JSweet: insights on motivations and design

A transpiler from Java to JavaScript

Renaud Pawlak
renaud.pawlak@jsweet.org
http://www.jsweet.org

November 16, 2015

Abstract

In this position paper, we present JSweet, a Java to JavaScript transpiler, which motivations originate in that its design takes advantage of the TypeScript transpiler to generate sound JavaScript code and APIs. We explain how JSweet uses the TSD repository to automatically generate all the JavaScript frameworks and libraries APIs in Java, so that they can be directly used from any JSweet-enabled Java program. We also explain how JSweet implements a two-layer design that allows Java and TypeScript to double-check the programs against the APIs. This double verification ensures that the source and transpiled programs are rigorously equivalent. We illustrate the capabilities of JSweet with two simple examples, showing the original and transpiled codes.

1 What is JSweet and why did we do it?

In the past few years, many source-to-source compilers (a.k.a *transpilers*) have been created to improve JavaScript. Among them, the most well-known are CoffeeScript [9], Dart [17], and TypeScript [11], but the list of such languages is much longer. Transpilers have become such an attractive alternative for programming Web applications, that some initiatives such as GPM [7] have raised to standardize the transpilation process in ECMAScript 6. Most transpilers over JavaScript propose a new syntax (CoffeeScript, Dart), or a syntax extension to JavaScript (TypeScript). Some approaches, such as Hop for Haskell [8] and the OCaml bytecode translator [19], propose to start from an existing language.

In this position paper, we discuss an approach called JSweet that falls in the latter category, starting from the Java language. JSweet is an Open Source Java to JavaScript transpiler that aims at programming modern Web applications (i.e. HTML + JavaScript) in plain Java, using our favorite Java IDEs. There are commonly two typically-given reasons why people would benefit such a transpiler.

- Firstly, with HTML5 and its new JavaScript API, modern Web browsers have become the most universal execution platform, efficient and plugable on most operating systems,. For instance, the support of SVG and the canvas API allows the programmers to develop cutting edge UI, CSS transforms allows for animations that harmlessly take advantage of the GPU, CSS media queries supports adaptive design, debugging and profiling is natively supported by modern

browsers, and so on. This makes it the most standard way to build cross-platform mobile applications, including mobile games [1].

- Secondly, Java remains the most efficient language for building complex applications, with its world-wide community support for various IDEs. Its simple syntax and strong typing (but not too strong), makes it a good compromise for most developer profiles. For instance, Eclipse's Java incremental compiler and refactoring tools make Java the most comfortable language to handle the complexity of large scale projects.

However, beyond these obvious reasons, many technical and "business"-oriented choices need to be made in order to build such a transpiler. In particular, the crucial issue to be tackled is the way to access the existing libraries, since it is a fact that, without libraries, a programming language is pointless. So the question that rapidly arises is: which libraries, and how do I get access to them?

1.1 The in-depth API transformation approach

In the past, many Java to JavaScript transpilers such as GWT [5] or Java2Script [6] made the choice to transform calls to a Java API (such as SWT for instance) into calls to native JavaScript libraries. This approach has the advantage of seamlessly leveraging the use of JavaScript APIs, so that Java programmers do not need to learn JavaScript at all. When GWT came out, it generated a lot of buzz and hope from the Java community, because most Java programmers do not want to learn JavaScript, but also because such a tool could save a lot of time (and money) when porting legacy Java code to build modern Web applications. As a consequence, people got all exited, because they hoped they could potentially save a lot of money by:

- not having to learn JavaScript at all and therefore being able to re-use Java skills and experienced teams to build Web applications,
- rapidly porting legacy Java software to Web software,
- building more complex applications than JavaScript ones because they would benefit static typing and professional-quality tools and IDEs.

However, real life does not work this way... In practice, there are several flaws to this kind of approaches. Building a transpiler that supports up-to-date Java and JavaScript APIs is extremely difficult and involves a significant delay when compared to projects that directly use the latest APIs natively. The root cause for this is that, in most cases, Java and JavaScript APIs are designed quite differently since the targeted runtime environments (a JRE and a Web Container) differ in many ways. As a consequence, the required in-depth transformations make the system complex to understand and, consequently to debug. In practice, when building more complex applications, it is naive to think that one will not need to understand the details of the generated code, when fine-tuning the Web applications, and trying to use lastest state-of-the-art frameworks. In short, we can enumerate at least two main reasons why this approach is not viable.

1. The impedance mismatch is high between the Java APIs and the JavaScript ones, which requires complex and in-depth transformations – as an example, try to imagine how complex it would be to build a transpiler for porting automatically all C++ applications to Java!

2. The Web languages, tools, and APIs evolve so fast, that it is mostly impossible to keep up with the pace and the variety of JavaScript frameworks out there.

So, why JSweet at all then?

1.2 The syntax mapper approach

Being agreed that trying to transform Java APIs calls to JavaScript APIs is a too complex task, there is a more realistic approach that consists of completely dropping the Java APIs, and actually using the JavaScript APIs in the Java language. With such an approach, the Java to JavaScript transpiler becomes a *syntax mapper* between the Java syntax and the JavaScript syntax, all the rest remaining unchanged. This approach was chosen by ST-JS [4] (Strongly Typed JavaScript), for instance. ST-JS proposes a Maven plugin and API bridges to well-known frameworks such as jQuery [2] in Java. As a consequence, programmers can use the JavaScript APIs directly in Java, and the transpiler just transforms the Java syntax to plain JavaScript syntax, so that it can run in a Web container.

Of course, with such an approach, programmers will not be able to automatically convert most legacy Java code to JavaScript code. For instance, a Swing-based application will never convert to JavaScript, since there is no corresponding Swing API in JavaScript. As we discussed in the previous section, this limitation is however a good and healthy limitation. Additionally, knowing that Java APIs will not be automatically transpiled, it does not mean that such a transpiler cannot help in manually porting legacy code, or sharing some part of the application between a Java and a JavaScript application. For instance, if the data model (transfer objects or persistent data layer) is written with the Java syntax, and if it does not use too much of the Java APIs, it is possible to share the same code in the Java-written server part, and in the JavaScript-written client part of the same application.

As a consequence, we believe that the syntax mapper approach is the best solution to bridge Java and JavaScript. Still, it remains the issue of accessing the JavaScript APIs from Java. How do we deal with fast pace releases and the variety of frameworks available in the JavaScript community? ST-JS does not bring any good solution for this problem, since the programmers will need to write JavaScript's API briges in Java, which is still a Herculean task.

1.3 The JSweet approach

JSweet basically improves the syntax mapper approach by bringing two major improvements compared to existing projects.

1. A JavaScript API repository automatically generated from the latest up-to-date well-typed TypeScript definitions available in TSD [16]. This repository includes more than 300 JavaScript libraries and is widely and freely accessible. The libraries dependencies are automatically mapped to Maven, and their potential side effects are handled by the transpiler, which is a key point of our approach.

2. A strong and flexible typing of JavaScript programs in Java, with types such as functional types, union types, tuple types, and string types.

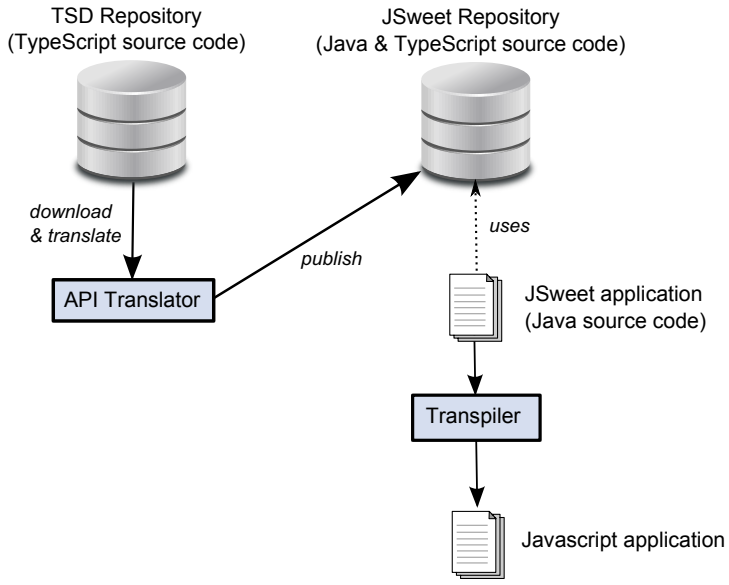
These improvements are made possible since JSweet leans on the TypeScript language and community [10], which is supported by Microsoft and Google. At the end

of the day, JSweet can be seen as a re-implementation of TypeScript in the Java syntax. It supports the equivalent interfaces, classes, enums, modules, and definitions structures. It bends the Java generics and uses advanced generation techniques in order to map to the excellent TypeScript type system. It takes advantage of existing TypeScript definition files to generate well-typed Java APIs for existing TypeScript APIs. Last but not least, it supports interface merge (like in TypeScript) between dependent APIs (for instance, jQuery plugins can impact the default jQuery APIs when used from JSweet).

In the remainder of this paper, we will present how JSweet was designed (Section 2). Through two examples, we will hopefully give you a head start on what you can expect if you decide to use it for programming Web applications (Section 3).

2 The JSweet design

JSweet consists of two parts: the *API translator* and the *transpiler*. The API translator is responsible for generating Java API for JavaScript libraries (including core APIs), and package those into Maven projects. This API generation is done upfront and is necessary to program JSweet applications, however final JSweet programmers do not need to use this part directly since they simply need to bind to the already-translated JSweet repository.



JSweet heavily relies on TypeScript to generate the APIs, but also to transpile to JavaScript. As we will explain in the upcoming sections, through its design, JSweet offers the same typing warranties, code quality, and up-to-date APIs as the excellent and widely used TypeScript transpiler, but for a Java syntax.

2.1 The API translator

The JSweet translator is an online automated service that extracts more than 300 TypeScript definition files from TSD [16] (from DefinitelyTyped [18]), which corresponds

to hundreds of JavaScript libraries and frameworks. It then transforms these definitions to Java APIs following the JSweet conventions. The API translator also translates the core JavaScript libraries (including the DOM) to Java. Each API (TypeScript and Java definitions), is then packaged in a Maven project and deployed as a JSweet library (a.k.a a JSweet *candy*) in the JSweet candy repository. Each packaged version of a given library corresponds to a given version in TSD, so that the JSweet repository updates its candies at the same pace as TSD.

The translator relies on a TypeScript parser, which generates a visitable modifiable AST. Then, a set of AST scanners are applied to the AST. Similarly to processors in the Spoon Java transpiler [14], each scanner performs elementary translation operations so that the initial TypeScript code structure will map the target Java code structure. Then, a Java pretty printer is applied to the final transformed AST to generate the Java source code, that defines the usable API from a JSweet program. Note that the translator may use several documentation sources, such as online Mozilla Developer Network (MDN) [12] in order to fill in Javadoc documentation for the generated Java code when the original TypeScript API misses documentation.

The following code excerpt shows the TypeScript for the `HTMLCanvasElement` interface in `lib.dom.d.ts`, the core TypeScript DOM library (`[...]` indicates snipped out parts).

```
interface HTMLCanvasElement extends HTMLElement {
  /**
   * Gets or sets the width of a canvas element on a document.
   */
  width: number;
  /**
   * Gets or sets the height of a canvas element on a document.
   */
  height: number;
  /**
   * Returns an object that provides methods and [...]
   * @param contextId The identifier (ID) of the type [...]
   */
  getContext(contextId: "2d"): CanvasRenderingContext2D;
  [...]
}
```

This TypeScript definition is automatically translated by the JSweet API translator to the following Java code:

```
1  package jsweet.dom;
2
3  /**
4   * The <strong><code><span>HTMLCanvasElement</span></code>
5   * </strong> interface provides properties and methods for [...]
6   */
7  @Interface
8  public class HTMLCanvasElement extends HTMLElement {
9    /**
10    * Gets or sets the width of a canvas element on a document.
11    */
12   public double width;
```

```
13    /**
14     * Gets or sets the height of a canvas element on a document.
15     */
16    public double height;
17    /**
18     * Returns an object that provides methods and [...]
19     *
20     * @param contextId
21     *          The identifier (ID) of the type of canvas [...]
22     */
23    native public CanvasRenderingContext2D getContext(
24        jsweet.util.StringTypes._2d contextId);
25    [...]
26  }
```

There are a few interesting points to be mentioned when comparing the original code to the generated code.

- Firstly, the JSweet translator generates `@Interface` annotated classes instead of interfaces (line 7), in order to allow field accesses in the APIs rather than getters and setters (like it is usually the case in Java). This design choice makes the application code closer to the JavaScript and TypeScript code: here the `width` and `height` fields can be accessed directly, which makes the code more readable. Although it is not the case here, the translator supports multiple inheritance by injecting the parent interfaces members into the classes, and by using an `@Extends` annotation to indicate that a dynamic cast is possible. As a direct consequence of this design choice, one can notice line 23 the use of the `native` modifier, which indicates that the `getContext` method is simply a prototype for a JavaScript method (it is important to remember that none of the generated code will be executed since it is a bridge to an existing JavaScript library, or to built-in JavaScript objects and functions).

- Secondly, at line 3, the `HTMLCanvasElement` class holds a Javadoc that was injected by the translator by fetching the online document from MDN, since the original definition does not contain any documentation for the interface.

- Thirdly, it is important to understand that the translator creates auxiliary types to ensure as strong typing as the TypeScript definitions. Line 24 shows the use of a `StringTypes` constant, which is automatically generated when a TypeScript string type (which do not exist in Java) is used.

JSweet and TypeScript definitions are meant to be fully equivalent. Since the TypeScript type system is richer than the Java one in most cases, JSweet uses conventions, extra annotations, classes and interfaces to generate the fully equivalent Java code.

2.2 The transpiler

. This part is a lightweight Java to TypeScript transpiler, which delegates the JavaScript code generation to the TypeScript compiler (*tsc*). The transpiler takes as input the Java sources of the program to be transpiled, the JSweet libraries as packaged *candies* and some transpilation options. The ouput is the JavaScript sources, and the `js.map` files, a.k.a. the *source map* files. A source map file is a standard JavaScript auxiliary file,

which contains the source position mapping between the JavaScript code elements, and the source code from which the JavaScript was transpiled. It is supported by all main browsers and allows the debugging of languages that are transpiled to JavaScript (such as Dart, CoffeeScript, and TypeScript). The JSweet transpiler is able to redirect the source map files generated by the TypeScript compiler (*tsc*), so that they map to the original Java source code, thus enabling the debugging of JSweet Java code directly in your favorite browser.

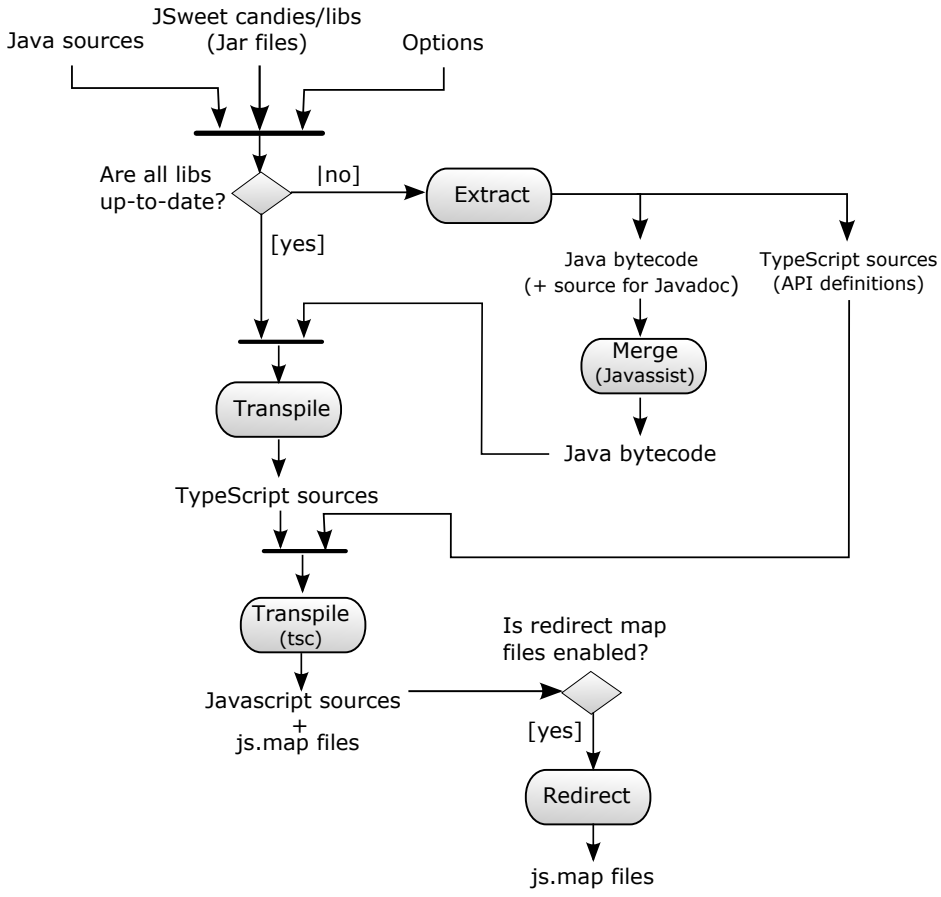


Figure 1: JSweet transpiler design

The simplified activity diagram of Figure 1, which we explain next, shows the main sub-processes (actions) that take part in the JavaScript generation.

1. If some candies have not been extracted yet, or if some have changed, they need to be updated, else we skip this part. The update process first cleans all temporary files and performs the following actions.

   (a) **Extract**: TypeScript definition files (.d.ts) are extracted so that they can be used in the upcoming transpilation process. Additionally, the extract task scans the JSweet definitions for types to be merged. For instance, a jQuery plugin can define other methods to the main `JQuery` class, but since the

plugin is not packaged in the same candy as jQuery itself, we need to extract all the classes to be merged and apply the following action.

(b) **Merge**: TypeScript natively supports the merging of interfaces, but not Java. Thus, this merge step allows JSweet to ensure that a given interface defined in various candies, can be merged on-the-fly so that the Java programmers can access its full definition. We first use the Javassist tool [3] to inject the methods directly at a bytecode level to the target classes.

2. **Transpile (Java → TypeScript)**: the TypeScript generation is the heart of the JSweet transpiler. It is implemented as a subclass of `com.sun.tools.javac.tree.TreeScanner` and is driven by core JSweet annotations. It requires the API definitions in the class path: all candy jars + the previously merged classes, which are added at the top of the class path so that they take precedence. Note that it is *javac* that performs all the basic type checking, since a JSweet program is, before all, a Java program that compiles against the JSweet APIs.

3. **Transpile (TypeScript → JavaScript with *tsc*)**: once the Java source code has been transpiled, if there is no Java compilation or JSweet transpilation problems, then JSweet delegates to TypeScript the final production of JavaScript code, which also generates the source map files. For transpilation, *tsc* requires the TypeScript that was generated from the original Java source files, but also, all the TypeScript definition files extracted from the candies. This step rigorously ensures that the Java and the TypeScript program are the same w.r.t the library APIs.

4. **Redirect**: if the transpiler option tells to redirect source map files to the Java files (which is the default), then the TypeScript generated source map files are redirected to the original Java files.

As we can see, the JSweet transpiler is designed so that the Java programs ensure the same level of type safety as TypeScript programs. Indeed, the Java programs are checked first against the Java APIs, and second against the TypeScript APIs, which ensures their equivalence. JSweet thus heavily relies on both Java and TypeScript's type checkers to ensure the correctness of JSweet programs. Additionally, the JSweet transpilation process (Java to TypeScript) performs the required syntactic and structural transformations to the code, and extra checks to ensure that the programmers write JSweet translatable code.

3 Examples

This section shows some typical use cases through simple examples.

3.1 Simple jQuery example

This example shows a very simple use case of the jQuery API [2], which is taken from the jQuery's website. The HTML document on which relies the program is just a Click me! link to jQuery official site (line 8).

```
1 <html>
```

```
  <head>
     <script src="lib/jquery.min.js"></script>
     <script src="lib/jquery-ui.min.js"></script>
    <script
        src="js/org/jsweet/examples/jquery/JQuery.js"></script>
  </head>
  <body>
     <a href="http://jquery.com/">Click me!</a>
  </body>
</html>
```

The JSweet code below uses jQuery to prevent the link to be opened by the user and shows an alert (line 13). To do so, it simply installs a click handler using the jQuery API (line 12-16). In the handler, it calls the `JQueryEventObject.preventDefault()`, which, as its name says, disables the default behavior of the clicked element (here, the link will not be opened).

```
package org.jsweet.examples.jquery;

import static def.jquery.Globals.$;
import static jsweet.dom.Globals.alert;
import static jsweet.dom.Globals.console;
import static jsweet.dom.Globals.document;

public class JQuery {
  public static void main(String[] args) {
    $(document).ready(() -> {
      console.info("starting JQuery example");
      $("a").click((event) -> {
        alert("JQuery is preventing this link to be opened");
        event.preventDefault();
        return null;
      });
      return null;
    });
  }
}
```

Let us now focus on that JSweet code with more details. As one can see, it is at first sight surprisingly close to a TypeScript program. It is made possible through the use of *static imports* and of Java 8 *lambda expressions*.

Java's *static imports* allow access to global variables such as `document` and `console` (lines 5-6), but also to global functions. Here, the `alert()` function is accessed globally (line 3), as well as the jQuery's famous `$` function imported line 3. Note that the `$` function is defined in `def.jquery.Globals` class, which means two things.

1. The `$` function belongs to the jQuery library, translated from TSD [16]. By convention, the JSweet translator generates libraries API definition in the `def` package. Each library is placed in a `def.libname` package. Core libraries, such as the DOM, are handled slightly differently and are placed in `jsweet` packages:

   (a) `jsweet.lang`, which contains the JavaScript and JSweet core APIs (prim-

itive objects, JSweet annotations, ...),

(b) `jsweet.dom`, which contains the DOM APIs,

(c) and `jsweet.util`, which contains some JSweet helpers (mainly for code simplification).

2. Similarly to `alert()`, `document`, and `console`, `$` is defined in a class called `Globals`, which means that it is globally defined in its module. Here, `Globals` being at the root of the `def.jquery` library, `$` corresponds to a JavaScript global function.

JavaScript intensively uses functions (lambdas) in its APIs. JSweet takes advantage of the Java 8 *functional interfaces* to make the code as close as possible to the JavaScript/TypeScript code. In our example, lambdas are used twice.

- Line 10: a function is passed to `document.ready(...)`, so that it can be asynchronously executed when the document has been fully loaded.
- Line 12: a function is passed to `$.click(...)`, so that it can be asynchronously executed when the links of the document (selected with `$("a")`) are clicked.

Java's lambda expressions can be used in JSweet programs since the JSweet API translator translates the TypeScript functional types to Java 8 functional interfaces. The prototype of `JQuery.click(...)` is thus translated as shown below, where the TypeSript functional type is replaced by a well-typed functional interface.

```
// Original TypeScript definition (from jquery.d.ts):
click(handler: (eventObject: JQueryEventObject) => any):
    JQuery;
// Transpiled Java definition:
import java.util.function.Function;
[...]
native public JQuery click(Function<JQueryEventObject,Object>
    handler);
```

This prototype allows the use of the lambda expression lines 12-16, which is passed as the click event's handler. In Java the `event` variable is well typed to the `JQuery-EventObject` type, like in TypeScript. On contrary to TypeScript, the "`return null;`" statement is mandatory in Java because the `handler` parameter is typed as a function that returns an object.

Finally, for the interested readers, let us show now the transpiled code for this example. Remember that there are two transpilation steps (step #1: Java → TypeScript, step #2: TypeScript → JavaScript), of which the resulting code is shown hereafter (note that *tsc* defaults to ECMAScript 3).

Transpilation step #1 (Java → TypeScript):

```
module org.jsweet.examples.jquery {
   export class JQuery {
      public static main(args: string[]) {
         $(document).ready(() => {
            console.info("starting JQuery example");
            $("a").click((event) => {
```

```
                alert("JQuery is preventing this link to be
                    opened");
                event.preventDefault();
                return null;
            });
            return null;
        });
    }
  }
}
org.jsweet.examples.jquery.JQuery.main(null);
```

Transpilation step #2 (TypeScript → JavaScript with *tsc*):

```
var org;
(function (org) {
   var jsweet;
   (function (jsweet) {
      var examples;
      (function (examples) {
         var jquery;
         (function (jquery) {
            var JQuery = (function () {
               function JQuery() {
               }
               JQuery.main = function (args) {
                  $(document).ready(function () {
                     console.info("starting JQuery example");
                     $("a").click(function (event) {
                        alert("JQuery is preventing this link
                            to be opened");
                        event.preventDefault();
                        return null;
                     });
                     return null;
                  });
               };
               return JQuery;
            })();
            jquery.JQuery = JQuery;
         })(jquery = examples.jquery || (examples.jquery =
             {}));
      })(examples = jsweet.examples || (jsweet.examples = {}));
   })(jsweet = org.jsweet || (org.jsweet = {}));
})(org || (org = {}));
org.jsweet.examples.jquery.JQuery.main(null);
//# sourceMappingURL=JQuery.js.map
```

3.2 Canvas animation example

This example shows a program that uses the HTML canvas API to draw an animated red circle that fills up counterclockwise. This program relies on an HTML document

shown hereafter, in which the used canvas is defined line 6. Also, the JSweet-generated JavaScript file is included in the document line 3.

```
1 <html>
2   <head>
3     <script src="js/org/jsweet/examples/canvas/Canvas.js"/>
4   </head>
5   <body>
6     <canvas id="canvas" style="position: absolute; border:
          solid black 1px"></canvas>
7   </body>
8 </html>
```

The JSweet code is shown hereafter. It defines a `Canvas` class with a `main` method that creates an instance of that class. The instance simply accesses the canvas element through the DOM API (`jsweet.dom`) and creates a 2D rendering context (line 24). It then calls the `draw()` function (line 25) that actually performs one animation step and requests the next one by using the `Window.requestAnimationFrame(...)` function (line 38). It is the state of the `Canvas` object that defines the animation state with the `angle` field (line 15), which goes from `0` to `Math.PI*2` (line 36). Each animation step, that is to say each time the `draw()` method is called within an animation frame, the angle is incremented by `0.05` (line 37).

```
1  package org.jsweet.examples.canvas;
2
3  import static jsweet.dom.Globals.console;
4  import static jsweet.dom.Globals.document;
5  import static jsweet.dom.Globals.window;
6  import jsweet.dom.CanvasRenderingContext2D;
7  import jsweet.dom.HTMLCanvasElement;
8  import jsweet.lang.Math;
9  import jsweet.util.StringTypes;
10
11 public class Canvas {
12
13   private HTMLCanvasElement canvas;
14   private CanvasRenderingContext2D ctx;
15   private double angle = 0;
16
17   public static void main(String[] args) {
18     new Canvas();
19   }
20
21   public Canvas() {
22     console.info("creating canvas drawing example");
23     canvas = (HTMLCanvasElement)
           document.getElementById("canvas");
24     ctx = canvas.getContext(StringTypes._2d);
25     draw();
26   }
27
28   private void draw() {
29     ctx.fillStyle = "red";
```

```
30      ctx.clearRect(0, 0, canvas.width, canvas.height);
31      ctx.beginPath();
32      ctx.moveTo(canvas.width / 2, canvas.height / 2);
33      ctx.lineTo(canvas.width, canvas.height / 2);
34      ctx.arc(canvas.width / 2, canvas.height / 2, canvas.width /
            2, 0, angle);
35      ctx.fill();
36      if (angle < Math.PI * 2) {
37        angle += 0.05;
38        window.requestAnimationFrame((time) -> {
39          this.draw();
40        });
41      }
42    }
43  }
```

Transpilation step #1 (Java → TypeScript):

```
module org.jsweet.examples.canvas {
   export class Canvas {
      private canvas: HTMLCanvasElement;

      private ctx: CanvasRenderingContext2D;

      private angle: number = 0;

      public static main(args: string[]) {
         new Canvas();
      }

      public constructor() {
         console.info("creating canvas drawing example");
         this.canvas =
             <HTMLCanvasElement>document.getElementById("canvas");
         this.ctx = this.canvas.getContext("2d");
         this.draw();
      }

      private draw() {
         this.ctx.fillStyle = "red";
         this.ctx.clearRect(0, 0, this.canvas.width,
             this.canvas.height);
         this.ctx.beginPath();
         this.ctx.moveTo(this.canvas.width / 2,
             this.canvas.height / 2);
         this.ctx.lineTo(this.canvas.width, this.canvas.height
             / 2);
         this.ctx.arc(this.canvas.width / 2,
             this.canvas.height / 2, this.canvas.width / 2, 0,
             this.angle);
         this.ctx.fill();
         if((this.angle < Math.PI * 2)) {
            this.angle+=0.05;
            window.requestAnimationFrame((time) => {
```

```
                this.draw();
            });
        }
    }
}
org.jsweet.examples.canvas.Canvas.main(null);
```

Transpilation step #2 (TypeScript → JavaScript with *tsc*):

```
var org;
(function (org) {
    var jsweet;
    (function (jsweet) {
        var examples;
        (function (examples) {
            var canvas;
            (function (canvas) {
                var Canvas = (function () {
                    function Canvas() {
                        this.angle = 0;
                        console.info("creating canvas drawing
                            example");
                        this.canvas =
                            document.getElementById("canvas");
                        this.ctx = this.canvas.getContext("2d");
                        this.draw();
                    }
                    Canvas.main = function (args) {
                        new Canvas();
                    };
                    Canvas.prototype.draw = function () {
                        var _this = this;
                        this.ctx.fillStyle = "red";
                        this.ctx.clearRect(0, 0, this.canvas.width,
                            this.canvas.height);
                        this.ctx.beginPath();
                        this.ctx.moveTo(this.canvas.width / 2,
                            this.canvas.height / 2);
                        this.ctx.lineTo(this.canvas.width,
                            this.canvas.height / 2);
                        this.ctx.arc(this.canvas.width / 2,
                            this.canvas.height / 2, this.canvas.width
                            / 2, 0, this.angle);
                        this.ctx.fill();
                        if ((this.angle < Math.PI * 2)) {
                            this.angle += 0.05;
                            window.requestAnimationFrame(function
                                (time) {
                                _this.draw();
                            });
                        }
                    };
                    return Canvas;
```

```
            })();
            canvas.Canvas = Canvas;
        })(canvas = examples.canvas || (examples.canvas =
            {}));
    })(examples = jsweet.examples || (jsweet.examples = {}));
})(jsweet = org.jsweet || (org.jsweet = {}));
})(org || (org = {}));
org.jsweet.examples.canvas.Canvas.main(null);
//# sourceMappingURL=Canvas.js.map
```

3.3 More examples...

The JSweet website [13] shows more complex and elaborated online examples, which can also be downloaded from Github. In particular, more complex versions of the examples above are available, as well as other examples to demonstrate the wide-range capabilities of JSweet. Theses examples include:

- A small AngularJS application for managing invitations to events. It includes a few controllers, and views for the UI.
- A small block breaking game, programmed as an animated multiple-layer canvas, and that uses (CSS) transforms. This game works well in iOS and Android Web containers, thus demonstrating the ability of JSweet to program HTML games for mobile devices.
- A promises example that shows in the Web container some progress bars for asynchronous concurrent tasks.
- A ray tracer example, which is a JSweet implementation of the ray tracer example found on the TypeScript website [10]. It draws a nice 3D scene on a canvas.
- A small application that uses the well-known Backbone, JQuery and Underscore libraries to implement a local-storage-based persistent todo list. It is a JSweet version of the TodoMVC example that can be found on the TypeScript website [10].

4 Conclusion (and a few words on performance)

In this paper, we have presented and discussed JSweet, a Java to JavaScript transpiler, which relies on TypeScript to provide a production-ready Java environment for developing JavaScript applications. The main strength of JSweet is that it brings to Java the best of TypeScript. First, it maps to Java most of the constructs and types that are introduced in TypeScript [11], and that are used to program well-typed JavaScript. Second it translates the DefinitelyTyped [18] APIs available in the TSD [16] repository to Java, so that all the frameworks and libraries out there become immediately available to JSweet programmers.

From a performance standpoint, the Java to TypeScript transpiler part is fast, since it is a syntax mapper as explained Section 1.2. It uses `javac` to build the Java program AST and uses a tree scanner to output the code. One pass on each compilation unit (file) is necessary to create the TypeScript code and save it into a TypeScript file. So,

JSweet induces a tiny overhead to `javac` since it performs some local analysis to generate valid TypeScript code. Some aspects of this analysis may be improved, for instance the lookup of fields and methods in the class hierarchy, which is necessary to avoid duplicates (which are not allowed in TypeScript), but also to generate the `this` prefix in member accesses, which are mandatory in TypeScript, but not in Java. This point may become a bottleneck in the future, but right now, it is not a high-priority optimization.

On the contrary, the TypeScript compiler (`tsc`), which is used to generate the final JavaScript code is actually much slower when launched from the command line than the `javac`-driven Java to TypeScript transpiler. To overcome this issue that is particularly problematic within the JSweet Eclipse plugin, in future releases, we will work on using TypeScript with `node.js`.

If not done yet, we strongly encourage Java programmers (but even JavaScript ones) to install JSweet from `http://www.jsweet.org` to draw their own conclusions on JSweet usability and performance. For programmers who are interested in going further, they can read the paper on the JSweet to TypeScript comparison [15], which give more advanced details on the way JSweet ensures safer coding through typing and semantics.

About the author

Renaud Pawlak is an R&D consultant and holds a Ph.D in Software Engineering. He got recognized by the academic community through his work on Aspect-Oriented Programming, of which he was one of the pioneers. He is the is the co-author of the book "Foundations of AOP for J2EE Development", published by Apress, and the co-creator of the Open Source AOPAlliance API, which is used by hundreds Java projects and libraries (source: Maven Central). He is also the initiator of the Spoon Java transpiler project at INRIA, and the co-founder and former CTO of IDCapture, a company creating mobile and WEB applications for the construction business.

Acknowledgments

We would like to thank the EASYTRUST™ company for being an early partner and user of the JSweet technology. EASYTRUST™ is a French Software Editor specialized in Asset Management Solutions and one of Oracle's partners. They have helped in designing an testing the JSweet transpiler and APIs.

References

[1] Joshua Barnett. Building a cross-platform mobile game with HTML5. `http://www.caffeinatednightmare.com/mindflip/`.

[2] Bear Bibeault and Yehuda Kats. *jQuery in Action*. Dreamtech Press, 2008.

[3] Shigeru Chiba. Javassist-a reflection-based programming wizard for java. In *Proceedings of OOPSLA98 Workshop on Reflective Programming in C++ and Java*, page 174, 1998.

[4] Alexandru Craciun. ST-JS : Strongly Typed JavaScript. Borrowing Java's syntax to write type-safe JavaScript. http://st-js.github.io/.

[5] Robert Hanson and Adam Tacy. *GWT in action: easy Ajax with the Google Web toolkit*. Dreamtech Press, 2007.

[6] Peter Klima and Stephan Selinger. *Towards Platform Independence of Mobile Applications*. Springer, 2013.

[7] Kyriakos-Ioannis D Kyriakou, Ioannis K Chaniotis, and Nikolaos D Tselikas. The gpm meta-transcompiler: Harmonizing javascript-oriented web development with the upcoming ecmascript 6 harmony specification. In *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*, pages 176–181. IEEE, 2015.

[8] Florian Loitsch and Manuel Serrano. Hop client-side compilation. *Trends in Functional Programming*, 8:141–158, 2007.

[9] Alex MacCaw. *The Little Book on CoffeeScript*. O'Reilly Media, Inc., 2012.

[10] Microsoft Corporation. TypeScript website. http://www.typescriptlang.org/.

[11] Microsoft Corporation. Typescript, language specification, version 1.5. http://www.typescriptlang.org, 2015.

[12] Mozilla. Mozilla Developer Network. https://developer.mozilla.org.

[13] Renaud Pawlak. JSweet: programming JavaScript applications with Java. http://www.jsweet.org.

[14] Renaud Pawlak, Martin Montperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software Practice and Experiments*, 2015.

[15] Renaud Pawlak. Jsweet typing and semantics: a comparison to typescript. http://www.jsweet.org, 2015.

[16] Bart van der Schoor and Diullei Gomes. TSD: TypeScript Definition manager for DefinitelyTyped. http://definitelytyped.org/tsd/.

[17] Kathy Walrath and Seth Ladd. *What is Dart?* O'Reilly Media, 1st edition, 2012.

[18] Boris Yankov. DefinitelyTyped: The repository for high quality TypeScript type definitions. https://github.com/borisyankov/DefinitelyTyped.

[19] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 301–312. ACM, 2011.