Building OpenJFX

Building a UI toolkit for many different platforms is a complex and challenging endeavor. It requires platform specific tools such as C compilers as well as portable tools like Gradle. Which tools must be installed differs from platform to platform. While the OpenJFX build system was designed to remove as many build hurdles as possible, it is often necessary to build native code and have the requisite compilers and toolchains installed. On Mac and Linux this is fairly easy, but setting up Windows is more difficult.

For a ready made build, try the Community Build page or the JDK9 early access.

Before you start

Do you really want to build OpenJFX ? We would like you to, but there are some great community builds that may work for you too. JavaFX is bundled by default in desktop editions of JDK8 and the early access JDK9.

Before you start to build OpenJFX yourself, you will need to know which build you are going to need. While there are many common elements, Java 8 and Java 9 JFX can be quite different to build and use.

With Java 8, JFX is bundled, but could be considered an overlay. In fact, building OpenJFX using the OpenJFX repository for JDK 8 produced an output that could indeed just be used as an overlay on top of a JDK 8 image.

In Java 9 and the module system, FX is now an integral part of the runtime environment - at least for the desktop. (JavaFX is not part of the ARM JRE for example). Because of this tie, there is no provision or capability for the output of the OpenJFX 9 build to be used as an overlay. It is still possible however to develop and enhance OpenJFX, and use that result to build an OpenJDK.

Platform Prerequisites

Windows

You need to have the following tools installed:

- Cygwin. Some packages to make sure are installed are:
    - openssh
    - bison
    - flex
    - g++
    - gperf
    - make
    - makedepend
    - mercurial

    - perl
    - zip
    - unzip
- DirectX SDK June 2010. Microsoft DirectX SDK (June 2010) headers are required for building the JavaFX SDK. This DirectX SDK can be downloaded from Microsoft DirectX SDK (June 2010). If the link above becomes obsolete, the SDK can be found from the Microsoft Download Site (search with "DirectX SDK June 2010"). The location of this SDK will normally be set with the environment variable DXSDK_DIR at installation time. The default location is normally "C:/Program Files/Microsoft DirectX SDK (June 2010)/". If DXSDK_DIR is not set, the build process may look for it in the default location or "C:/DXSDK/".
- Microsoft Visual Studio 10 SP1 (express edition works). The compiler and other tools are expected to reside in the location defined by the variableVS100COMNTOOLS which is set by the Microsoft Visual Studio installer.

Mac

To configure your Mac, make sure you have at least version 10.7 installed. Install the latest version of Xcode and that you have the developer tools installed. You can install them by using the menus within Xcode: XCode -> Preferences -> Downloads -> Components. Install the latest JDK 8 build. In order to build WebKit, you will also need to install QT 5.2 (because WebKit uses QMake).

IMPORTANT: If you have a different version of X code (say one that is compatible with OS X 10.9), you will need to add the following line to your `~/.gradle/gradle.properties` file:

MACOSX_MIN_VERSION=10.9

Depending on the version of X code that you have, the value of MACOSX_MIN_VERSION may need to be different (ie. 10.8). If you do not set this variable correctly, the C code will not build.

Linux

Setting up a Linux build configuration is fairly straightforward. These build instructions were used for the "official" build platform of Ubuntu 10.04, but also on the latest Ubuntu 12.10. First, run the following command to install all the required development packages:

Ubuntu 14.04, 15.10, 16

```
sudo apt-get update
sudo apt-get install ksh bison flex gperf libasound2-dev libgl1-mesa-dev \
    libgstreamer0.10-dev libgstreamer-plugins-base0.10-dev libjpeg-dev \
    libpng-dev libx11-dev libxml2-dev libxslt1-dev libxt-dev \
    libxxf86vm-dev pkg-config x11proto-core-dev \
    x11proto-xf86vidmode-dev libavcodec-dev mercurial \
libgtk2.0-dev libgtk-3-dev \
    libxtst-dev libudev-dev libavformat-dev
```

You will also need to install QT 5.2 in order to build WebKit (because WebKit uses Qmake). With Ubuntu 16, this will satisfy the requirements:

```
sudo apt-get install cmake ruby
```

Ubuntu 14.10

 currently not recommended for building ARM because of packaging conflicts with libgl1-mesa-dev and the compatibility libraries needed for ARM.

Oracle Enterprise Linux 7 and Fedora 21

```
yum install mercurial bison flex gperf ksh pkgconfig \
    libpng12-devel libjpeg-devel libxml2-devel \
libxslt-devel systemd-devel glib2-devel  gtk2-devel \
libXtst-devel pango-devel freetype-devel
```

Linux ARM

Building OpenJFX for Linux ARM has only been tested on as a cross build from Linux and MacOSX. The process is only regularly used on Linux. Follow the steps for a Linux build setup first, and then refer to the steps for Cross Building for ARM Hard Floa t.

Common Prerequisites

Java SE 8 for developing for Java 8

Since each release of OpenJFX is paired with a corresponding release of the JDK, you should make sure that you have a recent (preferably the

latest) promoted build of the JDK available.The current supported build can be downloaded from the Java SE 8 download page. Some make it a practice to always run against the latest promoted build, others will stick with an older build until they finally can't build OpenJFX with it anymore, and then update. Whichever method you chose, you need to have a reasonably recent version of JDK 8 installed.

The OpenJFX build requires a Java JDK 8 that does not have the JFX jar present. The build scripts check for this condition, and will refuse to continue if it is found.This jar is found in the JDK at (your JDK)/jre/lib/ext/jfrt.jar. A common practice is to have a copy the JDK for general use, and another for building, and then remembering to set your PATH when building to the proper JDK. There are many ways to copy the JDK, and which one to use will depend on which OS you are using.

Some versions of Linux may have an installed Java that is not new enough, and may already be in your path, which will cause confusion as you try to build.

Make sure you have the right Java in your path with:

```
java -version
```

Java SE 9 for JDK 9

Download and install the current early access from JDK9 Early Access Download. This version will be used for testing clients and for unit test runs. (And eventually for compilation). Current minimun is build 148 (check with java -version). Note that this version will change as we move closer to the release.

Mercurial

OpenJFX, as with OpenJDK, uses Mercurial as the source control system. You must install some support for using Mercurial. Many (if not all) IDEs include built in support, although the tooling is generally not as good as you might get from a standalone tool.

For Linux, the Mercurial package is included in the list of required packaged that were installed.

Popular options include SourceTree from Atlassian, TortoiseHg for Windows, or the command line tools from Mercurial.

Gradle

You must also install Gradle. We are using Gradle 3.1 for 9-dev, and Gradle 1.8 for the older 8-dev (IMPORTANT: Only these versions are regularly tested).

Note: gradle is available as a Ubuntu package, but check the version. This command should work after you set JAVA_HOME:

```
gradle -version
```

Environment Variables

At a minimum, you will need to have gradle in your path.

For JFX 8

- set JAVA_HOME and JDK_HOME to point to the top of the JDK 8 installation (that has jfxrt.jar removed).
- set gradle 2.11 in your path
- ensure Ant 1.8.2 is in your path

For JFX 9

- set JAVA_HOME and JDK_HOME to point to the top of the JDK 9 ea 148+.
- set gradle 3.1+ in your path
- ensure Ant 1.8.2 is in your path
- set the following env variable to allow gradle to run with the latest JDK 9:

```
export _JAVA_OPTIONS="-Dsun.reflect.debugModuleAccessChecks=true
--add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.util=ALL-UNNAMED
--add-opens=java.base/java.lang.invoke=ALL-UNNAMED --add-opens=java.base/java.io=ALL-UNNAMED
--add-opens=java.base/java.util.concurrent=ALL-UNNAMED
--add-opens=java.base/java.text=ALL-UNNAMED"
```

Note: on windows, these paths MUST be in dos format, though you can use the 'right' slashes ('/'). Test your settings with:

```
"$JAVA_HOME/bin/java" -version

    "$JIGSAW_HOME/bin/java" -version

    gradle -version

    ant -version
```

Getting the Sources

All OpenJFX sources are held in mercurial repositories. As mentioned in Repositories and Releases, we have several different repositories for you to choose from.

```
# for 8u-dev the "stable" stream matching JDK8
hg clone http://hg.openjdk.java.net/openjfx/8u-dev/rt

# for the active development stream targeted for JDK9
hg clone http://hg.openjdk.java.net/openjfx/9-dev/rt
```

(Note: Historically you also had to clone the "jfx" repository in the forest that you cared about. However we have modified our approach, such that we no longer promote the use of a forest, and instead are putting all of our sources in a single repository, presently named "rt").

Using Gradle on The Command Line

Before diving directly into building OpenJFX, lets get our feet wet by learning what kinds of things we can call from the command line, and how to get help when we need it. The first command you should execute is *tasks*:

```
rbair$ gradle tasks
The CompileOptions.useAnt property has been deprecated and is scheduled to be removed
in Gradle 2.0. There is no replacement for this property.
:tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------


Default tasks: assemble


Basic tasks
-----------
clean - Deletes the build directory and the build directory of all sub projects
javadoc - Generates the JavaDoc for all the public API
jfxrt - Creates the jfxrt.jar
sdk - Creates an SDK


Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
ccDecora - Compiles native sources for Decora
ccGlass - Compiles native sources for Glass
ccPrism - Compiles native sources for Prism
ccPrismSW - Compiles native sources for PrismSW
classes - Assembles the main classes.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
javahDecora - Generates JNI Headers for Decora
javahGlass - Generates JNI Headers for Glass
javahPrism - Generates JNI Headers for Prism
```

```
javahPrismSW - Generates JNI Headers for PrismSW
linkDecora - Creates native dynamic library for Decora
linkGlass - Creates native dynamic library for Glass
linkPrism - Creates native dynamic library for Prism
linkPrismSW - Creates native dynamic library for PrismSW
native - Compiles and Builds all native libraries for Graphics
nativeDecora - Generates JNI headers, compiles, and builds native dynamic library for
Decora
nativeGlass - Generates JNI headers, compiles, and builds native dynamic library for
Glass
nativePrism - Generates JNI headers, compiles, and builds native dynamic library for
Prism
nativePrismSW - Generates JNI headers, compiles, and builds native dynamic library for
PrismSW
stubClasses - Assembles the stub classes.
testClasses - Assembles the test classes.


Documentation tasks
-------------------
javadoc - Generates Javadoc API documentation for the main source code.


Help tasks
----------
dependencies - Displays all dependencies declared in root project 'javafx'.
dependencyInsight - Displays the insight into a specific dependency in root project
'javafx'.
help - Displays a help message
projects - Displays the sub-projects of root project 'javafx'.
properties - Displays the properties of root project 'javafx'.
tasks - Displays the tasks runnable from root project 'javafx' (some of the displayed
tasks may belong to subprojects).


IDE tasks
---------
cleanIdea - Cleans IDEA project files (IML, IPR)
cleanIdeaWorkspace - Deletes the javafx.ipw file
cleanNetBeans - Deletes generated NetBeans files
idea - Generates IDEA project files (IML, IPR, IWS)
netBeans - Creates the NetBeans project files for JavaFX


Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.


To see all tasks and more detail, run with --all.


BUILD SUCCESSFUL
```

```
Total time: 4.883 secs
```

The *tasks* task is extremely helpful. You use it to discover all the other things you can do with this build file. You notice at the top of the output the phrase "All tasks runnable from root project". The "root" project is "javafx". That is, we are in the root project. Below the root project are a series of sub projects, some of which are referred to as modules or "components". But more about those later.

Gradle then tells us what the default tasks are. In this case, our default task is the 'sdk' task. This is the task that will be executed if you just call 'gradle' alone without providing any additional arguments. After this comes a listing of different tasks, broken out by group. The first group is the "Basic" group which contains the tasks you may find yourself using most often. These are all named and have a description provided. For example, if I wanted to execute the 'clean' task, then I would do so like this:

```
rbair$ gradle clean
```

Finally, the *tasks* task gives us a useful hint that we can pass the --all argument in order to see all of the tasks in more detail. This produces a lot more output, but really gives an in depth look at what tasks are available for you to call.

I mentioned above that our root project is called "javafx", and that we have sub-projects in the gradle build. To see all of the projects available to you, execute the *projects* task (which you will notice was in the "Help tasks" group produced by the *tasks* task). This lists not just what projects are available, but what their name is, and what the project hierarchy is.

```
rbair$ gradle projects
The CompileOptions.useAnt property has been deprecated and is scheduled to be removed
in Gradle 2.0. There is no replacement for this property.
:projects


------------------------------------------------------------
Root project
------------------------------------------------------------


Root project 'javafx'
+--- Project ':base'
+--- Project ':build-tools'
+--- Project ':controls'
+--- Project ':designTime'
+--- Project ':fxml'
+--- Project ':graphics'
|    +--- Project ':graphics:effects-jsl'
|    \--- Project ':graphics:prism-jsl'
+--- Project ':swing'
\--- Project ':swt'
To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :base:tasks


BUILD SUCCESSFUL


Total time: 4.194 secs
```

Projects in gradle are named according to their depth. So the root project is simply named "javafx". The immediate subprojects are all prefixed with a ":". Sub-subprojects have their parents in their name, for example, ":graphics:effects-jsl". When you execute a command such as *gradle assemble* what actually happens is that Gradle locates the *assemble* task on all projects and executes them. (TODO Is this entirely accurate?)

There are a couple other tricks-of-the-trade that you should be aware of. You can execute any gradle command with --info or --debug in order to

get more output. Running in --info mode provides some additional debugging output that is very useful when things go wrong. In particular, our build system will output certain crucial variables that are being used to perform the build:

```
rbair$ gradle projects
Starting Build
Settings evaluated using settings file
'/Users/rbair/Projects/JavaFX/graphics-8.0/javafx/settings.gradle'.
Projects loaded. Root project using build file
'/Users/rbair/Projects/JavaFX/graphics-8.0/javafx/build.gradle'.
Included projects: [root project 'javafx', project ':base', project ':build-tools',
project ':controls', project ':designTime', project ':fxml', project ':graphics',
project ':swing', project ':swt', project ':graphics:effects-jsl', project
':graphics:prism-jsl']
Evaluating root project 'javafx' using build file
'/Users/rbair/Projects/JavaFX/graphics-8.0/javafx/build.gradle'.
OS_NAME: mac os x
JAVA_HOME: /Library/Java/JavaVirtualMachines/jdk1.8.0.jdk_b81/Contents/Home/jre
JDK_HOME: /Library/Java/JavaVirtualMachines/jdk1.8.0.jdk_b81/Contents/Home
BINARY_STUB:
file:///Library/Java/JavaVirtualMachines/jdk1.8.0.jdk_b81/Contents/Home/jre/lib/ext/jf
xrt.jar
HUDSON_JOB_NAME: not_hudson
HUDSON_BUILD_NUMBER: 0000
PROMOTED_BUILD_NUMBER: 00
PRODUCT_NAME: OpenJFX
RAW_VERSION: 8.0.0
RELEASE_NAME: 8.0
RELEASE_MILESTONE: ea
The CompileOptions.useAnt property has been deprecated and is scheduled to be removed
in Gradle 2.0. There is no replacement for this property.
Evaluating project ':base' using empty build file.
Evaluating project ':build-tools' using empty build file.
Evaluating project ':controls' using empty build file.
Evaluating project ':designTime' using empty build file.
Evaluating project ':fxml' using empty build file.
Evaluating project ':graphics' using empty build file.
Evaluating project ':swing' using empty build file.
Evaluating project ':swt' using empty build file.
Evaluating project ':graphics:effects-jsl' using empty build file.
Evaluating project ':graphics:prism-jsl' using empty build file.
All projects evaluated.
Selected primary task 'projects'
Tasks to be executed: [task ':projects']
:projects


------------------------------------------------------------
Root project
------------------------------------------------------------


Root project 'javafx'
+--- Project ':base'
+--- Project ':build-tools'
+--- Project ':controls'
+--- Project ':designTime'
+--- Project ':fxml'
+--- Project ':graphics'
|    +--- Project ':graphics:effects-jsl'
```

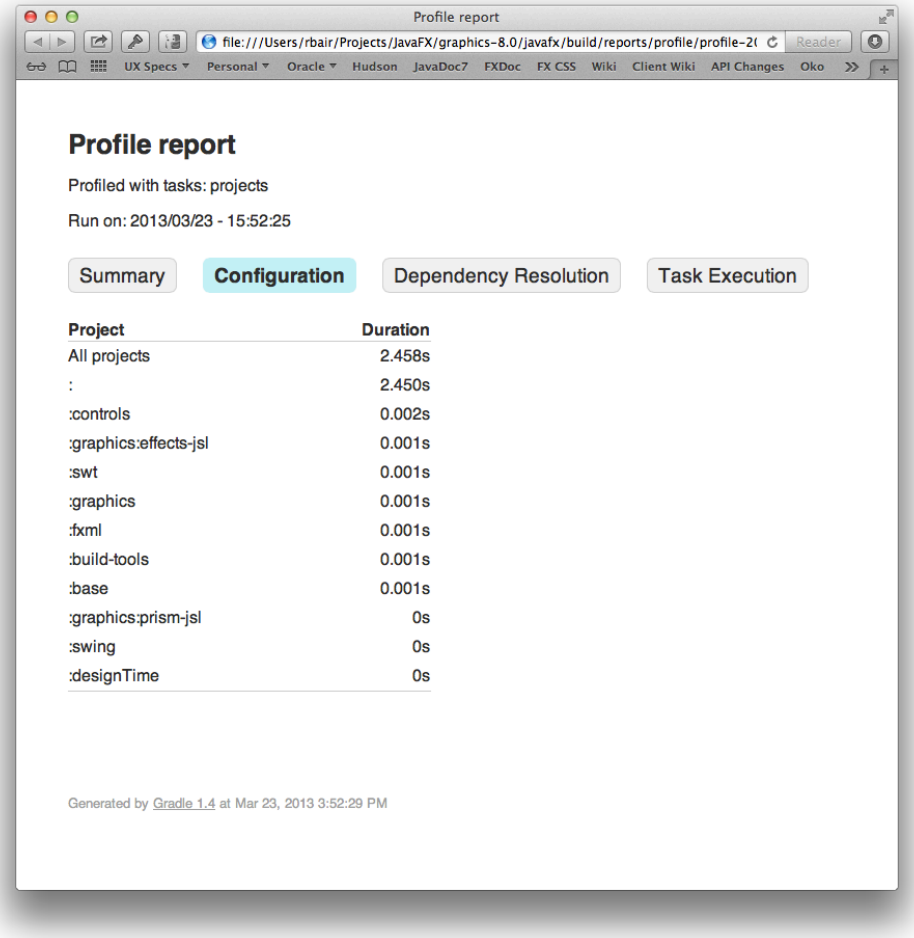
```
|    \--- Project ':graphics:prism-jsl'
+--- Project ':swing'
\--- Project ':swt'
To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :base:tasks


BUILD SUCCESSFUL
```

```
Total time: 4.194 secs
```

Among all this output is a list of several important properties, such as JDK_HOME. These properties are essential to the behavior of the build system, so if something goes wrong, you can check that you are building with the right binar stub and the right JDK (hint: nearly everything is based on JDK_HOME – if you have that set right, the rest of the Java build should *just work*).

One more trick is the --profile argument. You can perform any gradle task and use the --profile argument. This will cause gradle to keep track of how long various parts of the build took, and will produce an HTML report in build/reports/profile. The report breaks down how much time was spent in configuration, dependency resolution, and task execution. It further breaks it down by project. This gives useful metrics for tracking down which parts of the build take the longest and hopefully tighten up the build times.



Build and Test

There are three main things you may want to do on a regular basis when working on JavaFX: building, testing, and creating documentation. Lets look at each of these in turn.

The simplest basic task to build is the *sdk* task. The *sdk* task will compile all Java sources and all native sources for your target platform. It is the default task which is executed if you do not supply a specific task to run. It will create the appropriate sdk directory and populate it with the native dynamic libraries and the jfxrt.jar. Because the SDK is not distributed with documentation, the javadocs are not created as part of the *sdk* task by default. Once the *sdk* task has completed, you will have and SDK distribution which you could run against (modulo any closed-bits) or give to somebody else to run.

```
rbair$ gradle
The CompileOptions.useAnt property has been deprecated and is scheduled to be removed
in Gradle 2.0. There is no replacement for this property.
:base:processVersion
:build-tools:generateGrammarSource
:build-tools:compileJava
:build-tools:processResources
:build-tools:classes
:build-tools:jar
:base:compileJava
[snip out a whole bunch of stuff]
:jfxrt
:sdk


BUILD SUCCESSFUL


Total time: 1 mins 45.184 secs
```

You can find the built SDK in the build directory:

```
rbair$ pwd
/Users/rbair/open-jfx/graphics/javafx

rbair$ ls -l build/
drwxr-xr-x  3 rbair  staff  102 Mar 23 17:39 sdk
drwxr-xr-x  3 rbair  staff  102 Mar 23 17:39 tmp

rbair$ ls build/sdk/rt/lib/
ext/                    libdecora-sse.dylib    libprism-common.dylib
javafx.properties       libglass.dylib         libprism-sw.dylib
```

The *sdk* task will build an OpenJFX SDK for your particular Operating System. The "host" build will be named sdk, and any cross builds will have a prefix like armv6hf-sdk. Multiple different sdks may be built concurrently, and all will reside within the build directory when completed (see <<Cross Builds>> for more information). Gradle automatically handles the downloading of all dependencies (such as Antlr and SWT).

For more information on build properties, see Customizing the Build.

Cross Builds

The build is configured to support *cross builds*, that is, the ability to build an SDK for a platform other than the one you are building from. There are multiple gradle files located in buildSrc which represent specific *compile targets*. These include:

- win.gradle
- mac.gradle
- linux.gradle
- android.gradle
- ios.gradle
- armv6sf.gradle
- armv6hf.gradle

Each of these have specific prerequisites that must be met before they can be built. win.gradle can only be used on Windows, mac.gradle on Mac, and linux.gradle on Linux. Android can be cross built from Mac, Windows, or Linux so long as the Android SDK and NDK are installed and the build knows where to find them. iOS can be cross built on Mac. ARM (soft float and hard float) can be cross built from Linux.

By default, the OpenJFX build system will only build the SDK for the desktop platform you are building from. To ask it to build for a specific compile target, you must pass a COMPILE_TARGETS property to the build system, instructing it which to build. This is a comma separated list. Assuming you have already setup the prerequisites for building ARM (for example, when targeting the Raspberry PI), you would invoke gradle like

this:

```
rbair$ gradle -PCOMPILE_TARGETS=armv6hf
```

Customizing the Build

The build can be customized fairly extensively through the use of Gradle properties. Gradle provides many ways to supply properties to the build system. However the most common approach will be to use a gradle.properties file located in the rt directory. Simply make a copy of gradle.properties.template and then edit the resulting gradle.properties file to customize your build.

```
rbair$ cp gradle.properties.template gradle.properties
```

The gradle.properties file that you have just created is heavily documented and contains information on all the different configuration options at your disposal. Some of the most common are:

- Enabling building of native source code (Prism, Glass, GStreamer, WebKit, etc)
- Specifying the build configuration (Release or Debug)
- Enabling building of JavaDoc
- Customizing the JDK_HOME
- Supplying compiler LINT options

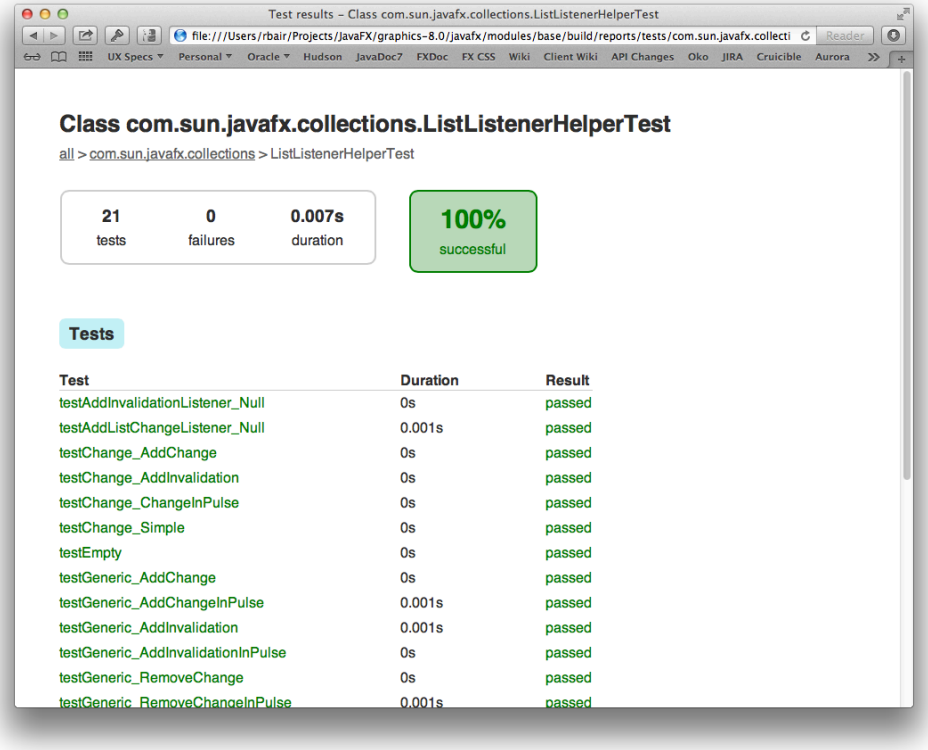
Arguably the most important property in the build is the JDK_HOME property. Almost all other properties are derived automatically from this one. The JDK_HOME is by default based on the java.home System property, which is set automatically by the JVM based on which version of Java is executed. Typically, then, the version of Java you will be using to compile with will be the version of Java you have setup on your path. You can of course specify the JDK_HOME yourself. Note also that on Windows, the version of the JDK you have set as JDK_HOME will determine whether you build 32 or 64 bit binaries.

Testing

The next basic task which you may want to perform is *test.* The *test* task will execute the unit tests. You generally will execute the top level *test* be cause unlike with Ant, Gradle will only re-execute those tests which have changed (or were dependent on code that was changed) on subsequent runs. You can of course execute *gradle cleanTest* in order to clean all the test results so they will run fresh. Or, if you want to execute only those tests related to a single project, you can do so in the normal fashion:

```
rbair$ gradle :base:test
The CompileOptions.useAnt property has been deprecated and is scheduled to be removed
in Gradle 2.0. There is no replacement for this property.
:base:processVersion UP-TO-DATE
:build-tools:generateGrammarSource UP-TO-DATE
:build-tools:compileJava UP-TO-DATE
:build-tools:processResources UP-TO-DATE
:build-tools:classes UP-TO-DATE
:build-tools:jar UP-TO-DATE
:base:compileJava UP-TO-DATE
:base:processResources UP-TO-DATE
:base:classes UP-TO-DATE
:base:compileTestJava UP-TO-DATE
:base:processTestResources UP-TO-DATE
:base:testClasses UP-TO-DATE
> Building > :base:test > 3411 tests completed, 45 skipped
```

Gradle gives helpful output during execution of the number of tests completed and the number skipped without dumping out lots of output to the console (unless you opt for --info). Also, once the tests complete, an HTML report is dumped to the project's build/reports/test directory (for example, modules/base/build/reports/test):



For the sake of performance, most of the tests are configured to run in the same VM. However some tests by design cannot be run in the same VM, and others cannot yet run in the same VM due to bugs or issues in the test. In order to improve the quality of the project we need to run as many tests as possible in the same VM. The more tests we can run on pre-integration the less likely we are to see failures leak into master. Being able to run 20,000 tests in a minute is extremely useful, but not possible, unless they run in the same VM. Something to keep in mind.

Overlay - JDK 8

After a successful build, the final step could be copying the results, overlaying them over an existing JDK. This step will replace any JavaFX/OpenJFX binaries in that runtime with your newly built binaries.

First execute the gradle task which creates an overlay bundle.

Zips

```
rbair$ gradle [-PCOMPILE_TARGETS=armv6hf] zips
# If you are cross compiling you will need the COMPILE_TARGETS flag
# with the appropriate target
# -PCOMPILE_TARGETS=armv6hf
```

This will create a zip bundle containing the OpenJFX binaries and is designed to be extracted into a JDK or JRE.

The created bundle is located in build/[*platform*-]bundles/javafx-sdk-overlay.zip. [*platform*-] will only be present for a cross build, and will be the name of the cross platform (for example armv6hf-bundles).

Please note that there might be some differences in the contents of the runtime found in build/[*platform*-]sdk/ and the contents of the bundles. The overlay bundle is designed to match JavaFX in a production JDK.

Within a JDK there is a directory that contains the Java Runtime Environment, "./jre". The zip bundle is designed to be extracted in the directory that contains "./jre".

As zip bundles do not always preserve permissions, sometimes it is necessary to modify the file permissions to match the others in the JRE. In particular, check the permissions on the extracted native libraries.

Sandbox Testing with JDK9

Using the results of a modular JDK9 OpenJFX build is quite simple. A "run" args file can be used to point to the overriding modules that are in your build. (args file support for java was added in JDK9) The file build/run.args and build/compile.args are created during the FX build process. The run.args file contains full paths to the overriding modules and shared libraries, and so must be recreated if you are using a copied or downloaded module set (for example from a nightly build). A script is provided that will recreate the xpatch.args file in the current directory:

```
bash tools/scripts/make_runargs.sh /absolute_path_to/modular-sdk
```

The following can be used to set up an alias that can be used to launch a JFX application, but using the FX binaries from your development tree. This alias will override the modules built into JDK9.

```
export JIGSAW_HOME="path_to_top_of_JDK9"

export JFX_BUILD="path_to_top_of_your_repo"

    export JFX_PATCH=$JFX_BUILD/build/run.args (or the path to one created by make_runargs.sh)

alias javafx='$JIGSAW_HOME/bin/java @$JFX_PATCH'
```

This alias uses the @argfile mechanism to include all that Xpatch/java.library.path verbosity to create a single command to run FX backed by your recently built binaries.

In Windows, the paths for the alias can be a bit tricky to get right, as the JDK wants native Windows paths, and cygwin often works better with a Unix path. Here is an example that works with Cygwin:

```
export JIGSAW_HOME=`cygpath -m "/cygdrive/c/Program Files/Java/jdk-9/"`

    export JFX_PATCH=`cygpath -m "$JFX_BUILD/build/run.args"`

     alias javafx='"$JIGSAW_HOME/bin/java" @$JFX_PATCH'
```

Integration with OpenJDK 9

With the module system in JDK 9, it is not possible to easily overlay an OpenJFX build over an existing JDK as was possible with JDK 8. It is possible to build an OpenJDK that included the updated OpenJFX modules.

To create an integrated OpenJDK 9 with OpenJFX requires two builds:

- OpenJFX for JDK 9
- OpenJDK 9, with a configure reference that includes your OpenJFX build.

The steps for building the OpenJDK are the same as for JDK 8. Use the repository path `http://hg.openjdk.java.net/jdk9/dev.`

Build OpenJFX first.

Configure the JDK with the following addition:

--with-import-modules=*_path_to_openjfx/9-dev*/build/modular-sdk

Then build the JDK as normal.

Understanding a JDK 9 Modular world in our developer build

The export of module packages is governed by two sets of files:

- module-info.java, the per module declarations
- module-info.java.extra, fragments of declarations used to augment standard JDK module-info.

During the build process, we generate some files for use by the build, and also by developers working in the sandbox.

- runs.args: for use with the java command, overrides as much as possible the FX modules in the JDK
    - must use absolute system paths internally, so cannot be easily copied without editing
    - can be rebuilt with tools/scripts/make_runargs.sh
    - cannot override with any local changes in module-info, so added packages may need an "--add-exports to be seen.
    - does not "re" grant any privileges that our default FX modules have with a security manager
- compile.args: arguments to allow for compile using the sandbox libraries
    - must use absolute system paths internally, so cannot be easily copied without editing
    - cannot override with any local changes in module-info, so added packages may need an "--add-exports to be seen.
- run.java.policy: a minimum permissions file to use with the security manager.
    - intended primarily as a base to start with before adding test specific permissions.

Each of these files has a "test" variant, for example "testrun.args". These files are altered to add in the "shims" version of the module. Note that the build/shims is not populated by the "sdk" task. Use the "copyGeneratedShims" or "test" task.

When dealing primarily with unit tests, additional arguments are needed to access non public API from within the unit tests. These additional

arguments have been placed in "addExports" that are local to the tests that need them. For example, "modules/javafx.graphics/src/test/addExports" contains all of the "--add-exports" clauses required to compile and run all of the graphics module junit tests. Care should be taken when modifying these files, as additions may mean that package module-info may need updates too. Keep in mind - if you are adding an "--add-exports" to ALL-UNNAMED so that a junit test can see the API, then the addExports the right place. If you are trying to fix access by another module, it likely is the wrong place.

Adding new packages in a modular world

JDK 9 Modules add complexity to the development chain, but particularly when adding new API and especially packages. Adding a new package or changing package visibility will be a multi step task that will require at least two change sets to implement.

Our developer sandbox build uses several items to work around module export during build and testing that you should be familiar with.

Create a "followup JBS" to cover the cleanup/removal of module access workarounds. Be sure to link this new followup JBS to the one you started with.

First Step - development

Modify affected modules module-info to reflect the proposed changes. These changes will only directly affect the current build java compile process. It is key to remember that the java ***runtime*** will ignore any changes to module-info, even while it uses "--patch-module".

The next modify buildSrc/addExport files to mirror changes that were made in the module-info files. Mark any additions with a comment containing the "Completion JBS" number, like this:

```
# to be removed by 81XXXXX

--add-exports=javafx.graphics/com.sun.javafx.newpackage=javafx.controls
--add-exports=javafx.graphics/com.sun.javafx.newpackage=ALL-UNNAMED
```

Note, if you add a junit test that for the new package, you will likely also need an export to ALL-UNNAMED (which the junit jar is a member of). The result may be two exports in buildSrc/addExport to add the temporary workarounds required for both development and test. If you are not modifying unit tests - do not add the ALL-UNNAMED line.

Complete development of your new package and adding unit test coverage, and all of the other process we normally do.

Your complete change set will now contain all of the delta required for the nightly build and test your changes. The promotion process will soon merge your module-info changes into the JDK. Once there is a promoted JDK that has the new module-info changes, it is possible to move to the second step.

One consideration - building a local development copy of the JDK is not difficult. In some cases, it may be useful to create a local developer JDK that incorporates the module-info changes, even before development of the changeset it complete. This developer JDK will honor the new package exports without the need of the changes to the addExport files. Note however, your change set may break the build if it has not be tested with the current minimum promoted JDK build.

Second Step - cleanup

Once the changes are promoted into a JDK, the second step to remove the addExports workarounds can be scheduled with the team lead.

As both the build machine and the other developers will need to update to the newer JDK build, this step will need to be coordinated.

Create a change set with:

- the now unneeded addExport lines removed
- the minimum JDK version used by the build has been updated to the new minimum

Test, review and commit as normal.