Preparations

A **runtime environment** is an execution environment that lets an application program access system resources and provides the tools the application needs to operate.

A **programming language** is a set of syntactic and semantic rules that describe a formal language and must be able to express the steps that a machine needs to perform to convert some input to an output.

Before the computer can perform those steps, though, the program needs to be converted to a form the computer can understand. Programs called **compilers** and **interpreters** perform this conversion. At their most basic level, compilers and interpreters are similar: they both translate a program written in a specific programming language to something that the computer can execute, typically referred to as "1s and 0s" since all information on digital computers gets stored as 1s and 0s.

Compilers and interpreters differ primarily in what they do with those 1s and 0s. Compilers produce an output file that the computer can run directly, perhaps after some additional processing called linking. Interpreters, however, don't produce 1s and 0s that the computer can run directly - instead, the interpreter runs the interpreted code directly, or perhaps passes it on to a companion program.

JavaScript, at its heart, is an interpreted language. However, for performance reasons, most modern JavaScript engines employ special kind of compiler called a **Just In Time (JIT) compiler** that takes the interpreted code one step further and lets the computer run the result.

When an application accesses an operating system's resources, something must ensure that the operating system provides them in a regulated and safe manner. An **Application Programming Interface (API)** describes the scheme and format that a programmer can use to securely access resources, with the operating system acting as an intermediary.

An API also refers to how applications talk to each other.

JavaScript in the browser has two main purposes: 1) to programmatically alter web pages based on user actions; and, 2) to exchange messages with a server over a network.

Node.js is a runtime environment that turns JavaScript into a general-purpose programming language that can run applications on almost any system.

A general-purpose programming environment, like Node.js, needs the following minimal capabilities:

- The ability to read and write disk files (disk I/O);
- The ability to read and write via the terminal (standard I/O);
- The ability to send and receive messages over a network (network I/O); and
- The ability to interact with a database.

Naming Conventions:

- Use **camelCase** formatting for most variable and function names.
- Some function names (i.e., constructor functions) should use **CamelCase**.
- Use uppercase names with underscores (i.e., SCREAMING_SNAKE_CASE) to represent constants that serve as **unchanging configuration values** in your program.

A REPL (read-eval-print-loop) for a programming language is an interactive environment where you can type commands and expressions in that language and get immediate results. To exit the Node.js REPL just hit 'CTRL+D'.

Once you create a JavaScript file, one with a **.js** extension, you can run the code in that file by typing the **node** command followed by the file name (i.e., **node myJSCode.js**). Use 'CTRL+C' to exit the program.

Running code from a file in a browser is more complicated than running it from the command line. To do so, you must first embed the code in an HTML file. You can do this by following the steps:

- Add a script tag to the HTML file that has the path to your **.js** file in the **src** attribute:
    - **<script src="example.js"></script>** — we put this in between the **<body>** attribute.

- Save the HTML file, then open it in a browser. Your code executes when the browser loads the file.
- Another, perhaps simpler, way to run JavaScript in the browser is to write all your JavaScript between the opening and closing **<script>** tags, don't use a **.js** file at all.

JavaScript's documentation focuses on the available data types, and the operations you can perform on those types.

Typically, a programming environment provides two main types of reusable code to an application: 1) components and operations that are part of the core language, often collectively referred to as the **standard library**; and, 2) components and operations specific to a runtime environment.

The official documentation of JavaScript standards can be found at **ECMA International**. When you really need to know how JavaScript is supposed to work, the standards are where you need to look.

For a more accessible read, the **Mozilla Developer Network (MDN)** provides a wealth of documentation about JavaScript as provided by the major browsers and Node.js. Not just for JavaScript, but HTML, CSS, and browser APIs as well.

For now, think of constructors as factories that create values of a particular type. A constructor represents a blueprint for a data type with the same name. The String constructor, for example, is a factory that creates values of type string.

The documentation page for a constructor describes how you can use that constructor to create values. It also lists the operations you can perform on values of that type.

In JavaScript, methods are functions that need a value that you can use to call the function. For example, to call the **toUpperCase()** string method, you need a string to call it:

- **'xyz'.toUpperCase()**

The MDN documentation shows two types of methods: **instance methods** and **static methods**. It uses the format **Constructor.prototype.methodName()** to refer to instance methods, and **Constructor.methodName()** for static methods.

MDN uses a number of different icons next to the property and method names on the left side of the screen. The following icons are the one you're most likely to encounter:

- A trash icon. This icon identifies deprecated items. A deprecated item may not work in future versions of the language, so you shouldn't use those items in new code.
- A thumbs-down icon. This icon identifies non-standard items. These items should not be used without understanding that they may not work on all platforms.
- A breaker icon. This icon identifies experimental items. These items should not be used in production applications as the behaviour may change or be removed at some time in the future.

As well as operations, some data type have properties associated with them. A data type's property is a noun; an operation is a verb. A property says something about the value and an operation does something with that value.

You can access a property by appending a dot and the name of the property to that value.

The big take away is that you apply instance methods to a value of the type that the constructor represents.

The Basics

Data Types

A core job of most programming languages is dealing with data. Different **data types** represent different kinds of data. Data types help programmers and their programs determine what they can and cannot do with a given piece of data.

JavaScript has five so-called **primitive data types**:

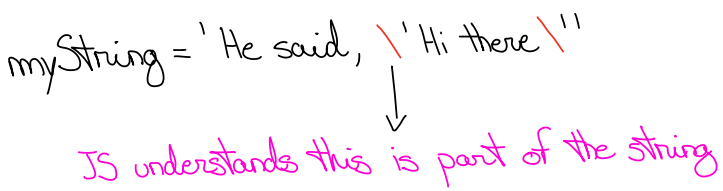
- String
- Number
- Undefined
- Null
- Boolean

Every type that is not a primitive type is an **object type**.

In most of your code, you should use the capitalized name when you're referring to a built-in feature of JavaScript.

Data type values can be represented by **literals**. A literal is any notation that lets you represent a fixed value in source code.

A **string** is a list of characters in a specific sequence. You write string **literals** with either single quotes or double quotes on either side of the text. Note that the quotes are **syntactic** components (i.e., not part of the value).



The backslash, or escape character (****), tells the computer that the next character isn't syntactic, but is part of the string.

A recent JS addition is template literals. They use backticks (**`**) and enable an operation called **string interpolation** (i.e., **${myVariable}**).

JS has a single data type, Number, that represents all types of numbers.

Boolean values represent an "on" or "off" state. There are two boolean literal values: **true** and **false**. Boolean values have a starring role when working with comparison operators.

In programming, we need a way to express the absence of a value. In JS, we do this with the value **undefined**. When a variable is not defined, its value is given by **undefined**. We can describe **undefined** as representing the absence of value.

Null is similar to **undefined**: it represents the intentional absence of a value. Often, **null** represents emptiness or nothing. The chief difference between **null** and **undefined** is that you must use **null** explicitly if you want to use it; **undefined** can arise implicitly.

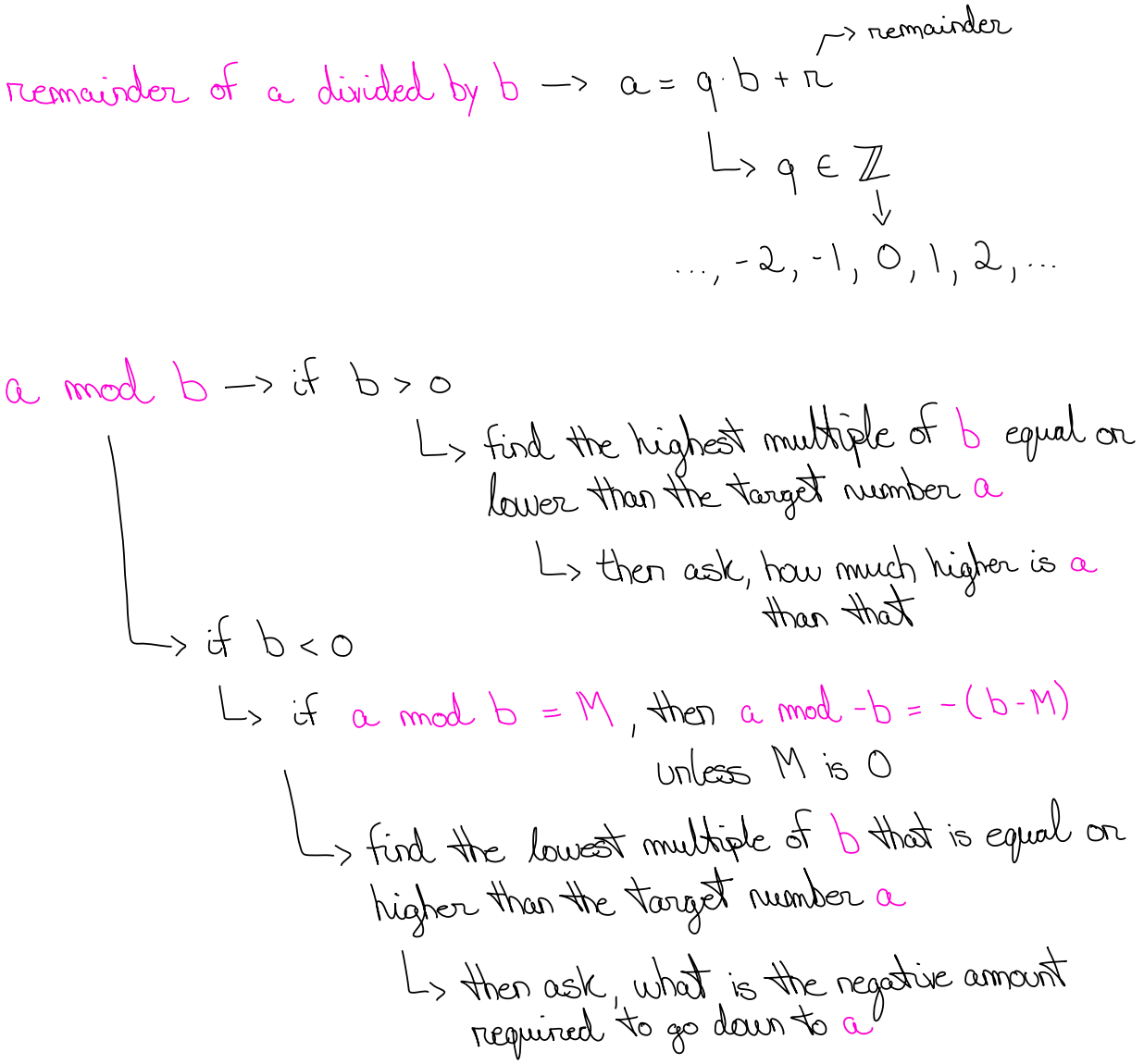
Every value you use in your JS programs has a data type. To see what type a particular value has, you can use the **typeof** operator (i.e., **typeof myVariable**). It returns a string that contains the type of its operand's value.

Operations

JS has a **remainder operator**, **%**. It returns the remainder of a division operation. For example, **16 % 5** returns 1.

Note that JS **%** operator computes the remainder of dividing two numbers; it does not compute the modulo value, nor does it have any built-in methods that will compute the modulo value. The difference between the two can be summarized as follows:

- Remainder operations return a positive integer when the first operand is positive, and a negative integer when the first operand is negative.
- Modulo operations return a positive integer when the second operand is positive, and a negative integer when the second operand is negative.



NaN values arise in two main situations:

- Undefined mathematical operations, such as dividing 0 by 0 or trying to take the square root of a negative number.
- Trying to convert a non-number value, such as **'Hello'**, to a number can also return **NaN**.

To determine whether a value is **NaN**, you can't use the usual comparison operators in a simple way. As it happens, **NaN** is the only value in JS that is not equal to itself. Instead, you should use either **Number.isNaN(myVariable)** or **Object.is(myVariable, NaN)**.

Other mathematical operations, such as **0/1**, return **Infinity**. You can safely use comparison operators to determine whether a value is **Infinity** or **-Infinity**.

When using **+**, if either operand is a string and the other is not, JS coerces the non-string operand to a string; thus, the result is always another string.

Explicit Coercion

The difference between explicit and implicit coercion is that explicit coercion lets you decide what you want to do, whereas implicit coercion lets the engine choose.

The **Number** function coerces a string to a number. It takes a sting value as an argument and returns a number if the string contains a valid numeric value.

You can also use the **parseInt** function to coerce strings to numbers. As the name suggests, **parseInt** parses an integer from a string. A surprising feature of this function is how it handles a string whose value begins with a numeric digit or a **+** or **-** sign followed by a digit. In both cases, it returns a numeric value, even though the remaining characters may not be numeric. It stops converting and ignores everything else once it encounters an invalid character.

A similar function, **parseFloat**, coerces a string to a floating-point (decimal) number.

The **String** function coerces numbers into strings.

Data Structures

The two most common data structures, or complex data types, that JS programmers use are arrays and objects.

JS organizes information into ordered lists using **arrays**. They may contain strings, numbers, booleans, or any other data type. In JS, array literals use square brackets **[ ]** surrounding a comma-delimited list of values, otherwise known as elements.

You can access an array element by placing its index inside brackets **[ ]**.

Arrays can also be written in a multi-line format, which is essentially useful for larger arrays or arrays with long values. The trailing comma on the last item in the list is optional, but a common practice.

The most important facts to remember about arrays are:

- The order of the elements is significant.
- Use index numbers to retrieve array elements.
- Index numbers are non-negative integers starting from **0**.

JS **objects** have many use cases, but the one that interests us most now is a dictionary-like data structure that matches keys with specific values. Essentially, a JS object is a collection of key-value pairs.

You can create objects using object literals, which have zero or more key-value pairs separated by commas all embedded within curly braces (**{ }**). A key-value pair associates a key and a given value. Each pair consists of a key, in the form of a string, and a value of any type. Key-value pairs in object literals use the key followed by a colon (**:**) and then the value.

Instead of index values (i.e., 0 or 1 or 2 or …), though, we use key names in string form.

Objects can also be written in a multi-line format, which is especially useful for larger objects or objects with long or complex values. The trailing comma on the last property in the object is optional, but a common practice.

Objects are the building blocks of programming.

Expressions and Return Values

An expression is anything that JS can evaluate to a value, even if that value is **undefined** or **null**. With few exceptions, almost everything you write in JS is an expression. JS expressions evaluate to a value that can be captured and used in subsequent code.

The evaluated value of your expression is the **return value**.

The term **log** is a synonym for printing or displaying something on the console.

It's important to realize that the output and return value are different concepts.

MDN Statements

JS also has **statements**. You can find a complete list of statements on MDN. Statements often include expressions as part of their syntax, but the statement itself is not an expression - its value cannot be captured and reused later in your code.

The key difference between a statement and an expression is that you can't capture a value from a statement.

A **statement** is a line of code commanding a task. Every program consists of a sequence of statements. By this definition, every chunk of code that can be treated as a single unit is a statement. This includes:

- variable, function, and class declarations
- loops and **if** statements
- **return** and **break** statements
- assignments such as **a = 3**
- standalone expressions such as **console.log("Hello")**

Most developers use the term "statement" in the broader sense: any syntactic unit of code that expresses an action for the computer to perform.

Expressions can be part of a statement, but not all statements can be part of an expression.

Exercises

When you compare two strings, JS performs a character-by-character comparison going from left to right.

Variables

Think of variables as containers that hold information: their purpose is to label and store data in memory so that your program can use it.

Variables and Variable Names

A variable is simply a named area of a program's memory space where the program can store data.

A variable name must accurately and succinctly describe the data that the variable contains.

Variable names are often referred to by the broader term, **identifiers**. In JS, identifiers refer to several things:

- Variable names declared by **let** and **var**
- Constant names declared by **const**
- Property names of objects
- Function names
- Function parameters
- Class names

The term **variable name** includes all of these identifiers except property names of objects. However, property names of the **global object** are usually included when discussing variable names.

JS has a bunch of other things that involve storing data in a named area of memory. The list looks a lot like the list of identifiers:

- Variables declared with **let** and **var**
- Constants declared with **const**
- Properties of the global object
- Function names
- Function parameters
- Class names

The most significant difference in this list compared to the list of identifiers is that not all object properties are variables; only those on the global object.

JS is also unusual in that we can think of function and class names as being variable names: in fact, they are. Functions and classes are actually values in JS, and their names are used in the same way as more traditional variables.

Declaring and Assigning Variables

A variable declaration is a statement that asks the JS engine to reserve space for a variable with a particular name. Optionally, it also specifies an initial value for the variable (i.e., the variable is **initialized** with a value).

There is a subtle difference in terminology surrounding the **=** token. When used in a declaration, the **=** is just a syntactic token that tells JS that you're going to supply an initial value for the variable. However, in an assignment, the **=** is called the **assignment operator**.

Declaring Constants

The **const** keyword is similar to **let**, but it lets you declare and initialize **constant** identifiers.

Constants have an immutable binding to their values. Unlike an ordinary variable, once you declare a constant, you cannot assign it a new value. The constant will continue to have that value until the constant is no longer needed.

Using constants is a great way to label a value with a name that makes your code more descriptive and easier to understand.

A standard convention when naming constants is to use all uppercase letters and digits in the name; if the name contains multiple words, use underscores to separate the words.

Variable Scope

A variable's **scope** determines where it is available in a program. The location where you declare a variable determines its scope.

In JS, variables declared with the **let** or **const** keywords have **block** scope. A block is a related set of JS statements and expressions between a pair of opening and closing curly braces.

In general, blocks appear in **if … else**, **while**, **do … while**, **for**, **switch**, and **try … catch** statements, or by themselves.

Function bodies are not technically blocks. However, they look and behave so much like blocks that many developers do not distinguish between them.

Where you define the variable determines the scope in which you can use and modify it.

Input/Output

Computers and software don't live in isolation. They are tools that solve-real world problems, which means that they must interact with the real world. A computer needs to take input from some source, perform some actions on that input, and then provide output.

Command Line Input

Node.js has an API called **readline** that lets JS programs read input from the command line. However, the API isn't straightforward or simple: it requires an understanding of asynchronous programming and higher-order functions. For now, we can use a simplified version of the readline library called **readline-sync**.

Example using this library:

- let rlSync = require('readline-sync');
- let name = rlSync.question("What's your name?\n");
- console.log(`Good Morning, ${name}!);

Line one uses Node's built-in **require** function to import **readline-sync** into your program. It returns the library as an object, which we assign to the **rlSync** variable.

In line two, we use **rlSync** to call the **question** method. This method displays its string argument, then waits for the user to respond. When the user types some text and presses Return, it returns that text to the program. Here, we assign that text to the variable **name** and use it to display a personalized greeting.

Input in the Browser

Browsers provide an environment that differs radically from that of Node.js: browser users must interact with JS programs in an entirely different way. Working with a browser's input controls requires a working knowledge of the Document Object Model (DOM).

However, you don't need to know about the DOM to get user inputs. Most browsers implement the **prompt** function which lets a program ask for and obtain text-based input from the user.

HTML filenames typically end with **.html**, and the **src** attribute in the **<script>** tag identifies the name of a JS file. When your browser encounters this tag, it loads the JS file and executes the code that it contains.

The **prompt** function works much like **rlSync.question**: it waits for the user to input some text and click the 'OK' button, and then returns the user's input as a string.

Functions

Most languages have a feature called procedures that let you extract the code and run it as a separate unit. In JS, we call these procedures **functions**.

Using Functions

Here's a function named **say**:

```
function say() {
        console.log("Hi!");
}
```

Functions are called by typing their name and providing some optional values that we call **arguments**. Arguments let you pass data from outside the function's scope into the function so it can access the data.

In the definition of a function, the names between parentheses are called **parameters**. The arguments are the values of those parameters. The parameter values inside the function are also called arguments.

You can think of parameters as placeholders, while arguments refer to the values that get stored in the placeholders.

Function names and parameters are both considered variable names in JS. Parameters, in particular, are **local variables**; they are only defined locally, within the body of the function.

Return Values

One common use case is to perform an operation and **return** a result to the call location for later use. We do that with **return values** and the **return** statement.

All JS function calls evaluates to a value. By default, that value is **undefined**; this is the **implicit return value** of most JS functions. However, when you use a **return** statement, you can return a specific value from a function. This is an **explicit return value**.

Outside of the function, there is no distinction between implicit and explicit return values, but it's important to remember that all functions return something unless they raise an exception, even if they don't execute a **return** statement.

When JS encounters the **return** statement, it evaluates the expression, terminates the function, and returns the expression's value to the location where we called it.

Functions that always return a boolean value (i.e., **true** or **false**) are called **predicates**.

Default Parameters

When you define a function, you sometimes want to structure it so that you can call it without an argument (i.e., we provide a default value to the function).

```
function say(words = "Hello") {
        console.log(words + "!");
}
```

Nested Functions

You can create functions anywhere, even nested inside another function. Such nested functions get created and destroyed every time the outer function runs.

Note that this has a mostly negligible effect on performance.

They are also private functions since we can't access a nested function from outside the function where it is defined.

Function & Scope

In JS, there are two types of variables based on where they're accessible: **global** variables and **local** variables. Global variables are available throughout a program, while local variables are confined to a function or a block.

We can use global variables within functions and we can even reassign global variables from inside a function.

Global variables can be useful in some scenarios (i.e., application-wide configuration). However, most developers discourage their use since they often lead to bugs. In general, you should limit the scope of your variables as much as possible; smaller variable scopes limit the risk that an outer scope might misuse the variable.

Parameters are local variables. We initialize it from the argument passed to the function. Parameters have a local scope within the function.

Local variables are short-lived; they go away when the function that corresponds to their scope stops running.

When we invoke the function, we start a new scope. If the code within that scope declares a new variable, that variable belongs to the scope. When the last bit of code in that scope finishes running, the local variables in that scope go away. JS repeats this process each time we invoke a function.

Functions vs. Methods

Thus far, we call a function by writing parentheses after its name and passing it zero or more arguments.

Method invocation occurs when you prepend a variable name or value followed by a period (**.**) to a function invocation such as **'xyzzy'.toUpperCase()**. We call such functions **methods**.

Mutating the Caller

Sometimes a method permanently alters the object that invokes the method: it **mutates the caller**.

Non-mutating methods like **toUpperCase()** often return a new value or object, but leave the caller unchanged.

The **pop()** method removes the last element from an array, but it does so destructively: the change is permanent.

We can also talk about whether functions mutate their arguments.

The **concat** method returns a new array that contains a copy of the original array combined with the additional elements that we supply with the arguments.

One non-obvious point here is that mutation is a concern when dealing with arrays and objects, but not with primitive values like numbers, strings, and booleans. Primitive values are **immutable**. That means their values never change: operations on immutable values always return new values.

Operations on **mutable** values (arrays and objects) may or may not return a new value and may or may not mutate data.

JS uses pass-by-value when dealing with primitive values and pass-by-reference with objects and arrays.

Function Composition

In a process called **function composition**, JS lets us use a function call as an argument to another function.

Three Ways to Define a Function

In JS, a **function declaration** looks like:

```
function functionName(zeroOrMoreArguments …) {
        // function body
}
```

A notable property of function declarations is that you can call the function before you declare it.

In JS, a **function expression** looks like:

```
let functionName = function(zeroOrMoreArguments …) {
        // function body
}
```

Function expressions have one key difference from a function declaration:

- You cannot invoke a function expression before it appears in your program.

JS functions are **first-class functions**. The key feature of first-class functions is that you can treat them like any other value. In fact, **all JS functions are objects**. Thus, you can assign them to variables, pass them as arguments to other functions, and return them from a function call.

Any function definition that doesn't have the word **function** at the very beginning of a statement is a function expression.

In JS, an **arrow function** looks like:

let greetPeople = () => console.log("Good Morning!");

Arrow functions are similar to function expressions, but they use a different syntax. The differences are not merely syntactic, however.

An interesting property of arrow functions is **implicit returns**. We can omit the **return** statement in arrow functions when and only when the function body contains a single expression. The expression may have subexpressions, but the entire expression must evaluate to a single value.

Suppose it contains two or more expressions or statements. In that case, you must explicitly return a value if you need it, and you must also use curly braces.

The Call Stack

One important aspect of function that all programmers need to understand is the concept of the **call stack**, or more casually, the **stack**.

The call stack helps JS keep track of what function is executing as well as where execution should resume when the function returns.

To do that, it works like a stack of books: if you have a stack of books, you can put a new book on the top or remove the topmost book from the stack. In much the same way, the call stack puts information about the current function on the top of the stack, then removes that information when the function returns.

The call stack initially has one item -- called a **stack frame** -- that represents the global (top-level) portion of the program. The initial stack frame is sometimes called the **main** function. JS uses this frame to keep track of what part of the main program it is currently working on.

The call stack has a limited size that varies based on the JS implementation. That size is usually sufficient for more than 10,000 stack entries. If the stack runs out of room, you will see a **RangeError** exception together with a message that mentions the stack.

A call stack can be seen as a "todo list" of function invocations.

Flow Control

When writing programs, you want your data to take the correct path. You want it to turn left or right, up, down, reverse, or proceed straight ahead when it's supposed to. We call this **flow control**.

Conditionals

A conditional is a fork (or multiple forks) in the road. The simplest conditionals use a combination of **if** statements with comparison and logical operators (**<**, **>**, **<=**, **>=**, **==**, **===**, **!=**, **!==**, **&&**, **||**).

You don't need a block when the **if** or **else** clause contains a single statement or expression. You need braces for a block when you want to execute multiple statements or expressions in a clause.

Comparisons

One thing to remember is that comparison operators return a boolean value: **true** or **false**.

The expressions or values that an operator uses are its **operands**.

The **strict equality operator** (**===**), also known as the **identity operator**, returns **true** when the operands have the same type and value, **false** otherwise.

The **strict inequality operator** (**!==**) returns **false** when the operands have the same type and value, **true** otherwise. Note that **!==** is the inverse of **===**.

The **non-strict equality operator** (**==**), also known as the **loose equality operator**, is similar to **===**. However, when the operands have different types, **==** attempts to coerce one of the operands to the other operand's type before it compares them, and it may coerce both operands in some cases.

The **non-strict inequality operator** (**!=**), also known as the **loose inequality operator**, is similar to **!==**. However, when the operands have different types, **!=** attempts to coerce one of the operands to the other operand's type before it compares them, and it may coerce both operands in some cases.

The **less than operator** (**<**) returns **true** when the value of the left operand has a value that is less than the value of the right operand, **false** otherwise.

The **greater than operator** (**>**) returns **true** when the value of the left operand has a value that is greater than the value of the right operand, **false** otherwise.

The **less than or equal to operator** (**<=**) returns **true** when the value of the left operand has a value that is less than or equal to the value of the right operand, **false** otherwise.

The **greater than or equal to operator** (**>=**) returns **true** when the value of the left operand has a value that is greater than or equal to the value of the right operand, **false** otherwise.

Logical Operators

The **not operator** (**!**) returns **true** when its operand is **false** and returns **false** when the operand is **true**. That is, it negates its operand.

The **and operator** (**&&**) returns **true** when both operands are **true** and **false** when either operand is **false**.

The **or operator** (**||**) returns **true** when either operand is **true** and **false** when both operands are **false**.

&& and **||** don't always return **true** or **false**, but they do when they operate on boolean values.

Short Circuits

The **&&** and **||** operators both use a mechanism called **short circuit evaluation** to evaluate their operands.

For example:

```
isRed(item) && isPortable(item)
isGreen(item) || hasWheels(item)
```

If the program determines that **item** is not red, it doesn't have to check whether it is portable. JS short-circuits the entire expression by terminating evaluation as soon as it determines that **item** isn't red.

Similarly, if the program determines that **item** is green, it doesn't have to check whether it has wheels. Again, JS short-circuits the entire expression once it determines that **item** is green.

Truthiness

JS can coerce any value to a boolean value, and that's what it does in conditional contexts like the **if** statement.

Thus, you can use any expression in a conditional expression. We often say that the expression **evaluates as** or **evaluates to** true or false.

When coercing a value to a boolean, JS treats the following value as false:

- **false**
- The number **0**. This includes all 3 variations of zero in JS:
    - **0**: The ordinary zero value.
    - **-0**: A negative zero. That's mathematical nonsense, but a real thing in JS.
    - **0n**: The **BigInt** version of zero.

- An empty string (‘ ’)
- **undefined**
- **null**
- **NaN**

Everything else evaluates as true.

We often use the term **falsy** to refer to values that evaluate as false, while the values that evaluate as true are **truthy**. We can also discuss **truthiness**: whether something is a truthy or falsy value.

The **&&** and **||** logical operators, as you’ll recall, use short-circuit evaluation. These operators work with truthy and falsy values too, and they can also return truthy values instead of boolean values. When using **&&** and **||**, the return value is always the value of the operand evaluated last.

Ternary expression:

let myVariable = (firstCriteria || secondCriteria) ? valueIfTrue : valueIfFalse;

Operator Precedence

JS has a set of **precedence** rules it uses to evaluate expressions that use multiple operators and sub-expressions. The following is a list of the comparison operations from the highest precedence (top) to lowest (bottom):

- **<=**, **<**, **>**, **>=** - Comparison
- **===**, **!==**, **==**, **!=** - Equality
- **&&** - Logical And
- **||** - Logical Or

We can use parentheses to override the precedence: sub-expressions in parentheses get evaluated before un-parenthesized expressions at the same depth in the main expression.

You should strive to use parentheses in any expression that uses two or more different operators.

The Ternary Operator

The **ternary operator** is a quick and easy way to write a short, concise, and simple if/else conditional. It uses a combination of the **?** and **:** symbols and takes 3 operands (hence, the name “ternary”):

1 == 1 ? ‘this is true’ : ‘this is not true’

The chief advantage that the ternary operator has over an **if/else** statement is that the entire structure is an expression. What that means is that we can treat the ternary expression as a value: we can assign it to a variable, pass it as an argument, and so on. Since **if/else** is a statement, we can’t capture its result to a variable.

Switch Statement

A **switch** statement is similar to an **if** statement, but it has a different interface. It compares a single value against multiple values for strict equality, whereas **if** can test multiple expressions with any condition.

Switch statements use the reserved words **switch**, **case**, **default**, and **break**.

For example:

```
let a = 5;

switch (a) {
        case 5:
                console.log(‘a is 5’);
                break;
        case 6:
                console.log(‘a is 6’);
                break;
```

```
    default:
        console.log('a is neither 5, nor 6');
        break;
}
```

The **switch** statement evaluates the expression, **a**, compares its value to the value in each **case** clause and then executes the statements and expressions associated with the first matching clause.

The statements and expressions in the **default:** clause run when the expression doesn't match any of the **case** clauses; it acts like the final **else** in an **if** statement.

The **break** statement in each **case** is crucial. Without a break, execution "falls through" to the next **case** clause.

We can apply the same code to multiple cases with a **switch** statement.

```
↳ case a:
  case b:
      // the block of code
      break;
```

Loops and Iterating

JS loops have several forms, but the main looping structures use a looping keyword, a condition, and a block.

While Loops

A **while** loop uses the **while** keyword followed by a conditional expression in parentheses and a block. The loop executes the block again and again for as long as the conditional expression remains truthy.

For example:

```
let counter = 1;
while (counter <= 10) {
    console.log(counter);
    counter = counter + 1;
}
```

The **increment operator** (**++**) increments its operand by 1; that is, it adds 1 to the existing value (i.e., **counter++**). There's a corresponding **decrement operator** (**--**) that decrements a variable's value by 1. That is, it subtracts 1 from the value.

There are two forms of **++**: one that comes before the variable name (the pre-increment operator), and one that comes after (the post-increment operator). Both increment the variable, but they differ in what gets returned by the expression. The pre-increment form returns the new value of the variable, while the post-increment form returns the previous value of the variable.

There are corresponding pre-decrement and post-decrement operators (i.e., **--a** and **a--**) that work in a similar way.

One of the most common uses of loops in programming is to iterate over an array's elements and perform some action on each element. By **iterate**, we mean that we process each element one at a time, in sequence from the first to the last element.

A **do/while loop** differs visibly from a **while** loop, but it's behaviour is almost identical. The crucial difference is that **do/while** always executes the code in the block at least once. In a **do/while loop**, the conditional check occurs at the end of the loop instead of the beginning which allows it to run the code at least once, even if the condition is falsy when the loop begins.

For example:

```
let answer;
do {
        answer = prompt("Do you want to do that again?");
} while (answer === 'y');
```

For Loops

For loops have the same purpose as **while** loops, but they use a condensed syntax that works well when iterating over arrays and other sequences. A **for** loop combines variable initialization, a loop condition, and the variable increment/ decrement expression all on the same line.

For example:

```
for (initialization; condition; increment) {
        // loop body
}
```

The sole difference between the **for** and **while** loops is the scope of any variables declared by the initialization clause. In the **while** statement, the scope includes the code that surrounds the loop; in the **for** statement, the scope is the **for** statement and its body.

for loops let you see and understand the looping logic at a single glance. The syntax also lets you move the **index** variable from the global scope into the scope of the **for** statement, and it helps make your code cleaner and more organized.

All 3 components of the **for** loop are optional.

Controlling Loops

JS uses the keywords **continue** and **break** to provide more control over loops.

continue lets you start a new iteration of the loop, while **break** lets you terminate a loop early.

When a loop encounters the **continue** keyword, it skips running the rest of the block and jumps ahead to the next iteration.

Without **continue**, your loops get cluttered with nested conditional logic.

Earlier, we said that you should always use blocks with **if** statements. A common exception to this rule occurs when using a **continue**, **break**, or **return** statement as the **if** clause. When changing the flow with these three statements, the single-line version of the **if** statement can make your code easier to read.

Array Iteration

JS arrays have several methods that iterate over the elements without using the looping syntax we've seen thus far.

Arrays are ordered lists.

One feature of JS that sets it apart from most other languages is that it has first-class functions. That means that functions are values: you can assign them to variables, pass them around as arguments to other functions, and even use them as return values in other functions.

When you pass a function as an argument to another function, that other function can call the function represented by the argument.

Most JS programmers prefer to use **array looping abstractions** like **forEach** to loop over arrays.

Recursion

Recursive functions are functions that call themselves. We say that recursion is another way to create loops in JS.

Every recursive function has a **baseline condition** that marks the end of the recursion and some code that recursively calls the function with a new argument. In most cases, the baseline condition returns a concrete value that gets reused as the code "unwinds" the recursive calls. Each unwind step uses the previous return value(s) to calculate an intermediate result that gets returned in the next step.

Arrays

An array is an ordered list of **elements**; each element has a value of any type. You can define an array by placing a list of values between brackets (**[ ]**).

Arrays are **heterogeneous**. Arrays can have anything in them, including objects and even other arrays.

Each element in an array has a unique index number that gives the position of the element in the array. Thus, we can say that arrays are **indexed lists** as well as ordered lists.

Modifying Arrays

To replace an element of an array, use brackets with the assignment operator (i.e., **array[0] = newValue**).

You can also use brackets to add new elements to an array.

Note that variables declared with **const** and initialized to an array are a little strange; while you can't change what array the variable refers to, you can modify the array's contents.

This type of behaviour stems from the "variables as pointers" concept. Essentially, a **const** declaration prohibits changing what thing the **const** points to, but it does not prohibit changing the content of that thing. Thus, we can change an element in a **const** array, but we can't change which array the **const** points to.

If you want the elements of the array to also be **const**, you can use the **Object.freeze** method such as:

const MyArray = Object.freeze([1, 2, 3])

It's important to realize that **Object.freeze** only works one level deep in the array. If your array contains nested arrays or other objects, the values inside them can still be changed unless they are also frozen.

The **push** method adds one or more elements to the end of an array. The **push** method appends its arguments to the caller (the array), which mutates the caller. It then returns the array's new length.

Don't forget that methods and functions perform actions and return values. You must be careful to distinguish between these two things.

The **concat** method is similar to **push**, but it doesn't mutate the caller. It concatenates two arrays and returns a brand new array that contains all the elements from the original array followed by all of the arguments passed to it.

The inverse of **push** is **pop**. While **push** adds an element to the end of the array, **pop** removes and returns the last element of the array. **pop** mutates the caller.

The **splice** method lets you remove one or more elements from an array, even those that aren't at the end of the array (i.e., **array.splice(3, 2)**).

we remove two elements starting at index position three

Iteration Methods

JS has a variety of built-in methods that iterate over the contents of an array.

To use **forEach**, you need a **callback** function that you pass to **forEach** as an argument. A callback function is a function that you pass to another function as an argument. The called function invokes the callback function when it runs. The **forEach** method invokes its callback once for each element, passing it the element's value as an argument.

forEach always returns undefined.

forEach works well when you want to use the values of an array's elements.

Suppose, though, that you want to create a new array whose values depend on the original contents of the array. The **map** method handles this situation more cleanly.

map returns a new array that contains one element for each element in the original array, with each element set to the return value of the callback.

The main thing to remember is that **forEach** performs simple iteration and returns **undefined**, while **map** transforms an array's elements and returns a new array with the transformed values.

The **filter** method is another array iteration method. It returns a new array that includes all elements from the calling array for which the callback returns a truthy value.

filter iterates over the elements of the array. During each iteration, it invokes the callback function, using the value of the current element as an argument. If the callback returns a truthy value, **filter** appends the element's value to a new array. Otherwise, it ignores the element's value and does nothing. When **filter** finishes iterating, it returns the array of selected elements: the elements for which the callback returned a truthy value.

filter doesn't mutate the caller.

The **reduce** method effectively reduces the contents of an array to a single value. It is, perhaps, one of the hardest array iterations methods to understand, but it is also one of the most fundamental.

You can build **forEach**, **map**, and **filter** with **reduce**, as well as a number of other iterative methods defined for Arrays.

reduce takes two arguments: a callback function and a value that initializes something called the **accumulator**. In its simplest form, the callback function takes two arguments: the current value of the accumulator and an element from the array. It returns a value that will be used as the accumulator in the next invocation of the callback.

For example:

```
let arr = [2, 3, 5, 7]
arr.reduce((accumulator, element) => accumulator + element, 0)
```

The **reduce** function isn't limited to computing numbers: you can also use it to compute strings, objects, arrays: anything.

reduce does not mutate the caller. It is possible that the callback might mutate the caller, but that's inadvisable, and not **reduce**'s fault.

Arrays Can Be Odd

Arrays are objects, one side effect of this is that the **typeof** operator doesn't return **'array'** when applied to an array. It returns **'object'**.

If you change an array's **length** property to a new, smaller value, the array gets truncated; JS removes all elements beyond the new final element.

If you change an array's **length** property to a new, larger value, the array expands to the new size. The new elements **do not get initialized**, which leads to some strange behaviour.

In general, JS treats unset array elements as missing, but the **length** property includes the unset elements.

You can create array “elements” with indexes that use negative or non-integer values, or even non-numeric values.

Since arrays are objects, you can use the **Object.keys** method to return an array’s keys -- its index values -- as an array of strings. Even negative, non-integer, and non-numeric indexes are included.

One quirk of this method is that it treats unset values in arrays differently from those that merely have a value of **undefined**. Unset values are created when there are “gaps” in the array; they show up as empty items until you try to use their value.

While the **length** property of Array includes unset values in the count, **Object.keys** only counts those values that have been set to some value.

Nested Arrays

Array elements can contain anything, including other arrays.

Array Equality

JS treats two arrays as equal only when they are the same array: they must occupy the same spot in memory. This rule holds for JS objects in general; objects must be the same object.

To check whether two arrays have the same elements, one option is to create a function that compares the elements of one array with the corresponding elements in the other.

Other Array Methods

The **includes** method determines whether an array includes a given element. Internally, **includes** uses **===** to compare elements of the array with the argument. That means we can’t use **includes** to check for the existence of a nested array or an object unless we have the same object or array we’re looking for.

The **sort** method is a handy way to rearrange the elements of an array in sequence. It returns a sorted array. This method mutates the caller.

The **slice** method extracts and returns a portion of the array. It takes two optional arguments. The first is the index at which extraction begins, while the second is where extractions ends.

If you omit the second argument, **slice** returns the rest of the array starting with the index given by the first argument. With the second argument, it returns the elements up to but excluding that index. If you don’t provide any arguments at all, **slice** returns a copy of the entire array: that is, returns a new array with the same elements as the original. That’s useful when you need to use a destructive method on an array that you don’t want to modify.

The **reverse** method reverses the order of an array. It is destructive: it mutates the array.

Objects

Object Oriented Programming is a programming paradigm that centers around modelling problems as **objects** that have **behaviour** (they perform actions) and **state** (they have characteristics that distinguish between different objects).

What are Objects

Objects store a collection of **key-value pairs**: each item in the collection has a name that we call the **key** and an associated **value**.

An object’s keys are strings, but the values can be any type, including other objects. We can create an object using **object literal** syntax:

```
let person = {
        name: ‘Jane’,
        age: 37,
        hobbies: [‘photography’, ‘genealogy’],
};
```

Though the keys are strings, we typically omit the quotes when the key consists entirely of alphanumeric characters and underscores.

We can access a specific value in an object in two ways: 1) dot notation and 2) bracket notation. For example:

person.name

or

person['age']

With dot notation, we place a dot and a key name after the variable that references the object. With bracket notation, we write the key as a quoted string and put it inside square brackets. Most developers prefer dot notation when they can use it. However, if you have a variable that contains a key's name, you must use bracket notation.

If you want to remove something from an existing object, you can use the **delete** keyword (i.e., **delete person.age**).

delete removes the key-value pair from the object and returns **true** unless it cannot delete the property (for instance, if the property is non-configurable).

Key-value pairs are also called object **properties** in JS. We can also use "property" to refer to the key name.

If a variable declared with **const** is initialized with an object, you can't change what object that variable refers to. You can, however, modify that object's properties and property values.

Essentially, a **const** declaration prohibits changing what thing the **const** points to, but it does not prohibit changing the content of that thing. Thus, we can change a property in a **const** object, but we can't change which object the **const** points to.

You can use **Object.freeze** with objects to freeze the property values of an object, just like you can with arrays.

As with arrays, **Object.freeze** only works one level deep in the object. If your object contains nested arrays or other objects, the values inside them can still be changed unless they are also frozen.

Objects vs. Primitives

Objects include, but aren't limited to, the following types:

- Simple Objects
- Arrays
- Dates
- Functions

Objects are complex values composed of primitive values or other objects. For example, an array object has a **length** property that contains a number: a primitive value. Objects are usually (but not always) mutable: you can add, remove, and change their various component values.

Primitive values are always immutable; they don't have parts that one can change. Such values are said to be **atomic**, they're indivisible. If a variable contains a primitive value, all you can do to that variable is use it in an expression or reassign it: give it an entirely new value. All operations on primitive values evaluate as new values.

Objects and primitive values are the data and functions that you can use in your program. Anything that isn't data or a function is neither a primitive value nor an object. That includes:

- variables and other identifiers such as function names
- statements such as **if**, **return**, **try**, **while**, and **break**
- keywords such as **new**, **function**, **let**, **const**, and **class**
- comments
- anything else that is neither data nor a function

Prototypes

An interesting and handy feature of JS objects is that they can **inherit** from other objects. When an object **a** inherits from object **b**, we say that **b** is the prototype of **a**. The practical implication is that **a** now has access to properties defined on **b** even though it doesn't define those properties itself.

The static method **Object.create** provides a simple way to create a new object that inherits from an existing object. **Object.create** creates a new object and sets the prototype for that object to the object passed in as an argument.

Iterations

The **for/in** loop behaves similarly to an ordinary **for** loop. The syntax and semantics are easier to understand since you don't need an initialization, ending condition, or increment clause. Instead, the loop iterates over all the keys in the object. In each iteration, it assigns the key to a variable which you can then use to access the object's values.

For example:

```
for (let prop in person) {
        console.log(person[prop]);
}
```

One feature — or downside, depending on how you look at it — of **for/in** is that it iterates over the properties of an object's prototypes as well. This behaviour is undesirable when you want to limit iteration to an object's **own properties** (i.e., properties it defined for itself, not properties it inherited).

We can use the **hasOwnProperty** method to get around that problem. It takes the name of a property and returns **true** if it is the name of one of the calling object's own properties, **false** if it is not (i.e., **object.hasOwnProperty(key)**).

The **Object.keys** static method returns an object's keys as an array. You can iterate over that array using any technique that works for arrays. It does not include any keys from the prototype objects.

For the most part, you can assume that the iteration order is the order in which the keys get added to the object. However, be careful: if you have any symbol keys or keys that look like non-negative integers ('0', '1', '2', etc.), JS will group the non-negative integers first followed by other-string valued keys, and, finally, the symbolic keys. For clarity, you should only rely on the iteration order when you know that all of the keys will be alphabetic.

Common Operations

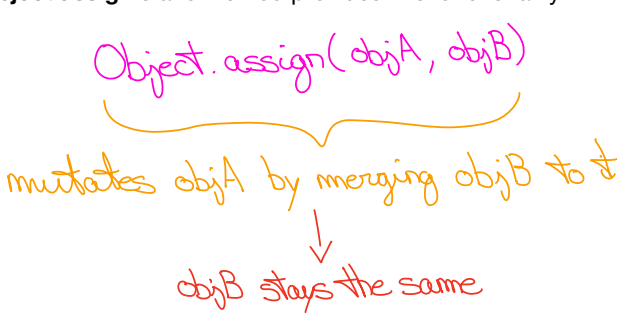
Unlike arrays, most JS objects don't have an abundance of methods that you can apply in your day to day usage.

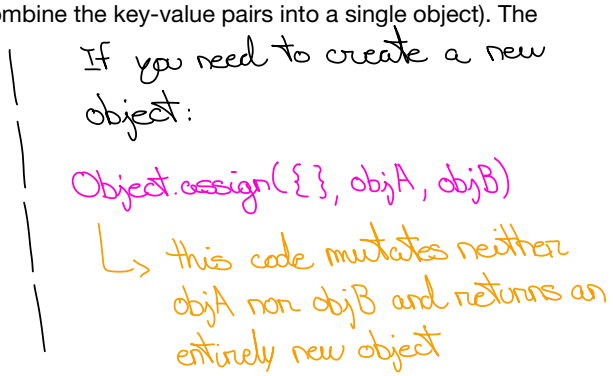
Most operations on objects involve iterating over the properties or their values. More often than not, you'll reach for methods that extract the keys or values of an object and then iterate over the resulting array.

Object.values is a static method that extracts the values from every own property in an object to an array. Be careful, remember that you can't predict the order of the values in the returned array.

Object.entries is a static method that returns an array of nested arrays. Each nested array has two elements: one of the object's keys and its corresponding value.

You may sometimes want to merge two or more objects (i.e., combine the key-value pairs into a single object). The **Object.assign** static method provides this functionality.





Objects vs. Arrays

When you need to choose between an object or an array to store some data, ask yourself a few questions:

- Do the individual values have names or labels? If yes, use an object. If the data doesn't have a natural label, an array should suffice.
- Does order matter? If yes, use an array.
- Do I need a stack or queue structure? Arrays are good at mimicking simple "last-in-first-out" stacks and "first-in-first-out" queues.

More Stuff

Variables as Pointers

Some variables act as pointers to a place (i.e., or address space) in memory, while others contain values.

Developers sometimes talk about **references** instead of pointers. You can say that a variable points to or references an object in memory, and you can also say that the pointers stored in variables are references.

Every time a JS program creates a new variable, JS allocates a spot somewhere in its memory to hold its value. With (most) primitive values, the actual value of the variable gets stored in this allocated memory. JS discards these variables and their values when it no longer needs them.

Each variable has a value, and reassigning values does not affect any other variables that happen to have the same value.

Variables that have primitive values store those values at the memory location associated with the variable.

In reality, string values aren't stored in variables in the same way as most primitive values, but they **act** like they are.

With objects, JS doesn't store the value of the object in the same place. Instead, it allocates additional memory for the object, and places a pointer to the object in the space allocated for the variable. Thus, we need to follow two pointers to get the value of our object from its variable.

While the pointer to the object can change, the object itself always has the same address.

The use of pointers has a curious effect when you assign a variable that references an object to another variable. Instead of copying the object, JS only copies the pointer. Some developers call this aliasing.

All primitive values are immutable. Two variables can have the same primitive value. However, since primitive values are stored in the memory address allocated for the variable, they can never be aliases. If you give one variable a new primitive value, it doesn't affect the other.

Reassignment applies to the item you're replacing, not the object or array that contains that item.

The takeaway of this section is that JS stores primitive values in variables, but it uses pointers for non-primitive values like arrays and other objects. Pointers can lead to surprising and unexpected behaviour when two or more variables reference the same object in the heap.

When using pointers, it's also important to keep in mind that some operations mutate objects, while others don't.

for/in and for/of

Two useful variants for the **for** loop are the **for/in** and **for/of** loops. These loops use a variant syntax to loop easily over object properties.

The **for/in** statement iterates over all enumerable properties of an object including any properties inherited from another object. Using this style of loop on an array, we will be iterating over the index values.

Using the **for/of** loop on an array allows us to iterate over the elements.

for/of is similar to **for/in**, but it iterates over the values of any “iterable” collection.

Method Chaining

The main takeaway is that you can call a method on the return value of another method. It’s not much different from function composition, but it uses a simpler syntax.

For example:

```
let str = ‘Pete Hanson’;
let names = str.toUpperCase().split(‘ ’).reverse().join(‘, ’);
console.log(names); // => HANSON, PETE
```

Regex

A **regular expression** — a **regex** — is a sequence of characters that you can use to test whether a string matches a given pattern. They have a multitude of uses:

- Check whether the string ‘**Mississippi**’ contains the substring **ss**.
- Print the 3rd word of each sentence from a list of sequences.
- Replace all instances of **Mrs** in some text with **Ms**.
- Does a string begin with the substring **St**?
- Does a string end with the substring **art**?
- Does a string contain any non-alphanumeric characters?
- Does a string contain any whitespace characters?
- Replace all non-alphanumeric characters in a string with a hyphen.

A regex looks like a string written between a pair of forward-slash characters instead of quotes (i.e., **/bob/**). You can place any string you want to match between the slashes, but certain characters have special meanings.

JS uses RegExp objects to store regex: note the spelling and case. Like other object, RegExp objects can invoke methods.

The method **test**, for instance, returns a boolean value based on whether a string argument matches the regex. For example, **/o/.test(‘bobcat’)** will return **true**.

Boolean values sometimes don’t provide enough information about a match. That’s when the **match** method for strings comes in handy. This method takes a regex as the argument and returns an array that describes the match. For example, **“bobcat”.match(/x/)** or **“bobcat”.match(/[bct]/g)** are some of the use cases.

If no match occurs, **match** returns the value **null**, which conveniently lets us use **match** in conditionals in the same way as **test**.

When a match occurs with a regex that contains the **/g** flag - a global match - the **match** method returns an array that contains each matching substring.

When **/g** isn’t present, the return value for a successful match is also an array, but it includes some additional properties:

- **index**: the index within the string where the match begins
- **input**: a copy of the original string
- **groups**: used for “named groups”

The array elements are also a bit different when **/g** isn’t present. In particular, the first element represents the entire matched part of the string. Additional elements represent capture group matches. Parentheses inside a regex define capture groups (i.e., **“bobcat”.match(/b((o)b)/)**).

Since **match** must generate information above and beyond a simple boolean value, it can have performance and memory costs. **test** is more efficient, so try to use it when you don’t need to know anything other than whether the regex matched the string.

Mixing **/g** and **test** may lead to surprising results. Keep in mind whether you need all matches or just a single match - if you need a single match, **/g** is inappropriate.

A regex can, in a single-line, solve problems that may require dozens of lines using other techniques. If you encounter a string matching problem that needs more than a simple substring search using the **indexOf** or **includes** method, remember to look into using regex.

The Math Object

The JS **Math** object provides a collection of methods and values that you can use without a complete understanding of how they work.

Dates

JS's **Date** constructor creates objects that represent a time and date. The objects provide methods that let you work with those values.

getDay returns a number for the day of the week: **0** represents Sunday, **1** represents Monday, and so on.

Exceptions

Not all errors in JS are silent. There are some situations where JS is less forgiving; that's where **exceptions** come into play. In such cases, JS **raises an error**, or **throws an exception**, then halts the program if the program does not **catch** the expression.

Exception handling is a process that deals with errors in a manageable and predictable manner. The reserved words **try** and **catch** (and sometimes **finally**) often occur in real-world JS programs.

JS's **try/catch** statement provides the means to handle exceptions. The basic structure looks like this:

```
try {
        // perform an operation that may produce an error
} catch (exception) {
        // an error occurred, do something about it
        // for example, log the error
} finally {
        // optional finally block; not used often
        // executes regardless of whether an exception occurs
}
```

Execution ultimately resumes with the code after the **try/catch** statement.

Don't try to catch every possible exception. If you can't do anything useful with the exception, let it go. Mishandling an exception is usually far more catastrophic than just letting the program fail.

The **throw** keyword raises an exception of the type specified as an argument, which is usually given as **new** followed by one of the error types such as **throw new TypeError("expected a number")**.

Exceptions are for **exceptional circumstances**: situations that your program can't control very easily, such as not being able to connect to a remote site in a web application.

For now, all you need to understand is that you **can** anticipate and handle errors that may occur in your program; a single unexpected input or other issue doesn't have to crash your entire application or introduce subtle bugs.

A special kind of exception occurs if the code can't be handled as valid JS. Such errors cause JS to raise a **SyntaxError**. A **SyntaxError** is special in that it occurs immediately after loading a JS program, but before it begins to run.

Three major takeaways:

- A **SyntaxError** usually has nothing to do with the values of any of your variables. You can almost always spot the error visually.

- A **SyntaxError** can occur long after the point where the error was. There can be many hundreds of lines between the point where the error is and the point where JS detects it.

- The code before and after the error does not run. That's because **SyntaxError**s are detected before a program begins running. This also shows that there are at least two phases in the life of a program — a preliminary phase that checks for syntax errors, and an execution phase.

There are some situations where JS can throw a **SyntaxError** after a program begins running.

Stack Traces

A **stack trace** reports the type of error that occurred, where it occurred, and how it got there.

In most cases, you can limit your attention to the lines that mention your JS code file(s) by name. Each filename in the trace includes a location specified as a line number and column number. The filename, line number, and column number together pinpoint the specific location where the failure occurred and how the program reached that point. Take note of the locations that pertain to your code.

The stack trace is a readable version of the call stack's content at the point an exception occurred.

Random Things:

Object.assign({}, copiedObject)

↳ creates a shallow copy of copiedObject

↳ it only copies pointers for nested objects

a shallow copy returns a new object that is a copy of the original object

however, only the object itself and any properties with primitive values are duplicated

properties that are themselves objects are copied "by reference" -- that is, only a pointer to the object is copied

Code appearing to the right of an = in a declaration is an expression

Even if a string isn't a number, JS coerces it to a number when a string and a number are mixed with ==

It doesn't matter which side of the == contains the string operand

If the string contains a non-numeric value, JS coerces it to NaN