Lodash library

Outline

LoDash overview

A modern JavaScript utility library delivering modularity, performance, & extras

Installation: npm install –save lodash

- Node.js consist of huge community of developers contributing tons of package to the NPM repository.
- Lodash is one such library which is successor of underscore.js.
- It is used to simplify your work of managing and editing objects and arrays by providing lots of utility methods to do so.

Lodash makes JavaScript easier by taking the hassle out of working with arrays, numbers, objects, strings, etc.
Lodash's modular methods are great for:

- Iterating Arrays, Objects, Functions & Strings
- Manipulating & testing values
- Creating composite functions

LoDash Overview

Module formats

Lodash is available in a variety of builds & module formats.
lodash & per method packages
lodash-es, babel-plugin-lodash, & lodash-webpack-plugin
lodash/fp
lodash-amd

JavaScript's available module specifications:

- CommonJS
- AMD: Asynchronous Module Definition
- UMD: Universal Module Definition

Read this article

Access to cdnjs.com

Web Installation:

```
<script src="lodash.js"></script>
<script
src="//cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js">
</script>
```

NodeJS: Installation

```
$ npm install --save lodash
$ npm install --save lodash@4.17.11
```

Use of lodash in NodeJS

```
var _ = require('lodash'); // Require the whole lodash package
var forEach = require('lodash/forEach'); // Require only forEach
```

loDash official documentation

Lo

Lodash 4 cookbook

You can download the Free Sample

Basic Concepts

Truthy and falsy values are very important when using lodash predicates.
false, 0, ""(empty string), null, undefined and NaN are falsy values in JavaScript.
All other values are truthy values.

SameValueZero algorithm of how to compare two values in lodash.
It's similar to JavaScript "strict equality comparison" (===), except the handling of NaN.
It always make developers confused as NaN === NaN returns false.
SameValueZero removes that confusion, so NaN is same to NaN in SameValueZero algorithm.

Predicates

Predicate functions only return truthy or falsy values.

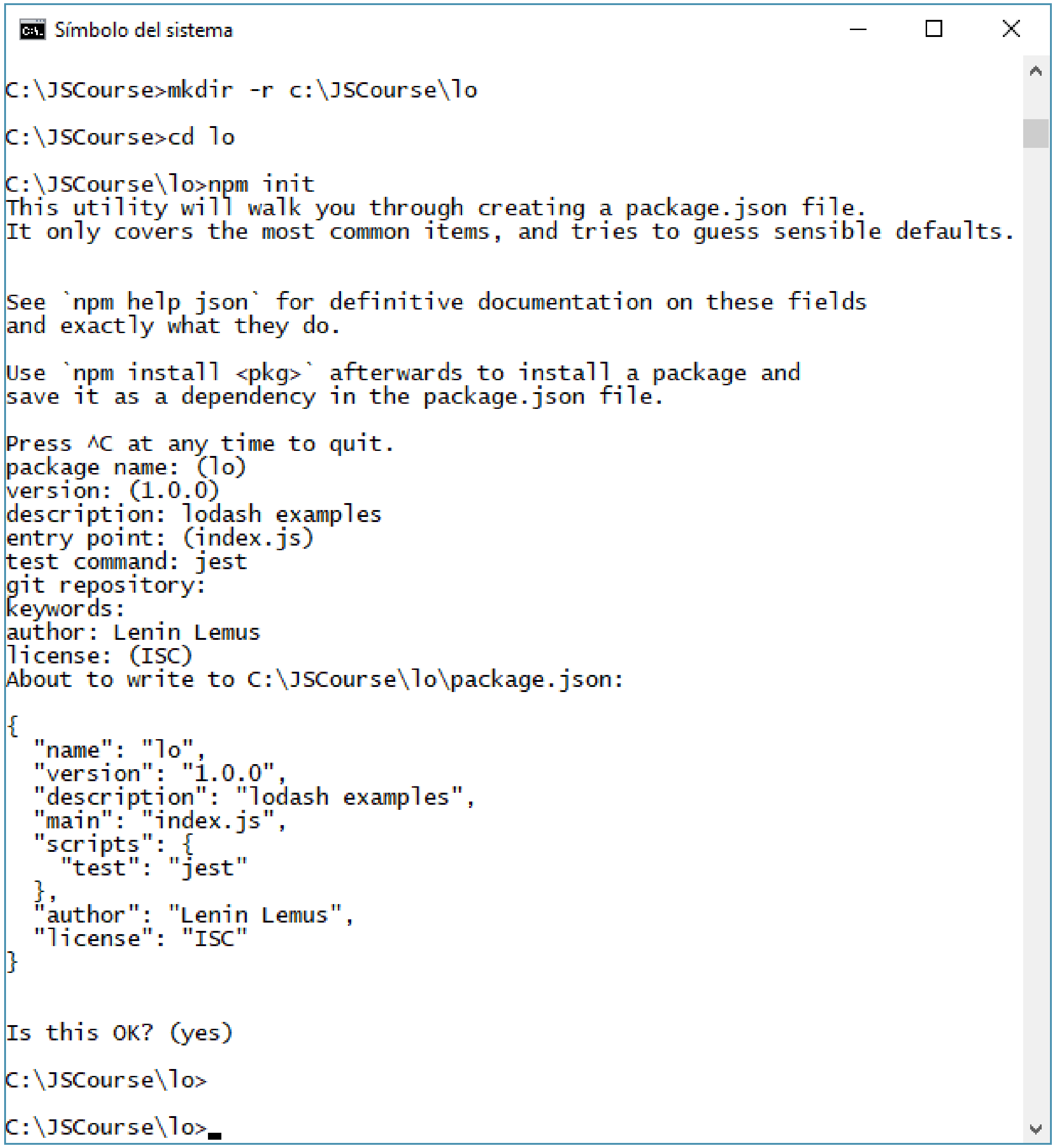
- Predicate functions can be written as plain old JavaScript functions
- Lodash also provides some helper functions to generate predicate functions for common use cases.

For example, when filtering a collection, a predicate function is required to determine what kind of elements should be kept.

Preparation:

1. Create the folder c:\JSCourse\lo
   mkdir –r c:\JSCourse\lo
2. Go inside c:\JSCourse\lo
   cd c:\JSCourse\lo
3. Execute the command: npm init
4. npm install jest lodash
5. mkdir __tests__
6. Open the folder c:\JSCourse\lo with the Visual Studio Code
7. From the editor create the file exercise01.js inside the folder __test__

Preparation



```
C:\JSCourse>mkdir -r c:\JSCourse\lo

C:\JSCourse>cd lo

C:\JSCourse\lo>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.


See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (lo)
version: (1.0.0)
description: lodash examples
entry point: (index.js)
test command: jest
git repository:
keywords:
author: Lenin Lemus
license: (ISC)
About to write to C:\JSCourse\lo\package.json:

{
  "name": "lo",
  "version": "1.0.0",
  "description": "lodash examples",
  "main": "index.js",
  "scripts": {
    "test": "jest"
  },
  "author": "Lenin Lemus",
  "license": "ISC"
}


Is this OK? (yes)

C:\JSCourse\lo>

C:\JSCourse\lo>
```

Installation

Sample Book

Basic concepts

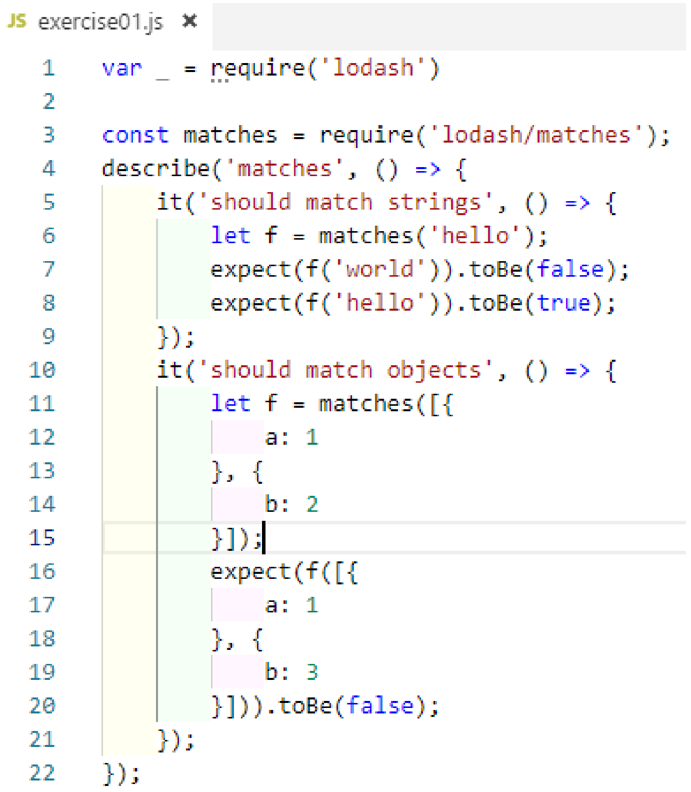
Predicates

- Matches
- Iteratees
- this binding
- Collections

Recipes

Matches

_.matches(source) takes a source object and creates a new function which performs a deep comparison between the given object and the source object.
_.matches supports comparison of different types of data, including booleans, numbers, strings, Date objects, RegExp objects, Object objects and arrays.



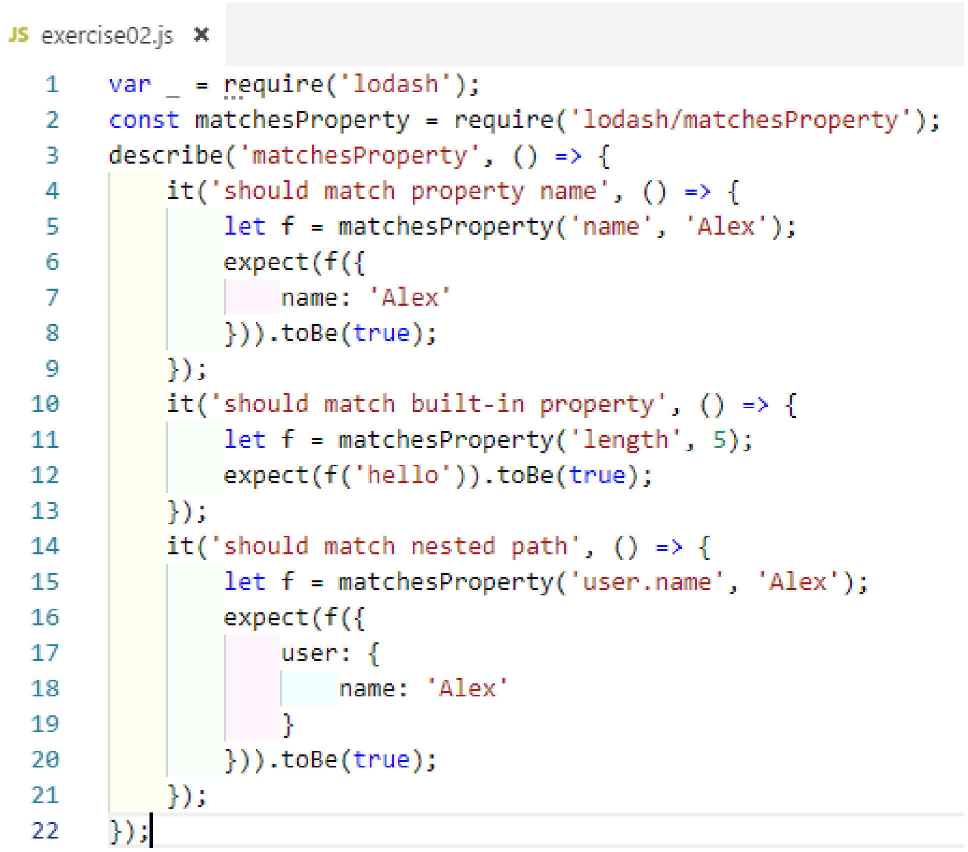
exercise01.js

```
var _ = require('lodash')

const matches = require('lodash/matches');
describe('matches', () => {
    it('should match strings', () => {
        let f = matches('hello');
        expect(f('world')).toBe(false);
        expect(f('hello')).toBe(true);
    });
    it('should match objects', () => {
        let f = matches([{
            a: 1
        }, {
            b: 2
        }]);
        expect(f([{
            a: 1
        }, {
            b: 3
        }])).toBe(false);
    });
});
```

Lo

Matches: matchesProperty

_.matchesProperty(path, value) takes a property path and the expected value of this property path to create a new function that checks if the given object's value of the same property path matches the expected value. Listing 2.2 shows how _.matchesProperty works by matching simple property name, built-in property and nested property path.



exercise02.js

```
var _ = require('lodash');
const matchesProperty = require('lodash/matchesProperty');
describe('matchesProperty', () => {
    it('should match property name', () => {
        let f = matchesProperty('name', 'Alex');
        expect(f({
            name: 'Alex'
        })).toBe(true);
    });
    it('should match built-in property', () => {
        let f = matchesProperty('length', 5);
        expect(f('hello')).toBe(true);
    });
    it('should match nested path', () => {
        let f = matchesProperty('user.name', 'Alex');
        expect(f({
            user: {
                name: 'Alex'
            }
        })).toBe(true);
    });
});
```

Matches: matchesProperty (cont.)

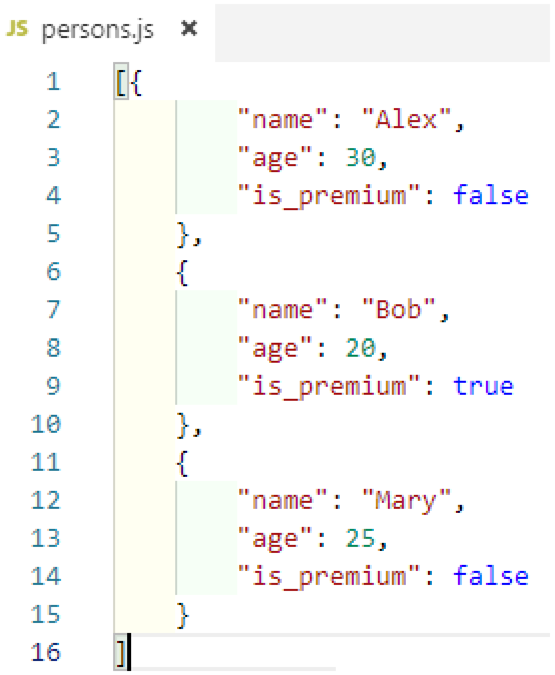
For lodash functions which accept predicates, e.g. _.find and _.filter, predicates can be specified using functions, strings, and objects.

- If a function is provided, it’s used directly.
  The predicate matches if the function returns a truthy value.
- If only a string is provided, it’s used to create a function using _.property as the predicate.
- If an array that contains a string and a value is provided, the string and the value are used to create a function using _matchesProperty as the predicate.
- If an object is provided, it’s used to create a function using _.matches as the predicate.

Lo

Matches: matchesProperty (cont.)

For example, given the array shown below

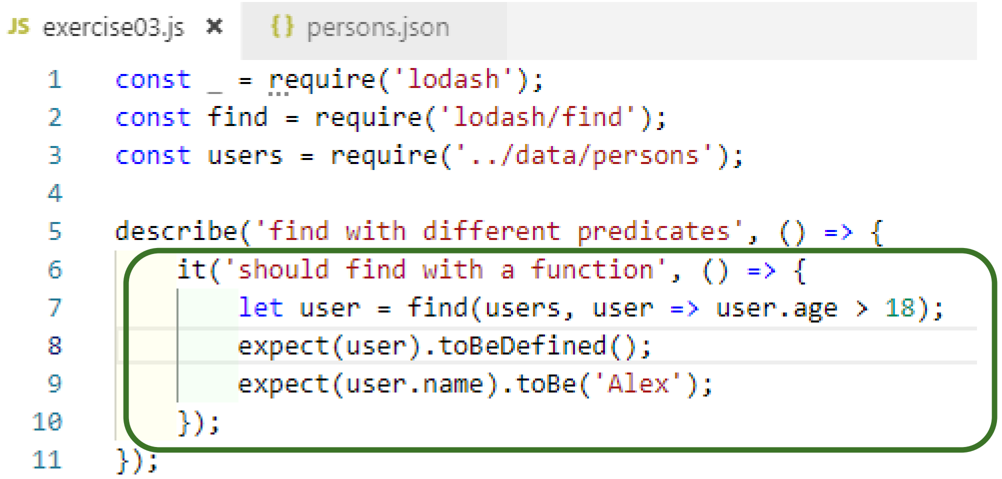

persons.js

```
[{
        "name": "Alex",
        "age": 30,
        "is_premium": false
    },
    {
        "name": "Bob",
        "age": 20,
        "is_premium": true
    },
    {
        "name": "Mary",
        "age": 25,
        "is_premium": false
    }
]
```

_.find returns the first matching element in the array.
A JavaScript function can be used as the predicate to _.find.

Next program finds the first element with age greater than 18 in the array users.

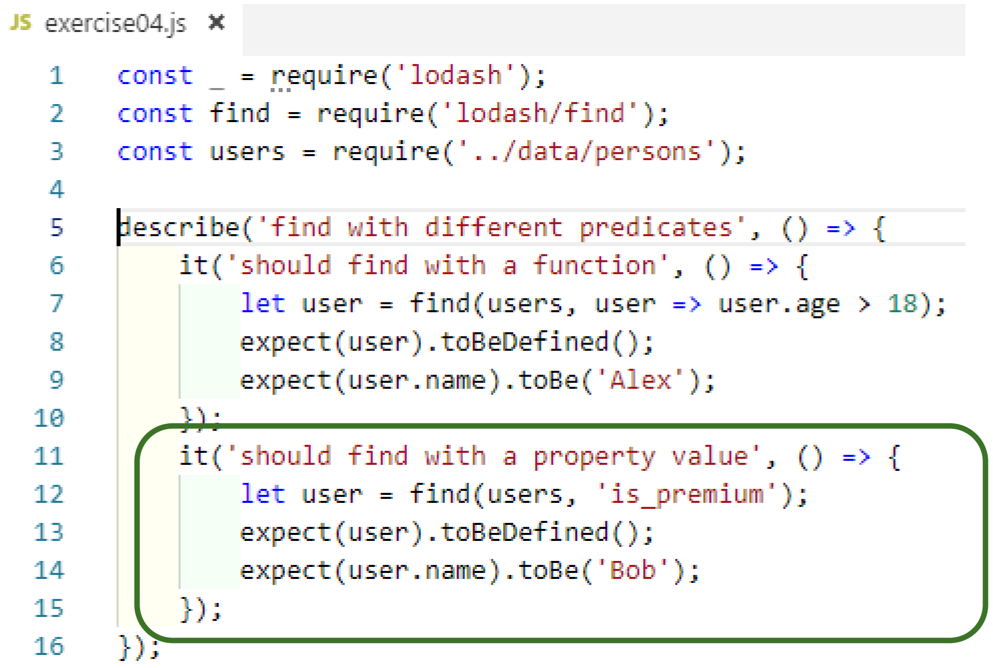
The result is the first element with name Alex.


exercise03.js persons.json

```
const _ = require('lodash');
const find = require('lodash/find');
const users = require('../data/persons');

describe('find with different predicates', () => {
    it('should find with a function', () => {
        let user = find(users, user => user.age > 18);
        expect(user).toBeDefined();
        expect(user.name).toBe('Alex');
    });
});
```

Matches: matchesProperty (cont.)

- If a string is passed as the predicate, it's treated as a property name of objects in the array.
- Next program finds the first element with truthy value of the property is_premium in the array.
- The actual used predicate is _.property('is_premium').
- The result is the second element with name Bob.



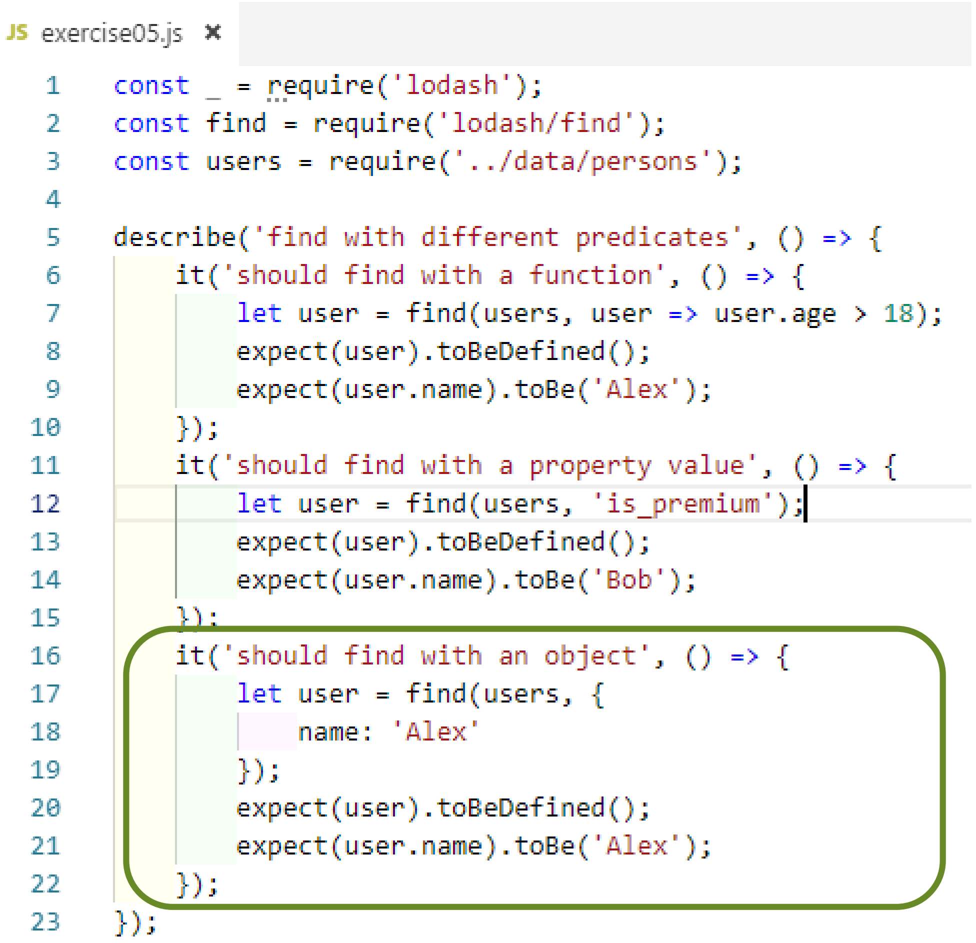
```javascript
const _ = require('lodash');
const find = require('lodash/find');
const users = require('../data/persons');

describe('find with different predicates', () => {
    it('should find with a function', () => {
        let user = find(users, user => user.age > 18);
        expect(user).toBeDefined();
        expect(user.name).toBe('Alex');
    });
    it('should find with a property value', () => {
        let user = find(users, 'is_premium');
        expect(user).toBeDefined();
        expect(user.name).toBe('Bob');
    });
});
```

Lo

Matches: matchesProperty (cont.)

- If an object is passed as the predicate, it’s treated as a search example
- Objects in returned results must have exactly the same values for all the corresponding properties provided in the search example

Next code finds the first element with the value of the property name equals to Alex in the array.
The actual used predicate is _.matches({ name: 'Alex' }).

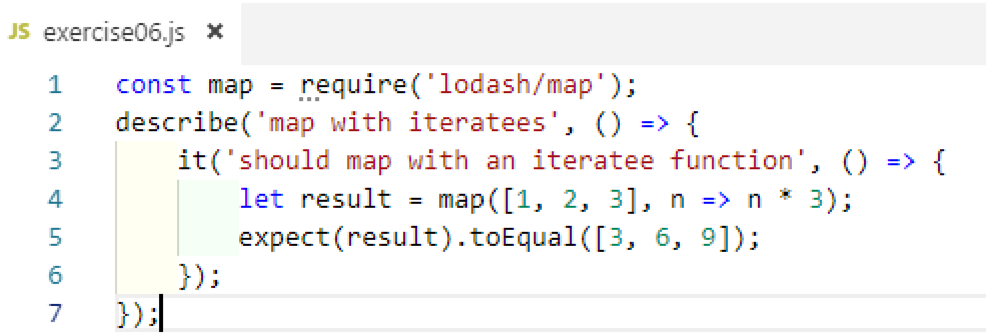

exercise05.js

```
const _ = require('lodash');
const find = require('lodash/find');
const users = require('../data/persons');

describe('find with different predicates', () => {
    it('should find with a function', () => {
        let user = find(users, user => user.age > 18);
        expect(user).toBeDefined();
        expect(user.name).toBe('Alex');
    });
    it('should find with a property value', () => {
        let user = find(users, 'is_premium');
        expect(user).toBeDefined();
        expect(user.name).toBe('Bob');
    });
    it('should find with an object', () => {
        let user = find(users, {
            name: 'Alex'
        });
        expect(user).toBeDefined();
        expect(user.name).toBe('Alex');
    });
});
```

Iteratees

- Iteratees are used by lodash functions which require iterating through a collection.
- Iteratee is invoked for each element in the collection and the result is used instead of the original element.
- Iteratees are typically used to transform collections.
    - A typical usage of iteratee is in the function _.map.
    - The second argument of _.map is the iteratee.
    - The result of applying iteratee to each element in the collection is collected and returned.

Next code use a function to transform input array [1, 2, 3] to [3, 6, 9].

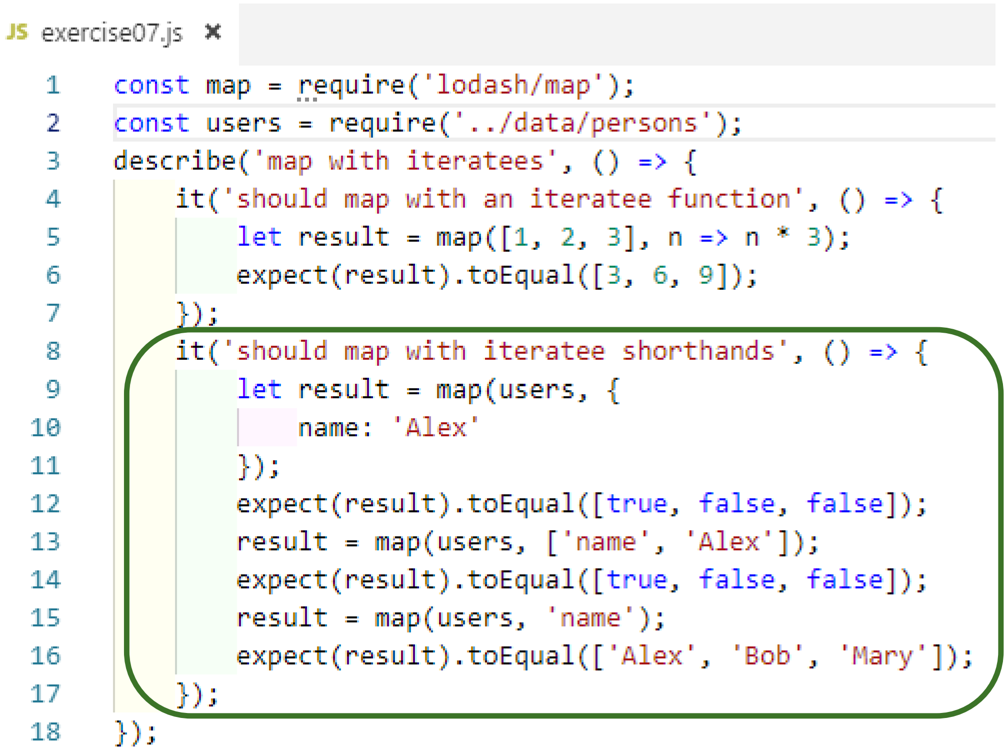


```
const map = require('lodash/map');
describe('map with iteratees', () => {
    it('should map with an iteratee function', () => {
        let result = map([1, 2, 3], n => n * 3);
        expect(result).toEqual([3, 6, 9]);
    });
});
```

Iteratee shorthand

- When iteratee functions are required, we can also use the similar syntax as predicate functions to quickly create them.
- These iteratee shorthands use methods _.matches, .matchesProperty or _.property behind the scene.

In the second invocation of _.map in code shown below, the second argument of _.map must be an array to indicate that it uses _.matchesProperty.



exercise07.js

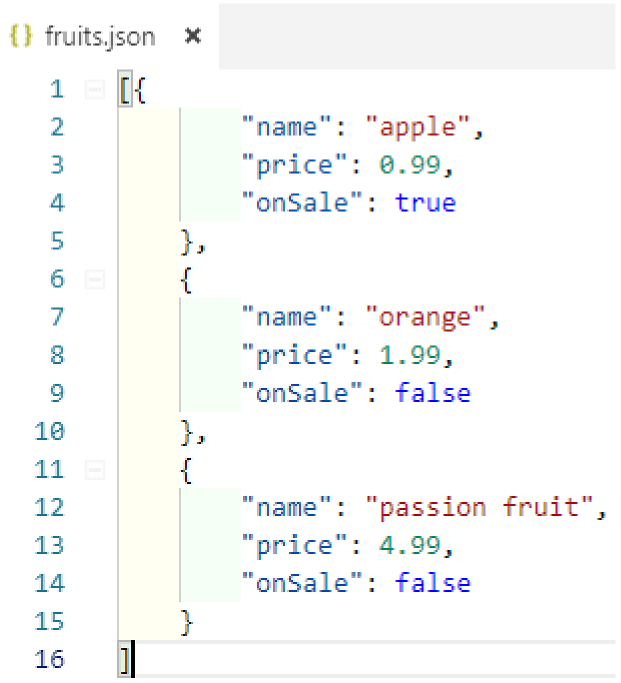
```
const map = require('lodash/map');
const users = require('../data/persons');
describe('map with iteratees', () => {
    it('should map with an iteratee function', () => {
        let result = map([1, 2, 3], n => n * 3);
        expect(result).toEqual([3, 6, 9]);
    });
    it('should map with iteratee shorthands', () => {
        let result = map(users, {
            name: 'Alex'
        });
        expect(result).toEqual([true, false, false]);
        result = map(users, ['name', 'Alex']);
        expect(result).toEqual([true, false, false]);
        result = map(users, 'name');
        expect(result).toEqual(['Alex', 'Bob', 'Mary']);
    });
});
```

Lo

Collections

A collection is an object that contains iterable elements.

- In lodash, collections can be arrays, objects, and strings.
- Lodash has a rich set of functions to work with collections.
- We use the following JSON array as the sample data fruits for some code samples

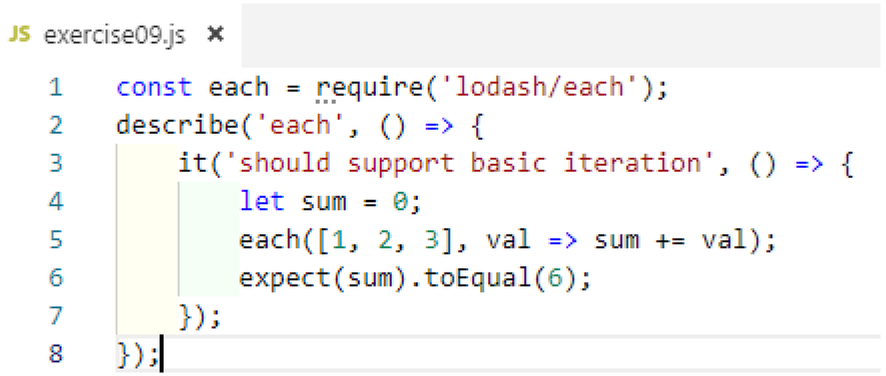

fruits.json

```json
[{
        "name": "apple",
        "price": 0.99,
        "onSale": true
    },
    {
        "name": "orange",
        "price": 1.99,
        "onSale": false
    },
    {
        "name": "passion fruit",
        "price": 4.99,
        "onSale": false
    }
]
```

Lo

Collections: each

- _.each(collection, [iteratee=_.identity]) and _.eachRight(collection, [iteratee=_.identity]) iterate over elements in the collection and invoke the iteratee function.
- The difference is that _.eachRight iterates from right to left
- _.forEach is an alias of _.each,
- _.forEachRight is an alias of _.eachRight.

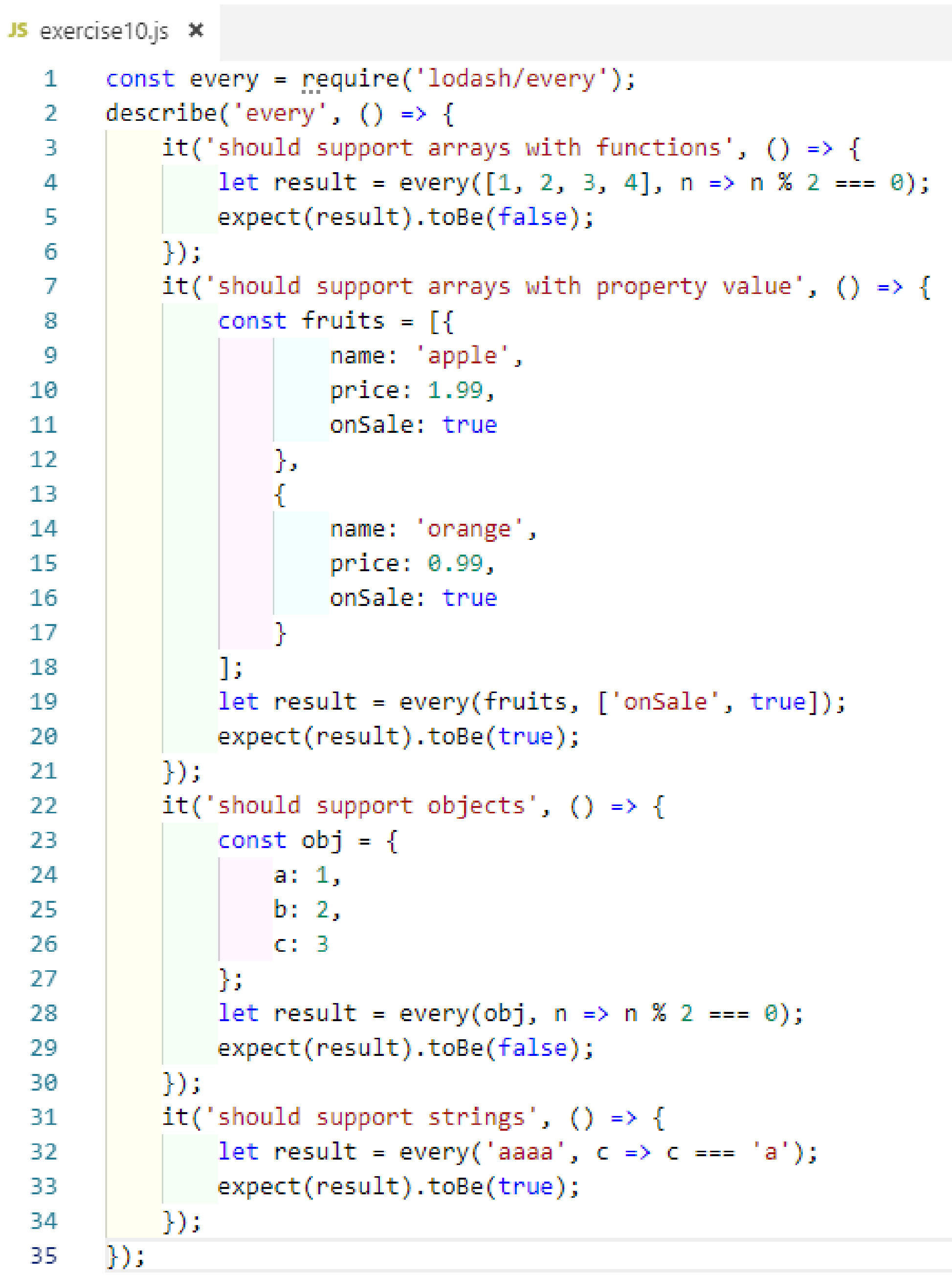

exercise09.js

```
const each = require('lodash/each');
describe('each', () => {
    it('should support basic iteration', () => {
        let sum = 0;
        each([1, 2, 3], val => sum += val);
        expect(sum).toEqual(6);
    });
});
```

Lo

Collections: Every and some

.every(collection, [predicate=.identity]) checks if all elements in the collection match the given predicate.

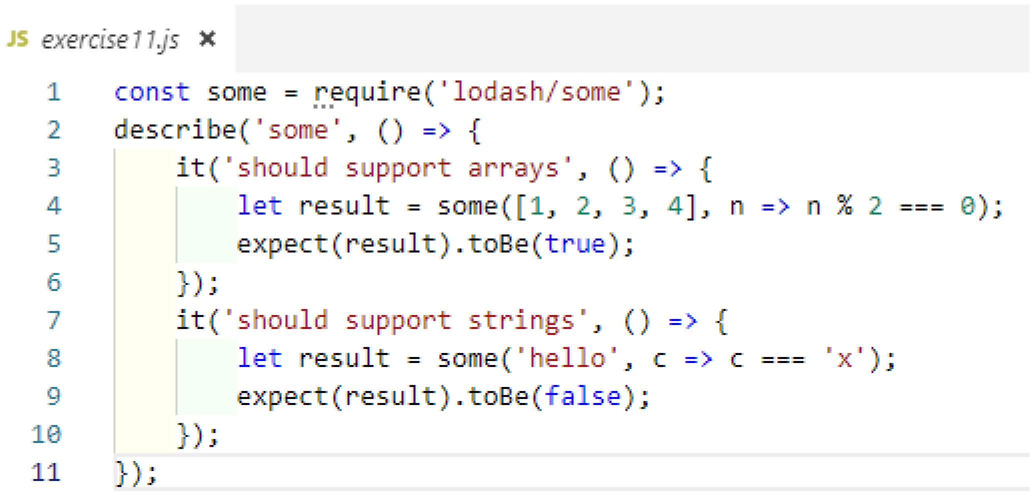


exercise10.js

```
const every = require('lodash/every');
describe('every', () => {
    it('should support arrays with functions', () => {
        let result = every([1, 2, 3, 4], n => n % 2 === 0);
        expect(result).toBe(false);
    });
    it('should support arrays with property value', () => {
        const fruits = [{
                name: 'apple',
                price: 1.99,
                onSale: true
            },
            {
                name: 'orange',
                price: 0.99,
                onSale: true
            }
        ];
        let result = every(fruits, ['onSale', true]);
        expect(result).toBe(true);
    });
    it('should support objects', () => {
        const obj = {
            a: 1,
            b: 2,
            c: 3
        };
        let result = every(obj, n => n % 2 === 0);
        expect(result).toBe(false);
    });
    it('should support strings', () => {
        let result = every('aaaa', c => c === 'a');
        expect(result).toBe(true);
    });
});
```

Collections: Every and some

- _.some(collection, [predicate=_.identity]) is the opposite of _.every
- which checks if any element in the collection matches the given predicate.
- _.some doesn't need to iterate the entire collection and the iteration exits as soon as a matching element is found.

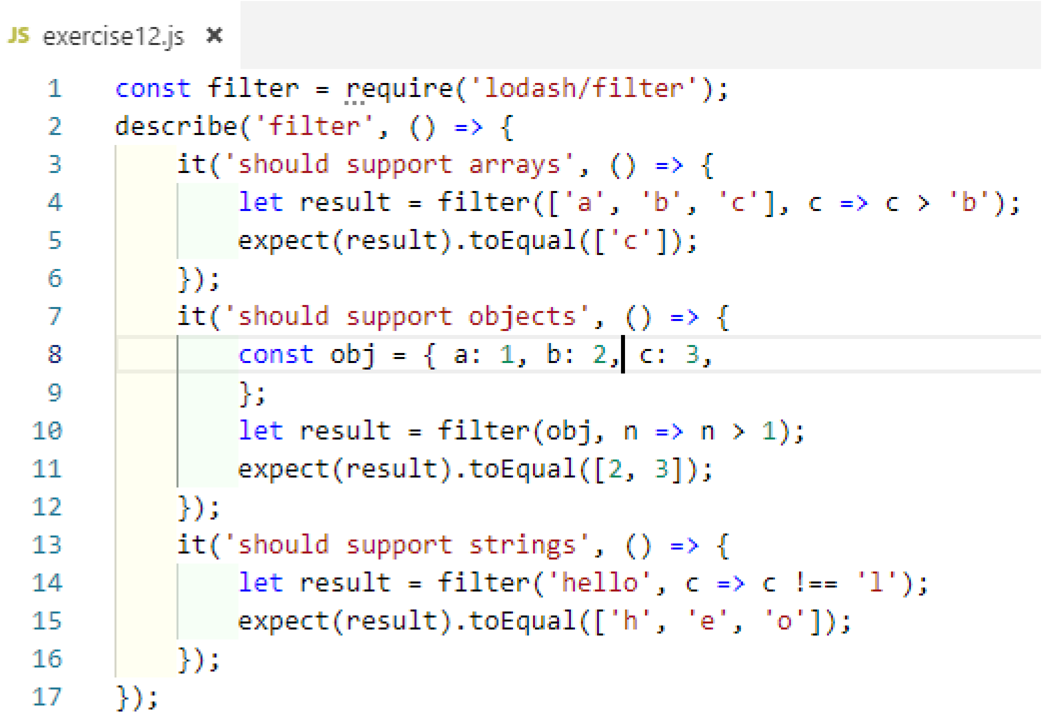


```
const some = require('lodash/some');
describe('some', () => {
    it('should support arrays', () => {
        let result = some([1, 2, 3, 4], n => n % 2 === 0);
        expect(result).toBe(true);
    });
    it('should support strings', () => {
        let result = some('hello', c => c === 'x');
        expect(result).toBe(false);
    });
});
```

Filter and reject

- _.filter(collection, [predicate=_.identity]) filters a collection by returning elements matching the given predicate.
- _.reject(collection, [predicate=_.identity]) is the opposite of _.filter that returns elements not matching the given predicate.
- When _.filter is used to filter objects, only values of matching properties are returned.
- If you want to keep the original object structure, use _.pick or _.omit instead.

When _.filter is used on strings, matching characters are returned in an array

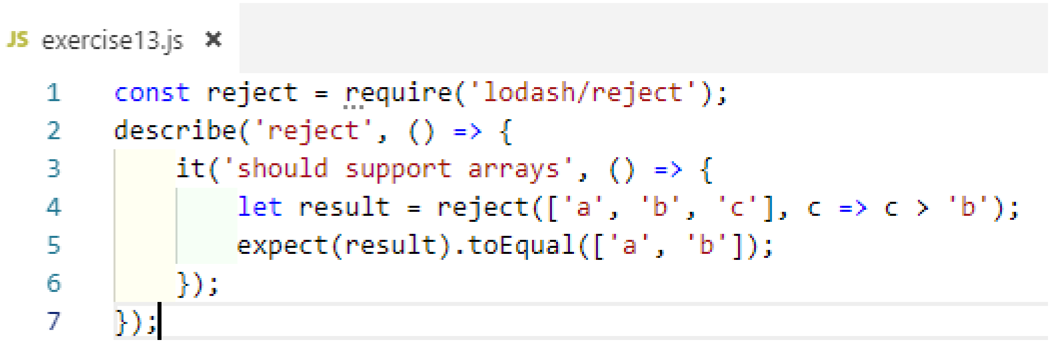


```
const filter = require('lodash/filter');
describe('filter', () => {
    it('should support arrays', () => {
        let result = filter(['a', 'b', 'c'], c => c > 'b');
        expect(result).toEqual(['c']);
    });
    it('should support objects', () => {
        const obj = { a: 1, b: 2, c: 3,
        };
        let result = filter(obj, n => n > 1);
        expect(result).toEqual([2, 3]);
    });
    it('should support strings', () => {
        let result = filter('hello', c => c !== 'l');
        expect(result).toEqual(['h', 'e', 'o']);
    });
});
```

Lo

Collections: Filter and reject

- Example of _.reject



exercise13.js

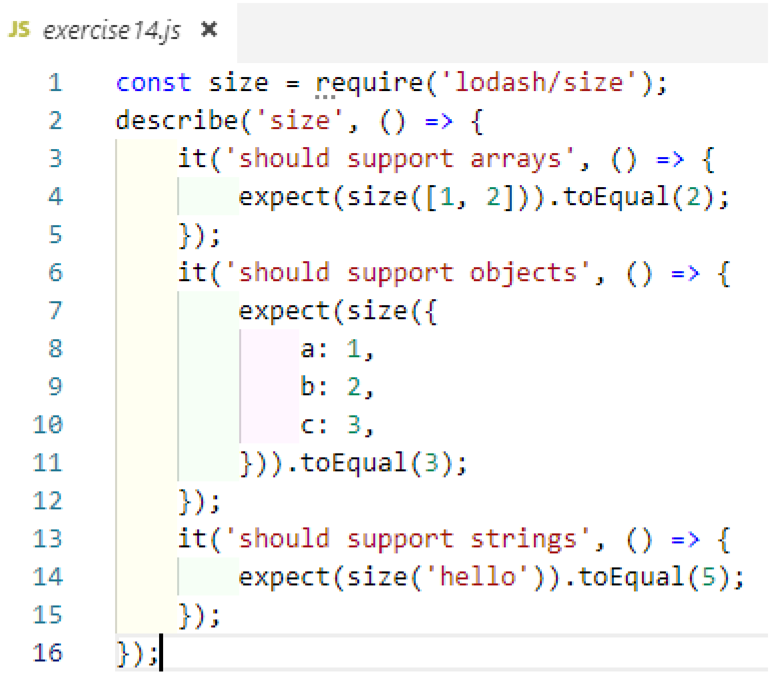
```
const reject = require('lodash/reject');
describe('reject', () => {
    it('should support arrays', () => {
        let result = reject(['a', 'b', 'c'], c => c > 'b');
        expect(result).toEqual(['a', 'b']);
    });
});
```

- _.filter and _.reject always return a new array.
- The input collection is not modified.
- A new array of filtered or rejected elements, object property values or characters is returned.

Lo

Collections: Size

- _.size(collection) gets the size of a collection. For arrays, the size is the array's length, same as the array's property length.
- For objects, the size is the number of own enumerable properties, i.e. the length of the array returned by _.keys.
- For strings, the size is the string's length.

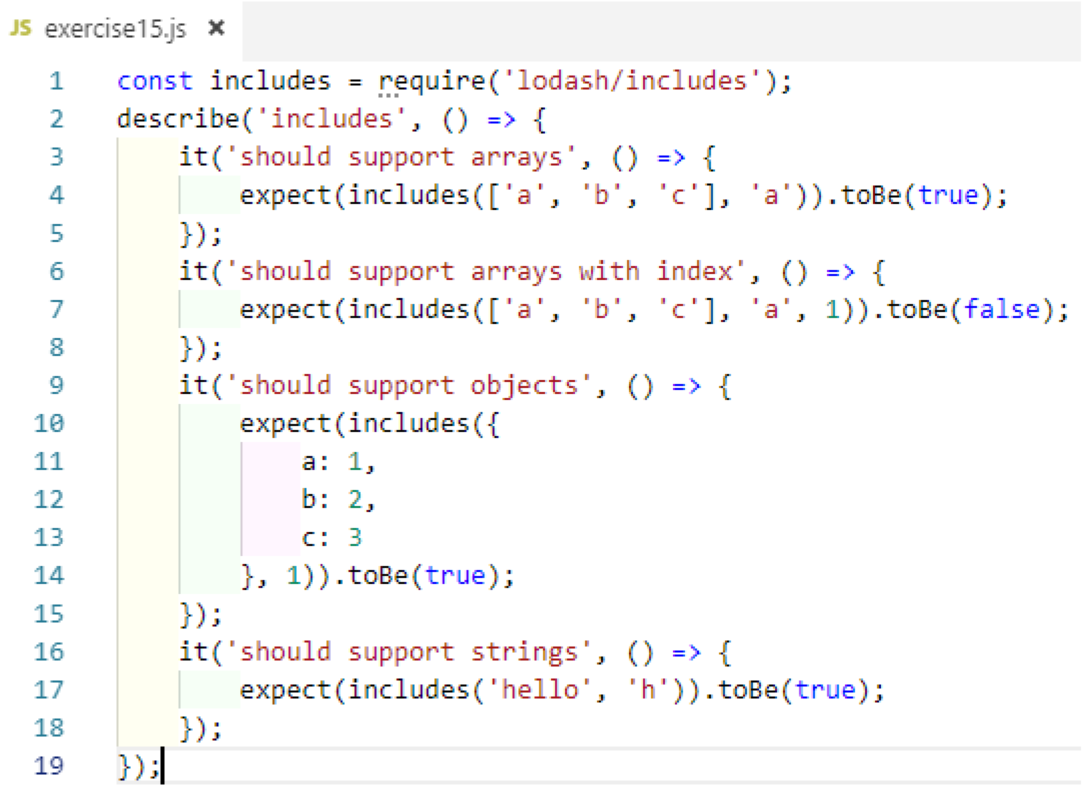


exercise14.js

```
const size = require('lodash/size');
describe('size', () => {
    it('should support arrays', () => {
        expect(size([1, 2])).toEqual(2);
    });
    it('should support objects', () => {
        expect(size({
            a: 1,
            b: 2,
            c: 3,
        })).toEqual(3);
    });
    it('should support strings', () => {
        expect(size('hello')).toEqual(5);
    });
});
```

Collections: Includes

- _.includes(collection, value, [fromIndex=0]) checks if a collection contains the given value.
- An optional index can be provided as the starting position to search.
- If the collection is an object, values of this object's properties, i.e. the result of _.values, are searched instead. _.includes uses the SameAsZero algorithm to check equality



exercise15.js

```
const includes = require('lodash/includes');
describe('includes', () => {
    it('should support arrays', () => {
        expect(includes(['a', 'b', 'c'], 'a')).toBe(true);
    });
    it('should support arrays with index', () => {
        expect(includes(['a', 'b', 'c'], 'a', 1)).toBe(false);
    });
    it('should support objects', () => {
        expect(includes({
            a: 1,
            b: 2,
            c: 3
        }, 1)).toBe(true);
    });
    it('should support strings', () => {
        expect(includes('hello', 'h')).toBe(true);
    });
});
```

Collections: Sample

```
var _ = require('lodash');
console.log(_.sample(['a', 'b', 'c']) );
// -> 'a'
_.sample({
    a: 1,
    b: 2,
    c: 3
});
// -> 1
console.log( _.sample('hello') );
// -> 'h'
console.log( _.sampleSize('hello', 2) );
// -> ['h', 'l']
```

Collections: Shuffle

```
var _ = require('lodash');
console.log( _.shuffle(['a', 'b', 'c']) );
// -> ['b', 'c', 'a']
console.log( _.shuffle({
a: 1,
b: 2,
c: 3
}) );
// -> [1, 2, 3]
console.log( _.shuffle('hello') );
// -> ['l', 'l', 'o', 'h', 'e']
```

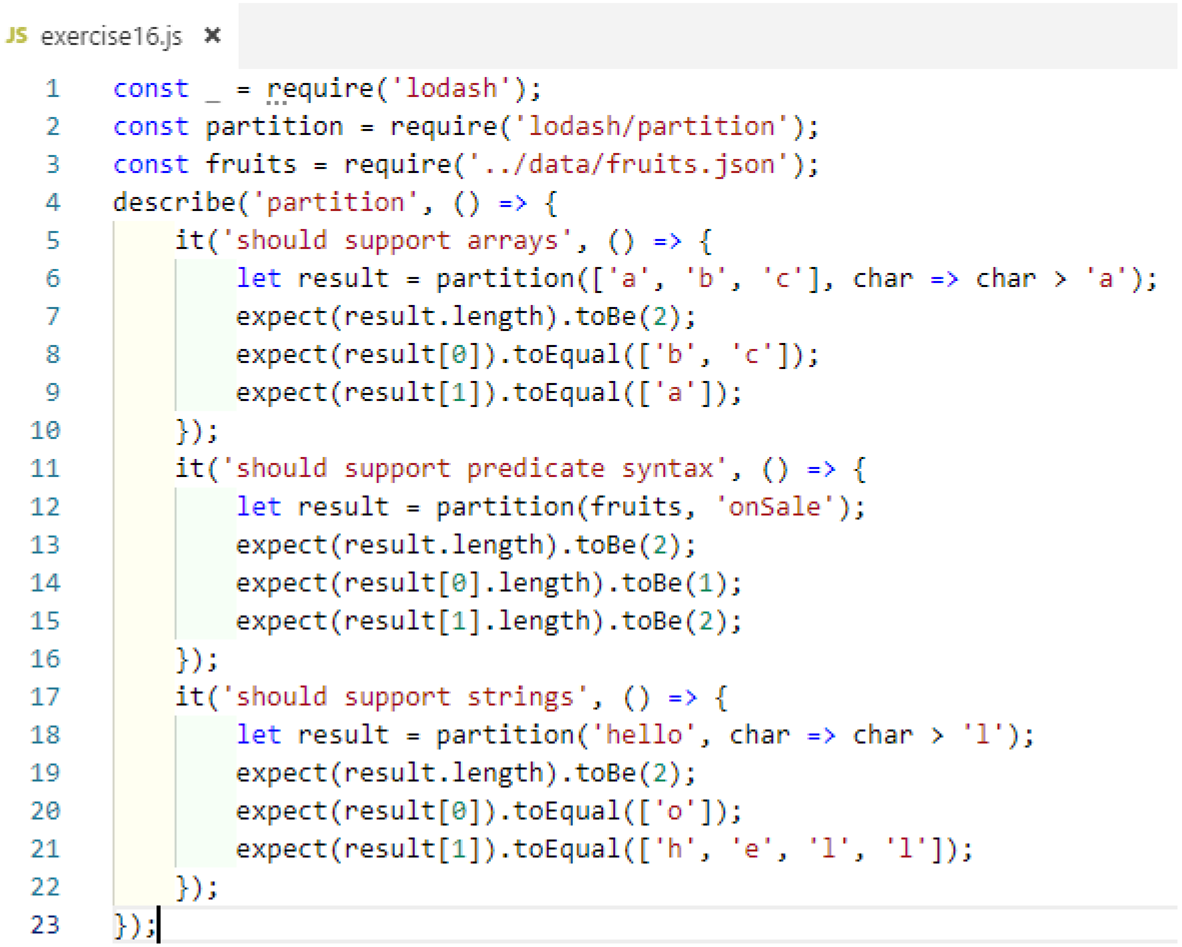
Due to the random nature of _.sample, _.sampleSize and _.shuffle, most likely the result will be different when running code shown above on your local machine.

Lo

Collections: Partition

- _.partition(collection, [predicate=_.identity]) splits a collection into two groups based on the result of invoking the predicate on each element.

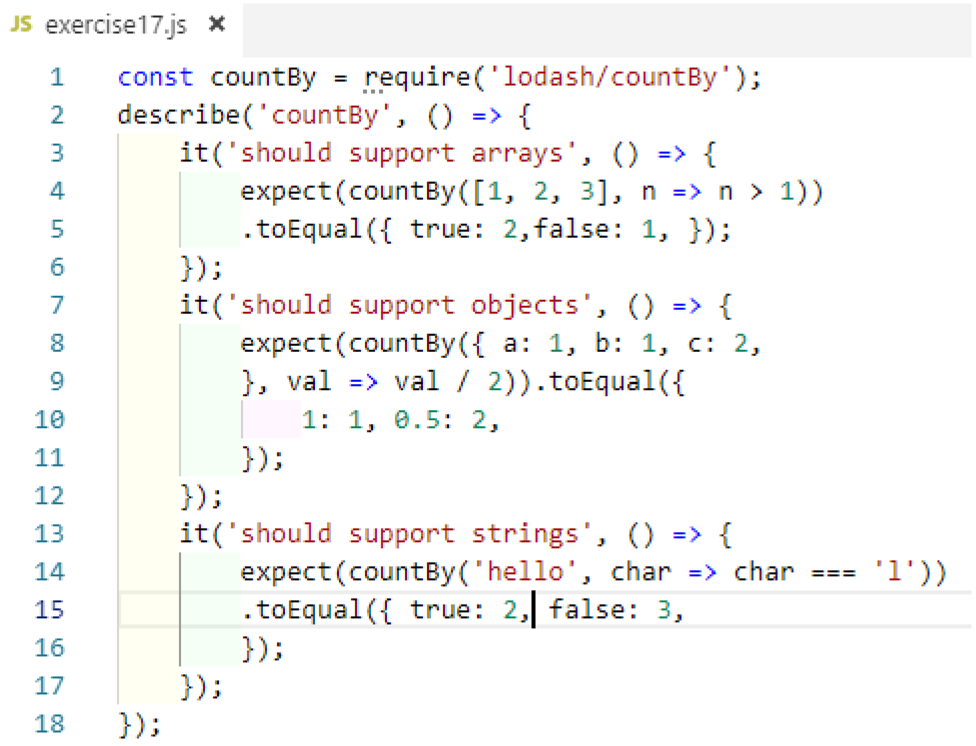
The first group contains elements for which the predicate returns a truthy value, while the second group contains elements for which the predicate returns a falsy value


exercise16.js

```
const _ = require('lodash');
const partition = require('lodash/partition');
const fruits = require('../data/fruits.json');
describe('partition', () => {
    it('should support arrays', () => {
        let result = partition(['a', 'b', 'c'], char => char > 'a');
        expect(result.length).toBe(2);
        expect(result[0]).toEqual(['b', 'c']);
        expect(result[1]).toEqual(['a']);
    });
    it('should support predicate syntax', () => {
        let result = partition(fruits, 'onSale');
        expect(result.length).toBe(2);
        expect(result[0].length).toBe(1);
        expect(result[1].length).toBe(2);
    });
    it('should support strings', () => {
        let result = partition('hello', char => char > 'l');
        expect(result.length).toBe(2);
        expect(result[0]).toEqual(['o']);
        expect(result[1]).toEqual(['h', 'e', 'l', 'l']);
    });
});
```

Collections: Count by

- _.countBy(collection, [iteratee=_.identity]) applies a function to each element in the collection and counts the number of occurrences of each result.
- The counting result is returned as an object with the applied result as the keys and the count as the corresponding values.



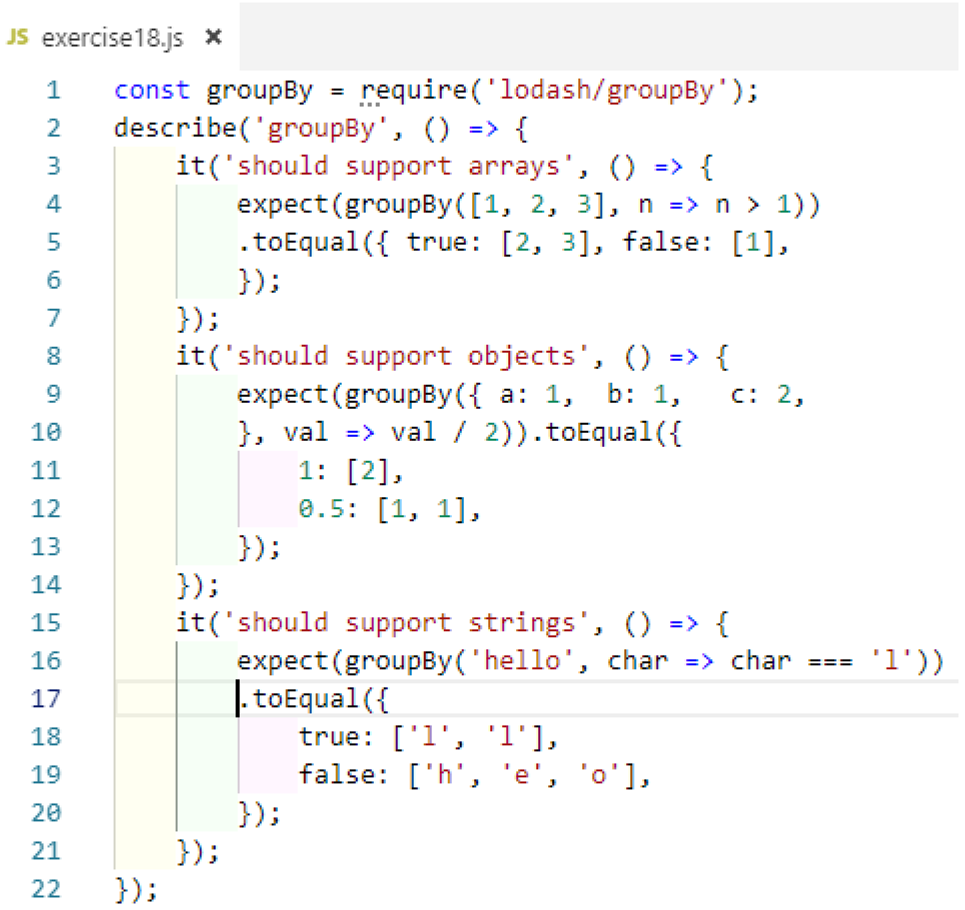
```
const countBy = require('lodash/countBy');
describe('countBy', () => {
    it('should support arrays', () => {
        expect(countBy([1, 2, 3], n => n > 1))
        .toEqual({ true: 2,false: 1, });
    });
    it('should support objects', () => {
        expect(countBy({ a: 1, b: 1, c: 2,
        }, val => val / 2)).toEqual({
            1: 1, 0.5: 2,
        });
    });
    it('should support strings', () => {
        expect(countBy('hello', char => char === 'l'))
        .toEqual({ true: 2, false: 3,
        });
    });
});
```

Lo

Collections: Group by and key by

- _.groupBy(collection, [iteratee=_.identity]) applies a function to each element in the collection and groups the elements by the result. Elements having the same result will be in the same group.
- The grouping result is returned as an object.

- The keys in the object are the applied results, while the values are arrays of elements which generate the corresponding result.

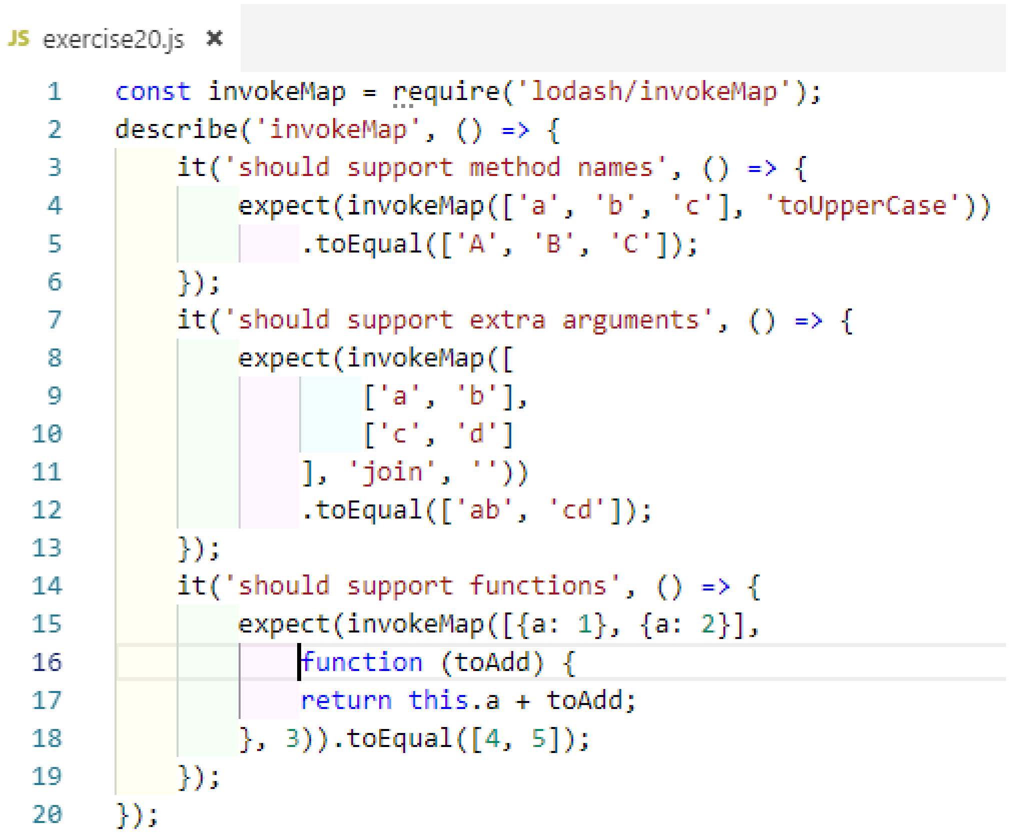


exercise18.js

```javascript
const groupBy = require('lodash/groupBy');
describe('groupBy', () => {
    it('should support arrays', () => {
        expect(groupBy([1, 2, 3], n => n > 1))
        .toEqual({ true: [2, 3], false: [1],
        });
    });
    it('should support objects', () => {
        expect(groupBy({ a: 1,  b: 1,   c: 2,
        }, val => val / 2)).toEqual({
            1: [2],
            0.5: [1, 1],
        });
    });
    it('should support strings', () => {
        expect(groupBy('hello', char => char === 'l'))
        .toEqual({
            true: ['l', 'l'],
            false: ['h', 'e', 'o'],
        });
    });
});
```

Collections: invokeMap

- _.invokeMap(collection, path, [args]) invokes a method on each element in the collection and returns the results in an array.
- The method to invoke is specified by the path, can be the function's name or the function itself.
- Additional arguments can also be provided for the method invocation.
- In the code shown, when the function is invoked, this references to the current element.



exercise20.js

```
const invokeMap = require('lodash/invokeMap');
describe('invokeMap', () => {
    it('should support method names', () => {
        expect(invokeMap(['a', 'b', 'c'], 'toUpperCase'))
            .toEqual(['A', 'B', 'C']);
    });
    it('should support extra arguments', () => {
        expect(invokeMap([
                ['a', 'b'],
                ['c', 'd']
            ], 'join', ''))
            .toEqual(['ab', 'cd']);
    });
    it('should support functions', () => {
        expect(invokeMap([{a: 1}, {a: 2}],
            function (toAdd) {
            return this.a + toAdd;
        }, 3)).toEqual([4, 5]);
    });
});
```

Collections: Map and reduce

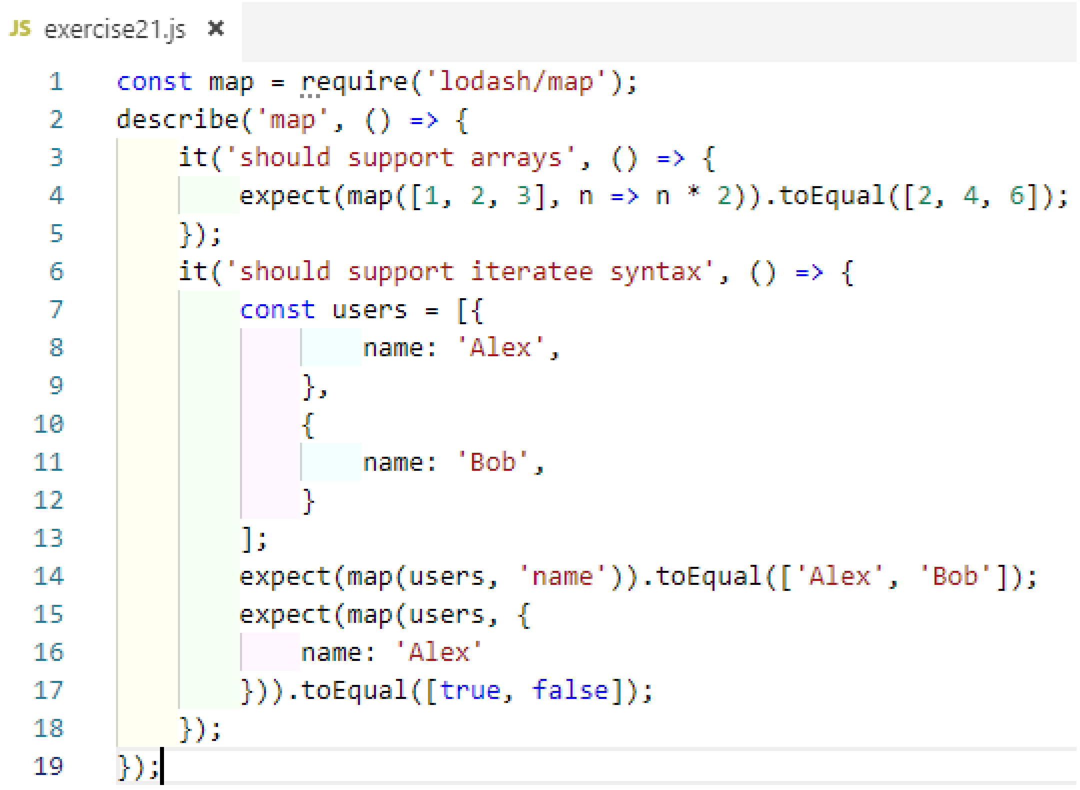
Map and reduce are common operations when processing collections.

- Map transforms a collection into another collection by applying an operation to each element in the collection.
- Reduce transforms a collection into a single value by accumulating results of applying an operation to each element.
- The result of the last operation is used as the input of the current operation.

Lo

Collections: Map and reduce: Map

.map(collection, [iteratee=.identity]) is the generic map function. We can use the different iteratee syntax.

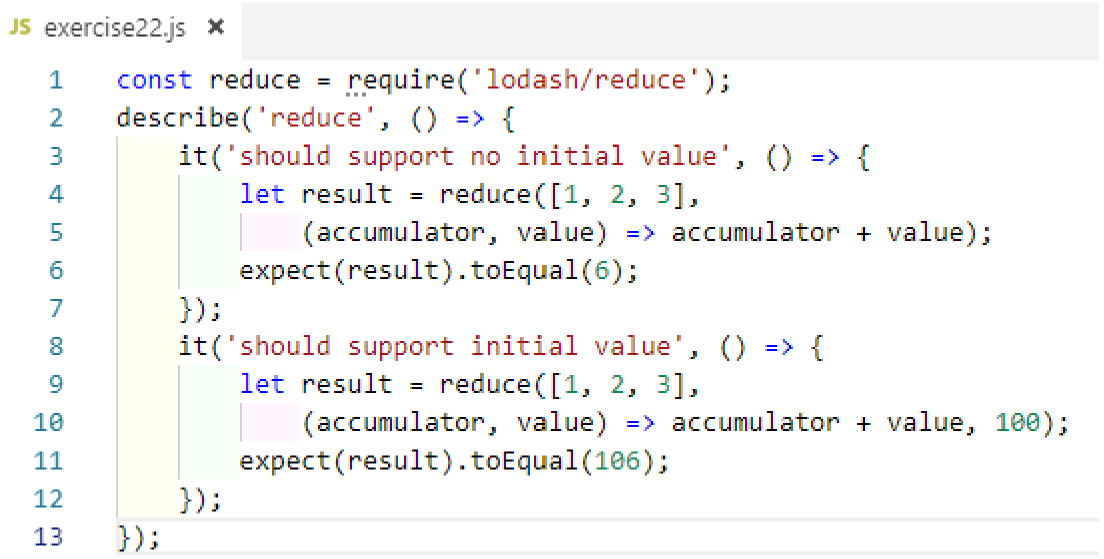

exercise21.js

```js
const map = require('lodash/map');
describe('map', () => {
    it('should support arrays', () => {
        expect(map([1, 2, 3], n => n * 2)).toEqual([2, 4, 6]);
    });
    it('should support iteratee syntax', () => {
        const users = [{
                name: 'Alex',
            },
            {
                name: 'Bob',
            }
        ];
        expect(map(users, 'name')).toEqual(['Alex', 'Bob']);
        expect(map(users, {
            name: 'Alex'
        })).toEqual([true, false]);
    });
});
```

Collections: Map and reduce: Reduce

.reduce(collection, [iteratee=.identity], [accumulator]) has similar arguments list with _.map, except that it accepts an optional value as the initial input of the first reduce operation.

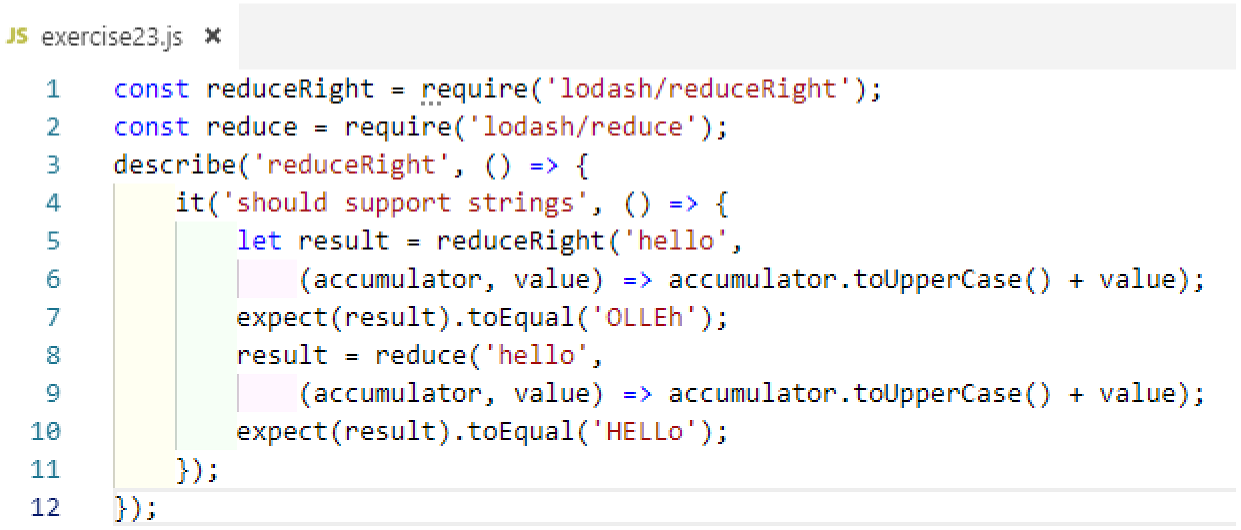
- If the initial value is not provided, the first element in the collection is used instead.
- The provided iteratee function will be invoked with four arguments, accumulator, value, index/key and collection.
- accumulator is the current reduced value, while value is the current element in the collection.
- The returned result of the iteratee function invocation is passed as the accumulator value of the next invocation.



```javascript
const reduce = require('lodash/reduce');
describe('reduce', () => {
    it('should support no initial value', () => {
        let result = reduce([1, 2, 3],
            (accumulator, value) => accumulator + value);
        expect(result).toEqual(6);
    });
    it('should support initial value', () => {
        let result = reduce([1, 2, 3],
            (accumulator, value) => accumulator + value, 100);
        expect(result).toEqual(106);
    });
});
```

Collections: Map and reduce: Reduce

.reduceRight(collection, [iteratee=.identity], [accumulator] is similar with _.reduce) except _.reduceRight iterates all the elements from right to left.



```
const reduceRight = require('lodash/reduceRight');
const reduce = require('lodash/reduce');
describe('reduceRight', () => {
    it('should support strings', () => {
        let result = reduceRight('hello',
            (accumulator, value) => accumulator.toUpperCase() + value);
        expect(result).toEqual('OLLEh');
        result = reduce('hello',
            (accumulator, value) => accumulator.toUpperCase() + value);
        expect(result).toEqual('HELLo');
    });
});
```

Collections: Search

Search is a very common task in programming.

- Search is performed on iterable collections with given conditions.
- The return result is the first element in the collection matching the condition, or undefined if no matching element is found.

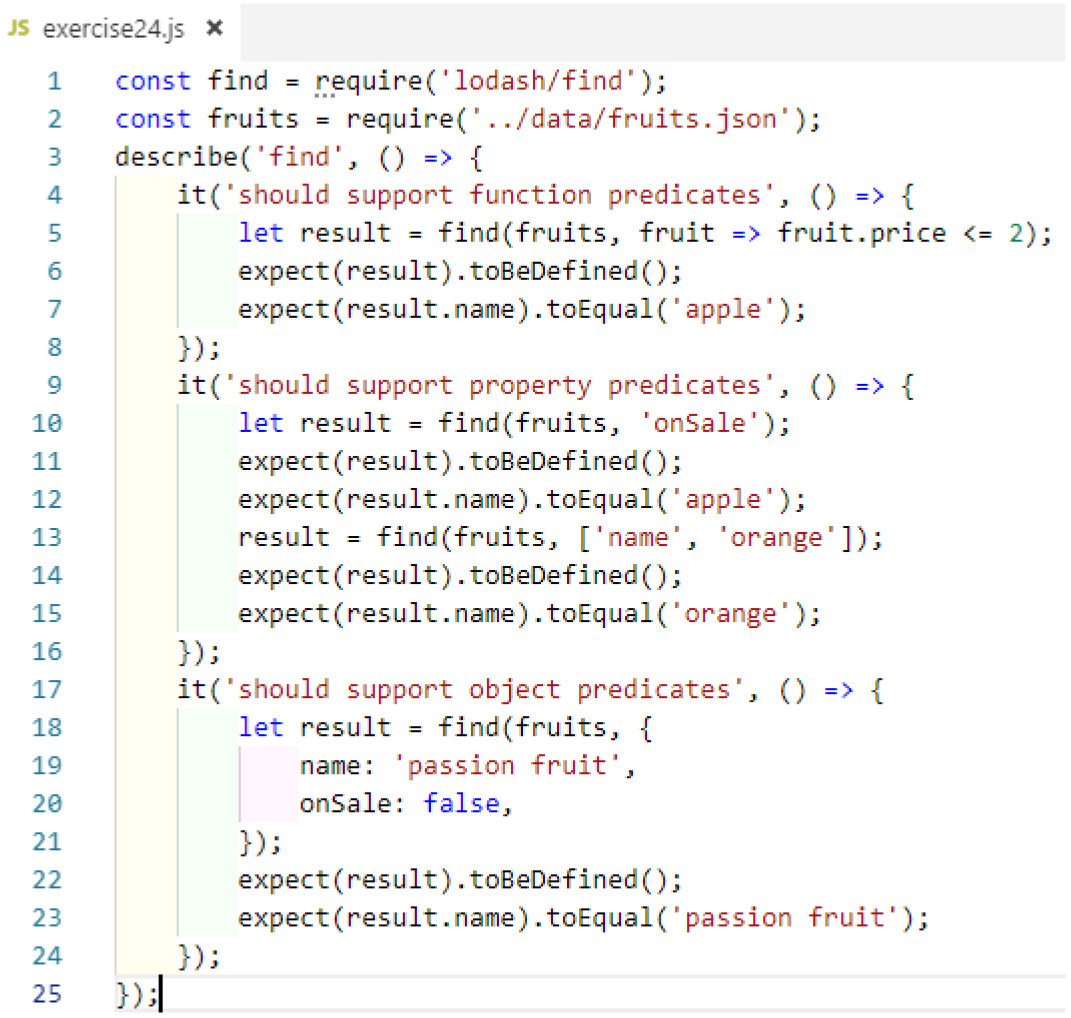
- _.find(collection, [predicate=_.identity], [fromIndex=0]) is the generic function to search in collections.

- When invoking _.find, the collection itself and the search condition should be provided.
- We can also provide an optional starting index for the search.
- _.find supports the same predicate syntax.
- If a function is provided as the predicate, the function is invoked for each element in the array until the function returns a truthy value.
- The function is invoked with three arguments: the currently iterated element, index or key of the element and the collection itself

Lo

Collections: Search

_find:



exercise24.js

```
const find = require('lodash/find');
const fruits = require('../data/fruits.json');
describe('find', () => {
    it('should support function predicates', () => {
        let result = find(fruits, fruit => fruit.price <= 2);
        expect(result).toBeDefined();
        expect(result.name).toEqual('apple');
    });
    it('should support property predicates', () => {
        let result = find(fruits, 'onSale');
        expect(result).toBeDefined();
        expect(result.name).toEqual('apple');
        result = find(fruits, ['name', 'orange']);
        expect(result).toBeDefined();
        expect(result.name).toEqual('orange');
    });
    it('should support object predicates', () => {
        let result = find(fruits, {
            name: 'passion fruit',
            onSale: false,
        });
        expect(result).toBeDefined();
        expect(result.name).toEqual('passion fruit');
    });
});
```

Installation

Sample Book

Basic concepts

Predicates

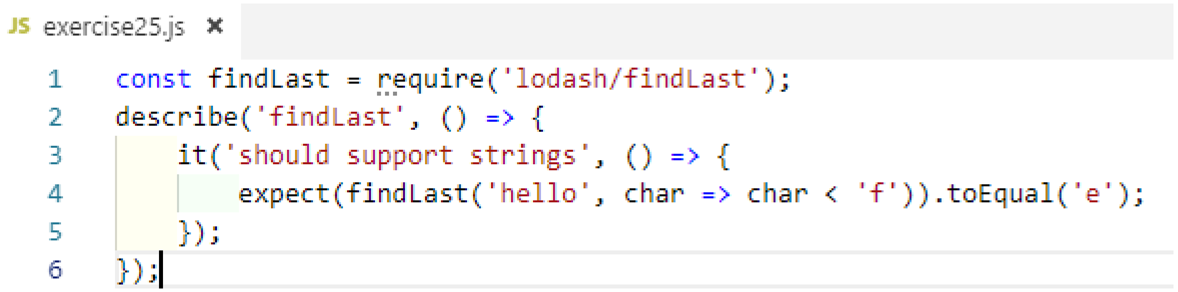
- Matches
- Iteratees
- this binding
- Collections

Recipes

Lo

Collections: Search : _findLast

.findLast(collection, [predicate=.identity], [fromIndex=collection.length-1])
is similar with _.find,
but _.findLast iterates over all elements of the collection in reverse order.
For arrays, it searches from the last element. For strings, it searches from the last character.
For objects, it searches from the last element of the array of property names returned by _.keys.



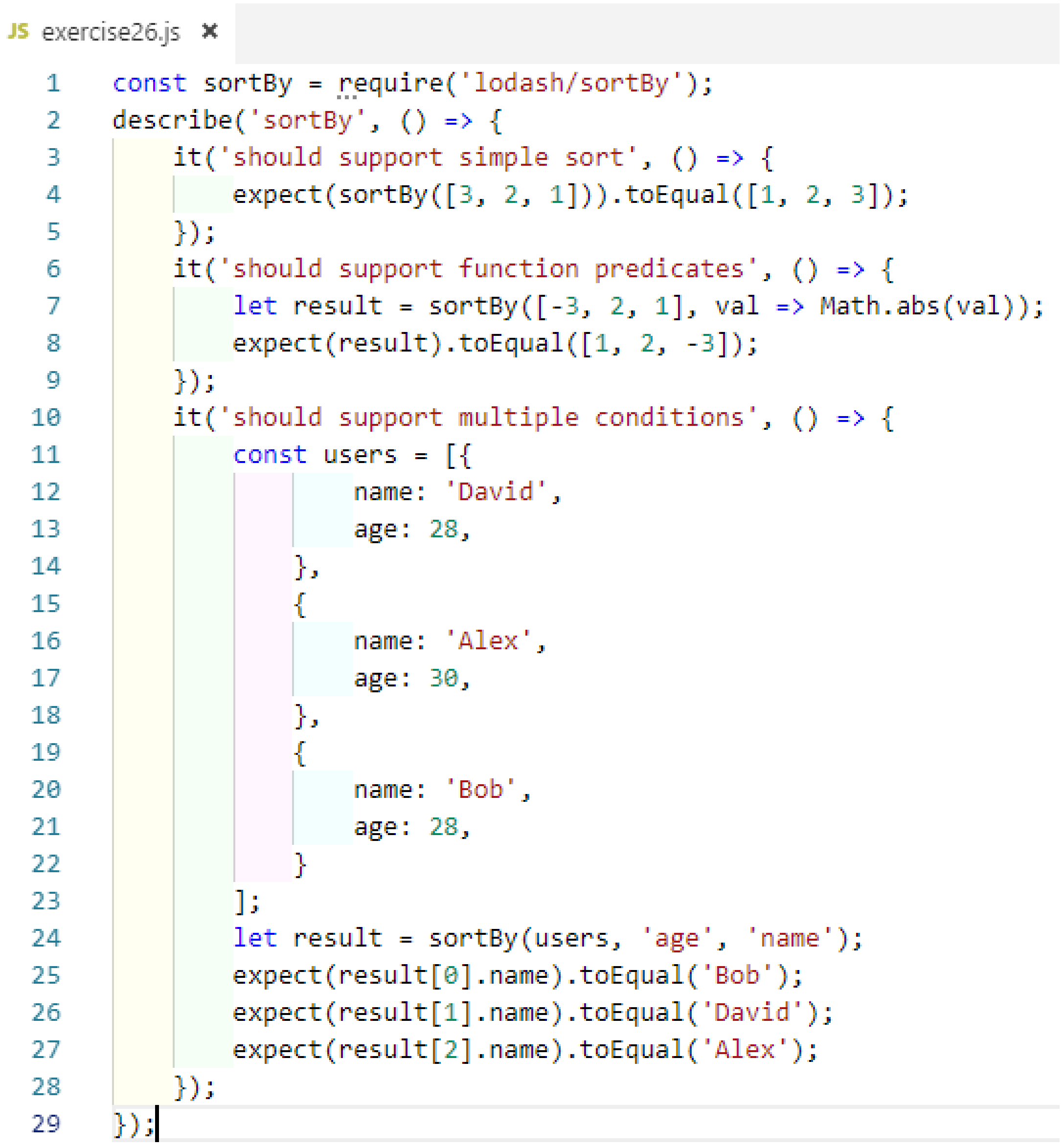
```
const findLast = require('lodash/findLast');
describe('findLast', () => {
    it('should support strings', () => {
        expect(findLast('hello', char => char < 'f')).toEqual('e');
    });
});
```

Collections: Sort

.sortBy(collection, [iteratee=.identity]) sorts a collection in ascending order with results after applying the iteratee function to each element in the collection.

- The sort is stable, which means it preserves original order for elements with equality.
- We can use multiple iteratees as sort conditions.
- If multiple elements in the collection have the same value for the first property name, those elements are sorted using the second property name, and so on.

Collections: Sort



exercise26.js

```javascript
const sortBy = require('lodash/sortBy');
describe('sortBy', () => {
    it('should support simple sort', () => {
        expect(sortBy([3, 2, 1])).toEqual([1, 2, 3]);
    });
    it('should support function predicates', () => {
        let result = sortBy([-3, 2, 1], val => Math.abs(val));
        expect(result).toEqual([1, 2, -3]);
    });
    it('should support multiple conditions', () => {
        const users = [{
                name: 'David',
                age: 28,
            },
            {
                name: 'Alex',
                age: 30,
            },
            {
                name: 'Bob',
                age: 28,
            }
        ];
        let result = sortBy(users, 'age', 'name');
        expect(result[0].name).toEqual('Bob');
        expect(result[1].name).toEqual('David');
        expect(result[2].name).toEqual('Alex');
    });
});
```

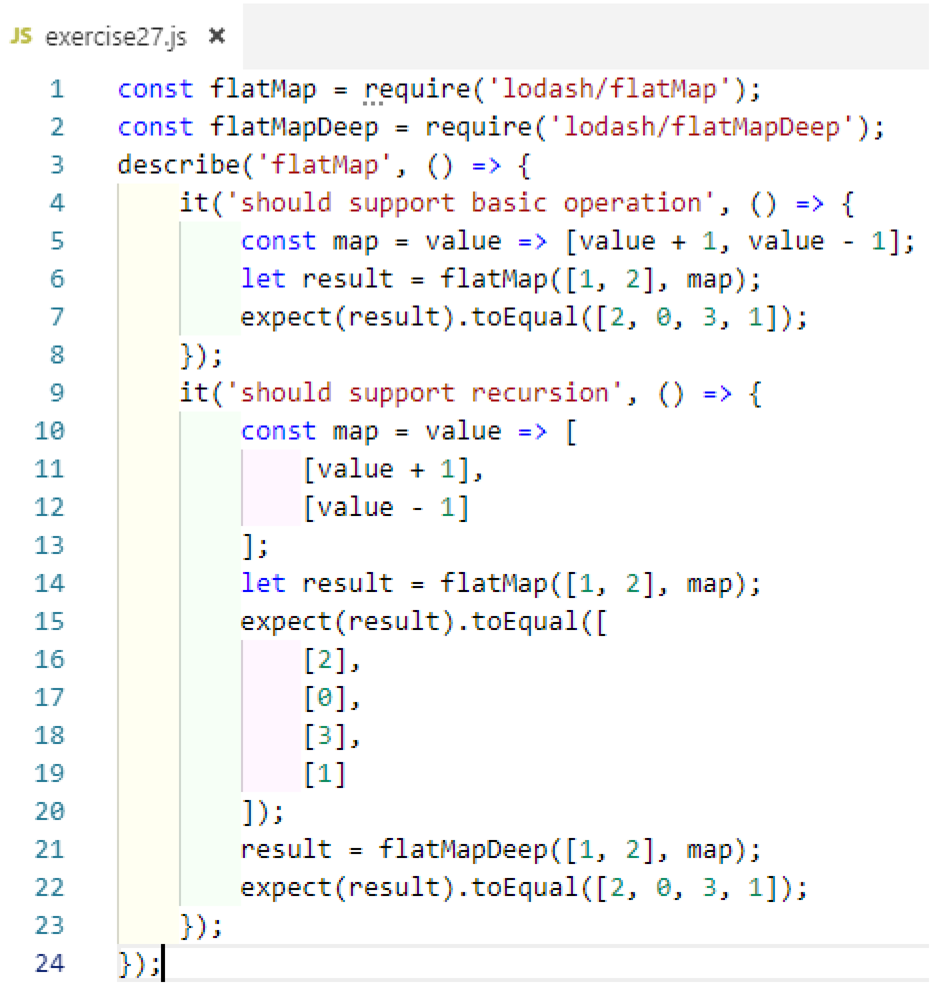
Collections: flatMap

.flatMap(collection, [iteratee=.identity]) invokes an iteratee function to each element in a collection.

- The result of each iteratee function invocation is an array.
- All result arrays are concatenated and flattened into a single array as the final result.
- _.flatMapDeep(collection, [iteratee=_.identity]) is similar with _.flatMap except that _- .flatMapDeep recursively flattens the result array until it’s completely flattened.
- _.flatMapDepth(collection, [iteratee=_.identity], [depth=1]) is similar with _.flatMapDeep except that it only flattens the result the given times.
- The default value of depth is 1, so _.flatMapDepth(array, iteratee) is the same as _.flatMap(array, iteratee).

Lo

Collections: flatMap



exercise27.js

```javascript
const flatMap = require('lodash/flatMap');
const flatMapDeep = require('lodash/flatMapDeep');
describe('flatMap', () => {
    it('should support basic operation', () => {
        const map = value => [value + 1, value - 1];
        let result = flatMap([1, 2], map);
        expect(result).toEqual([2, 0, 3, 1]);
    });
    it('should support recursion', () => {
        const map = value => [
            [value + 1],
            [value - 1]
        ];
        let result = flatMap([1, 2], map);
        expect(result).toEqual([
            [2],
            [0],
            [3],
            [1]
        ]);
        result = flatMapDeep([1, 2], map);
        expect(result).toEqual([2, 0, 3, 1]);
    });
});
```

Recipes:

1. Filter an object’s properties
2. Push an array of elements into an array
3. Create a unique array of objects
4. Convert an array to an object

Recipes: Recipe 1

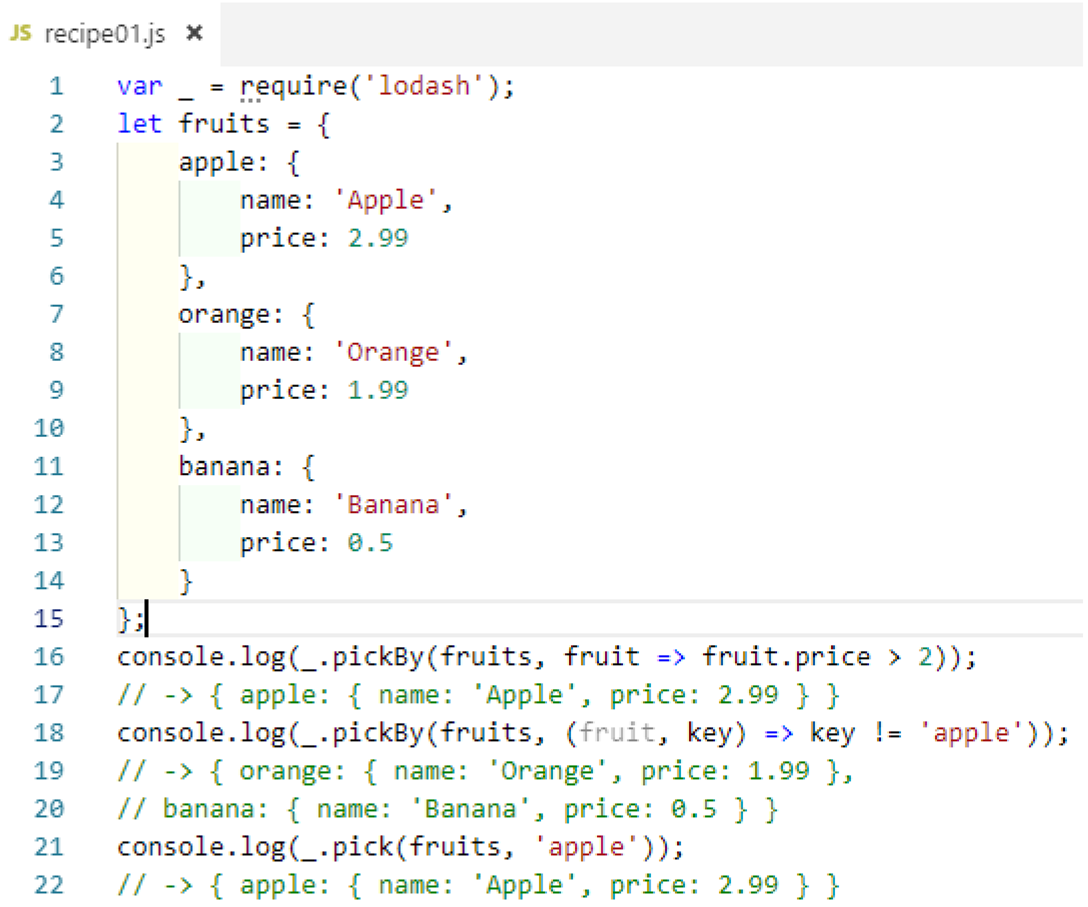
Scenario Filter a given object by removing certain properties.

Although _.filter and _.reject can be applied to objects, they cannot be used for this scenario, because _.filter and _.reject return an array of property values after filtering. _.pick, _.pickBy, _.omit and _.omitBy should be used instead.

Lo

Recipes: Recipe 1 (cont.)

Scenario: Filter a given object by removing certain properties.



recipe01.js

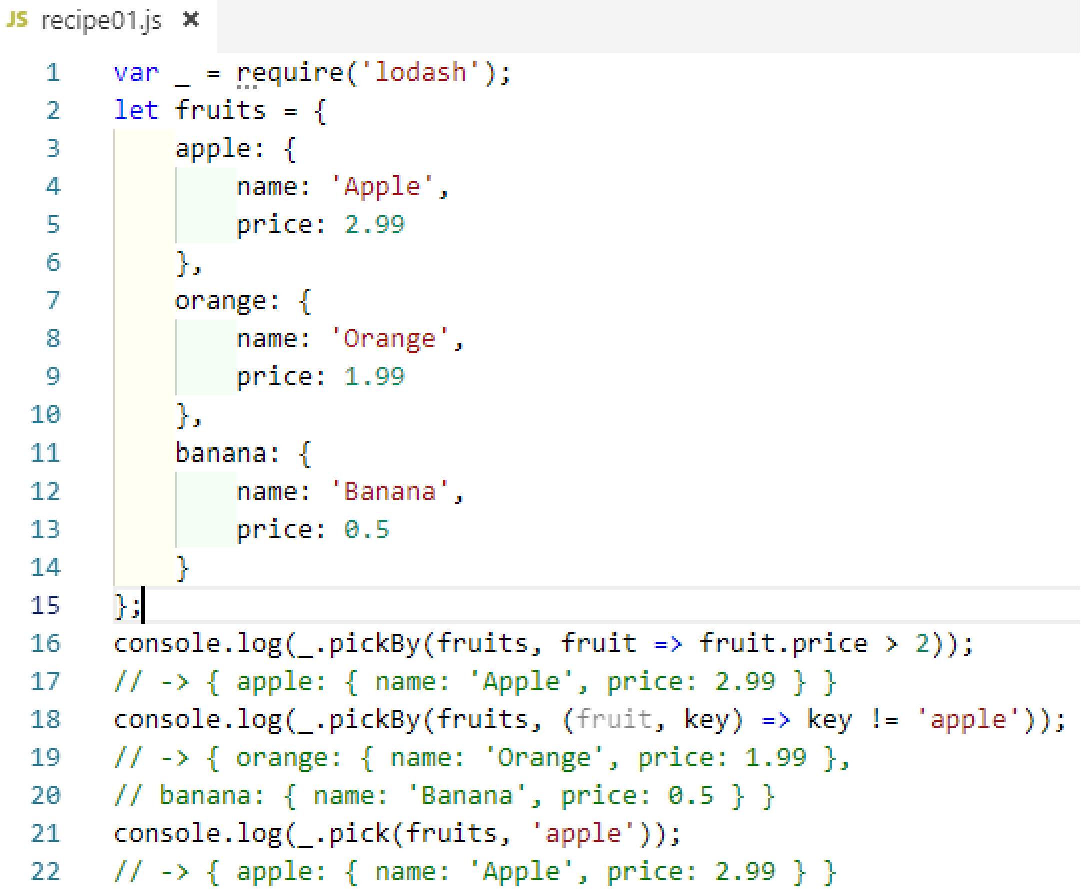
```
var _ = require('lodash');
let fruits = {
    apple: {
        name: 'Apple',
        price: 2.99
    },
    orange: {
        name: 'Orange',
        price: 1.99
    },
    banana: {
        name: 'Banana',
        price: 0.5
    }
};
console.log(_.pickBy(fruits, fruit => fruit.price > 2));
// -> { apple: { name: 'Apple', price: 2.99 } }
console.log(_.pickBy(fruits, (fruit, key) => key != 'apple'));
// -> { orange: { name: 'Orange', price: 1.99 },
// banana: { name: 'Banana', price: 0.5 } }
console.log(_.pick(fruits, 'apple'));
// -> { apple: { name: 'Apple', price: 2.99 } }
```

When a predicate function is passed to _.pickBy or _.omitBy, it’s invoked with three arguments: property value, property name and the object itself.

Lo

Recipes: Recipe 1 (cont.)

Scenario Filter a given object by removing certain properties.



recipe01.js

```javascript
var _ = require('lodash');
let fruits = {
    apple: {
        name: 'Apple',
        price: 2.99
    },
    orange: {
        name: 'Orange',
        price: 1.99
    },
    banana: {
        name: 'Banana',
        price: 0.5
    }
};
console.log(_.pickBy(fruits, fruit => fruit.price > 2));
// -> { apple: { name: 'Apple', price: 2.99 } }
console.log(_.pickBy(fruits, (fruit, key) => key != 'apple'));
// -> { orange: { name: 'Orange', price: 1.99 },
// banana: { name: 'Banana', price: 0.5 } }
console.log(_.pick(fruits, 'apple'));
// -> { apple: { name: 'Apple', price: 2.99 } }
```

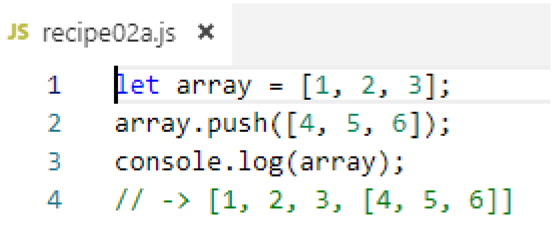
When a predicate function is passed to _.pickBy or _.omitBy, it’s invoked with three arguments: property value, property name and the object itself.

Lo

Recipes: Recipe 2. Push an array of elements into an array

Scenario: Given an array of elements, push those elements into another array

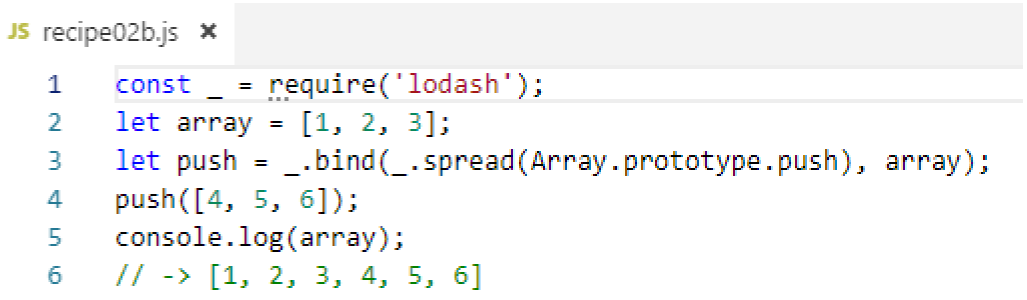
If using Array’s push method, the whole array will be pushed as a single element



recipe02a.js

```javascript
let array = [1, 2, 3];
array.push([4, 5, 6]);
console.log(array);
// -> [1, 2, 3, [4, 5, 6]]
```

The first solution is to use _.spread to wrap push method to accept arrays as arguments.



recipe02b.js

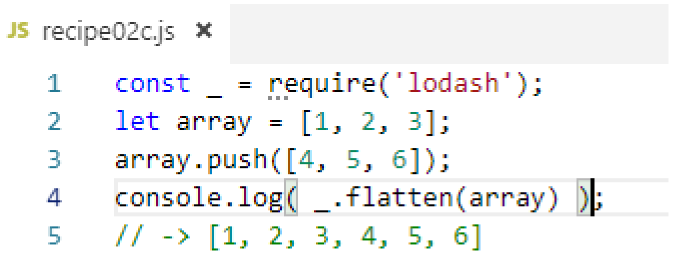
```javascript
const _ = require('lodash');
let array = [1, 2, 3];
let push = _.bind(_.spread(Array.prototype.push), array);
push([4, 5, 6]);
console.log(array);
// -> [1, 2, 3, 4, 5, 6]
```

Lo

Recipes: Recipe 2. Push an array of elements into an array (cont.)

Scenario: Given an array of elements, push those elements into another array

The second solution is to push the array first, then use _.flatten to flatten the array



```
const _ = require('lodash');
let array = [1, 2, 3];
array.push([4, 5, 6]);
console.log( _.flatten(array) );
// -> [1, 2, 3, 4, 5, 6]
```

Recipes: Recipe 3. Create a unique array of objects

Scenario: Given an array of objects, remove duplicate values from the array

_.uniq and uniqBy functions can be used to remove duplicate values from an array, but it only uses SameAsZero algorithm to compare values.

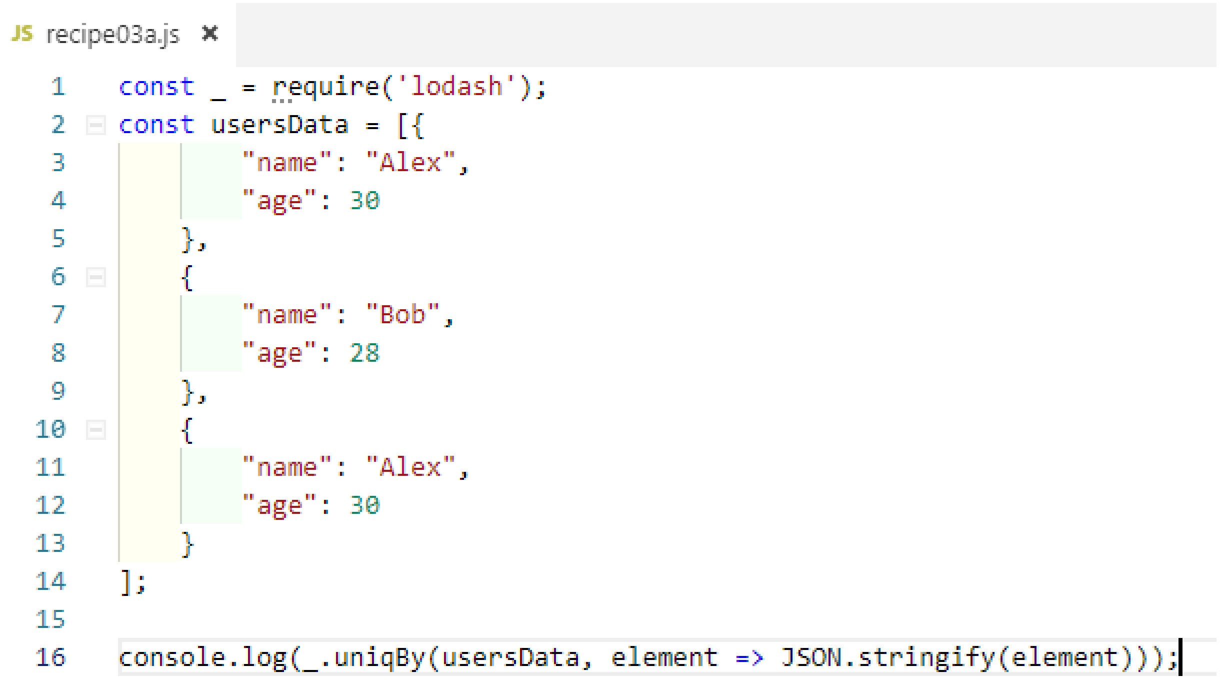
To perform the deep comparison for elements in the array , we need to convert each element into a single value.

For example, if the property name is the unique key for each element, use _.uniqBy(array, 'name').
If there is no unique key, you can convert the element into a JSON string.

Lo

Recipes: Recipe 3. Create a unique array of objects (cont.)

Scenario: Given an array of objects, remove duplicate values from the array



recipe03a.js

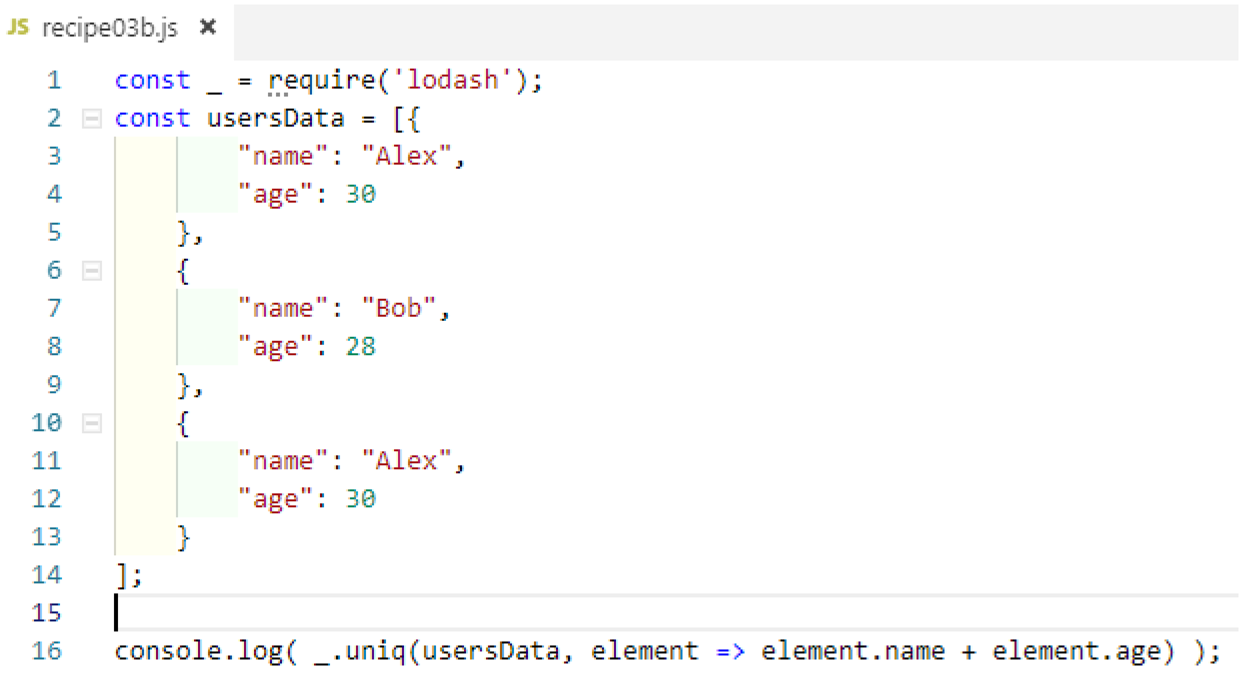
```javascript
const _ = require('lodash');
const usersData = [{
        "name": "Alex",
        "age": 30
    },
    {
        "name": "Bob",
        "age": 28
    },
    {
        "name": "Alex",
        "age": 30
    }
];

console.log(_.uniqBy(usersData, element => JSON.stringify(element)));
```

Lo

Recipes: Recipe 3. Convert an array to an object (cont.)

Scenario: Given an array of objects with IDs, convert the array to an object with IDs as the keys and array elements as the values.

JSON serialization may generate different results for objects with the same value due to the undermined property enumeration order. For a more consistent result, we should create our own object serialization format. In code shown below, we concatenate name and age properties as the serialization format to determine uniqueness.


recipe03b.js

```
const _ = require('lodash');
const usersData = [{
        "name": "Alex",
        "age": 30
    },
    {
        "name": "Bob",
        "age": 28
    },
    {
        "name": "Alex",
        "age": 30
    }
];

console.log( _.uniq(usersData, element => element.name + element.age) );
```

Lo

Recipes: Recipe 4. Convert an array to an object

Scenario: Given an array of objects with IDs, convert the array to an object with IDs as the keys and array elements as the values.

Input:

```
[
{ "id": "user001", "name": "Alex" },
{ "id": "user002", "name": "Bob" }
]
```

Output:

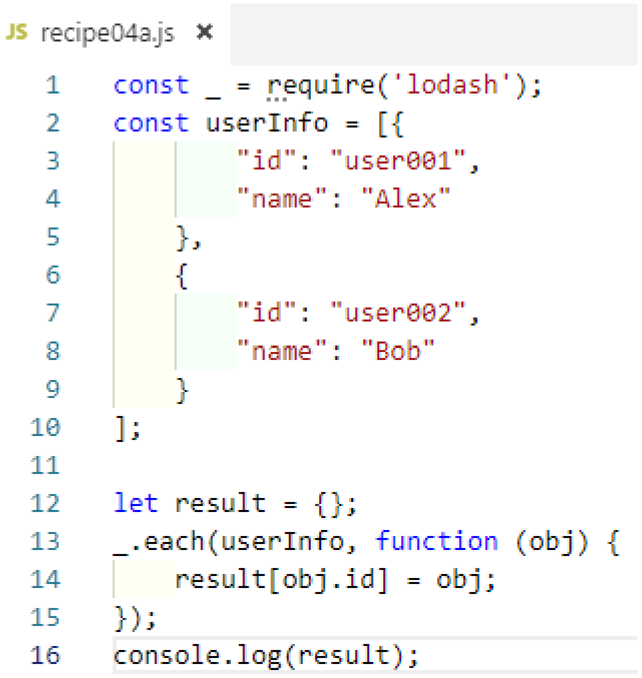
```
{
"user001": { "id": "user001", "name": "Alex" },
"user002": { "id": "user002", "name": "Bob" }
}
```

Lo

Recipes: Recipe 4. Convert an array to an object (cont.)

Scenario: Given an array of objects with IDs, convert the array to an object with IDs as the keys and array elements as the values.

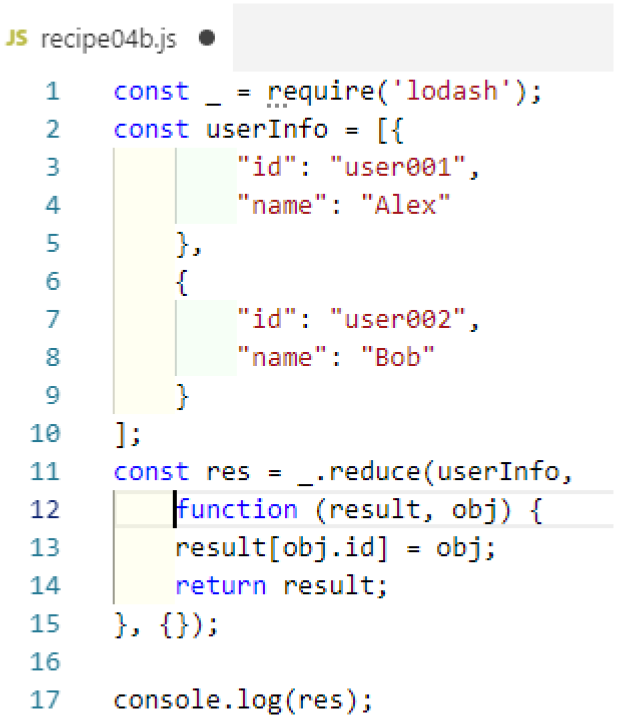
One solution is to use _.each to iterate the array and set each property in the result object



recipe04a.js

```javascript
const _ = require('lodash');
const userInfo = [{
        "id": "user001",
        "name": "Alex"
    },
    {
        "id": "user002",
        "name": "Bob"
    }
];

let result = {};
_.each(userInfo, function (obj) {
    result[obj.id] = obj;
});
console.log(result);
```

A better solution is to use _.reduce



recipe04b.js

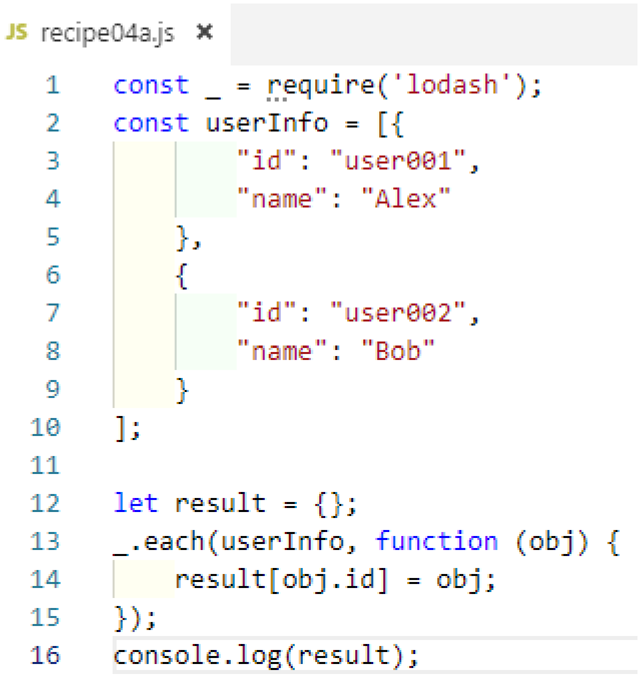
```javascript
const _ = require('lodash');
const userInfo = [{
        "id": "user001",
        "name": "Alex"
    },
    {
        "id": "user002",
        "name": "Bob"
    }
];
const res = _.reduce(userInfo,
    function (result, obj) {
    result[obj.id] = obj;
    return result;
}, {});

console.log(res);
```

Lo

Recipes: Recipe 4. Convert an array to an object (cont.)

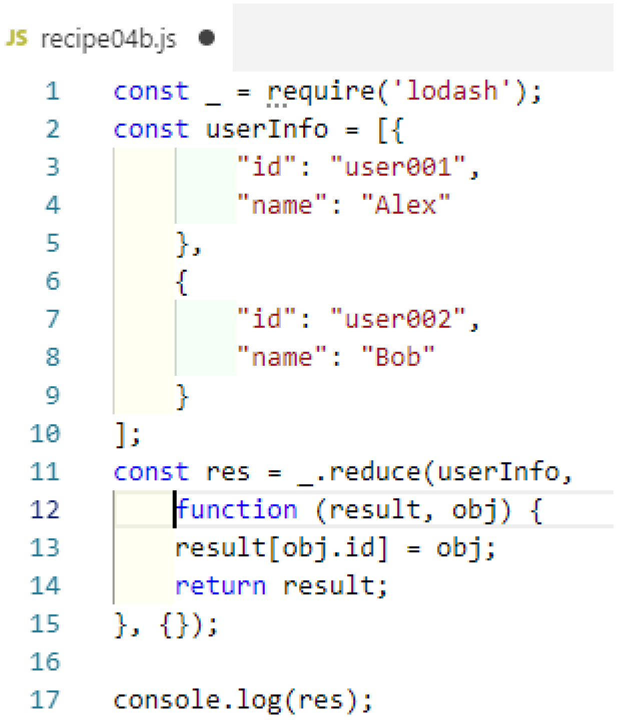
Scenario: Given an array of objects with IDs, convert the array to an object with IDs as the keys and array elements as the values.

One solution is to use _.each to iterate the array and set each property in the result object


recipe04a.js

```javascript
const _ = require('lodash');
const userInfo = [{
        "id": "user001",
        "name": "Alex"
    },
    {
        "id": "user002",
        "name": "Bob"
    }
];

let result = {};
_.each(userInfo, function (obj) {
    result[obj.id] = obj;
});
console.log(result);
```

A better solution is to use _.reduce


recipe04b.js

```javascript
const _ = require('lodash');
const userInfo = [{
        "id": "user001",
        "name": "Alex"
    },
    {
        "id": "user002",
        "name": "Bob"
    }
];
const res = _.reduce(userInfo,
    function (result, obj) {
    result[obj.id] = obj;
    return result;
}, {});

console.log(res);
```