

Chapter 15. JavaScript 4: Objects and Arrays

Table of Contents

Objectives	2
15.1 Introduction	2
15.2 Arrays	2
15.2.1 Introduction to arrays	2
15.2.2 Indexing of array elements	3
15.2.3 Creating arrays and assigning values to their elements	3
15.2.4 Creating and initialising arrays in a single statement	4
15.2.5 Displaying the contents of arrays	4
15.2.6 Array length	4
15.2.7 Types of values in array elements	6
15.2.8 Strings are NOT arrays	7
15.2.9 Variables — primitive types and reference types	8
15.2.10 Copying primitive type variables	9
15.2.11 Copying reference type variables	9
15.2.12 Array activities	10
15.2.13 Arrays concept map	12
15.3 The JavaScript object model	13
15.3.1 Introduction to the JavaScript object model	13
15.3.2 Objects in JavaScript	13
15.3.3 Naming conventions	14
15.3.4 Creating objects and variables referring to objects	15
15.3.5 Object properties and methods	16
15.3.6 Some important JavaScript objects	19
15.3.7 The 'Triangle' object	21
15.3.8 User defined objects	21
15.3.9 Implementing instance variables and methods	23
15.3.10 The JavaScript object search chain	25
15.3.11 Object activities	28
15.4 Arrays as objects	29
15.4.1 Array method: join	29
15.4.2 Array method: sort	30
15.4.3 Array method: reverse	31
15.4.4 Single and multi-dimensional arrays	32
15.5 Objects as associative arrays	33
15.5.1 Enumerating associative arrays with FOR/IN loop statements	33
15.5.2 Using FOR/IN for numerically index arrays	34
15.5.3 Activities and further work	36
15.6 Review Questions	41
15.7 Discussion Topics	42
15.8 Answers	42
15.8.1 Discussions of Exercise 1	43
15.8.2 Discussions of Exercise 2	43
15.8.3 Discussions of Activity 1	44
15.8.4 Discussions of Activity 2	44
15.8.5 Discussions of Exercise 3	45
15.8.6 Discussions of Activity 3	45
15.8.7 Discussions of Activity 4	47
15.8.8 Discussions of Activity 5	47
15.8.9 Discussions of Activity 6	48
15.8.10 Discussions of Activity 7	48

15.8.11	Discussions of Activity 8	49
15.8.12	Answers to Review Questions	51
15.8.13	Contribution to Discussion Topics.....	51

Objectives

At the end of this chapter you will be able to:

- Understand the basic features of JavaScript arrays;
- Understand the fundamental elements of JavaScript arrays;
- Write HTML files using JavaScript arrays;
- Explain the JavaScript object model;
- Use arrays as objects.

15.1 Introduction

Most high level computer programming languages provide ways for groups of related data to be collected together and referred to by a single name. JavaScript offers objects and arrays for doing so. JavaScript arrays are rather different from arrays in many programming languages: all arrays are objects (as in many other languages), but they are also *associative* arrays. Also, all objects can also be used as if they, too, are arrays.

This chapter is organised into four sections. It introduces arrays and objects separately, then considers arrays as objects, then finally considers objects as (associative) arrays. Many important concepts are covered in this unit, although much of the object technology concepts have been introduced in earlier units.

When working with variables, an important distinction has to be made: does the variable contain the value of a primitive type, or does it contain a reference to a (non-primitive) collection of data. A thorough grounding in the concepts covered in this chapter is necessary to both be able to understand the sophisticated JavaScript scripts written to support complex websites, and to be able to begin developing JavaScript solutions yourself for real world problems. It is important to work through examples until you understand them; write your own programmes that use and test your learning. Programming is learnt through doing as much, or more so, than by reading.

15.2 Arrays

15.2.1 Introduction to arrays

Arrays provide a tabular way to organise a collection of related data. For example, if we wished to store the seven names of each weekday as Strings, we could use an array containing seven elements. This array would be structured as follows:

Index	Value
weekDays[0]	"Monday"
weekDays[1]	"Tuesday"
weekDays[2]	"Wednesday"
weekDays[3]	"Thursday"
weekDays[4]	"Friday"
weekDays[5]	"Saturday"
weekDays[6]	"Sunday"

As can be seen all of these different String values are stored under the collective name `weekDays`, and a number (from 0 to 6) is used to state which of these `weekDays` values we specifically wish to refer to. So by referring to `weekDays[3]` we could retrieve the String "Thursday".

DEFINITION - Array

An array is a tabular arrangement of values. Values can be retrieved by referring to the array name together with the numeric index of the part of the table storing the desired value.

As you may have spotted, by having a loop with a numeric variable we can easily perform an action on all, or some sub-sequence, of the values stored in the array.

15.2.2 Indexing of array elements

As can be seen from the above figure, there are seven elements in the `weekDays` array.

DEFINITION — element

Arrays are composed of a numbered sequence of elements. Each element of an array can be thought of as a row (or sometimes column) in a table of values.

The seven elements are indexed (numbered) from zero (0) to six (6). Although it might seem strange to start by numbering the first element at zero, this way of indexing array elements is common to many high-level programming languages (include C, C++ and Java), and has some computational advantages over arrays that start at 1.

Note

The index of an array element is also known as its subscript. The terms array index and array subscript can be used interchangeably. In this unit we consistently use the term *index* for simplicity.

Exercise 1

Answer the following questions about the `weekDays` array:

- What is the first element?
- What is the last element?
- What is the 4th element?
- What is the value of the first element?
- What is the value of the 4th element?
- What is the element containing String "Monday"?
- What is the element containing String "Saturday"?
- What is the index of the element containing String "Monday"?
- What is the index of the element containing String "Saturday"?

15.2.3 Creating arrays and assigning values to their elements

There are a number of different ways to create an array. One piece of JavaScript code that creates such an array is as follows:

```
// VERSION 1
var weekDays = new Array(7); weekDays[0] = "Monday"; weekDays[1] =
"Tuesday"; weekDays[2] = "Wednesday"; weekDays[3] = "Thursday";
weekDays[4] = "Friday"; weekDays[5] = "Saturday"; weekDays[6] =
"Sunday";
```

The first (non-comment) line is:

```
var weekDays = new Array(7);
```

This line declares a new variable called `weekDays` and makes this new variable refer to a new `Array` object that can hold seven elements.

Note

The concept of arrays as objects is discussed later this unit.

The seven statements that follow this line assign the Strings *"Monday"* - *"Sunday"* to the array elements *weekDays[0]* to *weekDays[6]* respectively.

15.2.4 Creating and initialising arrays in a single statement

Another piece of JavaScript that would result in the same array as VERSION 1 above is the following:

```
// VERSION 2 - all in one line
var weekDays = new Array( "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday", "Sunday" );
```

This single statement combines the declaration of a new variable and `Array` object with assigning the seven weekday Strings. Notice that we have not had to specify the size of the array, since JavaScript knows there are seven Strings and so makes the array have a size of seven elements.

The above examples illustrate that arrays can either be created separately (as in VERSION 1), and then have values assigned to elements, or that arrays can be created and provided with initial values all in one statement (VERSION 2).

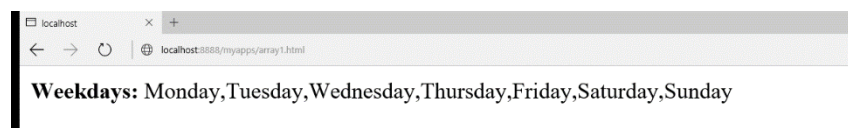
When declaring the array, if you know which values the array should hold you would likely choose to create the array and provide the initial values in one statement. Otherwise the two-stage approach of first creating the array, and then later assigning the values, is appropriate.

15.2.5 Displaying the contents of arrays

The easiest way to display the contents of an array is to simply use the `document.write()` function. This function, when given an array name as an argument, will display each element of the array on the same line, separated by commas. For example, the code:

```
var weekDays = new Array( "Monday", "Tuesday", "Wednesday", "Thursday",
    "Friday", "Saturday", "Sunday" );
document.write( "<b> Weekdays: </b>" + weekDays );
```

Produces the following output in a browser:



15.2.6 Array length

The term *length*, rather than *size*, is used to refer to the number of elements in array. The reason for this will become clear shortly.

As illustrated in the VERSION 1 code above, the size of an array can be specified when an array is declared:

```
var weekDays = new Array(7);
```

This creates an array with seven elements (each containing an undefined value):

Index	Value
weekDays[0]	<undefined>
weekDays[1]	<undefined>
weekDays[2]	<undefined>
weekDays[3]	<undefined>
weekDays[4]	<undefined>
weekDays[5]	<undefined>
weekDays[6]	<undefined>

In fact, while this is good programming practice, it is not a requirement of the JavaScript language. The line written without the array size is just as acceptable to JavaScript:

```
var weekDays = new Array();
```

this creates an array, with no elements:

Index	Value

In this second case, JavaScript will make appropriate changes to its memory organisation later, once it identifies how many elements the array needs to hold. Even then, JavaScript is a can extend the size of an array to contain more elements than it was originally defined to contain.

For example, if we next have the statement:

```
weekDays[4] = "Friday";
```

the JavaScript interpreter will identify the need for the weekDays array to have at least five elements, and for which the 5th element is the String "Friday".

Index	Value
weekDays[0]	<undefined>
weekDays[1] weekDays[2]	<undefined> <undefined>
weekDays[3]	<undefined>
weekDays[4]	"Friday"

If this were then followed by the statement:

```
weekDays[6] = "Sunday";
```

the JavaScript interpreter will identify the need for the weekDays array to now have seven elements, of which the 5th contains the String "Friday" and the 7th contains "Sunday":

Index	Value
weekDays[0]	<undefined>
weekDays[1]	<undefined>
weekDays[2]	<undefined>
weekDays[3]	<undefined>
weekDays[4]	"Friday"

Index	Value
weekDays[5]	<undefined>
weekDays[6]	"Sunday"

Once created, an array has a length property. This stores the number of elements for which JavaScript has made space. Consider the following statements:

```
var weekDays = new Array(7); var months = new Array();
var bits = new Array(17, 8, 99);
```

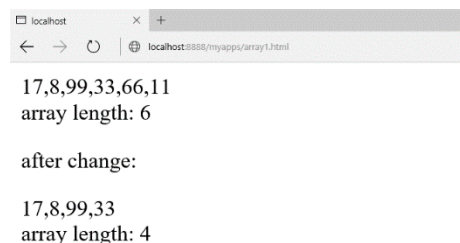
The length of each of these arrays is as follows:

- weekDays.length = 7
- months.length = 0
- bits.length = 3

One needs to be careful, though, since making the length of an array smaller can result in losing some elements irretrievably. Consider this code and the following output:

```
var bits = new Array(17, 8, 99, 33, 66, 11);
document.write(bits);
document.write("<br> array length: " + bits.length);
bits.length = 4;
document.write("<p> after change: </p>");
document.write(bits);
document.write("<br> array length: " + bits.length);
```

The browser output from such code is:



As can be seen, after the statement `bits.length = 4;` the last two array elements have been lost.

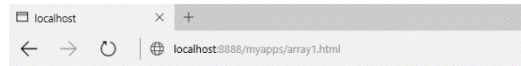
15.2.7 Types of values in array elements

In most high-level programming languages arrays are typed. This means that when an array is created the programmer must specify the type of the value to be stored in the array. With such languages, all elements of an array store values of a single type. However, JavaScript is not a strongly typed programming language, and a feature of JavaScript arrays is that a single array can store values of different types.

Consider the following code:

```
var things = new Array(); things[0] = 21;
things[1] = "hello";
things[2] = true;
document.write("<p>[0]: " + things[0]);
document.write("<p>[1]: " + things[1]);
document.write("<p>[2]: " + things[2]);
```

As can be seen, this is perfectly acceptable JavaScript programming:



[0]: 21

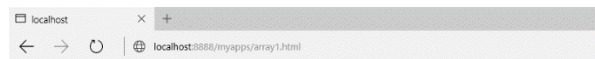
[1]: hello

[2]: true

When the value of an array element is replaced with a different value, there is no requirement for the replacement value to be of the same type. In the example below, array element 1 begins with the String "hello", is then changed to the Boolean value false, and changed again to the number 3.1415:

```
var things = new Array(); things[0] = 21;
things[1] = "hello"; things[2] = true; things[1] = false; things[1]
= 3.1415;
document.write("<p>[0]: " + things[0]);
document.write("<p>[1]: " + things[1]);
document.write("<p>[2]: " + things[2]);
```

As can be seen, this changing of array element values of different types works without problems:



[0]: 21

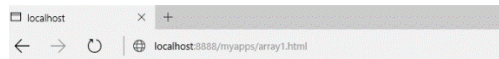
[1]: 3.1415

[2]: true

We can confirm that a single array can store values of different types by displaying the return value of the `typeof` function:

```
var things = new Array(); things[0] = 21;
things[1] = "hello";
things[2] = true;
document.write("<p>type of things[0]: " + typeof things[0] );
document.write("<p>type of things[1]: " + typeof things[1] );
document.write("<p>type of things[2]: " + typeof things[2] );
```

The output of the above code is as follows:



type of things[0]: number

type of things[1]: string

type of things[2]: boolean

15.2.8 Strings are NOT arrays

In many programming languages, text strings are represented as arrays of characters. While this makes sense in non-object oriented languages, there are a number of advantages of representing data such as text as objects (see later this unit).

You will only obtain *undefined* values if you attempt to refer to particular characters of Strings using the square bracket array indexing syntax.

For example, the code

```
var firstName = "Matthew";
document.write("second letter of name is: " + firstName[1]);
```

It is also easy to confuse String and Array objects because they both have a length property. So the code:

```
var firstName = "Matthew";
document.write("second letter of name is: " + firstName[1]);
document.write("<p> length of 'firstName' " + firstName.length);
```

is valid, and we do see the number of characters of the String displayed:

However, the similarity is because both *Strings* and *Arrays* are objects — see later in this unit for a detailed discussion of JavaScript objects.

15.2.9 Variables — primitive types and reference types

When one begins to work with collections of values (e.g. with arrays or objects), one needs to be aware of the variable's value. A variable containing a primitive type value is straightforward: for example consider the numeric variable age in this code:

```
var age = 21;
```

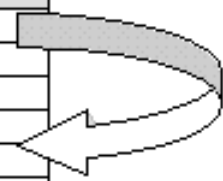
Memory location	value
age	21

However, the situation is not so simple when we consider an array variable. For example the ageList array of three ages defined as follows:

```
var ageList = new Array( 3 ); ageList[0] = 5;
ageList[1] = 3;
ageList[2] = 11;
```

It is not the case that ageList is the name for a single location in memory, since we know that this array is storing three different values. The location in memory named ageList is actually a reference to the place in memory where the first value in the array can be found. The diagram below attempts to illustrate this:

Memory location	value
AgeList	001729
	001727
	001728
[0]	5
[1]	3
[2]	11
	001732
	001733
	001734



Note

There is nothing special about the locations 001727 etc., these numbers have been made up and included to illustrate unnamed locations in memory.

So we can think of the variable ageList as referring to where the array values can be found.

The implication of the difference between variables of primitive types and reference types is rather important,

especially in the case of copying or overwriting the values of reference variables.

15.2.10 Copying primitive type variables

With primitive type variables it is straightforward to copy and change values. Consider the following code:

```
var name1 = "ibrahim"; var name2 = "james";
```

Initially the memory looks as follows:

Memory Location	Value
name1	"Ibrahim"
name2	"James"

Then if we execute the following lines:

```
name1 = name2; name2 = "fred";
```

the values in memory will be:

Memory Location	Value
name1	"James"
name2	"fred"

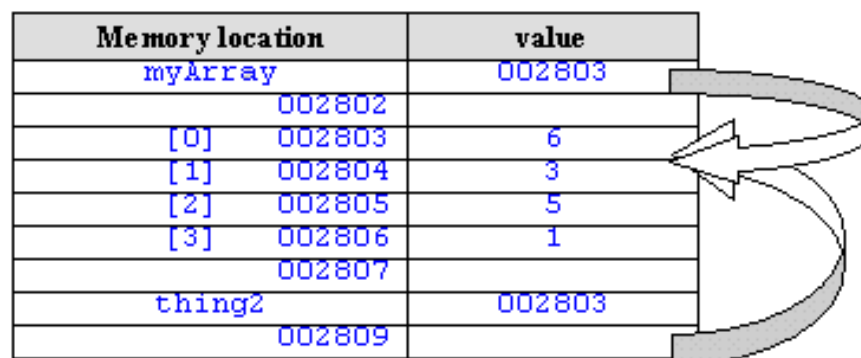
What is happening is that when one variable is assigned the value of a second, a copy of the second variable's value is placed into the location in memory for the first — so a copy of the value "james" from name2 was copied into the location for name1.

15.2.11 Copying reference type variables

Things are different when one is working with variables of reference types. Consider the following code, where first an array called *myArray* is created with some initial values, next a variable called *thing2* is assigned the value of *myArray*:

```
var myArray = new Array( 6, 3, 5, 1 ); var thing2 = myArray;
```

Since *myArray* is a reference to the array values in memory, what has been copied into *thing2* is the reference — so now both *myArray* and *thing2* are referring to the same set of values in memory:



The implications are that if a change is made to the array values, since both *myArray* and *thing2* are referring to the same values in memory, both will be referring to the changed array.

Exercise 2

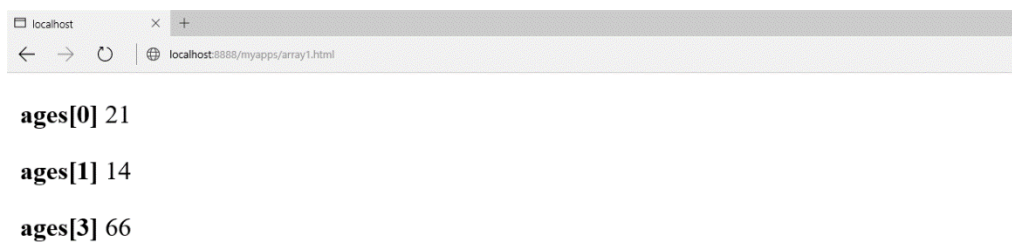
What is the value of `myArray[2]` after the following statements have been executed?

```
var myArray = new Array( 6, 3, 5, 1 ); var thing2 = myArray;  
  
thing2[1] = 27;  
thing2[2] = 19;  
thing2[3] = 77;
```

15.2.12 Array activities

Activity 1: Creating and displaying elements of an array

Write an HTML file that creates an array called `ages` containing the following numbers: 21, 14, 33, 66, 11. Write code so that the values of array elements 0, 1 and 3 are displayed. The screen should look similar to the following when the page is loaded:



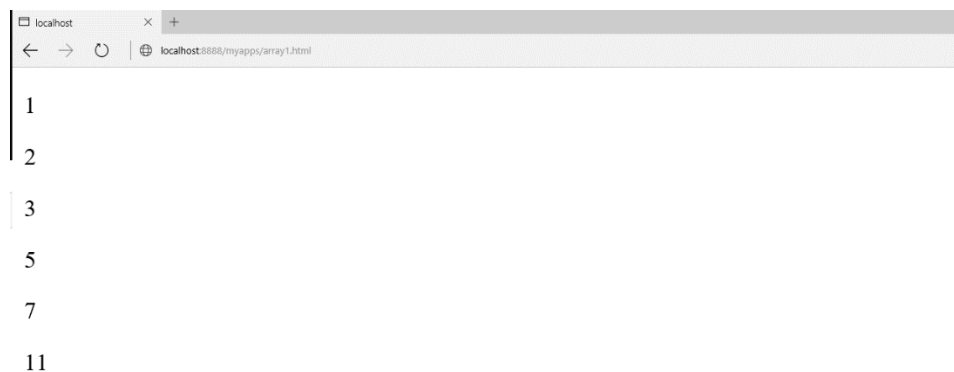
You can find a discussion of this activity at the end of the unit.

Activity 2: Iterating through array contents

Create an array called `primes` containing the following numbers: 1, 2, 3, 5, 7, 11.

Write a while loop that will iterate (loop) through each element of the array, displaying each number on a separate line. Your code should be written in a general way, so if more numbers were added to the array (say 13, 17 and 19) the loop code would not need to be changed.

The browser output should be something like:

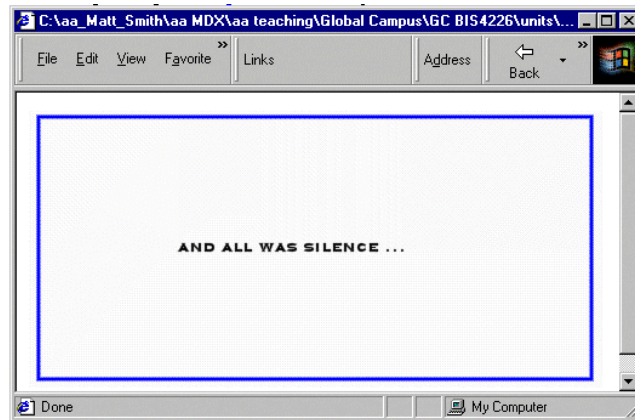


You can find a discussion of this activity at the end of the unit.

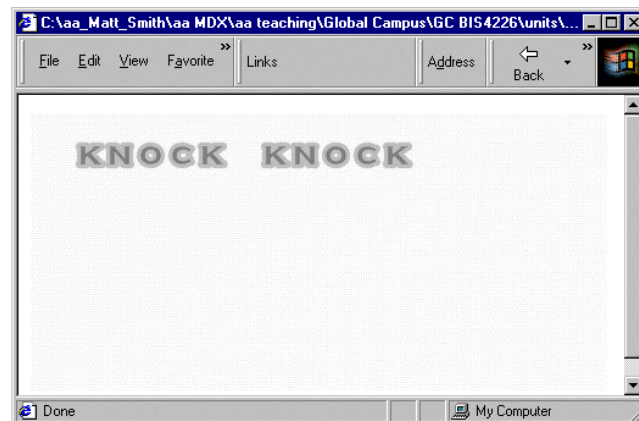
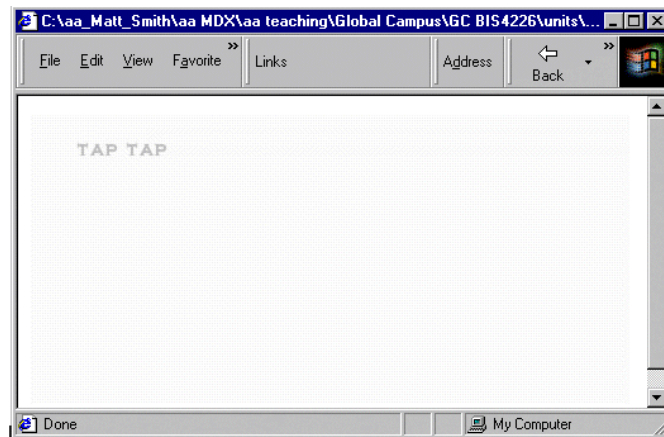
Activity 3: Animation using an array of Image objects

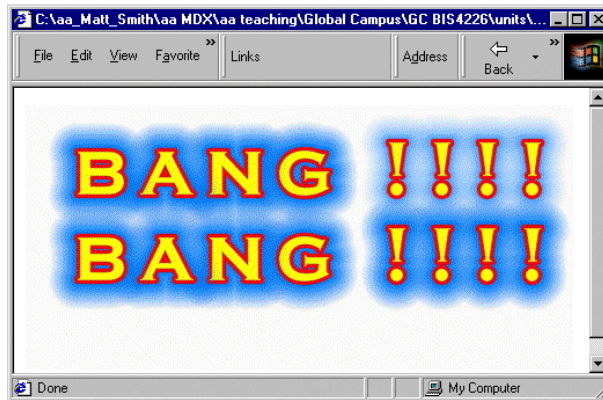
Create a Web page that shows an image, and when the mouse goes over this image the image is replaced with an

animation of 3 other images before returning back to the original image. Make the original image: silence.gif, with browser output as follows:



Make the three images that are animated: taptap.gif, knockknock.gif and BANGBANG.gif, with browser outputs as follows:





Create an array of Image objects called images. You only need to refer to the src property as in the following lines:

```
// set up image array images[0] = new Image();
images[0].src = "taptap.gif";
```

In the body section of the HTML file display the silence image (naming it noise) with an appropriate *onMouseOver* JavaScript one-liner:

```
<BODY>
IMG          NAME="noise"          SRC="silence.gif"
onMouseOver="startAnimation()">
</BODY>
```

Create a function *startAnimation()* that sets the delay time and the imageNumber to display first in the animation. Your *startAnimation()* function should then make a call to an *animate()* function written as follows:

```
function animate()
{
    document.noise.src    =    images[    imageNumber    ].src;
    imageNumber++;
    delay += 250;

    if( imageNumber < 4 )

        setTimeout( "animate()", delay );

    // if imageNumber = 4 the animation has finished
    // and this function can terminate
}
```

You might create a simple version with an unchanging delay of 500 milliseconds initially. Notice how the line:

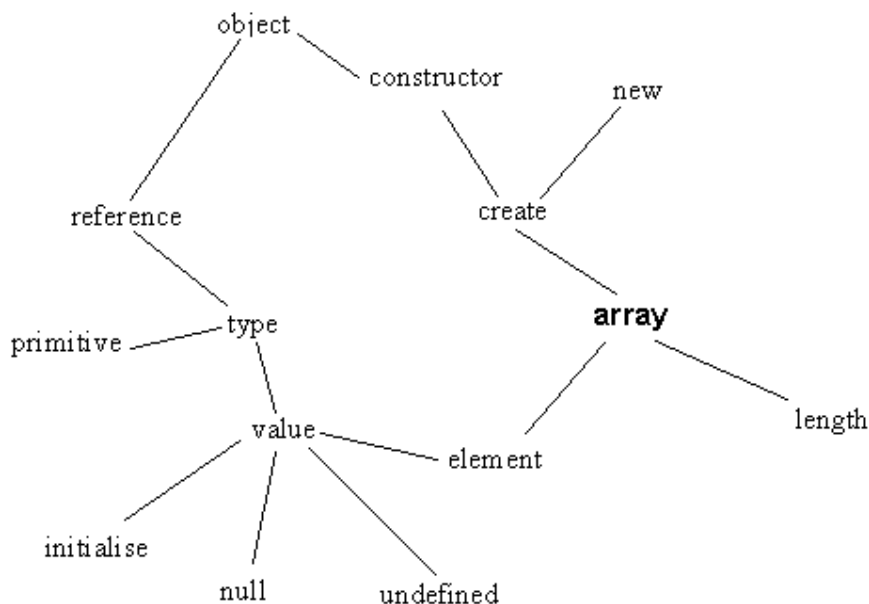
```
document.noise.src = images[ imageNumber ].src;
```

refers to the NAME of the image from the IMG tag, and resets the src property of this Image object to the appropriate array Image object src.

You can find a discussion of this activity at the end of the unit.

15.2.13 Arrays concept map

This figure illustrates the way different array concepts can be related.



15.3 The JavaScript object model

15.3.1 Introduction to the JavaScript object model

It is practically impossible to write useful JavaScript without using object properties and methods — even though many novice programmers do not realise they are making use of JavaScript objects.

Consider the following line of code:

```
document.write( "Sorry, that item is out of stock" );
```

The above line illustrates the use of many important object concepts, since it involves creating a new object ("Sorry..." is an instance built from the **String** constructor function), passing a reference to this new object as an argument to a method of another object (*write()* is a method of the document object).

JavaScript has a powerful and flexible object model. Unlike the Java programming language, JavaScript is a class-less language: the behaviour and state of an object is not defined by a class, but rather by other objects (in the case of JavaScript, this object is called the object's *prototype*. While this major difference to languages such as Java and C++ often leads people to conclude that the JavaScript object model is a simple one, the JavaScript object model is powerful enough to allow a JavaScript programmer to extend its single inheritance abilities to allow for implementations of multiple inheritance, but also most of the functionality that JavaScript is considered to have "missing". All of this may also be done dynamically, at runtime — in other words, prototypes are defined while the JavaScript programme is executing, and can be updated with new methods and properties as and when they are required, and not only at compile time as with languages such as Java.

As you progress through this part of the unit, you will both learn new ways to use and define your own objects and constructor, and also you will come to more fully understand many JavaScript programming concepts you have learned up to now. The remainder of this introduction provides a more detailed overview of the JavaScript object model presented briefly in the first JavaScript unit.

15.3.2 Objects in JavaScript

A software system is considered to be a set of components — **objects** — that work together to make up the system. Objects have **behaviour** (defined by functions and methods) that carries out the system's work, and **state**, which affects its behaviour. As we previously mentioned, objects may be created and used by programming a **constructor function** to create objects, or **instances**. A constructor is merely a JavaScript function that has been designed to be used with the *new* keyword. The constructor defines what properties an object should have (the object's state), as well as its methods.

Programming an object involves declaring **instance variables** to represent the object's state, and writing **instance methods** that implement behaviour. The separate parts of an object's state are often called its **properties**. Thus, you might have a bank account object which had properties such as *holder* of account, *address* of holder, *balance* and *credit limit*; its methods would include the likes of *credit*, *debit* and *change address*.

JavaScript represents functions (which are used to implement methods, and to create objects) as data of type *Function* (notice the capital F), so in fact there is less difference between **data properties** and **functions** (or methods) than in most other object-based programming languages. Functions can be passed and stored in variables, and any function stored in a particular variable can be executed when required.

An object is said to **encapsulate** its state (often simply called its data) and its behaviour. In the case of JavaScript, this means that each object has a **collection** of properties and methods.

A JavaScript object can **inherit** state and behaviour from any other object. This object is called a **prototype**.

An object can obtain properties and methods from the constructor function from which the object is created; from its prototype, through inheritance; and from any properties and methods added separately to the object after it has been constructed (remember that Functions are just data that can be freely assigned to variables, as and when required). Two objects made from the same constructor function can have different properties and methods from each other, since extra properties and methods can be added to each object after they have been created.

Associated with inheritance is the separate concept of **polymorphism**, the ability to call a method on an object irrespective of the constructor function used to create the object, and expect the appropriate method to be called for the given object. For example, there can be two constructor functions, *Square()* and *Circle()*, which create objects representing squares and circles, respectively. The constructor can create these objects so that they each have an *area()* method that returns the area of the object. Of course, the method will operate differently for square and circle objects, but any variable holding either a square or a circle object can have the *area* method called on it, and polymorphism will ensure that the correct *area* method is called for the appropriate object.

15.3.3 Naming conventions

Although it can initially seem rather pedantic, the following naming convention is used throughout this unit, and is widely used in various programming standards. Using a standard convention helps a programmer to understand the code that they are reading, and you are advised to adopt this convention in your own programming.

All words of a **constructor** are spelt with an initial capital letter. Examples include:

- Document
- Form
- Triangle
- Array
- String
- BankAccount
- CreditCard

The first word in an **instance** (object) name is spelt with all lower case letters; all subsequent words making up the object name are spelt with an initial capital, as per usual. Examples include:

- Document.
- ageField.
- triangle1.
- nameArray.
- customerAccount1.

The first word of a **method** name is spelt in lower case; all subsequent words making up the object name are spelt with initial capitals. Examples include:

- area.
- setBalance.
- isGreaterThan.
- postTransaction.

The first word of a property is spelt in lower case; all subsequent words making up the object name are spelt with initial capitals. Examples include:

- triangle1.colour.
- width.
- firstName.
- addressLine1.
- Value.
- fontSize.

Exercise 3

Which of the following are constructors:

- door1
- House
- MotorCar
- homeHampus

You can find a discussion of this Exercise at the end of the unit.

15.3.4 Creating objects and variables referring to objects

Objects are created in a number of ways, although the most common is by the use of the *new* operator. This is a unary, prefix operator, which means it is placed before a single operand. The new operator should be followed by a constructor function. Here are some examples:

```
new Object();  
var accountNumbers = new Array( 5 ); var firstName = new  
String( "Jose" );
```

Where is the object created?

Sufficient free memory is located in which to create the new object. The new operator returns a **reference** to the location in memory where the new object has been created. In the first of the three examples above (*new Object();*), no variable is assigned the result of the object creation, therefore there is no way for the JavaScript programmer to refer to this object later in the programme. In almost all cases, a variable will be assigned the reference to the location of the newly created object.

In the second example above the **reference variable** *accountNumbers* is assigned the reference to the location of the newly created **Array** object.

Note

Variables do not store objects, they store a reference to the object located elsewhere in memory.

Creating String objects

Although the following way of creating **String** objects is perfectly acceptable JavaScript:

```
var firstName = new String( "Jose" );
```

it is usually much more convenient to use the special syntax provided for creating **String** literal objects. The same result could be achieved with the following statement:

```
var firstName = "Jose";
```

String objects can be created in two different ways because the creation of **String** objects is such a common feature of programming that the developers of the JavaScript language provided this second, simpler syntax especially for Strings.

Creating Function objects

Just as the creation of Strings is so common the JavaScript developers provided a special syntax, the same has been done for make the creation of **Function** objects easier. As you will know from earlier units a function object can be created in the following way:

```
function myFunction( arg1,arg2, -, argN )
{
  //function statements go here
}
```

Functions are themselves objects (of the type Function), and can be created in the same way as objects using the **Function** constructor:

```
var myFunction = new Function( arg1, arg2, -, argN,
  "//function statements" );
```

In most cases the former way of creating **Function** objects is more straightforward.

15.3.5 Object properties and methods

Objects have both data properties and methods (function properties). Often a property of one object is an object in its own right (or perhaps another kind of collection, such as an array). An example is the *forms* array property of the *document* objects — this property contains details (i.e. an array) for all the forms on a Web page.

Properties

Variables (data) that "belong" to objects are called **properties**. In earlier units you have seen examples of properties, such as an array's *length* property:

```
var myArray = new Array( 5 );

alert( "length of array is: " + myArray.length );
```

In the example above the *length* property of the *myArray* object is accessed by the dot notation:

```
myArray.length
```

Methods

Methods are object properties containing functions. In earlier units you have seen examples of methods, for example the *reverse* method of an array:

```
var myArray = new Array( 3 ); myArray[0] = 11;
myArray[1] = 22;
```



```
myArray[2] = 33;

alert( myArray.reverse() );
```

In the example above the *reverse()* method of the *myArray* object is invoked by the dot notation:

```
myArray.reverse()
```

As can be seen, the only difference from (data) properties is when methods are invoked with the use of the *()* (parentheses) operator — this is a special operator expecting a variable holding a Function object to the left, and optional operands (the function arguments) between the parentheses.

It is important to realise that methods are **properties**, but ones which can also be invoked since they are properties that hold Function objects.

The 'dot' syntax

What is sometimes known as **dot** notation is used to indicate which properties an object owns. Ownership of properties can exist to any number of levels deep, and the dot notation can be used at every level. fine.

The use of the full stop (dot) can be thought of as meaning "the thing on the right belongs to the object (named collection) on the left". The general form for properties and methods to be invoked is as follows:

```
<object>.<property>
<object>.<method>()
```

Common examples include:

```
var myArray = new Array( 3 ) ; myArray.length;
var catpic = new Image();
catpic.src = "images/cartoons/sillycat1.gif";
```

In each of the above examples, the dot syntax is used to refer to a named piece of data (property) of an object. So we can refer to the *length* property belonging to the **Array** object *myArray*, and the *src* property of the **Image** object *catpic*.

We can see extensive use of the dot notation in the line:

```
document.forms[0].age.value;
```

This line refers to the **value** property of the **age** property (itself an object) of the **form** object at index **0** of the **forms[]** array object of the **document** object. Code using such a reference to the value of a form's text input box is as follows:

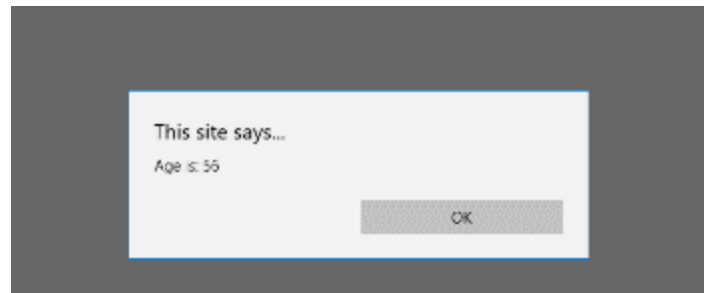
```
<script>
    function showAge()
    {
        var newAge = document.forms[0].age.value; alert("Age is: " +
newAge );
    }
</script>

<form> Enter age:
<INPUT TYPE="text" NAME="age" SIZE=10>
<INPUT TYPE="button" NAME="button1"
VALUE="click me" SIZE=10 onClick="showAge()">

</form>
```

The browser output is:

localhost x +
localhost:8080/myapps/array1.html
Enter age:



Variables are properties

When a JavaScript programme is being executed, a special object, called the 'global' object, is created. The global object exists until the programme terminates. All variables of a JavaScript programme are **properties** of this global object.

Functions are methods

In the same way that all variables of a JavaScript programme are in fact properties of the global object, likewise all the functions of a JavaScript programme are **methods** of the global object.

Functions are properties too

Since methods are just properties that hold Function objects, functions, being methods of the global object, are also stored in properties of the global object.

15.3.6 Some important JavaScript objects

Although you have been programming with objects for some weeks in JavaScript, you may not have been aware of some of the objects you have been working with. This section briefly describes important JavaScript objects.

You may wish to consult on-line and textual sources for further details of each JavaScript object.

The 'global' object

When the JavaScript interpreter starts up, there is always a single **global** object instance created. All other objects are properties of this object. Also, all variables and functions are properties of this global object.

For client-side JavaScript, this object is the instance *window* of the constructor **Window**.

The Window object for client-side JavaScript: window

When an HTML document is loaded into a browser, a single Window object instance, named *window*, is created. All other objects are properties of this window object. Since everything is a property of the window object, there is a relaxation of the dot notation when referring to properties. Thus each reference to a variable, function or object is not required to start with "window." — although this would be a more "accurate" notation to use, it is far less convenient.

Thus instead of writing:

```
document.write( "Hello" );
```

We could write:

```
window.document.write( "Hello" ); and so on.
```

One of the properties of the window object is the status property. This property is a String value that is displayed in the browser window's status bar. You can change the status bar with, or without, an explicit reference to the window object. Both these lines have the same effect:

```
window.status = "this is a test";  
status = "this is a test";
```

The Document object: document

When an HTML document is loaded into a frame of a browser window, a Document object instance — named *document* — is created for that frame. This document object, like most objects, has a collection of properties and methods. Perhaps the most frequently used method of the document object is the *write()* method:

```
document.write( "sorry, that item is out of stock<p>" );
```

One useful property of the document object is the forms property. This is actually an array of Form objects with an element for each form defined in the document. So we could refer to the first form defined in the document as follows:

```
document.forms[0]
```

The 'call' object

When a function (or method) is executed, a temporary object is created that exists for as long as the function is executing. This object is called the **call** object, and the arguments and local variables and functions are properties of this object.

It is through use of this call object that functions/methods are able to use local argument and variable/ function names that are the same as global variables/functions, without confusion. The call object is JavaScript's implementation of the concepts of variable/function **scoping** — i.e. determining which piece of memory is referred to by the name of a variable or function, when there are global and local properties with the same name.

String objects

Strings are objects. A frequently used property of String is *length*. String includes methods to return a new **String** containing the same text but in upper or lower case. So we could create an upper case version of the **String "hello"** as follows:

```
var name = "hello";  
alert( name.toUpperCase() );
```

It should be noted that none methods belonging to string objects never change the string's value, but they may return a new String object, such as in the example above.

Array objects

Since Arrays are rather an important topic in the own right, Array objects are given their own section at the end of this unit — although you may wish to skip ahead to read that section now to help your understanding of object with this more familiar example.

Function objects

You should refer back to the earlier unit on functions.

As previously stated, functions are of the type Function, and can be treated as data variables or properties, or they can be treated as sub-programmes and executed using the () operator.

Math objects

The Math object provides a number of useful methods and properties for mathematical processing. For example,

Math provides the following property:

```
Math.PI
```

that is useful for many geometric calculations.

An example of some other methods provided by the Math object include:

```
Math.abs( );  
Math.round( );  
Math.max( n1, n2 );
```

These may be used in the following way:

```
document.write( "<p>PI is " + Math.PI );
document.write( "<p>The signless magnitudes of the numbers -17 and 7 are "
               + Math.abs( -17 ) + " and " + Math.abs( 7 ) );
document.write( "<p>The interger part of 4.25 is " + Math.round( 4.25 )

);document.write( "<p>The larger of 17 and 19 is " + Math.max(17, 19) );
```

15.3.7 The 'Triangle' object

To focus our investigation of user defined constructors and objects, we will create 'Triangle' objects. We shall be defining the Triangle constructor function as follows:

- **Constructor Function** - *Triangle(initWidth, initHeight)*
- **Properties** - *numSides, width, height, colour*
- **Methods** - *larger(t1, t2), area()*

As you work through this section on JavaScript objects you will become familiar with not only with Triangle objects, but with the definition and use of your own constructors and objects.

The following is an example of some code using Triangles:

```
var triangle1 = new Triangle(10, 20);
document.write("<p><b>triangle1 has height: </b>" + triangle1.height );
document.write("<p><b>triangle1 has width: </b>" + triangle1.width );
document.write("<p><b>triangle1 has area: </b>" + triangle1.area() );
document.write("<p><b>triangle1 has colour: </b>" + triangle1.colour );
```

15.3.8 User defined objects

Constructor functions

Objects are created with the new operator and a constructor function. The new operator requires the name of a function to its right (with optional arguments), and will return a reference to the memory location of the newly created object.

Examples of creating objects include:

```
var myArray1 = new Array( 3 ); var myArray2 = new Array();
var firstName = new String( "Jonathan" ); var catPicture = new
Image();
```

If we assume we have defined a Triangle constructor, we can create new objects in just the same way as above:

```
var triangle1 = new Triangle( 5, 10 ); var triangle2 = new
Triangle( 10, 20 ); var triArray = new Array( 3 );
triArray[0] = new Triangle( 17, 92 );
```

A constructor function creates a new object, possibly initialising some of these new objects properties based on arguments provided when the function was called. We can see from the above that the Array() constructor function can create an array if provided with no arguments, or a numeric argument (for the size of the array), or a number of arguments (for the initial elements of the array).

Our Triangle constructor requires two arguments — the first is the width of the new triangle, the second is the height. Our Triangle constructor function looks as follows:

```
function Triangle( newWidth, newHeight )
```

```

{
  this.width = newWidth; this.height = newHeight;
}

```

Constructor functions make use of the special variable *this* — when a function is called with the *new* operator, a new object is created in memory, which the *this* variable refers to. Using this variable, the constructor is able to assign values to new properties called *width* and *height*. When a constructor function finishes, the reference to the newly created object is returned.

So after the following line is executed:

```
var triangle1 = new Triangle( 5, 10 );
```

the object *triangle1* should have a *width* property with value 5 and a *height* property with value 10. This can be investigated by using the dot notation and some *document.write()* statements:

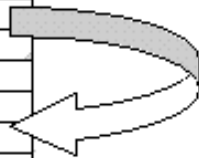
```

document.write("<p> width is : " + triangle1.width );
document.write("<p> height is : " + triangle1.height );

```

We can imagine memory to be arranged something like the following (remember that all object variables are reference variables):

Memory location	value
<i>triangle1</i>	001729
001727	
001728	
<i>obj01.width</i>	5
<i>obj01.height</i>	10
001731	
001732	
001733	
001734	



Functions are objects with properties

Now we have a constructor function, *Triangle*, we can use this function object as an object that defines *Triangles*.

Object 'prototypes'

A constructor function object has a special property named *'prototype'* — this is an object through which inheritance is implemented for all objects created from the constructor: every object created from the constructor can inherit the properties and methods of the prototype object.

For example, by creating a new property or method for the prototype, this property or method is made available to all objects created from the constructor. Prototypes are a very useful way of defining methods for objects, and for setting default object properties.

For example, let us imagine that we wish all our **Triangle** objects to have the property *colour* initially set to the String "blue". To provide this property to all **Triangle** objects created in the future, we can assign the property to the *Triangle* prototype as follows:

```
Triangle.prototype.colour = "blue";
```

Likewise, if we wish to make a method available to all *Triangle* objects, we need to assign a function to an appropriate property of the prototype. As useful method for *Triangle* objects is the calculation of their area. A function to calculate the area of a triangle could be defined as follows:

```

function triangleArea()
{

  // triangle height is half (width * height) var area = 0.5
  * this.height * this.width;
}

```

```

    // return calculation to 2 decimal places return
    Math.round( area * 100 ) / 100;
}

```

Notice how this function has been written to refer to whatever object it is called from by using the variable *this*.

We now need to assign this function to a property of the **Triangle** prototype. The method name *area* seems a good choice:

```
Triangle.prototype.area = triangleArea;
```

Notice, we are not executing this function, so we do not use the `()` operator. We are making `Triangle.area` refer to the same Function object as `triangleArea`.

After adding the above lines to a script, we can now see if a newly created Triangle object has a colour property and can use its area method:

```

var triangle1 = new Triangle( 5, 10 );

document.write("<p> colour property is: " +
triangle1.colour); document.write("<p> area method
returns: "+ triangle1.area() );

```

Remember, to invoke a function/method we must follow the function name with the `()` operator as above.

15.3.9 Implementing instance variables and methods

There is a distinction made to where variables and methods belong:

- **instance variables** — these are properties that can be different for each instance object (e.g. `triangle1` might have *height* 25, while `triangle2` has *height* 40).
- **instance methods** — methods that each object can apply to itself, and that has access to the properties of the object (e.g. `triangle1` can call its *area* method, this method returns a value calculated using the *height* and *width* properties of variable1).
- **class variables** — in a class-based object model, these are variables useful / relevant to the class as a whole, but that do not have anything to do with any particular instance of the class (e.g. the number of sides of a Triangle is 3, the value of PI is 3.1415926535). JavaScript, too, has the concept of "class variables", but the variables do not belong to classes, but rather to the constructor functions (which themselves are Function objects). These kinds of variables do not require an instance created from the Triangle or Math constructors in order for them to have meaning.
- **class methods** — Similar to class variables, these are methods that are useful / relevant to the constructor function, and does not require an instance of any object created by the constructor in order for it to be useful.

We shall briefly consider how each of these is implemented in JavaScript.

Instance properties and instance methods

The previous section on the prototype property illustrates precisely what is meant by instance properties (properties) and instance methods:

- A property added to the prototype, or created inside the constructor, is an instance variable (property).
- A method added to the prototype (or in the constructor) is an instance method.

In fact, since a method is just a special kind of property, instance variables and instance methods can all be considered to be implemented as instance properties of the prototype.

You may find it useful to clearly comment the implementation of instance properties and methods in your code, such as the following lines illustrate:

```
////////////////////////////////////////
// instance (object) methods
////////////////////////////////////////

function triangleArea()
{

// triangle height is half (width * height) var area = 0.5 * this.height *
this.width;

// return calculation to 2 decimal places return Math.round( area * 100) /
100;
}

// add instance method "area" to the "prototype" property of "Triangle"
// (i.e. add this method to the "prototype" property of the "Triangle"

Triangle.prototype.area = triangleArea;

////////////////////////////////////////
// instance (object) variables (properties)
////////////////////////////////////////
// the default colour for triangles is "blue" Triangle.prototype.colour =

"blue";
```

Class properties

A "class properties" can be implemented by assigning a value to a property of the constructor object. This has been done for values such as *Math.PI*.

For example, we can implement the class property numSides with a value of 3 to the Triangle constructor as follows:

```
////////////////////////////////////////
// class properties
////////////////////////////////////////
// add property "numSides" Triangle.numSides = 3;
```

At any later point in the code we can then refer to this class property using the dot notation. For example:

```
document.write( "the number of sides of a triangle is: " +
Triangle.numSide
```

Class properties can be referred to even if no instances have been constructed (since the property belongs to the constructor, and not to the prototype).

Class methods

A class method, such as *Math.abs()*, is implemented in the same way as a class property — a function is assigned to a property of the constructor function.

For example, we can we can implement a Triangle class method called larger(). This method requires two

arguments, each a Triangle object, and returns the String "first" if the first Triangle is the larger, otherwise the method returns "second" (i.e. if the second is the same or larger it returns "second"). First we must create a function:

```
function triangleLarger(t1, t2)
{
  if( t1.area > t2.area ) return "first";
  else
    return "second";
}
```

t1 and t2 are the two Triangle object references passed as arguments. This function makes use of the area instance method for each of the two Triangle objects — the larger triangle is defined to be the one with the larger area.

Then we must assign this function to an appropriately named property of the constructor object:

```
// add function to constructor (i.e. add to constructor function object

Triangle.larger = triangleLarger;
```

Notice, we are not executing this function, so we do not use the () operator. We are making Triangle.larger refer to the same Function object as triangleLarger.

The method can then be used as follows:

```
var triangle1 = new Triangle(10, 20);
var triangle2 = new Triangle(5, 10);
document.write("<p><b>the larger of triangles 'triangle1' and 'triangle
document.write( Triangle.larger( triangle1, triangle2 ) );
```

15.3.10 The JavaScript object search chain

When an object is created, a reference is created to the location in memory where the object's properties and methods are stored. Also, a reference is created to the object's prototype. When JavaScript wishes to retrieve the value of one of this object's properties, or to invoke one of this object's methods, it will first search the details stored for the object, and then if the desired property/object is not found it follows the reference to the object's prototype, where it continues to search there for the property or method. Again, if the property/method is not found in the prototype, it searches the prototype object's own prototype, and so on. Eventually, if not found, JavaScript will reach the object **Object** — this is the 'parent' of all built-in and user-defined objects.

If the property or method is not found, the value *undefined* is returned.

It is important to bear in mind that objects are **reference** types, as are prototype objects (since they are themselves only objects).

Overriding inheritance

In the previous sections we have investigated the JavaScript features of prototypes and object inheritance. However, any object can override an inherited property or method by defining its own.

In our Triangle example, we have created a prototype *colour* property with a value of the **String** "blue". This means that, by default, all our **Triangle** objects will inherit this property and value. However, let us consider that while for most objects the colour blue is fine, for the object *triangleR* we may wish to have a *colour* property with value "red". The programming of this situation is very straightforward:

```
var t1 = new Triangle( 10, 20 );
var triangleR = new Triangle( 6, 6 ); triangleR.colour =
"red";
```

After execution of the above lines, part of the memory will look as follows:

Memory location	value
(Triangle) t1	001726
	001727
	001728
obj01.width	001729
obj01.height	001730
	001731
Triangle.prototype	001732
	001733
obj02.colour	001734
obj02.area	001735
	001736
	001737
(Triangle) triangleR	001738
	001739
	001740
obj03.width	001741
obj03.height	001742
obj03.colour	001743
	001744

When JavaScript comes across code referring to the colour property of object *triangleR*:

```
document.write( "colour of triangleR is " + triangleR.colour );
```

it will follow the reference for *triangleR* to location 001741 and search for a property colour. It will succeed at location 001743. The value of this property will be returned, and this value, "red", will be passed to the *document.write()* method. Since the property value was found at the object referred to by *triangleR*, no search is made of the **Triangle** object *prototype*. In this way, object *triangleR* has been said to have overridden the inherited colour property with its own value of "red".

It is important to notice that although the object referred to by *triangleR* has overridden its inherited colour property, there has been no change to any of the other **Triangle** objects. So, for example, object *t1* still inherits the colour property of *Triangle.prototype*.

Note - the difference between JavaScript and other object- language

In JavaScript, a specific object can override what it inherits from its object prototype. All other objects inheriting from the prototype can remain unaffected by this change. This differs from many other languages, such as Java, where any changes in inheritance are reflected across all objects of that given type.

The dynamic nature of inheritance through prototype references

A final concept to appreciate regarding prototype based inheritance in JavaScript is that the inheritance of prototype properties and methods is dynamic: if an object is created and the prototype is subsequently changed, the changes will be reflected in the state and behaviour of any object inheriting from it.

We shall illustrate this dynamic inheritance with the following code:

```
var myTriangle = new Triangle( 6, 8 ); var triangle2 = new
Triangle( 11, 22 );
document.write("<p> colour of myTriangle is " +
myTriangle.colour ); document.write("<p> colour of triangle2
is " + triangle2.colour );
Triangle.prototype.colour = "green";
document.write("<p> colour of myTriangle is " +
myTriangle.colour ); document.write("<p> colour of triangle2
is " + triangle2.colour );
```

After the execution of the first two lines:

```
var myTriangle = new Triangle( 6, 8 ); var triangle2 = new Triangle( 11, 22 );
```

part of the memory will look as follows:

Memory location	value
(Triangle) myTriangle	001726
	001727
	001728
obj01.width	001729
obj01.height	001730
	001731
Triangle.prototype	001732
	001733
obj02.colour	001734
obj02.area	001735
	001736
	001737
(Triangle) triangle2	001738
	001739
	001740
obj03.width	001741
obj03.height	001742
	001743
	001744

So when the first two document.write() statements are executed:

```
document.write("<p> colour of myTriangle is " + myTriangle.colour );
document.write("<p> colour of triangle2 is " + triangle2.colour );
```

JavaScript retrieves the inherited colour "blue" for each object. However, after execution of the line that changes the prototype property:

```
Triangle.prototype.colour = "green";
```

memory is changed to look as follows:

Memory location	value
(Triangle) myTriangle	001726
	001727
	001728
obj01.width	001729
obj01.height	001730
	001731
Triangle.prototype	001732
	001733
obj02.colour	001734
obj02.area	001735
	001736
	001737
(Triangle) triangle2	001738
	001739
	001740
obj03.width	001741
obj03.height	001742
	001743
	001744

Thus when the last two `document.write()` statements are executed, the reference is followed to the *colour* value at location 001734 "green", and it is this changed property value that is inherited by both the **Triangle** objects.

It is important to understand, then, that **changing a prototype dynamically changes the inherited properties and methods for all objects sharing the prototype from that point onwards**, and should not be an action used lightly, or when some simpler solution is possible.

15.3.11 Object activities

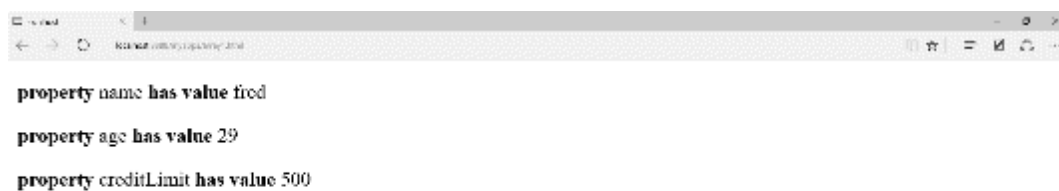
Activity 4 - Iterating through object properties

Create an object `customer1` with the following properties and values:

Object — <code>customer1</code>	
Property name	Property value
<code>name</code>	<code>"fred"</code>
<code>age</code>	<code>29</code>
<code>creditLimit</code>	<code>500</code>

Write a *for/in* loop that will iterate through the properties of this object, displaying each property name and value a separate line.

The browser output should look something like the following:



You can find a discussion of this activity at the end of the unit.

Activity 5 — Constructor function for Rectangle

Create a constructor function for **Rectangle** objects. The properties of new **Rectangle** objects to be initialised in the constructor are its *length* and *height*.

Class — <code>Rectangle</code>	
	Constructor function
	<code>Rectangle(initLength, initHeight)</code>
Instance Properties	Instance Methods
<code>length</code>	
<code>height</code>	

Your constructor function should be designed to create new **Rectangle** objects with code such as the following (note the *length* is the first argument, and the *height* the second):

```

var rect1 = new Rectangle( 15, 20 ); document.write("<p> length is : " +
rect1.length ); document.write("<p> height is : " + rect1.height );

```

You can find a discussion of this activity at the end of the unit.

15.4 Arrays as objects

Arrays are objects, and have properties and methods like any other object. The prototype of the array object can be illustrated by the following figure:

Class — <code>Array</code>	
Instance Properties	Instance Methods
<code>length</code>	<code>join()</code>
	<code>sort()</code>
	<code>reverse()</code>
	<code>concat()</code> recent addition to JavaScript
	<code>slice()</code> recent addition to JavaScript
	<code>splice()</code> recent addition to JavaScript
	<code>push()</code> recent addition to JavaScript
	<code>pop()</code> recent addition to JavaScript
	<code>shift()</code> recent addition to JavaScript
	<code>unshift()</code> recent addition to JavaScript
	<code>toString</code> recent addition to JavaScript
	<code>toSource()</code> recent addition to JavaScript

Earlier this unit we dealt in detail with the *length* property of arrays. The first three methods of the **Array** object list above are discussed below. You are advised to consult on-line sources and books for details of the many other **Array** functions.

15.4.1 Array method: join

The *join()* method is a straightforward way of converting from the tabular, numerically indexed structure of an **Array** to a simple text **String**. *join()* copies all elements of an **Array** into a **String** object, separating each element value with the provided String argument, or with a comma "," if no argument is provided.

The general form of the *join()* method is:

```
<array>.join( <separator> );
```

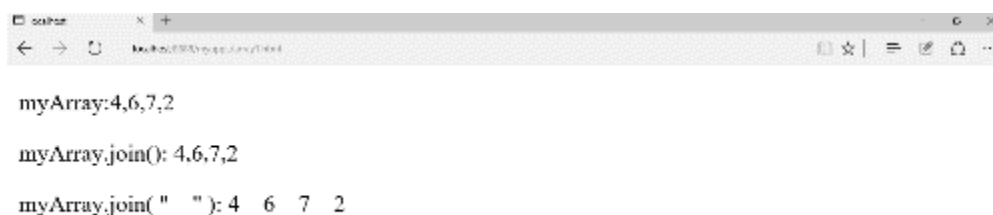
where <array> is a variable holding an array, and <separator> (if present) is a String to be used to separate each array element. The *join()* method is called by default when an **Array** object is passed as a parameter to a *document.write(...)* statement.

An example of some code using the *join()* method is:

```
var myArray = new Array( 4, 6, 7, 2 );

document.write("<p> myArray:" + myArray );
document.write("<p> myArray.join(): " + myArray.join() );
document.write("<p> myArray.join( \" _ \" ): ' + myArray.join( \" _ \" ) );
```

The output of the code is:



As can be seen from the output, calling *document.write(...)* with simply the **Array** variable as an argument is the

same as providing the **Array** name and its default *join()* method. However, when the *join()* method is called with the argument " _ " this String is returned as the separator for each array element in the String returned by the method.

15.4.2 Array method: sort

As its name suggests, the *sort()* Array method provides a mechanism for changing the order of the elements of an **Array** according to provided criteria.

sort()'s default ordering is ascending, alphanumeric order. This produces some unintuitive results for numeric values:

```
var myArray = new Array( 4, 16, 7, 2 );
document.write("<p> myArray: " + myArray );
document.write("<p> myArray.sort(): " + myArray.sort() );
```

Since 16 comes **alphabetically** before 2 (1 is alphanumerically lower than 2), it appears first in the sorted array. To sort array elements according to numerical (or some other) order an ordering function has to be provided as an argument to the *sort()* method. This ordering **function** must accept two arguments and return a numeric indication indicating which of the two arguments should occur before the other. If the function returns the value N, then:

- $N < 0$: if the first argument should appear before the second.
- $N > 0$: if the second argument should appear before the first.
- $N = 0$: if the two arguments are equivalent.

An example of a function that places elements into ascending numeric order is:

```
function smaller(n1, n2)
{
    return (n1 - n2);
}
```

The above function *smaller()* returns a negative value if the first argument is the smaller, zero if the two numbers are equal, and a positive number if the second argument is the smaller.

A more explicit (although less elegant) function, this time to return the larger of two numbers, can be written as follows:

```
function larger(n1, n2)
{
    if( n1 > n2 )
        return -1; // "n1" first else if (n1 < n2)
        return 1; // "n2" first else
        return 0;
}
```

As we can clearly see, this *larger()* function returns -1 if the first argument is larger, 1 if the second argument is larger, and zero if the arguments are the same. An example of some code that illustrates both the default (alphabetical) sorting, and sorting using the *larger()* function above, is as follows:

```
var myArray = new Array( 4, 16, 7, 2 );
document.write("<p> myArray: " + myArray );
document.write("<p> myArray.sort(): " + myArray.sort());

//////////
// function to return the smaller of 2 numbers
////////// so will order numbers in DESCENDING order

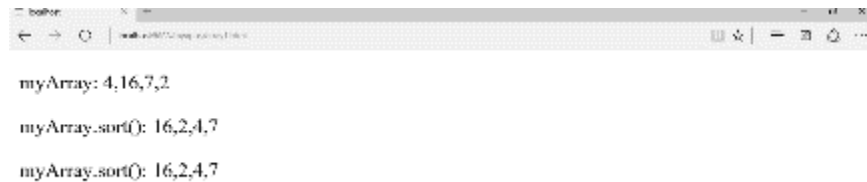
function larger(n1, n2)
{
    if( n1 > n2 )
```

```

    return -1; // "n1" first else if (n1 < n2)
    return 1; // "n2" first else
    return 0;
}
document.write("<p> myArray.sort(): " + myArray.sort(larger));

```

The browser output from the above code is:



15.4.3 Array method: reverse

The *reverse()* method does not return any values; rather, it rearranges the **Array** elements into reverse order (i.e. reverse order of the numerical indices from 0..(length-1)).

Consider an array `myArray` with the following values:

array <code>myArray</code>	
Index	Value
<code>myArray[0]</code>	"hello"
<code>myArray[1]</code>	false
<code>myArray[2]</code>	16

If the *reverse()* method is invoked on `myArray`, the following is obtained:

array <code>myArray</code>	
Index	Value
<code>myArray[0]</code>	16
<code>myArray[1]</code>	false
<code>myArray[2]</code>	"hello"

Some code to demonstrate this is as follows:

```

var myArray = new Array(3);

myArray[0] = "hello";
myArray[1] = false;
myArray[2] = 16;

document.write("<p> myArray: " + myArray );

myArray.reverse();

document.write("<p> after reversing...");

```

```
document.write("<p> myArray: " + myArray );
```

The browser output of the above code is:



```
myArray: hello,false,16
after reversing...
myArray: 16,false,hello
```

15.4.4 Single and multi-dimensional arrays

The discussion of arrays have so far concerned what are known as single-dimension arrays, which are arrays forming an arrangements of values that can be accessed using a single numeric index, and can be pictured as a row of boxes, each box being numbered (by the index) and holding a value. Most programming languages provide facilities for multi-dimensional arrays, which are arrays requiring multiple numeric indices. A two dimensional array can be pictured as a book shelf, where each shelf is itself a single-dimension array.

In JavaScript, a two-dimensional array — holding months and the days of the month — can be created as follows:

```
var delivery = new Array(12); delivery[0] = new Array(31);

// January
delivery[1] = new Array(29); // allow for Feb in leap years
delivery[2] = new Array(31); // March...
delivery[11] = new Array(31); // December
```

Notice that a two dimensional array is created by setting each element of an single-dimension array to be arrays themselves.

This array could be used to represent the days on which a company could possibly perform a delivery, and wee could place true/false values into each element to indicate whether a delivery is possible on that day or not. If a delivery is not possible on the 1st or 2nd of January, but is possible on the third, we might write:

```
delivery[0][0] = false; // no delivery possible 1st Jan
delivery[0][1] = false; // no delivery possible 2nd Jan
delivery[0][2] = true; // delivery is possible 3rd Jan
```

However, let us assume that due to poor climate and annual staff holidays that the company cannot perform deliveries during August. JavaScript's flexible nature allows us to do the following:

```
delivery[7] = false; // August - no deliveries
```

Notice that a multi-dimensional array can easily have different sizes for its different rows, and some rows need not even be arrays at all.

To make working with multi-dimensional arrays easier, one can either add new methods to the Array prototype, or create new constructor functions which use Array as their prototype. Consider the following constructor function which attempts to make creating multi-dimensional arrays easier. It takes two arguments — the size of the first and second dimensions — and creates a two dimensional array:

```
function two_dim_array(length_d1, length_d2)
{
    // loop to make each element of "this" an
    // array with length "length_d2"

    var i = 0;
    while( i < length_d1 )
```



```

        {
            this[i] = new Array( length_d2 );
            i++;
        }

        // make the "length" property of array "this"
        // an object with properties
        // "d1" = "length_d1"
        // "d2" = "length_d2"

        this.length = { d1:length_d1, d2:length_d2 };
    }
    two_dim_array.prototype = Array;

    var myArray = new two_dim_array(3,2);

```

15.5 Objects as associative arrays

While it is clear that arrays are JavaScript objects, JavaScript also allows objects and their properties to be accessed as if they were arrays. Normal JavaScript arrays index their values in order using integer indices. When treating JavaScript objects as arrays, Strings are used as indices to associate each property value with the object name. Arrays of this kind are called "associative arrays".

To refer to an object property using array notation, simply pass the property name as a String to the array square brackets applied to the object, as follows:

```
objectName[ "propertyName" ]
```

An example of the use of associative array notation for accessing object properties is illustrated in the following code:

```

var object1 = new Object;
object1.name = "Joanne"; object1.age = 27;
object1.nationality = "British";
document.write(<p> property name: " + object1["name"] );
document.write("<p> property age: " + object1["age" ] );
document.write("<p> property nationality: " +
object1["nationality" ] )
var myArray = new Array(5);
document.write("<p> array length: " + myArray["length" ] );

```

It is important to note that although the square bracket indexing is used with associative arrays, it is objects that are being processed, and that these objects must be created with constructor functions (and the *new* keyword), or with object literals — associative arrays are not created using the built-in Array object.

15.5.1 Enumerating associative arrays with FOR/IN loop statements

One form of loop statement created for the processing of associative arrays is the FOR/IN statement. This statement allows the processing of all user-defined properties of arrays (and all inherited user-defined properties).

An example of the FOR/IN statement is:

```

var object1 = new Object;

object1.x = 29;
object1.y = 6;
object1.z = 55;

for( var property in object1)

```

```

{
  document.write("<p>the value of property " + property + "
  is " + objec
}

```

Note

It is important to note that the developers of JavaScript did not prescribe the order in which the FOR/IN statement processes object properties. Therefore the output of such statements may vary from browser to browser, and the ordering cannot be relied upon to always be the same.

If the order is important, either more sophisticated programming is required, or the code must be tested on the specific browsers the JavaScript needs to run on — along with a clear warning that the code may not work on other browsers. Obviously, it is best to use this statement in such a way that the order in which properties are processed does not matter.

The real usefulness of the FOR/IN statement is that it allows the iterative processing of all properties of an object, in the same way that a numeric loop variable can be used to iteratively process all elements of a normal array. This technique is mainly useful where

- The order of the properties either does not matter, or can be resolved
- All (or at least most) properties of an object need to be processed in the same way
- There are a large number of properties, so that a FOR/IN statement saves both programme statements and helps avoid the chance of forgetting to process an object property

15.5.2 Using FOR/IN for numerically index arrays

FOR/IN statements can be used to enumerate normal, numerically index arrays. For example, consider the code:

```

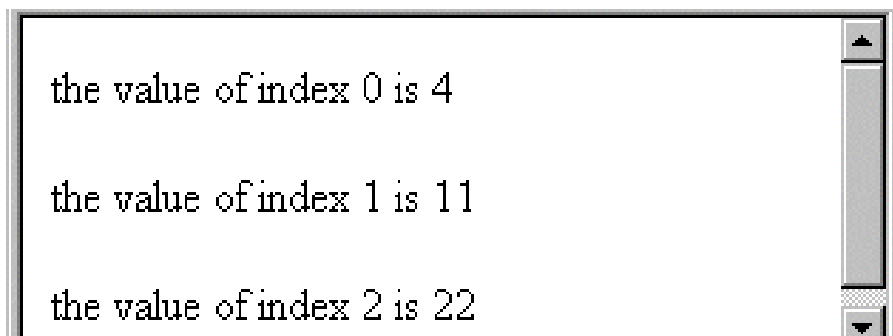
var object1 = new Array( 3 );

object1[0] = 4;
object1[1] = 11;
object1[2] = 22;

for( var index in object1)
{
  document.write("<p>the value of index " + index + " is " +
  object1[ index])
}

```

The above code produces the following output on a browser:



However, as with any use of FOR/IN, there is no guarantee that the elements will be retrieved in the numerical index order. To ensure that elements are enumerated in numerical sequence (starting at 0), use a loop such as the following:

```
var object1 = new Array( 3 ); object1[0] = 4;  
object1[1] = 11;  
object1[2] = 22;
```

```

var index = 0;

while( index < object1.length )
{
    document.write("<p>the value of index " + index + " is " +
    object1[ i

    index++;
}

```

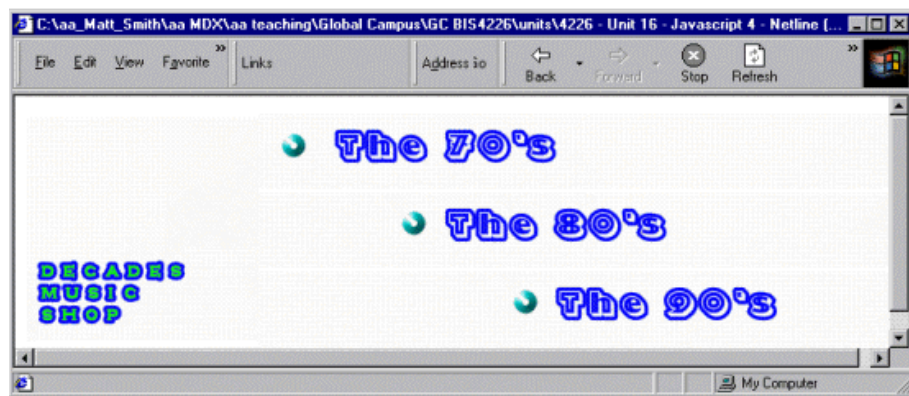
Note

You may wish to read about the more elegant FOR loop statement for simple numerical loops such as the one above.

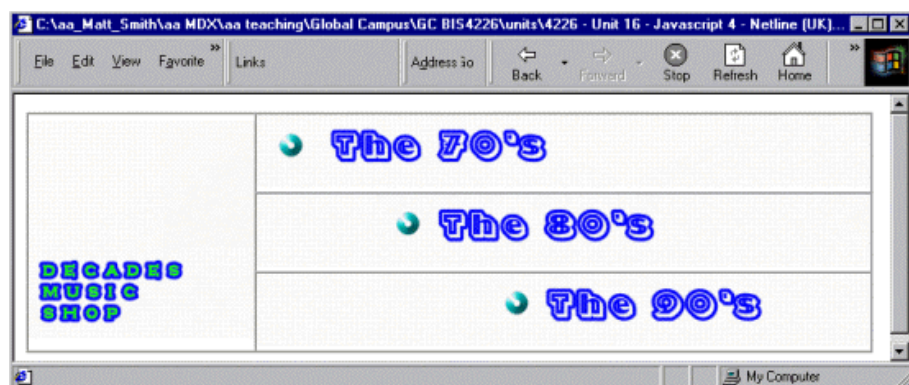
15.5.3 Activities and further work

Activity 6 — Using Arrays of Images to improve "Decades Music Shop"

The company "Decades Music Shop" has set up an on-line music ordering website. The site looks as follows:

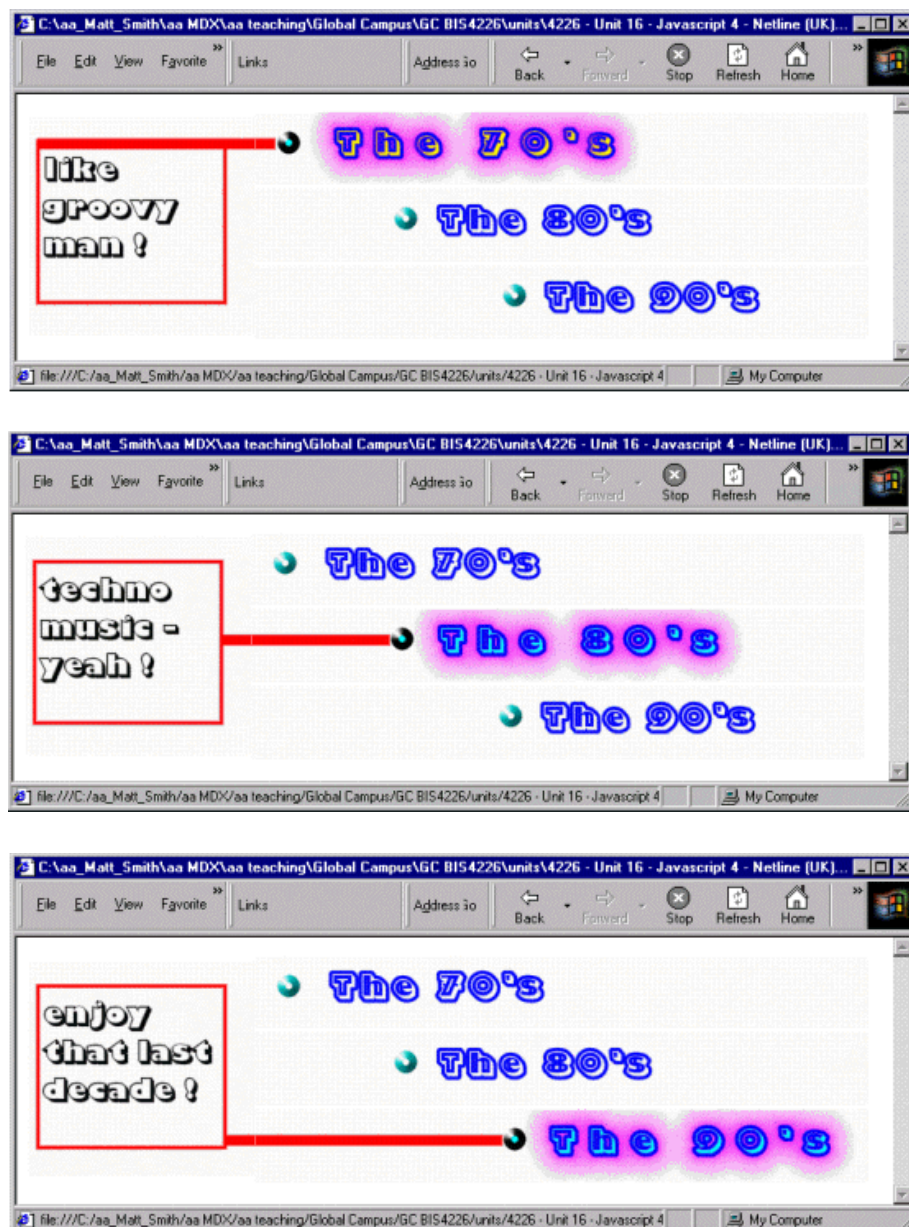


The Webmaster who set up the site has made this a visually interactive home page by arranging the four images in a table. The table has three rows for "The 70s", "The 80s" and "The 90s" and the first column is an image spanning all 3 rows. The same page, with the table border set to 1 is as follows:



The images for "The 70s", "The 80s" and "The 90s" change when the mouse is over them — i.e. they have been made rollover images via the onmouseover and onmouseout event

attributes of the IMG tag. The three screenshots below show how the page changes when the mouse is over each of these images in turn:



The current music shop Webmaster does not know about Arrays, and so has implemented a separate JavaScript function for each images onMouseOver and onMouseOut event — i.e. the Webmaster has had to create two functions for each image. The functions for the 70s image are as follows:

```
function select70s()
{
    document.the70s.src = "the70s_selected.gif";
    document.textArea.src = "the70s_selected_TEXT.gif";
}

function deselect70s()
{
    document.the70s.src = "the70s_UNselected.gif";
    document.textArea.src = "UNselected_TEXT.gif";
}
```

```
}
```

The HTML table is created with the following code:

```
<TABLE CELLSPACING=0 CELLPADDING=0 WIDTH=202 BORDER=0>

<TR>
  <TD rowspan = 3>
    <IMG NAME="textArea" SRC="UNselected_TEXT.gif">
  </TD>

  <TD>
    <A HREF="the70s.html" onMouseOver="select70s();"
      onMouseOut="deselect70s();">
    <IMG NAME="the70s" SRC="the70s_UNselected.gif"
      BORDER=0></A>
  </TD>
</TR>

<TR>
  <TD>
    <A HREF="the80s.html" onMouseOver="select80s();"
      onMouseOut="deselect80s();">
    <IMG NAME="the80s" SRC="the80s_UNselected.gif"
      BORDER=0></A>
  </TD>
</TR>

<TR>
  <TD>
    <A HREF="the90s.html" onMouseOver="select90s();"
      onMouseOut="deselect90s();">
    <IMG NAME="the90s" SRC="the90s_UNselected.gif"
      BORDER=0></A>
  </TD>
</TR>

</TABLE>
```

Notice how the 70s image *onMouseOver* and *onMouseOut* event attributes call the two functions listed above.

Your task is to reduce the number of required functions from six to two. You are to create a single function called *selectImage()* and another called *deselectImage()*, which are passed the number of the image to select or deselect. Refer to the 70s image as number 0, the 80s image as number 1 and the 90s image as number 2.

You are to achieve this reduction of functions by creating three arrays, called *selectedImages*, *deselectedImages*, and *imageNames*. They contain Strings holding the name of the relevant image as stated in the HTML *IMG NAME="..."* tag. Each array should contain an *Image* object whose *src* property has been set to the appropriate image.

changing a document image when referring to an array can be done in a way similar to this:

```
document.the80s.src = selectedImages[1].src;
```

If your function calls its index argument *imageNumber*, you can create a *String* that contains the statement you wish executed, and then use the *eval()* function to execute that *String*:

```

var imageStatement = "document." + imageNames[imageNumber ] ;

imageStatement += ".src = selectedImages[" + imageNumber + "].src";

eval( imageStatement );

```

So if imageNames[imageNumber] contains "the80s" and then the variable imageStatement will contain precisely the statement you wish to execute!

You can find a discussion of this activity at the end of the unit.

Activity 7 — Using associative arrays of images to improve "Decades Music Shop"

Associative arrays offer a way to associate names rather than numbers with array elements.

Your task is to make the music_arrays.html more meaningful by replacing your arrays of images with objects that have a property for each image. This means that the need for an array of image names is removed, the need for an eval statement is removed, and the argument to the select and deselect functions becomes a more meaningful String, such as "the70s", or whatever IMG NAME has been set to.

HINT 1: Create an associative array (object) with images as properties.

Replace your array creation statements with object creation. Then replace array element assignment statements with object property statements:

```

var selected = new Object; selected.the70s = new Image();
selected.the70s.src = "the70s_selected.gif";

```

Thus we now have an object (i.e. an associative array) named selected, that can have its Image properties accessed via the property name, e.g.:

```

selected["the70s"].src;

```

HINT 2: Pass the property name as an argument from onMouseOver and onMouseOut Replace your

HTML IMG lines with something similar to the following:

```

<A HREF="the70s.html" onMouseOver="selectImage('the70s');"
  onMouseOut="deselectImage('the70s');">

<IMG NAME="the70s" SRC="the70s_UNselected.gif" BORDER=0>

</A>

```

(remember, since you cannot have double quotes within double quotes, you will need to mix double and single quotes for the onMouseOver JavaScript one-liner String, since you are passing a String argument as part of your JavaScript statement).

You can find a discussion of this activity at the end of the unit.

Activity 8 — Using user-defined object with methods to improve "Decades Music Shop"

The previous activity greatly reduced the amount of code, and improved the readability and meaningfulness of function calls.

However, at present we have three objects containing our images, and two isolated functions. We can tidy things up even further by encapsulating the select and deselect functions into user-defined, rollover image objects.

Create a constructor function called *RolloverImage()* as follows:

Class — <code>RolloverImage</code>	
	Constructor function <code>RolloverImage(</code> <code>nameString,</code> <code>selectedIMG,</code> <code>unselectedIMG,</code> <code>selectedTEXT,</code> <code>unselectedTEXT)</code>
Instance Properties	Instance Methods
<code>name</code>	<code>select()</code>
<code>selected</code>	<code>deselect()</code>
<code>unselected</code>	
<code>selectedTEXT</code>	
<code>unselectedTEXT</code>	

Where the instance properties are as follows:

String	name	the name of the image this
Image	selected	the selected version of the
Image	unselected	the unselected version of the
Image	selectedTEXT	the selected version of the
Image	unselectedTEXT	the unselected version of the TEXT IMG (in fact this is the same for

Create the instance methods for the prototype — these should be the methods *select()* and *deselect()*.

Create an object in which to store your RolloverImage objects, call this object *rolloverImages*.

Create, as properties of your object *rolloverImages*, three instances of RolloverImage named *the70s*, *the80s* and *the90s*. This lets you use statements such as:

```
rolloverImages.the70s.select()
```


for onMouseOver events, and so on.

HINT: The HTML table entries should call object methods.

Your HTML table entries should look something like the following:

```
<TD>

<A HREF="the70s.html"
onMouseOver="rolloverImages.the70s.select()" onMouseOu
<IMG NAME="the70s"
SRC="rolloverImages.the70s_UNselected.gif" BORDER=0>
</A>

</TD>
```

You can find a discussion of this activity at the end of the unit.

15.6 Review Questions

1. What is an array?

You can find the answer end of the unit.

2. What is the role of the number 4 in the line:

```
Alert( myArray[4] );
```

You can find the answer end of the unit.

3. How many elements, and what are their indices, in the array ages created in the statement:

```
var ages = new Array(3)
```

You can find the answer end of the unit.

4. Consider this code working with numeric variables:

```
var a1 = 5; var b1 = a1;

document.write( "<p> a1 = " + a1 ); b1 = 7;

document.write("<p> a1 = " + a1 );
```

and this code working with array variables

```
var a2 = new Array( 2 );

a2[0] = 5;
a2[1] = 6;
var b2 = a2;
document.write("<p> a2 = " + a2 ); b2[0] = 7;

document.write("<p> a2 = " + a2 );
```

Browser output when this code is executed is as follows:

```
a1 = 5  
  
a1 = 5  
  
a2 = 5,6  
  
a2 = 7,6
```

Why is it that a2 is changed when b2 is changed, but a1 is not changed when b1 changes? You can find the answer end of the unit.

5. Why does document start with a lower case in the statement: `document.write("sorry, product out of stock");` You can find the answer end of the unit.
6. What output will the following code produce — try to work this out on paper, don't enter the code and try it out yet!

```
<script src="triangle_class.js"></script>  
  
<script>  
  
    var triangle1 = new Triangle(10, 20); var t2 = triangle1;  
  
    t2.width = 5;  
  
    document.write("triangle1 has width: " + triangle1.width );  
  
</script>
```

You can find the answer end of the unit.

7. Are JavaScript objects just data structures? You can find the answer end of the unit.

15.7 Discussion Topics

1. An object is just an array indexed by property names.

You can some contribution to this topic at the end of the unit.

2. Why should programmers try to avoid changing prototypes in the middle of code where objects using the prototype are being created and used?

You can some contribution to this topic at the end of the unit.

15.8 Answers

15.8.1 Discussions of Exercise 1

The answers and discussion of the questions are as follows:

An element of an array is referred to with the array name and the index in square brackets. The first element is that indexed with 0, so it is `weekDays[0]` and so on:

- The first element is: `weekDays[0]`
- The last element: `weekDays[6]`
- The 4th element: `weekDays[4]`
- The value of `weekDays[0]` is "Monday" (we can see this from the figure), and so on.
- The value of the first element: "Monday"
- The value of the 4th element: "Friday"
- The element of the array is referring to the part of `weekDays` that contains the value, i.e. `weekDays[0]` or `weekDays[6]` and so on.
- The element containing String "Monday": `weekDays[2]`
- The element containing String "Saturday": `weekDays[5]`

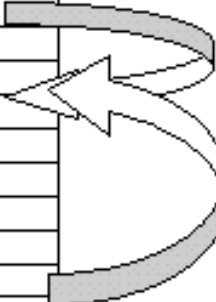
The index of the array element is referring to the number of the `weekDays` element containing the given value; since "Monday" is stored in `weekDays[2]`, the index of this element is 2:

- The index of the element containing String "Monday": 2
- The index of the element containing String "Saturday": 5

15.8.2 Discussions of Exercise 2

After execution of the first two lines, memory looks as follows:

Memory location		value
<code>myArray</code>		002803
	002802	
<code>[0]</code>	002803	6
<code>[1]</code>	002804	3
<code>[2]</code>	002805	5
<code>[3]</code>	002806	1
	002807	
<code>thing2</code>		002803
	002809	

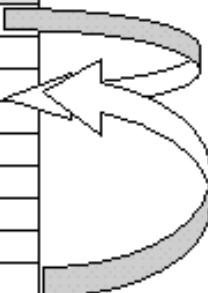


Since `thing2` refers to the same array values as `myArray`, after the three lines:

```
thing2[1] = 27;  
thing2[2] = 19;  
thing2[3] = 77;
```

memory will have been changed as follows:

Memory location		value
myArray		002803
	002802	
[0]	002803	6
[1]	002804	27
[2]	002805	19
[3]	002806	77
	002807	
thing2		002803
	002809	



Therefore the value of *myArray[2]* is now 19.

15.8.3 Discussions of Activity 1

The following code will fulfil the requirements:

```
<html> <script> <!--
var ages = new Array( 21, 14, 33, 66, 11 );
document.write( "<p><b>ages[0] </b>" + ages[0]);
document.write( "<p><b>ages[1] </b>" + ages[1]);
document.write( "<p><b>ages[3] </b>" + ages[3]);

// --> </script> </html>
```

This first line:

```
var ages = new Array( 21, 14, 33, 66, 11 );
```

creates an array, and initialises it with the 5 values. This could have also been achieved as follows:

```
var ages = new Array( 5 ); ages[0] = 21;
ages[1] = 14;
ages[2] = 33;
ages[3] = 66;
ages[4] = 11;
```

The *document.write(...)* lines display the values of each specified array element.

15.8.4 Discussions of Activity 2

To create the primes array we can write the following:

```
var primes = new Array( 6 );
```

To put the values into the array we can write the following:

```
primes[0] = 1;
primes[1] = 2;
primes[2] = 3;
primes[3] = 5;
primes[4] = 7;
primes[5] = 11;
```

Alternatively we could create and initialise the array all in a single line:

```
var primes = new Array( 1, 2, 3, 5, 7, 11 );
```

To iterate through the array we need an index variable (we'll use *i*) and a while loop that will keep looping

while i is less than the value of the length property of the array. By using the length property in our loop condition we make the loop general, in that it will loop through the whole array no matter the length of the array.

```
var i = 0;
while (i < primes.length)
{
    document.write("<p> " + primes[i] );
    i++;
}
```

15.8.5 Discussions of Exercise 3

Constructor functions are named with an initial capital letter, so the following are the names of constructor functions:

- House
- MotorCar

The others may be objects, or methods, or property names:

- door1
- homeHampus

15.8.6 Discussions of Activity 3

An example of an HTML file resulting in the desired animation could be the following:

```
<HTML> <HEAD> <SCRIPT>  <!--

////////////////////////////////////
// function to:
// display image
// increment image number
// increase delay time by 1/4 second
// if not last image then delay and call self
////////////////////////////////////
function animate()
{
    document.noise.src = images[ imageNumber ].src;
    imageNumber++;
    delay += 250;

    if( imageNumber < 4 )
        setTimeout( "animate()", delay );

    // if imageNumber = 4 the animation has finished
    // and this function can terminate
}

////////////////////////////////////
// function to:
// set imageNumber for "taptap" image
// set delay to 1/4 second (250 milliseconds)
// execute the animate() function
//////////////////////////////////// function
startAnimation()
{
    imageNumber = 0;
```

```
delay = 250; animate();  
}
```

```
// set up image array
var images = new Array( 4 );

images[0] = new Image(); images[0].src = "taptap.gif";
images[1] = new Image(); images[1].src =
"knockknock.gif"; images[2] = new Image(); images[2].src
= "BANGBANG.gif"; images[3] = new Image(); images[3].src
= "silence.gif";

// declare the global variables for the imageNumber and

the delay duration
var imageNumber;
var delay;

//--> </SCRIPT> </HEAD>

<BODY>

<IMG NAME="noise" SRC="silence.gif"
onMouseOver="startAnimation()">

</BODY> </HTML>
```

It is important you understand this programme before moving on the later object activities.

15.8.7 Discussions of Activity 4

The object *customer1* can be created as follows:

```
var customer1 = new Object();
```

The properties of this object can be created and assigned as follows:

```
customer1.name = "fred"; customer1.age = 29;
customer1.creditLimit = 500;
```

A *for/in* loop to iterate through and display the properties on separate lines could be:

```
for( var prop in customer1 )
{
  document.write( "<p><b> property </b>" + prop + "<b> has
value </b> " + customer[prop] )
}
```

15.8.8 Discussions of Activity 5

Your constructor function must be named *Rectangle*. It should take two arguments, named along the lines of *newLength* and *newHeight*. The function should assign the value of the first argument to *this.length*, and the value of the second argument to *this.width*:

```
function Rectangle( newLength, newHeight )
{
  this.length = newLength; this.height = newHeight;
}
```

15.8.9 Discussions of Activity 6

Your code should set up the three image arrays:

```
var selected = new Array(3); selected[0] = new Image();
selected[0].src = "the70s_selected.gif"; selected[1] =
new Image(); selected[1].src = "the80s_selected.gif";
selected[2] = new Image(); selected[2].src =
"the90s_selected.gif";

var unselected = new Array(3); unselected[0] = new
Image();
unselected[0].src = "the70s_UNselected.gif";
unselected[1] = new Image(); unselected[1].src =
"the80s_UNselected.gif"; unselected[2] = new Image();
unselected[2].src = "the90s_UNselected.gif";

var selectedTEXT = new Array(3); selectedTEXT[0] = new
Image();
selectedTEXT[0].src = "the70s_selected_TEXT.gif";
selectedTEXT[1] = new Image(); selectedTEXT[1].src =
"the80s_selected_TEXT.gif"; selectedTEXT[2] = new
Image(); selectedTEXT[2].src =
"the90s_selected_TEXT.gif";
```

Your two functions should look as follows:

```
function selectImage(imageNumber)
{
var selectStatement = "document." +
imageName[imageNumber]

+ ".src = selected[imageNumber].src;"; eval(
selectStatement );
document.textArea.src = selectedTEXT[imageNumber].src;
}

function deselectImage(imageNumber)
{
var deselectStatement = "document." +
imageName[imageNumber] + ".src = unselect eval(
deselectStatement );
document.textArea.src = "UNselected_TEXT.gif";
}
```

Your onMouseOver and onMouseOut event one-liners should look similar to the follows:

```
onMouseOver="selectImage(0);" onMouseOut="deselectImage(0)"
```

(where the 0 should be replaced by the appropriate image number);

15.8.10 Discussions of Activity 7

Your objects with Image properties should be set up in a way similar to the following:

```
var selected = new Object; selected.the70s = new Image();
selected.the70s.src = "the70s_selected.gif"; selected.the80s = new
Image(); selected.the80s.src = "the80s_selected.gif";
```



```

selected.the90s = new Image(); selected.the90s.src =
"the90s_selected.gif";

var unselected = new Object(); unselected.the70s = new Image();
unselected.the70s.src = "the70s_UNselected.gif"; unselected.the80s =
new Image(); unselected.the80s.src = "the80s_UNselected.gif";
unselected.the90s = new Image(); unselected.the90s.src =
"the90s_UNselected.gif";

var selectedTEXT = new Object(); selectedTEXT.the70s = new Image();
selectedTEXT.the70s.src = "the70s_selected_TEXT.gif";
selectedTEXT.the80s = new Image(); selectedTEXT.the80s.src =
"the80s_selected_TEXT.gif"; selectedTEXT.the90s = new Image();
selectedTEXT.the90s.src = "the90s_selected_TEXT.gif";

```

Your select and deselect functions can be simplified to the following:

```

function selectImage(imageName)
{
    document[imageName].src = selected[imageName].src;
    document.textArea.src = selectedTEXT[imageName].src;
}

function deselectImage(imageName)
{
    document[imageName].src = unselected[imageName].src;
    document.textArea.src = "UNselected_TEXT.gif";
}

```

See listing music_associative_arrays.html for a complete HTML file using the above code.

15.8.11 Discussions of Activity 8

Your constructor function should look something like the following:

```

function RolloverImage( nameString, selectedIMG,
unselectedIMG, selectedTEX
{
    this.name = nameString; this.selected = new
Image(); this.selected.src = selectedIMG;
this.unselected = new Image(); this.unselected.src
= unselectedIMG; this.selectedTEXT = new Image();
this.selectedTEXT.src = selectedTEXT;
this.unselectedTEXT = new Image();
this.unselectedTEXT.src = unselectedTEXT;
}

```

Your two instance methods, and their addition to the prototype should look similar to the following:

```

function selectImage()
{
    // access the IMG NAME by treating 'document' as an
associative array document[this.name].src =

this.selected.src;

textArea.src = this.selectedTEXT.src;
}

```

```

function deselectImage()
{
    // access the IMG NAME by treating 'document' as
    an associative array
    document[this.name].src = this.unselected.src;
    textArea.src = this.unselectedTEXT.src;
}

RolloverImage.prototype.select = selectImage;
RolloverImage.prototype.deselect = deselectImage;

```

The rolloverImages object, and the three properties that are instances of RolloverImage should look similar to the following:

```

var rolloverImages = new Object();

rolloverImages.the70s = new RolloverImage( "the70s",
"the70s_selected.gif", "the70s_UNselected.gif",
"the70s_selected_TEXT.gif", "UNselected_TEXT.gif");

rolloverImages.the80s = new RolloverImage( "the80s",
"the80s_selected.gif", "the80s_UNselected.gif",
"the80s_selected_TEXT.gif", "UNselected_TEXT.gif");

rolloverImages.the90s = new RolloverImage("the90s",
"the90s_selected.gif", "the90s_UNselected.gif",
"the90s_selected_TEXT.gif", "UNselected_TEXT.gif");

```

Your HTML table should look as follows:

```

<TABLE CELLSPACING=0 CELLPADDING=0 WIDTH=202 BORDER=0>

<TR>
<TD rowspan = 3>
<IMG NAME="textArea" SRC="UNselected_TEXT.gif">
</TD>

<TD>
<A HREF="the70s.html" onMouseOver="rolloverImages.the70s.select()"
onMouseOut="rolloverImages.the70s.deselect()">
<IMG NAME="the70s" SRC="the70s_UNselected.gif" BORDER=0>
</A>
</TD>
</TR>

<TR>
<TD>
<A HREF="the80s.html" onMouseOver="rolloverImages.the80s.select()"
onMouseOut="rolloverImages.the80s.deselect()">
<IMG NAME="the80s" SRC="the80s_UNselected.gif" BORDER=0>
</A>
</TD>
</TR>

<TR>
<TD>
<A HREF="the90s.html" onMouseOver="rolloverImages.the90s.select()"
onMouseOut="rolloverImages.the90s.deselect()">
<IMG NAME="the90s" SRC="the90s_UNselected.gif" BORDER=0></A>

```

```
</TD>
</TR>

</TABLE>
```

15.8.12 Answers to Review Questions

1. An array is a tabular collection of data, accessed via a numeric index. Each element of an array can be a primitive value, or a reference to an object (possibly another Array object).
2. The number 4 is the index of the 5th element in the array (remember the first element is stored in array element `myArray[0]`). The index of an array states the element to which it is being referred.
3. There are 3 elements in this array. They are:

```
ages[0] ages[1] ages[2]
```

(remember, arrays are indexed with 0 as the first element index, not 1.)

4. Simple numeric variables like `a1` and `b1` store values, and when the value of variable `a1` is assigned to the value of `b1`, a copy of the value 5 is placed into variable `b1`. When `b1` changes the value it stores this has no effect on the number 5 stored elsewhere in memory in variable `a1`.

Array variables contain a reference to the location in memory where the array elements are stored. Copying an array variable means copying the reference, so in the above code variable `b2` is referring to the same array as `a2`, therefore changing the array elements referred to by `b2` will also change the elements referred to by `a2`.

5. In this statement `document` is a variable that contains a reference to the document instance of the `Document` constructor. When a browser loads and interprets an HTML file containing JavaScript, it creates a single instance based on `Document`, called `document`. This object is made available as a property of the global object to JavaScript programmers, so that they can refer to the properties and methods of browser `document`.
6. The answer is that `triangle1.width` has a value of 5. This is because when `t2` is assigned the value of `triangle1`, it is assigned the reference to `triangle1`. So when a change is made to the object referred to by `t2`, this is also the object referred to by `triangle1`.

Browser output is as follows:

```
triangle1 has width: 5
```

7. Yes they are. In fact more so that for many other object languages, since methods are just another form of data.

15.8.13 Contribution to Discussion Topics

1. Yes this statement is true — associative arrays and objects are the same thing. Looking at objects as associative arrays, however, provides a potentially more flexible and dynamic way to access properties and methods.

However, associative arrays should be distinguished from numerically indexed arrays, the latter having an explicit order to their elements.

2. When a prototype changes, the changes are inherited by all objects sharing the prototype at that point in time. Therefore if an object was previously created and used, and then its prototype changed, performing the same operations on the object will not necessarily result in the same behaviour. This could lead to inconsistent software system behaviour, and might be a very difficult bug to locate and repair.

Few circumstances exist where one might desire such changing behaviour from objects in the middle of

execution of a programme.