

Advanced JavaScript Programming

Regular Expressions

Lesson 1, Activity 2: **Getting Started**

Regular expressions are used to do sophisticated pattern matching, which can often be helpful in form validation. For example, a regular expression can be used to check whether an email address entered into a form field is syntactically correct. JavaScript supports Perl-compatible regular expressions.

There are two ways to create a regular expression in JavaScript:

1. Using literal syntax

```
var reExample = /pattern/;
```

2. Using the `RegExp()` constructor

```
var reExample = new RegExp("pattern");
```

Assuming you know the regular expression pattern you are going to use, there is no real difference between the two; however, if you don't know the pattern ahead of time (e.g, you're retrieving it from a form), it can be easier to use the `RegExp()` constructor.

JavaScript's Regular Expression Methods

The regular expression method in JavaScript has two main methods for testing strings: `test()` and `exec()`.

The `exec()` Method

The `exec()` method takes one argument, a string, and checks whether

that string contains *one or more* matches of the pattern specified by the regular expression. If one or more matches is found, the method returns a result array with the starting points of the matches. If no match is found, the method returns `null`.

The `test()` Method

The `test()` method also takes one argument, a string, and checks whether that string contains a match of the pattern specified by the regular expression. It returns `true` if it does contain a match and `false` if it does not. This method is very useful in form validation scripts. The code sample below shows how it can be used for checking a social security number. Don't worry about the syntax of the regular expression itself. We'll cover that shortly.

Code Sample:

RegularExpressions/Demos/SsnChecker.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>ssn Checker</title>
<script type="text/javascript">
var reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]{4}$/;

function checkSsn(ssn) {
  if (reSSN.test(ssn)) {
    alert("VALID SSN");
  } else {
    alert("INVALID SSN");
  }
}
</script>
</head>
<body>
  <form onsubmit="return false;">
    <input type="text" name="ssn" size="20">
    <input type="button" value="Check">
  </form>
</body>
</html>
```

```
onclick="checkSsn(this.form.ssn.value);">
</form>
</body>
</html>
```

Let's examine the code more closely:

1. First, a variable containing a regular expression object for a social security number is declared.
2. Next, a function called `checkSsn()` is created. This function takes one argument: `ssn`, which is a string. The function then tests to see if the string matches the regular expression pattern by passing it to the regular expression object's `test()` method. If it does match, the function alerts "VALID SSN". Otherwise, it alerts "INVALID SSN".
3. Last, A form in the body of the page provides a text field for inserting a social security number and a button that passes the user-entered social security number to the `checkSsn()` function.

Flags

Flags appearing after the end slash modify how a regular expression works.

- The `i` flag makes a regular expression case insensitive. For example, `/aeiou/i` matches all lowercase and uppercase vowels.
- The `g` flag specifies a global match, meaning that all matches of the specified pattern should be returned.

String Methods

There are several String methods that take regular expressions as arguments.

The search() Method

The `search()` method takes one argument: a regular expression. It returns the index of the first character of the substring matching the regular expression. If no match is found, the method returns `-1`.

```
"Webucator".search(/cat/); //returns 4
```

The split() Method

The `split()` method takes one argument: a regular expression. It uses the regular expression as a delimiter to split the string into an array of strings.

```
"Webucator".split(/[aeiou]/);

/*
returns an array with the following values:
"W", "b", "c", "t", "r"
*/
```

The replace() Method

The `replace()` method takes two arguments: a regular expression and a string. It replaces the first regular expression match with the string. If the `g` flag is used in the regular expression, it replaces all matches with the string.

```
"Webucator".replace(/cat/, "dog"); //returns Webudogor
"Webucator".replace(/[aeiou]/g, "x"); //returns Wxbxcxtxr
```

The match() Method

The `match()` method takes one argument: a regular expression. It

returns each substring that matches the regular expression pattern.

```
"Webucator".match(/[aeiou]/g);  
  
/*  
returns an array with the following values:  
"e", "u", "a", "o"  
*/
```

Lesson 1, Activity 3: **Regular Expression Syntax**

A regular expression is a pattern that specifies a list of characters. In this section, we will look at how those characters are specified.

Start and End (^ \$)

A caret (^) at the beginning of a regular expression indicates that the string being searched must start with this pattern.

- The pattern `^foo` can be found in "food", but not in "barfood".

A dollar sign (\$) at the end of a regular expression indicates that the string being searched must end with this pattern.

- The pattern `foo$` can be found in "curfoo", but not in "food".

Number of Occurrences (? + * { })

The following symbols affect the number of occurrences of the preceding character (or characters if parenthesis are used): `?`, `+`, `*`, and `{ }`.

A questionmark (?) indicates that the preceding character should appear zero or one times in the pattern.

- The pattern `foo?` can be found in "food" and "fod", but not "faod".

A plus sign (+) indicates that the preceding character should appear one or more times in the pattern.

- The pattern `foo+` can be found in "fod", "food" and "foood", but not "fd".

A asterisk (*) indicates that the preceding character should appear zero or more times in the pattern.

- The pattern `fo*d` can be found in "fd", "fod" and "food".

Curly brackets with one parameter (`{ n }`) indicate that the preceding character should appear exactly `n` times in the pattern.

- The pattern `fo{3}d` can be found in "foood" , but not "food" or "fooodd".

Curly brackets with two parameters (`{ n1 , n2 }`) indicate that the preceding character should appear between `n1` and `n2` times in the pattern.

- The pattern `fo{2,4}d` can be found in "food","foood" and "fooodd", but not "fod" or "fooooood".

Curly brackets with one parameter and an empty second parameter (`{ n , }`) indicate that the preceding character should appear at least `n` times in the pattern.

- The pattern `fo{2,}d` can be found in "food" and "fooooood", but not "fod".

Common Characters (`.` `\d` `\D` `\w` `\W` `\s` `\S`)

A period (`.`) represents any character except a newline.

- The pattern `fo.d` can be found in "food", "foad", "fo9d", and "fo*d".

Backslash-d (`\d`) represents any digit. It is the equivalent of `[0-9]`.

- The pattern `fo\d` can be found in "fo1d", "fo4d" and "fo0d", but not in "food" or "fodd".

Backslash-D (`\D`) represents any character except a digit. It is the equivalent of `[^0-9]`.

- The pattern `fo\D` can be found in "food" and "foad", but not in "fo4d".

Backslash-w (`\w`) represents any word character (letters, digits, and the underscore (`_`)).

- The pattern `fo\w` can be found in "food", "fo_d" and "fo4d", but not in "fo*d".

Backslash-W (`\W`) represents any character except a word character.

- The pattern `fo\W` can be found in "fo*d", "fo@d" and "fo.d", but not in "food".

Backslash-s (`\s`) represents any whitespace character (e.g, space, tab, newline, etc.).

- The pattern `fo\s` can be found in "fo d", but not in "food".

Backslash-S (`\S`) represents any character except a whitespace character.

- The pattern `fo\S` can be found in "fo*d", "food" and "fo4d", but not in "fo d".

Grouping (`[]`)

Square brackets (`[]`) are used to group options.

- The pattern `f[aeiou]d` can be found in "fad" and "fed", but not in "food", "faed" or "fd".
- The pattern `f[aeiou]{2}d` can be found in "faed" and "feod", but not in "fod", "fed" or "fd".
- The pattern `[A-Za-z]+` can be found in "Webucator, Inc.", but not in "13078".

Negation (`^`)

When used as the first character within square brackets, the caret (`^`) is used for negation.

- The pattern `f[^aeiou]d` can be found in "fqd" and "f4d", but not in "fad" or "fed".

Subpatterns (`()`)

Parentheses (`()`) are used to capture subpatterns.

- The pattern `f(oo)?d` can be found in "food" and "fd", but not in "fod".

Alternatives (`|`)

The pipe (`|`) is used to create optional patterns.

- The pattern `foo$|^bar` can be found in "foo" and "bar", but not "foobar".

Escape Character (`\`)

The backslash (`\`) is used to escape special characters.

- The pattern `fo\d` can be found in "fo.d", but not in "food" or "fo4d".

Lesson 1, Activity 5: **Backreferences**

Backreferences are special wildcards that refer back to a subpattern within a pattern. They can be used to make sure that two subpatterns match. The first subpattern in a pattern is referenced as `\1`, the second is referenced as `\2`, and so on.

For example, the pattern `([bmpw])o\1` matches "bob", "mom", "pop", and "wow", but not "bop" or "pow".

A more practical example is matching the delimiter in social security numbers. Examine the following regular expression:

```
var pattern = /^d{3}([\ - ]?)d{2}([\ - ]?)d{4}$/;
```

Within the caret (^) and dollar sign (\$), which are used to specify the beginning and end of the pattern, there are three sequences of digits, optionally separated by a hyphen or a space. This pattern will be matched in all of the following strings (and more):

- 123-45-6789
- 123 45 6789
- 123456789
- 123-45 6789
- 123 45-6789
- 123-456789

The last three strings are not ideal, but they do match the pattern. Backreferences can be used to make sure that the second delimiter matches the first delimiter. The regular expression would look like this:

```
^\d{3}([\- ]?)\d{2}\1\d{4}$
```

The `\1` refers back to the first subpattern. Only the first three strings listed above match this regular expression.

Lesson 1, Activity 6: Form Validation with Regular Expressions

Regular expressions make it easy to create powerful form validation functions. Take a look at the following example:

Code Sample:

RegularExpression/Demos/Login.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Login</title>
<script type="text/javascript">
var reEmail = /^(\w+[\-\.\.])*\w+@(\w+\.)+[A-Za-z]+$/;
var rePassword = /^[A-Za-z\d]{6,8}$/;

function validate(form){
    var email = form.Email.value;
    var password = form.Password.value;
    var errors = [];

    if (!reEmail.test(email)) {
        errors[errors.length] = "You must enter a valid email address.";
    }

    if (!rePassword.test(password)) {
        errors[errors.length] = "You must enter a valid password.";
    }

    if (errors.length > 0) {
        reportErrors(errors);
        return false;
    }

    return true;
}

function reportErrors(errors){
    var msg = "There were some problems...\n";
    for (var i = 0; i<errors.length; i++) {
        var numError = i + 1;
        msg += "\n" + numError + ". " + errors[i];
    }
    alert(msg);
}
```

```

</script>
</head>
<body>
<h1>Login Form</h1>
<form method="post" action="Process.html" onsubmit="return validate(this);">

Email: <input type="text" name="Email" size="25"><br/>
Password: <input type="password" name="Password" size="10"><br/>
*Password must be between 6 and 10 characters and
can only contain letters and digits.<br/>

<input type="submit" value="Submit">
<input type="reset" value="Reset Form">
</p>
</form>
</body>
</html>

```

This code starts by defining regular expressions for an email address and a password. Let's break each one down.

```
var reEmail = /^(\w+\.)*\w+@(\w+\.)+[A-Za-z]+$;/
```

1. The caret (^) says to start at the beginning. This prevents the user from entering invalid characters at the beginning of the email address.
2. (\w+ [\- \ .]) * allows for a sequence of word characters followed by a dot or a dash. The * indicates that the pattern can be repeated zero or more times. Successful patterns include "ndunn.", "ndunn-", "nat.s.", and "nat-s-".
3. \w+ allows for one or more word characters.
4. @ allows for a single @ symbol.
5. (\w+ \ .) + allows for a sequence of word characters followed by a dot. The + indicates that the pattern can be repeated one or more times. This is the domain name without the last portion (e.g, without the "com" or "gov").
6. [A-Za-z]+ allows for one or more letters. This is the "com" or

"gov" portion of the email address.

7. The dollar sign (\$) says to end here. This prevents the user from entering invalid characters at the end of the email address.

```
var rePassword = /^[A-Za-z\d]{6,8}$/;
```

1. The caret (^) says to start at the beginning. This prevents the user from entering invalid characters at the beginning of the password.
2. [A-Za-z\d]{6,8} allows for a six- to eight-character sequence of letters and digits.
3. The dollar sign (\$) says to end here. This prevents the user from entering invalid characters at the end of the password.

Lesson 1, Activity 8: **Advanced Form Validation**

Duration: 25 to 40 minutes.

1. Open [RegularExpressions/Exercises/FormValidation.js](#) for editing.

- Write additional regular expressions to check for:

1. Proper Name

- starts with capital letter
- followed by one or more letters or apostrophes
- may be multiple words (e.g, "New York City")

2. Initial

- zero or one capital letters

3. State

- two capital letters

4. US Postal Code

- five digits (e.g, "02138")
- possibly followed by a dash and four digits (e.g, "-1234")

5. Username

- between 6 and 15 letters or digits

2. Open [RegularExpressions/Exercises/Register.html](#) for editing.

- Add validation to check the following fields:

1. first name

2. middle initial

3. last name

- 4. city
- 5. state
- 6. zip
- 7. username

3. Test your solution in a browser.

Challenge

1. Add regular expressions to test Canadian and United Kingdom postal codes:
 - Canadian Postal Code - A letter followed by a digit, a letter, a space, a digit, a letter, and a digit (e.g, M1A 1A1)
 - United Kingdom Postal Code - 2 letters followed by 1 or 2 numbers, an optional whitespace, 1 number, and 2 letters. (e.g, WC12 3XX)
2. Modify Register.html to check the postal code against these two new regular expressions as well as the regular expression for a US postal code.

Solution:

RegularExpressions/Solutions/FormValidation.js

```
// Regular Expressions
var reEmail = /^(\w+[\-\.\.])*\w+@(\w+\.)+[A-Za-z]+$/;
var rePassword = /^[A-Za-z\d]{6,8}$/;
var reProperName = /^(([A-Z][A-Za-z']+) )*[A-Z][A-Za-z']+$/;
var reInitial = /^[A-Z]$/;
var reState = /^[A-Z]{2}$/;
var rePostalUS = /^\d{5}(\-\d{4})?$/;
var reUsername = /^[A-Za-z\d]{6,15}$/;
---- C O D E   O M I T T E D ----
```

Solution:

RegularExpressions/Solutions/Register.html

```

<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Registration Form</title>
<script type="text/javascript" src="FormValidation.js"></script>
<script type="text/javascript">

function validate(form){
    var firstName = form.FirstName.value;
    var midInitial = form.MidInit.value;
    var lastName = form.LastName.value;
    var city = form.City.value;
    var state = form.State.value;
    var zipCode = form.Zip.value;
    var email = form.Email.value;
    var userName = form.Username.value;
    var password1 = form.Password1.value;
    var password2 = form.Password2.value;
    var errors = [];

    if (!reProperName.test(firstName)) {
        errors[errors.length] = "You must enter a valid first name.";
    }

    if (!reInitial.test(midInitial)) {
        errors[errors.length] = "You must enter a one-letter middle initial.";
    }

    if (!reProperName.test(lastName)) {
        errors[errors.length] = "You must enter a valid last name.";
    }

    if (!reProperName.test(city)) {
        errors[errors.length] = "You must enter a valid city.";
    }

    if (!reState.test(state)) {
        errors[errors.length] = "You must enter a valid state.";
    }

    if (!rePostalUS.test(zipCode)) {
        errors[errors.length] = "You must enter a valid zip code.";
    }

    if (!reUsername.test(userName)) {
        errors[errors.length] = "You must enter a valid username.";
    }

```

```

}
---- C O D E   O M I T T E D ----

```

Challenge Solution:

RegularExpressions/Solutions/FormValidation-challenge.js

```

// Regular Expressions
---- C O D E   O M I T T E D ----

var rePostalUS = /^\d{5}(\-\d{4})?$/;
var rePostalCA = /^[A-Z]\d[A-Z] \d[A-Z]\d$/;
var rePostalUK = /^[A-Z]{2}[0-9]{1,2} ?[0-9]{1}[A-Z]{2}$/;
---- C O D E   O M I T T E D ----

```

Challenge Solution:

RegularExpressions/Solutions/Register-challenge.html

```

<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Registration Form</title>
<script type="text/javascript" src="FormValidation-challenge.js"></script>
<script type="text/javascript">

function validate(form){
---- C O D E   O M I T T E D ----

    if (!rePostalUS.test(zipCode)
        && !rePostalCA.test(zipCode)
        && !rePostalUK.test(zipCode))
    {
        errors[errors.length] = "You must enter a valid postal code.";
    }
---- C O D E   O M I T T E D ----

    return true;
}
---- C O D E   O M I T T E D ----

```

Lesson 1, Activity 9: Cleaning Up Form Entries

It is sometimes nice to clean up (or sanitize) user entries immediately after they are entered. This can be done using a combination of regular expressions and the `replace()` method of a string object.

The `replace()` Method Revisited

Earlier in the 'Getting Started' lesson, we showed how the `replace()` method of a string object can be used to replace regular expression matches with a string. The `replace()` method can also be used with backreferences to replace a matched pattern with a new string made up of substrings from the pattern. The example below illustrates this.

Code Sample:

[RegularExpressions/Demos/SsnCleaner.html](#)

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>ssn Cleaner</title>
<script type="text/javascript">
var reSSN = /^(\d{3})[\- ]?(\d{2})[\- ]?(\d{4})$/;

function cleanSsn(ssn){
  if (reSSN.test(ssn)) {
    var cleanedSsn = ssn.replace(reSSN, "$1-$2-$3");
    return cleanedSsn;
  } else {
    alert("INVALID SSN");
    return ssn;
  }
}
</script>
</head>
<body>
<form onsubmit="return false;">
  <input type="text" name="ssn" size="20">
  <input type="button" value="Clean SSN"
    onclick="this.form.ssn.value = cleanSsn(this.form.ssn.value);">
```

```
</form>  
</body>  
</html>
```

The `cleanSsn()` function is used to "clean up" a social security number. The regular expression contained in `reSSN`, `^(\d{3})[\-]?(\d{2})[\-]?(\d{4})$`, contains three subexpressions: `(\d{3})`, `(\d{2})`, and `(\d{4})`. Within the `replace()` method, these subexpressions can be referenced as `$1`, `$2`, and `$3`, respectively.

When the user clicks on the "Clean SSN" button, the `cleanSsn()` function is called. This function first tests to see that the user-entered value is a valid social security number. If it is, it then cleans it up with the line of code below, which dash-delimits the three substrings matching the subexpressions.

```
var cleanedSsn = ssn.replace(reSSN, "$1-$2-$3");
```

It then returns the cleaned-up social security number.

Lesson 1, Activity 11: **Cleaning Up Form Entries**

Duration: 15 to 25 minutes.

1. Open [RegularExpressions/Exercises/PhoneCleaner.html](#) for editing.
2. Where the comment indicates, declare a variable called `cleanedPhone` and assign it a cleaned-up version of the user-entered phone number. The cleaned up version should fit the following format: (555) 555-1212
3. Test your solution in a browser.

Challenge

Some phone numbers are given as a combination of numbers and letters (e.g, 877-WEBUCATE). As is the case with 877-WEBUCATE, such numbers often have an extra character just to make the word complete.

1. Add a function called `convertPhone()` that:
 - strips all characters that are not numbers or letters
 - converts all letters to numbers
 - ABC -> 2
 - DEF -> 3
 - GHI -> 4
 - JKL -> 5
 - MNO -> 6
 - PQRS -> 7
 - TUV -> 8
 - WXYZ -> 9

- passes the first 10 characters of the resulting string to the `cleanPhone()` function
 - returns the resulting string
2. Modify the form, so that it calls `convertPhone()` rather than `cleanPhone()`.
 3. Test your solution in a browser.

Solution:

RegularExpressions/Solutions/PhoneCleaner.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Phone Cleaner</title>
<script type="text/javascript">
var rePhone = /^\(?(?([2-9]\d\d)\)?[\-\.\ ]?([2-9]\d\d)[\-\.\ ]?(\d{4})$/;

function cleanPhone(phone) {
  if (rePhone.test(phone)) {
    var cleanedPhone = phone.replace(rePhone, "($1) $2-$3");
    return cleanedPhone;
  } else {
    alert("INVALID PHONE");
    return phone;
  }
}

</script>
</head>
<body>
<form onsubmit="return false;">
  <input type="text" name="Phone" size="20">
  <input type="button" value="Convert Phone"
    onclick="this.form.Phone.value = cleanPhone(this.form.Phone.value);">
</form>
</body>
</html>
```

Challenge Solution:

RegularExpressions/Solutions/PhoneCleaner-challenge.html

```

<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Phone Checker</title>
<script type="text/javascript">
var rePhone = /^\(?(?([2-9]\d\d)\)?[\-\.\ ]?([2-9]\d\d)[\-\.\ ]?(\d{4})$/;

---- C O D E   O M I T T E D ----

function convertPhone(phone) {
  var convertedPhone;
  convertedPhone = phone.replace(/^[A-Za-z\d]/g, "");
  convertedPhone = convertedPhone.replace(/[ABC]/gi, "2");
  convertedPhone = convertedPhone.replace(/[DEF]/gi, "3");
  convertedPhone = convertedPhone.replace(/[GHI]/gi, "4");
  convertedPhone = convertedPhone.replace(/[JKL]/gi, "5");
  convertedPhone = convertedPhone.replace(/[MNO]/gi, "6");
  convertedPhone = convertedPhone.replace(/[PQRS]/gi, "7");
  convertedPhone = convertedPhone.replace(/[TUV]/gi, "8");
  convertedPhone = convertedPhone.replace(/[WXYZ]/gi, "9");
  return cleanPhone(convertedPhone.substr(0, 10));
}
</script>
</head>
<body>
  <form onsubmit="return false;">
    <input type="text" name="Phone" size="20">
    <input type="button" value="Convert Phone"
      onclick="this.form.Phone.value = convertPhone(this.form.Phone.value);">
  </form>
</body>
</html>

```