**Lodash**

In this project, you will be implementing some of the most exciting functionality from the widely-popular [lodash.js](https://lodash.com/docs/4.17.10/) library. You will be implementing ten methods that add new functionality for numbers, strings, objects, and arrays.

In implementing each method, we will follow these four steps:

1. **Specify** the functionality of the method we are implementing
2. **Ideate** a game plan for how to implement this functionality in code
3. **Implement** our game plan
4. **Test** our code to ensure it works as expected

We encourage you to try to complete the "Ideate" and "Implement" steps on your own before consulting our suggestions for each. It may be difficult at points, but working through difficult problems on your own will be incredibly helpful in your journey to become a stronger developer. Once you've come up with a solution on your own, or if you have become so stuck you are no longer being productive, check out our steps to see our suggestions for how to solve the problem.

There is no right or wrong answer when it comes to programming. As a result, don't be frustrated if the solution we present is different than the solution you came up with. We are merely trying to challenge you to consider many different solutions. Your solution is equally as valid as ours. Consider the one you were going to write and then consider ours. Whichever you pick in the end is completely your decision, and we support it completely.

You have the choice of writing this project within the Codecademy environment to the right or locally on your own computer by downloading the starting code [here](https://s3.amazonaws.com/codecademy-content/programs/programming-with-javascript/lodash.zip). Feel free to proceed in whichever environment you are most comfortable with.

With all of that said, let's get started implementing some awesome new functionality!

Mark the tasks as complete by checking them off

**Create Lodash Object**

1.

Before we get started implementing our new methods, we need to create an object to contain them. This object will represent our library containing all the functionality we add to it.

Create a new variable called \_ that is initialized to an empty object.

This is an important step, so we want to make sure you start on the right foot. This is what your code should currently look like:

const \_ = {};

2.

We've written test files for each task in this project. Let's run the first test suite to ensure your lodash object was initialized correctly.

To run a file using node, we type the node command in a command line followed by the name of the file. For example, to run the main file we are working on, we would run node \_.js.

Our test files are all located in the **test/** directory. To run the test suite for this task, type node test/lodash.js in your terminal and then press enter. The test will either throw errors if something is not currently working properly in your code or will print a success message to the console if your code is good to go.

**Implement \_.clamp()**

3.

**Specify**: The first method we will implement is [.clamp()](https://lodash.com/docs/4.17.10#clamp). Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work.

Here is a summary of the method:

* .clamp() takes three arguments: a number, a lower bound, and an upper bound
* .clamp() modifies the provided number to be within the two provided bounds
* If the provided number is smaller than the lower bound, it will return the lower bound as the final number
* If the number is larger than the upper bound, it will return the upper bound as the final number
* If the number is already within the two bounds, it will return the number as the final number

You can see a diagram demonstrating this functionality [here](https://s3.amazonaws.com/codecademy-content/programs/programming-with-javascript/lodash-assets/clamp.svg).

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

4.

**Ideate**: There are a number of different ways to implement this method. One that might have come to your mind would be to use control flow to compare the current value and to the bounds and return the proper result. We are going to present a different solution in these steps so that you can keep considering unexpected ways to solve problems.

1. Add the .clamp() method to the lodash object.
2. Use Math.max() to clamp the number by the lower bound. The return value of Math.max() called with the number and the lower bound will be the larger of the two values, meaning it will be clamped by the lower bound.
3. Use Math.min() to clamp the number by the upper bound. The return value of Math.min() called with the number and the upper bound will be the smaller of the two.
4. Return the final value of these two operations, which will be the clamped number.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

5.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called clamp.
2. Add three parameters to this method: number, lower, and upper.
3. Within the method, create a variable called lowerClampedValue that is set equal to the return value of Math.max() called with number and lower.
4. Create a variable called clampedValue that is set equal to the return value of Math.min() called with lowerClampedValue and upper.
5. Return clampedValue as our final value from the method.

Once you've finished implementing this method, move on to the next step to test it.

6.

**Test**: To test that our .clamp() method works as expected, run node test/clamp.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Once all of the tests are passing, congratulate yourself! You've implemented the first method of this project! This is very exciting. Hopefully, you're beginning to feel like a developer.

When you're ready, move on to the next method.

**Implement .inRange()**

7.

**Specify**: The next number method we will implement is [.inRange()](https://lodash.com/docs/4.17.10#inRange). Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work.

Here is a summary of the method:

* .inRange() takes three arguments: a number, a start value, and an end value
* .inRange() checks to see if the provided number falls within the range specified by the start and end values
* If the provided number is smaller than the start value, .inRange() will return false
* If the provided number is larger than or equal to the end value, .inRange() will return false
* If the provided number is within the start and end values, .inRange() will return true
* If no end value is provided to the method, the start value will be 0 and the end value will be the provided start value
* If the provided start value is larger than the provided end value, the two values should be swapped

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

8.

**Ideate**: As always, there are a number of different solutions that could work to solve this problem. This is just one solution.

1. Add the .inRange() method to the lodash object.
2. Check to see if the end value is undefined. If so, set the end value equal to the start value and then set the start value equal to 0.
3. Check to see if the start value is larger than the end value. If so, swap the two values. Note: We will need to use a temporary variable to do this. To understand why, imagine if we tried to swap values without one. We might start by setting the end value equal to the start value. When we then go to set the start value equal to the end value, the end value will have already been overwritten and the swap can't be completed. To solve this, we create a variable that will temporarily store the end value to access when we need to set the new start value and complete the swap.
4. Using boolean operators, find out if the number is in the specified range.
5. Return the value of the previous operation from the method.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

9.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called inRange.
2. Add three parameters to this method: number, start, and end.
3. Within the method, create an if statement that checks to see if end is undefined.
4. Within the if statement block, set end equal to start. Then set start equal to 0.
5. After the previous if statement, add another if statement. This if statement should check whether start is bigger than end.
6. Within the if statement block, swap the two start and end values. Create a variable called temp that is set to the current end value. Then set end equal to start. Finally, set start equal to temp.
7. After our second if statement, create a variable called isInRange and set it equal to a boolean expression that checks if start is smaller than or equal to number and if number is smaller than end.
8. Finally, return the value of isInRange from the method.

Once you've finished implementing this method, move on to the next step to test it.

10.

**Test**: To test that our .inRange() method works as expected, run node test/in-range.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Congratulations, you've finished implementing all of the number methods!

When you're ready, move on to the next method.

**Implement .words()**

11.

**Specify**: Let's start implementing some string methods! The first string method we will implement is [.words()](https://lodash.com/docs/4.17.10#words). We will be writing a slightly modified version of this method to save you some time. Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work. Then read below to see which pieces of functionality you will implement.

Here is a summary of what your method should do:

* .words() takes one argument: a string
* .words() splits the string into an array of words
* A word is defined by a space-separated string of characters, so each space character, ' ', indicates the end of one word and the start of the next
* Note: You may have noticed in the documentation that this function has a pattern parameter. Your method does not need to accept the additional pattern parameter, we will only split our string into words based on spaces

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

12.

**Ideate**: This solution probably has the most potential solutions of the methods we have implemented thus far. We've opted to use a built-in JavaScript method to make this method as short and readable as possible.

1. Add the .words() method to the lodash object.
2. Use the string [.split()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/split) method to split the provided string on space characters into an array of words.
3. Return the array of words generated in the previous step.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

13.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called words.
2. Add one parameter to this method: string.
3. Within the method, create a variable called words and set its value equal to string split on space characters ' ' using the .split() method.
4. Return the value of words from the method.

Once you've finished implementing this method, move on to the next step to test it.

14.

**Test**: To test that our .words() method works as expected, run node test/words.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Congratulations! You've finished implementing your first string method.

When you're ready, move on to the next method.

**Implement .pad()**

15.

**Specify**: The next string method we will implement is [.pad()](https://lodash.com/docs/4.17.10#pad). We will be writing a slightly modified version of this method to save you some time. Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work. Then read below to see which pieces of functionality you will implement.

Here is a summary of what your method should do:

* .pad() takes two arguments: a string and a length
* .pad() adds spaces evenly to both sides of the string to make it reach the desired length
* Extra padding is added to the end of the string if an odd amount of padding is required to reach the specified length
* Your method does not need to accept the additional chars parameter; we will only add space characters to pad our string

You can see a diagram demonstrating this functionality [here](https://s3.amazonaws.com/codecademy-content/programs/programming-with-javascript/lodash-assets/pad.svg).

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

16.

**Ideate**: Again, as noted with previous game plans, there are many solutions to this problem. We chose this one for readability.

1. Add the .pad() method to the lodash object.
2. Check to make sure the target length is longer than the current string length. If not, return the unpadded version of the string.
3. Find the amount of padding to add to the start of the string by finding the difference between the target length and the string length, dividing by two, and rounding down the resulting number. We round down so that any uneven padding gets added to the end of the string, not the beginning, as specified in the instructions.
4. Find the amount of padding to add to the end of the string by subtracting the string length and the starting padding length (calculated above) from the target length.
5. Generate the padded string by adding the amount of starting padding and ending padding calculated above to each side of the current string.
6. Return the padded version of the string.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

17.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called pad.
2. Add two parameters to this method: string and length.
3. Within the method, add an if statement that checks if length is shorter than or equal to string's length. If so, return string.
4. Create a variable called startPaddingLength and set its value equal to the difference of length and string's length, divided by 2, and rounded down by using Math.floor().
5. Create a variable called endPaddingLength and set its value equal to length minus string's length minus startPaddingLength.
6. Create a new variable called paddedString and set its value equal to the space character, ' ', repeated startPaddingLength number of times (using the string .repeat() method), concatenated with string, concatenated with the space character repeated endPaddingLength number of times.
7. Return the value of paddedString from the method.

Once you've finished implementing this method, move on to the next step to test it.

18.

**Test**: To test that our .pad() method works as expected, run node test/pad.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Congratulations, you've finished implementing all of the new string methods!

When you're ready, move on to the next method.

**Implement \_.has()**

19.

**Specify**: Let's begin implementing some new object methods! The first object method we will implement is [.has()](https://lodash.com/docs/4.17.10#has). We will be writing a slightly modified version of this method to save you some time. Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work. Then read below to see which pieces of functionality you will implement.

Here is a summary of what your method should do:

* .has() takes two arguments: an object and a key
* .has() checks to see if the provided object contains a value at the specified key
* .has() will return true if the object contains a value at the key and will return false if not
* Your method does not need to accept the additional path parameter; we will only check for unnested values

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

20.

**Ideate**: Let's come up with a game plan for implementing this method.

1. Add the .has() method to the lodash object.
2. Access the current value at the specified key in the object.
3. Check to see if the value at that key is undefined.
4. Return true if the value is undefined and false if not.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

21.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called has
2. Add two parameters to this method: object and key
3. Within the method, create a variable called hasValue and set its value equal to the result of checking to see if the current value of object at key does not equal undefined.
4. Return the value of hasValue from the method.

Once you've finished implementing this method, move on to the next step to test it.

22.

**Test**: To test that our .has() method works as expected, run node test/has.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Congratulations! You've finished implementing the first new object method.

When you're ready, move on to the next method.

**Implement \_.invert()**

23.

**Specify**: The next object method we will implement is [.invert()](https://lodash.com/docs/4.17.10#invert). Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work.

Here is a summary of what your method should do:

* .invert() takes one argument: an object
* .invert() iterates through each key / value pair in the provided object and swaps the key and value

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

24.

**Ideate**: Let's come up with a game plan for implementing this method.

1. Add the .invert() method to the lodash object.
2. Create a new object to represent our inverted object.
3. Iterate through each key in the provided object.
4. Set the original object's value at that key to be a key on our inverted object and set the value of that key to be the original object's key.
5. Return the inverted object.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

25.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called invert.
2. Add one parameter to this method: object.
3. Within the method, create a variable called invertedObject and set its value equal to an empty object.
4. Using a for ... in loop, iterate through each key in object.
5. Within the loop, create a variable called originalValue and set it equal to the value at the current key in object.
6. Still within the loop, set the value at originalValue on invertedObject to be the current key.
7. Finally, outside of the loop, return the value of invertedObject from the method.

Once you've finished implementing this method, move on to the next step to test it.

26.

**Test**: To test that our .invert() method works as expected, run node test/invert.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Congratulations! When you're ready, move on to the next method.

**Implement \_.findKey()**

27.

**Specify**: The final object method we will implement is [.findKey()](https://lodash.com/docs/4.17.10#findKey). Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work.

Here is a summary of what your method should do:

* .findKey() takes two arguments: an object and a *predicate function* — a function that returns a boolean value
* .findKey() iterates through each key / value pair in the provided object and calls the predicate function with the value
* .findKey() returns the first key that has a value that returns a truthy value from the predicate function
* .findKey() returns undefined if no values return truthy values from the predicate function

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

28.

**Ideate**: Let's come up with a game plan for implementing this method.

1. Add the .findKey() method to the lodash object.
2. Iterate through each key in the provided object.
3. Call the provided predicate function with the value at that key.
4. If the predicate function returns a truthy value, return the current key from the method.
5. After the loop, return undefined, since no values returned a truthy value from the predicate function.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

29.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called findKey.
2. Add two parameters to this method: object and predicate. We will name our predicate function parameter predicate since this is the name used in the Lodash documentation.
3. Within the method, use a for ... in loop to iterate through each key in object.
4. Within the loop, create a variable called value and set it equal to the value at the current key in object.
5. Still within the loop, create another variable called predicateReturnValue and set it equal to the result of calling predicate with value.
6. Finally, still within the loop, use an if statement to check to see if predicateReturnValue is truthy. If it is, return the current key from the method.
7. Outside of the loop, return undefined to address all cases where no truthy values were returned from predicate.

Once you've finished implementing this method, move on to the next step to test it.

30.

**Test**: To test that our .findKey() method works as expected, run node test/find-key.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Congratulations, you've implemented all of the object methods! These are starting to get a little tricky. If you're feeling overwhelmed at all, that's normal. Just keep tackling these problems one at a time, and you'll soon find that you'll be able to tackle problems like these faster and faster.

When you're ready, move on to the next method.

**Implement \_.drop()**

31.

**Specify**: It's time to start implementing methods for our final data type: arrays! The first array method we will implement is [.drop()](https://lodash.com/docs/4.17.10#drop). Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work.

Here is a summary of what your method should do:

* .drop() takes two arguments: an array and a number representing the number of items to drop from the beginning of the array
* .drop() returns a new array which contains the elements from the original array, excluding the specified number of elements from the beginning of the array
* If the number of elements to drop is unspecified, your method should drop one element

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

32.

**Ideate**: Let's come up with a game plan for implementing this method.

1. Add the .drop() method to the lodash object.
2. Check to see if the number of items to drop was set. If not, set the number equal to 1.
3. Create a new copy of the original array with the specified number of elements dropped from the beginning of the array. We will use the array .slice() method to accomplish this.
4. Return the modified copy of the array from the method.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

33.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called drop.
2. Add two parameters to this method: array and n.
3. Within the method, use an if statement to check if n has been set.
4. Within the if statement block, set n equal to 1.
5. Outside of the if statment, create a new variable called droppedArray and set its value to be a copy of the array missing the first n elements by using .slice().
6. Return droppedArray from the method.

Once you've finished implementing this method, move on to the next step to test it.

34.

**Test**: To test that our .drop() method works as expected, run node test/drop.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Congratulations, you've implemented the first array method!

When you're ready, move on to the next method.

**Implement \_.dropWhile()**

35.

**Specify**: The next array method we will implement is [.dropWhile()](https://lodash.com/docs/4.17.10#dropWhile). Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work.

Here is a summary of what your method should do:

* .dropWhile() takes two arguments: an array and a predicate function
* The supplied predicate function takes three arguments: the current element, the current element index, and the whole array
* .dropWhile() creates a new copy of the supplied array, dropping elements from the beginning of the array until an element causes the predicate function to return a falsy value

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

36.

**Ideate**: Let's come up with a game plan for implementing this method.

1. Add the .dropWhile() method to the lodash object.
2. Iterate through the array until you find an element that causes the predicate to return a falsy value. We will use [.findIndex()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/findIndex) to iterate through the array since it is built to iterate through an array until it finds an element that returns a specific value.
3. Use our previous .drop() method to drop the elements prior to the one that returned a falsy value.
4. Return the modified copy of the array from the method.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

37.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called dropWhile.
2. Add two parameters to this method: array and predicate.
3. Within the method, create a new variable called dropNumber and set its value equal to the return value of a call to findIndex on array.
4. Pass an anonymous callback function to findIndex that takes two arguments: element and index.
5. Within the callback function, return the negated return value of predicate called with element, index, and array. We negate the value (use !) since we are looking to drop elements until the predicate returns a falsy value. However, .findIndex() is looking for the first truthy value. So, we make every truthy value falsy and vice versa to get the value we are looking for.
6. After the entire dropNumber declaration, create a new variable called droppedArray and set its value to the return value of this.drop() called with dropNumber. We are using this since .drop() is a method on the \_ object which is the current object we are working from, and therefore the current value of this. Calling \_.drop() would also have worked but is a less common practice.
7. Return droppedArray from the method.

Once you've finished implementing this method, move on to the next step to test it.

38.

**Test**: To test that our .dropWhile() method works as expected, run node test/drop-while.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Congratulations! This method wasn't especially long, but it used a lot of advanced concepts. Great job working through it!

When you're ready, move on to the next method.

**Implement \_.chunk()**

39.

**Specify**: The last array method we will implement is [.chunk()](https://lodash.com/docs/4.17.10#chunk). Read the explanation and examples of the method in the linked documentation to get a sense of how the method should work.

Here is a summary of what your method should do:

* .chunk() takes two arguments: an array and a size
* .chunk() breaks up the supplied array into arrays of the specified size
* .chunk() returns an array containing all of the previously-created array chunks in the order of the original array
* If the array can't be broken up evenly, the last chunk will be smaller than the specified size
* If no size is supplied to the method, the size is set to 1

You can see a diagram demonstrating this functionality [here](https://s3.amazonaws.com/codecademy-content/programs/programming-with-javascript/lodash-assets/chunk.svg).

When you've ideated a game plan for how to implement this functionality, move on to the next step to see how we plan to do it.

40.

**Ideate**: Let's come up with a game plan for implementing this method.

1. Add the .chunk() method to the lodash object.
2. Check to see if a size has been supplied. If not, set the size equal to 1.
3. Create an empty array that will contain all of the generated array chunks.
4. Loop through the array. In each turn of the loop, create an array chunk of the specified size, add it to the final array, and increase the loop counter by the chunk size to go to the next chunk. We will use a for loop to do this, since no iterator method does exactly what we want and a while loop won't auto-increment.
5. Return the array of array chunks from the method.

Once you have tried implementing this game plan in code, move on to the next step to see how we do it.

41.

**Implement**: Let's implement our game plan in code.

1. Add a method to our \_ object called chunk.
2. Add two parameters to this method: array and size.
3. Within the method, write an if statement that checks to see if size is undefined.
4. Within the if statement block, set size equal to 1.
5. After the if statement, create a variable called arrayChunks and initialize it to an empty array.
6. Write a for loop that loops through array and has a counter that increments by size each turn of the loop.
7. Within the for loop, create a variable called arrayChunk and set it equal to the chunk of the array going from the current loop index to the current loop index plus size. You an use .slice() to accomplish this.
8. Still within the for loop, add arrayChunk to the end of arrayChunks. You can use .push() to accomplish this.
9. Finally, outside of the for loop, return arrayChunks from the method.

Once you've finished implementing this method, move on to the next step to test it.

42.

**Test**: To test that our .chunk() method works as expected, run node test/chunk.js in your terminal. Don't worry if any errors appear, work through them one by one until your code runs as expected.

Congratulations, you've implemented all of the methods of this project! This is a huge accomplishment.

We hope you now have the confidence to go out and build your own multi-million-download open-source library! It may seem like a crazy statement at this stage in your coding journey, but you've demonstrated you have the skills to do it.

What's next for this project? We suggest you try implementing even more methods from the lodash library and address additional edge cases for the methods you wrote. We also encourage you to clean up your code by following the [Google JavaScript style guidelines](https://google.github.io/styleguide/jsguide.html). Making sure your code is easy to read and follows best practices is a great way to set yourself apart as a developer.

Take a moment to congratulate yourself on how far you've come. You truly deserve it!

Report a Bug

If you see a bug or any other issue with this page, please report it [here](https://www.codecademy.com/programs/522912ef246eb1f1d7bd3a2dc1679dd5/content-items/fb6220182ee4317cc3eaf380b71175de).