The Checklist

Pure functions are the atomic building blocks in FP, adored for their simplicity and testability. Not every function is pure, however, and this section discusses how to judge one for yourself. (5 min. read)

Pure Functions

A function must pass two tests to be considered "pure"

1. Same inputs *always* return same outputs

2. No side-effects

Ramda obeys these rules to facilitate FP in JavaScript. Let's zoom in on each one.

Same Input => Same Output

Compare this:

```
const add = (x, y) => x + y;

add(2, 4); // 6
```

To this:

```
let x = 2;

const add = (y) => {
  x += y;
};

add(4); // x === 6 (the first time)
```
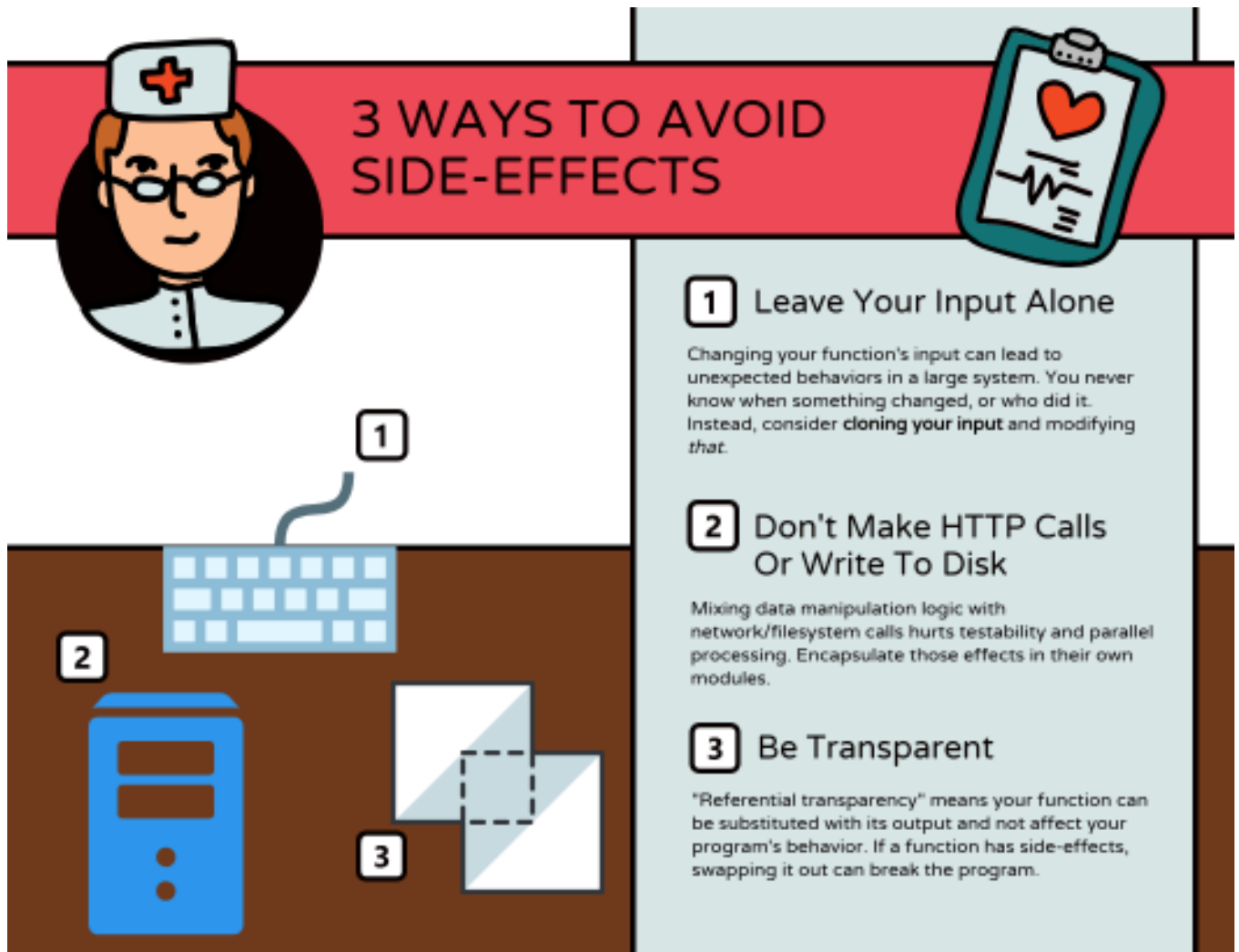
The first example returns 6 if given 2 and 4, no matter when you call it. The second example increments an outside variable by the given `y`.

This **shared state** introduces a time dependency–you get different results depending on when you called the function. The first time results in 6, next time is 10 and so on.

Which version's easier to reason about? Which one's less likely to breed bugs that happen only under certain conditions? Which one's more likely to succeed in a multi-threaded environment where time dependencies can break the system? Definitely the first one.

No Side-Effects

3 WAYS TO AVOID SIDE-EFFECTS

1. Leave Your Input Alone

Changing your function's input can lead to unexpected behaviors in a large system. You never know when something changed, or who did it. Instead, consider **cloning your input** and modifying *that*.

2. Don't Make HTTP Calls Or Write To Disk

Mixing data manipulation logic with network/filesystem calls hurts testability and parallel processing. Encapsulate those effects in their own modules.

3. Be Transparent

"Referential transparency" means your function can be substituted with its output and not affect your program's behavior. If a function has side-effects, swapping it out can break the program.

This test itself is a check-list. A few examples of side-effects are

1. Mutating your input
2. `console.log`
3. HTTP calls (like AJAX/fetch)
4. Changing the filesystem
5. Querying the DOM

Basically any "side work" you perform that isn't related to calculating a function's final output. Violating this rule can lead to the state management issues we just discussed.

Here's an impure function with a side-effect

```
const impureDouble = (x) => {
  console.log('doubling', x);

  return x * 2;
};
```

```
const result = impureDouble(4);



console.log({ result });
```

We hardly call this a side-effect because, in all practicality, it won't harm us. We'll still get the same outputs, given the same inputs.

This, however, may cause a problem

```
const impureAssoc = (key, value, object) => {
  object[key] = value;
};

const person = { name: 'Bobo' };
const result = impureAssoc('shoeSize', 400, person);

console.log({ person, result });
```

The variable, `person`, has been forever changed because our function introduced an assignment statement.

Shared state means `impureAssoc`'s impact isn't fully obvious anymore. Understanding its effect on a system now involves tracking down every variable it's ever touched and knowing their histories.

Shared state = timing dependencies.

We can purify `impureAssoc` by simply returning a new object with our desired properties.

```
const pureAssoc = (key, value, object) => ({
  ...object,
  [key]: value
});

const person = { name: 'Bobo' };
const result = pureAssoc('shoeSize', 400, person);

console.log({ person, result });
```
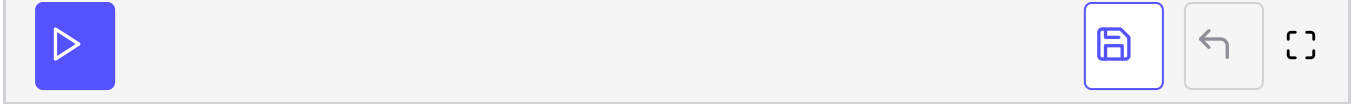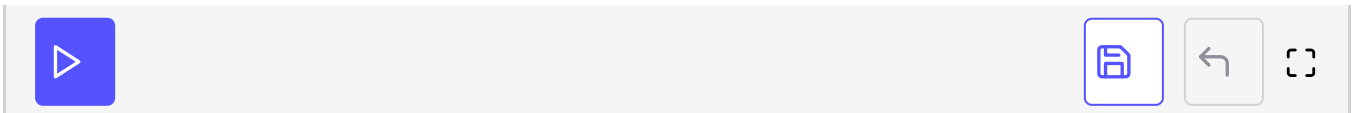
Now `pureAssoc` returns a testable result and we'll never worry if it quietly mutated something elsewhere.

You could even do the following and remain pure

```
const pureAssoc = (key, value, object) => {
  const newObject = { ...object };

  newObject[key] = value;

  return newObject;
};

const person = { name: 'Bobo' };
const result = pureAssoc('shoeSize', 400, person);

console.log({ person, result });
```

Mutating your input can be dangerous, but mutating a copy of it is no problem. Our end result is still a testable, predictable function that works no matter where/when you call it. The mutation's limited to that small scope and you're still returning a testable result.

Summary

- A function's pure if it's free from side-effects and returns the same output, given the same input.

- Side-effects include: mutating input, http calls, writing to disk, printing to the screen.

- You can safely mutate your input's *clone*, just leave the original one untouched.

- Spread syntax is the easiest way to clone objects and arrays.

```javascript
const person = { name: 'Bobo' };
const personClone = { ...person };

const people = ['Bobo', 'Barry', 'Bruce'];
const peopleClone = [...people];

personClone.name = 'Sam';
peopleClone[0] = 'Sam';

console.log({
  person,
  personClone,
  people,
  peopleClone
});
```