Functional Programming

- Functional programming is a programming paradigm
- A programming paradigm is a style of building the structure and elements of computer programs
- Functional programming is a declarative programming paradigm, which means programming is done with expressions or declarations instead of statements

Key Concepts

- Pure functions / Avoid side effects
- Referential transparency
- Immutability
- Lazy evaluation
- Function composition
    - Higher-order functions
    - Currying
    - Recursion

Pure Functions

- A pure function is a function that:
    - Given the same input, will always return the same output
    - Produces no side effects

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

Immutability

```
const arr = [1, 2, 3];
arr.push(4);
console.log(arr); // [1, 2, 3, 4]
```

```
const arr = [1, 2, 3];
const newArr = arr.concat(4);
console.log(arr); // [1, 2, 3]
console.log(newArr); // [1, 2, 3, 4]
```

Higher-Order Functions

- A higher-order function is a function that:
  - Takes one or more functions as arguments
  - Returns a function as its result

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

function operate(a, b, func) {
  return func(a, b);
}

console.log(operate(1, 2, add)); // 3
```

Currying

- Currying is the process of taking a function with multiple arguments and returning a series of functions that take one argument and eventually resolve to a value

`map`, `filter`, and `reduce`

map-filter-reduce.js

ES6+ Features

- Arrow functions
- Spread/rest operators
- Destructuring
- Default parameters
- Template literals
- Class
- `let` & `const`
- Promises (will cover later)

Arrow Functions

- no `arguments` object
- no `this` binding
- no `prototype` property

```
const add = (a, b) => {
  return a + b;
};

const sum = (a, b) => a + b;

const square = x => x * x;

const foo = () => ({ bar: 1 });
```

`this` in Arrow Functions

- Arrow functions do not have their own `this`
- The value of `this` inside an arrow function remains the same throughout the lifecycle of the function and is always bound to the value of `this` in the closest non-arrow parent function

```
const obj = {
  name: 'John',
  sayHi: function () {
    console.log(`Hi, I'm ${this.name}`);
  }
};

obj.sayHi(); // Hi, I'm John
```

```
const obj = {
  name: 'John',
  sayHi: () => {
    console.log(`Hi, I'm ${this.name}`);
  }
};

obj.sayHi(); // Hi, I'm undefined
```

Spread/Rest Operators

```
const arr = [1, 2, 3];
const newArr = [...arr, 4, 5, 6];
console.log(newArr); // [1, 2, 3, 4, 5, 6]
```

```
const obj = { a: 1, b: 2 };
const newObj = { ...obj, c: 3 };
console.log(newObj); // { a: 1, b: 2, c: 3 }
```

```
function foo(...args) {
  for (let i = 0; i < args.length; i++) {
    console.log(args[i]);
  }
}
```

Destructuring

```
const arr = [1, 2, 3];
const [a, b, c] = arr;
console.log(a, b, c); // 1 2 3
```

```
const obj = { a: 1, b: 2, c: 3 };
const { a, b, c } = obj;
console.log(a, b, c); // 1 2 3
```

Template Literals

```javascript
const name = 'John';
const age = 30;
console.log(`Hi, I'm ${name} and I'm ${age} years old`);
```

Shallow vs. Deep Copy

shallow-deep-copy.js

```js
const obj = { a: 1, b: 2 };
const newObj = obj;
newObj.a = 3;
console.log(obj.a); // 3
```

```js
const obj = { a: 1, b: 2 };
const newObj = { ...obj };
// const newObj = JSON.parse(JSON.stringify(obj));
newObj.a = 3;
console.log(obj.a); // 1
```

Map and Set

map-set.js

- `Map` is a collection of keyed data items, just like an `Object`
- `Set` is a collection of unique values
  - based on strict equality
  - `has()` method faster than `indexOf()` / `includes()`

Map vs. Object

- `Map` is iterable - `for...of`
- `Map` keys can be any type
  - `Object` keys are always strings and symbols
- `Map` keys are ordered
- `Map` size is easily retrieved
- `Map` is faster than `Object` for frequent key/value manipulation in some cases

Date and Time

- `Date` object represents a single moment in time in a platform-independent format

```
const now = new Date();
console.log(now); // 2018-01-01T00:00:00.000Z
```

```
const date = new Date(2018, 0, 1);
console.log(date); // 2018-01-01T00:00:00.000Z
```

```
const date = new Date('2018-01-01');
console.log(date); // 2018-01-01T00:00:00.000Z
```

Regular Expressions

- A regular expression is a sequence of characters that forms a search pattern
- Regular expressions can be used to perform all types of text search and text replace operations
- https://deerchao.cn/tutorials/regex/regex.htm
- https://www.regexpal.com/

```
const str = 'Hello World';
const regex = /hello/i;
console.log(regex.test(str)); // true
```

Regular Expressions (cont'd)

character	matches
`.`	any character except newline
`\w`	word character
`\W`	non-word character
`\d`	digit
`\D`	non-digit
`\s`	whitespace
`\S`	non-whitespace

flag	description
`g`	global match
`i`	case-insensitive
`m`	multiline
`s`	dotall
`u`	unicode

Regular Expressions (cont'd)

character	matches
`{n}`	exactly n times
`{n,}`	at least n times
`{n,m}`	at least n times but no more than m times
`?`	0 or 1 time
`*`	0 or more times
`+`	1 or more times

character	matches
`*?`	0 or more times, ungreedy
`+?`	1 or more times, ungreedy
`??`	0 or 1 time, ungreedy
`{n,m}?`	at least n times but no more than m times, ungreedy
`{n,}?`	at least n times, ungreedy