Introduction to JavaFX for Game Development

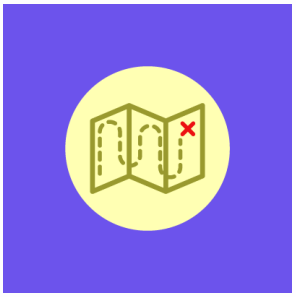
by Lee Stemkoski 19 May 2015

Difficulty: Intermediate Length: Medium Languages: English

Java JavaFX Programming How to Learn

JavaFX is a cross platform GUI toolkit for Java, and is the successor to the Java Swing libraries. In this tutorial, we will explore the features of JavaFX that make it easy to use to get started programming games in Java.

This tutorial assumes you already know how to code in Java. If not, check out Learn Java for Android, Introduction to Computer Programming With Java: 101 and 201, Head First Java, Greenfoot, or Learn Java the Hard Way to get started.



ROUNDUPS

12 Gamedev Engines and Platforms (and the Best Ways to Learn Each of Them)

Michael James Williams

Installation

If you already develop applications with Java, you probably don't need to download anything at all: JavaFX has been included with the standard JDK (Java Development Kit) bundle since JDK version 7u6 (August 2012). If you haven't updated your Java installation in a while, head to the Java download website for the latest version.

Basic Framework Classes

Creating a JavaFX program begins with the Application class, from which all JavaFX applications are extended. Your main class should call the `launch()` method, which will then call the `init()` method and then the `start()` method, wait for

the application to finish, and then call the `stop()` method. Of these methods, only the `start()` method is abstract and must be overridden.

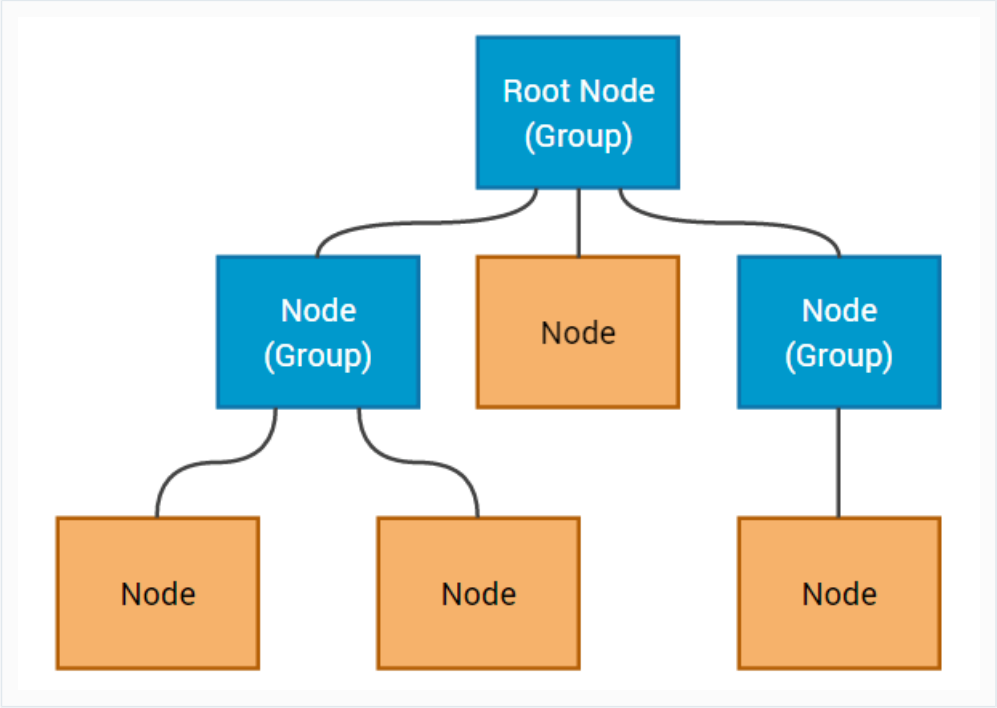
The Stage class is the top level JavaFX container. When an Application is launched, an initial Stage is created and passed to the Application's start method. Stages control basic window properties such as title, icon, visibility, resizability, fullscreen mode, and decorations; the latter is configured using StageStyle. Additional Stages may be constructed as necessary. After a Stage is configured and the content is added, the `show()` method is called.

Knowing all this, we can write a minimal example that launches a window in JavaFX:

```
01  import javafx.application.Application;
02  import javafx.stage.Stage;
03
04  public class Example1 extends Application
05  {
06      public static void main(String[] args)
07      {
08          launch(args);
09      }
10
11      public void start(Stage theStage)
12      {
13          theStage.setTitle("Hello, World!");
14          theStage.show();
15      }
16  }
```

Structuring Content

Content in JavaFX (such as text, images, and UI controls) is organized using a tree-like data structure known as a scene graph, which groups and arranges the elements of a graphical scene.



Representation of a JavaFX Scene Graph.

A general element of a scene graph in JavaFX is called a Node. Every Node in a tree has a single "parent" node, with the exception of a special Node designated as the "root". A Group is a Node which can have many "child" Node elements. Graphical transformations (translation, rotation, and scale) and effects applied to a Group also apply to its children. Nodes can be styled using JavaFX Cascading Style Sheets (CSS), quite similar to the CSS used to format HTML documents.

The Scene class contains all content for a scene graph, and requires a root Node to be set (in practice, this is often a Group). You can set the size of a Scene specifically; otherwise, the size of a Scene will be automatically calculated based on its content. A Scene object must be passed to the Stage (by the `setScene()` method) in order to be displayed.

Rendering Graphics

Rendering graphics is particularly important to game programmers! In JavaFX, the Canvas object is an image on which we can draw text, shapes, and images, using its associated GraphicsContext object. (For those developers familiar with the Java Swing toolkit, this is similar to the Graphics object passed to the `paint()` method in the JFrame class.)

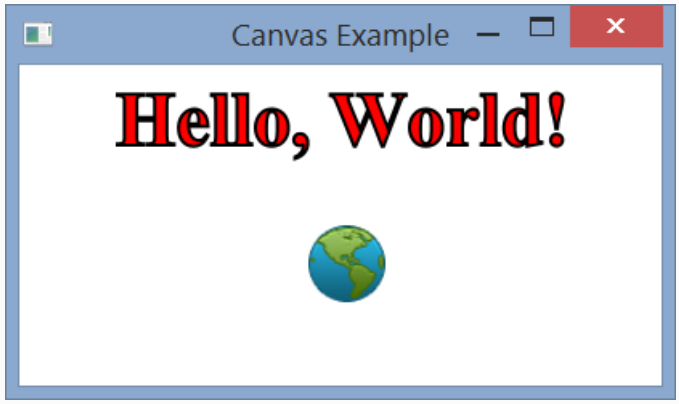
The GraphicsContext object contains a wealth of powerful customization abilities. To choose colors for drawing text and shapes, you can set the fill (interior) and stroke (border) colors, which are Paint objects: these can be a single solid Color, a user-defined gradient (either LinearGradient or RadialGradient), or even an ImagePattern. You can also apply one or more Effect style objects, such as Lighting, Shadow, or GaussianBlur, and change fonts from the default by using the Font class.

The Image class makes it easy to load images from a variety of formats from files and draw them via the GraphicsContext class. It's easy to construct procedurally generated images by using the WritableImage class together with the PixelReader and PixelWriter classes.

Using these classes, we can write a much more worthy "Hello, World"-style example as follows. For brevity, we'll just

include the `start()` method here (we'll skip the import statements and `main()` method); however, complete working source code can be found in the GitHub repo that accompanies this tutorial.

```
01  public void start(Stage theStage)
02  {
03      theStage.setTitle( "Canvas Example" );
04
05      Group root = new Group();
06      Scene theScene = new Scene( root );
07      theStage.setScene( theScene );
08
09      Canvas canvas = new Canvas( 400, 200 );
10      root.getChildren().add( canvas );
11
12      GraphicsContext gc = canvas.getGraphicsContext2D();
13
14      gc.setFill( Color.RED );
15      gc.setStroke( Color.BLACK );
16      gc.setLineWidth(2);
17      Font theFont = Font.font( "Times New Roman", FontWeight.BOLD, 48 );
18      gc.setFont( theFont );
19      gc.fillText( "Hello, World!", 60, 50 );
20      gc.strokeText( "Hello, World!", 60, 50 );
21
22      Image earth = new Image( "earth.png" );
23      gc.drawImage( earth, 180, 100 );
24
25      theStage.show();
26  }
```



The Game Loop

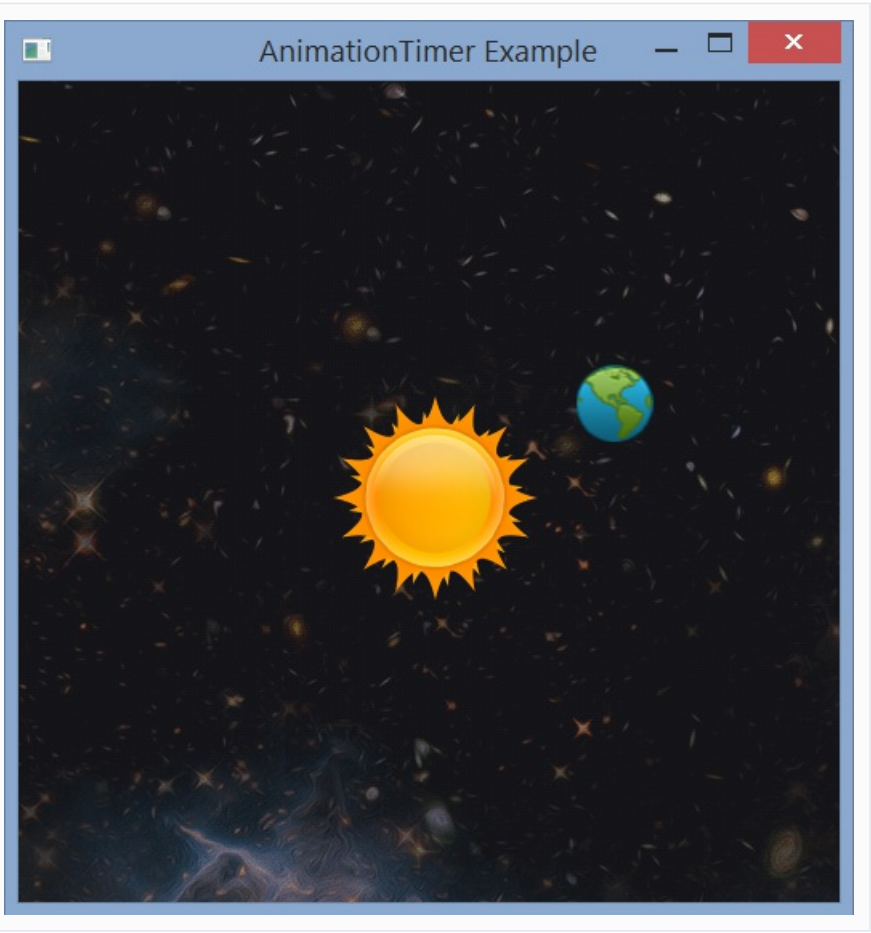
Next, we need to make our programs *dynamic*, meaning that the game state changes over time. We'll implement a game loop: an infinite loop that updates the game objects and renders the scene to the screen, ideally at a rate of 60 times per second.

The easiest way to accomplish this in JavaFX is using the AnimationTimer class, where a method (named `handle()`) may be written that will be called at a rate of 60 times per second, or as close to that rate as is possible. (This class does not have to be used solely for animation purposes; it is capable of far more.)

Using the AnimationTimer class is a bit tricky: since it is an abstract class, it cannot be created directly—the class must be extended before an instance can be created. However, for our simple examples, we will extend the class by writing an anonymous inner class. This inner class must define the abstract method `handle()`, which will be passed a single argument: the current system time in nanoseconds. After defining the inner class, we immediately invoke the `start()` method, which begins the loop. (The loop can be stopped by calling the `stop()` method.)

With these classes, we can modify our "Hello, World" example, creating an animation consisting of the Earth orbiting

around the Sun against a starry background image.




```
01  public void start(Stage theStage)
02  {
03      theStage.setTitle( "Timeline Example" );
04
05      Group root = new Group();
06      Scene theScene = new Scene( root );
07      theStage.setScene( theScene );
08
09      Canvas canvas = new Canvas( 512, 512 );
10      root.getChildren().add( canvas );
11
12      GraphicsContext gc = canvas.getGraphicsContext2D();
13
14      Image earth = new Image( "earth.png" );
15      Image sun   = new Image( "sun.png" );
16      Image space = new Image( "space.png" );
17
18      final long startNanoTime = System.nanoTime();
19
20      new AnimationTimer()
21      {
22          public void handle(long currentNanoTime)
23          {
24              double t = (currentNanoTime - startNanoTime) / 1000000000.0;
25
26              double x = 232 + 128 * Math.cos(t);
27              double y = 232 + 128 * Math.sin(t);
28
29              // background image clears canvas
30              gc.drawImage( space, 0, 0 );
31              gc.drawImage( earth, x, y );
32              gc.drawImage( sun, 196, 196 );
33          }
34      }.start();
35
36      theStage.show();
37  }
```

There are alternative ways to implement a game loop in JavaFX. A slightly longer (but more flexible) approach involves the Timeline class, which is an animation sequence consisting of a set of KeyFrame objects. To create a game loop, the Timeline should be set to repeat indefinitely, and only a single KeyFrame is required, with its Duration set to 0.016 seconds (to attain 60 cycles per second). This implementation can be found in the `Example3T.java` file in the GitHub repo.

Frame-Based Animation

Another commonly needed game programming component is frame-based animation: displaying a sequence of images in rapid succession to create the illusion of movement.

Assuming that all animations loop and all frames display for the same number of seconds, a basic implementation could be as simple as follows:

```
01  public class AnimatedImage
02  {
03     public Image[] frames;
04     public double duration;
05
06     public Image getFrame(double time)
07     {
08        int index = (int)((time % (frames.length * duration)) / duration);
09        return frames[index];
10     }
11  }
```

To integrate this class into the previous example, we could create an animated UFO, initializing the object using the code:

```
1  AnimatedImage ufo = new AnimatedImage();
2  Image[] imageArray = new Image[6];
3  for (int i = 0; i < 6; i++)
4     imageArray[i] = new Image( "ufo_" + i + ".png" );
5  ufo.frames = imageArray;
6  ufo.duration = 0.100;
```

...and, within the AnimationTimer, adding the single line of code:

```
1  gc.drawImage( ufo.getFrame(t), 450, 25 );
```

...at the appropriate spot. For a complete working code example, see the file `Example3AI.java` in the GitHub repo.

Handling User Input

Detecting and processing user input in JavaFX is straightforward. User actions that can be detected by the system, such as key presses and mouse clicks, are called *events*. In JavaFX, these actions automatically cause the generation of objects (such as KeyEvent and MouseEvent) that store the associated data (such as the actual key pressed or the location of the mouse pointer). Any JavaFX class which implements the EventTarget class, such as a Scene, can "listen" for events and handle them; in the examples that follow, we'll show how to set up a Scene to process various events.

Glancing through the documentation for the Scene class, there are many methods that listen for handling different types of input from different sources. For instance, the method `setOnKeyPressed()` can assign an EventHandler that

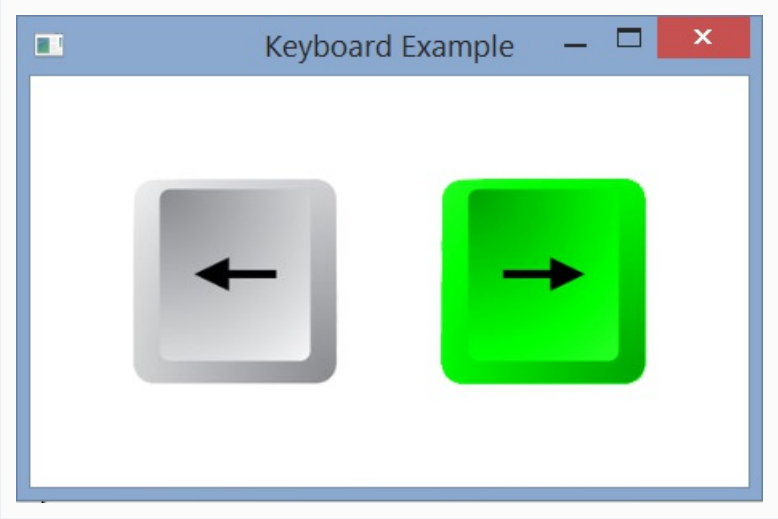
will activate when a key is pressed, the method `setOnMouseClicked()` can assign an EventHandler that activates when a mouse button is pressed, and so on. The EventHandler class serves one purpose: to encapsulate a method (called `handle()`) that is called when the corresponding event occurs.

When creating an EventHandler, you must specify the *type* of Event that it handles: you can declare an `EventHandler<KeyEvent>` or an `EventHandler<MouseEvent>`, for example. Also, EventHandlers are often created as anonymous inner classes, as they are typically only used once (when they are passed as an argument to one of the methods listed above).

Handling Keyboard Events

User input is often processed within the main game loop, and thus a record must be kept of which keys are currently active. One way to accomplish this is by creating an ArrayList of String objects. When a key is initially pressed, we add the String representation of the KeyEvent's KeyCode to the list; when the key is released, we remove it from the list.

In the example below, the canvas contains two images of arrow keys; whenever a key is pressed, the corresponding image becomes green.



The source code is contained in the file `Example4K.java` in the GitHub repo.

```
public void start(Stage theStage)
   {
      theStage.setTitle( "Keyboard Example" );

      Group root = new Group();
      Scene theScene = new Scene( root );
      theStage.setScene( theScene );

      Canvas canvas = new Canvas( 512 - 64, 256 );
      root.getChildren().add( canvas );

      ArrayList<String> input = new ArrayList<String>();

      theScene.setOnKeyPressed(
         new EventHandler<KeyEvent>()
         {
            public void handle(KeyEvent e)
            {
               String code = e.getCode().toString();

               // only add once... prevent duplicates
               if ( !input.contains(code) )
                  input.add( code );
            }
         });

      theScene.setOnKeyReleased(
         new EventHandler<KeyEvent>()
         {
            public void handle(KeyEvent e)
            {
               String code = e.getCode().toString();
               input.remove( code );
            }
         });

      GraphicsContext gc = canvas.getGraphicsContext2D();

      Image left = new Image( "left.png" );
      Image leftG = new Image( "leftG.png" );

      Image right = new Image( "right.png" );
      Image rightG = new Image( "rightG.png" );

      new AnimationTimer()
      {
         public void handle(long currentNanoTime)
         {
            // Clear the canvas
            gc.clearRect(0, 0, 512,512);

            if (input.contains("LEFT"))
               gc.drawImage( leftG, 64, 64 );
            else
               gc.drawImage( left, 64, 64 );

            if (input.contains("RIGHT"))
               gc.drawImage( rightG, 256, 64 );
            else
               gc.drawImage( right, 256, 64 );
         }
      }.start();

      theStage.show();
   }
```

Handling Mouse Events

Now let's look at an example that focuses on the MouseEvent class rather than the KeyEvent class. In this mini-game, the player earns a point every time the target is clicked.

Since the EventHandlers are inner classes, any variables they use must be final or "effectively final", meaning that the variables can not be reinitialized. In the previous example, the data was passed to the EventHandler by means of an ArrayList, whose values can be changed without reinitializing (via the `add()` and `remove()` methods).

However, in the case of basic data types, the values cannot be changed once initialized. If you would like the EventHandler to access basic data types that are changed elsewhere in the program, you can create a wrapper class that contains either public variables or getter/setter methods. (In the example below, `IntValue` is a class that contains a `public` `int` variable called `value`.)

```
01  public void start(Stage theStage)
02  {
03     theStage.setTitle( "Click the Target!" );
04
05     Group root = new Group();
06     Scene theScene = new Scene( root );
07     theStage.setScene( theScene );
08
09     Canvas canvas = new Canvas( 500, 500 );
10
11     root.getChildren().add( canvas );
12
13     Circle targetData = new Circle(100,100,32);
14     IntValue points = new IntValue(0);
15
16     theScene.setOnMouseClicked(
17        new EventHandler<MouseEvent>()
18        {
19           public void handle(MouseEvent e)
20           {
21              if ( targetData.containsPoint( e.getX(), e.getY() ) )
22              {
23                 double x = 50 + 400 * Math.random();
24                 double y = 50 + 400 * Math.random();
25                 targetData.setCenter(x,y);
26                 points.value++;
27              }
28              else
29                 points.value = 0;
30           }
31        });
32
33     GraphicsContext gc = canvas.getGraphicsContext2D();
34
35     Font theFont = Font.font( "Helvetica", FontWeight.BOLD, 24 );
36     gc.setFont( theFont );
37     gc.setStroke( Color.BLACK );
38     gc.setLineWidth(1);
39
40     Image bullseye = new Image( "bullseye.png" );
41
42     new AnimationTimer()
43     {
44        public void handle(long currentNanoTime)
45        {
46           // Clear the canvas
47           gc.setFill( new Color(0.85, 0.85, 1.0, 1.0) );
48           gc.fillRect(0,0, 512,512);
49
50           gc.drawImage( bullseye,
51              targetData.getX() - targetData.getRadius(),
52              targetData.getY() - targetData.getRadius() );
53
54           gc.setFill( Color.BLUE );
55
56           String pointsText = "Points: " + points.value;
57           gc.fillText( pointsText, 360, 36 );
58           gc.strokeText( pointsText, 360, 36 );
59        }
60     }.start();
61
62     theStage.show();
63  }
```

The full source code is contained in the GitHub repo; the main class is `Example4M.java`.

Creating a Basic Sprite Class With JavaFX

In video games, a *sprite* is the term for a single visual entity. Below is an example of a Sprite class that stores an image and position, as well as velocity information (for mobile entities) and width/height information to use when calculating bounding boxes for the purposes of collision detection. We also have the standard getter/setter methods for most of this data (omitted for brevity), and some standard methods needed in game development:

- `update()` : calculates the new position based on the Sprite's velocity.
- `render()` : draws the associate Image to the canvas (via the GraphicsContext class) using the position as coordinates.
- `getBoundary()` : returns a JavaFX Rectangle2D object, useful in collision detection due to its intersects method.
- `intersects()` : determines whether the bounding box of this Sprite intersects with that of another Sprite.

```
01  public class Sprite
02  {
03     private Image image;
04     private double positionX;
05     private double positionY;
06     private double velocityX;
07     private double velocityY;
08     private double width;
09     private double height;
10
11     // ...
12     // methods omitted for brevity
13     // ...
14
15     public void update(double time)
16     {
17        positionX += velocityX * time;
18        positionY += velocityY * time;
19     }
20
21     public void render(GraphicsContext gc)
22     {
23        gc.drawImage( image, positionX, positionY );
24     }
25
26     public Rectangle2D getBoundary()
27     {
28        return new Rectangle2D(positionX,positionY,width,height);
29     }
30
31     public boolean intersects(Sprite s)
32     {
33        return s.getBoundary().intersects( this.getBoundary() );
34     }
35  }
```

The full source code is included in `Sprite.java` in the GitHub repo.

Using the Sprite Class

With the assistance of the Sprite class, we can easily create a simple collecting game in JavaFX. In this game, you assume the role of a sentient briefcase whose goal is to collect the many money bags that have been left lying around by a careless previous owner. The arrow keys move the player around the screen.

This code borrows heavily from previous examples: setting up fonts to display the score, storing keyboard input with an ArrayList, implementing the game loop with an AnimationTimer, and creating wrapper classes for simple values that need to be modified during the game loop.

One code segment of particular interest involves creating a Sprite object for the player (briefcase) and an ArrayList of Sprite objects for the collectibles (money bags):

```
Sprite briefcase = new Sprite();
briefcase.setImage("briefcase.png");
briefcase.setPosition(200, 0);

ArrayList<Sprite> moneybagList = new ArrayList<Sprite>();

for (int i = 0; i < 15; i++)
{
    Sprite moneybag = new Sprite();
    moneybag.setImage("moneybag.png");
    double px = 350 * Math.random() + 50;
    double py = 350 * Math.random() + 50;
    moneybag.setPosition(px,py);
    moneybagList.add( moneybag );
}
```

Another code segment of interest is the creation of the `AnimationTimer`, which is tasked with:

- calculating the time elapsed since the last update
- setting the player velocity depending on the keys currently pressed
- performing collision detection between the player and collectibles, and updating the score and list of collectibles when this occurs (an Iterator is used rather than the ArrayList directly to avoid a Concurrent Modification Exception when removing objects from the list)

- rendering the sprites and text to the canvas

```
new AnimationTimer()
{
    public void handle(long currentNanoTime)
    {
        // calculate time since last update.
        double elapsedTime = (currentNanoTime - lastNanoTime.value) / 1000000000.0;
        lastNanoTime.value = currentNanoTime;

        // game logic

        briefcase.setVelocity(0,0);
        if (input.contains("LEFT"))
            briefcase.addVelocity(-50,0);
        if (input.contains("RIGHT"))
            briefcase.addVelocity(50,0);
        if (input.contains("UP"))
            briefcase.addVelocity(0,-50);
        if (input.contains("DOWN"))
            briefcase.addVelocity(0,50);

        briefcase.update(elapsedTime);

        // collision detection

        Iterator<Sprite> moneybagIter = moneybagList.iterator();
        while ( moneybagIter.hasNext() )
        {
            Sprite moneybag = moneybagIter.next();
            if ( briefcase.intersects(moneybag) )
            {
                moneybagIter.remove();
                score.value++;
            }
        }

        // render

        gc.clearRect(0, 0, 512,512);
        briefcase.render( gc );

        for (Sprite moneybag : moneybagList )
            moneybag.render( gc );

        String pointsText = "Cash: $" + (100 * score.value);
        gc.fillText( pointsText, 360, 36 );
        gc.strokeText( pointsText, 360, 36 );
    }
}.start();
```

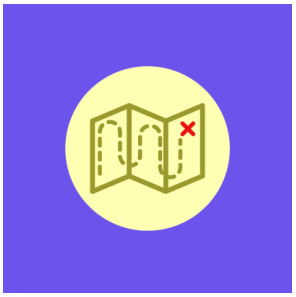
As usual, complete code can be found in the attached code file (`Example5.java`) in the GitHub repo.

Next Steps

- There is a collection of introductory tutorials on the Oracle website, which will help you learn common JavaFX tasks: Getting Started with JavaFX Sample Applications.
- You may be interested in learning how to use Scene Builder, a visual layout environment for designing user interfaces. This program generates FXML, which is an XML-based language that can be used to define a user interface for a JavaFX program. For this, see JavaFX Scene Builder: Getting Started.
- FX Experience is an excellent blog, updated regularly, that contains information and sample projects of interest to JavaFX developers. Many of the demos listed are quite inspirational!
- José Pereda has excellent examples of more advanced games built with JavaFX in his GitHub repository.
- The JFxtras project consists of a group of developers that have created extra JavaFX components which provide commonly needed functionality currently missing from JavaFX.
- The JavaFXPorts project enables you to package your JavaFX application for deployment on iOS and Android.
- You should bookmark the official references for JavaFX, in particular, Oracle's JavaFX guide and the API documentation.
- Some well-reviewed books on JavaFX include Pro JavaFX 8, JavaFX 8 - Introduction by Example, and, of particular interest to game developers, Beginning Java 8 Games Development.

Conclusion

In this tutorial, I've introduced you to JavaFX classes that are useful in game programming. We worked through a series of examples of increasing complexity, culminating in a sprite-based collection-style game. Now you're ready to either investigate some of the resources listed above, or to dive in and start creating your own game. The best of luck to you in your endeavors!



ROUNDUPS

12 Gamedev Engines and Platforms (and the Best Ways to Learn Each of Them)

Michael James Williams

Lee Stemkoski

A professor of mathematics and computer science with interests in 3D graphics and game development.

Weekly email summary

Subscribe below and we'll send you a weekly email summary of all new Game Development tutorials. Never miss out on learning about the next big thing.

Email Address

Update me weekly

View on Github

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

24 Comments Tuts+ Hub Login

Recommend 5 Share Sort by Best

Join the discussion…

Chris • a year ago
10/10 best tutorial on anything ever
6 • Reply • Share ›

Todd • a year ago
Hi, nice tutorial, I just wanted to add something that was holding me back.
I am using Java 8 with Netbeans 8.0.2 and had to make a small change to run the examples.
In place of:
Image earth = new Image("earth.png");
I had to use:
Image earth = new Image(getClass().getResourceAsStream("earth.png"));

It has something to do with the fact that Netbeans 8.0.2 packs files into a .jar file to run the project. For some reason it fails to find the image unless you use the above to find the image at runtime. Or at least that's what I think is happening.
2 • Reply • Share ›

Radosza Belián ↗ Todd • a year ago
Hi.
Try Image earth = new Image("file:earth.png"); instead. It worked for me.
1 • Reply • Share ›

Michael James Williams Mod ↗ Radosza Belián • a year ago
Thanks for the tip!

Reply • Share ›

Sakshi Goyal ➚ Todd • 6 months ago
thanks
Reply • Share ›

ykzg htvo • a year ago
Nice tutorial!

Nagel: I observe earth animation stuttering / lurching too.

I have found that if

1) I set the stage to StageStyle UTILITY
=> theStage.initStyle(StageStyle.UTILITY);
or

2) if I set the stage to fullscreen

=> theStage.setFullScreen(true);

than stuttering goes away.

Happy coding
2 Reply • Share ›

Michael James Williams Mod ➚ ykzg htvo • a year ago
Thank you for sharing this solution!
Reply • Share ›

SwedishDev • a year ago
Excellent tutorial, I learnt a lot! I noticed some improvements that could be made to the example code, I made a pull request on your GitHub repo.
1 Reply • Share ›

Michael James Williams Mod ➚ SwedishDev • a year ago
Wow, thanks so much! I've merged the request :)
1 Reply • Share ›

SwedishDev ➚ Michael James Williams • a year ago
My pleasure :)! Maybe update this article to use the new code aswell? I'd also like to request a tutorial on 3D game development with JavaFx, would be cool :D!
Reply • Share ›

Michael James Williams Mod ➚ SwedishDev • a year ago
Good call! I'll put it on my list to update the samples at some point. Point noted about 3D JavaFX gamedev :)
Reply • Share ›

SwedishDev ➚ Michael James Williams • a year ago
Sweet!
Reply • Share ›

Owen Gartside (UCC) • 2 years ago
Hi, Excellent set of tutorials, I've learnt so much. However I'm currently at Example4M "Click the Targets" and it doesn't appear to work even when I copy your GitHub code in. See Fig1, it says that I need to import javafx.scene.shape.circle; which seems fair although your code does not have it. but when I do, 2 more uncorrectable errors appear, see Fig2.

Also and perhaps more importantly, "IntValue" isn't recognised/cant be resolved to a type.

(EDIT: it looks as if it's one big image, however they are 2 separate images starting at "//Creates new circle object...." and not all once piece of code.)

Any ideas?

Thanks again for this tutorial.

Owen G

1 • Reply • Share ›

Todd ➚ Owen Gartside (UCC) • a year ago

If you look in the repository 'IntValue' is a reference to IntValue.class custom class.

2 • Reply • Share ›

Michael James Williams Mod ➚ Todd • a year ago

Thanks for the explanation!

Reply • Share ›

hiro • a year ago

Thank you, nice tutorial!

Can I translate this tutorial to Japanese and use the images in this tutorial?
I would like to spread this tutorial.

1 • Reply • Share ›

Michael James Williams Mod ➚ hiro • a year ago

Hey hiro! I'm afraid we don't allow anyone to republish our content on other sites (even translated versions), but if you're keen on translating the content to another language, check out the Tuts+ Translation Project: http://tutsplus.com/translate-.... You could translate it to Japanese and then post a summary and link on your own site, if you're interested. Thanks!

1 • Reply • Share ›

hiro ➚ Michael James Williams • 9 months ago

I've just finished translating!

1 • Reply • Share ›

Michael James Williams Mod ➚ hiro • 9 months ago

So I see! Thanks so much, hiro - I notice it's already online :) http://gamedevelopment.tutsplu...

1 • Reply • Share ›

hiro ➚ Michael James Williams • 10 months ago

(I've been busy for these days)

Anyway, thanks for giving advices!
I'll start to try it.

1 • Reply • Share ›

JunHo Seo • 2 months ago

Nice tutorial. Thanks!

Reply • Share ›

Arista sandy • 5 months ago

good tutorial , could you give me information how javafx run in browser ?

Reply • Share ›

java_an • a year ago

best java8 tutorial

Reply • Share ›

Raymond Nagel • a year ago

Hello,

I didn't have any issue running the examples - thanks for making the code available. I have done similar things with sprites using overridden paint methods in Swing. The animation in both cases is flicker-free. But one thing bugs me: as the sprites move around, they lurch slightly... the movement is rhythmically interrupted. I'm assuming this is because of JavaFX's retained mode drawing; but is there no way to avoid this and get smooth/steady movement? Thanks...

-RN

Reply • Share ›