Introducing FXML

A Markup Language for JavaFX

Greg Brown, 8/15/2011

FXML is a scriptable, XML-based markup language for constructing Java object graphs. It provides a convenient alternative to constructing such graphs in procedural code, and is ideally suited to defining the user interface of a JavaFX application, since the hierarchical structure of an XML document closely parallels the structure of the JavaFX scene graph.

This document introduces the FXML markup language and explains how it can be used to simplify development of JavaFX applications.

Elements

In FXML, an XML element represents one of the following:

- A class instance
- A property of a class instance
- A "static" property
- A "define" block
- A block of script code

Class instances, instance properties, static properties, and define blocks are discussed in this section below. Scripting is discussed in a later section.

Value Elements

Class instances can be constructed in FXML in several ways. The most common is via instance declaration elements, which simply create a new instance of a class by name. Other ways of creating class instances include referencing existing values, copying existing values, and including external FXML files. Each is discussed in more detail below.

Instance Declarations

If an element's tag name begins with an uppercase letter (and it is not a "static" property setter, described later), it is considered an instance declaration. When the FXML loader (also introduced later) encounters such an element, it creates an instance of that class.

As in Java, class names can be fully-qualified (including the package name), or they can be imported using the "import" processing instruction (PI). For example, the following PI imports the `javafx.scene.control.Label` class into the current FXML document's namespace:

```
<?import javafx.scene.control.Label?>
```

This PI imports all classes from the `javafx.scene.control` package into the current namespace:

```
<?import javafx.scene.control.*?>
```

Any class that adheres to JavaBean constructor and property naming conventions can be readily instantiated and configured using FXML. The following is a simple but complete example that creates an instance of `javafx.scene.control.Label` and sets its "text" property to "Hello, World!":

```
<?import javafx.scene.control.Label?>
<Label text="Hello, World!"/>
```

Note that the `Label`'s "text" property in this example is set using an XML attribute. Properties can also be set using nested property elements. Property elements are discussed in more detail later in this section. Property attributes are discussed in a later section.

Classes that don't conform to Bean conventions can also be constructed in FXML, using an object called a "builder". Builders are discussed in more detail later.

Maps
Internally, the FXML loader uses an instance of `com.sun.javafx.fxml.BeanAdapter` to wrap an instantiated object and invoke its setter methods. This (currently) private class implements the `java.util.Map` interface and allows a caller to get and set Bean property values as key/value pairs.

If an element represents a type that already implements `Map` (such as `java.util.HashMap`), it is not wrapped and its `get()` and `put()` methods are invoked directly. For example, the following FXML creates an instance of `HashMap` and sets its "foo" and "bar" values to "123" and "456", respectively:

```
<HashMap foo="123" bar="456"/>
```

fx:value
The `fx:value` attribute can be used to initialize an instance of a type that does not have a default constructor but provides a static `valueOf(String)` method. For example, `java.lang.String` as well as each of the primitive wrapper types define a `valueOf()` method and can be constructed in FXML as follows:

```
<String fx:value="Hello, World!"/>
<Double fx:value="1.0"/>
<Boolean fx:value="false"/>
```

Custom classes that define a static `valueOf(String)` method can also be constructed this way.

fx:factory
The `fx:factory` attribute is another means of creating objects whose classes do not have a default constructor. The value of the attribute is the name of a static, no-arg factory method for producing class instances. For example, the following markup creates an instance of an observable array list, populated with three string values:

```
<FXCollections fx:factory="observableArrayList">
    <String fx:value="A"/>
    <String fx:value="B"/>
    <String fx:value="C"/>
</FXCollections>
```

Builders
A third means of creating instances of classes that do not conform to Bean conventions (such as those representing immutable values) is a "builder". The builder design pattern delegates object construction to a mutable helper class (called a "builder") that is responsible for manufacturing instances of the immutable type.

Builder support in FXML is provided by two interfaces. The `Builder` interface defines a single method named `build()` which is responsible for constructing the actual object:

```
public interface Builder<T> {
    public T build();
}
```

A `BuilderFactory` is responsible for producing builders that are capable of instantiating a given type:

```
public interface BuilderFactory {
    public Builder<?> getBuilder(Class<?> type);
}
```

A default builder factory, `JavaFXBuilderFactory`, is provided in the `javafx.fxml` package. This factory is capable of creating and configuring most immutable JavaFX types. For example, the following markup uses the default builder to create an instance of the immutable `javafx.scene.paint.Color` class:

```
<Color red="1.0" green="0.0" blue="0.0"/>
```

Note that, unlike Bean types, which are constructed when the element's start tag is processed, objects constructed by a builder are not instantiated until the element's closing tag is reached. This is because all of the required arguments may not be available until the element has been fully processed. For example, the `Color` object in the preceding example could also be written as:

```
<Color>
    <red>1.0</red>
    <green>0.0</green>
    <blue>0.0</blue>
</Color>
```

The `Color` instance cannot be fully constructed until all three of the color components are known.

When processing markup for an object that will be constructed by a builder, the `Builder` instances are treated like value objects - if a `Builder` implements the `Map` interface, the `put()` method is used to set the builder's attribute values. Otherwise, the builder is wrapped in a `BeanAdapter` and its properties are assumed to be exposed via standard Bean setters.

<u><fx:include></u>
The `<fx:include>` tag creates an object from FXML markup defined in another file. It is used as follows:

```
<fx:include source="filename"/>
```

where *filename* is the name of the FXML file to include. Values that begin with a leading slash character are treated as relative to the classpath. Values with no leading slash are considered relative to the path of the current document.

For example, given the following markup:

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<VBox xmlns:fx="http://javafx.com/fxml">
    <children>
        <fx:include source="my_button.fxml"/>
    </children>
</VBox>
```

my_button.fxml

```
<?import javafx.scene.control.*?>
<Button text="My Button"/>
```

the resulting scene graph would contain a `VBox` as a root object with a single `Button` as a child node.

Note the use of the "fx" namespace prefix. This is a reserved prefix that defines a number of elements and attributes that are used for internal processing of an FXML source file. It is generally declared on the root element of a FXML document. Other features provided by the "fx" namespace are described in the following sections.

`<fx:include>` also supports attributes for specifying the name of the resource bundle that should be used to localize the included content, as well as the character set used to encode the source file. Resource resolution is discussed in a later section.

<u><fx:reference></u>
The `<fx:reference>` element creates a new reference to an existing element. Wherever this tag appears, it will effectively be replaced by the value of the named element. It is used in conjunction with either the `fx:id` attribute or with a script variables, both of which are discussed in more detail in later sections. The "source" attribute of the `<fx:reference>` element specifies the name of the object to which the new element will refer.

For example, the following markup assigns a previously-defined `Image` instance named "myImage" to the "image" property of an ImageView control:

```
<ImageView>
```

```
    <image>
        <fx:reference source="myImage"/>
    </image>
</ImageView>
```

Note that, since it is also possible to dereference a variable using the attribute variable resolution operator (discussed later in the *Attributes* section), `fx:reference` is generally only used when a reference value must be specified as an element, such as when adding the reference to a collection:

```
<ArrayList>
    <fx:reference source="element1"/>
    <fx:reference source="element2"/>
    <fx:reference source="element3"/>
</ArrayList>
```

For most other cases, using an attribute is simpler and more concise.

<u><fx:copy></u>
The `<fx:copy>` element creates a copy of an existing element. Like `<fx:reference>`, it is used with the `fx:id` attribute or a script variable. The element's "source" attribute specifies the name of the object that will be copied. The source type must define a copy constructor that will be used to construct the copy from the source value.

At the moment, no JavaFX platform classes provide such a copy constructor, so this element is provided primarily for use by application developers. This may change in a future release.

Property Elements
Elements whose tag names begin with a lowercase letter represent object properties. A property element may represent one of the following:
- A property setter
- A read-only list property
- A read-only map property

<u>Property Setters</u>
If an element represents a property setter, the contents of the element (which must be either a text node or a nested class instance element) are passed as the value to the setter for the property.

For example, the following FXML creates an instance of the `Label` class and sets the value of the label's "text" property to "Hello, World!":

```
<?import javafx.scene.control.Label?>
<Label>
    <text>Hello, World!</text>
</Label>
```

This produces the same result as the earlier example which used an attribute to set the "text" property:

```
<?import javafx.scene.control.Label?>
<Label text="Hello, World!"/>
```

Property elements are generally used when the property value is a complex type that can't be represented using a simple string-based attribute value, or when the character length of the value is so long that specifying it as an attribute would have a negative impact on readability.

Type Coercion
FXML uses "type coercion" to convert property values to the appropriate type as needed. Type coercion is required because the only data types supported by XML are elements, text, and attributes (whose values are also text). However, Java supports a number of different data types including built-in primitive value types as well as extensible reference types.

The FXML loader uses the `coerce()` method of `BeanAdapter` to perform any required type conversions. This method is capable of performing basic primitive type conversions such as `String` to `boolean` or `int` to `double`, and will also convert `String` to `Class` or `String` to `Enum`. It will also convert a list of comma-separated values to an array type. Additionally, custom conversions can be implemented by defining a static `valueOf()` method on the target type.

Read-Only List Properties
A read-only list property is a Bean property whose getter returns an instance of `java.util.List` and has no corresponding setter method. The contents of a read-only list element are automatically added to the list as they are processed.

For example, the "children" property of `javafx.scene.Group` shown in an earlier example is a read-only list representing the group's child nodes:

```
<?import javafx.scene.*?>
<?import javafx.scene.shape.*?>
<Group xmlns:fx="http://javafx.com/fxml">
    <children>
        <Rectangle fx:id="rectangle" x="10" y="10" width="320" height="240"
            fill="#ff0000"/>
        ...
    </children>
</Group>
```

As each sub-element of the `<children>` element is read, it is added to the list returned by `Group#getChildren()`.

Read-Only Map Properties
A read-only map property is a bean property whose getter returns an instance of `java.util.Map` and has no corresponding setter method. The attributes of a read-only map element are applied to the map when the closing tag is processed.

The "properties" property of `javafx.scene.Node` is an example of a read-only map. The following markup sets the "foo" and "bar" properties of a `Label` instance to "123" and "456", respectively:

```
<?import javafx.scene.control.*?>
<Button>
    <properties foo="123" bar="456"/>
</Button>
```

Note that a read-only property whose type is neither a `List` nor a `Map` will be treated as if it were a read-only map. The return value of the getter method will be wrapped in a `BeanAdapter` and can be used in the same way as any other read-only map.

<u>Default Properties</u>
A class may define a "default property" using the `@DefaultProperty` annotation defined in the `javafx.fxml` package. If present, the sub-element representing the default property can be omitted from the markup.

For example, if the `Group` class were to define a default property of "children", the `<children>` element would no longer be required:

```
<?import javafx.scene.*?>
<?import javafx.scene.shape.*?>
<Group xmlns:fx="http://javafx.com/fxml">
    <Rectangle fx:id="rectangle" x="10" y="10" width="320" height="240"
        fill="#ff0000"/>
    ...
</Group>
```

Taking advantage of default properties can significantly reduce the verbosity of FXML markup.

Static Properties
An element may also represent a "static" property (sometimes called an "attached property"). Static properties are properties that only make sense in a particular context. They are not intrinsic to the class to which they are applied, but are defined by another class (often, the parent container of a control).

Static properties are prefixed with the name of class that defines them. For example, The following FXML invokes the static setter for `GridPane`'s "rowIndex" and "columnIndex" properties:

```
<GridPane>
    <children>
        <Label text="My Label">
            <GridPane.rowIndex>0</GridPane.rowIndex>
            <GridPane.columnIndex>0</GridPane.columnIndex>
        </Label>
    </children>
</TabPane>
```

This translates roughly to the following in Java:

```
GridPane gridPane = new GridPane();

Label label = new Label();
label.setText("My Label");

GridPane.setRowIndex(label, 0);
GridPane.setColumnIndex(label, 0);

gridPane.getChildren().add(label);
```

The calls to `GridPane#setRowIndex()` and `GridPane#setColumnIndex()` "attach" the index data to the `Label` instance. GridPane then uses these during layout to arrange its children appropriately. Other containers, including `AnchorPane`, `BorderPane`, and `StackPane`, define similar properties.

As with instance properties, static property elements are generally used when the property value cannot be efficiently represented by an attribute value. Otherwise, static property attributes (discussed in a later section) will generally produce more concise and readable markup.

Define Blocks

The `<fx:define>` element is used to create objects that exist outside of the object hierarchy but may need to be referred to elsewhere.

For example, when working with radio buttons, it is common to define a `ToggleGroup` that will manage the buttons' selection state. This group is not part of the scene graph itself, so should not be added to the buttons' parent. A define block can be used to create the button group without interfering with the overall structure of the document:

```
<VBox>
    <fx:define>
        <ToggleGroup fx:id="myToggleGroup"/>
    </fx:define>
    <children>
        <RadioButton text="A" toggleGroup="$myToggleGroup"/>
        <RadioButton text="B" toggleGroup="$myToggleGroup"/>
        <RadioButton text="C" toggleGroup="$myToggleGroup"/>
    </children>
</VBox>
```

Elements in define blocks are usually assigned an ID that can be used to refer to the element's value later. IDs are discussed in more detail in later sections.

Attributes

An attribute in FXML may represent one of the following:
- A property of a class instance
- A "static" property
- An event handler

Each are discussed in more detail in the following sections.

Instance Properties

Like property elements, attributes can also be used to configure the properties of a class instance. For example, the following markup creates a `Button` whose text reads "Click Me!":

```
<?import javafx.scene.control.*?>
<Button text="Click Me!"/>
```

As with property elements, property attributes support type coercion. When the following markup is processed, the "x", "y", "width", and "height" values will be converted to `double`s, and the "fill" value will be converted to a `Color`:

```
<Rectangle fx:id="rectangle" x="10" y="10" width="320" height="240"
    fill="#ff0000"/>
```

Unlike property elements, which are applied as they are processed, property attributes are not applied until the closing tag of their respective element is reached. This is done primarily to facilitate the case where an attribute value depends on some information that won't be available until after the element's content has been completely processed (for example, the selected index of a `TabPane` control, which can't be set until all of the tabs have been added).

Another key difference between property attributes and property elements in FXML is that attributes support a number of "operators" that extend their functionality. The following operators are supported and are discussed in more detail below:

- Location resolution
- Resource resolution
- Variable resolution

<u>Location Resolution</u>

As strings, XML attributes cannot natively represent typed location information such as a URL. However, it is often necessary to specify such locations in markup; for example, the source of an image resource. The location resolution operator (represented by an "@" prefix to the attribute value) is used to specify that an attribute value should be treated as a location relative to the current file rather than a simple string.

For example, the following markup creates an `ImageView` and populates it with image data from "my_image.png", which is assumed to be located at a path relative to the current FXML file:

```
<ImageView>
```

```
    <image>
        <Image url="@my_image.png"/>
    </image>
</ImageView>
```

Since `Image` is an immutable object, a builder is required to construct it. Alternatively, if `Image` were to define a `valueOf(URL)` factory method, the image view could be populated as follows:

```
<ImageView image="@my_image.png"/>
```

since the value of the "image" attribute would be converted to a URL by the FXML loader, then coerced to an `Image` using the `valueOf()` method.

Resource Resolution
In FXML, resource substitution can be performed at load time for localization purposes. When provided with an instance of `java.util.ResourceBundle`, the FXML loader will replace instances of resource names with their locale-specific values. Resource names are identified by a "%" prefix, as shown below:

```
<Label text="%myText"/>
```

If the loader is given a resource bundle defined as follows:

```
myText = This is the text!
```

the output of the FXML loader would be a `Label` instance containing the text "This is the text!".

Variable Resolution
An FXML document defines a variable namespace in which named elements and script variables may be uniquely identified. The variable resolution operator allows a caller to replace an attribute value with an instance of a named object before the corresponding setter method is invoked. Variable references are identified by a "$" prefix, as shown below:

```
<fx:define>
    <ToggleGroup fx:id="myToggleGroup"/>
</fx:define>
...
<RadioButton text="A" toggleGroup="$myToggleGroup"/>
<RadioButton text="B" toggleGroup="$myToggleGroup"/>
<RadioButton text="C" toggleGroup="$myToggleGroup"/>
```

Assigning an `fx:id` value to an element creates a variable in the document's namespace that can later be referred to by variable dereference attributes, such as the "toggleGroup" attribute shown above, or in script code, discussed in a later section. Additionally, if the object's type defines an "id" property, this value will also be passed to the objects `setId()` method.

Expression Binding
Attribute variables as shown above are resolved once at load time. Later updates to the variables value are not automatically reflected in any properties to which the value was assigned. In many cases, this is sufficient; however, it is often convenient to "bind" a property value to a variable or expression such that changes to the variable are automatically propagated to the target property. Expression bindings can be used for this purpose.

An expression binding also begins with the variable resolution operator, but is followed by a set of curly braces which wrap the expression value. For example, the following markup binds the value of a text input's "text" property to the "text" property of a Label instance:

```
<TextInput fx:id="textInput"/>
<Label text="${textInput.text}"/>
```

As the user types in the text input, the label's text content will be automatically updated.

Only simple expressions that resolve to property values or page variables are currently supported. More complex expressions involving boolean or other operators may be supported in the future.

Static Properties
Attributes representing static properties are handled similarly to static property elements and use a similar syntax. For example, the earlier `GridPane` markup shown earlier to demonstrate static property elements could be rewritten as follows:

```
<GridPane>
    <children>
        <Label text="My Label" GridPane.rowIndex="0" GridPane.columnIndex="0"/>
    </children>
</TabPane>
```

In addition to being more concise, static property attributes, like instance property attributes, support location, resource, and variable resolution operators, the only limitation being that it is not possible to create an expression binding to a static property.

Event Handlers
Event handler attributes are a convenient means of attaching behaviors to document elements. Any class that defines a "`setOn<Event>()`" method can be assigned an event handler in markup, as can any observable property (via an "on*<Property>*Change" attribute).

FXML supports two types of event handler attributes: script event handlers and controller method event handlers. Each are discussed below.

Script Event Handlers
A script event handler is an event handler that executes script code when the event is fired, similar to event handlers in HTML. For example, the following script-based handler for the button's "onAction" event uses JavaScript to write the text "You clicked me!" to the console when the user presses the button:

```
<?language javascript?>
...

<VBox>
    <children>
        <Button text="Click Me!"
            onAction="java.lang.System.out.println('You clicked me!');"/>
    </children>
</VBox>
```

Note the use of the `language` processing instruction at the beginning of the code snippet. This PI tells the FXML loader which scripting language should be used to execute the event handler. A page language must be specified whenever inline script is used in an FXML document, and can only be specified once per document. However, this does not apply to external scripts, which may be implemented using any number of supported scripting languages. Scripting is discussed in more detail in the next section.

Controller Method Event Handlers
A controller method event handler is a method defined by a document's "controller". A controller is an object that is associated with the deserialized contents of an FXML document and is responsible for coordinating the behaviors of the objects (often user interface elements) defined by the document.

A controller method event handler is specified by a leading hash symbol followed by the name of the handler method. The method is expected to conform to the signature of a standard event handler; that is, it should take a single argument of a type that extends `javafx.event.Event` and should return void (similar to an event delegate in C#). For example:

```
<VBox fx:controller="com.foo.MyController"
    xmlns:fx="http://javafx.com/fxml">
    <children>
        <Button text="Click Me!" onAction="#handleButtonAction"/>
    </children>
</VBox>
```

Note the use of the `fx:controller` attribute on the root element. This attribute is used to associate a controller class with the document. If `MyController` is defined as follows:

```
package com.foo;

public class MyController {
    public void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
    }
}
```

the text "You clicked me!" will be written to the console when the user presses the button, same as in the previous script-based example.

Controllers are discussed in more detail in a later section.

Scripting

The `<fx:script>` tag allows a caller to import scripting code into or embed script within a FXML file. Any JVM scripting language can be used, including JavaScript, Groovy, and Clojure, among others. Script code is often used to define event handlers directly in markup or in an associated source file, since event handlers can often be written more concisely in more loosely-typed scripting languages than they can in a statically-typed language such as Java.

For example, the following markup defines a function called `handleButtonAction()` that is called by the action handler attached to the `Button` element:

```
<?language javascript?>

<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns:fx="http://javafx.com/fxml">
    <fx:script>
    importClass(java.lang.System);

    function handleButtonAction(event) {
       System.out.println('You clicked me!');
    }
    </fx:script>

    <children>
        <Button text="Click Me!" onAction="handleButtonAction(event);"/>
    </children>
</VBox>
```

Clicking the button triggers the event handler, which invokes the function, producing output identical to the previous examples.

Script code may also be defined in external files. The previous example could be split into an FXML file and a JavaScript source file with no difference in functionality:

example.fxml

```
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<VBox xmlns:fx="http://javafx.com/fxml">

```

```
    <fx:script source="example.js"/>

    <children>
        <Button text="Click Me!" onAction="handleButtonAction(event);"/>
    </children>
</VBox>
```

example.js

```
importClass(java.lang.System);

function handleButtonAction(event) {
   System.out.println('You clicked me!');
}
```

It is often preferable to separate code from markup in this way, since many text editors support syntax highlighting for the various scripting languages supported by the JVM. It can also help improve readability of the source code and markup.

Note that script blocks are not limited to defining event handler functions. Script code is executed as it is processed, so it can also be used to dynamically configure the structure of the resulting output. As a simple example, the following FXML includes a script block that defines a variable named "labelText". The value of this variable is used to populate the text property of a `Label` instance:

```
<fx:script>
var myText = "This is the text of my label.";
</fx:script>

<Label text="$myText"/>
```

Controllers

While it can be convenient to write simple event handlers in script, either inline or defined in external files, it is often preferable to define more complex application logic in a compiled, strongly-typed language such as Java. As discussed earlier, the `fx:controller` attribute allows a caller to associate a "controller" class with an FXML document. A controller is a compiled class that implements the "code behind" the object hierarchy defined by the document.

As shown earlier, controllers are often used to implement event handlers for user interface elements defined in markup:

```
<VBox fx:controller="com.foo.MyController"
    xmlns:fx="http://javafx.com/fxml">
    <children>
        <Button text="Click Me!" onAction="#handleButtonAction"/>
    </children>
</VBox>
```

```
package com.foo;

public class MyController {
    public void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
    }
}
```

In many cases, this is sufficient. However, the `javafx.fxml.Initializable` interface can be used when more control over the behavior of the controller and the elements it manages is required. This interface defines a single method, `initialize()`, which is called once on an implementing controller when the contents of its associated document have been completely loaded:

```
public void initialize(URL location, Resources resources);
```

This allows the implementing class to perform any necessary post-processing on the content. It also provides the controller with access to the resources that were used to load the document and the location that was used to resolve relative paths within the document (often equivalent to the location of the document itself).

For example, the following code overrides the `initialize()` method to attach an action handler to a button in code rather than via an event handler attribute, as was done in the previous example. The `button` instance variable is injected by the loader as the document is read. The resulting application behavior is identical:

```
<VBox fx:controller="com.foo.MyController"
    xmlns:fx="http://javafx.com/fxml">
    <children>
        <Button fx:id="button" text="Click Me!"/>
    </children>
</VBox>
```

```
package com.foo;

public class MyController implements Initializable {
    public Button button;

    @Override
    public void initialize(URL location, Resources resources)
        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("You clicked me!");
            }
        });
    }
}
```

@FXML

Note that, in the previous examples, the controller member fields and event handler methods were declared as public so they can be set or invoked by the loader. In practice, this is not often an issue, since a controller is generally only visible to the FXML loader that creates it. However, for developers who prefer more restricted visibility for controller fields or handler methods, the `javafx.fxml.FXML` annotation can be used. This annotation marks a protected or private class member as accessible to FXML.

For example, the controllers from the previous examples could be rewritten as follows:

```
package com.foo;

public class MyController {
    @FXML
    private void handleButtonAction(ActionEvent event) {
        System.out.println("You clicked me!");
    }
}
```

```
package com.foo;

public class MyController implements Initializable {
    @FXML private Button button;

    @Override
    public void initialize(URL location, Resources resources)
        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                System.out.println("You clicked me!");
            }
        });
    }
}
```

In the first version, the `handleButtonAction()` is tagged with `@FXML` to allow markup defined in the controller's document to invoke it. In the second example, the `button` field is annotated to allow the loader to set its value.

Note, however, that `@FXML` can currently only be used with trusted code. Because the FXML loader relies on reflection to set member fields and invoke member methods, it must call `setAccessible()` on any non-public field. `setAccessible()` is a privileged operation that can only be executed in a secure context. This may change in a future release.

FXMLLoader

The `FXMLLoader` class is responsible for actually loading an FXML source file and returning the resulting object graph. For example, the following code loads an FXML file from a location on the classpath relative to the loading class and localizes it with a resource bundle named "com.foo.example". The type of the root element is assumed to be a subclass of `javafx.scene.layout.Pane`:

```
URL location = getClass().getResource("example.fxml");
ResourceBundle resources = ResourceBundle.getBundle("com.foo.example");

FXMLLoader fxmlLoader = new FXMLLoader();
fxmlLoader.setLocation(location);
fxmlLoader.setResources(resources);
fxmlLoader.setBuilderFactory(new JavaFXBuilderFactory());

InputStream inputStream = null;
Pane root;
try {
    inputStream = location.openStream();
    root = (Pane)fxmlLoader.load(inputStream);
} finally {
    if (inputStream != null) {
        inputStream.close();
    }
}
```

While it is possible (and occasionally necessary) to create a actual instance of `FXMLLoader` this way, it is usually more convenient to use one of the static `load()` methods defined by `FXMLLoader` to process FXML content:

```
public static <T> T load(URL location) { ... }
public static <T> T load(URL location, ResourceBundle resources) { ... }
public static <T> T load(URL location, ResourceBundle resources,
    BuilderFactory builderFactory) { ... }
```

The first version takes the location of the FXML document to load and returns the root element of the processed file. The second version adds a `Resources` argument that will be used to localize the processed content. The third version adds a `BuilderFactory`. The first two versions delegate to this method and pass an instance of `JavaFXBuilderFactory` by default.

Note that the output of an `FXMLLoader#load()` operation is an instance hierarchy that reflects the actual named classes in the document, not `org.w3c.dom` nodes representing those classes. Internally, `FXMLLoader` uses the `javax.xml.stream` API (also known as StAX) to load an FXML document. StAX is an extremely efficient event-based XML parsing API that is conceptually similar to its W3C predecessor, SAX. It allows an FXML document to be processed in a single pass, rather than loaded into an intermediate DOM structure and then post-processed.