Hint on retaining backward compatibility

How to inform users that a plugin's data structure is not backward compatible

See Marking a new plugin version as incompatible with older versions

Scenario: Adding a new field

Persistence is XML-based, and when the data is read back from disk and XML doesn't contain data for a particular field, the existing field value is left as-is. "As-is" normally means the JVM default value (0, null, etc.), because the persistence doesn't invoke your constructor to create your object, like Java serialization.

There are a few exceptions to this, where objects first create themselves and load XML onto itself. This includes Jenkins and Item, but these exceptions are limited to the root object of persistence. In these exceptional cases, the values set in the constructor will survive.

If you want to fill in your field with a non-trivial default value, you can write the `readResolve` method, which gets invoked right after your object is resurrected from persistence. `readResolve` is called by XStream, but is not part of a Jenkins class hierachy, so there is no override. Just put it right in your class alongside your fields, for example in the build pipeline plugin:

```
    protected Object readResolve() {
        if (gridBuilder == null) {
            if (selectedJob != null) {
                gridBuilder = new DownstreamProjectGridBuilder(selectedJob);
            }
        }
        return this;
    }
```

If you need to force the *Manage Old Data* screen to list jobs, builds, etc. using your data in the old format so that it can be saved in the new format in bulk, you cannot use `readResolve` since it will not notify this system of the problem. Instead you must create a static nested class called `ConverterImpl` extending `XStream2.PassthruConverter`, which should clean up the storage of your instance and finally call `OldDataMonitor.report` to record the conversion.

Scenario: Remove a field

Removing a field requires that you leave the field with the transient keyword. When Jenkins reads the old XML, XStream will set the value for this field, but it will no longer be written back to XML when data is saved.

Scenario: Rename a field

Renaming a field is a combination of the above: mark your old field as transient, declare your new field, and then migrate the data from the old format to the new in your `readResolve()` method:

```
    protected transient Long myObjectId;
    protected List<Long> myObjectIds;


    protected Object readResolve() {
        if (myObjectId != null) {
           myObjectIds = Arrays.asList(myObjectId)
        }
        return this;
    }
```

Scenario: Make a class abstract and introduce concrete subtypes

Before

- You decide to extend a class and create new choosable classes, e.g. more browsers for a SCM-plugin.
- The old data structure looked like this when you had only one class `SCMBrowser`:

```
    <browser>
      <url>http://yahoo.com/</url>
    </browser>
```

After

- Now you decide to add a new `NewSCMBrowser`, all your `SCMBrowsers` are extending `SCMBrowserBase` and your XML suddenly looks like this:

```
    <browser class="org.jenkinsci.plugins.foo.NewSCMBrowser">
      <url>http://yahoo.com/</url>
    </browser>
```

or

```
    <browser class="org.jenkinsci.plugins.foo.SCMBrowser">
      <url>http://yahoo.com/</url>
    </browser>
```

- With new jobs, no problem. Old jobs however will probably break.
- In your `SCMBrowserBase` class add a method `readResolve` (see XStream FAQ):

```
   // compatibility with earlier plugins
    public Object readResolve() {
        if (this.getClass()!=SCMBrowserBase.class) {
            return this;
        }
        // make sure to return the default SCMBrowser only if we no class is given in config.
        try {
            return new SCMBrowser(url.toExternalForm());
        } catch (MalformedURLException e) {
            throw new RuntimeException(e);
        }
    }
```

Scenario: rename a class

Sometimes, you need to rename packages or class names. If your serialization data includes a fully qualified class name (which happens for example if you have a collection of them), then a measure must be taken to maintain backward compatibility.

To do this, use `XSTREAM2.addCompatibilityAlias(String,Class)` to register aliases. You need to do this against the right XStream instance, as a few different instances are used to persist different parts of data.

`Items.XSTREAM2` is used for serializing project configuration, and `Run.XSTREAM2` is used for serializing build and its associated Actions.

For example, to alias `Foo` in the "old" package to the "updated" one, you can use this method call:

```
Items.XSTREAM2.addCompatibilityAlias("org.acme.old.Foo", org.acme.updated.Foo.class);
```

To ensure your alias is registered early in the Jenkins boot sequence, you can use the Initializer annotation on a static method, e.g. in your `DescriptorImpl`:

```
@Initializer(before = InitMilestone.PLUGINS_STARTED)
public static void addAliases() {
    Items.XSTREAM2.addCompatibilityAlias("org.acme.old.Foo", Foo.class);
}
```

Scenarion: rename or move the descriptor class of a plugin

Since 1.507 Descriptor#getConfigFile() is overridable and XmlFile can be instantiaded with any XStream instance.

TODO Examplecode (Weltramschaf)