



Advance Internet Programming

Lecture: Khim Rada

1- What is java Script ?

JavaScript is a scripting language that enables web developers/designers to build more functional and interactive websites.

JavaScript is the most popular scripting language in the world. It is the standard language used in web pages, but it is also widely used by desktop apps, mobile phone apps, and internet servers.

2- Why using JavaScript?

Common uses of JavaScript include:

- Alert messages
- Popup windows
- Dynamic dropdown menus
- Form validation
- Displaying date/time

3- How JavaScript can do in webpage?

- ❖ JavaScript can read and change the content of HTML elements.
- ❖ JavaScript can read and change the style of HTML elements.
- ❖ JavaScript can be used to validate data, like validating forms input.
- ❖ JavaScript can be set to execute when something happens, like when a user clicks on an HTML element.

4- JavaScript vs. Java

JavaScript and Java are two completely different languages, in both concept and design.


❖ **Java** (developed by Sun Microsystems) is a more complex programming language in the same category as C and C++.



❖ **JavaScript** = ECMAScript (*European Computer Manufacturers Association*)

ECMA-262 is the official JavaScript standard.

JavaScript was invented by Brendan Eich at Netscape, and appeared for the first time in Netscape Navigator (a no longer existing web browser) in 1995.




❖ JavaScript is an object-based, client-side scripting language that you can use to make Web pages more dynamic. To make sense of such a definition, let's look at its important parts one by one.

5- Object Based


Object based means that JavaScript can use items called objects. However, the objects are not class based (meaning no distinction is made between a class and an instance); instead, they are just general objects.

6- Server-Side Languages

A **server-side** language needs to get information from the Web page or the Web browser, send it to a program that is run on the host's server, and then send the information back to the browser.




Therefore, an intermediate step must send and retrieve information from the server before the results of the program are seen in the browser.



A **server-side** language often gives the programmer options that a client-side language doesn't have, such as saving information on the Web server for later use, or using the new information to update a Web page and save the updates.

7- Client-Side Languages

A **client-side** language is run directly through the client being used by the viewer. In the case of JavaScript, the client is a Web browser. Therefore, JavaScript is run directly in the Web browser and doesn't need to go through the extra step of sending and retrieving information from the Web server.



A **client-side** language can also access special features of a browser window that may not be accessible with a server-side language. However, a client-side language lacks the ability to save files or updates to files on a Web server like a server-side language can.

8- The `< script >` Element

The primary method of inserting JavaScript into an HTML page is via the `< script >.....</Script` element. A JavaScript can be put in the `<body>` and in the `<head>` section of an HTML page. For example:



```
<head>
```

```
    <title>My JavaScript page</title>
```

```
    <script language="Javascript" type="text/javascript">
```

```
        alert("Welcome to my JavaScript page!")
```

```
    </script>
```

```
</head>
```

```
<body bgcolor="#FFFFFF">
```


```
<h2>This script pops up a message box for the  
user.</h2>
```

```
</body>
```

9- JavaScript Statements

JavaScript statements are "commands" to the browser. The purpose of the statements is to tell the browser what to do. Example:

```
document.write("Good Morning !");
```

```
<html>
```

```
<body>
```

```
<script type="text/javascript">
```

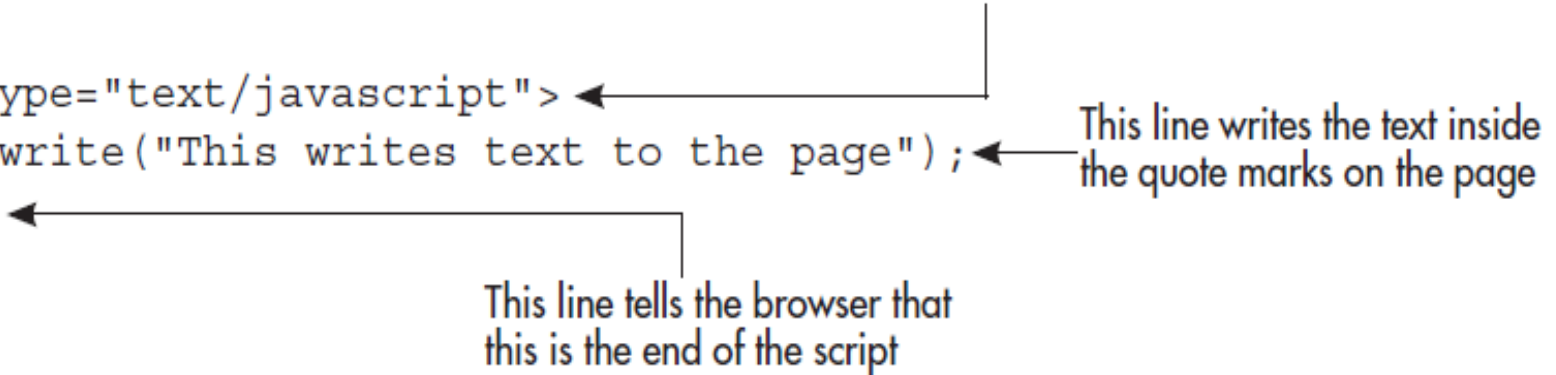
```
document.write("This writes text to the page");
```

```
</script>
```

```
</body>
```

```
</html>
```

This tag tells the browser
that JavaScript follows




This line writes the text inside
the quote marks on the page

This line tells the browser that
this is the end of the script



Data type

A data type in a programming language is a set of data with values having predefined characteristics. Examples of data types are: String, Number, Boolean, Array, Object, Null, Undefined . Example:



```
var message = "some string";
```

```
alert(typeof message); //"string"
```

```
alert(typeof(message)); //"string"
```

```
alert(typeof 95);          //"number"
```

1- Number

JavaScript has only one type of number. Numbers can be with, or without decimals. Example:



```
<body>
```

```
<script>
```

```
var x=1;
```

```
var y=2; var z=x+y;
```

```
document.write(x + "<br>");
```

```
document.write(y + "<br>");
```

```
document.write(z + "<br>");
```

```
</script>
```

```
</body>
```

2- String

A string is a variable which stores a series of characters like "Thyda".

A string can be any text inside quotes. You can use simple or double quotes:



```
<body>
```

```
<script>
```

```
var Ch1="who is she?";
```

```
var Ch2="She is called 'Thyda';
```

```
var Ch3='Hi! "Thyda";
```

```
document.write(Ch1 + "<br>")
```

```
document.write(Ch2 + "<br>")
```

```
document.write(Ch3 + "<br>")
```

```
</script>
```

```
</body>
```




The result is:

who is she?

She is called 'Thyda'

Hi! "Thyda"


3- Boolean

Booleans are often used in conditional testing. Booleans can have only two values: true or false. Fro example:

```
var found = true;  
var lost = false;
```

4- Array

An array is a way of storing data of similar types for easy access later in a script. In JavaScript, an array is basically a user-defined object that is typically accessed in a different way than other objects are accessed. It uses a single variable name to store multiple values.



```
var Books=new Array();  
Books[0]="Head 1";  
Books[1]="Head 2";  
Books[2]="Head 3";
```

Variables

ECMAScript variables are loosely typed, meaning that a variable can hold any type of data. Every variable is simply a named placeholder for a value.

1- Declaration Viriabies

To define a variable, use the **var** operator (note that **var** is a keyword) followed by the variable name. like this:

```
var coolcar;
```

```
var message;
```

```
Var x;
```

```
Var Result;
```

2- Assign Variable

To assign a value to a variable, you use the JavaScript assignment operator, which is the equal to (=) symbol. If you want to declare a variable and assign a value to it on the same line, use this format:

var variablename=variablevalue;



For example:

var carname= Toyota;

var message=How are you !;

Var x=12;

Var Result=30;


3- Making a Call to a Variable

The following code shows how to write the value of a variable to a Web page using the `document.write()` method:


```
<script language="JavaScript">  
var mybook="javascriptbook";  
document.write(mybook);  
</script>
```

4- Adding Variables to Text Strings

The preceding code just prints the value of the variable in the browser. If you want that variable to print along with some other text in a string, the `document.write()` command becomes more complex.



The text string needs quotes around it if it has not been defined as a variable, and the variable needs to be on its own. You use the addition operator (+) to add the value of the variable to the string, as shown in this example:



```
<script type="text/javascript">  
var mybook="javascriptbook";  
document.write("I am reading "+mybook);  
</script>
```



Controls Statement

ECMA - 262 describes several statements (also called *flow - control statements*). Statements can be simple, such as telling a function to exit, or complicated, such as specifying a number of commands to be executed repeatedly.


1- Making Decision

Very often when you write code, you want to perform different actions for different decisions. You can use conditional statements in your code to do this. In JavaScript we have the following conditional statements:


2- The if Statement

One of the most frequently used statements in most programming languages is the if statement. The if statement has the following syntax:


```
if (condition)
    statement1
else
    statement2
```


The condition can be any expression; it doesn't even have to evaluate to an actual Boolean value. ECMAScript automatically converts the result of the expression into a Boolean by calling the Boolean() casting function on it. If the condition evaluates to true, statement1 is executed; if the condition



evaluates to false , statement2 is executed. Each of the statements can be either a single line or a code block (a group of code lines enclosed within braces). Consider this example:



```
if (i > 25)
  alert("Greater than 25."); //one-line
  statement
else {
  alert("Less than or equal to 25.");
  //block statement
}
```



```
var mynum=1;  
var mymessage;  
    if (mynum==1)  
        mymessage="You win!";  
    else  
        mymessage="Sorry! Try again!";
```

3- The Else IF

The If Else If statement is more powerful than the previous. This is because you can specify many different outputs based on many different conditions - all within the one statement. You can also end with an "else" to specify what to do if none of the conditions are true.



Syntax:

if (condition1)


statement1

else if (condition2)

statement2

else


statement3




```
if (time<10)
{
    x="Good morning";
}
else if (time<20)
{
    x="Good day";
}
else
{
    x="Good evening";
}
```

4- Switch


The switch statement allows you to take a single variable value and execute a different block of code based on the value of the variable. If you wish to check for a number of different values, this can be an easier method than the use of a set of nested if/else statements.



Now, you need to see the general syntax for a full switch statement. The following code is an example of how a switch statement looks:



```
switch (expression) {  
    case value1: statement1  
    break;  
    case value2: statement2  
    break;  
    case value3: statement3  
    break;  
    case value4: statement4  
    break;  
    default: other statement  
}
```




```
switch (i) {  
    case 25:  
        alert("25");  
    break;  
    case 35:  
        alert("35");  
    break;  
    case 45:  
        alert("45");  
    break;  
    default:  
        alert("Other");  
}
```

```
switch ("hello world") {  
    case "hello" + " world":  
        alert("Greeting was found.");  
    break;  
    Case "goodbye":  
        alert("Closing was found.");  
    break;  
    default:  
        alert("Unexpected message was  
            found.");  
}
```



Loop

A loop is a block of code that allows you to repeat a section of code a certain number of times, perhaps changing certain variable values each time the code is executed.



Loops are useful because they allow you to repeat lines of code without retyping them or using cut and paste in your text editor. This not only saves you the time and trouble of repeatedly typing the same lines of code, but also avoids typing errors in the repeated lines.

1- Using Loops in JavaScript

In order to see how loops can really be helpful to you, you need to take a look at the different loop structures that you can use in JavaScript. The loop structures covered in this section are the *for*, *while*, and *do while* loops.

2- For Loop

The for statement is also a pretest loop with the added capabilities of variable initialization before entering the loop and defining post loop code to be executed. Here 's the syntax:

```
for (var variablename=number; variablename  
expression ; variablename increment)
```



The following example code shows how this can be done using a for loop:

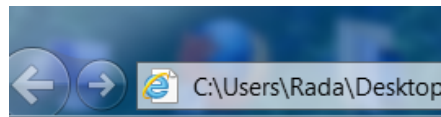
```
var count = 10;  
for (var i=0; i < count; i++)  
{  
  alert(i);  
}
```



This for loop is the same as the following:

```
var count = 10;  
var i;  
for (i=0; i < count; i++)  
{  
  alert(i);  
}
```

```
document.write("Get some repeated text.<br />");  
for (var count=1;count<11;count++) {  
document.write("I am part of a loop!<br />");  
}
```



Get some repeated text.
I am part of a loop!
I am part of a loop!
I am part of a loop!
I am part of a loop!
I am part of a loop!
I am part of a loop!
I am part of a loop!
I am part of a loop!
I am part of a loop!
I am part of a loop!

3- While Loop


The while statement is a pretest loop. This means the escape condition is evaluated before the code inside the loop has been executed. Because of this, it is possible that the body of the loop is never executed. Here 's the syntax:

while(expression) statement



And here 's an example of its usage:

```
var i = 0;  
while (i < 10)  
{  
    i += 2;  
}
```




```
var count=1;
while (count<6) {
    JavaScript Code Here
    count++;
}
```

← A variable is assigned a value to count the loop

← The while statement begins with a comparison

← The count variable is adjusted so that you do not have an endless loop



```
<script>
function myFunction(){
var x="", i=0;
while (i<5) {
x=x + "The number is " + i + "<br />"; i++;
}
document.getElementById("demo").innerHTML
=x;
}
</script>
```




Result is :

The number is 0

The number is 1

The number is 2

The number is 3

The number is 4

4- Do While Loop

The do - while statement is a post - test loop, meaning that the escape condition is evaluated only after the code inside the loop has been executed. The body of the loop is always executed at least once before the expression is evaluated. Here ' s the syntax:



do

{

statement

}

while (expression);



And here 's an example of its usage:

```
var i = 0;  
do  
{  
    i += 2;  
}  
while (i < 10);
```



do

{

x=x + "The number is " + i + "
";

i++;

}

while (i<5);



Result is :

The number is 0

The number is 1

The number is 2

The number is 3

The number is 4




Functions

Functions are the core of any language, because they allow the encapsulation of statements that can be run anywhere and at any time. Functions in ECMAScript are declared using the **function** keyword, followed by a set of arguments and then the body of the function.

1- Structuring Functions

Now that you understand what functions are and why you want to use them, you need to learn how to structure them in your scripts. A function needs to be declared with its name and its code. There are also some optional additions you can use to make functions even more useful.




You can import one or more variables into the function, which are called *parameters*. You can also return a value to the main script from the function using the *return* statement. You will start by looking at how the function begins.

2- Declaring Functions

On the first line of a function, you declare it as a function, name it, and indicate whether it accepts any parameters. To declare a function, you use the reserved word `function`, followed by its name, and then a set of parentheses:

`function functionname()`




For example, to name your function **myfunction** and indicate that it does not use any parameters, the first line looks like this:


```
function myfunction()
```

3- Defining the Code for Functions

Curly brackets (`{ }`) surround the code inside the function. The opening curly bracket marks the beginning of the function's code; then comes the code; and, finally, the closing curly bracket marks the end of the function, in this format:



```
function myfunction()  
{  
  JavaScript code here  
}
```



The browser will execute all of the code inside the curly brackets when the function is called (as you will learn later). When the browser gets to the closing curly bracket, it knows the function has ended. The browser will move to the next line of code or continue whatever it was doing before the function was called.



For example:

```
function reallycool()  
{  
  Var a= 7; var b = 5; var c = 10-;  
}
```


4- Naming Functions

❑ Using Case in Function Names

Function names are case sensitive, just like variable names. This means that **myfunction**, **MYFUNCTION**, and **MyFunction** represent different functions. Remember that you need to call your functions using the same letter cases as you used in their declarations.



❑ Using Allowed Characters and Avoiding Reserved Words


The characters that are allowed for function names are the same as those you can use for variable names:

- The function name must begin with a letter or an underscore character (_).
- The function name cannot contain any spaces.

5- Adding Parameters to Functions


Parameters are used to allow a function to import one or more values from somewhere outside the function. Parameters are set on the first line of the function inside the set of parentheses, in this format:

Function *functionname(variable1,variable2)*




Any value brought in as a parameter becomes a variable within the function, using the name you give it inside the parentheses.

For example, here is how you would define a function **myfunction** with the parameters (variables) para1 and para2:



```
function myfunction(para1,para2) {  
JavaScript code here  
}
```

Notice that in JavaScript, you do not use the var keyword when you set the parameters for a function. JavaScript declares the variables automatically



when they are set as parameters to a function, so the **var** keyword is not used here. For example, a line like this one is invalid:


Incorrect (not used)




function reallycool(var para1, var para2)

6- Using Function Parameter Values

When you assign parameters to a function, you can use them like any other variables. For example, you could give the para1 variable value to another variable by using the assignment operator, as in this example:



```
function myfunction(para1,para2) {  
var para3=para1;  
}
```





This assigns the value of the **para1** parameter to a variable named **para3**. Instead of assigning its value to another variable, you could just use the **para1** parameter in the function, as in this example:


```
function myfunction(para1,para2) {  
  document.write("My work is a "+para1);  
}
```

7- Using Multiple Parameters

You may have noticed that the previous example had two parameters but used only one parameter. A function can have as few or as many parameters as you wish. When you assign multiple function parameters, the function doesn't need to use all of them. It can use one parameter, a few, or none.




The only rule is that if you have more than one parameter, you need to separate each parameter with a comma, so that the browser knows what to do. In the previous example, the second parameter was not used. Here is how you could change the function to use both parameters:



```
function myfunction(para1,para2) {  
  document.write("My work is a  
"+para1+" and I rest at "+para2);  
}
```


8- Adding Return Statements to Functions

A return statement is used to be sure that a function returns a specific value to the main script, to be used in the main script. You place the return statement as the last line of the function before the closing curly bracket and end it with a semicolon.




Most often, the value returned is the value of a variable, using the following format:

return variablename;



Example: you want to write a function that returns the result of adding two strings together. You could use a return statement, as in this:

```
function get_added_text() {  
  var textpart1="This is ";  
  var textpart2="fun!";  
  var added_text=textpart1+textpart2;  
  return added_text;
```



In addition to returning a variable value, you can return a simple value or even nothing. All of the following would be valid return statements:

return "This is work"; returns a string value

return 42; returns a numeric value


return true; returns a Boolean value

return null; returns a null value

return; returns a nothing

9- Calling Functions in Your Scripts

Now that you know how the function itself works, you need to learn how to call a function in your script.




A call to a function in JavaScript is simply the function name along with the set of parentheses (with or without parameters between the opening and closing parentheses), ending with a semicolon, like a normal JavaScript statement:

functionname();

10- Calling a Function from Another Function

Calling a function within another function can be a useful way to organize the sequence in which your events will occur. Usually, the function is placed inside another function that has a larger task to finish.



When you place a function call within a function, you should define the function that will be called before you define the function that calls it, per the earlier suggestion that a function should be defined before it is called.




Here is an example of two functions,
where the second function calls the
first one:

```
function update_alert(){  
    window.alert("Welcome! This site is updated  
daily!");  
}  
  
    function call_alert() {  
        update_alert();  
    }  
  
    call_alert();
```



Operator Types


An *operator* is a symbol or word in JavaScript that performs some sort of calculation, comparison, or assignment on one or more values. In some cases, an operator provides a shortcut to shorten the code so that you have less to type.



Common calculations include finding the sum of two numbers, combining two strings, or dividing two numbers. Some common comparisons might be to find out if two values are equal or to see if one value is greater than the other. JavaScript uses several different types of operators:


1- The Mathematical Operators

For a mathematical calculation, you use a mathematical operator. The values that you use can be any sort of values you like. For instance, you could use two variables, two numbers, or a variable and a number. A few of these operators are able to perform a task on a single variable's value.




You can also use the addition operator when one of the values is a variable, as in this example:


```
var part2="To be countinue."  
window.alert("I begin and "+part2);
```



The mathematical operators and their functions are summarized in The following Table:



Operator	Symbol	Function
Addition	+	Adds two values
Subtraction	−	Subtracts one value from another
Multiplication	*	Multiplies two values
Division	/	Divides one value by another
Modulus	%	Divides one value by another and returns the remainder
Increment	++	Shortcut to add 1 to a single number
Decrement	--	Shortcut to subtract 1 from a single number
Unary negation	−	Makes a positive negative or a negative positive



For example: The subtraction operator is used to subtract the value on its right side from the value on its left side, as in mathematics.

```
var num1=10;
```


```
var num2=3;
```

```
var theresult=num1-num2;
```

```
window.alert(theresult);
```

2- The Assignment Operators


Assignment operators assign a value to a variable. They do not compare two items, nor do they perform logical tests. The next slide you see how the basic assignment operator, the single equal sign (`=`), is used to give an initial value or a new value to a variable, as in this example:



- ❖ `var population=4500;`
- ❖ `var mytime=0;`

Please the following table for more:

Operator	Symbol	Function
Assignment	=	Assigns the value on the right side of the operator to a variable
Add and assign	+=	Adds the value on the right side of the operator to the variable on the left side, and then assigns the new value to the variable
Subtract and assign	-=	Subtracts the value on the right side of the operator from the variable on the left side, and then assigns the new value to the variable
Multiply and assign	*=	Multiplies the value on the right side of the operator by the variable on the left side, and then assigns the new value to the variable
Divide and assign	/=	Divides the variable on the left side of the operator by the value on the right side, and then assigns the new value to the variable
Modulus and assign	%=	Takes the integer remainder of dividing the variable on the left side by the value on the right side, and assigns the new value to the variable




For example: The `*=` operator multiplies the value on the right side of the operator by the variable on the left side. The result is then assigned to the variable.

```
var mymoney=1000;  
var multby=2;  
mymoney*=multby;
```


3-The Comparison Operators

Comparison operators are often used with conditional statements and loops in order to perform actions only when a certain condition is met. Since these operators compare two values, they return a value of either true or false, depending on the values on either side of the operator.




This table summarizes the comparison operators, which are discussed in more detail in the following:

Operator	Symbol	Function
Is equal to	==	Returns true if the values on both sides of the operator are equal to each other
Is not equal to	!=	Returns true if the values on both sides of the operator are not equal to each other
Is greater than	>	Returns true if the value on the left side of the operator is greater than the value on the right side
Is less than	<	Returns true if the value on the left side of the operator is less than the value on the right side
Is greater than or equal to	>=	Returns true if the value on the left side of the operator is greater than or equal to the value on the right side
Is less than or equal to	<=	Returns true if the value on the left side of the operator is less than or equal to the value on the right side
Strict is equal to	===	Returns true if the values on both sides are equal and of the same type
Strict is not equal to	!==	Returns true if the values on both sides are not equal or not of the same type



For example: The Is-Not-Equal(!=) operator is the opposite of the == operator. Instead of returning true when the values on each side of the operator are equal, the != operator returns true when the values on each side of it are *not* equal.




The only way this operator returns a false value is if it finds that the values on both sides of the operator are equal. The following table shows:

Comparison	Return Value	Reason
<code>4!=3</code>	True	4 and 3 are not equal numbers
<code>"Cool"!="cool"</code>	True	Strings do not have the same capitalization, so they are not equal
<code>4!=4</code>	False	4 is equal to 4
<code>"cool"!="cool"</code>	False	Strings are exactly alike, so they are equal

4- The Logical Operators


The three logical operators allow you to compare two conditional statements to see if one or both of the statements is true and to proceed accordingly.




The logical operators can be useful if you want to check on more than one condition at a time and use the results. Like the comparison operators, the logical operators return either true or false, depending on the values on either side of the operator.



Operator	Symbol	Function
AND	&&	Returns true if the statements on both sides of the operator are true
OR		Returns true if a statement on either side of the operator is true
NOT	!	Returns true if the statement to the right side of the operator is not true



For example : The OR Operator (||)
The logical operator OR returns true if the comparison on either side of the operator returns true. So, for this to return true, only one of the statements on one side needs to evaluate to true.




To return false, the comparisons on both sides of the operator must return false. The following table shows some examples of comparisons using the OR operator.

Statement	Return Value	Reason
$(2==2) \parallel (3>5)$	True	Comparison on the left is true
$(5>17) \parallel (4!=9)$	True	Comparison on the right is true
$(3==3) \parallel (7<9)$	True	Both comparisons are true
$(4<3) \parallel (2==1)$	False	Both comparisons are false
$(3!=3) \parallel (4>=8)$	False	Both comparisons are false




Object-Oriented Programming

Like most modern computer languages, JavaScript highly relies on the theory of class for its functionality. This also means that you must make sure you understand the concept of class and object to take advantage of JavaScript.

- 
- ❖ If you are coming from C++ or Pascal, you must be careful with the way JavaScript handles classes.
 - ❖ It is somewhat different and, because JavaScript is referred to as "loose", this flexibility doesn't make sense to a C++ or Pascal programmer.

A Class

- ❖ In the first lesson, we had to introduction to class in order to explain what an object was. Let's consider a (human or animal) hand.



❖ To describe it, you would use characteristics such as left or right, the number of fingers, whether the nails are trimmed or not. These are the words used to describe a hand. You can create a list to represent this description as follows:




Hand

[

Left or Right
Number of Fingers
Nails are Trimmed
Hand Is Opened

]



in computer programming, the list of items used to describe something is called a class. In our example, the Hand is a class.


This Member

Every time you create a new class, there is a member automatically created for it. This member is called `this` and it represents the class itself. Furthermore, the `this` member has direct access to all properties of its parent class.

Class Creation


A constructor is a special function that is used to create a class. The name of this function also becomes the name of the class. Here is an example:

```
function Hand()  
{  
    }  
}
```




In this case, Hand becomes the name of the new class. To specify the properties of a class, you can list them in the parentheses of the constructor. Here is an example:


```
function Hand(side, fingers, nailsState,  
handOpenState) {  
}
```



Using this in a constructor, you can create a list of the properties of the class. To do this, type this, followed by the period operator, followed by the name of the new property you want to create. To complete the creation of the property, you can assign it to an argument passed in the parentheses of the constructor. Here is an example:




```
function Hand(side, fingers, nailsState,  
handOpenState) {  
  this.LeftOrRight = side;  
  this.NumberOfFingers = fingers;  
  this.NailsAreTrimmed = nailsState;  
  this.HandIsOpened = handOpenState;  
}
```



These assignments actually create the class and gives it meaning. For this example, we now have a class called Hand and that has the following four properties: LeftOrRight, NumberOfFingers, NailsAreTrimmed, and HandIsOpened. [See example](#)

An Object


Once you know what is necessary to describe something, you can provide a value for each member of its list. Because the class is only a list of items, if you want to describe something that is based on that list, you can first name it. For example, you can name a hand HumanHand. Another hand can be DogHand, and so on.



This means that, before describing a hand to a person, you must specify a name for that hand.

Instantiating an Object: The new Operator

Like the variables we used in previous lessons, before using a class, you must declare it. This is also referred to as instantiating a class or creating an object. To create an object, type the `var` keyword, followed by the name you want to use for the object, followed by the assignment operator, followed by the `new` operator, followed




by the name of the class and its parentheses. In the parentheses of the class, you can provide a value for each argument. Here is an example:

```
var dogtHand = new Hand("Right", 5,  
                        false, true)
```


[See example 1](#)

Methods

While the properties are used to describe or characterize an object, some objects may be able to perform actions. For example, you can ask the hand to open. The action of opening a hand is called a method. You can also ask a hand to close itself. This action also is referred to as a method.




A method of a class is created like a function. Since a function cannot be created inside of another function, in this case inside of the class' constructor, when defining the function, you can indicate that it belongs to the class with the use of the `this` object and the access to the defined member(s) of the class.



After creating the method, to add it to your class, assign it to the desired member name using the this object in the constructor of the class as we did previously. [See example2](#)

Passing an Object as Argument


Like a normal value, an object can be passed to a function. Only the name of the object is passed to the function. This means that you must know what the class on which the object is based looks like.



When the object has been passed as argument, you can use its name in the body of the function and access any of its properties and methods, if any, as you see fit. [See example3](#)

Returning an Object From a Function

Like a normal value, you can declare a local variable of an object in a function, process that object, and then return it when the function exists.



When a function returns an object, when calling that function, you can assign it to variable declared from the same type of object that function returns. [See example](#)



Built-In Classes

When the user opens a web page, the object that displays it is called a browser. Internally, the browser is primarily a window application. To access this window, JavaScript provides a class called window.


The Window

To create a new window object, which corresponds to opening a new instance of the browser, you can call the `window.open()` method. Its syntax is:


```
window.open("URL", "name", "attributes")
```


Attributes of a Window


The "name" parameter lets you assign a target name to the newly opened window for links on your page to target. The 3rd parameter, "attributes", is where you control the look of the opened window with. Here are the possible values this parameter accepts.




The **Attribute** of a window are set in the third argument of the `open()` method. Each **Attribute** has a specify name. To set more than one **Attribute**, separate them with a comma.



By default, when you create a new window, it displays normally with a **menu**, a **toolbar**, and the rest. Such a window may use the maximum area of the window. Otherwise, when creating it, you can specify the dimensions you want the new window to have.



The distance from the left to the right borders of the browser is represented by the **width** attribute. To control this aspect, set the desired value, as a natural number to the **width** attribute.



The distance from the top to the bottom borders of the window is represented by the **height** attribute. To control it, set the desired value to this attribute.


[See the table of attributes](#)



Lets see how these attributes are passed in to affect the look of a pop up window:

```
<form> <input type="button" value="Click  
here to see"  
onclick="window.open('page2.htm',  
'win1','width=300,height=200,menubar');"/>  
</form>
```


[See example](#) [Result](#)



Here is another example with no attributes turned on, except the size changed:

```
<input type="button" value="Click here to see"
onclick="window.open('L6.htm',
'win1','width=600,height=500,status=0');"/>
```

[See example](#) [result](#)




Both of the above windows are non resizable. This brings us to an important point. Consider the below:

```
<input type="button" value="Click here to  
see"> onclick="window.open('page2.htm',  
'win1','width=600,height=500,status,resizable  

```

[See example](#) [Result](#)



When this method is called without an argument, it creates an empty window that can be a copy of the window that created it:

```
<input type="button" name="btnNewWindow"  
value="Create New Window"  
onClick="window.open();">
```

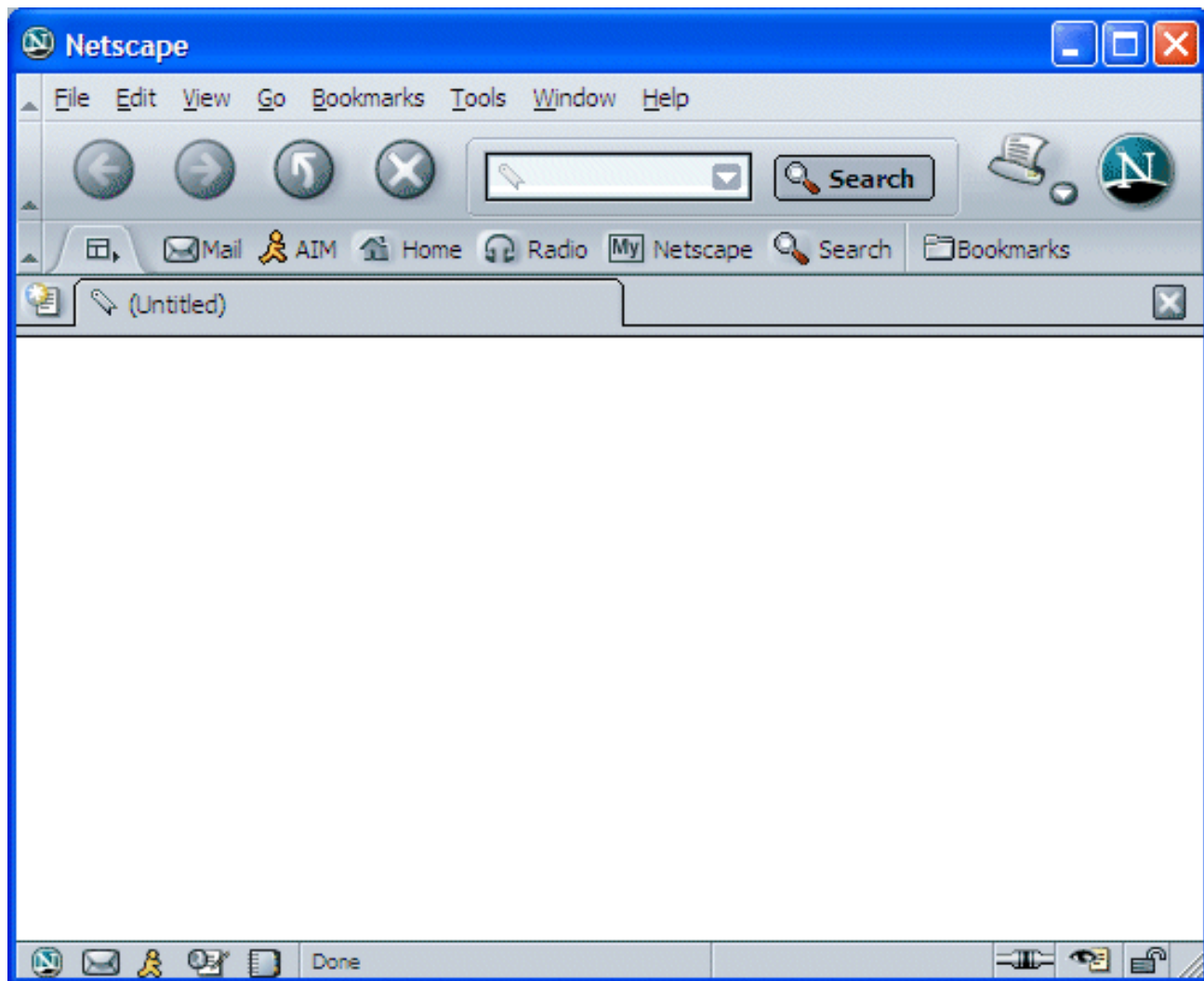
Methods of a Window

To support the creation of a new window, we saw that you can call the `open()` method of the window class. After using a window, a user can close it. To programmatically close a window, you can call the `close()` method.

The Document Class

The main area of the browser is called a document. It may be blank if you created or decided to display it as an empty window:

`Window.open();` see [example](#), [result](#).



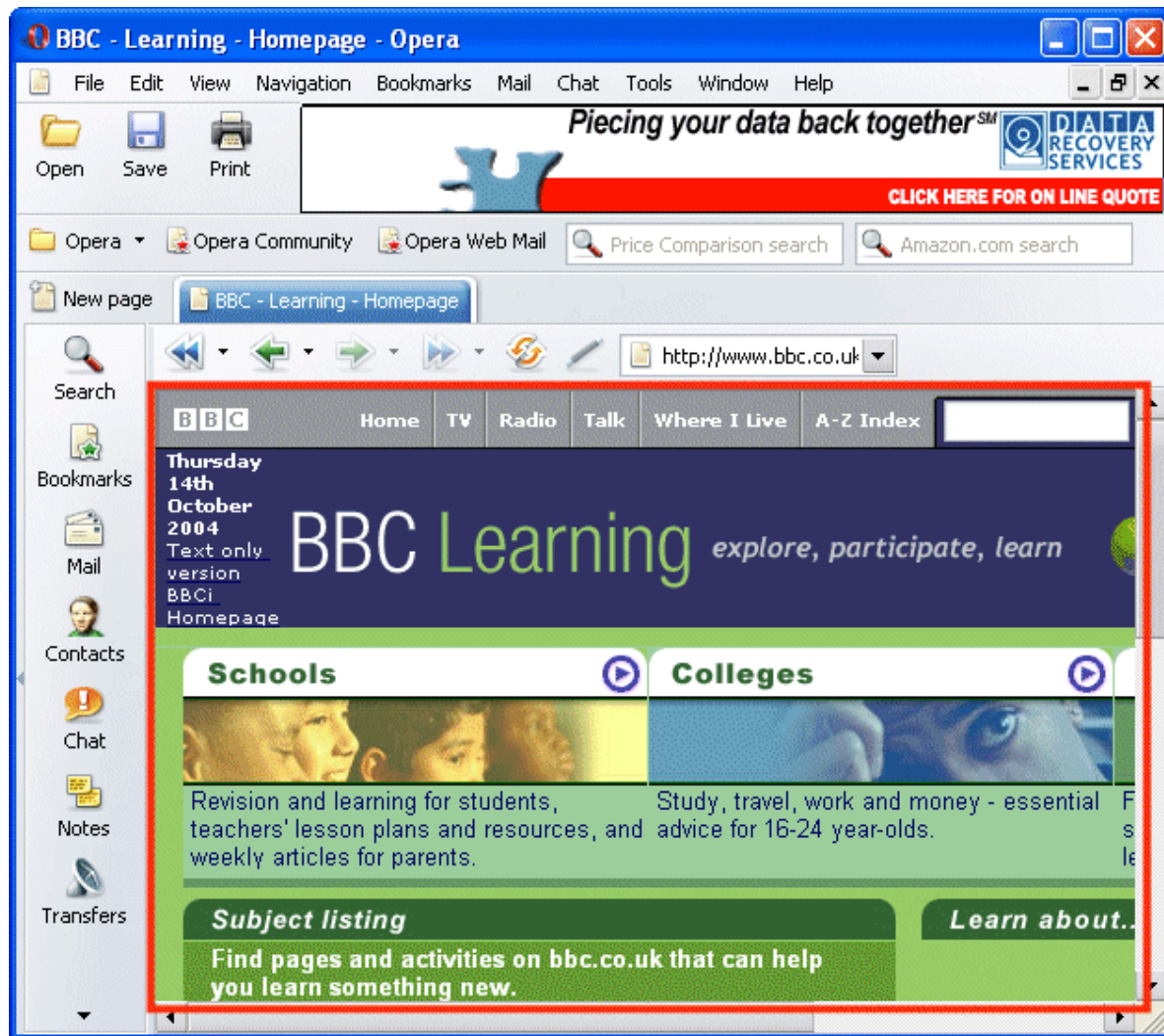


It may also be filled with the contents of its URL.

The area that displays the content of web page is represented by the document class.


```
Window.open('Page1.html');
```

See [example](#), [result](#).




The Document Properties


In its title bar, the browser displays the title of the document. The title of the document is primarily created in the head section of the page. To get the title of a document, access its title property. Here is an [example](#) , [result](#)




In previous lessons, we created forms to explore those lessons. Whenever you create a form in a web page, that form becomes a property of the document object. To access that form, simply use the period operator applied to the **document** class.



In the body of the document, everything may have been created by the initiator of the page. This includes all the colors used by the various sections of the document. Still, if you want, you can change the background color of a page using the **bgColor** property of the **document** class.



To set this color, you can assign the name or the hexadecimal format of the color. Here is an example that changes the background color of the page to pink (fuchsia).see [example](#), [result](#)



To control the general font color of the text in the page, use the **fgColor** property by assigning it either the name or the hexadecimal format of the desired color. See [example](#) , [result](#).

Review

- ❖ Consider1
- ❖ Consider2
- ❖ Consider3 , result
- ❖ Test




Working with the DOM

In many cases, working with the DOM is fairly straightforward, making it easy to re - create with JavaScript what normally would be created using HTML code. There are, however, times when using the DOM is not as simple as it may appear.

1- Dynamic Scripts

The `< script >` element is used to insert JavaScript code into the page, using either the `src` attribute to include an external file or by including text inside the element itself. Dynamic scripts are those that don't exist when the page is loaded but are included later by using the DOM.




As with the HTML element, there are two ways to do this: pulling in an external file or inserting text directly.

Consider the following:

`< script >` element:

`< script type="text/javascript" src="client.js" >`

`< /script >`




The DOM code to create this node is as follows:

```
function loadScript(url){  
  var script = document.createElement("script");  
  script.type = "text/javascript";  
  script.src = url;  
  document.body.appendChild(script);  
}
```

2- Dynamic Styles

CSS styles are included in HTML pages using one of two elements. The `< link >` element is used to include CSS from an external file, whereas the `< style >` element is used to specify inline styles. Similar to dynamic scripts, dynamic styles don't exist on the page when it is loaded initially; rather, they are added after the page has been loaded.



```
var link = document.createElement("link");  
link.rel = "stylesheet";  
link.type = "text/css";  
link.href = "styles.css";  
var head =  
document.getElementsByTagName("head")[0];  
head.appendChild(link);
```

3- Manipulating Tables

Suppose you want to create the following HTML table using the DOM:

//create the table

```
var table = document.createElement("table");  
table.border = 1;  
table.width = "100%";
```



//create the tbody

```
var tbody = document.createElement("tbody");  
table.appendChild(tbody);
```




//create the first row


```
var row1 = document.createElement("tr");  
tbody.appendChild(row1);  
var cell1_1 = document.createElement("td");  
cell1_1.appendChild(document.createTextNode("Cell  
1,1"));  
row1.appendChild(cell1_1);  
var cell2_1 = document.createElement("td");  
cell2_1.appendChild(document.createTextNode("Cell  
2,1"));  
row1.appendChild(cell2_1);
```

4- Using Node Lists


Understanding a NodeList object and its relatives, NamedNodeMap and HTMLCollection , is critical to a good understanding of the DOM as a whole.



Each of these collections is considered “live,” which is to say that they are updated when the document structure changes such that they are always current with the most accurate information.



In reality, all NodeList objects are queries that are run against the DOM document whenever they are accessed. For instance, the following results in an infinite loop:




```
var divs =  
document.getElementsByTagName("div");  
  
for (var i=0; i < divs.length; i++){  
var div = document.createElement("div");  
document.body.appendChild(div);  
  
}
```



String Manipulation

To work with strings in JavaScript, you need to learn about the various methods that handle them. The methods come from the JavaScript String object.



This chapter first explains what the String object is and how to create strings that use its properties and methods. Then, the String object's properties and methods are discussed in more detail so you can see how they work. Finally, you'll code a script that uses some of the properties and methods you've learned.


1- String Object

The string object makes it easier to handle strings or The String object is used to manipulate a stored piece of text.

The following methods and properties are available for strings:

2- The split() Method

The space character " " we mentioned will be our *delimiter* and it is used by the split function as a way of breaking up the string. Every time it sees the *delimiter* we specified, it will create a new element in an array. The first argument of the split function is the *delimiter*.



```
<script type="text/javascript">  
var myString = "123456789";  
var mySplitResult = myString.split("5");  
document.write("The first element is " +  
mySplitResult[0]);  
document.write("<br /> The second  
element is " + mySplitResult[1]);  
</script>
```




Display:


The first element is 1234

The second element is 6789

3- The `replace()` Method


JavaScript's String Object has a handy function that lets you replace words that occur within the string. The string *replace* function has two arguments:

- SearchFor - What word is going to be replaced. This can be a string or a regular expression.



➤ ReplaceText - What the word will be replaced with. This needs to be a string.

replace returns the new string with the replaces, but if there weren't any words to replace, then the original string is returned.



```
<script type="text/javascript">
  var visitorName = "Chuck";
  var myOldString = "Hello username! I hope
you enjoy your stay username.";
  var myNewString = myOldString.replace
("username", visitorName);
  document.write("Old string = " +
myOldString);
  document.write("<br />New string = " +
myNewString); </script>
```




Display:

Old string = Hello username! I hope you
enjoy your stay username.


New string = Hello Chuck! I hope you enjoy
your stay username.

4- The search() Method

This string function takes a regular expression and then examines that string to see if there are any matches for that expression. If there is a match, it will return the position in the string where the match was found. If there isn't a match, it will return -1.



The most important thing to remember when creating a regular expression is that it must be surrounded with slashes */regular expression/*.



```
<script type="text/javascript">  
var myRegExp = /Pisith/;  
var string1 = "Today John went to the store and  
talked with Pisith.";   
var matchPos1 = string1.search(myRegExp);  
  if(matchPos1 != -1)  
document.write("There was a match at position " +  
matchPos1);  
  else  
document.write("There was no match in the first  
string");  
</script>
```




Display:

There was a match at position 45

5- The String Length() Method

Advanced scripters will often need to know how long a JavaScript string is. For example, if a **webdeveloper** was creating a submission form that required the username to be no longer than 20 characters, then she would need to check the length of the string before allowing the user to submit data.



```
<script type="text/javascript">
```

```
var myString = "123456";
```

```
var length = myString.length;
```

```
document.write("The string is this long: " +  
length); // Same thing, but using the  
property inside the write function
```

```
document.write("<br />The string is this  
long: " + myString.length);
```

```
</script>
```



Display:


The string is this long: 6

The string is this long: 6



Date, Time and Timers


The Date object is useful when you want to display a date or use a timestamp in some sort of calculation. In Java, you can either make a Date object by supplying the date of your choice, or you can let JavaScript create a Date object based on your visitor's system clock. It is usually best to let JavaScript simply use the system clock.



When creating a Date object based on the computer's (not web server's!) internal clock, it is important to note that if someone's clock is off by a few hours or they are in a different time zone, then the Date object will create a different times from the one created on your own computer.

1- Date today(Current)

To warm up our JavaScript Date object skills, let's do something easy. If you do not supply any arguments to the Date constructor (this makes the Date object) then it will create a Date object based on the visitor's internal clock.



```
<h4>It is now  
<script type="text/javascript">  
<!--  
var currentTime = new Date()  
//-->  
</script> </h4>
```





Display:


It is now

2- Getting Date Object

The Date object has been created, and now we have a variable that holds the current date! To get the information we need to print out, we have to utilize some or all of the following functions:

- 
- `getTime()` - Number of milliseconds since 1/1/1970 @ 12:00 AM
 - `getSeconds()` - Number of seconds (0-59)
 - `getMinutes()` - Number of minutes (0-59)
 - `getHours()` - Number of hours (0-23)

- 
- `getDay()` - Day of the week(0-6). 0 = Sunday, ... , 6 = Saturday
 - `getDate()` - Day of the month (0-31)
 - `getMonth()` - Number of month (0-11)
 - `getFullYear()` - The four digit year (1970-9999)



Now we can print out the date information. We will be using the *getDate*, *getMonth*, and *getFullYear* methods in this example.



<h4>It is now

```
<script type="text/javascript">
```

```
<!--
```

```
var currentTime = new Date()
```

```
var month = currentTime.getMonth() + 1
```

```
var day = currentTime.getDate()
```

```
var year = currentTime.getFullYear()
```

```
document.write(month + "/" + day + "/" + year)
```

```
//-->
```

```
</script>
```

```
</h4>
```




Display:

It is now 9/23/2012 !

Notice that we added 1 to the *month* variable to correct the problem with January being 0 and December being 11. After adding 1, January will be 1, and December will be 12.

3- Timer in page

Now, instead of displaying the date we, will display the format you might see on a typical digital clock -- HH:MM AM/PM (H = Hour, M = Minute).



```
<h4>It is now <script type="text/javascript">
<!--
var currentTime = new Date()
var hours = currentTime.getHours()
var minutes = currentTime.getMinutes()
if (minutes < 10){
minutes = "0" + minutes }
document.write(hours + ":" + minutes + " ")
  if(hours > 11){
document.write("PM") }
  else {
document.write("AM") }
//--> </script> </h4>
```



Display:


It is now 9:08 AM

Above, we check to see if either the *hours* or *minutes* variable is less than 10. If it is, then we need to add a zero to the beginning of minutes. This is not necessary, but if it is 9:08 AM, our clock would output "9:8 AM", which doesn't look very nice at all!




Arrays and Array Function

An array is a variable that can store many variables within it. Many programmers have seen arrays in other languages, and they aren't that different in JavaScript.



The following points should always be remembered when using arrays in JavaScript:

- The array is a special type of variable.
- Values are stored into an array by using the array name and by stating the location in the array you wish to store the value in brackets. Example:
`myArray[2] = "Hello World";`




➤ Values in an array are accessed by
The array name and location of the
value. Example: `myArray[2];`

➤ JavaScript has built-in functions for
arrays, so check out these built-in
array functions before writing the code
yourself!


1- Using Arrays in JavaScript

To access the elements of an array, you use what is often called an “index number” that allows you access to each element of the array by its position in the array. For instance, the syntax to assign the first element of an array to a variable is shown in the following example:




```
var varname = arrayname[0];
```

You would replace *varname* with a variable name and *arrayname* with the name of the array you wish to access. The **0** in brackets after *arrayname* is the index number for the first element of the array.



The index number is **0** because arrays begin counting at **0** instead of **1**; thus, you need to be careful that you do not get confused about the index number of an element in an array. The first element has an index number of **0**, the second has an index number of **1**, the third has an index number of **2**, and so on.




Inside the square brackets immediately following the array name is where you place the index number of the element you wish to access in the array. To see this in action, Consider the example of the four students. The array for the name of each student was defined as in the following code:




```
var Stu_list = new Array("Thida",  
"Sok", "Rasmey","Ry");
```

Now, suppose you want to assign the value of the first element in the array (Thida,in this case, since that is the first element in the list) to a variable named **tall_student**.



Remember that the first element has an index number of 0; so, to get the value of the first element assigned to the variable, you would use the following code:

```
var tall_student = Stu_list[0];
```



Now you could write a short script to write it on the page. The following example writes the value on the page as part of a sentence:



```
<body>
```

```
<script type="text/javascript">  
var Stu_list = new Array("Thida","Sok","  
Rasmey","Ry");  
var tall_student = Stu_list[0];  
document.write("The tallest student in class  
is "+tall_student);  
</script>
```

```
</body>
```



```
<body>
```

```
<script type="text/javascript">  
var Stu_list = new Array("Thida","Sok",  
"Rasmey","Ry");  
document.write("The tallest student in class  
is "+Stu_list[0]);  
</script>
```

```
</body>
```


2- Creating Arrays

You have seen one way to define an array, but there are other methods of doing so that will come in handy later in the chapter. The following methods of defining arrays will allow you to use the method best suited for various situations.


3- Space Now and Assign Later

One method of defining an array is to assign a certain amount of space (elements) to an array, and then allow the values to be assigned later in the script. This is done using a single number inside the parentheses when defining the array:




```
var Stu_list = new Array(4);
```

This creates an array named **Stu_list**, which will initially have four elements (you can add elements to an array later if you wish). Keep in mind the index numbers will be 0, 1, 2, and 3. you will give the four elements of the array values later in the script.




if you want to assign a value of
“Borey” to the third position in the
array (index number 2), you would
use the following code:

```
Stu_list[2] = "Borey";
```



You can use a line such as this anywhere after the array has been defined. The technique used in the preceding code also allows you to add elements to an array if you decide you need more elements later in the script. You can also use a loop to assign all the elements of the array, but that will be discussed later in the chapter when you work with arrays and loops.




So, suppose you would like to add a fifth element to your **Stu_list** array. By assigning it a value (index number 4), you can add a new name to the array in the fifth position:

```
Stu_list[4] = "Piesy";
```


This adds a fifth element to your **Stu_list** array with a value of Pat.

4- Space Now and Assign Now


Another way you can define an array is very similar to the one just covered. The only difference is that some or all of the elements of the array are assigned right after the new array is created.



To see an example, consider the **Stu_list** array again. The following code shows how you can assign the elements of an array right after the line that creates the array space:



```
var Stu_list = new Array(4);  
Stu_list[0]="Thida";  
Stu_list[1]="Sok";  
Stu_list[2]="Rasmey";  
Stu_list[3]="Ry";
```



This time all four elements are assigned as the array is created. This method is advantageous if you want to assign each element but do not wish to use one long list of parameters on the same line.

5- Array Name Using No Parameters


Another option for defining an array is to define it with no parameters. This creates an array with no elements, but you can add elements later, as with the other options for defining arrays.

To create an array with no elements, you create an array with no parameters:




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


This creates an array named s_list with no elements. You can still add more elements later in the script by using the next index number in the sequence. So, if you decide later that you want the array to contain one element, you could use the following code:



```
var Stu_list= new Array();  
var x = 17;  
var y = x+2;  
var my_message = "Hi!";  
Stu_list[0]="Thida";
```



If you assign a value to an element that has a higher index number, the array expands to have a slot for any elements that come before it. For instance, take a look at the following code:




```
var s_list = new Array();  
var x = 17;  
var y = x+2;  
var my_message = "Hi!";  
Stu_list[29]="Thida";
```



6- Properties and Methods of the Array

As with other objects, an instance of the JavaScript Array object is able to use various properties and methods of the Array object.




As you will see as you read through the descriptions of the individual properties and methods of the Array object, you access the properties by using the array name followed by the property or method name you wish to use.


7- The constructor Property

The constructor property contains the value of the function code that constructed an array:


```
function Array() { [native code] }
```



Property	Description
constructor	Refers to the constructor function used to create an instance of an object
index	Used when an array is created by a regular expression match
input	Used when an array is created by a regular expression match
length	Contains a numeric value equal to the number of elements in an array
prototype	Allows the addition of properties and methods to objects such as the JavaScript Array object



The constructor property is mainly useful for informational purposes or for comparisons. To access the property, you use the array name followed by the dot operator and the constructor keyword. The following is the general syntax for using an Array object property: arrayname.**property**



So, to use the constructor property, you need to create an array so that you have an instance of the Array object to use. Instead of creating a new array, the s_list array from earlier is used here, allowing you to use the following code:


```
var Stu_list = new Array(4);  
window.alert(Stu_list.constructor)  
;
```

8- The index and input Properties


To understand the index and input properties, you first need to understand regular expressions (which are used to match text strings), which requires a lengthy explanation, as provided in The last slide.

9- The length Property

The length property returns a number that tells you how many elements are in an array.



To use the length property, you use the name of the array and then add the length property afterward. The following code shows an example of the length property being used to tell the viewer how many elements are in an array:



```
var Stu_list = new Array(4)
```

```
Stu_list[0]="Thida";
```

```
Stu_list[1]="Sok";
```

```
Stu_list[2]="Rasmey";
```


```
Stu_list[3]="Ry";
```

```
window.alert("The array has  
"+Stu_list.length+" elements");
```

This will send an alert that says **"The array has 4 elements"** to the viewer.


10-The prototype Property

The prototype property allows you to add properties and methods to an object that already exists, such as the Array object. By using this property, you can add properties to the object from outside the constructor function.




For example, if you decide that the Array object needs another property for one of your pages, you could use the prototype property to assign a new property using the following syntax:

Array.prototype.new_property=default_value;



You would replace *new_property* with the name you wish to use for your new property and then replace *default_value* with a default value for that property.

So, if you want to add a new property named attitude to the Array constructor function on your page and give it a default value of "cool", you could use code such as the following:




```
Array.prototype.attitude = "cool";  
var Stu_list = new Array();  
window.alert("This place is "+Stu_list.attitude);
```

```
.....  
var fish = new Array();  
fish.attitude = "wide-eyed";  
window.alert("Fish are "+fish.attitude);
```

Arrays Methods

Now that you know the properties of the Array object, this section presents the methods that you can use to do different things with your arrays.

Following the table, each method is described in more detail.



Method	Description
<code>concat()</code>	Combines the elements of two or more arrays into one new array
<code>join()</code>	Combines the elements of an array into a single string with a separator character
<code>pop()</code>	Removes the last element from an array and then returns the removed element if needed
<code>push()</code>	Adds elements to the end of an array and then returns the numeric value of the new length of the array if needed
<code>reverse()</code>	Reverses the direction of the elements in an array: the first element is moved to the last slot and the last element is moved to the first slot, and so on



shift()	Removes the first element from an array and then returns that element if needed
unshift()	Adds elements to the beginning of an array and returns the numeric value of the length of the new array if needed
slice()	Pulls out a specified section of an array and returns the section as a new array
splice()	Removes elements from an array or replaces elements in an array
sort()	Sorts the elements of an array into alphabetical order based on the string values of the elements
toString()	Combines the elements of an array into a single string with a comma as a separator character

1- The concat() Method

The **concat()** method is used to combine (or concatenate) the elements of two or more arrays and return a new array containing all of the elements.

```
var fruits = new Array("oranges","apples");  
var veggies = new Array("corn","peas");  
var fruits_n_veggies = fruits.concat(veggies);
```

2- The `join()` Method

The `join()` method is used to combine the elements of an array into a single string, with each element separated by a character sent as a parameter to the method.



```
<body>
```

```
<script type="text/javascript">
```

```
var fruits = new Array("oranges","apples","pears");
```

```
var fruit_string = fruits.join();
```

```
document.write("The new string is "+fruit_string);
```

```
</script>
```


```
</body>
```

Result: The new string is oranges,apples,pears

3-The pop() Method

The **pop()** method is used to remove the last element from an array.

```
var fruits = new  
Array("oranges","apples","pears");  
fruits.pop();
```



This creates an array named fruits with three elements (oranges, apples, pears). Then, the last element is removed using the `pop()` method, shortening the array to have only the first two elements (oranges, apples).

4- The push() Method

The **push()** method is used to add elements to the end of an array.

```
var fruits = new Array("oranges","apples");  
fruits.push("pears");
```

5- The reverse() Method

The **reverse()** method is used to reverse the order of the elements in an array.

```
var fruits = new Array("oranges","apples","pears");  
fruits.reverse();
```


6- The shift() Method

The shift () method is used to remove the first element of an array.

```
var fruits = new Array("oranges","apples","pears");  
fruits.shift();
```

7- The unshift() Method

The **unshift()** method is used to add elements to the beginning of an array.


```
var fruits = new Array("apples","pears");  
fruits.unshift("oranges");
```

8- The slice() Method


The **slice()** method is used to slice a specified section of an array and then to create a new array using the elements from the sliced section of the old array.

arrayname.slice(start,stop)

```
var fruits = new  
Array("oranges","apples","pears","grapes");  
var somefruits = fruits.slice(1,3);
```



This slices the second element (index number 1) through the third element (index number 2) of the array. It does not pull out the fourth element (index number 3) because 3 is the index number after 2, which is the index number of the last element designated to be removed.



The new array named `somefruits` contains the sliced elements (apples, pears).

9- The splice() Method

The **splice()** method allows you to remove or replace elements within an array.

```
var fruits = new  
Array("oranges","apples","pears","grapes");  
var somefruits = fruits.splice(2,1);
```

10- The sort() Method


The **sort()** method sorts an array in alphabetical order (like a directory listing).

```
var fruits = new  
Array("oranges","apples","pears","grapes");  
fruits.sort();
```




Error Handling


Every major web application needs a good error - handling protocol and most good ones do, though it is typically on the server side of the application.




In fact, great care is usually taken by the server - side team to define an error – logging mechanism that categorizes errors by type, frequency, and any other metric that may be important.



Error handling has slowly been adopted on the browser side of web applications even though it is just as important. An important fact to understand is that most people who use the Web are not technically savvy — most don't even fully comprehend what a web browser is, let alone which one they're using.




In the best case, the user has no idea what happened and will try again; in the worst case, the user gets incredibly annoyed and never comes back. Having a good error - handling strategy keeps your users informed about what is going on without scaring them.




To accomplish this, you must understand the various ways that you can trap and deal with JavaScript errors as they occur.

1- The try - catch Statement


ECMA - 262, 3rd Edition, introduced the **try - catch** statement as a way to handle exceptions in JavaScript. The basic syntax is as follows, which is the same as the **try - catch** statement in Java:




```
try {  
    //code that may cause an error  
}  
catch (error) {  
    //what to do when an error occurs  
}
```




Any code that might possibly **throw** an error should be placed in the try portion of the statement, and the code to handle the error is placed in the **catch** portion, as shown in the following example:



```
try {  
  window.someNonexistentFunction();  
}  
catch (error){  
  alert("An error happened!");  
}
```




If an error occurs at any point in the **try** portion of the statement, code execution immediately exits and resumes in the catch portion. The **catch** portion of the statement receives an object containing information about the error that occurred.



Unlike other languages, you must define a name for the error object even if you don ' t intend to use it.

For example:


```
try {  
    window.someNonexistentFunction();  
} catch (error){  
    alert(error.message);  
}
```




This example uses the message property when displaying an error message to the user. The message property is the only one that is guaranteed to be there across Internet Explorer (IE), Firefox, Safari, Chrome, and Opera even though each browser adds other information.

2- The finally Clause


The optional finally clause of the try - catch statement always runs its code no matter what. If the code in the try portion runs completely, the finally clause executes; if there is an error and the catch portion executes, the finally portion still executes.




There is literally nothing that can be done in the try or catch portion of the statement to prevent the code in finally from executing, which includes using a return statement. Consider the following function:



```
function testFinally(){  
  try {  
    return 2;  
  } catch (error){  
    return 1;  
  } finally {  
    return 0;  
  }  
}
```




This function simply places a return statement in each portion of the try - catch statement. It looks like the function should return 2, since that is in the try portion and wouldn't cause an error.



However, the presence of the finally clause causes that return to be ignored; the function returns 0 when called no matter what. If the finally clause were removed, the function would return 2.

3- Error - handling strategies

Error - handling strategies have traditionally been confined to the server for web applications. There ' s often a lot of thought that goes into errors and error handling, including logging and monitoring systems.



The point of such tools is to analyze error patterns in the hopes of tracking down the root cause as well as understanding how many users the error affects.

4- Identify Where Errors Might Occur

The most important part of error handling is to first identify where errors might occur in the code. Since JavaScript is loosely typed and function arguments aren't verified, there are often errors that become apparent only when the code is executed.




In general, there are three error categories to watch for:

- ❖ Type coercion errors
- ❖ Data type errors
- ❖ Communication errors



❖ Type Coercion Errors:

Type coercion errors occur as the result of using an operator or other language construct that automatically changes the data type of a value.



The two most common type coercion errors occur as a result of using the equal (`==`) or not equal (`!=`) operator and using a non - Boolean value in a flow control statement such as `if` , `for` , and `while` .



❖ *Data Type Errors*

Data type errors most often occur as a result of unexpected values being passed into a function. For example:



Ex1:

```
function reverseSort(values)
{
if (values){ //avoid!!!
values.sort();
values.reverse();
}
}
```



Ex2:

```
function reverseSort(values){  
    if (values != null){ //avoid!!!  
        values.sort();  
        values.reverse();  
    }  
}
```



Ex3:

```
function reverseSort(values){  
    if (typeof values.sort == "function"){  
        //avoid!!!  
        values.sort();  
        values.reverse();  
    }  
}
```




It is fixed:

```
function reverseSort(values){  
  if (values instanceof Array){ //fixed  
    values.sort();  
    values.reverse();  
  }  
}
```



❖ Communication Errors

The first type of communication error involves malformed URLs or post data. This typically occurs when data isn't encoded using `encodeURIComponent()` before being sent to the server.




The `encodeURIComponent()` method should always be used for query string arguments. To ensure that this happens, it 's sometimes helpful to define a function that handles query string building, such as the following:


```
function addQueryStringArg(url, name, value){  
    if (url.indexOf("?") == -1){  
        url += "?";  
    } else {  
        url += "&";  
    }  
    url += encodeURIComponent(name) + "=" +  
           encodeURIComponent(value);  
    return url;  
}
```




Using Cookies

A *cookie* is a small text file that is stored on the end user's computer. It can be referenced any time the viewer returns to your site, provided the cookie hasn't been deleted or expired. Of course, if the viewer doesn't accept cookies, then a cookie won't be able to be set or referenced later. Keep the following points in mind when using cookies:


- 
- ✓ Cookies must be 4KB (4000 characters) each or less.
 - ✓ A browser can accept up to only 20 cookies from a single domain.
 - ✓ If a number of viewers don't accept cookies, this eliminates any advantages of your
 - ✓ cookie(s) to those viewers.



Netscape invented cookies to help users browse your site more effectively. For instance, if you use a script on your main page that sends one or more alerts while the page is loading, you won't want that to happen every time the viewer goes to another page on your site and then returns to the home page.




The alerts pop up each time the page loads, because HTTP lacks state persistence. Cookies fill that gap because they allow the browser to “remember” that the viewer has seen the pop-up alert before and thus not display it on subsequent page visits.



With cookies, you can fix problems like these for any viewers who have cookies enabled.

1- Setting a Cookie

Setting a basic cookie is as easy as giving a value to the cookie property of the document object. The only restriction is that you can't have spaces, commas, or semicolons inside the string.




For example, you can set a cookie that will store a name so that you can identify it if you set more than one cookie later. The following code sets a basic cookie:

```
function set_it() {  
document.cookie="name=tasty1";  
}
```




Note:

Remember when setting a cookie not to use spaces, commas, or semicolons inside the string that sets the cookie data.




The preceding code sets a cookie with a value of name=**tasty1** when the function is called. You can set any delimiter you want, though (or none at all, but setting delimiters allows you to store multiple values like a query string), so the following code would work as well:




```
function set_it() {  
document.cookie="name:tasty1";  
}
```


The only problem is that you may need to use a space, comma, or semicolon in your cookie at some point.




For instance, you might want to add in some text that tells what your favorite kind of cookie is. Adding the additional information isn't very difficult as long as the value does not need a space, as shown in the following code:



```
function set_it() {  
  document.cookie="name=tasty1&fav=  
  Sugar";  
}
```




As you can see, the value of the cookie is being formatted in name=value pairs, and each pair is separated with an ampersand (&). Again, you can choose any type of separators you want. The following code is fine as well:



```
function set_it() {  
  document.cookie="name:tasty1|fav:Sugar";  
}
```


The following code shows how you could use the `escape()` method to set a cookie with a space in it:



```
function set_it() {  
  var thetext="name=tasty1&fav=Soda"  
  var newtext=escape(thetext);  
  document.cookie=newtext;  
}
```

2- Allowing User Input


By using the `escape()` method, you can prompt the viewer for the information, escape it, and then use it in the cookie. The following code shows how to do this to get the viewer's favorite type of cookie:



```
function set_it() {  
  var thefav=window.prompt("Enter  
your favorite type of cookie","");  
  var  
  thetext="name=tasty1&fav="+thefav;  
  var newtext=escape(thetext);  
  document.cookie=newtext;  
}
```

3- Setting an Expiration Date

Adding an expiration date to a cookie will keep it from being deleted once the browser is closed, or it can be used to expire a cookie you no longer want to use. To set an expiration date, add a little more to your string for the cookie, as shown in the following code:




```
function set_it() {  
var thetext="quote=I have a quote";  
var expdate=";expires=Mon, 30 Mar  
2009 13:00:00 UTC";  
var newtext=escape(thetext);  
newtext+=expdate;  
document.cookie=newtext;  
}
```

4- Reading a Cookie

Reading cookies is fairly simple if you have only a single cookie set and want to read it. To read the cookie, you just need to get the value of the `document.cookie` property from the browser:

```
function read_it() {  
    var mycookie=document.cookie;  
}
```



```
function read_it() {  
var mycookie=document.cookie;  
var fixed_cookie=  
unescape(mycookie);  
var thepairs= fixed_cookie.split("&");  
}
```



Thank You !