

# CMSC 124 Final Project

Tumulak, Patricia Lexa U.,      Valles, Oscar Vian L.

December 31, 2020

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Javascript</b>                                     | <b>1</b>  |
| 1.1      | Purpose and Motivations . . . . .                     | 1         |
| 1.2      | History . . . . .                                     | 1         |
| 1.3      | Language Features . . . . .                           | 4         |
| 1.4      | Paradigms . . . . .                                   | 5         |
| 1.5      | Language Evaluation Criteria . . . . .                | 7         |
| <b>2</b> | <b>Rust</b>   | <b>9</b>  |
| 2.1      | Purpose . . . . .                                     | 9         |
| 2.2      | History . . . . .                                     | 9         |
| 2.3      | Language Features . . . . .                           | 11        |
| 2.4      | Paradigms . . . . .                                   | 14        |
| 2.5      | Language Evaluation Criteria . . . . .                | 16        |
| <b>3</b> | <b>Feature 1: Event Listeners</b>                     | <b>18</b> |
| 3.1      | Description . . . . .                                 | 18        |
| 3.2      | Advantages in having the feature . . . . .            | 18        |
| 3.3      | Advantages in not having the feature . . . . .        | 18        |
| 3.4      | How to implement similar functionality in B . . . . . | 18        |
| <b>4</b> | <b>Feature 2: Objects</b>                             | <b>20</b> |
| 4.1      | Description . . . . .                                 | 20        |
| 4.2      | Advantages in having the feature . . . . .            | 20        |
| 4.3      | Advantages in not having the feature . . . . .        | 20        |
| 4.4      | How to implement similar functionality in B . . . . . | 20        |
|          | <b>References</b>                                     | <b>21</b> |

# 1 Javascript

## 1.1 Purpose and Motivations

JavaScript or JS was first made to provide a lightweight programming language for NetScape that would make web development more accessible instead of requiring deeper training. Today, it is now one of the most widely used programming languages and is mainly used to build websites and web-based applications. This is because it allows the creation of interactive elements for web pages that enhances the user experience. While HTML and CSS give web pages structure, JavaScript gives it responsiveness that engages the user. It is also not limited to just web technology but is also used in game development and mobile applications.

## 1.2 History

### 1.2.1 Mocha

JavaScript was developed by Brendan Eich in September 1995 when he was tasked to develop a “Scheme for the web browser” — a simple, dynamic, lightweight, and powerful scripting language with syntax that resembled Java for NetScape. It would be accessible to non-developers such as designers. The first version of JavaScript was made in only just 10 days. JavaScript was originally named Mocha, then called LiveScript, and finally renamed to JavaScript in December 1995 to make it sound closer to Java and was presented as a scripting language for client-side tasks in the browser.

### 1.2.2 ES1 and 2

With Microsoft’s development of their own web browser, Internet Explorer, they developed their own language similar to Javascript called JScript. With the rapid growth of the internet, the need to standardize JavaScript was realized. NetScape tapped the European Computer Manufacturers Association (ECMA) to make a standardized language. In June 1997, the first version of the ECMAScript, labelled ECMA-262, was released. Due to trademark reasons, ECMA could not use JavaScript for the name of the standardized language and so, JavaScript is its commercial name.

ECMAScript 2 or ES2 was released in June 1998 with relatively no new features to the language and only fixed a few inconsistencies between the ECMA

and ISO standard for JavaScript.

### **1.2.3 ES3**

ECMAScript 3 (or ES3) was released in December 1999 with changes to features were made such as regular expressions, exceptions and try/catch blocks, do-while block, the operators in and instanceof, and more.

It was also during this time that AJAX (asynchronous JavaScript and XML) was born which was a technique that allowed pages to be updated asynchronously using JavaScript and browser built-in XMLHttpRequest object. The term AJAX was coined by Jesse James Garrett.

### **1.2.4 ES3.1 and ES4**

As soon as ES3 was released in 1999, work on ES4 had already begun. The goal for this version of ECMAScript was to design features that allowed JavaScript to be used on the enterprise scale. However, conflict within the committee that worked on it (with representatives from Adobe, Mozilla, Opera, Microsoft, and Yahoo) started to arise. Some parties within expressed concern that ES4 was beginning to get “too big and was out of control”. These were words by Douglas Crockford, an influential JavaScript developer from Yahoo. Microsoft also supported Doug’s concerns and eventually, the group split off to work on ES4 and another separate idea called ES3.1 which was a simpler proposal with no new syntax and only practical improvements. ES4 ended up being too complex and was finally scrapped in 2008. Eventually ES4 found its way into the market as ActionScript developed by Adobe which was the scripting language supported by Flash.

jQuery is a JavaScript library that was initially released in August 2006. Created by John Resig, it allows developers to add extra functionality to web-pages. According to W3Techs, 74.4% of the top 10 million websites use jQuery as of February 2020.

NodeJS, a server-side runtime for JavaScript, was introduced in May 2009 by Ryan Dahl. This was built on Chrome’s V8 engine and it included an event loop. This helped build real-time web applications that scale. NodeJS also enabled developers to build a web app stack using only one programming language. This paradigm is called JavaScript Everywhere.

### 1.2.5 ES5

ECMAScript 3.1 was completed and released in December 2009, exactly 10 years after ES3. ECMAScript 3.1 was renamed ECMAScript 5 by the committee to avoid confusion. It was supported by Firefox 4, Chrome 19, Safari 6, Opera 12.10, and Internet Explorer 10. ES5 featured updates to the language such as getter/setters, reserved words, new methods for Object, Array, and Date, JSON support, among others. This did not require any changes to syntax.

Another iteration of ES5 called ECMAScript 5.1 was released in 2011. However, this did not provide new features but only clarified ambiguous points.

### 1.2.6 ES6 (ES2015) and ES7 (ES2016)

2015 introduced a huge leap forward for JavaScript with the release of ES6 or ES2015 with the introduction of features such as promises, let and const bindings, generators, classes, arrow functions, spread syntax, among others.

It was also during 2015 that ReactJS, the framework that solidified modern day declarative UI patterns, was introduced. It took some concepts of AngularJS with declarative UI but improved them unidirectional data flow, immutability, and the use of the virtual DOM.

June 2016 saw the release of the 7th edition of ECMAScript — ES2016. This was a smaller release with few new features introduced such as the exponential operator (\*\*), keywords for asynchronous programming and the Array.prototype.includes function.

### 1.2.7 ES8, 9, 10 (ES2017, 2018, 2019)

For the next three years, more features were added in subsequent editions of ECMAScript. This included but not limited to features such as functions for easy Object manipulation, rest/spread operators for object literals, asynchronous iteration, and changes to Array.sort and Object.fromEntries.

### 1.2.8 ES11 (ES2020)

Published in June 2020, ECMAScript 2020 included new functions, the primitive type BigInt for integers that were arbitrarily sized, the nullish coalescing operator, and the globalThis object.

## 1.3 Language Features

### 1.3.1 Java-like syntax

JS shares syntax with Java such as the use of brackets, semicolons to end statements, return statements, if and do..while statements

### 1.3.2 Dynamic typing

JS supports dynamic typing, allowing a variable's type to be determined/defined based on the value stored.

```
let x = 42
x = "Forty-two"
x = new Array(42)
```

Figure 1: Dynamic Types in JavaScript

Figure 1 presents a case where different data types are stored in the same variable. This is allowed in JavaScript and no errors would be raised when this is run.

### 1.3.3 Prototypal Inheritance

Prototypal inheritance is a feature in JavaScript where each object has a special hidden property called `[[Prototype]]` that is a reference to another object or is null. This prototype serves as the basis of the current object. Whenever a property is read from an object and it is missing, the prototype provides the value of the object. Figure 2 illustrates this.

```
const animal = {
  alive: true
}

const bird = {
  fly: true,
  __proto__: animal
}

console.log(bird.alive) // Console: true
```

Figure 2: Dynamic Types in JavaScript

### 1.3.4 Interpreted Language

JS script is interpreted by the JavaScript interpreter — a built-in component of the web browser. In recent years however, just in time compilation is used for JS code such as in Chrome’s V8 engine.

### 1.3.5 Client-side validations

JS is a client-side scripting language. This means that JS functions can run even after the webpage has been loaded without communication with the server because the source code is processed by the client’s web browser instead of the web server. This makes JS very useful for things such as forms with the capability to validate errors in user input before sending the data to the server.

### 1.3.6 Arrow functions

Useful light-weight syntax that further simplified and shortened function syntax and lessened the number of lines of code.

### 1.3.7 Website Document Manipulation

The most common feature that JavaScript is used for is manipulating a website’s document structure. This is made possible by using the Document Object Model (DOM)

```
const link = document.querySelector('.link')  
  
link.textContent = 'Modified Text'  
link.href = 'https://modified.link.here'
```

Figure 3: DOM Manipulation

Figure ?? is a basic example of how a website’s content may be programmatically modified through the use of JavaScript.

## 1.4 Paradigms

JavaScript is a multi-paradigm language. It supports object-oriented programming, functional programming, and event-driven programming.

### 1.4.1 Object-Oriented Programming

JavaScript supports object-oriented programming through the use of object prototypes and prototypal inheritance. Objects in JavaScript are also mutable, meaning new instances of objects can be given new methods and properties.

```
// Defining a Class
class Person {
  constructor(first, last, age){
    this.first = first,
    this.last = last,
    this.age = age
  }

  get name() {
    return `${this.first} ${this.last}`
  }
}

// Creating an Object
const ana = new Person("Ana", "Canete", 21)

// Adding a new property to an object
ana.course = "BS Bio"
```

Figure 4: Objects in JavaScript

Figure 4 shows how a class is defined, an object is created and how a new property can be added in JavaScript

### 1.4.2 Functional Programming

The use of first-class functions, arrow functions (which are basically lambdas), and closures make the language functional as well.

First-class functions can be passed as an argument to another function, returned from another function, and stored in an array, variable, or object.

Lambdas (arrow functions in JavaScript) make code more succinct. Its body can be an expression or a statement block.

Closures are the collection of all variables within the scope of a function when it is created. It is the function and the lexical environment wherein the function was declared.

Figure 5 shows an arrow function being returned from another function. It



```

function foo(){
    const num = 42

    // An arrow function as the value being returned
    return () => console.log(num)
}

const bar = foo()

bar() // Console: 42

```

Figure 5: Arrow Functions in JavaScript

is stored in a variable and is then executed. The arrow function retains the scope and environment where it was declared, making it work and still print the variable `num` to the console.

### 1.4.3 Event-Driven Programming

Event-driven programming is a paradigm wherein a program is designed to respond to events such as user's actions. In JavaScript, this is usually done with an event loop and event listeners that listens or waits for these event triggers and then responds with an event handler which is a function that is called once an event is detected.

```

document.addEventListener('wheel', (event) => console.log(event))

```

Figure 6: Adding Event Listeners

Figure 6 illustrates how an event listener is added to the document. This event listener calls the arrow function every time a `'wheel'` event is detected in the document.

## 1.5 Language Evaluation Criteria

### 1.5.1 Simplicity

JavaScript, with its small number of primitive constructs and feature multiplicity, is highly writable. Due to this, its readability diminishes. However, JavaScript also does not support operator overloading, which also helps its readability.

### **1.5.2 Orthogonality**

JavaScript is capable of executing operations on variables of different data types due to its dynamically typed nature. This gives the language high orthogonality. This implies that JavaScript is fairly readable in terms of orthogonality but a high degree of orthogonality diminishes its writability and its reliability. This is because errors might be undetected.

### **1.5.3 Data Types**

JavaScript has a variety of different primitive data types including strings, numbers, booleans, null, undefined, symbol, and object. Data types in JS are also dynamically typed. Due to the data types not being explicitly defined, this makes it harder to read but also easier to write.

### **1.5.4 Syntax Design**

JavaScript's syntax design shares some similarities with Java such as the use of braces to start and end blocks of statements. JavaScript also uses a number of special and reserved words. This all helps enhance the language's readability and writability.

### **1.5.5 Support for Abstraction**

JavaScript supports abstraction since it allows the creation of classes, functions, and arrow functions. This makes it writable and reliable.

### **1.5.6 Expressivity**

Functions that would require multiple lines of code in other languages such as C could be written and simplified in JavaScript with only a few lines because of the language having operators that make code succinct, making the language have high expressivity. This means the language is very writable as well as reliable.

### **1.5.7 Type Checking**

JavaScript, being dynamically typed, makes it difficult to run type checks before program execution. There are some tools to conduct this but it adds another step to the process of building code. Type checking during execution also results

in the performance suffering and a tendency for mistakes. These makes type checking in JavaScript not very reliable.

### **1.5.8 Exception Handling**

JavaScript has measures for handling exceptions such as try-catch statements. This allows for intercepting run-time errors, making corrections, and continuing with execution. This makes the language reliable.

### **1.5.9 Restricted Aliasing**

Every data type aside from the primitive types (strings, numbers, booleans, null, undefined, and symbol) are considered objects in JavaScript and are referred to as reference types. Aliasing is possible for objects, arrays, and functions since they are passed by reference. Due to this, its reliability suffers.

## **2 Rust**

### **2.1 Purpose**

Rust was created by Graydon Hoare as a hobby in 2006. He described the language as “compiled, concurrent, safe, systems programming language”. This was initially developed to solve common problems in C and C++ that were related to memory management using built-in guards through the compiler and the design of the language. In addition, Graydon also added known and loved ideas from other languages into a systems language. He also wanted to revive old ideas from the 70s and 80s based on the theory that circumstances have changed and the design tradeoffs that used to favor C and C++ have shifted.

Today, Rust has expanded into different industries and use cases. Some of these include embedded devices, command line interface tools, web assembly, and networking.

### **2.2 History**

#### **2.2.1 The Personal Years (2006–2010)**

Graydon Hoare started working on a compiled, concurrent, safe, systems programming language as a hobby and as a research project. These were some of the following descriptions that Hoare wrote as Rust was beginning: Memory Safety,

Typestate system, Mutability control, Side-effect control, and Garbage Control. Some of these remained throughout the years, some were also scrapped. At this point, ~90% of the language features and ~70% of the runtime was roughly working.

### **2.2.2 The Graydon Years (2010–2012)**

During this time, Rust was adopted by Mozilla for use in their Servo project. This project was a rewrite of the Gecko rendering engine used for Firefox. Development of both Servo and Rust was parallel. Features were dogfooded by the Rust and Servo team and the language was iterated upon based on the feedback from both teams. During this time period Graydon became a benevolent dictator for life-like figure for Rust, much like how Linus Torvald is a benevolent dictator for life-like figure for Linux.

### **2.2.3 The Typesystem Years (2012–2014)**

During this time period, the team grew and incorporated more experts that had experience with advanced type systems. Due to this, the type system also grew as well. As the type system grew, functionality found in the language was transferred into libraries. In addition, the package manager of the language, Cargo, was also implemented during this period. At this time, Graydon stepped down from the project. This allowed the project to be able to democratically improve the project, since a singular authority to determine what is best for the language is not present.

### **2.2.4 The Release Years (2015–May 2016)**

Rust released 1.0.0-alpha back on January 9, 2015, 1.0.0-beta1 on February 16, 2015 and a 1.0.0 on May 15, 2015. The 1.0.0 release guarantees that Rust as a language will continue to change, but it will do so in a backwards-compatible manner. This meant that stability will be maintained in its various versions. Four aspects of the language were also improved during this period: ecosystem, tooling, stability, and community.

### **2.2.5 The Production Years (May 2016–Present)**

Nowadays, Rust has released version 1.48.0 and is widely used in various industries. Companies like Firefox, Dropbox, Cloudflare, and NPM all use Rust

in some form or another. Rust has also gained popularity in developer circles, being the most loved programming language for two years in a row, according to StackOverflow. The number of Crates found in Crates.io has also ballooned to 51,833, as of December 30, 2020.

## 2.3 Language Features

This section is heavily based on the book *The Rust Programming Language* by Klabnik and Nichols, 2018. Code examples are directly lifted from the book.

### 2.3.1 Immutability

In Rust, by default variables are immutable. In practice, once a value is bound to a name, the value cannot be changed. This prevents bugs where assumptions are made about values that will never change, but other parts of the code does change that value. The compiler guarantees that once a variable is declared to be immutable, it is immutable in all parts of the code.

```
fn main() {  
    let x = 5;  
    println!("The value of x is: {}", x);  
    x = 6;  
    println!("The value of x is: {}", x);  
}
```

Figure 7: Immutable Variable

Figure 7 will result in a compilation error because `x` is reassigned a value. However, mutability of a variable may still be possible by adding the keyword `mut` before assigning the variable.

```
fn main() {  
    let mut x = 5;  
    println!("The value of x is: {}", x);  
    x = 6;  
    println!("The value of x is: {}", x);  
}
```

Figure 8: Mutability Declaration

Figure 8 will now compile successfully and print the expected values.

Even if variables are immutable by default, constants are also available in Rust. One key difference between variables and constants is the ability to change its mutability. Variables can be mutable, while constants will always be immutable

### 2.3.2 Data Types

Rust is a statically typed language where the compiler needs to know the type of the variables at compile time. Despite that, it is not strongly typed. The compiler may infer the type based on what the value is and how it is used. Figure 7 shows this clearly where the type of the variable is not explicitly stated. However, there are cases where the compiler cannot accurately determine what the type is, hence a type annotation is required

```
fn main() {  
    let guess: u32 = "42".parse().expect("Not a number!");  
}
```

Figure 9: Annotations on a Variable

Figure 9 shows an example where the variable `guess` is given the annotation `u32` to show that the datatype of the variable is expected to be an unsigned 32bit integer. Otherwise, if not annotated, the compiler will spit out an error since the final type is not clear

### 2.3.3 Ownership

One of Rust's key defining feature is Ownership. It is a way for Rust to make memory safety guarantees without the need of a garbage collector. There are three main rules that Rust follows in handling ownership:

- Each value in Rust has a variable that's called its owner.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

These rules do not differ if the memory used is taken from the stack or from the heap. When the variable goes out of scope, the memory is always returned.

However, there is a difference when copying values from one variable to another. An example of copying values from a variable in the stack is shown in Figure 10

```
fn main() {
    let x = 5;
    let y = x;
}
```

Figure 10: Copying Variables from the Stack

This example is simple, since a new variable is created and the value is copied. After they go out of scope, both variables will be dropped. However, this is different when the variable is stored in a heap as shown in Figure 11

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
}
```

Figure 11: Copying Variables from the Heap

Variable `s1` is stored on the heap as the type `String` is dynamically allocated. When `s1` is copied from `s2`, `s1` is invalidated and dropped. This is because when `s1` is copied into `s2`, the contents of the heap is not copied, only the pointer to the starting character and other data pertaining to `s1` is copied. If `s1` is not dropped, a double free may occur once both variables go out of scope. Hence, the first variable, `s1` is dropped. This effectively means that `s1` is moved to `s2`, instead of being copied.

Rust also presents a way to deep copy the contents of a value stored on the heap through a method called `clone()`

### 2.3.4 Smart Pointers

The four most common smart pointers used in Rust are the following: `Box<T>`, `Rc<T>`, `RefCell<T>`

`Box<T>` stores a pointer on the stack that points to data found on the heap. It can be used during the following situations:

- When a type's size is not known on compile time and a value of that type needs an exact size
- When a large amount of data needs to be stored and ownership needs to be transferred, but a copy should not happen.

`Rc<T>` allows values to have multiple owners. This type keeps track of the references that point to a value, and is destroyed when the number of references reaches zero.

`RefCell<T>` is almost the same as `Box<T>`. The main difference from the two is the rules of ownership apply to `RefCell<T>` on runtime, rather than compile time. When the rules are broken on `Box<T>`, the compile errors out, when the rules are broken on `RefCell<T>`, the program panics and exits.

## 2.4 Paradigms

This section is heavily based on the book *The Rust Programming Language* by Klabnik and Nichols, 2018. Code examples are directly lifted from the book.

### 2.4.1 Concurrent

Rust allows safe concurrency through the ownership and type systems that Rust has. These features of Rust allow the error-checking of the compiler to find concurrency errors during compile-time rather than at runtime.

An simple example of a concurrent program is shown in Figure 12:

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Figure 12: Spawning Multiple Threads



### 2.4.2 Functional

Rust has the following features that allow functional programming: Closures and Iterators.

Closures are anonymous functions that can be saved in a variable or passed as arguments. They capture the scope from where they are defined. An example of a closure in Rust is shown in Figure 13

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

Figure 13: Closures in Rust

Iterators are responsible for iterating over each item. An example of an iterator in Rust is shown in Figure 14 where `v1_iter` is an iterator that iterate over the values in the vector `v1`

```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    for val in v1_iter {
        println!("Got: {}", val);
    }
}
```

Figure 14: Iterators in Rust

### 2.4.3 Generic

Rust allows generic programming through generics. Generics are stand-ins for other concrete types or properties that are not yet defined. A function with a Generic, shown in Figure 15, where the function `largest<T>(list: &[T])`, can take any list with arbitrary data types like `i32`, `u32`, `u8`, etc. Generics are like templates found in C++.

```
fn main() {
    let v1 = vec![1, 2, 3];

    let v1_iter = v1.iter();

    for val in v1_iter {
        println!("Got: {}", val);
    }
}
```

Figure 15: Generics in Rust

### 2.4.4 Imperative

Rust is imperative by nature, as it was influenced by C and C++. All figures shown before present code that provide evidence of the imperative nature of Rust.

## 2.5 Language Evaluation Criteria

### 2.5.1 Simplicity

Rust is not that simple, it has many basic constructs, with complicated concepts and meanings. Due to this Rust suffers from readability, writability, and reliability

### 2.5.2 Orthogonality

Rust is very orthogonal, every possible combination of primitives is possible. There is a very little amount of exceptions present in the language. This makes it very writable, readable, and reliable

### 2.5.3 Data Types

Rust's main feature is its robust data type system. Rust has adequate facilities for defining data types and structures through the usage of Structs and Generics. This gives great writability, readability, and reliability

### 2.5.4 Syntax Design

Rust's syntax design is based on C and C++, with additions from scripting languages and functional programming. It has explicit special words, but has truncated them. Most evidently with functions written as `fn`, mutable as `mut`, constant as `const`. Increasing writability but greatly reducing readability.

### 2.5.5 Support for Abstraction

Rust support for abstraction is great through functions and generics.

### 2.5.6 Expressivity

Rust is not that expressive, for the sake of achieving reliability. Most of the time, there is a singular 'Rust Way' of solving a problem. This way usually greatly reduces the errors that may occur and increases readability. One great example is the removal of the shorthand `count++` in favor of `count = count + 1` or `x += 1` as it is less ambiguous for readers. This makes it more reliable, but less writable.

### 2.5.7 Type Checking

Due to its robust data type system, type checking is also very strong in Rust. It is not strongly typed, as variable types may be implied by the compiler but type annotations help when the compiler cannot find a unique type for a variable. Type checking is done at compile-time and is very thorough with expressive error messages. These makes rust very reliable.

### 2.5.8 Exception Handling

Exception handling in Rust is exemplary. The compiler enforces code to always handle error cases. However, Rust does not have exceptions, but rather has two types of errors. `/rust::Result<T,E>` for recoverable errors and a `/rust::panic!` macro for unrecoverable errors. This gives the programmer better ways to handle errors in the program. This greatly increases Rust's reliability.

### 2.5.9 Restricted Aliasing

Aliasing is very restricted in Rust because of one key feature in the language: Ownership. Ownership prevents a lot of errors that may occur due to aliasing. This is one of the main reasons for Rust's reliability

## 3 Feature 1: Event Listeners

### 3.1 Description

Event listeners in JavaScript follow an Observer Pattern where events, also known as the subject in the Observer Pattern, notify event listeners or observers that are subscribed to the events. This is used in JavaScript to react to events that are asynchronous in nature. Examples include a `'KeyDown'` event, a `'wheel'` event, or a `'load'` event.

### 3.2 Advantages in having the feature

This allows Event-Driven programming in the language, and allows an easy way to handle asynchronous events

### 3.3 Advantages in not having the feature

It reduces the simplicity of the programming language since it introduces another paradigm to the language

### 3.4 How to implement similar functionality in B

The snippet shown on Figure 16 by Shemsedinov, 2020 shows how an EventEmitter is implemented in Rust:

```

use std::cmp::Eq;
use std::collections::HashMap;
use std::hash::Hash;

type HandlerPtr<T> = Box<Fn(&T)>;

pub struct EventEmitter<T: Hash + Eq, U> {
    handlers: HashMap<T, Vec<HandlerPtr<U>>>,
}

impl<T: Hash + Eq, U> EventEmitter<T, U> {
    pub fn new() -> Self {
        Self {
            handlers: HashMap::new(),
        }
    }

    pub fn on<F>(&mut self, event: T, handler: F)
    where
        F: Fn(&U) + 'static,
    {
        let event_handlers = self.handlers.entry(event).or_insert_with(|| vec![]);
        event_handlers.push(Box::new(handler));
    }

    pub fn emit(&self, event: T, payload: U) {
        if let Some(handlers) = self.handlers.get(&event) {
            for handler in handlers {
                handler(&payload);
            }
        }
    }
}

impl<T: Hash + Eq, U> Default for EventEmitter<T, U> {
    fn default() -> Self {
        Self::new()
    }
}

```

Figure 16: Event Emitter in Rust

## 4 Feature 2: Objects

### 4.1 Description

Objects couple data and methods together and hides implementation details that manipulate the data inside an object. JavaScript makes use of objects extensively, making most of the types in the language, like Arrays, Dictionaries, and Strings objects.

### 4.2 Advantages in having the feature

This feature allows the Object-Oriented Programming in the language. This carries the benefits that OOP brings, like reusability, encapsulation, inheritance, among others.

### 4.3 Advantages in not having the feature

It reduces the simplicity of the programming language since it introduces another paradigm to the language

### 4.4 How to implement similar functionality in B

A snippet shown on Figure 17 by Klabnik and Nichols, 2018 shows how methods and data could be coupled together. However, this lacks some OOP functionality like inheritance.

```
struct Rectangle {  
    width: u32,  
    height: u32,  
}  
  
impl Rectangle {  
    fn area(&self) -> u32 {  
        self.width * self.height  
    }  
}
```

Figure 17: OOP-Like Functionality in Rust

## References

- Avram, A. (2012). Interview on rust, a systems programming language developed by mozilla. <https://www.infoq.com/news/2012/08/Interview-Rust/>
- Brown, K. (2020). Javascript: How did it get so popular? <https://news.codecademy.com/javascript-history-popularity/>
- DeGroat, T. (2019). The history of javascript: Everything you need to know. <https://www.springboard.com/blog/history-of-javascript/>
- ECMA International. (2011). *Standard ecma-262 - ecmascript language specification* (5.1). <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- Features of javascript - 13 vital javascript features you must learn! (2019). <https://data-flair.training/blogs/features-of-javascript/>
- Fireship. (2019). The weird history of javascript. <https://www.youtube.com/watch?v=Sh6lK57Cuk4>
- Javascript features. (n.d.). <https://www.studytonight.com/javascript/javascript-features>
- Klabnik, S. (2016). The history of rust. *Applicative 2016 on - Applicative 2016*. <https://doi.org/10.1145/2959689.2960081>
- Klabnik, S., & Nichols, C. (2018). *The rust programming language*. no starch Press.
- Manipulating documents. (n.d.). [https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side\\_web\\_APIs/Manipulating\\_documents](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents)
- Peyrott, S. (2017). A brief history of javascript. <https://auth0.com/blog/a-brief-history-of-javascript/>
- Rust for js developers. (n.d.). <https://www.codegram.com/blog/rust-for-js-developers/>
- Shemsedinov, T. (2020). EventEmitter. <https://github.com/HowProgrammingWorks/EventEmitter>
- Usage statistics of javascript libraries for websites. (n.d.). [https://w3techs.com/technologies/overview/javascript\\_library](https://w3techs.com/technologies/overview/javascript_library)
- Uses of javascript: How and when javascript application is suited. (2020). <https://www.educba.com/uses-of-javascript/>