JavaScript:

Intermediate JavaScript Coding From The Ground Up

The information in the following pages is broadly considered to be a truthful and accurate account of facts, and as such any inattention, use or misuse of the information in question by the reader will render any resulting actions solely under their purview. There are no scenarios in which the publisher or the original author of this work can be in any fashion deemed liable for any hardship or damages that may befall them after undertaking information described herein.

Additionally, the information in the following pages is intended only for informational purposes and should thus be thought of as universal. As befitting its nature, it is presented without assurance regarding its prolonged validity or interim quality.

Table of Contents

INTRODUCTION

Thank you for downloading JavaScript: Intermediate JavaScript Coding from The Ground Up! This book is the second entry of the DIY JavaScript series. It is preceded by the book JavaScript: Beginner JavaScript Coding from The Ground Up, and assumes that the user is familiar with the contents of that book, including beginner JavaScript code and syntax. The books should be taken in chronological order for optimal results. This book will cover intermediate JavaScript manipulation techniques, with code examples to match the concepts explained.

Thank you again for downloading this book! You have many choices available to you for furthering your JavaScript coding knowledge. Thank you for selecting the DIY JavaScript series as your tool of choice!

CHAPTER 1

IF-ELSE STATEMENTS IN JAVASCRIPT

Often in our day to day life, we take several decisions. Some decisions are small, and some have higher significance. Programming is greatly inspired by the human life. Let me relate the concept of decision statements in a programming language with real-life decisions. Let us assume you are making tea. While making tea, you add sugar. Now when you decided to add sugar, you made a decision unconsciously. Since you wanted the tea to be sweetened, you added sugar. If you had prepared the tea for a diabetic patient, you would not have added the sugar. So, this real-life example can be converted to programming world in the following way:

```
if(you want the tea to be sweet)
{
addSugar();
}
```

In programming, whenever you encounter a situation where you need to make a decision, you make use of conditional statements that will help the program to decide its flow based on whether the condition was true or false.

The most common way to implement decision-making in a JavaScript program is through if-else statements. However, there are other variants of if-else. We shall discuss them later in this chapter. Let us first understand how if-else works.

Whenever a condition is encountered, a decision is made based on whether the condition was true or false. If the condition is true, the immediate next block is executed or else it is skipped.

JavaScript, like any other programming language, supports the other variants of an if-else statement. These are:

1. if statement
2. if-else statement

3. else-if statement

The above three variants are used as per the requirement. Let us discuss each one of them in detail. We will also see some examples.

1. **The if statement**

   The most common and the most basic one of the three variants is a simple if statement. First of all, let us understand when we can make use of this. This statement is used when we just have a single condition to check before executing a set of statements, and we require that rest of the statements following that is executed nevertheless. This will be much easier to understand through an example but first, let us look at the syntax for writing an “if- statement.”

```
if (expression)
{
//Set of statements to be executed if
the expression results in a true value;
}
```

   The expression can either be true or false. If the expression is true, the set of statements following the “if“ will be executed or else it is ignored. Let me explain it to you using an example in which we

check the age of a person and displays whether or not he is eligible to vote.

```
varageTobeChecked=23;
if (ageTobeChecked>=18)
{
      alert( “The person with age” +
      ageTobeChecked + “ ,is eligible to
      vote”);
}
```

In the above example, since the value of the variable *ageTobeChecked*is greater than 18, an alert will pop with the message “The person with age23, is eligible to vote”.

2. The if-else statement

The if-else is used when you need to make choices between two sets of code based on certain condition or conditions. If the condition is true, one set of statements are executed or else the second set of statement is executed. Its syntax is almost similar to the previous type but with a small addition.

```
if (expression)
   {
   //Set of statements to be executed if
   the expression results in a //true
   value;
   }
else
   {
   //Set of statements to be executed if
   the expression results in a false
   //value;
   }
```

If the expression is true, the first set of statements is executed or else the second set is executed. In the last example, you only get an alert message if the person was eligible to vote but no alert popped up when the age was lesser than 18. If-else is used for such cases. Let us understand this through an example:

```
varageTobeChecked=16;
if (ageTobeChecked>=18)
{
      alert( “The person with age” +
      ageTobeChecked + “ ,is eligible to
      vote”);
}
```

```
else
{
      alert( “The person with age” +
      ageTobeChecked + “ ,is ineligible
      to vote”);
}
```

3. The else-if statement

Whenever there are several conditions to be checked and a set of statements are to be executed, we make use of else-if statements. Let us first look at the syntax and then we will understand it better through an example.

```
if (expression1)
{
//Set of statements to be executed if
the expression1 results in a true
value;
}
else if (expression2)
{
//Set of statements to be executed if
the expression2 results in a true
value;
}
```

```
else
{
//set of statements to be executed if
none of the expressions listed above
are true;
}
```

If expression1 is true, the first set of statements is to be executed. Else if expression2 is true, the second set of statements is executed. In case both the expressions result in a false value, the else part is executed. There can be any number of else-if statements.

```
varpurchasePrice=1600;
if (purchasePrice< 1000)
{
      alert( “There is no discount
      available”);
}
else if(purchasePrice>=1000
&&purchasePrice<=1600)
{
      alert( “The available discount is
      10%”);
}
```

```
else
{
      alert( “The available discount is
      20%”);
}
```

In the above example, since the purchase price satisfies the second condition, the first block is ignored, and an alert will pop up with the message “The available discount is 10%”.

CHAPTER 2

COMPARISON OPERATORS IN JAVASCRIPT

What are comparison operators?

As the name suggests, these operators are used to compare two variables or values. The final value of the expression made up of these operators is either true or false. The general syntax of the usage of these operators consists of the two operands to be compared which are placed on the either side of the operator.

Listed below are the comparison operators in JavaScript.

Equality operator [==]

The equality operator is used to check equality of the two operands. This operator is a non-strict type comparison operator which means it first attempts to

convert the operands to the same type, if they are not of the same type, and then compares them, i.e., it doesn't compare type along with values of the operands.

When comparing two objects, this operator compares the reference of the objects, i.e., the memory address of the objects which means two objects can be same only if they point to the same memory address. But when comparing an object with a non-object type value or variable, this operator attempts to convert the object value to a primitive type value and then compare it.

Examples:

```
if( 7 == 7 ) console.log('true');
     //prints 'true' on console

if( 7 == '7' )console.log('true');
     //prints 'true' on console

if( 7 == 4 )console.log('true'); //doesn't
prints anything

if( true == 1 ) console.log('true');
     //prints 'true' on console

if( true == 7 ) console.log('true');
     //doesn't prints anything
```

```
if( false == 0 ) console.log(‘true’);
     //prints ‘true’ on console

if( false == 1 ) console.log(‘true’);
     //doesn’t prints anything

//comparing objects:

var ob1 = { test: ‘example’ },

    ob2 = { test: ‘example’ };

var ob3 = ob1;

if( ob1 == ob2 ) console.log(‘true’);
     //doesn’t prints anything
```

```
if( ob1 == ob3 ) console.log(‘true’);
     //prints ‘true’ on console

//comparing object with a non-object type
value:

var ob = new String(‘test’);//string type
OBJECT

if( ob == ‘test’ )
console.log(‘true’);//prints ‘true’ on
console
```

Strict equality operator [===]

This operator also checks equality of the two operands, but it differs from the equality operators as it also takes the type of the two operands into consideration while comparing them, i.e., there is no attempt made to convert the operands into the same types before comparing.

Examples:

```
if( 7 === 7) console.log('true');
     //prints 'true' on console

if( 7 === '7') console.log('true');
     //doesn't prints anything

if( true === 1) console.log('true');
     //doesn't prints anything

if( true === true) console.log('true');
     //prints 'true' on console
```

Inequality operator [!=]

This operator is just the opposite of equality operator; it checks whether the operands are unequal or not. This is a non-strict type comparison operator, which means it first attempts to convert the operands to the same type, if they are not of the same type, and then

compares them, so the type of an operand is not taken into consideration while comparing them.

When checking if two objects are unequal, this operator compares the reference of the objects, i.e., the memory address of the objects, which means two objects can be different only if they point to different memory addresses. But when comparing an object with a non-object type value or variable, this operator attempts to convert the object value to a primitive type value and then checks if they are unequal.

Example:

```
if( 7 != 7 ) console.log(‘true’);
     //doesn’t prints anything

if( 7 != ‘7’) console.log(‘true’);
     //doesn’t prints anything

if( 7 != 4) console.log(‘true’);
     //prints ‘true’ on console

if( true != 1) console.log(‘true’);
     //doesn’t prints anything

if( true != 7) console.log(‘true’);
     //prints ‘true’ on console

if( false != 0 ) console.log(‘true’);
     //doesn’t prints anything
```

```
if( false != 1) console.log(‘true’);
     //prints ‘true’ on console

//comparing objects:

var ob1 = { test: ‘example’ },

    ob2 = { test: ‘example’ };

var ob3 = ob1;

if( ob1 != ob2) console.log(‘true’);
     //prints ‘true’ on console

if( ob1 != ob3) console.log(‘true’);
     //doesn’t prints anything

//comparing object with a non-object type
value:

var ob = new String(‘test’);//string type
OBJECT

if( ob != ‘test’) console.log(‘true’);
     //doesn’t prints anything
```

Strict inequality operator [!==]

This operator checks whether the two operands are different with respect to both their types and values,

whereas the inequality operator didn't take the type of operand into consideration.

Examples:

```
if( 7 !== 7) console.log('true');
     //doesn't prints anything

if( 7 !== '7') console.log('true');
     //prints 'true' on console

if( true !== 1) console.log('true');
     //prints 'true' on console

if( true !== true) console.log('true');
     //doesn't prints anything
```

Greater than operator [>]

This operator checks whether the operand on the left side is greater than the operand on the right side, if it is then true is returned else false.

Example:

```
if( 7 > 2) console.log('true');
     //prints 'true' on console

if( 2 > 7) console.log('true');
     //doesn't prints anything
```

```
if( 7 > 7) console.log(‘true’);
     //doesn’t prints anything
```

Greater than or equal to operator [>=]

This operator checks whether the operand on the left side is greater than or equal to the operand on the right side.

Example

```
if( 7 >= 2 ) console.log(‘true’);
     //prints ‘true’ on console

if( 2 >= 7 ) console.log(‘true’);
     //doesn’t prints anything

if( 7 >= 7 ) console.log(‘true’);
     //prints ‘true’ on console
```

Smaller than operator [<]

This operator checks whether the operand on the left side is smaller than the operand on the right side, if it is then true is returned else false.

Example:

```
if( 7 < 2) console.log(‘true’);
     //doesn’t prints anything

if( 2 < 7) console.log(‘true’);
     //prints ‘true’ on console

if( 7 < 7 ) console.log(‘true’);
     //doesn’t prints anything
```

Smaller than or equal to operator [<=]

This operator checks whether the operand on the left side is smaller than or equal to the operand on the right side.

Example

```
if( 7 <= 2) console.log(‘true’);
     //doesn’t prints anything

if( 2 <= 7) console.log(‘true’);
     //prints ‘true’ on console

if( 7 <= 7) console.log(‘true’);
     //prints ‘true’ on console
```

CHAPTER 3

LOGICAL OPERATORS

What are logical operators?

Logical operators are used to making up a single expression (a criterion) by combining expressions, methods or even variables which are said to be the operands of logical operators. For example if you want to check if a variable's value lies between 5 and 21, you will use a logical operator to make up a criteria for the same, which would be "`x > 5 && x < 21`", where '&&' is the logical operator and on its both sides are the expressions specifying conditions for the criteria.

They are typically used with boolean variables or methods and expressions which return a boolean type value. The expressions formed up using logical operator returns value of one of its operand.

Listed below are the logical operators in JavaScript.

Logical AND [&&]

This operator returns value of its first operand if it's a boolean type false value or can be converted to it*(the list of values which can be converted to boolean type false is available at the end of this chapter)*, else it returns the value of its second operand. Thus the whole expression's value will result in a boolean type true value only if both of its operand value is true.

Examples:

```
var x = 4;

if ( x > 2 && x < 5) console.log("true");

else console.log("false");

//prints 'true' on the console screen


if ( x > 10 && x < 20) console.log("true");

else console.log("false");

//prints 'false' on the console screen
```

```
console.log( 'This' && 'That' ); //prints
“That” as any value other than one listed
at end of chapter can be converted to true

console.log( false&& 'That' ); //prints
“false” on the console screen

console.log( ‘That’ && false ); //prints
“false” on the console screen
```

Logical OR [||]

This operator returns value of its first operand if it’s a boolean type true value or can be converted to it; else it returns the value of its second operand. Thus the whole expression’s value will result in a boolean type true value only if any of its operand value is true.

Examples:

```
var x = 4;

if ( x > 10 || x <20) console.log(“true”);

else console.log(“false”);

//prints ‘true’ on the console screen


if ( x > 10 || x <2) console.log(“true”);
```

```
else console.log(“false”);

//prints ‘false’ on the console screen



console.log( ‘This’ || ‘That’ ); //prints
“This” on the console screen

console.log( false || ‘That’ ); //prints
“That” on the console screen

console.log( ‘That’ || false ); //prints
“That” on the console screen
```

Logical NOT [!]

This operator returns boolean type false value if the value of its first operand is a boolean type true value or can be converted to it, else it returns a boolean type true value. Unlike other logical operators, this operator works on one operand only.

Examples:

```
var x = true;

if( ! x ) console.log(“true”);

else console.log(“false”);
```

```
//prints ‘false’ on the console screen


var y = false;

if( ! y ) consle.log(“true”;

else console.log(“false”);

//prints ‘true’ on the console screen


console.log( ! ‘That’ ); //prints ‘false’ on the console screen

console.log( ! ‘’ );//prints ‘true’ on the console screen, as ‘’ can be converted to false as mentioned at the end of the chapter
```

Using logical operators as a replacement for if-else

You can substitute the if-else block with logical operators. In logical AND(&&) and logical OR(||), the value of the second operand is evaluated depending on the value of the first operand; this is what makes it possible for these operators to be substituted in place of if-else block.

Examples:

```
if( x > 10 ) someMethod();

//the above statement can be subsituted by:

(x > 10) && someMethod();



if( ! x > 10) someOtherMethod();

// the above statements can be subsituted
by

( x > 10 ) || someOtherMethod();
```

Order of evaluation of logical operator in a single statement

If there are multiple logical operators in a single expression they will follow the order of precedence, i.e., the ones inside brackets() will be evaluated first and then the order of evaluation will be from left to right.

Example:

```
console.log( false && true  || true );
     // prints ‘true’ on the console screen
```

```
console.log( false && (true || true) );
     // prints false on the console screen
```

Values that can be converted to boolean type false

Here is the list of the values which can be converted to boolean type false value:

- undefined
- null
- 0
- NaN
- "or ‘’ , i.e., an empty string

CHAPTER 4

BITWISE OPERATORS

What are bitwise operators?

Bitwise operators work on binary numbers. The operands corresponding to bitwise operators are converted to 32-bit binary number before the operation takes place. But the return value of the expression formed from bitwise operators is a standard JavaScript numerical value.

Listed below are the bitwise operators in JavaScript.

Bitwise AND [&]

This operator compares each bit of one operand to the bits of the other operand, from right to left, and then sets the bit at that position as 1 in the return value if both operand's bit is 1 at that position else it sets it as 0, this is known as AND operation.

Example:

```
console.log( 5 &6 ); //prints ‘4’ on
console screen, explanation below
```

In the above example, first 5 and 6 are converted to their binary form which is 0101 and 0110 respectively, and then the AND operation is performed on each bit, which has been demonstrated below:

0 1 0 1

0 1 1 0

0 1 0 0

The resultant binary is 0100 which is equal to ‘4’ the in decimal system, hence ‘4’ is printed.

Bitwise OR [|]

This operator compares each bit of one operand to the bits of the other operand, from right to left, and then sets the bit at that position as 1 in the return value if either of the operand’s bit is 1 at that position and if both operands have 0 bit in that position then the result’s bit at that position is set to 0 too, this is known as OR operation.

Example:

```
console.log( 5 | 6 ); //prints ‘7’ on
console screen, explanation below
```

In the above example, first 5 and 6 are converted to their binary form which is 0101 and 0110 respectively, and then the OR operation is performed on each bit, which has been demonstrated below:

0 1 0 1

0 1 1 0

0 1 11

The resultant binary is 0111 which is equal to ‘7’ in the decimal system.

Bitwise XOR [^]

This operator compares each bit of one operand to the bits of the other operand, from right to left, and then sets the bit at that position as 1 in the return value if either of the bit is 1, and the other one is 0 at that position else it sets it as 0, this is known as XOR operation.

Example:

```
console.log( 5 ^ 6 ); //prints ‘3’ on
console screen, explanation below
```

In the above example, first 5 and 6 are converted to their binary form which is 0101 and 0110 respectively, and then the XOR operation is performed on each bit, which has been demonstrated below:

0 1 0 1

0 1 1 0

0 01 1

The resultant binary is 0011 which is equal to ‘3’ in the decimal system.

Bitwise NOT [~]

Unlike other bitwise operators, this operator works on one operand only. This operator inverts the bits of its operand, i.e., changes all the 0 bits to 1 and vice-versa.

Example:

```
console.log( ~5 ); //prints ‘-6’ on console
screen, explanation below
```

In the above example, first 5 is converted to itsbinary form which is 00000000000000000000000000000101 (32-bit binary), and then its each bit is inverted, which has been demonstrated below:

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0

The value of the resultant binary is equal to '-6' in the decimal system.

Left shift [<<]

This operator shifts the bits of the first operand to the left by the times specified as the second operand. Excess of bits that are shifted off to the left are discarded, and the extra bits added from the right are all 0.

Example:

```
console.log(5 << 6 ); //prints '320' on console screen, explanation below
```

In the above example, first 5 is converted to its binary form which is 00000000000000000000000000000101, then each of its bit is shifted to left by 6 times, which results in the binary number 00000000000000000000000101000000 which is equivalent of ‘320’ in the decimal system.

Signed Right Shift [>>]

This operator shifts the bits of the first operand to the right by the times specified as the second operand. Excess of bits that are shifted off to the right are discarded, and the extra bits added from left are same as the leftmost bit before the operation, i.e., 0 is pushed from left if the leftmost bit was 0 and 1 is pushed if 1 was the leftmost bit.

This is called ‘Signed’ left shift because as the same bit as the leftmost bit before the operation is pushed from the left, the sign of the value doesn’t change.

Example:

```
console.log(5 >> 6 ); //prints ‘0’ on
console screen, explanation below
```

In the above example, first 5 is converted to its binary form which is 00000000000000000000000000000101, then each of

its bit is shifted to the right by 6times, which results in the binary number 00000000000000000000000000000000 which is equivalent of ‘0’ in the decimal system.

Zero fill Right Shift [>>>]

This operator shifts the bits of the first operand to the right by the times specified as the second operand. Excess of bits that are shifted off to the right are discarded, and the extra bits added from the right are all 0.

Example:

```
console.log(-9>>> 2 ); //prints ‘1073741821’ on console screen, explanation below
```

In the above example, first -9 is converted to its binary form which is 11111111111111111111111111110111, then each of its bit is shifted to the right by 2 times, which results in the binary number 00111111111111111111111111111101 which is equivalent of ‘1073741821’ in the decimal system.

CHAPTER 5

SWITCH STATEMENTS IN JAVASCRIPT

If statement works great when there are a lesser number of conditions to be checked. However, if the case arises where we need to compare the same variable for multiple values and then execute some set of statements based on whether or not those comparisons resulted in a true value, it's a better choice to go with switch case. Switch case can be used to perform different operations on different conditions.

Before we dive any deeper into what and how of switch statements, let us have a look at the syntax.

```
switch (expression) {
    case n1:
        statements;
        break;
```

```
    case n2:
        statements;
        break;

    case n3:
        statements;
        break;

    default:
        Statements;
}
```

Let us see how this syntax works. The expression written inside the bracket is evaluated. The resulting value is compared with each of the cases. The statements written next to the case which matches the result is executed. You might be wondering what the default case is for! Well, in case none of the cases written below, matches with the result of the expression, then the default case is executed. You will understand it well through an example.

The example demonstrated below is a famous example which is used to explain the relevance of switch statement. Let us say we are interested in writing a program which compares the value of the two variables.

```
var a = 50;
var b = 50;
switch (true) {
case a == b:
console.log(“Equal”);
break;

case a < b:
console.log(“a is less than b”);
break;

case a > b:
console.log(“a is greater than b”);
break;

default:
console.log(“Something went wrong);
}
```

Let us see what happened in the above program. We initialized the variable x and y to 50. Since the expression is true, the switch case will execute in any case. The Switch then goes on to check each case. So Equal is displayed in the console. In case we would have set the variable x to 100 or any other value other than 50, the default case would have been executed. The Switch statement can also be used to set discount to be given based on the

purchase value. There are many such cases where implementing switch cases makes sense. You might wonder why the break keyword is written after every case. I will explain the reason soon.

Break keyword is significant because of the way the switch case works. After the evaluation of the expression, the result has to be compared with the cases listed below in that order. As soon as a match is found, the statements corresponding to that particular case is to be executed. All the other cases listed after that case have to be neglected. The break statement ensures that execution exists from the switch block after a match has been found. So if you think, writing break after the last case would make no sense. The execution will anyway, exit the switch case. We at would sometimes refrain from writing break whenever we want all the cases to be executed. Let me demonstrate that using an example.

```
var animal = ‘cow';
switch (animal) {
case 'Cow':
case 'dog':
case 'goat':
case 'cat':
console.log('This is a pet animal');
break;
```

```
case‘lion':
case ‘tiger’:
case ‘fox’:
console.log(“This is a wild animal”);

default:
console.log('Animal entry does not exist');
}
```

The above program clearly shows why it is not necessary to write a break statement in each of the switch cases. So let us understand exactly what happened here. The value of the variable animal is compared with all the cases until a match is found. Even though a match is found, it continues to look till break is encountered. Here we have smartly clubbed similar cases and made sure only one set of statements are written for one type of the case.

We have seen the use of default keyword in all the above example programs. So is it compulsory to write default statement? The answer is a big “NO” In case you are sure that the default case will never be encountered, you can choose to omit it. However, writing a default case is always recommended. So let me point out few points regarding the default case in the switch statement:

- The default case need not be compulsorily present in the switch statement.
- The default case need not be the last statement. Use the break keyword in case there are cases following it.

The Switch statement is not a replacement for if statement. Both are equally useful. Their usefulness depends on the situation.

CHAPTER 6

LOOPS IN JAVASCRIPT

What are loops?

Loops help us to execute a particular block of statements repeatedly until a condition is met. This is one of the most important features of a programming language, and all major programing language has loops. Loops are really helpful when you need to execute the same block of statements many times but with different values.

For example, say you need to add all the elements of an array of size 7, then you will do it like this without loop:

```
var values = [10, 20, 30, 40, 50, 60, 70];
var sum = 0;
sum += values[0];
sum += values[1];
sum += values[2];
```

```
sum += values[3];
sum += values[4];
sum += values[5];
sum += values[6];
console.log( sum ); // prints ‘28’ on the
console screen
```

The same code can be written with the help loop, like this:

```
var values = [10, 20, 30, 40, 50, 60, 70];
var sum = 0;
for(var i = 0; i < 7; i++)
     sum += values[i];

console.log( sum ); // prints ‘28’ on the
console screen
```

You see how small and easy this program became with loops.

Different types of loops

JavaScript provides different methods for looping, but essentially they all do the same thing, i.e., repeat a block of statements. Discussed below are the different types of loop in JavaScript.

The ‘for’ loop

The general syntax of the ‘for’ loop is as follows:

```
for([initial statement], [condition],
[incremental expression])
{
     Block of statements to repeat
}
```

Description:

- [initial statement]: This statement is executed before the loop begins. Generally, a variable is declared here which value changes in every iteration of the loop.
- [condition]: The loop continues to iterate until the condition becomes false, i.e., the loop executes as long as the condition mentioned remains true.
- [incremental expression]: This statement executes after each repetition of the loop. Generally, we change the value of the variable in the condition clause of the loop in this statement.

Example Usage:

```
for(var i = 0; i < 5; i++)
{
      console.log( i );
}
```

The result of the above code is:

0

1

2

3

4

Note that when ‘i’ becomes 5 the condition becomes false, so the number 5 is not printed on the console screen.

Each statement in the ‘for’ loop is optional but the semicolons(;) are not optional, so the following for loops valid too:

```
for(var i = 0;;)      // VALID
for(; i < 4;)         // VALID
for(;; i++)           // VALID
for(;;)          // VALID
```

The ‘for-in’ loop

The general syntax of the ‘for-in’ loop is as follows:

```
for(variable in object)
{
     Block of statements to repeat
}
```

This loop is used to iterate through each user-defined property of an object, in each iteration the name of the user-defined properties of the object passed in ‘for’ statement are successively assigned to the variable passed in the ‘for’ statement in string format.

Example usage:

```
var myObj = { ‘prop1’: 21, ‘prop2’: 42, ‘prop3’: 63};

for( var x in myObj ) {
```

```
      console.log( x + ‘ ’ + myObj[x] );
}
```

The result of the above code is:

prop1 21

prop2 42

prop3 63

The ‘while’ loop

The general syntax of the ‘while’ loop is as follows:

```
while(condition)
{
      Block of statements to repeat
}
```

This loop continues to iterate until the condition becomes false, i.e., the loop executes as long as the condition mentioned remains true.

Example usage:

```
var i = 0;
while(i < 5)
{
      console.log( i );
      i++; // incrementing the value of ‘i’
}
```

The result of the above code is:

0

1

2

3

4

The ‘while’ loop is equivalent to this form of the ‘for’ loop:

```
for(;condition;) {
      Block of statements to repeat
}
```

The ‘do-while’ loop

The general syntax of the ‘do-while’ loop is as follows:

```
do
{
     Block of statements to repeat
} while(condition)
```

This loop continues to iterate until the condition becomes false, i.e., the loop executes as long as the condition mentioned remains true.

Example usage:

```
var i = 0;
do
{
     console.log( i );
     i++; // incrementing the value of ‘i’
} while(i < 5)
```

The result of the above code is:

0

1

2

3

4

This loop is similar to the ‘while’ loop but the difference is that the condition is checked after the block of statements to be repeated is executed once, whereas in the ‘while’ loop, first the condition is checked and then based on the result of the evaluation of the condition, the loop iterates.

Example demonstrating the difference in the ‘while’ and the ‘do-while’ loop:

```
var i = 999;
while(i < 5) // the condition is false!
{
      console.log( i ); // does NOT gets
executed since the condition is false
}

do
{
```

```
      console.log( i ); // prints ‘999’ on the console screen
      //this block of the statement(s) gets executed one time even if the condition is false from the start!

} while(i < 5) // the condition is false!
```

The ‘continue’ statement

The ‘continue’ statement is used to skip to the next iteration of the loop, i.e. when the ‘continue’ statement is executed all the statements below it in the loop block are not executed, and the loop skips to the next repetition of the loop.

Example:

```
for(var i = 0; i < 10; i++) {
      if( i == 3 || i == 5 || i == 8) continue;
      console.log( i );
}
console.log(‘ LOOP STOPPED ’);
```

The result of the above code is:

0

1

2

4

6

7

9

LOOP STOPPED

You see how the numbers 3, 5 and 8 were not printed on the console screen? This is because of the ‘continue’ statement! Whenever the ‘i’ value became 3, 5 or 8 the condition in the ‘if’ clause became true and the ‘continue’ statement was executed, and so the console.log() method was skipped for that value of ‘i.'

The ‘continue’ statement can be used in any type of loop, not only in ‘for’ loop.

The ‘break’ statement

The ‘break’ statement is used to terminate the loop completely, i.e., when this statement is executed, the control flow of the program directly jumps outside the loop in which this statement was used.

Now let’s see the same example that we used in the case of ‘continue’ statement but with ‘break’ statement in place of the ‘continue’ statement :

```
for(var i = 0; i < 10; i++) {
      if( i == 3 || i == 5 || i == 8) break;
      console.log( i );
}
console.log(‘ LOOP STOPPED ’);
```

The result of the above code is:

0

1

2

LOOP STOPPED

As you can see, when the first time the ‘break’ statement was executed, the loop was completely terminated, hence the rest of the numbers were not printed.

Conclusion

Now that you have learned how to use loops in JavaScript, you can do things that you weren’t able to do before, as some things just can’t be done without loops, pretty much anything which involves repeating of a particular block of statements a dynamic number of times cannot be done without loops. Also, your production speed will increase, as, with the help of the loops, the program can be made much smaller.

CHAPTER 7

ARRAYS IN JAVASCRIPT

What is an array?

An array is a collection of items. It is basically a JavaScript object that is used to store an ordered collection of items. All the items can be accessed by using a single name which is the array name. Unlike Java, arrays in JavaScript can have non-homogenous data stored in them, i.e., items in the array can be of different data types.

An array is one of the most useful entities in any programming language; a programming language is incomplete without an array. Often we find the need to have a list of items, and it is easier to do so with arrays, instead of having a separate variable for each item, you can refer to the whole list with just one variable if you use arrays.

Initializing arrays in JavaScript

There are two ways of initializing an array in JavaScript which is listed below.

- Method 1:
  This is the most commonly used method to create an array, in this method array is declared with the help of square brackets [ ], where the items of the array are placed between these square brackets separated by commas.
  Example:

```
var myArray = []; //initialzing an
empty array

var myArray2 = [ ‘this’, ‘is’, ‘an’,
‘array’];
```

- Method 2:
  This method involves initialization of array with the help of ‘new’ keyword which is used to initialize an object of Array ‘class’.
  Example:

```
var myArray = new Array();
//initialzing an empty array
```

```
var myArray2 = new Array('this', 'is', 'an', 'array');
```

Accessing elements of array

Array's elements or items are accessed by referring to their index number which is the position of element or item in the array. The index number in JavaScript starts from 0, not from 1, so index number of the first element will be 0 and that of the second one will be 1 and so on.

Elements can be accessed by typing array name followed by the index number of the desired element enclosed within square brackets [ ].

Example:

```
var myArray = [ 'this', 'is', 'an', 'array'];

console.log( myArray.0 ); // WRONG SYNTAX, array's elements are not object's properties

console.log( myArray[0] ); // CORRECT SYNTAX, prints 'this' on the console screen

console.log( myArray[2] ); //CORRECT SYNTAX,prints 'an' on the console screen
```

Unlike objects, in arrays, elements cannot be referred to by string indexes. But arrays can store objects as one of its elements which can have string index.

Example:

```
var myArray = [ ‘this’, ‘is’, { ‘an’ : ‘example’ }];

console.log( myArray.an );// WRONG SYNTAX

console.log( myArray[2].an );//CORRECT SYNTAX, prints ‘example’ on the console screen
```

Nested array

It is possible to have nested arrays, i.e., arrays within arrays, in JavaScript. You just have to declare an array, as you normally do, as an element of another array.

Example:

```
var myArray = [ ‘an array’, [‘nested’, ‘array’]];

console.log( myArray[0] ); //prints ‘an array’ on the console screen
```

```
console.log( myArray[0][0] ); //prints
‘nested’ on the console screen
```

Modifying and adding elements to array

You can directly assign values to array elements by referring them with their index number, the same way as you do to access them.

Example:

```
var myArray = [ ‘this’, ‘is’, ‘an’,
‘array’];

console.log( myArray[2] ); // prints ‘an’
on the console screen

myArray[2] = ‘change’;

console.log( myArray[2] ); // prints
‘change’ on the console screen
```

You can add values to array in the exact manner as you assign values to array elements, if that particular index is bigger than array’s size then it gets created automatically.

Example:

```
var myArray = [ ‘this’, ‘is’, ‘an’,
‘array’];

myArray[12] = ‘change’; // index 12 doesn’t
exists but gets created when we assign
value to this index

console.log( myArray[12] ); // prints
‘change’ on the console screen
```

You can also add elements to the array by using .push() method about which you will read later in this chapter.

Size or length of an array

You can get the size/length of an array by using the ‘length’ property of the array object. As the index starts from 0, the valid indexes for an array of length ‘n’ is from 0 to (n – 1);

Example:

```
var myArray = [ ‘this’, ‘is’, ‘an’,
‘array’, ‘example’];

console.log( myArray.length ); //prints ‘5’
on the console screen
```

```
console.log( myArray[ myArray.length – 1 ]
); //prints ‘example’ on the console screen
```

When you add a new value to an array by using an index number that did not exist previously, the array length is updated accordingly.

Example:

```
var myArray = [ ‘this’, ‘is’, ‘an’,
‘array’, ‘example’];

console.log( myArray.length ); //prints ‘5’
on the console screen

myArray[12] = ‘change’;

console.log( myArray.length ); //prints
‘13’ on the console screen
```

Looping through array

Looping through an array in JavaScript is rather simple, there are many approaches to do so, some of the most common and easy ways to do so are discussed below.

- Using array's length:

  This method is simple and straightforward; it involves getting an array's length using 'length' property of the array object and then looping from 0 to the obtained length.
  Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];

var len = myArray.length;

// notice that we use '<' operator below not '<=' here
for( var i = 0; i < len; i++) {
     console.log( myArray[i] );
}

//above example prints each element of the array in separate line on the console screen
```

- Using .forEach() method:

  Example:

```
var myArray = [ 'this', 'is', 'an', 'array', 'example'];
```

```
myArray.forEach(function(item, index,
array) {
      console.log( index, item );
});

//above example prints index along
with the corresponding element of the
array in separate line on the console
screen
```

- for-of method:
  This method involves usage of the ‘of’ keyword in for loop.
  Example usage:

```
var myArray = [ ‘this’, ‘is’, ‘an’,
‘array’, ‘example’];

for (var item ofmyArray) {
  console.log(item );
}

//above example prints each element of
the array in separate line along with
other properties of the object on the
console screen
```

Some commonly used array object methods

- `.pop()`
  This method removes the last element from the array and returns it.
  Example usage:

```
var myArray = [ ‘this’, ‘is’, ‘an’, ‘array’, ‘example’];

console.log( myArray.length );//prints ‘5’ on the console screen

console.log( myArray.pop() );// prints ‘example’ on the console screen

console.log( myArray.length );//prints ‘4’ on the console screen, as the last element has been removed using .pop() method, so now length becomes 4
```

- `.push(item1, item2 ... itemN)`
  This method adds element(s) to the end of the array. The element(s) to be added are passed as an argument of this array.
  Example:

```
var myArray = [ ‘this’, ‘is’, ‘an’, ‘array’, ‘example’];
```

```
console.log( myArray.length );//prints
‘5’ on the console screen

myArray.push(‘more’,
‘elements’);//adding 2 new elements at
the end of the array

console.log( myArray.length );//prints
‘7’ on the console screen, as two new
elements have been added
```

- `.shift()`
  Same as .pop() method but instead of removing the last element of the array, this method removes the first element of the array and returns it.
  Example:

```
var myArray = [ ‘this’, ‘is’, ‘an’,
‘array’, ‘example’];

console.log( myArray.length );//prints
‘5’ on the console screen

console.log( myArray.shift() );//
prints ‘this’ on the console screen

console.log( myArray.length );//prints
‘4’ on the console screen, as the
first element has been removed using
```

```
.shift() method so now length becomes
4
```

- `.unshift(item1, item2 ... itemN)`
  Same as .pop() method but instead of adding element(s) to the end of the array, this method adds the element(s) at the start of the array. Example usage:

```
var myArray = [ ‘this’, ‘is’, ‘an’,
‘array’, ‘example’];

console.log( myArray.length );//prints
‘5’ on the console screen

myArray.unshift(‘more’,
‘elements’);//adding 2 new elements at
the end of the array

console.log( myArray[0] ); );//prints
‘more’ on the console screen

console.log( myArray.length );//prints
‘7’ on the console screen, as two new
elements have been added
```

- `.join(separator)`
  This method returns a string consisting of the elements of the array with the argument passed as the separator between them. The

separator argument is optional if it is not passed then the separator is taken as a comma.
Example:

```
var myArray = [ ‘this’, ‘is’, ‘an’, ‘array’, ‘example’];

console.log( myArray.join(‘-’) );
//prints “this-is-an-array-example” on the console screen
```

- .slice(startIndex, endIndex):
  This method extracts the part of the array starting from startIndex and ending at (endIndex - 1) and returns it as a new array; the original array stays unaffected. Both of the arguments are optional, if not specified then whole array copy is returned.
  Example:

```
var myArray = [ ‘this’, ‘is’, ‘an’, ‘array’, ‘example’];

var newArray = myArray.slice(1, 3);

console.log( newArray ); // prints [‘is’, ‘an’] on the console screen
```

- `.indexOf(searchItem)`
  This method returns the first index of the element which is being searched for based on the argument passed, returns -1 if no such element was found.

```
var myArray = [ ‘this’, ‘is’, ‘an’, ‘array’, ‘an’,‘example’];

console.log( myArray.indexOf(‘an’) );//prints ‘2’ on the console screen

console.log(myArray.indexOf(‘random’) );//prints ‘-1’ as no such element is there in the array
```

- `.lastIndexOf(searchItem)`
  This method is same of .indexOf() method but instead of returning the first index of the searched element it returns the last index of it, if a match was found, else it returns -1.

```
var myArray = [ ‘this’, ‘is’, ‘an’, ‘array’, ‘an’,‘example’];

console.log( myArray.lastIndexOf(‘an’) );//prints ‘4’ on the console screen
```

```
console.log(myArray.indexOf('random')
);//prints '-1' as no such element is
there in the array
```

Arrays are very useful. So learn the array concepts very well. I hope this chapter was helpful.

CHAPTER 8

TYPE CASTING

What is type casting?

Type casting means data type conversion. It may be explicit or implicit. Explicit type casting is the conversion done by the programmer manually usually by use of a function whereas implicit type casting is the conversion done by the compiler automatically.

Implicit type casting

Implicit type casting is done by the compiler automatically when we mix variables of the different data type in one expression, in layman language, it is the conversion done by JavaScript's compiler to convert the "wrong" data type to the "right" data type. Some of the situations where implicit type conversion occurs are discussed below.

- Whenever you try to output a variable, i.e., through functions like console.log() and alert() or assigning DOM functions, they are automatically converted to string type by calling .toString() method.

  Examples:

```
var myArray = [‘this’, ‘is’, ‘an’, ‘example’];

alert( myArray ); //shows ‘this,is,an,example’ in the alert dialog box

var date = new Date();

alert( date ); //shows the current date and time in ISO string format, example ‘Tue Aug 01 2017 18:45:27 GMT+0530 (India Standard Time)’
```

- Using ‘+’ operator with a string and any other data type in one expression results in compiler converting other data types to string and then concatenating them all.
  Examples:

```
console.log( ‘test’ + 2 + 1 );
//prints ‘test21’ on the console
screen

console.log( “2” + 1); //prints ‘21’
on the console screen

console.log( “2” + null ); //prints
‘2null’ on the console screen
```

So what if you want to add digits and then concatenate it with a string like in the first example? Don’t worry there is a workaround for that; you can do it if you enclose the numbers within parentheses (), this will make their order of precedence higher thus leading to the execution of number addition being done first and this they are converted to string after they are added.
Example:

```
console.log( ‘test’ + (2 + 1) );
//prints‘test3’ on the console screen
```

- Using ‘-,' ‘*’ or ‘/’ operator in an arithmetic expression leads to all of the operands of expression being converted to a number type.
  Examples:

```
console.log( “2” - 1); //prints ‘1’ on
the console screen

console.log( “20” – “5” ); //prints
‘15’ on the console screen

console.log( “20” / “5” ); //prints
‘4’ on the console screen

console.log( “20” * “5” ); //prints
‘100’ on the console screen
```

Explicit type casting

Often we need to convert one data type to another manually, the process of doing so is called explicit type casting. We shall learn about different approaches to do so below.

Converting to string data type

There are three ways of converting other data type to String data type, which are:

- String() wrapper function:
  This method involves passing the value to be converted to String() wrapper function which converts the argument to the string data type and returns it.
  Examples:

```
var x = 21;
var y = String( x );

console.log( typeof x ); //prints
‘number’ on the console screen

console.log( typeof y ); //prints
‘string’ on the console screen

console.log( typeof String(null) );
//prints ‘string’ on the console
screen
```

- .toString() method:
  You can use the .toString() method on all data types to convert them to string data types. However, you cannot run .toString() method on a null or undefined type value but you can run it on NaN type value.
  Examples:

```
var x = 21;
var y = x.toString;

console.log( typeof x ); //prints
‘number’ on the console screen

console.log( typeof y ); //prints
‘string’ on the console screen
```

```
var z = null.toString(); //WRONG, you
cannot run .toString() method on null

var a = undefined.toString();//WRONG

var b = NaN.toString();// WORKS

console.log( typeof b );//prints
‘string’ on the console screen

var bool = false.toString();//works on
boolean too

console.log( typeof bool );//prints
‘string’ on the console screen
```

- Using ‘+’ operator with an empty string:
  In this method, we indirectly use implicit type casting by using ‘+’ operator with an empty string, so they are converted to a string type value automatically during compile time. But since the programmer is doing this intentionally, specifically to convert it to string data type, so it can be termed as an explicit type data.

  Examples:

```
var x = 21;
var y = ‘’ + x;
```

```
console.log( typeof x ); //prints
‘number’ on the console screen

console.log( typeof y ); //prints
‘string’ on the console screen

console.log( typeof ( ‘’ + null ) );
//prints ‘string’ on the console
screen
```

Converting to number data type

Boolean type values false and true are converted to 0 and 1 respectively when converting to number data type. If values cannot be converted to be a number, they become NaN which is a 'number' data type value though NaN stands for “Not a Number.” Here are the ways to convert other data types to number data type:

- Number() wrapper function:
  This method involves passing the value to be converted to Number() wrapper function which converts the argument to the number data type and returns it.Examples:

```
var x = “21”;
var y = Number( x );// y’s value is 21
now
```

```
console.log( typeof x ); //prints
‘string’ on the console screen

console.log( typeof y ); //prints
‘number’ on the console screen

console.log( typeof Number(null) );
//prints ‘number’ on the console
screen

var z = Number( “test” );// z is now
‘NaN’

console.log( Number(true) );//prints
‘1’ on the console screen
console.log( Number(false) );//prints
0’ on the console screen

var date = Number(new Date());

console.log( date );// prints current
date and time as timestamp, i.e.,
number of milliseconds passed since
1st of Jan 1970, example:
1501596947774
```

- Using ‘+’ operator before the value to be converted:
  This is a short way of converting a data type to number data type.

Examples:

```
var x = “21”;
var y = + x; // y’s value is 21 now

console.log( typeof x ); //prints
‘string’ on the console screen

console.log( typeof y ); //prints
‘number’ on the console screen

var z = + “test”;

console.log( z ); //prints ‘NaN’ on
the console screen
```

- Using parseInt() method:

  The value to be converted is passed as this function’s argument, and it is returned as number data type. Note that this function converts the argument to an integer, i.e., no decimal places in the number. If the argument can’t be converted ‘NaN’ is returned. Unlike Number() method if there are spaces present between numbers in a string type argument the first number before space is returned, whereas in the Number() method ‘NaN’ is returned in case of a space being present.

Example:

```
var x = “21”;
var y = parseInt(x); // y’s value is
21 now

console.log( typeof y ); //prints
‘number’ on the console screen

console.log( parseInt(“21 32”)
);//prints ‘21’ on the console screen

console.log( parseInt(“test 21 32”)
);//prints ‘NaN’ on the console screen

console.log( parseInt(“45.59”));
//prints ‘45’ on the console screen
```

This chapter talked about type casting. I hope you have got a clear understanding of type casting in JavaScript after reading this chapter.

CHAPTER 9

ERROR HANDLING

What are errors?

Errors are basically unexpected conditions that may or may not halt the execution of a program during the runtime of a program. Errors can arise due to programmer's mistake, unexpected input or other such reasons. Any program is prone to errors. Hence it becomes necessary to have a way of handling them without disrupting program's flow or have an alternative flow to make sure the application doesn't function in an unexpected manner.

Handling errors

In JavaScript error handling is done with the help of four statements: 'try,' 'catch,' 'finally,' and 'throw.'

try-catch blocks

The try-catch blocks are the fundamental building blocks of error handling in JavaScript. The code that is expected to produce an error is placed inside the ‘try’ block and right at end of the ‘try’ block catch block is placed where the code to handle the error is placed. Catch block has an argument in which the details of the error is passed.

General Syntax:

```
try {
      //code that may produce an error here
 }
catch(errObject) {
      //code to handle error when it is
produced goes here
}
```

The process of capturing the errors and forwarding it is called ‘throwing’ so when an error occurs the interpreter ‘throws’ an error which we ‘catch’ using the catch block, think of it as throwing and catching a ball.

If any statement inside the try block throws an error, the control is shifted to catch block immediately. If no error is thrown then catch block is skipped.

Example:

```
try {
     nonExistentFunction(); //running a
function that isn’t available in the
current context
}
catch(err) {
console.log( err.message ); //shows error
detail if it occurs
}
```

In the above example since the function in the try block isn’t available in current context hence an error is thrown, which is caught in the catch block and thus the output of the above program is “nonExistentFunction is not defined”.

‘finally’ block

The ‘finally’ block is placed after the try-catch block. This block executes every time; it doesn’t matter whether an error is thrown by the code in ‘try’ block or not, unlike catch block the ‘finally’ block will execute every time.

Example:

```
try {
      // code to be tested for error
}
catch(err) {
      // error handling code
}
finally {
      console.log('This runs every time');
}
```

The error object

As you know when an error is thrown its details gets passed along as an object in the 'catch' block, we will talk about this object here.

The error object has two main properties which are:

· name

This property indicates the error name; we can use this property to check which type of error has been thrown.

· message

This property contains the details about the thrown error in the form of a message; this property is used to find out which segment of the program caused the error.

There are five types of predefined errors that can be thrown by JavaScript which has been discussed below.

ReferenceError

This error is thrown when you try to use a function or a variable that is not declared or is not available in the current context.

Example:

```
try {
     var x = y * 5;
}
catch(err) {
     console.log( err.name );// prints ‘ReferenceError’ on the console screen
     console.log( err.message );// prints ‘y is not defined’ on the console screen
}
```

RangeError

This error is thrown when you try to use a value that is not in the range of legal values.

Example:

```
try {
     var x = var x = new
Array(99999999999999999999999999);
}
catch(err) {
     console.log( err.name );// prints
‘RangeError’ on the console screen
    console.log( err.message );// prints
‘Invalid array length’ on the console
screen
}
```

SyntaxError

This error is thrown when the wrong syntax is used somewhere in the program.

Example:

```
try {
     eval( 'var x =  +;)' );
}
catch(err) {
```

```
     console.log( err.name );// prints
‘SyntaxError’ on the console screen
     console.log( err.message );// prints
‘Unexpected token ;’ on the console screen,
since there is expected to be a value after
the + operator in the eval statement but a
; was found.
}
```

URIError

This error is thrown when illegal characters are used in the URI parameter of a URI handling function.

Example:

```
try {
     decodeURIComponent('%');
 }
catch(err) {
     console.log( err.name );// prints
‘URIError’ on the console screen
      console.log( err.message );// prints
‘URI malformed’ on the console screen
}
```

TypeError

This error is thrown when a function as an object property is used and that function isn't a property of that object

Example:

```
var x = 45;
try {
      x.abcd();
 }
catch(err) {
      console.log( err.name );// prints 'TypeError' on the console screen
       console.log( err.message );// prints 'num.abcd is not a function' on the console screen
}
```

Throwing custom errors

You can throw custom errors with the help of the 'throw' statement. You can throw errors as an object, number, string or a boolean type value using this statement, but it is recommended that you throw the error as an object which has same general structure as other error objects have, i.e., they have the 'name'

and the ‘message’ property, you can have additional properties.

Example:

```
var input = 160;
try {
      if(input > 100) throw
{‘name’:’CustomError’, ’ message’:’Input
value is bigger than 100’};
}
catch(err) {
      console.log( err.name ); //prints
‘CustomError’ on the console screen
      console.log( err.message ); //prints
‘Input value is bigger than 100’
}
```

You can also utilize the pre-made JavaScript error objects specified above in the ‘throw’ statement.

Example:

```
var input = “MY URI”;
try {
      //doing something with URI
      if( input.length < 20 ) throw new
URIError(‘URI should have more than 20
characters’);
```

```
}
catch(err) {
      console.log( err.name ); //prints
‘URIError’ on the console screen
      console.log( err.message ); //prints
‘URI should have more than 20 characters’
}
```

Conclusion

Now that you have learned how to handle and produce custom errors in JavaScript, it is imperative that you use this in your code in production. A good application is the one which can handle unexpected situations and keep on running without abruptly disrupting the flow of the program. Errors are going to be “thrown,” make sure your application can “catch” them all.

CHAPTER 10

REGULAR EXPRESSIONS

What are regular expressions?

Regular expressions represent a pattern. In JavaScript, regular expressions can be used to perform operations such as searching a pattern, replacing a pattern, checking if a string matches a given pattern or breaking of a string into smaller strings based on a specific pattern. In JavaScript, regular expressions are objects.

Making a regular expression

There are two ways by which you can create regular expressions in JavaScript, which are:

- Using regular expression literal:
  In this method, the pattern is specified within two slashes // followed by the character(s) which are known as modifiers. You will learn

about what modifiers are later in this chapter; modifiers are optional.
Syntax:

```
/pattern/modifiers
/pattern/
```

If this method is used, then the regular expression is compiled when the script is loaded and thus if the regular expression is constant, this method improves the performance of the program.

- Using constructor of RegExp object:
  In this method, we use the 'new' keyword to initialize a new object of RegExp which stands for Regular Expression. The pattern is passed as the first argument to the RegExp constructor and the modifier as the second argument, note that the modifier argument here is optional too.
  Syntax:

```
new RegExp( "pattern", "modifiers" );
new RegExp("pattern");
```

If this method is used, then the regular expression is compiled at runtime, this method is generally used where regular expression is not a constant.

Modifiers

A modifier is used to change the way a match of pattern based on regular expression is done. Listed below are the modifiers and the details on how they affect the operation.

- `i`
  Makes the match case insensitive, by default matching done is with regular expressions is case sensitive.
- `m`
  Makes the match of pattern extend to multiple lines one if there are more than one lines in the string that the match is being performed on.
- `g`
  By default, the operation stops after finding the first match for the pattern, but if this modifier is used the operations don't stop at the first match and thus performs a 'g'lobal match.

Simple patterns

If you want to match a sequence of character directly, then you can write it in the place of the pattern. For example, the regular expression /test/ will match any string with 'test' in it, so there would be a match for a string like 'this is a test, ' but there would not be any

match for string like ‘this is a est’ because the character ‘t’ is missing from the sequence specified.

Certain characters denotes something special in a regular expression like ‘*’ or ‘.’, if you want to match these characters directly then you need to escape them, i.e., add a backslash before them. So for matching the exact string ‘te*st’, the regular expression will be /te*st/. You will learn about all the characters with a special significance in regular expressions in the very next section.

Special character(s)

Here is a list of special character(s) that can be used for a regular expression and what they do:

- ^

  If present at the start of the regular expression, this character signifies that the following pattern should be matched from the beginning of the input. If the ‘m’ modifier is used, then the starting of each line is also tested for the match.

  For example, the regular expression /^test/ will not match anything in the string ‘this is a test’ as ‘test’ is not present at the beginning of the

string, but the same regular expression will match with the string ‘test is going on’.

- `$`
  If present at the end of the regular expression, this character signifies that the pattern should be matched with the end of the line. If the ‘m’ modifier is used then the end of each line is tested for the match.

  For example, the regular expression /test$/ will not match anything in the string ‘test is going on’ as ‘test’ is not present at the end of the string, but the same regular expression will match the string ‘this is a test.'

- `*`
  This character is used to match the expression preceding this character zero or more times.

  For example, the regular expression /te*st/ will match ‘t’ followed by zero or more ‘e’ and that followed by ‘st’, so the string ‘this is a teeeeeest’ as well as the string ‘this is a tst’ will be matched but the string ‘this is teees’ won’t be matched since a ‘t’ is missing from the sequence to be matched.

- `+`
  This character is used to match the expression preceding this character one or more times, i.e., if the preceding expression occurs at-least once.

  For example, the regular expression /te*st/ will match the string ‘this is a teeeeeest’ but won’t match the string ‘this is a tst’ or the string ‘this is a tees’.

- `?`
  This character is used to match the expression preceding this character zero or one time.

  For example, the regular expression /te?st/ will match the string ‘this is a test’ and ‘this is a tst’ but won’t match the string ‘this is a teest’ or the string ‘this is a tes’.

- `{x}`
  This form of expression is used to match exactly x occurrences of the preceding expression, where x is a positive integer.

  For example, the regular expression /te{3}st/ will match the string ‘this is a teeest’ but won’t

match the string ‘this is a test’ or the string ‘this is a teeeeest’.

- `{x, y}`
  This form of expression is used to match any number of occurrences between x and y of the preceding expression, where x and y both are positive integers and x is less than or equal to y.

  For example, the regular expression /te{3, 6}st/ will match the string ‘this is a teeest’ and ‘this is a teeeeest’ but won’t match the string ‘this is a test’ or the string ‘this is a teeeeeeest’.

- `{x, }`
  This form of expression is used to match at-least x occurrences of the preceding expression, where x is a positive integer.

  For example, the regular expression /te{3,}st/ will match the string ‘this is a teeest’ but won’t match the string ‘this is a test’ or the string ‘this is a teest’.

- `.`
  This character is used to match any character except a newline one.

  For example, the regular expression /te.st/ will match the string ‘this is a tOst’ and ‘this is a tZst’ but won’t match the string ‘this is a tst’.

- `[abc]`
  This form of expression is used to match any character or expression that is present between the square brackets.

  For example, the regular expression /t[eyz]st/ will match the string ‘this is a test’ and ‘this is a tzst’ but won’t match the string ‘this is a tpst’ or the string ‘this is a trst’.

  You can use ranges like a-z and 0-9 which matches all the character from ‘a’ to ‘z’ and ‘0’ to ‘9’ respectively.For example, the regular expression /t[a-z]st/ will match the string ‘this is a test’ and ‘this is a tkst’ but won’t match the string ‘this is a t9st’ or the string ‘this is a t2st’, but the regular expression /t[a-z0-9]st/ will match all the strings mentioned in this particular example.

- `[^abc]`
  This form of expression is used to match anything EXCEPT the character(s) or expression(s) that is present between the [^ and].

  For example, the regular expression /t[^eyz]st/ will match the string ‘this is a tpst’ and ‘this is a trst’ but won’t match the string ‘this is a test’ or the string ‘this is a tyst’.

  You can use the ranges, just like in the above-mentioned expression, in this form of expression too.For example, the regular expression /t[^a-z]st/ will match the string ‘this is a t9st’ and ‘this is a t8st’ but won’t match the string ‘this is a test’ or the string ‘this is a tast’.

- `(abc)`
  This form of expression is called a capture group. The pattern between the () is matched normally but is stored in memory for later use; you will learn about this expression usage in methods like .exec() later in this chapter.

Meta Characters

These are the special character sequences that are used to match certain characters or range of characters. Some of the most used metacharacters are:

- \w
  Matches any word character, i.e., any alphanumeric character including underscore.
- \W
  Matches any non-word character. All the characters that are not matched by \w are matched by this.
- \d
  Matches any digit, i.e., any character from 0 to 9.
- \D
  Matches any non-digit, i.e., any character except from 0 to 9.
- \w
  Matches any whitespace character, i.e., characters like tab, space, line ends etc.
- \W
  Matches any non-whitespace character, i.e., any character except whitespace characters like tab, space, line ends, etc.

Using regular expressions

Now that you have learned about how to make up a regular expression, we can move on to learn about how to put them into action. Listed below are the methods which you can use with a regular expression.

- `regexObject.test(string)`
  This method tries to match the regular expression with the string passed as the argument, if it is matched, this method returns true else false.
  Examples:

```
var result1 = /example/.test('this is an example');
var result2 = /example/.test('what is this');

console.log( result1 ); //prints 'true' on the console screen
cosnole.log( result2 );//prints 'false' on the console screen
```

- `regexObject.exec(string)`
  This method tries to match the regular expression with the string passed as the argument, if it is matched, this method returns an array filled with information else it returns

null. The returned array first item is the matched string and the second is the first capture group present if any and third item is the second capture group present if any and so on.

Examples:

```
var result1 = /exam(pl)e/.test('this is an example');
var result2 = /example/.test('what is this');

console.log( result1[0] ); //prints 'example' on the console screen
console.log( result[1] ); //prints 'pl' on the console screen
cosnole.log( result2 ); //prints 'null' on the console screen
```

- `string.search(regex)`

  This method tries to find a match of the regex passed as the argument. If found, it returns the index(position) of the match else returns -1. Note that the index begins from 0, not from 1. So the first character's index in a string is 0 and the second character's index is 1.

  Examples:

```
var result1 = 'an example'.search(/a.p/);
```

```
var result2 = ‘an
example’.search(/55/);

console.log( result ); //prints ‘5’ on
the console screen
cosnole.log( result2 ); //prints ‘-1’
on the console screen
```

- `string.replace(regex, replacer)`
  This method tries to find a match of regex passed as the first argument if found it replaces the match with the second argument(replacer).
  Example:

```
var original = “this is an example”;

//replcaes first s and a with 9
var new = original.replace(/[sa]/,
‘9’);

//replcaes each s and a with 9, since
the‘g’ modifier is used
var new2 = original.replace(/[sa]/g,
‘9’);

console.log(new);//prints ‘thi9 is an
example’ on the console screen
console.log(new2);//prints ‘thi9 i9 9n
ex9mple’ on the console screen
```

We can also use capture groups in the regular expression in this function and then use them in the replacer string. If in the replacer string we use a $ sign followed by a number then that $ sign with the number is replaced by the capture group of that index. For example, $1 gets replaced by the first capture group in the regular expression and $2 by the second one and so on.

Example:

```
var original = “this is an example”;

//places _ on both side of each s and
a
var new = original.replace(/([sa])/g,
‘_$1_’);

console.log(new);//prints ‘thi_s_ i_s_
_a_n ex_a_mple’ on the console screen
```

In this chapter, you have learned about regular expressions. I hope you have got a basic idea of regular expressions in JavaScript after reading this chapter.

CHAPTER 11

HOISTING

What is hoisting?

The word hoist means to lift or raise up by means of some mechanical device like a pulley, but in JavaScript, hoisting means that the functions and variable declarations are moved to the top of the scope or context they are declared in, i.e., lifted up as in hoist, thus the word ‘hoist’ing.

The declarations aren’t really moved to the top, they are just put first into the compiled code.

Variable hoisting

For example the following code:

```
x = 21;
console.log( x );
var x;
```

Really compiles as this:

```
var x;
x = 21;
console.log(x);
```

As you can see the declaration has moved to the top of the current context, this is what hoisting is.

Let's understand this with a more suitable example:

```
console.log( x ); // prints ‘undefined’ on
the console screen
console.log( y ); // throws a
ReferenceError saying ‘y is not defined’
var x;
```

This happens because ‘x’ being declared in the current context, though it is below the console.log line, it is moved at the top of the context and as there is no assignment done to ‘x’, it is set to ‘undefined’ by default, whereas in case of ‘y’ it is not available anywhere in current context, therefore, a ReferenceError is thrown.

Note that the assignment operation is not hoisted, only the declaration is.

Example:

```
console.log(x); //prints ‘undefined’ on the console screen
var x; // only this statement is hoisted
x = 21; // this is not
```

You must be wondering about, what if we assign the value to variable while declaring it, like ‘var x = 21;’, though here it seems that you are assigning value to variable while declaring it but in fact internally the variable is first declared and then the value is assigned to it, so it is equivalent to ‘var x; x = 21;’. So if you declare a variable like this below the line where it is being used but in the same context then only declaration part is hoisted.

Example:

```
console.log(x); //prints ‘undefined’ on the console screen
var x = 21;
```

This is how the above code is compiled:

```
var x;
console.log(x); //prints ‘undefined’ on the
console screen
x = 21;
```

Function hoisting

Just as variable hoisting, functions are also hoisted. As a result the functions can be called even before they are declared, given that it is declared in the current context or scope.

Example:

```
test();
function test() {
     console.log(‘This is a test’);
}
//prints ‘This is a test’ on the console
screen
```

It is to be noted that function expressions, i.e., functions that are assigned to variables through the assignment operator ‘=’ are not hoisted. The case of hoisting function expression is same as assigning a value a variable, which has been discussed above.

Example:

Just as variable

```
test(); // throws a TypeError saying ‘test
is not a function’
testRandom(); // throws ReferenceError
saying ‘testRandom is not defined’
var test = function {
     console.log(‘This is a test’);
};
```

In the above example, the difference in types of error is caused due to hoisting. As visible the variable ‘test’ is declared in the current scope therefore its declaration is moved at top and hence it is declared, though its value is undefined at the point where it is used, so a TypeError is thrown which indicates that the ‘test’ is defined but is not a function, whereas the function ‘testRandom’ is not declared anywhere, so a ReferenceError is thrown.

Order of precedence of hoisting

Functions are always hoisted over variable declaration.

Example:

```
function test() {
      //function code
}
var test;

console.log( typeof test ); //prints
‘function’ on the console screen
```

Even if the position of the variable line and function line are swapped in the above example, the output will remain the same.

But functions are not hoisted over variable declaration, given the assignment is done above the line where it is being used.

Example:

```
function test() {
      //function code
}
var test = 21; //now assignment is being
done too
console.log( typeof test ); //prints
‘number’ on the console screen
```

Even if the position of the variable line and function line are swapped in the above example, the output will remain the same.

As mentioned this is valid as long as the assignment is done above the line where it is being used, when the assignment is done below, then the function is hoisted over the assignment.

Example:

```
function test() {
     //function code
}
var test;

console.log(typeof test);//prints
‘function’ on the console screen
test = 21;
console.log(typeof test);//prints ‘number’
on the console screen
```

Conclusion

It is always the best practice to declare variables or functions on the top of the scope where they will be used to avoid any confusion, even though due to hoisting variable declaration will be moved to the top of the scope or context automatically.

CHAPTER 12

‘USE STRICT’ IN JAVASCRIPT

What is ‘use strict’?

‘use strict’ is not a statement but a literal expression, it indicates that the code should be executed in the strict mode, it is a more restricted variant of JavaScript, you will learn about strict mode later in this chapter. This directive was added in ECMAScript 5 and is rather new and will not work in old browsers.

Using strict mode

You can apply strict mode to entire code or to specific functions; you cannot apply it to specific code blocks like that of a ‘for’ loop.

Applying strict mode to the whole script

To apply strict mode to the whole script, you need to put the statement ‘use strict’; at the top of the script,

i.e., before any other code. If you use this expression in the middle of the code, then it will not have any effect.

Example:

```
//top of script
‘use strict’; // or “use strict’; works too
//other code now
var example = 45;
```

You cannot use a strict mode script with a non-strict mode script; you can only use strict mode script with other strict mode scripts and non-script mode scripts with other non-strict mode scripts.

Applying strict mode to functions

To apply strict mode to a function, you need to put the statement ‘use strict’; at the top of the function body, i.e., before any other code in the function body.

Example:

```
//strict mode is NOT applicable here
function example() {
     ‘use script’;// or “use strict’; works too
     // strict mode is applicable here
```

```
}
//strict mode is NOT applicable here
```

The strict mode

In strict mode, there are certain differences in the semantics of JavaScript, the purpose of these changes is to help us to write more “secure” JavaScript. The strict mode not only makes changes to syntax but also to runtime behavior. The strict mode also helps in future proofing; you will learn how in the last segment of this chapter.

Changes which occur in strict mode are discussed below.

Silent errors become errors

Silent errors are the mistakes, generally minor mistakes, in the code which are accepted in the non-strict mode but in the strict mode, these mistakes lead to the throwing of errors. In the non-strict mode, not reporting these minor mistakes(silent errors) fixed the issue for short-term but these silent errors can sometimes lead to major bugs in the long run, thus by eliminating silent errors, the strict mode makes JavaScript more “secure”, as these mistakes are now reported and can be fixed promptly.

Listed below are silent errors which become errors in strict mode:

- When you perform an assignment operation on an undeclared variable it creates a new property on the global object and doesn't throw any error in the non-strict mode but in the strict mode this is not allowed and a ReferenceError is thrown.
  Example:

```
var y = 5;
x = y + 10; // works, even though it is not declared anywhere

//other script:
'use strict'; // strict mode activated!
var y = 5;
x = y + 10; // doesn't works, a ReferenceError is thrown
```

- Some assignment operations fail without throwing any error in the non-strict mode, but the same assignment operations throw an error in the strict mode. These failed assignment operations happen in the following conditions:

- When you try to assign a value to non-writable global variables like ‘NaN or ‘undefined’.
  Example:

```
‘use strict’;
var NaN = 21; // throws a TypeError
Var ‘undefined’ = 21; // throws a TypeError
```

- When you try to assign values to read-only properties of an object.
  Example:

```
‘use strict’;
var example = {};
// now we will define a read-only property
Object.defineProperty(example, ‘test’, {value:21, writable:false});
example.test = 99; // throws TypeError
```

- When you try to assign values to get-only properties of an object.
  Example:

```
‘use strict’;
// now we will define get-only
property
var example = { get test() {
return 21; } };
example.test = 99; // throws
TypeError
```

- When you try to assign values to NEW properties of a non-extensible object, an object can be made non-extensible with the help of Object.preventExtensions() method.
  Example:

```
‘use strict’;
var example = {};
// now we will make the object
non-extensible
Object.preventExtensions(example
);
example.newProperty = 99; //
throws TypeError
```

- In the strict mode, you cannot access the variables declared in the eval function, whereas in the non-strict mode you can.
  Example:

```
‘use strict’;
eval(‘ var test = 21 ’);
console.log( test ); // throws
ReferenceError but in the non-strict
mode prints ‘21’ on the console screen
```

- Deleting variables, functions and undeletable properties of an object throw error in the strict mode whereas in the non-strict mode deleting these had no effect.
  Example:

```
‘use strict’;
function test(x) { console.log(x); }
delete test; // throws SyntaxError

var test2 = 21;
delete test2; // throws SyntaxError


Delete Object.prototype; // throws
TypeError
```

- In the non-strict mode, if you use the duplicate parameters in the function prototype, function declaration line is called its prototype, and then, if you use that duplicated parameter in the function body, the name of the variable being duplicated will always refer to the last

parameter with duplicate name, but in strict mode having duplicate parameters will thrown an error.
Example:

```
‘use strict’;
function test(x, x) { //throws
SyntaxError
     console.log(x);
}
```

- In JavaScript’s non-strict mode you can use the octal number system by adding the prefix ‘0’ to a number, but since octal number system is not a part of ECMAScript 5 and the leading ‘0’ can create confusion, so in the strict mode prefixing a number with a ‘0’ throws a SyntaxError, you can use the prefix ‘0o’ in strict mode for octal numbers.

```
‘use strict’;
var x = 021; //throws SyntaxError
var y = 0o21; // works, this is an
octal number
```

- The strings ‘arguments’ and ‘eval’ are reserved keyword in strict mode and cannot be used as a function or a variable name.
  Example:

```
‘use strict’;
var arguments = 21; //throws
SyntaxError
var eval = 21; //throws SyntaxError
```

Future Proofing

Strict mode helps in future proofing as certain keywords which will be reserved for the future versions of ECMAScript are prohibited to be used as variable or function name. These keywords include:

- public
- private
- protected
- static
- let
- yield
- interface
- implements
- package

Example:

```
‘use strict’;
var yield = 21; //throws SyntaxError
```

Conclusion

Using strict mode helps us to avoid silly mistakes, but it also makes it difficult for us to optimize the code, but since it helps in future proofing too, it is recommended for novice developers to code in the strict mode whereas experienced developers may code in the non-strict mode to have more flexibility and optimization.

CAPTER 13

FORM VALIDATION WITH JAVASCRIPT

What is form validation?

Form data is checking for validity of data entered in the form and providing feedback if any value was not valid. Have you ever noticed that on major sites when you leave a required field empty and try to submit the form and it doesn't submit instead a message is shown to you that required field is empty? That is form validation, though it is not limited to checking if the field is empty only, it also includes checking for any other condition like if a numeric value is within a range of values or any such custom conditions.

Why form validation?

It is crucial that the data in the form is checked for its validity before sending it to the server to prevent our

application from malfunctioning. Though the backend on the server may have these checks, checking it beforehand and stopping it from being sent if it is invalid saves our server's bandwidth.

Though HTML5 supports basic form validation using some attributes in the elements, we will learn how to do it with JavaScript since doing it with JavaScript is more flexible.

Binding function to listen to form submit event

In this segment we will learn how to bind function to listen to form submit event, which means that this function will be called when the form is submitted by the user either by pressing the submit button or the 'Enter' key on the keyboard, this is where we will check form values for their validity.

All the JavaScript code examples in this chapter will be based on the following form:

```
<form name='myForm' action='/target.php'
method='post' id='myForm'>
      Number:   <input type='text'
name='fnum' id='fnum'>
      Required: <input type='text'
name='frequ' id='frequ'>
```

```
      Range:    <input type='text'
name='range' id='range'>
      <input type='submit' value='Submit'>
</form>
```

Every time the form is submitted the form element emits a 'submit', we will use this fact to bind a function to the submit event using the .addEventListener() method in the following manner:

```
var form =
document.getElementById('myForm'); // get
our form as object in JavaScript
// now let's bind the listener function to
submit event
form.addEventListener('submit', function
(event) {
      // this function will run when the
form is submitted
}, false);
```

Preventing form from being submitted

You must have noticed the 'event' argument in the function; we use that event argument to prevent the form from being submitted. This is done by using the

method .preventDefault() on the event object passed as the function argument.

Example:

```
form.addEventListener("submit", function
(event) {
      // in this example form is never
submitted since the following
      // method stops the submitting process
every time
      event.preventDefault();
}, false);
```

Getting form input box's value

To get form's input box's value, we will first retrieve the input box element as an object in JavaScript using the document.getElementById() method. Then we can easily set or get the value of input box using the 'value' property of the object retrieved.

Example:

```
var inputbox =
document.getElementById('frequ');
// getting value
alert( 'Value of fnum ID input box is' +
inputbox.value );
```

```
//setting value
inputbox.value = ‘custom value’;
```

Basic Validation

Now let’s combine what we have learned up to now to setup basic form validation. Some of the basic validation examples are discussed below.

- Required field validation:
  This can be done by retrieving the input box’s value and checking if it is empty or not.

  Example:

```
form.addEventListener("submit",
function (event) {
      var val =
document.getElementById('frequ').value
;
      if(val == ‘’) {
            alert( ‘Required field is
empty’ ); // showing user message
            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

- Number field validation:
  To do this, we will first try converting the value to a numeric data type by prefixing it with a '+' operator and then checking if it's a number by using Number.isInteger() method.

  Example:

```
form.addEventListener("submit",
function (event) {
      var val =
document.getElementById('fnum').value;
      if( ! Number.isInteger( +val ) )
{
            alert( 'Field must have
numeric value only' ); // showing user
message
            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

- Simple number range validation:
  This can be simply done by first converting the value to number data type and then comparing the value with our range values.

Example:

```
form.addEventListener("submit",
function (event) {
      var val =
document.getElementById('range').value
;
      val = +val; // converting it to
number data type

      //now let’s compare
      if( val < 5 || val > 10 ) {
            alert( ‘Field must have
numerical value BETWEEN 5 and 10 only’
); // showing user message
            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

You will see more complex validation examples in the next chapter.

Conclusion

It is always considered the best practice to add form validation code no matter how small or insignificant the code maybe, in addition to what has been told

about why the form validation is important it also helps the user to fill the form properly by providing useful feedback.

CHAPTER 14

VALIDATION EXAMPLES

Introduction

In this chapter, we will see more complex examples of form validation along with examples showing how to validate radio boxes, check boxes, and list type inputs.

Text based input field validation examples

For the examples in this section of the chapter we will be using the following HTML form:

```
<form name='myForm' action='/target.php'
method='post' id='myForm'>
      Field:   <input type='text'
name='myInput' id='myInput'>
      <input type='submit' value='Submit'>
</form>
```

Examples:

- Limiting input text length within a range:
  For this, we will first get the input text and the use the ‘length’ property to get its length and then compare it with our desired length.

  Limiting input text length to be not more than 15:

```
var form = document.getElementById('myForm');

form.addEventListener("submit", function (event) {
      var val = document.getElementById('myInput').value;
      if( val.length > 15 ) {
            // showing user message:
            alert( ‘Text length should not be more than 15!’ );

            event.preventDefault(); // stopping form from being submitted
      }
}, false);
```

Limiting input text length to be within 4 and 10:

```
var form =
document.getElementById('myForm');

form.addEventListener("submit",
function (event) {
      var val =
document.getElementById('myInput').val
ue;
      if( val.length < 4 || val.length
> 10  ) {
            // showing user message:
            alert( ‘Text length should
be between 4 and 10’ );

            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

- Allowing only alphabets in the input:
  We can do this by many ways, but the most efficient way is to do it with regular

  expressions. We will be using .test() method to test our input of its validity.

```
var                form                =
document.getElementById('myForm');

form.addEventListener("submit",
function (event) {
      var val =
document.getElementById('myInput').val
ue;
      if( /^[a-zA-Z]*$/.test(val) ==
false ) {
            // showing user message:
            alert( ‘Text should only
consist of alphabets!’ );

            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

The regular expression `/^[a-zA-Z]*$/` matches the input from start to end because of the ‘^’ and ‘$‘ and only looks for zero or more occurrences of alphabets, the range ‘a-zA-Z’ used, includes all the alphabets of the English language in both upper and lower cases. This regular expression will also match blank input because it looks for zero or more occurrences of alphabets, if you do not want to match blank

then you can replace the '*' in the expression with '+', which means one or more occurrence.

- Allowing only numbers in the input:
  As mentioned in the previous example the most efficient way of doing this by using the regular expression.

```
var form = document.getElementById('myForm');

form.addEventListener("submit", function (event) {
    var val = document.getElementById('myInput').value;
    if( /^[0-9]*$/.test(val) == false ) {
        // showing user message:
        alert( 'Text should only consist of numbers!' );

        event.preventDefault(); // stopping form from being submitted
    }
}, false);
```

- Not allowing some characters in the input:
  We will do this by regular expression too, but in the ‘if’ clause we will not check for falsity as we have been doing in the last two examples, i.e., we will not put that ‘== false’ part in the ‘if’ clause, as now we will write a regular expression to find match for any character that is not allowed, and if any character is found result will be true which will lead to stopping of submission process.

  Not allowing the characters ‘x’, ‘y’ and ‘z’:

```
var form = document.getElementById('myForm');

form.addEventListener("submit", function (event) {
    var val = document.getElementById('myInput').value;
    if( /[xyz]+/.test(val) ) { // no ‘ == false’ here
        // showing user message:
        alert( ‘Text should not have the characters x, y and z’ );
```

```
            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

Not allowing any upper case alphabets:

```
var                form                =
document.getElementById('myForm');

form.addEventListener("submit",
function (event) {
      var val =
document.getElementById('myInput').val
ue;
      if( /[A-Z]+/.test(val) ) { // no
‘ == false’
            // showing user message:
            alert( ‘Text should not
have any uppercase letter!’ );

            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

Note: In the regular expressions used in this particular example, replacing the '+' with '*' will break our validity check, since '*' checks for ZERO or more occurrences so if the character to be omitted doesn't exist in the input text, then too the output of the .test() method will be true.

- Validation the email input:
  This can be done only by regular expressions, technically it can be done by other ways, but they aren't practical. The regular expression for validating an email is pretty long as there are many things to keep in mind for an email to be valid.

```
// regular expression for a valid email:
var emailRegex = /^(([^<>()\[\]\.,;:\s@\"]+(\.[^<>()\[\]\.,;:\s@\"]+)*)|(\".+\"))@(([^<>()\.,;\s@\"]+\.{0,1})+[^<>()\.,;:\s@\"]{2,})$/;

var form = document.getElementById('myForm');

form.addEventListener("submit", function (event) {
```

```
      var val =
document.getElementById('myInput').val
ue;
      if( emailRegex.test(val) ==
false ) {
            // showing user message:
            alert( 'Invalid email
entered!' );

            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

- Allowing the input text, to begin with, certain characters only:
  We can do this by writing a regular expression that checks whether the input begins with those specific characters or not and then negating that test by comparing it to 'false'.

  Allowing the input text, to begin with, the characters 'x', 'y' or 'z' only:

```
var                form                =
document.getElementById('myForm');
```

```
form.addEventListener("submit",
function (event) {
      var val =
document.getElementById('myInput').val
ue;
      if( /^[xyz]+.*/.test(val) ==
false ) {
            // showing user message:
            alert( ‘Text should begin
with x, y, or z’ );

            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

Allowing the input text to only begin with a number:

```
var                form                =
document.getElementById('myForm');

form.addEventListener("submit",
function (event) {
      var val =
document.getElementById('myInput').val
ue;
```

```
     if( /^[0-9]+.*/.test(val)  ==
false) {
          // showing user message:
          alert( 'Text should begin
with any number' );

               event.preventDefault(); //
     stopping form from being
     submitted
     }
}, false);
```

- Not allowing the input text to begin with certain characters:
  We can do this by removing the '== false' part from the previous example.

  Not allowing the input to begin with 'x', 'y' or 'z':

```
var                form                =
document.getElementById('myForm');

form.addEventListener("submit",
function (event) {
     var val =
document.getElementById('myInput').val
ue;
     if( /^[xyz]+.*/.test(val) ) {
```

```
            // showing user message:
            alert( ‘Text should NOT
begin with x, y, or z’ );

            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

Not allowing the input to begin from any number:

```
var                form                =
document.getElementById('myForm');

form.addEventListener("submit",
function (event) {
      var val =
document.getElementById('myInput').val
ue;
      if( /^[0-9]+.*/.test(val) ) {
            // showing user message:
            alert( ‘Text should NOT
begin with any number’ );

            event.preventDefault(); //
      stopping form from being
      submitted
```

```
        }
    }, false);
```

CheckBox based validation examples

We can check if a CheckBox is checked or not by first retrieving the CheckBox element as an object, using document.getElementById() method and then using its ‘checked’ property to check if the checkbox is checked or not, if it is then this property is set to ‘true’ and if it is not then it is set to ‘false’.

For the examples in this section of the chapter we will be using the following HTML form:

```
<form name='myForm' action='/target.php'
method='post' id='myForm'>
     <input type="checkbox" name='box1'
id="box1">CheckBox 1
     <br> <input type="checkbox"
name='box2' id="box2">CheckBox 2
     <br> <input type="checkbox"
name='box3' id="box3">CheckBox 3
     <br> <input type='submit'
value='Submit'>
</form>
```

Examples:

- Box 1 should be checked:

```
var form =
document.getElementById('myForm'),
      box1 =
document.getElementById('box1'),
      box2 =
document.getElementById('box2'),
      box3 =
document.getElementById('box3');

form.addEventListener("submit",
function (event) {
      if( ! box1.checked ) {
            alert(‘Box 1 should be
checked!’);
            event.preventDefault(); //
      stopping form from being
      submitted
      }
}, false);
```

- At least one of the box is checked:
  For this, we can just add the ‘checked’ property of each box and compare the result.

In the addition process of ‘checked’ property of the boxes, first all the ‘true’s are converted to 1 and all the ‘false’s are converted to 0 (implicit type casting), and then the addition is performed, so if two boxes are checked then result of addition will be 2 and if three are checked then result of addition will be 3.

Checking if at least one box is checked:

```
var form =
document.getElementById('myForm'),
      box1 =
document.getElementById('box1'),
      box2 =
document.getElementById('box2'),
      box3 =
document.getElementById('box3');

form.addEventListener("submit",
function (event) {
      var sum = box1.checked +
box2.checked + box3.checked;
      if( sum < 1 ) {
            alert(‘At least one box
should be checked!’);
            event.preventDefault(); //
      stopping form from being
      submitted
```

```
        }
    }, false);
```

Radio button based validation examples

Radio buttons are almost like CheckBoxes; the difference is that in a group of radio buttons, only one can box be checked at a time whereas in CheckBoxes any number of boxes can be checked. So the validation process can be done in the same way.

The HTML form with radio buttons:

```
<form name='myForm' action='/target.php'
method='post' id='myForm'>
     <input type="radio" name='box'
id="box1">CheckBox 1
     <br> <input type="radio" name='box'
id="box2">CheckBox 2
     <br> <input type="radio" name='box'
id="box3">CheckBox 3
     <br> <input type='submit'
value='Submit'>
</form>
```

Example of validation:

```
var form =
document.getElementById('myForm'),
      box1 =
document.getElementById('box1'),
      box2 =
document.getElementById('box2'),
      box3 =
document.getElementById('box3');

form.addEventListener("submit", function
(event) {
      var sum = box1.checked + box2.checked
+ box3.checked;
      if( sum == 0 ) {
            alert('Check one radio
button!');
            event.preventDefault(); //
      stopping form from being submitted
      }
}, false);
```

Note that for a group of radio buttons where only one can be changed, all radio buttons must have the same 'name' attribute, as you can see in the example. If the name attribute of any box is changed, then that

box will belong to other group and selection of that button wouldn't affect the selection of the group of other radio buttons.

The fact that for the same group of radio button the 'name' property' is same can be used to achieve an even shorter validation process, in which we can retrieve the list of all radio buttons in a group by document.getElementsByNames() method and then loop through the list to get the sum.

Example:

```
var form =
document.getElementById('myForm'),
      boxes =
document.getElementByName('box');

form.addEventListener("submit", function
(event) {
      var sum = 0;
      //looping through each box and getting
sum
      for(var i = 0; i < boxes.length;
i++)sum += boxes[i].checked;
      if( sum == 0 ) {
            alert('Check one radio
button!');
```

```
            event.preventDefault(); //
      stopping form from being submitted
      }
}, false);
```

Select list based validation examples

The select list HTML form:

```
<form name='myForm' action='/target.php'
method='post' id='myForm'>
      <select name="myList" id="myList">
            <option
      value="notAnOption">Select a
      option</option>
            <option value="a">Option
      1</option>
            <option value="b">Option
      2</option>
            <option value="c">Option
      3</option>
      </select>
<br> <input type='submit' value='Submit'>
</form>
```

The validation of select list is rather simple if we retrieve the ‘select’ element as an object using document.getElementById() method and then check

its ‘value’ property, it will give us the value of currently selected option in the list.

Example:

```
var form = document.getElementById('myForm'),

    list = document.getElementById('myList');

form.addEventListener("submit", function (event) {
    if( list.value == "notAnOption" ) {
        alert(‘Select an option from the list!’);
        event.preventDefault(); // stopping form from being submitted
    }
}, false);
```

I hope the validation examples given in this chapter were useful.

CHAPTER 15

DEBUGGING JAVASCRIPT CODE

What is debugging?

Getting errors of all sorts while programming is common, be it syntax, runtime or logical error. In long programs it becomes difficult to diagnose these errors, especially the logical errors are harder to trace. When you run a code with an error you don't see any popups related to errors or any sort of information related to error directly on the screen, so it becomes crucial that you learn how to find these errors. The process of tracking errors in a program is called debugging.

Described below are the methods for debugging.

Using browser's in-built debugging application

Almost all the modern browsers are equipped with tools for debugging JavaScript code. Almost all

JavaScript debug tools consist of a part called console, all the errors thrown are showed in this screen, given the debug console was open at the time the JavaScript code was running.

Listed below are the major browsers and the way by which you can access the debugging tools:

- Google Chrome:
  By pressing the `F12` key or by pressing the key combination `Ctrl + Shift + J`

- Firefox:
  In Firefox the tool for debugging JavaScript code is not natively built, you need to install an extension known as Firebug for it. After installing this extension, you can press the key `F12` to access the debugging tool.

- Internet Explorer:
  First press the F12 key and then click on the Console option.

- Safari:
  Go to Preferences menu then head to Advanced option and check the “Enable Show Develop menu in menu bar” in it. Now a new option named “Develop” will appear in the menu

bar, click on it then click on the “Show Error Console” option.

- Opera:
  Go to ‘Tools’ menu then click on ‘Advanced’ option and then click on the option ‘Developer Tools.'

Using the console.log() function

You may remember seeing the function ‘console.log’ in the code examples in prior chapters, the ‘console’ in the function name refers to this debug console mentioned in the previous segment of this chapter and the information passed to this function via arguments is printed on that debug console screen.

Example:

```
var x = 21 * 2;
console.log( x ); // prints ‘42’ on the
debug console screen
```

Wise use of this function accelerates the debugging process; you can log useful information in the debug console to see if there is any variable has an unexpected value.

Watch variables

Watch variables are the variables whose values are monitored and reported on change throughout the runtime of a program. This functionality is a part of debug tools of a browser, and not all browsers have this functionality, but many majors browsers have.

Breakpoints

Breakpoints are the points where the execution of a JavaScript program pauses, and when it is paused, you can analyze values of various variables in the program at that instance. You can then easily resume the program execution usually through a play or resume button. Different browsers have different ways of inserting a breakpoint, and not all browsers support breakpoints natively. Generally, option of adding breakpoints is available in debug tools of the browser.

The ‘debugger’ keyword

They keyword debugger pauses the execution of the program and calls the debugging functionally of the browser if it's available if it is not then this keyword has no effect. It is like adding a breakpoint to the code at the point where this keyword is being used.

Example:

```
var x = 21 * 2;

debugger;
// pauses the execution of program if
debugging functionality  is available in
the browser

console.log( x );
// doesn’t output anything unless resume
execution button is pressed, given that the
execution paused in the first place
```

The console.trace() function

This function helps us to understand the execution flow of the program, it prints the stack trace, i.e., the list of functions called to reach the point where this function is being called from, on the debug console screen. The list of functions is generally printed with the line numbers of the function

Example:

```
function b() {
     console.trace();
}
```

```
function a() {
      b();
}
a();
```

The output of the above program will be:

```
console.trace();
      b();
      a();
      <anonymous>;
```

The order is in reverse as it is a stack, i.e., information is inserted in the list from the top, so the first function being called is at the bottom of the list.

The alert() function

This function is similar to console.log for debugging, but instead of printing information on the debug console, passed to this function as its argument, this function shows a popup containing that information.

Example:

```
var x = 21 * 2;
alert( x ); // shows a popup containing the string ‘42’
```

Conclusion

With the knowledge of debugging your JavaScript program, your productivity will be faster since you can now trace and fix those errors and move on to doing other useful things. It is very probable that there will be an error(s) in a long program. The best of the programmers debug their code more often than you think they do.