Java Script

Finally, the name actually means what it should.

Gianna Julio
CMPT331 - Theory of Programming Languages
Marist University
May 9, 2025

Contents

1 Introduction

Java Script (pronounced "Java **pause** Script") is a modern, hybrid programming language designed for seamless full-stack development. It combines the strict, expressive type system and functional capabilities of Scala with the gradual typing and ecosystem flexibility of TypeScript. Java Script compiles to both JVM bytecode and JavaScript, enabling shared logic across the frontend and backend. It differs from TypeScript and Scala in the following ways:

1. **Full-stack code sharing:** Seamlessly compiles to both JVM bytecode and JavaScript, enabling shared logic across frontend and backend.

2. **Gradual typing for migration:** Supports both dynamic scripting (like JavaScript) and strict type safety (like Scala).

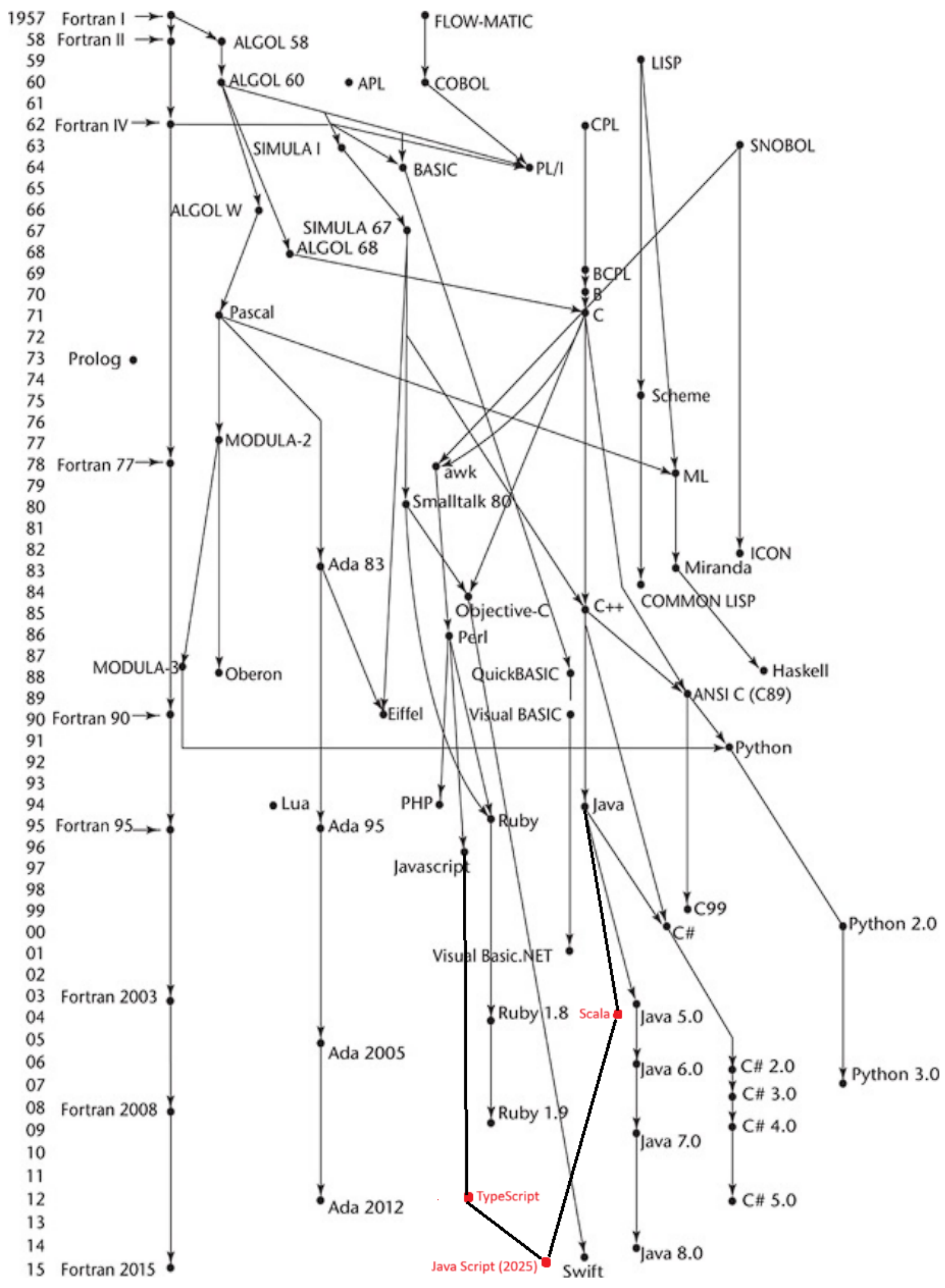
3. **Dual ecosystem access:** Direct interoperability with Java/Scala libraries on the JVM and npm/JS modules in the browser, unifying two ecosystems under one syntax.

4. **The name actually makes sense.** Java-inspired Scala and TypeScript (aka improved JavaScript) come together to make a language that is truly a combination of its namesakes.

1.1 Genealogy

Java Script comes from exactly where it sounds like it should. It is based on more modern versions of Java and JavaScript (Scala and TypeScript, respectively). Java Script inherits traits from:

- **Scala:** strong typing, functional programming/object oriented blend, JVM interoperability
- **TypeScript:** gradual typing, JS compatibility, browser deployment



1.2 "Hello, World!"

```
console.log("Hello, TypeScript!") // Top-level statement (JS-style)

@main def run() = {
  println("Hello from JVM!") // JVM entry point (Scala-style)
}

print("Hello, World!") // unified print to output on both platforms
```

1.3 Program structure

- **Modules:** file-based or explicit
- **Objects/Classes:** mixing Scala's object and TypeScript's class
- **Top-Level Functions:** like JavaScript/TypeScript

1.3.1 Example Program

This example demonstrates Java Script's hybrid structure through a minimal user service.

```
// Shared module (works on JVM & JS)
module UserService {

  // TypeScript-style type + Scala case class
  case class User(name: string, id: int)

  // Singleton object (Scala-style)
  object Database {
    private val users = List(
      User("Alice", 1),
      User("Bob", 2)
    )

    // Hybrid method: TypeScript arrow + Scala collection
    def findUser(id: int): User | null =
      users.find(u => u.id == id).orNull
  }

  // Async function (works as Promise/Future)
  def getUser(id: int): Promise[User] =
    Promise.resolve(Database.findUser(id))
}

// Frontend usage (JS)
const user = await UserService.getUser(1)
console.log(`Hello ${user?.name}`)

// Backend usage (JVM)
@main def run(): Unit = {
  val user = UserService.getUser(2).await
  println(s"Found: ${user.name}")
}
```

The program begins by defining data with a `User` class that blends Scala's case class syntax with TypeScript-style type annotations. The main logic resides in a Database object, using Scala's singleton object while incorporating JavaScript-like arrow functions for operations. A key feature is the `getUser` function which returns a Promise/Future - this single implementation works identically in both frontend and backend environments. The example shows how Java Script maintains Scala's organizational structure while adopting TypeScript's type flexibility and JavaScript's concise function syntax. As a result, the same method call adapts to each platform, using await in browser JavaScript and matching patterns in JVM Scala.

1.4 Types and Variables

- **Gradual Typing:** Types optional (inferred like TS/Scala).
- **Primitives:** `string`, `number`, `boolean`, `bigint` (TS-like names).
- **Collections:** `Array[T]` (JS-style), `List[T]` (Scala-style).

```
        let x = 10  // Inferred as 'number'
        var y: string = "foo"  // Explicit type
        const list = List(1, 2, 3)  // Scala-style list
```

Java Script features two fundamental type categories: primitive types and object types. Primitive types (like `number`, `boolean`, and `string`) store values directly, behaving exactly like JavaScript primitives when compiled to JS and like JVM primitives when compiled for the Java Virtual Machine. Object types (including classes, arrays, and collections) are reference types that work consistently across both platforms - when compiled to JavaScript they behave like JS objects with reference semantics, while on the JVM they map to corresponding Java/Scala object types. A key characteristic of Java Script's object types is their shared behavior across platforms: assigning an object to a new variable creates a reference rather than a copy, meaning mutations through one reference are visible through all others, whether running on Node.js, in browsers, or on the JVM. This unified handling of references ensures consistent behavior for objects regardless of the target platform, while primitive values maintain the expected platform-specific optimizations. See Section 3 for details.

1.5 Visibility

- `public` (default)
- `private` (Scala-style scoping)
- `protected`, `internal` (JVM-only)

Java Script provides four visibility modifiers to control access to types and members: `public`, `private`, `protected`, and `internal`. By default, all declarations are public, matching TypeScript's approach for JavaScript interoperability. Private members are accessible only within their containing class or object, enforced at compile time in both JVM and JS targets. The protected modifier follows Scala/JVM semantics, restricting access to subclasses while also being accessible from the same package. For cross-platform compatibility, Java Script introduces internal visibility, visible everywhere within the same module (compilation unit) on both platforms, but unavailable outside it. This ensures consistent access control whether the code runs on JavaScript or the JVM.

1.6 Statements Differing from Parent Languages

1.6.1 if

```
let x = 10
if (x > 5) {
  print("High")
} else {
  print("Low")
}
```

Uses TypeScript-style parentheses () and unified print for both platforms. Additionally, no parentheses are required around conditions in Scala, but are enforced for cross-platform consistency.

1.6.2 for

```
// compiles to JS for-of and Scala ranges
for (i <- 1 to 5) {
  print(i)
}

// JS-style array iteration
for (item of [1,2,3]) {
  print(item)
}
```

Uses `<-` syntax from Scala for ranges, `of` from TypeScript for arrays, and unified print.

1.6.3 match

```
let result = match (someValue) {
  case x: number => `Number ${x}`
  case "hello" => "Greeting"
  case _ => "Default"
}
```

Uses TypeScript type guard, Scala literal match, and a wildcard default case.

2 Lexical Structure

2.1 Programs

A Java Script program consists of one or more source files (`.jsc` extension) containing a sequence of Unicode characters. Conceptually, compilation follows these stages:

1. **Lexical Analysis:** Converts source text into tokens, supporting both TypeScript-style and Scala-style interpolated strings, as well as single-line (`//`), and multi-line (`/* */`), and Doc (`/** **/`) comments.
2. **Syntactic Analysis:** Parses tokens into an Abstract Syntax Tree (AST) using hybrid rules.
3. **Platform-Specific Code Generation:**
   - **JavaScript Target:** Modules with TypeScript-compatible types.
   - **JVM Target:** Generates JVM bytecode with Scala-like optimizations.

2.2 Grammars

This specification presents the syntax of the Alan++ programming language where it differs from Scala and TypeScript.

2.2.1 Lexical grammar where different from Parent Languages

Java Script combines the keywords for Scala-style methods, pattern matching, and singletons with TypeScript-style bindings and aliases. A unique platform check keyword is added, as well as the internal visibility modifier.

```
<keyword> ::= def | let | match | type | object | isJVM
<modifier> ::= private | protected | public | internal
```

2.2.2 Syntactic grammar where different from Parent Languages

Scala-style def functions and TypeScript arrow functions (=¿) are both valid, letting developers choose their preferred style. Similarly, for loops support both Scala's range syntax (x ¡- 1 to 5) and TypeScript-style iteration (for...of). The match expression adopts Scala's pattern matching but requires explicit =¿ for cases, avoiding ambiguity. These choices preserve familiarity while ensuring consistent behavior across platforms. The unified `print` as well as platform-specific print functions are also specified below.

```
<function> ::=
    | def <identifier> ( [<params>] ) [: <type>] = <expr>
    | ( <params> ) => <expr>

<for> ::=
    | for ( <identifier> <- <expr> to <expr> ) <block>
    | for ( <identifier> of <expr> ) <block>

<match> ::= match ( <expr> ) { case <pattern> => <expr> }

<print> ::= print ( <expr> ) | println ( <expr> )
```

```

    <js-print> ::= console.log ( <expr> )
    <jvm-print> ::=
        | System.out.print ( <expr> )
        | System.out.println ( <expr> )
```

2.3 Lexical Analysis

2.3.1 Comments

Java Script supports three types of comments for flexible documentation. Single-line comments begin with '//' and continue to the end of the line, identical to both TypeScript and Scala. Delimited comments use '/* ... */' and can span multiple lines, following conventional C-style syntax. Additionally, Java Script adopts Scala's nested comment capability, allowing '/* outer /* inner */ still-outer */' constructs for temporarily disabling commented blocks. Unlike TypeScript (which forbids nesting) but like Scala, these nested comments are fully supported. All comment forms are ignored by the compiler and have no runtime representation, whether targeting JVM or JavaScript platforms. Documentation comments using '/** ... **/' follow Scala's and TypeScript's doc conventions.

2.4 Tokens

Java Script recognizes the following categories of tokens, which form the basic building blocks of source code:

- **Identifiers** – Names for variables, functions, and types (e.g., `x`, `calculateTotal`)
- **Keywords** – Reserved words with special meaning:
  - Declaration: `def`, `let`, `type`, `object`
  - Control flow: `match`, `if`, `else`
  - Platform: `isJVM`
- **Literals**:
  - *Integer*: `42`, `0xFF`, `0b1010`
  - *Floating-point*: `3.14`, `6.022e23`
  - *Boolean*: `true`, `false`
  - *String*: `"hello"`
  - *Character*: `'A'` (JVM-only, compiled to JS as single-char strings)
  - *Null/Undefined*: `null`, `undefined` (aliases in JVM)
- **Operators**:
  - Arithmetic: `+`, `-`, `*`, `/`, `%`
  - Comparison: `==`, `!=`, `===`, `!==` (JS-style strict equality)
  - Logical: `&&`, `||`, `!`
  - Bitwise: `&`, `|`, `^`, `~`

  - Assignment: `=`, `+=`, `-=`, etc.

- **Punctuators** – Syntax delimiters:
  - Grouping: `(`, `)` (grouping)
  - Blocks: {, } (blocks)
  - Arrays: `[`, `]` (arrays)
  - Miscellaneous: `:`, `,`, `;`, `.`, `=>` (type annotations, separators, lambda arrows)

Note: whitespace and comments are not tokens but separate adjacent tokens

2.4.1 Keywords different from Parent Languages

New Keywords

- `isJVM` - Platform detection (compiles to `true/false`)
- `internal` - Module-level visibility

Removed Keywords

- From Scala:
  - `implicit`
  - `trait` - Simplified inheritance model
  - `with` - Conflicting semantics
- From TypeScript:
  - `enum` (redefined) - Modified implementation
  - `namespace` - Replaced by `module`
  - `declare` - Simplified type system
  - `public` - Now default visibility

Modified Keywords

- `match` - Requires `=>` (unlike Scala's optional)
- `type` - Supports both:
  - TypeScript unions (`|`)
  - Scala type aliases
- `object` - Works with both:
  - Scala singleton objects
  - JavaScript object literals

Note: minimal changes should maintain backward compatibility with both ecosystems

3 Type System

Java Script uses a hybrid gradual type system, blending static and dynamic typing to be compatible with both Scala and TypeScript. During compilation, it enforces strong typing where explicit declarations exist (like Scala), catching type errors early, while allowing untyped variables (as in TypeScript) for flexibility. Static type checking is applied to all explicit declarations.

3.1 Type Rules

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 == e_2 : \text{boolean}} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 > e_2 : \text{boolean}}$$

$$\frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 \mathrel{!=} e_2 : \text{boolean}} \qquad \frac{\begin{array}{l} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 < e_2 : \text{boolean}} \qquad \frac{\begin{array}{l} S \vdash e_1 : \text{any} \\ S \vdash e_2 : \text{any} \end{array}}{S \vdash e_1 = e_2 : \text{any}}$$

```
let x: any = "5"
let y: any = 5
print(x == y)  // Typechecks as ‘: boolean‘ with warning
```

Java Script organizes its type system into three categories to work across both the JVM and JavaScript platforms. Primitive types like numbers, booleans, and characters are value types that get copied when passed, ensuring predictable behavior matching each platform's native handling. Object types, including classes and arrays, use reference semantics, allowing shared mutable states like in both parent languages. Java Script also introduces a special hybrid type `any` to resolve differences between the two platforms.

3.2 Value Types (differing from Parent Languages)

In Java Script, value types (primitives) behave nearly identically to their native platform implementations. On the JVM, numbers and booleans map directly to Java primitives int, double, and boolean. When compiled to JavaScript, these same types use JS primitives number and boolean. The only semantic difference is for char values: on the JVM they become primitive char types, while in JavaScript they compile to single-character strings.

3.3 Reference Types (differing from Parent Languages)

In Java Script, reference types (objects, arrays, classes) maintain consistent behavior across platforms with some adaptations. On the JVM, they compile to regular Scala objects with standard reference semantics, while in JavaScript, they become native JS objects that are immutable (like Scala) unless explicitly declared. Data structures like Array[T] map directly to Java ArrayList and JS Array. The biggest difference is null handling: Java Script unifies JS undefined/null into a single null for all reference types.

4 Example Programs

4.1 Caesar Cipher (Encrypt)

```
module Cipher

// Unified function works on JVM/JS
def encrypt(text: string, shift: int): string =
  text.map(c =>
    if (c.isLetter) {
      val base = if (c.isUpper) 'A'.code else 'a'.code
      val shifted = (c.code - base + shift) % 26
      (base + (if (shifted < 0) shifted + 26 else shifted).toChar
    } else c
  ).mkString("")

// Example usage (cross-platform)
@main def run(): Unit = {
  val message = "Hello, World!"
  val encrypted = encrypt(message, 3)  // "Khoor, Zruog!"

  if (isJVM) println(encrypted)
  else console.log(encrypted)
}
```

4.2 Caesar Cipher (Decrypt)

```
module Cipher

def decrypt(text: string, shift: int): string =
  text.map(c => {
    if (c.isLetter) {
      val base = if (c.isUpper) 'A'.code else 'a'.code
      val shifted = (c.code - base - shift) % 26
      (base + (if (shifted < 0) shifted + 26 else shifted)).toChar
    } else c
  }).mkString("")

// Example Usage
@main def demo(): Unit = {
  const secret = "Khoor, Zruog!"
  console.log(decrypt(secret, 3))  // Prints "Hello, World!"
}
```

4.3 Factorial

```
module MathUtils

// Gradual typing: TypeScript-style annotation with Scala 'def'
def factorial(n: int): bigint =
  if (n <= 1) 1n
  else n * factorial(n - 1)

// Platform-adaptive output
@main def run(): Unit = {
  val num = 5
  val result = factorial(num)

  if (isJVM) {
    println(s"Factorial of $num is $result")  // Scala string interpolation
  } else {
    console.log(`Factorial of ${num} is ${result}`)  // TS template literal
  }
}
```

4.4 Merge Sort

```
module Sorting

// TypeScript-style generics with Scala's 'def'
def mergeSort[T](arr: Array[T])(using ord: Ordering[T]): Array[T] = {
  if (arr.length <= 1) arr
  else {
    val mid = arr.length / 2
    val (left, right) = arr.splitAt(mid)

    // Recursive parallel sorting
    val sortedLeft = mergeSort(left)
    val sortedRight = mergeSort(right)

    // Hybrid merge with pattern matching
    merge(sortedLeft, sortedRight)
  }
}

// Private helper (Scala-style with TS type annotations)
private def merge[T](left: Array[T], right: Array[T])(using ord: Ordering[T]):
    Array[T] = {
  var result = Array.empty[T]  // Mutable for performance
  var (i, j) = (0, 0)

  while (i < left.length && j < right.length) {
    result = if (ord.lt(left(i), right(j)))
      result :+ left(i += 1)
    else
      result :+ right(j += 1)
  }

  // Concatenate remaining elements (JS spread + Scala ++)
  result ++ left.slice(i) ++ right.slice(j)
}

// Usage
@main def demo(): Unit = {
  val numbers = Array(3, 1, 4, 1, 5, 9, 2, 6)
  val sorted = mergeSort(numbers)

  if (isJVM) println(sorted.mkString(", "))
  else console.log(sorted.join(", "))
}
```

4.5 Quicksort

```
module Sorting

// Gradual typing: TS generics + Scala ordering
def quickSort[T](arr: Array[T])(using ord: Ordering[T]): Array[T] =
  if (arr.length <= 1) arr
  else {
    val pivot = arr(0)
    val (left, right) = arr.slice(1).partition(ord.lt(_, pivot))
    quickSort(left) ++ Array(pivot) ++ quickSort(right)
  }

// Platform-adaptive demo
@main def run(): Unit = {
  val data = Array(5, 3, 8, 1, 2)
  val sorted = quickSort(data)

  if (isJVM) {
    println("Sorted (JVM): " + sorted.mkString(", "))
  } else {
    console.log(`Sorted (JS): ${sorted.join(", ")}`)
  }
}
```

4.6 Stack

```
module Collections

class Stack[T] {
  private var elements: List[T] = List.empty  // Scala List, compiles to JS Array

  // TypeScript-style type parameter with Scala 'def'
  def push(item: T): Unit = {
    elements = item :: elements  // Scala cons operator
  }

  def pop(): T = {
    if (elements.isEmpty) {
      if (isJVM) throw new Exception("Empty stack")
      else throw Error("Empty stack")  // Platform-appropriate error
    }
    val head = elements.head
    elements = elements.tail
    head
  }

  def peek: T | null = elements.headOption.orNull  // TS union type + Scala Option

  def size: int = elements.length  // Unified property access
}

// Usage
@main def stackDemo(): Unit = {
  val stack = new Stack[number]()  // TypeScript-style angle brackets

  stack.push(10)
  stack.push(20)

  if (isJVM) {
    println(s"Top: ${stack.peek}")  // Scala string interpolation
    println(s"Popped: ${stack.pop()}")
  } else {
    console.log(`Top: ${stack.peek}`)  // TS template literal
    console.log(`Popped: ${stack.pop()}`)
  }
}
```