Capture of variables

```
let makeWallet = (sum) => {
  return (pay) => {
    sum -= pay;
    return sum;
  };
};

let sum1 = 1000;
let payFromWallet1 = makeWallet(sum1);
let sum2 = 500;
let payFromWallet2 = makeWallet(sum2);

let balance = payFromWallet1(50);

console.log(`balance is ${balance}`);
// balance is 950

balance = payFromWallet2(70);

console.log(`balance is ${balance}`);
// balance is 430

balance = payFromWallet1(150);

console.log(`balance is ${balance}`);
// balance is 800
```

Currying

```
let carry = (f) => (a) => (b) => f(a, b);

let avg = (a, b) => (a + b) / 2;
let n1 = avg(1, 3);
// n1 is 2

let avg1 = carry(avg)(1);
// avg1 is avg func with first param = 1
let n2 = avg1(5);
// n2 is 3 = (1 + 5) / 2

console.log(`n1 is ${n1}`); // n1 is 2
console.log(`n2 is ${n2}`); // n2 is 3
```

Function as a parameter

```
let numbers = [2, 3, 1];
numbers.sort((a, b) => a - b);
// numbers is [3, 2, 1]
console.log(`numbers is [${numbers}]`);

// Up to ES6
numbers.sort(function (a, b) {
  return a - b;
});

// numbers is [3, 2, 1]
console.log(`numbers is [${numbers}]`);
```

Function as a return value

```
let makeSum = () => (a, b) => a + b;

let sumFunc = makeSum();
let sum = sumFunc(5, 8);

console.log(`sum is ${sum}`);
// sum is 13

// Up to ES6
var makeSum2 = function () {
  return function (a, b) {
    return a + b;
  };
};

var sumFunc2 = makeSum2();
var sum2 = sumFunc2(7, 8);

console.log(`sum2 is ${sum2}`);
// sum2 is 15
```

Memoization

```
function memoize(fun) {
  let memo = new Map();
  return (x) => {
    if (x in memo) {
      return memo[x];
    }
    let r = fun(x);
    memo[x] = r;
    return r;
  };
}

function fibonacci(x) {
  return x <= 1 ? x : fibonacci(x - 1) + fibonacci(x - 2);
}

let memFibonacci = memoize(fibonacci);

for (let i in [1, 2]) {
  let start = new Date();
  let f37 = memFibonacci(37);
  let milliseconds = new Date() - start;

  console.log(`${i}: f37 is ${f37}`);
  console.log(`${i}: milliseconds is ${milliseconds}`);
}
// prints:
// 1: f37 is 24157817
// 1: milliseconds is 245
// 2: f37 is 24157817
// 2: milliseconds is 0

let start = new Date();
let f38 = memFibonacci(38);
let milliseconds = new Date() - start;

console.log(`f38 is ${f38}`);
console.log(`milliseconds is ${milliseconds}`);
// f38 is 39088169
// milliseconds is 357
```

Memoization (Recursive)

```
function memoize(fun) {
  let memo = new Map();
  let memoFun = (x) => {
    if (memo.has(x)) {
      return memo.get(x);
    }
    let r = fun(memoFun, x);
    memo.set(x, r);
    return r;
  };
  return memoFun;
}

let fib = (f, x) => (x > 1 ? f(x - 1) + f(x - 2) : x);

let memFibonacci = memoize(fib);

for (let i in [1, 2]) {
  let start = new Date();
  let f37 = memFibonacci(37);
  let milliseconds = new Date() - start;

  console.log(`${i}: f37 is ${f37}`);
  console.log(`${i}: milliseconds is ${milliseconds}`);
}
// prints:
// 1: f37 is 24157817
// 1: milliseconds is 1
// 2: f37 is 24157817
// 2: milliseconds is 0

let start2 = new Date();
let f38 = memFibonacci(38);
let milliseconds2 = new Date() - start2;

console.log(`f38 is ${f38}`);
console.log(`milliseconds is ${milliseconds2}`);
// f38 is 39088169
// milliseconds is 1
```

Modify captured variables

```
let x = 5;
let addYtoX = (y) => (x += y);
addYtoX(3);

console.log(`x is ${x}`);
// x is 8

// Up to ES6
var x1 = 5;
var addYtoX1 = function (y) {
  x1 += y;
};
addYtoX1(1);

console.log(`x1 is ${x1}`);
// x1 is 6
```

Recursion

```
let fibonacci = (x) => {
  return x <= 2 ? 1 : fibonacci(x - 1) + fibonacci(x - 2);
};

let f10 = fibonacci(10);
// f10 is 55
console.log(`f10 is ${10}`);
```

Void function as a parameter

```
let checkAndProcess = (number, process) => {
  if (number < 10) {
    process(number);
  }
};

let process = (number) => console.log(number * 10);

checkAndProcess(5, process);
// printed: 50
```

With multiple operators

```
function Point(x, y) {
  this.x = x;
  this.x = y;
}

let getDistance = (p1, p2) => {
  let d1 = Math.pow(p1.x - p2.x, 2);
  let d2 = Math.pow(p1.y - p2.y, 2);
  return Math.sqrt(d1 + d2);
};

let point1 = new Point(0, 0);
let point2 = new Point(5, 5);
let distance = getDistance(point1, point2);
// distance is 7.071
console.log(`distance is ${distance}`);

// Up to ES6
function Point1(x, y) {
  this.x = x;
  this.y = y;
}

var getDistance1 = function (p1, p2) {
  var d1 = Math.pow(p1.x - p2.x, 2);
  var d2 = Math.pow(p1.y - p2.y, 2);
  return Math.sqrt(d1 + d2);
};

var point3 = new Point1(0, 0);
var point4 = new Point1(7, 7);
var distance2 = getDistance1(point3, point4);

console.log(`distance2 is ${distance2}`);
// distance2 is 9.899
```

With multiple parameters

```
let avgFunc = (a, b) => (a + b) / 2;
let avg = avgFunc(3, 5);

console.log(`avg is ${avg}`); // avg is 4

// Up to ES6
var avgFunc2 = function (a, b) {
  return (a + b) / 2;
};

var avg2 = avgFunc2(7, 5);

console.log(`avg2 is ${avg2}`); // avg is 6
```

With one parameter

```
let powOfThree = (power) => Math.pow(3, power);
let pow3 = powOfThree(3);

// Up to ES6
var powOfTwo = function (power) {
  return Math.pow(2, power);
};

var pow8 = powOfTwo(8);

console.log(`pow3 is ${pow3}`); // pow3 is 27
console.log(`pow8 is ${pow8}`); // pow8 is 256
```

Without return value

```
let add2AndPrint = (a) => console.log(a + 2);

add2AndPrint(5);
// printed 7

// Up to ES6
var add3AndPrint = function (a) {
  console.log(a + 3);
};

add3AndPrint(5);
// printed 8
```

Multi-threaded operations

Asynchronous call with a result

```
function add(a, b) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(a + b);
    }, 3000);
  });
}

add(5, 3).then((result) => console.log(result));
```

Error handling

```
// reject the result if not a number is passed
function getIntAfter1Second(i) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let val = +i;
      isNaN(val) ? reject(val) : resolve(val);
    }, 1000);
  });
}

// Since ES9
getIntAfter1Second(42)
  .then((i) => console.log("i is", i))
  .catch((e) => console.log("Error:", e))
  .finally(() => console.log("First finally!"));

async function add(a, b) {
  a = await getIntAfter1Second(a);
  b = await getIntAfter1Second(b);
  return a + b;
}

add(1, "two")
  .then((r) => console.log("Result is", r))
  .catch((e) => console.log("Error:", e))
  .finally(() => console.log("Second finally!"));
```

Keywords “async” and “await”

```
function add(a, b) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(a + b);
    }, 3000);
  });
}

async function test() {
  return await add(5, 3);
}

// Start async function and await for add() result
test().then((result) => console.log(result));
```

Start of a new thread

```
function add(a, b) {
  return a + b;
}

// ECMAScript 6 feature
setTimeout(() => {
  let result = add(3, 5);
  console.log(result);
}, 3000);
console.log("main thread");
// output:
// main thread
// result: 8
```

Start of a new thread and waiting

```
async function showAddResult(a, b) {
  return new Promise((resolve) => {
    setTimeout(() => {
      let result = a + b;
      console.log(result);
      resolve(result);
    }, 3000);
  });
}

async function test() {
  // ECMAScript 6 feature
  // Start async function and wait for add() result
  await showAddResult(3, 5);
  console.log("main function");
  // output:
  // result:
  // main function
}

test();
```

Design Patterns

Creational patterns: Abstract factory

```
// concrete product A1
class ProductA1 {
  testA() {
    console.log("test A1");
  }
}

// concrete product A2
class ProductA2 {
  testA() {
    console.log("test A2");
  }
}

// concrete product B1
class ProductB1 {
  testB() {
    console.log("test B1");
  }
}

// concrete product B2
class ProductB2 {
  testB() {
    console.log("test B2");
  }
}

// concrete factory 1
class Factory1 {
  createA() {
    return new ProductA1();
  }
  createB() {
    return new ProductB1();
  }
}

// concrete factory 2
class Factory2 {
  createA() {
    return new ProductA2();
  }
```