

EECS 368

Programming Language Paradigms

David O. Johnson

Fall 2022

Reminders

- Assignment 1 due (today): 11:59 PM, Wednesday, September 7
- Assignment 2 due: 11:59 PM, Monday, September 19

Any Questions?

In-Class Problem Solution

- 5-(9-2) In-Class Problem Solution.pptx

Any Questions?

Higher-Order Functions

- Higher-order functions are functions that operate on other functions, either:
 - taking functions as arguments
 - returning functions
- Since we have already seen that functions are regular values, there is nothing particularly remarkable about the fact that such functions exist.
- The term comes from mathematics, where the distinction between functions and other values is taken more seriously.

Higher-Order Function Example

- It is common for a program to do something a given number of times.
- You can write a for loop for that, like this:

```
for (let i = 0; i < 10; i++) {  
  console.log(i);  
}
```
- Can we abstract “doing something N times” as a function?

Higher-Order Function Example

- It is common for a program to do something a given number of times.
- You can write a for loop for that, like this:

```
for (let i = 0; i < 10; i++) {  
  console.log(i);  
}
```

- Can we abstract “doing something N times” as a function?
- Well, it’s easy to write a function that calls console.log N times.

```
function repeatLog(n) {  
  for (let i = 0; i < n; i++) {  
    console.log(i);  
  }  
}
```


Higher-Order Function Example

- But what if we want to do something other than logging the numbers?
- Since “doing something” can be represented as a function and functions are just values, we can pass our action as a function value.

```
function repeat(n, action) {  
  for (let i = 0; i < n; i++) {  
    action(i);  
  }  
}
```

```
repeat(3, console.log);  
// → 0  
// → 1  
// → 2
```

Higher-Order Function Example

- We don't have to pass a predefined function to repeat.
- Often, it is easier to create a function value on the spot instead.

```
let labels = [];  
repeat(5, i => {  
  labels.push(`Unit ${i + 1}`);  
});
```

```
console.log(labels);  
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

- The **function value** represented as an arrow function is wrapped in the **parentheses** of the call to **repeat**.

Creating New Functions

- We can have functions that create new functions:

```
function greaterThan(n) {  
  return m => m > n; //returning a new function  
}  
  
let greaterThan10 = greaterThan(10); //m => m > 10  
console.log(greaterThan10(11)); //11 => 11 > 10  
// → true
```

Changing Other Functions

- We can have functions that change other functions:

```
function noisy(f) { //makes f more verbose
  return (...args) => {
    //first by printing out arguments
    console.log("calling with", args);
    //then printing out arguments and result of function
    let result = f(...args);
    console.log("called with", args, ", returned", result);
    return result;
  };
}
noisy(Math.min)(3, 2, 1);
// → calling with [3, 2, 1]
// → called with [3, 2, 1] , returned 1
```

Providing New Types of Control Flow

- We can even write functions that provide new types of control flow:

```
function unless(test, then) {  
  if (!test) then(); //do nothing if test is false, then do arrow function  
}
```

```
repeat(3, n => {  
  unless(n % 2 == 1, () => { //if n is divisible by 2, i.e., no remainder  
    console.log(n, "is even");//then print out "n is even"  
  });  
});  
// → 0 is even  
// → 2 is even
```

Any Questions?

Built-In High-Order Array Methods

- Arrays provide a number of built-in useful higher-order methods.
- We will look at four to see how high-order functions are used:
 - `forEach`
 - `filter`
 - `map`
 - `reduce`

forEach Array Method

- There is a built-in array method, `forEach`, that provides something like a for/of loop as a higher-order function.
- You can use `forEach` to loop over the elements in an array.

```
["A", "B"].forEach(l => console.log(l));
```

```
// → A
```

```
// → B
```


Script Data Set

- To illustrate the other higher-order array methods, we will use the **Script Data Set (SCRIPTS)**.
- The Script Data Set contains information about the 140 non-Latin character scripts defined in Unicode.
- For example, here's a sample of Tamil script:

இன்னா செந்தார பூதூதல் சிவநாண
நன்னாயம் செய்து விடல்.

- The binding contains an array of objects, each of which describes a script.

```
{  
  name: "Coptic", //name of script  
  ranges: [[994, 1008], [11392, 11508], [11513, 11520]], //Unicode ranges assigned to it  
  direction: "ltr", //direction in which it is written  
    // "ltr" for left to right  
    // "rtl" for right to left (the way Arabic and Hebrew text are written)  
    // "ttb" for top to bottom (as with Mongolian writing)  
  year: -200, //the (approximate) origin time  
  living: false, //whether it is still in use  
  link: https://en.wikipedia.org/wiki/Coptic\_alphabet //link to more information  
}
```

filter Array Method

- **filter** filters out the elements in an array that don't pass a test:

```
function filter(array, test) {  
  let passed = [];  
  for (let element of array) {  
    if (test(element)) {  
      passed.push(element);  
    }  
  }  
  return passed;  
}
```

```
console.log(filter(SCRIPTS, script => script.living));  
// → [{name: "Adlam", ...}, ...]
```

- **filter** is a standard array method.
- The example defined the function only to show what it does internally.
- In a real JavaScript program, we would use **filter** like this instead:

```
console.log(SCRIPTS.filter(s => s.direction == "ttb"));  
// → [{name: "Mongolian", ...}, ...]
```

map Array Method

- **map** transforms an array by applying a function to all of its elements and building a new array from the returned values.
- The new array will have the same length as the input array, but its content will have been mapped to a new form by the function.

```
function map(array, transform) {  
  let mapped = [];  
  for (let element of array) {  
    mapped.push(transform(element));  
  }  
  return mapped;  
}
```

```
let rtlScripts = SCRIPTS.filter(s => s.direction == "rtl");  
console.log(map(rtlScripts, s => s.name));  
// → ["Adlam", "Arabic", "Imperial Aramaic", ...]
```

- **map** is a standard array method.
- The example defined the function only to show what it does internally.
- In a real JavaScript program, we would use **map** like this instead:

```
console.log(Math.round(average(  
  SCRIPTS.filter(s => s.living).map(s => s.year))));  
// → 1165
```

reduce Array Method

- **reduce** (sometimes called **fold**) computes a single value from an array.
- It builds a value by repeatedly taking a single element from the array and combining it with the current value.

```
console.log([1, 2, 3, 4].reduce, (a, b) => a + b, 0));  
// → 10
```

- where:

`[1, 2, 3, 4]` is the array to reduce

`(a, b) => a + b` is the combination rule

`0` is the starting value

- If your array contains at least one element, you are allowed to leave off the start argument.
- The method will take the first element of the array as its start value and start reducing at the second element.

Any Questions?

Summary

- Higher-order functions are functions that operate on other functions, either:
 - taking functions as arguments
 - returning functions
- We can have higher-order functions that:
 - create new functions
 - change other functions
 - provide new types of control flow
- Arrays provide a number of useful higher-order methods:
 - The `forEach` method loops over the elements in an array.
 - The `filter` method returns a new array containing only the elements that pass the predicate function.
 - The `map` method transforms an array by putting each element through a function.
 - The `reduce` method combines all the elements in an array into a single value.

Any Questions?

In-Class Problem

1. What will the function in the blue box below calculate for a script from the Script Data Set?
2. What will console.log display?
3. Describe what this code from below does:

```
SCRIPTS.reduce((a, b) => {  
  return characterCount(a) < characterCount(b) ? b : a;  
})
```

```
function characterCount(script) {  
  return script.ranges.reduce((count, [from, to]) => {  
    return count + (to - from);  
  }, 0);  
}  
  
console.log(SCRIPTS.reduce((a, b) => {  
  return characterCount(a) < characterCount(b) ? b : a;  
}));
```