PharoJS: Transpiling Pharo Classes to JS ECMAScript 5 versus ECMAScript 6

Noury Bouraqadi[1], Dave Mason[2]

[1]IMT Nord Europe - France
[2]Toronto Metropolitan University, Toronto, Canada

Abstract
PharoJS is an open source infrastructure (framework + middleware + tools) that allows developing and testing in Pharo Smalltalk, applications which ultimately run on a JavaScript (JS) interpreter. Historically, and up to Pharo version 9, PharoJS generated JS code following the ECMAScript 5 (ES5) standard. Since, ES5 has no explicit support for classes, PharoJS transpiler had to generate JS code that mimicked the Pharo class structures, behaviors and hierachies. After migrating PharoJS to Pharo version 10, we decided to adopt the ECMAScript 6 (ES6) standard. One of the major features provided by ES6 is a set of new constructs to explicitly support class-based programming. In this paper, we describe the changes we have made to PharoJS to switch from ES5 of ES6. We describe the consequences of adopting ES6 on generated code, and the positive impact on all peformance metrics.

Keywords
Javascript, Transpilation, Class-Based Programming, Performance

1. Introduction

JavaScript (JS) is a mainstream language that is used in various application domains. Originally designed for client-side Web development, JS-based software can be found nowadays on servers, desktops, smartphones, as well as small micro-controllers. The size of the JS community, as well as the important actors involved (e.g. Google) support a large investment in JS. The result is highly performant JS Virtual Machines (VM), and an ever-expanding ecosystem of tools, frameworks, and libraries.

We introduced PharoJS [1, 2] to build software using the Pharo language and its IDE, while still benefiting from the JS ecosystem. With PharoJS, development is done 100% in Pharo. For final test stages, and for deployment, the Pharo code is transpiled to JS.

In this paper we compare two implementations of PharoJS. The historical one, relies on ECMAScript 5 (ES5). The latest one, introduced for Pharo 10, adopts the more modern ES6. We support this study by performing various benchmarks.

In the following, section 2 discusses the differences between ES5 and ES6 with respect to class-based programming. Then, section 3 presents PharoJS class transpilation, and how it evolved from ES5 to ES6 style. Section 4 reports metrics we have collected that compare the two versions of PharoJS. Before concluding remarks, section 5 discusses some related work.

IWST 2023: International Workshop on Smalltalk Technologies. Lyon, France; August 29th-31st, 2023
noury.bouraqadi@imt-nord-europe.fr (N. Bouraqadi); dmason@torontomu.ca (D. Mason)
https://nootrix.com/noury (N. Bouraqadi); https://sarg.torontomu.ca/dmason/ (D. Mason)

2. Classes in ECMAScript 5 vs. ECMAScript 6

2.1. From Prototypes to Classes

JS is a prototype-based Object-Oriented (OO) language. Each object contains a collection of values termed *properties*. These are variables owned by the object, which may reference other objects.

Objects are white boxes. Since there is no encapsulation, properties of any object can be read or written anywhere.

JS provides single inheritance among objects. Each object references a parent through the `__proto__` property. Missing properties are looked up in the parent and ancestors through the inheritance chain. The root of the inheritance tree is an object that has no prototype. That is, its `__proto__` property is a null reference.

Beside prototypical objects, JS core concepts include functions and arrays. All of these can have user-defined properties. There are additional types, including symbols, booleans, numbers and strings that have pre-defined properties, but they cannot have new properties defined.[1] All of the above are objects in the sense that they can have fields accessed, including invoking methods. There are also two special values for null references which are not objects: **undefined** and **null**.

The concept of class was introduced later by the ECMAScript standard committee. To ensure backward compatibility, concepts related to classes were introduced in two steps, on top of the concepts of functions and prototypes. First, ECMAScript 5 (ES5) introduced *constructor functions* along with the **new** operator to define and instantiate classes [3]. Then, ECMAScript 6 (ES6) introduced the **class**, **extends**, and **super** keywords to explicitly define classes and better support inheritance [4].

2.2. ECMAScript 5 Style Classes

ES5 has no explicit support for classes. Nevertheless, it's possible to emulate them. This is done by using functions as constructors along with the **new** operator. Such functions initialize new objects that they reference using the **this** pseudo-variable. In this section we describe how. A complete example is provided in appendix A.

The current object in the context of a ES5 function is referenced in functions bound to it via the **this** pseudo-variable. This is similar to the **self** pseudo-variable in Smalltalk. For the following explanations of how **this** is bound, assume the following preamble:

```
function setXY(newX, newY){
    this.x = newX;
    this.y = newY;
}
let aPoint = {};
```

[1]Although `__proto__` for these objects can have properties that will be inherited.

2.2.1. Default property access

When a function is called other than by dot syntax, any reference to **this** refers to the default context. This is `global` in NodeJS and `window` in a web browser. Therefore calling:

```
setXY(10,20);
```

would have the (presumably unintended) consequence of setting `x` and `y` in the `global` or `window` object.

2.2.2. Explicitly bound property access

A function can be bound to any object, making **this** reference it. This is can be done explicitly by calling `bind()` as shown in the following example:

```
(setXY.bind(aPoint))(10,20);
```

In this code, function `setXY` is bound to an empty object `aPoint`. After binding, the function is evaluated with parameters `10` and `20`.

In this and all following examples, the `aPoint` structure is initialized to become `{x: 10, y: 20}`.

2.2.3. Direct property access

When a method is invoked using the dot syntax, such as:

```
aPoint.set = setXY;
aPoint.set(10,20)
```

the object `aPoint` is examined for a property called `set` which is called with **this** bound to the object `aPoint`. This example has the drawback that the object `aPoint` now is larger than the desired `x` and `y` properties, because it also has the `set` property (and in a realistic program many more properties).

2.2.4. `__proto__` property access

If the object doesn't have the target property, the `__proto__` reference is followed to a prototype object. And, this continues if the target property is not found there.

```
aPoint.__proto__ = {set:setXY};
aPoint.set(10,20)
```

This version doesn't use any additional space because the `__proto__` property is defined for all objects. Additionally, objects assigned to the `__proto__` property are available to all related instances.

2.2.5. Constructor property access

When invoking the **new** operator on a function, the JS interpreter creates a new object. Then, the new object is bound to the function playing the role of a constructor. Next, the bound function is performed to initialize the object. The result of the **new** operator is a newly created and initialized object.

To mimic classes sharing methods among their instances, the ES5 approach relies on the constructor function's `prototype` property. This property references an object that serves as the prototype for new objects. Upon invoking the **new** operator, the `__proto__` of the newly created object is set to the value of the constructor function's `prototype` property. So, any slot added to the prototype will be available to new objects. Assuming we had the constructor function `Point` defined as part of the following listing:

```
function Point(){} // Constructor function
Point.prototype = {set: setXY};
aPoint = new Point();
aPoint.set(10,20);
```

The `new Point()` returns a new object with the `__proto__` property set to the `prototype` property of the constructor function. The result is the object containing the same properties as in `__proto__`.

2.2.6. Inheritance

Lastly, to implement a a mechanism like class inheritance, ES5 requires making the `prototype` property of the "subclass" constructor function inherit from the one of the "superclass" constructor functions. Thus, we rely on prototype inheritance to achieve the inheritance of instance methods.

However, we still need to ensure that instance initialization is also inherited. To do so, we call the "superclass" constructor function in the "subclass" constructor function.

Implicitly, constructor functions play the role of classes. Class side methods are simply defined properties of constructor functions. To achieve Pharo-like parallel inheritance for classes as well as their instances, we ensure in PharoJS for ES5 that a "subclass" constructor function inherits from the "superclass" constructor function.

The last brick is to support **super** message sends. We achieve it by binding **this** in a "subclass" method to the overridden "superclass" method to be called. This is illustrated in the following JS code snipet. We define an instance method `instanceMethod` in a "subclass" `Subclass`. It looks up the overriden instance method starting from its "superclass" accessed via `__proto__`. Then, the inherited method is peformed on the receiver (**this**).

```
Subclass.prototype.instanceMethod = function(){
    Subclass.__proto__.prototype.instanceMethod.apply(this);
}
```

2.3. ECMAScript 6 Style Classes

ES6 introduced a set of statements to explicitly support class-based programming. Those are mostly syntactic sugar. Generated entities are very close to what we have in ES5. Appendix B provides a full example.

The keyword **class** allows defining classes. A class definition may include some methods as well as a `constructor` function, that takes care of initializing new instances. Class-side methods can be defined using the **static** keyword.

Under the hood, ES6 classes are just functions. All methods defined in a class, as well as the `constructor` are properties of the class' `prototype`. An instance has its `__proto__` property set to reference the `prototype` of its class.

The class definition may include a reference to a superclass using the **extends** reserved keyword. The **super** keyword allow calling overridden constructors or methods, including class-side methods (keyword **static**).

If no superclass is provided, the root `Object` superclass is used. Oddly enough, this is true only for the instance side. For the class side, a class without an explicit superclass will have its `__proto__` referencing `prototype` of the `Function` JS class. This is a source of metaclass compatibility issues [5, 6]. Instance methods from `Object` may send messages to the class. However `Object` class methods are not inherited when no superclass is provided in the class definition.

3. PharoJS Class Transpilation

In this section we discuss how PharoJS transpiles Pharo classes to equivalent JS objects. PharoJS does support transpiling code from Pharo versions 7 and up to 10. For Pharo versions 7 up to 9, PharoJS generates JS code that complies with ES5. For Pharo 10, we have 2 different PharoJS versions:

- one generates ES5 style JS, and
- the latest one produces ES6 style JS code.

In the remainder of this paper, we focus on these 2 versions of PharoJS for Pharo 10. We do use them for comparison (see section 4). References to Pharo denote Pharo 10.

Regardless of the generated JS class definition style (ES5 or ES6), there are various cases and issues to tackle:

- General case with typically application specific Pharo classes and traits.
- Referencing JS classes in Pharo.
- Pharo-like class-metaclass parallel inheritance hierarchies.
- Class initialization.

3.1. General Case

The typical general case is one with a new user-defined, application-specific class. The definition of a such class as well as all its methods from both instance and class side are transpiled from Pharo to JS. Depending on the PharoJS version, we use either the ECMAScript 5 or the ECMAScript 6 style (see section 2).

3.1.1. Traits

In the current version of PharoJS we provide partial support for reflection, but we ignore traits as entities. Traits are non-existant in the generated JS. While classes are materialized as objects, traits are implicitly conveyed without reification. For every transpiled Pharo class that uses a trait, we generate JS code that includes the transpiled version of all trait methods.

3.1.2. Instance Variables and Slots

Pharo instance variables materialize as JS instance properties. They are not explicitly declared in the resulting JS code because, in JS, property declaration is implicit upon the first reference. A JS object structure is dynamic. Fields come into existence upon the first write to a property.

The current PharoJS version provides some support for reflection on instance variables. However, this is currently limited to `instVarNamed:` and `instVarNamed:put:`. Slot meta-objects are unsupported for the time being.

3.1.3. Class Variables

Pharo class variables are shared among the class, all its subclasses, and all their instances. Class variables are transpiled as properties of a JS object referenced by a property `cp$` of the owning class. All references to class variables are transpiled as direct accesses to that object. We rely for this on JS's lack of encapsulation.

For example suppose a Pharo class `MyClass` has 2 class variables `X` and `Y`. The JS `MyClass` class resulting from transpilation will include the initialization code for property `cp$`.

In ES5, we assign property `cp$` as following.

```
MyClass.cp$ = {
        X: null,
        Y: null
    };
```

In ES6, static property `cp$` is included in the class definition.

```
class MyClass {
     static cp$ = {
        X: null,
        Y: null
    };
}
```

References to class variables result into the same JS code for both ES5 and ES6. For example, read and write accesses to `X` and `Y`, in instance or class methods are transpiled as following:

```
// Read class variables
MyClass.cp$.X;
MyClass.cp$.Y;
// Write class variables
MyClass.cp$.X = 123;
MyClass.cp$.Y = 'Hello Pharo';
```

3.1.4. Shared Pool Dictionaries

A shared pool is conceptually a set of variables that are shared among multiple related classes and their instances. In Pharo, shared pools are defined as subclasses of the `SharedPool` class. Variables belonging to shared pools are defined as class variables of `SharedPool` subclasses.

Upon transpilation of shared pools to JS, we adopt the same approach as Pharo. A shared pool from Pharo is transpiled as JS a class. Variables belonging to a shared pool are transpiled to class variables as presented in the dedicated section 3.1.3. For example, the transpilation of a shared pool `MySharedPool` with variables `X` and `Y` will produce the following ES6 style JS code[2].

```
class MySharedPool extends Object{
    cp$ = {
        X: null,
        Y: null
    };
}
```

References to a variable belonging to some shared pool are transpiled to direct accesses. We rely on the lack of encapsulation in JS. The following listing provide JS code generated for read and write of variables `X` and `Y` from the above `MySharedPool` pool dictionary.

```
class MyClass{
    instanceMethod(){
        return MySharedPool.cp$.X + MySharedPool.cp$.Y;
    }
    static classMethod(newX, newY){
        MySharedPool.cp$.X = newX;
        MySharedPool.cp$.Y = newY;
    }
}
```

3.2. Referencing JS Classes in Pharo

We support referencing JS classes in Pharo through the concept of a *JS class placeholder*. A JS class placeholder is simply a Pharo class that answers **true** to message `isJsClassPlaceholder`.

[2] ES5 style JS code is smilar though obviously class declarations are different.

JS class placeholders are critical for Pharo's code portability to JS. Indeed, Pharo and JS core library APIs are different. So, we extensively use JS placeholders to extend JS core library classes such as `Object` and `Array` with methods that support Pharo APIs. So, Pharo classes that use Pharo core libaries can be transpiled as they are, without any change.

We need JS class placeholders in 3 different situations:

- Transpile a Pharo class that inherits from a JS class, typically Object.
- Send a message to a JS class, typically for instance creation.
- Extend JS classes, typically adding extra methods.

In the first two cases, JS class placeholders are empty classes. Their definitions, are simply skipped.

The last case is about JS class extensions. Methods and class variables if any, are transpiled and linked to the referenced JS class.

3.3. Implementing JS Class Extensions

In ES5 style JS implementing class extensions is straight forward. We use the same approach as for the general case of defining a class following ES5 style JS. Methods are transpiled to functions that are assigned to properties of the constructor function of it's prototype (see section 2.2). Class variables are transpiled as described in section 3.1.3.

In ES6, so called *monkey patching* an existing class is trickier. ES6 accepts adding extra properties referencing functions or objects to an already defined class. However, ES6 forbids creating functions outside classes that contain messages sent to **super**. Our workaround has 2 steps.

1. First, we transpile class extensions to anonymous classes with the same superclass as the target class.
2. Next, we assign all properties of the anonymous class to the target JS class we want to extend.

This, is illustrated in the following code where we show how to extend a presumably existing JS class `ExistingJsClass`. Please note that the actual generated code does the same operations as listed below, but in a slightly more optimized and less readable style.

```
let existingJsSuperclass = ExistingJsClass.__proto__;
let classExtension = class extends existingJsSuperclass {
    instanceMethod(){/* ... */}
    static classMethod(){/* ... */}
    static cp$ = {X: null, Y: null}
}
ExistingJsClass.prototype.instanceMethod = classExtension.
   prototype.instanceMethod;
ExistingJsClass.classMethod = classExtension.classMethod;
ExistingJsClass.cp$ = ExistingJsClass.cp$ ?? {};
Object.assign(ExistingJsClass.cp$, classExtension.cp$);
```

It is important to point out that the anonymous class (referenced by variable `classExtension`) must inherit from the superclass of the extended class (`ExistingJsClass.__proto__`). This is because messages to **super** are statically bound at compile time.

The above class extension transpilation strategy supports multiple extensions for the same class. In case of collisions, methods from the last applied extensions overrwrite all the previous ones. However, we ensure the union of class variables. We rely on the `Object.assign()` static method to copy all class variables declared in a class extension to the target extended class.

3.4. Parallel Inheritance Hierarchies

Part of the PharoJS generated code in ES5 style deals with the inheritance link. It ensures we have parallel inheritance hierarchies (see section 2.2.6). With ES6, this is taken care of by JavaScript. However, some JS classes from core or third party libaries may suffer from the metaclass compatibility issue (see section 2.3).

Since we extend some JS classes such as `Object` with some instance and class side methods, we need to ensure they are inherited so we can seamlessly use them in all subclasses, including ones from core JS or provided by third party libraries. This is why we need to ensure parallel inheritance also for JS classes referenced in PharoJS applications. Such JS classes have placeholders in Pharo. So, upon transpiling placeholders, we generate code that fixes the inheritance links. The code is basically the same for both ES5 and ES6.

3.5. Class Initialization

Pharo provides class side `initialize` methods to allow developers perform initializations after loading classes from source code. Examples are assigning initial values to class variables or shared pools.

In the JS code generated for PharoJS, the `initialize` message is sent to all classes that implement a class side `initialize` method. We include in this list, any existing JS class with a class extension that introduces a class side `initialize` method.

Sometimes class initializations require other classes. Iceberg, the Pharo's package management system, ensures that all dependencies are loaded before sending `initialize` to newly loaded classes. To account for dependencies, PharoJS appends the `initialize` messages at the end of generated JS code. We thus ensure class initializations are performed after all classes are completely defined, and have clean parallel inheritance hierarchies.

4. Benchmarks

4.1. Metrics, Environment, and Applications

We ran benchmarks to compare the JS generated by 2 versions of the PharoJS transpiler. The only difference was that one supports EcmaScript 5 style JS classes, while the other generates

JS code with classes following the Ecmascript 6 style. In this section, we report metrics we have obtained, as well as our benchmarking process.

4.1.1. Metric Dimensions, and Measurement Process

We have conducted benchmarks to compare the 2 PharoJS implementations with respect to the following 4 metrics:

1. JS Generation Time: This is the time taken to transpile Pharo code into a standalone working JS code;
2. JS Code Size: This is the size of JS code generated in the previous step;
3. JS Load Time: This is the load time of the generated JS;
4. JS Execution Time: This is the duration to run JS code generated from the Pharo code.

JS Generation Time PharoJS generates standalone, ready to use, JS. This is done using `PjExporter` and its subclasses that wrap the PharoJS transpiler and perform the following sequence of operations:

- Transpile all classes reachable from a starting point class.
- Transpile extra classes such as `PjClass` that import some kernel class methods from Pharo's kernel metaclasses such as `Behavior` or `ClassDescription`.
- Transpile class extensions that provides Pharo API to core JS classes such as `Object` or `Array`.
- Inject a few handwritten JS functions that allow bootstrapping core classes and support Pharo-like parallel hierarchies.

To test JS generation time we have introduced the `PjExportBenchmarks` class in package `PharoJs-Benchmarking-Application`. We performed 100 iterations of the export of `PjNodeTimingApplication` and `PjBrowserTimingApplication`. Each export time was measured using the `timeToRunWithoutGC` message sent to the block performing the export. Before the 100 iterations we performed an initial `Smalltalk garbageCollect`.

JS Code Size This metric results in a fixed value for a given machine, only varying if we generate for another device. However, there are differences between JS code generated for NodeJS, or for a web browser. For example, the two JS environments have different sets of global variables. This is why we do the size comparison for both.

To generate the 2 files for each JS environment, we evaluate the following expressions in a Pharo playground:

```
PjNodeTimingApplication exportApp.
PjBrowserTimingApplication exportApp.
```

JS Load Time To measure load time, we rely on JS console timers [7]. We generate JS code using PharoJS by sending `exportApp` message as described above. Then we do the following manual changes:

- Remove the last line that sends the `start` message to the entry-point class. This is because we don't want to measure the execution, but only the load duration.
- Insert `console.time('PharoJS');` as the first statement of the generated JS file.
- Append `console.timeEnd('PharoJS');` at the end of the generated JS file.

Note that the measured time includes class initializations as well as inheritance and metaclass links setup.

The modified JS code displays time required for 1 iteration. To have some statistically meaningful data, we run 100 iterations for each environment. No warmup was required as these benchmarks are dominated by I/O and we observed no appreciable variance. On NodeJS we introduce a shell script that calls `node <generatedJsFile>` 100 times. For the web browser, we appended the following lines to the PharoJS generated JS code:

```
if(!localStorage.iterationsCount){
  localStorage.iterationsCount = 100;
  localStorage.nextIteration = 1;
}else{
  localStorage.nextIteration = 1 + Number(localStorage.
     nextIteration);
}
if(Number(localStorage.nextIteration) < Number(localStorage.
    iterationsCount)){
  window.setTimeout(()=>{window.location.reload();});
}else{
  console.log(">>> PharoJS load benchmark finished");
  localStorage.clear();
}
```

We rely on `localStorage` to save the iteration index. Then, we trigger page reload at the end of the script. This is repeated until the desired number of iterations (100) is reached. Note that by default, the web browser clears the log after each reload. So, to collect measurements, we activate `Preserve log` in the browser's developer tools.

JS Execution Time For each metric, we measured the median time from 10 runs following 5 warmup runs. The duration of a single run was measured as the sum of 50 to 300 iterations. We have chosen the number of iterations to be large enough, while keeping the duration of each benchmark under 10 minutes.

4.1.2. Applications used for Benchmarks

The PharoJS transpiler requires an entry point class. The transpiler converts to JavaScript all classes reachable from the entry-point class. Exporter adds some extra code to allow bootstrapping core classes, and sending the message `start` to the entry-point class.

For the purpose of benchmarking we relied on our benchmarking infrastructure gathered under `PjTimingApplication` (provided as part of the PharoJS libraries). It is used in the following classes that provide some utility methods (e.g. for logging), as well as interpreter specific support methods:

- `PjNodeTimingApplication` : targets NodeJS.
- `PjBrowserTimingApplication` : targets JS interpreters embedded in web browsers

`PjTimingApplication` measures execution time for transpiling various Pharo language constructs and objects (e.g. non-return blocks). But, the PharoJS evolution addressed in this paper focused only on class generation. So, we consider only execution time metrics for two popular micro-benchmarks involving multiple classes and message sends:

- Richards implemented in package `PharoJs-Benchmarking-Richards`
- DeltaBlue implemented in package `PharoJs-Benchmarking-DeltaBlue` .

The Richards benchmark was initially written in BCPL by Martin Richards from the Computer Laboratory of the University of Cambridge, UK. It is "a machine and language independent benchmark test designed for the comparison of System Implementation Languages and their implementations on various machines by various compilers." [8]

The DeltaBlue benchmark is based on an incremental constraint solver, originally implemented in Smalltalk [9]. Multiple implementations exist also for various other languages [10].

4.1.3. Environment

Transpilation was done using 64-bit Pharo 10.0.0 build 536 images. PharoJS was installed from 2 branches sharing a common parent commit:

- `pharo10` : implements ECMAScript 6 style JS class generation for Pharo 10.
- `ecmascript5` : implements ECMAScript 5 style JS class generation for Pharo 10.

The 2 PharoJS versions are loaded by evaluating the following expression in a playground, replacing `<branchName>` with the name of the desired branch.

```
Metacello new
   baseline: 'PharoJS';
   repository: 'github://PharoJS/PharoJS:<branchName>';
   load
```

Pharo images run on a Pharo 100 Darwin x86 64-bit VM updated on 2022-12-07 at 20:44:51.
We run benchmarks on 2 different JS interpreters:

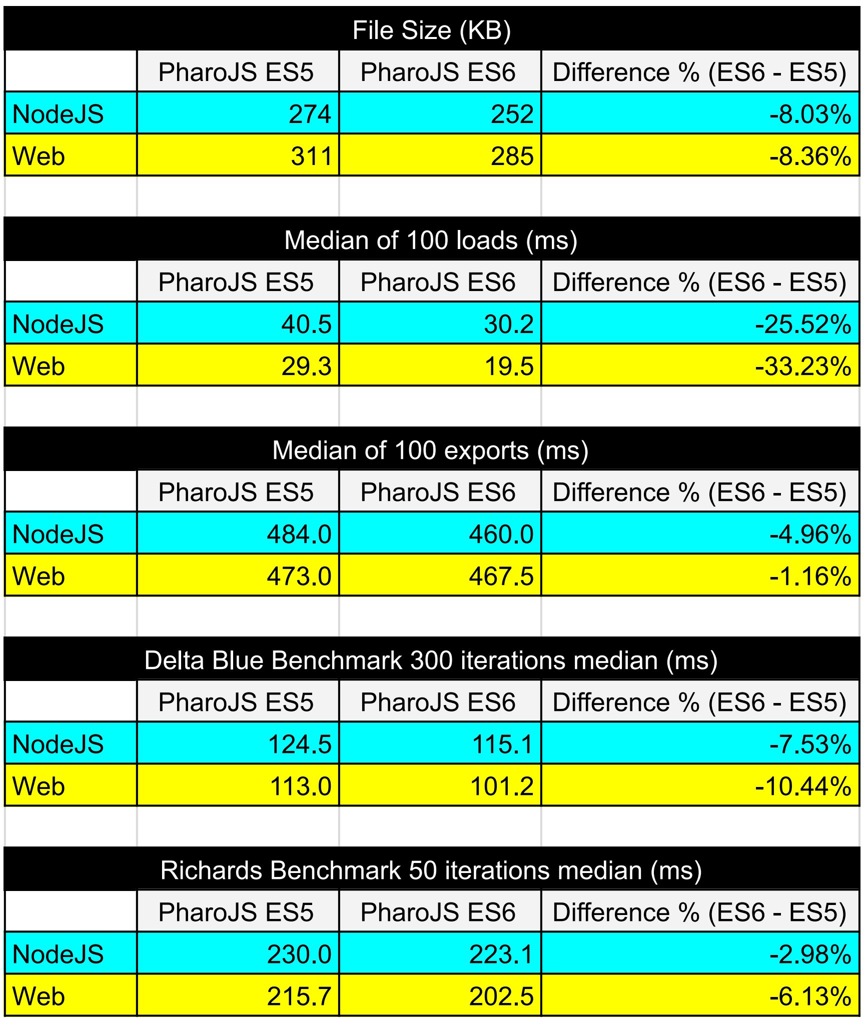
- NodeJS v19.8.1
- Web browser application on Google Chrome Version 111.0.5563.146 (Official Build) (x86_64)

In both cases, the computer is a Mac Book Pro 2 with:

- CPU: 8-Core Intel Core i9, clocking at 2.3 GHz
- RAM: 32 GB 2667 MHz DDR4
- OS: Mac OS X Ventura 13.2.1 (22D68)
- Hard drive: 1 TB SSD, PCI-Express with APFS File System

4.2. Benchmark Results and Discussion

As presented above, we use 2 different applications to compare differences between ES5 and ES6 styles. One is a NodeJS application, for which transpilation results in a JS code generated from 82 Pharo classes. The second is a single page web application, for which JS code is obtained from 85 Pharo classes. All metrics are summarized by tables in figure 1.



File Size (KB)			
	PharoJS ES5	PharoJS ES6	Difference % (ES6 - ES5)
NodeJS	274	252	-8.03%
Web	311	285	-8.36%

Median of 100 loads (ms)			
	PharoJS ES5	PharoJS ES6	Difference % (ES6 - ES5)
NodeJS	40.5	30.2	-25.52%
Web	29.3	19.5	-33.23%

Median of 100 exports (ms)			
	PharoJS ES5	PharoJS ES6	Difference % (ES6 - ES5)
NodeJS	484.0	460.0	-4.96%
Web	473.0	467.5	-1.16%

Delta Blue Benchmark 300 iterations median (ms)			
	PharoJS ES5	PharoJS ES6	Difference % (ES6 - ES5)
NodeJS	124.5	115.1	-7.53%
Web	113.0	101.2	-10.44%

Richards Benchmark 50 iterations median (ms)			
	PharoJS ES5	PharoJS ES6	Difference % (ES6 - ES5)
NodeJS	230.0	223.1	-2.98%
Web	215.7	202.5	-6.13%

Figure 1: Summary of Benchmark Results

We can see that the numbers for ES6 are always smaller than the ones for ES5. Meaning that latest PharoJS version generating ES6 style JS code always out-performs the previous version with ES5.

Generation time and load time are coupled to code size. Since ES5 has no explicit support for classes, we need to generate code that mimicks the desired features, such as subclass to superclass inheritance relationships, or messages sent to **super**. ES6 instead provides statements and constructs that directly support them. This results in less code to generate and load.

Execution time is also positively affected with the transition from ES5 to ES6 style JS code. Generating code for ES5 that correctly captures the Smalltalk semantics produces some very non-idiomatic code, whereas the structure for ES6 is much more idiomatic. We suspect that this is a result of JS compiler and VM optimizations, for example when looking up methods in a class hierarchy.

5. Related Work

Related work falls into two categories: code compilation or transpilation into Javascript, and looking at the relationship of ES5 to ES6 code.

5.1. Javascript as a Target for Language Implementation

Because Javascript is so ubiquitous, it is a very popular target for code generation. There are numerous projects to translate to Javascript from a variety of languages [11]. There are more than a dozen different projects for each of such popular object-oriented languages as Ruby, Python, Java, and C#, in addition to Smalltalk and Java. Detailed analysis of all of them is beyond the scope of this paper. We will focus on four of these systems.

5.1.1. Amber [12]

Amber is a development IDE that runs in the browser. It predates PharoJS, so was created in ES5 (actually probably ES3 or earlier), and has not upgraded. Its performance was already worse than PharoJS, so with the PharoJS performance boost from ES6 syntax, Amber falls further behind.

5.1.2. SqueakJS [13]

SqueakJS is handwritten Javascript implementation of a Smalltalk VM. The primary JS code is a Smalltalk VM/interpreter and the interpreter is a single object, with almost all operations calling methods on, or otherwise manipulating **this**.

While it is written in ES5 or earlier, we speculate that it probably doesn't suffer a significant performance hit from not having moved to ES6.

5.1.3. Powerlang-JS [14, 15]

Powerlang-JS parallels Pharo's Slang - that is, the transpiler only needs to be able to compile the Smalltalk interpreter, which doesn't use the full power of Smalltalk. It was written relatively recently, so was written from the beginning to use ES6.

5.1.4. JSweet [16, 17]

There are over a dozen systems that transpile from Java, but perhaps the most relevant is JSweet. It has a similar model to PharoJS, in that it doesn't attempt to support traditional Java interaction models such as Swing, but rather focuses on clean interaction with native Javascript packages. Because Java has a simpler execution model (no class-side inheritance, no equivalent to class variables, no DNU) and because JSweet targets TypeScript, the transpilation is more straightforward. But, since Java has static typing, mapping to Javascript libraries becomes more complex. Since JSweet targets TypeScript, the use of JS5 or JS6 is left up to the Typescript transpiler. But, the simpler execution model means there would be less of a performance hit for staying with ES5.

5.2. Relationship of ES6 to ES5

5.2.1. Refactoring Legacy JavaScript Code to Use Classes: The Good, The Bad and The Ugly [18]

This paper discusses various mechanisms that were chosen to migrate ES5 code to ES6. They found some of the required changes difficult to implement. We did not experience these problems with our migration because we had our own idiosyncratic implementation of classes already, so it was easy to map them to ES6 constructs. This shows an advantage of writing code in a mature, stable language like Smalltalk, and then using a tool like PharoJS to generate Javascript code - nothing to do on the part of the application programmer beyond recompilation.

5.2.2. On the Usage of New JavaScript Features through Transpilers: The Babel Case [19, 20, 21]

Babel is a tool to provide JS backward compatability. It allows the programmer or transpiler to target ES6 features. Babel takes care of converting them to work with earlier levels of ECMAScript.

5.2.3. Lebab transpiles your ES5 code to ES6/ES7 [22]

Lebab is a tool to bring ES5 code up to date with newer standards. Theoretically we could have left PharoJS generating ES5 and letting Lebab translate to ES6. However the code generated by PharoJS ES5 is quite idiosyncratic, and Lebab is not terribly sophisticated with recognising non-standard code patterns.

6. Conclusion & Future Work

Summary. Prior to ECMAScript 6 (ES6), JavaScript had no explicit support for class-based Object Oriented Programming. PharoJS's previous version therefore had to generate code that emulated classes and their inheritance. We have discussed how we have converted the transpilation of the Pharo/Smalltalk class structure in Javascript from classic ECMAScript 5 (ES5) to more modern ES6 Javascript.

This change has resulted in a more human-readable Javascript code. It also brought a nice size and speed benefit, as shown by benchmarks we have run. For those outside the Smalltalk world, this should provide a tutorial on how to represent a somewhat tricky class structure in ES5 or ES6 Javascript.

Even though ES6 provides support for classes and inheritance, transpiling Pharo classes to ES6 has it's own challenges. We discussed the rather frequent cases where we need to extend JS core or third party classes with extra methods. These class extensions allow JS objects to align with the Pharo/Smalltalk semantics, as well as to introduce application specific methods to built-in classes. We showed that the use of intermediate anonymous classes allows addressing the tricky case of methods that send messages to **super**.

Future Work. One of the next stages in supporting Javascript development from a Smalltalk base would be to provide better debugging support. Debugging support at the level that is possible with Smalltalk would be onerous and have a very significant performance cost. One step that would support this would be to generate a source map that would tie Javascript code back to the Smalltalk source, and support the debuggers available in browsers or NodeJS. Another possible path would be to use the Debug Adapter Protocol [23, 24].

We are also working to improve the inter-operation of PharoJS and standard Javascript libraries, in both directions. Currently it is fairly easy to access Javascript libraries from PharoJS, although the mapping from Smalltalk selectors to Javascript function names can be quite arbitrary. We also have limited support for building libraries with PharoJS that can be seamlessly accessed from other Javascript code, and we would like to improve this support.

Appendix

A. Example of Class Definition in ECMAScript 5 Style

```
// A class is actually a constructor function
function Counter() {
    this.count = 0;
}
// Instance methods
Counter.prototype.increment = function () {
    this.count = this.count + 1;
}
// Class methods
Counter.createDefaultInstance = function(){
    return new this();
}
Counter.getDefaultInstance = function(){
    if(this.defaultInstance == null){
        return this.createDefaultInstance();
    }
    return this.defaultInstance;
}
Counter.resetDefaultInstance = function(){
    this.defaultInstance = null;
}
// "Subclass" as a constructor function
function CircularCounter() {
    // Call superclass constructor
    Counter.apply(this);
    this.max(999);
}
// Ensure instance methods are inherited
CircularCounter.prototype.__proto__ = Counter.prototype;
// Ensure class methods are inherited
CircularCounter.__proto__ = Counter;
CircularCounter.prototype.max = function (maximum) {
    this.maxValue = maximum;
}
// Override inherted instance method
CircularCounter.prototype.increment = function () {
    if (this.count == this.maxValue) {
        return this.count = 0;
    }
    // Call overridden instance method
    CircularCounter.prototype.__proto__.increment.apply(this);
}
// Override inherted class method
CircularCounter.createDefaultInstance = function(){
    let counter = Counter.createDefaultInstance.apply(this);
    counter.max(3);
    return counter;
}
// Avoid subclass read access superclass property
CircularCounter.defaultInstance = null;

let cc = CircularCounter.getDefaultInstance();
console.log(cc);
for (i = 0; i < 5; i++) {
    cc.increment();
    console.log(cc.count);
}
```

B. Example of Class Definition in ECMAScript 6 Style

```
class Counter {
    constructor() {
        this.count = 0;
    }
    // Instance methods
    increment() {
        this.count = this.count + 1;
    }
    // Class methods
    static createDefaultInstance(){
        return this.defaultInstance = new this();
    }
    static getDefaultInstance (){
        if(this.defaultInstance == null){
            return this.createDefaultInstance();
        }
        return this.defaultInstance;
    }
    static resetDefaultInstance (){
        this.defaultInstance = null;
    }
}

class CircularCounter extends Counter {
    constructor(){
        // Call superclass constructor
        super();
        this.max(999);
    }
    max(maximum) {
        this.maxValue = maximum;
    }
    // Override inherted instance method
    increment() {
        if (this.count == this.maxValue) {
            return this.count = 0;
        }
        // Call overridden instance method
        super.increment();
    }
    // Override inherted class method
    static createDefaultInstance(){
        let counter = super.createDefaultInstance();
        counter.max(3);
        return counter;
    }
}
// Avoid subclass read access superclass property
```

```
CircularCounter.defaultInstance = null;

let cc = CircularCounter.getDefaultInstance();
console.log(cc);
for (i = 0; i < 5; i++) {
    cc.increment();
    console.log(cc.count);
}
```

References

[1] N. Bouraqadi, D. Mason, Mocks, proxies, and transpilation as development strategies for web development, in: J. Laval, A. Etien (Eds.), IWST'16: Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies, ESUG, Association for Computing Machinery, New York, NY, United States, Prague, Czech Republic, 2016.

[2] N. Bouraqadi, D. Mason, Test-driven development for generated portable javascript apps, Science of Computer Programming 161 (2018) 2–17.

[3] S. Stefanov, K. C. Sharma, Object-Oriented JavaScript, second edition ed., Packt Publishing, 2013.

[4] M. Haverbeke, Eloquent Javascript, 3rd edition ed., No Starch Press, 2018.

[5] M. N. Bouraqadi-Saâdani, T. Ledoux, F. Rivard, Safe Metaclass Programming, in: Proceedings of OOSPLA'98, ACM, 1998.

[6] N. Bouraqadi, Safe metaclass composition using mixin-based inheritance, Journal of Computer Languages and Structures 30 (2004) 49–61. Special issue: Smalltalk Language.

[7] Mozilla, Console - webapis | mozilla, 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/console#timers, [Online; accessed 2023-05-03].

[8] M. Richards, Bench - internet archive - way back machine, 1999. URL: https://web.archive.org/web/20110805035442/http://www.cl.cam.ac.uk/~mr10/index.html, [Online; accessed 2023-05-03].

[9] B. N. Freeman-Benson, J. Maloney, The DeltaBlue algorithm: An incremental constraint hierarchy solver, in: Proceedings of the Eighth Annual International Phoenix Conference on Computers and Communications, IEEE Computer Society, 1989, pp. 538–539.

[10] S. Marr, Are we fast yet? comparing language implementations with objects, closures, and arrays, https://github.com/smarr/are-we-fast-yet/, 2022. [Online; accessed 2023-05-02].

[11] J. Ashkenas, List of languages that compile to javascript, 2023. URL: https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS, [Online; accessed 2023-05-13].

[12] H. Vojčík, N. Petton, Amber smalltalk, 2022. URL: https://www.amber-lang.net/, [Online; accessed 2023-05-13].

[13] V. Freudenberg, Squeakjs, 2023. URL: https://squeak.js.org/, [Online; accessed 2023-05-13].

[14] J. Pimás, PowerlangJS: A quick way to get your smalltalk to the web?, in: FAST Workshop 2022 on Smalltalk Related Technologies, 2022. URL: https://openreview.net/forum?id=DRUVWUDX_z.

[15] J. Pimás, Powerlang-js github repository, 2022. URL: https://github.com/powerlang/

powerlang-js, [Online; accessed 2023-05-13].

[16] R. Pawlak, Jsweet: Insights on motivations and design, A transpiler from Java to JavaScript. EASYTRUST 16 (2015).

[17] R. Pawlak, Jsweet home page, 2018. URL: https://www.jsweet.org/, [Online; accessed 2023-05-13].

[18] L. H. Silva, M. T. Valente, A. Bergel, Refactoring legacy javascript code to use classes: The good, the bad and the ugly, in: G. Botterweck, C. Werner (Eds.), Mastering Scale and Complexity in Software Reuse, Springer International Publishing, Cham, 2017, pp. 155–171.

[19] T. Nicolini, A. Hora, E. Figueiredo, On the usage of new javascript features through transpilers: The babel case, IEEE Software early access (2023) 1–12. doi:10.1109/MS.2023.3243858.

[20] MRajaelM, Babel: The key to using the latest javascript features in any environment, 2023. URL: https://medium.com/@mrajaeim/babel-the-key-to-using-the-latest-javascript-features-in-any-environment-e317e7814c8f, [Online; accessed 2023-04-25].

[21] B. Team, Babel, 2023. URL: https://babeljs.io/, [Online; accessed 2023-04-25].

[22] community, Lebab: Transpiling ES5 code to ES6/ES7, 2023. URL: https://github.com/lebab/lebab, [Online; accessed 2023-05-13].

[23] Microsoft, What is the debug adapter protocol?, 2021. URL: https://microsoft.github.io/debug-adapter-protocol/, [Online; accessed 2023-07-12].

[24] C. Chiarulli, Understanding debug adapters: Bridging ides and debuggers, 2023. URL: https://medium.com/@chrisatmachine/understanding-debug-adapters-bridging-ides-and-debuggers-c31a3b02b30a, [Online; accessed 2023-07-12].