Data types, Functions& Objects in JS

Function:

Functions in JavaScript are declared using the `function` keyword. A function declaration creates a function that's a *Function* object having all the properties, methods, and behaviors of Function objects. By default, functions return the value `undefined`; to return any other value, the function must have a return statement that consists of the `return` keyword followed by the value to be returned (this can be a literal value, a variable, or even a call to a function).

Function expression:
A function expression is very similar to (and has almost the same syntax as) a function statement. The main difference between a *function expression* and a *function statement* is the function *name*, which can be omitted from a function expression to create an anonymous function. Function expressions are often used as Immediately Invoked Function Expressions (IIFEs), which run as soon as they're defined.

Eg:

```
function main(input) {

    /**
    *   Defines an unnamed function and assigns it to a variable named square.
    *   @param {Number} x
    *   @returns {Number} The value of argument squared.
    **/
    var square = function(x) {
        return x * x;
    };

    // Print the value returned by passing input as x to the
    // anonymous function referenced by variable square
```

```
    console.log(square(input));
}
```

Types of functions:

- **Named functions**
- **Anomymous functions**
- **Immediately invoked function expressions.**

Eg of custom function :

```
function findBiggestFraction(a, b){
      a>b ? console.log(“a ”, a) : console.log(“b”, b);
}
var a = 3/4;
var b = 5/7;
findBiggestFraction(a, b);
```

Lexical Structure

The lexical structure of a programming language is the set of elementary rules that tells you *how* to write programs in that language. It's essentially the lowest-level syntax of a language and specifies such things as what variable names look like, the delimiter characters for comments, and how one program statement is separated from the next.

Character Set

- JavaScript programs are written using the *Unicode* character set. Unicode is a superset of *ASCII* and *Latin-1*.
- JavaScript is a case-sensitive language.
- JavaScript ignores spaces that appear between tokens in programs. For the most part, JavaScript also ignores line breaks.

Printing Output

We use the *console.log* method to write data to standard output in JavaScript.

```
//Example
function main() {
    console.log("You entered the following text in the Input box:");

    const input = readLine();
    console.log(input);
}
```

Comments

JavaScript supports two styles of comments, as demonstrated below.

Inline Comments

Any text between a `//` and the end of a line is ignored by JavaScript and treated as a comment:

```
console.log("This is an instruction that won't be ignored.");
// This is an inline comment and will be ignored
```

Block Comments

Any text between `/*` and `*/` is also treated as a comment:

```
console.log("This is an instruction that won't be ignored.");
/*
 * This is a block comment and will be ignored
 */

console.log("This is an instruction that won't be ignored.");

/*
 * This is part of our block comment and will be ignored
 * This is part of the same block comment and will be ignored
 */
```

Literals

A literal is a data value that appears directly in a program. For example:

// The integer number twelve:

12

// The floating-point number one-point-two:

1.2

// A string of text:

"Hello, World."

// Another string:

'Hi!'

// A boolean value:

true

// The absence of an object:

null

More complex expressions can serve as array and object literals.

// An object initializer:

{x: 1, y: 2}

// An array initializer:

[1, 2, 3, 4, 5]

Identifiers:

An identifier is simply a name that you can specify and use as a means of referring back to a specific value or other piece of code. In JavaScript, identifiers are used to name variables and functions, as well as to provide labels for certain code loops.

A JavaScript identifier must begin with a letter, an underscore (_), or a dollar sign ($). Subsequent characters can be letters, underscores, dollar signs, or digits (i.e., the numbers
through

). Like many other languages, JavaScript doesn't allow digits as the first character of an identifier because it makes them more easily distinguishable from numbers.

// Some valid identifiers are:
x
variable_name
sum13
_variable
$variable

A number of identifiers are reserved words or keywords, meaning they are part of a set of predefined words that have special meaning in the language itself. You cannot use these words as identifiers in your programs. For example, for and function are reserved words in JavaScript. In addition, there are a number of predefined global variables and functions; it's important to avoid using these predefined names for your own variables and functions.

Optional Semicolon:

Like many programming languages, JavaScript uses the semicolon (;) to separate statements from each other. This is important as it makes the meaning of your code clear; without a separator, the end of one statement might appear to be the beginning of the next (and vice versa). In JavaScript, you can usually omit the semicolon between two statements as long as those statements are written on separate lines.

Data Types

JavaScript's Data Types

The latest ECMAScript standard defines seven data types:

- A *primitive* value or data type is data that is not an object and has no methods. All primitives are immutable, meaning they cannot be changed. There are six primitive types:
    - *Number*
    - *String*
    - *Boolean*
    - *Symbol*
    - *Null*
    - *Undefined*
- The seventh data type is *Object*

Number Data Type

According to the ECMAScript standard, all numbers are double-precision 64-bit binary format IEEE 754-2008, meaning there is no specific type for integers.

Maximum Value for a Number

The `MAX_VALUE` property has a value of approximately

. Values larger than `Number.MAX_VALUE` are represented as `Infinity`.

Minimum Value for a Number

The `MIN_VALUE` property is the smallest positive value of the *Number* type closest to

, not the most negative number, that JavaScript can represent.

`MIN_VALUE` has a value of approximately . Values smaller than `Number.MIN_VALUE` ("underflow values") are converted to 0.

Symbolic Numbers

There are three symbolic number values:

- `Infinity`: This is any number divided by

, or an attempt to multiply `Number.MAX_VALUE` by an integer

- .

- `-Infinity`: This is any number divided by , or an attempt to multiply `Number.MAX_VALUE` by an integer
- .
- `NaN`: This stands for "Not-a-Number" and denotes an unrepresentable value (i.e.,
-).

The *isSafeInteger* Method

A *safe integer* is an integer that:

- Can be exactly represented as an IEEE-754 double precision number, and
- Whose IEEE-754 representation cannot be the result of rounding any other integer to fit the IEEE-754 representation.

The `Number.isSafeInteger()` method determines whether the provided value is a number that is a safe integer.

Maximum Safe Integer

The `Number.MAX_SAFE_INTEGER` constant has a value of

, or

.

Minimum Safe Integer

The `Number.MIN_SAFE_INTEGER` constant has a value of

, or .

String Data Type

A string value is a chain of zero or more Unicode characters (i.e., letters, digits, and punctuation marks) that we use to represent text. We include string literals in our scripts by enclosing them in single (`'`) or double (`"`) quotation marks. Double quotation marks can be contained in strings surrounded by single quotation marks (e.g., `'"'` evaluates to `"`), and single quotation marks can be contained in strings surrounded by double quotation marks (e.g., `"'"` evaluates to `'`).

Notice that JavaScript does not have a type to represent a single character. To represent a single character in JavaScript, you create a string that consists of only one character. A string that contains zero characters ("") is an empty (zero-length) string.

Unlike in languages like C, JavaScript strings are immutable. This means that once a string is created, it is not possible to modify it. However, it is still possible to create another string based on an operation on the original string. For example:

- A substring of the original by picking individual letters or using `String.substr()`.
- A concatenation of two strings using the concatenation operator (`+`) or `String.concat()`.

Boolean Data Type

A boolean represents a logical entity and can have one of two literal values: `true`, and `false`.

Symbol Data Type

Symbols are new to JavaScript in ECMAScript Edition 6. A Symbol is a unique and immutable primitive value and may be used as the key of an Object property.

Null Data Type

The null data type is an internal type that has only one value: `null`. This is a primitive value that represents the absence of any object value. A variable that contains null contains no valid number, string, boolean, array, or object. You can erase the contents of a variable (without deleting the variable) by assigning it the null value.

Undefined Data Type

The undefined value is returned when you use an object property that does not exist, or a variable that has been declared, but has never had a value assigned to it.

The *typeof* Operator

As demonstrated in some of the code examples above, we can use the `typeof` operator to determine the type associated with a variable's current value

Variables

Dynamic Typing:

JavaScript is a loosely typed or *dynamic* language, meaning you don't need to declare a variable's type ahead of time and the language autmatically determines a variable's type while the program is being processed. That also means that you can reassign a single variable to reference different types.

Naming:

JavaScript is a case-sensitive language, meaning that a variable name such as `myVariable` is different from the variable name `myvariable`. Variable names can be of any length, and the rules for creating legal variable names are as follows:

- The first character must be either an ASCII letter (uppercase or lowercase) or an underscore (_). Note that a number *cannot* be used as the first character.
- Subsequent characters can be ASCII letters, underscores, or digits (e.g., the numbers

through).

- The variable name must not be a reserved word.

Declaration and Initialization:

The first time a variable appears in your script is considered its *declaration*. The first mention of the variable sets it up in memory, and the name allows you to refer back to it in your subsequent lines of code. You should declare variables using the `var` keyword before using them. If you do not initialize a variable that was declared using the `var` keyword, it automatically takes on the value `undefined`.

Coercion:

In JavaScript, you can perform operations on values of different types without raising an exception. The JavaScript interpreter implicitly converts, or coerces, one of the data types to that of the other, then performs the operation. The rules for coercion of string, number, or boolean values are as follows:

- If you add a number and a string, the number is coerced to a string.
- If you add a boolean and a string, the boolean is coerced to a string.
- If you add a number and a boolean, the boolean is coerced to a number.

```
// Write code that uses console.log to print the sum of the 'firstInteger' and 'secondIn
teger' (converted to a Number        type) on a new line.
  console.log(firstInteger + Number(secondInteger));

  // Write code that uses console.log to print the sum of 'firstDecimal' and 'secondDe
cimal' (converted to a Number           type) on a new line.
  console.log(firstDecimal + Number(secondDecimal));
```

```
  // Write code that uses console.log to print the concatenation of 'firstString' and 'se
condString' on a new line. The       variable 'firstString' must be printed first.
  console.log(firstString + secondString);
```

Recursion

This is an extremely important algorithmic concept that involves splitting a problem into two parts: a *base case* and a *recursive case*. The problem is divided into smaller subproblems which are then solved recursively until such time as they are small enough and meet some base case; once the base case is met, the solutions for each subproblem are combined and their result is the answer to the entire problem.

If the base case is not met, the function's recursive case calls the function again with modified values. The code must be structured in such a way that the base case is reachable after some number of iterations, meaning that each subsequent modified value should bring you closer and closer to the base case; otherwise, you'll be stuck in the dreaded infinite loop!

It's important to note that any task that can be accomplished recursively can also be performed iteratively (i.e., through a sequence of repeatable steps). Recursive solutions tend to be easier to read and understand than iterative ones, but there are often performance drawbacks associated with recursive solutions that you're going to want to evaluate on a case-by-case basis. Typically, we use recursion when each recursive call significantly reduces the size of the problem (e.g., if we can halve the dataset during each recursive call). Regardless of the advisability of recursively solving a problem, it's extremely important to practice and understand *how* to recursively solve problems.

Variable Declaration Keywords:

`var`

We use the *var* keyword to declare variables. The scope of a variable declared using this keyword is within the context wherever it was declared. For variables declared outside any function, this means they are globally available throughout the program. For variables declared within a function, this means they are only available within the function itself.

`let`

We use the *let* keyword to declare variables that are limited in scope to the block, statement, or expression in which they are used. This is unlike the *var* keyword, which defines a variable globally or locally to an entire function regardless of block scope.

`const`

We use the *const* keyword to create a *read-only* reference to a value, meaning the value referenced by this variable cannot be reassigned. Because the value referenced by a constant variable cannot be reassigned, JavaScript *requires* that constant variables always be initialized.