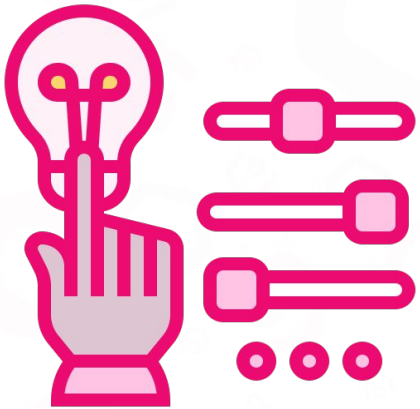
CodingLab Powered by RBK

Bootstrap



Improved Reduce

An explanation of Improved Reduce

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

CodingLab Powered by RBK

Improved reduce & Optional Params

Often when using **reduce** we can infer what the accumulator should be based on the input array -- it would be convenient in these cases if we could omit the third argument.

Let's explore the idea of **optional parameters** by giving reduce this capability.

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, start) {
  var acc = start;



  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

This is the version of **reduce** that we are familiar with.

CodingLab Powered by BBK

Improved Reduce

```
function reduce(arr, f, start) {
  var acc = start;



  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
}, 0);
// => 6
```

And we can use it to sum an array of numbers as shown in the blue box.

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, start) {
  var acc = start;



  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
}, 0);
// => 6
```

We would like to be able to omit the third parameter and have **`reduce`** make an educated guess about where it should start. **Q:** What can reduce use to *guess* the best starting point?

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, start) {
  var acc = start;



  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => ???
```

A: The **first value** in the input array!
Q: However, if we try this with our existing version of reduce, what answer will we get?

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, start) {
  var acc = start;



  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => NaN
```

A: Not a number. Why is this? Well, the first time `f` is invoked, `acc` is `undefined`, and `undefined + 1` is `NaN`.

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, start) {
  var acc = start;



  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => NaN
```

We now know that, if `start` is not provided, its value will be `undefined` -- how can we make use of this information to instead provide a starting value?

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, acc  ) {



  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => NaN
```

First, to aid in clarity, we'll rename `start` to `acc` and remove the line `var acc = start`.

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, acc  ) {
  if (acc === undefined) {


  }

  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => NaN
```

Now, we can check if the initial value for the accumulator has been supplied by comparing its value to `undefined`. If it has not been supplied, it will be `undefined`...

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, acc  ) {
  if (acc === undefined) {
    acc = arr[0];

  }

  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => NaN
```

...so we can instead assign it to the value of the first element of the array!

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, acc  ) {
  if (acc === undefined) {
    acc = arr[0];

  }

  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => ???
```

Q: What will the result of using reduce with this version now produce?

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, acc  ) {
  if (acc === undefined) {
    acc = arr[0];

  }

  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => 7
```

A: It produces 7 -- that's better than NaN...but it's not the correct answer. What's going on?

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, acc  ) {
  if (acc === undefined) {
    acc = arr[0];

  }

  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => 7
```

The problem is that now, the first value in the array -- **1** -- is being reduced **twice.** How can we keep this from happening? Somehow we need to start iterating from the *second* element instead of the first one...

CodingLab Powered by RBK

Improved Reduce

```
function reduce(arr, f, acc  ) {
  if (acc === undefined) {
    acc = arr[0];
    arr = arr.slice(1);
  }

  each(arr, function(element, i){
    acc = f(acc, element, i);
  });

  return acc;
}
```

```
reduce([1, 2, 3], function(acc, num)
{
  return acc + num;
});
// => 6
```

...and the simplest solution is to just reassign our our array to be all of the original elements **except** for the first one. Now, `reduce` produces the correct solution.

CodingLab
Powered by BK

That's it

For Improved Reduce