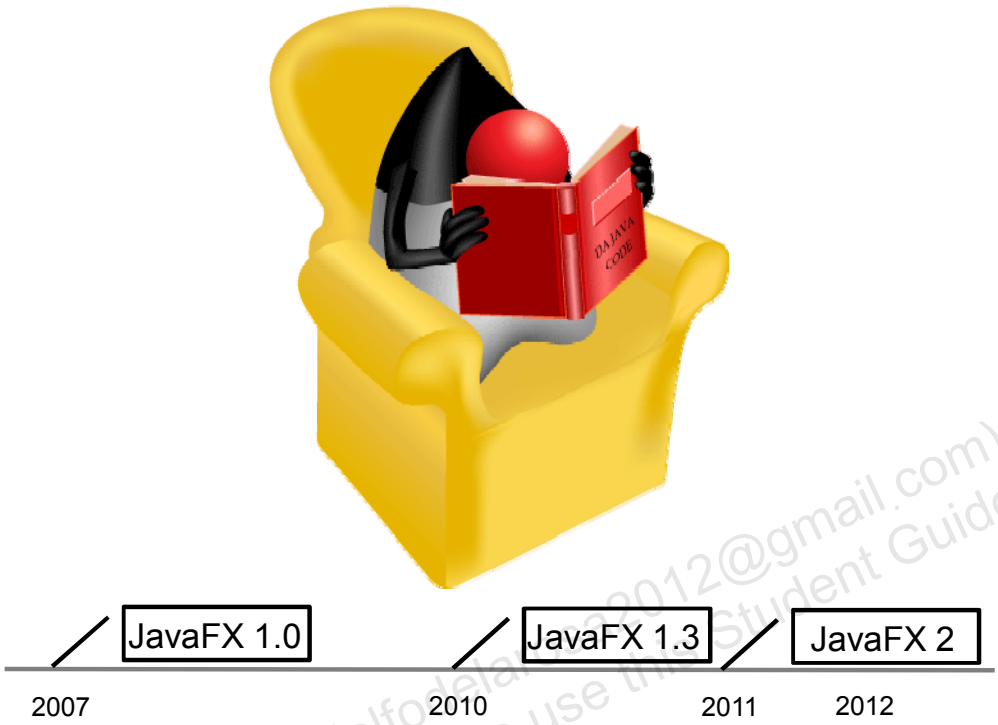
B

JavaFX History and Architecture

ORACLE

The Story of JavaFX

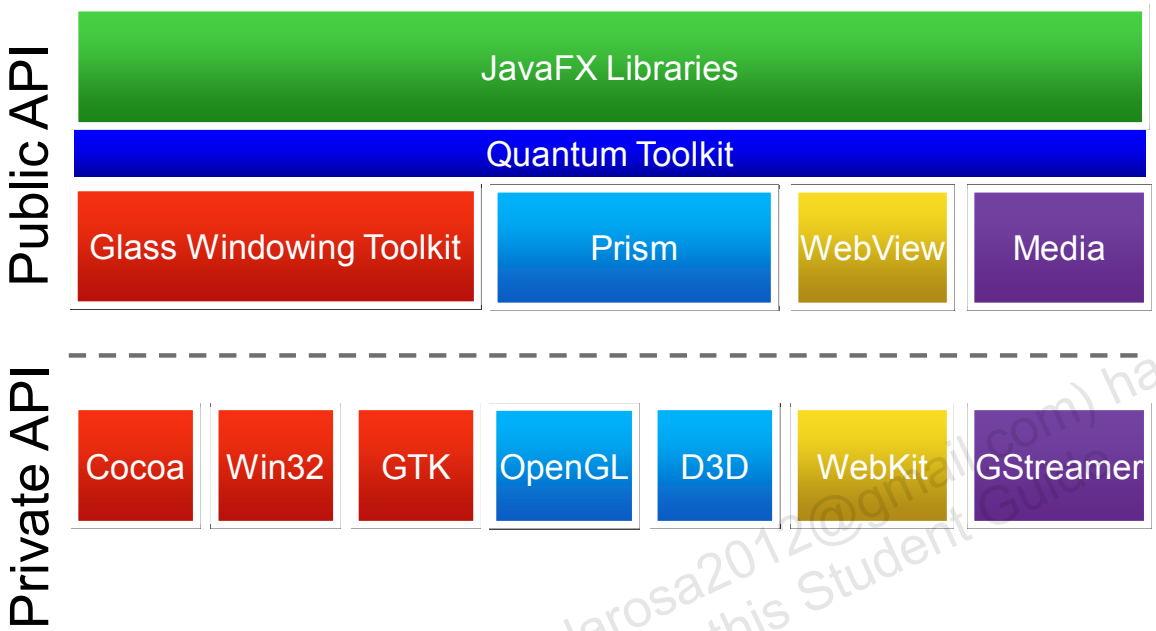


JavaFX started life as F3 (form follows function), which was the brainchild of Chris Oliver when he worked for a company named SeeBeyond. SeeBeyond was acquired by Sun Microsystems, who renamed F3 to JavaFX, and announced it at JavaOne 2007. Chris joined Sun during the acquisition and continued to lead 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.

At the JavaOne 2007 conference, Sun Microsystems introduced the JavaFX platform to help content developers and application developers to create content-rich applications for mobile devices, desktops, televisions, and other consumer devices. The initial offering consisted of the JavaFX Mobile platform and the JavaFX Script language. Multiple public releases were delivered after the initial announcement; the 1.3 version was released in April 2010.

After Oracle's acquisition of Sun Microsystems, Oracle announced during the JavaOne 2010 conference that support for the JavaFX Script language would be discontinued. However, it was also announced that the JavaFX Script APIs would be ported to Java and would be released as part of the JavaFX 2 platform. This announcement meant that the JavaFX capabilities would become available to all Java developers, without the need for them to learn a new scripting language. With this announcement, Oracle committed to making JavaFX the premier environment for rich client applications.

JavaFX Architecture and APIs



The diagram describes 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. In addition, there are several private APIs available through JavaFX, and although these components are not exposed publicly, their descriptions can help you to better understand what runs a JavaFX application.

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.

The Glass Windowing Toolkit is the lowest level framework for the JavaFX 2 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.

AWT and Glass

AWT	Glass Windowing Toolkit
Manages its own event queue	Uses native operating system's event queues
Multi-threaded approach (Native half of AWT runs on one thread and the Java level runs on another thread)	Single-threaded approach (runs on the same thread as the JavaFX application)