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 and the JDK. 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 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.

If you are looking for instructions to build FX for JDK 8uNNN, they have been archived here.

Before you start

Do you really want to build OpenJFX? We would like you to, but the latest stable build is already available on the JavaFX website, and JavaFX 8 is bundled by default in Oracle JDK 8 (9 and 10 also included JavaFX, but were superseded by 11, which does not). There are also some great community builds that may work for you.

We are exploring making this easier, by enabling a developer to build a set of javafx.* modules that can be used with a clean OpenJDK build (without the javafx.* modules). Stay tuned.

Platform Prerequisites

Building WebKit as part of building JavaFX is optional and requires additional steps; these are detailed per operating system below. If you do not build WebKit, you can use pre-built libraries as detailed here.

Windows

You will need Windows 10 or later (Windows 10 is recommended) 64-bit OS.

You need to have the following tools installed:

- Cygwin. Some packages to make sure are installed are:
    - `openssh`
    - `zip`
    - `unzip`
    - `make` (needed to compile media)
    - `makedepend` (needed for media)
    - Optional: `git`
- Microsoft Visual Studio 2022. You can use the Enterprise, Professional, or Community edition or the command line BuildTools. The `Desktop development with C++` workload is required at most, but it may be possible to install individual components to satisfy the requirements.

If you build WebKit (it is not built by default) you will need the following additional tools:

- Cmake 3.22.3 or later, available from the Cmake download site
- Additional Cygwin tools:
    - `bison`
    - `flex`
    - `gperf`
    - `perl (5.10 or later)`
    - `python3`
    - `ruby (2.5 or later)`

All commands on this page are run inside Cygwin (and not in Windows CMD).

The JavaFX build will automatically locate your Visual Studio installation, as long as you installed it in the default location. You no longer need to set any env variables to point to your VS 2022 installation, unless you installed Visual Studio in a non-standard location, for example, the `D:` drive instead of the default `C:` drive.

Missing paths issue

The initial build process that generates the needed resources is done by the `buildSrc` folder. On Windows, it tries to locate all the needed tools and write their paths to the `build\windows_tools.properties` file. If it fails, the file is left blank, which results in a fatal error. In this case, define the `VSCOMNTOOLS` variable (older versions of JavaFX used `VS150COMNTOOLS`) to point to the `VC\Auxiliary\Build` directory in your Visual studio Installation. For example, use the following if you installed Visual Studio 2022 on the `D:` drive.

```
export VSCOMNTOOLS="D:\\Program Files\\Microsoft Visual Studio\\2022\\Community\\VC\\Auxiliary\\Build"
```

Note the use of the double backslash in the `VSCOMNTOOLS` env var. This is needed because the cygwin shell uses the '\' as an escape character.

If these definitions aren't persisted between launches of Cygwin, you can either set them in the Windows Environment Variables UI or in the `/home/$user$/.bash_profile` file (these are ran on startup). Use `export -p` to verify that the env variables are set correctly.

Mac

You will need macOS 12 (Monterey) or later.

Install the following software:

- Xcode 14 or later (14.3 is recommended)
- Xcode developer command line tools – you can install them by using the menus within Xcode: XCode -> Preferences -> Downloads -> Components
- git

If you build WebKit (it is not built by default) you will need the following additional tools:

- Cmake 3.22.3 or later, available from the Cmake download site
- gperf

Linux

Setting up a Linux build configuration is fairly straightforward. These build instructions were used for Ubuntu 18.04.

Ubuntu 18.04

First, run the following command to install all the required development packages:

```
sudo apt-get update
sudo apt-get install libavformat-ffmpeg57 libgl1-mesa-dev \
    libx11-dev pkg-config x11proto-core-dev git \
    libgtk2.0-dev libgtk-3-dev
```

If you build WebKit (it is not built by default) you will need the following additional tools:

- Cmake 3.22.3 or later, available from the Cmake download site
- bison
- flex
- gperf
- perl (5.10 or later)
- python3
- ruby (2.5 or later)

The following should satisfy the requirements (but check the version of cmake) :

```
sudo apt-get install cmake bison flex gperf ruby
```

Ubuntu 20.04

Same as Ubuntu 18.04 with the following changes for `sudo apt-get install`:

1. Change `libavformat-ffmpeg57` to `libavformat58`
2. Add `libxxf86vm-dev`

Oracle Enterprise Linux 7 and Fedora 21

We use Oracle Linux 7 to build the javafx.* modules that we ship with the Oracle JDK releases. Here are the packages you will need:

```
yum install mercurial git bison flex gperf pkgconfig \
    gtk2-devel gtk3-devel pango-devel freetype-devel
```

CentOS 8

Run the following commands (using Java 11 here as an example):

1. `sudo yum update`
2. `sudo yum install git bison flex pkgconfig gtk2-devel gtk3-devel \`
   `pango-devel freetype-devel libXtst-devel java-11-openjdk-devel ant gcc-c++`
3. `sudo yum install epel-release`
4. `sudo yum config-manager --set-enabled PowerTools`
5. `sudo yum update`
6. `sudo yum install libstdc++-static`
7. `sudo alternatives --config java`

   (specify Java 11)

Common Prerequisites

OpenJDK

OpenJFX N is formally compatible with JDK N and N-1. For OpenJFX 22, download OpenJDK 21 or later to use as the boot JDK to build and test OpenJFX. We recommend to use the latest version, however, Gradle might not support that version, so a version that Gradle supports might also be required to run Gradle itself (though it will use the latest version of the JDK through toolchain support).

Git

OpenJFX (and OpenJDK) transitioned to Git as part of Project Skara. The OpenJFX repo is hosted on GitHub at openjdk/jfx. We encourage developers to become familiar with Git and GitHub.

Many (if not all) IDEs include built in support. For example, Eclipse uses EGit, which can be downloaded through the built-in update site http://download.eclipse.org/releases/latest/ under *Collaboration* > *Java implementation of Git*.

For Linux, the `git` package is included in the list of required packaged that were installed. On Windows, you can also install git as a Cygwin package.

Popular GUI options include SourceTree for Windows or Mac from Atlassian and TortoiseGit for Windows.

Gradle

Gradle is the primary build tool for building OpenJFX. Since the repository includes a Gradle wrapper that will download the correct Gradle version when needed, you do not need to manually install Gradle. The current and minimum Gradle versions are defined in the source code. If you want to generate a wrapper yourself (for example, you want to build OpenJFX with a different Gradle version), then you will need to install Gradle.

The `sh gradlew` command used throughout this document can be replaced with `gradle` when not using the wrapper.

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

```
gradle -version
```

Ant

You will need Apache Ant 1.10.5 to build the OpenJFX apps (IMPORTANT: there are known issues with ant 1.9.x, so use either version 1.10.5 or 1.8.2).

Environment Variables

Set the following environment variables:

- set `JAVA_HOME` and `JDK_HOME` to point to the root of your jdk-N release
- add `$JAVA_HOME/bin` to your `PATH`
- if you do not use the wrapper, add `gradle-x.y/bin` to your `PATH` where `x.y` is the version
- add `apache-ant-1.10.5/bin` to your `PATH`

Note: on windows, the JAVA_HOME and JDK_HOME variables ***must*** be in DOS format (e.g., "C:/Program Files/..." rather than "/cygdrive/c/Program Files/..."), although you can use forward slashes ('/'). Test your settings with:

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

IMPORTANT: Any time you change env settings or install new software after a failed build of JavaFX you should execute the following three commands:

```
sh gradlew --stop
rm -rf build
sh gradlew clean
```

The first is needed to stop any gradle daemons that might be running (by default gradle starts a daemon that is used to speed up subsequent builds). There was a bug in the gradle daemon that causes gradle to ignore any env variables set after the daemon is started (see JDK-8193288). Additionally, on Windows platforms, the gradle daemon can sometimes interfere with your ability to delete files that it keeps open. If you run into problems you can stop the gradle daemon with "gradle --stop" (or disable the gradle daemon altogether).

The second is needed because the OpenJFX build caches the results of a previous configuration, in such a way that it can cause gradle clean to fail.

Getting the Sources

All OpenJFX sources are held in https://github.com/openjdk/jfx (see Repositories and Releases). To clone the repo from the command line, use:

```
# for the active development stream, currently targeted for JDK 14
git clone https://github.com/openjdk/jfx.git
```

Other tools will have a clone option.

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*:

```
$ sh gradlew tasks
...
:tasks
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Default tasks: sdk

Basic tasks
-----------
buildModuleBaseWin - creates javafx.base property files
buildModuleGraphicsWin - copies javafx.graphics native libraries
buildModuleLibsWin
buildModuleMediaWin - copies javafx.media native libraries
buildModuleSWTWin - copies SWT JAR
buildModuleWebWin - copies javafx.web native libraries
clean - Deletes the build directory and the build directory of all sub projects
cleanAll - Scrubs the repo of build artifacts
javadoc - Generates the JavaDoc for all the public API
javafxSwtWin - Creates the javafx-swt.jar for the win target
sdkWin

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.
buildModulesWin
```

```
buildModuleWin
buildModuleZipWin
buildNeeded - Assembles and tests this project and all projects it depends on.
buildRunArgsWin
ccWinFont - Compiles native sources for font for win
ccWinGlass - Compiles native sources for glass for win
ccWinIio - Compiles native sources for iio for win
ccWinPrism - Compiles native sources for prism for win
ccWinPrismD3D - Compiles native sources for prismD3D for win
ccWinPrismES2 - Compiles native sources for prismES2 for win
ccWinPrismSW - Compiles native sources for prismSW for win
classes - Assembles main classes.
clean - Deletes the build directory.
cleanNative - Clean all native libraries and objects for Graphics
cleanNativeDecora - Clean native objects for Decora
cleanNativeFont - Clean native objects for font
cleanNativeGlass - Clean native objects for glass
cleanNativeIio - Clean native objects for iio
cleanNativePrism - Clean native objects for prism
cleanNativePrismD3D - Clean native objects for prismD3D
cleanNativePrismES2 - Clean native objects for prismES2
cleanNativePrismSW - Clean native objects for prismSW
createMSPfile
generateD3DHeaders - Generate headers by compiling hlsl files
jar - Assembles a jar archive containing the main classes.
jslcClasses - Assembles jslc classes.
linkWinFont - Creates native dynamic library for font for win
linkWinGlass - Creates native dynamic library for glass for win
linkWinIio - Creates native dynamic library for iio for win
linkWinPrism - Creates native dynamic library for prism for win
linkWinPrismD3D - Creates native dynamic library for prismD3D for win
linkWinPrismES2 - Creates native dynamic library for prismES2 for win
linkWinPrismSW - Creates native dynamic library for prismSW for win
native - Compiles and Builds all native libraries for Graphics
nativeDecora - Generates JNI headers, compiles, and builds native dynamic library for Decora
nativeFont - Generates JNI headers, compiles, and builds native dynamic library for font for all compile targets
nativeGlass - Generates JNI headers, compiles, and builds native dynamic library for glass for all compile
targets
nativeIio - Generates JNI headers, compiles, and builds native dynamic library for iio for all compile targets
nativePrism - Generates JNI headers, compiles, and builds native dynamic library for prism for all compile
targets
nativePrismD3D - Generates JNI headers, compiles, and builds native dynamic library for prismD3D for all
compile targets
nativePrismES2 - Generates JNI headers, compiles, and builds native dynamic library for prismES2 for all
compile targets
nativePrismSW - Generates JNI headers, compiles, and builds native dynamic library for prismSW for all compile
targets
rcFont - Compiles native sources for font
rcGlass - Compiles native sources for glass
rcIio - Compiles native sources for iio
rcPrism - Compiles native sources for prism
rcPrismD3D - Compiles native sources for prismD3D
rcPrismES2 - Compiles native sources for prismES2
rcPrismSW - Compiles native sources for prismSW
shadersClasses - Assembles shaders classes.
shimsClasses - Assembles shims classes.
stubClasses - Assembles stub classes.
testapp1Classes - Assembles testapp1 classes.
testapp2Classes - Assembles testapp2 classes.
testapp3Classes - Assembles testapp3 classes.
testapp4Classes - Assembles testapp4 classes.
testapp5Classes - Assembles testapp5 classes.
testapp6Classes - Assembles testapp6 classes.
testClasses - Assembles test classes.
toolsClasses - Assembles tools classes.

Build Setup tasks
-----------------
init - Initializes a new Gradle build.
wrapper - Generates Gradle wrapper files.
```

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

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in root project 'rt'.
components - Displays the components produced by root project 'rt'. [incubating]
dependencies - Displays all dependencies declared in root project 'rt'.
dependencyInsight - Displays the insight into a specific dependency in root project 'rt'.
dependentComponents - Displays the dependent components of components in root project 'rt'. [incubating]
help - Displays a help message.
model - Displays the configuration model of root project 'rt'. [incubating]
projects - Displays the sub-projects of root project 'rt'.
properties - Displays the properties of root project 'rt'.
tasks - Displays the tasks runnable from root project 'rt' (some of the displayed tasks may belong to
subprojects).

Publishing tasks
----------------
generateMetadataFileForJavafxPublication - Generates the Gradle metadata file for publication 'javafx'.
generateMetadataFileForMavenPublication - Generates the Gradle metadata file for publication 'maven'.
generatePomFileForJavafxPublication - Generates the Maven POM file for publication 'javafx'.
generatePomFileForMavenPublication - Generates the Maven POM file for publication 'maven'.
publish - Publishes all publications produced by this project.
publishJavafxPublicationToMavenLocal - Publishes Maven publication 'javafx' to the local Maven repository.
publishJavafxPublicationToMavenRepository - Publishes Maven publication 'javafx' to Maven repository 'maven'.
publishMavenPublicationToMavenLocal - Publishes Maven publication 'maven' to the local Maven repository.
publishMavenPublicationToMavenRepository - Publishes Maven publication 'maven' to Maven repository 'maven'.
publishToMavenLocal - Publishes all Maven publications produced by this project to the local Maven cache.

Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.
To see all tasks and more detail, run gradle tasks --all
To see more detail about a task, run gradle help --task <task>

BUILD SUCCESSFUL in 19s
1 actionable task: 1 executed
```

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 "rt". 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, executing the 'clean' task would be done like this:

```
$ sh gradlew 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.

As mentioned above, the root project is called "rt", 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.

```
$ sh gradlew projects
...
:projects
------------------------------------------------------------
Root project
------------------------------------------------------------
Root project 'rt'
+--- Project ':apps'
+--- Project ':base'
+--- Project ':controls'
+--- Project ':fxml'
+--- Project ':graphics'
+--- Project ':media'
+--- Project ':swing'
+--- Project ':swt'
+--- Project ':systemTests'
\--- Project ':web'
To see a list of the tasks of a project, run gradle <project-path>:tasks
For example, try running gradle :apps:tasks
BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

Projects in gradle are named according to their depth. So the root project is simply named "rt" (or whatever your top directory is named). 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.

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 or give to somebody else to run.

```
$ sh gradlew
...
:buildModulesLinux
:buildRunArgsLinux
:buildModules
:createTestArgfilesLinux
:sdkLinux
:sdk
BUILD SUCCESSFUL in 1m 48s
127 actionable tasks: 127 executed
```

You can find the built SDK in the build/modular-sdk directory:

```
$ pwd
/Users/kcr/jfx-dev/rt

$ ls -l build/
-rw-r--r-- 1 kcr kcr 1621 Dec 22 09:54 compile.args
drwxr-xr-x 2 kcr kcr 4096 Dec 22 09:54 libs/
-rw-r--r-- 1 kcr kcr   47 Dec 22 09:54 linux_freetype_tools.properties
-rw-r--r-- 1 kcr kcr  681 Dec 22 09:54 linux_gtk2.properties
-rw-r--r-- 1 kcr kcr  799 Dec 22 09:54 linux_gtk3.properties
-rw-r--r-- 1 kcr kcr  255 Dec 22 09:54 linux_pango_tools.properties
drwxr-xr-x 9 kcr kcr 4096 Dec 22 09:54 modular-sdk/
-rw-r--r-- 1 kcr kcr 1916 Dec 22 09:54 run.args
-rw-r--r-- 1 kcr kcr 1379 Dec 22 09:54 run.java.policy
-rw-r--r-- 1 kcr kcr 1304 Dec 22 09:54 test.java.policy
-rw-r--r-- 1 kcr kcr 1551 Dec 22 09:54 testcompile.args
-rw-r--r-- 1 kcr kcr 1846 Dec 22 09:54 testrun.args
drwxr-xr-x 3 kcr kcr 4096 Dec 22 09:54 tmp/
```

The *sdk* task will build an OpenJFX SDK for your particular platform. Gradle automatically handles the downloading of all dependencies (such as Antlr and SWT located under `\rt\build\libs`).

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

Platform Builds

NOTE: cross-build support is currently untested in the mainline jfx-dev/rt repo

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:

```
$ sh gradlew -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.

```
$ 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.

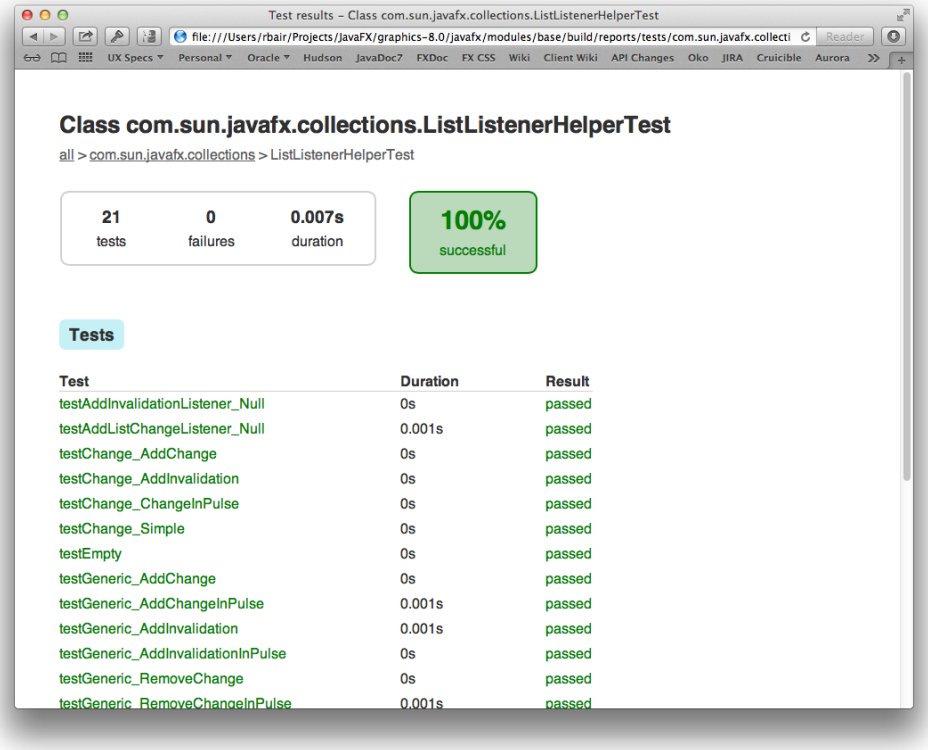
Arguably the most important property in the build is the JDK_HOME property, which will be set to the value of $JAVA_HOME if you haven't explicitly set it. Almost all other properties are derived automatically from this one. The JDK_HOME is by default based on the java.home System property, which is set 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 for all projects (all modules). If you want to execute only those tests related to a single project, you can do so in the normal fashion:

```
$ sh gradlew :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.

Running system tests with Robot

When running a system test that requires the Robot API, additional flags need to be passed:

```
sh gradlew -PFULL_TEST=true -PUSE_ROBOT=true :systemTests:test --tests TestClassName
```

Testing with JDK 9 or JDK 10

Using the results of a modular 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 JDK 9) 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 JAVA_HOME="path_to_top_of_JDK"

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='$JAVA_HOME/bin/java @$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 JAVA_HOME=`cygpath -m "/cygdrive/c/Program Files/Java/jdk-9/"`

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

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

Integration with OpenJDK

With the module system in JDK 9 and later, 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 with OpenJFX requires two builds:

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

See the following instructions for building OpenJDK. Use the following repository path: http://hg.openjdk.java.net/jdk/jdk.

Build OpenJFX first.

Configure the JDK with the following addition:

--with-import-modules=*_path_to_jfx-dev_*/rt/build/modular-sdk

Then build the JDK as normal.

Understanding a JDK 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

The JDK Module System adds 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.