# JAVASCAR

## Language Specification

**Version .01**

**(Since I don't know what you expected on the home page, (home page? really?) I mean, front cover, I'm going to go ahead and just put my name, which was missing on yours (OCD intended))**

-Joey Carmello

# 1. Introduction

JavaSCAR, Which stands for "JavaScript - **S**trict **C**omments **A**re **R**equired" is a language dedicated to the readability of code by enforcing strict rules on where comments must be placed and what they must include. The language is much like JavaScript, it's major difference being the **C**omment **E**nforcement **N**otation, or CEN;. The three most notable differences include:

1. The CEN comment blocks. Although comments have not been replaced in order for developers to go above and beyond with their comments, CEN blocks offer an alternative by making what you intend to do very very obvious. CEN blocks are indicated by a leading /~.
2. A "Main" block has been added to more clearly define where the start point to the program actually is
3. A much stricter instruction set, several of JavaScript's most vague features such as inferred variable types have been replaced with declarations in favor of stronger readability and reliability.
4. Specific keywords have been added to indicate the scope of a variable.

## 1.1 Hello world

```
/~System.initialize: Main Block
Main {
        /~System.alert: string Output
        "Hello World".alert;
}
```

## 1.2 Program structure

The key organizational concepts in JAVASCAR are as follows:

1. File entry occurs at the "Main" block, preceded by the appropriate CEN line for file initialization.
2. Execution of instructions moves sequentially, blocks of code are bounded off by curly braces
3. CEN script precedes almost every line of code.

***Example: printing an array***

```
/~System.initialize: Main Block
Main {
        /~Var.declare: Array Full Of integers
        Local Array exampleArray = new Array(3,5,6,7,8);

        /~Control.loop: Transverse Array By Index i
        for (Local Integer i = 0; i < exampleArray.length; i++)
        {
                /~System.alert: integer Output
                exampleArray[i].alert;
        }
}
```

The above code begins with the common CEN script for an opening execution, begins by defining a Local Array and populating it on declaration. The CEN block preceding that indicates that this array will contain only Integers and that it will be pre-populated, as indicated by the "Full" keyword in the declaration. The loop creates a Local Integer as its index, and outputs the current Integer at each index in the Array.

## 1.3 Types and variables

Every variable in JAVASCAR is treated as a ***reference type*** more specifically ***class instances***. This allows any variable created to have its own member functions. I believe that a class member should have its class functions which utilize the personal attributes of that class best. While this may lead to more memory allocations than perhaps are necessary, it will add to the readability and reliability by guaranteeing that functions called on any given variable or member will be using the appropriate functions for that variable or member.

## 1.4 Statements Differing from JavaScript

**Statement** **Example**

Expression statement

```
/~System.initialize: Main Block
Main {
        /~Var.declare: integer
        Integer thisVariable = 5;

        /~Var.operation: addition
        thisVariable += 13;

        /~System.alert: integer Output
        thisVariable.alert;
}
```

if statement

```
/~System.initialize: Main Block
Main {
        /~Var.declare: integer
        Integer thisVariable = 5;

        /~Control.select: If
        if (args.Length = 0) {
                /~System.alert: integer Output
                thisVariable.alert;
        }
        /~Control.select: Else
        else {
                /~System.alert: integer Output
                thisVariable.alert;
        }
}
```

## 1.5 Classes and objects

New classes are created using class declarations.

The following is a declaration of a simple class named Tree:

```
/~System.initialize: Class
Class Tree {
        /~System.initialize: Main Block
        Main {
        }
}
```

Instances of classes are created using the new operator, which allocates memory for a new instance, invokes the 'Main' function to initialize the instance, and returns a reference to the instance. The 'Main' function replaces the constructor of a Class. If no parameters are specified in the 'Main" function, it is assumed nothing can be passed to it and you do not have to include them in the declaration of a new member of that Class. A capital 'Tree' is used to indicate that the Variable being declared is a user-defined class and that the CEN is to look for the appropriate constructor.

```
/~Var.declare: Tree
Tree oak := new Tree;
/~Var.declare: Tree
Tree maple := new tree;
```

### 1.5.1 Accessibility

Each member of a class has an associated accessibility, which controls the regions of program text that are able to access the member. There are three possible forms of accessibility. These are summarized in the following table.

| Accessibility | Meaning |
|---|---|
| Local | Access is restricted to the block it resides |
| Friend | allows an anonymous function access to any variables within the scope the anonymous function was called |
| Party | Access is open, kind of a mash-up of a static Class variable and a Global |

### 1.5.2 Fields

A field is a variable that is associated with a class or with an instance of a class.

```
/~System.initialize: Class
Class Tree {
        /~Var.declare: dynamic
        Dynamic leaves = 0;

        /~System.initialize: Main Block
        Main {
                /~Var.operation: addition
                this.leaves = 10;
        }
}
```

### 1.5.3 Functions

A *function* is a callable action of either a Class, or a static action of a given file. Functions are used much as they are in JavaScript; anonymous functions can be used as well as delegates to have variables act with certain functional qualities.

#### 1.5.3.1 Constructors

JAVASCAR uses the 'Main' function as its constructor. The 'Main' function defines what happens when a new instance of that class is defined. A constructor is called using the 'new' keyword. The CEN for implementing a construction is of the format: */~Var.declare Class* where *Class* is the capitalized name of the class to be instantiated.

#### 1.5.3.2 Properties

Class members have **implicitly** defined **getters** and **setters**. These **accessors** can be overridden by implementing the *getVariable* and *setVariable* functions, where *Variable* is the member to be accessed.

#### 1.5.3.3 Parties

A **Party** is a member that enables a class or object to provide notifications. It is much like the events JavaScript has always known and loved, as they are bound and called in just about the same way. Parties are attached to the DOM and referenced in the same way as JavaScript events. The following example shows a *onClick* party being attached to an html button, without the use of party planners.

***HTML***

```
<input type="button" id="testButton" onclick="testClick" value="click me!" />
```

**JavaSCAR**

```
/~Var.declare: Function void
function testClick {
        /~System.alert: string Output
        alert("Lets Party!");
}
```

Alternatively, you use a **party planner**, to directly bind a Party to a DOM object through JAVASCAR itself. The following is an example of this, accomplishing the same task as the previous through the use of a party planner:

***HTML***
```
<input type="button" id="testButton" value="click me!" />
```

**JAVASCAR**
```
/~System.initialize: Main Block
Main {
        /~Var.declare: Party Planner
        document.getElementById('testButton').planParty('click', testClick);
}

/~Var.declare: Function void
function testClick {
        /~System.alert: string Output
        alert("Let's Party!");
}
```
The 'planParty' function can be called on any DOM object that an event can be attached to. In this case, the 'click' event is bound to the 'testButton' input button, which when clicked, invokes the 'testClick' function set as the second parameter of the planParty function.

## 1.6 Arrays

An **array** is declared using the same style of CEN as any other variable. There are two ways to declare an array. In the first, you insert the data to be placed in it on declaration; in the CEN, you declare the Array to be 'Full Of' the type of which the contents will be associated with. Such as the following:

```
        /~Var.declare: Array Full Of integers
        Array exampleArray = new Array(3,5,6,7,8);
```
In the second, you declare the Array as any other variable, and load data in after the fact; in the CEN, you declare the Array to be Empty Of' the type of which the contents will be associated with.

```
        /~Var.declare: Array Empty Of integers
        Array exampleArray = new Array;

        /~Var.assign: Array
        exampleArrray[] = 4;
```
This code is shorthand for inserting a value into the next sequentially available slot in the array. You can also insert directly to an index.

```
        /~Var.assign: Array
        exampleArrray[5] = 20;
```

# 2. Lexical structure

## 2.1 Programs

A JAVASCAR ***program*** consists of one or more ***source files***. A source file is an ordered sequence of characters.

JavaSCAR programs are interpreted rather than compiled:

1. The browser finds the source file that was requested by the client, the browser acts as an interpreter which scans the file's code for those Unicode characters for those Unicode characters.

2. Lexical analysis, which translates a stream of Unicode input characters into a stream of tokens.

3. Syntactic analysis, which translates the stream of tokens into executable code.

## 2.2 Grammars

This specification presents the syntax of the JavaSCAR programming language where it differs from JavaScript.

### 2.2.1 Lexical grammar where different from JavaScript

The major lexical difference between JavaScript and JavaSCAR is the CEN statements that appear before almost every line of code. CEN statements are broken up into tokens and the statements proceeding them are all included as one statement.

### 2.2.2 Syntactic ("parse" ) grammar where different from JavaScript

Execution in JavaSCAR programs always begins at the 'Main' Block. The Main Block can be anywhere in the file, but all code outside the Main Block is inconsequential to the performance of the program. Statements appearing randomly outside of any defined function in a file are overlooked completely.

### 2.2.3 Grammar notation

The lexical and syntactic grammars are presented using **BNF *grammar productions***. Each grammar production defines a non-terminal symbol and the possible expansions of that non-terminal symbol into sequences of non-terminal or terminal symbols. In grammar productions, *non-terminal* symbols are shown in italic type, and terminal symbols are shown in a fixed-width font.

The first line of a grammar production is the name of the non-terminal symbol being defined, followed by a colon. Each successive indented line contains a possible expansion of the non-terminal given as a sequence of non-terminal or terminal symbols. For example, the production:

*while-statement:*
*/~Control.loop: While*
while ( *boolean-expression* ) *{*
*CEN statement*
*embedded-statement*
*}*

defines a *while-statement* to consist of the CEN statement */~Control.loop: While*, followed by a newline, followed by the token while, followed by the token "(", followed by *a boolean- expression*, followed by the token ")", followed by a '{', followed by an *embedded-statement w/ appropriate CEN statement before it, followed by an ending '}'.*

## 2.3 Lexical analysis

### 2.3.1 Line terminators

Line terminators divide the characters of a JAVASCAR source file into lines. (I didn't even know of some of these characters, but I really cannot say they should be excluded)

*new-line:*
- Carriage return character (U+000D)
- Line feed character (U+000A)
- Carriage return character (U+000D) followed by line feed character (U+000A)
- Next line character (U+0085)
- Line separator character (U+2028)
- Paragraph separator character (U+2029)

### 2.3.2 Comments

The major different between comments in JavaScript and comments in JavaSCAR is that JavaSCAR uses Comment Enforced Notation to get across a lot of the basics comments usually may convey. JavaScript comments, the // and /* */ notation, can be used to give more description to the statements, but CEN will be required to generalize the idea behind any statement. CEN statements appear before any line of code (except lines where just a '}' or '{' appear on).

### 2.3.3 White space

White space is defined as any character with Unicode class Zs (which includes the space character) as well as the horizontal tab character, the vertical tab character, and the form feed character. Tokens in JavaSCAR are separated just as they are in JavaScript.

*whitespace:*
- Any character with Unicode class Zs
- Horizontal tab character (U+0009)
- Vertical tab character (U+000B)
- Form feed character (U+000C)

## 2.4 Tokens

There are several kinds of tokens: identifiers, keywords, literals, operators. White space and comments are not tokens, though they act as separators for tokens.

*token:*
- *identifier*
- *keyword*
- *integer-literal*
- *real-literal*
- *character-literal*
- *string-literal*
- *operator-or-punctuator*

Valid tokens in JavaSCAR: Var.declare, Var.assign, Var.destroy, System.alert, System.initialize, System.function, Control.loop, Control.select, function, integer, string, float, Array, Class, Main, Dynamic, Full Empty, Of, call, and any user-defined identifiers or names.

### 2.4.1 Keywords different from JavaScript

A ***keyword*** is an identifier-like sequence of characters that is reserved, and cannot be used as an identifier.

*New keywords:*

Local Class friend Dynamic Party

- all of the CEN statements:

/~Var /~System /~Control

*Removed keywords:*

var

# 3. Basic concepts

## 3.1 Application Startup

***Application startup*** occurs when the execution environment calls a designated method, which is referred to as the application's entry point. This entry point method is always named Main, and it is always defined by the following signature:

```
/~System.initialize: Main Block
Main {
}
```

While JavaScript programs do not have this 'Main' block at all, I find it increases the readability by defining a clear location for code to begin execution. Doing so reduces errors such as when code segments are placed outside of all function statements in JavaScript, making variables global and potentially creating unnecessary side effects. The main function cannot return a value, and other functions cannot override the Main function.

## 3.2 Application termination

JavaSCAR programs terminate when execution reaches the end of the 'Main' block defined in a file. The curly bracket at the end of this function marks the exact end point of the program. Execution control returns to the calling function or execution environment, whichever is above in the execution hierarchy.

## 3.3 Scope

*JavaSCAR will be a statically scoped language. Forced Local Variable declaration helps define exactly which variables are being referenced. The first example shows a general static structure for a basic program.*

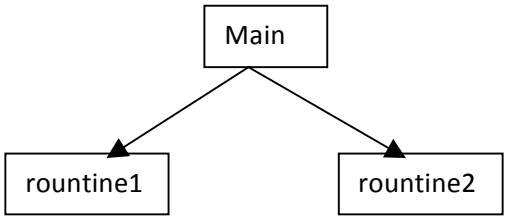
```
/~System.initialize: Main Block
Main {
        /~Var.declare: integer
        integer x = 5;

        /~System.function: Call
        routine1;
}

/~Var.declare: Function void
function routine1 {
        /~Var.declare: integer
        integer x = 6;

        /~System.function: Call
        routine2;
}

/~Var.declare: Function void
function routine2{
        /~System.alert:integer Output
        Party x.alert;
        /~System.alert:integer Output
        Friend x.alert;
}
```




This program will alert 5, and then 6. The 'Party' keyword retreives the value of x at the highest point in the call stack as possible, whereas the 'Friend' keyword retrieves the value of x from the function who called the subroutine it is in.

### 3.4 Automatic memory management

JAVASCAR employs automatic memory management, which frees developers from manually allocating and freeing the memory occupied by objects. Automatic memory management policies are implemented by a ***garbage collector***. It differs from JAVASCRIPT in the following ways:

1. At the end of a scope's execution, all objects that were explicitly instantiated during this scope are removed, unless they are being pointed at from outside the scope.
2. The 'Local' keyword is used to mark a particular variable as local to the current scope it is in.

# 4. Types

Every variable in JAVASCAR is treated as a ***reference type*** more specifically a ***class instance***.

## 4.1 Reference types in JavaSCAR

*Examples*

*To define a reference variable in JavaSCAR (which is all of them), use the /~Var.assign type CEN statement*

```
/~Var.assign: integer
integer exampleInteger = 4;
```

*You are then able to call integer class functions on the defined integer member. For example:*

```
/~System.function: Call
exampleInteger.divide(2);
```

*Would invoke the 'divide' function on the exampleInteger member.* In the same way, you can invoke functions on what would usually be considered primitive values. The following example is correct:

```
/~System.function: Call
6.divide(2);
```

# 5. Variables

Variables represent storage locations. Every variable has a type that determines what values can be stored in the variable. JAVASCAR requires that all variable type be explicitly defined on creation (unless the 'dynamic' type is used).

## 5.1 Variable categories

*Variable are of the following types:*

*integer*
/~Var.assign: integer
integer exampleInteger = 4;

*float*
/~Var.assign: float
float exampleFloat = 4.4245943505;

*string*
/~Var.assign: string
string exampleString= "this is a string!";

*Array*
/~Var.assign: Array
Array exampleArray = [2,3,4,5,6,7];

*Dynamic*
/~Var.assign: *Dynamic*
Dynamic exampleDynamic = 4;
/~Var.assign: *Dynamic*
exampleDynamic = "You can just change this to a string since it's dynamic";

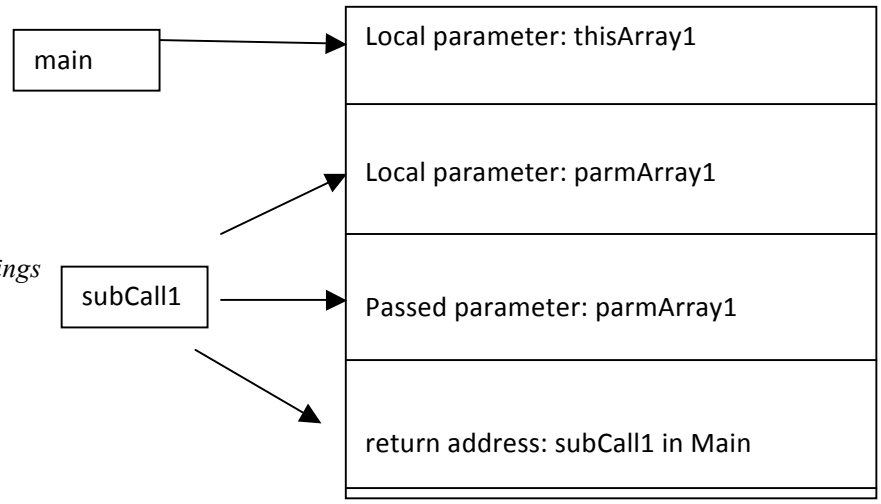
or of course, anything user-defined in a class. The following are modifiers for variables that control their scope, as seen in section 3.3 on scope. They can be pre-pended to any variable to change the accessibility of the variable.

*Party*
*Friend*
*Local*

# 6. Parameter Passing

### 6.1 Method

In JavaSCAR, all parameter passing is In-Out, and always pass by reference. Pointers are always sent to called functions since all variables and members are reference-based. This saves time because of the lack of copying, and no redundant storage in memory.




### 6.2 Examples

```
/~System.initialize: Main Block
Main {
        /~Var.declare: Array Empty of strings
        Array thisArray1 = [];

        /~System.function: Call
        subCall1(thisArray1);
}

/~Var.declare: Function void
function subCall1 (Array parmArray1) {
        /~Var.declare: Array Full Of strings
        Local Array parmArray1 = ["n", "w", "d", "g", "h"];

        /~Var.assign: Array
        Friend parmArray1 = ["y", "b", "v", "c"];
}
```

Here you can see the implications of using the Local and Friend keywords. The Local keyword allows you to create a brand new instance of Array with the exact same name as the parameter passed, without overwriting it. The Friend keyword forces the statement to utilize the parmArray1 variable that was passed in from the calling program. This example allows two variables with the same name to exist within the scope, if for some reason you really wanted to do that.

# 7. Conversions

A ***conversion*** enables an expression to be treated as being of a particular type.

### 7.1 Implicit conversions

The only instance of implicit conversion in JavaSCAR is in regard to what would normally be considered primitive types. When a function is invoked on an integer or what have you, it is implicitly referring to the integer Class instead, which has all the Class functions needed to interact with it. For example:

```
/~System.function: Call integer
67.multiply(5);
```

67 is implicitly treated as the integer type in order to use the integer class's multiply function. You cannot, on the other hand, attempt to call the multiply function on a string which appears to be like an integer. For instance:

```
/~System.function: Call integer
"67".multiply(5);
```

*Will throw an exception because the string class does not have a multiply function within it.*

### 7.2 Explicit conversions

In order to combat the above issue where the multiply function perhaps should be able to work on the given string, an explicit conversion could be used as follows:

```
/~System.function: Call integer
"67".conversion.multiply(5);
```

*All classes contain the 'conversion' function, which looks ahead at the next function, and attempts to convert the invoking member into an appropriate useable form for that function. In this case, the conversion function looks at multiply, sees that it is a member of the integer class, and appropriately converts 67 to an integer for the given operation.*

# 8. Statements

Examples of statements in JAVASCAR that differs from JAVASCRIPT , as noted in the introduction one:

1. The CEN comment blocks. Although comments have not been replaced in order to allow developers to go above and beyond with their comments, CEN blocks offer an alternative by making what you intend to do obvious. CEN blocks are indicated by a leading /~. Here are what they mean.

/~ System
These are for system related tasks such as initializing the program or a class, function calls, and system alerts

/~Var
These allow the allocation of memory for new variables, the reassignment of variables, the definition of functions, and the use of operators on variables.

/~Control
These indicate that a control structure is to be used, as well as how it is carried out, such as a for and while loop, and if else blocks.

2. A "Main" block has been added to more clearly define where the start point to the program actually is. This is used purely to better show where a file starts. The Main block cannot receive arguments and does not return a value.

```
/~System.initialize: Main Block
Main {
        .. program execution begins here...
}
```

3. A much stricter instruction set, several of JavaScript's most vague features such as inferred variable types have been replaced with declarations in favor of stronger readability and reliability. Now, if you want to produce a variable that can hold any data type (or at least change data types) you need to use the 'Dynamic' keyword as the variable type.

```
/~Var.declare: Dynamic
Dynamic f = "a string?";

/~Var.assign: Dynamic
f = 2;
```

4. Specific keywords have been added to indicate the scope of a variable.

```
/~System.initialize: Main Block
Main {

        /~Var.declare: integer
        local integer x = 4;
}
/~Var.declare: function void
function partyItUp {
        /~Var.declare: integer
        integer x = 7;

        /~System.function: call
        Party x.alert;
}
```

The above will alert 4 because the Party Keyword jumps access to the x defined outside the scope of the subroutine. Despite being defined as local, 'x' can be called by any subroutine that tries to call it with the 'Party' keyword.

```
/~System.initialize: Main Block
Main {
        /~Var.declare: integer
        integer y = 4;

        /~System.function: Call
        subCall1;
}
/~Var.declare: Function void
function subCall1 {
        /~System.function: call
        Friend y.alert;
}
```

The above will alert 4 because the Party Keyword jumps access to the x defined outside the scope of the subroutine. If 'y' was defined as local you would not be able to access it with a friend call.

# 9. Example Programs

**Reversing an array:**

```
/~System.initialize: Main Block
Main {
        /~Var.declare: string
        string my_str="A String to reverse";

        /~Var.declare: integer
        integer i=my_str.length;

        /~Var.assign:integer
        i = i - 1;

        /~Control.loop: Transverse Array By Index x
        for (Local integer x = Friend i; x >= 0; x--)
        {
                /~System.function: call
                my_str.charAt(x).alert;
        }
}
```

**sorting an array:**

```
/~System.function: call
array.sort(sortfunction);

/~Var.declare: Function boolean
function sortfunction(a, b){

        /~System.return: boolean
        return a < b;
}
```

**find position of element in array:**

```
/~Var.declare: Function integer
function getPosition(arrayName,arrayItem)
{
	/~Control.loop: Transverse Array By Index i
	for(Local integer i = 0; i < arrayName.length; i++)
	{
		/~Control.select: If
		if(arrayName[i]==arrayItem)
		{
			/~System.return: boolean
			return i;
		}
	}
}
```

**find even/odd numbers:**

```
/~Var.declare: Function boolean
function iseven(someNumber)
{
	/~Control.select: If
	if (someNumber % 2 == 0)
	{
		/~System.return: boolean
		return true;
	}
	/~Control.select: Else
	else {
		/~System.return: boolean
		return false;
	}
}
```

# 10. Conclusion

On the outside, JavaSCAR looks like a weighed down version of JavaScript, but with key new features, I believe it vastly surpasses it in usability. The new Comment Enforced Notation forces a programmer into good habits; by making the programmer think before he or she inserts whatever code he or she wants, it is a requirement to explicitly say what you want to do before doing it. Consequently, when the programmer goes back to look at his or her code after a period of time away from it, they will easily be able to follow along. This is especially true if the individual reading the code is not the programmer who wrote it. They would be able to follow complex programs along a line of CEN comments, and not worry about the specifics until after getting a general overview of the code.

The majority of CEN statements have a field for data type. This is imperative towards the goals JavaSCAR attempts to meet: to make programming more readable and harder to make logical errors, the reliability will be improved. The 'var' keyword used in JavaScript has been replaced by declarative and expressive alternatives such as 'integer' and 'float'. JavaScript even allows you to declare variables without the 'var' keyword, making code even harder to follow as anything could be assigned a value and you would not know it happened without directly following the code. CEN statements direct the reader's attention to exactly what the programmer is trying to do with the code.

By instituting a 'Main' block, reliability is drastically improved as the programmer will always know what exactly is contained in the activation record of his or her program. In JavaScript, you could define a variable on the very bottom of a file, and it would be included in the runtime environment because there is no clear-cut definition of where a program starts and ends. The file itself is the limit, and I believe that this reduces the level of reliability. A 'Main' block marks the exact point in the file where program execution will begin. Somebody foreign to the code can look at a JavaSCAR file and not have to spend much time figuring out what happens when the file is executed.

Unfortunately, with these improvements in readability and reliability, writability suffers. The number of ways to do things has been severely decreased with the subtraction of 'var' keyword and the addition of CEN statements. Programmers must learn all of the appropriate CEN statements to go along with the tasks they intend to do. Further, CEN statements can possibly get in the way, forcing a programmer to take longer in what they are trying to by including them as they go. An IDE could possibly add CEN statements on-the-fly as the programmer typed out their code, correctly perhaps minor inaccuracies in variable types and scope.