Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Understanding the Adoption Trends of JavaScript Modern Features

Rafael Campos Nunes

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2023

Understanding the Adoption Trends of JavaScript Modern Features

Abstract—
JavaScript is a versatile programming language conceived in the 1990s. Besides its somewhat long history, it continues to exert a profound and enduring influence, empowering websites with dynamic capabilities through its interactions with browsers and rendered documents. In the last decade, though, its scope extends far beyond the web, finding utility in backend development, desktop applications, and IoT devices. To circumvent the needs of modern programming, JavaScript has undergone a remarkable evolution since its inception, with the groundbreaking release of its sixth version in 2015, introducing a plethora of new features and establishing an annual versioning system. While the adoption of new JavaScript features promises numerous benefits to developers and their projects, the process of integrating them into existing codebases poses a persistent challenge. This process requires a judicious assessment of project requirements, compatibility with targeted engines (such as NodeJS, Web browsers, and the like), and the potential advantages they confer. However, the strategies employed by developers to effectively incorporate these features into their projects remain elusive. To shed light on the prevailing trends, we present the results of a comprehensive software mining repository study that aims to characterize the trends in the adoption of modern JavaScript features. After mining the source code history of 100 JavaScript open-source projects, we find extensive use of JavaScript modern features like Arrow Function Declarations, Async Declarations, Const Declarations, Let Declarations, and Object Destructuring present in more than 80% of our dataset. Our findings also reveal that (a) the widespread adoption of modern features happened between one and two years after the release of the version in 2015 and (b) a consistent trend toward increasing the adoption of modern JavaScript language features in open-source projects.

Index Terms—**Software Engineering, Modern language features, Rejuvenation, Software Evolution, Software Repository Mining**

I. Introduction

The World Wide Web is accessible by almost any imaginable device and comes in excellent availability, altogether with ease to develop in this platform. The technologies used to create the Web might differ from when it started, but the JavaScript language, born in the nineties, was initially conceived to increase the interactivity on web pages. Currently, JavaScript is used far more than just the Web, and there are JavaScript implementations that fulfil the needs for a wide range of fields, from backend and desktop development to emergent technologies such as smart contracts and IoT. JavaScript is also the target platform for languages such as Reason and TypeScript, and renderer frameworks such as Vue.js, ReactJS, and so many more [10].

To embrace the many different scenarios of use, the JavaScript language has changed frequently since its first release. Indeed, the first JavaScript standard appeared as an ECMA specification in June 1997 (ECMA-262) [7]. The sixth version of the language (ECMAScript 6) is considered a breakthrough for the JavaScript language evolution, bringing many modern program language features such as classes, arrow functions, and promises [17]. After the release of ECMAScript 6, new versions of the language are being released on early bases.

Adopting modern JavaScript features brings numerous benefits to developers, including expressiveness and security [8]. Nonetheless, developers face the challenge of incorporating these features into legacy codebases because adapting to the newest JavaScript features entails a careful trade-off involving project requirements, potential benefits of modern JavaScript features, and compatibility with the targeted browsers and runtime engines, to cite a few. While rejuvenating old code to leverage the improved readability and performance of new features can be time-consuming, it might be worthwhile in the long run —which motivates our general research question: "How modern JavaScript features have been introduced, championed or ignored in existing legacy code?" [14]. Similar research questions have been explored before, though focusing on Kotlin [12] and Typescript [16], for instance. The Kotlin [12] study analyzed 26 language features on 387 source code repositories and the Typescript [16] study analyzed 13 features on 454 source code repositories. Surprisingly, regardless of the popularity of the JavaScript language, just a few studies have explored how specific features of the language are being used [10].

To fulfil this gap, our research identified the adoption of 15 modern JavaScript features on 98% of 100 JavaScript open-source project repositories we mined from GitHub and found it took between one and two years from the 2015 version release to the modern features be widely used by developers on their projects. Detailed results are presented in Section IV.

The main contributions our work makes are the following:

- A study presenting the results of the adoption of modern JavaScript features in the development of projects over time.
- Provide the programming language community with valuable insights into the dynamics of language feature adoption, thereby informing analogous studies pertaining to JavaScript features.
- Provide insights into the usage patterns of JavaScript features, highlighting the prevalence and importance of

specific features in modern JavaScript development.

II. JavaScript Evolution

The ECMA Script, known worldwide as JavaScript, is a language initially designed for the Internet. Although it is widely used in many other fields of software development besides the web, it was not created with that intent in the beginning. Specifically, it was born in an era when Web browsers were still giving their first steps. Brendan Eich tailored the crafting of the JavaScript language, which was called Mocha back then. Soon after the first release on the Netscape Navigator, it followed what was called LiveScript. The third development iteration was named JavaScript, as it borrowed some syntactic features from the Java programming language [17].

To relieve everyone of the problems of having different JavaScript flavours, the Netscape company asked the European Computer Manufacturers Association (ECMA) to standardize the JavaScript syntax and semantics as a general-purpose, cross-platform, vendor-neutral scripting language. This effort led to the ECMA-262 standard, and other companies that created browsers could use it to implement JavaScript [11]. Nowadays, JavaScript is ubiquitous and has significantly changed in these two decades of development. There are 13 versions of the language, which is broadly used to develop frontend and backend components of Web, mobile and desktop applications, ranging from simple CRUD applications to more sophisticated ones such as game interfaces. On the backend, JavaScript is used with a runtime (as NodeJS) that exposes APIs commonly found in other programming languages, allowing programmers to implement systems for domains with highly demanding software requirements (such as finance). Next, we trace the evolutionary history of JavaScript, highlighting the key features that have propelled JavaScript's growth and widespread adoption.

A. Initial Versions of JavaScript

The initial version of JavaScript, based on the ECMAScript 1 standard, introduced fundamental programming features like variables, functions, and conditional statements [17]. In the subsequent versions, ECMAScript versions 2 and 3, the language was extended to improve error handling, enhance the support for regular expression, and introduce new array manipulation capabilities [17]. The version 4 was never published as they stalled its development because of internal discussions about the direction of the language [17].

```
function f(arr) {
  if (arr.length > 2)
    throw 'Exception Exception';
}

var arr = [1,2,3,4];

try {
  f(arr);
} catch (e) {
  console.log('thrown exception: '+ e);
}
```

JavaScript uses exceptions and the try/catch statement blocks to handle errors that might occur when executing code.

The ECMAScript 5 standard was released in 2009 and underwent notable enhancements. ECMAScript 5 introduced new built-in objects and methods, improving string manipulation, JSON handling, object-oriented programming features, and array manipulation methods, such as `reduce()`, `filter()`, and `map()` [15], [17]. ECMAScript 5 represented a significant advancement by introducing the strict mode, which improved error checking and established a more predictable execution environment [8] by not letting programmers use undeclared variables in code.

To exemplify some of the built-in methods and the new JSON functionality look at the following piece of code.

```
var o = {"n": "Leafar", "s":["Senun", "Sopmac"]}
var so = '{"n": "Leafar", "s":["Senun", "Sopmac"]}';

var parsedObject = JSON.parse(so);
var stringifiedObject = JSON.stringify(o);

function addExpression(s) {
    var r = s + ' expression';

    return r;
}

// Array["Senun expression", "Sopmac expression"]
o.s.map(addExpression);
```

The previous code shows the usage of the JSON language object that allows the developer to easily transform an object into a string and vice-versa. Given that JSON is one of the most used standard for exchanging information across web services it is a significant addition to the language. It also features the map function that executes the passed function to each argument of the array, resulting in another array with different elements.

B. Modern JavaScript

The ECMAScript 6 standard was released in 2015 and introduced modern programming features, such as: `for/of` loop and iterable objects, template literals, rest parameters that allow functions to be invoked with an indefinite number of arguments as an array, and a new way to define functions known as arrow functions; all of this enhances the language's object-oriented programming capabilities [8], [17]. Moreover, it introduced modules, providing a modular approach to code organization and promoting code reuse [15]. These additions improved code readability, contextual separation of code, and provided developers with more expressive syntax. ECMAScript 6 also included the `let` and `const` keywords for block-scoped variables, destructuring assignments for concise variable assignments, improved iterators and generators to facilitate advanced control flow, and `Promises`, a mechanism for handling asynchronous operations [8], [15]. The changes of this version were so substantial that the term "Modern JavaScript" is considered any code composed with post-ES6 syntax [13]. In the following excerpt of code it shows some features of the ECMAScript 6.

```
'use strict';

let f = (x) => {
    return x * 2;
}

let e = [1,2,4,5,3];

e.map(f); // results in [2,4,8,10,6]
```

In the previous code we can see the usage of the strict mode, let assignments, anonymous functions (arrow functions) and also the new built in function map. It shows the new capabilities of the language, providing more succinct code.

The ECMAScript versions 7 to 11, from 2016 to 2020, were published on a new yearly publication cycle proposed by Technical Committee 39 (TC39), the committee responsible for the standardization of the ECMAScript programming language, and they continued to refine existing features and introduce new constructs [17]. On 2017, ECMAScript 7 introduced the `includes()` method for array search values, the exponentiation operator, and the `async/await` keywords, simplifying asynchronous programming and enhancing code readability [8]. ECMAScript 2018 brought spread operators to copy the properties of an existing object into a new object, and added the `for/await` loop to work with the new asynchronous iterator [8]. Version 2019 (ECMAScript 10) introduced new functions for array manipulation as `flat` and `flatMap` on `Array.prototype` for flattening arrays, and a few minor updates to syntax and semantics [8], bringing even more functional programming features into the language. The ECMAScript 11 was released in 2020, introducing the `BigInt` data type and the global object `This`, the nullish coalescing operator, for providing default values, and optional chaining, for safe property access, simplifying common programming patterns and providing more concise syntax [8].

The ECMAScript 12 brought numeric separators, to improve program readability by allowing the use of underscore characters as separators within numeric literals; the `replaceAll()` method to simplify string manipulation by providing a direct means to replace all instances of a substring; the inclusion of `WeakRef` and `FinalizationRegistry`, to effective management of weak references and cleanup operations by the garbage collector; the `any` Promise combinator, that resolves as soon as any promise in an array resolves, offering a streamlined approach to handling multiple asynchronous operations; and the logical assignment operators `??=, ||= and &&=`, enabling concise assignment based on logical conditions. [2]

ECMAScript standard version 13 introduce the top-level await feature, enabling parent modules wait for the child modules to execute first. New class elements were also introduced, including visibility modifiers and static fields and methods, to reinforce encapsulation and data privacy within classes. ECMAScript 13 also introduced the `cause` property on `Error` objects, allowing the recording of causation chains in errors. Another addition is the `at` method for `Strings, Arrays, and TypedArrays`, enabling relative indexing. Furthermore, static blocks within classes was added, facilitating per-class evaluation initialization. Lastly, an additional flag `d` to regular expressions was included, providing matched indexes to indicate the start and end positions of a matched string [2]

III. Study Settings

The study aims to understand how JavaScript developers adhere to the newly released features of the language. Specifically, we look at a subset of changes in the ECMAScript 6th standard and also two features from the ECMAScript 8th, particularly *async* and *await*. We chose to start the study with the 6th standard because it is considered to be the modern JavaScript [13] as it not only updates the language with new features but also adds new terminology and semantic variations to the standard itself [17]. In order to execute this study we mine the source code history of open-source JavaScript projects with a tailored tool developed by the team and investigate the following research questions.

- (RQ1) To what extent do JavaScript repositories rely on features from the ES6 standard?
- (RQ2) When did JavaScript developers start using features from the ES6 standard?
- (RQ3) Is there any trend in the adoption of modern JavaScript features?

The first question would answer to what extent the modifications of the language are being adopted by developers and if those modifications are meant for the development of JavaScript applications. For instance, arrow functions give the developer a lot of expressiveness and in JavaScript, where functions are first-class citizens, one would use it rather extensively. From another perspective, classes might not be important as JavaScript already have a mechanism that provides encapsulation using prototypes.

Besides having the quantitative answer for how much a feature from the standard is being used, it is important to know when such modifications began to take place. This answers how confident developers are to switch from an already established solution to something that might be more secure or take less time to write. The usage of scoped variables, for instance, is one of the instances where developers might jump in early as JavaScript provides a function bases hoisting that, because of its browser nature — and now beyond browsers —, might confuse a developer as it isn't quite clear where or to who a variable might be visible.

Lastly, JavaScript is known for its rapid iteration cycle when developing applications but how that reflects on feature adoption by developers? The answer to the third question might show us if there was an effort to update code bases and when such effort took place.

A. Dataset

The dataset containing the 100th most rated JavaScript repositories was selected using the SEART tool [6]. The parameters of the search were JavaScript projects whose number of commits was more than 1000 (one thousand), the creation date to be before 2012 with at least one update on January of 2023. After having the list of repositories to be studied we used a script to download all of them into a folder in May and subsequently ran our tool to extract all the features displayed on Table II. The dataset accounts for approximately 1.2GB of data and that number represents a surface analysis of 32069 JavaScript files summing up to more than 5 million lines of code. A random peek into the dataset provides the following insights from cloc.

TABLE I
RANDOM PEEK INTO THE PROJECT DATASET

Project Name	Files	Lines of code
eslint	1297	364064
chart.js	632	54028
svg.js	169	18186
jsdoc	500	20980
mocha	136	24868
shields	1223	73910
meteor	1340	198926

B. Data Collection

We developed JSMiner, a tool that traverses the source code history of projects and collects the usage scenarios of the features we are interested in (see Table II). JSMiner uses a JavaScript ANTLR grammar [1] to generate a Java software language engineering infrastructure for analysing JavaScript programs.

TABLE II
JAVASCRIPT FEATURES WE COLLECT BY ANALYSING THE SOURCE CODE HISTORY OF JAVASCRIPT OPEN-SOURCE PROJECTS.

Feature Name	Release	Date
let	ECMAScript 6	2015
const	ECMAScript 6	2015
arrow functions	ECMAScript 6	2015
classes	ECMAScript 6	2015
destructuring	ECMAScript 6	2015
promises	ECMAScript 6	2015
modules	ECMAScript 6	2015
generators	ECMAScript 6	2015
default parameters	ECMAScript 6	2015
spread and rest parameters	ECMAScript 6	2015
async and await	ECMAScript 8	2017

After doing the necessary modifications on the grammar files, we use ANTLR to generate a lexer, a parser, and a visitor that we can use to traverse the JavaScript files for a given revision of a project and collect the information we need. We implemented a suite of test cases to confirm that our parser correctly recognizes JavaScript files and contains the usage scenarios of the features. We then proceed with the execution of the tool against the project revisions in our dataset (Section III-A). The traversal process uses the Visitor pattern in the Abstract Syntax Tree (AST). While traversing a project, JSMiner collects the usage of each feature and outputs this information in a CSV file that it contains between revisions.

JSMiner is designed not to go over every commit, but to aggregate changes over some interval — the period we use in this research is seven days. JSMiner goes from the first commit on the previously defined date and then steps seven days forward aggregating every change (commits) in that interval and analyzing the changes in features usage.

The algorithm used to extract and count features is simple. Given a root directory JSMiner will look for Github folders and for each folder it will create a set of commits to be analysed going from an initial interval to an end interval defined when executing the program. Having the list of commits (data points) it starts checking in on each of them and traversing every JavaScript file within the folder for features we are interested in, sequentially it counts each feature and possible errors that occur when executing the parser on a file. The algorithm uses concurrency to accelerate the parsing step using all available processors available in the system (logical and physical).

C. Data Analysis

To answer those three questions we use a combination of quantitative analysis and also a conservative heuristic to determine whether or not there's a trend in the adoption of modern JavaScript features. More specifically, we use the linear regression technique to derive the answer to the third question.

The quantitative analysis uses a ratio of used features (see Table II) per line of code on every repository in this study and it also accounts for the first occurrence of a given feature on the code base. We also rely on using a central tendency metric called median for each feature to verify if their usage is constant across the entire project.

To evaluate feature usage patterns in the studied dataset we used a mathematical tool called regression [18], associating time and feature usage as variables. This part of the study uses an R script that we developed to ingest the output of JSMiner (a time series in CSV). Another way that we used to evaluate why patterns present themselves in a particular manner in the code base is to manually verify commits at specific points of time to find, for instance, the reason why the usage percentage of a particular feature has dropped over time.

D. Environment

The developed tool is compiled into a JAR (Java ARchive) using the OpenJDK [4] on version 19.0.2 and Maven [3]. The resulting file is then executed using the JRE (Java Runtime Environment) that accompanies the JDK (Java Development Kit) from OpenJDK.

IV. RESULTS

This section describes the results of an analysis of every JavaScript project in our dataset. First, our analyses include the adoption of modern JavaScript features by projects, the frequency of the feature across all projects, and the date the feature appeared for the first time in our dataset (Section IV-A). Section IV-B presents the adoption trends for modern JavaScript features in open-source software repositories.

A. Summarizes results for all repositories

This section presents the results of the analysis conducted on various features in terms of projects' adoption, occurrences, and first occurrence dates. The table III below provides a summary of the findings:

TABLE III
PERCENTAGE OF APPLICATIONS THAT USE A FEATURE, NUMBER OF OCCURRENCES, AND FIRST OCCURRENCE DATES.

Features	Projects Adoption (%)	Occurrences (#)	First Occurrence
Array Destructuring	52	1015	2012-02
Arrow Function Declarations	98	358711	2012-02
Async Declarations	89	35289	2012-02
Await Declarations	65	20345	2015-03
Class Declarations	59	2243	2014-01
Const Declarations	92	142689	2012-02
Default Parameters	68	2515	2015-02
Export Declarations	50	7797	2013-02
Import Statements	61	23262	2013-02
Let Declarations	88	39899	2013-02
Object Destructuring	80	9446	2012-02
Promise All() and Then()	46	694	2012-03
Promise Declarations	65	1522	2012-11
Rest Statements	55	1272	2015-03
Spread Arguments	60	1733	2015-03

The analyzed projects show high adoption rates for several features. Specifically, the features Arrow Function Declarations, Const Declarations, Async Declarations, Let Declarations, and Object Destructuring are adopted in 98%, 92%, 89%, 88%, and 80% of the projects, respectively. In our dataset, the usage of Default Parameters, Await Declarations, Promise Declarations, Import Statements, and Spread Arguments ranges from 60% to 68% across projects. Furthermore, the features Class Declarations, Rest Statements, Array Destructuring, Export Declarations, and Promise All() and Then() are adopted by 46% to 59% of the projects in our dataset. These findings highlight the widespread acceptance and utilization of these features among the analyzed projects.

The median distributions of the analyzed JavaScript features vary widely. Arrow function declarations have a median distribution of 1694, while const declarations have a median of 624. Other features, such as async declarations and let declarations, have median distributions of 137 and 126, respectively. These findings demonstrate the non-uniform adoption patterns and usage of different JavaScript features across our dataset. However, the Jquery-ui project alone contains 29628 occurrences of Arrow Function Declarations (8.26% of the total). We observe the same lack of uniformity for Const Declarations and Let Declarations.

RQ1: To what extent do JavaScript systems rely on modern JavaScript features?

Our findings suggest that JavaScript developers extensively use Async Declarations, Const Declarations, Arrow Function Declarations, Let Declarations, and Object Destructuring in their projects. These features are widely adopted, with adoption rates of 98%, 92%, 89%, 88%, and 80% respectively in the projects analyzed. Moreover, features such as Default Parameters, Await Declarations, Promise Declarations, Import Statements, and Spread Arguments are employed in a significant portion, ranging from 60% to 68% across the dataset. In contrast, Class Declarations, Rest Statements, Export Declarations, and Promise All() and Then() exhibit moderate adoption rates, ranging from 46% to 59% in the analyzed projects. These findings provide insights into the usage patterns of JavaScript features, highlighting the prevalence and importance of specific features in modern JavaScript development.

Next, we examined the first occurrences of a set JavaScript features to gain insights into their timeline of introduction. The earliest feature to appear was Array Destructuring, which first emerged in February 2012. It was followed by Arrow Function Declarations and Const Declarations, which also made their debut in February 2012. The features Async Declarations, Export Declarations, Import Statements, and Object Destructuring were introduced a month later in March 2012. Subsequently, the year 2013 witnessed the arrival of Let Declarations, Promise Declarations, and Import Statements in February, while Export Declarations was first observed in July. The features Await Declarations and Rest Statements came into existence in March 2015. Notably, Class Declarations and Promise All() and Then() were introduced earlier in January 2014 and March 2012, respectively. These findings shed light on the historical timeline of JavaScript features within our dataset, providing information about their initial appearances.

The first appearance of some of the features occurred between two and three years before the actual release of ES2015 (ES6) and ES2017 (ES8). The reason for that is because the majority of features were supported by browsers before the release of ES6 and ES8. For example, on June 25, 2013, Arrow Function Declarations was released on Firefox and Firefox for Android. Other features, such as Const Declarations, were added to Safari in 2011 and Opera in 2006.

RQ2: When did JavaScript developers start using modern JavaScript features?

Our findings suggest that JavaScript developers embraced modern JavaScript features like Arrow Function Declarations, and Async Declarations before their official standardization, indicating an early adoption and utilization of these language enhancements in JavaScript projects. Furthermore, other modern JavaScript features such as Const Declarations, Let Declarations, Object Destructuring, Default Parameters, Await Declarations, Promise Declarations, Import Statements, Spread Arguments, Class Declarations, Rest Statements, Export Declarations, and Promise All() and Then() are widely adopted between 1-2 years after the introduction of the feature in a new release JavaScript version.

B. *Trends of features adoption*

To estimate the Trends for our features, we conduct a regression analysis on our time series-based dataset using the tslm function available in the Forecast R package. Figure 1 present results of the regression analysis indicate a statistically significant upward trend for the adoption of the features in our study (Const Declarations, Let Declarations, Object Destructuring, Default Parameters, Await Declarations, Promise Declarations, Import Statements, Spread Arguments, Class Declarations, Rest Statements, Export Declarations, Arrow Function Declarations, and Async Declarations).

In addition to the clear trend of adopting new JavaScript features, Figure 1 showcases interesting situations where the overall usage count of specific language features diminishes. Upon careful analysis, we identified two primary factors contributing to this phenomenon. Firstly, certain commits purposefully revert contributions aimed at modernizing the codebase. Secondly, the merging of branches in commits often leads to this disruptive pattern of feature adoption, characterized by contributions that decrease the feature adoption count, followed by subsequent contributions that restore it to the original count.

RQ3: Is there any trend in the adoption of modern JavaScript features in JavaScript applications?

Our analysis indicates a discernible trend in the adoption of modern JavaScript features in JavaScript applications. The regression analysis conducted on our dataset reveals statistically significant upward trends for features such as Const Declarations, Let Declarations, Object Destructuring, Default Parameters, Await Declarations, Promise Declarations, Import Statements, Spread Arguments, Class Declarations, Rest Statements, Export Declarations, Arrow Function Declarations, and Async Declarations. These findings, as illustrated in Figure 1, suggest a consistent and increasing adoption of these features over time.

V. Discussions

In this section, we discuss the implications of our findings (Section V-A) and the limitations that may threaten the validity of our research (Section V-B).

A. *Implications of the Findings*

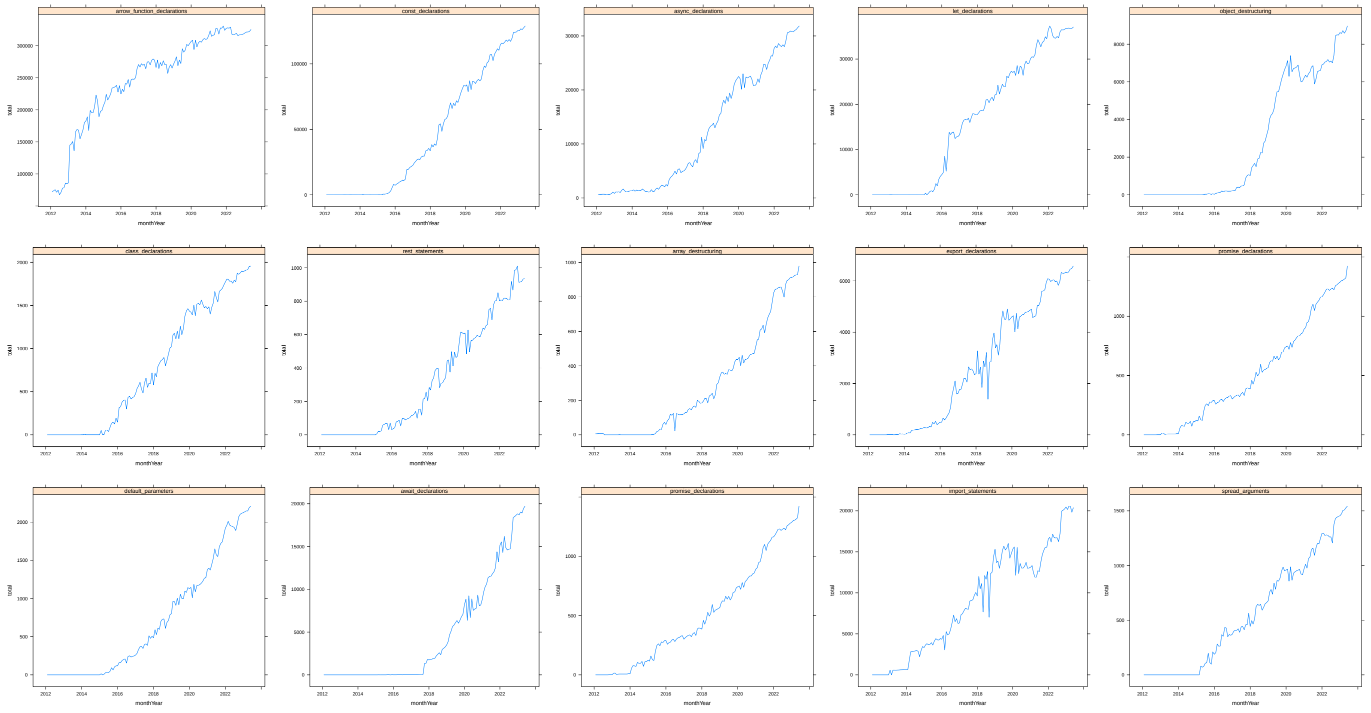
The findings of this study have some implications for the field of JavaScript development. Firstly, the extensive usage of features such as Async Declarations, Const Declarations, Arrow Function Declarations, Let Declarations, and Object Destructuring highlights their significance and popularity among JavaScript developers. These features have become widely adopted, indicating their importance in modern JavaScript development practices. Secondly, the early adoption of features like Arrow Function Declarations and Async Declarations suggests that JavaScript developers have been quick to embrace and utilize language enhancements even before their official releases (ES6, and ES8). This demonstrates the JavaScript developers' eagerness to leverage new features and leverage their benefits in JavaScript projects. Additionally, the prevalence of other modern JavaScript features, including Default Parameters, Await Declarations, Promise Declarations, Import Statements, Spread Arguments, Class Declarations, Rest Statements, Export Declarations, and Promise All() and Then(), emphasizes their widespread adoption within a relatively short time frame after their introduction. These implications underline the continuous evolution and rapid acceptance of modern JavaScript features in the development open-source community.

B. *Threats to Validity*

In terms of internal validity, our study focused on examining a specific subset of JavaScript features to ensure unbiased results. We selected this subset based on the features presented in Part 4 (Modernizing JavaScript) [17].

For external validity, our analysis involved 100 out of 609 JavaScript projects (16.42%) sourced from the GitHub community. These selected projects met specific criteria, including a development history of more than ten years and recent

Fig. 1. Distribution of all features usage across the JavaScript projects in our dataset.



contributions (after January 01, 2023). It is important to acknowledge that our dataset represents only a small fraction of the total number of applications, limiting the generalizability of our findings. Additionally, since our study focused solely on open-source repositories, we cannot generalize our results to industry projects. Nevertheless, we firmly believe that our study's findings hold relevance for developers in general, offering valuable insights into their JavaScript development practices.

VI. Related Works

The related works for our study reflect researchs focused on programming language features adoption for different languages, as TypeScript and Kotlin, and other research focused on selected in-depth JavaScript specific features.

Mateus and Martinez [12] conducted a similar study about feature adoption by developers for the Kotlin program language used in Android development. The research gathered empirical data based on 387 Android applications' source code repositories and identified when features were used for the first time, just like our method in this study. Their findings analyzing 26 features show that 15 of these are used on at least 50% applications and that 24 features are incorporated gradually into the source code over time. While Kotlin differs from JavaScript, this research aligns with our goals of analyzing language feature adoption and evolution.

Additionally, the research proposed by Scarsbrook et al. [16] offers a comprehensive examination of the adoption of TypeScript by analyzing 454 open-source repositories on GitHub. The authors studied the adoption of 13 language features over a three-year span (2020-2022) and the findings show that some features are largely adopted by the community while others rarely are used. Roughly 35% of projects adopt the latest release within the first three months after release. Similar to our work, the authors released an analysis and data-gathering tool to help the programming language community. Although focusing on TypeScript, this work presents valuable insights into the dynamics of language feature adoption, thereby informing analogous studies pertaining to JavaScript features.

Silva et al. [10] study focuses on the usage of classes in JavaScript. The authors extracted data from 50 popular JavaScript projects available on GitHub, ranked with at least 1,000 stars and that have at least 150 commits to the study research date(June 2014). Their findings show that 74% of the selected projects have classes implemented on the source code and about 40% make a relevant usage of classes, and that there is no significant relation between project size and class usage. Although Silva et al. [10] work focuses specifically on classes, the study contributes to the understanding of adoption rates of language constructs as JavaScript features.

Alves et al. [5] research quantifies the adoption of functional programming concepts in JavaScript by developers. The authors mined 91 GitHub repositories to identify the usage of language features such as immutability, lazy evaluation, higher-order functions, whose has been growing throughout time, and recursion and callbacks & promises, whose adoption decreased during the same researched period. Despite being a study that somewhat restricted the number of language features analyzed, it is related to our research on the adoption of these features by developers over a period of time.

A study published by Gokhale et al. [9] presents a tool and a refactoring technique designed to facilitate the development transition from synchronous to asynchronous APIs in JavaScript projects. Through the use of static analysis on the source codes the authors identified GitHub projects which had calls to synchronous API functions implemented, and they selected 12 random projects. The refactoring technique and tool proposed by the authors migrated the selected projects' API calls to use the asynchronous JavaScript language features named Promises and Async/Await. This study is related to our on the mining and use of static analysis on the source codes to find determined JavaScript language features within the projects.

Some of these related works refer more closely to our research on identifying the adoption by developers of the features released when a programming language has a new version. Other works describe mining and static analysis techniques similar to ours in search for specific JavaScript language features on projects' source code. Our study endeavours to deepen the understanding of JavaScript feature adoption over time and by its released versions, aiming to provide additional contributions to the programming community besides these related works provided.

VII. CONCLUSIONS

This study provides insights into the adoption patterns, timeline, and trends of modern JavaScript features in JavaScript applications of open-source communities. The analysis demonstrates the extensive utilization of modern JavaScript features, indicating their significance in contemporary JavaScript development. The early adoption of certain features showcases the forward-thinking approach of JavaScript developers and their readiness to incorporate cutting-edge language enhancements into their projects rejuvenating their codebases. Furthermore, the identified trends in the adoption of modern JavaScript features affirm their increasing popularity and the growing adoption of them. The findings of this research contribute to a deeper understanding of the evolving landscape of JavaScript development and highlight the importance of staying abreast of new features to leverage their potential effectively. As JavaScript continues to evolve, future studies can build upon these findings to explore the impact of these features on software quality, developer productivity, and overall application performance. In future work, we intend to find developers' rejuvenation efforts to introduce the modern features of the JavaScript language and build a catalog of transformations to help other developers, communities, and organizations maintain their current programs.

REFERENCES

1. ANTLR. https://www.antlr.org/. Accessed on February 13th, 2023.
2. ECMA international. https://262.ecma-international.org. Accessed on May 25th, 2023.
3. Maven. https://maven.apache.org/. Accessed on February 13th, 2023.
4. OpenJDK. https://openjdk.org/. Accessed on February 13th, 2023.
5. F. Alves, D. Oliveira, F. Madeiral, and F. Castor. On the bug-proneness of structures inspired by functional programming in javascript projects, 06 2022.
6. O. Dabic, E. Aghajani, and G. Bavota. Sampling projects in github for MSR studies. In 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, pages 560–564. IEEE, 2021.
7. E. Ecma. 262: Ecmascript language specification. Technical report, ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr,, 1997.
8. D. Flanagan. JavaScript: The Definitive Guide : Master the World's Most-used Programming Language. O'Reilly Media, Incorporated, 2020.
9. S. Gokhale, A. Turcotte, and F. Tip. Automatic migration from synchronous to asynchronous javascript apis. Proc. ACM Program. Lang., 5(OOPSLA), oct 2021.
10. L. Humberto Silva, M. Ramos, M. T. Valente, A. Bergel, and N. Anquetil. Does JavaScript software embrace classes? In SANER 2015 : International Conference on Software Analysis, Evolution, and Reengi pages 73 – 82, Montreal, Canada, Mar. 2015.
11. J. Keith. DOM Scripting. Apress, 2005.
12. B. G. Mateus and M. Martinez. On the adoption, usage and evolution of kotlin features in android development. In Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery.
13. J. Morgan and A. Stewart. Simplifying JavaScript: Writing Modern JavaScript with ES5, ES6, and Beyond. The Pragmatic Programmers. Pragmatic Bookshelf, 2018.
14. C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: How new features are introduced, championed, or ignored. In Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, page 3–12, New York, NY, USA, 2011. Association for Computing Machinery.
15. J. Resig, B. Bibeault, and J. Maras. Secrets of the JavaScript Ninja, Second Edition. ITpro collection. Manning Publications, 2016.
16. J. D. Scarsbrook, M. Utting, and R. K. L. Ko. Typescript's evolution: An analysis of feature adoption over time, 2023.
17. A. Wirfs-Brock and B. Eich. Javascript: The first 20 years. Proc. ACM Program. Lang., 4(HOPL), jun 2020.
18. X. Yan and X. Su. Linear Regression Analysis: Theory and Computing. World Scientific, 2009.