ACADEMIND

What Makes Up A Function?

```
add(5, 7);
```

```
function add(n1, n2) {
  return n1 + n2;
}
```

Calling the function should be readable

The number and order of arguments matter

Working with the function should be easy / readable

The length of the function body matters

ACADEMIND

What Makes Up A Function?

```
add(5, 7);
```

```
function add(n1, n2) {
  return n1 + n2;
}
```

Calling the function should be readable

The number and order of arguments matter

Working with the function should be easy / readable

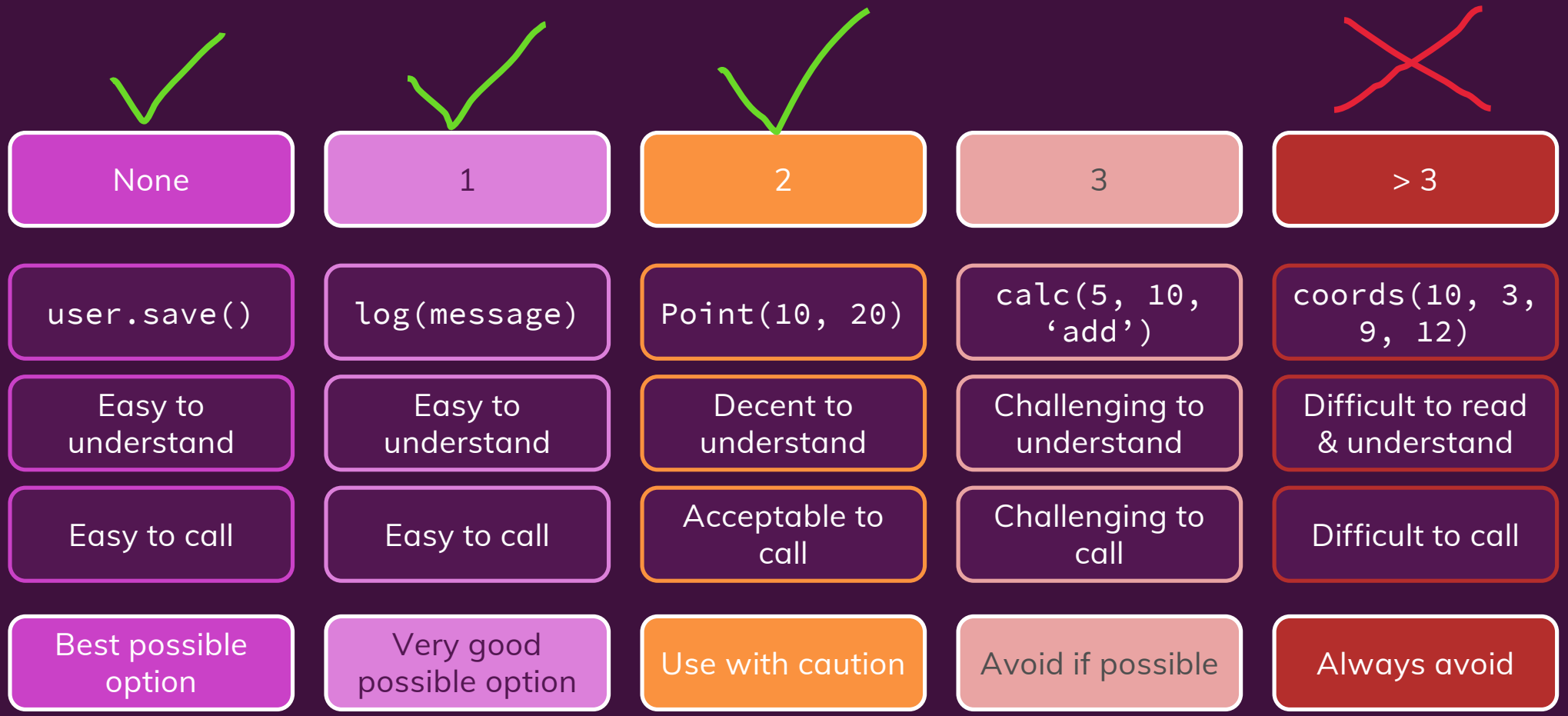
The length of the function body matters

ACADEMIND

Minimize the number of parameters

ACADEMIND

The Number Of Function / Method Parameters



None	1	2	3	> 3
`user.save()`	`log(message)`	`Point(10, 20)`	`calc(5, 10, 'add')`	`coords(10, 3, 9, 12)`
Easy to understand	Easy to understand	Decent to understand	Challenging to understand	Difficult to read & understand
Easy to call	Easy to call	Acceptable to call	Challenging to call	Difficult to call
Best possible option	Very good possible option	Use with caution	Avoid if possible	Always avoid

ACADEMIND

Output Parameters

Try to **avoid** output arguments – especially if they are **unexpected**

`createId(user)`

Not great – user gets modified in an unexpected way

`addId(user)`

Okay – user gets modified, but the function implies it

`user.addId()`

Great – it's obvious, that the user will get modified

ACADEMIND

What Makes Up A Function?

```
add(5, 7);
```

```
function add(n1, n2) {
  return n1 + n2;
}
```

Calling the function should be readable

The number and order of arguments matter

Working with the function should be easy / readable

The length of the function body matters

ACADEMIND

Functions Should Be Small

ACADEMIND

Functions Should Do Exactly One Thing

ACADEMIND

What Is “One Thing”?

ACADEMIND

"One Thing"

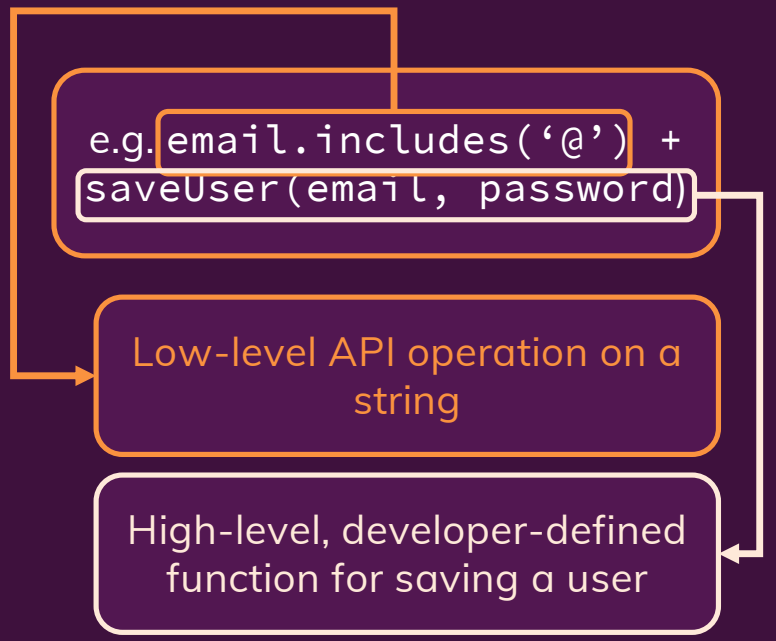
Different Operations

e.g. Validate + Save User Input

Operation 1 + Operation 2

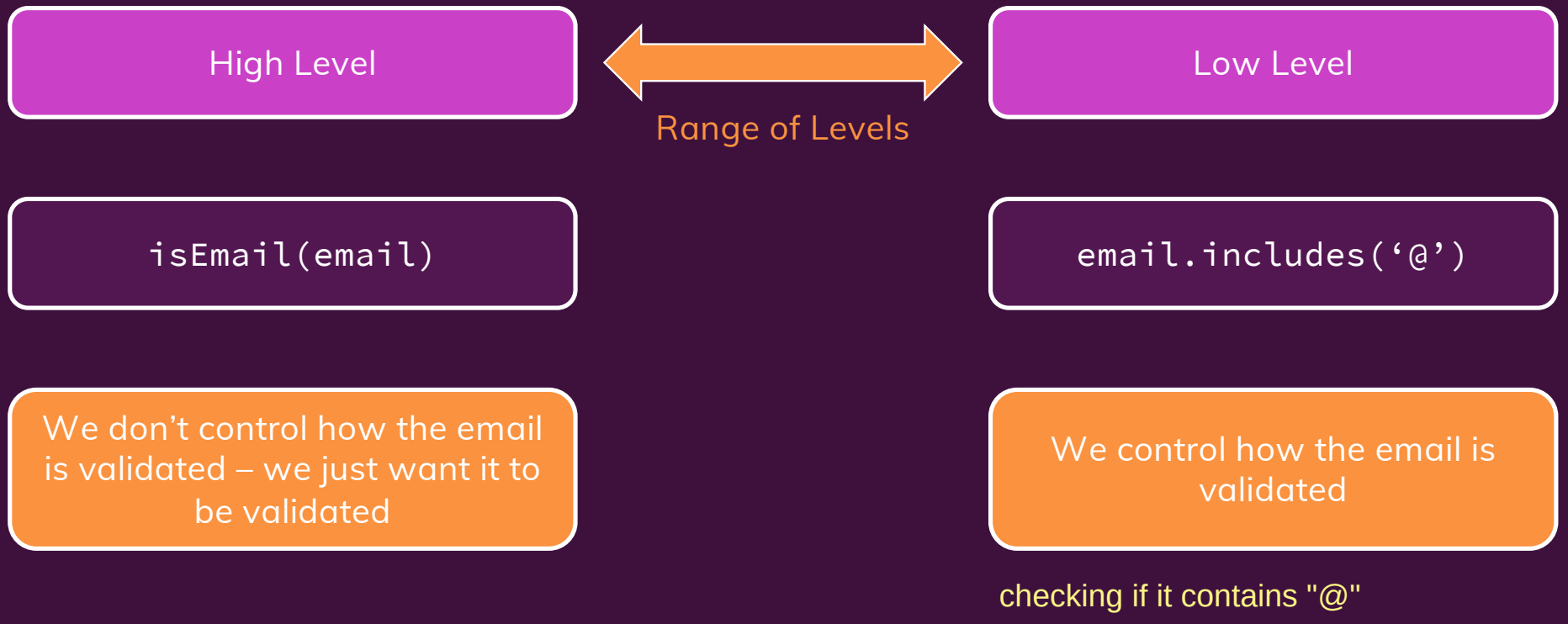
+

Different **Levels of Abstraction**



ACADEMIND

Understanding “Levels of Abstraction”



The Problem With Multiple Levels Of Abstraction

High Level

```
isEmail(email)
```

This is easy to read – there is no room for interpretation

clear that we are finding out if something is an email or not

Low Level

```
email.includes('@')
```

This might be technically clear, but the interpretation must be added by the reader

reader has to interpret that you are validating an email address

ACADEMIND

Functions & Abstraction

Functions should do work that's **one level of abstraction below their name**

```
function emailIsValid(email) {
  return email.includes('@');
}
```

low level code but u add a function so its easier to interpret by function name

This function should return yes/ no (true/false) based on the email validity

```
function saveUser(email) {
  if (email.includes('@')) { … }
  // …

}
```

This function should orchestrate all the steps that are required to save a user

easy to understand

ACADEMIND

Try Not To Mix Levels Of Abstraction

```
if (!email.includes('@')) {
  console.log('Invalid email!')
} else {
  const user = new User(email)
  user.save()
}
```

We need to read, understand and interpret the different steps

export lower level operations into other functions

```
if (!isEmail(email)) {
  showError('Invalid email!')
} else {
  saveNewUser(email)
}
```

We just need to read the different steps

easier to understand!

ACADEMIND

Keeping Functions Short

Rule of Thumb

Extract code that works on the same functionality

```
user.setAge(31)
user.setName('Max')
```

merge into a new method

```
user.update({age: 31, name: 'Max'})
```

Extract code that requires more interpretation than the surrounding code

```
if (!email.includes('@')) {…}
saveNewUser(email)
```

using functions instead

```
if (!isValid(email)) {…}
saveNewUser(email)
```

ACADEMIND

Reusability Matters *(Sometimes)*

ACADEMIND

Don’t Repeat Yourself (DRY)

DRY = “Don’t Repeat Yourself”

Don’t write the same code more than once

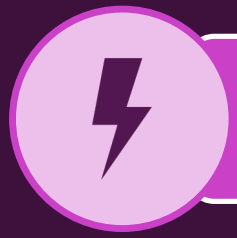
Signs of code which “is not DRY”

- You find yourself copy & pasting code
- You need to apply the same change to multiple places in your codebase

ACADEMIND

Use Common Sense

Opinion: Split Functions Reasonably



Being as **granular** as possible **won't automatically improve readability**

The opposite might be the case!

Make reasonable decisions and don't split if ...

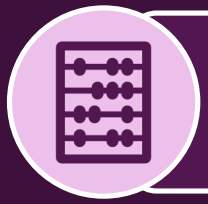
- ... you're just renaming the operation
- ... finding the new function will take longer than reading the extracted code
- ... can't produce a reasonable name for the extracted function

ACADEMIND

Try Keeping Functions Pure

Input
(Parameters)

Output
(Return value)



The same input always yields the same output

No side effects

ACADEMIND

What's a Side Effect?

```
function createUser(email, password) {
  const user = new User(email, password);
  startSession(user);
  return user;
}
```

could be an expected side effect

side effect

not a pure function with this statement!

could change something outside of this function

A **side effect** is an operation which does **not** just act on function inputs and change the function output but which instead **changes the overall system / program state**

Side effects are not automatically bad – we do need them in our programs. But **unexpected side effects should be avoided**. if unexpected, then it is bad

ACADEMIND

Avoid Unexpected Side Effects

Naming matters!

The name of a function should signal or imply that a side effect is likely to occur

`saveUser(…)`	`isValid(…)`	`showMessage(…)`	`createUser(…)`
Side effect expected	Side effect **not** expected	Side effect expected	Side effect **not necessarily** expected

saved user to database
affect overall state of user

showing a message changes
the program state

ACADEMIND

Handling Side Effects

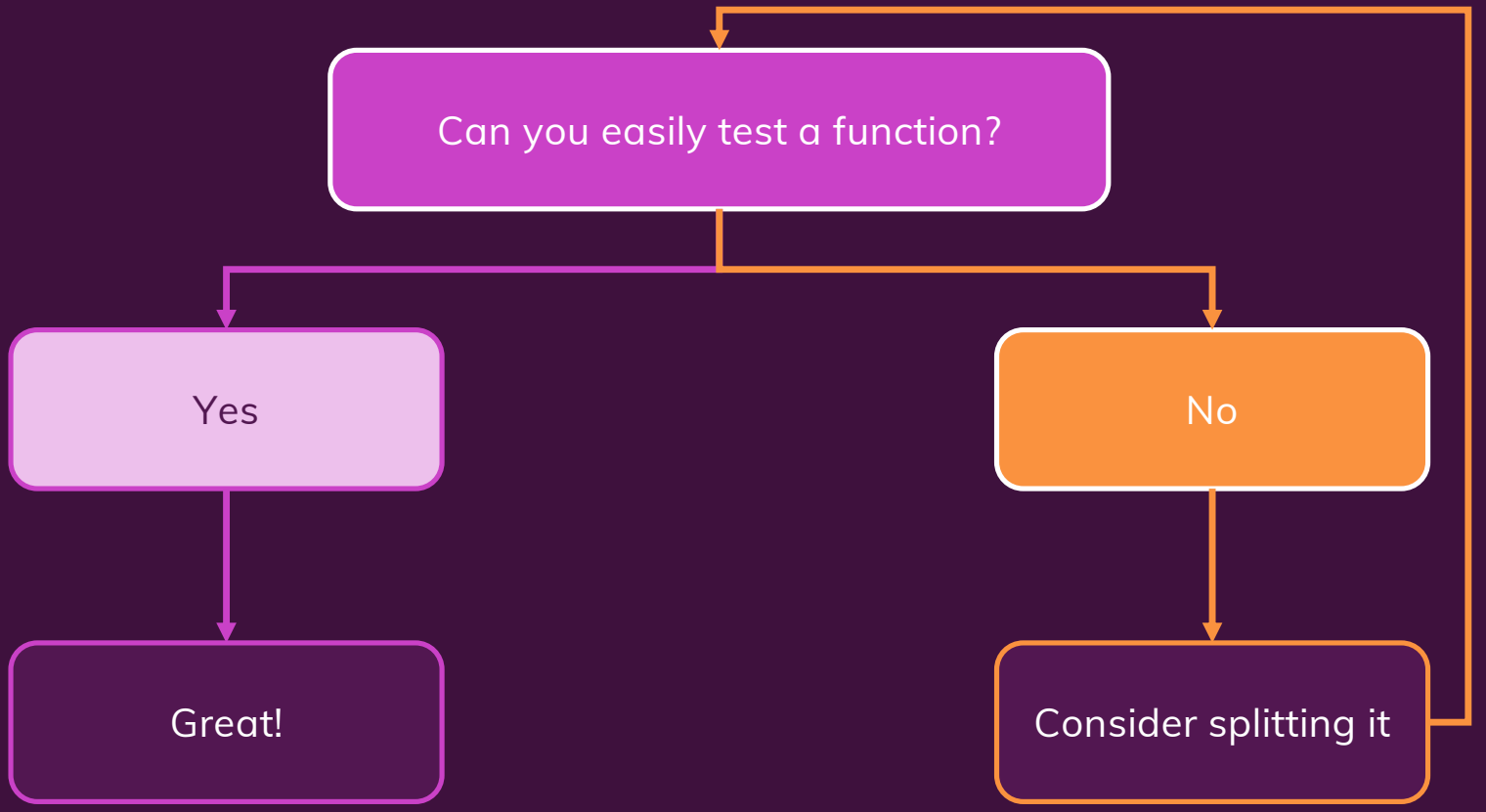
Your functions should **not** have any **unexpected side effects**

If you have / need a side effect

- Choose a function name which implies it
- Move the side effect into another function / place

ACADEMIND

Unit Testing Helps!



splitting up the functions
so it is easier to test