F27WD - LECTURE NOTES 4.1

JavaScript Language fundamentals

When you create HTML documents, you'll almost always add behaviour to make the document more interactive. You might want to enable some HTML elements to perform actions. Maybe you want to add a fast and fully-featured auto-completion menu to your search box displaying results "as you type". You can use JavaScript to accomplish these tasks because JavaScript is the programming language of the web.

Understanding JavaScript (JS)

- Using expressions to produce data
- Using statements
- Working with functions
- Browser's built-in alert, prompt, and confirm functions
- Using variables
- Conditional programming
- Implementing code loops

JavaScript is a programming language that mainly is used in the design of the user interface of web applications. JavaScript is normally executed (interpreted) by a web browser. JavaScript is not related to Java, although it is a curly-brace language like Java.

JS is *dynamically typed*, which means that when you create a variable, you don't need to specify

its type. JS variables are used to store data values and they can hold a value of any data type. For example:

```
var msg = 'Hello world';    // msg is a string variable and its value is Hello world
var num = 10.3  // num is a numeric variable and its value is 10.3
```

var is one of the many JS keywords. You will use it to declare and initialise variables. msg and num are names of variables or identifiers.

Identifiers are case sensitive and should always start with a letter, an underscore (_), or a dollar sign ($). Hyphens are not allowed; they are reserved for subtractions only. See the “Using Statements” section for more information on variables.

JS Terminology

When you want to create a program, you typically find that the program must access and manipulate data because data is at the root of all systems. You collect data, manipulate data, store data, retrieve data, display data, and so on.

Most data can be broken into smaller pieces called values. JavaScript defines a value type as an **object**, a **primitive** value, or a **function**.

“An *object* is a collection of properties, and a property is an association between a name (key) and a value. A property's value can be a *function*, in which case the property is known as a *method*.”

Examples of objects could be a person, an animal, an HTML document, etc.; anything that can have properties with values.

Primitive value is a data that is represented at its lowest level of the language implementation and, in JavaScript, is one of the following types: undefined, *null*, Boolean, number, or string.

A *function* is a callable object, and a function that is a member of an object is called a *method*.

JavaScript also defines the following built-in objects: the *global* object, the *Object* object, the *Function* object, the *Array* object, the *String* object, the *Boolean* object, the *Number* object, the *Math* object, the *Date* object, the *RegExp* object, the *JSON* object, and several variations of *Error* objects.

Using expressions to produce data

An *expression* is a piece of code that produces a value. An expression can be a simple value, or it can contain operands and operators. Mathematical symbols, such as the plus, minus, divide,

and multiply signs, are examples of operators. Numbers are operands. The operators cause an operation to be executed by using the operands on each side of the operator. Note that an operand can be a value, or it can be another expression.

ARITHMETIC

We need numbers to perform arithmetic operations. Many arithmetic operations, such as addition, subtraction, multiplication, and division, perform an operation on two numeric values to produce a resultant numeric value. In JavaScript, you might write an expression to do something like this:

```
7 + 3 * 8
```

This is an expression with operands and operators. The plus sign (+) and multiplication sign (*) are operators. The numbers are operands. The operators cause an operation to be executed, using the operands on each side of the operator.

OPERATOR PRECEDENCE

JavaScript executed operator in a specific way. Thus, in the previous expression, the multiplication sign has a higher precedence than the addition sign, so 3 is first multiplied by 8 to give 24, and then 7 is added to 24 to get 31. The addition sign (+) and subtraction sign (–) have the same precedence. The multiplication sign (*) and division sign (/) have the same precedence, which is higher than the addition and subtraction signs. If you are working with an expression in which multiple operators have the same precedence, you just apply the operators from left to right.

In the previous example, what would you do if you actually wanted to add 7 and 3 first? You can use parentheses to indicate the order of precedence. Parentheses have the highest precedence, so the expression within the parentheses will be executed first. Here is the modified expression:

```
(7 + 3) * 8
```

MODULO

The modulo (%) operator performs an implied division and returns the remainder. For example, 25 % 7 produces a result of 4, this is the remainder after dividing 25 by 7.

Understanding the string type

A string is a collection of characters and is used to represent text. You can create a string by enclosing in single or double quotes. It doesn't matter whether you use single or double quotes as long as the starting and ending delimiters match, as in the following examples:

```
"Every good boy deserves fudge"
'The quick fox jumps over the lazy dog'
'The doctor said "Today is your lucky day!" '
```

```
"I'm going to be happy when I get my degree!"
```

The third example demonstrates the use of single quotes when you need to embed double quotes within the string. The fourth example demonstrates the use of double quotes when you need to embed a single quote within the string. If you need to embed double or single quotes into the string, you can use the backslash (\) character to escape the single or double quote, as shown in below example:

```
'The doctor said "I\'m pleased to announce that it\'s a girl!"
' "The doctor said \"I'm pleased to announce that it's a girl!\" "
```

Some other common escape sequences are \t to embed a tab and \n to embed a new line.

You will use the plus sign to concatenate strings. The following is an example in which several strings are concatenated to produce one large string:

```
'Hickory Dickory Dock.' + "The mouse ran up the clock." + 'The clock struck one...'
```

You can also concatenate numeric values and strings:

```
'7' + 7 + 1
```

This expression will produce 771 and not 15 because the first 7 is a string and when you concatenate (add) a string with a number, both of them are concatenated as strings and produce a string ('77').

Understanding the Boolean type

The Boolean type can contain the values *true* and *false*. Although you can use these values directly, it's more common to produce a true or false value by comparing two values, as shown in the following examples that use the less-than (<) and greater-than (>) signs:

```
10 < 9
20 > 3
```

The first example produces a false value, whereas the second one produces a true value. The next examples use the less-than-or-equal-to (<=) sign and the greater-than-or- equal-to (>=) sign:

```
5 <= 4
7 >= 8
```

The first expression evaluates to false, and the second expression evaluates to false. You can also use the equals (==) and not equals (!=) operators, as shown in the following example:

```
'Apples'!='Oranges'
10 == 13 - 3
```

Both of these expressions evaluate to true.

LOGICAL OPERATORS

JavaScript provides three logical operators: *and* (&&), *or* (||), and *not* (!). These operators can be used in expressions to produce a Boolean value. The *and* operator will produce a true value if both operands evaluate to true; else it produces a false value. The *or* operator will produce a true value if either operand evaluates to true; else it produces a false value. The *not* operator is a unary operator that will invert the operand, so if the operand evaluates to true, the result is false, and vice versa. Consider the following examples:

```
'Apples' == 'Oranges' && 5 > 3
5 > 10 || 4 < 2
3 < 10 && 10 > 8
7 > 5 || 1 > 2
!(7 > 5 || 1 > 2)
```

The first example uses the *and* operator and produces a false result because the first operand evaluates to false. The second example uses the *or* operator and produces a false result because neither operand evaluates to true. The third example uses the *and* operator and produces a true result because both operands evaluate to true. The fourth example uses the *or* operator and produces a true result because the first operand (7 > 5) evaluates to true. The fifth example uses the *or* operator and the *not* operator. Inside the parentheses, the expression evaluates to true, but the not operator inverts the true to produce a false result.

Using statements

In JavaScript, a statement is a command that is terminated with a semicolon. A statement is different from an expression because an expression just produces a value, and then JavaScript discards the value. A statement tells the software that is executing JavaScript what to do. The software could be a web browser. A statement can be a command to store the result of an expression so it can be reused in other statements.

Using variables

One way to store the results of an expression is to assign the results to a variable. A *variable* is a named reference to a location in memory for storing data. To create a variable, use the JavaScript keyword var, as in the following example:

```
var totalCost = 3 * 21.15;
```

In this example, a variable named *totalCost* is created, and the result of 3 * 21.15 is assigned to the *totalCost* variable. After this variable is created, it can be an operand in other expressions, such as the following:

```
var totalCost = 3 * 21.15;

var tax = totalCost * .05;
```

The value you assign to a variable is not permanent; it is called a variable because you can change it. The following examples modify the *totalCost* variable:

```
var totalCost = 3 * 21.15;
totalCost = totalCost * .1;
totalCost *= .1;
```

The first example initialises *totalCost*. The second example reads the value of *totalCost*, multiplies the value by .1, and stores the result back into *totalCost*. This overwrites the old value with the new value. The third example is a shortcut of the second example. It uses the *= syntax to indicate that you want to multiply the existing value by .1 and store the result in the same variable.

Rules for naming variables

You must adhere to the following rules when naming JavaScript variables.

- A variable name can contain numbers, but they cannot begin with a number.
- Variable names must not contain any type of operators. Illegal examples are *a-b*, *&>x*, *cost*5*.
- Variable names must not contain any punctuation marks of any kind other than the underscore (_) and dollar sign ($). Legal examples are *vehicle_identification*, *first_name*, *last_name*, *$cost*, *total$*. Illegal examples are *thisDoesn'tWork*, *begin;end*, *Many#s*.
- Variable names must not contain any spaces.
- Variable names must not be JavaScript keywords, but they can contain keywords. Illegal examples are function, char, class, for, var. Legal examples are *theFunction*, *forLoop*, *myVar*.
- Variable names are case-sensitive. Examples of different-case variables are *MyData*, *myData*, *mydata*, *MYDATA*. All of them are different variables.

Naming variables

When you create a variable, give the variable a name that is descriptive enough that you don't need a comment to describe what it is. It is not good if you need to add a comment to describe the variable usage. If you name the variable in a way that does not require a comment, the meaningful name will be readable throughout your code. Here are some good and bad examples of variable naming:

```
//bad examples
var last;     //last accessed date
var current; //current vehicle
var changed; //the vehicle make was changed

//good examples
var lastAccessedDate;
var currentVehicle;
var vehicleMakeWasChanged;
```

Notice the casing that is used in the good examples. The recommended naming way is to use "camel casing", which means you start a variable name in lowercase and then capitalise the first letter of each subsequent word that makes up the variable name. Although a variable name can contain the dollar sign and the underscore, it is better not to use them.

Working with functions

A *function* is a grouping of statements that are executed when you call the function. Functions promote code reuse because you can call the function many times from within your code. Functions can have parameters, which enable you to pass data into the function. Functions can also have a return value, so you can return the results of the function to the caller.

Using function declarations

A function can be declared by using the function keyword and then providing a name (also known as an *identifier*), the optional list of parameters enclosed in parentheses, and a set of curly braces with the grouping of statements, as follows:

```
function Add(x, y) {
   return x + y;
}
```

This is an example of a *function declaration*, in which the function is called Add and has two parameters, *x* and *y*. The function has a grouping of statements, denoted by the curly braces (also known as a code block). This function has only one statement, but it could have many statements.

When you *call* the function from your code, you are *invoking* or *applying* the function. An example of calling, invoking, or applying the Add function is as follows:

```
var a = 5; var b = 10;
var c = Add(a, b);
```

In this example, three variables are declared. Variables *a* and *b* are initialised with data to be passed as *arguments* to the Add function. Variable *c* will contain the return value of the Add function. The Add function will receive the arguments into its *x* and *y* parameters. Finally, the return statement will add *x* and *y* and return the result, which is assigned to variable *c*.

DISTINGUISHING "ARGUMENT" AND "PARAMETER"

Many people use the terms "argument" and "parameter" synonymously, but these terms are different. Arguments represent the values you pass to the function (variables *a* and *b* in the previous example), whereas the parameters represent the values received from the caller (variables *x* and *y* in the previous example).

Function declarations may be called before the function declaration is declared because the function declarations are resolved when the JavaScript is parsed. The following example will run properly even though the call to the Add function is before the Add function:

```
var a = 5; var b = 10;
var c = Add(a, b);

function Add(x, y) {
        return x + y;
}
```

However, “function expressions” are different; they cannot be used before they have been declared.

Using function expressions

These functions produce a value of type function. You can assign function expressions to variables or execute them directly. Example of a function expression being created and assigned to a variable:

```
var addFunction = function(x,
y){
  return x + y;
};
var c = addFunction(5, 10);
```

First, notice that *addFunction* is called after the function expression is assigned to the *addFunction* variable. If you tried to call *addFunction* before the assignment of the function expression to the *addFunction* variable, an exception would be thrown.

The *addFunction* variable is of type *function*, in which the function expression is created by using the *function* keyword to create a function with no name (also known as an *anonymous function*), and then the function expression is assigned to the variable. An anonymous function has no name or identifier. Although function expressions can be named or anonymous, it’s considered better to leave the function anonymous to minimise confusion.

Deciding which arguments

JavaScript is very loose when passing arguments to functions. If you have too many arguments, JavaScript just discards the extras. If you don’t have enough arguments, the parameter values for missing arguments will be undefined.

The advantage of this loose behaviour is that you can add parameters to a method that has already been created and is already being called. The existing callers will continue to work. The drawback of this loose behaviour is that you might inadvertently pass an incorrect quantity of arguments to the function, and you get no indication of a problem.

Using the browser's built-in alert, prompt, and confirm functions

When writing web applications, the browser provides the following functions that can present data to and collect data from the user:

- **alert** Used to display a message to the user in a modal window. The user clicks the OK button to close the message window. The following code produces the alert window shown in Figure 4.1-1:

```
alert('Here is an alert');
```

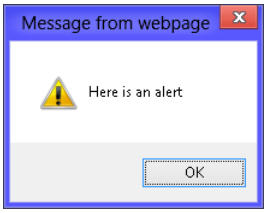

FIGURE 4.1-1 The alert window showing a message and an OK button

- **prompt** Used to query the user for input by displaying a modal message prompt and a text box for the user to enter data into. The text box can be supplied a default value. The user can close the window by pressing Enter or clicking the OK or Cancel buttons. If the user didn't select "Cancel", the prompt function returns the data that the user typed in the text box. The following code produces the prompt in Figure 4.1-2:

```
var promptResult = prompt('This is a prompt for information', 'default value');
```

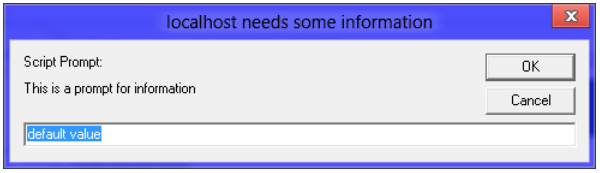

FIGURE 4.1-2 The prompt window collecting text

- **confirm** Used to query the user for OK or Cancel by displaying a modal message window. The confirm function returns either true (when the OK button is clicked) or false (when the Cancel button is clicked):

```
var confirmResult = confirm('Do you confirm?');
```

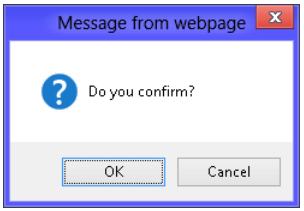

FIGURE 4.1-3 The confirm window presents a message and OK and Cancel buttons

These built-in functions, and any functions you write, can be overwritten very easily because the function name is a variable. Consider the following scenario, in which a different function is assigned to the prompt function:

```
prompt = function(){
    return 'hello
    again';
};
```

This code replaces the behaviour of the **prompt** built-in function with a function that always returns the string, 'hello again'. The function name is represented by a variable, and you can change its value dynamically. This is the same for function declarations and function expressions.

Using variables

Scoping variables

Scoping is the context within a computer program in which a variable name is valid and can be used to access the variable. In JavaScript, there are essentially two scopes, global and local. A variable with a global scope is accessible from anywhere in the program. A variable with a local scope is accessible only from within the function in which the variable is defined, so you can think of local scope as being function scope. Variables that are declared anywhere inside the function will have a local function scope. To avoid confusion, you should declare all function variables at the top of the function.

BE CAREFUL NOT TO CREATE GLOBAL VARIABLES IMPLICITLY

If you do not use the **var** keyword when you declare a variable, the variable is automatically created, but it will have a global scope. Consider the following code example:

```
totalCost = 3 * 21.15;
tax = totalCost * 1.05;
```

If this code were inside a function, you might have thought that *totalCost* and *tax* would be automatically created with a local scope, but that's not the case. As it turns out, *totalCost* and *tax* will always be created with a global scope, and these variables will be accessible from anywhere in the program. You should always declare variables by using the var keyword.

Converting to a different type

In many scenarios, you will want to convert a variable from one type to another. For example, the prompt function always returns a string, but you have prompted the user to enter a number. For conversions, you can use the **Number** and **String** functions.

Using the Number function

The Number function attempts to convert the object argument that is passed into the function to a number. If the object argument cannot be converted to a number, a value of *NaN* (not a number) is returned. (Performing any mathematical operation with a non-numeric value will produce a result of NaN.)

Consider the following example, in which the user is prompted to enter his or her age, the age is incremented, and a message is displayed, using the new age:

```
var age = prompt('Enter age', '');
alert('You will soon be ' + age + 1 + ' years old!');
```

When you run this code and input a number such as 20, the message displays as You Will Soon Be 201 Years Old! because age is a string, so the plus sign automatically converts the number 1 to a string and concatenates to the age, resulting in an age of 201.

By using the Number function, you can convert the age to a number and then add 1, as shown in the following example:

```
var age = prompt('Enter age', '');
alert('You will soon be ' + Number(age) + 1 + ' years old!');
```

If you try to run this code, you get the same age response of 201 years old. What happened? The first plus sign has a string on the left side and the Number function on the right. Although you converted to a number, the first plus sign automatically converted the number back to a string and returned the same value of 201.

To solve the problem, put parentheses around the math to ensure that age receives the proper value before being converted to a string and concatenated as follows:

```
var age = prompt('Enter age', '');
alert('You will soon be ' + (Number(age) + 1) + ' years old!');
```

When you run this code, you get a response of You Will Soon Be 21 Years Old! because the age is converted to a number and incremented before any of the string concatenation occurs.

Using the String function

The String function attempts to convert the object argument that is passed into the function into a string. If you have two numbers such as 20 and 19, and you want to concatenate them like strings and display the resulting string of 2019; you might try the following code:

```
var x = 20;
var y = 19;
alert(x + y);
```

This example will display a value of 39 because the plus sign added the values instead of concatenating them as strings. To solve the problem, you can use the String function to convert the numbers to strings as follows:

```
var x = 20;
var y = 19;
alert(String(x) + String(y));
```

This example will display a value of 2019 because the numbers were converted to strings, and then the plus sign concatenated the values.

Conditional programming

You will often need to execute certain code when an expression evaluates as true and execute different code when the expression evaluates to false. This is when the *if* and *switch* keywords can help.

Using the if/else keywords

Consider the scenario in which the user is prompted to enter his or her age, one is added to the age, and a message is displayed. Here is the code that was used earlier when the built-in prompt function was discussed:

```
var age = prompt('Enter age', '');
alert('You will soon be ' + (Number(age) + 1) + ' years old!');
```

What happens if someone enters I Don't Know for the age? This string is not numeric, and the displayed message will be You Will Soon Be *NaN* Years Old. You might want to provide a more specific message when the user input is not a number. This can be accomplished by using the *if/else* keywords with a built-in function called **isNaN**, as follows:

```
var age = prompt('Enter age', '');

if(isNaN(age))
   alert('You need to enter a valid number');
   else
alert('You will soon be ' + (Number(age) + 1) + ' years old!');
```

The isNaN function returns true if the object ('age') is not numeric and false if it is numeric.

The **if** keyword is used with an expression that is surrounded by parentheses and evaluates to a Boolean value. This expression is used in your code to steer the program flow based on the result of the isNaN function, which, when true, displays the message, You Need To Enter A Valid Number. Notice that the **if** keyword has a corresponding **else** keyword that is used to provide an alternate program flow when the if expression evaluates to false. The **else** keyword is optional.

The previous example executes a single statement when true and a single statement when false. If you need to execute multiple statements when true or false, you must surround the statements with curly braces to indicate that you have a code block to execute as follows:

```
var age = prompt('Enter age', '');
if(isNaN(age)) {
        age = 0;
        alert('You need to enter a valid number');
}
else {
        age = Number(age) + 1;
        alert('You will soon be ' + age + ' years old!');
}
```

As a rule, you should consider using curly braces all the time. This enables a user to add code into the code block without having to think about whether the curly braces exist.

You can also create chained (also known as cascading) **if** statements by adding another **if** after the **else** keyword. Here is an example of a cascading **if**:

```
var age = prompt('Enter age', '');
if(isNaN(age)){
        age = 0;
        alert('You need to enter a valid number');
}
else if(Number(age) >= 50) {
        age = Number(age) + 1;
        alert('You're old! You will soon be ' + age + ' years old!');
}
else if(Number(age) <= 20) {
        age = Number(age) + 1;
        alert('You're a baby! You will soon be ' + age + ' years old!');
}
else
{
        alert('You will soon be ' + (Number(age) + 1) + ' years old!');
}
```

In this example, the first conditional test checks to see whether the age is not a number. The **else** condition is checking to see whether the age is greater than or equal to 50. The next **else** condition is checking to see whether the age is less than or equal to 20. Finally, the last **else** condition is just displaying a default message.

Using the switch keyword

The **switch** keyword can be used when a single value is to be examined and, based on its value, there could be several outcomes. Consider the following example that uses the **switch** keyword, in which the user is prompted to select a colour for the car that he or she is purchasing:

```
var carColor = prompt('What color car would you like to buy?',
'white'); switch (carColor) {
   case 'red':
      alert('Red is a fancy
```

```
        choice!'); break;
     case 'black':
        alert('Black looks nice, but you need to wash
        often!'); break;
     case 'white':
        alert('White is in stock and you get a discount!');
        break;
     default:
       alert('The color:' + carColor + ' is not
  known.'); break;
  };
```

In this example, the user is prompted to enter a car colour and presented with a default value of white. The **switch** keyword examines *carColor* to see whether its value matches any of the values provided in the cases. If there is a match, the code within the case is executed until the code reaches either a break or a return keyword. If there is no break or return, the code continues into the next case or default.

How do you use the **switch** with numeric ranges? Consider previous cascading **if** example. Four different categories of responses—if the age was not a number, if it was greater than or equal to 50, if it was less than or equal to 20, or if it was any other number—received four different messages. The following is a rewrite of the cascading **if** as a **switch**:

```
  var age = prompt('Enter your age',
  ''); age = Number(age);
  switch (true)
     { case
     isNaN(age):
        age = 0;
        alert('You need to enter a valid
        number'); break;
     case (age >= 50):
        age = Number(age) + 1;
        alert("You're old! You will soon be " + age + " years old!");
        break;
     case (age <= 20):
        age = Number(age) + 1;
     alert("You're a baby! You will soon be " + age + " years
     old!"); break;
  default:
     alert('You will soon be ' + (Number(age) + 1) + ' years
     old!'); break;
};
```

In this example, the trick is to use *switch(true)*, which enables you to use conditional statements with each case that evaluates as true or false.

Determining whether a variable has a value

You will often want to determine whether a variable has a value. You can use the if keyword to

determine this. The if keyword evaluates as true if the variable has a value; it evaluates as false when the variable is either *undefined* or *null*. The difference between undefined and *null* is minimal, but there is a difference. A variable whose value is undefined has never been initialised. A variable whose value is null was explicitly given a value of null, which means that the variable was explicitly set to have no value. If you compare undefined and null by using the null==undefined expression, they will be equal.

Consider the following example, in which you want to determine whether *myVar* has a value:

```
if(myVar){
   alert('myVar has a value');
}
else {
   alert('myVar does not have a value');
}
```

If *myVar* is 0, *NaN*, empty string, null, or undefined, a message is displayed, stating that *myVar* does not have a value. If *myVar* contains any other value, a message is displayed stating that *myVar* has a value.

No value coalescing operators

Often, you want to check a variable to see whether it has a value and, if the variable has a value, use either the variable or a default value. JavaScript offers a rather simple way to accomplish this: use the || (**or**) operator.

The following is an example of using the **or** operator to accomplish this task:

```
var customer = prompt('Please enter your name');
alert('Hello ' + (customer || 'Valued Customer'));
```

In this example, if customer has a value, the **or** operator will be evaluated as true, and the actual value is returned. Because the **or** operator is short-circuiting, there is no evaluation of the second operand. If customer has no value, the **or** operator returns the second operand, even if it's null.

It's interesting to note that or operators can be chained, as shown in this example, in which the user is prompted for a name and then prompted for a company name:

```
var customer = prompt('Please enter your name');
var companyName = prompt('Please enter your company name');
alert('Hello ' + (customer || companyName || 'Valued
Customer'));
```

In this example, the alert message will contain the value of the first variable that has a value or Valued Customer if none of the variables has a value. The && (**and**) operator exhibits similar behaviour but returns the first empty value instead of the first non-empty value. If all variables have a value, Valued Customer is returned. There isn't much real-world value to this behaviour, but it is a tool in your JavaScript toolbox.

Determining whether two values have the same type and are equal

When JavaScript evaluates the following expressions, they all evaluate as true:

```
null == undefined false == 0;
'' == 0;
'123' == 123
```

JavaScript attempts to convert the operands to types that are compatible before per- forming the equality check. However, you might want these expressions to evaluate as false because the types are different.

To perform a type and equality comparison, JavaScript provides the === and the !=== operators. The following expressions evaluate as false:

```
null === undefined false === 0;
'' === 0;
'123' === 123
```

Implementing code loops

Code loops are an important part of every programming language because many programs need to repeat a sequence a given number of times or repeat a sequence until some value has changed. JavaScript gives us the *while*, *do,* and *for* keywords to perform looping operations.

Implementing the while loop

The *while* keyword can be used to create a loop that accepts a loop expression enclosed within parentheses. The loop can contain a single statement or a code block that executes as long as the loop expression evaluates as true. Here is an example of the while loop:

```
var x = 10;
while(x > 0) {
        x--;
        alert("The value of x is " + x);
}
```

In this example, as long as the loop expression evaluates as true, the loop continues. Each time through the loop, *x* is decremented, using the x-- statement. After *x* is decremented, an alert message is displayed that shows the current value of *x*. Note that the code block can execute zero to many times, based on the value of *x* when the program pointer reaches the while loop. If *x* were initialised to zero, the loop would not execute.

Implementing the do loop

The *do* keyword can be used to create a loop that executes one to many times. The do statement starts with the word "*do*," followed by a mandatory set of curly braces containing a code block that will be executed each time the loop executes, followed by the while keyword and

a loop expression that is enclosed in parentheses.

The most compelling reason to use the do loop is that it executes at least once because the loop expression is evaluated after the loop executes. It can be difficult to think of a real-world implementation of this loop, but consider when a login screen needs to be displayed to collect the user name and password, and the login screen will be redisplayed if the login credentials are not correct. The following example should provide some clarity to this implementation:

```
var retries = 0;
do{
        retries++;
        showLoginScreen();
} while(!authenticated() && retries < 3);
if(retries==3){
        alert('Too many tries');
        return;
}
```

In this example, a *retries* variable is first created and initialised to zero. Next, the do loop executes. Inside the loop's code block, the *retries* variable is incremented, and a call is made to a *showLoginScreen* function, which will display a login screen that prompts for a user name and password. After the user enters the appropriate information and closes the login screen, the loop expression is evaluated. The authenticated function checks the user name and pass- word and returns true if the user should be authenticated. The loop will continue as long as the user is not authenticated and the *retries* count is less than three.

Implementing the for loop

The *for* loop is typically used when you know how many times the loop will execute, and you want a *loop counter* variable. The *for* loop uses the following syntax:

```
for (var variable=startvalue; variable < endvalue; variable = variable + increment)
{
        code to be executed
}
```

Within the parentheses are three sections, separated by semicolons. The two semicolons must exist, even if you leave a section empty. The first section enables you to declare and initialise a loop variable. This section executes once when the program reaches this loop. The second section is the loop expression, which is called prior to executing the loop code, to determine whether the loop should execute. If the loop expression evaluates as true, the loop code executes. The third section is the loop modification section. This is when you might want to increment (or decrement) the loop variable. This section is executed after the loop code executes for each loop.

You might use the for loop when you know that you want to loop a specific number of times and, in the loop code, you want to access a *counter* variable, as shown in the following example:

```
for (var counter = 0; counter < 10; counter++) {
        alert('The counter is now set to ' + counter);
}
```

In this example, a *counter* variable is created with the var keyword. Be careful to use the var keyword to avoid creating a global variable by mistake. The loop will continue as long as the counter variable is less than 10. Each time the loop executes, the counter is incremented, using the counter++ syntax. The *counter* variable is used in the loop code to display a mes- sage, but the counter could certainly be used for more elegant tasks.

Breaking out of a loop

As your loop logic becomes complicated, you might find that you need a way to exit the loop by using a conditional check within the loop code. For scenarios such as this, you can use the break keyword to exit the immediate loop. Note that the break keyword will exit only from the current loop. If you are in a nested loop, you will exit only one level.

In the following scenario, a loop is created to determine whether a number is a prime number, and the break keyword is used to exit the loop if the number to test is determined not to be a prime number:

```
var numberToTest = prompt('Type number here.', '');
var index = 2;
var isPrime = true;
while (index < numberToTest) {
        if (numberToTest % index == 0) {
                isPrime = false;
                break;
        }
        index++;
}
if (isPrime) {
        alert(numberToTest + ' is a prime number');
}
else {
        alert(numberToTest + ' is not a prime number because it is divisible by ' +
index);
}
```

In this example, the modulo (%) operator determines whether index can be divided into the number to test without producing a remainder. If so, the number to test is not a prime number. As soon as a number is found to be nonprime, there is no need to continue looping, so the break keyword is used to exit the loop.

Handling errors

When writing code, you always want to make sure that your code does not cause an *exception*. An *exception* is an error that occurs at runtime due to an illegal operation during execution. You should validate your variables before performing an operation that could throw an exception. For example, before you divide one variable (numerator) by another variable (denominator), verify that the denominator will never be zero. However, you cannot always check for every potential error. For example, you are reading from a network stream when the network connection is abruptly lost. For situations like this, you can use the *try (try block)*, *catch (catch block)*, or *finally (finally block)* keywords.

The *try* block is used with a code block that contains the code that might fail. You want to try to execute the code block. The *try* block requires a *catch* block, a *finally* block, or both.

The *catch* block will have the exception passed to it, so you have access to the exception within your code. The *catch* block is automatically executed if the code in the *try* block throws an exception. In that case, program execution immediately jumps to the *catch* block without executing further statements in the *try* block.

The *finally* block is executed after the *try* block successfully completes or the *catch* block completes. The intent of the *finally* block is to provide a place for cleanup code because the *finally* block executes regardless of whether an exception was thrown.

The following code example illustrates the use of *try*, *catch*, and *finally* blocks:

```
try{
   undefinedFunction()
   alert('Made it, so undefinedFunction exists')
}
catch(ex){
   alert('The following error occurred: ' + ex.message)
}
finally{
   alert('Finally block executed')
}
```

In this example, if the *undefinedFunction* function exists and doesn't throw an exception, you get two alerts the first alert is Made It, So undefinedFunction Exists, and the second alert is Finally Block Executed.

If the *undefinedFunction* function does not exist, an exception is thrown, and you receive two alerts: the first alert is The Following Error Occurred, and the second alert is Finally Block Executed. An exception was thrown because *undefinedFunction* didn't exist, and the program immediately jumped to the *catch* block without executing the rest of the *try* block.

Remember that the finally block always executes either after the *try* block completes successfully or after the *catch* block executes. If the *catch* block throws an exception, the *finally* block executes before the exception is passed to the calling routine.

Check that you understand

- JS is *dynamically typed*, so when you create a variable, you don't need to specify its type. The type checking is performed when the program is run. JavaScript has six primitive types: *string*, *number*, *undefined*, *null*, *Boolean*, and *symbol*. It has also a compound type: *object*
- Scoping is the context within a computer program in which a variable name is valid and can be used to access the variable. In JS, there are essentially two scopes, global and local.
- Use two equal signs (==) to test for equality, and use three equal signs (===) to test for same type and equality.
- The *while* loop executes zero to many times, and the *do* loop executes one to many times. The *for* loop executes zero to many times and has a counter variable. Use the *break* keyword to exit from the current loop.
- Use the try, catch, and finally keywords to handle exceptions.