7.1. Strings as Collections

Throughout the first chapters of this book we have used strings to represent words or phrases we wanted to print out. Our definition of a string was simple: a string is a sequence of characters inside quotes.

In this chapter we explore strings in much more detail. Strings come with a special group of operations that can be carried out on them, known as methods. Strings are also what is called a collection data type. Let's look at what this means.

7.1.1. Collection Data Types

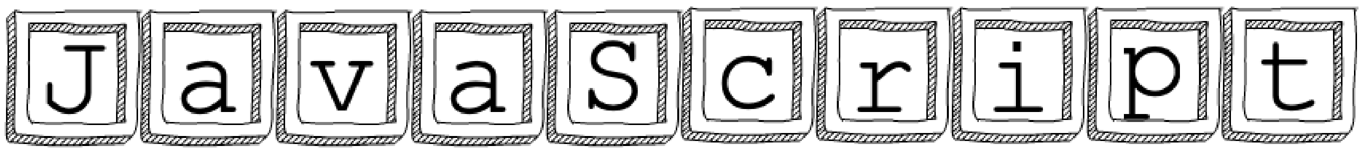
Data types that are comprised of smaller pieces are called **collection data types**, or simply **collection types**. Depending on what we are doing, we may want to treat a value of a collection data type as a single entity (the whole collection), or we may want to access its parts.

A **character** is a string that contains exactly one element, such as `'a'`, `"?"`, or even `" "` (a single space character).

Note

Some programming languages, such as Java and C, represent characters using their own data type. For example, Java has the data type `char`. JavaScript, however, does not consider strings and characters to be different types.

We can think of strings as being built out of characters. In this way, strings can be broken down into smaller pieces.



A string is made up of characters, which are strings of length 1.

Strings are made up of smaller pieces, characters. Other data types, like `number` and `boolean`, are not composed of any smaller parts.

7.1.2. Ordered Collections

We defined strings as *sequential* collections of characters. This means that the individual characters that make up the string are assumed to be in a particular order from left to right. The string `"LaunchCode"` is different from the string `"CodeLaunch"`, even though they contain the exact same characters.

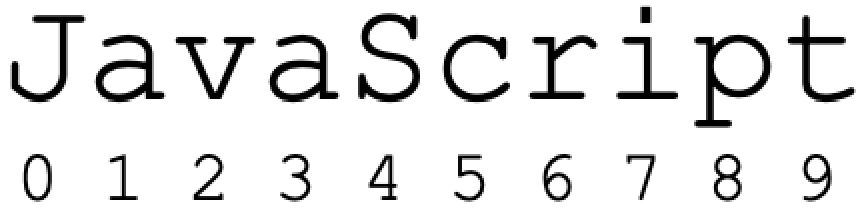
Collection types that allow their elements to be ordered are known as **ordered collections**, for reasons that will become clear to you very soon.

7.2. Bracket Notation

Understanding strings as sequential collections of characters gives us much more than just a mental model of how they are structured. JavaScript provides a rich collection of tools—including special syntax and operations—that allows us to work with strings.

Bracket notation is the special syntax that allows us to access the individual characters that make up a string. To access a character, we use the syntax `someString[i]`, where `i` is the **index** of the character we want to access. String indices are integers representing the position of a character within a given string, and they start at 0. Thus, the first character of a string has index 0, the second has index 1, and so on.

Consider the string `"JavaScript"`. The `"J"` has index 0, the first `"a"` has index 1, `"v"` has index 2, and so on.



The indices of a string.

An expression of the form `someString[i]` gives the character at index `i`.

Example

This program prints out the initials of the person's name.

```
let jsCreator = "Brendan Eich";
let firstInitial = jsCreator[0];
let lastInitial = jsCreator[8];

let outputStr = "JavaScript was created by somebody with initials " +
   firstInitial + "." +
   lastInitial + ".";

console.log(outputStr);
```

Console Output

```
JavaScript was created by somebody with initials B.E.
```

What happens if we try to access an index that doesn't exist, for example -1 or an index larger than the length of the string?

Try It!

```
let jsCreator = "Brendan Eich";

console.log(jsCreator[-1]);
console.log(jsCreator[42]);
```

repl.it

Question

What does an expression using bracket notation evaluate to when the index is invalid (the index does not correspond to a character in the string)?

7.2.1. Check Your Understanding

Question

If `phrase = 'Code for fun'`, then `phrase[2]` evaluates to:

a. `"o"`
b. `"d"`
c. `"for"`
d. `"fun"`

Question

Which of the following returns `true` given `myStr = 'Index'`? Choose all correct answers.

a. `myStr[2] === 'n';`
b. `myStr[4] === 'x';`
c. `myStr[6] === ' ';`
d. `myStr[0] === 'I';`

Question

What is printed by the following code?

```
let phrase = "JavaScript rocks!";
console.log(phrase[phrase.length - 8]);
```

a. `"p"`
b. `"i"`
c. `"r"`
d. `"t"`

7.3. Strings as Objects

Beyond bracket notation, there are many other tools we can use to work with strings. Talking about these tools requires some new terminology.

7.3.1. Object Terminology

In JavaScript, strings are objects, so to understand how we can use them in our programs, we must first understand some basics about objects.

An **object** is a collection of related data and operations. An operation that can be carried out on an object is known as a **method**. A piece of data associated with an object is known as a **property**.

Example

Suppose we had a `square` object in JavaScript. (While no such object is built into JavaScript, we will learn how we could make one in a later chapter.)

Since a square has four sides of the same length, it should have a property to represent this length. This property could be called `length`. For a given square, it will have a specific value, such as 4.

Since a square has an area, it should have a method to calculate the area. This method could be called `area`, and it should calculate the area of a square using its `length` property.
You can think of methods and properties as functions and variables, respectively, that "belong to" an object. Properties and methods are accessed using **dot notation**, which dictates that we use the object name, followed by a `.`, followed by the property or method name. When using a method, we must also use parentheses as we do when calling regular functions.

Example

Returning to the `square` example, we can access its length by typing `square.length`.

We can calculate the area by calling `square.area()`.
Referencing `length` or `area` by itself in code *does not* give you the value of `square.length` or carry out the calculation in `square.area()`. It does not make sense to refer to a property or method without also referring to the associated object. Typing simply `length` or `area()` is ambiguous. There might be multiple squares, and it would be unclear which one you were asking about.

Example

We have already encountered one object, the built-in object `console`, which we use to output messages.

```
console.log(typeof console);
```

Console Output

```
object
```

JavaScript reports that the type of `console` is indeed `object`.

When calling `console.log`, we are calling the `log` method of the `console` object.
We will learn quite a bit more about objects in this course, including how to use objects to create your own custom data types. This powerful JavaScript feature allows us to bundle up data and functionality in useful, modular ways.

7.3.2. Strings Are Objects

The fact that strings are objects means that they have associated data and operations, or properties and methods as we will call them from now on.

Every string that we work with will have the same properties and methods. The most useful string property is named `length`, and it tells us how many characters are in a string.

Example

```
let firstName = "Grace";
let lastName = "Hopper";

console.log(firstName, "has", firstName.length, "characters");
console.log(lastName, "has", lastName.length, "characters");
```

Console Output

```
Grace has 5 characters
Hopper has 6 characters
```

Every string has a `length` property, which is an integer.

The `length` property is the only string property that we will use, but there are many useful string methods. We will explore these in depth in the section String Methods, but let's look at one now to give you an idea of what's ahead.

The `toLowerCase()` string method returns the value of its string in all lowercase letters. Since it is a method, we must precede it with a specific string in order to use it.

Example

```
let nonprofit = "LaunchCode";

console.log(nonprofit.toLowerCase());
console.log(nonprofit);
```

Console Output

```
launchcode
LaunchCode
```

Notice that `toLowerCase()` does not alter the string itself, but instead *returns* the result of converting the string to all lowercase characters. In fact, it is not possible to alter the characters within a string, as we will now see.

7.3.3. Check Your Understanding

Question

Given `word = 'Rutabaga'`, why does `word.length` return the integer 8, but `word[8]` is `undefined`?

7.4. String Immutability

If an object cannot be changed, we say that it is **immutable**. Strings are immutable, which means we cannot change the individual characters within a given string. While we can access individual characters using bracket notation, attempting to change individual characters simply does not work.

Example

```
1 let nonprofit = "Launchcode";
2
3 console.log(nonprofit);
4 nonprofit[6] = "C";
5 console.log(nonprofit);
```

Console Output

```
Launchcode
Launchcode
```

We attempted to change the value of the character at index 6 from `'c'` to `'C'`, by using an assignment statement along with bracket notation on line 4 (perhaps to align with official LaunchCode branding guidelines). However, this change clearly did not take place. In many programming languages strings are immutable, and while trying to change a string in some languages results in an error, JavaScript simply ignores our request to alter a string.

It is important to notice that immutability applies to string *values* and not string variables.

Example

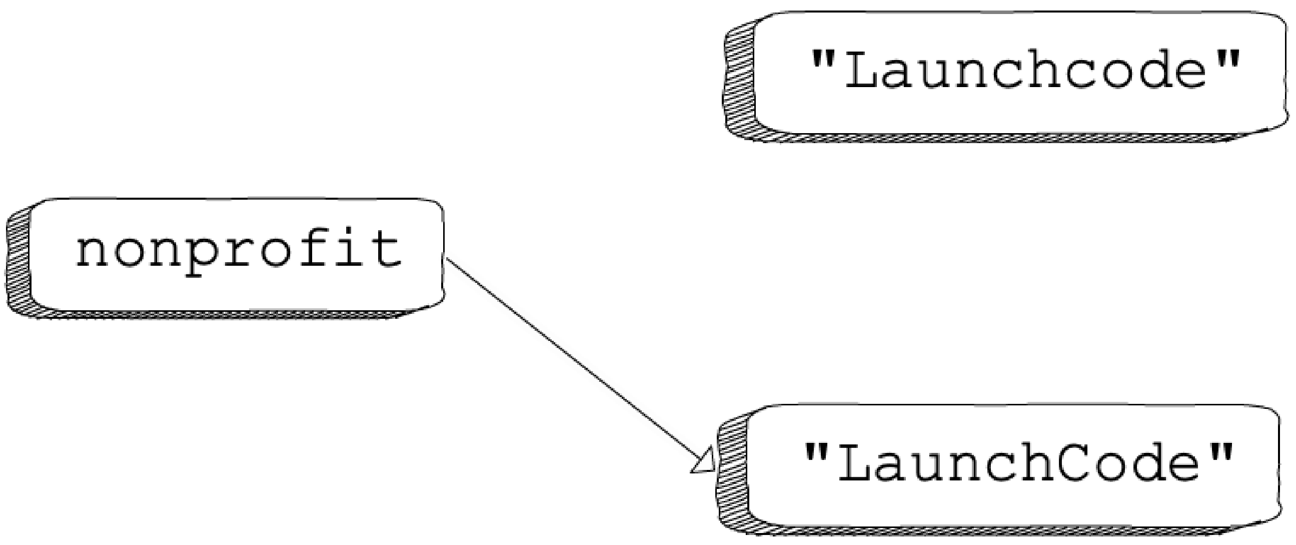
We can set a variable containing a string to a different value.

```
1 let nonprofit = "Launchcode";
2 nonprofit = "LaunchCode";
3
4 console.log(nonprofit);
```

Console Output

```
LaunchCode
```

In this example, the change made on line 2 is carried out. The difference between this example and the one above is that here we are modifying the value that the variable is storing, and not the string itself. Using our visual analogy of a variable as a label that “points at” a value, the second example has the following effect:



When the value of a variable storing a string is changed, the variable then points to a new value, with the old value remaining unchanged.

7.4.1. Check Your Understanding

Question

Given `pet = 'cat'`, why do the statements `console.log(pet + 's');` and `pet += 's';` NOT violate the immutability of strings?

7.5. String Methods

JavaScript provides many useful methods for string objects. Recall that a method is a function that "belongs to" a specific object. Methods will typically result in some operation being carried out on the data within an object. For strings, this means that our methods will typically transform the characters of the given string in some way.

As we have learned, strings are immutable. Therefore, string methods will not change the value of a string itself, but instead will *return* a new string that is the result of the given operation.

We saw this behavior in the `toLowerCase` example.

Example

```
let nonprofit = "LaunchCode";

console.log(nonprofit.toLowerCase());
console.log(nonprofit);
```

Console Output

```
launchcode
LaunchCode
```

While `nonprofit.toLowerCase()` evaluated to `"launchcode"`, the value of `nonprofit` was left unchanged. This will be case for each of the string methods.

7.5.1. Common String Methods

Here we present the most commonly-used string methods. You can find documentation for other string methods at:

- W3Schools
- MDN

Common String Methods

Method	Syntax	Description
indexOf	`stringName.indexOf(substr)`	Returns the index of the first occurrence of the substring in the string, and returns -1 if the substring is not found.
toLowerCase	`stringName.toLowerCase()`	Returns a copy of the given string, with all uppercase letters converted to lowercase.

Common String Methods

Method	Syntax	Description
toUpperCase	`stringName.toUpperCase()`	Returns a copy of the given string, with all lowercase letters converted to uppercase.
trim	`stringName.trim()`	Returns a copy of the given string with the leading and trailing whitespace removed.
replace	`stringName.replace(searchChar, replacementChar)`	Returns a copy of `stringName`, with the first occurrence of `searchChar` replaced by `replacementChar`.
slice	`stringName.slice(i, j)`	Return the substring consisting of characters from index `i` through index `j-1`.

Tip

String methods can be combined in a process called **method chaining**. Given `word = 'JavaScript';`, word.toUpperCase() returns `JAVASCRIPT`. What would `word.slice(4).toUpperCase()` return? Try it at repl.it.

7.5.2. Check Your Understanding

Follow the links in the table above for the `replace`, `slice`, and `trim` methods. Review the content and then answer the following questions.

Question

What is printed by the following code?

```
let language = "JavaScript";
language.replace('J', 'Q');
language.slice(0,5);
console.log(language);
```

a. `"JavaScript"`
b. `"QavaScript"`
c. `"QavaSc"`
d. `"QavaS"`

Question

Given `language = 'JavaScript';`, what does `language.slice(1,6)` return?

a. `"avaScr"`
b. `"JavaSc"`
c. `"avaSc"`
d. `"JavaS"`

Question

What is the value of the string printed by the following program?

```javascript
let org = "  The LaunchCode Foundation ";
let trimmed = org.trim();

console.log(trimmed);
```

a. `"  The LaunchCode Foundation "`
b. `"The LaunchCode Foundation"`
c. `"TheLaunchCodeFoundation"`
d. `" The LaunchCode Foundation"`

7.6. Encoding Characters

If you had microscope powerful enough to view the data stored on a computer's hard drive, or in its memory, you would see lots of 0s and 1s. Each such 0 and 1 is known as a **bit**. A bit is a unit of measurement, like a meter or a pound. Collections of computer data are measured in bits; every letter, image, and pixel you interact with on a computer is represented by bits.

We work with more complex data when we program, including numbers and strings. This section examines how such data is represented within a computer.

7.6.1. Representing Numbers

A **byte** is a set of 8 bits. Bytes look like 00101101 or 11110011, and they represent a **binary number**, or a base-2 number. A binary number is a number representation that uses only 0s and 1s. The numbers that you are used to, which are built out of the integers 0…9, are **decimal numbers**, or base-10 numbers.

Since each bit can have one of two values, each byte can have one of $2^8 = 256$ different values.

It may not be obvious, but every decimal integer can be represented as a binary integer, and vice versa. There are 256 different values a byte may take, each of which can be used to represent a decimal integer, from 0 to 255.

Note

We will not go into binary to decimal number conversion. If you are interested in learning more, there are many tutorials online that can show you the way.
In this way, the bits in a computer can be viewed as integers. If you want to represent values greater than 255, just use more bits!

7.6.2. Representing Strings

Strings are collections of characters, so if we can represent each character as a number, then we'll have a way to go from a string to a collection of bits, and back again.

7.6.2.1. Character Encodings

Unlike the natural translation between binary and decimal numbers, there is no natural translation between integers and characters. For example, you might create a pairing of 0 to `a`, 1 to `b`, and so on. But what integer should be paired with `$` or a tab? Since there is no natural way to translate between characters and integers, computer scientists have had to make such translations up. Such translations are called **character encodings**.

There are many different encodings, some of which continue to evolve as our use of data evolves. For instance, the most recent versions of the Unicode character encoding include emoji characters, such as 🌮.

7.6.2.2. The ASCII Encoding

Most of the characters that you are used to using—including letters, numbers, whitespace, punctuation, and symbols—are part of the **ASCII** (pronounced *ask-ee*) character encoding. This standard has changed very little since the 1960s, and it is the foundation of all other commonly-used encodings.

Note

ASCII stands for American Standard Code for Information Interchange, but most programmers never remember that, so you shouldn't try to either.
ASCII provides a standard translation of the most commonly-used characters to one of the integers 0…127, which means each character can be stored in a computer using a single byte.

ASCII maps `a` to 97, `b` to 98, and so on for lowercase letters, with `z` mapping to 122. Uppercase letters map to the values 65 through 90. The other integers between 0 and 127 represent symbols, punctuation, and other assorted odd characters. This scheme is called the **ASCII table**, and rather than replicate it here, we refer you to an excellent one online.

In summary, strings are stored in a computer using the following process:

1. Break a string into its individual characters.
2. Use a character encoding, such as ASCII, to convert each of the characters to an integer.
3. Convert each integer to a series of bits using decimal-to-binary integer conversion.

Fun Fact

JavaScript uses the UTF-16 encoding, which includes ASCII as a subset. We will rarely need anything outside of its ASCII subset, so we will usually talk about "ASCII codes" in JavaScript.

7.6.3. Character Encodings in JavaScript

JavaScript provides methods to convert any character into its ASCII code and back.

The string method `charCodeAt` takes an index and returns the ASCII code of the character at that index.

Example

```
let nonprofit = "LaunchCode";

for (let i = 0; i < nonprofit.length; i++) {
  console.log(nonprofit.charCodeAt(i));
}
```

Console Output

```
76
97
117
110
99
104
67
111
100
101
```

To convert an ASCII code to an actual character, use `String.fromCharCode()`.

Example

```
let codes = [76, 97, 117, 110, 99, 104, 67, 111, 100, 101];
let characters = "";

for (let i = 0; i < codes.length; i++) {
   characters += String.fromCharCode(codes[i]);
}

console.log(characters);
```

Console Output

```
LaunchCode
```

7.7. Special Characters

Aside from letters, numbers, and symbols, there is another class of characters that we will occasionally use in strings, known as **special characters**. These characters consist of special character combinations that all begin with `\` backslash). They allow us to include characters in strings that would be difficult or impossible to include otherwise, such as Unicode characters that are not on our keyboards, control characters, and whitespace characters.

The most commonly-used special characters are `\n` and `\t`, which are the newline and tab characters, respectively. They work as you would expect.

Example

```
console.log("A message\nbroken across lines,\n\tand indented");
```

Console Output

```
A message
broken across lines,
   and indented
```

We can also represent Unicode characters (most of which aren't on a normal keyboard) using special character combinations of the form `\uXXXX`, where the Xs are combinations referenced by the Unicode table. This allows us to use character sets other than the basic Latin characters that English is based on, such as Greek, Cyrillic, and Arabic, as well as a wider array of symbols.

Example

```
console.log("The interrobang character, \u203d, combines ? and !");
```

Console Output

```
The interrobang character, ‽, combines ? and !
```

We can also use the backslash, `\`, to include quotes within a string. This is known as **escaping** a character.

Example

```
console.log("\"The dog's favorite toy is a stuffed hedgehog,\" said Chris");
```

Console Output

```
"The dog's favorite toy is a stuffed hedgehog," said Chris
```

7.7.1. Check Your Understanding

Question

Which of the options below print `'Launch'` and `'Code'` on separate lines?

a. `console.log('Launch\nCode');`
b. `console.log('Launch/nCode');`
c. `console.log('Launch', 'Code');`
d. `console.log('Launch\tCode');`
e. `console.log('Launch/tCode');`

7.8. Template Literals

Earlier, we used *concatenation* to combine strings and variables together in order to create specific output:

Example

```
let name = Jack;
let currentAge = 9;

console.log("Next year, " + name + " will be " + (currentAge + 1) + ".");
```

Console Output

```
Next year, Jack will be 10.
```

Unfortunately, this process quickly gets tedious for any output that depends on multiple variables. Often, concatenation requires multiple test runs of the code in order to check for syntax errors and proper spacing within the output. Fortunately, JavaScript offers us a better way to accomplish this process.

Template literals allow for the automatic insertion of expressions (including variables) into strings.

While normal strings are enclosed in single or double quotes (`'` or `"`), template literals are enclosed in back-tick characters, `` ` ``. Within a template literal, any expression surrounded by `${ }` will be evaluated, with the resulting value included in the string.

Example

Template literals allow for variables and other expressions to be directly included in strings.

```
let name = "Jack";
let currentAge = 9;

console.log(`Next year, ${name} will be ${currentAge + 1}.`);
```

Console Output

```
Next year, Jack will be 10.
```

Besides allowing us to include data in strings in a cleaner, more readable way, template literals also allow us to easily create multi-line strings without using string concatenation or special characters.

Example

```
let poem = `The mind chases happiness.
The heart creates happiness.
The soul is happiness
And it spreads happiness
All-where.

– Sri Chinmoy`;

console.log(poem);
```

Console Output

```
The mind chases happiness.
The heart creates happiness.
The soul is happiness
```

```
And it spreads happiness
All-where.

- Sri Chinmoy
```

Note

The ECMAScript specifications define the standard for JavaScript. The 6th edition, known as ES2015, added template literals. Not only are template literals relatively new to JavaScript, but you may encounter environments—such as older web browsers—where they are not supported.

7.8.1. Check Your Understanding

Question

Mad Libs are games where one player asks the group to supply random words (e.g. “Give me a verb,” or, “I need a color”). The words are substituted into blanks within a story, which is then read for everyone’s amusement. In elementary school classrooms, giggles and hilarity often ensue. TRY IT!

Refactor the following code to replace the awkward string concatenation with template literals. Be sure to add your own choices for the variables.

```javascript
let pluralNoun = ;
let name = ;
let verb = ;
let adjective = ;
let color = ;

console.log("JavaScript provides a "+ color +" collection of tools — including " + adjective + " syntax and " + pluralNoun + " — that allows "+ name +" to "+ verb +" with strings.")
```

repl.it

7.9. Exercises: Strings

7.9.1. Part One

1. Identify the result for each of the following statements:

   a. `'JavaScript'[8]`
   b. `"Strings are sequences of characters."[5]`
   c. `"Wonderful".length`
   d. `"Do spaces count?".length`

   There's no code snippet for this one, just try it on your own with old-fashioned pen and paper!

2. The `length` method returns how many characters are in a string. However, the method will NOT give us the length of a number. If `num = 1001`, `num.length` returns `undefined` rather than 4.

   a. Use type conversion to print the length (number of digits) of an integer.
   b. Print the number of digits in a DECIMAL value (e.g. `num = 123.45` has 5 digits but a length of 6).
      i. Modify your code to print out the length of a decimal value EXCLUDING the period.
   c. What if `num` could be EITHER an integer or a decimal? Add an `if/else` statement so your code can handle both cases. (Hint: Consider the `indexOf()` or `includes()` string methods).

   Code it at repl.it

7.9.2. Part Two

1. Remember, strings are *immutable*. Consider a string that represents a strand of DNA: `dna = " TCG-TAC-gaC-TAC-CGT-CAG-ACT-TAa-CcA-GTC-cAt-AGA-GCT    "`. There are some typos in the string that we would like to fix:

   a. Use the `trim()` method to remove the leading and trailing whitespace, and then print the results.
   b. Change all of the letters in the dna string to UPPERCASE and print the result.
   c. Note that if you try `console.log(dna)` after applying the methods, the original, flawed string is displayed. To fix this, you need to *reassign* the changes back to `dna`. Apply these fixes to your code so that `console.log(dna)` prints the DNA strand in UPPERCASE with no whitespace.

   Code it at repl.it

2. Let's use string methods to do more work on the DNA strand:

   a. Replace the gene `'GCT'` with `'AGG'`, and then print the altered strand.
   b. Look for the gene `'CAT'` with `indexOf()`. If found print, `'CAT gene found'`, otherwise print, `'CAT gene NOT found'`.
   c. Use `slice()` to print out the fifth gene (set of 3 characters) from the DNA strand.
   d. Use a template literal to print, `"The DNA string is ___ characters long."`
   e. Just for fun, apply methods to `dna` and use another template literal to print, `'taco cat'`.

   Code it at repl.it

7.9.3. Part Three

1. If we want to turn the string `'JavaScript'` into `'JS'`, we might try `.remove()`. Unfortunately, there is no such method in JavaScript. However, we can use our cleverness to achieve the same result.

    a. Use string concatenation and two `slice()` methods to print `'JS'` from `'JavaScript'`.
    b. Without using `slice()`, use method chaining to accomplish the same thing.
    c. Use bracket notation and a template literal to print, `"The abbreviation for 'JavaScript' is 'JS'."`
    d. Just for fun, try chaining 3 or more methods together, and then print the result.

    Code it at repl.it

2. Some programming languages (like Python) include a `title()` method to return a string with Every Word Capitalized (e.g. `'title case'.title()` returns `Title Case`). JavaScript has no `title()` method, but that won't stop us! Use the string methods you know to print `'Title Case'` from the string `'title case'`.

    Code it at repl.it