FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Usability improvements to JavaScript/ECMAScript
Student:	Bc. Jan Jindráček
Supervisor:	Ing. Konrad Siek, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

JavaScript is the most popular programming language ever. It follows a One JavaScript philosophy to achieve backwards compatibility: old features are never removed or fixed, but new features are introduced, new syntax must inter-operate with old syntax. Therefore, JS carries a lot of historical baggage causing ambiguity and redundancy that makes JS difficult to learn and maintain.

The goal is to explore whether dropping backwards compatibility makes it possible to develop a drop-in replacement for JS that is:
- is interoperable with JS and existing frameworks,
- legible and easy to learn for JS devs,
- more usable than JS.

The student will:
- identify ambiguities and redundancies in JS and propose specific solutions,
- propose a new language based on the above
- develop an interpreter or compiler for this language,
- perform experiments, surveys and/or interviews to show the improved usability over JS
- do case studies showing the extent of interoperability with JS and its ecosystem.

Electronically approved by Ing. Michal Valenta, Ph.D. on 19 February 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Usability improvements to JavaScript/ECMAScript

Bc. Jan Jindráček

Department of Software Engineering
Supervisor: Ing. Konrad Siek, Ph.D.

May 4, 2021

Acknowledgements

I wish to thank Ing. Konrad Siek, Ph.D. for his patience, help, advice and encouragement in guiding me to this point.

Declaration

In Prague on May 4, 2021 . . . .

Citation of this thesis

Jindráček, Jan. *Usability improvements to JavaScript/ECMAScript.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021. Also available from: ⟨`https://npmjs.com/package/jonscript`⟩.

Abstrakt

JavaScript je nejpopulárnější programovací jazyk na světě a je nedílnou součástí dnešních webových stránek. Nicméně, i přes jeho popularitu je zdrojem frustrace pro vývojáře, kteří s tímto jazykem pracují. Tato práce identifikuje hlavní problémy tohoto jazyka jako nedostatek konzistence základních vlastností jazyka, což často vede k neintuitivní sémantice. Tento problém je dlouho známý a byl již částečně vyřešen několika jazyky, jazykovými extenzemi, striktním módem a lintery. Tato práce představuje JonScript, jazyk, který má jednoduchou syntaxi, jednoduše se používá, a je kompatibilní s existujícími knihovnami v JavaScriptu. Jeho výhody spočívají v univerzálním použití arrow funkcí, funktorovými výrazy, přetěžování operátorů, automatickým použitím operátoru async a pattern matchingu. Také zjednodušuje a zavádí konzistenci k funkcionalitě známé z JavaScriptu: syntax stringů, dědičnost, vytváření třídních instancí a výrazům obsahujícím nedefinované vlastnosti objektů. Tato práce obsahuje případovou studii, která slouží jako příklad použitelnosti JonScriptu a jeho kompatibilitě s jQuery a Redux knihovnami. Také obsahuje porovnání s výkonností vůči TypeScriptu, která poukazuje na vyšší výkonnostní nároky JonScriptu, což ukazuje na potřebu optimalizace.

Klíčová slova Sémantika v programovacích jazycích, Funkcionalita a konstrukty v jazycích, Kompilátor, JavaScript, TypeScript, JonScript

Abstract

JavaScript is the most popular language and the backbone of web development. However, despite its popularity, it is a source of frustration for its developers. This thesis identifies JavaScript's main problem to be a lack of consistency within its features (often subtle) which leads to counterintuitive semantics. The problem is well known and has been partially addressed by a number of languages, extensions, JavaScript strict mode, and linters. This thesis introduces JonScript, a simple and easy-to-use language on top of JavaScript that is compatible with the JavaScript ecosystem. JonScript's features include universally applied arrow function semantics, functor syntax, operator overloading, async inference, and pattern matching. It also simplifies and regularizes a number of features with respect to JavaScript: string syntax, inheritance, instantiation, and expression of undefined object members. The thesis contains case studies showing the usability of JonScript and compatibility with jQuery and Redux. It also contains a performance evaluation showing overhead with respect to TypeScript, suggesting optimization is required.

Keywords Semantics of Programming Languages, Language Constructs and Features, Compiler, JavaScript, TypeScript, JonScript

Contents

List of Figures

Chapter 1

Introduction

JavaScript is the world's most used language [44]. It is the backbone of web development. JavaScript has many amazing features and a wide range of frameworks and extensions.

1.1 Utility of functional programming

JavaScript contains powerful functional programming features, which give it the ability to easily create high-level abstractions without any boilerplate code.

JavaScript adheres to the principle of function as first class citizen. This means that you can, for example, easily add inversion control using function call as a default parameter value of a function. Or, as it is common in functional programming, you can create a function to which you pass another function as a parameter or return a function from a function. Furthermore, the standard library in JavaScript includes powerful functions for array manipulation, `map`, `filter`,[1] `sort` [2] and others, reminiscent of LISP mapping functions [30]. Modern programming languages that are not considered functional programming languages support there features: C#,[3] Lua,[4] Python,[5] Go,[6] Java[7] and PHP[8] all have functional programming features.

In order to showcase just how useful functional programming is, here is an example, where you need to create map-reduce algorithm function. This function counts the length of the words in a string. This type of algorithm

[1] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter`

[2] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort`

[3] `https://www.codeproject.com/Articles/375166/Functional-Programming-in-Csharp`

[4] `https://luafun.github.io/intro.html`

[5] `https://realpython.com/python-functional-programming`

[6] `https://blog.logrocket.com/functional-programming-in-go/`

[7] `https://www.geeksforgeeks.org/functional-programming-in-java-with-examples`

[8] `https://phptherightway.com/pages/Functional-Programming.html`

is useful when creating histograms or other statistics about a data set. I am creating two different versions of this algorithm in JavaScript in order to point to the differences between imperative and functional programming.

In the first version of the algorithm, map-reduce is implemented without any functional programming features. In the example below we create a function which splits any string by spaces. For simplicity of presentation assume that the string does not contain punctuation. After that, we iterate over the array of sub-strings (words) and add their lengths in a hash map. This function returns the completed map.

```
const wordCount = text => {
    const split = text.split(" ");
    const map = {};
    for (let i = 0; i < split.length; i++) {
        if (map[split[i].length]) {
            map[split[i].length] += 1;
        } else {
            map[split[i].length] = 1;
        }
    }
    return map;
};
```

In the next example we create the same algorithm, but with JavaScript functional programming features. We split a string by spaces into words, just as before, but after, we apply a `SELECT`-like function which converts an array of words into an array of word lengths. After this operation, we add each new length into a hash map through the `reduce` function (e.g. a historgram).

```
const wordCount = text => text
    .split(" ")
    // This is equivalent to SELECT query in SQL
    .map(word => word.length)
    // Copy map, rewrite element at index 'length'
    .reduce((p, length) => ({
        ...p, [c]: (p[length] || 0) + 1
    }), {});
```

In the first example, I am using imperative programming. While this is not incorrect in any way, it does not allow for any abstraction. This means that if I need to write a similar function, I either need to start from scratch, or move to object-oriented programming features.

The second is shorter, but the main advantage is that by simply using parameter for the map function, you can create an abstract map-reduce algorithm and use it elsewhere. The example of such abstraction is shown in the example in the next code listing: a general-purpose map-reduce algorithm, that can take any mapping function and an array and add the results into a hash map. It returns the said hash map.

```
/**
 * Library function to provide map-reduce functionality
 * {values} Array of input values
 * {map} Function to map values passed in into the desired type
 */
const mapReduce = (
    values,
    map,
) => values
    // Pass map function into .map() - function as parameter
    .map(map)
    // Finally, reduce the array into a hash map
    .reduce((hashmap, value) => ({
        ...hashmap,
        [value]: ((hashmap[value] || 0) + 1)
    }));
```

Now we have defined a generic function with two type parameters that can take any map function. Using this generic reduce function we can simplify other similar functions—not just the word count. See the next listing for an example of a function that uses this abstraction. This function counts how many words start with certain letter. The simplification here occurs thanks to us being able to pass a simple mapping function directly into another function—the map-reduce.

```
/**
 * Counts how many words start with certain letter
 */
const firstLetterCount = text => mapReduce(
    text.split(" "),
    word => word[0]
);
```

1.2 Powerful resources at the disposal of JavaScript developers

Another advantage JavaScript developers have is a large amount of free resources at hand to help them, thanks to large community support with package distributions. The best known package repository of these is NPM [48]. Some of the best known packages are: TypeScript [10], ESLint [43], React [18], Knockout [32], Vue [27] and Angular [17]. TypeScript and ESLint are compile-time libraries which affect the quality of the code produced, while React, Knockout, Vue and Angular are frameworks, used to manipulate the HTML user interface.

Thanks to such a large ecosystem, modern JavaScript developers also have access to language extensions, such as JSX [19]. JSX allows you to directly add HTML tags to your JavaScript code and have them render through JavaScript in your browser. JSX constructs are reminiscent of those found in Razor [40] templates in C#. React and Angular take advantage of this by providing powerful component based systematic approaches to web development.

1.3 Problematic behaviour within JavaScript

With such amazing features, one would think that JavaScript is the perfect language. This, however, is not the case. Consider that some of the most used packages in JavaScript are there to help you write cleaner and more understandable code (ESLint and TypeScript). This hints at a deeper problem. Developers like to mightily complain about this language, despite its features [16, 26, 28].

The main problem with JavaScript is that the language is comparatively old, has undergone many changes, and—much like Java—maintains full backwards compatibility. This causes a major problem—there are many ways of doing the same thing [33, 37, 39]. To explain myself further, I will now begin to explore a simple use case.

1.3.1 Problematic variable declaration

Consider an example, where I want to define an object with one property. This property will have a function assigned to it, which returns another object with just one property. While this seems easy to understand and do, there are several issues a developer will encounter along the way. To start with, I will need to define a variable—there are three ways of doing it:

- `const` defines a constant variable,
- `let` defines a block-scoped variable, and
- `var` defines a function-scoped variable.

Those statements differ in semantics. If you try and reassign into constant variable (`const`), JavaScript throws an exception upon assignment. You can re-assign to block-scoped variable (`let`), and to the function-scoped variable (`var`). There is not much special about `let`, except for the differences between `let` variables and function-scoped variables. Function-scoped variables (`var`) are hoisted—you can access them in their scope *before* they are defined. Their values will be `undefined`.[9] Also, `var` can be redefined—you can define the same variable multiple times in the same scope, and when defined inside the global scope, `var` becomes the property of the global object (usually `window`).[10] Therefore, `var` defined variables will mutate the object, even before they are assigned to.

In our case, let us choose `const`, since we will not be reassigning the value anywhere:

```
const Module = {}; // assigns an empty object
```

1.3.2 Problematic property definition

Next, we are going define to create a property on `Module`. There are three distinct ways of doing that:

1. by creating a property: `o.property = value`,
2. by using a get/set syntax (this means defining a getter and setter function), and
3. by calling the function `Object.defineProperty`.

Using the get/set syntax requires that you define a getter, setter, or both. When you use the get/set syntax, you run the risk of unexpected behaviour, if you want to create a read-only property—if you try to create a read-only property by using a getter without setter during property declaration and then try to assign to this property, JavaScript will throw an exception (assuming you are using strict mode).[11] However, you can remove the getter-only property using the `delete` operator from the object and then assign a new value into it.

```
const o = {
    get length() { return this._len },
    _len: 5
};

// You may assume that a getter cannot be assigned to, but:
delete o.length;
o.length = 6; // o.length is now a regular property

console.log(a.length); // Prints 6
console.log(a._len);   // Prints 5
```

[9] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined

[10] https://developer.mozilla.org/en-US/docs/Web/API/Window

[11] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

To truly create a read-only property, one must use `Object.defineProperty`. I showcase the use of this method by an example found in this [12] documentation.

```
const object1 = {};

Object.defineProperty(object1, 'property1', {
  value: 42,
  writable: false,
  // Non configurable properties cannot be deleted in scrict mode
  configurable: false,
});

// throws an error in strict mode
object1.property1 = 77;

// expected output: 42
console.log(object1.property1);
```

For now, let us use a simple assignment. Now, our code looks like this:

```
const Module = {
    Example: ?
}
```

1.3.3 Problematic function declaration

Now we need to define a function, which returns an object. There are four ways to define functions in the text below. Each way has certain subtle differences:

- Classic JavaScript function—`function name() { return 1; }`;
- arrow function expression;
- instantiating an instance of the Function class; or
- a class definition.

We discuss each of these below.

[12] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty

Standard JavaScript function First, we will discuss standard JavaScript functions. Standard functions definition is initialized by the keyword `function`, followed by an optional name and a code block. Function may have a set of parameters, last one of which can be a spread (variable length argument list interpreted as an array).

The result of calling a function may depend on its execution scope. One can access this scope by the `this` keyword. Function context in a global scope without strict mode refers to `window`—in browsers, `window` is a global namespace object. It contains the entire standard API. When strict mode is used, the scope is not defined. Context can be redefined by using functions `call`, `apply` or `bind` on said function. You can redefine the contents of the `this` object by running the function in a different scope.

Each function also has the property `length`, which refers to the number of parameters and the 'arguments' object, which refers to function arguments. The `arguments` object also has the `callee` property, which refers to the function itself—similar to the aforementioned context, `callee` is not defined when using the strict mode.

Arrow function expression Second, let us discuss the arrow function expression. An arrow function expression does not change its context based on execution. Arrow function expression context is set and frozen when such function is defined. Also, an arrow function expression cannot use the `yield` operator, nor have its context manipulated by `bind`, `call` or `apply` methods. Lastly, an arrow function cannot be used for constructors and does not have the `new.target` property, which lets you see whether or not it has been called using the operator `new`.

Function class instance Third, we can create a function by using the `Function` class [1]. This means you convert a string into a function, in a similar way to how `eval`[13] works. This is highly insecure and not recommended - for obvious reasons, since it allows for cross-site scripting (XSS) [11] attacks on your site under certain conditions, when a malicious script is placed within the Function class constructor.

Function as class And last, but not least, we could define the function as class. Classes in JavaScript are essentially just functions, which return an object. Some classes need not to be instantiated with the `new` keyword—you can instantiate those classes as if you were calling a function. Also, you can instantiate a class with the `new` keyword, without using parentheses `()`. If you create your own class and call it without using `new`, an exception will

[13]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval`

occur. Class definition can be mutated at any time by adding properties to their `prototype` property.[14]

Let's choose the arrow function for simplicity and continue. Next, we need to define an array and return the desired value.

```
const Module = {
    Example: () => {

    }
}
```

1.3.4 Multiple array definitions

There are three ways of defining an array: instantiating the Array class constructor with the `new` keyword, instantiating the Array class without the `new` keyword (as if it was a regular function), or creating the array literal (`[]`). The difference between the array literal, and calling Array class, is that is there is a single numeric parameter passed into the Array class, it will generate an array of empty elements of certain length. There are no other differences between these, so let us simply define a variable containing an array by using the array literal and now we have:

```
const Module = {
    Example: () => {
        const map = [1, 2, 3];
    }
}
```

Since JavaScript has a strong support of functional programming, we can take the `map` property (similar in function to `SELECT` in SQL) and assign it into separate variable.

```
const Module = {
    Example: () => {
        const map = [1, 2, 3].map;
    }
}
```

Since JavaScript only has one kind of return statement, we can use it and return the desired value:

[14] `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes`

```
export const Module = {
    Example: () => {
        const map = [1, 2, 3].map;
        return {
            result: map(i => i + 1),
        };
    },
};
```

1.4 Problem definition and exploring possible solutions

In the text above, I outline at least six different ways of creating the same object. Their differences are not just syntactic, they change the semantics of the program as well.

The algorithm in the last example will throw an exception when executed. This is due to the `map` function of an array being context-dependent.[15] This means that after assignment into `map` variable it will throw an exception when called, since it is context is undefined. You can fix this problem by either wrapping the statement in an arrow function expression, or by using the function `bind` - this would create a new `map` function with it's context set from then on.

The problem, at its core, is that there are not just many ways of doing the same thing in JavaScript, but many ways of doing relatively similar things, that differ from each other in subtle details.

This invites a question—what can be done about the state of things? In general, the solution should prevent developers from encountering hidden complexities due to backwards compatibility or strange design choices. The solution should also be compatible with current JavaScript ecosystem, in order to allow users to use the massive amount of packages in existence. Last, but not least, the solution should not take experienced developers a long time to master, or change the positive aspects of the language.

Such solution could be creating a new language, or a language extension which fits the above mentioned traits. Such language would need to have a similar syntax to modern JavaScript, but it's inside logic should be intuitive, without changing how an imported code from a package functions within the context of this new language.

There were many attempts to create such solution in the past. These attempts are discussed at great detail in Chapter 5 (Related Work). Two of the most significant attempts are CoffeeScript and TypeScript. TypeScript, however, cannot fix certain run-time errors, and while CoffeeScript comes close

[15]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

to my own work, my work arguably provides more features than CoffeeScript does. In summary, while these attempts solve parts of the problem, the problem has not been solved in full.

1.5 Thesis statement

Thesis. *It is possible to create a programming language which is interoperable with JavaScript. The requirements for this language are:*

a. Consistency—one way of doing things;

b. Compatibility—the language will be compatible with household frameworks in JavaScript; and

c. Ease of use—JavaScript programmers, whether beginners or advanced ones, will be able to easily master it.

1.6 Chapters supporting proposition

This proposition will be supported by following chapters: Chapter 2 contains an overview of all features of this language, which support the proposed requirements, and goes over syntactic structure of my proposed language. Chapter 3 contains a description of technologies used during development of this proposed language, and description of technical details on compile process, as well as a detailed description of future work. Chapter 4 goes over the testing done to ensure correctness and to measure performance of this language. Chapter 5 dives into similar projects in the past—how they work, what they accomplish and where they may be insufficient. I conclude with Chapter 6, that summarizes what was accomplished in this thesis and contains an overview of future work.

Chapter 2

Design

This chapter critiques specific features of JavaScript and provides counter-proposals to them. Together, these counter-proposals produce a programming language that fits the requirements from set out in the thesis statement. Besides critiquing JavaScript features, I also describe high-level design of my proposed solution—the language JonScript. This chapter also discusses JonScript syntax structure and semantics in concrete ways and showcases examples of how the language works.

2.1 Feature analysis

JavaScript is famous for its problematic idiosyncratic behaviour. In this section I list them and provide concrete improvement proposals. Furthermore, I show that these improvements are useful in improving developer and end user experience.

2.1.1 Static typing

General concept Static types [25] are present in programming languages to validate data structures, prevent application errors during run-time by not allowing operations that violate the type systems rules, and to allow compile-time optimization.

JavaScript implementation JavaScript does not have static typing. Instead, you can check the type of any data structure during run time.[16] This is called dynamic typing. You cannot, for example, restrict what data structure will be passed into a function as a parameter, or to be assigned to any variable. If you need to enforce types, you must do it dynamically, at run time.

[16]`https://www.w3schools.com/js/js_datatypes.asp`

Problematic behaviour occurs when a program wrongly assumes the type of variable, property, or function parameter. This causes errors during run time, that are hard to detect, because of the variety of data structures that can be assigned to any variable or property, or passed as a function parameter. Here is an example of this behavior. We try to create a function called `add`. It takes two parameters and adds them together with `+`. Then, we call this function with unexpected parameters and discuss the result.

```
const add = (a, b) => a + b;
add(5, 4); // returns 9
add("5", 4); // returns "54"!
```

The result of `add("5", 4)` is `"54"`. This is due to the `+` operator acting as concatenation operation instead of addition, because it dispatches on a string argument. This is preventable by checking the types of parameters entering the function and raising an exception[17] if they are not numbers.

Source of behaviour Without a static type system, JavaScript is more accessible to new developers.[18] Not having to specify what value is stored within variables, properties or arguments also makes the language less verbose. This also makes it easier for JavaScript developers to duck-type.[19] Duck typing can be used to avoid excessive conditional statements.[20]

Consequences Due to the lack of static typing, it is more difficult to predict the behaviour of JavaScript code. This makes it harder for JavaScript developers to create complex applications and libraries, and results in bugs affecting the end user [38].

Solution JonScript contains a subset of TypeScript typing and is compatible with TypeScript packages. This means that you can define typed methods on classes and use a certain amount of type coercion known from TypeScript, through user-defined type guards [14]. This typing is essential for many other features discussed below, in Section 2.1.4, or Section 2.1.2.

[17] https://www.w3schools.com/js/js_errors.asp
[18] https://fosterv222.medium.com/coding-languages-typed-vs-untyped-d29c7e0b3713
[19] https://hackernoon.com/learning-duck-typing-in-javascript-qa3g35nc
[20] http://adripofjavascript.com/blog/drips/using-duck-typing-to-avoid-conditionals-in-javascript.html

2.1.2 Function contexts

General concept Function context[21] is understood as the data structure from which a method is being called. Function context is very similar to the keyword `this` in C#. In C#, the `this` keyword[22] refers to the current instance of a class.

JavaScript implementation Each data structure carries its own context. That is when a method is called on an object, you can access the object properties within said method by using `this` keyword. In JavaScript, you can define a function using `this` without explicitly stating to which object `this` refers to. This means that you can re-use the same function as a method for multiple objects.

Problematic behaviour Function context can be accidentally changed during run time by passing a method as a parameter, or by assigning a method to a variable, to the context of the data structure in which the aforementioned method is executed. This behavior depends on which kind of method is used. If a method is defined as arrow function expression,[23] and not a standard function, its context is set forever during declaration, to the data structure in which it was declared. If however, an arrow function expression was not used to define a method, an unexpected behavior can occurs when you call this method. Here is an example, where passing a method as a function parameter leads to an exception. In this example, I wrote a function that takes another function as parameter (`funct`) and then tries to execute the function and convert and return the resulting value as a number.

```
const convertToNumber = funct => Number(funct());
convertToNumber((2).valueOf); // throws an exception
```

Source of behaviour Function context behaviour in JavaScript increases code re-usability by allowing you to define a method for multiple classes (or objects) and then simply assign it to one of their properties—and utilize their context within said function. The reason for this behaviour is that originally, JavaScript did not have the syntax for classes that contain methods [47]. The usage of `this` keyword allowed for a top-level definition of methods.

[21]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this
[22]https://docs.microsoft.com/cs-cz/dotnet/csharp/language-reference/keywords/this
[23]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Consequences This behaviour lessens JavaScript potential to be used as a functional programming language—by not allowing certain methods to be, safely, used as first-class citizen functions, JavaScript not only increases the amount of potential errors a programmer can make, due to context changes, but also makes the programmer use either a wrapper function—wrapping a method in an arrow function expression (which came in the ECMAScript version 6 (ES6) [50] version of the language—therefore, the context re-assignment problem partially stems from backwards compatibility) to pass it safely, or, use the `bind` method present on each function to define its context and prevent this problem. Alternatively, you can use closures to this end as well[24] and assign the desired context to a separate variable (usually `self`, or `that` and use that variable instead of `this`. All of these solutions, except the arrow function expression, bloat the code however.

Solution JonScript stops context reassignment from happening. If the same example was created in JonScript, the `valueOf` methods function context would be automatically bound to the number `(2)`. This automatic binding effectively makes all functions in JonScript behave like arrow function expressions—when any method is accessed on an object, this binding takes place.

Here is a rough approximation of what JonScript does when you try to define a variable to which a method is assigned:

```
const convertToNumber = funct => Number(funct());
convertToNumber(
    (() => const _e = (2); return _e.valueOf.bind(_e))
); // Safely binds the method to 2
```

2.1.3 Multiple programming paradigms

General concept Functional programming [49] is one of modern programming paradigms, which focuses on creating a software by using pure functions. This means there is no shared state, nor are there function side-effects. This style of programming is declarative. You use a function definition to declare what do you want to do—the implementation details are then hidden within the functions that are passed as parameters. This style of programming is closely related to the function as first-class citizen principle. When a function is a first class citizen, it can be used in the same fashion as any other data structure—as a parameter, a return value or as value assigned to a variable.

[24]`https://salesforce.stackexchange.com/questions/159399/what-is-this-pattern-for-self-this`

JavaScript implementation JavaScript implements the function as first-class citizen principle—in JavaScript, you can use function as you would any other data structure. Functions even have their own properties and methods. Given these facts, you could argue that JavaScript functions are in fact functors—objects[25] that have an overloaded application operator (`()`). JavaScript also fully supports imperative and object-oriented[26] styles of programming. This means that it is equally possible to write JavaScript code utilizing multiple different approaches.

Problematic behaviour Giving programmers total freedom of using multiple different paradigms together without limitations causes the JavaScript developer to essentially develop their own dialect of the language—using a subset of possible programming styles.[27] This causes difficulty when figuring out the correct approach to developing complex applications in JavaScript.

Source of behaviour Originally, the author company of JavaScript, Netscape, wanted JavaScript to be simple, dynamically typed object-oriented language [47], similar in syntax to C. However, the author, Brendan Eich found LISP-like first class citizen functions to be attractive. Due to the multitude of developers working their own ideas into the language (and pressure from Netscape), JavaScript ended up with multiple programming paradigms.

Consequences Because of the nigh-unlimited number of possible ways of writing code, JavaScript developers working in a team frequently resort to using linters—tools [43] designed to flag code that, while correct and functional, does not conform to the standards set by the team.

Solution By removing imperative programming from JonScript, the language gains larger resemblance to functional programming languages like LISP, which helps programmers lean more on functional programming features [23].

However, JonScript is not a purely functional language—you can define both classes and objects. Both can have public and private properties, and can inherit from each other. This makes JonScript a highly functional, object-oriented language. With this combination, JonScript offers something both to fans of functional style, and to everyone else that likes object-oriented approach, with classes and encapsulation. By enforcing a single, consistent programming style within JonScript, we avoid the multiple-paradigm issue.

[25]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions`
[26]`https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_JS`
[27]`https://javascript.plainenglish.io/javascript-functional-vs-oop-fb5fbf15a35d`

2.1.4 Quasi prototypal inheritance

General concept A class declaration is a blueprint for creating objects. An object of a class is called an instance [45]. The instance of a class has properties and methods defined within the class declaration. Class definitions also provide encapsulation—they tell the object instance which properties should be public—accessible by other instances—and which should be private, or protected—only available to select class instances. Classes can inherit properties and methods from a parent class. This means that a class instance can have methods and properties that are defined in the parent class. An instance of a class is also an instance of its parent. There are two kinds of inheritance systems, prototypal and class based, with significant differences in semantics.

In typical class based inheritance systems, a class cannot remove, or redefine, parent methods and properties—unless the language allows you to declare them as `abstract`,[28] or `virtual`.[29] Also, in typical class based systems, classes can have only one parent class (such as Java or C#). In general, class definitions are created during compile-time—and stay static during the entire run of the program.

On the other hand, prototypal inheritance [41] means that a class definition is simply a function that returns an object. Classes within prototypal inheritance are called prototypes. Prototypes implement inheritance by simply composing objects together within the prototype. This means that a prototype may have multiple parents and can redefine all inherited properties and methods. In general, prototypes are created during run time and they are mutable during run time. One important benefit of the prototypal inheritance is that it solves the fragile base class problem [31].

JavaScript implementation JavaScript uses its own kind of prototype-based inheritance [9]. A class in JavaScript is special kind of function, returning an object. Each of those objects contains information about its prototype.[30] This is what makes the class function different from a regular one. When calling a method on any object, JavaScript looks at the direct properties of the object first (which can be added during run-time). If if does not find anything, it finds said objects prototype and tries to find the method there. If the method is not found, it continues in a similar fashion onto the parent of the prototype. Prototypes in JavaScript cannot use multiple inheritance.

Problematic behaviour Most experienced developers who arrive to JavaScript coming from other modern languages are not expecting a proto-

[28] `https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html`

[29] `https://www.infoworld.com/article/2895408/exploring-virtual-and-abstract-methods-in-c.html`

[30] `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes`

typal inheritance system.[31] This effect is compounded by JavaScript adding a new keyword in ES6—`class`. ES6 JavaScript classes look as if they were written in Java.[32] Therefore, most developers assume that JavaScript uses a class-based inheritance system. This leads to unexpected and counterintuitive behaviour, such as expecting the class definitions to be immutable, objects not to be able to become instances of completely different class, or that class definitions cannot have dynamically generated properties.

Source of behaviour JavaScript tries to be as inviting as possible to new developers coming to it from other languages—we speculate that that is why the `class` keyword was added.[33] Furthermore, JavaScript prototypal inheritance allows for a great flexibility—by changing existing class definitions you can customize behaviour of any API you want to a much greater extent than you would be able to do otherwise.

Consequences Having a less-known inheritance system and compounding it with its own specific implementation makes JavaScript inheritance counterintuitive. This is further compounded by the ES6 added syntax. This results in developers working with a system they do not properly understand, which leads to developers under-utilizing the inheritance system and to unexpected behaviour of JavaScript applications.

Solution JonScript offers a true version of prototype-based inheritance, with multiple parents. In addition, JonScript also completely removes the keyword `this` from the language—this is useful, since it shortens property access. Furthermore, JonScript prototype-based inheritance is compatible with original JavaScript one.

2.1.5 Forgiveness

General concept In this thesis, language forgiveness is defined as a feature that applies transformations to values to prevent errors and to cause code that would have been erroreous to execute with assumed intended semantics. An example of language forgiveness would be an interpolated string in C#.[34] In C#, you can concatenate numbers with strings within interpolated strings and even though string and number types are different, C# automatically converts the number in question to its string representation. An example of

[31]`https://www.toptal.com/javascript/es6-class-chaos-keeps-js-developer-up`
[32]`https://everyday.codes/javascript/please-stop-using-classes-in-javascript/`
[33]`https://www.digitalocean.com/community/tutorials/understanding-classes-in-javascript`
[34]`https://docs.microsoft.com/cs-cz/dotnet/csharp/language-reference/tokens/interpolated`

unforgiving behaviour would be, in C#,[35] a division by zero—this would result in an exception being thrown and the run-time possibly being interrupted, if such exception is not caught.

JavaScript implementation JavaScript has forgiving arithmetic, logical and relational operators.[36] Unless you change the default behaviour of Object, Number or String classes by overriding standard API methods, or change the prototype of their class instances during run-time, these operators never throw exceptions—they accept every kind of data structure within the language. JavaScript is not forgiving when it comes to accessing properties on null references. If you try to access a property on a null value, the operation will result in an exception.[37] Similarly, JavaScript is not forgiving when you try to call a data structure that is not a function.[38] In ES2020 [2], JavaScript added the optional chaining operator `?.`.[39] This operator can be used to access properties on an object, items within an array of call a function and will not result in an exception if the value in question is a null reference. Instead, the operation returns an empty result (`undefined`). This operator will not protect you from an exception being thrown when calling a data structure that is not a function, and you cannot use this operator when instantiating classes.

Problematic behaviour This kind of partial forgiveness—throwing an error when accessing properties on the `null` (or `undefined`) value—is problematic when working within complex data structures. For example, when you try to access a property on a complex data structure that a server returned as a response to your request, you might only care whether this particular property exists within the structure or not—if this is the case, your application crashing when trying to access this property is not desirable behaviour.

Here is an example of how you may have to do an operation without exceptions being thrown: Lets assume that the server can return either `{ a: { b: c: { d: 5 } } }` structure, or `{ a: 1 }`. We only care if property `d` exists within the structure (named `response`).

[35] `https://docs.microsoft.com/cs-cz/dotnet/api/system.dividebyzeroexception?view=net-5.0`

[36] `http://speakingjs.com/es5/ch09.html`

[37] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors/No_properties`

[38] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors/Not_a_function`

[39] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining`

```
if (response != null
    && response.a.b != null
    && response.a.b.c != null
    && response.a.b.c.d != null) {
    // do stuff
}
```

This way we can always make sure that our program will not crash on null reference exception. However, for each property that may not be defined, we need an extra null check. This will bloat our code. There is a better way of doing this with the aforementioned optional chaining operator:

```
if (response.a?.b?.c?.d != null) {
    // do stuff
}
```

However, the problem with optional chaining operator is that we can forget to use it. Programmers are human after all, and constantly remembering to annotate every property access that may lead to a null reference exception can be difficult and prone to errors.

Source of behaviour By having null reference exceptions, JavaScript conforms to a well-known standard behaviour that most developers who have used other programming languages are familiar with.[40] This may help developers orient themselves better when working in JavaScript. Null reference exception tells you that you have tried to access something that does not exist. The reasoning is similar to throwing an exception when calling a data structure that is not a function. This behaviour also helps to prevent typos within your code.

Consequences Incomplete forgiveness in JavaScript leads to a website crashing on null reference exception. If uncaught, these exceptions can completely interrupt the user progressing through a website. Similar situation occurs when you try and call a data structure that is not a function.

Solution JonScript uses automatic optional chaining whenever you access an item within an array, a property within an object, or try to call a method, or a function. JonScript is also forgiving when it comes to arithmetic, logical operators and comparative operators: no operation will cause a crash, but if the operation would result in an invalid, or unexpected value, it simply returns a null value (`undefined`). More on operators in Section 2.1.11.

[40] `https://www.geeksforgeeks.org/null-pointer-exception-in-java/`

2.1.6 String literals

General concept A string is a sequence of characters. Most object-oriented languages represent string by a String class.[41] This class encapsulates the underlying sequence of characters and adds useful methods for manipulating text within the programming language. In modern programming object-oriented languages, such as C#,[42] Java,[43] or Rust,[44] you instantiate a string class instance by using the `"Text here"` text within quotes.

JavaScript implementation JavaScript has three kinds of strings:[45]

1. string surrounded by double quotes—`"string"`,
2. string surrounded by single quotes—`'string'`, and
3. a string surrounded by backticks (grave accents)—`` `string` ``.

The first two, single, and double quoted strings, only differ in that they escape each other quotes—therefore, `"'"` and `'"'` are both valid strings in JavaScript. The third kind, `` `string` ``[46] (added in ES6) is template literal string. This kind of string supports string interpolation (allows embedded expressions) and is a multi-line string—the single and double quoted strings can only be multi-line if you escape the newlines present within them by a backslash `\`.

Problematic behaviour In modern popular programming languages, such as C#[47] or Java,[48] single quotes mean that a value is a `char`. A `char` is a single character value. JavaScript does not have `char`. This causes a counterintuitive behaviour, where developers may assume that `'c'` is a character and will treat is as such—this can lead to unexpected behaviour, especially when using the `+` operator—a C# developer, new to JavaScript, may assume that a `+` operation on `'c'` would result in a new character—but instead, it simply concatenates `'c'` with the value passed into the `+` operation.

[41] https://realpython.com/oop-in-python-vs-java/
[42] https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/strings
[43] https://www.geeksforgeeks.org/strings-in-java
[44] https://www.geeksforgeeks.org/r-strings
[45] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String
[46] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals
[47] https://docs.microsoft.com/cs-cz/dotnet/csharp/language-reference/builtin-types/char
[48] https://www.tutorialspoint.com/java/java_characters.htm

Source of behaviour The single and double quoted strings in JavaScript exist due to preference—if you are, working with a string containing HTML code with element attributes, such as `'<p align="right"></p>'`[49] it is better to use single quoted strings to avoid having to escape the double quotes in HTML. However, if you are, for example, working with text representation that contains single quotes as contractions in English, such as `"I'm"`, it is better to use double quotes and avoid additional escapes.

Consequences JavaScript developers can become confused by the unusual variety of quotes around strings JavaScript offers. Even if the developers do not assume that single quotes mean the `char` type, this confusion can result in inconsistent usage of quotes within application code. While this seems innocuous, the fact that JavaScript projects can have linters [5] set up in a way that disallows programmers to use these strings interchangeably suggests that the inconsistent usage of quotes around strings is perceived as a problem. [50]

Solution JonScript only uses template strings and double quoted strings, thereby avoiding the confusing differences between single and double quoted strings. There is no difference between JonScript double quoted string and JavaScript double quoted strings.

2.1.7 `null` and `undefined`

General concept A null value in most modern object-oriented programming languages (like C#)[51] is a value on a property of a data structure that indicated such property is empty (has no value). This value in most modern programming languages has its own special type (like C#, or Java).[52] When trying to access a property on a null value, either one of these three things happens:

1. the application crashes, (in C),
2. the operation throws an exception (in Java or C#), or
3. the operation uses type coercion to another value (SQL).[53]

[49] `https://www.w3docs.com/snippets/javascript/when-to-use-double-or-single-quotes-in-javascript.html`
[50] `https://eslint.org/docs/rules/quotes`
[51] `https://docs.microsoft.com/cs-cz/dotnet/csharp/language-reference/keywords/null`
[52] `https://www.javatpoint.com/null-keyword-in-java`
[53] `https://www.tutorialspoint.com/sql/sql-null-values.htm`

JavaScript implementation JavaScript has two null-like values, `null`[54] and `undefined`.[55] The `null` value has the same type as an object, evaluates within logical conditions as false-like, and within arithmetic operations as zero. On the other hand, the `undefined` has its own run-time type, but still evaluates as false-like. However, if you use `undefined` within arithmetic operations, the resulting value is `NaN`.[56] Both of these values are evaluated as equal[57] when using the `==` operator, but will be unequal when using the `===` operator. They also have different semantic meaning—`null` means that a property value, variable or parameter is empty—a value may be set in its place later on (or was deleted earlier), while `undefined` denotes that a property does not exist, or that a value was declared, but not yet assigned to. When using comparison operators, such as greater-than, or less-than on these two values, the result is always false.

Problematic behaviour Can be shown by defining a function called `arrayify` that takes a single parameter `aValue` and returns an array containing this single parameter. This function simply returns the value passed into it as an element within an array containing only said value passed into the function. By doing this with both `null` and `undefined` I showcase the problematic behaviour of having two slightly different null-like values. When this function receives `undefined` as a parameter, it returns an array withotu any elements. However, if it receives `null`, it returns an array with a single item inside: the `null` value.

```
const arrayify = aValue => new Array(aValue);
arrayify(undefined); // results in an empty array
arrayify(null); // results in an array with null inside
```

This is due to the fact that a function, or a constructor will treat an undefined parameter as if it was no there. While this is consistent with the meaning of `null` an `undefined`, it is not very intuitive for programmers who are used to `null` values from C#, Java, or other modern programming languages.

[54] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null

[55] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined

[56] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/NaN

[57] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

Source of behaviour According to the author of JavaScript, this behaviour was a design mistake.[58] This mistake was not fixed from the language due to backwards compatibility concerns.

Consequences JavaScript has two different null-like values, each with their own set of subtle semantic differences. This creates counterintuitive behaviour when trying to check for null-like values—such as having to check for two possible values instead of one.

```
const check = aValue => aValue !== null && aValue !== undefined;
```

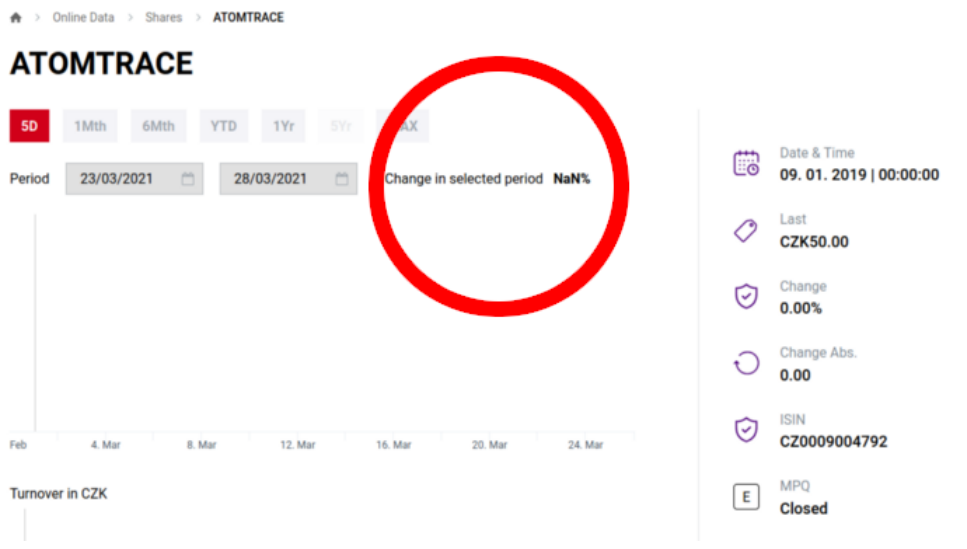
Solution JonScript combines these two values into a single entity, called `nil`. This entity behaves the same as JavaScript value `undefined`.

2.1.8 `NaN`

General concept According to the IEEE 754 standard,[59] `NaN` is a value denoting that a certain result is not a numeric result. This special value is the result of an invalid operation, such as dividing zero by zero, or by dividing an infinity with an infinity. This value is never equal, greater, lesser, greater or equal, lesser or equal than any other value except for itself. This value is unequal to every other value within the standard, including itself. When used in arithmetic operators, this value consumes all other values—`1 + NaN` equals `NaN`, `1 - NaN` equals `NaN` and so on.

JavaScript implementation JavaScript uses the value `NaN` when dividing zero by zero, infinity by infinity, or when a function that is supposed to return a numerical result receives invalid input. The value `NaN` is not a numeric primitive—instead, it is a variable in the global scope. Similar to zero, this value is treated as false-like within logical conditions and behaves like the IEEE 754 standard when placed within comparison operators. However, when concatenating the value `NaN` with a string, JavaScript coerces the value `NaN` into a string form—`"NaN"`. Therefore, `NaN + "a"` results in the string `"NaNa"`. JavaScript also considers the value `NaN` to have run-time type of number, just like any other numerical value. If you want to find out whether a value is `NaN` or not, you can use either the function `isNaN`, or the function `Object.is`. The former will try to coerce the value to a number, so `isNaN("dog")` as well as `isNaN(NaN)` will evaluate as true, and the latter can be used like this, when checking the value of `aValue`:

[58] https://twitter.com/brendaneich/status/1140668264109891590
[59] https://ieeexplore.ieee.org/document/8766229



Period 23/03/2021 28/03/2021

Change in selected period NaN%

Date & Time 09. 01. 2019 | 00:00:00

Last CZK50.00

Change 0.00%

Change Abs. 0.00

ISIN CZ0009004792

MPQ Closed

Figure 2.1: Prague Stock Exchange displaying NaN% when looking up specific time horizon for stocks

```
const aValue = 0 / 0; // results in NaN
Object.is(aValue, NaN); // results in true
```

Problematic behaviour Since `NaN` can be displayed as `"NaN"` on a website, if you use `NaN` within concatenation operation, this can create confusing artefacts in your application for the end user. While this may seem like an issue that is easy to solve, there are professional websites still struggling with displaying `NaN` correctly. A concrete example of this behaviour would be the Prague Stock Exchange.[60]

Furthermore, JavaScript considering the `NaN` to be a number can cause problems if you try to do a run-time type check on a value. Since you need to do a special check for this numeric value, it is easy to forget and therefore very easy to incorrectly sanitize function parameters, if you do not want your function to work with `NaN` as if it was a regular number.

Source of behaviour The concatenation behaviour of `NaN` is due to `NaN.toString()` resulting in `"NaN"`. This is similar to the behaviour of other languages which contain the value `NaN`, like C++[61] and Python.[62] The nu-

[60]`https://pse.cz`
[61]`http://www.cplusplus.com/reference/cmath/nan-function/`
[62]`https://towardsdatascience.com/5-methods-to-check-for-nan-values-in-in-python-3f21ddd17eed`

meric run time typing of `NaN` as a number also makes sense, given that `NaN` is part of the IEEE 754. When working with an infinite value, which is also a part of this standard, its run time type is also a number.

Consequences Given the difficulty of working with `NaN`, numerical operations in JavaScript frequently yield problematic behaviour on websites which may confuse clients. Most users do not recognize the text `"NaN"` as a numerical value and will instead be confused by such values displayed. Also, while `NaN` not being equal to itself is a part of the IEEE 754 standard, this is not common knowledge amongst developers.[63] This causes further errors in JavaScript development. The author of JavaScript himself makes jokes about this behaviour.[64]

Solution Similar to how `null` is treated in JonScript, JonScript simply converts any `NaN` value that results from an operation into `undefined`. If any operation returns an object that contains `NaN` within its properties, JonScript will also convert this value to `undefined` when it is being accessed—this does not however change the object in question—the accessing operation merely gives you `undefined` instead of `NaN`. Interestingly, JavaScript already does a very similar thing (except it converts `NaN` into `null` and not `undefined`) when parsing an object into its string JavaScript Standard Object Notation (JSON)[65] representation. Any property that has the value `NaN` will be converted into a property that contains `null`.[66]

2.1.9 Null as property

General concept In many modern programming languages, like C#, there is a distinction between accessing a property that does not exist—in C#, this throws an exception,[67] and accessing a property with null value. The latter usually simply yields the value null. What about removing properties? In most modern, statically typed programming languages there is no way of removing a property, or a method from an object—you can only assign a null value to a property.

[63] http://adripofjavascript.com/blog/drips/the-problem-with-testing-for-nan-in-javascript.html

[64] https://twitter.com/brendaneich/status/819360853476507649

[65] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

[66] https://levelup.gitconnected.com/manipulating-json-strings-in-javascript-5c9423841fa3

[67] https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/cs1061

JavaScript implementation In JavaScript, you can access non-existent properties on an object without an exception being raised. There are three operations that act as a way of removing a property from an object:

- by assigning the value `null`,
- by assigning the value `undefined`, or
- by using the `delete`[68] operator.

The differences between `null` and undefined were already discussed above in Section 2.1.7. The `delete` operator does something different—it remove the property from an object completely. While a removed (or nonexistent) property will still yield `undefined` when accessed, there are differences between a property that has the value of `undefined` and a property that does not exist. The JavaScript standard API treats them differently—any function, whose job is to enumerate properties on an object (like `Object.keys`,[69] `Object.values`[70] and `Object.entries`[71]) will treat a property that has `undefined` assigned to it as it it was still a part of the object—this means, for example, that `Object.keys`, which returns an array of strings for each property of an object will return a string key for a property that has the value of `undefined`.

Problematic behaviour Since there are three different ways of removing a property from an object with varying effects, the resulting behaviour is counterintuitive. Consider an example, where I try to remove a property from an object named `object`:

```
const object = {
    property: 1,
};
object.property = undefined;
object.property === undefined; // is true! but:
// hasOwnProperty is a method on Object.prototype
object.hasOwnProperty("property") // is true as well!
```

[68]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete`
[69]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys`
[70]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values`
[71]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries`

Source of behaviour Since JavaScript objects are similar to hash-maps (except for behaviour related to prototypes), the behaviour of property removal as discussed above is a logical consequence of previously stated facts and is consistent with other operations within the language. The `delete` operator then acts as a way of doing additional operations that you simply are not able to do through simple assignment.

Consequences Having three subtly different ways of doing the same thing is counterintuitive, since it makes it difficult to decide which of these ways is appropriate given the circumstances. This negatively affects beginner developers, and developers coming to JavaScript from other languages that have no such concept as removing properties from an object. If these developers choose to either assign `undefined` or `null` to a property as a way of removing it, they can encounter null reference exceptions when iterating over object properties.

Solution When assigning `undefined` value to an object or a class, JonScript automatically deletes the property to which this value was assigned. This does not affect compile-time typing within the language in any way. Also, if you are using methods `Object.keys`, `Object.values` or `Object.entries` on an object that contains `null`, `undefined`, or `NaN` properties, these methods will act as if those properties were not present on that object. This behaviour is important when working with objects from imported packages, or from the standard JavaScript API. This functionality will be extended to cover the rest of JavaScript standard API in future versions of JonScript.

2.1.10 Pattern matching and type-checking

General concept Pattern matching [46] is the act of checking a data structure against a pattern. This pattern is made up of values that the data structure constituents are compared against. The match is always exact—the data structure either matches the pattern, or it does not. In this thesis, run-time type checking is considered as a way of matching an object against a pattern consisting of a type. A match occurs when an object is an instance of said type.

JavaScript implementation JavaScript has three ways of checking a data structure against a pattern (or type):

- the `switch` statement,
- the `typeof` operator,
- the `RegExp` class, and
- the `instanceof` operator.

The switch statement is the same in JavaScript as it is in other modern object-oriented programming languages, such as Java, and will not coerce the values used within itself. The `RegExp`[72] class provides regular expressions for pattern matching within a string. The `typeof`[73] operator is an infix operator that, when applied on a value, returns a string based on the type of value passed into it. This operator can return these seven values:

1. `"number"` for numbers,
2. `"string"` for strings,
3. `"object"` for objects, arrays, class instances or null,
4. `"undefined"` for `undefined`,
5. `"function"` for functions,
6. `"bigint"` for a special kind of large number without floating point,[74]
7. `"symbol"` for symbols.[75]

The other operator for determining the type of a value, `instanceof`[76] is a binary comparison operator that returns a boolean value. Left hand side should contain an object you are trying to match with a class definition that is on the right hand side. This operator will try to determine whether or not a value is an instance of a prototype. The operator will also consider inheritance, so a child class instance will match to its parent class definition using `instanceof`. This operator will only work on functions and objects—values, to which the

[72] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp
[73] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof
[74] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt
[75] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol
[76] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof

`typeof` operator returns either `"object"`, or `"function"`, with the exception of `null`, to which `typeof` returns `"object"`, but the `instanceof` operator always returns false on comparison. If you pass a primitive string, number, boolean, symbol, bigint value into `instanceof` and try to compare them with String, Number Boolean class definitions, or the object BigInt, or the object Symbol, the result will always be false.

Problematic behaviour JavaScript does not have a general-purpose pattern matching operator, despite its functional programming support—an operator similar to `match` in Scala [34] would improve JavaScript capacity to be more verbose and create abstractions more easily.

Another problematic part of JavaScript related pattern matching comes to play when trying to determine the type of an object—an important feature in a language without static typing. Consider an example, where I write a function `checkInstance` that checks a parameter `aValue` against the parameter `aPrototype` to see if `aValue` is an instance of `aPrototype`, regardles of whether `aValue` is an object or a value:

```javascript
const checkInstance = (aValue, aPrototype) => {
    // this is not enough
    if (aValue instanceof aPrototype) {
        return true;
    }
    // check primitive strings
    if (typeof aValue === "string"
        && aPrototype === String) {
        return true;
    }
    // check primitive numbers
    if (typeof aValue === "number"
        && aPrototype === Number) {
        return true;
    }
    // and so on...
};
```

As you can see with this example, creating a custom function to simply check the basic language types in JavaScript is arguably a long and verbose process. This should not be the case, as most modern programming languages can do similar operations more easily—for example, the `is` operator in C#.[77]

Source of behaviour The lack of a general-purpose pattern matching operator is something to be expected, since the inspiration for JavaScript during

[77]https://docs.microsoft.com/cs-cz/dotnet/csharp/language-reference/operators/type-testing-and-cast

its creation was Java, which does not have such an operator. This absence may be ameliorated with the ECMAScript pattern matching proposal.[78]

The differences between `typeof` and `instanceof` are due to the differences between primitive values of string, number or boolean and their class counterparts. For example, a String class[79] will behave differently than a string primitive value when the operator `===` is used to compare it. Furthermore, you can define additional properties on a String class instance during run-time, as opposed to a string primitive. Therefore, it makes sense to have operators that distinguish between the two.

Consequences Given the state of things in JavaScript regarding pattern matching, if JavaScript developers want to add pattern matching algorithms, they have to import packages to help them.[80] Furthermore, given the counter-intuitive nature of `typeof` and `instanceof` type-checking mechanism, developers are faced with additional challenges when they want to correctly check a type of variable, parameter or an object property.

Solution JonScript offers an improvement on the operator `typeof` and `instanceof`—by offering a general-purpose pattern matching operator. The `is` operator correctly matches both primitive and class instances as strings, numbers, boolean values etc. But that is not all—`is` also offers pattern matching that is found in other programming languages—for example, it can match any object onto an object pattern, similar to Scala. When using `is` JonScript uses user-defined type guards that TypeScript offers[81] for conditional type coercion.

2.1.11 Operators and operator overloading

General concept Operators[82] are language concepts designed to perform a specific arithmetic, logical, or relational operation. They are similar to function calls, but differ syntactically and in their semantics from functions or methods in any specific language. Operator overloading is an act of defining a method that a compiler, or an interpreter will utilize instead of standard operator behaviour on a specific instance of class in order to perform a custom action when an operator is applied to such instance.

[78] `https://github.com/tc39/proposal-pattern-matching`
[79] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/String`
[80] `https://www.npmjs.com/package/ts-pattern`
[81] `https://www.typescriptlang.org/docs/handbook/advanced-types.html`
[82] `https://brainly.in/question/26216828`

JavaScript implementation JavaScript supports all logical, arithmetic and relational operators found in Java.[83] Furthermore, it adds several additional operators.[84] In addition to these, for the purposes of this section, object property access and function call will be considered as operators. JavaScript supports a limited amount of operator overloading. You can overload the property access operators `.` and `[]` by defining getters and setters on object properties, or by creating an object proxy through the Proxy class.[85] The proxy class allows you to control any access to any of the objects properties and allows you to override the default behaviour. Furthermore, you can overload the `()` operator by creating a functor—in this sense a functor is simply a function which had additional properties defined during run-time. And lastly, you can override the arithmetic operators—but only to an extent—by redefining the `valueOf`[86] method within a class declaration. This will not change how the operator acts by itself, but it does allow you to create custom representation of your class. This method also will not allow you to specify the return type of such operation—it merely affects how a class will behave when coerced into a string or a number.

Problematic behaviour Operator forgiveness as implemented within JavaScript leads to counterintuitive behaviour. This is due to type coercion. There are many examples of this behaviour [8], but here is an example of one of the arguably worst: if you compare two objects, called `a` and `b` that are unequal to each other (`a != b` is true), one will never be less than (`a < b` is false), nor greater than (`a > b` is false) the other. However, both of these structures will be less or equal to each other (`a <= b` is true), and greater or equal to each other (`a >= b` is also true). This behaviour is due to JavaScript operators `<`, `>`, `>=` and `<=` using type coercion by converting `a` and `b` structures into strings. This coercion results in both `a` and `b` being compared as the string `"[object Object]"` (by conversion to a Primitive [4]). Since both string version of `a` and string version of `b` are equal to each other, `a >= b` and `a <= b` are true, but `a > b` and `a < b` are false.

Furthermore, the limited amount of operator overloading hobbles JavaScript developers if they wish to create classes for managing currencies, matrices or complex numbers—since true arithmetic operator overloading would nicely encapsulate operations for these classes. Lack of overloading affects the comparison operators as well in a similar way.

The behaviour of equality operators, `==` and `===`, is counterintuitive as well. Consider this example with the equality operator `==`:

[83] https://www.tutorialspoint.com/java/java_basic_operators.htm
[84] https://www.w3schools.com/jsref/jsref_operators.asp
[85] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy
[86] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/valueOf

```
[] == 0 // is true
0 == "0" // is also true
[] == "0" // is not true
```

Therefore, the equality operator `==` is not transitive. Transitivity is meant to be provided by the `===` operator. While triple equals is transitive, consider this example:

```
// is not true
new String("Hello") === new String("Hello")
 // is true
new String("Hello") == new String("Hello")
```

The triple equals operator employs no coercion when it comes to objects—instead, it only compares their references, even in places where a developer experienced in modern programming language, such as Java[87] or C#[88] would expect a value-based comparison.

Source of behaviour Operator forgiveness works based on type coercion[89] that allows JavaScript to not crash upon invalid operations. Type coercion in JavaScript always tries to give the best approximate result to what a programmer meant. This means that even if at all possible, website will always display at least somewhat correct content. While JavaScript does not yet have complete operator overloading, there is a pending proposal for this feature.[90] However, the author of JavaScript himself called the operator type coercion ”insane”.[91]

Consequences The type coercion system is confusing, unwieldy and frequently causes difficult situations for developers.[92]

Solution JonScript gives you a smaller set of arithmetic and comparison operators that utilize less type coercion. First, I will showcase how arithmetic operators work within the language based on parameter types in Table 2.2. Each operation listed in the table is symmetric: If you use a string class with a string primitive (or class), you end up with a class value. Also, when using `+`

[87]`https://www.javatpoint.com/string-comparison-in-java`
[88]`https://www.tutorialsteacher.com/articles/compare-strings-in-csharp`
[89]`https://developer.mozilla.org/en-US/docs/Glossary/Type_coercion`
[90]`https://github.com/tc39/proposal-operator-overloading`
[91]`https://twitter.com/brendaneich/status/1053029515968970754`
[92]`http://mauricio.github.io/javascript-from-hell/#/`

Operator	Left hand side	Right hand side	Type of result
+	string	string	string
+	string	String class	String class
+	string	Number class	string
+	string	number	string
+	String class	String class	String class
+	string	undefined	string
+	String class	undefined	String class
+	Number class	Number class	Number class
+	number	Number class	Number class
+	number	number	number
-	number	number	number
-	number	Number class	Number class
*	number	number	number
*	number	Number class	Number class
/	number	number	number
/	number	Number class	Number class
%	number	number	number
%	number	Number class	Number class

Figure 2.2: Binary arithmetic operators in JonScript

for concatenation between a string, or String class and `undefined`, the result will be the string, or String class instance that the operator was run on.

Relational operators are simplified There is no triple equals. Instead, double equals works as triple equals—but with one important caveat: it employs coercion that makes it able to compare string classes with string primitives—same goes for numbers and their class counterparts. Other comparative operators will only return `true` when comparing numbers with numbers (or their class counterparts), or with strings and strings (or their class counterparts).

JonScript also contains the `await`[93] unary operator, which resolves asynchronous calls[94] and ternary conditions[95]—ternary conditions (`condition ? then : else`). execute and return the `then` branch if condition is true-like, otherwise they execute and return the `else` branch.

[93]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await

[94]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[95]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

If you need to throw an exception within your code, JonScript offers the `throw` inline expression for that purpose. This is unary, prefix operator that throws any parameter passed inside as an exception.

On the other hand, if you need to catch an exception, JonScript contains the `try-catch-finally` ternary operator. If an expression within `try` throws an exception, the mandatory `catch` clause will handle it—this part requires a function whose first parameter is the exception thrown. If there is no exception thrown, the `try` clause simply returns the result of the expression passed into it. The optional `finally` clause can change this process further by accepting what the `try` or `catch` clauses return and applies to them whichever function you have passed into it.

Furthermore, JonScript allows overloading relational and arithmetic operators. You can do this by defining one of these methods on any object. Here is a list of special methods for overloading operators:

1. `plus`, for the `+` operator,
2. `minus`, for the `-` operator,
3. `divide`, for the `\` operator,
4. `multiply`, for the `*` operator,
5. `module`, for the `%` operator, and
6. `compare` for relational operators.

The `compare` works like this: when two values are equal, it should return `0`, a positive number when the left hand side is greater than the right hand one and negative, when the opposite is true.

Operator overloading works in a symmetric fashion—it does not matter whether or not the overloaded object is on the left hand side, or the right hand side. An exception to this behaviour is a situation where both objects have an overload—the left hand side overload will be applied.

Logical operators `&&` and `||` behave just like they do in JavaScript.[96] The `&&` is a binary operator that, if the left hand side is false-like, it will return the left hand side, but if the left hand side is true-like, it returns the right hand side. The `||` returns left hand side if it is true-like and the right hand side if the left hand side is false-like. Both of these operators short-circuit.

2.1.12 The Boolean class

General concept Class counterparts of the boolean type exist in many modern object-oriented programming languages, such as C#[97] and Java.[98]

[96] https://www.w3schools.com/js/js_comparisons.asp
[97] https://docs.microsoft.com/cs-cz/dotnet/api/system.boolean?view=net-5.0
[98] https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html

These values exist as a consequence of a class-based model, since it is beneficial to be able to call methods on these values.

JavaScript implementation JavaScript standard API has the aforementioned functionality as well—there is a Boolean class.[99] Boolean class instance `valueOf` method returns its boolean value. When converted to string using `toString`, the class instance returns string representation of its boolean value.

Problematic behaviour Boolean class instances are typed as objects. Objects, besides `null`, are always true-like values.[100] Therefore, if you use `new Boolean(false)` in a condition, the expression will be evaluated as true and `then` branch of the condition will execute. This gets even stranger when using the `==` operator, since this operator will coerce the Boolean class as a boolean primitive when comparing this class to a boolean value. Here is a simple example with two variables, `a` and `b`:

```
// what do you think the value of "a" will be?
const a = new Boolean(false) ? true : false;
// what do you think the value of "b" will be?
const b = new Boolean(false) == false;
```

Because of the aforementioned behaviour, both `a` and `b` are true. Considering how the Boolean class acts in other languages, such as C# and Java, this behaviour is very counterintuitive. Even the official documentation specifies that the Boolean class should not be used as a replacement for the boolean value.[101]

Source of behaviour This behaviour is consistent with how JavaScript treats classes, objects, and logical evaluations. Objects, besides null, are always true-like within logical expressions and conditions. Also, the `==` operator always tries to do type coercion on values passed into it—therefore, the aforementioned example is logical from this perspective. However, the author calls this decision unfortunate, and has stated that he is not proud of it.[102]

Consequences The Boolean class is not recommended for use. It is hard to find any utility for this class that cannot be achieved with simple logical operators within JavaScript. If you, for example, need to know if a value of `aValue`

[99] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean
[100] https://developer.mozilla.org/en-US/docs/Glossary/Truthy
[101] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean
[102] https://brendaneich.com/2008/04/popularity/

if true-like, or false-like, you can use either `!!aValue`, or `Boolean(aValue)`. Both approaches return a simple boolean value.

Solution JonScript will not create Boolean class through its automatic `new` keyword insertion.

2.1.13 The `new` keyword

General concept The `new` keyword in languages similar to C#[103] or Java[104] is used to instantiate a class. In C++ however, this operator is used for memory allocation,[105] instead of the original C function `malloc`.[106]

JavaScript implementation The `new`[107] keyword lets you create an instance of a class. This keyword can also be used when calling any function. When used in a conjunction with a call to a standard JavaScript function, the function/constructor, depending on its implementation (a function can find out whether or not a `new` has been used to call it through looking a the property `new.target`),[108] can return a different value based on the presence of this keyword. This feature is used in several standard JavaScript API functions, such as Date, String, Number or Boolean—these functions return a different value when called without the `new` keyword. However, if you create your own class definition using the keyword `class`, not using `new` during a call to constructor results in an exception. Furthermore, you can call a class constructor with merely the `new` keyword and omit the parentheses.

Problematic behaviour This behaviour of returning a different value based on the presence of the `new` keyword is counterintuitive, since this behaviour is not present in modern object-oriented languages like C# or Java. Developers used to these languages would not expect the missing `new` to affect what a function/class constructor returns. For example, when calling the Date class constructor without the `new` keyword, it yields a string, not a Date class instance. Here are the examples of the aforementioned behaviour:

[103] `https://docs.microsoft.com/cs-cz/dotnet/csharp/language-reference/keywords/new-modifier`
[104] `https://www.javatpoint.com/new-keyword-in-java`
[105] `https://www.geeksforgeeks.org/new-vs-operator-new-in-cpp/`
[106] `https://www.tutorialspoint.com/c_standard_library/c_function_malloc.htm`
[107] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new`
[108] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target`

```
    const a = new Date; // "a" becomes a Date object
    const b = Date(); // "b" becomes a string
    const c = new Date(); // "c" becomes a Date object
```

Source of behaviour When calling a class constructor without the keyword `new`, you are specifically calling a function and not a class constructor. It is therefore not unexpected behaviour that this function would return a different result.

Consequences Not needing to use the keyword `new` leads to many situations, where a bug occurs in an application because of a developer oversight. This oversight is that much easier to make since there are classes within the JavaScript standard API that do not react to a missing `new` keyword during constructor call—such as the Array,[109] or RegExp classes.

Solution JonScript automatically detects whether or not the `new` keyword is appropriate (through looking for a class signature within TypeScript typing) and inserts it into the code by itself—therefore removing the need for the `new` keyword within the language in the first place.

2.1.14 The `async` keyword

General concept The keyword `async` is a part of a programming pattern called async/await [21]. Popular programming languages, such as Python [22] and C# [35] support this feature. It is used to handle asynchronous, non-blocking operations through function annotation. The `async` before a function means that the value it returns is a promise. A promise is a data structure that contains an operation and its return value. While the value does not exist initially, the promise will contain this value once the operation finishes. The use of the operator `await` can then resolve the promise, and return the unwrapped value.

JavaScript implementation In JavaScript, the `async`[110] keyword used before a function acts as a wrapper that takes the return value of a function and does either one of two things: if the value that this function returns is a promise, it simply returns that promise, and if the value that the function

[109] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

[110] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

returns is not a promise, it returns a promise that resolves into the aforementioned value. Furthermore, if you try to define a function that uses the `await`[111] operator without `async`, an exception will be thrown.

Problematic behaviour When a programmer is already using the keyword `await` within a method or a function, they must add `async`—this is redundant and should be automatic. The `async` is a necessary formality when `await` is already used within a function.

Source of behaviour Await/async is a well-known pattern in several other languages, such as C#[112] or Python.[113] When this implementation is already present in other well-known languages, it is easier for developers coming to JavaScript from other languages to understand how to use `await` in functions or methods if the feature is used in the same way in JavaScript.

Consequences When programmers wish to extend a deeply embedded function to use await/async pattern, they must then add the keyword `async` to all functions in which they wish to await the return value of the embedded function—and this addition of `async` will spread exponentially through their code.

Solution Whenever a function uses `await` within its body, JonScript automatically inserts `async` into its definition. This way, you can use `await` anywhere you wish without bloating your code.

2.1.15 Functors

General concept A functor, as understood in C++, is a class instance that acts like a function.[114] In C++, this is achieved through overloading the `()` operator on a class definition. Sometimes, a functor may be referred to as a stateful function.[115]

JavaScript implementation Functors in JavaScript are created differently from C++. A functor is a function that has additional properties created on itself during run-time. Since it is impossible to actually overload the operator `()` as you would in C++, this is the way to achieve the same behaviour in JavaScript.

[111] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await`
[112] `https://docs.microsoft.com/cs-cz/dotnet/csharp/programming-guide/concepts/async/`
[113] `https://docs.python.org/3/library/asyncio-task.html`
[114] `https://www.cprogramming.com/tutorial/functors-function-objects-in-c++.html`
[115] `https://basarat.gitbook.io/typescript/main-1/statefulfunctions`

Problematic behaviour While functors by themselves are not problematic, there is no simple way of creating them—you need to write a specific merging function between an object and a function as parameters that returns the desired functor. The lack of functor definition syntax results in a code-bloat.

Source of behaviour Functors require overloading the application operator `()`. See Section 2.1.11 for more details.

Consequences Programmers that wish to utilize functors in JavaScript, as they would in C++ need to write their own custom function that creates them and use it each time that they want to create a functor.

Solution JonScript is capable of creating functor objects and classes with a single expression. Also, JonScript can easily create constructs, where a class may inherit the overload from it's parent.

2.2 Syntax and semantics

This section will go through the syntax and semantics of my newly created language—a description of how to define methods, classes, objects, and expressions with a description of how they work. I will first enumerate each important point in JonScript syntax. There are points within this enumeration, where JonScript does not differ from TypeScript. At such points, there are appropriate links to resources to the appropriate TypeScript semantics, in order to not reiterate TypeScript syntax.

Variable name JonScript has stricter variable name rules than JavaScript or TypeScript. In JonScript, a variable name can only consist of letters from A to Z, uppercase and lowercase, the dollar sign and numbers from 0 to 9. A variable name cannot start with a number. The intended convention is for class properties to start with a lowercase letter, and module and class names to start with an uppercase letter.

List of imports Each JonScript file can have one list of imports, similar to TypeScript import syntax,[116] defined before anything else. These imports either denote a local file dependency, or a NPM package dependency. There are two kinds of imports shown below. There are no separators between imports.

[116] `https://www.typescriptlang.org/docs/handbook/modules.html`

Default import A default import consists of the `import` keyword, a variable name to be used as a reference to the imported dependency, and a string literal denoting either the path to a dependency—this may be a path to a folder, if such folder contains a file named `index.js`, or `index.ts`—this is consistent with how imports in TypeScript behave. This path may contain a name of the imported NPM package as root—this is a standard feature of TypeScript imports as well. A default import will import all exported variables from a dependency. Consider this example, where I import every exported variable from package `jquery`:

```
import $ from "jquery"
```

Named import Named imports work in a similar fashion as default imports, except that instead of a variable name to be used as a reference to the imported dependency, named imports contain a list of variable names, separated by commas within curly brackets. Each variable name can be followed by an optional keyword `as` and another variable name. This specifies that you want the imported variable to be referred to as the other variable name within the file. Consider an example where I want to import two exported variables (named `uniq` and `orderBy` from the lodash package:

```
import { uniq, orderBy } from "lodash"
```

In the next example, I import the same variables under different names:

```
import { uniq as unique, orderBy as sort } from "lodash"
```

If I use the renamed import example instead of the original one, the exported variable `uniq` will be referred to as `unique` and the variable `orderBy` as `sort` within the file.

Module Each JonScript file needs to include exactly one module containing prototypes. A module consists of a variable name and curly braces. Between the curly braces, there is a list of classes without separator. Consider this example, where I create and empty module named `NameOfModule`:

```
NameOfModule {

}
```

Prototype Each prototype is comprised of template types, parameters, private properties, inheritance and public properties. A prototype is an alternative to a class within a prototype inheritance enviroment. A prototype definition consists of:

1. a variable name (unique to other prototypes within the same module),
2. an optional list of type templates (each with unique name), surrounded by triangle braces, separated by commas,
3. a list of parameters surrounded by parentheses (each with unique name), and
4. a pair of curly braces, which contain a list of private properties delimited by commas, inheritance statements delimited by commas, and a public API, in this order.

Parameters Each parameter declared in JonScript consists of a parameter name—which works the same as a generic variable name, but with one exception. The `_` character may replace the parameter name. The `_` parameter means a placeholder—a method parameter that will not be referred to within the function body. Next, each parameter can have the colon sign, or a question mark with a colon sign, after which a type definition follows. After type definition, the optional keyword `=` can be placed, with an expression following after it. This expression will be the default parameter value. While the colon sign simply indicates a place for the parameter type, question mark with a colon sign indicates that this parameter is an optional one. There is also a special kind of parameter, which is denoted by three dots before its name. These dots denote a rest parameter,[117] which can only occur as the last—or only—parameter defined within a parameter list. This parameter signals an unlimited amount of parameters of certain type will be accepted during a call. Here is an example of a list of three parameters. The first parameter, `a`, is a string, the second parameter, `b`, is a number, and the third parameter, `c`, is a spread parameter of numbers.

```
    (a: string, b: string, ...c: number[])
```

[117] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-0.html

Types JonScript typing is essentially just TypeScript typing of parameters and return values of methods. Here is a list of possible syntax for type declaration in JonScript:

a type name This way, you can refer to a defined type elsewhere in the current scope,

a type operation Either a type intersection (`type1 & type2`), denoting a merge between two types, or a type union (`type1 | type2`), denoting that a parameter can be either of two different types,[118]

a generic type reference This reference[119] takes a type name, and a list of types surrounded by less-than sign on the left and more-than sign on the right,

an arrow function expression type This type consists of a pair of parentheses, in which a list of parameters is defined, separated by commas, and an arrow sign `=>`—after this sign, there is a return type, and

an object type Which consists of a pair of curly braces, between which a list of properties is found, separated by commas—each property consists of a property name and its type—the format of which is the same as if it was a parameter type.

There are two special template types in JonScript, that are not found in TypeScript—`Class` and `Object`. The former is a shorthand way of referencing a return type of a method, function, or a prototype. The latter is a way of referring to a type of variable (as replacement for `typeof` in TypeScript).

Templates JonScript templates work like a subset of TypeScript templates—a template consists of three parts—a template name (which works in the same way as a variable name), an optional keyword `extends` followed by a type, and an optional keyword `=`, followed by a type. The type which comes after `extends` enables you to limit what kind of type can be used within the generic type and the keyword `=` allows you to set a default template type, when this template is used. Here is a list of templates surrounded by square brackets. First template, `T`, can be any type, second template, `E`, must be type which inherits from string (such as string enumeration), and third template type, `F`, will be a number by default (but can be any other type if specified).

```
    <T, E extends string, F = number>
```

[118]https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html
[119]https://www.typescriptlang.org/docs/handbook/2/generics.html

Private properties Private property consists of a variable name and an expression, separated by colon. A comma is placed after the expression. References to private variables within scope work from top to bottom—bottom properties can reference top ones. A special type of private property is `_`. This property refers to an ignored property. This property is not accessible and essentially just serves as a way to express something to which you will not refer to in the future. A private property cannot have the same name a method-/constructor parameter. Consider this example, where I create a single prototype, named `EmptyPrototype` and one private property `privateProperty` with value of `1`:

```
EmptyPrototype() {
   privateProperty: 1,
}
```

Inheritance statements An inheritance statement consists of a three-dot token `...` and an expression, ending with a comma. Right after such statement is defined, two things happen: All public properties and methods of this object will immediately be accessible within the body of the prototype and the very same properties will now also be included amongst public properties of the prototype instance, unless overridden by other inheritance statements of the prototypes own public properties. Consider this example, where I create a single module, with a single prototype, `Child` and one import—`Module` containing a single prototype—`Parent`:

```
import { Module } from "parent"
ChildModule {
    Child() {
        ...Module.Parent(), // Inheritance statement
    }
}
```

Public API A public API can be defined as either one of these two things:

1. as a list of properties—this list consists of a pair of curly braces, within which is a list of properties in a similar format as private properties. Each public property within this list must have a unique name, can be named the same as private property or method/constructor parameter, but if it is named the same as an inherited property, it must either share its type, or be `nil`. Public properties cannot be `_` (ignored). Public properties are delimited by commas—the exception to this is a list of properties with a single member—this member does not have to be delimited by a comma, or

2. as an expression—any expression within JonScript can be used as a public API.

Public API provides public properties to its class or object.

To show how a public API within JonScript works, I showcase several examples of public APIs. The first example will be a simple module with a class and a single public property (`publicProperty`) with a value of `1`:

```
Module {
    Class() {
        {
            publicProperty: 1,
        }
    }
}
```

Next, I showcase how to create a prototype with an overloaded application operator `()` with an expression public API—the expression in question is a method declaration.

```
Module {
    Class() {
        () => 5
    }
}
```

Comments JonScript has two C-style comments, single-line and multi-line. Single-line comments are denoted by two forward slashes (`//`) and multi-line comments are denoted by a pair of slashes with stars (`/**/`).

Expressions

This subsection goes over the semantics of all the expressions that can be used in JonScript and explains how they function.

Method/Function declaration Each method defined in JonScript is a first class, higher order anonymous function (also, an arrow function expression) assigned to a prototype, or object, property. A function consists of a list of parameters surrounded by parentheses—after the parameter declaration, there is an optional colon followed by a return type. After, an arrow syntax token (`=>`) is present, followed by an expression that the method returns. The only difference from TypeScript arrow function expressions, similar to private properties, is that there can be multiple `_` parameters—denoting that certain parameters should be ignored (rather than at most one in JavaScript).

Objects Objects have the same syntactic structures as class bodies. Therefore, each object can have private properties, inheritance statements and a public API, either comprised of public properties, or an expression. This principle embodies the statement that prototypal "classes" are just functions that return an object. Consider an example, where I create a class with a single public property (`publicObject`) containing an object. This object inherits a from a single imported `Parent` prototype, has a single private property named `privateProperty` with the value of `"Hello"`, and a single public property with the value of `"Hello World"`:

```
import { ParentModule } from "parentModule"
Module {
    Prototype {{
        publicObject: {
            privateProperty: "Hello",
            ...ParentModule.Parent(),
            {
                publicProperty: privateProperty + " World",
            }
        },
    }}
}
```

Functors You create a functor by adding a function expression to the top of an object body. This is normally possible to do in JavaScript only through multiple statements and problematic to do in TypeScript. Consider an example, where I create a simple functor with two public properties `a` and `b`. When called, this functor will return the result of the expression `a + b`.

```
Module {
   Class() {{
       () => a + b,
       a: 5,
       b: 10,
   }}
}
```

Try-catch-finally expression This ternary expression consists of three keywords: `try`, `catch` and `finally` (which is optional). After `try`, an expression is required. After catch and finally, a function is expected. Try will evaluate an expression—if this expression throws an exception, catch block executes a function that has been passed into it. If the `finally` is present, the resulting value of either the `try` expression, or the `catch` executed function will be passed into the function defined after the `finally` keyword. This expression returns either the value returned by catch on expression, or the value passed into try clause, if `finally` is not present. If the `finally` is present, then this expression returns the return value of the executed function that was passed into it. Consider this example, of a simple simple try-catch expression catching from a function call of the function `toCatch` and print it into a `console.error`. Finally, the expression returns the number `5`:

```
try
    toCatch()
catch
    exception => console.error(exception)
finally
    () => 5
```

Next, I will showcase the same example from before, except without the `finally` part:

```
try
    toCatch()
catch
    exception => console.error(exception)
```

This expression now returns the value `nil`, if the expression `toCatch()` throws an exception—or it returns the return value of `toCatch()`, if it does not.

Await expressions Await expression is an unary prefix operator, which consists of the `await` keyword and an expression. If the expression within this expression is a promise, it will be resolved. If not, the expression is simply returned as-is. Await expressions [7] act much the same as their TypeScript counterparts. Here is an example of an await expression awaiting a promise resolution from the variable `aPromise`.

```
await aPromise
```

Throw expressions Throw expression is an unary prefix operator, which consists of the `throw` keyword an an expression. This expression is thrown as an exception. This behaviour is consistent with the TypeScript `throw` statement.[120] Here is an example of throwing a string `"Hello World"`.

```
throw "Hello World"
```

Property access expressions There are two kinds of property access within JonScript. The first one consists of two variable names with the `.` keyword between them. Alternatively, you can use a variable name and a pair of square brackets with an expression between them to the same effect. Here is an example of two property accesses–one through a dot, one though square brackets. I showcase accessing a property `aProperty` on the variable `anObject`.

```
anObject.aProperty
anObject["aProperty"]
```

Function call expressions A function call expression consists of a function name as reference, an optional template, which is a list of types surrounded with less-than on the left and more-than sign on the right with delimiting commas between them, and a pair of parentheses with a list of parameters between them, also delimited by commas. Here is an example of calling a function `aFunction`, which accepts three numeric parameters.

```
aFunction(1, 2, 3)
```

Array expressions An array expression consists of a pair of square brackets with a list of expressions between them, delimited by commas. Each element in the list can be preceded by the `...` keyword. This denotes the use of the

[120] `https://basarat.gitbook.io/typescript/type-system/exceptions`

spread operator, which merges the expression as if it was an array. Consider this example, where I merge together two arrays and one object:

```
    [...[1, 2, 3], ...[1, 2, 3], ...{{ a: 5 }}]
```

The result of this expression will be an array containing the following numbers, in this particular order: `[1, 2, 3, 1, 2, 3, 5]`. The spread operator in JonScript automatically converts any object it receives into an array of property values. The ordering of these values is determined by the order in which they were defined on the aforementioned object. String values and String classes merge as if they were arrays of characters. If you try to merge any kind of numerical value, `nil`, or a function, the resulting array elements will not be affected.

Logical expressions There are three kinds of logical expressions within JonScript:

1. the `!` unary prefix operator, which returns a true boolean, if its parameter is false-like, and the false boolean, if its parameter is true-like,
2. the `||` binary infix operator, which returns the right hand side argument, if the left hand side argument is false-like, and returns the left hand side argument, if it is true-like, and
3. the `&&` binary infix operator, which returns the right hand side argument, if the left hand side argument is true-like, and returns the left hand side argument, if the left hand side argument is false-like.

Relational expressions There are six kinds of relational expressions within JonScript. Since their function was already described in Section 2.1.11, this will only be a brief overview of their syntax:

1. The equality binary infix operator `==`,
2. The less-than binary infix operator `<`,
3. The more-than binary infix operator `>`,
4. The less-or-equal-than binary infix operator `<=`,
5. The more-or-equal-than binary infix operator `>=`,
6. The non-equality binary infix operator `!=`.

Condition expressions JonScript contains a single ternary conditional operator. This operator consists of an conditional expression, the `?` keyword, a consequent expression following it, and `:` keyword, with an alternative expression following as well. If the conditional expression before the `?` keyword is true-like, the consequent expression is executed and returned. If the conditional expression is false like operator is false-like, the alternative expression is executed and returned. Here is an example of conditional expression, with a conditional expression being the variable `aCondition`, the consequent expression being the variable `aConsequent` and the alternative expression being `anAlternative`.

```
    aConsequent ? aCondition : anAlternative
```

Is operator The `is` operator is a binary logical operator, that also acts as a TypeScript type-guard `is`.[121] The `is` operator's left hand side is an expression to be evaluated and the right hand side contains a pattern for the left hand side to be checked against. This operator provides pattern matching to JonScript. Consider an example, where I check an object (`anObject`) against an object pattern. This pattern consists of an object with three properties. The first property, `a` is a String class constructor. This property will match any string or a string class instance. The second property, `b`, contains a Date class instance of a concrete date. This property will match to either the Date object representing the same date, a string which represents this concrete date, or a numerical value of milliseconds from the UNIX epoch before the aforementioned date. The third property, `c`, contains an array of two values, `1` and `2`. The `c` property will match to an array of two elements, `1` and `2`.

```
    anObject is {{
        a: String,
        b: Date(2020, 1, 1),
        c: [1, 2],
    }}
```

String literals There are two kinds of string literals within JonScript: the first one consists of a pair of double quotes with text between them—working just like the double-quoted string[122] in JavaScript does. The other kind of string is a template literal string—this string consists of a pair of grave accents between text. If this text contains the dollar sign with curly brackets after it, the value inside the curly brackets will be evaluated as an expression and

[121] `https://www.typescriptlang.org/docs/handbook/advanced-types.html`
[122] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String`

concatenated within the string. Here is an example of concatenation of double-quoted string (`"Hello "`), and a template string literal, with a variable `aName` inside it.

```
"Hello " + `World ${aName}`
```

Boolean literals There are two boolean literals within JonScript, `true` an `false`.

Number literals JonScript number literals consist of positive and negative floating-point numbers, including a zero, without `NaN`. See Section 2.1.8.

Nil literal The `nil` literal value within JonScript acts as a way to denote a null (empty) value. This value can also be used within type definitions to the same effect, denoting an empty (void) type. This literal is compiled into `undefined`.

2.3 Code example

This section showcases JonScript functional programming capacity by creating two algorithms as examples in it. The first example is algorithm is a fibonacci calculation algorithm. This example contains a single module `MathModule`, in it a single class `Fibonacci`, with a single number type parameter `index`, which will have a function declaration expression as its public API. When this function executes, the resulting number is the member of the fibonacci sequence at the index denoted by `index`. The concrete implementation of fibonacci sequence calculator is placed in private method named `calculate`:

```
MathModule {
    Fibonacci(index: number) {
        calculate: (
            index: number,
            first: number = 0,
            second: number = 1
        ) => !index
            ? first
            : calculate(index - 1, second, second + first),
        () => calculate(index)
    }
}
```

The second example is a queue algorithm. This example is a generic Queue prototype. The Queue prototype has three basic queue methods: `push`, `pop` and `top`. The first method enqueues an element (and return queue), the second method returns a queue with a dequeued first element, and the third method returns the first element in line to be dequeued.

```
    QueueModule {
        Queue<T>(...items: T[]) {{
            push: (item: T) => Queue(item, ...items),
            pop: () => Queue(items.slice(1)),
            top: () => items[0],
        }}
    }
```

Note that this example uses TypeScript type inference[123] in order to simplify the use of generics—one does not need to use explicit generics when defining the Queue class—the type for generics will be inferred from the elements passed into its constructor.

[123] `https://www.typescriptlang.org/docs/handbook/type-inference.html`

Chapter **3**

Implementation

This chapter describes the technical details of the JonScript implementation. It covers the technologies used to create this language, as well as an overview of the build process and what the future development of JonScript.

3.1 Broad overview of the build process

I this section I describe the implementation details without focusing on particular technologies that were used to accomplish them. There are four main parts to the JonScript build process:

Pre-processor Which resolves references to JonScript files and breaks the JonScript files into syntactic tokens;

Parser Which is the main part of the build process, where JonScript files are compiled into TypeScript;

Post-processing Which analyzes the resulting TypeScript code and does additional processing to implement type-dependent changes within the code, such as automatic `new` and `async` insertions; and

JavaScript compilation Which compiles the resulting TypeScript into JavaScript. This JavaScript may, depending on configuration outside of JonScript, then be subject to other changes, such as minification or uglification.

I chose to compile JonScript to TypeScript, over asm.js and webassembly [15]. The obstacle in choosing WebAssembly is its lack of access to Document Object Model[124] (DOM), making it less suitable for regular web development.[125] The reason for not choosing asm.js is that it is deprecated.[126]

[124] `https://www.w3schools.com/js/js_htmldom.asp`
[125] `https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts`
[126] `https://developer.mozilla.org/en-US/docs/Games/Tools/asm.js`

3.2 Distribution

JonScript is distributed as an NPM package, to be used as a webpack plugin. For description of how webpack is used, refer to Section 3.3.6. You can download this package to be used as plugin from NPM.[127]

3.3 Technologies

This section goes over a list of packages from NPM repository which JonScript depends on, as well as detailed explanation how they are related to the compilation process. This list does not contain every single package used—rather, it focuses on the most important ones. For a full list of dependencies, you can refer to the jonscript[128] and jonscript-util[129] (utility package for the language) respectively.

3.3.1 NodeJS

NodeJS [13] is a very popular run-time for JavaScript.[130] It is based on the Google Chrome JavaScript engine. This project uses this run-time to compile JonScript into TypeScript.

3.3.2 NPM

NPM is a package management system and therefore controls all other dependencies of JonScript used during the build process. NPM is also responsible for package versioning and allowing others to download my newly-published language.

3.3.3 Typescript

JonScript compiles into TypeScript rather than directly into JavaScript. Compiling JonScript this way allows JonScript to take advantage of TypeScript type system.

[127] `https://npmjs.com/package/jonscript`
[128] `https://npmjs.com/package/jonscript`
[129] `https://npmjs.com/package/jonscript-util`
[130] `https://insights.stackoverflow.com/survey/2019#technology`

JonScript uses the TypeScript typing system to implement these features:

1. methods keeping context after assignment (for details about this process, see Section 2.1.2),
2. removing the `this` keyword from JonScript,
3. removing the `new` keyword from JonScript (for details about this process, see Section 2.1.13),
4. removing the `async` keyword from JonScript (for details about this process, see Section 2.1.14).

Implementing these features manually would have meant generating complex boilerplate code. Instead, TypeScript provides these features. This would have led to code-bloat and much slower run-time. If I wanted to implement automatic `new` within JonScript without the help of TypeScript typing, I would need to check each function for the property `constructor.name` before calling it. Worse yet, if I wanted to ensure that methods keep their context after assignment, I would need to check each and every property access, to see if it yielded a function. In case I wanted to remove `this` keyword from the language, I would need to generate inherited properties as local variables (since JonScript provides a `this`-free access to inherited properties as well as it does to private and public ones). The only way of doing this is using the `with` keyword—which is not recommended for use.[131] The only features that I would be able to implement easily without TypeScript would be `async` removal, since this would only require JonScript parser to do additional lookup for the operator `await` within function body. However, since the package ts-morph (more detailed description is in Section 3.3.5) offers this feature already, I decided to use their implementation instead.

As consequence, JonScript is only able to use either packages written in TypeScript, or JavaScript packages have TypeScript typing accessible. Given the popularity of TypeScript [6], most popular packages either are written in TypeScript, or have TypeScript compatible typing either included right away, or the typing exists as a separate package. In case you wish to use purely JavaScript package within JonScript, you can define your own typing for such package. Alternatively, you can import the pure JavaScript package into a TypeScript file and import the TypeScript file instead—since JonScript allows you to import local TypeScript files.

3.3.4 antlr4ts

In order to implement the language parser, I used ANother Tool for Language Recognition 4 (ANTLR4) [36]. ANTLR4 is a LL(*) left-to-right, leftmost

[131] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/with`

derivation parser generator written in Java. ANTLR is widely used for language implementation. To utilize ANTLR4 within my project, I imported this the package antlr4ts.[132] This package takes ANTLR4 grammar and outputs a parser written in TypeScript. This parser was then used to parse JonScript into a JonScript Abstract Syntax Tree (AST).

3.3.5 Ts-morph

In order to compile JonScript to TypeScript, I used the package ts-morph. [133] Ts-morph is a compilation and AST manipulation framework for TypeScript. JonScript is compiled into TypeScript by compiling the JonScript AST into a TypeScript AST, managed by this package. After that, ts-morph generates TypeScript source code. After compiling the code into TypeScript, I also use ts-morph to perform JonScript post-processing features mentioned in the Section 3.3.3.

3.3.6 Webpack

Webpack is a household name within modern JavaScript development.[134] Webpack allows you to insert unlimited compilers and then add directions as to which files they will compile and where they should output the resulting files. Webpack simplifies compilation configuration.

JonScript exists as a webpack plugin. This was done so that the TypeScript files emitted by JonScript compilation could be compiled into JavaScript. When you wish to use TypeScript in the browser, you do so by compiling it into a JavaScript bundle, alongside a collection of map files (map files will be included if you wish to debug your code in a browser). JonScript is designed to be integrated within standard TypeScript application.

3.4 Parser description

This section describes how JonScript is parsed from each JonScript file to the resulting JavaScript code to be run in the browser. First, the ANTLR generated parser takes each individual file and parses them into a stream of tokens. JonScript then uses the ANTLR bindings onto each of these tokens and uses mapping functions to parse the ANTLR bound tokens to an AST of TypeScript. The ts-morph package then saves the generated AST into a file. Then, all generated files are subjected to post processing. The package ts-morph is responsible for this as well. It loads an AST from each individual file at the same time and then mutates the tree in order to implement the

[132] `https://www.npmjs.com/package/antlr4ts`
[133] `https://www.npmjs.com/package/ts-morph`
[134] `https://www.npmjs.com/package/webpack`

post-processing features. This double loading is done due to the fact that ts-morph cannot recognize TypeScript types until the AST in question is saved to a file. Then, all TypeScript files are loaded by webpack and compiled into JavaScript.

3.5 Future release features and shortcomings

This section dives into problems that are either not yet implemented in JonScript, due to time and resource constraints, or will require further research, or are likely impossible to solve due to an underlying JavaScript constraints.

3.5.1 Features that will be implemented in future release

These features were not implemented in the first JonScript purely because of a lack of time and resources—they will be rolled out in near future. There is not much further research required for these, since they are rather straight forward.

Implement JSX The JSX [19] is a language extension for JavaScript, which allows you to model HTML tags as JavaScript expressions. As discussed in the introduction, the JSX is of great help when trying to model user interface in your application.

Sanitize unwanted types JonScript needs to be able to crash on compilation if it encounters the `any`, or `unknown` types—since they will interfere with its capacity to do post-processing.

Fix spread operator on array-like values In standard JavaScript API, there are several methods that return values that are similar to arrays. These values can be converted to regular arrays in JavaScript by using the spread operator.[135] I will be adding this functionality into JonScript as well. Currently, the workaround is the use of `Array.from`,[136] which converts array-like objects to simple arrays.

Fix template literal string escape sequences JonScript needs additional escape characters when using `$`, `{` or `}` within template literal strings.[137] JavaScript only needs to escape `$` character within such string. This will be

[135] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

[136] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from

[137] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

fixed with a better parser in the future release. Since this bug fix was not considered significant enough to be fixed in the first release, it will be fixed in the future one by implementing a more capable parser.

Introduce map files into compilation process Map files [6] are a great tool for debugging your program. In essence, they allow you to map the compiled JavaScript code onto whatever the original language that was compiled into JavaScript was. This is highly useful, as you can then use a debugger which correctly goes over line-by-line of the original code, therefore greatly decreasing the difficulty in debugging compiled code. Since JonScript uses the package webpack for its build process and webpack already supports the creation of map files, an algorithm will be added to provide this functionality.

Visual Studio Code Extension for JonScript Visual Studio Code[138] is an open-source IDE developed by Microsoft. Since it supports a large amount of extensions making it compatible with a number of languages, I will develop an open-source extension for JonScript, offering features like syntax highlighting, debugging and code references.

Descriptors for properties in classes JavaScript has some measure of support for class, property and method descriptors[139] on both classes and objects. In future versions of JonScript these features will be added and possibly improved.

3.5.2 Features that require more research

These features either need to be researched further, in order to fit them neatly into another category, or there are multiple ways of implementing or designing them. Therefore, they are in a category on their own—likely, they will be fixed in a future release, but I do not yet have a clear outline of the concrete solution.

Add regex expressions JavaScript has regular expression literals, which can be created a string surrounded by forward slashes (`/`). For now, JonScript only supports the use of RegExp class for creating regular expressions. In future release, JonScript will support the creation of these regular expression literals.

Rename properties to protect against keyword conflicts JavaScript has many keywords and sometimes, when a property name is the same as a keyword within the language an exception, or a build failure may occur.

[138] `https://code.visualstudio.com/`

[139] `https://medium.com/jspoint/a-quick-introduction-to-the-property-descriptor-of-the-javascript-objects-5093c37d079`

There may be a renaming scheme to protect against this behaviour in the future release—at least for private properties.

Fix possible-null types Since JonScript implements automatic optional chaining and TypeScript can be set up to crash on an unchecked null reference, this creates a problem—either all types in JonScript will be compiled as possibly null,[140] therefore avoiding this issue, or the default JonScript TypeScript compiler will be set up to ignore this error, or the non-null assertion operator may be deployed to fix this error.

Add the is operators compatibility with JonScript classes While the `is` operator is an improvement over both `typeof` and `instanceof`, it still lacks one very important feature—it cannot yet match classes directly created in JonScript—at least not out-of-the box for now.

This can be ameliorated by using a public property of a class for a match with `is` and letting TypeScript do its type coercion by using a union type between the pertinent types.

Optimize the build process Given that JonScript has eliminated the need for `this` keyword in your code, it needs a way of accessing properties within a class. This is achieved through dynamically creating local variables. This, however, seems to slow down the build process significantly, when inheritance is involved. Further research is required on how to make JonScript compile faster.

Allow access to private properties within same-type classes So far you cannot access a private property of another class instance of certain prototype within a context of the same prototype. Each instance can only access its own private properties. Future versions of JonScript should allow you to access private properties of another instance of the same-type class.

3.5.3 Shortcomings without a clear fix

This subsection describes problems with JavaScript engine to which I have not found a solution. In general, these problems relate to compromises among the JonScript tenets in the Chapter 1 (Introduction), and to the JavaScript engine and how it treats certain values.

No deep object sanitize In above section I write about the removal of `NaN` and `null` from the language, as well as JonScript being compatible with the JavaScript ecosystem. A question arose from these two facts—what happens, if an object contains an embedded property with the `NaN` value? JonScript

[140]`https://www.typescriptlang.org/docs/handbook/advanced-types.html`

guarantees that such property, when directly accessed, will act as if it was simply `undefined`. However, because of performance, JonScript does not remove every `NaN` or `null` that happens to exist within its object structure. This means that if you pass an object from outside of JonScript into a function that was imported as well, their behaviour might change based on these values. This behaviour also makes sense if the function or method that accepts such object depends on these values being what they are.

Subtle differences between values and their class counterparts While JonScript ameliorates most of the differences between, say, String class and string value, two differences still prevail because of the JavaScript engine. These differences are the inability to add properties onto a string primitive and the fact that a class instance can never be false-like. Therefore, the class instance of number zero, or the class instance of empty string will not be false-like, even though their values will. It is possible that these classes will be removed within future release.

3.6 Lessons learned

During the implementation process of JonScript, I have learned how difficult it is to plan and execute a creation of a whole new language from bottom-up perspective. Furthermore, there needs to be an immense effort to document features properly, in order to make the new language understandable to newcomers. While I truly believe that the features implemented in JonScript are of use to both beginner and experienced JavaScript developers, it may be beneficial to think about moving JonScript features from a separate language to TypeScript pre-processor—implementing JonScript features directly within standard TypeScript syntax.

Chapter 4

Evaluation

This chapter evaluates the implementation of JonScript in two ways. First, I evaluate the correctness of the implementation with respect both to the specification and the stated goals. Second, I evaluate the run-time performance of JonScript compiled JavaScript code in comparison to JavaScript compiled from TypeScript.

4.1 Correctness of the implementation

This section focuses on whether JonScript fulfills the specification from the Chapter 2 (Design), and the stated goals in the Chapter 1 (Introduction). This section also contains a list of descriptively named unit tests, all of which my solution passes. These unit tests were designed to show correctness of the implementation—not only that it compiles into JavaScript properly, but also that the features stated in the Chapter 2 (Design) are functional.

4.1.1 Unit test list

To further support my arguments about the utility and correctness of JonScript, I developed thirty unit tests. These tests set out to ensure the correctness of JonScript and its most important features. All of these unit tests pass. See Table 4.1.

4.1.2 Evaluation of the stated goals

In the thesis statement, I declared three goals: Ease of use, consistency and compatibility. This subsection presents arguments for all three of these goals in terms of whether or not they were achieved and to what extent.

Ease of use JonScript syntax is largely similar to TypeScript syntax. This leads to easier adoption of this language from developers that are already

Name of test	Test result
Overload the plus operator	Passed
Overload division operator and then convert resulting `NaN` to `undefined`	Passed
Calculate Fibonacci sequence	Passed
Use standard JavaScript API method `reduce` on array	Passed
Access properties on `undefined` value	Without error
Custom QuickSort algorithm	Passed
Inheritance, inherits function public API	Passed
Import local TypeScript files	Passed
Do named import and then rename it from TypeScript file	Passed
Import multiple named imports from local TypeScript file	Passed
Import and inherit from a prototype from renamed JonScript module	Passed
Use default import, which imports all exported variables	Passed
Import and utilize JonScript module	Passed
Use single parameter anonymous functions without parentheses	Passed
Run QuickSort with generic types	Passed
Create classes which use inversion control	Passed
Use the `is` operator	Passed
Use spread operators on arrays with `nil` members	Passed
Import and inherit from JonScript module	Passed
Create a single-expression functor	Passed
Use a function type as a type of parameter	Passed
Use the Class template type to get type of prototype imported from another module	Passed
Use with multi-level inheritance with prototypes having multiple parents	Passed
Remove inherited properties from public API	Passed
One object can inherit from another	Passed
Use object type as method parameter type	Passed
Create multi-level object inheritance from other objects	Passed
Create complex prototype system with pattern matching, and with automatic `async` and `new` keyword insertion	Passed
Can inherit from an array in prototype	Passed
Define an expression as public API in prototype	Passed
Call `Array.from` despite `from` being a keyword in JonScript	Passed

Table 4.1: Unit test results

used to TypeScript. Furthermore, JonScript increased compatibility with functional programming paradigm (due to single-expression functors, function consistency improvements and removal of side-effect operators). Due to this, developers can gain greater use of function as first class citizen principle, and higher order functions. However, it could be argued that due to slow build time, JonScript has not fully achieved this stated goal. Further research will be necessary to determine how exactly to ameliorate (and measure) this issue.

Consistency Second goal of JonScript is to fix inconsistent and hard to predict behaviour JavaScript is known for. The Chapter 2 (Design) and Chapter 1 (Introduction) describe these inconsistencies in detail. Here is a list of ways how JonScript was able to correct these:

1. function context behaviour,
2. null value behaviour,
3. class initialization behaviour,
4. operator behaviour with regards to type coercion,
5. single and double quoted strings, and
6. run-time type checking of String class instances, Number class instances, and their class-less counterparts.

However, JonScript was not able to ameliorate the differences between strings and String classes and numbers and their class counterparts entirely: the Number class of number zero and String class of empty string are still evaluated as true-like in JonScript, despite the empty string and number being false-like. The ideal solution would be to make the classes in question behave in a false-like manner, when they represent the empty string and number zero respectively. This however was not possible due to how JavaScript evaluates objects.

Furthermore, JonScript is still not able to add properties during run-time to values that are not an object, nor a function. This creates another difference in behaviour between strings, numbers and their class counterparts.

Compatibility JonScript has been designed to be largely compatible with either TypeScript written NPM packages, of JavaScript NPM packages that have TypeScript typing. In order to showcase basic compatibility, I created a test project as a case study, containing a questionnaire about common JavaScript pitfalls, written in JonScript, to be displayed in a browser. This project can be found within the test folder of the JonScript git repository.[141] In order to display the questions within the questionnaire, I have imported the household name package jQuery. [142]. Furthermore, in order to store the answers to each questions, I have imported another household name package—Redux. [143] The Redux package was responsible for predictable state updates within the project. The JonScript application works with both packages without issues.

4.2 Performance testing

While better performance was not one of the stated goals of JonScript, it is still relevant to the utility of this language. To test the performance of JonScript, I implement a set of micro benchmark applications and measure performance against TypeScript.

4.2.1 Test subjects

I tested two sets of JavaScript code, one compiled from TypeScript, and one compiled from JonScript. Since TypeScript does not affect the run-time, the TypeScript test-times represent how a vanilla JonScript algorithm would perform. In effect, I measure the overhead that JonScript features introduce into an equivalent JavaScript program.

4.2.2 Benchmark algorithms

I measure three algorithms:

Fibonacci sequence An i-th element in the fibonacci sequence, which was implemented as a recursive end-tail method,

Merge sort Stable, out-place sorting algorithm, which was implemented as a recursive method sorting elements, and

Langton's ant Algorithm [20], which is a cellular automaton in two dimensional field executed over k steps.

[141] `https://gitlab.fit.cvut.cz/jindrj14/master-thesis`
[142] `https://jquery.com`
[143] `https://redux.js.org`

Benchmark for JavaScript	Min	Max	Mean	Median	Std Dev
5000th fibonacci number	5	15	7.47	7.0	2.19
1000th Langtons ant step	8	29	14.96	17.0	4.80
Merge Sort (700 elements)	3	13	6.85	6.5	1.61
Benchmark for JonScript	**Min**	**Max**	**Mean**	**Median**	**Std Dev**
5000th fibonacci number	1128	1447	1224.30	1255.0	80.06
1000th Langtons ant step	125	302	152.89	147.5	26.39
Merge Sort (700 elements)	73	140	80.92	79.0	8.01

Table 4.2: Microbenchmark performance summary

4.2.3 Testing enviroment

These algorithms were tested on my personal computer with these specifications:

1. Processor: AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx $\times$ 4,
2. Random Access Memory (RAM): 8 GB 1600 MHz DDR3L SDRAM,
3. Physical memory: 1 TB 5400 rpm SATA SSHD, and
4. Operating system: Linux Mint 20 Cinnamon.

These tests were ran in NodeJS version 12.22.1 enviroment.

4.2.4 Results

In Table 4.2 you can see various averages measured during the test. These results represent times of run in milliseconds. There were 100 measured iterations for each test. For Langton's ant and Fibonacci, each of those results were for a hundred runs. This means that a 5000th fibonacci member, and the 1000th step of Langton's automaton, were computed a hundred times in each of those hundred tests runs. Merge sort ran once per iteration. All values in this table are in milliseconds.

In order to better understand the differences between the test subjects, I present three graphs, one for each test. The graphs are violin plots which show the distribution of execution times within each test. The Y-axis is the execution time. The X-axis is the tested code, either compiled from JonScript, or compiled from TypeScript. The shape of the violin describes the distribution of results from a single test in a single test subject. The graphs are to-scale

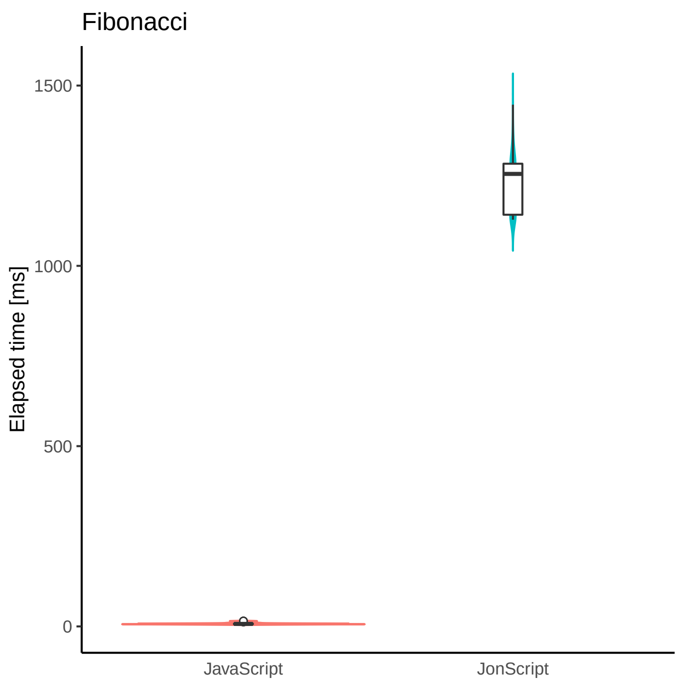


Figure 4.1: Fibonacci performance

with each other. The violin is widest in the median execution time for each test and test subject.

There are boxplots within each violin. They describe the distribution of results and provide additional information. In each boxplot, there is a horizontal line representing median execution time, which splits the dataset in two. There are two boxes, one above, and one below the horizontal line both of which extend to the lower and upper quartile respectively. Each of them ends at the median of the lower/upper half of the dataset. In each boxplot, there are whiskers, top and bottom. The top whisker shows the distribution between the upper half median and the maximum exection time. The bottom whisker shows the same information for lower half median and minimum execution time. The outliers are excluded from these and plotted separately with points per outlying execution time.

In Figure 4.1 you can see the execution times plot for the fibonacci sequence calculation algorithm. In Figure 4.2 you can see the execution times plot for the Langtons ant algorithm. In Figure 4.3 you can see the execution times plot for the merge sort algorithm.

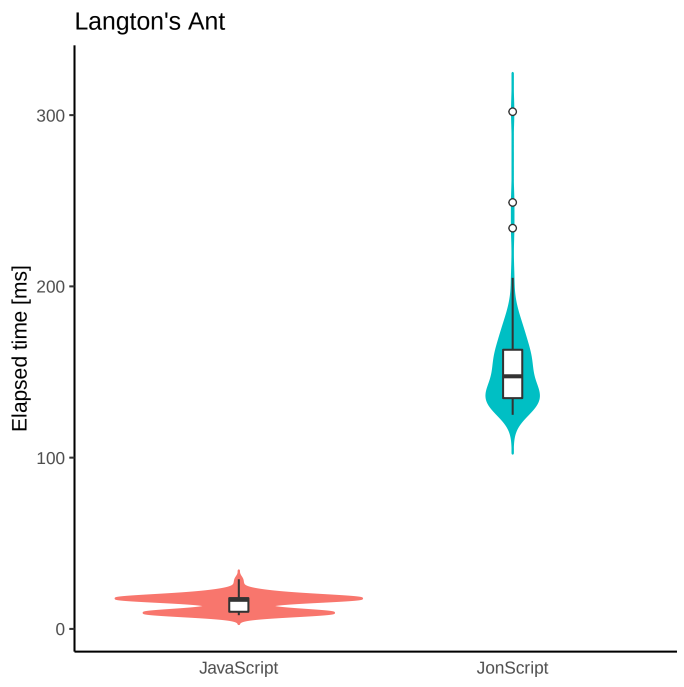


Figure 4.2: Langton’s ant performance

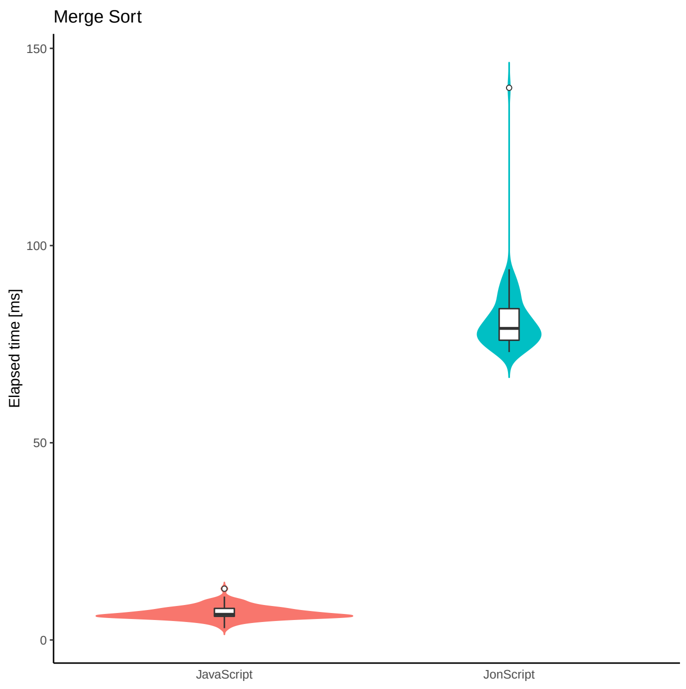


Figure 4.3: Merge sort performance

4.2.5 Analysis

Unfortunately, JonScript suffers severe overhead compared to TypeScript. In each of the tests run, the median time for TypeScript compiled JavaScript code execution was lower than the JonScript alternative. These measurements were relatively stable (e.g. low standard deviation compared to the median). The differences in mean for each algorithm ranged from 74.07 ms (for merge sort algorithm) to 1216.83 ms (for Fibonacci).

The deoptimization possibly stems from this list of suspect factors:

1. custom operator behaviour (custom type coercion),
2. function context safety,
3. automatic optional null chaining,
4. unification of null values (`null` and `undefined`), and
5. automatic checking and removal of `NaN` values.

Optimization The suspected problems that have caused the deoptimization will require further testing and profiling to ascertain whether or not solving them will improve the compiled JavaScript code from JonScript performance during run-time.

Chapter 5

Related Work

This chapter introduces other works developed in the field of programming languages aiming at ameliorating the idiosyncrasies of JavaScript. This chapter is split into three sections. The first section is dedicated strictly to the JavaScript strict mode, in order to discuss whether or not it can ameliorate the issues JonScript fixes. The second section deals with alternatives to JonScript and the third section deals with linters.

5.1 Strict mode

Strict mode[144] is a declaration which can be placed either into the global context or the function body. This statement works by limiting functionality of JavaScript to improve developer experience and limit severe errors within an application. Strict mode was introduced in ES5 [3]. The most important issues this mode fixes for JavaScript developers are: throwing errors when you assign a value to a variable that has not been defined, changing the value of function context of a function executed in global context from the `window`[145] object to the value `undefined`,[146] and throwing an error when you use the `delete`[147] operator on a prototype object[148] (e.g. remove the prototype).

While the features of strict mode are very useful in preventing bugs in JavaScript code, they do not address any of the issues that JonScript addresses, which were discussed at length in Chapter 1 (Introduction) and Chap-

[144]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode`

[145]`https://developer.mozilla.org/en-US/docs/Web/API/Window`

[146]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined`

[147]`https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/delete`

[148]`https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes`

ter 2 (Design). Furthermore, JonScript already uses strict mode automatically, due to its usage of TypeScript.

5.2 Languages and language extension

This section describes popular, interesting alternatives to JonScript and how they compare to what JonScript does. These alternatives either fix a portion of problems within JonScript, or they improve developer experience of JavaScript developers in similar way to JonScript.

5.2.1 TypeScript

TypeScript is a language, which is a superset of JavaScript. Therefore, all JavaScript code is valid TypeScript code. TypeScript extends JavaScript by providing compile-time typing, which does not influence the run-time and is executed in the browser by being compiled into JavaScript. The compiler gives the developer useful information by warnings and error messages in case of a typing mismatch.

TypeScript has powerful type inference system [29]. This is significant, since it allows one not to have to explicitly define function return types, object types, variable types and so on. As an improvement over normal TypeScript usage, JonScript makes it so that every object and prototype can have recursive type definitions. Normally, TypeScript would only allow this on class/prototype definitions.

TypeScript is an integral part of JonScript structure. JonScript utilizes TypeScript typing when importing packages, declaring methods, type-checking, using type inference and post-processing. So, where does JonScript add value over pure TypeScript? In the effects it has on run-time. TypeScript tries as hard as possible to keep developers from making mistakes—but JonScript lets you write intuitive code, extends the language possibilities by adding additional syntactic and semantic features—all of this, while still utilizing TypeScript.

5.2.2 CoffeeScript

CoffeeScript [12] aims to remove Java influence from the JavaScript language, leaving it with more similarities to a functional programming language.[149] To that end, it adds functional programming features within the language, such as re-creating ternary conditional operator with the keyword `if`. CoffeeScript is compatible with any JavaScript library and with the JSX syntax. CoffeeScript can be used with TypeScript (through the compiled-coffee package).[150]

[149] `https://www.npmjs.com/package/coffeescript`
[150] `https://www.npmjs.com/package/compiled-coffee`

The benefits of CoffeeScript are one-to-one compilation into JavaScript, easy syntax, and guaranteed compatibility. JonScript differs from CoffeeScript philosophy that it is more active in trying to change the underlying JavaScript behaviour—for example, JonScript uses its own, more consistent, operator type coercion—JonScript will not produce `"NaN"` string when working with invalid numerical operations (such as dividing zero by zero) and then concatenating the result with a string. JonScript also introduces function context change safeguards for all methods called within JonScript. Furthermore, JonScript provides automatic `new` and `async` keyword insertions.

5.2.3 JSX

While the JSX [19] syntax extension does not try to fix any of the problematic behaviour in JavaScript, it is still worth mentioning, since it does extend the JavaScript syntax in order to improve developer experience. JSX allows the developer to input HTML-like syntax into JavaScript code. This syntax is then compiled into JavaScript expressions. In essence, both JSX and CoffeeScript add syntactic benefits, while JonScript adds both syntactic and semantic benefits.

5.2.4 JS++

JS++ is a JavaScript language extension that imposes strict C++-like typing and does run-time coercion on values based on these types.[151] JS++ is similar to TypeScript, in that it uses a typing structure to improve developer experience. JS++ supports imperative, functional and object-oriented programming paradigms. While JS++ is similar to JonScript in how it aims to fix the JavaScript run-time problems, it can be argued that JonScript has more functional programming features.

5.2.5 Amber

Amber is a SmallTalk [24] dialect that can be compiled into JavaScript.[152] Since SmallTalk has comparatively small developer community,[153] it is more difficult to find people who work in it. While Amber fixes JavaScript problematic behaviour, it is not easy to use for current JavaScript developers—simply put, it differs too much in syntax.[154] JonScript keeps much of the JavaScript (or rather, TypeScript) syntax as-is. This allows easier switch from JavaScript

[151] `https://www.onux.com/jspp`

[152] `https://www.npmjs.com/package/amber`

[153] `https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages-loved`

[154] `https://www.gnu.org/software/smalltalk/manual/html_node/Defining-methods.html`

or TypeScript to JonScript. Furthermore, Amber is not a forgiving language—for example, when trying to concatenate `NaN` with a string, it throws an exception. This is arguably undesirable behaviour, since throwing an exception on such operation may break the website and not display any content to the end-user.

5.3 Linters

Linters [43] are code analysis tool that work by giving developers useful information about their code. They do so by recognizing patterns and keywords within code that may be dangerous. Linters use static code analysis to gain this information. Linters can be connected to an Integrated Development Enviroment (IDE) to provide additional syntax highlighting alongside useful advice within the IDE [42].

5.3.1 ESLint

ESLint is a popular JavaScript linter.[155] ESLint identifier problematic patterns within JavaScript code. For example, ESLint can be set up to flag any function definition that is not an arrow function expression to try to prevent function context errors. In this way, ESLint is similar to JonScript. However, JonScript goes a step beyond warnings—it simply fixes the issues both during compile-time and during run-time. Furthermore, JonScript is its own language—this means that JonScript can add more value than simply removing the bad parts of JavaScript—such as adding single-expression functors.

5.3.2 Other linters

The problematic nature of JavaScript semantics compelled many developers to create their own linters, different from ESLint. Similarly to ESLint, they only deliver warnings and do not improve the language. I mention them for completeness. They include JSLint,[156] JSHint,[157] JSCS,[158] and TSLint.[159]

[155]`https://www.npmjs.com/package/eslint`
[156]`https://jslint.com`
[157]`https://jshint.com`
[158]`https://jscs-dev.github.io`
[159]`https://palantir.github.io/tslint`

Chapter 6

Conclusion

This thesis identifies the problematic behaviour of JavaScript and as a lack of consistency within its API, and in subtle variations in its grammar with significant semantic effects. Furthermore, the thesis identified problems with JavaScript counterintuitive quasi-prototypal inheritance system, counterintuitive type coercion when using operators, such as concatenation, arithmetic operators and relational operators. These issues are not addressed by existing solutions. This thesis addresses these problems by introducing a new languages, called JonScript, which compiles into JavaScript, via TypeScript, and has these three guiding principles: consistency, compatibility with household frameworks in JavaScript, and ease of use.

The changes to JavaScript grammar and semantics include the universal use of arrow functions, automatic application of the `async` operator when the `await` operator is used, automatic application of the `new` operator when instantiating a class, and intuitive type coercion system when using operators. These changes make JonScript source code more consistent than JavaScript.

JonScript is rigorously tested for correctness and I have developed a case study project to showcase its compatibility with jQuery and Redux. JonScript is easy to use for both experienced JavaScript developers, since its syntax and semantics are similar enough to JavaScript, while its consistency makes it easy for new developers to start using it alongside the JavaScript veterans.

However, JonScript incurs an overhead within run-time due to safety features it implements. These features are:

- automatic optional chaining for constructor calls, property and element access and function calls,
- safety from unexpected changes in function context,
- improvements on JavaScript inheritance system,
- improvements on operator type coercion, and
- consistency when operating with null values.

Despite the overhead, an argument could be made that safety features which improve both developer and the end-user experience by avoiding common JavaScript pitfalls which can break an application are worth the overhead. Nevertheless, the future versions of JonScript will contain optimizations ameliorating these problems.

JonScript is distributed as a plugin for Webpack. It is publically available at NPM JonScript repository. The package has been downloaded 168 times at the time of writing.

6.1 Threats to validity

Due to TypeScript type system affecting the JonScript run-time, there is a possibility that an error in TypeScript typing would invalidate JonScript features. However, given that JonScript programmers have the freedom to import TypeScript files into JonScript, these problems can be addressed by fixing the problematic types in question—therefore, even if this threat occurs, it would not invalidate the thesis.

6.2 Unsolved problems and future work

Given the scope of development of JonScript and the time and resources I have at my disposal during the creation of this thesis, there are leftover issues not addressed in the language that will require future attention. This list of issues will be split into three parts: First, a list of issues that are not yet fixed/implemented due to lack of time and resources. Second, a list of issues that require future research, and third, a list of issues that are impossible to fix due to JavaScript functionality, or due to compatibility or run-time performance concerns.

6.2.1 Issues to be implemented in future release

Here is a list of issues that will be implemented in a future release of JonScript:

1. add support JSX syntax. JSX syntax allows developers to insert HTML-like syntax which will be compiled into JavaScript expressions to simplify application UI definition—for details, see Section 3.5.1,

2. unify type references to null-like values—types such as `null`, or `void` can still be defined in JonScript—for details see Section 3.5.1,

3. fix spread operator on array-like values—they should work as they do in regular JavaScript within spread operators, see Section 3.5.1,

4. fix literal string escape sequences, since they require more escape characters than the same expressions in JavaScript, see Section 3.5.1,

5. utilize map files to help JonScript developers with debugging see subsection—map files are used to map compiled JavaScript code onto another language it originated from—see Section 3.5.1,

6. develop a Visual Studio Code extension for JonScript developers, for syntax highlighting, debugging and code reference—see Section 3.5.1,

7. add syntax for prototype and method attributes—see Section 3.5.1, and

8. add regular expression literals—see Section 3.5.2.

6.2.2 Issues that require more research

Here is a list of issues that may be fixed in the future release, if possible, or that have multiple possible solutions and the correct approach to solving them requires more research:

1. automatic renaming of properties to protect against keyword conflicts—Section 3.5.2,

2. fix type errors concerning null references—Section 3.5.2,

3. fix JonScript prototype compatibility with the `is` operator—Section 3.5.2,

4. optimize the build process—Section 3.5.2, and

5. allow access to private properties within same-type classes—Section 3.5.2.

6.2.3 Performance

I created three different performance tests using benchmark algorithms to measure the difference between JavaScript code that was compiled from JonScript and JavaScript code that was compiled from TypeScript. JonScript have consistently performed worse than TypeScript in each of these benchmarks.

6.2.4 Issues that cannot be addressed

These issues cannot be fixed due to either JavaScript itself, concerns of compatibility, or concerns of heavily sub-optimal run-time:

1. cannot recursively sanitize imported objects to remove `null` and `NaN` values (see Section 3.5.3)—this is not practical to solve, since it would further decrease performance and potentially cause compatibility issues, and

2. cannot remove all subtle differences between non-object values and their class counterparts—this is not fixable due to how JavaScript engines treat false-like values (see Section 3.5.3).

6.3 Future work

The results of performance evaluation show that I need to optimize run time of JonScript. Furthermore, I will research optimization of compilation time, and implement more unit tests, improve documentation, and fix the aforementioned issues.

Bibliography

[1] How to create a function from a string in JavaScript. `https://www.geeksforgeeks.org/how-to-create-a-function-from-a-string-in-javascript/`, Oct 2019. [Online; accessed 3. May 2021].

[2] ECMA-262 - Ecma International. `https://www.ecma-international.org/publications-and-standards/standards/ecma-262`, Mar 2021. [Online; accessed 1. May 2021].

[3] ECMAScript Language Specification - ECMA-262 Edition 5.1. `https://262.ecma-international.org/5.1`, Jan 2021. [Online; accessed 1. May 2021].

[4] ECMAScript® 2022 Language Specification. `https://tc39.es/ecma262/#sec-abstract-relational-comparison`, Apr 2021. [Online; accessed 1. May 2021].

[5] ESLint User Guide. Configuring ESLint. `https://eslint.org/docs/user-guide/configuring`, May 2021. [Online; accessed 3. May 2021].

[6] Introduction to JavaScript Source Maps-HTML5 Rocks. `https://www.html5rocks.com/en/tutorials/developertools/sourcemaps`, Apr 2021. [Online; accessed 18. Apr. 2021].

[7] TypeScript 3.7 documentation. `https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html`, Apr 2021. [Online; accessed 18. Apr. 2021].

[8] Enrique Amodeo. JavaScript Horror Show. `https://eamodeorubio.github.io/thejshorrorshow`, Apr 2013. [Online; accessed 3. May 2021].

[9] Sammie Bae. JavaScript objects. In *JavaScript Data Structures and Algorithms*, pages 83–88. Springer, 2019.

[10] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[11] Prithvi Bisht and VN Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2008.

[12] Trevor Burnham. *CoffeeScript: accelerated JavaScript development.* Pragmatic Bookshelf, 2015.

[13] Mike Cantelon, Marc Harter, TJ Holowaychuk, and Nathan Rajlich. *Node.js in Action.* Manning Greenwich, 2014.

[14] Ivo Gabe de Wolff and Jurriaan Hage. Refining types using type guards in TypeScript. In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 111–122, 2017.

[15] Massimo DiPierro. The rise of JavaScript. *Computing in Science & Engineering*, 20(1):9–10, 2018.

[16] Richard Kenneth Eng. JavaScript cannot be fixed! - JavaScript Non Grata - Medium. `https://medium.com/javascript-non-grata/as-others-have-noted-the-fundamental-problem-with-web-development-is-that-javascript-is-a-broken-7f9675048c77`, Mar 2018.

[17] Yakov Fain and Anton Moiseev. *Angular 2 Development with TypeScript.* Manning Publications Company, 2017.

[18] Cory Gackenheimer. *Introduction to React.* Apress, 2015.

[19] Cory Gackenheimer. JSX Fundamentals. In *Introduction to React*, pages 43–64. Springer, 2015.

[20] Anahi Gajardo, Andre Moreira, and Eric Goles. Complexity of Langton's ant. *Discrete Applied Mathematics*, 117(1-3):41–50, 2002.

[21] Mariana Goranova, Elena Kalcheva-Yovkova, and Stanimir Penkov. Task-based asynchronous pattern with async and await. In *International Scientific Conference Computer Science*, page 150, 2015.

[22] Caleb Hattingh. *Using Asyncio in Python: Understanding Python's Asynchronous Programming Features.* O'Reilly Media, Inc., 2020.

[23] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[24] John Hunt. *Smalltalk and object orientation: an introduction.* Springer Science & Business Media, 2012.

[25] Sebastian Kleinschmager, Romain Robbes, Andreas Stefik, Stefan Hanenberg, and Eric Tanter. Do static type systems improve the maintainability of software systems? An empirical study. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 153–162. IEEE, 2012.

[26] Brian Kovacs. Why JavaScript Sucks! (for now...). *Medium*, Sep 2018. `https://medium.com/@briankovacs/why-javascript-sucks-for-now-6bd30de6eafc`.

[27] Alex Kyriakidis and Kostas Maniatis. *The Majesty of Vue. js.* Packt Publishing Ltd, 2016.

[28] David Luecke. JavaScript — The weird parts. *Medium*, May 2018. `https://medium.com/@daffl/javascript-the-weird-parts-8ff3da55798e`.

[29] Dan Maharry. *TypeScript revealed.* Apress, 2013.

[30] John McCarthy. History of LISP. In *History of programming languages*, pages 173–185. 1978.

[31] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *European Conference on Object-Oriented Programming*, pages 355–382. Springer, 1998.

[32] Jamie Munro. *Knockout. JS: building dynamic client-side web applications.* O'Reilly Media, Inc., 2014.

[33] Chris Nwamba. Declaring JavaScript Variables: var, let and const. `https://scotch.io/courses/10-need-to-know-javascript-concepts/declaring-javascript-variables-var-let-and-const`, May 2021. [Online; accessed 3. May 2021].

[34] Martin Odersky et al. The Scala programming language. 2008. `https://www.scala-lang.org`.

[35] Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. A study and toolkit for asynchronous programming in C#. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1117–1127, 2014.

[36] Terence Parr. *The definitive ANTLR 4 reference.* Pragmatic Bookshelf, 2013.

[37] Dmitri Pavlutin. 5 Differences Between Arrow and Regular Functions. *Dmitri Pavlutin Blog*, Mar 2021. `https://dmitripavlutin.com/differences-between-arrow-and-regular-functions`.

[38] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165, 2014.

[39] Ashutosh K. Singh. Three Different Ways to Create Objects in JavaScript. *Medium*, Sep 2020. `https://betterprogramming.pub/three-different-ways-to-create-objects-in-javascript-d3595d693296`.

[40] Ralph Steyer. Razor–Syntax und View Engine. In *Webanwendungen mit ASP. NET MVC und Razor*, pages 41–46. Springer, 2017.

[41] Antero Taivalsaari. Classes vs. prototypes - some philosophical and historical observations. In *Journal of Object-Oriented Programming*, pages 44–50. SpringerVerlag, 1996.

[42] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie van Deursen. Why and how JavaScript developers use linters. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 578–589. IEEE, 2017.

[43] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. The adoption of JavaScript linters in practice: A case study on ESLint. *IEEE Transactions on Software Engineering*, 46(8):863–891, 2018.

[44] Liam Tung. Programming language popularity: JavaScript leads—5 million new developers since 2017. `https://www.zdnet.com/article/programming-language-popularity-javascript-leads-5-million-new-developers-since-2017/`.

[45] Kathy Walrath. *The Java Tutorial.* Addison-Wesley, Feb 1996.

[46] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11. IEEE, 1973.

[47] Allen Wirfs-Brock and Brendan Eich. Javascript: the first 20 years. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–189, 2020.

[48] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 351–361, 2016.

[49] Richard Wyatt. Understanding functional programming. *Journal of Computing Sciences in Colleges*, 18(5):109–117, 2003.

[50] Nicholas C Zakas. *Understanding ECMAScript 6: the definitive guide for JavaScript developers.* No Starch Press, 2016.

Appendix

A JonScript ANTLR4 Syntax

```
TRY: 'try';
CATCH: 'catch';
FINALLY: 'finally';
THROW: 'throw';
IMPORT: 'import';
EXTENDS: 'extends';
EQUALS: '=';
IS: 'is';
AS: 'as';
FROM: 'from';
ARROW: '=>';
INHERITS: '...';
QMARK: '?';
COLON: ':';
LIST_BEGIN: '[';
LIST_END: ']';
EVAL_BEGIN: '(';
EVAL_END: ')';
OBJECT_BEGIN: '{';
OBJECT_END: '}';
COMMA: ',';
ATOMIC: NUMBER | STRING | BOOLEAN | NIL;
NIL: 'nil';
BOOLEAN: 'true' | 'false';
NUMBER:
    NEGATIVE_NUMBER
    | NATURAL_NUMBER
    | DECIMAL_NUMBER
    | NEGATIVE_DECIMAL_NUMBER
    | ZERO
    | INFINITY;
fragment ZERO: '0';
fragment NEGATIVE_NUMBER: '-' [1-9][0-9]*;
fragment NATURAL_NUMBER: [1-9][0-9]*;
fragment DECIMAL_NUMBER: [0-9]+ '.' [0-9]+;
fragment NEGATIVE_DECIMAL_NUMBER: '-' [0-9]+ '.' [0-9]+;
fragment INFINITY: 'Infinity';
SINGLE_COMMENT: '//' .*? NEWLINE -> skip;
MULTILINE_COMMENT: '/*' .*? '*/' -> skip;
NEWLINE: ('\n' | '\r\n') -> channel(HIDDEN);
SPACE: ('\t' | ' ') -> channel(HIDDEN);
fragment ESC_STRING: ('\\\\')* '\\"';
STRING: '"' (ESC_STRING | ~'"')* '"';
INTERPOLATED_SEQUENCE_EMPTY: '``';
fragment ESC_DOLAR: ('\\\\')* '\\$';
fragment ESC_OBJECT_BEGIN: ('\\\\')* '\\{';
fragment ESC_OBJECT_END: ('\\\\')* '\\}';
fragment ESC_INTERPOLATED: ('\\\\')* '\\`';
fragment INTERPOLATED_SEQUENCE_CONTENTS: (
        ESC_DOLAR
        | ESC_OBJECT_BEGIN
        | ESC_OBJECT_END
        | ESC_INTERPOLATED
        | ~('`' | '$' | '{' | '}')
    );
INTERPOLATED_SEQUENCE_STRING:
    INTERPOLATED_SEQUENCE_BORDER INTERPOLATED_SEQUENCE_CONTENTS*? INTERPOLATED_SEQUENCE_BORDER;
INTERPOLATED_SEQUENCE: (
        INTERPOLATED_SEQUENCE_BORDER
        | OBJECT_END
    ) INTERPOLATED_SEQUENCE_CONTENTS*? (
        INTERPOLATED_SEQUENCE_BORDER
        | INTERPOLATED_SEQUENCE_EXPR_START
    );
fragment INTERPOLATED_SEQUENCE_BORDER: '`';
fragment INTERPOLATED_SEQUENCE_EXPR_START: '${';
LOGICAL_OPERATORS: '&&' | '||';
MULTIPLYDIVIDEMOD_OPERATORS: '*' | '/' | '%';
PLUSMINUS_OPERATORS: '+' | '-';
EQUALITY: '==';
NEQUALITY: '!=';
EQUAL_OR_GREATER: '>=';
EQUAL_OR_LESS: '<=';
LESS_THAN: '<';
MORE_THAN: '>';
TYPE_OPERATORS: TYPE_OR | TYPE_AND;
fragment TYPE_OR: '|';
fragment TYPE_AND: '&';
OBJECT_OPERATORS: '.';
PREFIX_UNARY_OPERATORS: '!' | 'await';
IGNORE: '_';
VAR: [a-zA-Z0-9$]+;

parent: imported* module;

importVar: VAR | (VAR AS VAR);
defaultImport: VAR;
```

```
imported:
    IMPORT (
        defaultImport
        | (OBJECT_BEGIN (importVar COMMA)* importVar OBJECT_END)
    ) FROM ATOMIC;

module: VAR OBJECT_BEGIN classdef* OBJECT_END;

classdef:
    VAR template? (
        EVAL_BEGIN (ignoreableVar COMMA)* (
            INHERITS? ignoreableVar
        )? EVAL_END object
    );

assignment: (VAR COMMA)
    | ((VAR | IGNORE) COLON (object | expression) COMMA);
varAssignment: (VAR COMMA)
    | (VAR COLON (object | expression) COMMA);
singleVarAssignment: VAR | (VAR COLON (object | expression));
publicObject:
    OBJECT_BEGIN (
        functor
        | singleVarAssignment
        | ((functor COMMA)? varAssignment*)
    ) OBJECT_END;
publicApi: (publicObject | expression);
inheritance: INHERITS expression COMMA;

object:
    OBJECT_BEGIN assignment* inheritance* publicApi?
    OBJECT_END;
functor: (
        VAR
        | (
            template? EVAL_BEGIN (ignoreableVar COMMA)* (
                INHERITS? ignoreableVar
            )? EVAL_END (COLON type)?
        )
    ) ARROW (object | expression);

ignoreableVar: (VAR QMARK? (COLON type)? (EQUALS expression)?)
    | IGNORE;

template:
    LESS_THAN (templateContents COMMA)* templateContents
        MORE_THAN;
templateContents: VAR extendsType? equalsType?;
extendsType: EXTENDS type;
equalsType: EQUALS type;

type:
    VAR
    | type LIST_BEGIN LIST_END
    | typeEval
    | tuple
    | type op = TYPE_OPERATORS type
    | (VAR | accessType) templateType
    | ATOMIC
    | functionType
    | objectType
    | accessType;
accessType: VAR OBJECT_OPERATORS (accessType | VAR);
typeEval: EVAL_BEGIN type EVAL_END;
tuple: LIST_BEGIN (type (COMMA type)*)? LIST_END;
templateType: LESS_THAN type (COMMA type)* MORE_THAN;
paramType: VAR QMARK? COLON type;
functionType:
    template? EVAL_BEGIN (paramType (COMMA paramType)*)? EVAL_END
         ARROW type;
objectType:
    OBJECT_BEGIN (
        objectParamType (COMMA objectParamType)* COMMA?
    )? OBJECT_END;
objectParamType: VAR QMARK? COLON type;

expression:
    ATOMIC
    | VAR templateType?
    | functor
    | expression obj = OBJECT_OPERATORS expression
    | expression obj = OBJECT_OPERATORS (
        THROW
        | IMPORT
        | AS
        | IS
        | TRY
        | CATCH
        | FINALLY
        | EXTENDS
        | FROM
    )
    | expression EVAL_BEGIN expressionList? EVAL_END
    | expression LIST_BEGIN expression LIST_END
    | eval
    | LIST_BEGIN expressionList? LIST_END
    | PREFIX_UNARY_OPERATORS expression
    | INTERPOLATED_SEQUENCE_EMPTY
    | INTERPOLATED_SEQUENCE_STRING
    | INTERPOLATED_SEQUENCE interpolated? INTERPOLATED_SEQUENCE
    | expression op = MULTIPLYDIVIDEMOD_OPERATORS expression
    | expression op = PLUSMINUS_OPERATORS expression
    | expression cmp = (
        EQUAL_OR_GREATER
        | EQUAL_OR_LESS
        | LESS_THAN
        | MORE_THAN
    ) expression
    | expression cmp = (EQUALITY | NEQUALITY) expression
    | expression logic = LOGICAL_OPERATORS expression
    | expression IS expression
    | THROW expression
    | object
    | TRY expression CATCH expression (FINALLY expression)?
    | expression QMARK expression COLON expression;

eval: EVAL_BEGIN expression EVAL_END;
interpolated: (INTERPOLATED_SEQUENCE | expression) interpolated?;
expressionList: expressionItem (COMMA expressionItem)* COMMA?;
expressionItem: (INHERITS? expression);
```

B Contents of enclosed USB drive

```
root
├── README.md ..... Contents description and build process for JonScript
├── src ......................................... JonScript source code
├── thesis.pdf ..................................... The thesis PDF
└── latex-source ........................... The thesis LaTeX sources
```