Object Literals and Duck-Typing in GWT (DRAFT)

Goktug Gokdogan - Google
Last updated: Nov 6, 2014

Objective

We would like to provide a Java idiom that could be mapped to duck-types in javascript, while providing a readable notation then can be translatable into object literals that optimizes well.

Background

One of the commonly used idioms that exists in Javascript and we are missing in the Java is the duck typing. A few years back, Ray Cromwell wrote an article related to this that you can take a look but I'll go with a simple sample to explain what we are missing and why it is important for Nextgen GWT JsInterop.

In javascript, a lot of APIs utilizes literal objects with optional properties and methods. For example:

```
var mapOptions = {
  zoom: 8,
  mapTypeId: google.maps.MapTypeId.ROADMAP,
  center: new google.maps.LatLng(41.850033, -87.6500523),
};
var map = new google.maps.Map(element, mapOptions);
```

We can handle such APIs with current JsInterop:

```
@JsType
interface Map {
  static Map create(Element e, MapOptions options) {
    return js("new google.maps.Map($0, $1)", e, options);
  }
}

interface MapOptions {
  @JsProperty void setZoom(double number);
  @JsProperty double getZoom();
  @JsProperty void setMapTypeId(int id);
  @JsProperty int getMapTypeId();
  @JsProperty void setCenter(LatLon center);
  @JsProperty LatLon getCenter();
}

MapOptions options = JsObject.create();
```

```
options.setZoom(8);
options.setMapTypeId(...);
options.setCenter(...);
Map map = Map.create(element, options);
```

However, there are some problems with this.

First problem is; there is huge amount of boilerplate (see the declaration of MapOptions class). We want to reduce the boilerplate. Ideally, we could have defined a java class with fields and mark it with @JsType however that will cause generation of prototype and this not what we want.

Second problems is; we are losing nice readable structured view that is available in javascript. See the following example:

```
Object.defineProperties(obj, {
    newDataProperty: {
        value: 101,
        writable: true,
        enumerable: true,
        configurable: true
    },
    newAccessorProperty: {
        set: function (x) {
             ..
        },
        get: function () {
             ..
        },
        enumerable: true,
        configurable: true
    }});
```

Assuming we already put all the boilerplate for the interfaces; this translates into:

```
JsMap properties = JsMap.create();

ObjectProperty property1 = ObjectProperty.create();
property1.setValue(101);
property1.setWritable(true);
property1.setEnumerable(true);
property1.setConfigurable(true);
properties.set("newDataProperty", property);
```

```
ObjectProperty property2 = ObjectProperty.create();
property2.set( (x) -> {
  ...
} );
property2.get( (x) -> {
  ...
});
property2.setEnumerable(true);
property2.setConfigurable(true);

/* I actually forgot this line in my first attempt */
properties.set("newAccessorProperty", property);

JsObject.defineProperties(obj, properties);
```

And the last issue is; this generates an imperative javascript block instead of using better optimizable object literal syntax.

Design

The basic idea to introduce new type of JsType called ‘literal types’ into our type system.

```
@JsType(literal=true)
class MapOptions {
  public double zoom;
  public int mapTypeId;
  public LatLon center;
}
```

When a JsType is marked with ‘literal’ attribute (hence a ‘literal type’), the generated javascript will no longer have a prototype or constructor function.
Anonymous subclasses of literal types are special and called ‘literal initializers’. Literal initializer can be used to instantiate and initialize the instance in place:

```
MapOptions options = new MapOptions() {{
    zoom = 8;
    mapTypeId = ...
    center = ...
}};
```

This will generate following javascript:

```
var mapOptions = {
  zoom: 8,
  mapTypeId: ...,
  center: ...,
};
```

As there is no prototype, any type can be casted to a literal type as there will be no runtime cast-checking:

```
@JsType(literal=true)
abstract class JsStruct extends JsObject {
  final <T> T as(Class<T> clazz) {
    return js("$0", this);
  };
}

MapOptions options = new JsStruct() {
  double zoom = 8;
  int mapTypeId = ...
  LatLon center = ...
}.as(MapOptions.class);
```

Imagine how this can be utilized as a super simple rpc solution:

```
Xhr.post("/update", new JsStruct() {
  String name = "Joe";
  int age = 15;
  JsStruct address = new JsStruct() {
    String street = "Crittenden";
    int zipCode = 94043;
  };
});
```

or an efficient map initializer:

```
JsMap st = new JsMap() { String a = "xyz", b = "klm"; }

// generates
var a = { a:”xyz”, b: “klm”}.
```

Let’s see a more complex example that brings all of these together:

```
JsObject.defineProperties(obj, new JsStruct() {
  ObjectProperty newDataProperty = new ObjectProperty() {{
      enumerable = true;
      configurable = true;
      value = 101;
      writable = true;
  }};
  ObjectProperty newAccessorProperty = new ObjectProperty() {{
    enumerable = true;
    configurable = true;
    set = (x) -> {
      ....
    };
    get = () -> {
      ....
    };
  }};
});
```

In order to make this work, we need to make some hard guarantees for these types:
NO_PROTOTYPE: a literal object cannot have a prototype.
NO_CLINIT: a literal object cannot have <clinit>.
TRIVIAL_INITIALIZATION: initialization should be trivial enough to be translatable into a javascript object literal.

To provide NO_PROTOTYPE guarantee, the compiler will enforce:

- A literal type can only set the prototype attribute to Object.
- A literal type can only extend other literal types but any other type can extend them.
- A literal type declaration cannot initialize the fields.
- A literal type declaration can have only final methods marked with @JsFinal.

For NO_CLINIT guarantee:

- A literal type declaration cannot have a static initializer.
- A literal type declaration can only declare a static fields that are ‘constants’ per JLS.

For TRIVIAL_INITIALIZATION guarantee:

- A literal type declaration cannot have a constructor.
- A literal type declaration cannot initialize the fields
- A literal type declaration cannot be an inner class.
- A literal type initializer can only have initialization expressions.
- An initialization expression cannot refer to instance members.

Additionally, setting literal attribute on an interface is an error.