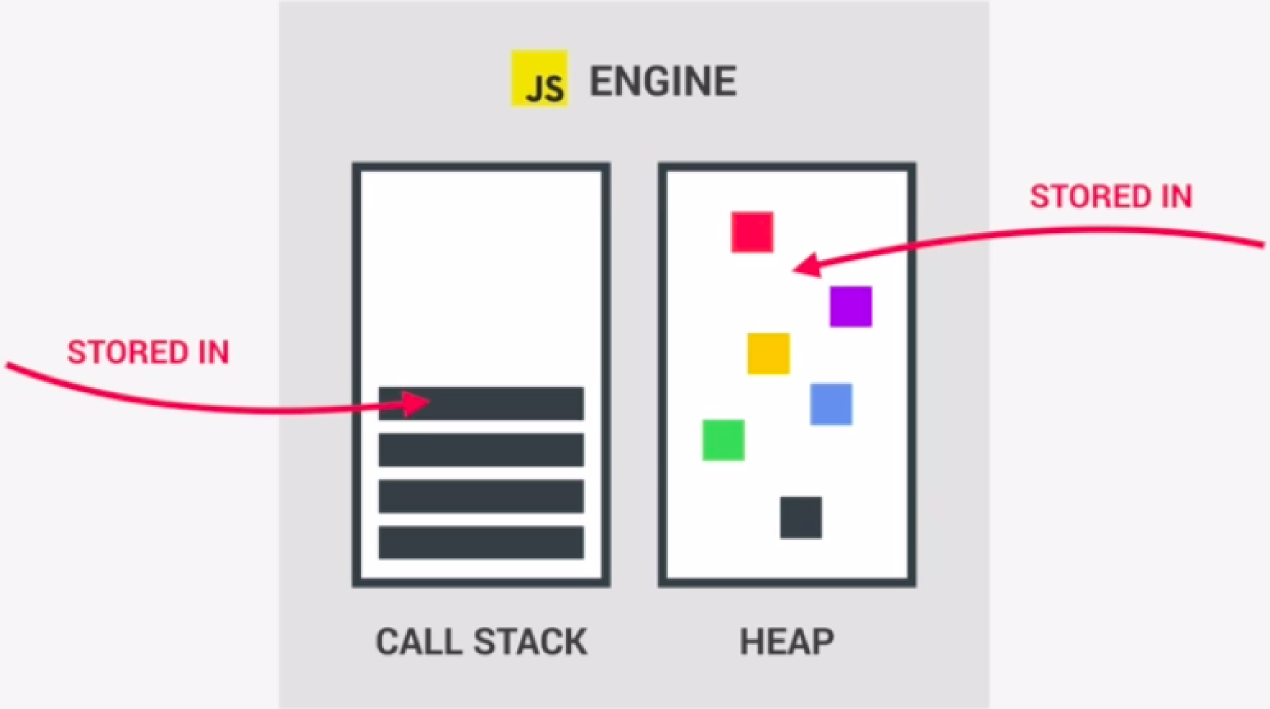
REVIEW: PRIMITIVES, OBJECTS AND THE JAVASCRIPT ENGINE

PRIMITIVES

- Number
- String
- Boolean
- Undefined
- Null
- Symbol
- BigInt

PRIMITIVE TYPES



OBJECTS

- Object literal
- Arrays
- Functions
- Many more...

REFERENCE TYPES

PRIMITIVE VS. REFERENCE VALUES

👉 **Primitive** values example:

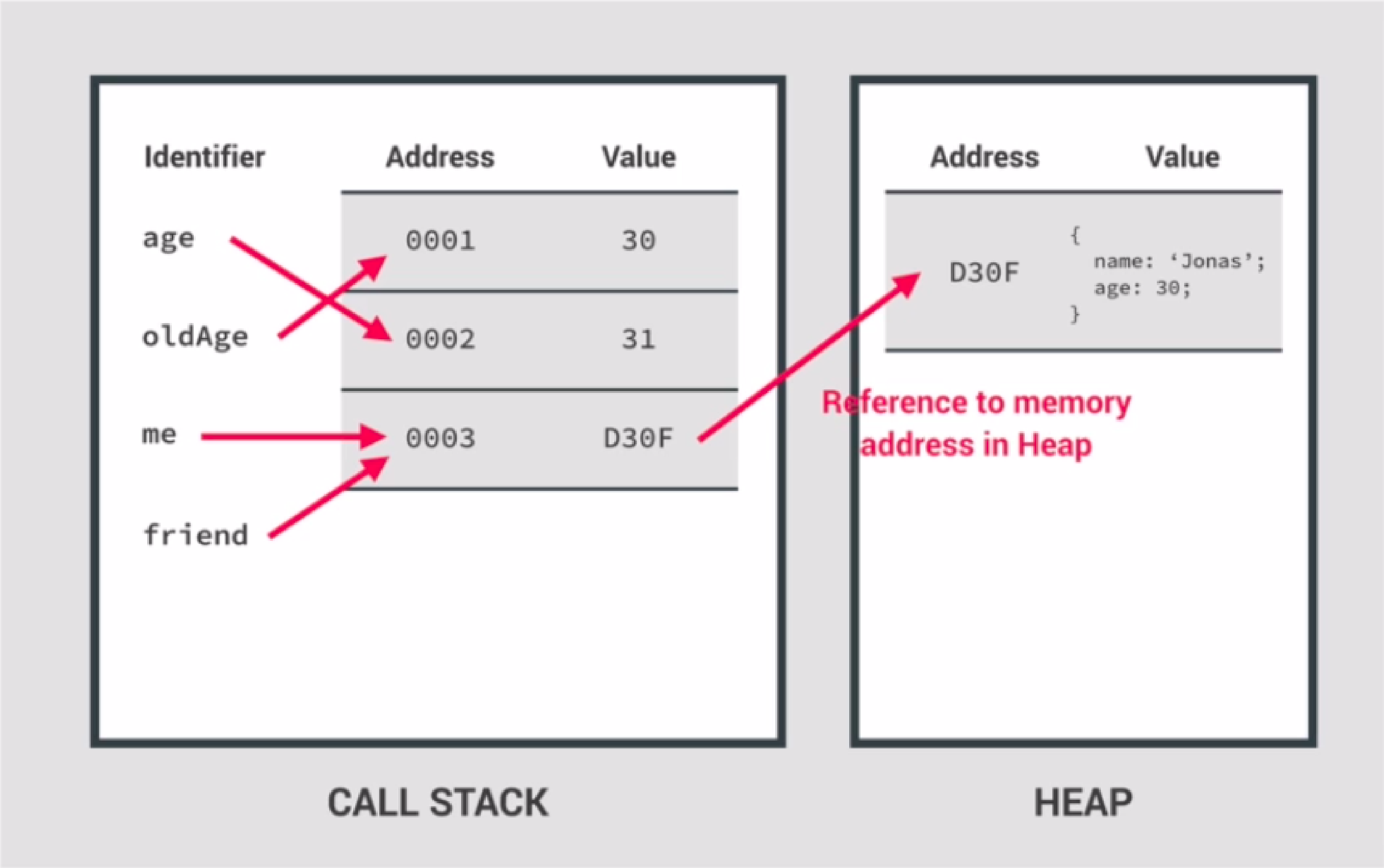
```
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30
```

👉 **Reference** values example:

```
const me = {
  name: 'Jonas',
  age: 30
};
const friend = me;
friend.age = 27;

console.log('Friend:', friend);
// { name: 'Jonas', age: 27 }

console.log('Me:', me);
// { name: 'Jonas', age: 27 }
```



Identifier	Address	Value
age	0001	30
oldAge	0002	31
me	0003	D30F
friend		

Address	Value

D30F	{
	name: 'Jonas';
	age: 30;
	}

Reference to memory address in Heap

CALL STACK

HEAP

PRIMITIVE VS. REFERENCE VALUES

- **Primitive** values example:

```
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30
```

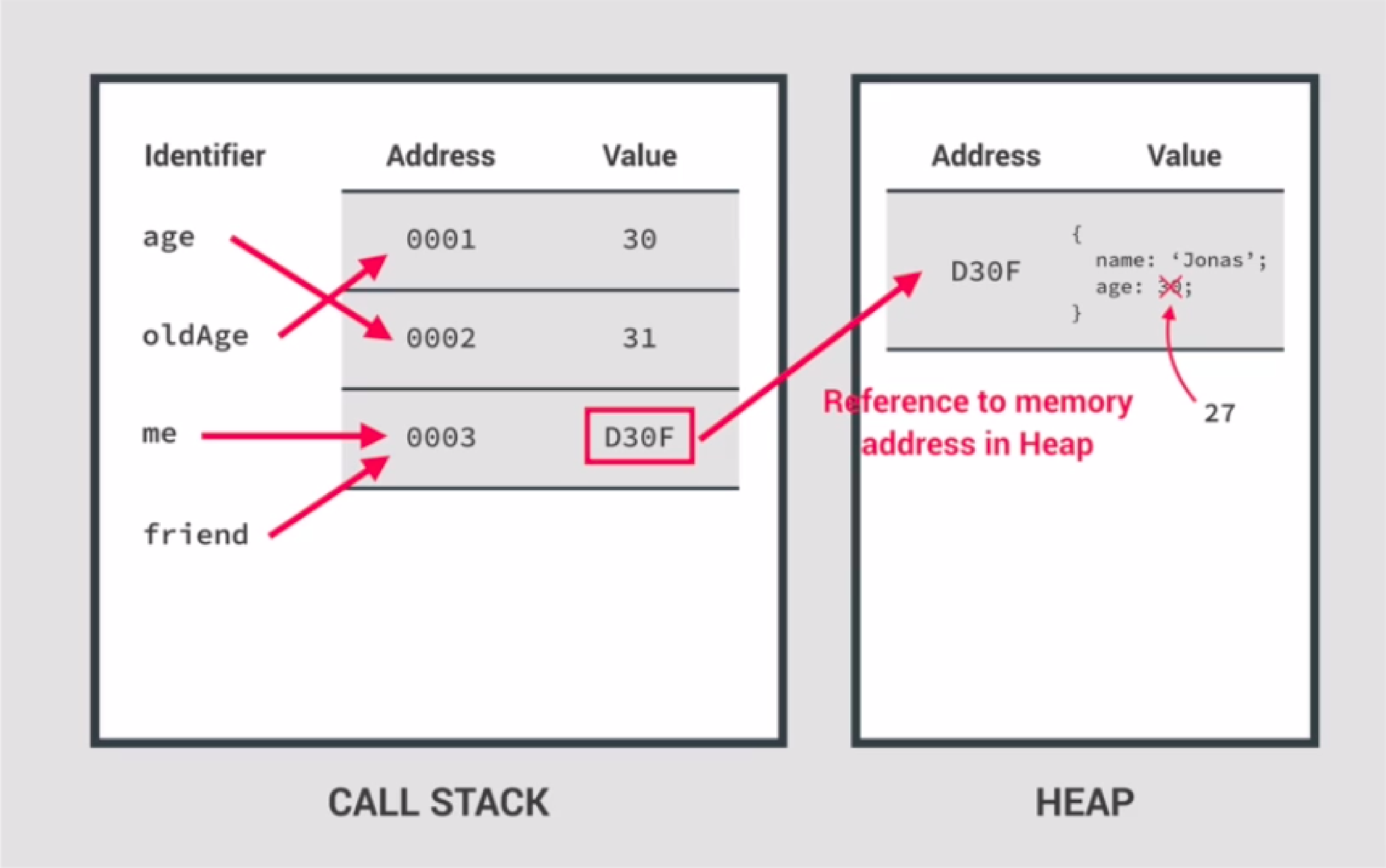
- **Reference** values example:

```
const me = {
  name: 'Jonas'
  age: 30
};
const friend = me;
friend.age = 27;

console.log('Friend:', friend);
// { name: 'Jonas', age: 27 }

console.log('Me:', me);
// { name: 'Jonas', age: 27 }
```

No problem, because we're NOT changing the *value* at address 0003!



Identifier	Address	Value
age	0001	30
oldAge	0002	31
me	0003	D30F
friend		

CALL STACK

Address	Value
D30F	{ name: 'Jonas'; age: ~~30~~; }

Reference to memory address in Heap

27

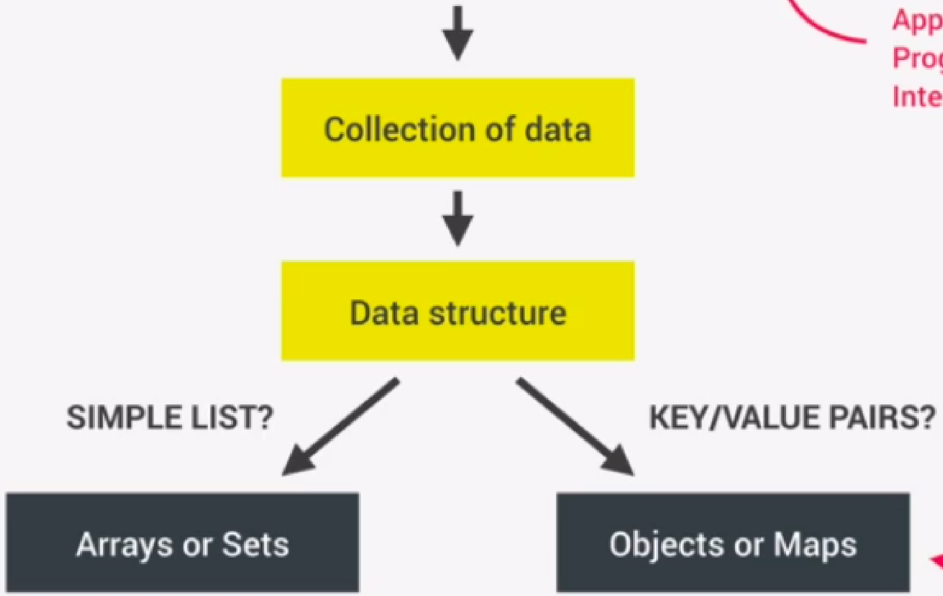
HEAP

DATA STRUCTURES OVERVIEW

SOURCES OF DATA

1. **From the program itself:** Data written directly in source code (e.g. status messages)
2. **From the UI:** Data input from the user or data written in DOM (e.g tasks in todo app)
3. **From external sources:** Data fetched for example from web API (e.g. recipe objects)

Application Programming Interface

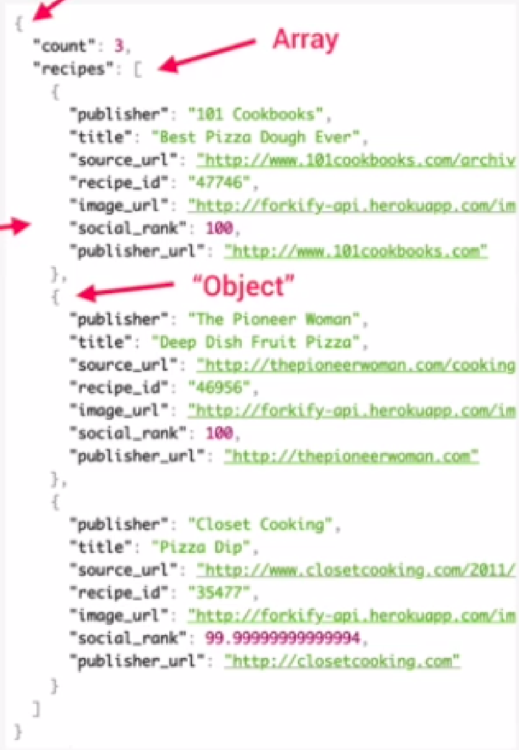


Keys allow us to *describe* values

“Object”

Array

“Object”



```json
{
  "count": 3,
  "recipes": [
    {
      "publisher": "101 Cookbooks",
      "title": "Best Pizza Dough Ever",
      "source_url": "http://www.101cookbooks.com/archiv
      "recipe_id": "47746",
      "image_url": "http://forkify-api.herokuapp.com/im
      "social_rank": 100,
      "publisher_url": "http://www.101cookbooks.com"
    },
    {
      "publisher": "The Pioneer Woman",
      "title": "Deep Dish Fruit Pizza",
      "source_url": "http://thepioneerwoman.com/cooking
      "recipe_id": "46956",
      "image_url": "http://forkify-api.herokuapp.com/im
      "social_rank": 100,
      "publisher_url": "http://thepioneerwoman.com"
    },
    {
      "publisher": "Closet Cooking",
      "title": "Pizza Dip",
      "source_url": "http://www.closetcooking.com/2011/
      "recipe_id": "35477",
      "image_url": "http://forkify-api.herokuapp.com/im
      "social_rank": 99.99999999999994,
      "publisher_url": "http://closetcooking.com"
    }
  ]
}
```

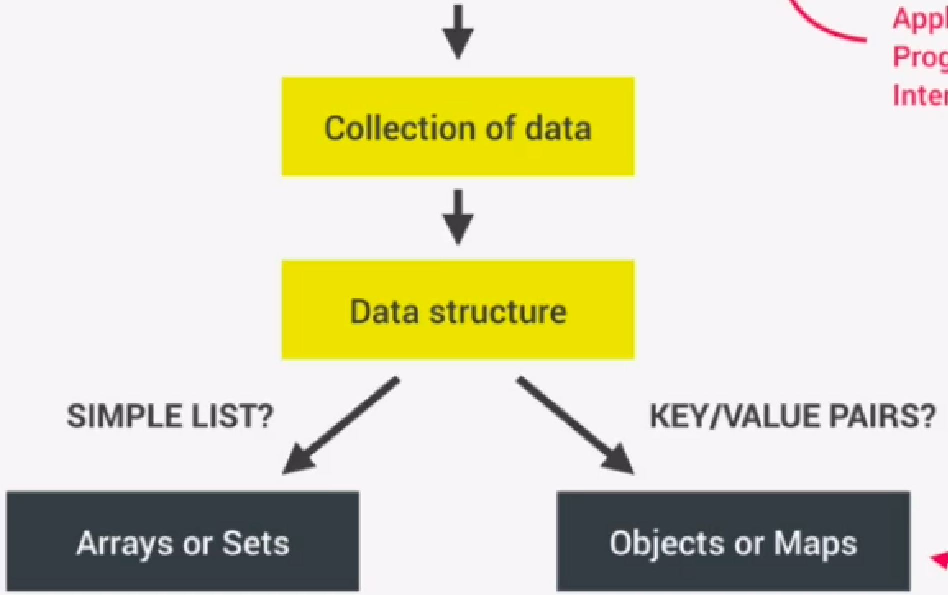
👉 JSON data format example

DATA STRUCTURES OVERVIEW

SOURCES OF DATA

1. **From the program itself:** Data written directly in source code (e.g. status messages)
2. **From the UI:** Data input from the user or data written in DOM (e.g tasks in todo app)
3. **From external sources:** Data fetched for example from web API (e.g. recipe objects)

Application Programming Interface



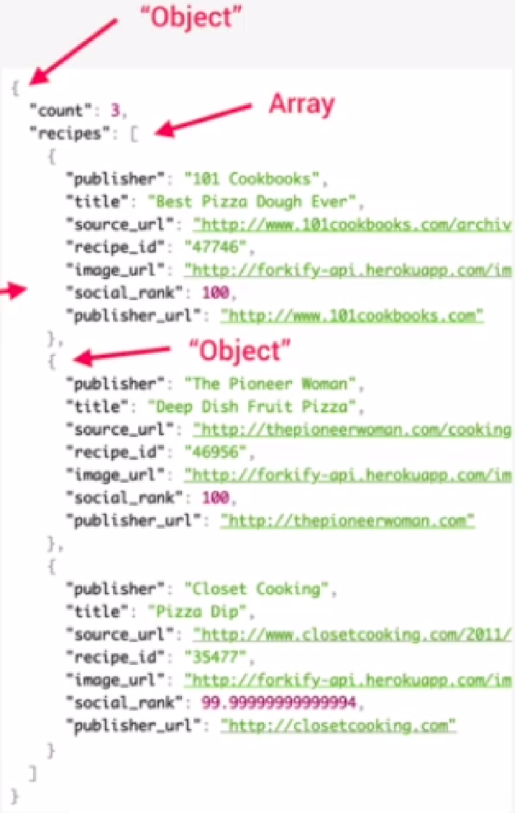
Keys allow us to *describe* values

OTHER BUILT-IN:

- WeakMap
- WeakSet

NON-BUILT IN:

- Stacks
- Queues
- Linked lists
- Trees
- Hash tables



```
{
  "count": 3,
  "recipes": [
    {
      "publisher": "101 Cookbooks",
      "title": "Best Pizza Dough Ever",
      "source_url": "http://www.101cookbooks.com/archiv
      "recipe_id": "47746",
      "image_url": "http://forkify-api.herokuapp.com/im
      "social_rank": 100,
      "publisher_url": "http://www.101cookbooks.com"
    },
    {
      "publisher": "The Pioneer Woman",
      "title": "Deep Dish Fruit Pizza",
      "source_url": "http://thepioneerwoman.com/cooking
      "recipe_id": "46956",
      "image_url": "http://forkify-api.herokuapp.com/im
      "social_rank": 100,
      "publisher_url": "http://thepioneerwoman.com"
    },
    {
      "publisher": "Closet Cooking",
      "title": "Pizza Dip",
      "source_url": "http://www.closetcooking.com/2011/
      "recipe_id": "35477",
      "image_url": "http://forkify-api.herokuapp.com/im
      "social_rank": 99.99999999999994,
      "publisher_url": "http://closetcooking.com"
    }
  ]
}
```

- JSON data format example

ARRAYS VS. SETS AND OBJECTS VS. MAPS

ARRAYS VS. SETS

ARRAYS

```
tasks = ['Code', 'Eat', 'Code'];
// ["Code", "Eat", "Code"]
```

- Use when you need **ordered** list of values (might contain duplicates)
- Use when you need to **manipulate** data

SETS

```
tasks = new Set(['Code', 'Eat', 'Code']);
// {"Code", "Eat"}
```

- Use when you need to work with **unique** values
- Use when **high-performance** is *really* important
- Use to **remove duplicates** from arrays

OBJECTS VS. MAPS

OBJECTS

```
task = {
  task: 'Code',
  date: 'today',
  repeat: true
};
```

- More "traditional" key/value store ("abused" objects)
- Easier to write and access values with `.` and `[]`

- Use when you need to include **functions** (methods)
- Use when working with JSON (can convert to map)

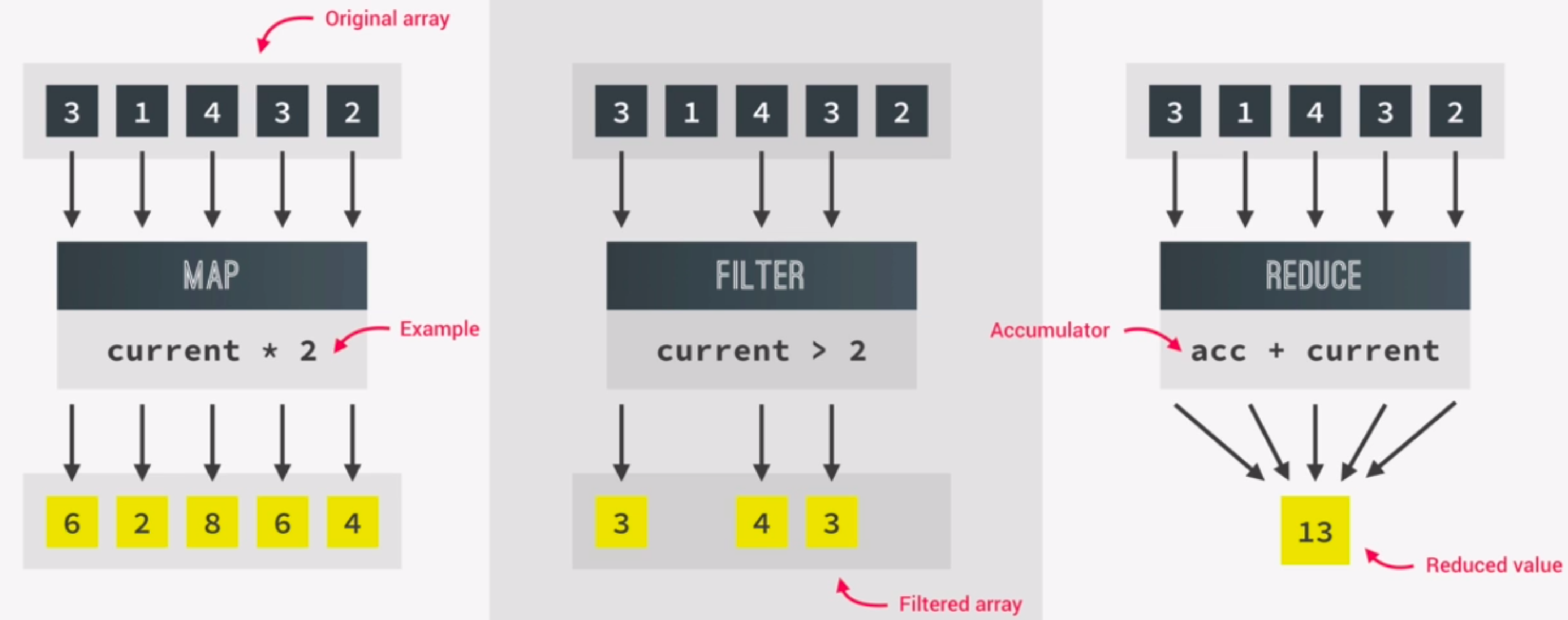
MAPS

```
task = new Map([
  ['task', 'Code'],
  ['date', 'today'],
  [false, 'Start coding!']
]);
```

- Better performance
- Keys can have **any** data type
- Easy to iterate
- Easy to compute size

- Use when you simply need to map key to values
- Use when you need keys that are **not** strings

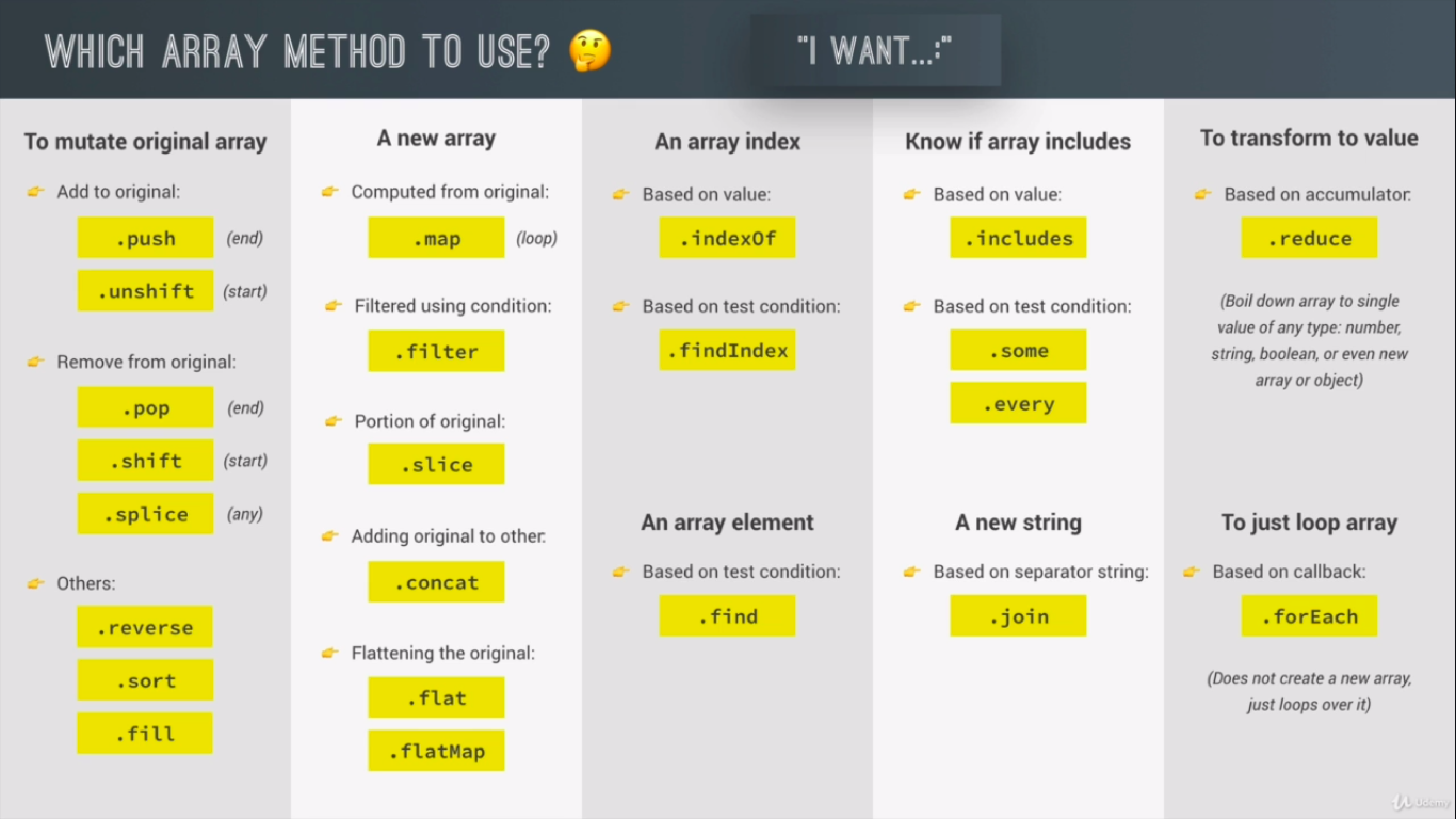
DATA TRANSFORMATIONS WITH MAP, FILTER AND REDUCE



- `map` returns a **new array** containing the results of applying an operation on all original array elements

- `filter` returns a **new array** containing the array elements that passed a specified **test condition**

- `reduce` boils ("reduces") all array elements down to one single value (e.g. adding all elements together)



WHICH ARRAY METHOD TO USE? 🤔

"I WANT...:"

To mutate original array

- Add to original:
  - `.push` *(end)*
  - `.unshift` *(start)*
- Remove from original:
  - `.pop` *(end)*
  - `.shift` *(start)*
  - `.splice` *(any)*
- Others:
  - `.reverse`
  - `.sort`
  - `.fill`

A new array

- Computed from original:
  - `.map` *(loop)*
- Filtered using condition:
  - `.filter`
- Portion of original:
  - `.slice`
- Adding original to other:
  - `.concat`
- Flattening the original:
  - `.flat`
  - `.flatMap`

An array index

- Based on value:
  - `.indexOf`
- Based on test condition:
  - `.findIndex`

An array element

- Based on test condition:
  - `.find`

Know if array includes

- Based on value:
  - `.includes`
- Based on test condition:
  - `.some`
  - `.every`

A new string

- Based on separator string:
  - `.join`

To transform to value

- Based on accumulator:
  - `.reduce`

(Boil down array to single value of any type: number, string, boolean, or even new array or object)

To just loop array

- Based on callback:
  - `.forEach`

(Does not create a new array, just loops over it)