JSweet: how does it compare to TypeScript?

A focus on typing and semantics

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

Abstract

JSweet is a Java to JavaScript transpiler, which relies on TypeScript for typing and JavaScript code generation. In this position paper, we discuss the JSweet typing and semantics by comparing them to TypeScript ones. We explain how the JSweet transpiler tweaks the Java language to support most features introduced by TypeScript, including typing features such as *string types*, *union types*, *functional types*, and *object types*. Also, we show how the transpiler generates JavaScript code that aims at mimicking the Java semantics, thus fixing some TypeScript semantic issues inherited from JavaScript. Finally, we highlight some strengths and weaknesses of JSweet compared to TypeScript, and we explain that besides some limitations, JSweet can be seen as a good alternative to TypeScript. It ensures a level of type safety similar to TypeScript, and adds the rigorous Java syntax and semantics to it, thus making JavaScript programming safer and more intuitive to Java programmers.

1 Introduction

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. JSweet relies on TypeScript [4] and bring the best of it to Java programmers.

In this position paper, we discuss how JSweet and TypeScript share common constructs, but we also discuss their fundamental differences. The goal of this discussion is to understand better the capabilities of JSweet, but also its limitations. Hence, programmers know better what to expect when programming applications with JSweet. We assume that the readers are familiar with concepts such as typing and semantics. Programmers interested in "just" using JSweet should probably skip this article and read the documentation available on JSweet's website [5]. In any case, we recommend that you read first the position paper [6], which presents the JSweet transpiler with a more general perspective.

Our discussion is organized as follows. Section 2 describes the JSweet design and how it closely relates to TypeScript. It points out the cross-validation mechanism that ensures that JSweet APIs and generated code are as correct as the TypeScript ones. In Section 3, we discuss the commonalities and the differences in terms of typing. We explain how JSweet uses auxiliary types to map all TypeScript typing constructs, including the more complex ones, such as union types. In Section 4, we highlight some semantic differences, which are implemented by the transpiler so that the programs behaviors match the expected behavior of Java programs. That way, Java programmers

can rely on their knowledge of Java and will not be confused with runtime differences between Java and JSweet. Finally, Section 5 studies other differences, such as optional parameters and function overloading.

2 Overview

The most important point to understand about JSweet, is that it transpiles from Java to JavaScript by going trough TypeScript, which allows a double verification of the program: one occurring in Java, with the Java type checker, and another one occurring in TypeScript, with the *tsc* transpiler. This two-phase verification process ensures that the JSweet-enabled Java application is rigorously equivalent to the generated TypeScript one, thus ensuring safer programs.

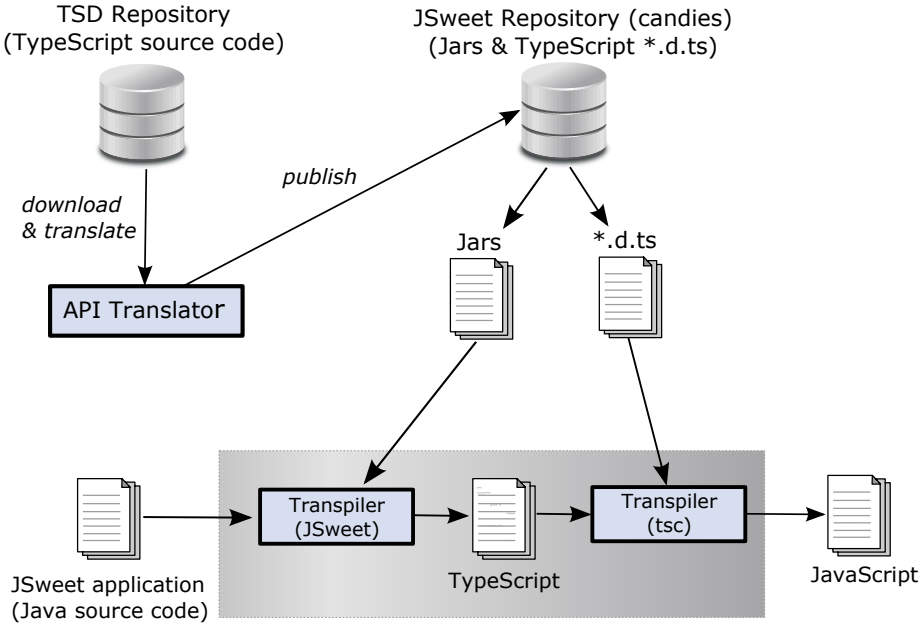

Figure 1: JSweet design

Figure 1 shows the JSweet design and explains how JSweet implements this double verification.

1. Each JavaScript API is translated from the TSD [7] repository, which contains hundreds of JavaScript libraries, plugins, frameworks, and components. The corresponding Java API is packaged in a Jar file along with the original `*.d.ts` files and deployed as a Maven artifact in the JSweet candies repository (for instance, jQuery group id is `org.jsweet.candies`, and its artifact id is `jquery`).

2. When JSweet transpiles from Java to TypeScript (bottom-left-hand side of the figure), it uses the generated Java API from the candies, and when it transpiles to JavaScript (bottom-right-hand side of the figure), it uses the original TypeScript definition files.

We have chosen this design because it cross-validates the source code transpilation (Java → TypeScript) and the API definition files translation (TypeScript → Java). On one hand, it ensures that the transpiled TypeScript code is correct with regard to the APIs (otherwise, the `tsc` transpilation would fail). On the other hand, it ensures that the generated APIs from the `*.d.ts` is fully equivalent to the original definitions (otherwise, the `tsc` transpilation, which uses the original APIs, would also fail).

On a language perspective, JSweet ensures that the API definition remain equivalent in terms of typing by using as much as possible of the Java typing. For instance, the following snippet of a TypeScript definition file...

```
// Original TypeScript definition (from jquery.d.ts):
click(handler: (eventObject: JQueryEventObject) => any):
    JQuery;
```

... translates to an equivalent definition in Java:

```
// Java definition translated by the API translator:
import java.util.function.Function;
[...]
native public JQuery click(Function<JQueryEventObject,Object>
    handler);
```

Finally, since Java and TypeScript remain similar languages, the transpilation between the two can be done quite efficiently. In theory, we "just" need to map the Java syntax to the TypeScript one. For instance, the simple jQuery program hereafter...

```
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;
    });
  }
}
```

... will transpile to a very similar TypeScript program:

```
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);
```

In practice however, this transpilation technique is not as simple as it may seem, since many subtle (and less subtle) differences exist between Java and TypeScript. These differences are interesting to be understood to be fully aware of JSweet capabilities and limitations with regard to TypeScript. In the remainder of this paper, we focus on explaining how JSweet maps to TypeScript and what are the main differences between the two languages.

3 Typing

Mapping TypeScript types to Java types poses several problems. The first problem, which is discussed in Section 3.1, is that core Java types, including primitive types such as numbers, boolean, strings, and even raw objects need to be mapped to the core JavaScript types. However, the Java API differs from the JavaScript one and we need a way to bridge the two and to ensure that the programmers will not misuse the API by using Java concepts that are not allowed in JavaScript. The second difficulty to be overcome is that TypeScript defines a very elaborated type system with various typing constructs that don't exist as is in Java. From Section 3.2, we show how the Java type system can be used to implement TypeScript typing constructs through the use of auxiliary types.

3.1 Core types and objects

In JSweet, core JavaScript objects (`Object`, `Array`, `Number`, `String`, `Error`, and so on) are defined in the `jsweet.lang` package. However, when using literals in Java programs, these values are instances of `java.lang` classes. So, in order to be able to manipulate literals and objects easily in JSweet programs, the JSweet API uses `java.lang` classes for the following types: `Object`, `String` and `Number` (it actually uses the `double` primitive type most of the time). Using Java types here is important since it allows the programmers to pass Java literals, for instance a string literal to the JQuery `$` function:

```
$("p").addClickListener(...)
```

However, in some cases, the programmers may want to access the JavaScript API (defined in `jsweet.lang`) on a core type, but will not be able to get immediate access to it since the API returns a Java type. For instance, if programmers want to use

the JavaScript function `replace(regexp, function)` on a Java string returned by an API, they would need to cast it to a JSweet string (`jsweet.lang.String`). The `jsweet.util.Globals` class defines convenient static methods to cast back and forth core Java objects to their corresponding JSweet objects. For instance the `string(...)` method will allow the programmer to switch from the Java to the JSweet strings and conversely.

These casts can be legitimately quite annoying to the programmers and lead to less readable code. To compensate this drawback, JSweet allows the use of the Java API when the methods have the same prototypes as the JavaScript ones. For instance, `String.indexOf`, `String.lastIndexOf`, `Object.toString`, `String.toLowerCase()`, and `String.toUpperCase()` are shared by both Java and JavaScript APIs, and are directly usable on JSweet and Java core objects.

Here is a JSweet example of a function that transforms a string with spaces to a title cased one. Note that the `str` and `tok` variables are API parameters and, as such, they are declared as `java.lang.String`. This implies the use of the `string(...)` cast function at lines 2, 5 and 6, to access the `replace`, `charAt` and `substr` JavaScript functions, which are not available in the Java API (note that `charAt` exists in Java but returns a primitive `char` type, which is not seen as a string).

```
1 public static String toTitleCase(String str) {
2   return string(str.toLowerCase()).replace(
3     new RegExp("\\w\\S*", "g"),
4     (tok, i) -> {
5       return string(tok).charAt(0).toUpperCase()
6         + string(tok).substr(1).toLowerCase();
7     });
8 }
```

In practice the use of such cast functions is limited to specific use cases. When casting is required, the Java compiler and the JSeet transpiler will always ensure type safety and report sound errors to avoid API misuses.

3.2 Mapping TypeScript enhanced typing features to Java

The TypeScript type system is richer than the Java one, because it is a response to the urge of typing JavaScript programs, which are inherently complex and dynamically typed. Besides core types, TypeScript allows various advanced type kinds that are not directly supported by Java:

- **Functional types**: functional types allow the programmer to say that they expect lambda objects with a given signature (parameter types and return type, written as `(p1:T1, ...)=>R`).
- **Object types**: an object type allows the programmer to say that objects with some properties and methods are expected. When using an object type, the name and the actual type of the object does not matter, what matters is if the object type defines the required properties and methods. For instance, an object type `{ length: number; }` will match all the types (classes and interfaces) that define the `length` property. In a way, object types can be seen as inlined anonymous type definitions, but they are actually some sort of constraint, which makes them extremely flexible.

- **String types**: a string type can be seen as string overload with a specific string value. For instance, `createElement(tag:string):HTMLElement` can be overloaded with `createElement(tag:"span"):HTMLSpanElement`, which implies that using the `"span"` string value as a parameter will return an `HTMLSpanElement` rather than an `HTMLElement`.
- **Tuple types**: they represent JavaScript array with individually tracked element types. For instance: `var t: [number, string] = [3, "three"];`
- **Union types**: a union type allows the programmer to say that several (potentially incompatible) types are allowed for a given object. For instance, `string | number` means that the object can be typed as a `string` or as a `number`. Note that a union type will not create a new type that would be the merge of the union-ed types, but it will ensure that the programmer actually uses one or the other of the union-ed types.

To support equivalent features in Java, JSweet uses *auxiliary types*. The idea behind auxiliary types is to create classes or interfaces that can hold the typing information through the use of type parameters (a.k.a *generics*), so that the JSweet transpiler can ensure type safety similarly to TypeScript. These interfaces can most of the time be automatically created by the API translator, or pre-created in the core JSweet API.

- For functional types, JSweet reuses the `java.Runnable` and `java.util.function` functional interfaces of Java 8. These interfaces only support up to 2-parameter functions. Thus, JSweet adds some support for more parameters in `jsweet.util.function`, since it is a common case in JavaScript APIs.
- For object types, the use of auxiliary classes (annotated with `@Interface` and `@ObjectType`) is mandatory. The API translator automatically generates such classes when object types are encountered in the TypeScript definitions. But the programmers must define them manually when wanting to use such types in their programs.
- For string types, the use of specific types and final instances of these types is the way to simulate string types in Java. For instance, if a `"span"` string type needs to be defined, a Java interface called `span` and a static final field called `span` in a `StringTypes` class will do the job.
- For tuple types, JSweet defines parameterized auxiliary classes `TupleN<T0, ... TN-1>`, which define `$0`, `$1`, ... `$N-1` public fields to simulate typed array accessed (field `$i` is typed with `Ti`).
- For union types, JSweet takes advantage of the `method overloading` mechanism available in Java [1]. For more general cases, it defines an auxiliary interface `Union<T1, T2>` (and `UnionN<T1, ... TN>`) in the `jsweet.util.union` package. By using this auxiliary type and a `union` utility method, programmer can cast back and forth between union types and union-ed type, so that JSweet can ensure similar properties as TypeScript union types.

In most cases, auxiliary types allow Java to perform type checks through generics, like it is the case for example for functional interfaces or for tuples. However, in some cases such as unions, JSweet implements extra checks to ensure the same typing constraints as the TypeScript ones.

3.3 Using auxiliary types: the union types case study

To illustrate the use of auxiliary types, we now focus on union types. The following code shows a typical use of union types in JSweet. It simply declares a variable as a union between a string and a number, which means that the variable can actually be of one of that types (but of no other types). The switch from a union type to a regular type is done through the `jsweet.util.Globals.union` helper method. This helper method is completely untyped, allowing from a Java perspective any union to be transformed to another type. It is actually the JSweet transpiler that checks that the union type is consistently used.

```
import static jsweet.util.Globals.union;
import jsweet.util.union.Union;
[...]
Union<String, Number> u = ...;
// u can be used as a String
String s = union(u);
// or a number
Number n = union(u);
// but nothing else
Date d = union(u); // JSweet error
```

The `union` helper can also be used the other way, to switch from a regular type back to a union type, when expected.

```
import static jsweet.util.Globals.union;
import jsweet.util.union.Union3;
[...]
public void m(Union3<String, Number, Date>> u) { ... }
[...]
// u can be a String, a Number or a Date
m(union("a string"));
// but nothing else
m(union(new RegExp(".*"))); // JSweet error
```

Thus, a simple set of auxiliary types can quite easily simulate the union types in TypeScript. Of course, this trick is not as concise and intuitive as the TypeScript union syntax. However, it is important to understand that in APIs, the use of TypeScript union types is replaced/complemented with *function overloading*, as it is the usual way in Java [1]. This makes it much simpler to use by the programmers, who do not need to know about union types anymore.

```
// general prototype (with union types)
native public void m(Union3<String, Number, Date>> u);
// actually generated API, using overloading
native public void m(String s);
native public void m(Number n);
native public void m(Date d);
```

Java overloading is a quite fundamental difference with TypeScript, and will be discussed further in Section 5.2, when talking about optional parameters.

4 Semantics

Semantics designate how a given program behaves when executed. It relies on an execution model and how each syntactic program element impacts that execution model in terms of side effects (for instance where and how variable values are accessed and modified). Although JSweet relies on the Java syntax, programs are transpiled to JavaScript and do not run in a JRE. As a consequence, the JavaScript semantics will impact the final semantics of a JSweet program compared to a Java program. In this section, we discuss some core semantic-related differences and explain how JSweet makes sure that the final semantics remain close to the Java one, so that Java developers are not confused with behavioral differences and feel safe when writing JSweet programs.

4.1 Conflict issues inherited from JavaScript

TypeScript inherits from JavaScript the way variables are handled. Thus, in TypeScript, variables are globally accessible and can hold any kind of objects, including the language entities, such as functions and modules. On contrary to Java, where variables, functions and packages are totally different concepts, JavaScript defines everything as a variable. Therefore, naming a module or a function after the name of an existing variable may cause unexpected execution behaviors, that will not be statically detected by the compiler. For instance, misspelling a variable may cause the program to bind to an existing global variable in the execution scope, which can be a function or a module, thus leading to potentially hard to debug programs.

JSweet, ensures closer match to the Java semantics. Firstly, JSweet removes the definition of global variables and functions, simply because Java does not allow them. Following the Java rules, globally scoped variables and functions must be `public static` members of a class. In JSweet, it is possible to simulate global variables and functions by placing them in a class called `Globals`, which by convention will be removed by the transpiler and lead to JavaScript global variables. In any case, when using such a variable, they need to be statically imported, and the compiler will complain with an ambiguous access error when several members of the same name are statically imported, therefore reducing the risks of confusion for the programmers, and leading to better structured and modular programs. Secondly, since JSweet enforces typing through Java, it is not possible to confuse variables and functions (fields and methods are different concepts), nor to confuse variables and modules (fields and packages are also different concepts). The JSweet transpiler makes sure that any name clash problem is detected statically.

For the sake of illustrating what was just explained, let us take here a very common JavaScript/TypeScript programming mistake, which consists of forgetting the parenthesis when calling a function. For instance, when drawing on a canvas, you can start a path with a call to *ctx.beginPath()*. If you forget the parenthesis, nor the completion (IDE's coding assistant) nor the compilation will give any clue that something is wrong. Even worse, it seems to work fine at runtime and does not crash. Indeed, *ctx.beginPath* just accesses the lambda and does nothing with it. This is clearly a semantics issue due to a lack of enforcement of semantic constraints at the compiler level. In JSweet, such statements are not permitted (and not even proposed by the completion), first because the Java compiler does not allow variable access statements (they are expressions and must be used in other expressions or statements), second because functions are not variables and cannot be confused with them. This strict Java behavior is at the end of the day quite comforting for most programmers.

4.2 Variable scoping in lambda expressions

In our opinion, variable scoping is one of the main weakness (not to say flaw) of JavaScript. Variables and functions are accessed by name, and any variable in the execution scope that matches the name will be returned. This leads to the global variable issue discussed in Section 4.1, but it also leads to a more complicated problem, which is well-known to advanced JavaScript programmers, but that can give headaches to Java programmers. Consider for instance the following TypeScript code:

```
1 var nodes : NodeList = document.querySelectorAll(".control");
2 for (var i : number = 0; i < nodes.length; i++) {
3   var element : HTMLElement = <HTMLElement> nodes[i];
4   element.addEventListener("keyup", (evt) => {
5     element.classList.add("hit");
6   });
7 }
```

At a first glance, most programmers would probably think that this piece of code installs a `keyup` event handler on each HTML element with the `control` CSS class, so that when a key is pressed, the `hit` class is added to the element. However, this does not work as expected! Indeed, the `element` variable accessed in the handler at line 5 does not bind to the value of `element` variable initialized line 3 while looping along the nodes. In fact, when the `keyup` event is triggered, the handler's execution scope is not the same as it was in the loop when the event handler was installed. As a consequence, JavaScript looks up for the variable called "`element`" in the execution scope and uses the first it finds. In that case, since the loop is over when the event listener is executed, the value of `element` that will be found in the scope is the last value taken by the `element` variable: the last element of the node list. It will be the case for all the installed event listeners – note that it can even get worse if the `element` variable is global and that the program changes the value of the element variable after the loop.

The code hereafter is the JSweet version of the previously discussed example and how it transpiles to TypeScript.

```
NodeList nodes = document.querySelectorAll(".control");
for (int i = 0; i < nodes.length; i++) {
  HTMLElement element = (HTMLElement) nodes.$get(i); // final
  element.addEventListener("keyup", (evt) -> {
    element.classList.add("hit");
  });
}
```

Transpiles to:

```
1 var nodes: NodeList = document.querySelectorAll(".control");
2 for(var i: number = 0; i < nodes.length; i++) {
3   var element: HTMLElement = <HTMLElement>nodes[i];
4   element.addEventListener("keyup", ((element) => {
5     return (evt) => {
6       element.classList.add("hit");
7   }})(element));
8 }
```

Note the difference with the TypeScript code we started from at the beginning of this section. In that new code, the JSweet transpiler ensures that the `element` variable is not subject to side effects in the lambda. To do so, the transpiled code transforms the code structure to ensure that the used variables are passed as parameters within an independent scope (wrapping lambda at lines 4-7). As a consequence, the semantics is the same as for a Java program, where variables accessed from a lambda must actually be `final` variables (but the `final` keyword is optional since Java 8), which is much more intuitive to Java programmers, but also for most JavaScript programmers...

4.3 Loosing *this*

Another side effect to that scoping issue, is that it is quite easy to trick the compiler and get some unexpected value for `this` in an object. The following simple TypeScript program shows such a problem:

```
1  module example {
2     class Example {
3        private i: number = 8;
4        public getAction(): () => void {
5           return this.action;
6        }
7        public action(): void {
8           console.log(this.i);
9        }
10    }
11    var instance: Example = new Example();
12    instance.getAction()();
13 }
```

Most programmers, when reading this code would conclude that it should output "`8`" in the console. However, when calling the action at line 12, `this` is actually worth `window`, which is the default value of `this` in the global scope of a Web browser. Indeed, by invoking the `action` lambda as a function, we are not in the scope of the `Example` instance anymore. This does not need to be seen as a bug and it is actually perfectly understandable to programmers who are used to functional programming, but it is quite hard to admit for Java programmers who understand the world as interacting objects and where it is expected that `this` should always be `this`, and not some random object. Whatever your standpoint is, we believe in that case that it is not acceptable that the TypeScript compiler does not report any error or warnings. At runtime, the program will indeed log `undefined`, because the `i` variable is not defined on the `window` instance!

The following code is the JSweet version of the same program. Note the use of the `Runnable` functional interface at line 6 and of the function reference at line 7. Functional programming lovers will dislike the lambda invocation at line 13, through the `getAction().run()` method, however, some Java programmers will find it clearer than the functional version that is written `getAction()()`.

```
1  package example;
2  import static jsweet.dom.Globals.console;
3
4  public class Example {
5    private int i = 8;
```

```
  public Runnable getAction() {
    return this::action;
  }
  public void action() {
    console.log(this.i); // this.i is 8
  }
  public static void main(String[] args) {
    Example instance = new Example();
    instance.getAction().run();
  }
}
```

This JSweet code transpiles to:

```
module example {
   export class Example {
      private i: number = 8;
      public getAction() : () => void {
         return () => { return this.action() };
      }
      public action() {
         console.log(this.i);
      }
      public static main(args: string[]) {
         var instance: Example = new Example();
         instance.getAction()();
      }
   }
}
example.Example.main(null);
```

The fundamental behavioral difference between this output and the original TypeScript example is quite subtle and probably does not catch your eye at first. It stands at line 6, through the way this code accesses the reference to the `action` method, that is to say through a wrapping lambda that scopes the `this` variable and makes it immutable. At the end of the day, passed the syntactic differences, the most important to us is that when running this program compiled by JSweet, it behaves as expected: it outputs "`8`" in the console.

5 Other language constructs

In the previous sections, we have just seen some important difference between JSweet and TypeScript, from a typing and semantics perspective. Many other differences can be discussed. This section intends to discuss some of them.

5.1 Indexed objects

One nice syntactic feature of TypeScript, it that is allows the definition of an indexed function on objects. This is useful when an object (besides arrays) is supposed to be a container of object. For instance, the `NodeList` type (from the DOM) defines an indexed function:

```
1 interface NodeList {
2    length: number;
3    item(index: number): Node;
4    [index: number]: Node;
5 }
```

The indexed function is defined at line 4. It means that when having a `NodeList` instance, one can access one of the nodes by using the square bracket syntax, like if it was an array, as shown here:

```
var nodeList : NodeList = ...
// get the last node
var node : Node = nodeList[nodeList.length-1];
```

In Java, it is not possible to use the square bracket syntax on arbitrary objects, unless dealing with an array type such as `Node[]` (which is allowed in JSweet and transpiles to a JavaScript array). So in JSweet, `$get` and `$set` naming conventions are used to simulate indexed functions on objects, which gives the following code:

```
var nodeList : NodeList = ...
// get the last node
var node : Node = nodeList.$get(nodeList.length-1);
```

To compensate this Java-inherited syntax limitation, JSweet supports the `Iterable` interface on indexed objects, so that the programmers can use *foreach* loops and avoid direct accesses to the indexed elements. Note that the JSweet API translator automatically detects the indexed function and makes the generated classes iterable. So, iterating on an indexed object in JSweet can be implemented as shown below.

```
NodeList nodes = ...
for (int i = 0; i < nodes.length; i++) {
  HTMLElement element = (HTMLElement) nodes.$get(i);
  [...]
}
// same as:
NodeList nodes = ...
for (Node node : nodes) {
  HTMLElement element = (HTMLElement) node;
  [...]
}
```

5.2 Optional parameters and fields

In JavaScript/TypeScript parameters can be optional, in the sense that a parameter value does not need to be provided when calling a function. In TypeScript, such an optional parameter is followed with a question mark (and potential an assignment when having a default value). This syntax is very concise and straightforward and does not exist in Java. The reason why it does exist is quite simple: there is another, more flexible way, to achieve the same effect in Java, through the use of *function overloading* [1]. Function overloading consists in allowing several functions to have the same name as

long as they have different prototypes. Using function overloading in Java, it is easy to implement optional parameters:

```
// TypeScript version:
m(s:string, n?:number):string;
// Java version:
native public String m(String s, double n);
native public String m(String s);
```

Typically, when implementing optional parameters with overloading, the `m(String s)` calls the `m(String s, double n)` with the `n` default value. Thus, JSweet allows this kind of (restrictive) overloading and transpiles it to optional parameters. As a consequence, JSweet allows optional parameters through overloading, which is syntactically speaking not as efficient as the question mark notation. Also, JSweet restricts Java overloading to the ones that can be implemented as optional parameters. For all the other cases of overloading, JSweet will report an error, which is clearly a loss compared to Java.

```
String m(String s, double n) { return s+n; }
// valid overloading (JSweet transpiles to optional parameter)
String m(String s) { return m(s, 0); }
// invalid overloading (JSweet error)
String m(String s) { return s; }
```

Optional parameters should not be confused with optional fields in interfaces. Although they share the same syntax, optional fields are used to report errors when the programmer forgets to initialize a mandatory field in an object. Supporting optional fields in JSweet is more straightforward than optional parameters and is done through the use of `@Optional` annotations. For instance:

```
@Interface
public class Point {
  public double x;
  public double y;
  @Optional
  public double z = 0;
}
```

It is the JSweet compiler that will check that the fields are correctly initialized, when constructing an object.

```
// no errors (z is optional)
Point p1 = new Point() {{ x=1; y=1; }};
// JSweet reports an error since y is not optional
Point p2 = new Point() {{ x=1; z=1; }};
```

Of course, compared to TypeScript, the syntax for defining optional fields and instantiating an object in JSweet is less concise (note the use of the instance initializer blocks right above). However, it uses standard Java and ensures the expected properties in terms of type safety and constraints, which is a good thing once the programmers get used to the syntax.

5.3 Other differences

Explaining all the differences in details would be out of the scope of this paper. To wrap up this comparison, we can however speak about other small interesting differences.

A first difference we can mention is the way enums are handled. In TypeScript, enums are implemented as a double entry map, which is not really intuitive to programmers. In JSweet, the Java Enum API is partially supported to ensure more readable code. A second difference is that the *varargs* are not correctly supported in TypeScript. Typically, when a function receives a *varargs* parameter, it cannot pass it to another function that takes a varargs parameter, so that *varargs* cannot be passed along. JSweet fixes that problem and follows the Java semantics of *varargs*, so that programmers can use them with confidence.

6 Conclusion

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 [4], and that are used to program well-typed JavaScript. Second it translates the DefinitelyTyped [8] APIs available in the TSD [7] repository to Java, so that all the frameworks and libraries out there become immediately available to JSweet programmers.

JSweet programs are globally less concise than TypeScript ones, due to the Java Syntax, which is not the one of a functional language. Also, from a syntax point of view, it generally less straightforward and intuitive (naming conventions, annotations, auxiliary types). However, in most common cases, programming in JSweet is as easy as with TypeScript, even easier in some cases, since it removes JavaScript-inherited-syntax and semantics that are really counter-intuitive to Java programmers. To summarize, here is a enumeration of JSweet main weaknesses and strengths that were discussed in this paper.

- **Weaknesses**
  - Some core API on `Object`, `String`, and `Number`, require the programmer to cast back and forth between `java.lang` and `jsweet.lang` objects. Core objects and how to use them is explained in section 3.1.
  - When advanced typing is required, TypeScript proposes built-in features, but JSweet leans on auxiliary types, which are less concise and less intuitive for the programmers. Auxiliary types are discussed in Section 3.2.
  - JSweet must rely on naming conventions for some language feature, such as `$get` and `$set` for indexed objects (see section 5.1), and the `Globals` classes for defining global variables and functions. The use of annotations and other Java specific elements can also be less straightforward to some programmers (see for example Section 5.2).
- **Strengths**
  - There is no confusion between variables and functions, variables and modules, as explained in Section 4.1.

- Variables in lambda expressions are made final so that there is no scope issue. Like in Java, side effects to variables that are used from within a lambda expression are reported as errors by the compiler (see Section 4.2).
- In TypeScript, it is possible to confuse the compiler about the value of `this`. In JSweet like in Java, `this` is always `this`, thus ensuring clean and intuitive Object-Oriented Programming. An example of such an issue is explained in Section 4.3.
- Varargs can be passed along (section 5.3).
- Enums have a clean API (section 5.3).

Finally, what JSweet brings to Web development and Web languages is a safer and more reliable semantics that is closer to Java and less surprising than the JavaScript (and TypeScript) ones. When programming within Eclipse, most Java programmers will feel safer and "at home", even when programming complex applications and using the latest Web frameworks such as Angular and Polymer. We believe that JSweet is a serious alternative to dropping Java or using dedicated Java-driven frameworks such as GWT [3] (or derivatives such as Vaadin [2]) when creating Web applications. Finally, using JSweet narrows the gap between the Java server and client side, so that programmers can use the same language for the UI and share models between the server and the client.

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.

References

[1] Joseph Yossi Gil and Keren Lenz. The use of overloading in Java programs. In *ECOOP 2010–Object-Oriented Programming*, pages 529–551. Springer, 2010.

[2] Marko Grönroos. *Book of Vaadin*. Lulu. com, 2011.

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

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

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

[6] Renaud Pawlak. Jsweet: insights on motivations and design. http://www.jsweet.org, 2015.

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

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