PUBLIC-FACING DOCUMENT

JsInterop v1.0:
Nextgen GWT/JavaScript Interoperability

Goktug Gokdogan - Google
Last updated: Oct 22, 2015

Overview

As more complex applications are developed with GWT, more and more projects started to use GWT together with other JavaScript technologies, like Closure. While in many cases, developers are just reusing already available JavaScript code in a GWT application, there are also projects that require GWT as part of a bigger application developed with JavaScript. With the introduction of Web Components, we are going to have even more complex JavaScript integration scenarios, like extending HTML Elements.

This document describes some essential pieces to provide better and easier interoperability with JavaScript.

Table of Contents

Migrating from JsInterop experimental release

Design

JsInterop is mostly about providing powerful tools to modify the JavaScript code generation to mimic hand-written JavaScript code and adapt the impedance mismatch between two languages so that they can work naturally together.

@JsType

JsType is used to describe the interface of a Javascript object. Calls to methods in a JsType are specially treated by GWT compiler for interoperability purposes. JsType creates a contract between the Java type system and the external JavaScript world. The contract says that any instance behind this interface **must have methods or properties with the original name in JavaScript** (i.e **not renamed, not pruned)**.

To illustrate this, let's take a look at an example:

```
package com.acme;

@JsType
public class Foo {
  public int x;
  public int y;

  public int sum() {
    return x + y;
  }
}
```

As the class is annotated with @JsType, it will preserve its original names for all of its public API. This makes the class easily accessible from JavaScript:

```
// Note that exporting of Java Objects to JavaScript to be accessed by their namespace (e.g. this
// sample) requires --generateJsInteropExports flag.
var foo = new com.acme.Foo();
foo.x = 40;
foo.y = 2;
foo.sum(); // will return 42!
```

As you would expect, the API contract defined by a JsType is still preserved through type hierarchy. However note that, as the JsType annotation is not inherited, the subtypes are not automatically exposed directly to JavaScript.

To illustrate this, let's take a look at an example:

```
// in Java
package com.acme;

@JsType
public class Foo {
  public static Foo createFooSubclass() {
    return new FooSubclass();
  }

  public int x;
  public int y;

  public int sum() {
    return x + y;
  }
}

public class FooSubclass extends Foo {
  @Override
  public int sum() {
    return x + y + 3;
  }
  public int bar() {
    return 35;
  }
}

// in JavaScript

new com.acme.FooSubclass(); // FAILS! FooSubclass itself is not marked with JsType.

// Let's create the instance with the helper in Foo.
var fooSub = com.acme.Foo.createFooSubclass();
fooSub.x = 40;
fooSub.y = 2;
fooSub.sum(); // will return 45!

// As FooSubclass itself is not marked @JsType, 'bar' is not part of the JavaScript API
fooSub.bar(); // FAILS! throws exception - bar is undefined.
```

Similar to class examples, interfaces can also be marked with @JsType annotation.
In that case any implementor of the interface will maintain the public JavaScript contract defined by the super interface:

```
// in Java
```

```
package com.acme;

@JsType
public interface FooInterface {
  public static FooSubclass createFooSubclass() {
    return new FooSubclass();
  }
  int sum();
}

public class FooSubclass implements FooInterface {
  @Override
  public int sum() {
    return 42;
  }
  public int bar() {
    return 35;
  }
}

// in JavaScript

var fooSub = com.acme.FooInterface.createFooSubclass();
fooSub.sum(); // will return 42!

// As FooSubclass itself is not marked @JsType, 'bar' is not part of the JavaScript API
fooSub.bar(); // FAILS! throws exception - bar is undefined.
```

@JsProperty / @JsMethod / @JsConstructor

In addition to the ability of defining the JavaScript API of a java object at class level via @JsType, members could be individually exposed as well for finer control.

```
// in Java
package com.acme;

Public class MyJavaObject {
  @JsProperty
  public int x = 42;

  public int y = 55;

  @JsConstructor
  public MyJavaObject() { ... }

  @JsMethod
  public void action1(String x) { /* some java code here */ }
```

```
  public void action2(String y) { /* some java code here */ }
}

// in JavaScript

var obj = new com.acme.MyJavaObject(); // works!
obj.x; // works!
obj.action1("action!!"); // works!

obj.y; // FAILS! Not annotated with @JsProperty
obj.action2("action!!"); // FAILS! Not annotated with @JsMethod
```

Actually, @JsType is mostly a shortcut for assigning these lower level annotations to all public members of the class.

Fine tune code generation with ‘name’ & ‘namespace’

If you noticed from previous examples, JsInterop system uses java identifier while providing the JavaScript API. This may not be always what is desired. For example; the code may want to be placed in a different namespace instead of following Java conventions, or the desired JavaScript name might be an invalid java identifier. In such cases, JsInterop gives you finer control on the names using name and namespace attributes.

```
// in Java
package com.acme;

@JsType(namespace = "acme", name = "MyJavaScriptObject")
public class MyJavaObject {
  @JsProperty(name = "answerToLife")
  public int x = 42;

  @JsMethod(name = “import”) // cannot use “import” as method name in java
  public void importImpl(String x) { /* some java code here */ }
}

// in JavaScript

var obj = new acme.MyJavaScriptObject(); // works!
obj.answerToLife; // works!
obj.import("action!!"); // works!

obj.x; // FAILS! renamed
obj.importImpl("action!!"); // FAILS! renamed
new com.acme.MyJavaObject(); // FAILS! renamed
```

In practice you usually don't want to individually provide a namespace for each class as that is repetitive and not refactoring friendly. In many cases, you want a shared namespace for all classes in the same package in that case, you can just customize the namespace via package-info.java using @JsPackage annotation:

```
// package-info.java
@jsinterop.annotations.JsPackage(namespace = "acme")
package com.acme.shared;
```

In this case all types inside com.acme.shared package will use default namespace "acme" instead of "com.acme.shared".
Note that a type may still override its namespace by setting the namespace attribute, package namespace only changes the default.

Native JsTypes

As described earlier @JsType creates a contract between the Java type system and the external JavaScript world. From the Java side, the contract says that any instance behind this interface **must have methods or properties of the name described by the interface**. This contract does not necessarily have to be provided in Java; actually it can be fulfilled by JavaScript objects available in native environment. In that case the contract can be marked with "isNative":

```
// in JavaScript
com = { acme: {} };
com.acme.Foo = function() {
  this.x = 40;
  this.y = 2;
};
com.acme.Foo.prototype.sum = function() { return this.x + this.y; };

// in Java
package com.acme;

@JsType(isNative = true)
public class Foo {
  public int x;
  public int y;

  public native int sum();
}

public class FooMain {
  public static void main() {
    Foo foo = new Foo();
```

```
    foo.sum(); // will return 42!
    foo.x = 50;
    foo.y = 5;
    foo.sum(); // will return 55!
  }
}
```

Please note that the methods in native types can only be native and there should be no initialization logic in the Java file (i.e. no assignment in field declarations & only constructors with empty body - except the @JsOverlay members which will see in advanced topics). Not conforming to the restriction will result in compilation error.

Such contracts could be also provided as an interface. This helps testability but also useful in cases where the contract is not backed by a single JavaScript object, or the type doesn't matter (e.g. object literals, objects with unknown origin):

```
// in Java
package com.acme;

@JsType(isNative = true)
public interface Foo {
  int getFoo();
}

public class Bar {
  @JsMethod
  public static int action(Foo foo) {
    return foo.getFoo();
  }
}

// in JavaScript

var foo = { getFoo: function() { return 42; } };
com.acme.Bar.action(foo); // will return 42!
```

See section Property setters & getters for handling of properties in native JsType interfaces.

Extending JavaScript types in Java

As we have seen in the previous section, JavaScript types are now representable like regular classes in Java. This enables powerful features like extending JavaScript objects (which will be very useful when web components start supporting ES6 classes).
This is as simple as extending a native JsType class like a regular class:

```
// in Java
package com.acme;

public class CustomElement extends HTMLElement {
  public void doFancy() { ... }
}

@JsType(namespace = JsPackage.GLOBAL, isNative = true)
public class HTMLElement {
  public String id;
  public native void setAttribute(String name, String value);
}
```

@JsFunction

Another powerful feature of JsInterop is the ability to represent JavaScript function contracts between Java and JavaScript. That means a JavaScript function can be easily called from Java or a Java object can be called like a function in JavaScript:

```
// in Java
package com.acme;

@JsFunction
public interface Foo {
  int exec(int x);
}

@JsType
public class Bar {
  public static int action1(Foo foo) {
    return foo.exec(40);
  }
  public static Foo action2() {
    return (x) -> x + 2;
  }
}

// in JavaScript

com.acme.Bar.action1(function(x) { return x + 2; }); // will return 42!

var fn = com.acme.Bar.action2();
fn(40); // will return 42!
```

Note that for a type to be marked as @JsFunction it needs to have a single abstract method.

More Advanced Topics

Property setters & getters

If a @JsProperty is added to a method instead of a property, it designates that they are not to be invoked as methods but rather treated as field getter or setters. This provides ability to set property contracts via interfaces.

```
@JsProperty(name = "x") void setX(int x)  => compiled to this.x = x
@JsProperty(name = "x") int getX() => compiled to this.x
```

Actually the name attribute can be omitted if the setter/getter follows Java bean naming conventions.

Overriding/implementing @JsProperty in concrete classes

When @JsProperty applied to a method inside a concrete class, or a concrete Java class implements an interface with @JsProperty methods on it, the @JsProperty method behaves exactly like an ES6 setter/getter method for the property. That means @JsProperty setter method will be called on any attempt to set the property and @JsProperty getter method will be called on when the property is looked up:

```
@JsType
public class Foo {
   @JsProperty
   public int getX() {
     return 42;
   }
}

// in JavaScript
var foo = new com.acme.Foo();
foo.x; // returns 42!
```

Overlay Methods in native JsTypes

Sometimes the API of a native type is not good enough for the needs of the Java apps and the developers may need to extend the API without wrapping the existing type to avoid extra code and performance hit. This could be achieved by adding overlay methods to native JsTypes:

```
@JsType(isNative=true)
public class FancyWidget {
   public native boolean isVisible();
   public native void setVisible(boolean visible);
```

```
  @JsOverlay
  public final void toggle() {
    setVisible(!isVisible());
  }
}
```

Note that the overlay methods cannot be called from JavaScript, cannot override any existing methods and needs to be marked as final. Underneath, the original type is not modified and the method is simply turned into a static dispatch.

The instanceOf operator and type casts

The instanceOf behavior is specialized for native JsTypes.
Native JsType classes follow JavaScript instanceof operator semantics. However as native JsType interfaces do not map to a single object in JavaScript, there is no sensible semantic to apply hence instanceof operator is forbidden for them.

Note that Java classes that extend or implement a native @JsType is still Java type and will continue to follow Java semantics.

On the casting side, a cast to a native JsType from an object that returns false to JavaScript instanceof will throw ClassCastException. On the other hand, casts to native JsType interfaces never throw an exception for the similar reasons explained above.

Here are some samples on how instanceOf behaves on different scenarios:

```
@JsType(namespace=GLOBAL, isNative=true)
public class HTMLElement extends Element { ... }

// Note that following points to same JavaScript class as the above class. This might happen
// by using two different libs in the same app that doesn't share the native class.
@JsType(namespace=GLOBAL, name="HTMLElement", isNative=true)
class AnotherHTMLElementAbstraction { ... }

class MyCustomElement extends HtmlElement { ... }

HTMLElement obj = new HTMLElement();
obj instanceOf Object == true
obj instanceOf HTMLElement == true
obj instanceOf AnotherHTMLElementAbstraction == true
obj instanceOf MyCustomElement == false
obj instanceOf Iterator == false
obj instanceOf JsWindow == false


MyCustomElement obj = new MyCustomElement();
obj instanceOf Object == true
obj instanceOf HTMLElement == true
```

```
obj instanceOf AnotherHTMLElementAbstraction == true
obj instanceOf MyCustomElement == true
obj instanceOf Iterator == false
obj instanceOf JsWindow == false
```

Caveats & Special cases

@JsType

1. @JsType is not transitive. An interface/class needs to be specifically marked with the annotation to be a @JsType.
2. A @JsType cannot have multiple methods/properties with the same JavaScript name (see Avoiding name collisions).
3. There could be only one @JsConstructor in a type and if other constructors exist they need to delegate to it.
4. A @JsType should @JsIgnore all public constructors that doesn't fit to above rule.

@JsType(isNative=true)

1. A native @JsType can only extend/implement native @JsTypes.
2. A native @JsType class can only have
    - native methods
    - uninitialized fields
    - Empty constructors
    - Final non-native JsOverlay methods that do not override any other methods

@JsFunction

1. A @JsFunction interface cannot extend other interfaces.
2. A @JsFunction interface can only have JsOverlay default methods.
3. A class that implements a @JsFunction type (directly or indirectly) cannot be a @JsType nor have JsConstructor/JsMethods/JsProperties.
4. A class cannot implement other interfaces while implementing @JsFunction interface.

Tips & Tricks

Using native @JsMethods to access global object

You can set the namespace of a native method to GLOBAL so that it will match to native methods in global. By this way you can provide quick access to global members.

```
public class FooMain {

  @JsMethod(namespace=GLOBAL)
  public static native boolean isNaN(double number);
```

```
  public static void main() {
    isNaN(0d/0d); // will return true!
  }
}
```

Avoiding name collisions

Because @JsType is used to create a JavaScript contract, the contract is bound with JavaScript semantics. One very important implication of that is, every member name in the contract should be unique. A very common way to cause a name collision is method overloading (see @JsType javadoc for more scenarios where a name collision might occur).
If a name collision happens, the following are some common ways to fix it:

- The member could be renamed to avoid the collision.
- A different JavaScript name can be supplied for the member (@JsMethod#name etc.).
- The visibility of the member can be changed to non-public.
- The member can be marked with @JsIgnore so it is ignored by JsInterop.

Note that name collision are allowed in native members since no code is generated for them.

JsInterop enforces this by a rule called “one live implementation”;
For a given class, as long as there is single method that is live to provide an implementation for the ‘name’; it is allowed.

Hide JsInterop entry points to decrease code size

All static members and constructors in @JsTypes act as entry points to GWT applications as they are reachable by native JavaScript code. Because of that GWT compiler cannot prune any code that are reachable from these nodes even when they are not referenced by Java code.
If the programmers are not careful about this, the application size can increase rapidly.

To keep the code size in control, any class marked with @JsType should be carefully inspected to remove unnecessary entry points and hide them from JavaScript. This could be achieved by either decreasing the visibility or adding @JsIgnore where changing visibility is not feasible:

```
@JsType
public class Foo {
  // @JsIgnore’d method is not exposed to JavaScript, hence will not serve as an entry point
  @JsIgnore
  public static void bar() { ... }

  // Hidden constructor is not exposed to JavaScript, hence will not serve as an entry point
  private Foo() { ... }
}
```

Also note that, some shared libraries including Java Standard Library emulation (aka JRE) need to have JsInterop exported classes/members to ease jsinterop. Hence these libraries can also act as entry points of the application. As a result, this will incur cost to all projects even if they don't need them. To avoid that exporting symbols is disabled by default and GWT compiler requires `-generateJsInteropExport` to be passed.
For projects that need to enable exporting but need further control of exported symbols, also `-includeJsInteropExport/-excludeJsInteropExport` flags provided for finer control.

Appendix

Migrating from JSNI to JsInterop

A simple JSNI method can be directly replace with native JsMethod.

```
public native double max(double x, double y) /*-{
  return Math.max(x, y);
}-*/;
```

could be written as:

```
@JsMethod(namespace="Math")
private static native double max(double x, double y);
```

Tip: If you are using many native methods from the same namespace, you can consider introducing a native JsType class to abstract it instead:

```
@JsType(isNative=true, namespace=JsPackage.GLOBAL)
final class Math {
  public static double PI;
  public static native double max(double x, double y);
  public static native double min(double x, double y);
  ....
  // Hide the constructor as it is not instantiable
  private Math() {}
}
```

Similarly, a more complex JSNI code could be re-written in Java by the help of native JsMethods:

```
public native double relativeTimeMillis() /*-{
  if ($wnd.performance) {
    return $wnd.performance.now();
  }
  return new Date().getTime();
}-*/;
```

could be written as:

```
@JsType(isNative=true)
interface Performance {
   double now();
}
```

```
@JsProperty(namespace=JsPackage.GLOBAL)
private static native Performance getPerformance();

public double relativeTimeMillis() {
  if (getPerformance() != null) {
    return getPerformance().now();
  }
  return new JsDate().getTime();
};
```

Sometimes, JSNI could be referring to other Java code using legacy JSNI syntax:

```
<example here>
```

In that case you need to mark the accessed member (method, field etc.) with corresponding JsInterop annotation (JsProperty, JsMethod etc.) so their names will not be mangled hence any reference can be written by the original name:

```
 <translated example>
```

Migrating from JSOs to JsInterop (Overlay types)

You can mark each method in a JSO with @JsOverlay (that's actually what they are) and replace the JSNI by following the instructions above.

< Examples >

Migrating from JsInterop experimental release

- ❏ Replace all c.g.gwt.core.client.js.* annotations with jsinterop.annotations.*
- ❏ Replace @JsExport on classes with @JsType
- ❏ Replace JsExport on methods with JsMethod (same for JsConstructor and JsProperty)
- ❏ Replace JsNoExport with JsIgnore
- ❏ Replace JsNamespace on package with JsPackage
- ❏ Replace JsNamespace on all other places with corresponding namespace attribute in JsType, JsMethod,JsProperty
- ❏ Replace JsType(prototype="a.b.C") with JsType(namespace="a.b", name="C", isNative=true); Not that, you can omit name or namespace if it is already correct for the type.
- ❏ Add isNative=true to @JsType if the JsType interface might be backed by a pure JavaScript object.
- ❏ As now JsType exports static methods and constructors as well, all public static methods/fields/constructors needs to be inspected and @JsIgnore should be added where necessary. If there are too many, you can remove JsType and add JsMethod/JsPropery/JsConstructor where necessary.
- ❏ Remove `-XjsInteropMode JS` flag and add `-generateJsInteropExports` flag.

If you are migrating shared code between multiple projects, you can perform above steps without deleting old annotations so that the shared code work in both modes until all dependent projects get migrated.