# MacZoop “Hello World” Tutorial
# (Revised for MacZoop 2.5)

This tutorial is designed to familiarise you with the very basic essentials of using MacZoop to create a working application. In time-honoured tradition, the application opens a window displaying “Hello World”. In part two, the window is display is made more sophisticated by adding the ability to scroll, print and draw additional graphics.

***Part One, the basics.***

In this section you will learn how to subclass ZWindow to make a new kind of window, and change the settings of the “Project Settings” file so that your window is created instead of the default type. This part of the tutorial is identical to that in the original MacZoop 2.1 manual.

In this chapter, we are going to see how we use MacZoop to actually do some real programming. Before you start this tutorial, make sure that you can compile and run the basic project without any problem. If there’s something wrong with your set-up, you need to fix it before attempting this- that way you won’t be trying to hit a moving target. You may also like to try to compile and run the demo project. This isn’t required for the tutorial, but it may help give you confidence in your set-up and MacZoop in general. It’s up to you...

OK, this tutorial guides you through some basic principles of MacZoop, and object-oriented programming in general. We are going to use MacZoop to create a custom, standalone Macintosh application, which will display the immortal words “Hello World!” in large, friendly letters in a nice on-screen window. While not of much intrinsic value, this will teach you:

• How to customise MacZoop to create your own window types rather than generic ones
• How to subclass a standard MacZoop object
• How to override object methods to do useful work
• A little about how MacZoop applications are put together

*A brand new project....*

Create a new project by cloning the minimal MacZoop project. Remember, create a new folder in your projects folder (call it “Hello World project” for example). Make sure you have a project file and your own ProjectSettings.h file. If you cloned the minimal project folder, discard all the other stuff in it for now.

Open the new project file in CodeWarrior. Open the settings, and change the application name (in the Target panel) to something suitable- say “Hello World”. Now, before doing anything else, make sure it compiles and runs cleanly without any problems. If this is not the case FIX it first! (refer to the troubleshooting information in the previous chapter).

*The Window Class*

The first thing we need to do is to define our window class. This is the object that will display the “Hello World!” text to the outside world. We don’t need anything too fancy- just a basic window with enough space to show the text. In fact the built-in window template will serve just fine. However, the built-in window object- ZWindow- doesn’t display anything except a vast expanse of crisp white, so we need to make a new class that will do this job. Since ZWindow already knows a great deal about how to be a window, we will use this as a starting point.

So:

• Open a new blank text file. This will become our class header. To do this, select New from the File menu in CodeWarrior.

Now, it’s not actually necessary to do this, but it is a good habit to form- add a comment to the top of the file that describes what it is. All MacZoop header files have a comment like this, so a good shortcut is simply to open one of them up and copy and paste that part into the new file. Then edit the info as you require.

This is going to be a header file, and we need to ensure that header files are only actually read by the compiler once. Not only is this more efficient, but many compilers will get confused if they read the same header more than once. So, below the comment area, add the following:

```
#pragma once

#ifndef __ZHELLOWINDOW__
#define __ZHELLOWINDOW__




#endif
```

What we’ve done is provide two ways to make sure the header is only read once. The line #pragma once tells the compiler- read this only once! For CodeWarrior, that is enough actually. But, MacZoop is not required to use CodeWarrior, and other compilers (particularly older ones) may not understand this instruction. So, just to make sure, we also add a more traditional way to ensure that the file is parsed only once (parsed is a computer nerd’s way of saying ‘read’). The other lines say IF you’ve never seen the word “__ZHELLOWINDOW__” before, then here it is... if the file has been read before, then the computer will have seen it before, and will then skip to the #endif, skipping anything in between. It’s in between that we are going to put our interesting stuff. By the way, that’s two underscore characters on each end of ZHELLOWINDOW. Actually some compilers treat word defined like this with double underscores in the same way as the #pragma once, so we’ve actually covered ourselves three ways here- all good practice, and I recommend it’s a habit you get into early.

OK, now for the more interesting stuff. Remember, everything we add must go after the #define __ZHELLOWINDOW__, and before the final #endif.

Type:

```
#include    "ZWindow.h"

class ZHelloWindow      : public ZWindow
{
public:
      ZHelloWindow( ZCommander* aBoss, const short id );
      virtual void      DrawContent();
};
```

Let's look at this line by line.

**#include ZWindow.h**.

This makes this header bring in another one- the one for ZWindow. That header is needed because it defines ZWindow, and as we are basing our new window on that one, so we need its definition.

```
class ZHelloWindow : public ZWindow
```

This tells the compiler we're defining a new class called ZHelloWindow, and we want it to be based on the public interface of ZWindow.

```
{
public:
```

This tells the compiler that the methods and data members we define following this word are part of our own public interface. Actually this public and private stuff is not really very important right now, so don't worry about it for the moment, just copy it down.

```
ZHelloWindow( ZCommander* aBoss, const short id );
```

This looks familar- it's just a function prototype. However, because the function has the same name as the class, it is treated as a special function. In fact this is the CONSTRUCTOR of the class- it is the function that will be called automatically when the class is created. We'll see how this is useful later. Functions that are defined within a class like this are called the methods of the class. Here, we can see that this method takes two parameters- a ZCommander*, which is another MacZoop type, and a short. We don't need to worry about this for now, since for this tutorial, all of that is taken care of for you.

```
virtual void    DrawContent();
```

Ah, this is more interesting. This is another method of our class. It takes no parameters, but it is declared as a VIRTUAL method. This means that it can be overridden by a subclass if desired, but more importantly for us here, it means that we are overriding the similar method of ZWindow. This is

where we are going to do our special drawing!

OK, we're done here. Save the file to your project folder, and be careful to name it "ZHelloWindow.h". Don't be tempted to give it another name for fun- the naming is important as we'll see.

So we have our new class defined, but we still haven't written any code. Let's do that now.

• Open another new blank file.
• Cut and Paste a comment block to describe the file. Edit the comment as you wish.
• Type:

```
#include    "ZHelloWindow.h"
#include    "MacZoop.h"

ZHelloWindow::ZHelloWindow( ZCommander* aBoss, const short id )
      : ZWindow( aBoss, id )
{
}
```

The first two lines simply include a couple of header files- the one we just created, and the MacZoop.h header file, which defines all sorts of useful MacZoop stuff that we might want to use. In fact we probably don't need it in the tutorial, but again, this is a good habit to get into since most real-world files will need it.

The next bit is the body of the CONSTRUCTOR. In fact, we don't need it to do anything special, but it must call the constructor of ZWindow, which we are built on top of. That's what's happening here- ZWindow's constructor is called, and the <aBoss> and <id> parameters are simply passed on to it. Now the interesting bit.

```
void  ZHelloWindow::DrawContent()
{
      TextFont( kFontIDTimes );
      TextSize( 48 );
      TextFace( bold + italic );

      MoveTo( 10, 50 );
      DrawString( "\pHello World!" );
}
```

This is the body of the DrawContent method for our class. It sets up the text drawing parameters, moves the pen location to 10 pixels horizontally and 50 pixels vertically, then draws the text string "Hello World!". That's all the code we need to write. Because we are based on ZWindow, we don't need to worry about how to make the window appear, or how it should behave- all we need to do is concentrate on how our particular window is different- ours draws "Hello World!", and that's all

we've coded. Right, save the file, as before, to your project folder, and name it "ZHelloWindow.cpp". Again, be sure to name it correctly.

Now we need to add our new source file to the project.

• With the new file frontmost, choose "Add Window" from the Project menu. You should see it appear in the project window. You can drag it wherever you want to suit yourself.

If you now compile and run your project, you should find that your code is compiled OK. If not, now is the time to go over it again and FIX it. Is that all that is needed? No. If you run your application now, you still get a blank window. That's because MacZoop is told to make ZWindow objects as standard, and you haven't told it anything about ZHelloWindows yet. So, quit the running app and follow on...

*Project Settings*

Your ProjectSettings.h file is the place you tell MacZoop what sort of window to make. Since this isn't part of the project, you have to open it by either double-clicking it in the Finder, or using the Open File... command in the File menu of CodeWarrior. As a shortcut, you can add this file to your project. Since it's a header, it can sit in the project file quite happily and not really have any effect, but it gives you a convenient way to open the file whenever you need to.

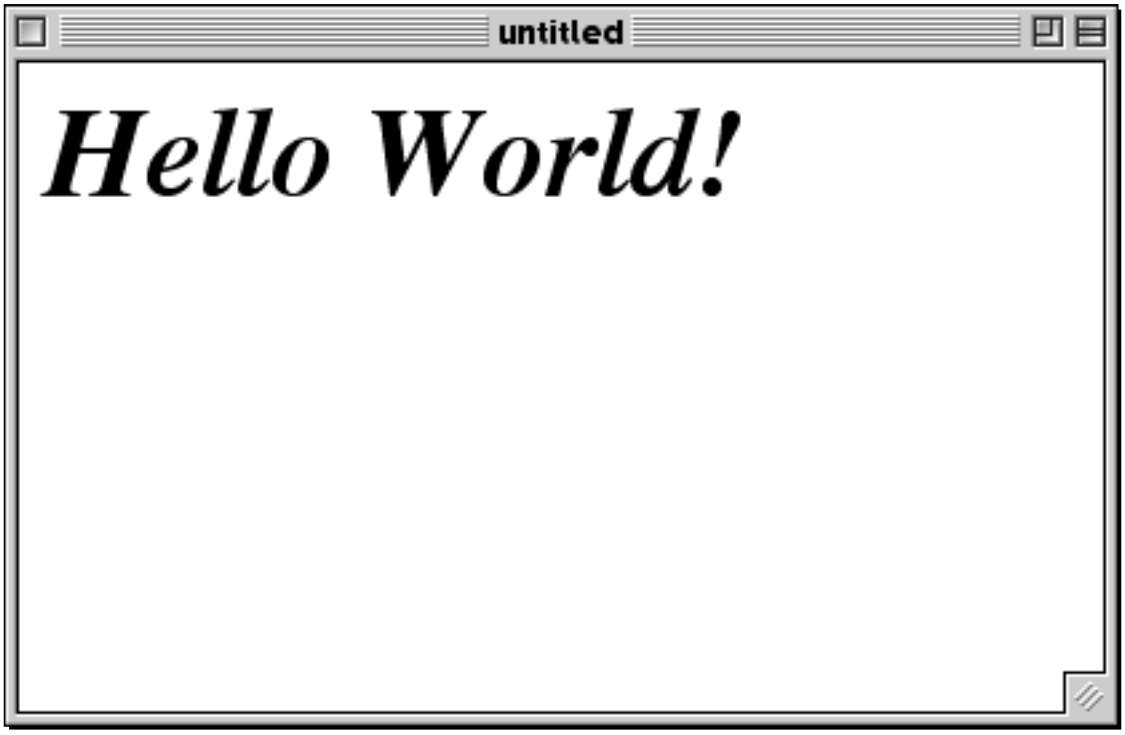
ProjectSettings.h contains a good number of tweakable parameters which affect how your app is built. What we are interested in here though is the window type to make. By default, no window type is specified. MacZoop knows to make generic ones in this case. Scroll down to about line 50. Actually, it doesn't matter where you add this, but the supplied file already has what you need on line 50, but commented out. It reads:

```
// #define        USER_DEFAULT_WINDOW_TYPE      ZMyFunkyWindow
```

change this to read:

```
#define           USER_DEFAULT_WINDOW_TYPE      ZHelloWindow
```

Note- we took away the comment slashes, and changed ZMyFunkyWindow to ZHelloWindow. When CodeWarrior encounters this line, it causes it to compile the code for ZApplication a little differently. It automatically #includes the file ZHelloWindow.h, and when instructed to do so, it makes a ZHelloWindow object instead of a ZWindow object. This is why the naming of the files and the class was so important- by putting that name here, MacZoop can find all the right files and make the right object. Save and close the file, then compile and run it again. This time, you should see:




Congratulations! You have successfully completed part one of the Hello World tutorial! Move the window partly offscreen, then on again- the text is refreshed. Open more windows using the New command- each one is a ZHelloWindow- we didn't just tell it how to create one window, but as many as we want of that type. As the various windows are moved around, you'll see that they refresh properly, grow shrink, close and generally behave as windows should. Yet the only code we wrote was to draw the text! This will, I hope, give you a glimpse of the power of using a framework such as MacZoop- all that donkey work is done for you.

*How it works*

When a window needs to be drawn, the Mac toolbox detects this and generates an update event. MacZoop retrieves the event and determines which window object is involved. It then asks the window object to redraw itself. The basic ZWindow takes care of the housekeeping tasks connected with handling an update, but then calls the DrawContent() method. Because this is a virtual method, the actual code that is called at runtime can be different for different windows, so although the code up to that point is shared with all windows, for our particular window is different from standard. Thus our code is called, and we draw the text. Note that when we wrote this code, we were not concerned with how and when the code is called- we can just trust that it will be. This is another new thing that programming with frameworks introduces- you generally write small fragments of code, and insert them where you know the framework will call them. You have to trust the framework will do the right thing, and in return, it trusts your code to do the right thing. It's all about mutual cooperation!

***Part Two, Introducing Views***

In this section, which is new for 2.5, you will learn how to harness the power of views to create far more powerful functionality with relatively little extra effort.

*What are views?*

In part one, we created a window and displayed some text in it. A window is itself a type of view - an area on the screen that we can draw graphics in and (though not really touched on so far) can respond to mouse clicks. In most real applications, we don't want to use a whole window for each separate graphic or user-interface element that we'd like to use. It would be absurd to open a whole new window for each text-entry field of a dialog, for example. So a view is a way to divide up the space within a window into separate logical areas, and, as we are about to see, to allow us to have a drawing area larger than the actual size of the window. A window is a view, but it can also contain views, which leads to an important point - views are hierarchical in nature, and every view can contain none, one or any number of further "child" views. It is not the aim of the tutorial to go into depth about views, but to simply show you by way of an example how they are used. Please read the views chapter for an in-depth discussion of views.

*Scrolling*

Scrolling is a way to provide a larger drawing area than can be displayed in a window. In fact the drawing area available is far larger than any window could be, due to the practical limitations of monitors. By attaching scrollbars, the user can move any part of a large drawing into view within the window. The view class in MacZoop that provides this basic functionality is called ZScrollView. We are going to create one of these objects and put it to work in our window. This will give us the basic ability to scroll. Then we are going to create a new view that displays our Hello World text and place this inside the scrollview. The scroll view will then allow us to scroll our text around. The key thing to understand as we go through this is that the view that displays the text does not need to know anything about how scrolling works - it just does. In this way, you can create displays that adopt complex behaviours without very much programming - in other words, as usual your effort is spent on the unique content of the application, and not on handling all the standard stuff.

*To the code...*

So, first thing to do is to modify our ZHelloWindow class so that we can create and add the scrollview to the window. Open the header file and change the class definition to this:

```
class ZHelloWindow      : public ZWindow
{
public:
      ZHelloWindow( ZCommander* aBoss, const short id );
      virtual void      InitZWindow();
};
```

We've removed DrawContent and added InitZWindow. We no longer need to override DrawContent

because once we're done, our special view class will be doing the drawing, not the window. We do wish to override InitZWindow however, since we need to create some extra objects as part of the window's initialisation.

Save the header file and open the .cpp file. Delete the DrawContent method altogether (leave the constructor as it was though) and add the following to the top of the file, just below the existing #include:

```
#include    "MacZoop.h"
#include    "ZScrollView.h"
```

And now add the method InitZWindow:

```
void  ZHelloWindow::InitZWindow()
{
      ZWindow::InitZWindow();

      ZScrollView*      myScrollView;

      FailNIL( myScrollView = new ZScrollView( this, NULL, TRUE, TRUE ));

      myScrollView->FitToParent();
      myScrollView->SetAutoSizing( kStdScrollViewSizing );
      myScrollView->SetBounds( 500, 500 );
}
```

Let's go through this and see what's happening, line by line.

```
      ZWindow::InitZWindow();
```

This calls the inherited InitZWindow method which is responsible for doing all of the standard window initialisation. You don't need to know what it does in detail, but it is vital that it is called before we do anything else, otherwise there won't really be a window to do anything with.

```
      ZScrollView*    myScrollView;
```

This just declares a temporary variable called "myScrollView" which is a pointer to a ZScrollView object.

```
      FailNIL( myScrollView = new ZScrollView( this, NULL,
                                                TRUE, TRUE ));
```

This line actually creates the scrollview object. The FailNIL just makes sure that if things go wrong, the error is handled and reported cleanly. The parameters that we pass to the constructor of ZScrollView are important.

The first parameter specifies which view is the “parent” view of the view we are creating - in this case, the window itself, so we pass <this>. You must always pass a valid parent to every view you create.
The second parameter, which we pass as NULL, is actually a pointer to a rectangle specifying the size and position of the view within the parent view. Since we want to position this afterwards, we can just pass NULL which initially sets the size and position to 0, 0, 0, 0.
The third and forth parameters are booleans which tell the scrollview which scrollbars it should create. In this case we want both horizontal and vertical bars, so we pass TRUE for both. This line creates the object and assigns it to the “myScrollView” variable.

```
myScrollView->FitToParent();
```

This line is a quick way to make the newly created view automatically position itself and size itself to fit exactly within the window, whatever its current size. We could have obtained the rectangle of the window and passed it in the constructor, but this way is easier. After this call, the scrollview will occupy the whole of the window content area.

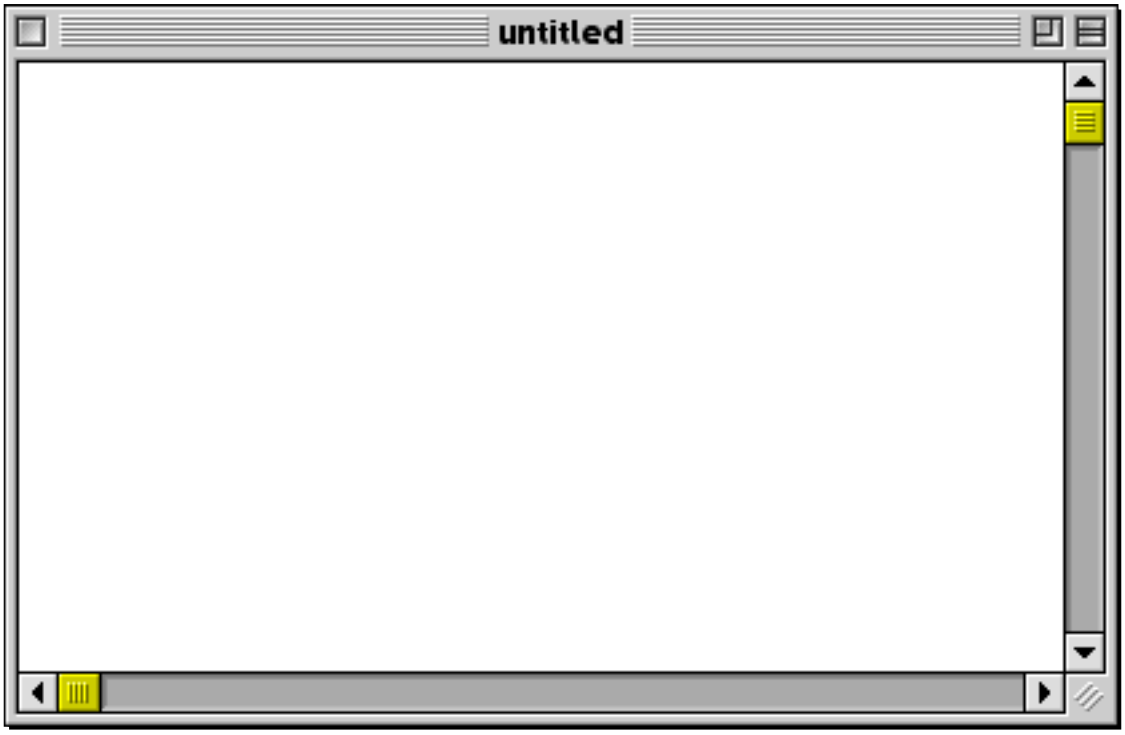
```
myScrollView->SetAutoSizing( kStdScrollViewSizing );
```

Our window is resizeable, and we would like our scroller to follow the size of the window such that the scrollbars remain correctly fixed to the right and bottom edges of the window. This line sets that up. Passing a constant of <kStdScrollViewSizing> informs the view that when its parent changes size, the top and left sides remain where they are, but the bottom and right edges move with the window. Many other combinations are possible to set up, but we are not interested in those for now.

```
myScrollView->SetBounds( 500, 500 );
```

The actual scrollable area of the view is called its bounds rect, and this may be larger than the view itself (the actual visible size of the view is called its frame). This line sets the bounds to 500 pixels on each side, which will allow us to scroll the view as long as the window is smaller than this.

The final step is to add ZScrollView.cpp to your project, recompile and run. We haven’t created our custom content yet, but it’s worth testing at this point to see just what we have achieved. You should see:




We have added fully operational scrollbars. Clicking on them will show that they apparently work, and resizing the window should work as you would expect. If you're feeling inquisitive, you can try commenting out the <SetAutoSizing> line above and see what happens when you enlarge the window.

*OK, onwards...*

Now for our custom content. We are going to create another view object, but this time we'll override its drawing method to draw our own content. By placing it inside the scrollview as a child, it will automatically scroll. The thing to note here is that as we code this simple view class, at no point do we need to care about how scrolling works.

Create a new header file and add this content:

```
#pragma once

#ifndef __ZHELLOVIEW__
#define __ZHELLOVIEW__

#include    "ZView.h"

class ZHelloView  : public ZView
{
public:
      ZHelloView( ZView* aParent, Rect* aFrame );
      virtual void      DrawContent();
};

#endif
```

This is very similar to our original ZHelloWindow, but now we are defining a ZHelloView, based on ZView. As before, we declare a constructor and a DrawContent method. Save this as ZHelloView.h. Create another new file and add this content, then save as ZHelloView.cpp:

```
#include    "ZHelloView.h"
#include    "MacZoop.h"

ZHelloView::ZHelloView( ZView* aParent, Rect* aFrame )
      : ZView( aParent, aFrame )
{
}

void  ZHelloView::DrawContent()
{
      TextFont( kFontIDTimes );
      TextSize( 48 );
      TextFace( bold + italic );
      MoveTo( 10, 50 );
      DrawString( "\pHello World!" );
}
```

Look familar? It should - it's virtually identical to the code for part one. Certainly the actual drawing code is identical. Add this file to your project. Next, we must create and add a view of this class to our window, so go back to ZHelloWindow.cpp, InitZWindow, and add this code to the end of the method:

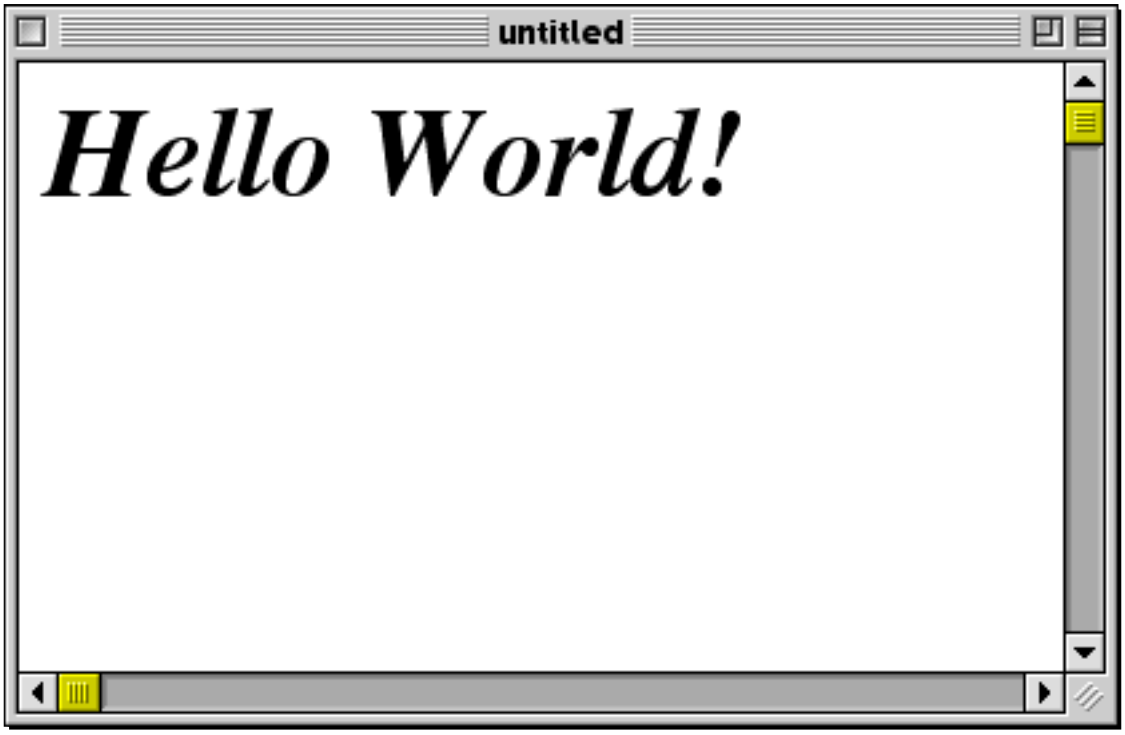
```
      Rect              r;
      ZHelloView*       myHelloView;

      SetRect( &r, 0, 0, 500, 500 );
      FailNIL( myHelloView = new ZHelloView( myScrollView, &r ));
```

Here we declare two variables, one a simple rectangle, the other a pointer to an object of type ZHelloView. We then set the rectangle to a reasonable size - here we match it to the scroll bounds, but it could be smaller - doing this is just convenient. Finally, we actually create the ZHelloView object. Note one very important thing - its parent is "myScrollView", not the window (i.e. "this"). All that remains is to add:

```
#include "ZHelloView.h"
```

to the list of includes at the top of the file, save, and run. You should see:




Well done, it's working! Now try scrolling the window - you'll see that it works - the text is redrawn correctly as it scrolls into view. Resize the window so that it is small - the text is correctly clipped. Dragging the scrollbar thumbs will live scroll the view as well. That's it! Scrolling for free...

One more thing to observe is that when we made the QuickDraw statements that actually drew this text we wrote:

```
MoveTo( 10, 50 );
```

And yet the text is actually being drawn in all sorts of different places relative to the window as we scroll it around. This shows another great advantage of views - each view effectively has its own private coordinate system, so you can always draw your content as if it were fixed in space. If it happens to be part of a scroller, well, MacZoop will deal with that. This makes creating content within a view an absolute doddle.

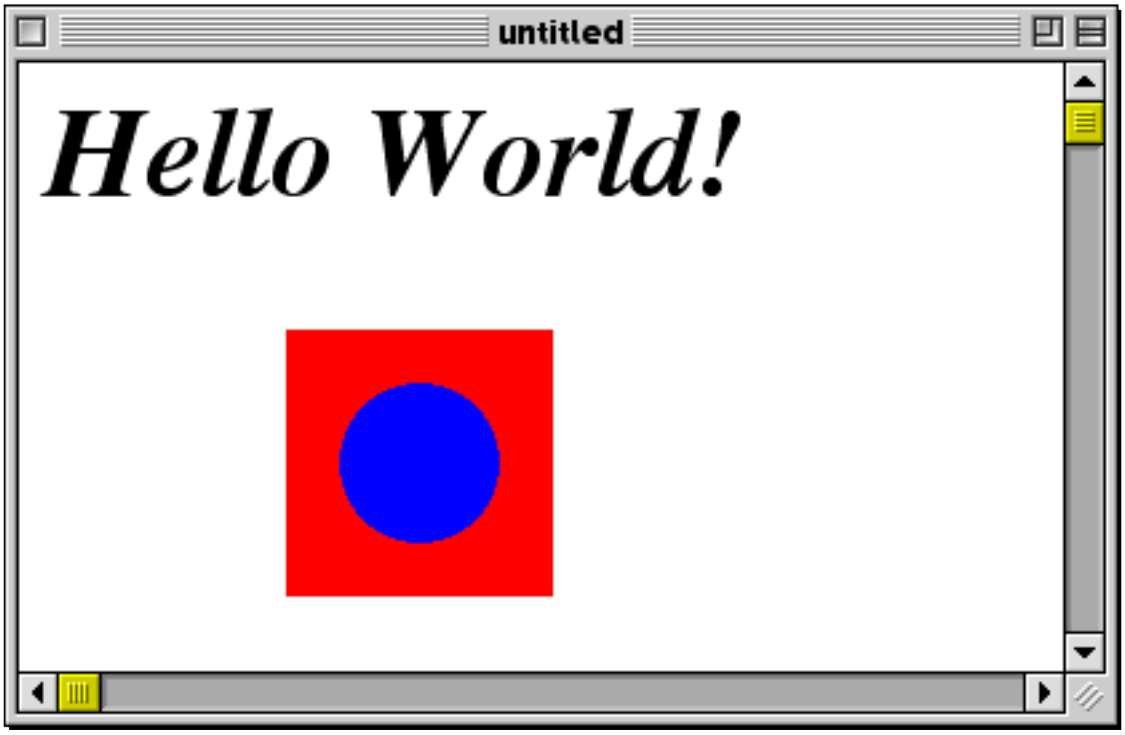
Just for fun, let's add a few more graphics to this view before moving on to printing.

Add this to ZHelloView::DrawContent:

```
Rect temp;

SetRect( &temp, 100, 100, 200, 200 );
RGBForeColor( &gRed );
PaintRect( &temp );
InsetRect( &temp, 20, 20 );
RGBForeColor( &gBlue );
PaintOval( &temp );
```

Run the application again, and you should see:




Feel free to experiment by adding other QuickDraw graphics calls as you wish - no matter what you do, you should find that the view will draw and scroll correctly.

*Printing.*

Finally, let's enable printing so that the contents of the window can be printed. This part shows you that supporting printing is straightforward, and again how views make life easier.

The first thing we need to do is make sure that the code that handles the actual printing is part of the project and available for use. Open your ProjectSettings.h file and set the following:

#define PRINTING_ON ON

In my copy this is on line 128 or thereabouts. You then must add ZPrinter.cpp to your project. This is found in :More Classes:Printing

Each individual window can be set to print or not, and by default, windows are set not to, so the next step is to add a little code to enable this for the window. Also, because a window may contain a number of views, you need to tell it which view is the one to print when the user requests the Print command. In this case, naturally we want to print our ZHelloView. So, add this to the end of ZHelloWindow::InitZWindow:

```
SetPrintable( TRUE );
SetPrintView( myHelloView );
```

That’s all there is to it. Compile, Run and print that window!

***Where to go from here***

If you haven't already done so, it's worth getting the demo project to compile and run, since it will give you confidence that nearly every part of MacZoop is working. If you're happy, then you are ready to use MacZoop in anger. The rest of this manual discusses the most important classes and what they do, and every class is described in the Class Reference towards the back half of the manual.

Personally, I find that the best way to learn something like a framework is to be goal-oriented. Decide what you want to create, then ask yourself "what do I need to get that done?". Read up on the classes you think will help you, and don't be afraid to experiment. If you get stuck, try something else. Don't be too ambitious- make each goal quite small and achieveable to avoid frustration, as you learn new things and the concepts start to sink in, new goals and possibilities will open up. Be prepared to write off your first couple of efforts as learning experiences- invariably as you learn the framework, more efficient ways to do things will become apparent that we not so obvious earlier on. In fact, programming with MacZoop should be easy- after a while. If you find yourself having to program your way in a very long-winded fashion around something, you're probably doing it wrong- but don't worry, it's all good stuff for learning. Good Luck!