Visual Programming Enviroments - Project

About java-fx

Paul stefan berbec

2015

About JavaFX

Contents

[2 History 3](#_Toc409072786)

[3 How does it work 6](#_Toc409072787)

[3.1 Scene graph 7](#_Toc409072788)

[3.2 Graphics system 8](#_Toc409072789)

[3.3 Glass window toolkit 9](#_Toc409072790)

[3.3.1 Threads 9](#_Toc409072791)

[3.3.2 Pulse 10](#_Toc409072792)

[3.4 UI Controls 10](#_Toc409072793)

[4 Example of application design 11](#_Toc409072794)

[4.1 Use CDI 11](#_Toc409072795)

[4.2 General design 11](#_Toc409072796)

[4.3 A different approach 12](#_Toc409072797)

[4.3.1 FXML Controller annotation 12](#_Toc409072798)

[4.3.2 Use the CDI magic 12](#_Toc409072799)

[4.3.3 The rest 13](#_Toc409072800)

# History



***JavaFX*** started life as the brainchild of Chris Oliver when he worked for a company named SeeBeyond. They had the need for richer user interfaces, so Chris created a language that he dubbed F3 (Form Follows Function) for that purpose. In the article, "Mind-Bendingly Cool Innovation" (cited in the Resources section at the end of this topic) Chris is quoted as follows. "When it comes to integrating people into business processes, you need graphical user interfaces for them to interact with, so there was a use case for graphics in the enterprise application space, and there was an interest at SeeBeyond in having richer user interfaces."

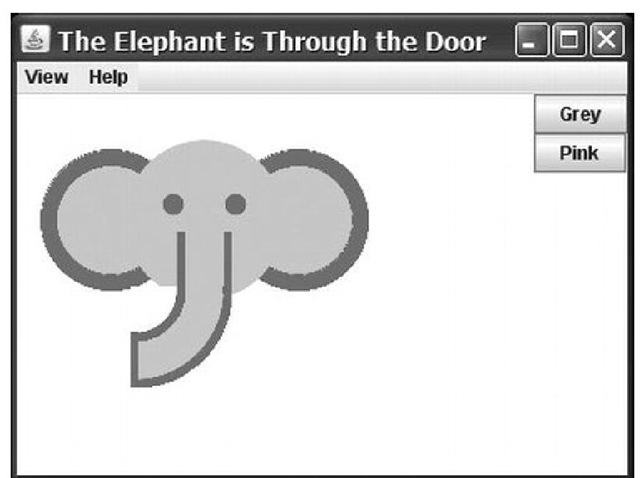
**SeeBeyond was acquired by Sun,** who subsequently changed the name of F3 to JavaFX, and announced it at JavaOne 2007. Chris joined Sun during the acquisition and continued to lead the development of JavaFX.

**The first version of JavaFX Script was** an interpreted language, and was considered a prototype of the compiled JavaFX Script language that was to come later. Interpreted JavaFX Script was very robust, and there were two JavaFX topics published in the latter part of 2007 based on that version. One was written in Japanese, and the other was written in English and published by Apress.

**While developers were experimenting** with JavaFX and providing feedback for improvement, the JavaFX Script compiler team at Sun was busy creating a compiled version of the language. This included a new set of runtime API libraries. The JavaFX Script compiler project reached a tipping point in early December 2007, which was commemorated in a blog post entitled "Congratulations to the JavaFX Script Compiler Team—The Elephant Is Through the Door." That phrase came from the JavaFX Script compiler project leader Tom Ball in a blog post, which contained the following excerpt.

**An elephant analogy came to me** when I was recently grilled about exactly when the JavaFX Script compiler team will deliver our first milestone release. "I can’t give you an accurate date," I said. "It’s like pushing an elephant through a door; until a critical mass makes it past the threshold you just don’t know when you’ll be finished. Once you pass that threshold, though, the rest happens quickly and in a manner that can be more accurately predicted."

**A screenshot of the silly,** compiled JavaFX application written by one of the authors,for that post is shown in Figure 1-1, demonstrating that the project had in fact reached the critical mass to which Tom Ball referred.

[](http://what-when-how.com/wp-content/uploads/2012/03/tmpA1_thumb.jpg)

**Much progress continued to be made on JavaFX in 2008:**

**• The NetBeans JavaFX** plug-in became available for the compiled version in March 2008.

**• Many of the JavaFX runtime** libraries (mostly focusing on the UI aspects of JavaFX) were rewritten by a team that included some very talented developers from the Java Swing team.

**• In July 2008,** the JavaFX Preview SDK was released, and at JavaOne 2008 Sun announced that the JavaFX 1.0 SDK would be released in fall 2008.

**• On December 4, 2008,** the JavaFX 1.0 SDK was released. This event increased the adoption rate of JavaFX by developers and IT managers because it represented a stable codebase.

**• In April 2009,** Oracle and Sun announced that Oracle would be acquiring Sun. The JavaFX 1.2 SDK was released at JavaOne 2009.

**• In January 2010**, Oracle completed its acquisition of Sun. JavaFX 1.3 SDK was released in April 2010, with JavaFX 1.3.1 being the last of the 1.3 releases.

**At JavaOne 2010,** JavaFX 2.0 was announced. The JavaFX 2.0 roadmap was published by Oracle on the Web page noted in the Resources section below, and includes items such as the following.

**• Deprecate the** JavaFX Script language in favor of using Java and the JavaFX 2.0 API. This brings JavaFX into the mainstream by making it available to any language (such as Java, Groovy, and JRuby) that runs on the JVM.

**• Make the compelling features** of JavaFX Script, including binding to expressions, available in the JavaFX 2.0 API.

**• Offer an increasingly** rich set of UI components, building on the components already available in JavaFX 1.3.

**• Provide a** Web component for embedding HTML and JavaScript content into JavaFX applications.

**• Enable** JavaFX interoperability with Swing.

**• Rewrite the** media stack from the ground up.

**JavaFX 2.0 was released at** JavaOne 2011, and has enjoyed a greatly increased adoption rate due to the innovative features articulated previously. Now that you’ve had the obligatory history lesson in JavaFX, let’s get one step closer to writing code by showing you were some examples, tools, and other resources are.

# How does it work

The JavaFX is a set of Java libraries designed to enable developers to create and deploy rich client applications that behave consistently across platforms.

The image bellow illustrates the architectural components of the JavaFX platform. The sections following the diagram describe each component and how the parts interconnect. Below the JavaFX public APIs lies the engine that runs your JavaFX code. It is composed of subcomponents that include the new JavaFX high performance graphics engine, called Prism; the new small and efficient windowing system, called Glass; a media engine, and a web engine.



^ Figure1

We will now follow with a description of the most important components from above

## Scene graph

The JavaFX scene graph, shown as part of the top layer in Figure 1 is the starting point for constructing a JavaFX application. It is a hierarchical tree of nodes that represents all of the visual elements of the application's user interface. It can handle input and can be rendered.

A single element in a scene graph is called a node. Each node has an ID, style class, and bounding volume. With the exception of the root node of a scene graph, each node in a scene graph has a single parent and zero or more children. It can also have the following:

* Effects, such as blurs and shadows
* Opacity
* Transforms
* Event handlers (such as mouse, key and input method)
* An application-specific state

Unlike in Swing and Abstract Window Toolkit (AWT), the JavaFX scene graph also includes the graphics primitives, such as rectangles and text, in addition to having controls, layout containers, images and media.

For most uses, the scene graph simplifies working with UIs, especially when rich UIs are used. Animating various graphics in the scene graph can be accomplished quickly using the javafx.animation APIs, and declarative methods, such as XML doc, also work well.

The javafx.scene API allows the creation and specification of several types of content, such as:

* **Nodes**: Shapes (2-D and 3-D), images, media, embedded web browser, text, UI controls, charts, groups, and containers
* **State**: Transforms (positioning and orientation of nodes), visual effects, and other visual state of the content
* **Effects**: Simple objects that change the appearance of scene graph nodes, such as blurs, shadows, and color adjustment

## Graphics system

The JavaFX Graphics System, shown in blue in Figure 1, is an implementation detail beneath the JavaFX scene graph layer. It supports both 2-D and 3-D scene graphs. It provides software rendering when the graphics hardware on a system is insufficient to support hardware accelerated rendering.

Two graphics accelerated pipelines are implemented on the JavaFX platform:

* **Prism** processes render jobs. It can run on both hardware and software renderers, including 3-D. It is responsible for rasterization and rendering of JavaFX scenes. The following multiple render paths are possible based on the device being used:
  + DirectX 9 on Windows XP and Windows Vista
  + DirectX 11 on Windows 7
  + OpenGL on Mac, Linux, Embedded
  + Java2D when hardware acceleration is not possible. The fully hardware accelerated path is used when possible, but when it is not available, the Java2D render path is used because the Java2D render path is already distributed in all of the Java Runtime Environments (JREs). This is particularly important when handling 3-D scenes. However, performance is better when the hardware render paths are used.
* **Quantum Toolkit** ties Prism and Glass Windowing Toolkit together and makes them available to the JavaFX layer above them in the stack. It also manages the threading rules related to rendering versus events handling.

## Glass window toolkit

The Glass Windowing Toolkit, shown in beige in the middle portion of [Figure 1](http://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm#BABDFFDG), is the lowest level in the JavaFX graphics stack. Its main responsibility is to provide native operating services, such as managing the windows, timers, and surfaces. It serves as the platform-dependent layer that connects the JavaFX platform to the native operating system.

The Glass toolkit is also responsible for managing the event queue. Unlike the Abstract Window Toolkit (AWT), which manages its own event queue, the Glass toolkit uses the native operating system's event queue functionality to schedule thread usage. Also unlike AWT, the Glass toolkit runs on the same thread as the JavaFX application. In AWT, the native half of AWT runs on one thread and the Java level runs on another thread. This introduces a lot of issues, many of which are resolved in JavaFX by using the single JavaFX application thread approach.

### Threads

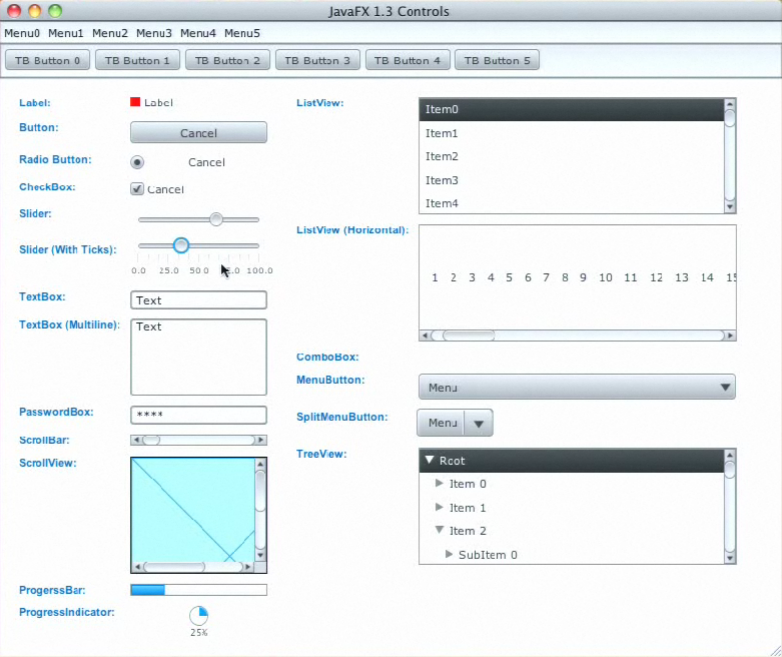
The system runs two or more of the following threads at any given time.

* **JavaFX application thread**: This is the primary thread used by JavaFX application developers. Any “live” scene, which is a scene that is part of a window, must be accessed from this thread. A scene graph can be created and manipulated in a background thread, but when its root node is attached to any live object in the scene, that scene graph must be accessed from the JavaFX application thread. This enables developers to create complex scene graphs on a background thread while keeping animations on 'live' scenes smooth and fast. The JavaFX application thread is a different thread from the Swing and AWT Event Dispatch Thread (EDT), so care must be taken when embedding JavaFX code into Swing applications.
* **Prism render thread**: This thread handles the rendering separately from the event dispatcher. It allows frame N to be rendered while frame N +1 is being processed. This ability to perform concurrent processing is a big advantage, especially on modern systems that have multiple processors. The Prism render thread may also have multiple rasterization threads that help off-load work that needs to be done in rendering.
* **Media thread**: This thread runs in the background and synchronizes the latest frames through the scene graph by using the JavaFX application thread.

### Pulse

A pulse is an event that indicates to the JavaFX scene graph that it is time to synchronize the state of the elements on the scene graph with Prism. A pulse is throttled at 60 frames per second (fps) maximum and is fired whenever animations are running on the scene graph. Even when animation is not running, a pulse is scheduled when something in the scene graph is changed. For example, if a position of a button is changed, a pulse is scheduled.

## UI Controls

The JavaFX UI controls available through the JavaFX API are built by using nodes in the scene graph. They can take full advantage of the visually rich features of the JavaFX platform and are portable across different platforms. JavaFX CSS allows for theming and skinning of the UI controls.

# Example of application design

## Use CDI

CDI is a modern design pattern that is for example part of JEE. By using dependency injection you can remove hard coded dependencies in your code. Here is a short example:

public class Controller {

@Inject

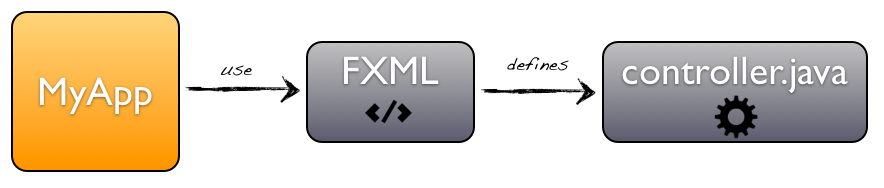
Model model;

public void action() {...}

}

If the Controller will be created by a CDI-Container the model field will be injected. This means that the CDI-Container will fill the field with a suitable instance of Model. How this model is created is not part of the Controller class. The controller only knows that he will get an injected instance of Model and can work with it.

## General design

In a normal business application you will have views and workflows. For example you will have an overview of some important business values and a form dialog with CRUD functions.  
Normally you will define your views by code or with the Scene Builder. If you use the Scene Builder you can transfer all your UI code into the fxml file and handle the logic inside a controller class. The controller can now access the business data by calling data stores or any other of your classes.  
[](http://www.guigarage.com/wordpress/wp-content/uploads/2013/05/cdi1.png)

This pattern is good and is working much better that your old Swing code. You have a clean separation between the view and the business logic. But there are still some problems here:

* The controller has hard dependencies to your business data classes.
* The controller class is defined inside the fxml file.
* Providing special business object instances only for one view workflow isn’t easy.

## A different approach



### FXML Controller annotation

First stept is to create the FXMLController annotation which will define the internal path to the fxml file.

@FXMLController("/ace/ucv/application/views/MyView.fxml")

public class MyController {

...

}

### Use the CDI magic

A first project that shows the use of CDI in JavaFX is [afterburner.fx](http://afterburner.adam-bien.com/) by [Adam Bien](http://about.adam-bien.com/). With this framework you can easily inject objects in your JavaFX controller.

Here is an example of our new simple controller:

@FXMLController("/ace/ucv/application/views/MyView.fxml")

public class MyController {

   @Inject

   Model myModel;

}

### The rest

After I added the @FXMLController annotation and CDI to my basic application it is really easy to create new views. The UI can be easily created by using the [Scene Builder](http://docs.oracle.com/javafx/scenebuilder/1/get_started/jsbpub-get_started.htm). Once this is done you can create the controller class and use CDI inside it. Cause you can inject everything that is needed it will be very easy to capsulate a view-controller-union as a single module. To create this I only need a line of code (as mentioned above):

Node myView FXMLCDI.create(MyController.class);

