OpenJFX unit tests

Understanding OpenJFX Unit tests.

OpenJFX contains many JUnit based unit tests that can be run using 'gradle test' as part of a build.

Source locations

There are two primary locations to find the unit tests in the repository:

- modules/*_module_name_*/src/tests
- tests/system/src/test/

In these locations, we have

- java/test/* - for all java source files for Unit tests
- java/* - if not in the test package, must be a shim class
- resources - for all related resources

In a non-modular build, there is nothing really separating the unit tests and the shims but convention.

The Shims

A shim is a java class that provides a backdoor into the core classes for testing purposes.They are special in a number of ways:

- they use the same extension classloader as the core classes
- they provide test entry points that circumvent restrictions on API visibility for unit testing.
- they usually must live in the same package as the API they are exposing (to avoid package private scope)
- they cannot use classes on the Application class loader - like the JUnit API (no import org.junit.* for example)
- with modules - they must live in a package referenced by the module-info for the class.

In the OpenJFX repository, the shims are any test class that does not reside in the test package.

It is best to keep these shim classes as simple as possible, reserving as much of the test logic in the Unit test classes as possible. Sometimes however, it makes sense to add more than just an accessor method, adding utility routines that may be easier to write from within the protected package.

With modules, the shim classes must live in a package already referenced in the module-info for the class, even if the package is not public. The addExports option can be used to loosen module restrictions, but not expose a package that is not already listed.

The Unit Tests

The unit tests run with the application class loader, and not the extension class loader as the core classes.

With modules, the unit tests run in the "unnamed" module by default. OpenJFX does not create a "test" module containing the tests.

All unit tests must be:

- JUnit tests
- part of the test package hierarchy (i.e. package test.*).
- Certain tests are optionally selected by gradle using run time options.
- @ignore(bug #) can be used to disable a test

Optional Tests with gradle toggles.

- -PFULL_TEST=true - long running tests, robot tests, or ones that flash windows making the host difficult to use
- -PUSE_ROBOT=true - enable Robot based tests
- -PHEADLESS_TEST=true - enable headless tests on some platforms
- -PAWT_TEST=true - enable AWT related tests
- -PUNSTABLE_TEST=true - enable tests tagged as unstable

All unit tests must:

- use the documented public API
- one of the shim classes
- in a few cases may be able to directly use internal API (like com.sun.*) classes with the right module addExport

When creating a new unit test, think about some of the special cases listed above and talk to the team about any special needs.

Currently all of the Robot based tests are in systemTests, and must live in the test.robot package.

Running a single test

Often a single test can be run for debugging using the --tests option to gradle:

gradle -PFULL_TEST=true -PUSE_ROBOT=true :systemTests:test --tests test.robot.javafx.embed.swing.RT32570Test

In this example, we want to run a single test in systemTests, which needs to be enabled with FULL_TEST

Disabling/Optional Unit Tests

There are times when a Unit test must be conditional or disabled. Here are some of the tricks for doing that.

The annotation @Ignore("*bug_id*") is used to disable a unit test on all platforms,

Unstable tests can be toggled using the conditions from within the test:

```
        assumeTrue(Boolean.getBoolean("unstable.test"));
```

Some tests are platform specific. These can be toggled with a line like this:

```
        assumeTrue(PlatformUtil.isMac() || PlatformUtil.isWindows());
```

Classpath

Gradle configures the classpath for each of the test modules. This classpath includes

- the JUnit jars
- each of the dependent modules (graphics has base tests in its classpath)

The dependent modules test classes are rarely used, but do contain items like the stub toolkit, so must be included.

Fun in a modular world

With the addition of modules in JDK9, there are some additional details that must be understood when dealing with the unit tests.

Module-info and the packages

It helps to review the module-info.java for the module you are working on to understand the visibility of the module.

The unit tests run in the "unnamed" module, and so by default can only see the public packages like this one:

```
      exports javafx.beans;
```

many packages are not exported as public like this one:

```
    exports com.sun.javafx to

        javafx.controls,

        javafx.graphics;
```

To access a class in this package, we will need to use an addExport option to override the default package protections for the test. Also note that package specifications do not include any sub packages - each must be explicitly mentioned.

Any packages not mentioned in the module info cannot be used for a shim, because the JDK does not allow addExport of unreferenced packages. In these cases, you will have to get clever with your shims, placing the test shim in a visible package, perhaps calling a second layer of shim in a hidden package.

Each of the build modules in OpenJFX has an addExports file that is imported using @argfile into the test invocation. The addExports file contains entries that export packages that are not public so that the unit tests in the unnamed module can see them. Here is an excerpt from one of the files:

```
-XaddExports:javafx.base/com.sun.javafx.collections=ALL-UNNAMED

-XaddExports:javafx.base/com.sun.javafx.property=ALL-UNNAMED

-XaddExports:javafx.base/com.sun.javafx=ALL-UNNAMED

-XaddExports:javafx.base/com.sun.javafx.event=ALL-UNNAMED
```

And example error, that indicates a missing export:

> java.lang.IllegalAccessError: class test.javafx.beans.Person (in unnamed module @0x22a67b4) cannot access class com.sun.javafx.property.PropertyReference (in module javafx.base) because module javafx.base does not export com.sun.javafx.property to unnamed module @0x22a67b4

Building an Xpatch

As part of the modular test build, we need to create a directory containing the freshly built core classes with the shim classes added in. This combined mix of classes can be used with the Xpatch command to override the modules in the JDK.

In the OpenJFX build, this results in -Xpatch:build/testing/modules

NOTE: Xpatch does not override module-info.

Finding our shared libraries

By default, java will look for the module shared libraries in the JDK, which may mean problems if was are trying to use the libraries we just built. To access those libraries, we need to use -Djava.library.path=build/sdk/lib/*_the_right_arch_* to find them

Capturing our classpath

During the gradle test task, an @argfile form of the test classpath is created for each of the modules. This can be used for reference or with manual command lines. An example of this file is: *build/testing/classpath_graphics.txt*.

GradleJUnitWorker

Gradle 2.11 does not understand JDK 9 modules. Normally, gradle uses its own JUnit worker to process the test environment, particularly the classpath before running each of the tests. GradleJUnitWorker was created to workaround many of the issues that were encountered. This workaround is launched by gradle, and it in turn launches the gradle JUnit worker with the appropriate JDK9 modular arguments.

Toggling worker.debug

In build.gradle, there is a toggle that can be used to create additional output from GradleJUnitWorker showing the command lines used to launch tests. This currently affects GradleJUnitWorker.java and the system Sandbox tests. Modify this line (temporarily), and run tests with the gradle -info option. (The -info option to gradle enables it to pass through stdout/stderr from the worker process).

> systemProperties 'worker.debug': false

Security

If a security manager is used, extra permissions are required with Xpatch. All of the classes in the Xpatch are treated as outside the module for the purposes of security grants.

For our current JUnit tests, this only applies to the Sandbox tests in :systemtests. To permit the Xpatch classes to operate properly, we generate a java.policy file for the core FX classes in the Xpatch modules. This java.policy file (build/testing/java.patch.policy) can be used standalone if no additional permissions are needed, or must be combined with the desired additional permissions to form a single java.policy file. The Sandbox tests combine the permissions in the text file indicated by the property worker.patch.policy with the test permissions, and use the resulting file when invoking the test app.

An example modular manual command line

The following is an example command line that can run a junit test from within a built Linux OpenJFX modular tree. This can be useful when debugging a single test class:

```
jdk-9/bin/java \

 -Xpatch:build/testing/modules \

 @tests/system/src/test/addExports \

 @build/testing/classpath_systemTests.txt \

 -Djava.library.path=build/sdk/lib/amd64 \

  org.junit.runner.JUnitCore \
```

test.sandbox.SandboxAppTest