**Writing regular Programs for Windows or Mac Computers**

To wrap up this course, here is a standard program written in Java that can execute on your regular computer. Java can be used to not only create Robot programs but also ‘standard’ computer programs as well as Apps for Android phones.

The particular program we want to create is a game reminiscent of the classic game “Space Invaders”. This program is fairly involved and uses a number of techniques that we did not need in our robot programs. However, the overall structure and language should look familiar, as should the general process of creating such a program.

We will start by selecting “File | New … | Java Project …”. Decide on a project name, such as “SpaceInvaders” and hit *Finish*. In that project we need to create several classes. Let’s start with the easiest one: the main class:

/\*

\* Main class: pretty much its only purpose is to contain the main method

\* that gets everything started. In this case, creates an ÏnvadersWindow

\* which contains all GUI elements.

\*/

**public** **class** InvadersMain

{

**public** **static** **void** main(String[] args)

{

InvadersWindow window = **new** InvadersWindow();

window.setTitle("Space Invaders 1.0");

window.setSize(600, 400);

window.setVisible(**true**);

}

}

That’s it, it is the entire class. Its purpose is to get the process of running a program going so the class contains only a main method, which in turn creates the main “windows” class. Note that in Java standard programs are “JFrames”.

Here is the listing for the second class in this project, the InvaderWindows class, as listed below:

**import** java.awt.BorderLayout;

**import** javax.swing.JFrame;

/\*

\* Represents the main window as well as methods to control the game. Acts as

\* 'switch' to pass requests from one item along to another.

\*/

**public** **class** InvadersWindow **extends** JFrame

{

// This window "has" an arena (where the game action happens) and a controller

// (containing the elements that control the program)

Arena arena = **null**;

Controller controller = **null**;

// Constructor method (no return type, same name as class). Called automatically

// when a new instance is created. Usually responsible for layout and setting up

**public** InvadersWindow()

{

arena = **new** Arena();

controller = **new** Controller(**this**);

setLayout(**new** BorderLayout());

add("Center", arena);

add("South", controller);

}

// Calls on the arena's start method to start a game

**public** **void** start()

{

arena.start();

}

}

Note that this class uses a layout manager to define where its various parts go. That is how elements are usually arranged. The most common layout managers are FlowLayout (which arranges all items in a row), BorderLayout (which has 5 parts named North, South, East, West, and Center), add GridLayout (which lays out its parts in a table where all cells have the same size). Working with layout managers takes a bit to get used to but once you are you will appreciate their flexibility and in particular their independence of screen size and resolution.

Next is the Arena class. This is the class where all the action happens and where the spaceships, gun, and bullets will be drawn. This class also contains a thread which will run and move the various elements on is own, independent from the user action.

Note that the thread will provide clock ‘ticks’ that can be used to move different elements at different speeds. Right now we just have a single spaceship, represented by a green rectangle that will move across the screen. Later we add a gun and bullets as well, all powered by the clock tick of the thread.

**import** java.awt.Color;

**import** java.awt.Graphics;

**import** java.awt.Image;

**import** javax.swing.ImageIcon;

**import** javax.swing.JPanel;

// contains the drawing of the various elements of the game, such as the

/\*

\* The Arena is where the main action happens. It is responsible for all graphics

\* and sounds, and runs the secondary thread to move the aliens, control the shots,

\* etc.

\*/

**public** **class** Arena **extends** JPanel **implements** Runnable

{

// The Thread that will act as secondary thread to control aliens and shots

Thread thread = **null**;

// (x,y) coordinates of the single 'alien'

**int** xUFO = 10;

**int** yUFO = 10;

// height, width of the single 'alien'

**int** wUFO = 40;

**int** hUFO = 40;

// direction of the alien ufo (1 moves left to right, -1 move right to left)

**int** dirUFO = 1;

// speed of the alien ufo, in ‘clock ticks’

**int** speedUFO = 20;

// speed of the thread, i.e. the alien movement by providing ‘clock ticks’

**int** sleepTime = 5;

**long** tick = 0L;

// Constructor. Is called automatically whenever a new Arena is created. Includes

// setup instructions and other initializations (we’ll add those later).

**public** Arena()

{

}

// Starts a new thread if no thread is currently active. The thread will execute

// the 'run' method automatically when started because of the **this** use.

**public** **void** start()

{

**if** (thread == **null**)

{

thread = **new** Thread(**this**);

thread.start();

}

}

// Stops the current thread from running. Can be restarted by calling 'start'

**public** **void** stop()

{

thread = **null**;

}

// Run method controls the game by moving the alien space ships, bullets, etc.

// Also needs to contain code checking for any hits, update the score, etc.

**public** **void** run()

{

**while** (thread != **null**)

{

**if** ((tick % speedUFO) == 0)

{

**if** (xUFO >= getWidth()-wUFO)

{

dirUFO = -1;

yUFO = yUFO + hUFO;

xUFO = getWidth() - wUFO - 1;

}

**else** **if** (xUFO < 0)

{

dirUFO = 1;

yUFO = yUFO + hUFO;

xUFO = 1;

}

xUFO = xUFO + dirUFO\*wUFO/2;

}

**try**

{

tick = tick + 1;

thread.*sleep*(sleepTime);

}

**catch** (InterruptedException e)

{

e.printStackTrace();

}

repaint();

}

}

// Draws everything, using variables for their locations. If the variables

// change, the corresponding object is drawn at its new location. Standard method

// for all Java program containing some custom drawing.

**public** **void** paintComponent(Graphics g)

{

// the following line must be the first line in this method

**super**.paintComponent(g);

g.setColor(Color.***green***);

g.fill3DRect(xUFO, yUFO, wUFO, wUFO, **true**);

}

}

The last class is the Contoller. It contains standard components such as buttons, labels, dropdown items, inpt fields, etc. that are used to control the game. In our case we start with a “Start” and “Quit” buttons as well as a field to display the current score. Note that the way the buttons work is standard for any Java program: the class implements ActionListener and therefore contains an ‘actionPerformed’ method, each button is defined in a field, and in the constructor the buttons are layed out via a layout manager and activated by calling their ‘addActionListener(this)’ method. Then an if-else-if statement in actionPerformed decides which button was clicked and what should happen.

**import** java.awt.FlowLayout;

**import** java.awt.event.ActionEvent;

**import** java.awt.event.ActionListener;

**import** javax.swing.JButton;

**import** javax.swing.JLabel;

**import** javax.swing.JPanel;

**public** **class** Controller **extends** JPanel **implements** ActionListener

{

InvadersWindow parent;

JButton start = **new** JButton("Start");

JButton quit = **new** JButton("Quit");

JLabel score = **new** JLabel("Score: 0");

**public** Controller(InvadersWindow \_parent)

{

parent = \_parent;

setLayout(**new** FlowLayout());

add(quit);

add(score);

add(start);

quit.addActionListener(**this**);

start.addActionListener(**this**);

}

**public** **void** updateScore(**int** newScore)

{

score.setText("Score: " + newScore);

}

@Override

**public** **void** actionPerformed(ActionEvent e)

{

**if** (e.getSource() == quit)

System.*exit*(0);

**else** **if** (e.getSource() == start)

parent.start();

}

}