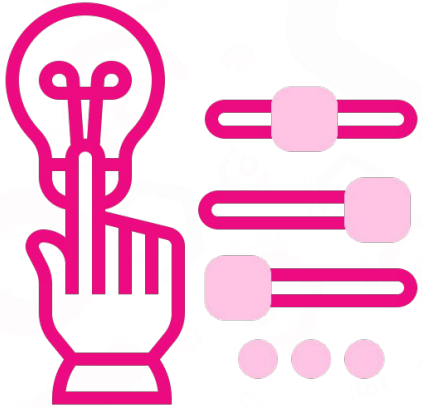
CodingLab Powered by RBK

Bootstrap



# Reduce

## An introduction to Reduce

Duration: 30 minutes
Q&A: 5 minutes by the end of the lecture

# Reduce: The Accumulator Pattern

**Reduce** is an abstraction for dealing with the *accumulator pattern*:

```
function something(array, ...) {
  var acc = <starting value>;
  each(array, function(element) {
    // update acc using element
  });
  return acc;
}
```

CodingLab Powered by RBK

## Reduce

```
function sum(nums) {



}
```

Let's start by looking at a familiar example of **accumulation**: computing the sum of an array of numbers. We can begin by declaring a function called `sum`.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;



  return total;
}
```

We know that we'll need to keep track of a running total...

CodingLab Powered by RBK

Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}
```

...and we can finish our sum function by iterating over the nums array and adding each number into our total.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}
```

What makes this an example of accumulation? Why do we call it *accumulation*?

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}
```

We can call this the ***accumulator***.

We use the **`total`** variable to store our accumulation, which starts at **`0`**...

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}
```

...we **iterate** over an array...

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}
```

...**combine** the accumulator with the next value (num in this case)...

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}
```

...update the accumulator...

CodingLab Powered by RBK

## Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}
```

...and finally **return** the accumulator when we are done iterating. This is what is meant by the *accumulator pattern*. How about another example?

CodingLab Powered by RBK

Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {




}
```

We are going to write a function that determines if every number in an array of numbers is even.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {




}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true
```

For instance, if we invoke this function on **`[2, 4, 6, 8]`**, we should get **`true`**, because all of those numbers are even.

CodingLab Powered by RBK

Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {




}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

However, if we invoke this function on **[2, 4, 5, 8]**, we should get **false**, because the number **5** is odd.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {

  each          (nums, function(        num) {

  });

}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

Our strategy will involve iterating over all of the numbers, but we need a way to keep track of whether or not each number we see is even or odd -- how can we do this?

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result      ;
  each          (nums, function(         num) {

  });

}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

Let's introduce an **accumulator** to keep track of whether or not the numbers so far are all even.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result       ;
  each          (nums, function(          num) {
                       num % 2 === 0;
  });

}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

We know that if `num % 2 === 0`, the number is even, and since *every number even* can be rephrased as the *first number is even* ***and*** *the second number is even* ***and*** *the third number is even*, etc...

CodingLab Powered by RBK

## Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result       ;
  each          (nums, function(          num) {
    result = result && num % 2 === 0;
  });

}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

We can use the logical **and** (&&) operator to express that `result` should be `true` if every number is even, and `false` if even *one* of them is odd.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result       ;
  each          (nums, function(          num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

We know that `result` should be what is returned by `everyNumberEven`...

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = ????;
  each          (nums, function(          num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

But what should the first value of `result` be?

CodingLab Powered by RBK

## Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

Since a single `false` will make the entire result `false` (due to the nature of &&), and `undefined` will be treated like `false`, we need to start with `true`. Now it works!

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

Let's take a moment to analyze the similarities and differences between these two completely unrelated functions. What is the same in both? What is different?

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(                    ) {



}
```

Let's start a function called **reduce** that we'll use to abstract the pattern.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(                    ) {
  var acc


  return acc;
}
```

First, the similarities. Both have **accumulator** variables that are returned at the end of the function.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(                ) {
  var acc
  each(     , function(element) {

  });
  return acc;
}
```

Both utilize iteration...

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(                ) {
  var acc
  each(      , function(element) {
    acc =
  });
  return acc;
}
```

...and we always update the accumulator while iterating. Now, let's turn the differences into parameters.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array           ) {
  var acc
  each(array, function(element) {
    acc =
  });
  return acc;
}
```

We will definitely need an array to iterate over...

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(          num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array    , start) {
  var acc = start;
  each(array, function(element) {
    acc =
  });
  return acc;
}
```

...and we also need to know what value to start the accumulator at.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array    , start) {
  var acc = start;
  each(array, function(element) {
    acc = ???
  });
  return acc;
}
```

The last thing that we need is a way to compute the **next** accumulator, given the **current** accumulator and an **element** in the array. What can we use to express this?

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
  each          (nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
  each          (nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array, f, start) {
  var acc = start;
  each(array, function(element) {
    acc = f(acc, element);
  });
  return acc;
}
```

How about a function? Our function will be given the **current accumulator** and an **element** in the array as arguments, and should return a **new accumulator**. Let's start refactoring!

CodingLab Powered by RBK

## Reduce

```
function sum(nums) {
  var total = 0;
        reduce(nums, function(        num) {
    total = total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
        reduce(nums, function(         num) {
    result = result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array, f, start) {
  var acc = start;
  each(array, function(element) {
    acc = f(acc, element);
  });
  return acc;
}
```

First, since `reduce` does iteration for us, let's replace `each` with `reduce`.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
        reduce(nums, function(        num) {
    return  total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
        reduce(nums, function(         num) {
    return   result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array, f, start) {
  var acc = start;
  each(array, function(element) {
    acc = f(acc, element);
  });
  return acc;
}
```

The function provided to `reduce` should **return** the next accumulator instead of doing assignment directly, so let's make that change.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
  var total = 0;
        reduce(nums, function(        num) {
    return  total + num;
  });
  return total;
}

function everyNumberEven(nums) {
  var result = true;
        reduce(nums, function(         num) {
    return   result && num % 2 === 0;
  });
  return result;
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array, f, start) {
  var acc = start;
  each(array, function(element) {
    acc = f(acc, element);
  });
  return acc;
}
```

`reduce` accomplishes the accumulation step by providing the accumulator as the first parameter to its function argument...

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
                0;
          reduce(nums, function(total, num) {
    return  total + num;
  });
  return
}

function everyNumberEven(nums) {
                 true;
          reduce(nums, function(result, num) {
    return   result && num % 2 === 0;
  });
  return
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array, f, start) {
  var acc = start;
  each(array, function(element) {
    acc = f(acc, element);
  });
  return acc;
}
```

...so we'll declare it as a parameter to the function and remove the variable that we used before. Only two steps remain:

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {
                0;
          reduce(nums, function(total, num) {
    return  total + num;
  });
  return
}

function everyNumberEven(nums) {
                true;
          reduce(nums, function(result, num) {
    return   result && num % 2 === 0;
  });
  return
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array, f, start) {
  var acc = start;
  each(array, function(element) {
    acc = f(acc, element);
  });
  return acc;
}
```

First, we need to take care of the **starting value** of the accumulator. How do we tell `reduce` what it should start the accumulator with?

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {

        reduce(nums, function(total, num) {
    return  total + num;
  }, 0);
  return
}

function everyNumberEven(nums) {

        reduce(nums, function(result, num) {
    return   result && num % 2 === 0;
  }, true);
  return
}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array, f, start) {
  var acc = start;
  each(array, function(element) {
    acc = f(acc, element);
  });
  return acc;
}
```

We provide it as the third argument to `reduce`! What is the last change that we need to make?

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {

  return reduce(nums, function(total, num) {
    return  total + num;
  }, 0);

}

function everyNumberEven(nums) {

  return reduce(nums, function(result, num) {
    return   result && num % 2 === 0;
  }, true);

}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false



function reduce(array, f, start) {
  var acc = start;
  each(array, function(element) {
    acc = f(acc, element);
  });
  return acc;
}
```

Because `reduce` results in its accumulator, the result of invoking `reduce` is what we want to return.

CodingLab Powered by RBK

# Reduce

```
function sum(nums) {

  return reduce(nums, function(total, num) {
    return  total + num;
  }, 0);

}

function everyNumberEven(nums) {

  return reduce(nums, function(result, num) {
    return   result && num % 2 === 0;
  }, true);

}
```

```
everyNumberEven([2, 4, 6, 8]);
// => true

everyNumberEven([2, 4, 5, 8]);
// => false
```

```
function reduce(array, f, start) {
  var acc = start;
  each(array, function(element) {
    acc = f(acc, element);
  });
  return acc;
}
```

And that's it!

CodingLab
Powered by BK

# That's it

For Reduce