Interoperability with JavaScript - JsInterop

Dr. Lofi Dewanto
https://lofidewanto.blogspot.de

Agenda

- JSNI (#deprecated)
- JsInterop
- Task

JSNI (JavaScript Native Interface)

JSNI

The GWT compiler translates Java source into JavaScript. Sometimes it's very useful to **mix** handwritten JavaScript into your Java source code. For example, the lowest-level functionality of certain core GWT classes are handwritten in JavaScript. GWT **borrows from the Java Native Interface (JNI) concept** to implement JavaScript Native Interface (JSNI). Writing JSNI methods is a powerful technique, but should be **used sparingly** because writing bulletproof JavaScript code is notoriously tricky. JSNI code is **potentially less portable** across browsers, more likely to **leak memory**, less amenable to Java tools, and **harder for the compiler to optimize**.

JSNI - Example

```
public static native void alert(String msg) /*-{
  $wnd.alert(msg);
}-*/;
```

JSNI - Bad Example

```
public static native int badExample() /*-{
  return "Not A Number";
}-*/;

 public void onClick () {
   try {
      int myValue = badExample();
      GWT.log("Got value " + myValue, null);
   } catch (Exception e) {
      GWT.log("JSNI method badExample() threw an exception:", e);
   }
 }
```

JSNI Problem

… in this case, neither the Java IDE nor the GWT compiler could tell that there was a type mismatch between the code inside the JSNI method and the Java declaration. The GWT generated interface code caught the problem at runtime in development mode. **When running in production mode, you will not see an exception**. JavaScript's dynamic typing obscures this kind of problem.

JSNI Tip and Docs

- Tip: Since JSNI code is just regular JavaScript, you will not be able to use Java debugging tools inside your JSNI methods when running in development mode. However, you can set a breakpoint on the source line containing the opening brace of a JSNI method, allowing you to see invocation arguments. Also, the **Java compiler and GWT compiler do not perform any syntax or semantic checks on JSNI code, so any errors in the JavaScript body of the method will not be seen until run time.**
- JSNI documentation: http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJSNI.html

JsInterop

JsInterop

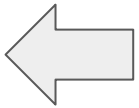
- JsInterop is a way of interoperating Java with JavaScript.
- It offers a better way of communication between the two using annotations instead of having to write JavaScript in your classes (using JSNI)

JsInterop: Exporting Java Type to JS

- JS → Java
- Annotation: `@JsType` == Preserve its original name
  - Exposes all the **public non-static fields and methods**, and tells the GWT compiler that the type is to be exported to a JavaScript type
- Annotating a class with `@JsType` is equivalent to annotating all its public non-static methods with `@JsMethod`, its constructor with `@JsConstructor` (only one `@JsConstructor` is allowed to exist

JsInterop: Exporting Java Type to JS

```java
package com.gwt.example;

@JsType
public class MyClass {

    public String name;

    public MyClass(String name) {
        this.name = name;
    }

    public void sayHello() {
        return "Hello" + this.name;
    }
}
```



```javascript
//the package name serves as a JS namespace
var aClass = new com.gwt.example.MyClass('World');

console.log(aClass.sayHello());

// result: 'Hello World'
```

JsInterop: Importing JS Type to Java

- Java → JS
- Constructors of native types **must be empty** and cannot contain any statement, except the call to the super class: **super()** (checked at compile time).
- Native types **cannot have non-native methods, unless annotated with** `@JsOverlay` (checked at compile time).
- `@JsType` is **not transitive**. If the child objects are to be exposed to JavaScript, they need to be annotated as well.

JsInterop: Importing JS Type to Java

```
@JsType(isNative = true, namespace = JsPackage.GLOBAL)
public class JSON {
    public static native String stringify(Object obj);

    public static native Object parse(String obj);
}
```

JsInterop: Consuming JS Function with Callback Arg

- Java → JS
- JsInterop can also be used to map a JavaScript function to a Java interface using **@JsFunction**. Unlike Java, methods can be used as arguments to other methods in JavaScript (known as callback argument).
- A JavaScript callback can be mapped to a **Java functional interface** (an inteface with only one method) annotated with `@JsFunction`.

JsInterop: Consuming JS Function with Callback Arg

```java
@JsFunction
public interface EventListenerCallback {

    void callEvent(Object event);
}
```

```java
@JsType(isNative = true)
public class Element {
    // other methods

    public native void addEventListener(String eventType, EventListenerCallback fn);
}
```

```java
Element element = DomGlobal.document.createElement("button");
// using Java 8 syntax
element.addEventListener("click", (event) -> {

    GWT.log("clicked!");
});
```

JsInterop: Consuming JS Function with Callback Arg

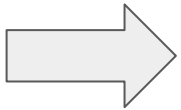
is (more or less) equivalent to the following JavaScript code:

```
var element = document.createElement("button");
element.addEventListener("click", (event) => {

  console.log("clicked!");
});
```

JsInterop - Expose Function to JS with Callback Arg

- JS → Java
- In the same fashion, if a Java type is to be passed as a callback, `@JsFunction` is to be used. The implementation of the callback can be done directly from JavaScript.

```
com.example.Bar.action1((x) => x + 2); // will return 42!

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



```
package com.example;

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

    public static int action1(Foo foo) {
        return foo.exec(40);
    }

    public static Foo action2() {
        return (x) -> x + 2;
    }
}
```

JsInterop - Additional Methods to Native Type

- Java → JS
- The JsInterop contract specifies that a native type may contain only native methods except the ones annotated with `@JsOverlay`.
- `@JsOverlay` allows adding a method to a native type (annotated with `@JsType(isNative=true)`) or on a default method of a `@JsFunction` annotated interface .
- The `@JsOverlay` contract specifies that the methods annotated should be final and should not override any existing native method. The annotated methods will not be accessible from JavaScript and can be used from Java only. `@JsOverlay` can be useful for adding utilities methods that may not be offered by the native type.

JsInterop - Additional Methods to Native Type

```
@JsType(isNative = true)
public class FancyWidget {

    public boolean visible;

    public native boolean isVisible();

    public native void setVisible(boolean visible);

    @JsOverlay
    public final void toggle() {
        visible = !visible;
    }
}
```

JsInterop

JsInterop documentation:

- http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJsInterop.html
- http://bit.ly/2rOd6o3
- http://www.gwtproject.org/javadoc/latest/jsinterop/annotations/package-summary.html

Example

- https://github.com/interseroh/gwt-jsinterop-sample

Task

- Export Java type to JS: JS → Java
  - `MyApp.click()`: implementation of your choice
- Import JS type to Java: Java → JS
  - `Window.alert("...")`: https://www.w3schools.com/jsref/met_win_alert.asp
  - `apple.js`: https://gist.github.com/lofidewanto/2c625d0f681918c8d387d7a7572af3ca

Result: https://github.com/interseroh/gwt-jsinterop-sample

References

- http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJSNI.html
- http://www.gwtproject.org/doc/latest/DevGuideCodingBasicsJsInterop.html