Lotto - a simple Cocoa app

Creating objects in Cocoa is fun! Programming should be fun, just like building with Lego blocks. Thanks to the versatile and well tested Cocoa framework it is too. Not only does this contain hundreds of nifty building blocks, you can design your own too. Of course, it takes time to master all the classes built into Cocoa. But, it is still possible to create useful programs with some familiarity with Objective-C and the basic Cocoa classes. This small program illustrates a few of the possibilities available to you, many more examples are available from Apple and other sources. Remember, the best way to find anything is to use Google!

A model application

Apple recommends that a Cocoa application is created according to the MVC model. This means that the data is separated from the graphical user interface and the program control. A data Model object handles the data, a View object the graphical display, and a Controller object interacts with the user. This makes for code that is easy to understand, maintain, and reuse.
Although this small app easily could be handled by a single source file, it was not difficult to subdivide into three objects according to the MVC model; LottoData, LottoView, and LottoControl.

Objects of class

Before we study the Lotto classes in detail, those of you who are unfamiliar with object oriented programming may appreciate a short introduction to the concept of objects and classes. An object is a self-contained unit (somewhat like a C struct) that combines data in variables and methods (functions) that operate on the data. Objects of the same kind belong to the same class. A class is both a blueprint and a factory that can be used to produce objects that are instances of the class. Objects communicate through messages that are requests to perform methods of the same name. A message expression looks like this:

```
[reciever message];
```

The reciever is either an object, another expression that evaluates to an object, a class name, or super (see below). A special case is 'self' which is a variable that refers to the object that performs the method. This is used when a method sends messages to its own class. Yes, this is tricky.
Let's say that a program requires several widget objects. For each widget, there will be an instance of the class ABWidget that specifies the looks and behavior of a particular widget. A sole ABWidget class object contains the methods that are the code used by all the widgets. To create a new widget we will send messages to the class that will allocate memory for the variables and initialize them as required. This is how it will look:

```
ABWidget *aWidget = [[ABWidget alloc] init];
```

The message +alloc will pass through ABWidget up the inheritance chain (se below) until it reaches the corresponding class method in the root object NSObject that will instantiate all new widgets by allocating memory for their variables. The method -init will perform any initializing that the instance object requires. The example also illustrates how the class name can be used in two different ways; as a name for statically typing the variable aWidget, and as a class object recieving a message.
An important concept in object oriented programming (OOP) is that of heredity. All objects (except NSObject) have a superclass above them, many also have their own subclasses that inherit the variables and methods of their supers. A subclass can expand a class by adding new methods and variables, it can also modify the inherited behaviour, either by completely overriding an existing method, or by incorperating an existing method in a new one. A new class without relatives should be a subclass of NSObject. Messages pass up the chain of inheritance towards the root until they find a matching method.
For a brief introduction to classes and Objective-C (ObjC), please see "The Objective-C Language" by Mike Beam <http://www.qreillynet.com/pub/a/mac/2001/05/04/cocoa.html>.

LottoData

Before you get more confused, it is best to go on with the example at hand. The unique class LottoData will create instances that contains a set of possible lotto numbers, here 35 of them. It has a method that creates the numbers, as well as a crucial instance method that scrambles them in a random order. The numbers are

stored in another object, an array. A pointer to this array is made available to the LottoView object that display the numbers.
LottoData creates the array when it catches the -init message, using its on init method. It is thus overriding the init method of the class above it. Many classes have more than one initializer – or constructor, the one with the most parameters is called the designated initializer. Thus, the LottoData init just calls its own designated initializer -initWithCount: using the deafult count of 35 as the parameter.

```
- (id)init {
    return [self initWithCount:35];
}
```

BTW, LottoControl uses init, but it could just as well use initWithCount: with a parameter. There are many ways to do things in Cocoa. Here comes the real stuff then.
We need two variables; a loop index and a number pointer. To let the class above LottoData do its own initializing, we must first pass the message on to super. If the super bit succeeds, we seed the C random generator with the time in seconds (if we wanted to create more than one instance of LottoData, we would have to modify the seed to give the Lotto sets different pseudorandom sequences, for example, by adding an index to the time for each instance).
The numbers are stored in an instance of the class NSMutableArray (mutable means that it can be modified). The 'numbers' variable that is declared in the class header file "LottoData.h" keeps a pointer to the array.

```
- (id)initWithCount:(int)count {
    int i;
    NSNumber *number;

    if ( self = [super init] ) {
        // initialize random number generator
        srandom(time(NULL));

       // create an array to hold the numbers
       numbers = [[NSMutableArray alloc] init];

        for ( i = 0; i < count; i++ ) {
            number = [[NSNumber alloc] initWithInt:i];
            [numbers addObject:number];
            [number release];
        }
    }
    return self;
}
```

Notice that the numbers that are stored in the array are objects of the class NSNumber, not simple integers. They have to be created and initialized, and their values are later read by the method -intValue. The last thing to do is to return a pointer to the class itself. This is eventually passed onto the View class by LottoControl. The return type 'id' is that of an untyped object, BTW.
Thanks to the richness of the Cocoa framework, the scrambling of the numbers is very easy to do. The method used is called -exchangeObjectAtIndex:withObjectAtIndex:. This looks weird at first, but you soon realize that this is a clever way of handling multiple parameters.

```
- (void)randomize {
    int i;
    int n = [numbers count];

    for ( i = 0; i < n; i++ ) {
        [numbers exchangeObjectAtIndex:i withObjectAtIndex:(random() % n)];
    }
}
```

An important concept in Cocoa is that variables as a rule are internal to the class. A nice effect of this is that you can use the same variable names in different classes – and the same method names too – but that is

another story. To give other classes access to the data in the variables it is customary to define "Accessor methods." It is simple to read a nonobject value of a variable, just return its contents. For objects, return the variable, i.e. a pointer to the object.

```
- (NSMutableArray *)numbers {
    return numbers;
}
```

Storing a nonobject value is also simple, just assign a new value. Objects are more complicated as one has to deal with memory management. We must make sure that the new data pointed to by the 'array' parameter is retained, while the memory used by the old data is released, and in the right order too. There are several possible ways of doing this, the example below shows how it could look in LottoData. Although this variable won't be set by other classes, the set method is used when disposing of the array (see below).

```
- (void)setNumbers:(NSMutableArray *)array {
    [array retain];
    [numbers release];
    numbers = array;
}
```

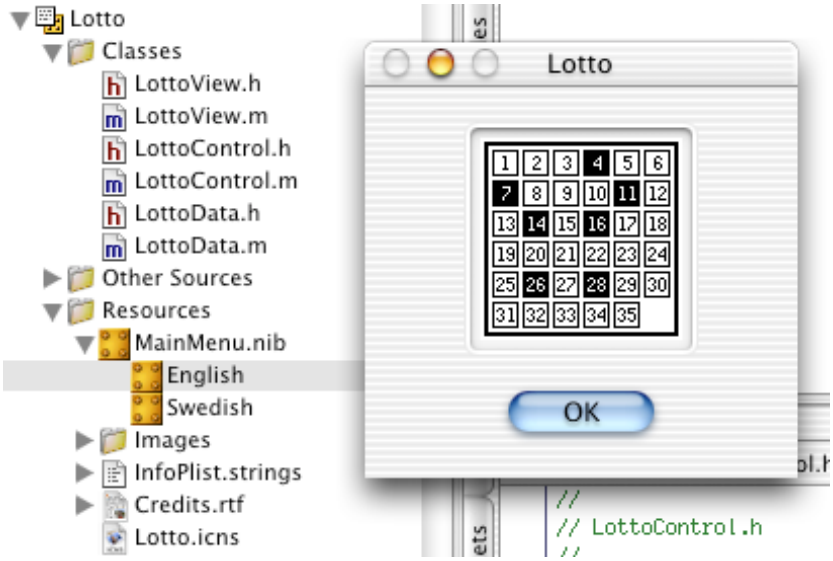
It is often good practice to use the accessor methods even inside ones own class, as this can simplify the inclusion of AppleScript and Undo/Redo management. This is exemplified in the last method of this class which deals with the cleanup that many classes have to consider for objects that are destroyed. At this point, they are sent the message -dealloc which must be passed onto the superclass. Although, Lotto app never gives up its data until it is terminated, its more powerful sibling LottofleX requires this method when the size of the lotto field is changed.

```
- (void)dealloc {
    NSLog(@"Destroying Data %@", self);
    [self setNumbers:nil]; // numbers is released in set method
    [super dealloc];
}
```

We could simply have used [numbers release], but instead sets the pointer to 'nil' with the accessor method that releases the array (sending retain to nil is OK). NSLog is similar to 'printf' in C, its output is visible in the console window of Project Builder where it can be used for debugging purposes. An addition to the printf formats, %@ is used by NSObject to execute the object's 'description' method. You can override this method in your own classes, as I have done in LottoData! The compiler directive @"Some text here" is a shortcut for creating an NSString, the Cocoa string object.
BTW, the 'NS' is said to signify NeXTStep, the origin of Cocoa. Although I have not done so in Lotto, it is a good idea to prefix your own class names with some suitable letter combination.

LottoView

In the original Lotto, written for Classic Mac OS, the numbers in the playing field were highlited by inverting a small rectangle of a PICT bitmap image displayed on the screen. This required only one simple instruction called InverRect. On the Apple Newton, which, BTW, uses a very nice object oriented programming environment, the same thing was accomplished by drawing a black rectangle over the numbers using the Exclusive OR (XOR) transfer mode. I suspected that something similar would be necessary in Cocoa. That proved to be correct but it was not at all that simple. However, before we go on, it may help to see how the finished application will look.



After much experimenting and searching online I learned that it is not trivial to composite freely to the screen, it was necessary to use an intermediate image, as shown below. Since then, Apple has published the example "TintedImage" that is well worth studying, if you are a Cocoa newbie like me.
In addition, the Quartz drawing methods involved require a transparent TIFF image with a premultiplied alpha channel. This is described in the "Aquatint" help <http://www.sticksoftware.com/software.html> (you can also use "tiffutil" from the Terminal application, look for information in Google). Here is how the final drawing looks in principle:

```
NSImage *lotto = [[NSImage alloc] initWithSize:canvas.size];
[lotto lockFocus];
[[NSColor blackColor] set];
[NSBezierPath fillRect:NSMakeRect(0, 0, 11, 11)]; // draw the black mask
[[self image] compositeToPoint:NSZeroPoint operation:NSCompositeXOR];
[lotto unlockFocus];
[lotto compositeToPoint:canvas.origin operation:NSCompositeSourceOver];
```

This is a mouthful, but in essense we create a new image object that combines the black masks (only one here) and the old playing field using XOR, and then draws this composite image to the screen. Notice the nesting of messages in the first line that is very common in ObjC. Thanks to this, you can accomplish a lot in a single line, and it is still quite readable. Here is another great example from the method that draws the seven masks:

```
n = [[[[self data] numbers] objectAtIndex:i] intValue];
```

This translates to: First find the LottoData object, then get its numbers array, find the number at the index, and convert this number object to an integer that is placed in the variable 'n.'
The actual drawing is done inside the method -drawRect: which is defined in NSView. It is overridden in Lotto's NSImageView that displays the playing field image and the nice frame around it. This is how the inheritance chain looks for this object: NSImageView : NSControl : NSView : NSResponder : NSObject.

LottoControl

LottoView was first designed graphically in Interface Builder (IB), the companion to Project Builder (PB) where you compose your code and compile it. IB creates Nibs, which are bundles of resources that are instantiated as objects and connections between them, either automatically when the program launches, or when requested to do so.
Lotto has one Nib with the default name – MainMenu.nib. Apart from the menu stuff, this contains two new objects, a window for the Lotto program, and a controller object – LottoControl.
While the LottoData class was created in PB, LottoControl was created in IB where it also was given two outlets and one action. An 'outlet' is simply a pointer to another object, while an 'action' refers both to the message and the corresponding method invoked by this in the 'target' of the action. This target/action pattern is central to the interaction between user controls and the Cocoa application. Outlets and actions are

connected by simply clicking and dragging in IB.
The GUI (Graphical User Interface) of Lotto is very simple, one button and an NSImageView with its frame. The outlets go to each one of these, while the action comes from the button that will send a message to the control whenever the user interacts with it.
With all connections made, IB created files for the controller in PB. If you examine the header file LottoControl.h, you will see the declarations for the outlets and the action.
The actual implementation in LottoControl.m is not complicated. When the Nib is unarchived and the objects are available, it recieves the message -awakeFromNib. The corresponding method first creates an instance of the LottoData class. The fact that it is autoreleased means that it will be disposed of when the current event loop terminates. By then it will already be in the hands of LottoView, however.

```
// use simple convenience class method that returns an autoreleased object
data = [LottoData lottoData];

// turn the data over to the view class which will retain it
[lottoView setData:data];
```

The data pointer is stored in a LottoView variable, by way of the LottoView outlet 'lottoView,' using its accessor method -setData.
Next, a flag is set in the button object that will make this send its action message continously when the button has been pressed for a while. This is where we make use of the button outlet 'lottoButton.' The continous action can also be accomplished by setting a switch in IB.

```
[lottoButton setContinuous:YES];
```

LottoControl is the target for the action method -play: that just tells the data to scramble itself, and the view that it should redraw its contents. The message -randomize is sent to the data object by way of the 'get' accessor method in LottoView.

```
- (IBAction)play:(id)sender {
    [[lottoView data] randomize];
    [lottoView setNeedsDisplay:YES];
}
```

For some reason, deminiaturizing the Lotto window from the Dock introduces an artifact in the game field. Updating this with -setNeedsDisplay: removes the artifact. This is easily accomplised using one of the notifications sent by the window handler when the window changes its properties. The notifications are sent to the windows 'delegate,' which here is LottoControl. This relationship is set in IB.

```
- (void)windowDidDeminiaturize:(NSNotification *)notification {
    [lottoView setNeedsDisplay:YES];
}
```

Well, that is about all there is to this little program. One other thing might be good to know, it is very easy to localize a program for different language areas. Initially, Lotto has one Nib file for English and one for Swedish. In addition, there are different versions of the credit text. You can see some of these in the image of the finished application above.

LottofleX

The simple version described here is probably only valid for the Swedish Lotto system, but it should not be difficult to adapt the program for other systems. The playing field can be modified in Graphic Converter, for example. The Aquatint documentation explains how to create a stenciled image, as noted above.
The more deveolped LottofleX uses another approach, creating the game field entirely in code. When the Lotto game size is set in a Preference panel, the playing field, and – if necessary – the window, change their sizes accordingly. The data set is also recreated with the required count of numbers. This version of the program is considerably more complex, and somewhat slower, but should support virtually all Lotto games. You can download LottofleX and its source code on the site too <http://mac.tidings.nu/lotto.html>.

Developer Tools

If you have not already done so, install the developer package in your new Mac – you will find 'Developer.mpkg' in /Applications/Installers/. It also comes with the complete Mac OS X package, and can be purchased from the Apple Developer site (after a free registration).
The developer installation includes tons of documentation including some more Cocoa examples, but it is often easier to find the right reference on the web using Google! For example, 'Cocoa examples Apple' will directly take you to <http://developer.apple.com/cocoa/>. The class documentation is best referenced with an application such as AppKiDo that is free from <http://homepage.mac.com/aglee/downloads/>.
Fire up BP and IB in the /Developer/Applications/ folder, open Lotto.pbproj, and have fun!

Bertil Holmberg
bertilholmberg@telia.com