# Introduction to Context

- Due No Due Date
- Points 1
- Submitting a website url

(https://github.com/learn-co-curriculum/phase-1-intro-to-context) (https://github.com/learn-co-curriculum/phase-1-intro-to-context/issues/new)

## Learning Goals

- Introduce execution context
- Define the term "record"
- Define the term "record-oriented programming"

## Introduction

Let's take a moment to appreciate where we are. We've reviewed the basic use and creation of functions. We've applied these skills in the context of collection-processing methods like `map` , `reduce` , and `forEach` . We're now ready to face one of the (infamously) most-challenging parts of JavaScript: working with execution context.

## Introducing Execution Context

When JavaScript code runs, the JavaScript engine sets aside a place in memory to store references to the variables and functions that are defined at the global level. This is referred to as the *global execution context*. In addition, any time a function is invoked, a separate space in memory known as the *function execution context* is created, with references to any variables or functions defined within that function.

Logically enough, when code is executed at the global level, the *execution context* is the global execution context. Similarly, when a function is invoked, the *execution context* is the execution context defined for that function. However, in addition to their *function execution context*, functions also have access to the execution contexts of their parents — the global context plus any functions they are enclosed in. In general, *execution context* refers to the full set of variables and methods available, either locally or via the scope chain, at the time that code is executed.

This concept will become clearer as you see it in action in upcoming lessons, but for now just know that the execution context is how JavaScript knows things like what variables are in scope, what methods are available, etc.

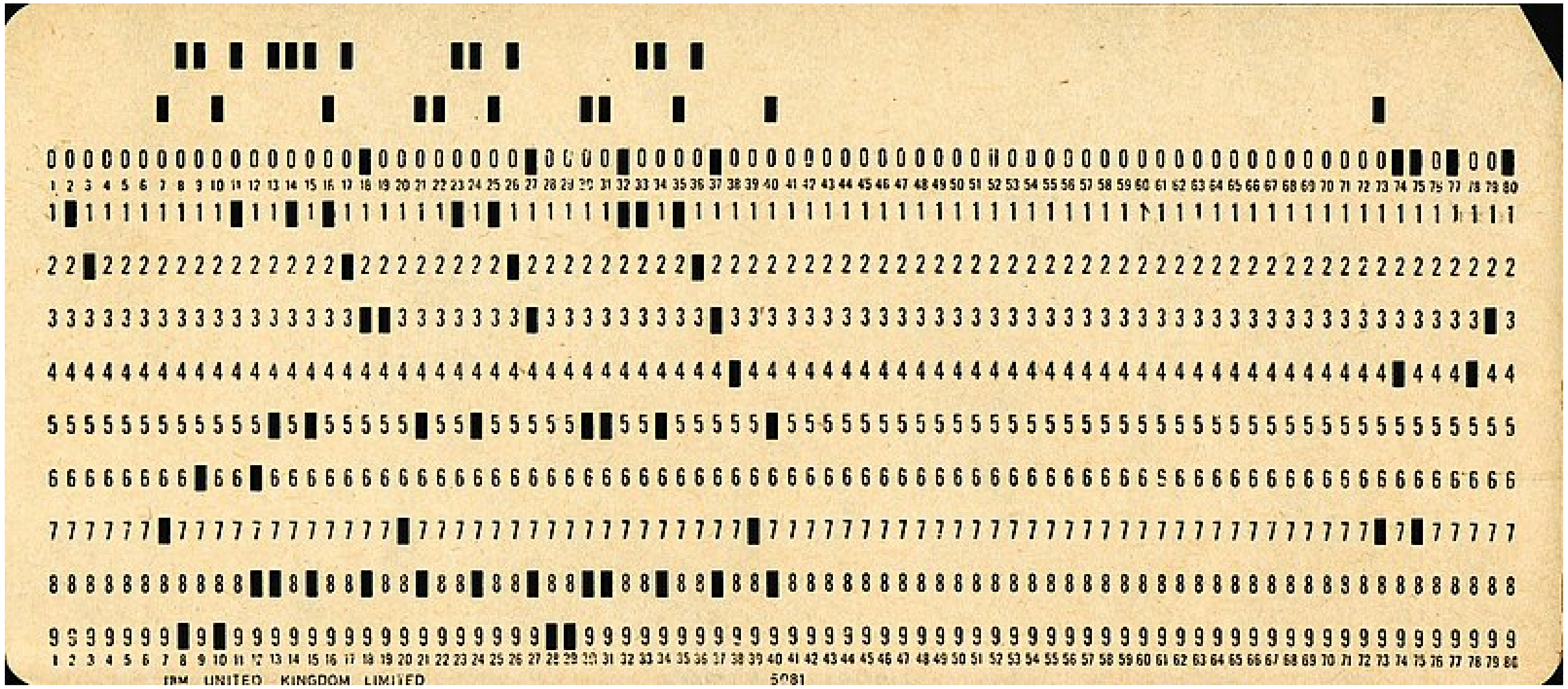
We're going to start this sequence of lessons off by defining five key terms. Each will be addressed in depth in upcoming lessons, but we want to introduce them now so you can say "Oh, this is important" when you see them later.

1. Execution Context: the space set aside in the JavaScript engine's memory containing references to the variables and methods that are currently in scope. At the global level, this is the *global execution context*; inside a function, it is the *function execution context* plus the execution contexts available to it via the scope chain.
2. `this` : refers to a special object that is part of the current execution context. The value of `this` is determined based on how and where the function is invoked. In future lessons, we'll explain how JavaScript determines what `this` is when a function is invoked.
3. `call` : This is a method *on a function* that calls the function, just like `()` . The first argument, traditionally called `thisArg` , is the object that should be used as the value of `this` when the function is invoked. You then list any other arguments you want to send to the function after the `thisArg` . An invocation of `call` looks like this: `Calculator.sum.call(multilingualMessages, 1, 2)` .
4. `apply` : This is a method *on a function* that calls the function, just like `()` . The first argument, traditionally called `thisArg` , is the object that should be used as the value of `this` when the function is invoked. You then pass additional arguments you want to send to the function as an `Array` after the `thisArg` . An invocation of `apply` looks like this: `Calculator.sum.apply(multilingualMessages, [1, 2])` .
5. `bind` : This method returns *a copy* of the function it's called on, but with the execution context "set" to the argument that's passed to `bind` . It looks like this: `sayHello.bind(greenFrog)("Hello") //=> "Mr. GreenFrog says *Hello* to you all."` .

Don't worry if this isn't making a lot of sense just yet — we will cover these concepts in depth soon. Before we do that, however, in this lab we'll create some context by practicing what we've already learned about JavaScript to build a time card application. This application is an example of a "record-oriented" application, which is quite simply an application that is used to process *records*. For this lab, the records are the time cards for each employee. Once we have a working application, we'll show how execution context, `this` , `call` , `apply` and `bind` can DRY up our code.

## Define the Term "Record"

Back in the old days (the 1960s and earlier) computers didn't have much memory. Records were stored on, if you can even believe this, small paper cards called punch-cards. They looked like this:



These cards, or "records," often had information on them in "fields." In the `first_name` field, you'd find a first name, etc... An employee's card would also include a record of how many hours they worked on each day. So when a business needed to figure out how much to pay each person for a week's work, something like the following would happen:

- Load up all the employees' cards into a tray
- Feed the tray of cards into the computer
- The computer would read in each card and calculate the hours worked for the week per card
- The computer would emit a new card with all the old data but this card would have a new field added called something like `wagesPaidInWeek33OfYear: 550`
- The computer would also print out a table containing the employees' names and how much each of them was owed

> **ASIDE**: Come to think of it, iterating over a collection, performing a transformation and emitting a new collection where every element has been transformed sounds an *awful* lot like `map` to us.

Then, the emitted pay ledger could be taken to the payroll department and the appropriate person could write (Write! With their hands! Using a pen and ink!) out paychecks to the employees.

Here's another use. If the executive team needed to know how much payroll cost the company in a given week, they'd (you guessed it!) load up all those punch cards in a tray and run them through a different program that calculated a total.

> **ASIDE**: Come to think of it, iterating over a collection, performing an evaluation on each element and emitting a new value based on those elements sounds an *awful* lot like `reduce` to us.

Ultimately, the "punch card" was an intermediate step between paper records and digital records. But it was during the punch-card era that computing really got big, so a *lot* of our ways of thinking about programming started by thinking about "records."

## Define the Term "Record-Oriented Programming"

**Record-oriented programming** (https://en.wikipedia.org/wiki/Record_(computer_science)) is a style of programming based on accessing records and processing them so that they're updated ( `map` -like) or so that their information is aggregated ( `reduce` -like). "Record-oriented" isn't a buzzword that we hear used very much, but for these next few lessons, we'll use it. Ask any programmer who's worked in large scale billing (at phone companies, insurers, etc.) or at a university (50,000 grade point averages), and you can bet they'll understand what the term means, though.

The amazing thing is that in the 21st century this style of programming is back in vogue! We're not using punch cards, but the ability to spin up hundreds of little computers in a cloud, hand them each a bundle of records, and get answers back is *cutting-edge!*

In fact, a program to do `map` and `reduce` operations at scale on a cloud was standardized in the 2000s. Guess what it's called? `mapReduce` (https://en.wikipedia.org/wiki/MapReduce) . It was pioneered and advanced as part of the secret sauce that made a small little company from Mountain View, California called Google become the giant it is today. Today you can use it under the name of **Apache Hadoop** (https://en.wikipedia.org/wiki/Apache_Hadoop)

The "Go" programming language is built around building and processing records at scale. Record-oriented programming is not likely to go away any time soon. Maybe it'll be the hot job posting buzzword any minute now!

## Lab

In this lab, we're going to build a time card and payroll application using the record-oriented approach. When someone enters the company's state of the art technical office, the employee has to insert their card in a time clock which will record the time they came in. When it's time to leave, the employee will "punch out."

For simplicity's sake, we'll make these assumptions:

1. Employees always check in **and** check out.
2. Employees always check in and out on the hour.
3. The time is represented on a 24-hour clock (1300 is 1:00 pm); this keeps the math easier and is the standard in most of the world.
4. When timestamps are needed, they will be provided as `String` s in the form: `"YYYY-MM-DD 800"` or `"YYYY-MM-DD 1800"` e.g. `"2018-01-01 2300"` .
5. Employees will never work across days, e.g., in at `2200` and out at `0400` the next day.

The lab tests will guide you toward a solution. When you encounter a failing test, look at how the test is calling the function that's missing or failing: how did it call the function, what arguments did it pass? What kind of thing did it expect back?

Take advantage of the collection-processing strengths you built up over the last few lessons.

## Extending the Challenge

If you have the time, you can learn more about JavaScript and remove the simplifying assumptions we wrote above. You can expand your learning by:

- Raising an exception if a `timeIn` is found without a matching `timeOut`
  - **Exception Handling in JavaScript** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error)
- Figuring out how to turn a time stamp into a construct that allows for you to handle cross-day times and start and end times that aren't on the hour
  - **Date Class Documentation** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)
- Raising errors if the time stamp is in an invalid format

While the bar set by the tests is at one level, you can turn this into a robust application, if you so desire!

Do your coding in `index.js` .

You should be guided by the tests as you work through the lab, but to help make the tests easier to read, we've also provided the *signatures* of the functions below. A function *signature* is the function name, the arguments it expects, and what the function returns. Pay close attention to the description of each function's expected behavior.

## `createEmployeeRecord`

- **Argument(s)**
  - A 4-element Array of a `String` , `String` , `String` , and `Number` corresponding to a first name, family name, title, and pay rate per hour
- **Returns**
  - JavaScript `Object` with keys:
  - `firstName`
  - `familyName`
  - `title`
  - `payPerHour`
  - `timeInEvents`
  - `timeOutEvents`
- **Behavior**
  - Loads `Array` elements into corresponding `Object` properties. *Additionally*, initialize empty `Array` s on the properties `timeInEvents` and `timeOutEvents` .

## `createEmployeeRecords`

- **Argument(s)**
  - `Array` of `Arrays`
- **Returns**
  - `Array` of `Object` s
- **Behavior**
  - Converts each nested `Array` into an employee record using `createEmployeeRecord` and accumulates it to a new `Array`

## `createTimeInEvent`

- **Argument(s)**

    - An employee record `Object`
    - A date stamp ( `"YYYY-MM-DD HHMM"` )
- **Returns**
    - The employee record
- **Behavior**
    - Add an `Object` with keys to the `timeInEvents` `Array` on the record `Object` :
    - `type` : Set to `"TimeIn"`
    - `hour` : Derived from the argument
    - `date` : Derived from the argument

## `createTimeOutEvent`

- **Argument(s)**
    - An employee record `Object`
    - A date stamp ( `"YYYY-MM-DD HHMM"` )
- **Returns**
    - The employee record
- **Behavior**
    - Add an `Object` with keys to the `timeOutEvents` `Array` on the record `Object` :
    - `type` : Set to `"TimeOut"`
    - `hour` : Derived from the argument
    - `date` : Derived from the argument

## `hoursWorkedOnDate`

- **Argument(s)**
    - An employee record `Object`
    - A date of the form `"YYYY-MM-DD"`
- **Returns**
    - Hours worked, an `Integer`
- **Behavior**

    - Given a date, find the number of hours elapsed between that date's timeInEvent and timeOutEvent

## `wagesEarnedOnDate`

- **Argument(s)**
    - An employee record `Object`
    - A date of the form `"YYYY-MM-DD"`
- **Returns**
    - Pay owed
- **Behavior**
    - Using `hoursWorkedOnDate` , multiply the hours by the record's payRate to determine amount owed. Amount should be returned as a number.

## `allWagesFor`

- **Argument(s)**
    - An employee record `Object`
- **Returns**
    - Pay owed for all dates
- **Behavior**
    - Using `wagesEarnedOnDate` , accumulate the value of all dates worked by the employee in the record used as context. Amount should be returned as a number. **HINT**: You will need to find the available dates somehow...

## `calculatePayroll`

- **Argument(s)**
    - `Array` of employee records
- **Returns**
    - Sum of pay owed to all employees for all dates, as a number
- **Behavior**
    - Using `wagesEarnedOnDate` , accumulate the value of all dates worked by the employee in the record used as context. Amount should be returned as a number.

## Conclusion

Congratulations! At the end of this lab, you should have built several incredibly simple functions that leveraged `map` and `reduce` to transform and aggregate data. Take a look at your code and see where you might be repeating yourself. These repetitions will be where we can bring in the innovation of execution context. We'll learn how we can DRY up our code using execution contexts in the next lesson.

It's also worth your time to take a look at the tests in `test/indexTest.js` . Because of this application's design, it's easy to test the functions that drive the application. Some programmers consider this style of programming to be optimal for the ease of testing and simplicity of code.

## Resources

- **Record / Record-Oriented Programming (https://en.wikipedia.org/wiki/Record_(computer_science))**
- **JavaScript Error Class (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error)**
- **JavaScript Date Class (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date)**