JavaOne™

ORACLE

Migrating Java UI Client Apps to the Modular JDK [CON4384]

Subtitle

Phil Race
Kevin Rushforth

JDK UI Client Group.

Note: The speaker notes for this slide include detailed instructions on how to reuse this Title Slide in another presentation.

Tip! Remember to remove this text box.

JavaOne ORACLE

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Note: The speaker notes for this slide include instructions for when to use Safe Harbor Statement slides.

Tip! Remember to remove this text box.

Session Topics

- Quick overview of the Jigsaw/Java Platform Module System.
- Impact on JDK source tree layout
- Impact on SE and JavaFX internals.
- Impact on Java SE and non-SE JDK client APIs
- Impact on SE and JavaFX client applications.
- Creating modules for Java SE Client Service Providers
- JavaFX Application Case Studies

Quick Project Jigsaw Overview

- Multiple JEPs and parts to Project Jigsaw
- JDK source code is restructured
- JDK runtime image is restructured (no more rt.jar)
- Defines the set of JDK modules (groups of related packages)
- Requires modules list modules on which they depend (JDK dependency graph)

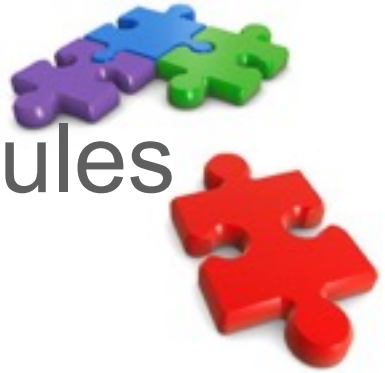
Project Jigsaw Overview (continued)

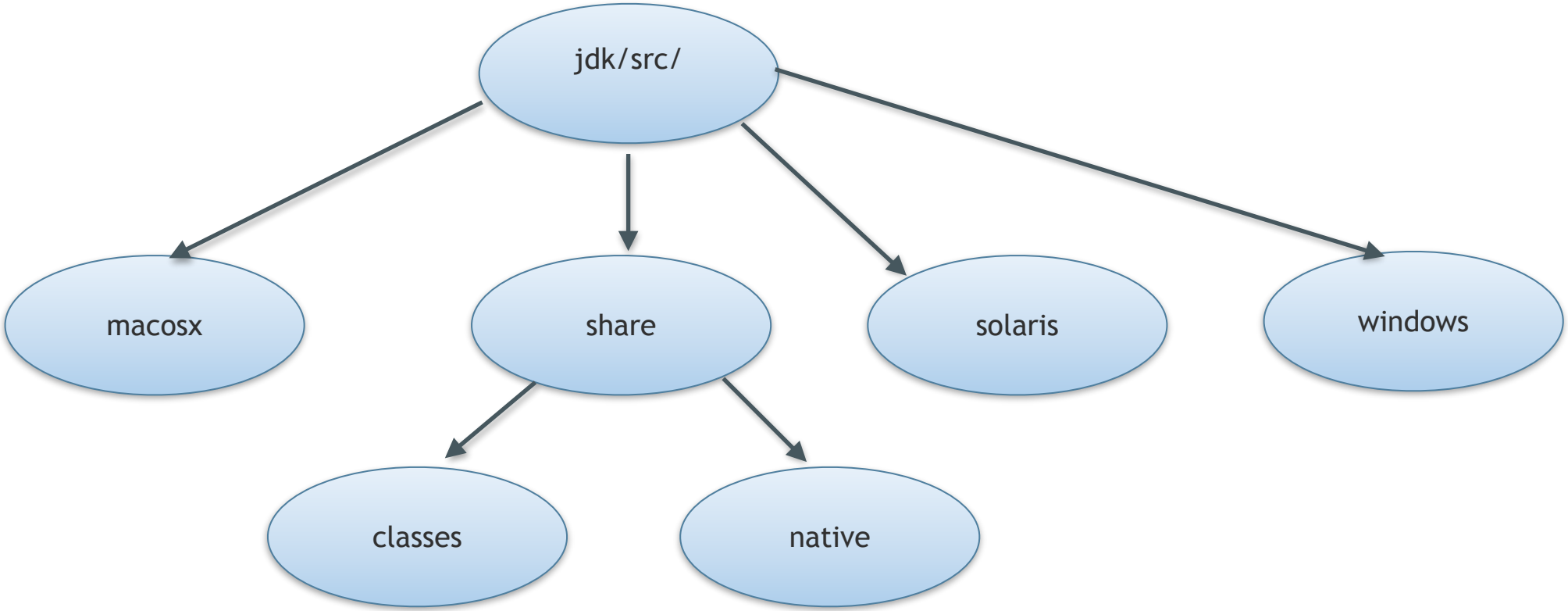
- Uses Service Providers for JDK to access code in external modules (no explicit dependency)
- Enforces encapsulation by denying access to packages not exported from a module
- Defines new language keywords and APIs for using the module system.
- Provides tools for building a modular JDK image

Some rules (consequences) of the Module System

- Internal JDK APIs are off-limits, even with no security manager
- Supportedness becomes a binary decision - no grey areas.
- SE modules cannot depend on non-SE modules - even internal ones.
- SE modules cannot export non-SE types.
- Exports are at the package level - all or nothing.
- No split packages

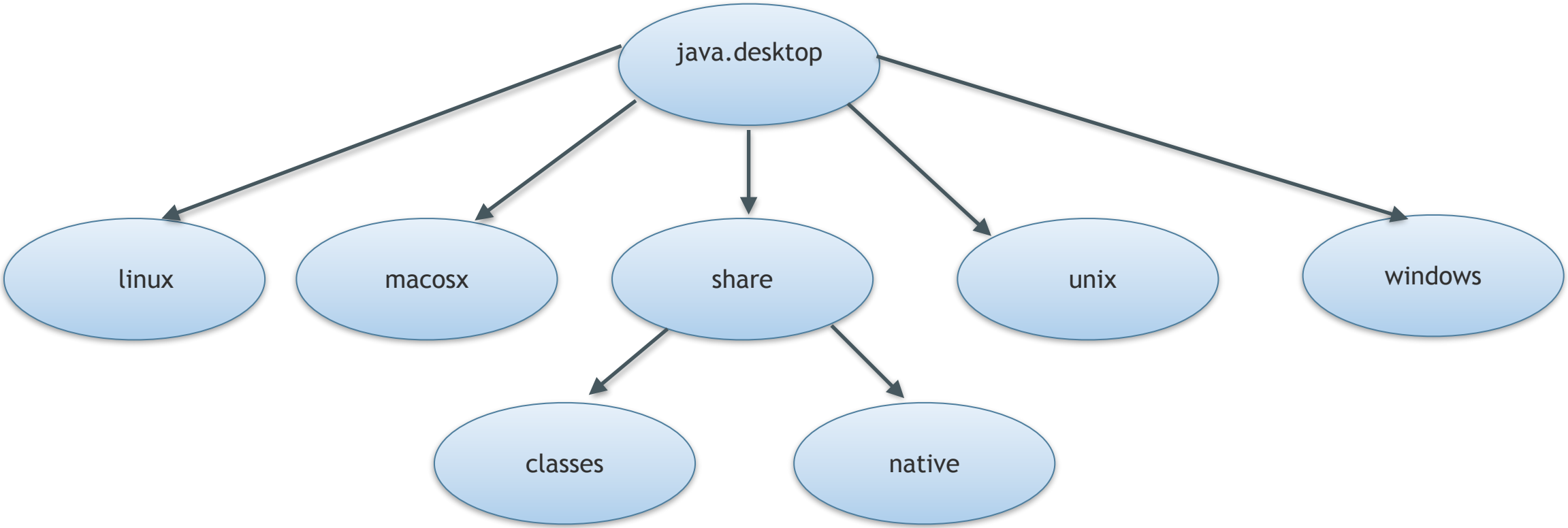
JDK Source Code restructuring and Client Modules





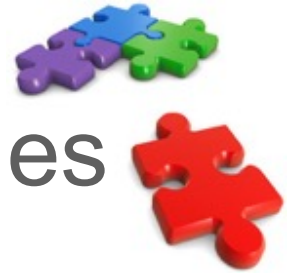
- Previously : JDK client code intermingled with all other source code.

New JDK source code organization



- Now JDK client code under its own hierarchy.

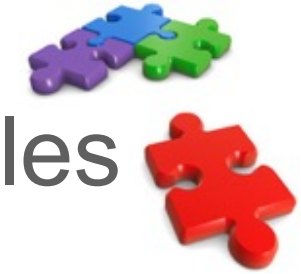
JDK Source Code restructuring and Client Modules



- New source code structure cleanly defines what is in the desktop module
- Mainly of interest to folks who use OpenJDK source
  - but eventually shows up in the runtime image.

- Maps to the "j*ava.desktop*" runtime module
  - contains all the binary code (Java classes and native, plus resources).

JDK Source Code restructuring and Client Modules



- One other very small client module : *java.datatransfer*
- Contains just the java.awt.datatransfer package
- Unbundled because non-client APIs used in embedded need it and the separation is possible (a rare case)
- java.desktop "*requires public java.datatransfer*" - so no need for apps to separately require it.
- This is the module system linkage granularity for client APIs in SE
  - Essentially all or nothing.

Impacts on JDK Client APIs and internals

- Breaks down in different ways
  - APIs retired
  - APIs adjusted / re-specified.
  - APIs made standard
  - Public replacement APIs provided for internal APIs
  - Internal changes made in the JDK in anticipation of module system constraints and rules.

Impacts on JDK Client APIs

Example 1: com.sun.image.codec.jpeg

- A non standard API from JDK 1.2 (1998)
  - Obsoleted by Image I/O in JDK 1.4 (2004)
  - Deprecated and hidden from new compilation (JDK 1.7)
  - No longer considered supported.
- Rules in effect
  - No exporting non-standard API from SE module
  - Binary choice of supportedness
- JDK action : deleted entirely
- User code action : migrate to Image I/O (javax.imageio)

Impacts on Client APIs

Example 2: java.awt.peer interfaces (this one was a toughie).

Package java.awt.peer is internal and undocumented.

But some of its types are exposed in public API

```
java.awt.Component : ComponentPeer getPeer()
java.awt.Font : FontPeer getPeer()
java.awt.Toolkit :
  ButtonPeer  createButton(Button target)
  LightweightPeer   createComponent(Component target)
java.awt.dnd.DragSource :
createDragSourceContext(java.awt.dnd.peer.DragSourceContextPeer, …)
```

Impact on Client APIs

Example 2: java.awt.peer interfaces (cont'd)

- Internal APIs in signatures of public classes very problematic
- Known to have been used historically by user code since JDK 1.0
- Rules in effect (nearly all of them)
  - No exporting non-standard API from SE module
  - Binary choice of supported-ness
  - Exports are at the package level - all or nothing.
  - No using internal JDK APIs

Impact on Client APIs

Example 2: java.awt.peer interfaces (cont'd)

- Option 1:
  - Make java.awt.peer standard
    - But these are properly internal to the desktop module
    - Needed only in ports of AWT itself (which implies a JDK port)

Impact on Client APIs

Example 2: java.awt.peer interfaces (cont'd)

- Option 2:
  - Change the documented and actual return type to Object.
    - A binary incompatibility
    - Cannot use it except as Object so not very useful.
    - JDK internal callers would need to use type-unsafe casting.

Impact on Client APIs

Example 2: java.awt.peer interfaces (cont'd)

- Option 3:
  - Stop documenting the methods: remove (only) from the spec.
  - Benefit: code that disables module boundary checking can still run
  - But too many oddities with TCK and type system.
    - Module system would still allow a call that does not try to resolve the return type to the internal type.
    - Analagous to an interface type being returned but implemented by an internal type.

Impact on Client APIs

Example 2: java.awt.peer interfaces (cont'd)

- Option 4:
  - Remove all the offending methods from the public API, making them internal to the desktop module.
    - Binary incompatible (obviously)
    - JDK code and application code will need to be updated.
  - The winner: Option 4: although it required a large set of JDK code changes and will require apps to be updated too
  - Doing nothing was not an option.

Impact on Client APIs

Replacing Application use of java.awt.peer classes.

- In practice most uses of peers we've found are one of two cases
- Usage 1: To see if a peer has been set yet

```
if (component.getPeer() != null)  { .. }
```

- Replace with JDK 1.1 API : Component.isDisplayable() :

```
public boolean isDisplayable() {
    return getPeer() != null;
}
```

Impact on Client APIs

Replacing Application use of java.awt.peer classes.

- Usage 2: To test if a component is lightweight

```
if (component.getPeer() instanceof LightweightPeer) ..
```

- Replace with JDK 1.2 API : `Component.isLightweight()` :

```
public boolean isLightweight() {
    return getPeer() instanceof LightweightPeer;
}
```

Impacts on JDK Client APIs

- **Example3: com.apple.eawt.desktop APIs**
  - Apple JDK supported these to provide access to OS X platform features
    - eg way to draw attention to app (bouncing dock icon)
    - handlers for standard system events

Impacts on JDK Client APIs

- **Example 3: com.apple.eawt.desktop APIs**
  - Rules in effect:
    - cannot export non-SE APIs
    - binary choice of supportedness
- Solution JEP 272: make standard APIs
  - extend java.awt.Desktop to provide similar functionality
  - query capability to test if platform provides a feature

Impacts on JDK Client APIs

- **Example 4: Java Access Bridge (an assistive technology)**
- Java Access Bridge (JAB) is not in the java.desktop module
  - It does not export any Java SE APIs
- java.awt.Toolkit responsible for loading JAB via reflection.
- Rules in effect:
    - SE module (ie java.desktop) cannot depend on non-SE module

Impacts on JDK Client APIs

- **Example 4: Java Access Bridge (an Assistive Technology (AT))**
- Solution: Support AT as a ServiceProvider
- AT will implement a new interface
  `javax.accessibility.AccessibilityProvider`
- JAB module will ***provide*** this service
- java.desktop will ***use*** the service.
  - java.util.ServiceLoader will take care of the loading/visibility

Impacts on JDK Client APIs

- **Example 5: Setting a Swing Look and Feel (L&F)**
- Swing provides two APIs to set the current L&F
  - `UIManager.setLookAndFeel(Class lafClass)`
  - `UIManager.setLookAndFeel(String lafName)`

Which one should you use ?

Impacts on JDK Client APIs

- **Example 5: Setting a Swing Look and Feel (L&F)**
- For built-in (JDK-provided) L&F ?
  - eg java.desktop module provides platform L&F
  - name of the class is exposed via

```
lafName = UIManager.getSystemLookAndFeelClassName()
Class lnfClass = Class.forName(lafName);
```

Impacts on JDK Client APIs

- **Example 5: Setting a Swing Look and Feel (L&F)**
- What about external / 3rd party L&F ?
- Either can work
  - If 3rd party L&F is an unnamed module (ordinary jar) you can instantiate
  - As a module it may export the class and package and you may ***require*** the module.

```
UIManager.setLookAndFeelClassName(lnfClass)
```

Using string name ?

`UIManager` will add a runtime read edge

```
UIManager.class.getModule().addReads(lnfClass.getModule())
```

Example of how JDK internals need to be updated to work with the module system

Impacts on JDK Client APIs

- **Example 6: Locating resources.**

  `javax.imageio.ImageReader.processWarning(String basename)`

  takes a resource bundle name but where should it look for the bundle ? Can client code provide additional localized resources for a standard (SE) Image I/O plugin ?

  Jigsaw expects resources in a named module to be located using the class loader which loaded the module.

Impacts on JDK Client APIs

- **Example 6: Locating resources.**
- Action:
  - Image I/O updated to restrict lookup
  - Now must come only from the same location as the plugin
  - For java.desktop provided services this is the ‘boot’ class loader.
  - For 3rd party external plugins this means they need to be provided by the plugin
- This all makes sense as you cannot localize some one else’s plugin
- Here jigsaw helped add clarity to what the code should have enforced all along.

Impact: Defining new public APIs

- Some internal APIs used by applications because there is still no public equivalent of useful functionality (but you need to tell us).
- Rule in effect
  - No using internal JDK APIs
- Solution : promote internal API to public
- Two small examples:

  sun.awt.ShellFolder (selected functionality)

  sun.awt.CausedFocusEvent
- Apps will need to change but can use reflection

Creating modules for Client Service Providers

- Several client APIs define Service Provider interfaces
  - javax.imageio (5)
  - javax.print (2)
  - javax.sound (8)
  - java.awt.im (Input Methods)
  - javax.accessibility (new in JDK 9 as discussed earlier)

Creating modules for Client Service Providers

- SPI support has been around since JDK 1.3
  - java.util.ServiceLoader can be used to load application-supplied implementation of interfaces
  - more useful than ever in JDK 9 - updated to work with module system
  - no need to know the dependent module, just the service name

Creating modules for Client Service Providers

- SE API defines the name of an interface
  - eg **`javax.imageio.spi.ImageReaderSpi`**
- Application provides a jar with a file :
  - META-INF/services/javax.imageio.ImageReaderSpi
  - Entry in that file defines the name of it's implementing class,eg **`simp.SIMPImageReader.Spi`**

Creating modules for Client Service Providers

- Creating a modular jar for a javax.imageio plugin:

  Step 1: Create module-info.java (new language syntax)

```
module simp /* name of the modular jar */ {
     requires java.desktop;
     exports simp to java.desktop; /* java package exported (see next slide) */
     provides javax.imageio.spi.ImageReaderSpi with simp.SIMPImageReaderSpi;
}
```

  Step 2: Create simp.jar as for JDK 8, but with module-info.java compiled in.

  Step 3: Add simp.jar to the ***module path :*** java -mp simp.jar …

Creating modules for Client Service Providers

- Creating a modular jar for a javax.imageio plugin:

  Why do we need “exports simp to java.desktop” ? Something of a special case in Image I/O

  - Image I/O plugin in the module provides the (String) name of its IIOMetadataFormat class
  - Core Image I/O IIOMetadata class provides a method that looks up this name and instantiates as necessary :

```
public IIOMetadataFormat getMetadataFormat(String)
```

- Unless the package is exported, it cannot be accessed. One module can add a read-edge to another module but *cannot* force the module to expose its internals.
- If designing today, it would be the plugin code that would be required to create the instance
- NB jigsaw is still evolving. This method ‘broke’ last week because it uses Class.forName(Module, String) on a potentially un-named module and the behaviour was changed in that case.

Creating modules for Client Service Providers

- Need your code to work with JDK 8 too ?
  - Not ready to require 9 and later ?
- Do nothing - Your JDK 8 “jar” is made into an “automatic module”
  - No module-info means it is a regular jar
  - Implicitly all packages are exported
  - ServiceLoader handles the dependency as for a regular module.
  - Put jar on the class path as now (not the module path)
- You do not get the encapsulation benefits, but no code changes are needed.

Impact: Re-specifying Java SE APIs

- A few cases where existing standard APIs do not play well with the module system
  - Cannot be fixed up ‘internally’. Case in point:
    - `javax.imageio.spi.ServiceRegistry.lookupProviders(Class)`
    - Is a wrapper around `ServiceLoader.load()`
      - pre-dates that as public API
      - lookups up an arbitrary Service Provider Interface
    - What is the catch ?

Impact: Re-specifying Java SE APIs

- To use a service a module must declare in its module-info.java

```
uses org.duke.SomeService
```

- ServiceLoader looks at its immediate caller which is java.desktop
- A long list of these in the module-info for java.desktop for standard SPIs but it does not include SomeService and consequently lookup will fail.
- JDK resolution: no need for this API since ServiceLoader is standard
  - As of JDK 1.9 it will load only the Image I/O specified service provider interfaces. Applications using it otherwise will break.

JDK Summary and Take Aways

- Examples provided are not by any means exhaustive
  - Image I/O examples could as easily have occurred elsewhere
- Other areas in client use core reflection
- Notably java.beans, javax.sound
- Tests that access internals needed to be updated

JDK Summary and Take Aways

- Uses of core reflection in the JDK code almost always need to add a runtime read edge - any time user code is delivered via a class name is problematic
- Service Providers are the preferred solution where possible
- Application code should test itself against the regular JDK 9 to see if affected by deleted or re-specified APIs
- Application code can test against the modular JDK 9 to see if affected by usage of internals (NB jdeps tool in JDK8u bin)
- Applications can benefit from encapsulation by adopting code delivery using modules.

JavaFX and Jigsaw

- JavaFX Modules
- Impact of Jigsaw on JavaFX Applications
- Impact of Jigsaw on JavaFX Internals
- JavaFX Sample Applications

JavaFX Modules

- JavaFX source code already organized as modules (as of JDK 8)

```
modules/
    base/
    controls/
    graphics/
    ...
```

JavaFX Modules

- In general, modules are named “javafx.nnnnn”
  - where “nnnnn” is the directory name under the modules directory

```
modules/
    base/          // javafx.base module
    controls/      // javafx.controls module
    graphics/      // javafx.graphics module
    ...
```

JavaFX Modules

- JavaFX classes and resources will be linked into the JDK
  - No more jfxrt.jar
- SWT interop will still be delivered separately:
  - jfxswt.jar file delivered with the JDK
  - Cannot be linked into runtime image because it depends on third-party swt.jar
  - jfxswt.jar will be an “automatic” module (meaning no module-info.class)
  - Implementation classes will go into a named javafx.internal.swt module

JavaFX Modules

JRE Modules

Public

```
javafx.base
javafx.controls
javafx.fxml
javafx.graphics
javafx.media
javafx.swing
javafx.web
```

Internal

```
javafx.deploy  [closed]
javafx.internal.swt
```

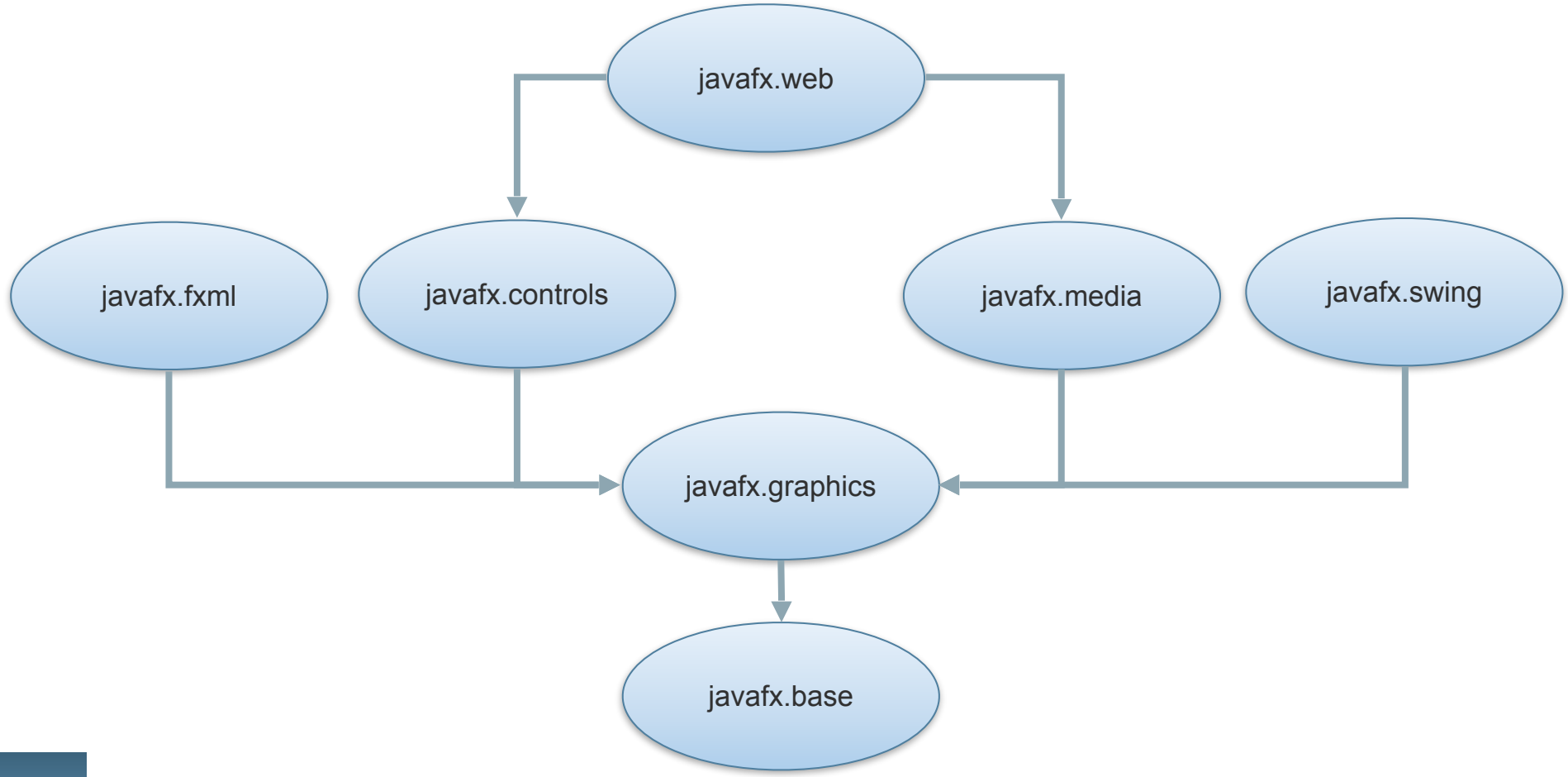
JDK Modules

Public

```
javafx.jmx
jdk.packager
jdk.packager.services
```

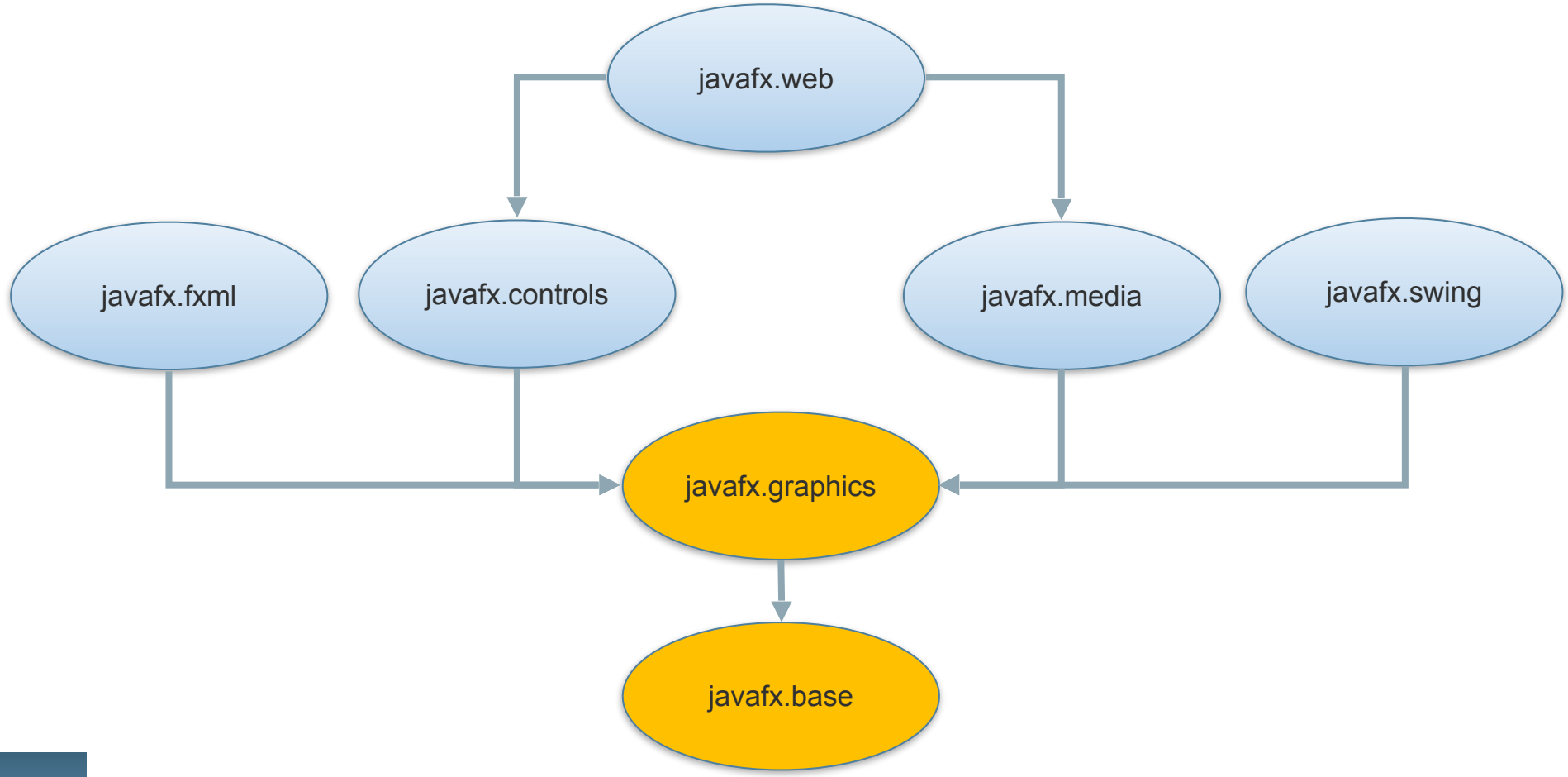
JavaFX Modules

- JavaFX module graph for runtime (JRE) modules: transitive reduction



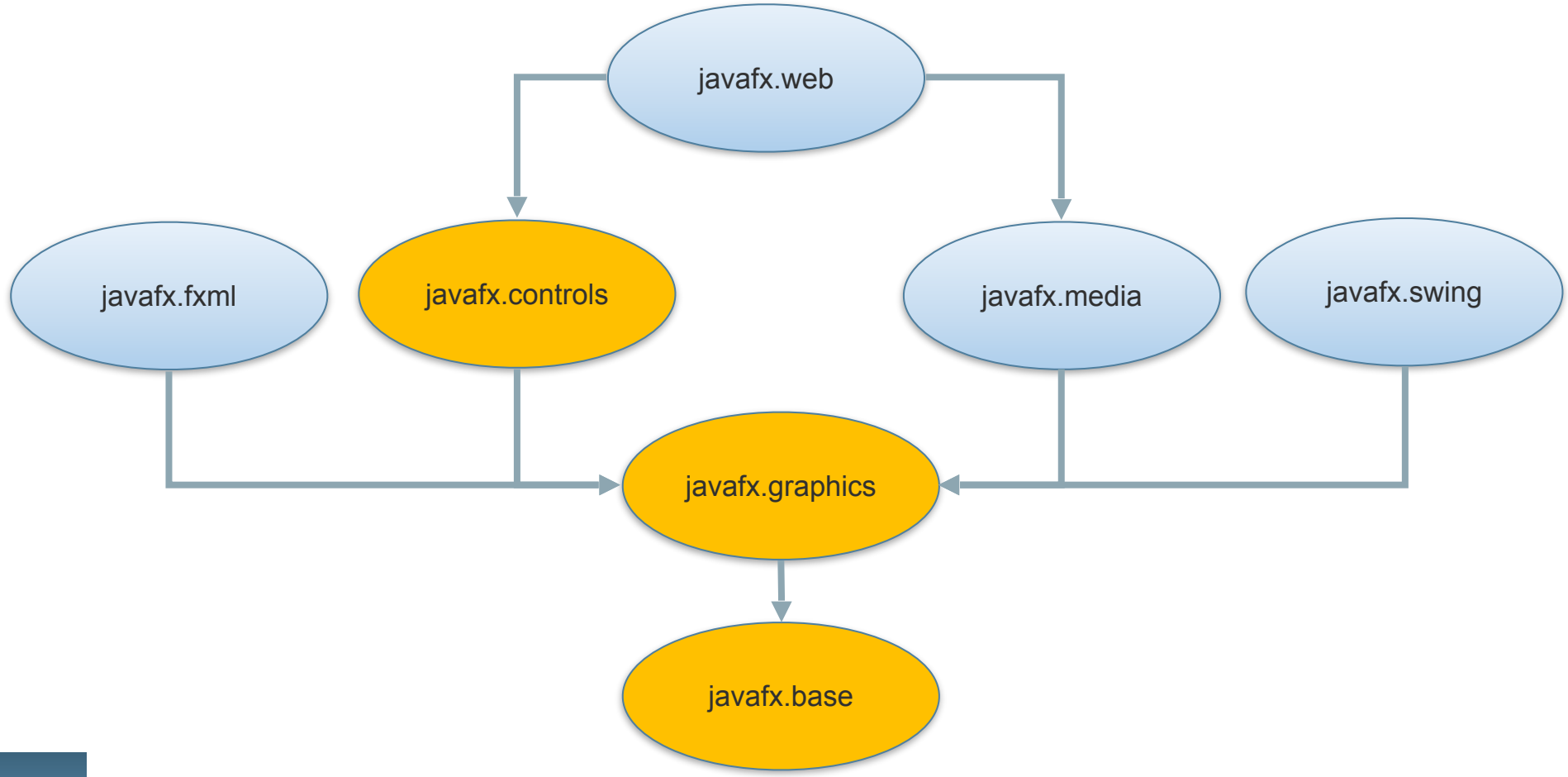
JavaFX Modules

- All JavaFX applications require javafx.graphics



JavaFX Modules

- All JavaFX UI applications require javafx.controls

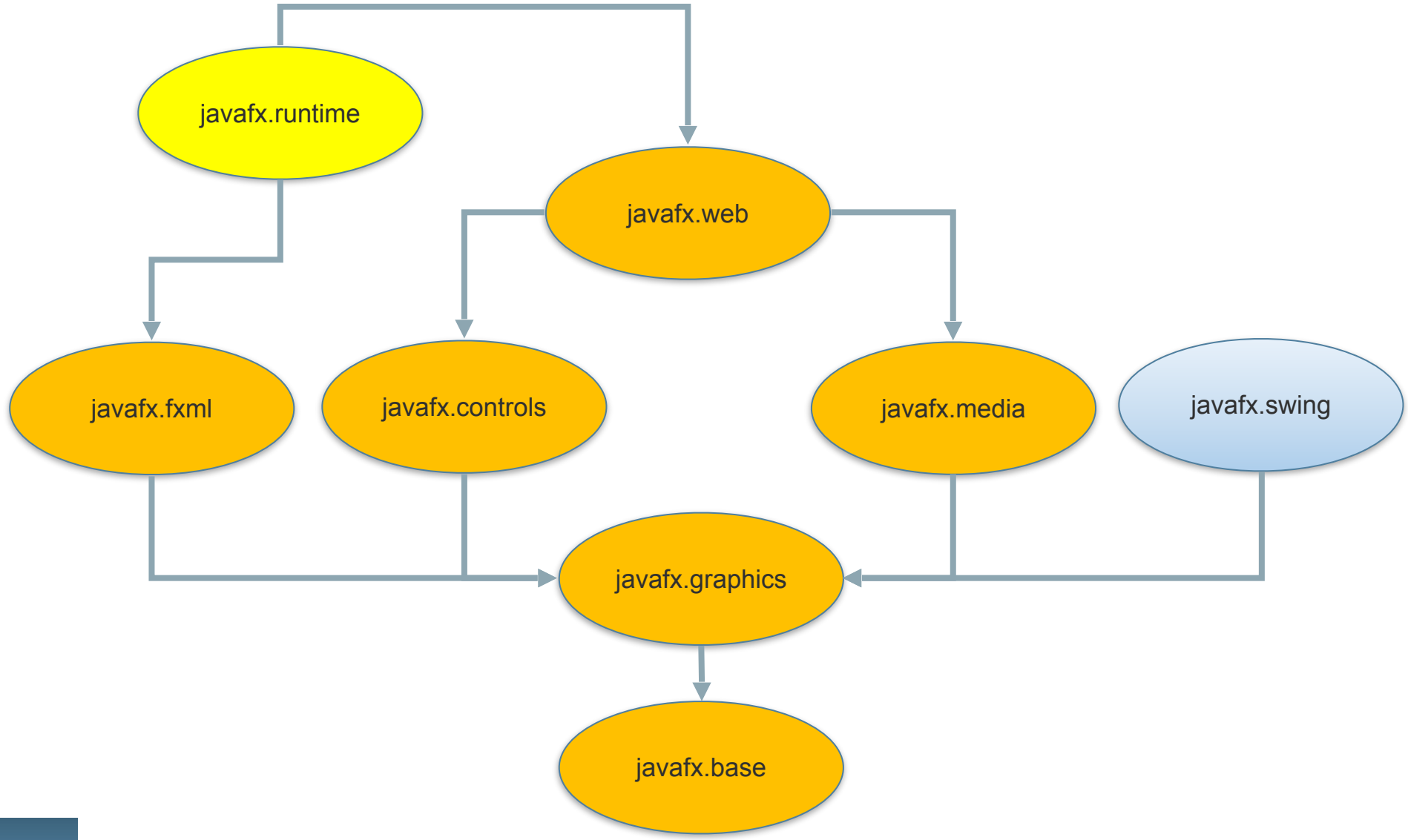


JavaFX Modules

- We will also provide an “aggregating” javafx.runtime module
  - Includes all the public javafx runtime modules except javafx.swing
  - Contains no classes or other artifacts
- A “pure” JavaFX application (no Swing interop) can just say:
  - **`requires javafx.runtime;`**

```
module javafx.runtime {
    requires public javafx.base;
    requires public javafx.graphics;
    requires public javafx.controls;
    requires public javafx.fxml;
    requires public javafx.media;
    requires public javafx.web;
}
```

JavaFX Modules



JavaFX Modules: Packages

- Only publicly documented packages are exported
  - All are in the "javafx.*" namespace

```
javafx.base module
    package javafx.beans;
    package javafx.beans.binding;
    package javafx.beans.property;
    package javafx.beans.property.adapter;
    package javafx.beans.value;
    package javafx.collections;
    package javafx.collections.transformation;
    package javafx.event;
    package javafx.util;
    package javafx.util.converter;
```

```
javafx.controls module
    package javafx.scene.chart;
    package javafx.scene.control;
    package javafx.scene.control.cell;
    package javafx.scene.control.skin;
```

```
javafx.fxml module
    package javafx.fxml;
```

JavaFX Modules: Packages

```
javafx.graphics module
    package javafx.animation;
    package javafx.application;
    package javafx.concurrent;
    package javafx.css;
    package javafx.css.converter;
    package javafx.geometry;
    package javafx.print;
    package javafx.scene;
    package javafx.scene.canvas;
    package javafx.scene.effect;
    package javafx.scene.image;
    package javafx.scene.input;
    package javafx.scene.layout;
    package javafx.scene.paint;
    package javafx.scene.shape;
    package javafx.scene.text;
    package javafx.scene.transform;
    package javafx.stage;
```

```
javafx.media module
    package javafx.scene.media;
```

```
javafx.swing module
    package javafx.embed.swing;
```

```
javafx.web module
    package javafx.scene.web;
```

JavaFX Modularity: Application Impact

- No access to internal “com.sun.*” JavaFX packages
  - Only explicitly exported packages are visible – javafx.*
  - Accessing a non-exported package will result in a compilation or runtime error
    - And no, you can’t simply use reflection to call **`setAccessible`**
    - It can be overridden with “**`-XaddExports`**” command line switch in extreme need

JavaFX Modularity: Application Impact

- Classes in non-modular applications are in the “unnamed” module
  - By default, unnamed module reads (requires) all named modules in system
  - Can access all publicly exported packages of all modules with no app changes
- Modular applications list their dependencies in module-info.java
  - Only public types of required modules are accessible
  - Here is a minimal module-info.java for an application that uses javafx.controls, .graphics, and .base modules (controls re-exports graphics and base)

```
module my.app {
  requires javafx.controls;
}
```

JavaFX Modularity: Impact on Internals

- Modules need to list dependencies and exports
  - Qualified exports can be used to allow other modules to access internals
  - Specified in module-info.java as follows:

```
module javafx.graphics {
    requires public javafx.base;                          ← uses / re-exports
    ...
    exports javafx.scene;                                 ← public API
    ...
    exports com.sun.glass.ui to javafx.media, javafx.web; ← qualified export
    ...
}
```

JavaFX Modularity: Impact on Internals

- No conditional dependencies
  - Use Module.addReads to access a new module at runtime
    - Used by frameworks or loaders to dynamically load classes
    - Example: FXML, Beans, Application launcher
  - Use Module.addExports to export a package to another module at runtime
    - JavaFX modules don't currently do this
    - We use qualified exports in module-info.java for all such cases
      - We will probably need this to export packages from javafx.internal.swt to jfxswt.jar
    - Usage:

```
Module thisModule = MyClass.getModule();
thisModule.addExports("my.pkg", otherClass.getModule());
```

JavaFX Internals: addReads

- JavaFX Beans

ReadOnlyPropertyDescriptor.java

```
 public ReadOnlyPropertyDescriptor(... Class<?> beanClass, ...) {
     ReflectUtil.checkPackageAccess(beanClass);
+    Module thisModule = this.getClass().getModule();
+    thisModule.addReads(beanClass.getModule();
```

- JavaFX Launcher

LauncherImpl.java

```
Module thisModule = LauncherImpl.class.getModule();
Module targetModule = appClass.getModule();
thisModule.addReads(targetModule);
```

JavaFX Internals: MethodUtil

- MethodUtil.invoke will not work for resources in module
  - Used by FXML both for internal and application methods

```
BeanAdapter.java (and a few other places)

-     value = MethodUtil.invoke(getterMethod, bean, (Object[]) null);
+     value = ModuleHelper.invoke(getterMethod, bean, (Object[]) null);

ModuleHelper.invoke(...) {
    Module thisModule = ModuleHelper.class.getModule();
    Module methodModule = m.getDeclaringClass().getModule();
    if (methodModule != thisModule) {
        return MethodUtil.invoke(m, obj, params);
    } else {
        return m.invoke(obj, params);
    }
}
```

JavaFX Modularity: Impact on Internals

- Reflection issues in FX media

Locator.java

```
 private static long getContentLengthLong(URLConnection connection) {
 ...
          try {
-             return connection.getClass().getMethod("getContentLengthLong");
+             return URLConnection.class.getMethod("getContentLengthLong");
          } catch (NoSuchMethodException ex) {
              return null;
          }
```

- If “connection” class is in another module, we get IllegalAccessError
  - The method we need is in URLConnection itself, so use that directly

JavaFX Modularity: Impact on Internals

These affect JavaFX applications, too

- Resource encapsulation changes:
  - ClassLoader.getResource() will not find resources in module
    - A modular app cannot simply use "/some/path/myresource.css" as stylesheet URL
  - Class.getResource() will find resources in modules -- but **only** in your own module
    - No more “dipping into” the internals to load another module's resource
  - This affected both JavaFX internals and (at least) two sample apps

JavaFX Modularity: Impact on Internals

These affect JavaFX applications, too

- JavaFX has several APIs that take a URL (or url String)
  - CSS Stylesheets
  - FXMLLoader
  - Image & Media
  - WebEngine
- A non-modular app can continue to use ClassLoader-relative URLs
- JavaFX internals, and modular applications cannot
  - Use Class.getResource() instead

JavaFX Internals: CSS Processing

- CSS processing: loading internal CSS files (e.g., modena.css)
- Two bad assumptions:
  - Internal JavaFX resources can be loaded using absolute path
    - Relying on classpath or lookup of jfxrt.jar
  - javafx.graphics module can load resources from javafx.controls module
  - Neither will work
- Solution:
  - Create a getResource() utility method in an internal controls package
    - Package is exported to javafx.graphics via qualified exports
    - The controls module gets its own resources and passes them to graphics

JavaFX Internals: JSObject

- JDK 8 has duplicate copy of `netscape.javascript.JSObject`
  - One in plugin.jar and one in jfxrt.jar
  - Used as the return type of HostServices.getWebContext()
- We must unify them in JDK 9
- JSObject has legacy method that takes an instance of AWT Applet
  - A bad thing to have done, but we are stuck with it
  - Changing it would break too many legacy applets
  - We do not want a hard API dependency from javafx.graphics on java.desktop

JavaFX Internals: JSObject

- Solution:
  - We are making an incompatible API change to getWebContext

```
HostServices.java:
  -     public final JSObject getWebContext();
  +     public final Object getWebContext();
```

- Applications that use getWebContext() will need to cast:

```
JSObject jsWin = (JSobject)getHostServices().getWebContext();
```

JavaFX Modularity: Unit Tests

- Work in progress
- Challenges:
  - Unit tests often need to access internal (implementation) packages
  - Some "white-box" tests need to access package-private data
  - JavaFX unit tests were not organized well
    - Many were in the same package as the core module, without need
  - Named modules cannot directly access types from unnamed modules
    - Using addReads is a runtime solution, but has to be done at runtime
  - Gradle does not work with the modular JDK
    - Their unit test harness makes assumptions about JDK class loader internals

JavaFX Modularity: Unit Tests

- Progress thus far:
  - Refactored tests to put actual unit tests in separate packages
  - Use "shim" classes to access package private data
    - These live in an existing package of the module they are testing
    - Appended to the module when tests are run using -Xpatch
  - Use @argfiles to specify the list of -XaddExports and -Xpatch
  - Tests for javafx.base, graphics, and controls are now running (standalone)
- Next steps:
  - Integrate this into gradle
  - Need custom test runner to be able to run this on JDK 9 + Jigsaw

JavaFX Modularity: Impact on Internals

- JavaFX Builders violated split package rule
  - In the same package as the classes they build, but in a different module
  - Already deprecated in 8 for unrelated reasons (unable to maintain compatibility when core classes evolve)
  - We removed them in JDK 9
- AWT peer change affected FX internals
  - Swing interop depends on java.awt.dnd.peer for Drag and Drop
  - One of the public interfaces FX uses has changed in JDK 9
  - FX will need to adapt to this before we switch to using JDK 9 as boot JDK

JavaFX Sample Applications: SceneBuilder

- SceneBuilder in FX 8 uses several internal packages
  - Most were due to Skins and CSS APIs not being public
    - JEP 253 fixes that
    - SceneBuilder helped inform public CSS API
      - (e.g., ParsedValueImpl, converter classes, error handling)
  - A few miscellaneous places had no equivalent public API
    - Internal picking code -- wrote a 10 line replacement function using public API
    - Window system toolkit listeners -- mainly used for native Apple menu handling
      - no replacement, but not critical functionality
- SceneBuilder in FX 9 now uses only public API
  - Runs on Jigsaw EA build without error

JavaFX Sample Applications: Ensemble8

- Eliminated use of deprecated builders
- No other changes needed to run as non-modular app
  - Runs in the unnamed module

- One change needed to allow it to run as a module:

```
// OLD
scene.getStylesheets().setAll("/ensemble/EnsembleStylesCommon.css");

//NEW
URL url = EnsembleApp.class.getResource("EnsembleStylesCommon.css")
scene.getStylesheets().setAll(url.toExternalForm());
```

JavaFX Sample Applications: Modena

- Eliminated deprecated builders
- Eliminated access to internal stylesheets:
  - Prior to 8u60, used internal skin class
    - com.sun.javafx.scene.control.skin.ButtonSkin.class.getResource("…")
    - This fails on attempt to access internal class (which has since moved anyway)
  - Currently in 9-dev, locates the "root" of jfxrt.jar from public class
    - Control.class.getResource("Control.class").toExternalForm()
    - Strips off "javafx/scene/control/Control.class"
    - Appends the internal path to modena.css
    - This fails because of resource encapsulation

JavaFX Sample Applications: Modena

- As mentioned earlier, application can no longer get at internals
  - This is a good thing: it stops apps depending on implementation details

- Problem: Reads internal css to test different color themes
- Solution: Use public API to load the unmodified css files
  - No longer need to read the contents of the css files

JavaFX Sample Applications: 3DViewer

- Eliminated deprecated builders

- Removed use of internal APIs
  - Used com.sun.geometry.* for some geometric computation
  - Necessary geometry utility classes now imported into application
  - Removed use of performance tracker

JavaFX Modularity: Impact on Applications

- All applications need to check and avoid:
  - Use of any JavaFX com.sun.* package
    - Many already have a public replacement
  - Access to any JavaFX resources (e.g., modena.css)
  - Use of deprecated builders (already gone in JDK 9)
- If you want to modularize your application, you also need to avoid:
  - Using ClassLoader.getResource (use Class.getResource instead)
  - Passing a “classpath-relative” URL string for resources in your module into CSS, FXML, WebView, etc.
    - Don’t do this: scene.getStyleSheets().add("/path/to/my/resource/resource.css");

Modular JDK with JavaFX: Availability

- Download early access builds from java.net
  - https://jdk9.java.net/jigsaw/
  - Includes JavaFX modules

- Source code for JDK Jigsaw prototype is here:
  - http://hg.openjdk.java.net/jigsaw/jake
- Source code for JavaFX Jigsaw protype is here:
  - http://hg.openjdk.java.net/openjfx/sandbox-9-jake/rt

Session Surveys

Help us help you!!

- Oracle would like to invite you to take a moment to give us your session feedback. Your feedback will help us to improve your conference.
- Please be sure to add your feedback for your attended sessions by using the Mobile Survey or in Schedule Builder.

JavaOne™

ORACLE