Strings and Regular Expressions

Strings are arguably one of the most important data types in programming. They're in nearly every higher-level programming language, and being able to work with them effectively is fundamental for developers to create useful programs. By extension, regular expressions are important because of the extra power they give developers to wield on strings. With these facts in mind, the creators of ECMAScript 6 improved strings and regular expressions by adding new capabilities and long-missing functionality. This chapter gives a tour of both types of changes.

Better Unicode Support

Before ECMAScript 6, JavaScript strings revolved around 16-bit character encoding (UTF-16). Each 16-bit sequence is a *code unit* representing a character. All string properties and methods, like the `length` property and the `charAt()` method, were based on these 16-bit code units. Of course, 16 bits used to be enough to contain any character. That's no longer true thanks to the expanded character set introduced by Unicode.

UTF-16 Code Points

Limiting character length to 16 bits wasn't possible for Unicode's stated goal of providing a globally unique identifier to every character in the world. These globally unique identifiers, called *code points*, are simply numbers starting at 0. Code points are what you may think of as character codes, where a number represents a character. A character encoding must encode code points into code units that are internally consistent. For UTF-16, code points can be made up of many code units.

The first 2^16^ code points in UTF-16 are represented as single 16-bit code units. This range is called the *Basic Multilingual Plane* (BMP). Everything beyond that is considered to be in one of the *supplementary planes*, where the code points can no longer be represented in just 16-bits. UTF-16 solves this problem by introducing *surrogate pairs* in which a single code point is represented by two 16-bit code units. That means any single character in a string can be either one code unit for BMP characters, giving a total of 16 bits, or two units for supplementary plane characters, giving a total of 32 bits.

In ECMAScript 5, all string operations work on 16-bit code units, meaning that you can get unexpected results from UTF-16 encoded strings containing surrogate pairs, as in this example:

```
var text = "𠮷";

console.log(text.length);           // 2
console.log(/^.$/.test(text));      // false
console.log(text.charAt(0));        // ""
console.log(text.charAt(1));        // ""
console.log(text.charCodeAt(0));    // 55362
console.log(text.charCodeAt(1));    // 57271
```

The single Unicode character `"𠮷"` is represented using surrogate pairs, and as such, the JavaScript string operations above treat the string as having two 16-bit characters. That means:

- The `length` of `text` is 2, when it should be 1.
- A regular expression trying to match a single character fails because it thinks there are two characters.
- The `charAt()` method is unable to return a valid character string, because neither set of 16 bits corresponds to a printable character.

The `charCodeAt()` method also just can't identify the character properly. It returns the appropriate 16-bit number for each code unit, but that is the closest you could get to the real value of `text` in ECMAScript 5.

ECMAScript 6, on the other hand, enforces UTF-16 string encoding to address problems like these. Standardizing string operations based on this character encoding means that JavaScript can support functionality designed to work specifically with surrogate pairs. The rest of this section discusses a few key examples of that functionality.

The codePointAt() Method

One method ECMAScript 6 added to fully support UTF-16 is the `codePointAt()` method, which retrieves the Unicode code point that maps to a given position in a string. This method accepts the code unit position rather than the character position and returns an integer value, as these `console.log()` examples show:

```
var text = "𠮷a";

console.log(text.charCodeAt(0));    // 55362
console.log(text.charCodeAt(1));    // 57271
console.log(text.charCodeAt(2));    // 97

console.log(text.codePointAt(0));   // 134071
console.log(text.codePointAt(1));   // 57271
console.log(text.codePointAt(2));   // 97
```

The `codePointAt()` method returns the same value as the `charCodeAt()` method unless it operates on non-BMP characters. The first character in `text` is non-BMP and is therefore comprised of two code units, meaning the `length` property is 3 rather than 2. The `charCodeAt()` method returns only the first code unit for position 0, but `codePointAt()` returns the full code point even though the code point spans multiple code units. Both methods return the same value for positions 1 (the second code unit of the first character) and 2 (the `"a"` character).

Calling the `codePointAt()` method on a character is the easiest way to determine if that character is represented by one or two code units. Here's a function you could write to check:

```
function is32Bit(c) {
    return c.codePointAt(0) > 0xFFFF;
}

console.log(is32Bit("𠮷"));         // true
console.log(is32Bit("a"));          // false
```

The upper bound of 16-bit characters is represented in hexadecimal as `FFFF`, so any code point above that number must be represented by two code units, for a total of 32 bits.

The String.fromCodePoint() Method

When ECMAScript provides a way to do something, it also tends to provide a way to do the reverse. You can use `codePointAt()` to retrieve the code point for a character in a string, while `String.fromCodePoint()` produces a single-character string from a given code point. For example:

```
console.log(String.fromCodePoint(134071));  // "𠮷"
```

Think of `String.fromCodePoint()` as a more complete version of the `String.fromCharCode()` method. Both give the same result for all characters in the BMP. There's only a difference when you pass code points for characters outside of the BMP.

The normalize() Method

Another interesting aspect of Unicode is that different characters may be considered equivalent for the purpose of sorting or other comparison-based operations. There are two ways to define these relationships. First, *canonical equivalence* means that two sequences of code points are considered interchangeable in all respects. For example, a combination of two characters can be canonically equivalent to one character. The second relationship is *compatibility*. Two compatible sequences of code points look different but can be used interchangeably in certain situations.

Due to these relationships, two strings representing fundamentally the same text can contain different code point sequences. For example, the character "æ" and the two-character string "ae" may be used interchangeably but are strictly not equivalent unless normalized in some way.

ECMAScript 6 supports Unicode normalization forms by giving strings a `normalize()` method. This method optionally accepts a single string parameter indicating one of the following Unicode normalization forms to apply:

- Normalization Form Canonical Composition (`"NFC"`), which is the default
- Normalization Form Canonical Decomposition (`"NFD"`)
- Normalization Form Compatibility Composition (`"NFKC"`)
- Normalization Form Compatibility Decomposition (`"NFKD"`)

It's beyond the scope of this book to explain the differences between these four forms. Just keep in mind that when comparing strings, both strings must be normalized to the same form. For example:

```
var normalized = values.map(function(text) {
    return text.normalize();
});

normalized.sort(function(first, second) {
    if (first < second) {
        return -1;
    } else if (first === second) {
        return 0;
    } else {
        return 1;
    }
});
```

This code converts the strings in the `values` array into a normalized form so that the array can be sorted appropriately. You can also sort the original array by calling `normalize()` as part of the comparator, as follows:

```
values.sort(function(first, second) {
    var firstNormalized = first.normalize(),
        secondNormalized = second.normalize();

    if (firstNormalized < secondNormalized) {
        return -1;
    } else if (firstNormalized === secondNormalized) {
        return 0;
    } else {
        return 1;
    }
});
```

Once again, the most important thing to note about this code is that both `first` and `second` are normalized in the same way. These examples have used the default, NFC, but you can just as easily specify one of the others, like this:

```
values.sort(function(first, second) {
    var firstNormalized = first.normalize("NFD"),
        secondNormalized = second.normalize("NFD");

    if (firstNormalized < secondNormalized) {
        return -1;
    } else if (firstNormalized === secondNormalized) {
        return 0;
    } else {
        return 1;
    }
});
```

If you've never worried about Unicode normalization before, then you probably won't have much use for this method now. But if you ever work on an internationalized application, you'll definitely find the `normalize()` method helpful.

Methods aren't the only improvements that ECMAScript 6 provides for working with Unicode strings, though. The standard also adds two useful syntax elements.

The Regular Expression u Flag

You can accomplish many common string operations through regular expressions. But remember, regular expressions assume 16-bit code units, where each represents a single character. To address this problem, ECMAScript 6 defines a `u` flag for regular expressions, which stands for Unicode.

The u Flag in Action

When a regular expression has the `u` flag set, it switches modes to work on characters, not code units. That means the regular expression should no longer get confused about surrogate pairs in strings and should behave as expected. For example, consider this code:

```
var text = "𠮷";

console.log(text.length);           // 2
console.log(/^.$/.test(text));      // false
console.log(/^.$/u.test(text));     // true
```

The regular expression `/^.$/` matches any input string with a single character. When used without the `u` flag, this regular expression matches on code units, and so the Japanese character (which is represented by two code units) doesn't match the regular expression. When used with the `u` flag, the regular expression compares characters instead of code units and so the Japanese character matches.

Counting Code Points

Unfortunately, ECMAScript 6 doesn't add a method to determine how many code points a string has, but with the `u` flag, you can use regular expressions to figure it out as follows:

```
function codePointLength(text) {
    var result = text.match(/[\s\S]/gu);
    return result ? result.length : 0;
}

console.log(codePointLength("abc"));    // 3
console.log(codePointLength("𠮷bc"));   // 3
```

This example calls `match()` to check `text` for both whitespace and non-whitespace characters (using `[\s\S]` to ensure the pattern matches newlines), using a regular expression that is applied globally with Unicode enabled. The `result` contains an array of matches when there's at least one match, so the array length is the number of code points in the string. In Unicode, the strings `"abc"` and `"𠮷bc"` both have three characters, so the array length is three.

W> Although this approach works, it's not very fast, especially when applied to long strings. You can use a string iterator (discussed in Chapter 8) as well. In general, try to minimize counting code points whenever possible.

Determining Support for the u Flag

Since the `u` flag is a syntax change, attempting to use it in JavaScript engines that aren't compatible with ECMAScript 6 throws a syntax error. The safest way to determine if the `u` flag is supported is with a function, like this one:

```
function hasRegExpU() {
    try {
        var pattern = new RegExp(".", "u");
        return true;
    } catch (ex) {
        return false;
    }
}
```

This function uses the `RegExp` constructor to pass in the `u` flag as an argument. This syntax is valid even in older JavaScript engines, but the constructor will throw an error if `u` isn't supported.

I> If your code still needs to work in older JavaScript engines, always use the `RegExp` constructor when using the `u` flag. This will prevent syntax errors and allow you to optionally detect and use the `u` flag without aborting execution.

Other String Changes

JavaScript strings have always lagged behind similar features of other languages. It was only in ECMAScript 5 that strings finally gained a `trim()` method, for example, and ECMAScript 6 continues extending JavaScript's capacity to parse strings with new functionality.

Methods for Identifying Substrings

Developers have used the `indexOf()` method to identify strings inside other strings since JavaScript was first introduced. ECMAScript 6 includes the following three methods, which are designed to do just that:

- The `includes()` method returns true if the given text is found anywhere within the string. It returns false if not.
- The `startsWith()` method returns true if the given text is found at the beginning of the string. It returns false if not.
- The `endsWith()` method returns true if the given text is found at the end of the string. It returns false if not.

Each methods accept two arguments: the text to search for and an optional index. When the second argument is provided, `includes()` and `startsWith()` start the match from that index while `endsWith()` starts the match from the second argument minus the length of the first argument; when the second argument is omitted, `includes()` and `startsWith()` search from the beginning of the string, while `endsWith()` starts from the end. In effect, the second argument minimizes the amount of the string being searched. Here are some examples showing these three methods in action:

```
var msg = "Hello world!";

console.log(msg.startsWith("Hello"));       // true
console.log(msg.endsWith("!"));             // true
console.log(msg.includes("o"));             // true

console.log(msg.startsWith("o"));           // false
console.log(msg.endsWith("world!"));        // true
console.log(msg.includes("x"));             // false

console.log(msg.startsWith("o", 4));        // true
console.log(msg.endsWith("o", 8));          // true
console.log(msg.includes("o", 8));          // false
```

The first six calls don't include a second parameter, so they'll search the whole string if needed. The last three calls only check part of the string. The call to `msg.startsWith("o", 4)` starts the match by looking at index 4 of the `msg` string, which is the "o" in "Hello". The call to `msg.endsWith("o", 8)` starts the search from index 7 (the second argument minus the length of the first argument), which is the "o" in "world". The call to `msg.includes("o", 8)` starts the match from index 8, which is the "r" in "world".

While these three methods make identifying the existence of substrings easier, each only returns a boolean value. If you need to find the actual position of one string within another, use the `indexOf()` or `lastIndexOf()` methods.

W> The `startsWith()`, `endsWith()`, and `includes()` methods will throw an error if you pass a regular expression instead of a string. This stands in contrast to `indexOf()` and `lastIndexOf()`, which both convert a regular expression argument into a string and then search for that string.

The repeat() Method

ECMAScript 6 also adds a `repeat()` method to strings, which accepts the number of times to repeat the string as an argument. It returns a new string containing the original string repeated the specified number of times. For example:

```
console.log("x".repeat(3));         // "xxx"
console.log("hello".repeat(2));     // "hellohello"
console.log("abc".repeat(4));       // "abcabcabcabc"
```

This method is a convenience function above all else, and it can be especially useful when manipulating text. It's particularly useful in code formatting utilities that need to create indentation levels, like this:

```
// indent using a specified number of spaces
var indent = " ".repeat(4),
    indentLevel = 0;

// whenever you increase the indent
var newIndent = indent.repeat(++indentLevel);
```

The first `repeat()` call creates a string of four spaces, and the `indentLevel` variable keeps track of the indent level. Then, you can just call `repeat()` with an incremented `indentLevel` to change the number of spaces.

ECMAScript 6 also makes some useful changes to regular expression functionality that don't fit into a particular category. The next section highlights a few.

Other Regular Expression Changes

Regular expressions are an important part of working with strings in JavaScript, and like many parts of the language, they haven't changed much in recent versions. ECMAScript 6, however, makes several improvements to regular expressions to go along with the updates to strings.

The Regular Expression y Flag

ECMAScript 6 standardized the `y` flag after it was implemented in Firefox as a proprietary extension to regular expressions. The `y` flag affects a regular expression search's `sticky` property, and it tells the search to start matching characters in a string at the position specified by the regular expression's `lastIndex` property. If there is no match at that location, then the regular expression stops matching. To see how this works, consider the following code:

```
var text = "hello1 hello2 hello3",
    pattern = /hello\d\s?/,
    result = pattern.exec(text),
    globalPattern = /hello\d\s?/g,
    globalResult = globalPattern.exec(text),
    stickyPattern = /hello\d\s?/y,
    stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello1 "
console.log(stickyResult[0]);   // "hello1 "

pattern.lastIndex = 1;
globalPattern.lastIndex = 1;
stickyPattern.lastIndex = 1;

result = pattern.exec(text);
globalResult = globalPattern.exec(text);
stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello2 "
console.log(stickyResult[0]);   // Error! stickyResult is null
```

This example has three regular expressions. The expression in `pattern` has no flags, the one in `globalPattern` uses the `g` flag, and the one in `stickyPattern` uses the `y` flag. In the first trio of `console.log()` calls, all three regular expressions should return `"hello1 "` with a space at the end.

After that, the `lastIndex` property is changed to 1 on all three patterns, meaning that the regular expression should start matching from the second character on all of them. The regular expression with no flags completely ignores the change to `lastIndex` and still matches `"hello1 "` without incident. The regular expression with the `g` flag goes on to match `"hello2 "` because it is searching forward from the second character of the string (`"e"`). The sticky regular expression doesn't match anything beginning at the second character so `stickyResult` is `null`.

The sticky flag saves the index of the next character after the last match in `lastIndex` whenever an operation is performed. If an operation results in no match, then `lastIndex` is set back to 0. The global flag behaves the same way, as demonstrated here:

```
var text = "hello1 hello2 hello3",
    pattern = /hello\d\s?/,
    result = pattern.exec(text),
    globalPattern = /hello\d\s?/g,
    globalResult = globalPattern.exec(text),
    stickyPattern = /hello\d\s?/y,
    stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello1 "
console.log(stickyResult[0]);   // "hello1 "

console.log(pattern.lastIndex);         // 0
console.log(globalPattern.lastIndex);   // 7
console.log(stickyPattern.lastIndex);   // 7

result = pattern.exec(text);
globalResult = globalPattern.exec(text);
stickyResult = stickyPattern.exec(text);

console.log(result[0]);         // "hello1 "
console.log(globalResult[0]);   // "hello2 "
console.log(stickyResult[0]);   // "hello2 "

console.log(pattern.lastIndex);         // 0
console.log(globalPattern.lastIndex);   // 14
console.log(stickyPattern.lastIndex);   // 14
```

The value of `lastIndex` changes to 7 after the first call to `exec()` and to 14 after the second call, for both the `stickyPattern` and `globalPattern` variables.

There are two more subtle details about the sticky flag to keep in mind:

1. The `lastIndex` property is only honored when calling methods that exist on the regular expression object, like the `exec()` and `test()` methods. Passing the regular expression to a string method, such as `match()`, will not result in the sticky behavior.
2. When using the `^` character to match the start of a string, sticky regular expressions only match from the start of the string (or the start of the line in multiline mode). While `lastIndex` is 0, the `^` makes a sticky regular expression no different from a non-sticky one. If `lastIndex` doesn't correspond to the

beginning of the string in single-line mode or the beginning of a line in multiline mode, the sticky regular expression will never match.

As with other regular expression flags, you can detect the presence of `y` by using a property. In this case, you'd check the `sticky` property, as follows:

```
var pattern = /hello\d/y;

console.log(pattern.sticky);    // true
```

The `sticky` property is set to true if the sticky flag is present, and the property is false if not. The `sticky` property is read-only based on the presence of the flag and cannot be changed in code.

Similar to the `u` flag, the `y` flag is a syntax change, so it will cause a syntax error in older JavaScript engines. You can use the following approach to detect support:

```
function hasRegExpY() {
    try {
        var pattern = new RegExp(".", "y");
        return true;
    } catch (ex) {
        return false;
    }
}
```

Just like the `u` check, this returns false if it's unable to create a regular expression with the `y` flag. In one final similarity to `u`, if you need to use `y` in code that runs in older JavaScript engines, be sure to use the `RegExp` constructor when defining those regular expressions to avoid a syntax error.

Duplicating Regular Expressions

In ECMAScript 5, you can duplicate regular expressions by passing them into the `RegExp` constructor like this:

```
var re1 = /ab/i,
    re2 = new RegExp(re1);
```

The `re2` variable is just a copy of the `re1` variable. But if you provide the second argument to the `RegExp` constructor, which specifies the flags for the regular expression, your code won't work, as in this example:

```
var re1 = /ab/i,

    // throws an error in ES5, okay in ES6
    re2 = new RegExp(re1, "g");
```

If you execute this code in an ECMAScript 5 environment, you'll get an error stating that the second argument cannot be used when the first argument is a regular expression. ECMAScript 6 changed this behavior such that the second argument is allowed and overrides any flags present on the first argument. For example:

```
var re1 = /ab/i,

    // throws an error in ES5, okay in ES6
    re2 = new RegExp(re1, "g");


console.log(re1.toString());            // "/ab/i"
console.log(re2.toString());            // "/ab/g"

console.log(re1.test("ab"));            // true
console.log(re2.test("ab"));            // true

console.log(re1.test("AB"));            // true
console.log(re2.test("AB"));            // false
```

In this code, `re1` has the case-insensitive `i` flag while `re2` has only the global `g` flag. The `RegExp` constructor duplicated the pattern from `re1` and substituted the `g` flag for the `i` flag. Without the second argument, `re2` would have the same flags as `re1`.

The `flags` Property

Along with adding a new flag and changing how you can work with flags, ECMAScript 6 added a property associated with them. In ECMAScript 5, you could get the text of a regular expression by using the `source` property, but to get the flag string, you'd have to parse the output of the `toString()` method as shown below:

```
function getFlags(re) {
    var text = re.toString();
    return text.substring(text.lastIndexOf("/") + 1, text.length);
}

// toString() is "/ab/g"
var re = /ab/g;

console.log(getFlags(re));          // "g"
```

This converts a regular expression into a string and then returns the characters found after the last `/`. Those characters are the flags.

ECMAScript 6 makes fetching flags easier by adding a `flags` property to go along with the `source` property. Both properties are prototype accessor properties with only a getter assigned, making them read-only. The `flags` property makes inspecting regular expressions easier for both debugging and inheritance purposes.

A late addition to ECMAScript 6, the `flags` property returns the string representation of any flags applied to a regular expression. For example:

```
var re = /ab/g;

console.log(re.source);     // "ab"
console.log(re.flags);      // "g"
```

This fetches all flags on `re` and prints them to the console with far fewer lines of code than the `toString()` technique can. Using `source` and `flags` together allows you to extract the pieces of the regular expression that you need without parsing the regular expression string directly.

The changes to strings and regular expressions that this chapter has covered so far are definitely powerful, but ECMAScript 6 improves your power over strings in a much bigger way. It brings a type of literal to the table that makes strings more flexible.

Template Literals

JavaScript's strings have always had limited functionality compared to strings in other languages. For instance, until ECMAScript 6, strings lacked the methods covered so far in this chapter, and string concatenation is as simple as possible. To allow developers to solve more complex problems, ECMAScript 6's *template literals* provide syntax for creating domain-specific languages (DSLs) for working with content in a safer way than the solutions available in ECMAScript 5 and earlier. (A DSL is a programming language designed for a specific, narrow purpose, as opposed to general-purpose languages like JavaScript.) The ECMAScript wiki offers the following description on the template literal strawman:

> This scheme extends ECMAScript syntax with syntactic sugar to allow libraries to provide DSLs that easily produce, query, and manipulate content from other languages that are immune or resistant to injection attacks such as XSS, SQL Injection, etc.

In reality, though, template literals are ECMAScript 6's answer to the following features that JavaScript lacked all the way through ECMAScript 5:

- **Multiline strings** A formal concept of multiline strings.
- **Basic string formatting** The ability to substitute parts of the string for values contained in variables.
- **HTML escaping** The ability to transform a string such that it is safe to insert into HTML.

Rather than trying to add more functionality to JavaScript's already-existing strings, template literals represent an entirely new approach to solving these problems.

Basic Syntax

At their simplest, template literals act like regular strings delimited by backticks (`` ` ``) instead of double or single quotes. For example, consider the following:

```
let message = `Hello world!`;

console.log(message);               // "Hello world!"
console.log(typeof message);        // "string"
console.log(message.length);        // 12
```

This code demonstrates that the variable `message` contains a normal JavaScript string. The template literal syntax is used to create the string value, which is then assigned to the `message` variable.

If you want to use a backtick in your string, then just escape it with a backslash (`\`), as in this version of the `message` variable:

```
let message = `\`Hello\` world!`;

console.log(message);               // "`Hello` world!"
console.log(typeof message);        // "string"
console.log(message.length);        // 14
```

There's no need to escape either double or single quotes inside of template literals.

Multiline Strings

JavaScript developers have wanted a way to create multiline strings since the first version of the language. But when using double or single quotes, strings must be completely contained on a single line.

Pre-ECMAScript 6 Workarounds

Thanks to a long-standing syntax bug, JavaScript does have a workaround. You can create multiline strings if there's a backslash (`\`) before a newline. Here's an example:

```
var message = "Multiline \
string";

console.log(message);       // "Multiline string"
```

The `message` string has no newlines present when printed to the console because the backslash is treated as a continuation rather than a newline. In order to show a newline in output, you'd need to manually include it:

```
var message = "Multiline \n\
string";

console.log(message);       // "Multiline
                            //  string"
```

This should print `Multiline String` on two separate lines in all major JavaScript engines, but the behavior is defined as a bug and many developers recommend avoiding it.

Other pre-ECMAScript 6 attempts to create multiline strings usually relied on arrays or string concatenation, such as:

```
var message = [
    "Multiline ",
    "string"
].join("\n");

var message = "Multiline \n" +
    "string";
```

All of the ways developers worked around JavaScript's lack of multiline strings left something to be desired.

Multiline Strings the Easy Way

ECMAScript 6's template literals make multiline strings easy because there's no special syntax. Just include a newline where you want, and it shows up in the result. For example:

```
let message = `Multiline
string`;

console.log(message);           // "Multiline
                                //  string"
console.log(message.length);    // 16
```

All whitespace inside the backticks is part of the string, so be careful with indentation. For example:

```
let message = `Multiline
               string`;

console.log(message);           // "Multiline
                                //                 string"
console.log(message.length);    // 31
```

In this code, all whitespace before the second line of the template literal is considered part of the string itself. If making the text line up with proper indentation is important to you, then consider leaving nothing on the first line of a multiline template literal and then indenting after that, as follows:

```
let html = `
<div>
    <h1>Title</h1>
</div>`.trim();
```

This code begins the template literal on the first line but doesn't have any text until the second. The HTML tags are indented to look correct and then the `trim()` method is called to remove the initial empty line.

A> If you prefer, you can also use `\n` in a template literal to indicate where a newline should be inserted: A> {:lang="js"} A> ~~~~~~~~ A> A> let message = `Multiline\nstring`; A> A> console.log(message); // "Multiline A> // string" A> console.log(message.length); // 16 A> ~~~~~~~~

Making Substitutions

At this point, template literals may look like fancier versions of normal JavaScript strings. The real difference between the two lies in template literal *substitutions*. Substitutions allow you to embed any valid JavaScript expression inside a template literal and output the result as part of the string.

Substitutions are delimited by an opening `${` and a closing `}` that can have any JavaScript expression inside. The simplest substitutions let you embed local variables directly into a resulting string, like this:

```
let name = "Nicholas",
    message = `Hello, ${name}.`;

console.log(message);       // "Hello, Nicholas."
```

The substitution `${name}` accesses the local variable `name` to insert `name` into the `message` string. The `message` variable then holds the result of the substitution immediately.

I> A template literal can access any variable accessible in the scope in which it is defined. Attempting to use an undeclared variable in a template literal throws an error in both strict and non-strict modes.

Since all substitutions are JavaScript expressions, you can substitute more than just simple variable names. You can easily embed calculations, function calls, and more. For example:

```
let count = 10,
    price = 0.25,
    message = `${count} items cost $${(count * price).toFixed(2)}.`;

console.log(message);       // "10 items cost $2.50."
```

This code performs a calculation as part of the template literal. The variables `count` and `price` are multiplied together to get a result, and then formatted to two decimal places using `.toFixed()`. The dollar sign before the second substitution is output as-is because it's not followed by an opening curly brace.

Template literals are also JavaScript expressions, which means you can place a template literal inside of another template literal, as in this example:

```
let name = "Nicholas",
    message = `Hello, ${
        `my name is ${ name }`
    }.`;
```

```
console.log(message);        // "Hello, my name is Nicholas."
```

This example nests a second template literal inside the first. After the first `${`, another template literal begins. The second `${` indicates the beginning of an embedded expression inside the inner template literal. That expression is the variable `name`, which is inserted into the result.

Tagged Templates

Now you've seen how template literals can create multiline strings and insert values into strings without concatenation. But the real power of template literals comes from tagged templates. A *template tag* performs a transformation on the template literal and returns the final string value. This tag is specified at the start of the template, just before the first ` character, as shown here:

```
let message = tag`Hello world`;
```

In this example, `tag` is the template tag to apply to the `` `Hello world` `` template literal.

Defining Tags

A *tag* is simply a function that is called with the processed template literal data. The tag receives data about the template literal as individual pieces and must combine the pieces to create the result. The first argument is an array containing the literal strings as interpreted by JavaScript. Each subsequent argument is the interpreted value of each substitution.

Tag functions are typically defined using rest arguments as follows, to make dealing with the data easier:

```
function tag(literals, ...substitutions) {
    // return a string
}
```

To better understand what gets passed to tags, consider the following:

```
let count = 10,
    price = 0.25,
    message = passthru`${count} items cost $${(count *
price).toFixed(2)}.`;
```

If you had a function called `passthru()`, that function would receive three arguments. First, it would get a `literals` array, containing the following elements:

- The empty string before the first substitution (`""`)
- The string after the first substitution and before the second (`" items cost $"`)
- The string after the second substitution (`"."`)

The next argument would be `10`, which is the interpreted value for the `count` variable. This becomes the first element in a `substitutions` array. The final argument would be `"2.50"`, which is the interpreted value for `(count * price).toFixed(2)` and the second element in the `substitutions` array.

Note that the first item in `literals` is an empty string. This ensures that `literals[0]` is always the start of the string, just like `literals[literals.length - 1]` is always the end of the string. There is always one fewer substitution than literal, which means the expression `substitutions.length === literals.length - 1` is always true.

Using this pattern, the `literals` and `substitutions` arrays can be interwoven to create a resulting string. The first item in `literals` comes first, the first item in `substitutions` is next, and so on, until the string is complete. As an example, you can mimic the default behavior of a template literal by alternating values from these two arrays:

```
function passthru(literals, ...substitutions) {
    let result = "";

    // run the loop only for the substitution count
    for (let i = 0; i < substitutions.length; i++) {
        result += literals[i];
        result += substitutions[i];
    }

    // add the last literal
    result += literals[literals.length - 1];

    return result;
}

let count = 10,
    price = 0.25,
    message = passthru`${count} items cost $${(count *
price).toFixed(2)}.`;

console.log(message);       // "10 items cost $2.50."
```

This example defines a `passthru` tag that performs the same transformation as the default template literal behavior. The only trick is to use `substitutions.length` for the loop rather than `literals.length` to avoid accidentally going past the end of the `substitutions` array. This works because the relationship between `literals` and `substitutions` is well-defined in ECMAScript 6.

I> The values contained in `substitutions` are not necessarily strings. If an expression evaluates to a number, as in the previous example, then the numeric value is passed in. Determining how such values should output in the result is part of the tag's job.

Using Raw Values in Template Literals

Template tags also have access to raw string information, which primarily means access to character escapes before they are transformed into their character equivalents. The simplest way to work with raw string values is to use the built-in `String.raw()` tag. For example:

```
let message1 = `Multiline\nstring`,
    message2 = String.raw`Multiline\nstring`;

console.log(message1);          // "Multiline
                                //  string"
console.log(message2);          // "Multiline\nstring"
```

In this code, the `\n` in `message1` is interpreted as a newline while the `\n` in `message2` is returned in its raw form of `"\\n"` (the slash and `n` characters). Retrieving the raw string information like this allows for more complex processing when necessary.

The raw string information is also passed into template tags. The first argument in a tag function is an array with an extra property called `raw`. The `raw` property is an array containing the raw equivalent of each literal value. For example, the value in `literals[0]` always has an equivalent `literals.raw[0]` that contains the raw string information. Knowing that, you can mimic `String.raw()` using the following code:

```
function raw(literals, ...substitutions) {
    let result = "";

    // run the loop only for the substitution count
    for (let i = 0; i < substitutions.length; i++) {
        result += literals.raw[i];      // use raw values instead
        result += substitutions[i];
    }

    // add the last literal
    result += literals.raw[literals.length - 1];

    return result;
}

let message = raw`Multiline\nstring`;

console.log(message);           // "Multiline\nstring"
console.log(message.length);    // 17
```

This uses `literals.raw` instead of `literals` to output the string result. That means any character escapes, including Unicode code point escapes, should be returned in their raw form. Raw strings are helpful when you want to output a string containing code in which you'll need to include the character escaping (for instance, if you want to generate documentation about some code, you may want to output the actual code as it appears).

Summary

Full Unicode support allows JavaScript to deal with UTF-16 characters in logical ways. The ability to transfer between code point and character via `codePointAt()` and `String.fromCodePoint()` is an important step for string manipulation. The addition of the regular expression `u` flag makes it possible to operate on code points instead of 16-bit characters, and the `normalize()` method allows for more appropriate string comparisons.

ECMAScript 6 also added new methods for working with strings, allowing you to more easily identify a substring regardless of its position in the parent string. More functionality was added to regular expressions, too.

Template literals are an important addition to ECMAScript 6 that allows you to create domain-specific languages (DSLs) to make creating strings easier. The ability to embed variables directly into template literals means that developers have a safer tool than string concatenation for composing long strings with variables.

Built-in support for multiline strings also makes template literals a useful upgrade over normal JavaScript strings, which have never had this ability. Despite allowing newlines directly inside the template literal, you can still use `\n` and other character escape sequences.

Template tags are the most important part of this feature for creating DSLs. Tags are functions that receive the pieces of the template literal as arguments. You can then use that data to return an appropriate string value. The data provided includes literals, their raw equivalents, and any substitution values. These pieces of information can then be used to determine the correct output for the tag.