

**author** David Aaron (bobjob) Muhar

Hey guys and welcome to my first LWJGL/OpenGL tutorial.

In this lesson we will be setting up a basic window to display an OpenGL context (so basically just making an empty window)

### **class Lesson01**

```
import org.lwjgl.Sys;
import org.lwjgl.input.Keyboard;
import org.lwjgl.input.Mouse;
import org.lwjgl.opengl.Display;
import org.lwjgl.opengl.DisplayMode;
import org.lwjgl.opengl.GL11;

public class Lesson01 {

    final static String TITLE =
        "Lesson 01 - Setting Up The LWJGL Display Window";
    final static int FRAME_RATE = 60;
    static int width = 800, height = 600;
    static boolean running = true;
    static boolean fullscreen = false;
```

each one of the lessons to follow will have a different titles that will be display on the top of the window

frame rate is not really used in this lesson this will be explained in the next lesson.

width and height are for the window size, or rather the resolution of for our OpenGL context frame.

running will be our main condition to terminate the program.

for the moment all lessons will be run in window so set the fullscreen to false.

### **start()**

```
public static void start() {
    try {
        initDisplay();
        initGL();
        loadResources();
        run();
        cleanup();
    } catch (Exception e) {
        e.printStackTrace();
        Sys.alert(TITLE, "Error 101: " + e.toString());
    }
}
```

start() will call our program template functions that we will be working from for all the lessons to follow.

start will basically be the same in all lessons each function called within start() will be explained in more detail later before there implementation.

InitDisplay() will setup.lwjgl display.  
initGL() will setup OpenGL settings.  
loadResources() will load program resources.  
run() will run main program loop.  
cleanup() will unload program resources.

dont worry to much about the error output at least not for the early lessons.

### initDisplay()

```
static void initDisplay() throws Exception {
    Display.setTitle(TITLE);
    Display.setFullscreen(false);

    DisplayMode displayMode = null;
    DisplayMode d[] = Display.getAvailableDisplayModes();

    for (int i = d.length - 1; i >= 0; i--) {
        displayMode = d[i];
        if (d[i].getWidth() == width &&
            d[i].getHeight() == height &&
            d[i].getFrequency() == FRAME_RATE &&

                (d[i].getBitsPerPixel() >= 24 && d[i].getBitsPerPixel() <= 32))

            break;
    }
    Display.setDisplayMode(displayMode);
    Display.setVSyncEnabled(fullscreen);

    Display.create();
    Display.update();
}
```

initDisplay() will set our desired screen settings. first we set out title for this lesson.

for now we will be running lessons in a window as problems are easier to debug at runtime.

Basically we get an array of DisplayModes for:

- \* resolution
- \* bit per pixel
- \* frequency

In window mode the frequency will be the same as the desktop

cycle through the available display modes  
if the desired display mode is not available  
another mode should be selected

the 24bpp is included for Linux bitsPerPixel  
the 32bpp is included for other OS bitPerPixel

if our display mode is compatibly then break; so  
we can use it

we then set the current display mode,  
aswell as vSync.  
vSyncn can be enabled in both window mode.  
but in these tutorials we will be using a capped frame rate.  
vSync can can conflict with monitor refresh rates that are  
over the desired frame rate.  
this is included if you desire to experiment  
with the fullscreen/boolean condition.  
This function if true enables v-sync if possible.

After the display mode is chosen and before rendering  
anything we must first create an OpenGL context for rendering  
with create();

The call to update() is not needed here, but I have found it has helped me  
with a few issues in the past.

#### **initGL()**

```
static void initGL() {  
    GL11.glClearColor(0f, 0f, 1f, 0f);  
}
```

initGL() will setup the OpenGL context just the way we like

*glClearColor() will make a call to OpenGL telling  
it what color should be set when the current color  
is cleared (in this program it will be used to set  
the color of the openGL context frame)*

we will be using color measued in float therefore  
for our lessons color channels will range from 0 to 1.

Usual opengl color layouts are as follows:

(float red, float green, float blue, float alpha)

so our OpenGL context should end up looking blue  
yay! our first OpenGL function!

#### **loadResources ()**

```
static void loadResources() {  
}
```

loadResources() will load all initial application resources.  
For this first lesson we have no resources to load  
but I wanted to include this function as we will be  
building on this template in future lessons.

## run()

```
static void run() {
    while (running) {
        if (Display.isCloseRequested()) running = false;
        Display.sync(FRAME_RATE);

        checkInput();
        logic();

        if (Display.isActive() && Display.isVisible())
            render();

        Display.update();
    }
}
```

run() will be our main program loop

using the default escape operation will cause this program to close  
sync() will cap our program to a healthy 60 frames per second.  
We then check for user interactions, before running our core logic program logic.

We will only want to render if the Display is visible or active. You may change this, but I have had "blue screens of death" when rendering two OpenGL contexts at the same time.

As far as OpenGL is concerned the update() function swaps the back buffer, or rather brings what has been rendered to the screen.

## CheckInput()

```
static void checkInput() {
    int mouseX = Mouse.getX();
    int mouseY = Mouse.getY();

    while (Mouse.next()) {
        int mouseButton = Mouse.getEventButton();
        if (Mouse.getEventButtonState()) {
            switch (mouseButton) {
                case 0: {
                    System.out.println("Mouse Down");
                    System.out.println("" + mouseX + " " + mouseY);
                    break;
                }
                case 1: {
                    System.out.println("Mouse Down");
                    System.out.println("" + mouseX + " " + mouseY);
                    break;
                }
            }
        }
    }
}
```

```

        case 2: {
            System.out.println("Mouse Down");
            System.out.println("" + mouseX + " " + mouseY);
            break;
        }
    }
} else {
    switch (mouseButton) {
        case 0: {
            System.out.println("Mouse Up");
            System.out.println("" + mouseX + " " + mouseY);
            break;
        }
        case 1: {
            System.out.println("Mouse Up");
            System.out.println("" + mouseX + " " + mouseY);
            break;
        }
        case 2: {
            System.out.println("Mouse Up");
            System.out.println("" + mouseX + " " + mouseY);
            break;
        }
    }
}

while (Keyboard.next()) {
    if (Keyboard.getEventKeyState()) {
        int key = Keyboard.getEventKey();

        switch (key) {
            case Keyboard.KEY_SPACE: {
                System.out.println("Space Bar");
                break;
            }
            case Keyboard.KEY_RETURN: {
                System.out.println("Return");
                break;
            }
        }
    }
}
}

```

checkInput() will check for any keyboard/mouse input

get our mouse co-ordinates before cycling through all mouse events. if the mouse is down getEvenButtonState() will return true. for this lesson we will just output the mouse coordinates to the console.

After the mouse check we will go through all remaining keyboard events, again nothing great, just to know that our input is working correctly we will output some strings to the console.

### **logic()**

```
static void logic() {  
}
```

logic() will really be the non graphical engine of our program.

there is no logic to be done in this first example but there is plenty to come so for the sake of a clean template I have included logic something to note about logic, any process not related to OpenGL drawing should be done in a separate function(logic) if we later decide to load dynamic resources or render at un-capped frame rates, we will need to still cap the application logic. Thats if we want the application to seem fluid. This can be achieved by moving logic into another Thread. If this doesn't make sense to you, thats not important as its not really relevant to anything you will be doing in these lessons.

### **render()**

```
static void render() {  
    GL11.glClearColor(GL11.GL_COLOR_BUFFER_BIT);  
}
```

render() will basically be doing our OpenGL rendering lets clear the screen using the OpenGL function `glClearColor()`, as we set our default clear color to blue in the `initGL()` function, when the openGL context is next updated it should be blue. And this would be our second openGL function!!!

### **cleanup()**

```
static void cleanup() {  
    Display.destroy();  
    System.out.println "[" + TITLE + "] has ended...";  
    System.exit(0);  
}
```

cleanup() will remove any unused resources.

In later lessons Ill be explaining how to free up OpenGL resources using this function.

### **main()**

```
public static void main(String args[]) {  
    Lesson01.start();  
}
```

Lets begin running our first lesson.

There you have it, your first LWJGL lesson. You should now know how to setup a window using LWJGL. Its best **NOT** to try and tweek any of this code, as this will be the template for later lessons. Any changes that need to be made will be done to this template. Instead of writing down all the code again, I will just make references to the functions name and then display the changes needed.

This Lesson is from Bobjob's LWJGL Lessons  
David Aaron (bobjob) Muhar (<http://users.on.net/~bobjob/>)

Im Currently involved in the JavaGaming.org community forums:  
(<http://www.javagaming.org/>)

So if you have any question or need help with Java/LWJGL check it out, and maybe I'll be there to help you =D