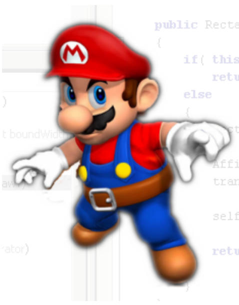


Games Programming

Philip Hanna 110CSC207

Tutorial 1: ‘Hello World’

Tutorial Overview:

This tutorial will explore how to build up a “Hello World” ‘game’ – well, it’s not really a game, but it touches upon all of the elements needed within a game: loading images, creating game objects and game layers, starting the game engine and updating/rendering graphical objects.

In terms of getting the most out of this tutorial, you will want to follow the outlined steps and develop your own version of the “hello world” game – i.e. please don’t just read the material, but rather try it out.

As with any tutorial, I’m particularly interested to receive feedback – when writing this tutorial I will try to include enough detail for you to carry out the steps, however, I depend upon your feedback to better understand the correct level at which to pitch the tutorial. Please send comments to P.Hanna@qub.ac.uk

What I assume you will have done prior to this tutorial:

Before attempting this tutorial you should have installed NetBeans, Java 1.6 and the CSC207 Code Repository. You can find instructions on how to do this within the CSC207 Code Repository document (available for download from the Core Documentation section of CSC207 on Queen’s OnLine). If you have problems getting any of the above to install, then please get in contact ASAP.

Phase 0: A design

Of course none of us would start writing any code without first developing a design as to what we intend to do... at the very least the design will help direct our thoughts and efforts concerning the code.

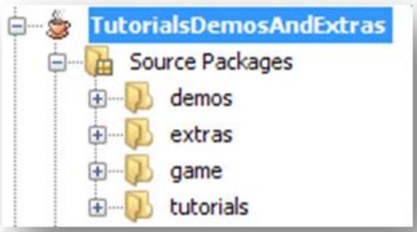
I’ve already mentioned this is going to be a “hello world” ‘game’, but what is it actually going to do? Well, my inspired vision for this game is that we will have a hello world image on screen that will follow the mouse as it is moved about the screen. Hardly an A-grade vision, but it will suffice.

So with this in mind, let’s start developing our game.

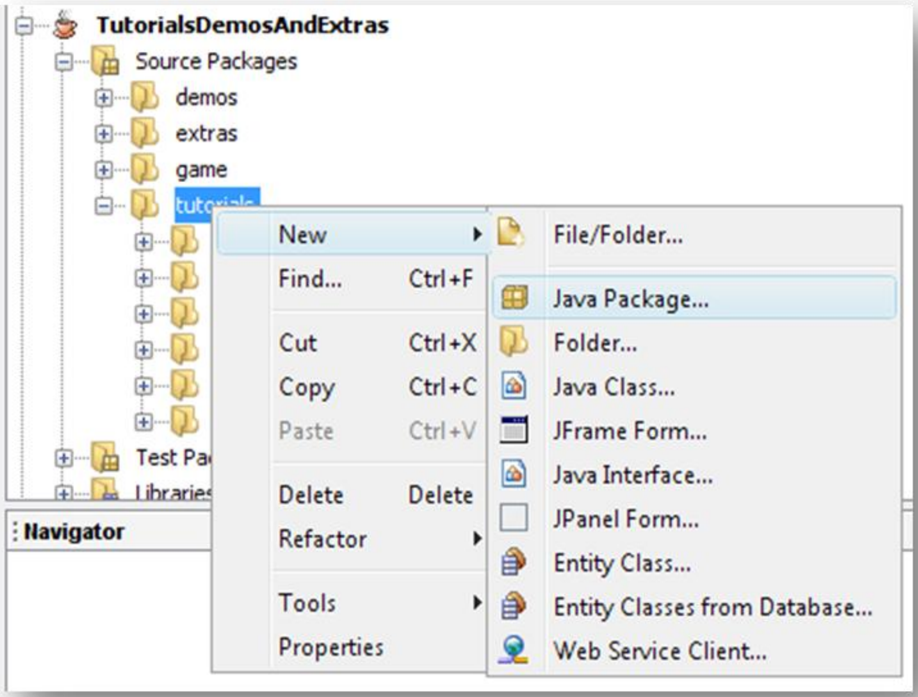
Phase 1: Creating a new package in which to build up the game

The first step is 'extending' the CSC207 Code Repository to create a package within which we can create our 'game' whilst having access to the classes within the game.engine and game.asset packages. Once we have done this, we will have an appropriate location within which to develop our game.

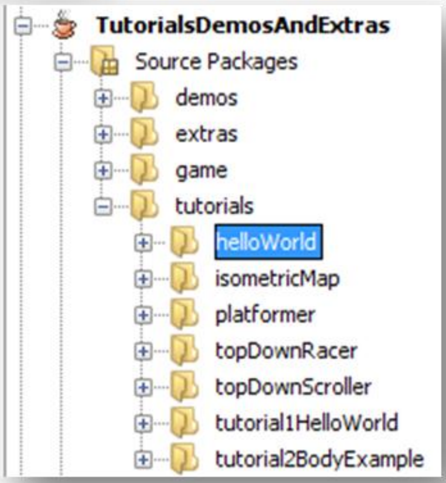
Step 1: Go to the Project tab within NetBeans (typically top-left of the screen) and expand the TutorialsDemosAndExtras project (you've already opened this project, haven't you?). Expand the Source Packages folder. This will show you all of the packages currently available within the repository (as shown to the right)



Next, right click on the 'tutorials' package entry, then 'New' and select the 'Java Package' option.



This will pop up a dialog box. Enter the name 'tutorials.helloWorld' as the package name and click on 'Finish'. If all goes well, you should now see a new package called 'helloWorld', within your tutorials package, i.e.:



Aside: Instead of extending an existing project, we could also have created a new NetBeans project and then copied across the game.core packages into the new project. You might want to try doing this once you complete this tutorial.

Phase 2: Extending and adapting the GameEngine class

We next want to create a class that extends the GameEngine class (located within the game.engine package). The GameEngine class provides a number of useful features for our game, namely:

- The ability to enter and exit Full Screen Exclusive Mode
- An update/render loop that will attempt to maintain a target number of updates per second
- Access to an asset manager (to manage our images and sounds) and a game input event manager (enabling us to check for keyboard and mouse input)
- Layer management (each GameEngine holds a number of layers – more on this later – and controls the update and render of the layers).
- Various other management feature which we won’t consider within this tutorial

Ok, lets not worry too much about the capabilities of the GameEngine (we’ll worry about them in later tutorials). Our aim is to create an “hello world” ‘game’ (which means we can ignore most of the things within the game engine – in fact, for most types of game development you can ignore the majority of the workings of the game engine and just let it get on with managing updates/renders/FSEM, etc.

So what do we do first of all? Let’s create a class that extends GameEngine. The steps to do this are as follows:

Step 1: Right click on the helloWorld package that was created within Phase 1. From the pop-up menu select ‘New’ and then ‘Java Class’. Next, enter ‘HelloWorld’ as the name of the class you wish to create and then hit ‘Finish’. This will create an HelloWorld.java class within the helloWorld package. Double click on the HelloWorld.java class to open it within the editor, ready for us to start entering some code.

Step 2: The class we just created does not extend any other class, nor does it have direct access to the game engine and game asset classes which we will need to use in order to write our game. Let’s remedy this problem. Firstly, introduce the following two import statements (after the ‘package tutorials.helloWorld’ statement, but before the ‘public class HelloWorld’ statement).

```
import game.assets.*;
import game.engine.*;
```

This will give your HelloWorld class access to all the classes within the game engine and game asset packages. Next, modify our class definition so that it extends GameEngine.

```
public class HelloWorld extends GameEngine {
```

We now have our own game engine which we will extend into our hello world game. In order to do this we will need to complete the following steps (usually in the identified order):

1. Load in the graphical and sound assets that our game uses
2. Construct and add any game layers that ‘contain’ the game to be played
3. Start the game running

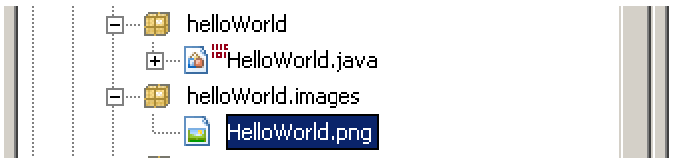
How do we accomplish the above? Thankfully the GameEngine will manage most of this process. In particular, whenever we call the gameStart(…) method, the game engine will go about its start-up process. We will still need to tell the game engine what images it should load, etc. Let’s do that now.

Phase 3: Loading assets

For this particular hello world example we will make use of a single image (entitled HelloWorld.png which you should have downloaded along with this document). We will firstly create an image directory within the helloWorld package in which to store the image (it’s generally a good idea to store your images/sounds, etc. within a tidy directory structure).

Step 1: To create a suitable folder to hold images, right click on the helloWorld package, then select ‘New’ and click on the top ‘File/Folder’ entry. This will pop up a dialogue. Click on the ‘Other’ category, then on ‘Folder’ within the ‘File Types’ pane. Click ‘Next’ At this point we can specify the name of the folder we wish to create. Enter ‘images’ as the folder name and click on ‘Finish’. This will create a helloWorld.images package.

Step 2: We next want to copy the HelloWorld.png image file into the folder we just created. To do this open Windows Explorer and copy the HelloWorld.png file into the CSC207 Code Repository TutorialsDemosAndExtras\src\tutorials\helloWorld\images folder (Aside: Do make sure it’s the src folder and not the build folder). After a couple of seconds NetBeans should detect that an image has been added as shown below:



If you double click on the image within NetBeans you can see what it looks like.

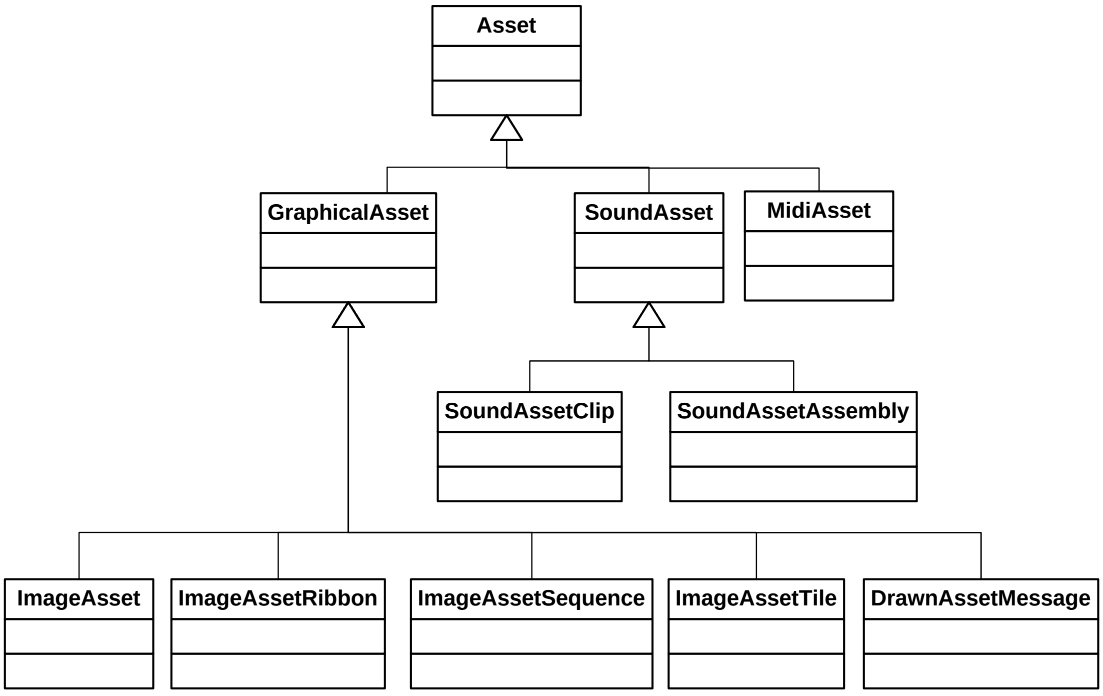
Step 3: Ok, lets go back and expand our HelloWorld.java class. Let's create a method called public boolean buildAssetManager() that loads in the various assets used by the game. Add the following method:

```
public boolean buildAssetManager()
{
   assetManager.addImageAsset("HelloWorldAsset",
         getClass().getResource("images/HelloWorld.png"));

   return true;
}
```

You might be wondering why this method returns true, or when are we going to call it. The buildAssetManager method is actually defined within the GameEngine class (i.e. we are overloading it within our helloWorld program). Whenever the gameStart() method of GameEngine is called (we'll do this later) one of the things that the GameEngine will do is to call this method to load any assets that the game uses. If this method returns true then the GameEngine knows it was possible to load all the images, etc. used by the game and it can carry on.

Good, once the class loads we will load in our HelloWorld.png image into an ImageAsset object.

Within the CSC207 Code Repository all images and sounds are held within asset objects. The top-level asset class is inventively called 'Asset'. I've provided the current class hierarchy below. As you can see a range of different types of asset are supported. However, for our purposes we only need a basic image (which won't scroll or be animated, etc.) – the class for us is ImageAsset.



All assets ‘live’ within an AssetManager instance. Within our code whenever we need an asset we will obtain it via the AssetManager, i.e. this class is our one-stop shop in terms of getting assets. Each GameEngine instance automatically created an AssetManager – this is the reason why we can simply invoke ‘assetManager.addImageAsset’ from within the buildAssetManager method (i.e. we don’t need to construct an AssetManger, we get one via inheritance from the GameEngine class).

Every asset has a name, in this case “HelloWorldAsset” which we set and can use later when we want to retrieve the asset from the AssetManager.

As an aside, the getClass().getResource(…) code basically asks Java to get the current class, which in this case is our HelloWorld instance, and to get a resource from a location relative to the location of the HelloWorld.class. In this case, we’re interested in the image folder that can be found in the same directory as the HomeWorld.class file.

Ok, at this point we’ve loaded all the assets that we will need within our game. The next step is to think about creating a suitable game layer.

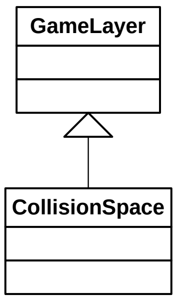
Phase 4: Creating a GameLayer

Within the code repository a game layer is basically a means of structuring and organising game objects in a manner that facilitates the update and render phase.

Why bother about having layers? Could we not simply have a big bunch of game objects and let the game engine update and render the objects? Yes, we could do this and for simple games it would be perfectly acceptable (in other words for this HelloWorld example constructing a game layer is not going to be beneficial for us – in fact, it’s going to require us to spend some effort in terms of building the class!).

However, as the complexity of the game increases we will find ourselves in situations where we only really want to update and render some of game objects and not others (i.e. for reasons of both performance and simplicity). Under such situations it is beneficial if we have some means of chopping up a large collection of game objects into separate chunks.

Anyway, let’s get on with creating our game layer. To do this we need to create a separate class that extends one of the GameLayer classes (there are currently two available, as shown below). Let’s do that now.



Step 1: Right click on the 'helloWorld' package, then select 'New' and then 'Java class'. Enter 'HelloWorldLayer' as the name of class and click 'Finish'.

Step 2: Double click on the HelloWorldLayer.java to open it for editing. As with HelloWorld.java, we will want to include suitable imports and then extend one of the available code repository classes (in this case GameLayer). Enter the following code (again paying attention to the location of the import statements, i.e. after the package definition but before the class definition).

```
import java.awt.*;
import game.engine.*;


public class HelloWorldLayer extends GameLayer {
```

Step 3: Following the above steps, NetBeans will complain that the constructor provided within HelloWorldLayer is invalid. It is – the GameLayer class does not provide a no-argument constructor. Instead each GameLayer upon construction must be provided with a name and a link to the GameEngine instance to which it belongs. Update the HelloWorldLayer constructor to that shown below:

```
public HelloWorldLayer( GameEngine gameEngine )
{
    super("HelloWorldLayer", gameEngine);
}
```

The above code will ensure that when HelloWorldLayer is constructed it automatically passes a 'HelloWorldLayer' layer name and game engine link to the GameLayer constructor.

Ok, we've now got our own version of GameLayer ready to hold the game objects need to implement our wonderful vision of an hello world 'game'. What next? Basically each game layer provides a glorified means of updating and drawing the objects that it contains. But, as of yet, we don't have any game objects within the game layer and have nothing to update or render. Hence, we will next create a game object that chases the mouse pointer as the player moves it around the screen.

Phase 5: Creating a GameObject

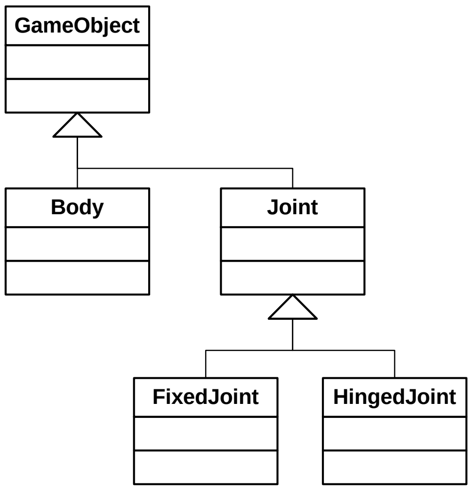
Yet another class to create? Well, yes, hopefully you did want an introduction to all of the key classes. In what will hopefully be a familiar series of steps, let's create a suitable game object.

Step 1: Right click on the 'helloWorld' package, then select 'New' and then 'Java class'. Enter 'MouseChaser' as the name of class and click 'Finish'.

As shown below there are a number of different types of GameObject. Which one do we pick? The basic one will suffice, i.e. GameObject, the other ones are for game objects that take part in the physics engine (we'll look at some of those in the next tutorial).

Step 2: Double click on the MouseChaser.java to open it for editing. As with the other classes add in the following code:

```
import game.engine.*;


public class MouseChaser extends GameObject {
```



Step 3: As with HelloWorldLayer, NetBeans will grumble that the MouseChaser constructor is invalid. In this case it's because all game objects need to know which game layer they belong to. Update the MouseChaser constructor to that shown below:

```
public MouseChaser(GameLayer gameLayer)
{
    super(gameLayer);
}
```

We've finally got things to the stage where we can start enriching the collection of objects. We'll start from the centre (chasing the mouse) and then work outwards (towards starting the game running within MouseChaser.java).

Phase 6: Chasing the mouse

How can we draw an image at the current location of the mouse? For a start we need to know the current location of the mouse. We also need to be able to draw an image.

Within the core repository, a GameInputEventManager class provides us with access to keyboard and mouse input events. This is the class that we want to use. But how do we use this class? Well, in the same way that a GameEngine instance will automatically create an AssetManager when it is constructed, the GameEngine instance will also create a GameInputEventManager (called inputEvent).

But how do we get access to the input events within our MouseChaser class? This is where the magic of inheritance will help us out – whenever we construct a GameObject (or indeed a GameLayer) the constructor automatically makes a copy of the GameInputEventManager, AssetManager and GameEngine objects. In other words, within a GameObject we can access the GameInputEventManager as inputEvent, the GameLayer as gameLayer and the GameEngine as gameEngine (our MouseChaser will automatically get these via inheritance)

Ok, so we can get access to input events within our game object. What about drawing things to the screen? Well, our MouseChaser class will inherit the following update and draw methods:

```
public void update()


public void draw( Graphics2D graphics2D )

public void draw( Graphics2D graphics2D, int drawX, int drawY )
```

These are the key methods within our MouseChaser class which are typically called many times each second. The GameEngine will trigger an update and render a certain number of times per second (defaulting to 60 times per second). During each update phase the GameEngine will call the update method of each active game layer. In turn, each game layer should update all the game objects that it contains. During the render phase the game engine will call the draw method of all visible layers. In turn, each layer will automatically draw all the objects that it contains.

In the vast majority of cases we will not need to modify the draw methods and instead just rely on the game engine and game layers to ensure that all objects are drawn. Hence, here we only need to change (i.e. overload) the update method so that it tracks the current mouse location. Let’s do that now.

Step 1: Update the MouseChaser class so that the following update method is added.

```
public void update()
{
    this.x = inputEvent.mouseXCoordinate;
    this.y = inputEvent.mouseYCoordinate;
}
```

With reference to the update method, you will see that we have assigned the mouse x and y coordinates to two inherited values, ‘x’ and ‘y’. Where did the ‘x’ and the ‘y’ come from? Well, each game object is defined by a number of different parameters (why don’t you open up the GameObject.java file within the game.Engine package and have a look?). Two of the variables are ‘x’ and ‘y’ which hold the location of the game object within the game layer. In this case, we’re changing the location of the game object to match the current location of the mouse pointer (why don’t you also open up GameInputEventManager from within the game.Engine package and have a look at the types of input method that are available?).

The last thing we need to do within MouseChaser is to get it to display the image we loaded in towards the start of this tutorial. The GameObject class is setup to permit each graphical game object to have a graphical realisation that is automatically displayed whenever the draw method is called. In other words, the only thing we need to do within this class is to the set the graphical realisation to the desired image.

The best place to do this is within the MouseChaser constructor. Let's do this now.

Step 2: Update the MouseChaser constructor as follows:

```
public MouseChaser(GameLayer gameLayer)
{
    super(gameLayer);

    setRealisationAndGeometry("HelloWorldAsset");
}
```

The inherited setRealisationAndGeometry method enables us to change the graphical realisation that the MouseChaser will automatically display whenever the draw method is called. We need to pass in the name of a graphical asset to this method (the method will automatically get the image from the AssetManager).

What about the 'AndGeometry' bit of the above method? The geometry describes the shape of the object (in terms of it's width, height, outline, etc.). Hence, whenever we call setRealisationAndGeometry(…) the game object's geometry is set to a rectangular shape of width and height equal to the image width and height. The geometry is used whenever we want to decide if two game objects overlap (i.e. they have collided). In this tutorial we don't actually use the geometry, and there is a setRealisation(…) method within the GameObject class, however, it's generally a good idea to always set the geometry.

Just to summarise, the completed MouseChaser class is shown below:

```
package tutorials.tutorial1HelloWorld;

import game.engine.*;

public class MouseChaser extends GameObject {

    public MouseChaser(GameLayer gameLayer) {
        super(gameLayer);

        setRealisationAndGeometry("HelloWorldAsset");
    }

    public void update() {
        this.x = inputEvent.mouseXCoordinate;
        this.y = inputEvent.mouseYCoordinate;
    }
}
```

We have constructed a game object that, during each update phase, will set its location to that of the mouse cursor and, during the draw phase, will draw the loaded hello world image at its current location.

Phase 7: Finishing the game layer

How that we have built a suitable game object we can return to our game layer (HelloWorldLayer). The first thing we need to ask ourselves when building a game layer is what game objects should the layer contain (and then go about constructing and adding such objects). The second thing we should then ask is how will the objects be updated during an update phase. Finally, we should ask how we want the objects to be drawn during the render phase. First things first: adding the game objects that we need.

Step 1: The HelloWorldLayer, as currently coded, does not contain any game objects. Lets create and add an instance of our MouseChaser game object. If we wanted the most straightforward coded solution, then, the following would suffice (however, we won't do it this way!)

```
public HelloWorldLayer(GameEngine gameEngine )
{
   super("HelloWorldLayer", gameEngine );


   MouseChaser mouseChaser = new MouseChaser( this );
   addGameObject( mouseChaser );
}
```

The above code creates a new instance of MouseChaser (recall that the MouseChaser constructor needs to be told to which game layer it belongs). We next use the inherited addGameObject method to add the created object to the layer. However, let's make things a little bit unnecessarily complicated (as I want to demonstrate the notion of game object collections).

A game object collection is simply a collection of game objects. Every game layer can define as many game object collections as needed. I would certainly recommend that game object collections are defined for any group of game objects that tend to be processed together as a whole (for example, all projectiles, ships, platform, etc. within a game). In this case we only have one object, but it will serve in terms of illustrating how we can use the game object collections.

Modify the constructor of HelloWorldLayer to that shown below:

```
public HelloWorldLayer(GameEngine gameEngine)
{
   super("HelloWorldLayer", gameEngine);

   addGameObjectCollection("Chasers");

   MouseChaser mouseChaser = new MouseChaser(this);
   addGameObject(mouseChaser, "Chasers");
}
```

We've created a new game object set called "Chasers" and have explicitly indicated that we would like the MouseChaser instance to be added to this game object set.

Step 2: Ok, we have created all the game objects that we need. The next question to answer is how we wish to update the various game objects. Thankfully we've already gone to the trouble of defining an update method within the MouseChaser class that does all the 'work'. Hence, within the game layer we simply need to make sure that we call the MouseChaser update method.

All GameLayer objects have (or inherit) a public void update method which will be called by the game engine every update tick. By default the GameLayer update method updates all objects contained within the layer. In other words, we don't actually need to overload this method to get it to update our MouseChaser (it will do it by default) – although if we do overload the update method we should certainly call super.update().

As we've gone to the trouble of adding a game object set, I'd like to show you how you can readily process all objects within a set. Add the following update method to HelloWorldLayer.

```
public void update()  {
   GameObjectCollection chasers = getGameObjectCollection("Chasers");
   for (int idx = 0; idx < chasers.size; idx++) {
      GameObject chaser = chasers.gameObjects[idx];
      chaser.update();
   }
}
```

Each GameObjectCollection instance simply contains an array of game objects (admittedly there is only one object in this particular example). The shown code will retrieve the game object collection, iterate over the collection and call the update method of each game object in the collection.

Step 3: Ok, we added our game object to the layer and ensured it is updated in a manner we are happy with. What about drawing the object? Again, all game layers have access to a public void draw(Graphics2D graphics2D) method. When this method is called it will automatically draw all game objects contained within the layer. In the majority of cases, the default GameLayer draw algorithm will suffice and we won't need to change anything. However, we do have a bit of extra work to do here (for reasons that will be explored fully whenever we look at page flipping). Basically, we need to clear the screen to a lovely pristine 'blank' state before we starting drawing any game objects. To do this, add in the following draw method:

```
public void draw(Graphics2D graphics2D)  {
    Color originalColour = graphics2D.getColor();
    graphics2D.setColor(Color.black);
    graphics2D.fillRect(
      0, 0, gameEngine.screenWidth, gameEngine.screenHeight);
    graphics2D.setColor(originalColour);

    super.draw(graphics2D);
}
```

The code shown above will clear the screen to black (by drawing a rectangle from position (0, 0) to (gameEngine.screenWidth, gameEngine.screenHeight). You will notice that we store the original colour and then set the Graphics2D back to the original colour. This is not really needed within this example, although it is good practice to leave the Graphics2D object in the same state as that received (the graphics instance will be used by other layers, etc. when they perform their draw).

We have created a new game layer that constructs and adds one MouseChaser instance and ensures that the mouse chaser will be updated and drawn.

Phase 7: Finishing the game engine

Now that we've got all the game layers and game objects that we need, we can think about how we will start the game running. Within our HelloWorld game engine we have already loaded in all the assets that we need. The next step is to construct and add in the game layers that we want to have available once the game starts running. In this case, we need to add an instance of HelloWorldLayer. To do this, add the following method into the HelloWorld class:

```
protected boolean buildInitialGameLayers()
{
    HelloWorldLayer helloWorldLayer = new HelloWorldLayer(this);
    addGameLayer(helloWorldLayer);

    return true;
}
```

In the same way that the GameEngine will automatically call the buildAssetManager method whenever the gameStart method is called to load in assets, it will also call the buildInitialGameLayers method to construct the initial game layers. Returning true informs the GameEngine that there were no problems in doing this.

The first line creates an instance of the HelloWorldLayer we have just finished. The next line adds the HellloWolrdLayer instance to this game engine.

We are now nearly setup and ready to go. But just before we start the game running we should think about global input and drawing. Each game engine inherits (from GameEngine) a method, protected void considerInput(), that is called at the start of each update and render phase to consider user input that is not layer specific. By default, the method will check to see if the ESC button is pressed (and, if so, exit the game). For this game the default inherited behaviour will suffice. Secondly, we should also think about additional drawing. In most of the examples I have written you will see that I display some text on the screen (for example showing FPS/UPS, key-board controls, etc.). This can be done by overloading the inherited gameRender method as shown below.

```
protected void gameRender( Graphics2D graphics2D )
{
        super.gameRender( graphics2D );
          // Whatever extra render code is needed here
}
```

The call to super.gameRender(…) ensures that the default behaviour of drawing all visible layers continues to be carried out. However, for our game we don’t really need to add anything (just to remember that, by default, ESC will exit the game).

Ok, we’re ready to start our game running. To do this, add the following code to the HelloWorld constructor.

```
        gameStart( 1024, 768, 32 );
```

The gameStart method is inherited from game engine and will start the update/render process. The three parameters are the target resolution (1024x768 in this example) and colour depth (32-bits in this example) at which FSEM should attempt to run the game. If we can't run the game at this resolution we will get an exception (a well written game would check to see which modes are available and possibly permit the user to select their desired mode).

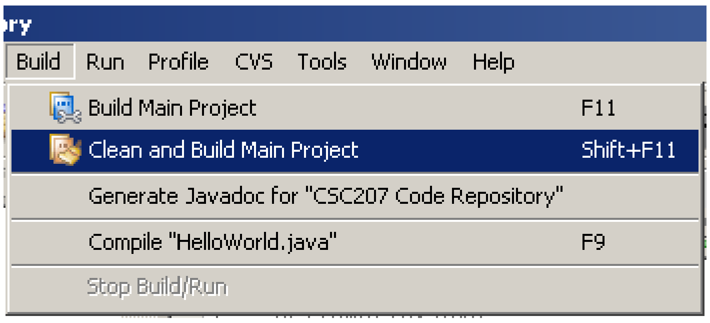
Ok, we're just about ready to go... except that we need a main method somewhere to construct and run an instance of HelloWorld. Add a main method into HelloWorld as follows:

```
public static void main(String[] args)
{
        HelloWorld instance = new HelloWorld();
}
```

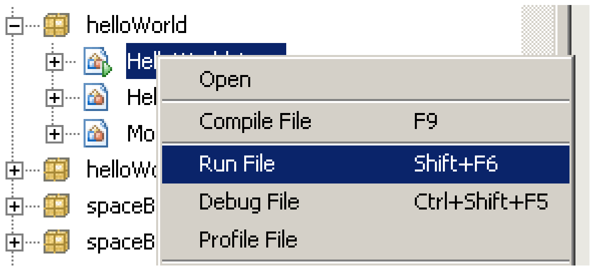
The completed HelloWorld class should look like that shown below:

```
package tutorials.tutorial1HelloWorld;

import game.engine.*;

public class HelloWorld extends GameEngine {

    public HelloWorld() {
        gameStart(1024, 768, 32);
    }

    protected boolean buildAssetManager() {
        assetManager.addImageAsset("HelloWorldAsset",
                getClass().getResource("images/HelloWorld.png"));

        return true;
    }

    protected boolean buildInitialGameLayers() {
        HelloWorldLayer helloWorldLayer = new HelloWorldLayer(this);
        addGameLayer(helloWorldLayer);

        return true;
    }

    public static void main(String[] args) {
        HelloWorld instance = new HelloWorld();
    }
}
```

Let’s run our game. Firstly, select the ‘Clean and Build Main Project’ from the ‘Build’ menu to ensure that we build all Java files and copy the HelloWorld.png from the src to build directory.


Build
Run
Profile
CVS
Tools
Window
Help
Build Main Project F11
Clean and Build Main Project Shift+F11
Generate Javadoc for "CSC207 Code Repository"
Compile "HelloWorld.java" F9
Stop Build/Run

Next, right click on HelloWorld and select ‘Run File’.



Congratulations (I hope!), you have just created and run your hello world game.

What next?

I hope that you would consider extending this single game in some way. Maybe we should have more than one image, maybe we should try to respond to keyboard input, etc. Whatever you feel to be useful. As always, please do feel free to some feedback and comments my way – it will be warmly welcomed.