Task 1: Create a Container class in JavaScript that:

- Stores a value
- Has a map(fn) method that applies the function to the stored value
- Has valueOf() and toString() methods

```
class Container {
  constructor(value) {
    this.value = value;
  }

  // Applies a function to the stored value and returns a new Container
  map(fn) {
    return new Container(fn(this.value));
  }

  // Returns the raw value
  valueOf() {
    return this.value;
  }

  // Returns a string representation
  toString() {
    return `Container(${this.value})`;
  }
}

const c = new Container(10);
const c2 = c.map(x => x * 2);

console.log(c.toString());   // Container(10)
console.log(c2.toString());  // Container(20)
console.log(c2.valueOf());   // 20
```

Task 2: Write the Hindley–Milner-style type signature for each of the following:

- A function that reverses a string
- A function that adds three curried arguments
- A function that checks if a number is positive

```
// reverseString :: String → String
const reverseString = (str) => str.split('').reverse().join('');
// addThree :: Number → Number → Number → Number
const addThree = x => y => z => x + y + z;

// isPositive :: Number → Boolean
const isPositive = num => num > 0;
```

Task 3: Implement a functor by extending your Container so that:

- map(fn) returns a new functor instance
- It satisfies the two functor laws (identity and composition)

```
class Functor {
  constructor(value) {
    this.value = value;
  }

  // Functor law: map returns a new functor
  map(fn) {
    return new Functor(fn(this.value));
  }

  valueOf() {
    return this.value;
  }

  toString() {
    return `Functor(${this.value})`;
  }
```

```
}

const id = x => x;

const f1 = new Functor(10);
const f2 = f1.map(id);

console.log(f1.valueOf() === f2.valueOf()); // true
const double = x => x * 2;
const addOne = x => x + 1;
const composed = x => double(addOne(x));

const f = new Functor(5);
const left = f.map(addOne).map(double);     // map(f).map(g)
const right = f.map(composed);              // map(g ∘ f)

console.log(left.valueOf() === right.valueOf()); // true
```

Task 4: Implement the Maybe abstract class and its two subclasses:

- Just(x) holds a non-null value
- Nothing() represents the absence of a value
- map(fn) applies the function in Just, skips it in Nothing

```
class Maybe {
  static of(value) {
    return value === null || value === undefined
      ? new Nothing()
      : new Just(value);
  }

  // Abstract method to override
  map(fn) {
    throw new Error("map() must be implemented in subclasses");
  }

  isNothing() {
    throw new Error("isNothing() must be implemented in subclasses");
```

```
  }

  toString() {
    return `${this.constructor.name}(${this.value})`;
  }

  valueOf() {
    return this.value;
  }
}

class Just extends Maybe {
  constructor(value) {
    super();
    this.value = value;
  }

  map(fn) {
    return Maybe.of(fn(this.value));
  }

  isNothing() {
    return false;
  }
}

class Nothing extends Maybe {
  constructor() {
    super();
    this.value = null;
  }

  map(_) {
    return this;
  }

  isNothing() {
    return true;
  }
}
```

```
const maybe1 = Maybe.of(10).map(x => x + 5);
console.log(maybe1.toString()); // Just(15)

const maybe2 = Maybe.of(null).map(x => x + 5);
console.log(maybe2.toString()); // Nothing()
```

Task 5: Write a function getField(attr) that uses Maybe to safely access object properties, avoiding runtime errors.

```
class Maybe {
  static of(value) {
    return value === null || value === undefined
      ? new Nothing()
      : new Just(value);
  }

  // Abstract method to override
  map(fn) {
    throw new Error("map() must be implemented in subclasses");
  }

  isNothing() {
    throw new Error("isNothing() must be implemented in subclasses");
  }

  toString() {
    return `${this.constructor.name}(${this.value})`;
  }

  valueOf() {
    return this.value;
  }
}


class Just extends Maybe {
  constructor(value) {
    super();
    this.value = value;
  }
```

```
  map(fn) {
    return Maybe.of(fn(this.value));
  }

  isNothing() {
    return false;
  }
}

class Nothing extends Maybe {
  constructor() {
    super();
    this.value = null;
  }

  map(_) {
    return this;
  }

  isNothing() {
    return true;
  }
}
const getField = (attr) => (obj) =>
  Maybe.of(obj).map(o => o[attr]);

const user = { name: "Alice", age: 30 };
const noUser = null;

const nameField = getField("name");

console.log(nameField(user).toString());    // Just("Alice")
console.log(nameField(noUser).toString());  // Nothing()
```

Task 6: Demonstrate functor law compliance for your Maybe class using concrete values and transformations.

```
class Maybe {
  static of(value) {
    return value === null || value === undefined
      ? new Nothing()
      : new Just(value);
  }

  // Abstract method to override
  map(fn) {
    throw new Error("map() must be implemented in subclasses");
  }

  isNothing() {
    throw new Error("isNothing() must be implemented in subclasses");
  }

  toString() {
    return `${this.constructor.name}(${this.value})`;
  }

  valueOf() {
    return this.value;
  }
}



class Just extends Maybe {
  constructor(value) {
    super();
    this.value = value;
  }

  map(fn) {
    return Maybe.of(fn(this.value));
  }

  isNothing() {
    return false;
```

```
  }
}

class Nothing extends Maybe {
  constructor() {
    super();
    this.value = null;
  }

  map(_) {
    return this;
  }

  isNothing() {
    return true;
  }
}
const id = x => x;
const double = x => x * 2;
const addOne = x => x + 1;
const composed = x => double(addOne(x)); // g ∘ f

//Identity law test
const m1 = Maybe.of(10);
const m1Mapped = m1.map(id);

console.log(m1.toString());      // Just(10)
console.log(m1Mapped.toString()); // Just(10)
console.log(m1Mapped.valueOf() === m1.valueOf()); // true

//Composition law test
const f = addOne;
const g = double;

const m2 = Maybe.of(10);
const left = m2.map(f).map(g);               // Just(22)
const right = m2.map(x => g(f(x)));          // Just(22)

console.log(left.toString());  // Just(22)
console.log(right.toString()); // Just(22)
console.log(left.valueOf() === right.valueOf()); // true
```

Task 7: Create a composed function using Maybe:

- Look up a user from a list
- Safely get their address field
- Chain calls to extract the city name

```
class Maybe {
  static of(value) {
    return value === null || value === undefined
      ? new Nothing()
      : new Just(value);
  }

  // Abstract method to override
  map(fn) {
    throw new Error("map() must be implemented in subclasses");
  }

  isNothing() {
    throw new Error("isNothing() must be implemented in subclasses");
  }

  toString() {
    return `${this.constructor.name}(${this.value})`;
  }

  valueOf() {
    return this.value;
  }
}


class Just extends Maybe {
  constructor(value) {
    super();
    this.value = value;
```

```
  }

  map(fn) {
    return Maybe.of(fn(this.value));
  }

  isNothing() {
    return false;
  }
}

class Nothing extends Maybe {
  constructor() {
    super();
    this.value = null;
  }

  map(_) {
    return this;
  }

  isNothing() {
    return true;
  }
}

const users = [
  { id: 1, name: "Alice", address: { city: "London" } },
  { id: 2, name: "Bob" },
  { id: 3, name: "Carol", address: { city: null } },
];

const findUserById = (id) =>
  Maybe.of(users.find(user => user.id === id));

const getField = (attr) => (obj) =>
  Maybe.of(obj).map(o => o[attr]);

const getUserCity = (id) =>
  findUserById(id)
    .map(getField("address"))
```

```
    .map(maybeAddr => maybeAddr?.valueOf())
    .map(getField("city"))
    .map(maybeCity => maybeCity?.valueOf());

console.log(getUserCity(1).toString()); // Just("London")
console.log(getUserCity(2).toString()); // Nothing()
console.log(getUserCity(99).toString()); // Nothing()
```

Task 8: Extend your functor into a Monad:

- Implement chain(fn) to flatten nested monads
- Add unwrap() to get the raw inner value

```
class Monad {
  constructor(value) {
    this.value = value;
  }

  // map returns a new monad with the function applied
  map(fn) {
    return new Monad(fn(this.value));
  }

  // chain flattens the nested monads
  chain(fn) {
    const result = fn(this.value);
    return result instanceof Monad ? result : new Monad(result);
  }

  // unwrap returns the innermost raw value
  unwrap() {
    let current = this;
    while (current instanceof Monad) {
      if (!(current.value instanceof Monad)) break;
      current = current.value;
    }
    return current.value;
  }

  valueOf() {
```

```
    return this.value;
  }

  toString() {
    return `Monad(${this.value})`;
  }

  static of(x) {
    return new Monad(x);
  }
}

const m = Monad.of(5)
  .map(x => x + 2)               // Monad(7)
  .chain(x => Monad.of(x * 3)); // Monad(21)

console.log(m.toString());      // Monad(21)
console.log(m.unwrap());        // 21

const nested = Monad.of(Monad.of(42));
console.log(nested.unwrap());   // 42 fully unwrapped
```

Task 9: Refactor the following to use monads:

```
const result = fn1(x);
if (result) {
  const next = fn2(result);
  if (next) {
    return fn3(next);
  }
}
return null;
```

Refactored:

```
return Maybe.of(x)
  .map(fn1)
  .map(fn2)
  .map(fn3)
  .valueOf();
```

Task 10: Add an ap() method to your Monad class and use it to apply a wrapped function to a wrapped value:

```
Monad.of(x => x + 1).ap(Monad.of(5)); // Monad(6)
```

```
class Monad {
  constructor(value) {
    this.value = value;
  }

  static of(x) {
    return new Monad(x);
  }

  map(fn) {
    return new Monad(fn(this.value));
  }

  chain(fn) {
    const result = fn(this.value);
    return result instanceof Monad ? result : new Monad(result);
  }

  ap(m) {
    if (typeof this.value !== 'function') {
      throw new Error("ap() expects a monad containing a function");
    }
    return m.map(this.value);
  }

  valueOf() {
    return this.value;
  }

  toString() {
    return `Monad(${this.value})`;
```

```
  }

  unwrap() {
    let current = this;
    while (current instanceof Monad && current.value instanceof Monad) {
      current = current.value;
    }
    return current.value;
  }
}

const fnMonad = Monad.of(x => x + 1);
const valMonad = Monad.of(5);

const result = fnMonad.ap(valMonad);

console.log(result.toString()); // Monad(6)
console.log(result.valueOf());  // 6
```

Task 11: Implement the Either monad with Left(error) and Right(value).

- map(fn) runs only on Right
- Left skips mapping

```
class Either {
  static of(left, right) {
    return right === null || right === undefined
      ? new Left(left)
      : new Right(right);
  }

  map(_) {
    throw new Error("map() must be implemented in Left or Right");
  }

  isLeft() {
    throw new Error("isLeft() must be implemented in Left or Right");
  }
```

```
  toString() {
    return `${this.constructor.name}(${this.value})`;
  }

  valueOf() {
    return this.value;
  }
}

class Left extends Either {
  constructor(error) {
    super();
    this.value = error;
  }

  map(_) {
    return this; // skip mapping
  }

  isLeft() {
    return true;
  }
}

class Right extends Either {
  constructor(value) {
    super();
    this.value = value;
  }

  map(fn) {
    return Either.of(null, fn(this.value));
  }

  isLeft() {
    return false;
  }
}

const success = Either.of(null, 10)
```

```
  .map(x => x + 1)
  .map(x => x * 2);

console.log(success.toString()); // Right(22)

const failure = Either.of("Something went wrong")
  .map(x => x + 1);

console.log(failure.toString()); // Left(Something went wrong)
```

Task 12: Write safeDivide(a, b) using Either:

- Return Right(result) or Left("Division by zero")

```
class Either {
  static of(left, right) {
    return right === null || right === undefined
      ? new Left(left)
      : new Right(right);
  }

  map(_) {
    throw new Error("map() must be implemented in Left or Right");
  }

  isLeft() {
    throw new Error("isLeft() must be implemented in Left or Right");
  }

  toString() {
    return `${this.constructor.name}(${this.value})`;
  }

  valueOf() {
    return this.value;
  }
}

class Left extends Either {
```

```
  constructor(error) {
    super();
    this.value = error;
  }

  map(_) {
    return this; // skip mapping
  }

  isLeft() {
    return true;
  }
}

class Right extends Either {
  constructor(value) {
    super();
    this.value = value;
  }

  map(fn) {
    return Either.of(null, fn(this.value));
  }

  isLeft() {
    return false;
  }
}

const safeDivide = (a, b) => {
  return b === 0
    ? new Left("Division by zero")
    : new Right(a / b);
};

const result1 = safeDivide(10, 2).map(x => x + 1);
console.log(result1.toString()); // Right(6)

const result2 = safeDivide(10, 0).map(x => x + 1);
console.log(result2.toString()); // Left(Division by zero)
```

Task 13: Implement Try.of(() => riskyOperation(), msg):

- Catch errors and wrap them in Left
- On success, return Right

```
// Base Either class
class Either {
  static of(left, right) {
    return right === null || right === undefined
      ? new Left(left)
      : new Right(right);
  }

  map(_) {
    throw new Error("map() must be implemented by subclasses");
  }

  isLeft() {
    throw new Error("isLeft() must be implemented by subclasses");
  }

  valueOf() {
    return this.value;
  }

  toString() {
    return `${this.constructor.name}(${this.value})`;
  }
}

// Left class: represents failure
class Left extends Either {
  constructor(error) {
    super();
    this.value = error;
  }

  map(_) {
```

```
    return this; // skip mapping
  }

  isLeft() {
    return true;
  }
}

// Right class: represents success
class Right extends Either {
  constructor(value) {
    super();
    this.value = value;
  }

  map(fn) {
    return Either.of(null, fn(this.value));
  }

  isLeft() {
    return false;
  }
}

// Try class for safe execution
class Try {
  static of(fn, msg) {
    try {
      const result = fn();
      return new Right(result);
    } catch (e) {
      return new Left(msg || e.message);
    }
  }
}

// A risky function that sometimes throws
const riskyOperation = () => {
  if (Math.random() < 0.5) {
    throw new Error("Random failure");
  }
```

```
  return "It worked!";
};

// Try running it safely
const result = Try.of(riskyOperation, "Default failure message");

// Output result
console.log(result.toString());  // Either Left("...") or Right("It
worked!")
```

Task 14: Write treeIsEmpty(tree) and treeRoot(tree) to:

- Return true/false based on tree contents
- Return the root value or null

```
// Functional Binary Tree Definition
const EmptyTree = () => (nonEmpty, empty) => empty();

const NewTree = (value, left, right) =>
  (nonEmpty, empty) => nonEmpty(value, left, right);

// treeIsEmpty :: Tree → Boolean
const treeIsEmpty = (tree) =>
  tree(
    () => false,  // non-empty case
    () => true    // empty case
  );

// treeRoot :: Tree → a | null
const treeRoot = (tree) =>
  tree(
    (value, _left, _right) => value,
    () => null
  );
// Create an example tree
const myTree = NewTree(
  10,
  EmptyTree(),
```

```
  EmptyTree()
);

const empty = EmptyTree();

console.log("Is tree empty?", treeIsEmpty(myTree));      // false
console.log("Root of tree:", treeRoot(myTree));          // 10

console.log("Is empty tree empty?", treeIsEmpty(empty)); // true
console.log("Root of empty tree:", treeRoot(empty));     // null
```

Bonus:

Task 15: Write treeInsert(value, tree) to insert a value into a binary search tree.

```
// Tree constructors using Church encoding
const EmptyTree = () => (nonEmpty, empty) => empty();

const NewTree = (value, left, right) =>
  (nonEmpty, empty) => nonEmpty(value, left, right);

const treeInsert = (newValue, tree) =>
  tree(
    (value, left, right) =>
      newValue <= value
        ? NewTree(value, treeInsert(newValue, left), right)
        : NewTree(value, left, treeInsert(newValue, right)),
    () => NewTree(newValue, EmptyTree(), EmptyTree())
  );

  let tree = EmptyTree();
tree = treeInsert(10, tree);
tree = treeInsert(5, tree);
tree = treeInsert(15, tree);
tree = treeInsert(7, tree);

// Convert to object for printing
const treeToObject = (tree) =>
```

```
  tree(
    (value, left, right) => {
      const result = { value };
      const leftObj = treeToObject(left);
      const rightObj = treeToObject(right);
      if (leftObj) result.left = leftObj;
      if (rightObj) result.right = rightObj;
      return result;
    },
    () => null
  );

console.log(JSON.stringify(treeToObject(tree), null, 2));
```

Task 16: Write treeSearch(value, tree) to return true if value exists, false otherwise.

```
// Functional binary tree (Church-encoded)
const EmptyTree = () => (nonEmpty, empty) => empty();

const NewTree = (value, left, right) =>
  (nonEmpty, empty) => nonEmpty(value, left, right);

const treeSearch = (searchValue, tree) =>
  tree(
    (value, left, right) =>
      searchValue === value
        ? true
        : searchValue < value
        ? treeSearch(searchValue, left)
        : treeSearch(searchValue, right),
    () => false
  );

  let tree = EmptyTree();
tree = NewTree(10,
  NewTree(5, EmptyTree(), NewTree(7, EmptyTree(), EmptyTree())),
  NewTree(15, EmptyTree(), EmptyTree())
```

```
);

console.log(treeSearch(7, tree));  // true
console.log(treeSearch(20, tree)); // false
```

You can download code from here.