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

```javascript
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

```javascript
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();
  }
  createB() {
    return new ProductB2();
  }
}

// client code
function testFactory(factory) {
  let productA = factory.createA();
  let productB = factory.createB();
  productA.testA();
  productB.testB();
}

testFactory(new Factory1());
// printed: test A1
// test B1
testFactory(new Factory2());
// printed: test A2
// test B2
// ConcreteBuilder 1
class TextBuilder {
  constructor() {
    this.text = "";
  }

  addText(value) {
    this.text += value;
  }

  addNewLine(value) {
    this.text += "\n" + value;
  }

  getText() {
    return this.text;
  }
}

// ConcreteBuilder 2
class HtmlBuilder {
  constructor() {
    this.html = "";
  }

  addText(value) {
    this.html += "<span>" + value + "</span>";
  }

  addNewLine(value) {
    this.html += "<br/>\n<span>" + value + "</span>";
  }

  getHtml() {
    return this.html;
  }
}

// Director
class TextMaker {
  makeText(textBuilder) {
    textBuilder.addText("line 1");
    textBuilder.addNewLine("line 2");
  }
}

// Client
let textMaker = new TextMaker();

let textBuilder = new TextBuilder();
textMaker.makeText(textBuilder);
let text = textBuilder.getText();
// text: line 1
//       line 2

let htmlBuilder = new HtmlBuilder();
textMaker.makeText(htmlBuilder);
let html = htmlBuilder.getHtml();
// html: <span>line 1</span><br/>
//       <span>line 2</span>

console.log(`text:\n${text}`);
console.log(`html:\n${html}`);
```

Creational patterns: Factory method

```
// Product
class Employee {
  test() {
    console.log("Employee");
  }
}

// ConcreteProduct
class Manager extends Employee {
  test() {
    console.log("Manager");
  }
}

// Creator
class Creator {
  // FactoryMethod
  createEmployee() {
    return new Employee();
  }

  // Some operation
  test() {
    this.createEmployee().test();
  }
}
// ConcreteCreator
class ManagerCreator extends Creator {
  // FactoryMethod
  createEmployee() {
    return new Manager();
  }
}

// Client
let creator = new Creator();
creator.test();
// printed: Employee

creator = new ManagerCreator();
creator.test();
// printed: Manager
```

Creational patterns: Prototype

```javascript
// Prototype
class Shape {
  constructor(lineCount) {
    this.lineCount = lineCount;
  }

  clone() {
    return new Shape(this.lineCount);
  }
}

// ConcretePrototype
class Square extends Shape {
  constructor(lineCount) {
    super(4);
  }
}

// Client
class ShapeMaker {
  constructor(shape) {
    this._shape = shape;
  }

  makeShape() {
    return this._shape.clone();
  }
}

let square = new Square();
let maker = new ShapeMaker(square);

let square1 = maker.makeShape();
let square2 = maker.makeShape();

console.log("square1.lineCount is", square1.lineCount);
// square1.lineCount is 4
console.log("square2.lineCount is", square2.lineCount);
// square2.lineCount is 4
```

Creational patterns: Singleton

```javascript
class Settings {
  constructor() {
    if (Settings.prototype._singletonInstance) {
      return Settings.prototype._singletonInstance;
    }

    Settings.prototype._singletonInstance = this;

    this.port = 0;
    this.host = "";
  }
}

// Client
let settings = new Settings();
settings.host = "192.168.100.1";
settings.port = 33;

let settings1 = new Settings();
// settings1.port is 33
console.log("settings1.port is", settings1.port);
// settings1.port is 33
```

Structural patterns: Adapter (Composition)

```javascript
// Adapter
class StringList {
  constructor() {
    this.rows = [];
  }

  // SpecificRequest()
  getString() {
    return this.rows.join("\n");
  }

  add(value) {
    this.rows.push(value);
  }
}
// Adapter
class TextAdapter {
  constructor(rowList) {
    this.rowList = rowList;
  }

  // Request()
  getText() {
    return this.rowList.getString();
  }
}
function getTextAdapter() {
  let rowList = new StringList();
  let adapter = new TextAdapter(rowList);

  rowList.add("line 1");
  rowList.add("line 2");
  return adapter;
}

// Client
let adapter = getTextAdapter();
let text = adapter.getText();
// text: line 1
// line 2
console.log(text);
// line 1
// line 2
```

Structural patterns: Adapter (Inheritance)

```
// Adaptee
class StringList {
  constructor() {
    this.rows = [];
  }

  // SpecificRequests()
  getString() {
    return this.rows.join("\n");
  }

  add(value) {
    this.rows.push(value);
  }
}

// Adapter
class TextAdapter extends StringList {
  constructor() {
    super();
  }

  // Request()
  getText() {
    return this.getString();
  }
}
function getTextAdapter() {
  let adapter = new TextAdapter();
  adapter.add("line 1");
  adapter.add("line 2");
  return adapter;
}

// Client
let adapter = getTextAdapter();
let text = adapter.getText();
// text: line 1
// line 2
console.log(text);
```

Structural patterns: Bridge

```
// Implementator
class TextImp {
  constructor() {
    this._rows = [];
  }

  getString() {
    return this._rows.join("\n");
  }
}

// RefinedAbstraction
class TextMaker {
  constructor(imp) {
    this.textImp = imp;
  }

  getText() {
    return this.textImp.getString();
  }

  addLine(value) {
    this.textImp.appendLine(value);
  }
} // ConcreteImplementator
class HtmlBuilder extends TextImp {
  constructor() {
    super();
  }

  appendLine(value) {
    this._rows.push("<span>" + value + "</span><br/>");
  }
}

// Client
let textMaker = new TextMaker(new TextBuilder());
textMaker.addLine("line 1");
textMaker.addLine("line 2");
let text = textMaker.getText();

let htmlMaker = new TextMaker(new HtmlBuilder());
htmlMaker.addLine("line 1");
htmlMaker.addLine("line 2");
let html = htmlMaker.getText();

console.log(text);
console.log(html);
// line 1
// line 2
// <span>line 1</span><br/>
// <span>line 2</span><br/>
```

Structural patterns: Composite

```
// Leaf
class Circle {
  draw() {
    console.log("Draw circle");
  }
}

// Leaf
class Square {
  draw() {
    console.log("Draw square");
  }
} // Composite
class CImage {
  constructor() {
    this.graphics = [];
  }

  add(graphic) {
    this.graphics.push(graphic);
  }

  remove(graphic) {
    let i = this.graphics.indexOf(graphic);
    this.graphics.splice(i, 1);
  }

  draw() {
    console.log("Draw image");
    for (let i in this.graphics) {
      if (this.graphics.hasOwnProperty(i)) {
        this.graphics[i].draw();
      }
    }
  }
}

// Client
let image = new CImage();
image.add(new Circle());
image.add(new Square());
let picture = new CImage();
```

```
picture.add(image);
picture.add(new CImage());
picture.draw();

// Draw image
// Draw circle
// Draw square
```

Structural patterns: Decorator

```
// Component
class Shape {
  // Operation()
  getInfo() {
    return "shape";
  }

  showInfo() {
    console.log(this.getInfo());
  }
}

// ConcreteComponent
class Square extends Shape {
  constructor() {
    super();
  }

  // Operation()
  getInfo() {
    return "square";
  }
} // Decorator
class ShapeDecorator extends Shape {
  constructor(shape) {
    super();
    this.shape = shape;
  }

  // Operation()
  getInfo() {
    return this.shape.getInfo();
  }
}

// ConcreteDecorator
class ColorShape extends ShapeDecorator {
  constructor(shape, color) {
    super(shape);
    this.color = color;
  }
  getInfo() {
    return this.color + " " + this.shape.getInfo();
  }
}

// Create a basic square
let square = new Square();

// Decorate the square with color
let coloredSquare = new ColorShape(square, "red");

// Show information about the colored square
coloredSquare.showInfo(); // red square
```

Structural patterns: Facade

```
// Complex parts
class Kettle {
  turnOff() {
    console.log("Kettle turn off");
  }
}

class Toaster {
  turnOff() {
    console.log("Toaster turn off");
  }
}

class Refrigerator {
  turnOff() {
    console.log("Refrigerator turn off");
  }
}

// Facade
class Kitchen {
  constructor(kettle, toaster, refrigerator) {
    this.kettle = kettle;
    this.toaster = toaster;
    this.refrigerator = refrigerator;
  }

  off() {
    this.kettle.turnOff();
    this.toaster.turnOff();
    this.refrigerator.turnOff();
  }
}
let kettle = new Kettle();
let toaster = new Toaster();
let refrigerator = new Refrigerator();
let kitchen = new Kitchen(kettle, toaster, refrigerator);
kitchen.off();

// Kettle turn off
// Toaster turn off
// Refrigerator turn off
```

Structural patterns: Flyweight

```
// ConcreteFlyweight
class Char {
  constructor(c) {
    this._c = c;
  }

  // Operation(extrinsicState)
  printSpan(style) {
    let span = '<span style="' + style + '">' + this._c +
"</span>";
    console.log(span);
  }
}

// FlyweightFactory
class CharFactory {
  constructor() {
    this.chars = {};
  }

  // GetFlyweight(key)
  getChar(c) {
    let character = this.chars[c];
    if (character == undefined) {
      character = new Char(c);
      this.chars[c] = character;
    }
    return character;
  }
} // Client
let factory = new CharFactory();
let charA = factory.getChar("A");
charA.printSpan("font-size: 40pt");

let charB = factory.getChar("B");
charB.printSpan("font-size: 12");

let charA1 = factory.getChar("A");
charA1.printSpan("font-size: 12");

let equal = charA === charA1;
// equal is true
```

```
console.log(equal);

// <span style="font-size: 40pt">A</span>
// <span style="font-size: 12">B</span>
// <span style="font-size: 12">A</span>
```

Structural patterns: Proxy

```
// Subject
class Graphic {
  constructor(fileName) {
    this._fileName = fileName;
  }

  getFileName() {
    return this._fileName;
  }
}

// RealSubject
class CImage extends Graphic {
  constructor(fileName) {
    super(fileName);
  }

  // Request()
  draw() {
    console.log("draw " + this._fileName);
  }
} // Proxy
class ImageProxy extends Graphic {
  constructor(fileName) {
    super(fileName);
  }

  getImage() {
    if (this._image == undefined) {
      this._image = new CImage(this._fileName);
    }
    return this._image;
  }

  draw() {
    this.getImage().draw();
  }
}

// Client
let proxy = new ImageProxy("1.png");
// operation without creating a RealSubject
```

```
let fileName = proxy.getFileName();
// forwarded to the RealSubject
proxy.draw();
// draw 1.png
console.log("fileName is", fileName);
// fileName is 1.png
```

Behavioral patterns: Chain of responsibility

```
// Handler
class Rescuer {
  constructor(code, next) {
    this._code = code;
    this._next = next;
  }

  // HandleRequest()
  help(code) {
    if (this._code === code) {
      this.toHelp();
    } else if (this._next != undefined) {
      this._next.help(code);
    }
  }
}

// ConcreteHandler
class Firefighter extends Rescuer {
  constructor(next) {
    super(1, next);
  }

  toHelp() {
    console.log("call firefighters");
  }
} // ConcreteHandler
class Police extends Rescuer {
  constructor(next) {
    super(2, next);
  }

  toHelp() {
    console.log("call the police");
  }
}

// ConcreteHandler
class Ambulance extends Rescuer {
  constructor(next) {
    super(3, next);
  }
  toHelp() {
    console.log("call on ambulance");
  }
}

let ambulance = new Ambulance();
let police = new Police(ambulance);
let firefighter = new Firefighter(police);
firefighter.help(1);
// printed: call firefighters
firefighter.help(2);
// printed: call the police
firefighter.help(3);
// printed: call the ambulance
```

Behavioral patterns: Command

```
// Invoker
class BankClient {
  constructor(cPut, cGet) {
    this._putCommand = cPut;
    this._getCommand = cGet;
  }

  putMoney() {
    this._putCommnad.execute();
  }

  getMoney() {
    this._getCommand.execute();
  }
}

// Reciever
class Bank {
  giveMoney() {
    console.log("money to the client");
  }

  recieveMoney() {
    console.log("money from the client");
  }
} // ConcreteCommand
class PutCommand {
  constructor(bank) {
    this._bank = bank;
  }

  execute() {
    this._bank.recieveMoney();
  }
}

// ConcreteCommand
class GetCommand {
  constructor(bank) {
    this._bank = bank;
  }
  execute() {
    this._bank.giveMoney();
  }
}

// Client
let bank = new Bank();
let cPut = new PutCommand(bank);
let cGet = new GetCommand(bank);
let client = new BankClient(cPut, cGet);
client.getMoney();
// printed: money to the client
client.putMoney();
// printed: money from the client
// TerminalExpression
class DivExpression {
  constructor(divider) {
    this._divider = divider;
  }

  interpret(i) {
    return i % this._divider === 0;
  }
}

// NonterminalExpression
class OrExpression {
  constructor(exp1, exp2) {
    this.exp1 = exp1;
    this.exp2 = exp2;
  }

  interpret(i) {
    return this.exp1.interpret(i) || this.exp2.interpret(i);
  }
} // NonterminalExpression
class AndExpression {
  constructor(exp1, exp2) {
    this.exp1 = exp1;
    this.exp2 = exp2;
  }

  interpret(i) {
    return this.exp1.interpret(i) && this.exp2.interpret(i);
  }
}

// Client
let divExp5 = new DivExpression(5);
let divExp7 = new DivExpression(7);
let orExp = new OrExpression(divExp5, divExp7);
let andExp = new AndExpression(divExp5, divExp7);

// 21 is divided by 7 or 5?
let result1 = orExp.interpret(21);
// 21 is not divided by 7 and 5
```

```
let result2 = andExp.interpret(21);
// 35 is divided by 7 and 5
let result3 = andExp.interpret(35);

console.log("21 is divided by 7 or 5?", result1);
// 21 is divided by 7 or 5? true
console.log("21 is divided by 7 and 5?", result2);
// 21 is divided by 7 and 5? false
console.log("35 is divided by 7 and 5?", result3);
// 35 is divided by 7 and 5? true
```

Behavioral patterns: Iterator

```
// ConcreteAggregate
class PrimeNumbers {
  constructor() {
    this.numbers = [2, 3, 5, 7, 11];
  }

  getIterator() {
    return new Iterator(this);
  }
}
// ConcreteIterator
class Iterator {
  constructor(primeNumbers) {
    this.index = 0;
    this.numbers = primeNumbers.numbers;
  }

  first() {
    this.index = 0;
  }

  next() {
    this.index++;
  }

  isDone() {
    return this.index >= this.numbers.length;
  }

  currentItem() {
    return this.numbers[this.index]; // Fix the typo here
  }
} // Client
let numbers = new PrimeNumbers();
let iterator = numbers.getIterator();
let sum = 0;

for (iterator.first(); !iterator.isDone(); iterator.next()) {
  sum += iterator.currentItem();
}

console.log(`sum is ${sum}`); // sum is 28
```

Behavioral patterns: Mediator

```
class Mediator {
  constructor() {
    this._switchers = [];
  }

  add(switcher) {
    this._switchers.push(switcher);
  }
}

// Colleague
class Switcher {
  constructor(mediator) {
    this._state = false;
    this._mediator = mediator;
    this._mediator.add(this);
  }

  sync() {
    this._mediator.sync(this);
  }

  getState() {
    return this._state;
  }

  setState(value) {
    this._state = value;
  }
} // ConcreteMediator
class SyncMediator extends Mediator {
  constructor() {
    super();
  }

  sync(switcher) {
    let state = switcher.getState();
    let switchers = this._switchers;
    for (let switcher of switchers) {
      switcher.setState(state);
    }
  }
}

// Client
let mediator = new SyncMediator();
let switcher1 = new Switcher(mediator);
let switcher2 = new Switcher(mediator);
let switcher3 = new Switcher(mediator);

switcher1.setState(true);
let state2 = switcher2.getState();
// state2 is false
let state3 = switcher3.getState();
// state3 is false
console.log("state2 is", state2);
console.log("state3 is", state3);

switcher1.sync();
state2 = switcher2.getState();
// state2 is true
state3 = switcher3.getState();
// state3 is true
console.log("state2 is", state2);
console.log("state3 is", state3);
// state2 is false
// state3 is false
// state2 is true
// state3 is true
```

Behavioral patterns: Memento

```
// State
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
}

class Memento {
  constructor(mState) {
    this._state = mState;
  }

  getState() {
    return this._state;
  }
}

// Originator
class Shape {
  constructor() {
    this.position = new Point(0, 0);
  }

  move(left, top) {
    this.position.x += left;
    this.position.y += top;
  }

  getMemento() {
    let state = new Point(this.position.x, this.position.y);
    return new Memento(state);
  }
  setMemento(memento) {
    this.position = memento.getState();
  }

  showPosition() {
    console.log(this.position.x + ", " + this.position.y);
  }
}
```

```
// Caretaker
class ShapeHelper {
  constructor(hShape) {
    this.stack = [];
    this.shape = hShape;
  }

  move(left, top) {
    this.stack.push(this.shape.getMemento());
    this.shape.move(left, top);
  }

  undo() {
    if (this.stack.length > 0) {
      this.shape.setMemento(this.stack.pop());
    }
  }
}

let shape = new Shape();
let helper = new ShapeHelper(shape);
helper.move(2, 3);
// shape.position is (2, 3)
shape.showPosition();
helper.move(-5, 4);
// shape.position is (-3, 7)
shape.showPosition();

helper.undo();
// shape.position is (2, 3)
shape.showPosition();
helper.undo();
// shape.position is (0, 0)
shape.showPosition();
```