

Week 02 (Programming Workshop)

Object Oriented Programming of a Simple SFML Application and Application States

Module Leader: Dr. Sud Sudirman

James Parsons Building, Byrom Street

Room BS/7.27

s.sudirman@ljmu.ac.uk

|  |
| --- |
| Liverpool John Moores University |

|  |
| --- |
| Contents  [Recap on Object Oriented Programming 3](#_Toc472270520)  [Structuring Your Graphics Application 9](#_Toc472270521)  [Application States 16](#_Toc472270522)  [Additional Notes 21](#_Toc472270523) |

# Recap on Object Oriented Programming

**Introduction**

This section will provide you with a quick recap on Object Oriented Programming and introduce you to the concept of inheritance. We are going to do this by creating a console (text-based) application that simulates fights between two military units. You can download an example of the application that you are making from the Blackboard.

**Expected Learning Outcome**

1. Understand basic Object Oriented Programming concepts
2. Understand the concept of Inheritance

**Pre-requisites**

In order to complete this section, you will need the following programming skills

1. Know how to start Visual Studio and create a new empty project.
2. Understand the concept of programming project, building a programming project, including header files, and linking libraries.
3. Understand basic C++ programming and using standard (std) library

**Creating the base soldier class**

1. Run Visual Studio and create a new **Empty Console Win32** project. If you are unsure about how to do this ask your tutor.
2. Add a new class to the project by right-clicking on the project name in the Solution Explorer and select Add » Class. Select C++ Class from the list of template classes and press Add.
3. In the next window, enter **soldier** in the Class name textbox. Press the Finish button.
4. Two files should have been created for you namely **soldier.h** and **soldier.cpp.** Open the header file and include iostream header.

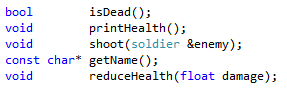


1. We are going to modify the soldier class’ declaration. First we change its default constructor to take a const char input:



We are going to leave the class’ destructor as it is.

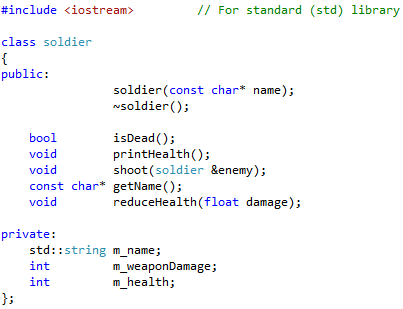
1. Next add the following **public** member functions:



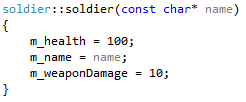
1. And add the following **private** member attributes:



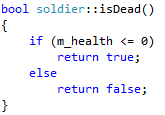
1. Compare your soldier class’ declaration to the following code to check if you have done it correctly or not.



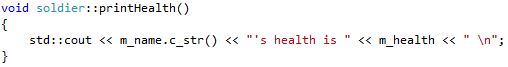
1. Now, open soldier.cpp. We are going to define the class’ member functions here. Let’s start with the class’ constructor. We need to replace the definition of the class’ constructor with the following:



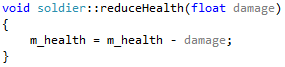
1. We will leave the destructor function as it is.
2. Now we define the isDead() member function.



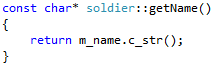
1. The printHealth() member function simply prints out the value of m\_name and m\_health members



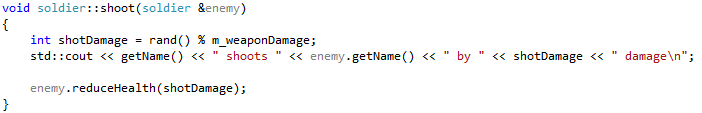
1. The reduceHealth() member function takes a number (float) and subtract it from m\_health



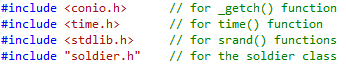
1. The getName() member function simply returns the name of the object.



1. The shoot() member function takes a reference to another soldier object, calculates a damage, prints a message and subtract the damage from the other soldier object’s health.



1. Add a new cpp file called main.cpp to the project.
2. Open main.cpp, include the following header files:



1. Create a new function called main()





1. Inside the main function, use time() to initialise the random seed. This will make any calls to rand() function to return different value each time the program runs.



1. Declare our soldier objects



1. Declare a Boolean variable to control our game loop



1. Create a do-while loop





1. Inside this loop, show the objects’ health



1. And pause until a key is pressed



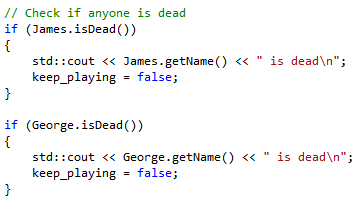
1. Get the two soldier object to shoot at each other



1. And pause until a key is pressed



1. Check if anyone is dead and print the appropriate message



The above concludes the do-while loop.

1. Just after the loop add a line to pause until a key is pressed



1. Build and run the program

**Deriving the grenadier class from the soldier class**

1. Add a new class to the project by right-clicking on the project name in the Solution Explorer and select Add » Class. Select C++ Class from the list of template classes and press Add. This should open a Generic C++ Class Wizard window.
2. In that window, enter **grenadier** in the Class name textbox. We will derive this new class publicly from one of SFML classes called NonCopyable. To do so, add **soldier** in the Base class textbox. Make sure the Access textbox is set to **public**. Press the Finish button.
3. Open grenadier.h. Change the constructor of the grenadier class to:



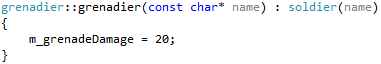
1. Add a **public** member function called throw\_grenade() that takes a reference to the soldier object.



1. Add a **private** member attribute

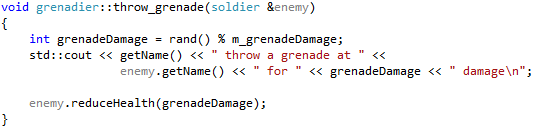


1. Open grenadier.cpp. Inside this class implement the class constructor:



Note that the grenadier constructor initialises the base constructor with the same input parameter. This will automatically initialise the class members.

1. Implement the throw\_grenade() member function as follows.



1. Open main.cpp and include the grenadier.h



1. Change the class type of George from soldier to grenadier



1. Change the action type of George from shoot() to throw\_grenade()



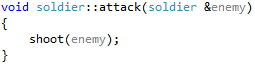
1. Build and run the program

**Class Specialisation by member function overloading**

1. In soldier.h, change the access classifier of the shoot() member function from public to private.
2. Declare a new public member function called attack()



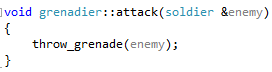
1. The definition of the function is to call the shoot() function



1. In grenadier.h,change the access classifier of the throw\_grenade() member function to private.
2. Declare a new public member function called attack()



When a grenadier object calls the attack() member function, it will execute this copy of the function instead of the base class’. The definition of this attack() function of the grenade class should call the throw\_grenade() function.



1. In main.cpp, we want to make a couple of changes. Instead of calling the shoot() function and throw\_grenade() function we instead call the attack() function.



1. Build and run the program.

# Structuring Your Graphics Application

**Introduction**

C++ is an Object Oriented Programming Language. The C++ code base in SFML makes full use of the Object Oriented feature of the C++ programming language. It is therefore important to structure your graphics application accordingly. In this workshop, you will create a class that will encapsulate all processes in the application. Justifiably we will call this class SFML\_Application.

In order to start this workshop, you need to have completed Workshop 01 from last week.

**Expected Learning Outcome**

1. Apply basic concepts of Object Oriented (OO) Programming in creating SFML application
2. Use OO Approach in structuring an application
3. Debug an OO source code.

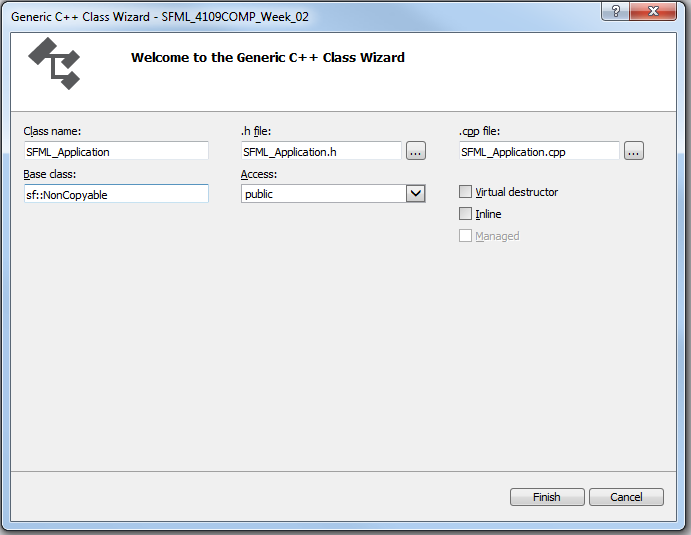
**Pre-requisites**

In order to complete this section, you will need the following programming skills

1. Know how to start Visual Studio
2. Understand the concept of programming project, building a programming project, including header files, and linking libraries.
3. Understand basic C++ programming

**Creating SFML\_Application class**

1. Using the steps detailed last week, create a duplicate of the solution from last week workshop and rename the project from **SFML\_4109COMP\_Week\_01** to **SFML\_4109COMP\_Week\_02**.
2. Open the new solution with Visual Studio.
3. Add a new class to the project by right-clicking on the project name in the Solution Explorer and select Add » Class. Select C++ Class from the list of template classes and press Add. This should open a Generic C++ Class Wizard window.
4. In that window, enter **SFML\_Application** in the Class name textbox. We will derive this new class publicly from one of SFML classes called NonCopyable. To do so, add **sf::NonCopyable** in the Base class textbox. Make sure the Access textbox is set to public. Press the Finish button.



1. The process should create two new files, namely **SFML\_Application.h** and **SFML\_Application.cpp**. Open the header file by double clicking its name in the Solution Explorer if the file is not opened already.

**Side Note:** There may be a bug in the auto class creation process that repeats the keyword “public” twice when deriving the class in SFML\_Application.h. If that occurs simply delete one of the word.

1. Still inside that header file, replace the include file from **“…\NonCopyable.hpp”** to **<SFML/Graphics.hpp>**.



When you create a class using the Generic C++ Class Wizard, the class declaration will already include default constructor and destructor member functions. While we cannot change the prototype of a class’ destructor, we can change the prototype of the constructor function as we see fit. In this occasion, we want the constructor to take two integer values as inputs specifying the size (width and height) of the render window.

1. So, change the constructor function declaration from

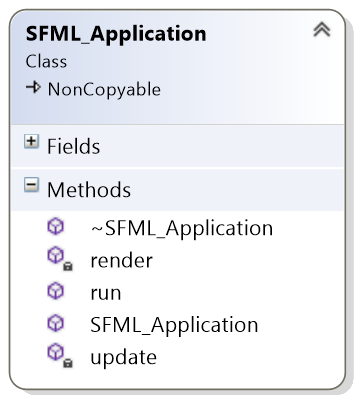


to



Our vision of the SFML\_Application class is that it has two functions that **update()** the application states and **render()** the screen. The running (in loop) and timing of the loop is controlled in another function called **run()**. That function will act as the interface to the SFML\_Application class to execute its function.

The structure of this class illustrated in the UML Class Diagram shown below.



1. Okay, let’s declare the run() function. It needs to be a **public** member function of the class.



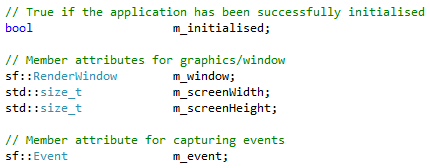
1. We need not expose the either the update() or render() member functions to the outside. Hence set them as **private** member functions



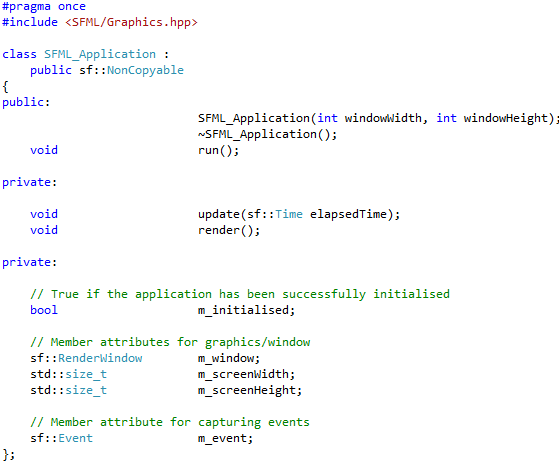
If you are unsure on how to set the encapsulation level (public or private) of a member function ask the tutor.

The class requires a number of member attributes (or fields) to perform its functions. Some of these attributes are the same as the ones we covered last week such as sf::RenderWindow, sf::Event, etc.

1. Add the following private member attributes



1. The content of SFML\_Application.h file should now look like this:

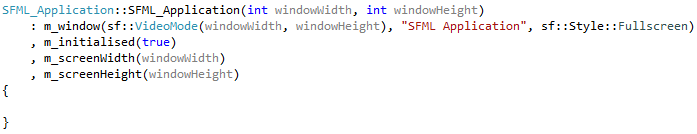


1. For now, we are done with the header file. Next, open SFML\_Application.cpp and amend the class’ constructor function’s interface to match its declaration by adding the two integer inputs.

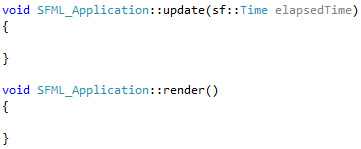


When an object of that class is created, the class constructor is called. We typically initialise all or some of the class member attributes in the constructor. We can do so inside the body of the constructor function or we can use the **Member Initializer List** technique. The difference is that the latter initialises the class member attributes just before the constructor is called, whereas the former do so as part of the constructor tasks. More information on Member Initializer List can be found on this URL <http://en.cppreference.com/w/cpp/language/initializer_list>

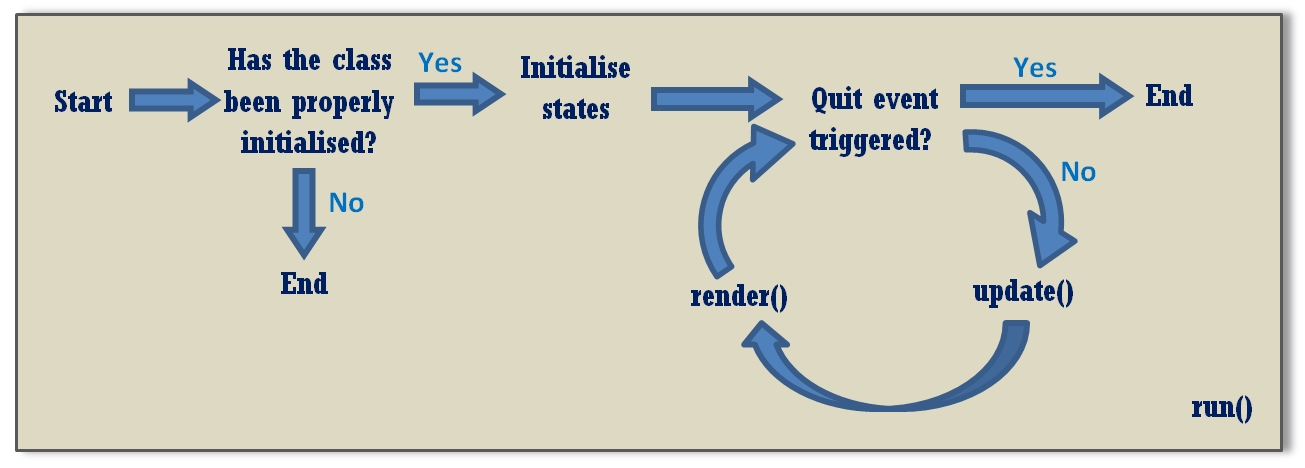
1. In this workshop, we will use the Member Initializer List to initialise all the class attributes. Modify the constructor definition to look like this:



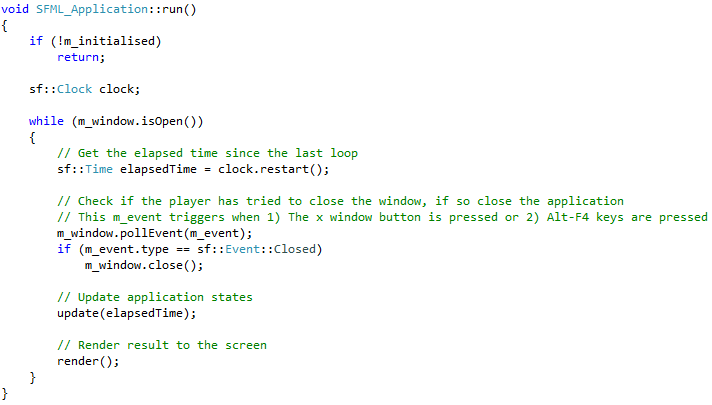
1. Create empty definitions of the update() and render() member functions as follows:



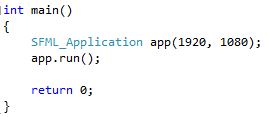
1. These functions will be executed in the class’ run() function in the manner illustrated in the following diagram.



The implementation of the run() function will look like the following



1. The next step is to use the new class in our main() function. Open main.cpp and replace the definition of the main() function to the following



1. You may notice that Visual Studio warns you that SFML\_Application is undefined. To solve this issue, make sure you include SFML\_Application.h in the main.cpp as well.



1. Build and run the program. You should see a blank white screen. You can exit this application by pressing Alt-F4.

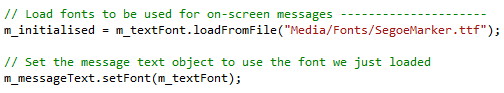
**Displaying Simple Text on Screen**

In this part, we are going to display a simple text message on the screen inside the game loop. Therefore, we need member attributes of type sf::Font and sf::Text.

1. Using File Explorer, create the following folder **/Media/Fonts** inside the **SFML\_4109COMP\_Week\_02** folder.
2. Download SegoeMarker.ttf from the Blackboard and save it inside this folder.
3. Back in Visual Studio, declare the following two objects as **private** members of the SFML\_Application class.

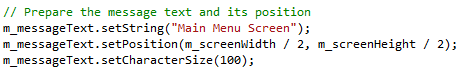


1. Inside the class constructor, load the font and set the text to use the font as the following:



Note that we use m\_initialised member attribute to store the result if we successfully loaded the font. If for some reasons the process failed then m\_initialised will be false, hence the game loop will not start.

1. Inside the **update()** member function we set the text, size and location.



1. Inside the **render()** member function we first clear the screen, render the text and update the display.



1. Build and run the program.

Note that the origin of the text is on its top-left corner.



1. If you want the text to be displayed centred at the text centre we need to set this origin to the centre of the text. We do this by finding out the text bounding rectangle and set the origin to its centre as follow:



The above lines of codes should be done inside the **update()** member function after the lines where you prepared the message’s text and position.

1. Build and rerun the program.

**Viewing the UML Class Diagram of the SFML\_Application class**

You can generate the UML class diagram of the SFML\_Application class simply by right-click on the project name in Solution Explorer. Select View » View Class Diagram from the pop-up menu.

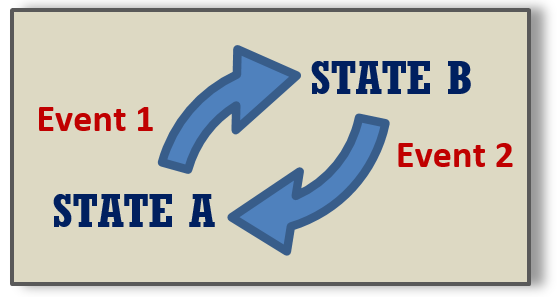
**Additional challenges**

1. Modify the program to make the text move from the left hand side of the screen to the right.
2. Modify the program again so that once the text has reached the right side of the screen it reverses direction to the left and vice versa.

# Application States

**Introduction**

A game can be in one of many different states. Consider one of your favourite RPGs. When you start the game it may display a welcome screen (TitleScreen state) and stay there until you press a key, Then It shows a menu (MainMenu state) where you can choose one of the many choices including start a new game (GamePlaying state), set game options (OptionMenu state), or quite the game (Exiting state). The logic flow of program that includes different states can be modelled and represented as a Finite-State-Machine (FSM) diagram. An example of an FSM diagram of a system with two states and two events that transition between them is illustrated below:



**Figure 1.** An FSM diagram of a system with two states and two events

In this part of the workshop, we will look at how we can modify the SFML\_Application class we created previously to accommodate different states of the application.

**Expected Learning Outcome**

1. Understand the concept of Application States
2. Implement Application States using enumeration and switch-case control structure

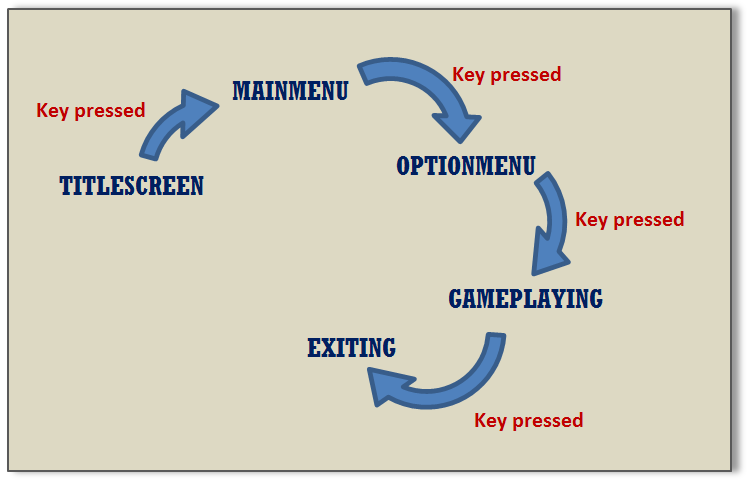
**Pre-requisites**

In order to complete this section, you will need the following programming skills

1. Understand the concept of programming project, building a programming project, including header files, and linking libraries.
2. Understand basic C++ programming

**The Finite State Machine**

The application state that we plan to achieve in this section is modelled by the FSM diagram shown below.



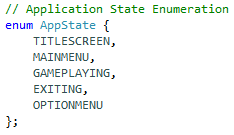
**Figure 2.** The Finite State Machine of the desired application state transitions

We will use C++ enumeration to represent the states. In C++, an enumeration provides context to describe a range of values which are represented as named constants called enumerators. More information on C++ enumeration can be found on <https://msdn.microsoft.com/en-gb/library/2dzy4k6e.aspx>

We will used unqualified (un-scoped) enumeration for our implementation. Since the states are related to the application states, it is best to declare them inside SFML\_Application class.

**The Implementation**

1. Open SFML\_Application.h and add the following enumeration as private field.



1. We need a member attribute that store the current state of the application. Declare it as a private member attribute m\_appState as shown below.



1. We can initialise this in the SFML\_Application constructor using the Member Initializer List. Add the following line to it.

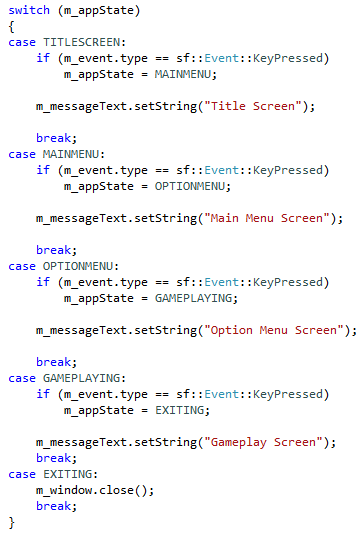


As a means to let us know, in which state the application is in currently, we are going to display different on-screen message when the application is in different states.

1. Inside the **update()** function we need to set the on-screen message depending on the current state. To do so you need to replace



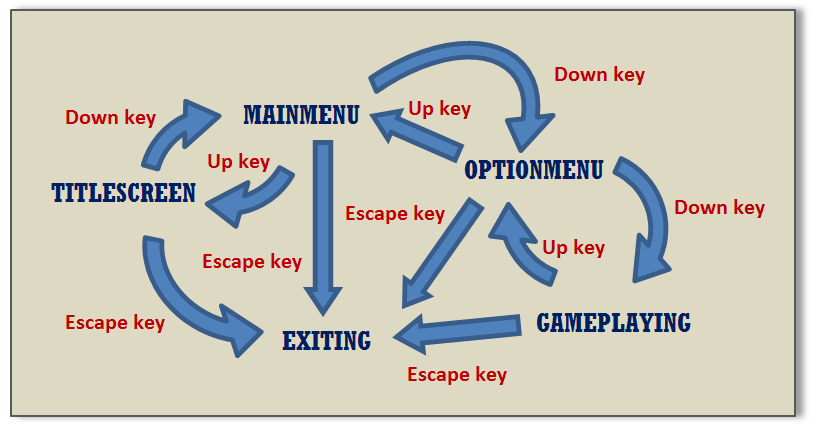
with



1. Build and run the program.

**Additional Exercises**

Implement a more complex FSM diagram illustrated in the figure below:

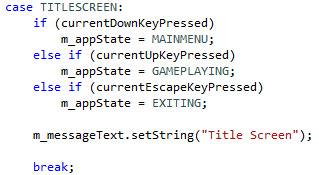


**Figure 3.** The FSM diagram of the target application

Hint: Use the following instructions to check if the Down, Up and Escape keys are pressed.



Hint: Use the following instruction to handle the transitions while the application is in TITLESCREEN state



Check Figure 3 to check what the situation for the other states is.

When you run the program, you may realise that the program polls the keypressed event very quickly. This could result in a very rapid traversing of the states too quickly and makes it harder for the user to go to a specific target. To solve that we need to only move to a new state if the key was previously up before it was pressed.

As an example, let’s consider the Down key. We need to store the previous state of that key.



And set it to false to start with



Then replace the logic



with



And store the value of the currentDownKeyPressed to the m\_previous DownKeyPressed, so that we can use it to check the previous key state in the next loop



Do the same with the Up key. Once you do that build and run the program and see if you can notice the difference.

# Additional Notes

Please use this space to add your own notes.