneğin QVector<T> sınıfının append üye fonksiyonu yeniden girişli fakat thread güvenli değildir. Yani biz aynı nesne ile append fonksiyonunu farklı thread'lerde çağırırsak bozulma oluşabilir. Ancak farklı nesnelerle çağırırsak bir sorun oluşmaz.

Sınıfların üye fonksiyonlarının thread güvenli ya da yeniden girişli olup olmadığı Qt dokümanlarında belirtilmektedir.

GUI Thread'ler ve İşçi Thread'ler

Microsoft en az bir pencere (widget) yaratan thread'lere GUI thread hiç pencere yaratmamış olan thread'lere ise işçi thread (worker thread) denilmektedir. Biz aslında bu zamana kadar hep işçi thread'lerle çalıştık. İşte hem QApplication sınıfının (aslında bu sınıfın taban sınıfından gelmektedir) hem de QThread sınıfının exec isimli fonksiyonları vardır. Bu fonksiyonlar mesaj döngüsü oluşturmaktadır. QApplication sınıfının exec fonksiyonu üzerinde daha önce durmuştuk. Burada QThread sınıfının da bir exec fonksiyonu olduğunu görüyoruz.

Qt'de QApplication sınıfının yaratıldığı thread'e GUI thread denilmektedir. Qt tasarımına göre tüm widget'lar bu thread tarafından yaratılmalıdır. Ancak biz sonraki konuda da ele alınacağı gibi bir işçi thread'in GUI thread'teki pencereleri güncellemesini sağlayabiliriz.

Qt'de Threadler Arası İşlemler

Qt'de aslında mesaj döngüsü (ya da olay döngüsü) yalnızca GUI uygulamaları tarafından kullanılan bir kavram değildir. Bir console uygulaması da mesaj döngüsüne sahip olabilir. Bir thread'ten diğer bir thread'e veri aktarmak bu mesaj döngüsü için içine sokularak düzenli biçimde yapılabilir. Örneğin en çok karşılaşılan isteklerden biri bir thread'in başka bir GUI thread'in yaratmış olduğu widget üzerinde birtakım güncellemeler yapmasıdır. (Örneğin bir işçi thread birtakım değerleri elde edip GUI penceresine yazdırmak isteyebilir.) Thread'ler arasında pencerelerin kullanımı işletim sistemi düzeyinde de önemli bir konudur. Çünkü tasarımda pek çok pencere kendi işlemlerini mesaj döngüsü yoluyla yapar. Mesaj döngüleri de thread temelinde çalışmaktadır. Thread'ler arası GUI işlemleri her platformda değişik bir tasarımla çözülmeye çalışılmıştır. Qt'nin de bu konuda kendine özgü bir çözüm yöntemi vardır.

Qt'de bir işçi thread'in GUI thread'teki bir pencere üzerinde güncelleme yapması dolaylı bir biçimde sinyal/slot mekanizması yoluyla ya da aşağı seviyeli "event" yöntemi yoluyla yapılmalıdır. "Event"ler sonraki bölümde ele alınacaktır. Sinyal/Slot mekanizması yoluyla pencereler üzerinde güncelleme şöyle yapılmalıdır:

- 1) İşçi thread sınıfında bir sinyal tanımlanmalıdır.
- 2) İşçi thread sınıfındaki sinyal GUI thread'teki bir sınıfın slot'una bağlanır.
- 3) İşçi thread bu sinyali emit eder (tetikler). Böylece bu slot içerisinde de ilgili GUI pencere işlemi yapılır. Aslında sonuç olarak bu slot çağırması GUI thread tarafından yapılacaktır.

Qt'de Thread Havuzlarının Kullanımı

Bazı IO yoğun uygulamalarda bir istek geldiğinde sıradaki isteği bekletmemek için o istek bir thread yaratılarak o thread'e havale edilmektedir. Özellikle TCP/IP Client-Server uygulamalarda server programlarda bu duruma sıklıkla karşılaşılmaktadır. Örneğin server thread bir client'tan istek alır, bunu doğrudan yapmak yerine bir thread oluşturarak onun yapmasını sağlar. Böylece server diğer client'lardan gelecek isteklere daha hızlı yanıt verebilmektedir. İşte bu tür uygulamalarda kısa süreli çalışan pek çok thread'in yaratılıp yok edilmesi gerekebilmektedir. Thread'lerin yaratılması ve yok edilmesi aslında işletim sistemi düzeyinde görece olarak uzun zaman alan işlemlerdir. Aynı zamanda thread'ler önemli ölçüde diğer kaynakları da kullanabilmektedir. İşte bu tür uygulamalarda thread havuzları sayesinde bir grup thread yaratılmış olarak "suspend durumda" havuzda bekletilir. Programcı bir thread'e gereksinim duyduğunda yaratılmış olan thread akışı programcıya verilir. Programcının thread ile işi bittiğinde thread yok edilmez. Yeni istekler için havuzda bekletilir. Böylece ikide bir thread'lerin yaratılması ve yok edilmesi maliyetlerinden kaçınılmış olur.

Thread havuzları Windows işletim sisteminde işletim sistemi düzeyinde API fonksiyonlarıyla desteklenmektedir. Ancak Linux, BSD ve Mac OS X sistemlerinde işletim sisteminin böyle bir desteği yoktur. Thread havuzları çeşitli framework'ler tarafından o framework'lere özgü bir biçimde sınıfsal olarak gerçekleştirilmektedir. Örneğin Java ortamında, .NET ortamında, Qt ortamında thread havuzlarını oluşturan ve bunlarla işlem yapılmasını sağlayan hazır sınıflar vardır.

Thread havuzlarının az sayıda thread'in uzun süre kullanıldığı uygulamalarda bir faydasının olduğu söylenemez. Yukarıda da açıklandığı gibi, çok sayıda kısa ömürlü thread'in yaratılıp yok edildiği sistemlerde thread havuzları performansı ciddi biçimde artırabilmektedir.

Qt'de thread havuzları QThreadPool isimli sınıfla gerçekleştirilmiştir. QThreadPool sınıfının kullanımı kabaca şöyledir:

- 1) QThreadPool nesnesi "singleton" kalıbı ile yaratılmaktadır. Yani toplamda aslında Qt'de tek bir QThreadPool nesnesi vardır. QThreadPool sınıfının static globalInstance isimli üye fonksiyonu bize bu nesneyi vermektedir. O halde biz öncelikle bu QThreadPool nesnesini aşağıdaki gibi elde etmeliyiz:

```
QThreadPool *threadPool = QThreadPool::globalInstance();
```

- 2) Thread havuzdan alınan thread QRunnable isimli sınıftan türetilmiş bir nesne üzerinde onun run üye fonksiyonunu çağırarak işlemine başlamaktadır. Bu durumda bizim öncelikle QRunnable sınıfından bir sınıf türetmemiz gerekmektedir. QRunnable sınıfıyla QThread sınıfının bir türetme ilişkisi yoktur. QRunnable soyut bir sınıftır.

- 3) Daha sonra QRunnable sınıfından türetilip run fonksiyonu override edilmiş sınıf türünden bir nesne dinamik olarak yaratılır ve onun adresiyle QThreadPool sınıfının start fonksiyonu çağırılır. Örneğin:

```
QThreadPool *threadPool = QThreadPool::globalInstance();
MyThread *myThread = new MyThread();
threadPool->start(myThread);
```

Aslında bu işlem tek satırda da aşağıdaki gibi yapılabilirdi:

```
QThreadPool::globalInstance()->start(new MyThread());
```

Yaratılan QRunnable nesnesi otomatik olarak run fonksiyonu bittiğinde QThreadPool sınıfı tarafından delete edilmektedir. Bu istenmiyorsa QRunnable sınıfın setAutoDelete fonksiyonu false parametresiyle çağırılmalıdır. QRunnable nesnesi delete edilmeden önce destroy isimli sinyal emit edilmektedir.

QThreadPool sınıfının activeThreadCount isimli üye fonksiyonu belli bir anda havuzdan alınarak kullanılmakta olan (henüz sonlanmamış olan) thread'lerin sayısını bize verir. Tabii havuzdan aldığımız thread'ler sonlandıkça bu sayı da azalacaktır.

QThreadPool sınıfının maxThreadCount isimli üye fonksiyonu thread havuzundaki toplam maksimum thread sayısını belirtmektedir. Yani bu değerden daha fazla thread havuzdan alınarak aynı anda kullanılamaz. Bu değer default olarak 8'dir. İstenirse setMaxThreadCount fonksiyonuyla değiştirilebilir. Burada belirtilen değerden daha fazla thread havuzdan alınıp start edilmek istenirse herhangi bir sorun oluşmaz. Ancak bu thread'ler start edilmez. Bekletilir. Havuzdan alınmış diğer thread'lerin işi bittiğinde çalışma bunlara verilir. QThreadPool sınıfının cancel isimli fonksiyonu bekletilen henüz çalıştırılmamış thread'leri iptal etmek için kullanılabilir. clear fonksiyonu ise henüz çalıştırılmamış fakat beklemekte olan tüm thread'leri iptal eder. QThreadPool sınıfının waitForDone isimli fonksiyonu havuzdaki çalışmakta ve beklemekte olan tüm thread'ler sonlanan kadar bloke beklemeye yol açmaktadır.

Qt'de Palet Kullanımı

Qt'de paletler pencerelerin (widget'ların) çeşitli bazı özelliklerini değiştirmek için kullanılan öğelerdir. Örneğin paletler sayesinde pencere üzerinde yazıların renkleri ve pencerelerin arka plan renkleri değiştirilebilmektedir. Ancak Qt'ye 4'lü versiyonlarla "Style Sheet" konusu eklenince palet kullanımına bir gerek kalmamıştır. Çünkü "Style Sheet"ler zaten palet kullanımının daha ileri bir uyarlaması gibidir.

Qt'de palet kavramı QPalette sınıfı ile temsil edilmektedir. QWidget sınıfındaki setPalette fonksiyonu palet değiştirmek için kullanılabilir. Böylece her widget'ın bir paleti söz konusu olabilmektedir. palet ile değiştirilebilecek öğeler sınırlıdır. Yukarıda da belirtildiği gibi "Style Sheet"ler çok daha geniş olanaklara sahiptir.

QPalette sınıfının setColor isimli fonksiyonu widget'ta renk set etmek için kullanılmaktadır.

```
void QPalette::setColor(ColorGroup group, ColorRole role, const QColor &color);
```

ColorGroup üç seçenekten oluşan bir enum'dur.

```
QPalette::Disabled  
QPalette::Active  
QPalette::Inactive
```

Active klavye odağı widget üzerindeykenki durumu, Disabled widget'ın klavye odağına kapatıldığı durumu, InActive klavye odağının widget üzerinde olmadığı durumu belirtir. ColorRole ise ilgili rengin widget'a ilişkin hangi öğenin rengi olduğunu belirlemekte kullanılır. Aşağıdaki renk rolleri vardır:

```
QPalette::Window  
QPalette::Background  
QPalette::WindowText  
QPalette::Foreground  
QPalette::Base  
QPalette::AlternateBase  
QPalette::ToolTipBase  
QPalette::ToolTipText  
QPalette::Text  
QPalette::Button  
QPalette::ButtonText  
QPalette::BrightText
```

QColor ise ilgili rengi belirtmektedir.

Örneğin bir QLineEdit widget'ının yazı ve zemin rengi şöyle set edilebilir:

```
QPalette palette;
palette.setColor(QPalette::Active, QPalette::Text, Qt::blue);
palette.setColor(QPalette::Active, QPalette::Base, Qt::yellow);

ui->m_lineEdit->setPalette(palette);
```

Burada zemin rengi için QPalette::Base renk rolünün kullanıldığına dikkat ediniz. Halbuki diğer widget'ların çoğu için QPalette::Window kullanılmaktadır. Örneğin bir label için şekil ve zemin renkleri şöyle değiştirilebilir:

```
QPalette paletteLabel;
paletteLabel.setColor(QPalette::Active, QPalette::WindowText, Qt::cyan);
paletteLabel.setColor(QPalette::Active, QPalette::Window, Qt::blue);
ui->m_label->setAutoFillBackground(true);
ui->m_label->setPalette(paletteLabel);
```

Örneğin bir düğmenin üzerindeki yazının rengini değiştirmek isteyelim:

```
QPalette paletteButton;
paletteButton.setColor(QPalette::Active, QPalette::ButtonText, Qt::red);
ui->m_buttonOk->setPalette(paletteButton);
```

Ana pencerenin zemin rengi ise şöyle değiştirilebilir:

```
QPalette paletteMainWindow;
paletteMainWindow.setColor(QPalette::Active, QPalette::Window, Qt::magenta);
ui->m_buttonOk->setPalette(paletteMainWindow);
```

Palet ayarlama işlemleri Qt'nin "Designer"ı ile görsel biçimde de yapılabilir.

Qt'de Style Sheet Kullanımı

Palet yöntemi widget özelliklerinin set edilmesinde kaba bir kullanım sunmaktadır. Ayrıca palet kod bakımından programcıya yük de oluşturmaktadır. Bunun yerine Qt'ye 4'lü versiyonlarla birlikte "Style Sheet" kullanımı sokulmuştur. Qt'nin "Style Sheet" tasarımı büyük ölçüde Web'teki CSS (Cascading Style Sheets)'lerden alınmıştır. Genel sentaks HTML CSS'tekine oldukça benzemektedir. Style Sheet sayesinde widget'ların yalnızca renkleri değil daha pek çok özellikleri değiştirilebilmektedir.

"Style Sheet" belli bir sentaksa sahip bir yazıdan ibarettir. Bu yazı oluşturulur ve ilgili sınıfların setStyleSheet fonksiyonuyla set edilir. QWidget sınıfının ve QApplication sınıfının setStyleSheet fonksiyonları vardır. "Style Sheet" üstten alta kalıtım yoluyla aktarılan bir özellik sunmaktadır. Yani biz bir üst pencerenin bazı özelliklerini setStyleSheet fonksiyonuyla değiştirdiğimizde bundan onun alt widget'ları da etkilenecektir.

Style Sheet sentaksı şöyledir:

```
<Seçici (Selector)> { <Özellik (Attribute) > : <Değer (Value)> ; ... }
```

Küme parantezinin solundaki hedef belirten öğeye seçici (selector) denilmektedir. Her seçici için önceden tanımlanmış birtakım özellikler vardır. Özelliklere atanacak değerler de önceden belirlenmiştir. Aslında yeni özellikleri programcının kendisi de oluşturabilir. Ancak burada bu ayrıntılar ele alınmayacaktır. Bir seçici belirlendiği zaman o belirleme seçicinin alt pencerelerine doğru etki göstermektedir. Başka bir deyişle yapılan Style Sheet belirlemeleri nerede yapılmışsa onun alt elemanlarını etkilemektedir. Eğer style sheet QApplication sınıfında belirlenmişse buradaki belirlemeler tüm uygulamayı etkiler. Eğer style sheet belirlemeleri ana pencere üzerinde yapılmışsa buradaki belirlemeler de o ana pencerenin içeriisindeki tüm pencerelerde etkili olmaktadır.

En basit seçici (selector) sınıf isminde oluşur. Tüm widget'lara uygulanabilen iki özellik (attribute) color ve background-color özellikleridir. Örneğin:

```

MainWindow::MainWindow(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QString styleSheet = "QLineEdit {color: red; background-color: blue}";

    setStyleSheet(styleSheet);
}

```

Burada biz ana pencerenin style sheet'ini set etmiş olduk. Bu style sheet içerisinde QLineEdit türünden nesnelerin şekil ve zemin renkleri belirlenmiştir. Neredeyse tüm widget'lara uygulanabilecek iki özellik "color" ve "background-color" özellikleridir. Bunlara atanacak değerlerin birer renk belirtmesi gerekir. Renk ise birkaç biçimde belirtilebilmektedir:

- 1) Doğrudan bazı renkler isimlerle belirtilebilir. Örneğin red, green, blue gibi
- 2) rgb(red, green, blue) style sheet fonksiyonuyla belirtilebilir.
- 3) rgba(red, green, blue, alpha) style sheet fonksiyonuyla belirtilebilir.
- 4) #rrggbb (burada rr, gg ve bb iki hex digit'ten oluşmalıdır.)
- 5) rgba(red%, green%, blue%, alpha%)

Seçici olarak bir sınıf ismi kullanıldığında eğer onun türemiş sınıfları için ayrıca beelirleme yapılamamışsa etki türemiş sınıfları da kapsar. Örneğin:

```

QWidget {background-color: rgb(130, 255, 0)}

```

Bu style ana pencereye iliştilirirse onun kendisi dahil olmak üzere QWidget'tan türetilmiş tüm alt pencereleri de etkileyecektir.

Tüm widget'lara uygulanabilecek diğer bir özellik de "background-image" özelliğidir. Bu özellik "url" isimli style shhet fonksiyonuyla set edilir. İstenirse yine kaynaktaki bir resim verilebilir. Resim küçükse default durumda sol-üst köşeden başlanarak önce soldan sağa, sonra yukarıdan aşağıya o resimli zemin doldurulur. Örneğin:

```

QLabel
{
    background-image: url(":/AbbeyRoad.jpg")
}

```

Resmi widget boyutuna getirmek için (stretch yapmak için) QFrame sınıfının border-image özelliğinden faydalanılır. Örneğin:

```

QFrame
{
    border-image: url(":/AbbeyRoad.jpg") 0 0 0 0 stretch stretch;
    border-width: 0px;
}

```

Birden fazla seçici eğer ortak özellikleri varsa tek hamlede de set edilebilir. Örneğin:

```

QLabel, QTextEdit, QLineEdit {background-color: blue};

```

Burada bu üç widget tünün ve onlardan türetilenlerin zemin renkleri mavi yapılmıştır.

Seçiciler birkaç biçimde oluşturulabilmektedir:

Seçici	Örnek	Uyuşan Widget'lar
--------	-------	-------------------

Sınıf İsmi	QWidget	Burada belirtilen sınıf ve burada belirtilen sınıftan türetilen sınıflar
. Sınıf İsmi	.QWidget	Yalnızca burada belirtilen sınıf, fakat bunun türemiş sınıfları değil
Sınıf ismi # nesne ismi	QWidget#m_widget1	Yalnızca ilgili sınıf türünden olup “object name” property’si belirtilen biçimde olanlar
Sınıf İsmi[property=”xxx”]	QWidget[x=”300”]	İlgili property değeri belirtilen gibi olan ilgili türden widget
Sınıf İsmi > Sınıf İsmi	QFrame > QDial	> işaretinin sağındaki türün üst penceresi eğer > solunda belirtilen türden ise. Örnekte QFrame’den türetilmiş QDial nesneleri uyacaktır.
Sınıf İsmi Sınıf İsmi ...	QFrame QDial	Her iki sınıftan da türetilmiş olan sınıflar uyuşur

Bazı seçiciler widget içerisindeki birtakım alt öğelere ilişkin olabilmektedir. Bunların listeleri de “Qt Style Sheet Reference” içerisinde verilmiştir. Alt öğelere ilişkin seçiciler :: karakterleriyle belirtilirler. Örneğin:

```
QCheckBox::indicator, QRadioButton::indicator {
    width: 20px;
    height: 20px;
}
```

Diğer bazı alt öğeler şunlardır:

- spin box ve scrool bar’ların oklarının büyüklüğü ::up-arrow ve ::down-arrow ile ayarlanabilmektedir.
- Combox’ın oku ::drop-down ile ayarlanabilmektedir.

QCheckBox ve QRadioButton sınıflarında kutucuk ya da yuvarlakçıkla yazı arasında mesafe spacing özelliği ile ayarlanabilmektedir. Örneğin:

```
QRadioButton, QCheckBox {spacing: 50;}
```

Bazı widget’ların bazı durumları için belirlemeler yapılabilir. Bu belirlemeler seçiciden sonra ‘:’ karakteriyle belirtilmektedir. Örneğin:

```
QCheckBox:hover {color: red }
```

Burada fare ile seçenek kutusu üzerine gelinip beklendiğinde seçenek kutusunun yazısı kırmızı olacaktır. Durumlar kombine edilebilir. Örneğin:

```
QCheckBox:checked:hover {color: red }
```

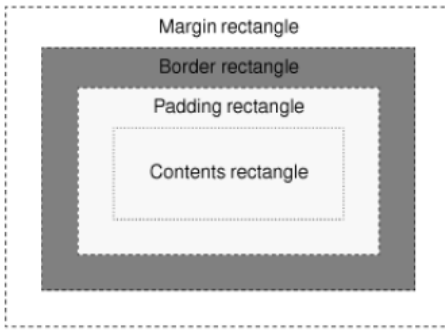
Burada seçen kutusu çarpılmışsa ve fare onun üzerinde bekletiliyorsa onun yazısı kırmızı yapılacaktır.

Tipik durum belirten öğeler şunlardır:

Figure 19.5. Some of the widget states accessible to style sheets

State	Description
:disabled	The widget is disabled
:enabled	The widget is enabled
:focus	The widget has input focus
:hover	The mouse is hovering over the widget
:pressed	The widget is being pressed using the mouse
:checked	The button is checked
:unchecked	The button is unchecked
:indeterminate	The button is partially checked
:open	The widget is in an open or expanded state
:closed	The widget is in a closed or collapsed state
:on	The widget is "on"
:off	The widget is "off"

Bazı widget'lar "box model" denilen bir modele uygun tasarlanmıştır. Bu modelde widget dört kademedan oluşur: İçerik, Padding, Border ve Margin. Padding widget içeriği ile widget border'ı arasındaki ayrılmış alandır. Margin ise widget'ın dışı için ayrılmış alandır. Yani başka bir widget margin'i delerek ona daha fazla yaklaştırılmaz.



Normal olarak bir widgetta tüm bu dört katman üst üste binmiştir. Yani sanki yalnızca "içerik kısmı" varmış gibi bir etki söz konusudur. Ancak biz "box model"i destekleyen bir widget için bu dört katmanı ayarlayabiliriz.

Örneğin border-radius özelliği dikdörtgenin köşelerini yuvarlaklaştırmak için kullanılabilir.

```
QPushButton {border: 4px groove gray; border-radius: 10px; }
```

Örneğin bir QListWidget nesnesinde yazıların renklerini ve seçimin rengini aşağıdaki gibi değiştirebiliriz:

```
QListWidget::item
{
    background: rgb(255,255,255);
}

QListWidget::item:selected
{
    background: purple;
}
```

Qt'de Veritabanı İşlemleri

Bu bölümde Qt'de veritabanı işlemleri ele alınacaktır. Ancak önce bazı genel bilgiler verilecektir.

Veritabanı Nedir?

Bilgilerin saklanması ve geri alınması için organize edilmiş dosyalara veritabanı denilmektedir. Veritabanları tek bir dosya olarak organize edilebildiği bir grup dosya biçiminde de organize edilebilir. Genellikle bu organizasyon istenen bilgilerin hızlı bir biçimde elde edilmesi amacıyla gerçekleştirilmektedir. Günümüzde uygulamaların pek çoğu küçük ya da büyük birtakım veritabanlarını kullanmaktadır.

Veritabanlarının organizasyonu için birtakım modeller (paradigmalar) kullanılmaktadır. Günümüzde bunlardan en çok tercih edileni “ilişkisel (relational)” veritabanı modelidir. Ancak farklı uygulamalarda farklı modellerin çeşitli avantajları vardır. Qt’de ana olarak ilişkisel model kullanılmaktadır.

Veritabanı Yönetim Sistemleri (Database Management Systems - DBMS)

Veritabanı işlemleri ticari uygulamalarda uygulamanın performansı üzerinde en etkili olan öğelerdendir. Bu nedenle geliştiriciler veritabanı işlemlerini mümkün olduğunca hızlı yapan araçlar kullanmak isterler. Eskiden veritabanı işlemleri kütüphaneler ile yapılıyordu. Yani bu konuda uzmanlaşmış kişilerin ya da şirketlerin yazmış olduğu kütüphanelerdeki fonksiyonlarla veritabanlarına kayır eklenip, sorgulamalar yapılıyordu. Ancak bu kütüphanelerin oldukça aşağı seviyeli bir yapısı vardı. Bunlarla çalışma genel olarak zordu. İşte ilk kez 70’li yılların sonlarına doğru “Veritabanı Yönetim Sistemi (VTYS)” ismi altında veritabanı işlemlerini yapan özel uygulamalar geliştirildi. Bu yazılımlar veritabanı işlemlerinden sorumlu oldular.

Bir yazılama VTYS denebilmesi için onun bazı özelliklere sahip olması gerekmektedir. Bunlardan bazıları şunlardır:

- 1) Aşağı Seviyeli Dosya Formatlarıyla Kullanıcının İlişisinin Kesilmiş Olması: VTYS’lerde kullanıcıların bilgilerin hangi dosyalarda ve nasıl organize edildiğini bilmelerine gerek kalmamaktadır. Yani adeta veritabanı kullanıcıya bir kara kutu biçiminde gösterilmektedir. Kullanıcı yalnızca ne yapacağını VTYS’ye iletir. İşlemleri VTYS yapar.
- 2) VTYS’ler yüksek seviyeli deklaratif dillerle kullanıcı isteklerini elde etmektedir. Bu dillerden en yaygın olanı “SQL (Structured Query Language)”dır. SQL asıl sorgulama işlemlerini yapan programların dili değildir. SQL kullanıcının VTYS’ye isteğini anlatmak için kullanılan bir dildir. VTYS bu isteği alır, motor kısmındaki C/C++ ile yazılmış kodlar yoluyla sonuçları elde eder ve kullanıcıya verir.
- 3) VTYS’ler client-server çalışma modeline sahiptir. Yani birden fazla kullanıcı VTYS’ye istekte bulunabilir. VTYS bu istekleri karşılar. Yani biz bir VTYS’yi bilgisayarımıza kurduğumuzda aynı zamanda bir server da kurmuş oluruz.
- 4) VTYS’lerde belli düzeylerde güvenlik mekanizması (safety and security) oluşturulmuştur. Yani bilgiler bu sistemlerde kolayca bozulmazlar ve çalınmazlar.
- 5) VTYS’lerin çoğu yardımcı birtakım araçlar içermektedir. Örneğin backup-restore programları, yönetici programlar, kütüphaneler vs.

Peki günümüzde en çok tercih edilen VTYS’ler nelerdir? Oracle firmasının Oracle isimli ürünü büyük veritabanları için kurumların en çok tercih ettiği VTYS’lerden biridir. Microsoft’un SQLServer isimli ürünü doğrudan Oracle ile rakip durumdadır. Pek çok kurum Sql Server’ı tercih etmektedir. Bunun dışında ücretli başka VTYS’ler de vardır. Ancak ücretsiz ve açık kaynak kodlu da pek çok VTYS geliştirilmiştir. MySql açık kaynak kodlu bir projedir. Ancak bazı haklarını daha sonra Oracle satın almıştır. Açık kaynak kodlu olarak devam etmektedir. Pek çok kaynak kodlu projeler ve Web tabanlı uygulamalar VTYS olarak MySQL’i kullanmaktadır. PostGre SQL diğer bir bedava ve açık kaynak kodlu VTYS’dir. Son dönemlerde gittikçe popülaritesi artmaktadır.

Bir grup VTYS aslında VTYS’lerin pek çok özelliğini barındırmasa da SQL kullanımına izin vermektedir. Bunların kurulum sorunları yoktur. Bunlar adeta bir veritabanı kütüphanesi gibi tek bir kütüphane dosyasından (örneğin DLL’den) oluşmuşlardır. Özellikle gömülü sistemlerde tercih edilmelerinden dolayı bunlara “Gömülü VTYS (Embedded DBMS)” de denilmektedir. Bunların en yaygın olanı şu günlerde SQLite’tir. SQLite hem Windows, hem Linux hem MAC OS X hem de mobil işletim sistemlerinde aynı biçimde kullanılmaktadır. Diğer çok kullanılan bir VTYS de Microsoft’un

“Jet Motorudur”. Bu Access olarak da bilinir. Access’in veritabanı motoru tüm Windows sistemlerinde Office yüklenmiş olmasa bile Windows’un bir parçası olarak bulunmaktadır.

SQLite Kurulumu ve Qt İçin Hazır Hale Getirilmesi

SQLite başından beri Qt tarafından içsel olarak (built-in) desteklenmektedir. Yani Qt sürümü ne olursa olsun SQLite sürücüsü zaten bulunuyor olacaktır. Qt kendği içerisinde SQLite’in taban kütüphanesine de zaten sahiptir. Dolayısıyla SQLite kullanmak için birşey yapmaya gerek olmayacaktır. Ancak SQLite üzerinde birtakım işlemleri görsel olarak yapabilmek için çeşitli kişiler ve kurumlarca yazılmış olan yönetici programlar vardır. Bunlardan biri “FireFox tarayıcısında bir plugin olarak çalışan SQLite Manager” programıdır. Diğerisi ise SQLite Studio olabilir.

MySql’in Kurulumu ve Qt İçin Hazır Hale Getirilmesi

MySql için çeşitli kurum programları vardır. Genellikle programcılar asıl VTYS server’ının yanı sıra bir de GUI arayüzlü yönetim programını da kurmak isterler. Bunun için “MySql Community Download” sayfasına gelinir. Burada “MySql Community Server” ve GUI aracı olarak da “MySql Workbench” kurulur. Server’ın kurulması sırasında bizden “root” kullanıcısı için parola da istenecektir.

Qt belli bir süredir artık MySql veritabanı sürücüsü içsel olarak (built-in) yüklü biçimde kurulmaktadır. Eskiden MySql’i Qt’de kurmak zaman alıcı bir işlemdi.

Microsoft Sql Server’ın Kurulumu ve Qt’de Hazır Hale Getirilmesi

Bilindiği gibi Sql Server paralı bir üründür. Ancak Microsoft MySql ile rekabet için bu ürünün biraz düşük güçlü bir versiyonunu parasız olarak dağıtmaktadır. Buna “Sql Server Express Edition” denilmektedir. Bu paket Microsoft’un sitesinden indirilerek kurulabilir. GUI aracı olarak Microsoft “Sql Server Management Studio” denilen yazılımı kullanmaktadır. Express Edition zaten kendi içerisinde “Management Studio”yu da içermektedir.

Qt’de “Sql Server” için “ODBC Veritabanı Sürücüsü” kullanılmaktadır. Bu sürücü bazı Qt dağıtımlarında (5.7 ve sonrası) hazır olarak gelmektedir. Eğer Qt versiyonu eskiyse bizim bu sürücüyü derleyip Qt’ye dahil etmemiz gerekir.

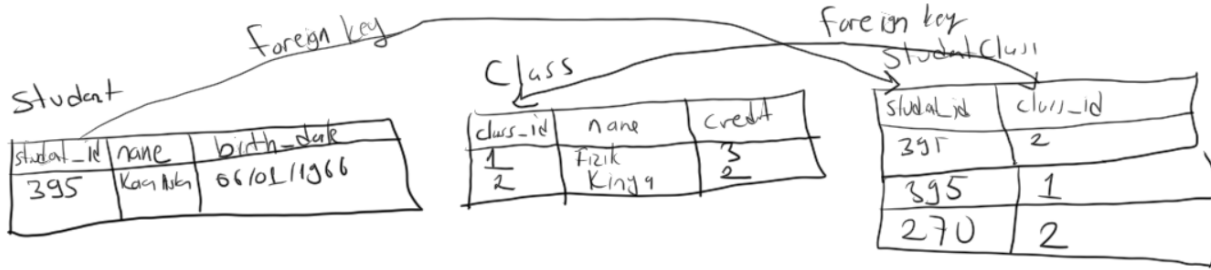
İlişkisel Veritabanları (Relational Databases)

İlişkisel veritabanları kabaca tablolardan (tables), tablolar da sütunlardan (fields) ve satırlardan (records) oluşmaktadır.

Adı Soyadı	NO
Ali Savaş	123
Kaan Aslan	513
—	—

Tablolarda yinelenmemesi garantisi verilen sütunlara “Birincil Anahtar (Primary Key)” denilmektedir. Genellikle her tablosunun bir tane birincil anahtara sahip olması tavsiye edilir.

İlişkisel veritabanlarında bilgiler birden fazla tabloda bulunuyor olabilir. Böylece bilgi elde edilirken birden fazla tablodan alınabilir. Örneğin:



İdeal olarak veritabanı tabloları tekrar bilgi içermemelidir. Örneğin Bir öğrencinin adı ve soyadı birden fazla tabloda gereksiz bir biçimde bulundurulmamalıdır. Tabii tablolar arasında geçiş yapmak için ortak bir anahtara gereksinim duyulur. Bunun oluşturulan sütunlara (alanlara) “yabancı anahtar (foreign key)” denilmektedir. Yukarıdaki veritabanında öğrencinin numarası onun hangi dersleri aldığı bilgisini elde etmek için kullanılmaktadır. Benzer biçimde öğrencinin derslerin id numaraları da o derslerin diğer bilgilerine erişmekte kullanılmaktadır. İşte büyük veritabanlarında böyle binlerce tablo bulunabilmektedir. Uygun bilginin elde edilmesi bir sürü tablo gezinerek yapılabilmektedir. Tabloların tekrarı engelleyecek biçimde düzenlenmesine veritabanı terminolojisinde “normalizasyon” denilmektedir. Veritabanı tablolarını tasarlamakta kullanılan yazılımsal araçlar da vardır. Fakat kursumuzda bunlar ele alınmayacaktır.

Temel SQL Bilgisi

Bu bölümde Qt için temel oluşturabilecek SQL bilgisi edineceğiz. SQL büyük harf küçük harf duyarlılığı olan bir dil değildir. Dolayısıyla SQL komutlarının büyük harf ya da küçük harfli yazılmaları sorun oluşturmaz. Pek çok programcı bir yazım stili olarak SQL komutlarını büyük harflerle belirtmektedir. Ayrıca SQL komutları ‘;’ ile sonlandırılmaktadır. Ancak pek çok VTYS komutlar ‘;’ ile sonlandırılmasa bile bir sorun oluşturmamaktadır.

Veritabanının Yaratılması

Şüphesiz ilk iş bir veritabanının sıfırdan yaratılmasıdır. Bu işleme bir kez gereksinim duyulacağı için VTYS’lerinin yönetim ekranından yapılabilir. SQL’de veritabanı yaratmak için CREATE DATABASE komutu kullanılmaktadır. Komutun genel biçimi şöyledir:

```
CREATE DATABASE <isim> ;
```

Veritabanı Tablolarının Yaratılması

Veritabanı yaratıldıktan sonra sıra tabloların yaratılmasına gelir. Yine tabloları biz VTYS’lerin sunduğu GUI araçlarıyla ya da doğrudan SQL cümleleriyle yaratabiliriz. Tabloların yaratımı sırasında tabloların isimleri, onların sütunlarının isimleri ve türleri tek tek belirtilir. Sütun türleri SQL standartlarında belirtilmiştir. Ancak farklı VTYS’ler kendi özgü türler de kullanmaktadır. Tipik sütun türleri şunlardır:

CHARACTER(n)	En fazla n karakterlik yazıların tutulacağı alan için kullanılır. Genellikle VTYS yazı n karakterden küçük olsa bile n karakterlik yeri ayırmaktadır.
VARCHAR(n)	En fazla n karakterli yazıların tutulabileceği alan için kullanılır Ancak n’den az karakterli yazılar için n karakterlik yer ayrılmaz.
SMALLINT	Bu tür genellikle 2 byte uzunluğunda olan tamsayıları belirtmektedir. Ancak VTYS’den VTYS’ye değişebilir.
INTEGER	Bu tür genellikle 4 byte uzunluğunda olan tamsayıları belirtmektedir. Ancak VTYS’den VTYS’ye değişebilir.
BIGINT	Bu tür genellikle 8 byte uzunluğunda olan tamsayıları belirtmektedir. Ancak VTYS’den VTYS’ye değişebilir.
FLOAT	Genellikle 8 byte’lık gerçek sayı türünü tutmak için kullanılmaktadır.
REAL	Genellikle 4 byte’lık gerçek sayı türünü tutmak için kullanılmaktadır.
DOUBLE	Genellikle 8 byte’lık gerçek sayı türünü tutmak için kullanılmaktadır.
DATE	Tarih bilgisi tutmak için kullanılır
TIME	Zaman bilgisi tutmak için kullanılır

BLOB	En fazla 2 byte uzunluğunda (65526) binary alan.
TINYBLOB	En fazla 1 byte uzunluğunda (256) binary alan
MEDIUMBLOB	En fazla 3 byte uzunluğunda (16777216) binary alan.
LONGBLOB	En fazla 4 byte uzunluğunda (4294967296) binary alan.
TINYTEXT	En fazla 1 karakter uzunluğunda (256) text alan
MEDIUMTEXT	En fazla 3 byte uzunluğunda (16777216) text alan.
TEXT	En fazla 2 byte uzunluğunda (65526) text alan.

Pek çok VTYS yukarıdaki türlerden başka pek çok tür de bulundurmaktadır. Örneğin MySQL’de işaretli tamsayı türleri de vardır.

Tablo yaratmak için CREATE TABLE Sql komutu kullanılır. Komutun yalın genel biçimi şöyledir:

```
CREATE TABLE <table_name>
(
    <column_name1> <data_type>(size),
    <column_name2> <data_type>(size),
    <column_name3> <data_type>(size),
    ....
);
```

Komutun ayrıntıları için Sql kaynaklarına başvurulabilir. Örneğin:

```
CREATE TABLE student_info(student_id INTEGER PRIMARY KEY AUTO, student_name VARCHAR(45),
student_bdate DATE);
```

Tabloya Kayıt Ekleme İşlemi

Tabloya kayıt eklemek için INSERT INTO Sql komutu kullanılır. Komutun genel biçimi şöyledir:

```
INSERT INTO <table_name> (column1,column2,column3,...)VALUES (value1,value2,value3,...);
```

Kayıt eklerken yazısal sütunlar ile tarih ve zaman sütunları tek tırnak içerisinde alınmalıdır. Bunun dışında sayısal sütun bilgileri doğrudan yazılır. Örneğin:

```
INSERT INTO student_info(student_name, student_bdate) VALUES ('John Lennon', '1940/10/9');
```

Insert işlemi sırasında biz bazı sütunları belirtmeyebiliriz. Ancak belirtmediğimiz sütunlar için default değer tanımlamasının yapılmış olması gerekir. Bazı sütunlar “Auto Increment” olabilmektedir. Bu durumda VTYS kayıt ekleme sırasında önceki değerlerin bir fazlasını bu sütuna değer olarak verir.

WHERE Cümleceği

WHERE cümleceği Sql’de bir komut değildir. Bazı komutların içerisinde kullanılan bir kalıptır. Örneğin DELETE FROM komutunun, SELECT komutunun WHERE cümleceği kısımları vardır. WHERE cümleceği koşul belirtmektedir. Koşul belirtilirken sütun isimleri ve temel karşılaştırma operatörleri kullanılabilir. Örneğin:

```
WHERE student_id > 1250
```

Koşullarda mantıksal AND, OR ve NOT operatörleri kullanılabilir. Örneğin:

```
WHERE student_name = 'Kaan Aslan' AND student_id > 2450
```

LIKE operatörü yazısal bir sütunun belli bir kalıba uygunluk koşulu için kullanılır. % joker karakteri “bundan sonra herhangi karakterler gelebilir” anlamındadır.

Örneğin:

```
WHERE student_name LIKE 'S%'
```

Burada ismi s ile başlayan öğrenciler için koşul verilmiştir.

Kayıt Silme İşlemi

Belli koşulları sağlayan kayıtların silinmesi DELETE FROM komutuyla yapılmaktadır. Komutun genel biçimi şöyledir:

```
DELETE FROM <tablo ismi> [WHERE cümlecığı]
```

Örneğin:

```
DELETE FROM student_info WHERE student_name = 'Ali Serçe'
```

Bu kmutla ismi Ali Serçe olan tüm kayıtlar silinmektedir. Örneğin:

```
DELETE FROM student_info WHERE student_id > 100
```

Burada id'si 100'den büyük tüm kayıtlar silinmektedir.

Kayıt Güncelleme İşlemi

Var olan kayıtların güncellenmesi UPDATE komutuyla yapılmaktadır. Komutun genel biçimi şöyledir:

```
UPDATE <tablo ismi> SET sütun1=değer1,sütun2=değer2,...WHERE <koşul>
```

Örneğin:

```
UPDATE student_info SET student_name = 'Ali Serçe' WHERE student_name = 'Kaan Aslan'
```

Burada ismi 'Kaan Aslan' olan kayıtların (bu kayıtların hepsi) ismi 'Ali Serçe' olarak değiştirilir. Komutun WHERE kısmının doğru girildiğinden emin olunuz. Aksi takdirde değişiklikler tüm kayıtlar üzerinde yapılır. Birincil anahtarın tabloda tek olması garanti edildiği için birincil anahtara dayalı kısıtlar güvenle kullanılabilir. Örneğin:

```
UPDATE student_info SET student_name = 'Ali Serçe' WHERE student_id = 152
```

Burada student_id birincil anahtar ise kesinlikle bu anahtara sahip olan tek bir kayıt vardır.

Koşulu Sağlayan Kayıtların Elde Edilmesi

Koşulu sağlayan kayıtların elde edilmesi için SELECT komutu kullanılmaktadır. SELECT komutunun genel biçimi oldukça ayrıntılıdır. Pek çok cümlecik (örneğin WHERE cümlecığı) komut içerisinde bulundurulabilmektedir.

SELECT komutun tipik kullanımı şöyledir:

```
SELECT <sütun listesi> FROM <tablo ismi> [WHERE cümlecığı]
```

Sütun listesi yerine '*' karakteri getirilirse tüm sütunlar anlaşılır. WHERE cümlecığı kullanılmazsa tüm kayıtlar anlaşılır. Örneğin MySQL'in örnek "World" veritabanı için şöyle bir SELECT komutu yazmış olalım:

```
SELECT Code FROM country WHERE Name LIKE 'T%'
```

Bu komutla ilk harfi T ile başlayan tüm ülkelerin ülke kodları elde edilecektir. Örneğin:

SELECT komutunda VTYS'nin hazır bazı fonksiyonları kullanılabilir. Her VTYS'nin birtakım hazır fonksiyonları vardır. Ancak bu konuda bir standart bulunmamaktadır. Örneğin MySQL'de DAYOFMONTH isimli fonksiyon bir tarihin gün değerini verir. Biz de bu sayede aşağıdaki gibi bir SELECT komutu oluşturabiliriz:

```
SELECT * FROM student_info WHERE DAYOFMONTH(student_bdate) < 10
```

Örneğin:

```
SELECT * FROM student_info WHERE MOD(age, 10) = 0
```

Burada yaşı 10'un katlarında olan öğrencilerin bilgileri listelenmek istenmiştir. Örneğin:

SELECT cümlesine ORDER BY cümleceği eklenebilir. ORDER BY cümleciğinin genel biçimi şöyledir:

```
ORDER BY sütun ismi [ASC|DESC], sütun ismi [ASC|DESC], ...;
```

Burada ASC (ascending) küçükten büyüğe, DESC (descending) büyükten küçüğe anlamına gelmektedir. Default durum ASC biçimindedir. Birden fazla sütun kullanılırsa birinci belirtilen sütundaki eşitlikler diğer sütunlara göre sıralanır.

VTYS'lerin fonksiyon listelerine onların dokümanlarından erişilebilir. Ancak bu fonksiyonların standart olmadığını yani her VTYS fonksiyonlarının birbirlerinden farklılık gösterebildiğini anımsatalım.

SELECT ile birden fazla tablodan bilgi alınabilir. Bu işleme genel olarak "join (birleştirme)" işlemi denilmektedir. Join işleminin INNER, LEFT, RIGHT ve FULL biçiminde türevleri vardır. Ancak bu join türevleri tüm VTYS'ler tarafından tam olarak desteklenemeyebilmektedir. Join işlemi denildiğinde default olarak INNER JOIN anlaşılmaktadır.

Join işlemi kartezyen çarpım işlemi biçiminde ele alınarak açıklanabilir. Bilindiği gibi iki kümenin kartezyen çarpımı sıralı ikililerden oluşmaktadır. Bu sıralı ikililerin ilk terimleri soldaki kümeden, ikinci terimleri sağdaki kümeden oluşturulur:

$$A \times B = \{ (a, b) \mid a \in A \text{ ve } b \in B \}$$

İşte biz iki tabloyu bu biçimde kartezyen çarpım işlemine sokarsak iki tablonun eleman sayılarının çarpımı kadar kayıt elde etmiş oluruz. Sonra bu kayıtlardan WHERE cümlesi ile belirtilen koşulu sağlayanlar seçilirse bu işleme INNER JOIN denilmektedir. INNER JOIN sentaksı şöyledir:

```
SELECT <sütun listesi> FROM table1 INNER JOIN table2 ON <koşul>
```

Sütun ve koşul kısımlarında her iki tablonun sütunları bulundurulabileceğinden dolayı bir çakışma söz konusu olabilir. Çatışma durumunda sütun isimleri tablo isimleriyle araya '.' karakteri konularak niteliklendirilebilir. Aslında SQL kullanıcıları çakışma olmasa da sütunları hep tablo isimleriyle niteliklendirmektedir. Örneğin MySQL'in örnek "world" veritabanı için aşağıdaki sorgulamayı yapıyor olalım:

```
SELECT city.Name, country.Name FROM city INNER JOIN country ON city.CountryCode = country.Code
```

Burada biz sonuç olarak city tablosundaki isimleri ile country tablosundaki isimleri beraber görüntülemek istemekteyiz. Ancak bu iki tablonun kartezyen çarpımındaki tüm satırlar için bu işlemler yapılmayacak. Yalnızca ON kısmında belirtilen koşulların sağlandığı satırlar elde edilecek. Bu işlemin sonucunda da biz tüm şehirlerin hangi ülkeye ilişkin olduğuna ilişkin bir liste elde ederiz.

Name	Name
Oranjestad	Aruba
Kabul	Afghanistan
Qandahar	Afghanistan
Herat	Afghanistan
Mazar-e-Sharif	Afghanistan
Luanda	Angola
Huambo	Angola
Lobito	Angola
Benguela	Angola

Örneğin:

```
SELECT country.Name, countrylanguage.Language, countrylanguage.Percentage FROM country INNER JOIN countrylanguage ON country.Code = countrylanguage.CountryCode
```

Name	Language	Percentage
Tonga	English	0.0
Tonga	Tongan	98.3
Trinidad and Tobago	Creole English	2.9
Trinidad and Tobago	English	93.5
Trinidad and Tobago	Hindi	3.4
Tunisia	Arabic	69.9
Tunisia	Arabic-French	26.3
Tunisia	Arabic-French-English	3.2
Turkey	Arabic	1.4
Turkey	Kurdish	10.6
Turkey	Turkish	87.6

INNER JOIN işlemi için alternatif bir sentaks daha vardır. Bu sentaks doğrudan birden fazla tablonun isminin geçtiği SELECT cümlesi sentaksıdır. Örneğin:

```
SELECT city.Name, country.Name FROM city INNER JOIN country ON city.CountryCode = country.Code
```

INNER JOIN işleminin eşdeğeri şöyle de yazılabilir:

```
SELECT city.Name, country.Name FROM city, country WHERE city.CountryCode = country.Code
```

Örneğin:

```
SELECT country.Name, countrylanguage.Language, countrylanguage.Percentage FROM country INNER JOIN countrylanguage ON country.Code = countrylanguage.CountryCode
```

INNER JOIN işleminin de eşdeğeri şöyle yazılabilir:

```
SELECT country.Name, countrylanguage.Language, countrylanguage.Percentage FROM country, countrylanguage WHERE country.Code = countrylanguage.CountryCode
```

LEFT JOIN işleminde sol taraftaki tablonun tüm satırları ve ON koşulunu sağlayan satırlar alınır. Sol taraftaki tablonun ON koşulunu sağlamayan satırlarının sağ taraf sütunları boş (NULL) biçimdedir. Örneğin:

```
SELECT city.Name, country.Name FROM city LEFT JOIN country ON city.CountryCode = country.Code AND country.Population > 50000000
```

Name	Name
Naogaon	Bangladesh
Sirajganj	Bangladesh
Narsinghdi	Bangladesh
Saidpur	Bangladesh
Gazipur	Bangladesh
Bridgetown	NULL
Antwerpen	NULL
Gent	NULL
Charleroi	NULL

RIGHT JOIN ise LEFT JOIN işleminin tersidir. Yani sağ taraftaki tablonun tüm satırları ve ON koşulunu sağlayan satırlar alınır. Sağ taraftaki tablonun ON koşulunu sağlamayan satırlarının sol taraf sütunları boş (NULL) biçimdedir. Örneğin:

```
SELECT city.Name, country.Name FROM city RIGHT JOIN country ON city.CountryCode = country.Code
AND country.Population > 50000000
```

Name	Name
NULL	Belgium
NULL	Benin
NULL	Burkina Faso
Dhaka	Bangladesh
Chittagong	Bangladesh
Khulna	Bangladesh
Rajshahi	Bangladesh

FULL JOIN pek çok VTYS tarafından desteklenmemektedir. Bu işlemde sol taraftaki ve sağ taraftaki tabloların bütün satırları ayrıca bir de koşulu sağlayan satırlar elde edilir. Ancak koşulu sağlamayan satırların diğer tablo karşılıkları boş (NULL) olarak elde edilir.

Aslında SQL burada anlatılanlardan daha ayrıntılı bir dildir. Ancak kursumuzda bu kadar bilgi yeterli görülmüştür. Fakat ne olursa olsun ne kadar çok SQL bilinirse o kadar etkin işlemler yapılabilir.

Qt’de VTYS İle Bağlantı Kurulması

Qt’de veritabanı işlemleri için ilk yapılacak işlem bir QSqlDatabase nesnesi elde etmektir. Bu nesne sınıfın başlangıç fonksiyonuyla değil VTYS türü belirtilerek addDatabase isimli static fonksiyonlarıyla elde edilir. Örneğin:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
```

addDatabase fonksiyonuna parametre olarak VTYS sürücüsünün ismi geçirilir. Bu isimler temel olarak şunlardan biri olabilir:

İsim	VTYS
QSQLITE	Sqlite
QODBC	Genel, Örneğin Microsoft SQL Server
QMYSQL	MySql
QDB2	IBM DB2
QPSQL	PostGre

Bir program birden fazla VTYS bağlantısı kullanabilir. Bu durumda ona isim vermek gerekir. İsimsiz bağlantıya (yukarıdaki gibi) “default bağlantı” denilmektedir. Bağlantı isim addDatabase fonksiyonunun ikinci parametresi olarak geçirilmelidir. Örneğin:

```
QSqlDatabase db1 = QSqlDatabase::addDatabase("QSQLITE", "Connection1");
QSqlDatabase db2 = QSqlDatabase::addDatabase("QSQLITE", "Connection2");
```

Bağlantı oluşturulduktan sonra işlem öncesinde open ile açılması ve işlem sonunda da close ile kapatılması gerekir. Ancak veritabanı bağlantısının oluşturulabilmesi için VTYS’nin bazı parametrelerinin set edilmesi gerekmektedir. Bunlar “VTYS’nin host adresi”, “kullanıcı ismi”, “parola” ve “veritabanı ismi”dir.

VTYS'nin host adresi QSqlDatabase sınıfının setHostName fonksiyonu ile, kullanıcı ismi setUsername fonksiyonu ile, parola setPassword fonksiyonu ile ve veritabanı ismi de setDatabaseName fonksiyonuyla set edilir. Örneğin:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");

db.setHostName("localhost");
db.setUserName("root");
db.setPassword("csd1993");
db.setDatabaseName("world");

db.open();
//...
db.close();
```

SQLite için yalnızca setDatabaseName ile veritabanı dosyasının yol ifadesi girilmelidir. Örneğin:

```
QSqlDatabase db = QSqlDatabase::addDatabase("SQLITE");
db.setDatabaseName("D:/Dropbox/Kurslar/Qt-Mayis-2016/Src/063-
SQLiteStudentDatabase/Student.db");

db.open();
//...
db.close();
```

Veritabanı bağlantısı kurulduktan sonra artık sıra SQL cümlesinin VTYS'ye iletilip onun çalıştırılmasına gelmiştir. Bu işlemler QSqlQuery sınıfıyla yapılmaktadır. QSqlQuery nesnesi default başlangıç fonksiyonuyla oluşturulabileceği gibi bağlantı ismi verilerek de oluşturulabilir. Default başlangıç fonksiyonuyla oluşturulan nesne her zaman default isimle (yani isim verilmeden)O yaratılmış bağlantı ile çalışmaktadır.

Bir işlemde oluşan hatanın nedeni QSqlQuery sınıfının lastError fonksiyonuyla elde edilebilir. Oluşturulan veritabanı bağlantısı QSqlDatabase sınıfının static removeDatabase fonksiyonuyla yok edilebilir.

SQL Cümlelerinin VTYS'ye Gönderilerek Çalıştırılması

Sql komutlarının VTYS'de çalıştırılması için QSqlQuery sınıfı kullanılır. QSqlQuery sınıfı türünden nesne sınıfın default başlangıç fonksiyonuyla yaratılabilir. Bu durumda default bağlantı kullanılmış olur ya da bağlantı ismi verilerek de nesne yaratılabilir. Ya da istenirse doğrudan isim yerine QSqlDatabase nesnesi de fonksiyona parametre olarak verilebilir. Örneğin:

```
QSqlDatabase db1 = QSqlDatabase::addDatabase("SQLITE");
QSqlDatabase db2 = QSqlDatabase::addDatabase("SQLITE", "otherConnection");
...
QSqlQuery query1; // db1 ile belirtilen default bağlantıyı kullanır
QSqlQuery query2("otherConnection"); // db2 ile belirtilen bağlantıyı kullanır
QSqlQuery query3(db2); // db2 ile belirtilen bağlantıyı kullanır
```

Komutun işletilmesi için QSqlQuery sınıfının exec fonksiyonları kullanılır. exec fonksiyonları bizden işletilecek SQL cümlesini yazı olarak almaktadır. Fonksiyonlar işlemin başarısına göre true ya da false ile geri dönerler.

Örneğin SQLite veritabanına bir aşağıdaki gibi bir kayıt ekleyebiliriz:

```
m_db = QSqlDatabase::addDatabase("SQLITE");
m_db.setDatabaseName("c:/Database/Student.db");
m_db.open();
//...
QSqlQuery query;
query.exec("INSERT INTO student_info(student_name, student_no, student_bdate) VALUES('Cevat Kelle', 24567, '1997/12/23')");
```

SQL SELECT Komutunun Uygulanması

Sql SELECT komutunun diğer komutlardan bir farkı vardır. Bu komut VTYS'den bize kayıt döndürmektedir. Bu nedenle bu komutun uygulanması ve döndürülen kayıtların elde edilmesi için başka bilgilere de gereksinim duyulmaktadır. Eğer QSqlQuery sınıfının exec fonksiyonlarıyla biz bir SELECT komutu uygulamış isek bu durumda koşulu sağlayan kayıtları tek tek bir iterator mantığı ile elde edebiliriz. QSqlQuery sınıfının next isimli fonksiyonu imleci (iteratörü) bir ileri götürür. Eğer listenin sonuna gelinmişse next fonksiyonu false ile gelinmemişse true ile geri döner. Başlangıçta da imleç ilk kaydın bir gerisindedir. (Yani imleci ilk kayda konumlandırmak için bir kez next uygulamak gerekir). İmleç uygun kayda konumlandırıldıktan sonra sınıfın value isimli fonksiyonlarıyla kaydın bilgileri (sütunları) elde edilebilir. Overload edilmiş iki value fonksiyonu vardır. Birincisi bizden select edilmiş sütunun indeks numarasını diğeri de ismini alır. value fonksiyonları bize değeri QVariant isimli bir sınıf biçiminde verir. QVariant farklı türleri içerisinde ebarındıran kompozit bir türdür. QVariant sınıfının toXXX isimli üye fonksiyonları bize nesnenin içerisindeki değeri XXX türü olarak verir. Örneğin biz student_info tablosundaki öğrenci isimlerini ve numaralarını şöyle elde edebiliriz:

```
QSqlQuery query;
query.exec("SELECT student_name, student_no FROM student_info");

while (query.next()) {
    qDebug() << query.value(0).toString();
    qDebug() << query.value(1).toInt();
}
```

QSqlQuery sınıfının first isimli fonksiyonu imleci ilk kayda, last fonksiyonu son kayda konumlandırır. Yukarıda da belirtildiği gibi next sonraki kayda ve previous ise önceki kayda konumlandırma yapar.

Select işleminden sonra elde edilen toplam kayıt sayısı sınıfın size isimli fonksiyonuyla alınabilir. at fonksiyonu ise o anda imlecin konumlandırıldığı kaydın toplam elde edilen kayıttaki numarasını verir.

SQL Komutlarında Parametre Kullanımı

QSqlQuery sınıfında oluşturulan SQL cümlesinde parametre kullanılabilir. Parametreler isimli ya da isimsiz olabilmektedir. Eğer parametreler isimliyse bunlar önce bir ':' karakteri sonra ona yapışık bir isimle belirtilir. İsimli parametreler diğer platformlarda olduğu gibi '?' karakteriyle belirtilmektedir.

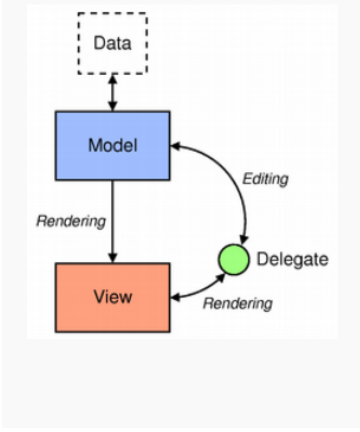
QSqlQuery sınıfında prepare isimli üye fonksiyon bizim parametrik komut girmemize olanak sağlamaktadır. prepare fonksiyonunda SQL cümlesi parametrik olarak verilir. Sonra QSqlQuery sınıfının bindValue fonksiyonlarıyla parametreler yerine onların değerleri yerleştirilir. Burada string ve tarih türlerinin tek tırnaklanması gerek yoktur. bindValue fonksiyonları zaten bunu bizim için yapmaktadır. En sonunda void parametrelili exec fonksiyonuyla sorgu VTYS'ye işletilir. Örneğin:

```
QSqlQuery query;
```

```
query.prepare("INSERT INTO person (student_name,student_no , student_bdate) VALUES (:name, :no, :bdate)");
query.bindValue(":name", "Ali Serçe");
query.bindValue(":no", 1234);
query.bindValue(":bdate", QDate(1980, 12, 21));
if (!query.exec()) {
    //...
}
```

Qt'de MVC (Model-View-Controller) Mimarisinin Temelleri

Aslında Qt'de veri katmanı ile görüntü (yani GUI) katmanını ve bunları yöneten katmanı tamamen birbirlerinden ayırmak mümkündür. Bu mimariye (ya da tasım kalıbına) MVC (Model-View-Controller) denilmektedir. MVC modelinde veri tedarik eden yani veriyi temsil eden sınıflara Model sınıfları denilmektedir. Tabii veriler Model sınıflarının içerisinde olmak zorunda değildir. Model sınıfları da veriyi başka yerden alıyor olabilir. Ancak bu veriyi uygun biçimde View katmanına iletmek Model sınıflarının görevidir. View sınıfları GUI elemanlardan oluşur. Bunlar verileri görüntülemek için kullanılırlar. View sınıfları belli arayüzlerle (sanal fonksiyonlarla) verileri Model sınıflarından alırlar. Bazı mimarilerde yalnızca Model ve View katmanları vardır. (Örneğin MFC'de mimari böyledir ve orada buna "Document/View" mimarisi denilmektedir.) Controller katmanı view ile kullanıcının etkileşimi sonucunda Model katmanının bundan haberdar edilmesi görevini yapar. Aslında Aslında Controller genel olarak Model ve View arasındaki etkileşimi sağlamaktadır.



Burada Delege bir çeşit Controller görevindedir.

MVC mimarisinden amaç veriyi oluşturan sınıflarla görüntüleyen sınıfları birbirlerinden ayırmak böylece bir veriyi değişik GUI sınıflarının zahmetsiz olarak kendi özgü bir biçimde görüntülemesini sağlamaktadır.

Qt'de XXXXModel biçiminde isimlendirilen çeşitli Model sınıfları ve XXXView biçiminde isimlendirilen çeşitli View sınıfları vardır. Bu View sınıflarının setModel isimli fonksiyonları herhangi bir model sınıfını alabilmektedir. Çünkü Model sınıflarının hepsi QAbstractItemModel isimli bir sınıftan türetilmiştir. Dolayısıyla setModel fonksiyonlarının parametresi de QAbstractItemModel türündendir.

Biz Qt programcısı olarak önce bir XXXModel sınıfı türünden nesneler oluşturacağız. Bu nesneye verilerimizi bağlayacağız. Sonra bir de XXXView GUI nesnesi yaratıp bu nesnenin setModel fonksiyonuyla Model nesnesini View nesnesiyle ilişkilendireceğiz. Qt'de kullanılan tipik model sınıfları şunlardır: QStringListModel, QStandardItemModel, QFileSystemModel, QSqlQueryModel, QSqlTableModel, QSqlRelationalTableModel.

```
QStringList sl;

sl << "Ali" << "Veli" << "Sealmi" << "Ayşe" << "Fatma";
QStringListModel *slm = new QStringListModel(sl);

ui->m_listView->setModel(slm);
ui->m_treeView->setModel(slm);
```

Burada görüldüğü gibi QSqltringListModel isimli Model sınıfının amacı birtakım yazıları view için tedarik etmektir. Bu model sınıfı yazıları QStringList sınıfıyla bizden almıştır. Biz bu model sınıfını QListView ve/veya QTreeView nesnelerine verebiliriz. Bunlar bu verileri kendilerine uygun biçimde görüntülerler.

Model sınıflarına eleman eklemek, onlardan eleman silmek ya da onları güncellemek mümkündür. Model sınıflarının içerisindeki elemanlara nasıl erişilmektedir? Model sınıfı içerisindeki verilerin birer index değerleri vardır. Ancak bu index değerleri int gibi bir türle değil QModelIndex isimli bir türle temsil edilmiştir. Index bilgisi için genel olarak üç anahtar değer kullanılmaktadır: Satır numarası, sütun numarası ve üst düğüm indeksi. Eğer söz konusu Model nesnesi sütun içermiyorsa oraya 0 gibi değer geçilebilir. Ya da söz konusu Model nesnesi üst düğüm içermiyorsa üst düğüm indeksi için boş indeks geçilebilir. Model içerisindeki herhangi bir elemanın indeksi Model sınıflarının index isimli fonksiyonlarıyla elde edilebilir:

```
QModelIndex QAbstractListModel::index(int row, int column = 0, const QModelIndex & parent =
QModelIndex()) const;
```

Örneğin:

```
QStringList sl;

sl << "Ali" << "Veli" << "Sealmi" << "Ayşe" << "Fatma";
QStringListModel *slm = new QStringListModel(sl);
QModelIndex index = slm->index(3);
```

Burada bir QStringListModel nesnesi oluşturulmuştur. Bu model nesnesinin sütun ve üst eleman özellikleri yoktur. Dolayısıyla burada 3'üncü elemanın indeksi doğrudan satır numarası verilerek elde edilmiştir. Daha sonra indeks verilerek model ilgili indeksteki veri data fonksiyonuyla elde edilebilir. Örneğin:

```
QModelIndex index = slm->index(3);
QVariant variant = slm->data(index, 0);
QDebug() << variant.toString();
```

Belli bir indeksteki veri setData fonksiyonuyla da set değiştirilebilir. Örneğin:

```
QModelIndex index = slm->index(3);
slm->setData(index, "Tacettin");
```

Model sınıflarının insertRow ve insertRows fonksiyonlarıyla biz yeni modele birer eleman (satır) ekleyebiliriz. Sonra onun verisini setData ile set edebiliriz. Örneğin:

```
slm->insertRow(1);
index = slm->index(1);
slm->setData(index, "Cumhur");
```

Modeldeki toplam eleman rowCount fonksiyonuyla elde edilebilmektedir. Eleman silmek için removeRow ve removeRows fonksiyonları kullanılmaktadır. Model sınıflarının sort fonksiyonları belli bir sütuna göre sıraya dizme işlemi yapar.

QFileSystem model sınıfı dosya sistemini bir veri modeli olarak temsil eder. Tabii bu model için ideal view QTreeView sınıfıdır. Örneğin:

```
QFileSystemModel *fsm = new QFileSystemModel();
fsm->setRootPath("c:/windows");
```

```
ui->m_listView->setModel(fsm);
ui->m_treeView->setModel(fsm);
```

QTableWidget Sınıfı

Pek çok farklı platformda “List View” isminde sütunlara ve satırlara sahip GUI pencereleri bulunmaktadır. Bu pencerelerin veritabanı için kullanılan özel biçimlerine de genellikle “Data Grid View” denilmektedir. Qt’de ListView elemanına en çok benzeyen widget QTableWidget, DataGridView elemanına çok benzeyen widget ise QTableView isimli widget’tır. QTableWidget sınıfı aslında QTableView sınıfından türetilmiştir. QTableView kendi içerisinde veritabanına yönelik veri bağlama (data binding) özelliklerine sahiptir. Dolayısıyla bu widget sayesinde veritabanı kayıtları kolay bir biçimde görüntülenebilmektedir. QTableWidget sınıfı şöyle kullanılır:

1) QTableWidget sınıfının setColumnCount ve setRowCount isimli fonksiyonları kontroldeki sütun ve satır sayısının belirlenmesi için kullanılır. (Qt Creator’ın designer’ında bu işlem görsel olarak yapılabilir. İşin başında özellikle kolonların belirlenmesi kolaylık sağlar.

2) Sütunları isimlendirmek için QTableWidget sınıfının setHorizontalHeaderLabels fonksiyonu kullanılır. Bu fonksiyon bizden sütun başlıklarını QStringList nesnesi olarak ister. Örneğin:

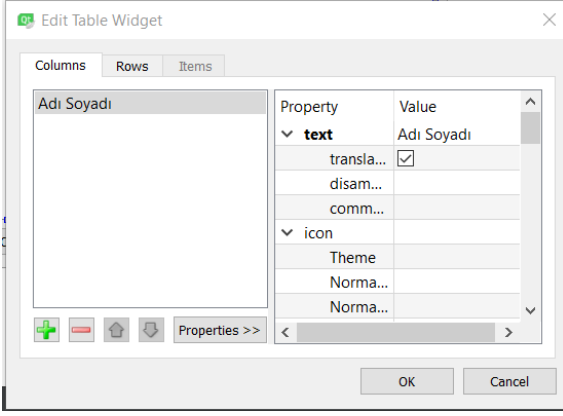
```
ui->m_tableWidget->setColumnCount(3);
ui->m_tableWidget->setRowCount(10);

QStringList sl;

sl << "Adı Soyadı" << "No" << "Doğum Tarihi";
ui->m_tableWidget->setHorizontalHeaderLabels(sl);

// ui->m_tableWidget->setHorizontalHeaderLabels(QString("Adı Soyadı;No;Doğum Tarihi").split(';'));
```

Bu işlemler de “designer’da” görsel olarak yapılabilir.



3) Şimdi sıra artık satırların oluşturulmasına gelmiştir. QTableWidgetItem penceresinde her hücre QTableWidgetItem isimli bir sınıfla temsil edilmiştir. QTableWidgetItem sınıfının setItem isimli üye fonksiyonları satır ve sütun numarasını ve QTableWidgetItem nesnesini alarak programlama yoluyla o hücreyi set eder. Örneğin:

```
QTableWidgetItem *qtwi = new QTableWidgetItem();
qtwi->setText("Ali Serçe");
qtwi->setTextColor(Qt::red);
qtwi->setBackgroundColor(Qt::yellow);

ui->m_tableWidget->setItem(0, 0, qtwi);
```

Burada QTableWidgetItem nesnesinin bizim tarafımızdan heap'te yaratılmış olması gerekmektedir. Çünkü QTableWidgetItem sınıfı QTableWidgetItem nesnelerinin adreslerini tutarak onlara erişir. Qt'de bu yaklaşım pek çok yerde kullanılmaktadır. Ancak programcı nesnenin kopyasını alarak mı nesnenin kullanıldığı, yoksa adresini alarak mı kullanıldığı konusunda tereddütler yaşayabilir. Bunu dokümantasyona bakmadan Qt'de anlamamanın bir yolu fonksiyon parametresini incelemektir. Eğer fonksiyon parametresi referans ise o aldığı nesnenin kopyasını kendisine alıp onu kullanmaktadır. Gösterici ise adresini alıp onu kullanmaktadır. Yani fonksiyonun parametresi gösterici ise bizim bu nesnenin o fonksiyona ilişkin sınıf nesneyi kullandığı sürece onu yaşıyor durumda olduğunu garanti etmemiz gerekir. Bu tür durumlarda bizim ilgili nesneyi heap'te new operatörüyle yaratmamız uygun olur. Fonksiyon bizden nesnenin adresini alıyorsa onun delete edilmesini kendisi yapmaktadır. Bizim ayrıca bu delete işlemi uğraşmamıza gerek yoktur. Örneğin biz QTableWidgetItem sınıfına heap'ta yarattığımız QTableWidgetItem nesnesinin adresini verdiğimizde bu QTableWidgetItem nesnesi yok edildiğinde bizim heap'te yarattığımız QTableWidgetItem nesnesi de delete edilecektir. Örneğin:

```
void setHorizontalHeaderLabels(const QStringList &labels);
void setItem(int row, int column, QTableWidgetItem *item);
```

Bu prototiplerden birinci fonksiyonun QStringList nesnesinin içeriğini alıp onu kendi içerisinde kullandığını, ikinci fonksiyonun ise nesnenin adresini alıp onu kendi içerisine hiç kopyalamadan onu adresi yoluyla kullandığını anlamalıyız.

QTableWidgetItem sınıfının horizontalHeaderItem ve verticalHeaderItem fonksiyonları yatay ve dikey başlık kısımlarını bize QTableWidgetItem nesnesi olarak verir. Benzer biçimde setHorizontalItem ve setVerticalItem fonksiyonları da bunların set edilmelerini sağlar.

Sınıfın selectedItems isimli fonksiyonu seçili olan tüm hücreleri bize QList<QTableWidgetItem*> olarak vermektedir. Yine programlama yoluyla setCurrentCell fonksiyonu ile o andaki aktif olan hücre değiştirilebilir.

Genel Olarak View Sınıfları ve QTableView Sınıfı

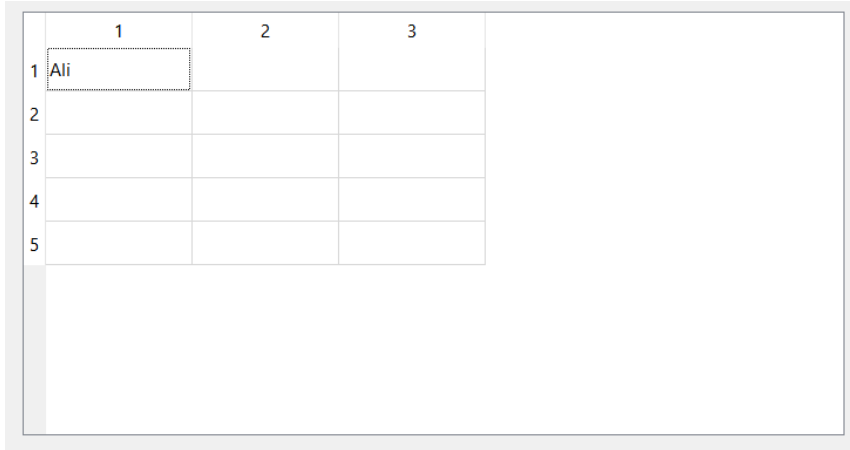
XXXView sınıfları Model sınıflarıyla birlikte kullanılmaktadır. XXXWidget sınıfları aslında XXXView sınıflarından türetilmiştir. Bu sınıflarda default bir model kullanılmıştır. Bu durumda biz aslında örneğin QListView, QTableView ya da QTreeView sınıflarını kullanırken bir model sınıfını da kullanmamız gerekir.

Qt'de QStandardItemModel isimli default bir model sınıfı tanımlanmıştır. Bu view sınıfları istenirse sıfırdan bu QStandardItemModel sınıfı ile oluşturulup kullanılabilir. Örneğin biz bir QTableView nesnesi ile birtakım verileri görüntülemek istersek önce bir model nesnesini oluşturmamız gerekir. Çeşitli model nesneleri var olsa da en klasik olanı QStandardItemModel sınıfıdır. Bu model sınıfı satır ve sütunlardan oluşan bir veriyi betimlemekte kullanılmaktadır. XXXView sınıflarının setModel fonksiyonları ile model bir view'ya set edilebilir.

QStandardItemModel sınıfının index isimli fonksiyonu bizden satır ve sütun değerini alarak ilgili elemanın index değerini QModelIndex olarak verir. Sonra setData fonksiyonuyla o satır ve sütundaki hücre set edilebilir. Set edilecek değer herhangi bir türden olabilir. Çünkü bunun ilgili parametresi QVariant türündendir. Örneğin:

```
QStandardItemModel *im = new QStandardItemModel(5, 3);
im->setData(im->index(0, 0), "Ali");
ui->m_tableView->setModel(im);
```

Burada biz 5 satır, 3 sütunluk bir model nesnesi oluşturduk, sonra onun 0, 0 elemanına Ali yazısını yerleştirdik. Sonra da model nesnesini QTableView nesnesine bağladık:



	1	2	3
1	Ali		
2			
3			
4			
5			

QStandardItemModel sınıfının insertRow isimli fonksiyonları belli bir satırı belli bir pozisyona insert ederler. Örneğin:

```
QStandardItemModel *im = new QStandardItemModel(5, 3);
im->setData(im->index(0, 0), "Ali");
im->setData(im->index(0, 1), "Veli");
im->setData(im->index(0, 2), "Selami");
im->insertRow(0);
im->setData(im->index(0, 0), "Sibel");
im->setData(im->index(0, 1), "Süleyman");
im->setData(im->index(0, 2), "Sacit");
ui->m_tableView->setModel(im);
```

Bu durumda şöyle bir görüntü elde edilecektir:



	1	2	3
1	Sibel	Süleyman	Sacit
2	Ali	Veli	Selami
3			
4			
5			
6			

QStandardItemModel sınıfının setHeaderData isimli fonksiyonları bizim modelin yatay ve dikey başlığındaki değerleri belirlememize olanak sağlar. Örneğin:

```
QStandardItemModel *im = new QStandardItemModel(5, 3);
im->setData(im->index(0, 0), "Ali Serçe");
im->setData(im->index(0, 1), "123");
im->setData(im->index(0, 2), "12/01/1999");
im->setHeaderData(0, Qt::Horizontal, "Adı Soyadı");
```

```
im->setHeaderData(1, Qt::Horizontal, "No");
im->setHeaderData(2, Qt::Horizontal, "Doğum Tarihi");
ui->m_tableView->setModel(im);
```

Şöyle görüntü elde edilecektir:

	Adı Soyadı	No	Doğum Tarihi
1	Ali Serçe	123	12/01/1999
2			
3			
4			
5			

MVC tasarım kalıbında Model sınıfının veriyi temsil ettiğini View sınıflarının ise görsel öğeleri temsil ettiğini anımsayınız. Bu durumda biz QTableView'daki içeriği model sınıfı ile oluştururuz, ancak görüntü biçimini (bold, italic vs.) view sınıfları ile oluştururuz. Örneğin yukarıdaki QTableView nesnesindeki sütun başlıklarını bold yapmak isteyelim:

```
QStandardItemModel *im = new QStandardItemModel(5, 3);
im->setData(im->index(0, 0), "Ali Serçe");
im->setData(im->index(0, 1), "123");
im->setData(im->index(0, 2), "12/01/1999");
im->setHeaderData(0, Qt::Horizontal, "Adı Soyadı");
im->setHeaderData(1, Qt::Horizontal, "No");
im->setHeaderData(2, Qt::Horizontal, "Doğum Tarihi");
ui->m_tableView->setModel(im);
```

```
QFont font("Times New Roman", 12);
font.setBold(true);
ui->m_tableView->horizontalHeader()->setFont(font);
```

Biz QStandardItemModel nesnesindeki belli bir hücrenin içerisindeki değeri nasıl elde edebiliriz? Önce index fonksiyonuyla satır ve sütun belirtip ilgili hücrenin indexini almaya çalışırız. Sonra o indeksi kullanarak data fonksiyonuyla oradaki veriyi elde ederiz. Örneğin:

```
QDebug() << m_im->data(m_im->index(0, 1)).toString();
```

QTableView sınıfının setSelectionBehavior isimli üye fonksiyonu seçimin türünü belirlememize olanak sağlar. Default durum hücrelerin bireysel seçilmesidir. Biz istersek tüm satırın seçilmesini sağlayabiliriz.

QTableView sınıfının currentIndex isimli fonksiyonu bize o anda aktif olan satıra ilişkin index değerini verir. Bu index değerinden hareketle biz data ya da item isimli fonksiyonları kullanarak oradaki veriyi elde edebiliriz. data bizden yalnızca index değerini alarak QVariant verir. item ise bizden satır ve sütun sumarasını alarak ilgili değeri QStandardItem nesnesi olarak verir.

QTableView sınıfının resizeColumnsToContents ve resizeRowsToContents fonksiyonları sütunları ve satırları içerindeki elemanlara göre fazla boşluk bırakmayacak biçimde ayarlar. Yine sınıfın hideColumn isimli fonksiyonu belli bir sütunun gösterilmemesini sağlar.

QTableView sınıfının currentIndex elemanı bize seçili olan satırın ilk sütununa ilişkin index değerini verir. Eğer seçim biçimi hücrese currentIndex bize seçilmiş hücrenin indeksini verir. eğer seçim satırsal ise bu durumda o satırın ilk sütununun indeksini verir. setCurrentIndex ise bunu programlama yoluyla değiştirmek için kullanılır.

Veritabanına İlişkin Model Sınıfları

Veritabanı için temelde üç model sınıfı vardır: QSqlQueryModel, QSqlTableModel ve QSqlRelationalTableModel.

QSqlQueryModel Sınıfı

QSqlQueryModel read-only bir model sınıfıdır. Yani biz bu model sınıfının verileri üzerinde ya da onu bir View sınıfına bağladığımızda orada değişiklik yapamayız. QSqlQueryModel sınıfı şöyle kullanılır:

1) Nesne dinamik olarak sınıfın başlangıç fonksiyonuyla yaratılır:

```
QSqlQueryModel *qm = new QSqlQueryModel();
```

2) Sınıfın setQuery fonksiyonuyla select cümlesi oluşturulur. Örneğin:

```
qm->setQuery("SELECT * FROM city");
```

3) Model nesnesi doğrudan QListView, QTableView gibi view sınıflarına bağlanabilir. Örneğin:

```
qDebug() << m_db.open();
```

```
QSqlQueryModel *qm = new QSqlQueryModel();  
qm->setQuery("SELECT * FROM city");  
ui->m_tableView->setModel(qm);
```

4) Eğer istenirse kayıtlar yine programlama yoluyla tek tek de ele geçirilebilir. QSqlQueryModel sınıfının record isimli fonksiyonları select edilmiş kayıtlar içerisindeki belli bir kaydı bize QSqlRecord türünden bir nesne olarak verir. QSqlRecord sınıfının value isimli fonksiyonları sütun id'sini ya da sütun ismini parametre olarak alıp bize QVariant biçiminde onun içeriğini verir. Örneğin veritabanındaki select ettiğimiz kayıtları şöyle görüntüleyebiliriz:

```
QSqlQueryModel *qm = new QSqlQueryModel();  
qm->setQuery("SELECT Name, CountryCode FROM city ORDER BY Name");  
  
for (int i = 0; i < qm->rowCount(); ++i) {  
    QSqlRecord rec = qm->record(i);  
    qDebug() << rec.value("Name").toString() << ", " << rec.value("CountryCode").toString();  
}
```

Şüphesiz biz INNER JOIN işlemiyle farklı tablolardan bilgileri benzer biçimde alıp görüntüleyebiliriz. Örneğin:

```
QSqlQueryModel *qm = new QSqlQueryModel();  
qm->setQuery("SELECT city.Name, country.Name FROM city, country WHERE city.CountryCode =  
country.Code ORDER BY city.Name");  
ui->m_tableView->setModel(qm);
```

QTableView'daki sütunlardaki isimler QTableView sınıfı kullanılarak değiştirilebileceği gibi aslında QSqlQueryModel sınıfı yoluyla da değiştirilebilir. Bunun için QSqlQueryModel sınıfının setHeaderData fonksiyonları kullanılır. Örneğin:

```
QSqlQueryModel *qm = new QSqlQueryModel();  
qm->setQuery("SELECT city.Name, country.Name FROM city, country WHERE city.CountryCode =  
country.Code ORDER BY city.Name");  
ui->m_tableView->setModel(qm);  
ui->m_tableView->verticalHeader()->setVisible(false);  
qm->setHeaderData(0, Qt::Horizontal, "Şehir İsmi");  
qm->setHeaderData(1, Qt::Horizontal, "Ülke İsmi");
```

Default durumda setModel yapıldığında tablolardaki sütun isimleri QTableView sınıfının başlık kısmında görüntülenir.

Kayıtları QTableView içerisine yerleştirdikten sonra o anda tabloda seçili olan satırı alabiliriz. Bunun için QTableView sınıfının selectedIndexes fonksiyonları kullanılır. Bu fonksiyonlar bize seçili olan elemanların indekslerini bir liste biçiminde vermektedir.

Şehir İsmi	Ülke İsmi
Aba	Nigeria
Abadan	Iran
Abaetetuba	Brazil
Abakan	Russian Federat...
Abbotsford	Canada
Abeokuta	Nigeria
Aberdeen	United Kingdom
Abha	Saudi Arabia
Abidjan	Côte d'Ivoire
Abiko	Japan

QSqlTableModel Sınıfı

Bu model sınıfı read-write işlemine izin vermektedir. Yani biz veritabanından belli kayıtkarı select ettikten sonra programlama yoluyla ya da View sınıfları yoluyla veritabanında değişiklikler yapabiliriz. QSqlTableModel sınıfının kullanımı şöyledir:

1) QSqlTableModel nesnesi default başlanıç fonksiyonuyla yaratılır. Sonra setTable fonksiyonuyla tablo, setFilter fonksiyonuyla koşul (WHERE cümlecisi) belirtilir. Nihayet select fonksiyonu çağrılarak veriler veritabanından elde edilir. Nesne default bağlantıyı kullanmaktadır. Eğer default bağlantı yerine başka bir bağlantı kullanılacaksa bu durumda QSqlDatabase nesnesi sınıfın başlanıç fonksiyonuna geçirilmelidir. Örneğin:

```
QSqlTableModel *tm = new QSqlTableModel();

tm->setTable("city");
tm->setFilter("Name LIKE 'K%'");
// Eşdeğeri: SELECT * from city WHERE Name LIKE K%
tm->select();
ui->m_tableView->setModel(tm);
ui->m_tableView->resizeRowsToContents();
ui->m_tableView->hideColumn(0);
```

2) Model sınıfının setHeaderData fonksiyonuyla sütun isimlerini değiştirebiliriz. Örneğin:

```
QSqlTableModel *tm = new QSqlTableModel();

tm->setTable("city");
tm->setFilter("Name LIKE 'K%'");
tm->setHeaderData(1, Qt::Horizontal, "Şehir İsmi");
tm->setHeaderData(2, Qt::Horizontal, "Ülke Kodu");
tm->setHeaderData(3, Qt::Horizontal, "Bölge");
tm->setHeaderData(4, Qt::Horizontal, "Nüfus");
tm->select();
```

3) QSqlTable model sınıfının editStrategy isimli fonksiyonu bize modlein editleme stratejisini verir. Üç strateji vardır:

QSqlTableModel::OnFieldChange: Bu durumda ilgili hücre değiştirildiğinde hemen değişiklik veritabanına yansıtılır.

QSqlTableModel::OnRowChange: Bu durumda değişiklik yeni bir satır aktif hale getirilince veritabanına yansıtılır.

QSqlTableModel::OnManualSubmit: Bu durumda değişiklik biz özellikle submitAll fonksiyonunu çağırdığımızda veritabanına yansıtılır.

Default durum satır değiştiğinde değişikliklerin veritabanına yansıtılmasıdır. Programcı isterse setEditStrategy fonksiyonuyla bu durumu değiştirebilir.

4) Biz aslında hiç view nesnesi kullanmadan yalnızca QSqlTableModel nesnesiyle de çeşitli veritabanı işlemlerini yapabiliriz. Tıpkı QSqlQueryModel sınıfında olduğu gibi QSqlTableModel sınıfının da record isimli üye fonksiyonu bize ilgili kaydı (yani satırı) QSqlRecord nesnesi biçiminde verir. Örneğin:

```
for (int i = 0; i < m_tm->rowCount(); ++i) {
    QSqlRecord rec = m_tm->record(i);
    qDebug() << rec.value(1).toString() + " " + rec.value("Population").toString();
}
```

Fakat QSqlTableModel sınıfı read/qwrite bir model sunmaktadır. Yani biz record üzerinde bir değişiklik yaptığımızda o değişiklik veritabanına yansıtılabilir. Örneğin:

```
QSqlRecord rec = m_tm->record(0);
qDebug() << rec.value(1).toString();
rec.setValue(1, "Ankara");
m_tm->setRecord(0, rec);
```

Burada önce ilgili Record nesnesi alınmış, sonra içerisinde değişiklik yapıldıktan sonra yeniden set edilmiştir. Bu tür değişiklikler QSqlTableModel sınıfının setData fonksiyonlarıyla da yapılabilir. Ancak setData fonksiyonu satır numarasıyla değil hücresel olarak index ile çalışmaktadır. Örneğin o anda seçili olan hücredeki bilgiyi değiştirecek olalım:

```
QModelIndex index = ui->m_tableView->currentIndex();
m_tm->setData(index, "Eskişehir");
m_tm->submitAll();
```

Benzer biçimde insert işlemi şöyle yapılabilir:

Kayıt silme işlemi için sınıfın removeRow ya da removeRows fonksiyonları kullanılmaktadır. Örneğin:

```
QModelIndex index = ui->m_tableView->currentIndex();
m_tm->removeRow(index.row());
qDebug() << m_tm->submitAll();
```

QSqlRelationalTableModel Sınıfı

Bu model sınıfı özellikle tablo geçişleri için düşünülmüştür. Tipik kullanımı şöyledir:

1) Öncelikle sınıfın setTable fonksiyonu ile veritabanı tablosu seçilir. Sonra setFilter fonksiyonu ile filtre belirtilebilir.

2) Daha sonra QSqlRelation nesneleri oluşturulur. QSqlRelation nesneleri oluşturulurken sınıfın başlangıç fonksiyonunda üç argüman girilir: Birinci argüman ilişki kurulacak tabloyu belirtir. İkinci argüman ilişki kurulacak tablodaki yabancı anahtarla eşleşecek sütunun ismini belirtmektedir. Üçüncü argüman ise eşitlik koşulu sağlandığında ilişki kurulacak tablodaki hangi değerın modele alınacağını belirtir. QSqlRelation nesnesi oluşturulduktan sonra sınıfın setRelation fonksiyonu ile bir sütun da belirtilerek eşleştirme yapılır:

Örneğin:

```
m_rtm = new QSqlRelationalTableModel();
m_rtm->setTable("city");
m_rtm->setRelation(2, QSqlRelation("country", "Code", "Name"));
```

```
m_rtm->setHeaderData(1, Qt::Horizontal, "Şehir");
m_rtm->setHeaderData(2, Qt::Horizontal, "Ülke");
m_rtm->setHeaderData(3, Qt::Horizontal, "Bölge");
m_rtm->setHeaderData(4, Qt::Horizontal, "Nüfus");
ui->m_tableView->setModel(m_rtm);
m_rtm->select();
```

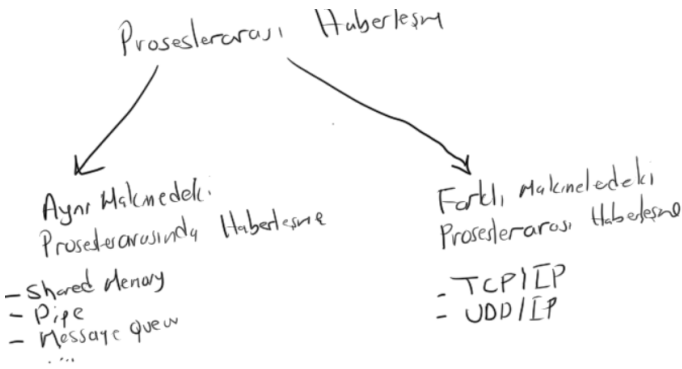
Burada city tablosundaki 2 numaralı sütun üç karakterli ülke kodunu belirtmektedir. İşte bu ülke kodu country tablosundaki "Code" alanı ile aynıysa bunun yerine country tablosundaki "Name" ile belirtilen ülkenin ismi getirilecektir.

	ID	Şehir	Ülke	Bölge	Nüfus
7	57	Huambo	Angola	Huambo	163100
8	58	Lobito	Angola	Benguela	130000
9	59	Benguela	Angola	Benguela	128300
10	60	Namibe	Angola	Namibe	118200
11	61	South Hill	Anguilla	Â-	961
12	62	The Valley	Anguilla	Â-	595
13	34	Tirana	Albania	Tirana	270000
14	55	Andorra la Vella	Andorra	Andorra la Vella	21189
15	33	Willemstad	Netherlands An...	Curaçao	2345
16	64	Dubai	United Arab Em...	Dubai	669181
17	65	Abu Dhabi	United Arab Em...	Abu Dhabi	398695
18	66	Sharja	United Arab Em...	Sharja	320095
19	67	al-Ayn	United Arab Em...	Abu Dhabi	225970
20	68	Ajman	United Arab Em...	Ajman	114395
21	69	Buenos Aires	Argentina	Distrito Federal	2982146
22	70	La Matanza	Argentina	Buenos Aires	1266461

QSqlRelationalTableModel sınıfı da read-write işlemine izin vermektedir. Yani QSqlTableModel sınıfıyla yapılan işlemler bu sınıfta da yapılabilir. Ancak bu işlemlerden sonra aslında iki tablonun da güncelleneceğine dikkat ediniz.

Proseslerarası Haberleşme

İşletim sistemlerinde çalışmakta olan programlara proses denilmektedir. Bir prosesin başka bir prosese veri göndermesi diğerinin de alması sürecine proseslerarası haberleşme denir. Proseslerarası haberleşme aynı makinenin prosesleri arasında haberleşme ve farklı makinelerin prosesleri arasında haberleşme olmak üzere iki kısma ayrılmaktadır. Tabii farklı makinelerin prosesleri arasında haberleşme aynı zamanda aynı makine üzerinde de uygulanabilmektedir.



Aynı makinenin prosesleri arasında haberleşmede işletim sistemleri birbirine benzer biçimde birkaç yöntem kullanmaktadır. Bu yöntemlerin bazılarını işletim sisteminden bağımsız bir biçimde Qt’de özel sınıflarla kullanabilmekteyiz. Örneğin Qt’nin QSharedMemory sınıfı “paylaşılan bellek alanları (shared memory)” denilen proseslerarası haberleşme yönteminde kullanılabilir.

Her ne kadar farklı makinelerin prosesleri arasındaki haberleşme aynı makinedeki proseslerarası haberleşme için de kullanılabilir olsa da aynı makinenin prosesleri arasındaki haberleşmeler genel olarak daha hızlıdır. Fakat biz kursumuzda ancak vakit kalırsa aynı makinenin prosesleri arasındaki haberleşme yöntemlerini göreceğiz.

Şüphesiz farklı makinelerdeki proseslerin haberleşmesi için makinelerin birbirlerine bağlandığı bir ağ ortamının donanımsal olarak hazır olması gerekmektedir. Farklı makinelerin prosesleri arasında haberleşme için önceden saptanmış bazı kuralların işletim sistemleri tarafından kabul edilmesi ve uygulanması gerekmektedir. Haberleşmede belirlenmiş bu kurallara protokol denilmektedir.

Bugün farklı makinelerdeki prosesleri haberleştirmek için pek çok protok ailesi bulunmaktadır. Örneğin Apple firmasının AppleTalk ailesi, NetBIOS gibi aileler vs. Ancak IP protocol ailesi bu aileler arasında en çok tercih edilenidir. Çünkü Internet 1983 yılında bu aileyi kullanmaya başlamıştır. Böylece bu aile kullanılarak oluşturulmuş yazılımlar hem bireysel ağlarda (intanet) hem de Inter ortamında çalışabilmektedir.

Protokol Katmanları

IEEE bir protokol ailesinin hangi katmanlardan oluşarak gerçekleştirileceğini belirten OSI isminde bir doküman yayınladı. OSI (Open System Interconnection) katmanları toplam yedi tanedir.

Layer 7	Application	Communication Application
Layer 6	Presentation	
Layer 5	Session	SIP, RTP, RTCP
Layer 4	Transport	TCP, UDP, SCTP
Layer 3	Network	Ipv4, IPv6
Layer 2	Data Link	Ethernet, etc
Layer 1	Physical Link	Coax, RF link, etc

OSI’de en aşağı katmana “fiziksel katman” denilmektedir. Bu katmanda tamamen iletişim için gereken donanımsal ekipmanlar tanımlanır (Örneğin kabloların ve konnektörlerin özellikleri gibi). Fiziksel katmanın yukarısında “veri bağlantı katmanı (data link layer)” bulunmaktadır. Bu katmanın görevi ağa bağlı olan birimler arasında donanımsal temelli haberleşme ortamını oluşturmaktır. Bu katmanda kullanılan adresleme yöntemi fiziksel yöntemdir. Yani burada ağa bağlı olan her birimin değişmez ve donanımdan kaynaklanan bir adresi vardır. Haberleşme bu adres yoluyla yapılmaktadır. Bugün kullandığımız Ethernet kartlarında kullanılan Ethernet protokolü tpiipik bir veri bağlantı katmanını tanımlar. Veri bağlantı katmanının üzerine “Ağ Katmanı (Netword Kayer)” oturtulmuştur. Bu katmanda artık adresleme fiziksel olmaktan çıkartılmış ve mantıksal bir sisteme geçilmiştir. Adresler mantıksal olduğu için değiştirilebilmektedir. İletim Katmanında (Transport Layer) artık bilginin güvenli bir biçimde ve port numaralandırmasıyla ve gerekliyse akış kontrolü iel gönderilip alınmasına yönelik belirlemeler bulunmaktadır. Oturum katmanı (Session Layer) oturum açma esaslarını belirleyen katmandır. Oturumlar istemci-sunucu tarzı haberleşmelerde bir güvenlik eşliğinde açılabilir. Sunum katmanında (Presentation Layer) şifreleme, sıkıştırma gibi ek birtakım işlemler uygulanabilmektedir. Nihayet “Uygulama Katmanı (Application Layer)” yazılımcıların kendi programlarını temsil eden katmandır. Bu bakımdan her uygulama kendine özgü bir protokol içerebilir.

IP Protokol Ailesi

IP protokol ailesi (IP protocol family) bugün en yaygın kullanılan ağ protokol ailesidir. Ailede üst üste yığılmış durumda olan pek çok protok vardır. Aileyi ismini IP (Internet Protocol) denilen protokol vermiştir. IP ailesi 1983 yılından beri Internet’in resmi protokol ailesidir.

IP protokol ailesi ilk kez 70’li yılların ilk yarısında geliştirilmeye başlanmıştır. Zamanla aileye yeni üyeler katılmıştır. Orijinal geliştiricileri Robet Kahn, Vint Cerf isimli kişilerdir. IP ailesi paket anahtarlama (packet switching) ağlar için düşünülmüştür. Paket anahtarlama bilgileri bir paket halinde toplanıp, paket olarak gönderilip alınır.

IP protokol ailesi OSI’nin 7 katmanına sahip değildir. Dört katmanlı bir ailedir:

OSI Model		Internet Model
7	Application	FTP, TFTP, HTTP, SMTP, DNS, TELNET, SNMP
6	Presentation	Application
5	Session	
4	Transport	
3	Network	IP (the Internet)
2	Data Link	Network Interface
1	Physical	

IP protokol ailesinde ağa bağlı birimlere “host” denilmektedir. Oost duruma göre bir bilgisayar, bir kamera bir yazıcı vs. olabilir. Ağa bağlı olan her host’a ismine IP numarası denilen mantıksal adres verilir. Sonra bu protokolda bilgiler paketlenerek hedef IP numaraları belirtilerek host’tan host’a rotalanarak gönderilip alınır. IPV4’te IP numaraları 4 byte uzunluktadır. IPV6’da 16 byte’a yükseltilmiştir. Şu anda hala ağırlıklı olarak IPV4 kullanılmaktadır. Elimizde yalnızca IP protoloü olsaydı biz ne yapabilirdik? Biz yalnızca bir grup bilgiyi bir IP adresiyle belirtilen host’a yollardık. Ancak karşı tarafın bunu aldığını biz bu prokolde bilemezdik. IP protokolü tipik olarak OSI’nin ağ katmanını temsil etmektedir.

IP protokolünün üzerine oturtulmuş OSI’nin “Aktarım Katmanını (Transport Layer)” temsil eden iki protokol vardır: TCP ve UDP:



TCP (Transmission Control Protocol) bağlantılı (connection oriented) bir protokoldür. Bağlantılı protokollerde iletişim için önce bir taraf diğer tarafa bağlanır. Sonra iletişim gerçekleştirilir. Bu bağlantı sırasında iki taraf birbirlerinin bilgilerini elde ettiği için bir akış kontrolü uygulanabilmektedir. TCP güvenilir (reliable) bir protokoldür. Yani bu protoklda IP paketleri karşı tarafa gönderildikten sonra karşı taraftan onaylar alınır, gerekirse yeniden gönderilir. İki taraf arasında bir konuşma geçmektedir. Tabii TCP gönderip alma işlemini IP prokolüyle yapar. Ancak IP paketlerine numaralar vererek büyük bilgilerin sıra bozulmadan gönderilip alınmasını mümkün hale getirmektedir.

TCP stream tabanlı bir protokoldür. Stream tabanlı protokollerde gönderilen bilgilerin byte’ları bir boru gibi sıraya dizilerek karşı tarafta oluşturulur. Böylece bilgiyi alan taraf istediği kadarını okuyup, okumasına daha sonra devam edebilir. Stream tabanlı protokollerin zıddı “datagram tabanlı” protokollerdir. Burada datagram “paket” anlamındadır. Datagram protokllerde gönderilen paket yine paket olarak alınır. Onun bir kısmı alınıp duruma göre diğer kısmı alınmaz.

UDP (User Datagram Protocol) güvenli olmayan bir protokoldür. Akış kontrolü içermez. Yani UDP’de bilgi gönderildiğinde karşı tarafın bunu alıp almadığını gönderen taraf bilemez. UDP datagram tabanlıdır. Yani bilgiler paket olarak gönderilip paket olarak alınır. UDP güvenilir olmasa da TCP’den oldukça hızlıdır. Bu nedenle hayati önem içermeyen periyodik birtakım bilgilerin iletilmesinde tercih edilmektedir. UDP bağlantısız (connectionless) bir protokoldür. Yani bir tarafın diğer tarafa bilgi göndermesi için önceden bağlantı oluşturmaya gerek yoktur.

TCP ve UDP’de bir önemli kavram daha devreye girmektedir: Port numarası. Port numarası bir iş yerindeki içsel numaralara (dahili numara) benzetilebilir. Aynı host’a giden bilgileri birbirinden ayırmak için düşünülmüş sayısal bir içsel adrestir. Böylece biz bu protokollerde aynı host’un farklı portlarına bilgiler gönderebiliriz. O host’ta o portu dinleyen programlar kendi portlarına gelen bilgileri alırlar. IPV4’te prot numaraları 2 byte, IPV6’da 4 byte’tır. Böylece IPV4’te port numaraları 0-65535 arasındadır. Ancak ilk 1024 port numarası (well known ports) IP ailesinin uygulama katmanındaki prorokollere ayrılmıştır. Örneğin 21 numaralı portu FTP, 22 numaralı portu SSH, 80 numaralı portu

HTTP protokolleri kullanılmaktadır. O halde bizim kendi programlarımızda port numarası olarak 1024'ün yukarısında bir değer seçmemiz uygun olur.

TCP	UDP
Bağlantılı (Connection Oriented)	Bağlantısız (Connectionless)
Güvenli (Reliable)	Güvenli Değil (Not Reliable)
Stream tabanlı	Datagram (paket) tabanlı
Yavaş	Hızlı

IP protokol ailesinin TCP üzerine oturtulmuş pek çok Uygulama Katmanı (Application Layer) protokolü vardır.



Client-Server Haberleşme Modeli

Client-Server tarzı haberleşme modelinde iki program yazılır. Bunlardan birine server (sunucu), diğerine client (istemci) denilmektedir. Server asıl işi yapan programdır. Client ise yalnızca server'a yapılacak işi bildirir. Client-Server modelinin sunduğu avantajlar şunlardır:

- 1) Server makineye bir kaynak (yazıcı, kamera vs.) bağlı olabilir. Server bunu paylaşmak isteyebilir. Böylece makinenin kontrolünü server program yapar. Client'lar da ona istekte bulunurlar. Yani bu model kaynak paylaşımında çok sık kullanılmaktadır.
- 2) Server makine çok güçlü özelliklere sahip olabilir. Böylece biz onun gücünden faydalanmak isteyebiliriz. Örneğin çok büyük iki matrisi server çarpabilir, sonucu client'a gönderebilir.
- 3) Client-Server model güvenlik için çokça kullanılmaktadır. Örneğin bilgiler güvenli bir yerdeki server'da tutulur. Client'lar ona erişerek işlemlerini kullanıcı ve password ile yaparlar. Örneğin banka ATM'lerindeki yazılım client tarafa ilişkindir. Banka bilgileri server'daki veritabanındadır. VTYS'leri de client-server tarzda çalışan sistemlerdir.
- 4) Bir grup arasında iletişim sağlayabilmek için grubun üyelerinin birbirleri ile iletişim kurması gerekmez. Herkes server ile ilişki kurar. Server aradaki mesaj koordinasyonunu sağlayabilir.
- 5) Bir server birden fazla client'a hizmet verecek biçimde yazılabilir. Bu da kaynakların verimli kullanılması anlamına gelir.

TCP/IP tipik olarak client-server çalışma modelini dikte ettirmektedir. O halde bu modelde biz iki program yazmak durumundayız: Server program ve client program. Şüphesiz server programların organizasyonu client programlara göre daha zordur.

IP Adresleri ve Host İsimleri

Sayılar bilgisayar sistemleri için daha uygundur. Ancak sayılar akılda kalıcı değildir. Bu nedenle genellikle bilgisayar sistemleri sayılarla işlemleri yaparken bunlara birer isim de karşılık getirilir. İşte IP ailesinde de bir host'un IP adresinin yanı sıra onun bir de ismi vardır. İsmi bilinen host'un IP adresi, IP adresi bilinen host'un ismi elde edilebilir. Aslında IP adresleriyle host isimleri bire bir eşlenmek zorunda değildir. Bir IP adresi için birden fazla host ismi, bir host ismi için de birden fazla IP adresi bulunabilir. Protokol IP adreslerinin sayısal değerleriyle oluşturulmuştur. O halde isimler önce IP adreslerine dönüştürülür. Qt kütüphanesi içerisinde bu işlem otomatik yapıldığı için programcı bunun farkına varmayabilir.

Internet'te host isimlerini IP adreslerine karşı getiren özel server'lar vardır. Bunlar için de özel protokoller tasarlanmıştır. Bu server'lara DNS (Domain Name Server) denilmektedir.

Internet'in Kısa Tarihi

Bilgisayarları birbirlerine bağlamak ilk kez 60' yıllarda insanların aklına gelmiştir. Soğuk savaş yıllarında Amerika Savunma Bakanlığına bağlı olan DARPA (Defense Advanced Research Project Agency) kurumu birkaç üniversite ile birlikte 1969 yılında ARPANET isimli bir proje başlattı. ARPANET ilk kez 1969 yılında uzak mesafeden dört üniversitenin birbirlerine bağlanmasıyla hayata geçirilmiş oldu. ARPANET'te daha sonra bazı devlet kurumları ve üniversiteeler katılmaya başlamıştır. 70'li yılların sonlarına doğru ARPANET Amerika'da gelişmeye başlamıştır. 1983 yılında ARPANET NCP (Network Control Protocol) protokolünü bırakarak IP ailesine ailesine geçmiştir. Ve art ık ağ Internet ismiyle yayılmaya devam etmiştir. Internet 80 yıllarda Avrupa'ya ve Türkiye'ye de geldi. Ancak tabi kişsel bilgisayarlar daha yeniydi ve Internet'e ancak Üniversitelerden ve bazı devlet kurumlarından, özel sektörden bağlanılabiliyordu. 1990-91 yıllarında HTTP protokü tasarlandı ve ilk Web sayfaları oluşturulmaya başlandı. 90'lı yılların ortalarına doğru tüm dünyada kişisel bilgisayarlarla servis sağlayıcılar sayesinde Internet'e girmek mümkün hale gelmiştir. Daha sonraları modern modem/router'larla yüksek hızlı evden erişimler sağlanmıştır.

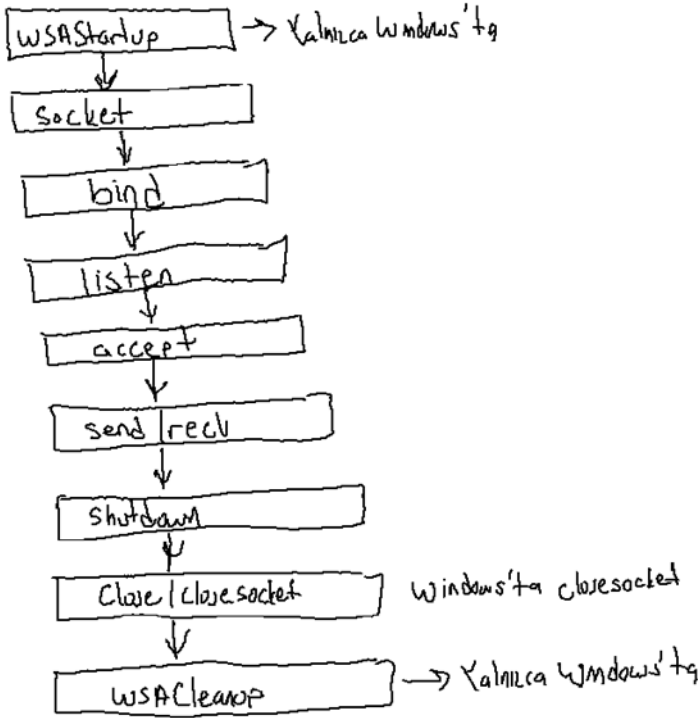
Internet ismi “internetworking” sözcüğünden gelmektedir. Internetworking “yerel ağların birbirlerine router isimli cihazlarla bağlanmalarıyla oluşturulmaktadır. Internetworking temel bir terimdir ve IP protokol ailesinin ismi buradan gelmektedir. Bugün Internet denildiğinde herkesin bağlandığı ARPANET'ten evrimleşen dev ağ aklıma gelir. (Internet yazarken l'yı büyük yazarsak bu ağ anlaşılır.) Şüphesiz mevcut protokoller sayesinde herkes kendi internetini kurabilir. Örneğin biz de birkaç arkadaşınızla ayrı bir Internet dünyası oluşturabiliriz. Hatta bazı ülkelerin bu biçimde kendilerine özgü Internet'leri vardır.

Soket Kavramı

Soket (socket) terimi maalesef isim olarak yanlış anlaşılmaya müsait bir terimdir. Bu bağlamda soket ağ haberleşmesi için kullanılan bir veri yapısını ve buna bağlı kütüphaneleri temsil etmektedir. Soket kütüphanesi ilk kez BSD türevi UNIX sistemleri için 1983'te gerçekleştirilmiştir. Bu kütüphane yalnızca IP ailesi için değil diğer aileler için de ortak bir arayüz sunmaktadır. Microsoft BSD'nin soket kütüphanesini alarak bir benzerini Windows sistemleri için oluşturmuştur. Buna Winsock denilmektedir. Bugün biz aşağı seviyeli olarak ağ haberleşmesi için Linux ve Mac OS X sistemlerinde BSD soket kütüphanesini, Windows sistemlerinde de Winsock kütüphanesini kullanmaktayız. Qt bu soket sistemini bize sınıfsal bir biçimde ve daha kolay kullanılacak bir biçimde sunmaktadır. Tabii Qt Linux ve Mac OS X sistemlerinde BSD türevi soket kütüphanesini Windows sistemlerinde de Microsoft'un Winsock kütüphanesini arka planda kullanmaktadır.

TCP Server Programın Organizasyonu

Qt kütüphanesi server programın daha kolay bir biçimde yazılmasına olanak sağlamaktadır. Ancak soket kütüphanesi kullanılarak tipik bir TCP server program aşağıdaki aşamalardan geçilerek yazılmaktadır:



Server programda sırasıyla önce bir soket yaratılmakta sonra o sokent bağlanmakta (bind edilmektedir). Soket bağlanması server'ın hangi porttan ve network kartından gelen bağlantıları kabul edeceğinin belirlenmesi sürecidir. Daha sonra soket listen fonksiyonuyla aktif dinlemene konumuna sokulur. Bundan sonra gelen bağlantı istekleri accept fonksiyonuyla kabul edilir. Sonra da send ve recv fonksiyonlarıyla bilgi gönderip alma işlemleri yapılır.

Qt yukarıdaki işlemleri kendi içerisinde yapabilen QTcpServer isimli bir sınıf bulunmaktadır. Bu sınıf sayesinde biz yukarıdaki adımları hızlı bir biçimde geçebilmekteyiz.

Qt'de soket sınıflarını kullanabilmek için qmake dosyasında "network" modülünü eklemek gerekmektedir.

Tipik bir TCP server programı Qt'de şu aşamalardan geçilerek oluşturulur:

- 1) Öncelikle bir tane QTcpServer sınıfı türünden nesne yaratılır. Bu yaratım sırasında aşağı seviyeli soket nesnesi de yaratılmaktadır. Nesnenin yaratımında QTcpServer sınıfının default başlangıç fonksiyonu kullanılır. Biz bu nesneyi, başka bir sınıfın veri elemanı olarak bildirebiliriz.
- 2) QTcpServer nesnesi ile sınıfın listen fonksiyonu çağrılarak arka planda soket bağlanması ve dinleme konumuna sokulması sağlanır.

```
bool QTcpServer::listen(const QHostAddress &address = QHostAddress::Any, quint16 port = 0)
```

Fonksiyonun birinci parametresi hangi network kartından gelen bağlantı isteğinin kabul edileceğini belirtir. Bu parametre QHostAddress::Any biçiminde girilirse host'a bağlı tüm network kartlarından gelen bağlantı isteklerinin değerlendirileceği anlaşılır. İkinci parametre bağlantıda kullanılacak port numarasını belirtmektedir.

Örneğin:

```
if (!m_server.listen(QHostAddress::Any, DefPort)) {
    QMessageBox::information(this, "Error", "Cannot listen socket");
    return;
}
```

listen işleminde akış bloke olmaz. Aslında soket dinlenmesi işlemi arka planda işletim sistemi tarafından yapılmaktadır. İşletim sistemi bizi ilgilendiren bir bağlantı isteği olduğunda onu bize iletmektedir. Konunun bazı ayrıntıları “Sistem Programlama ve İleri C Uygulamaları 1” kursunda ele alınmaktadır.

3) Artık sıra gelen bağlantı isteklerini kabul etmeye gelmiştir. Bu işlem aşağı seviyeli soket kütüphanelerinde accept fonksiyonuyla yapılmaktadır. Ancak Qt bunun için sinyal-slot mekanizmasını kullanmaktadır. QTcpServer sınıfının newConnection isimli aşağıdaki parametrik yapıya ilişkin bir sinyali vardır:

```
void newConnection();
```

Bizim de connect işlemi ile bu sinyale slot bağlamamız gerekir. Örneğin:

```
QObject::connect(&m_server, SIGNAL(newConnection()), this, SLOT(newConnectionHandler()));
```

Burada görüldüğü gibi bir bağlantı isteği geldiğinde newConnectionHandler isimli fonksiyon çağrılacaktır. İşte bu fonksiyon içerisinde QTcpServer sınıfının nextPendingConnection fonksiyonu ile bağlantı kurularak client ile konuşmakta kullanılacak QTcpSocket nesnesi elde edilir:

```
virtual QTcpSocket *nextPendingConnection();
```

Örneğin:

```
void MainWindow::newConnectionHandler()
{
    QTcpSocket *socket;

    socket = m_server.nextPendingConnection();
    //..
}
```

4) Artık bize verilen QTcpSocket nesnesi ile bağlandığımız client’a bilgi gönderip alabiliriz. Bu konu sonraki bölümde ele alınacaktır.

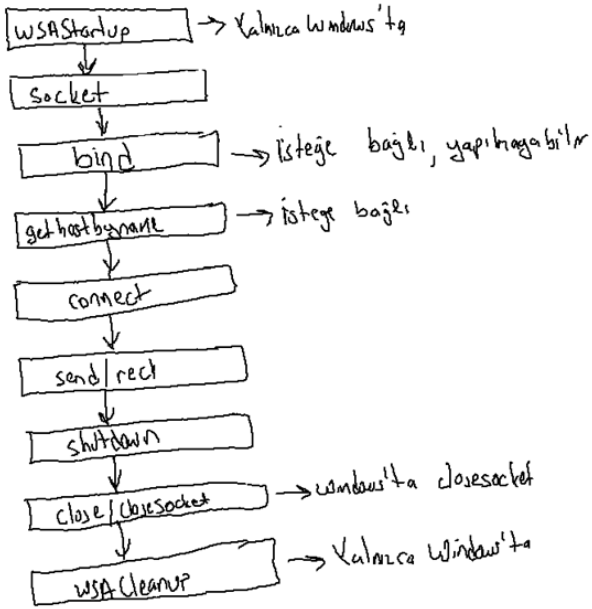
5) İşimiz bittiğinde elde edilen soketi QTcpSocket sınıfının close fonksiyonuyla kapatabiliriz.

Server bağlantıyı sağladıktan sonra QTcpSocket sınıfı içerisinde bağlanan client’ın IP numarasını ve port numarasını alabilir. Eğer client bind işlemi yapmadıysa işletim sistemi boş bir port numarasını oluşturulan sokete atayacaktır. Örneğin client host’un ip adresini ve port numarasını belirterek server’a bağlanmış olsun. Server bu bağlantıdan yeni bir QTcpSocket nesnesi elde edecektir. Bu nesneyi kullanarak client’ın ip adresini ve port numarasını alabilecektir.

QTcpSocket sınıfının localAddress ve localPort fonksiyonları o anda bağlı bulunan yerel host’tai IP adresini ve port numarasını bize verir. Halbuli peerAddress ve peerPort fonksiyonları ise karşı tarafın IP adresini ve port numarasını bize verir.

TCP Client Programın Organizasyonu

TCP client programın API düzeyinde soket kütüphanesi ile organizasyonu aşağıdaki gibidir:



Önce burada da bir soket nesnesi yaratılmış sonra isteğe bağlı olarak bağlanmıştır. Sonra host ismini ip numarasına dönüştürmek için gethostbyname fonksiyonu çağırılmıştır. Sonra bağlantı işlemini yapan connect fonksiyonun çağırıldığını görüyorsunuz. Bundan sonra artık bilgi gönderip alma işlemi benzer biçimde yapılmaktadır.

Qt'de client programının yazımında doğrudan QTcpSocket sınıfı kullanılır. Client programının yazımı sırasıyla şu aşamalardan geçilerek yapılmaktadır:

- 1) Önce QTcpSocket sınıfı türünden nesne yaratılır. Bu nesne yine sınıfın veri elemanı olarak alınabilir.
- 2) Client belli bir yerel portu kullanmak istiyorsa bind işlemi yapabilir. bind işlemi QTcpSocket sınıfının bind üye fonksiyonuyla yapılmaktadır. Eğer client bind işlemi yapmazsa işletim sistemi yerel port numarası olarak boştaki bir port numarasına sokete otomatik olarak atamaktadır.
- 3) Server'a bağlantı için sınıfın connectToHost üye fonksiyonu kullanılır. İki ayrı connectToHost fonksiyonu vardır. Bunlardan biri bağlanılacak host'un IP adresini diğeri ismini almaktadır:

```
void connectToHost(const QString &, quint16 , OpenMode , NetworkLayerProtocol );
void connectToHost(const QHostAddress &, quint16 , OpenMode );
```

- 3) Bağlantı sağlanınca QTcpSocket sınıfının connected isimli sinyali tetiklenmektedir:

```
void QAbstractSocket::connected();
```

Karşılıklı Bilgi Gönderip Alma İşlemi

Yukarıdaki adımlardan geçildikten sonra artık iki tarafın da elinde QTcpSocket nesneleri vardır. Server bu nesneyle client'a bilgi gönderip ondan gelen bilgileri alır, client ise server'a bilgi göndererek ondan gelen bilgileri alır. Bilgi gönderip alma işlemi QTcpSocket sınıfının read ve write fonksiyonlarıyla yapılabilmektedir. read fonksiyonlarının prototipleri şöyledir:

```
quint64 QIODevice::read(char *data, quint64 maxSize);
QByteArray QIODevice::read(quint64 maxSize);
```


Qt’de read işlemleri blokesiz modda yapılmaktadır. Yani o anda sokete hiçbir bilgigelmemişse ve biz read fonksiyonuyla okuma yapmak istemişsek read fonksiyonu 0 değeri ile geri döner. O halde biz sokete bilgi geldiğinde read işlemi yapmalıyız. Pekiyi sokete bilgi geldiğini nereden anlarız? İşte QTcpSocket sınıfının QIODevice taban sınıfından gelen readyRead sinyali bu amaçla kullanılabilir. Bu sinyal sokete bilgi geldiğinde emit edilmektedir. Bu durumda biz okuma işlemi bu sinyal için yazdığımız slot fonksiyonunda yapabiliriz.

Bilgi göndemek için write fonksiyonları kullanılmaktadır. Üç tane write fonksiyonu vardır:

```
qint64 QIODevice::write(const char *data, qint64 maxSize);
qint64 QIODevice::write(const char *data);
qint64 QIODevice::write(const QByteArray &byteArray);
```

Birinci fonksiyon belli adresten başlayarak belli miktarda bilgiyi göndermek için kullanılır. İkinci fonksiyon null karakter görene kadar byte’ları sokete yazar. Üçüncü fonksiyon ise QByteArray içerisindeki tüm byte’ları göndermek için kullanılmaktadır. Fonksiyonlar istenilen kadar byte’ı gönderemeyebilirler. Aslında bu fonksiyon gönderilecek byte’ları network tamponuna bırakıp sonlanmaktadır. Dolayısıyla fonksiyonun geri dönmesi bilginin karşı tarafın eline geçtiği anlamına gelmez. Bilgi network tamponuna bırakılmıştır. Orada paketlenerek (birden fazla paket biçiminde de olabilir) zaman içerisinde karşı tarafa gönderilecektir. write fonksiyonları da blokesiz modda çalışmaktadır. Yani bizim istediğimiz kadar byte’ın tamamı network tamponuna bırakılana kadar fonksiyonlar bizi bekletmez. Eğer network tamponu doluysa write fonksiyonu daha az bilgiyi yazarak hemen geri dönebilir. Ancak bu durum nadir oluşabilecek bir durumdur. Fakat bu programcının önlemini almalıdır.

Soketten bilgi gönderip alma işlemi yukarıda da görüldüğü gibi byte düzeyinde yapılmaktadır. read ve write fonksiyonları bizden char türünden bir adres ya da QByteArray nesnesi istemektedir. İşte birtakım bilgileri byte düzeyine dönüştürüp geri almak için QDataStream sınıfından faydalanılmaktadır. Bu sınıf temel olarak dosya işlemlerinin anlatıldığı bölümde ele alınmıştı. Örneğin:

```
QByteArray qba;
int a = 123, b = 124;
int x, y;

QDataStream qds1(&qba, QIODevice::ReadWrite);
qds1 << a << b;
//...

QDataStream qds2(&qba, QIODevice::ReadOnly);
qds2 >> x >> y;
```

Burada önce qds1 yardımıyla bir QByteArray nesnesine bilgiler yazılmış sonra qds2 yardımıyla oradan onlar geri alınmıştır.

Anahtar Notlar: QtCreator’da bir projenin birden fazla kez çalıştırılabilmesi için “Tools/Options/Build & Run/Stop Applications before building “ None seçeneğine ayarlanmalıdır.

Çok Client’lı Server Programların Yazımı

Pek çok server birden fazla client’ın isteklerini karşılayabilecek biçimde yazılmaktadır. Bunun için bazı tekniklerin kullanılması gerekebilmektedir. Öncelikle çok client’lı programlardaki en önemli sorun server’ın farklı client’larla konuşabilme yeteneğinin organize edilmesidir. Her framework bu organizasyon için kendisine uygun bir yöntem öngörmüştür. Biz burada QT’de kullanılması gereken tipik organizasyon üzerinde duracağız.

Öncelikle çok client’lı uygulamalarda server’ın her farklı client ile bağlandığında onun bilgilerini bir sınıfla temsil edip o türden bir sınıf nesnesinin içerisinde saklaması uygun olur. Sonra bu sınıf nesnelerini bir container içerisinde tutabilir. Örneğin her client’ın bilgisi ClientInfo isimli bir sınıfla temsil edilebilir:

```
class ClientInfo : public QObject {
    //...
```

```
};
```

Sınıfın QObject'ten türetilmesi uygun olur. Çünkü bunun sinyal/slot mekanizmasına dahil edilmesi gerekmektedir. Örneğin:

```
class ClientInfo : public QObject
{
    Q_OBJECT
public:
    şilislaiş

private:
    QTcpSocket *m_socket;
};
```

Server bağlantı kurduğu zaman bir ClientInfo nesnesi yaratıp ona gelen mesajları onun ele almasını sağlayabilir:

```
void MainWindow::newConnectionHandler()
{
    QTcpSocket *socket= m_server.nextPendingConnection();
    ClientInfo *ci = new ClientInfo(socket, this);

    QObject::connect(socket, SIGNAL(readyRead()), ci, SLOT(readyReadHandler()));
    m_clients.append(ci);
}
```

Her bağlanan client için readyRead sinyali ele alınmıştır. Böylece herhangi bir client'a bilgi geldiğinde o ClientInfo nesnesi üzerinden readyRead sinyali emit edilecektir. Slot fonksiyonu içerisinde bilginin hangi client'tan gelmiş olduğu anlaşılabilir. Burada server'ın aynı zamanda bağlanan client'lara ilişkin ClientInfo nesnelerini de bir container sınıfta (m_clients) tuttuğunu görüyorsunuz.

Soket Hatalarının Ele Alınması

Şimdiye kadar biz sokette bir hata oluştuğunda herhangi bir ele alım işlemi uygulamadık. Qt'de soket hataları sinyal/slot mekanizması kullanılarak ele alınmaktadır. Şöyle ki: QTcpSocket sınıflarının error isimli sinyalleri vardır. Ne zaman soket üzerinde bir hata oluşsa bu sinyal tetiklenir. Programcı da bu bu sinyalleri slotlara bağlayarak hatayı ele alır. error isimli sinyalin prototipi şöyledir:

```
void QAbstractSocket::error(QAbstractSocket::SocketError socketError);
```

Çok Client'lı Uygulamalarda Thread Havuzlarının Kullanımı

Qt'de çok çok client'lı server uygulamaları yazarken sinyaller hep aynı thread'e bağlanmış durumdadır. Yani başka bir deyişle aslında kodun tek bir akışı vardır. Bu durumda Örneğin kod bir client için readyRead sinyaline bağlanan slotta geldiğinde burada uzun süre bekleme yapılırsa diğer client'lara gelen bilgileri biz işleyemeyiz. Pekiyi bunun çözümü ne olabilir? Bunun en makul çözümü readyRead sinyaline bağlanan slotta client'tan gelen mesajları ayrı bir thread yaratarak o thread'e işletmektir. Tabii böyle bir thread'in ömrü kısa olacağı için onun yaratılıp yok edilmesi zaman kaybına yol açabilmektedir. İşte bu tür thread'lerin thread havuzları yoluyla yaratılması performansı artırmaktadır. Diğer bir yöntem sonraki başlıkta ele alınacak olan "her client için ayrı bir thread'in işin başında yaratılması" yöntemidir". Ancak bu yöntem Qt'de oldukça karmaşık bir organizasyona yol açmaktadır. Bu nedenle multiclient server uygulamalarında thread havuzlarının kullanılması en iyi çözümdür ("075-MultiClientServerWithThreadPool").

Qt'de thread havuzu için QThreadPool sınıfı kullanılmaktadır. QThreadPool sınıfı türünden global bir tane nesne yaratılmış ve hazır biçimde bekletilmektedir (Singleton). Bu nesnenin adresi QThreadPool::globalInstance() static fonksiyonuyla elde edilebilir. Sonra elde edilen bu nesne ile QThreadPool sınıfının start fonksiyonu çağrılır. start fonksiyonunun parametrik yapısı şöyledir:

```
void QThreadPool::start(QRunnable *runnable, int priority = 0);
```

start fonksiyonu parametre olarak bizden QRunnable türünden bir nesne adresi istemektedir. Fonksiyon bu sınıfın sanal run fonksiyonuna farklı bir thread akışında çağırır. Yani bizim QRunnable sınıfından bir sınıf türetilip run fonksiyonunu override etmemiz gerekmektedir.

Çok Client'lı Server Uygulamalarında Her Client İçin Ayır Thread Kullanımı

Çok client'lı server uygulamalarında diğer bir yöntem de her client için işin başında bir thread oluşturmaktır. Ancak client sayısı aşırı büyürse çok fazla sayıda thread sistem performansını düşürebilir. Biz burada önce bu yöntem üzerinde duracağız. Ancak bu yöntem Qt için oldukça karmaşık bir organizasyona yol açabilmektedir. Bu nedenle uygulamalarda bu yöntem yerine yukarıdaki "Çok Client'lı Uygulamalarda Thread Havuzlarının Kullanımı" başlığında açıklanan yöntem kullanılmalıdır.

Anımsanacağı gibi Qt'de QObject sınıfından türetilen nesnelerin thread sahipliği vardır. Yani bir QObject nesnesine hangi thread sahipse o nesne yalnızca o thread'te kullanılmalıdır. İşte maalesef server bir client ile bağlantı sağladığında elde edilen QTcpSocket nesnesinin sahipliği ana thread'te olur. Başka bir deyişle biz bu nesneyi yeni yaratacağımız client'ta kullanamayız.

Qt'de her client'ın ayrı bir thread'e sahip olması yöntemi şu adımlardan geçilerek gerçekleştirilmektedir:

1) Server soket nesnesi için QTcpServer sınıfı doğrudan kullanılmaz. Bu sınıftan türetme yapılarak türetilmiş sınıf kullanılır.

2) Türetien bu sınıfta QTcpServer sınıfındaki incomingConnection isimli sanal fonksiyon override edilir. Ne zaman bir bağlantı oluşsa aslında bu sanal fonksiyon da çağrılmaktadır. Dolayısıyla bizim artık newConnection isimli sinyali ele almamız gerekmemektedir.

```
void QTcpServer::incomingConnection(qintptr socketDescriptor);
```

Aslında QTcpSocket sınıfı kendi içerisinde işletim sistemi düzeyinde kullanılan soket handle değeri ile soket işlemlerini yapmaktadır. Bu sanal fonksiyon bize QTcpSocket nesnesini değil bu handle değerini vermektedir. Bu handle değeri biliniyorsa QTcpSocket nesnesi zaten oluşturulabilmektedir.

3) Artık yeni bir thread sınıfı QThread sınıfından türetilip incomingConnection fonksiyonundan alınan değer bu thread sınıfına başlangıç fonksiyonu parametresi olarak geçirilmelidir. Bu thread içerisinde de QTcpSocket nesnesi bu soket betimleyicisi ile yaratılabilir.

Yukarıdaki adımlar "076-MultiThreadClientServer" örnek kodunda uygulanmıştır. Bu örnek kodun Server tarafının açıklaması şöyle yapılabilir:

- Öncelikle QTcpServer sınıfından bir sınıf türetilip o sınıfta incomingConnection fonksiyonu override edilmiştir. Bunun nedeni bağlantı yapıldığında bağlanılan client'a ilişkin soket betimleyicisinin (socket descriptor) elde edilmek istenmesidir. Örneğimizde QTcpServer sınıfından türetilen sınıf MyServer isimli sınıftır.

- MainWindow sınıfı içerisinde MyServer sınıfı türünden nesne kullanılarak listen işlemi yapılmıştır. Böylece yeni bir client ile bağlantı sağlandığında MyServer sınıfının incomingConnection fonksiyonu çağrılacaktır. Bu fonksiyonun tanımlaması aşağıdaki gibi yapılmıştır:

```
void MyServer::incomingConnection(qintptr socketDescriptor)
{
    MainWindow *mw = static_cast<MainWindow *>(parent());
    QTcpSocket *socket = new QTcpSocket();
    socket->setSocketDescriptor(socketDescriptor);

    QString str = QString("%1, %2, %3, %4").arg(socket->localAddress().toString()).
        arg(socket->localPort()).arg(socket->peerAddress().toString()).arg(socket->peerPort());
    mw->addClientInfo(str);
    delete socket;
```

```

ClientInfo *ci = new ClientInfo(socketDescriptor, this);
QObject::connect(ci, SIGNAL(putMsg(QString)), mw, SLOT(putMsg(QString)));
ci->start();
}

```

Burada görüldüğü gibi bir ClientInfo nesnesi yaratılmış ve socketin betimleyicisi o nesneye verilmiştir. ClientInfo aslında bir thread nesnesidir. QThread sınıfından türetilmiştir. Fonksiyonun başındaki kodlar yeni bağlanan client'ın bilgileri MainWindows'daki QListWidget nesnesine eklenmesi ile ilgilidir.

Client-Server Haberleşme Sistemi

Yukarıda verdiğimiz örneklerde client ile server arasında yalnızca bir yazı gönderilmektedir. Ancak uygulama büyüdükçe client ile server arasında birtakım konuşmalar (yani mesajlaşmalar) gerçekleşmektedir. Bu konuşmalar bir protokolü akla getirir. Aslında bizim yazdığımız uygulamalardaki client ile server arasındaki mesajlaşmalar da bir çeşit uygulama düzeyinde protokol olarak değerlendirilebilir. Gerçekten de "telnet" gibi "ssh" gibi, "ftp" gibi protokoller de benzer biçimde tasarlanmışlardır.

Client ile server arasındaki mesajlaşmalar metin (text) tabanlı ya da binary tabanlı olabilir. Genel olarak binary mesajlaşmalar daha etkin olsa da metin tabanlı mesajlaşmalar daha pratiktir. IP ailesinin uygulama katmanındaki pek çok protokol metin tabanlı bir haberleşme kullanır.

Mesaj tabanlı haberleşmede client ve server isteklerini ve bu isteklerin sonuçlarını yazısal biçimde (herhangi bir encoding kullanılabilir) gönderip almaktadır. Yazının sonunda null karakter bulundurulabilir ya da New Line karakteri ('\n') bulundurulabilir. Metin tabanlı mesajlaşmada bir mesaj aşağıdaki gibi bir formata sahip olabilir:

"<mesaj türü> <argümanlar>"

Örneğin:

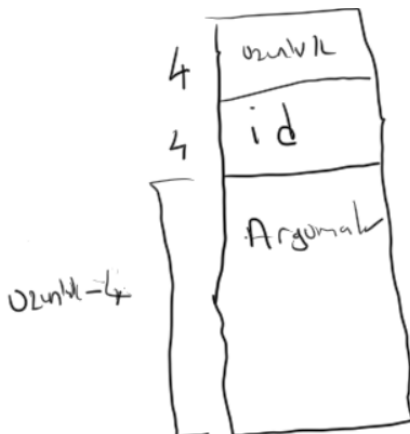
"LOGIN kaan"

Burada "LOGIN" mesajın türüdür. Onun argümanı da "nickname"dir. Örneğin:

"DIRLIST c:\windows"

Burada mesajın türü "DIRLIST" biçimindedir. Bu mesajla client server'daki bir dizinin içeriğini istiyor olabilir. mesajın argümanı da hangi dizinin içeriğinin istendiğidir.

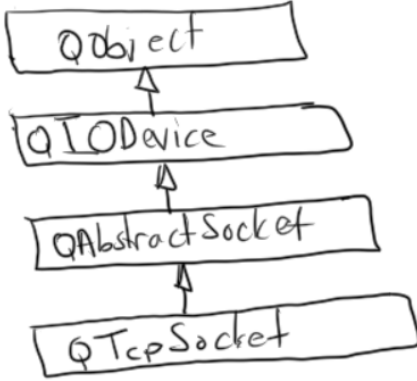
Binary haberleşmede her şey byte'lar biçiminde gönderilip alınır. Bu durumda mesajın ne mesajı olduğu onlara numara verilerek belirlenmektedir. Tabii mesajların başlarında mesajın toplam uzunluğunun kaç byte olduğunu belirten bir alanın bulunması gerekir. Örneğin binary mesajlaşmada bir mesaj aşağıdaki formata sahip olabilir:



Bu durumda mesajı alacak kişi önce soketten 4 byte'lık uzunluk bilgisini okur. Bundan sonra yeniden soketten bu sefer uzunluk kadar byte okur. Hepsini okuduktan sonra işleme sokar. Tabii mesajın argümanlar kısmının nelerden oluştuğu mesajın ne mesajı olduğuna (yani id'sine) bağlıdır.

Peki Qt'de soketten taaamolarak istediğimiz kadar byte'ı nasıl okuruz? Anımsanacağı gibi biz read fonksiyonunu çağırdığımızda read bizim talep ettiğimiz miktarı okuyana kadar beklememektedir. read o anda hazır olan byte sayısı kadar en fazla bize okuma yapmaktadır. Bu nedenle readyRead sinyalinde örneğin biz read fonksiyonu ile soketten 1000 byte okumak istesek 1000 byte'ın tamamını değil o anda sokete gelen byte miktarı kadar okuma yaparız. Bu miktar 1000'den az ise bizim bu işleme devam etmemiz gerekir.

Qt'de soket sınıfı olarak kullandığımız QTcpSocket sınıfının türetme şeması şöyledir:



Aslında read ve write fonksiyonları QIODevice sınıfından gelmektedir. QIODevice sınıfının byttdAvalibale fonksiyonu o anda sokette okunabilecek kaçbyte'lık bir bilgi bulunduğunu bize verir. Benzer biçimde canReadLine fonksiyonu da o anda sokette en az bir satırlık bilginin olup olmadığını bize vermektedir. Biz bu sayede read fonksiyonuyla ya da readLine fonksiyonuyla istediğimiz kadar bilgiyi soketten tam olarak okuyabiliriz.

Çok Clientlı Örnek Bir Chat Programı

Örnek chat programımızda protokol özet olarak aşağıdaki gibidir:

1) Client önce fiziksel olarak TCp ile server'a bağlanır. Sonra server'a mantıksal bağlanma için LOGIN mesajını gönderir. LOGIN mesajının formatı şöyledir:

"LOGIN <takma ad>"

2) Client ve server her mesaj sonra karşı tarafa onay anlamında bir mesaj gönderir. Eğer alınan mesaj başarılıysa "OK", "değilse de hata mesajının kendisini göndermektedir.

3) Server bağlanan tüm client'ları bir listede tutar. Yeni bir client bağlandığında server diğer tüm client'lara "USERLOGGEDIN" mesajı göndermektedir. Bu mesajın formatı şöyle olabilir:

"USERLOGGEDIN <takma ad>"

4) Bir client sistemden çıkmak istediğinde "LOGOUT" mesajını server'a gönderir. Diğer bütün client'lara da "USERLOGGEDOUT" mesajını göndermektedir. Bu mesajın formatı şöyle olabilir:

"USERLOGGEDOUT <takma ad>"

5) Client ortaya bir mesaj göndermek için server'a "DISTRIBUTE MSG" mesajını yollar. Mesajın formatı şöyledir:

"DISTRIBUTE MSG <mesaj metni>"

6) Server client'lara mesajı dağıtmak için "NEWMESAGE" mesajını yollar. Bunun da formatı şöyledir:

"NEWMESAGE <mesaj metni>"

Qt'de UDP Uygulamaları

Daha önceden de belirtildiği gibi UDP paket tabanlı bir protokoldür. UDP bağlantılı bir protokol değildir. Dolayısıyla bir connect işlemi gerekmez. O halde UDP uygulamalarında belirgin bir client-server mimarisi yoktur. Ancak genellikle mesajları alan tarafa server, gönderen tarafa client denilmektedir. Yani bir bağlantı olmasa da mesajları işleyen roldeki program server olarak isimlendirilebilir. UDP stream tabanlı olmadığı için biz gelen paketi parça parça okuyamayız. Onu tek hamlede paket olarak alırız. Gönderen taraf karşı tarafın paketi alıp almadığını bilemez. Her iki taraf da karşı tarafın aktif olup olmadığını ancak gönderim ve alım sırasında anlayabilir. Daha önceden de belirtildiği gibi periyodik fakat çok önemli olmayan bildirimler UDP ile aktarılmaktadır. (Örneğin bir makina çalıştığını saniyede bir UDP paketi göndererek server'a iletebilir. Bir oyun programı arabanın durumunu UDP paketi ile dış dünyaya iletebilir. Birtakım görüntüler UDP ile iletebilirler.

Qt'de UDP server uygulaması şu adımlardan geçilerek yazılır:

- 1) Server QUdpSocket sınıfı türünden bir nesne yaratır.
- 2) Server QUdpSocket sınıfının bind fonksiyonunu çağırarak soketi bağlar. Bu sırada hangi network kartından ve hangi porttan gelen paketleri alacağını belirtir.
- 3) readyRead sinyali yeni bir datagram geldiğinde tetiklenir. Server bu mesajı işleyerek gelen paketi QUdpSocket sınıfının readDatagram fonksiyonuyla okur.
- 4) Server isterse writeDatagram fonksiyonuyla karşı tarafa da paket gönderebilir.
- 5) İşlem sonucunda server close fonksiyonuyla soketi kapatır.

UDP client programın yazımı da şöyle yapılır:

- 1) QUdpSocket türünden bir nesne yaratılır.
- 2) bind işlemi yapılmak zorunda değildir. Göndeme için writeDatagram fonksiyonu kullanılır. writeDatagram fonksiyonunda gönderilecek host'un ip adresi ve port numarası belirtilmektedir.

Qt'de Event Mekanizması

Biz şimdiye kadar birtakım olaylara tepki vermek için sinyal-slot mekanizmasını kullandık. Bu mekanizmada bir olay olduğunda bir sinyal tetiklenmektedir (emit edilmektedir) bu sayede o sinyale bağlı slot fonksiyonları çağrılır. Örneğin bir QPushButton nesnesinin tıklandığında bu nesne clicked isimli sinyali tetikler (emit eder), böylece bu sinyale bağladığımız slot fonksiyonları çağrılır. Qt'de bize sunulmuş sinyallerle biz pek çok aşağı seviyeli işlemleri yapamamaktayız. Çünkü sinyal-slot mekanizması aşağı seviyeli bir event mekanizmasının üzerine kurulmuştur.



Peki ya acaba QPushButton sınıfını yazanlar kendi pencerelerinin üzerine tıkladığını nasıl anlayıp clicked sinyalinin tetiklerler? İşte farelin izlenmesi gibi, klavyenin izlenmesi gibi aşağı seviyeli mesaj işlemleri sinyal-slot mekanizmasıyla değil event mekanizmasıyla işletilmektedir.

Event kavramı QEvent isimli bir sınıfla temsil edilmiştir. Bu sınıftan çeşitli event sınıfları türetilmiştir. Event’lerin çoğu aşağı seviyeli GUI mesajları ilişkilidir. Bu nedenle bunların çoğu QWidget sınıfında bulunmaktadır. Event işlemesi sanal fonksiyonlar yoluyla yapılır. Yani tipik olarak işletim sisteminin GUI alt sistemi mesajı oluşturur. Bu mesajı Qt Framework elde eder. Sonra mesajın ilişkin olduğu nesneyi (büyük ölçüde QWidget nesnesini) belirler ve bu nesneyle o sınıfın ilgili sanal fonksiyonunu çağırır. Biz de ilgili event’i işlemek için o sınıftan türetme yapıp o sanal fonksiyonu override ederiz. Qt’de okunabilirlik için bu sanal fonksiyonlar xxxEvent biçiminde isimlendirilmiştir. Event fonksiyonları ilgili sınıfların protected bölümlerinde bulunmaktadır. Örneğin QWidget sınıfında bulunan tipik event’ler şunlardır:

- closeEvent : Pencere yok edilmeden önce oluşturulur.
- keyPressEvent ve keyReleaseEvent: Klavyeden bir tuşa basıldığında ve el klavyedeki tuştan çekildiğinde oluşturulur.
- mouseMoveEvent: Fare hareket ettirildiğinde oluşur. Bu mesaj fare hareket ettirilmiyorsa oluşturulmaz. Hareket sırasında kaç defa oluşturulacağı belli değildir. Sistemin genel yoğunluğuna bağlıdır.
- mousePressEvent ve mouseReleaseEvent: Farenin herhangi bir tuşuna basıldığında ve el çekildiğinde oluşturulur.
- paintEvent: Pencere içerisindeki görüntü bozulduğunda bunun yeniden çizilmesi için oluşturulur. Programcı çizimleri burada yapmalıdır.

Event fonksiyonlarının parametreleri oluşan olay hakkında bize bazı bilgileri vermektedir. Örneğin fare ilişkin event fonksiyonlarının parametresi QMouseEvent sınıfı türündendir. Bu sınıfın elemanları bize farenin konumu, hangi tuşuna basılmış olduğu gibi bilgileri vermektedir.

Qt’de Çizim İşlemleri

Windows gibi bazı sistemler pencere içerisindeki görüntüyü bizim için tutmamaktadır. Pencere içerisindeki görüntünün oluşturulması Windows sistemlerinde programcıya bırakılmıştır. Programcı pencere içerisine bir çizim yaptığında o pencerenin üzerine bir pencere getirilip tekrar açıldığında o çizim kaybolur. Windows bunu tutarak geri basmamaktadır. Bunun yerine WM_PAINT isimli mesajı thread’in mesaj kuyruğuna bırakır. Bu mesaj pencere içerisindeki görüntünün bozulduğu ve yeniden çizilmesi gerektiği anlamına gelmektedir. Dolayısıyla bu sistemlerde çizimler bu mesaj geldiğinde yapılmalıdır. Diğer işletim sistemlerini bazıları pencere içerisindeki çizimleri kendisi tutup basabilmektedir. Ancak Qt’nin “cross platform” özelliği ortak bir bölene göre tasarlanmıştır. Qt’de pencere içerisindeki çizim bozulduğunda framework tarafından QWidget sınıfının paintEvent isimli sanal fonksiyonu çağrılır. O halde çizimler bu fonksiyon içerisinde yapılmalıdır.

Çizim işlemlerinde QPainter isimli sınıf kullanılmaktadır. Bu sınıf sınıf türünden nesne çizimin yapılacağı QWidget nesnesinin adresi verilerek yaratılır.

```
void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    //...
}
```

QPainter sınıfının çizim işlemlerini yapan pek çok drawXXX fonksiyonu vardır. drawXXX fonksiyonları çizimi yaparken bazı çizim nesnelerini kullanmaktadır. İki önemli çizim nesnesi vardır: Kalem (pen) ve fırça (brush). Kalem nesnesi QPen sınıfı ile fırça nesnesi de QBrush sınıfı ile temsil edilmiştir. Bir kalem ve fırça nesnesi yaratılıp QPainter sınıfının setPen ve setBrush fonksiyonlarıyla kullanıma hazır hale getirilebilir. Çizim işlemlerinde orijin noktası çalışma sol-üst köşesidir. X değeri sağa doğru Y değeri aşağıya doğru artar.

QPainter sınıfının drawLine isimli fonksiyonları bizden iki nokta olarak bir doğru çizer. drawLines isimli fonksiyonlar ise bizden QVector<Point> ya da QPoint ya da QLine dizisi olarak her iki noktadan bir doğru oluşturup bunların çizimini yapar. Örneğin:

```
void MainWindow::paintEvent(QPaintEvent *event)
```

```

{
    QPainter painter(this);

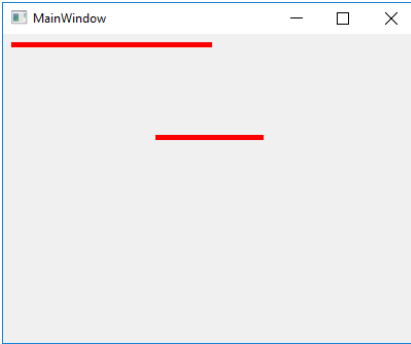
    QPen pen(Qt::red);
    pen.setWidth(5);

    painter.setPen(pen);

    QVector<QPoint> points;

    points << QPoint(10, 10) << QPoint(200, 10) << QPoint(150, 100) << QPoint(250, 100);
    painter.drawLines(points);
}

```



Örneğin:

```

void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    QPen pen(Qt::red);
    pen.setWidth(5);

    painter.setPen(pen);

    QLine lines[] = {QLine(10, 10, 200, 10), QLine(150, 100, 250, 100)};

    painter.drawLines(lines, 2);
}

```

drawPolyLine fonksiyonları bizden bir QPoint dizisi ve onun uzunluğunu alarak oradaki noktaları doğrularla birleştirmektedir. Örneğin:

```

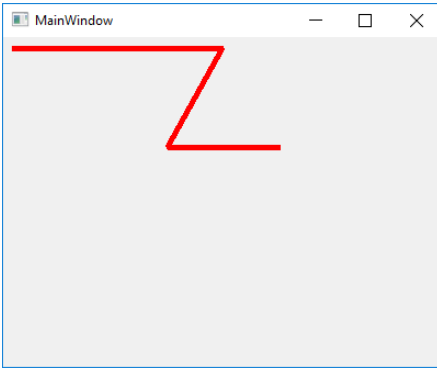
void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    QPen pen(Qt::red);
    pen.setWidth(5);

    painter.setPen(pen);

    QPoint points[] = {QPoint(10, 10), QPoint(200, 10), QPoint(150, 100), QPoint(250, 100)};
    painter.drawPolyline(points, 4);
}

```

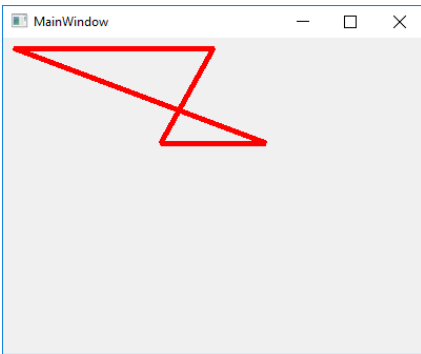
drawPolygon fonksiyonları drawPolyline gibidir. Ancak son nokta ilk noktayla birleştirilir. Örneğin:

```
void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    QPen pen(Qt::red);
    pen.setWidth(5);

    painter.setPen(pen);

    QPoint points[] = {QPoint(10, 10), QPoint(200, 10), QPoint(150, 100), QPoint(250, 100)};
    painter.drawPolygon(points, 4);
}
```

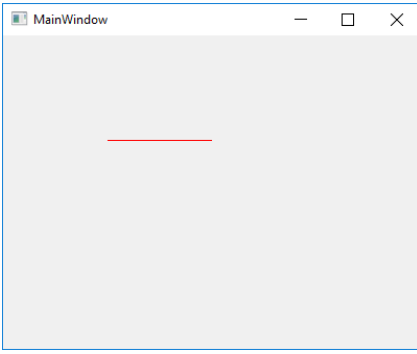


drawPoint fonksiyonları tek bir nokta(pixel) basmak için, drawPoints fonksiyonları ise birden fazla nokta basmak için kullanılır. Örneğin:

```
void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    QPen pen(Qt::red);
    painter.setPen(pen);

    for (int i = 0; i < 100; ++i)
        painter.drawPoint(100 + i, 100);
}
```



Çizilen nokta kalemin genişliğine de bağlıdır.

Örneğin bir sinüs eğrisi şöyle çizilebilir:

```
void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

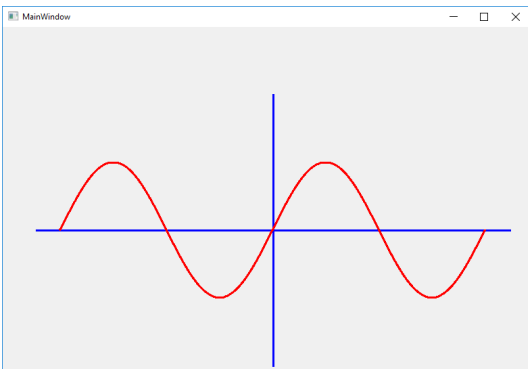
    QPen pen;

    pen.setColor(Qt::blue);
    pen.setWidth(3);
    painter.setPen(pen);

    double x, y;
    QVector<QPoint> points;
    int xOrg = 400, yOrg = 300, xUnit = 50, yUnit = 100;

    painter.drawLine(xOrg, yOrg - 200, xOrg, yOrg + 200);
    painter.drawLine(xOrg - 350, yOrg, xOrg + 350, yOrg);

    pen.setColor(Qt::red);
    painter.setPen(pen);
    for (x = -3.14 * 2; x < +3.14 * 2; x += 0.01) {
        y = sin(x);
        points.append(QPoint(xOrg + x * xUnit, yOrg - y * yUnit));
    }
    painter.drawPolyline(QPolygon(points));
}
```



QPainter sınıfının drawEllipse isimli fonksiyonları bizden bir dikdörtgen ister ve o dikdörtgen için iç teğet elips çizer. Örneğin:

```
void MainWindow::paintEvent(QPaintEvent *event)
{
```

```

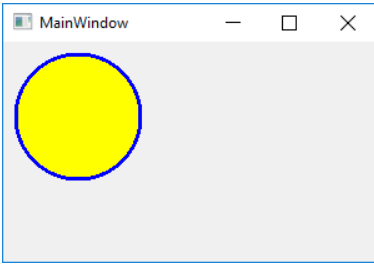
QPainter painter(this);

QPen pen;
pen.setColor(Qt::blue);
pen.setWidth(3);
painter.setPen(pen);

QBrush brush(Qt::yellow);
painter.setBrush(brush);

painter.drawEllipse(10, 10, 100, 100);
}

```



QPainter sınıfının drawPie isimli fonksiyonları elips dilimi çizmek için kullanılır. Bu fonksiyonlar bizden bir dikdörtgen ve iki de açı isterler. Açılardan 1/ 16 derece cinsindendir ve biri başlangıç açısını diğer de süpürme açısını belirtir.(saat yönünün tersi pozitif). Örneğin:

```

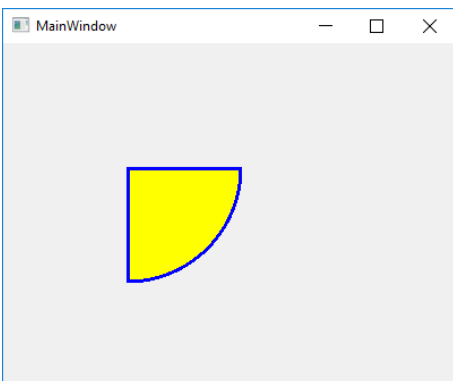
void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    QPen pen;
    pen.setColor(Qt::blue);
    pen.setWidth(3);
    painter.setPen(pen);

    QBrush brush(Qt::yellow);
    painter.setBrush(brush);

    painter.drawPie(QRect(10, 10, 200, 200), 0, -90 * 16);
}

```



drawArc isimli fonksiyonlar ise yalnızca yayı çizerler. Örneğin:

```

void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    QPen pen;

```

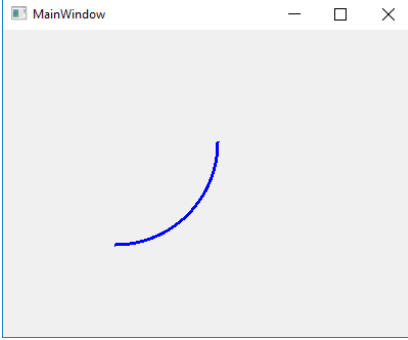
```

pen.setColor(Qt::blue);
pen.setWidth(3);
painter.setPen(pen);

QBrush brush(Qt::yellow);
painter.setBrush(brush);

painter.drawArc(QRect(10, 10, 200, 200), 0, -90 * 16);
}

```



Resimleri çizdirmek için üç fonksiyon grubu vardır: drawPixmap, drawPicture ve drawImage fonksiyonları. Resimleri temsil etmek Qt'de QImage ve QPixmap sınıfları vardır. Aslında her iki sınıf da benzer işlevselliğe sahiptir. Eğer elimizde bir QImage varsa biz onu drawImage fonksiyonla çizdirebiliriz. Eğer elimizde bir QPixmap nesnesi varsa onu da biz drawPixmap ile çizdirebiliriz. QImage sınıfı bize daha etkin olarak bitmap içerisindeki çizimleri değiştirme olanağı vermektedir. Ancak QPixmap de çizim işlemleri için idealdir. Birkaç parametrik yapıya sahip drawPixmap ve drawImage fonksiyonları vardır. Burada drawPixmap üzerinden parametrik yapıları açıklayalım. Aşağıdaki fonksiyon hiç büyültme küçültme yapmadan sol üst köşe koordinatını vererek resmi çizer:

```
void drawPixmap(int x, int y, const QPixmap & pixmap);
```

Bir resmi boyutlandırarak da çizebiliriz. Bunun için drawPixmap fonksiyonun aşağıdaki overload'ı kullanılabilir:

```
void drawPixmap(const QRect &rectangle, const QPixmap &pixmap);
void drawPixmap(int x, int y, int width, int height, const QPixmap &pixmap);
```

Fonksiyon bizden hedef dikdörtgenin koordinatlarını ve çizilecek resmi ister. Bir resmin belli bir dikdörtgensel alanı da boyutlandırılarak çizdirilebilir:

```
void drawPixmap(const QRect &target, const QPixmap &pixmap, const QRect &source);
```

Fare Yardımıyla Çizim Yapılması

Qt'de tüm çizimlerin paintEvent fonksiyonunda yapılması gerekmektedir. Gerçekten de QPainter nesnesi ile zaten pencereye dışarıda (yani paintEvent fonksiyonun dışında) çizim yapılamaz. O halde bizim fare mesajlarında uygun verilere oluşturup çizimi o verilere göre paintEvent fonksiyonunda yapmamız gerekir. paintEvent isimli event'i oluşturmak için QWidget sınıfından gelen update ve repaint fonksiyonları vardır. Bu iki fonksiyon arasında küçük bazı farklılıklar bulunmaktadır.

Örneğin bir yazboz tahtası (scratchpad) uygulaması şöyle yapılabilir:

```

/* Scratchpadwidget.hpp */

#ifndef SCRATCHPADWIDGET_HPP
#define SCRATCHPADWIDGET_HPP

#include <QtGui>
#include <QVector>

```

```

#include <QWidget>

class ScratchpadWidget : public QWidget
{
    Q_OBJECT
public:
    explicit ScratchpadWidget(QWidget *parent = 0);

protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);

signals:

public slots:

private:
    bool m_drawFlag;
    QPolygon m_line;
    QVector<QPolygon> m_lines;
};

#endif // SCRATCHPADWIDGET_HPP

/* scratchpadwizard.cpp */

#include "scratchpadwidget.hpp"

ScratchpadWidget::ScratchpadWidget(QWidget *parent) : QWidget(parent)
{
    m_drawFlag = false;
}

void ScratchpadWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.setRenderHints(QPainter::Antialiasing | QPainter::SmoothPixmapTransform);

    QPen pen(Qt::black);
    pen.setWidth(3);
    painter.setPen(pen);

    for (QPolygon line : m_lines)
        painter.drawPolyline(line);
    painter.drawPolyline(m_line);
}

void ScratchpadWidget::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        m_line.clear();
        m_line.append(event->pos());
        m_drawFlag = true;
    }
}

void ScratchpadWidget::mouseReleaseEvent(QMouseEvent *event)
{
    if (m_drawFlag) {
        m_lines.append(m_line);
        m_drawFlag = false;
    }
}

void ScratchpadWidget::mouseMoveEvent(QMouseEvent *event)

```

```
{
    if (m_drawFlag) {
        m_line.append(event->pos());
        update();
    }
}
```

Qt’de Bağlam Menüsünün Oluşturulması

Fare ile sağ tuşa basıldığında çıkan menüye bağlam menüsü (context menu) denilmektedir. Qt’de bağlam menüsü şöyle oluşturulur:

- 1) Önce QWidget sınıfının setContextMenuPolicy fonksiyonu Qt::CustomContextMenu argümanı ile çağrılarak bağlam menüsünün genel özelliği belirlenir.
- 2) QWidget sınıfından gelen customContextMenuRequested isimli sinyalin bir slotla bağlanması gerekir.
- 3) QMenu sınıfı türünden bir nesne yaratılıp menü elemanları addAction fonksiyonlarıyla bu menüye eklenir.
- 4) customContextMenuRequested sinyali farenin sağ tuşuna basıldığında tetiklenecektir. Bu sinyale bağlanan slot fonksiyonunda menü nesnesi ile popup fonksiyonu çağrılır. Ancak bu fonksiyon bağlam menüsünün koordinatlarını masaüstü orijinli olarak alır. Bu nedenle çalışma alanı orijinli noktayı masaüstü orijinli hale dönüştürmek için mapToGlobal fonksiyonundan faydalanılabilir.
- 5) Menüler için oluşturulan QAction nesnelerinin triggered sinyali işlenerek menü elemanları seçildiğinde uygun fonksiyonun çağrılması sağlanır.

QPainterPath Kullanımı

Karmaşık bazı çizimler tek bir nesne olarak tanımlanıp yapılabilirler. Genel olarak bu özelliğe “graphics path” denilmektedir. Qt’de “graphics path” işlemleri QPainterPath sınıfıyla temsil edilir. QPainterPath sınıfı şöyle kullanılır:

- 1) QPainterPath sınıfı türünden bir nesne yaratılır.
- 2) Sınıfın addXXX üye fonksiyonları ile sınıfa çeşitli geometrik şekiller eklenir.
- 3) Sonra QPainter sınıfının drawPath fonksiyonuyla belirlenen şekiller tek hamlede çizdirilebilir, fillPath fonksiyonuyla da bunların içi boyanabilir.
- 4) Şekiller baştan 0 orijinli olarak oluşturulabilir. Sonra onlar translate fonksiyonuyla istenildiği gibi ötelenebilirler.

Pencere İçerisine Yazıların Yazdırılması

Pencere içerisine yazı yazdırmak için DrawText fonksiyonları kullanılır. Birkaç overload edilmiş drawText fonksiyonu vardır:

```
void drawText(const QPoint &position, const QString &text);
void drawText(int x, int y, const QString &text);
```

Bu fonksiyon yazıyı yazının sol alt köşesi belirtilen koordinatta olacak biçimde yazdırır. Aşağıdaki fonksiyon yazıyı sarmalama da yaparak bir dikdörtgenin içerisine yazdırır.

```
void drawText(const QRectF &rectangle, const QString &text, const QTextOption &option = QTextOption());
```

Aşağıdaki fonksiyonda aynı zamanda hizalama da yapılabilir:

```
void drawText(const QRect &rectangle, int flags, const QString &text, QRect *boundingRect = Q_NULLPTR);
```

Tabii yazılar değişik fontlarla yazdırılabilirler. Bunun için Font sınıfı türünden bir nesne yaratıp bunu setFont fonksiyonuyla set etmek gerekir. Örneğin:

GraphicsView, GraphicsScene ve GraphicsItem Nesnelerinin Kullanımı

Qt'de biraz daha yüksek seviyeli çizim sınıfları vardır. Bu sınıfların temel kullanımı şöyledir:

1) Önce bir QGraphicsView türünden nesne yaratılarak ana pencereye yerleştirilir. Bu nesne Qt-Designer'da oluşturulabilmektedir.

2) Bundan sonra bir QGraphicsScene nesnesi yaratılır ve bu nesnenin içerisinde bu sınıfın addXXX fonksiyonlarıyla şekiller eklenir. Sonra da QGraphicsView sınıfının setScene fonksiyonu ile "scene" nesnesi "view" nesnesine bağlanır.

Default durumda orijin noktası "scene" ile belirtilen alanın tam ortasındadır (Böylece negatif koordinatlar da geçerlidir.)

Aslında QGraphicsScene nesnesine eklenen öğeler QGraphicsXXXItem nesneleridir. Aslında QGraphicsScene sınıfının addXX fonksiyonları bu nesneleri oluşturup onları eklemektedir. QGraphicsXXXItem sınıfları QGraphicsItem isimli bir abstract sınıftan türetilmiştir. Biz de doğrudan bu sınıf türünden nesneler yaratarak QGraphicsScene sınıfının addItem fonksiyonuyla eklemeyi yapabiliriz.

QGraphicsXXXItem sınıflarının pek çok faydalı elemanı vardır.

Chart Nesnelerinin Kullanımı

Qt'de birtakım grafiklerin kolay çizilmesi için QGraphicsView ve QGraphicsScene sınıfları temelinde bazı chart nesneleri bulundurulmuştur. Chart işlemleri için QChartView isimli view sınıfı kullanılmaktadır. Bu sınıf QGraphicsView sınıfından türetilmiştir. Bu sınıfa bir QChart nesnesi iliştilir. QChart nesneleri de

Qt Quick ve QML ile Programlama

Qt'de Quick Qt'nin GUI arayüzü oluşturmak için yeni bir framework'tür. Bu framework işlev bakımından .NET'in WPF'ine ya da Java FX'e benzetilebilir. Qt Quick ortamında GUI arayüzü QML diye isimlendirilen deklaratif bir dilde yazısal olarak oluşturulmaktadır. Zaten yeni eğilim artık kullanıcı arayüzlerinin projenin parçalarından ayrı olarak metinsel biçimde oluşturulmasına yöneliktir. Pek çok ortam bu paradigmaya kaymaya başlamıştır. Qt Quick ve QML özellikle mobil cihazları hedef alarak geliştirilmiştir. Ancak bu ortamda masaüstü uygulamalar da yapılabilir. Qt Quick ve Qml resmi olarak Qt'ye 4.7 versiyonuyla girmiştir. Ancak 5'li versiyonlarla programcılar tarafından tatnır hale gelmiştir.

QML (Qt Meta Language) kullanıcı arayüzü oluşturmak için kullanılan deklaratif bir dildir. JSON ve JS dillerine sentaks olarak oldukça benzemektedir. Kullanıcı arayüzü QML'de oluşturur, sonra bir sınıf yardımıyla bu QML kodları arayüze dönüştürülür.

Temel QML Sentaksı

Bir QML dosyası iç içe olabilen elemanlardan oluşmaktadır. Elemanların birer isimleri vardır. Bunların içerik tanımlamaları küme parantezi içerisinde yapılır. Eleman tanımlamanın genel biçimi şöyledir:

```
<Eleman İsmi> {  
    [ Tanımlamalar ]  
}
```

Eleman tanımlası çeşitli öğelerden oluşmaktadır. Bunlardan biri property tanımlamalarıdır. Bir property ilgili görsel elemanın bir özelliğini belirtir. Property aşağıdaki gibi set edilmektedir:

<property ismi> : <değer>

Normal olarak her satıra bir property tanımlaması yazılmaktadır. Ancak aynı satırda birden fazla property tanımlaması yapılırsa bunlar ‘;’ ile ayrılmak zorundadır. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 1000 ; height: 600
    title: qsTr("Merhaba Qt Quick")

    MainForm {
        anchors.fill: parent
        mouseArea.onClicked: {
            Qt.quit();
        }
    }
}
```

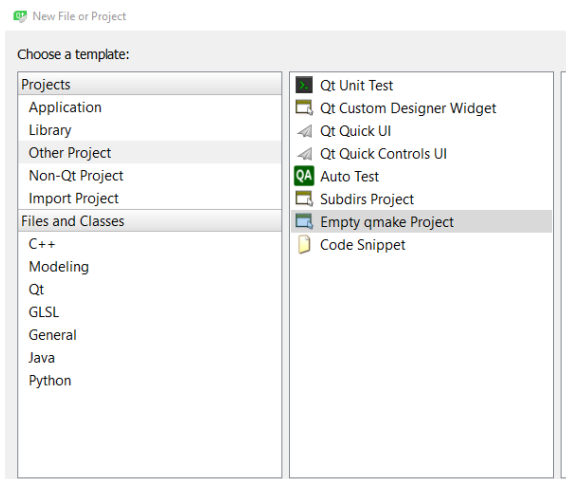
QML’de yazısal değerler tek tırnak içerisinde ya da çift tırnak içerisinde belirtilebilirler. Tek tırnak ile çift tırnak arasında bir farklılık yoktur. Qml’deki elemanlar kabaca iki gruba ayrılmaktadır: Görsel (visual) elemanlar ve Görsel olmayan elemanlar. Görsel elemanlar Item sınıfından türetilmiştir ve bunlarla belirli property’ler ortak olarak kullanılmaktadır.

QML elemanlarının (item’lerinin) dokümantasyonu tam olarak yapılmıştır. QML elemanları nesne yönelimli bir biçimde birbirlerinden türetilmiş olabilmektedir. Bu türetme hiyerarşisinin en yukarısında Item elemanı vardır.

Sıfırdan Bir QML Uygulamasının Oluşturulması

Sıfırdan bir QML uygulaması için şunlar yapılır:

1) QtCreator’da “Projects/Empty qmake Projects” seçilir.



2) qmake dosyasının (.pro dosyasının) içerisine aşağıdaki satır eklenir:

```
TEMPLATE = app
QT += qml quick
```

3) Sonra projeye bir .cpp kaynak dosyası eklenir (Örneğin main.cpp olabilir). Sonra aşağıdaki kod yazılır:

```
#include <QGuiApplication>
```



```
#include <QQmlApplicationEngine>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQmlApplicationEngine engine;
    engine.load(QUrl("qrc:/main.qml"));

    return app.exec();
}
```

Burada ana Application sınıfı QGuiApplication sınıfıdır. QQmlApplicationEngine sınıfı ise qml dosyasını okuyarak GUI arayüzünü oluşturmakta kullanılır. exec fonksiyonu yine mesaj döngüsü işlemlerini yapmaktadır. QQmlApplicationEngine sınıfının üç load fonksiyonu vardır. Bunlar URL olarak, dosya ismini alarak ya da doğrudan bir QByteArray olarak qml kodunu oradan okuyabilirler.

```
void load(const QUrl &url);
void load(const QString &filePath);
void loadData(const QByteArray &data, const QUrl &url = QUrl());
```

Pek çok uygulamada qml kodu bir kaynak olarak uygulamaya iliştilirip okuma oradan yapılmaktadır:

```
engine.load(QUrl("qrc:/main.qml"));
```

Burada QUrl sınıfı hem Intertte, hem yerel makinede hem de kaynakta yer belirtebilmektedir. Buraada qrc öneki ilgili main.qml dosyasının kaynak bulunduğunu belirtir.

4) Şimdi qml kodu “main.qml” isimli bir dosyaya yerleştirilir. O dosya da bir kaynak dosyası yaratıp onun içerisine yerleştirilmelidir. Tabii dosyanın isminin “main.qml” olması gerekmez. Ancak yukarıdaki programdaki isimle aynı isimli bir dosyanın yaratılması gerekir. main.qml dosyasının içeriği şöyle olabilir:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 1000
    height: 600
    color: "#FFFFFF00"
    title: qsTr("Merhaba Qt Quick")
}
```

Elemanların (Bileşenlerin) ID’leri

Bir Qml dosyasında yalnızca bir tek kök eleman bulunabilir. Bu nedenle birden fazla kök elemanlar farklı dosyalara yerleştirilirler. Elemanlar iç içe (nested) bildirilebilirler. İçteki eleman dıştaki elemanlarına “parent” anahtar sözcüğüyle erişir. Ancak Qml’de her elemanın isteğe bağlı olarak bir ismi de olabilir. Bu isim id property’si ile belirtilir. Eğer elemanlara isim verirsek onların property’lerine bu eleman isimleriyle erişebiliriz.

Gruplandırılmış Property’ler

Bazı property’lerin alt elemanları vardır. Bunlara gruplandırılmış property’ler (grouped properties) denir. Gruplandırılmış property’lerde her elemana ayrıcas nokta operatörüyle erişilebilir ya da küme parantezli sentak ile alt elemanlara nokta operatörü olmadan erişilebilir. Örneğin border gruplandırılmış property’sinin color elemanına aşağıdaki gibi değer atayabiliriz:

```
border.color: ‘#000000’
```

ya da aynı işlem şöyle yapılabilir:

```
border {  
    color: '#000000'  
}
```

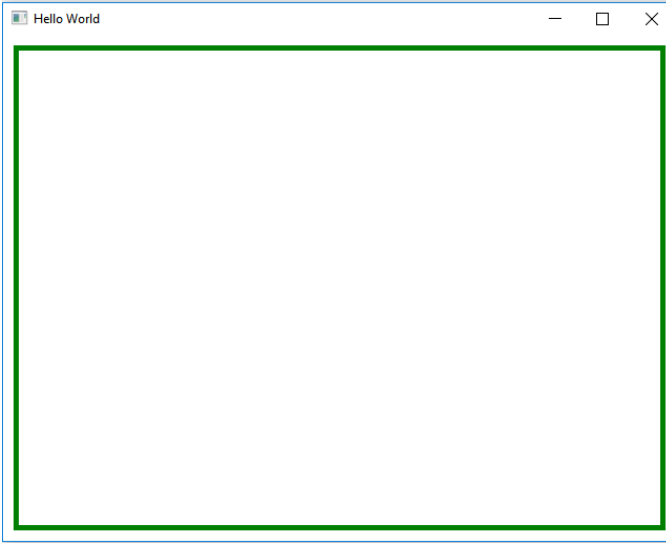
Örneğin:

```
import QtQuick 2.5  
import QtQuick.Window 2.2  
  
Window {  
    visible: true  
    width: 640  
    height: 480  
    title: qsTr("Hello World")  
  
    Rectangle {  
        id: myRectangle  
        x:10; y: 10  
        width: 400; height: 300  
        border {  
            color: 'green'  
            width: 5  
        }  
    }  
}
```

Property Değerlerine Erişim

Bir property'nin değeri başka bir property'de kullanılabilir. Eğer değeri kullanılacak property başka bir elemana ilişkinse o elemanın id'si isim olarak geçirilmelidir. Yukarıda da belirtildiği gibi üst (parent) elemana her zaman parent anahtar sözcüğüyle erişilebilir. Örneğin:

```
import QtQuick 2.5  
import QtQuick.Window 2.2  
  
Window {  
    id: root  
    visible: true  
    width: 640  
    height: 480  
    title: qsTr("Hello World")  
  
    Rectangle {  
        id: myRectangle  
        x:10; y: 10  
        width: root.width - 20; height: root.height - 20  
        border {  
            color: 'green'  
            width: 5  
        }  
    }  
}
```



Kullanıcı Tanımlı Property'ler (Custom Properties)

Programcı isterse property anahtar sözcüğü ile kendisi de property tanımlayabilir. Sonra onu istediği gibi diğer property'lerde kullanabilir. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    id: root
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")
    property int margin: 10

    Rectangle {
        id: myRectangle

        x: root.margin; y: root.margin
        width: root.width - root.margin * 2; height: root.height - root.margin * 2
        border {
            color: 'green'
            width: 5
        }
    }
}
```

Item Elemanının Bazı Önemli Property'leri

Tüm görsel QML elemanları Item denilen elemandan türetilmiştir. Dolayısıyla onun property'leri doğrudan kullanılabilir.

Item elemanının x, y, width ve height elemanları vardır. Tüm görsel elemanlarda bu property'ler konumlandırma ve boyutlandırma için gerekmektedir. Eğer width ya da height 0 olursa görsel eleman görüntülenemez.

Item elemanının opacity property'si elemanın arka planı gösterme derecesini belirtir.

Qml'de elemanlardaki üstlük-altlık (parent-child) ilişkisinde alt eleman koordinat olarak her zaman üst elemanı orijin olarak alır. Yani örneğin biz bir Text elemanı Rectangle eleman içerisinde alırsak. Bu Text Elemanın (0, 0) orijini Rectangle elemanın sol üst köşesidir. Elemanlar koordinat bakımından çakışık olabilirler. Bu durumda kimin yukarıda görüntüleneceği Z sırasına bağlıdır. Z sırası ileride ele alınacaktır. Elemanlar arasındaki üstlük altlık ilişkisinin diğer bir

önemi de alt elemanın property değerlerinin default olarak üst elemandan alınmasıdır. Yani alt ve üst elemanların ortak elemanları varsa alt elemanda bu ortak property'ler üst elemandan alınırlar. Bunun bazı kuralları da vardır. Elemanın color property'si yazının rengini belirtir. Text elemanı HTML formatlanmış bir yazıyı da görüntüleyebilir. Ancak yazının formatı textFormat property'si ile belirtilmektedir. Bu property default olarak AutoText biçimindedir. Bu durumda yazının formatına otomatik olarak karar verilir. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components")

    Rectangle {
        border.color: "black"
        width: 200; height: 200

        Text {
            x: 0; y: 0
            width: 200; height: 200
            font {
                pointSize: 14
                family: "Consolas"
            }
            color: "red"

            wrapMode: Text.WordWrap
            textFormat: Text.AutoText
            text: "<b>Hello</b> <i>World!</i>"
        }
    }
}
```

Rectangle Elemanı

Rectangle elemanı Item elemanın türetilmiştir. Dolayısıyla bir Item'in bütün özellikleri Rectangle'da vardır. Rectangle default olarak üst elemanın renginden bir sınır çizgisi çıkartır. border gruplandırılmış property.color property'si ile sınır çizgilerinin rengi, border.width property'si ile de kalınlığı değiştirilebilir. Çerçevenin iç rengi ise color property'si ile değiştirilmektedir. color temel türü kullanılabilecek tüm seçenekler

<https://www.w3.org/TR/SVG/types.html#ColorKeywords> adresinde verilmiştir.

radius property'si ise çerçevenin köşelerini yuvarlaklaştırmak için kullanılmaktadır. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    id: root
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")
    property int margin: 10

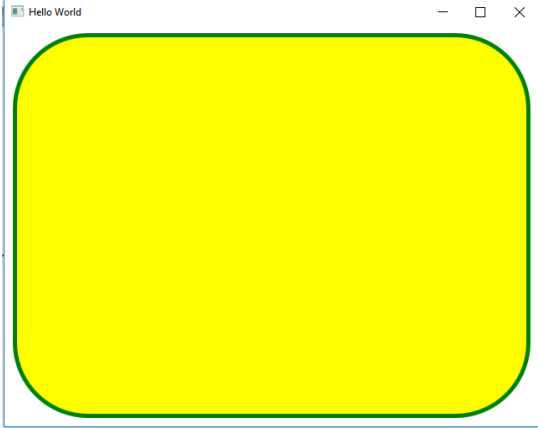
    Rectangle {
        id: myRectangle

        x: root.margin; y: root.margin
```

```

width: root.width - root.margin * 2; height: root.height - root.margin * 2
border {
    color: 'green'
    width: 5
}
color: 'yellow'
radius: 90
}
}

```



Elemanların Event'leri

Elemanlarda çeşitli event'ler bulunabilmektedir. Bu event'ler genellikle onXXX biçiminde isimlendirilmiştir. İlgili olay gerçekleştiğinde event elemana atanmış olan kod çalıştırılır. Örneğin onClicked isimli event (sinyal de diyebiliriz) oluştuğunda bir kodun çalıştırılması şöyle sağlanır:

```

onClicked: {
    <script kodu>
}

```

Hangi elemanların hangi event'lerinin olduğunu dokümantasyona bakarak anlayabiliriz. Örneğin:

```

import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 2.0

Window {
    id: root
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    Button {
        x: 10; y: 10
        text: "Ok"
        onClicked: {
            console.log("Ok")
        }
    }
}

```

MouseArea Elemanı

Bu eleman fare olaylarını izlemek için kullanılmaktadır. Yani elemanın amacı fare mesajlarının programcı tarafından ele alınmasını sağlamaktır. Biz görsel elemanımızın belli bir kısmını bu elemanla kaplayabiliriz ya da istersek hepsini kaplayabiliriz. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    id: root
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")
    property int count: 0

    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
        onPositionChanged: {
            console.log(count);
            ++count;
        }
    }
}
```

Farenin koordinatları elemanın mouseX ve mouseY property'lerinden elde edilebilir. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    id: root
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")
    property int count: 0

    MouseArea {
        anchors.fill: parent
        onClicked: {
            Qt.quit();
        }
        onPositionChanged: {
            console.log("X:" + mouseX + " Y:" + mouseY)
            ++count;
        }
    }
}
```

Normal olarak default durumda farenin yalnızca sol tuşu aktif durumdadır. Diğer tuşlarını aktif hale getirmek için elemanın acceptsButtons property'si set edilmelidir. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    id: root
    visible: true
```

```

width: 640
height: 480
title: qsTr("Hello World")
property int count: 0
color: "yellow"

MouseArea {
    x: 0; y: 0
    width: 100; height: 100
    acceptedButtons: Qt.LeftButton | Qt.RightButton

    onPressed: {
        if (pressedButtons == Qt.RightButton)
            root.color = "blue";
        else if (pressedButtons == Qt.LeftButton)
            root.color = "green";
    }

    onDoubleClicked: {
        Qt.quit();
    }

    onPositionChanged: {
        console.log("X:" + mouseX + " Y:" + mouseY)
        ++count;
    }
}
}

```

Text Elemanı

Text elemanı bir yazıyı göstermek için kullanılır. Bu elemanı işlev bakımından Widget kütüphanesindeki QLabel kontrolüne benzetebiliriz. Text elemanın text property'si görünülenecek yazıyı belirtir. Elemanın font isimli gruplandırılmış property'si yazının fontunu ayarlamakta kullanılır. Elemanın wrapMode isimli property elemanı sarmalama biçimi hakkında bilgi verir.

Bileşen Oluşturma İşlemi

Qml'in en önemli özelliklerinden biri hızlı biçimde kontrol (custom controls) oluşturabilmektir. Genel olarak programcının kendi oluşturduğu bu elemanlara bileşen denilmektedir. Bir bileşen bir dosya içerisinde bir kök elemanla gerçekleştirilir. Dosya kaynak tutulabileceği gibi, dosyası isteminde ya da uzak makinede bulunabilir. Örnek bir kaynak tabanlı bileşen şöyle oluşturulabilir:

1) Kaynak dosyasının (qml.qrc dosyasının) üzerine gelinip bağlam menüsünden Qt/Qml File seçilir. Aslında bu işlem yerine dosya Notepad gibi bir editörle yazılıp kaynağa eklenebilirdi. Burada yapılan da aynı şeydir. Bileşenin ismi dosyanın ismi olmaktadır.

2) Bileşenler için kök eleman Rectangle gibi bir eleman labileceği gibi genel olarak Item elemanı da olabilir. Bu durumda diğer elemanları biz bu Item elemanın içerisine yerleştiririz.

Örneğin:

```

/* MyButton.qml */

import QtQuick 2.0

Item {
    id: rootId
    property alias mouse: mouseArea

```

```

Rectangle {
    width: 80
    height: 50

    border.color: "black"
    color: "gray"

    Text {
        text: "Ok"
        anchors.centerIn: parent
    }

    MouseArea {
        id: mouseArea
        x: 0
        y: 0
        width: 80
        height: 50

        anchors.centerIn: parent
    }
}

```

Bir bileşeni kullanırken dışarıdan biz yalnızca ana elemana (kök elemana) erişebiliriz. Ancak bileşenin içerisindeki dosyadan id belirtilerek tüm elemanlara erişilebilmektedir. İşte dışarıdan bileşenin yalnızca kök elemanına erişilebildiği için bazı iç elemanları kök elemana isimsel olarak aktarmak gerekir. Bunun için property alias bildirimi kullanılır. Örneğin:

```
property alias mouse: mouseArea
```

Bu bildirim uygulandığında artık dışarıdan mouse kullanıldığında içerideki MouseArea elemanının id'si anlaşılacaktır. Şimdi bu bileşeni kullanalım:

```

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components")

    MyButton {
        x: 100
        y: 100
        mouse.clicked: {
            console.log("clicked")
        }
    }
}

```

Column ve Row Elemanları

Birtakım görsel öğeleri yatay ya da dikey dizmek için Column ve Row elemanları kullanılmaktadır. Column dikey, Row ise Yatay dizilim için kullanılır. Column ve Row elemanları da Item elemanında türetilmiştir. Dolayısıyla Item elemanındaki property'ler bunlarda da kullanılabilir.

Her iki elemanın da spacing property'si elemanlar arasındaki aralığı belirlemek için kullanılır. Örneğin:


```

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Custom Components")

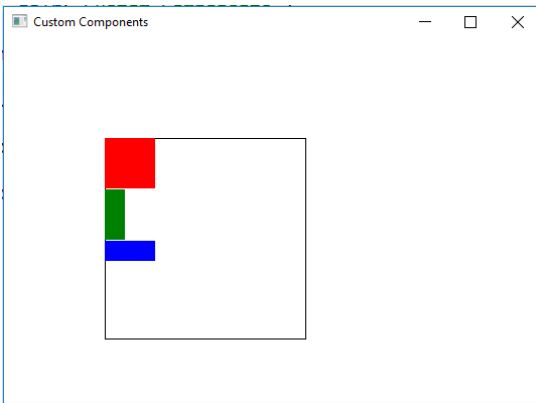
    Rectangle {
        x: 100; y: 100
        width: 200; height: 200

        border.color: "black"

        Column {
            spacing: 1

            Rectangle { color: "red"; width: 50; height: 50 }
            Rectangle { color: "green"; width: 20; height: 50 }
            Rectangle { color: "blue"; width: 50; height: 20 }
        }
    }
}

```



Grid Elemanı

Grid matrisel bir yerleşim sunmaktadır. Bu anlamda Column ve Row elemanlarının bir birleşimi gibi düşünülebilir. Grid elemanının columns ve rows property'leri grid'te kaç satır kaç sütun bulunacağını belirtir. Yalnızca columns property'si girilebilir. Bu durumda zaten elemanlar sırasıyla sonraki satırlara yerleştirilir. Elemanların nerede olacağı ayrıca belirtilmez. Eleman içerisindeki sıraya göre bu tespit yapılır. Yine Grid elemanında da spacing property'si vardır. Örneğin:

```

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Grid")

    Rectangle {
        x: 100; y: 100

        width: 112; height: 112
        color: "#303030"
    }
}

```

```

Grid {
    anchors.horizontalCenter: parent.horizontalCenter
    anchors.verticalCenter: parent.verticalCenter
    columns: 2
    spacing: 6

    Rectangle { color: "#aa6666"; width: 50; height: 50 }
    Rectangle { color: "#aaaa66"; width: 50; height: 50 }
    Rectangle { color: "#9999aa"; width: 50; height: 50 }
    Rectangle { color: "#6666aa"; width: 50; height: 50 }
}
}
}

```



Flow Elemanı

Burada elemanlar eklendikçe onların kapladıkları yer dikkate alınarak matrisel biçimde yerleştirme yapılır. Böylece satırlarda farklı sayıda elemanlar bulunabilir. Örneğin:

```

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Grid")

    Rectangle {
        color: "lightblue"
        width: 300; height: 200

        Flow {
            anchors.fill: parent
            anchors.margins: 4
            spacing: 10

            Text { text: "Text"; font.pixelSize: 40 }
            Text { text: "items"; font.pixelSize: 40 }
            Text { text: "flowing"; font.pixelSize: 40 }
            Text { text: "inside"; font.pixelSize: 40 }
            Text { text: "a"; font.pixelSize: 40 }
            Text { text: "Flow"; font.pixelSize: 40 }
            Text { text: "item"; font.pixelSize: 40 }
        }
    }
}

```

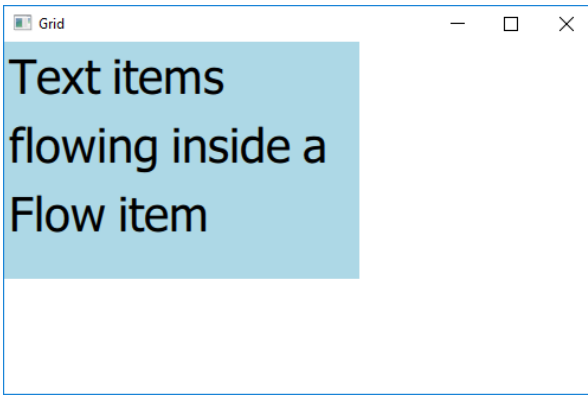


Image Elemanı

Image elemanı bir resmi görüntülemek için kullanılır. Elemanın source property'si gösterilecek resmi alır. Bu resim kaynaktaki bir resim, diskte yol ifadesi ya da bir URL olabilir. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Grid")

    Image {
        anchors.fill: parent
        source: "http://galeri4.uludagsozluk.com/112/abbey-road_164958.jpg"
    }
}
```

Burada "Abbey Road" resmi ana pencereyi kaplayacak biçimde gösterilir. Elemanın sourceSize gruplanmış property'si resmi ölçeklendirerek görüntüler. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Grid")

    Image {
        source: "http://galeri4.uludagsozluk.com/112/abbey-road_164958.jpg"
        anchors.centerIn: parent
        sourceSize.width: 200
        sourceSize.height: 300
    }
}
```

Item Elemanının Anchors Gruplandırılmış Property'si

Item elemanında anchors isimli bir gruplandırılmış property vardır. Bu property belli bir elemanın diğerine göre konumunu muhafaza etmek için kullanılmaktadır.

anchors.fill property'si ilgili elemanın belli bir elemanın içini dolduracağını belirtir. Üst eleman boyutlandırıldıkça alt eleman da boyutlandırılır. Örneğin:

```
// Box.qml

import QtQuick 2.0

Rectangle {
    id: rectangleId
    border.color: "red"
}

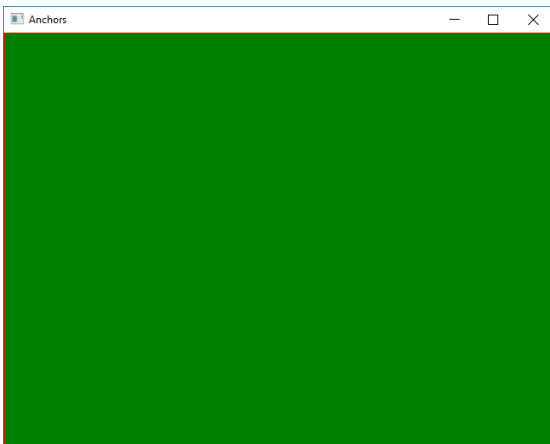
// main.qml

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    Box {
        anchors.fill: parent
        id: greenBox
        color: "green"
    }
}
```

anchors.fill property'si kullanıldıktan sonra artık alt elemanın x, y, width ve height property'lerinin hangi değere set edildiklerinin bir önemi kalmamaktadır.



anchors.centerIn property'sinde alt eleman üst elemanın ortasında görüntülenir. Ancak onun boyutlarını almaz. Alt elemanın ayrı boyutları (width, height) olmalıdır. Örneğin:

```
// box.qml

import QtQuick 2.0

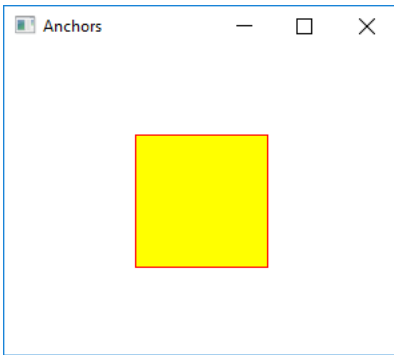
Rectangle {
    id: rectangleId
    border.color: "red"
    width: 100; height: 100
}
```

```
// main.qml

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")

    Box {
        id: yellowBox
        anchors.centerIn: parent
        color: "yellow"
        width: 100
        height: 100
    }
}
```



anchors.top, anchors.bottom, anchors.left ve anchors.right property'leri başka bir kardeş elemana referans edilerek verilir. Böylece o eleman diğerinin belli bir köşesine göre konumlanır. Örneğin:

```
// box.qml

import QtQuick 2.0

Rectangle {
    id: rectangleId
    border.color: "red"
    width: 100; height: 100
}
```

```
// main.qml

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")

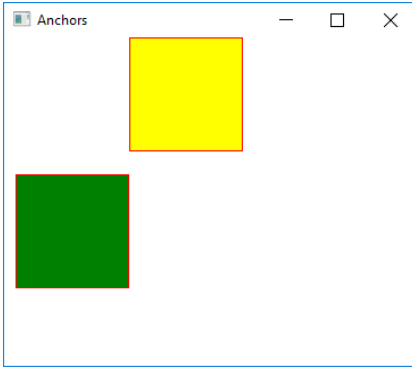
    Box {
        id: greenBox
        color: "green"
        x: 10; y: 120
        width: 100
    }
}
```

```

        height: 100
    }

    Box {
        id: yellowBox
        color: "yellow"
        anchors.left: greenBox.right
        width: 100
        height: 100
    }
}

```



Örneğin:

```

// box.qml

import QtQuick 2.0

Rectangle {
    id: rectangleId
    border.color: "red"
    width: 100; height: 100
}

// main.qml

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")

    Column {
        spacing: 10
        x: 50
        y: 10
        Box {
            id: greenBox
            color: "green"
            width: 100
            height: 100
        }

        Box {
            id: yellowBox
            color: "yellow"
            anchors.left: greenBox.right

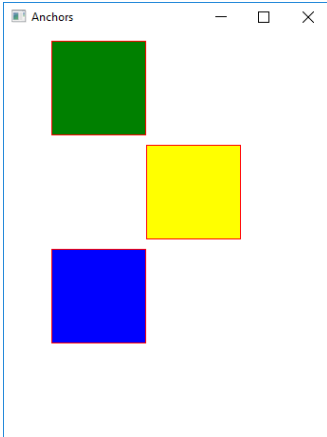
```

```

        width: 100
        height: 100
    }

    Box {
        id: blueBox
        color: "blue"
        width: 100
        height: 100
    }
}

```



`anchors.topMargin`, `anchors.bottomMargin`, `anchors.leftMargin`, `anchors.rightMargin` property'leri dört kenardan dışsal marjın bırakmak için kullanılır. `anchors.margins` ise dört kenarın hepsi için margin vermektedir. Örneğin:

```

// box.qml

import QtQuick 2.0

Rectangle {
    id: rectangleId
    border.color: "red"
    width: 100; height: 100
}

// main.qml

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")

    Box {
        id: greenBox
        color: "green"
        width: 100
        height: 100
    }

    Box {
        id: yellowBox
        color: "yellow"
        anchors.top: greenBox.bottom
    }
}

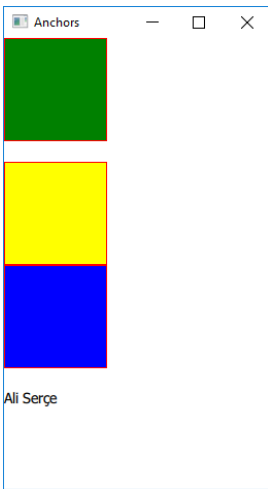
```

```

        width: 100
        height: 100
        anchors.topMargin: 20
    }

    Box {
        id: blueBox
        color: "blue"
        width: 100
        height: 100
        anchors.top: yellowBox.bottom
    }
    Text {
        text: "Ali Serçe"
        anchors.top: blueBox.bottom
        anchors.topMargin: 20
    }
}

```



Elemanların Döndürülmesi

Elemanları döndürmek için Item elemanından gelen rotation property'si kullanılır. Döndürme derece cinsindendir ve saat yönüne doğrudur. Örneğin:

```

// box.qml

import QtQuick 2.0

Rectangle {
    id: rectangleId
    border.color: "red"
    width: 100; height: 100
}

// main.qml

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")
}

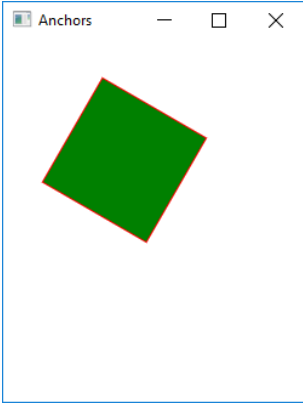
```



```

Box {
    id: greenBox
    antialiasing: true
    x: 50; y: 50
    width: 100; height: 100
    color: "green"
    rotation: 30
}
}

```



Repeater Elemanı

Repeater bir elemanı çoklamak için kullanılır. Ancak bu çoklama etkisi Row, Column, Grid, Flow gibi elemanların içerisinde etki gösterir. Elemanın count property'si Repeater içerisindeki eleman sayısını belirtir. model property'si herhangi bir model olabilir. Model konusu ileride daha ayrıntılı ele alınacaktır. Ancak model bir değer de içerebilir. Örneğin:

```

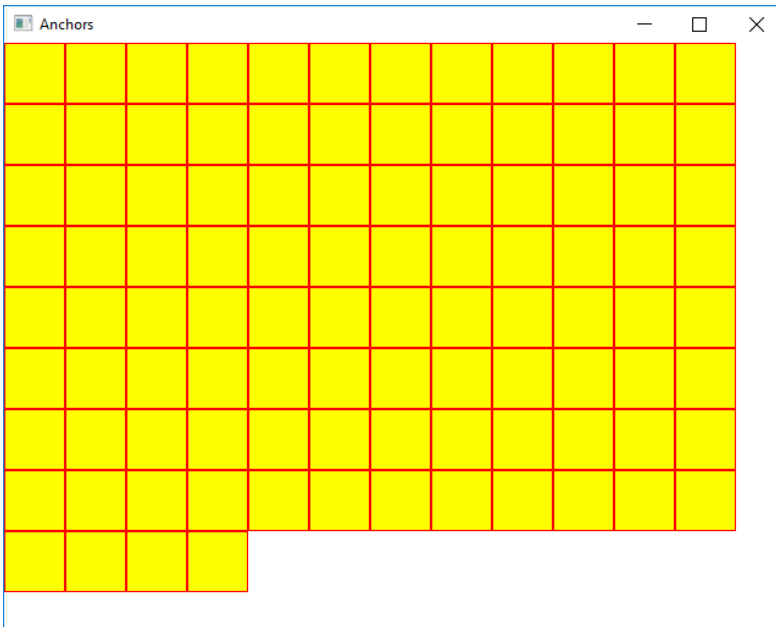
// main.qml

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")

    Flow {
        anchors.fill: parent
        Repeater {
            model: 100
            Rectangle {
                width: 50; height: 50
                border.width: 1
                border.color: "red"
                color: "yellow"
            }
        }
    }
}

```



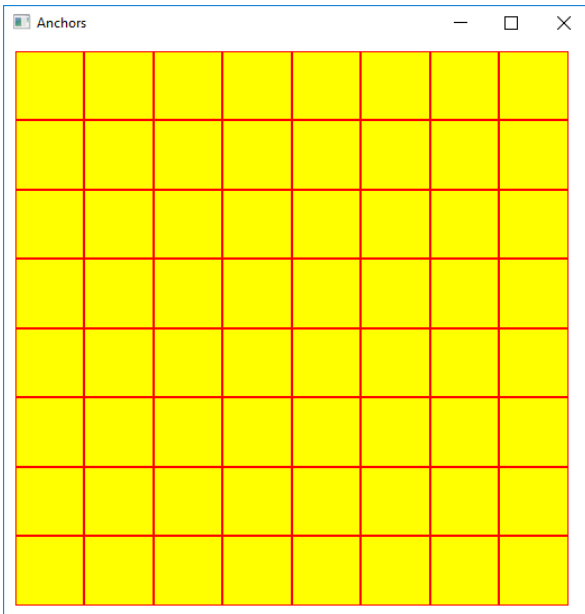
Örneğin:

```
// main.qml

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    id: root
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")

    Grid {
        id: idGrid
        anchors.fill: parent
        anchors.margins: 10
        columns: 8
        Repeater {
            model: 64
            Rectangle {
                width: (idGrid.width < idGrid.height ? idGrid.width / 8 : idGrid.height / 8)
                height: (idGrid.width < idGrid.height ? idGrid.width / 8 : idGrid.height / 8)
                border.width: 1
                border.color: "red"
                color: "yellow"
            }
        }
    }
}
```



Repeater'ın index property'si o anda yaratılmış olan elemanın indeks değerini bize verir. Bu yolla biz elemana özgü bazı şeyler yapabiliriz. Örneğin:

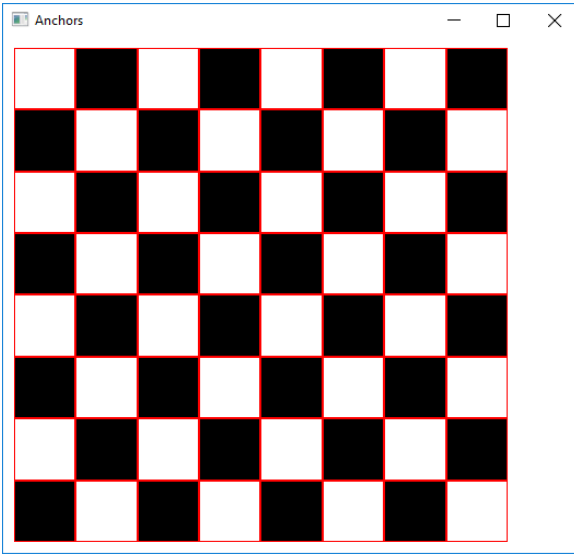
Örneğin:

```
// main.qml

import QtQuick 2.5
import QtQuick.Window 2.2

Window {
    id: root
    visible: true
    width: 640
    height: 480
    title: qsTr("Anchors")

    Grid {
        id: idGrid
        anchors.fill: parent
        anchors.margins: 10
        columns: 8
        property int min: (idGrid.width < idGrid.height ? idGrid.width / 8 : idGrid.height / 8);
        Repeater {
            model: 64
            Rectangle {
                width: idGrid.min
                height: idGrid.min
                border.width: 1
                border.color: "red"
                color: (index % 8 + Math.floor(index / 8)) % 2 == 0 ? "white" : "black"
            }
        }
    }
}
```



QML'de Kontroller

Aslında Qml'de pek çok kontrol mevcut temel elemanlarla kısa bir QML koduyla yazılabilmektedir. Ancak bazı kontroller hazır olarak da bulundurulmuştur. Bu kontrolleri kullanabilmek için versiyona göre Controls kütüphanesinin import edilmesi gerekir. Örneğin:

```
import QtQuick.Controls 2.0
```

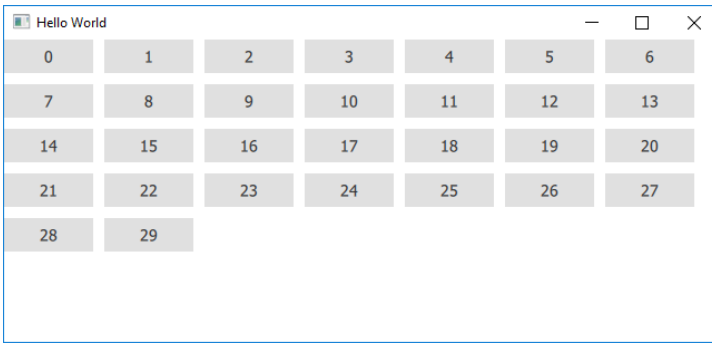
Button Kontrolü

Button kontrolü klasik düğme işlevini gerçekleştirir. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 2.0

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    Flow {
        anchors.fill: parent
        spacing: 10
        Repeater {
            model: 30
            Button {
                x: 10; y: 10
                width: 80; height: 30
                text: index
                onClicked: {
                    console.log( text + " button clicked")
                }
            }
        }
    }
}
```



Button kontrolünün onClicked event'i düğmeye basıldığında tetiklenir.

Label Kontrolü

Şüphesiz Label kontrolü yerine temel Text elemanı da kullanılabilir. Ancak Label'ın kontrol olmasından kaynaklanan birkaç avantajı da vardır.

TextField Kontrolü

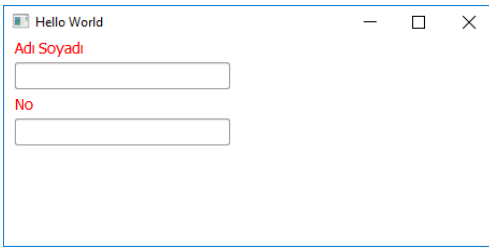
Bu kontrol klasik textbox (QLineEdit) yaratmakta kullanılır. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 1.4

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    Column {
        x: 10
        spacing: 5;
        Label {
            text: "Adı Soyadı"
            color: "red"
        }
        TextField {
            id: textFieldName
            width: 200
        }

        Label {
            text: "No"
            color: "red"
        }
        TextField {
            id: textFieldNo
            width: 200
        }
    }
}
```



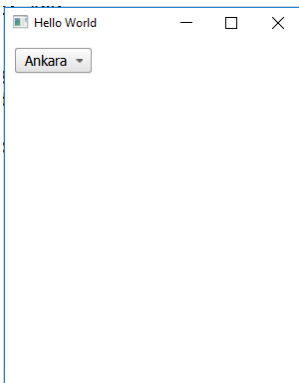
ComboBox Kontrolü

ComboBox kontrolü modelle çalışır. Model konusu ileride ele alınacaktır. Ancak model olarak bir string dizisi verilebilir. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 1.4

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    ComboBox {
        id: idComboBox
        x: 10; y: 10
        model: ["Ankara", "İzmir", "Adana", "İstanbul", "Siirt", "Eskişehir"]
    }
}
```



ComboBox elemanının currentIndex property'si seçilen elemanın indeksini, currentText property'si ise yazısını belirtir. currentIndex property'si read/write bir property'dir. Dolayısıyla belli elemanı seçmek için de kullanılabilir. Kontrolün editable property'si default olarak false durumdadır. Bu property true yapılırsa combobox edit edilebilir. Combobox'ın edit alanındaki yazı editText property'si ile elde edilir. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 1.4

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    ComboBox {
        id: idComboBox
        x: 10; y: 10
    }
}
```

```

        width: 100
        editable: true
        model: ["Ankara", "İzmir", "Adana", "İstanbul", "Siirt", "Eskişehir"]
    }

    Button {
        id: idButton
        x: 200; y: 10
        text: "Ok"
        onClicked: {
            print(idComboBox.currentText + " at index " + idComboBox.currentIndex)
            print("Edit text: " + idComboBox.editText)
        }
    }
}

```

Kontrolün count property'si kontrolde kaç elemanın bulunuyor olduğunu belirtir.

Kontrolün onCurrentIndexChanged isimli event'i seçili olan eleman değiştiğinde tetiklenir. Örneğin:

```

import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 1.4

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    ComboBox {
        id: idComboBox
        x: 50; y: 10
        width: 100
        editable: true
        currentIndex: 3
        model: ["Ankara", "İzmir", "Adana", "İstanbul", "Siirt", "Eskişehir"]
        onCurrentIndexChanged: {
            print(currentText)
        }
    }

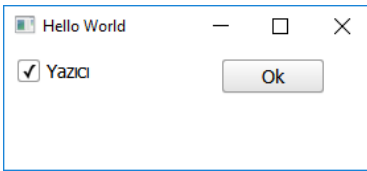
    Button {
        id: idButton
        text: "Ok"
        anchors.left: idComboBox.right
        anchors.top: idComboBox.top

        anchors.leftMargin: 20
        onClicked: {
            print(idComboBox.currentText + " at index " + idComboBox.currentIndex)
            print("Edit text: " + idComboBox.editText)
        }
    }
}

```

CheckBox Kontrolü

CheckBox kontrolü CheckBox isimli elemanla temsil edilir. Kontrolün kullanımı diğerlerine çok benzerdir. Kontrolün bool türden checked read/write property'si o anda kontrolün checked olup olmadığını bize verir ve konumunu değiştirmemize olanak sağlar. Örneğin:



Örneğin:

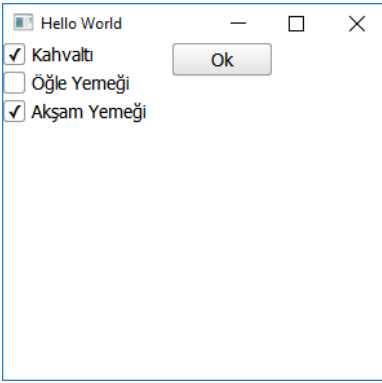
```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 1.4

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    Column {
        id: idColumn
        spacing: 5
        CheckBox {
            id: idCheckBoxBreakfast
            text: "Kahvaltı"
            checked: true
        }
        CheckBox {
            id: idCheckBoxLunch
            text: "Öğle Yemeği"
        }
        CheckBox {
            id: idCheckBoxDinner
            text: "Akşam Yemeği"
            checked: true
        }
    }

    Button {
        id: idButton
        text: "Ok"
        anchors.left: idColumn.right
        anchors.top: idColumn.top

        anchors.leftMargin: 20
        onClicked: {
            print("Kahvaltı: " + (idCheckBoxBreakfast.checked ? "Checked" :
"Unchecked"))
            print("Öğle Yemeği: " + (idCheckBoxLunch.checked ? "Checked" : "Unchecked"))
            print("Akşam Yemeği: " + (idCheckBoxDinner.checked ? "Checked" :
"Unchecked"))
        }
    }
}
```

RadioButton Kontrolü

Radyo düğmeleri RadioButton elemanı ile temsil edilmektedir. Radyo düğmelerinin bir grup oluşturması için düğmenin içerisine bir tane ExclusiveGroup elemanı yerleştirmek gerekir. Bu elemana bir id verdikten sonra RadioButton elemanlarının exclusiveGroup property'lerini bu id olarak set etmek gerekir. Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 1.4

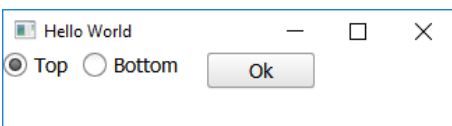
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    Row {
        id: idRow
        ExclusiveGroup { id: tabPositionGroup }
        spacing: 10
        RadioButton {
            text: "Top"
            checked: true
            exclusiveGroup: tabPositionGroup
        }
        RadioButton {
            text: "Bottom"
            exclusiveGroup: tabPositionGroup
        }
    }

    Button {
        id: idButton
        text: "Ok"
        anchors.left: idRow.right
        anchors.top: idRow.top

        anchors.leftMargin: 20
        onClicked: {

        }
    }
}
```



Seçilen eleman tek tek RadioButton kontrollerinin checked property'sine bakılarak anlaşılabilir. Örneğin:

```

import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 1.4

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    Row {
        id: idRow
        ExclusiveGroup { id: tabPositionGroup }
        spacing: 10
        RadioButton {
            id: idRadioTop
            text: "Top"
            checked: true
            exclusiveGroup: tabPositionGroup
        }
        RadioButton {
            id: idRadioBottom
            text: "Bottom"
            exclusiveGroup: tabPositionGroup
        }
    }

    Button {
        id: idButton
        text: "Ok"
        anchors.left: idRow.right
        anchors.top: idRow.top

        anchors.leftMargin: 20
        onClicked: {
            if (idRadioTop.checked)
                print("Top radio button checked")
            else if (idRadioBottom.checked)
                print("Bottom radio button checked")
        }
    }
}

```

QML'de JavaScript Kullanımı

Aslında QML içerisindeki kodlar JavaScript kodlarıdır. JavaScript ECMA tarafından standardize edilmiştir. QML içerisinde JavaScript büyük ölçüde Web uygulamalarındaki gibi kullanılabilir. Ancak bazı farklılıklar ve eksiklikler de bazı yerlerde bulunmaktadır. Kursumuzda JavaScript dili ele alınmamaktadır. Ancak dilin sentaksı C++'a oldukça benzer. Tabii JavaScript yorumlayıcı ile çalışan basit bir dildir. Javascript dinamik tür sistemine (dynamically typed) sahiptir. Değişkenlerin türleri onlara atanan değerle programın çalışma zamanı sırasında değişmektedir.

Javascript'te fonksiyon tanımlamak oldukça kolaydır. Tanımlama işleminin genel biçimi şöyledir:

```

function <isim> ([parametre listesi])
{
    //...
}

```

Fonksiyonlar herhangi bir elemanın içerisinde tanımlanabilirler. Ancak istenirse fonksiyonlar ayrı bir JavaScript dosyasına da yerleştirilebilirler. QtCreator'da bunu yapmak için fare ile kaynak dosyasının (qrc dosyasının) üzerine

gelinir ve farenin sağ tuşuna basılarak “Add New” seçilir. Buradan “Qt/JS File” seçilir. Dosyaya bir isim verilir. Sonra QML dosyasından bu JavaScript dosyası import edilir. Import işleminin sentaksı şöyledir:

import “JS dosyasının ismi” as <isim alan ismi>

Örneğin:

```
import QtQuick 2.5
import QtQuick.Window 2.2
import QtQuick.Controls 1.2
import "Util.js" as Util

Window {
    id: root
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")
    color: "yellow"

    TextField {
        id: idTextField
        x: 100; y: 10
        width: 200
    }

    Button {
        id: idButton
        x: 10; y: 10
        text: "Ok"

        onClicked: {
            Util.foo()
            print(idTextField.text)
            print(parseInt(idTextField.text) + 10);
        }
    }
}

// Util.js

function foo()
{
    print("foo")
}
```

JavaScript’te pek çok hazır global fonksiyon vardır. Biz bunları doğrudan QML kodlarında kullanabiliriz.

Fonksiyonların parametreleri bildirilirken tür belirtilmez. Yalnızca parametrelerin isimleri belirtilir. Örneğin:

```
function add(a, b)
{
    return a + b
}
```

Diziler köşeli parantezle ilkdeğer verilecek oluşturulurlar. Örneğin:

```
var a[] = [1, 2, 3, 4]
```

Bildirimde var anahtar sözcüğünün kullanılması zorunludur. Örneğin bir dizinin en büyük elemanına geri dönen bir fonksiyon yazmak isteyelim:

```
function getMax(a)
{
    var max = a[0]

    for (var i = 1; i < a.length; ++i)
        if (max < a[i])
            max = a[i]

    return max
}
```

Fonksiyon şöyle çağrılabilir:

```
Button {
    id: idButton
    x: 10; y: 10
    text: "Ok"

    onClicked: {
        var result = Util.getMax([3, 67, 45, 34, 18])
        print(result);
    }
}
```

Örneğin:

```
function concat(strArray)
{
    var str = ""

    for (var i = 0; i < strArray.length; ++i) {
        if (i != 0)
            str += ", "
        str += strArray[i]
    }

    return str
}
```

for in deyiminde döngü her yinelendikçe sıradaki indeks değeri döngü değişkenine atanır. Örneğin:

```
function getMax(a)
{
    var max = a[0]

    for (var i in a)
        if (max < a[i])
            max = a[i]

    return max
}
```

Diziler read-only değildir. push fonksiyonuyla dizilerin sonuna eleman eklenip, pop fonksiyonuyla sondaki eleman silinebilir. Örneğin:

```
Button {
    id: idButton
    x: 10; y: 10
    text: "Ok"
```

```
onClicked: {  
    var a = [1, 2, 3, 4, 5]  
    a.push(20)  
    a.push(30)  
    a.pop()  
  
    print(a)  
}  
}
```