**Functions**

The most basic purpose of a function is to contain code. A function's advanced features are that it can take parameters and return values.  
  
Once we have defined a function, we can call it from anywhere, and any number of times.  
  
<HTML>  
<HEAD>  
<TITLE>Functions</TITLE>  
<SCRIPT>  
function hello() {  
document.write('Hello');  
}  
hello();  
</SCRIPT>  
</HEAD>  
</HTML>  
  
As we see, Hello is written to the screen.  
  
In JavaScript, we do not have to define the function before we use it, so we could equally type:  
  
hello();  
function hello() {  
document.write('Hello');  
}  
  
To see the power of functions, we could also type:  
  
function hello() {  
document.write('Hello');  
}  
hello();  
hello();  
  
Now we see two Hello's - because we have called the function twice.  
  
Note that the code for a function MUST be contained by curly brackets {}, even if the code is only one line. So,  
  
function hello() document.write('Hello');  
  
is not legal JavaScript, although it looks like it should be.  
Parameters  
  
Functions can become more powerful. They can take parameters, which means that we can pass variables to a function for the function to work on.  
  
<HTML>  
<HEAD>  
<TITLE>Parameters</TITLE>  
<SCRIPT>  
function greeting(greetStr) {  
document.write(greetStr);  
}  
greeting('Hello');  
greeting('Goodbye');  
</SCRIPT>  
</HEAD>  
</HTML>  
  
(See example) Now, when we call the function we pass a string to it. Inside the function, this string becomes greetStr, and we can manipulate greetStr using terms we covered in previous tutorials.  
  
So, if we change the function to:  
  
document.write(greetStr+'?');  
  
our example would now produce the output: Hello?Goodbye?.  
  
JavaScript's loosely-typed nature can be seen here - try:  
  
greeting(5);  
  
You will see 5.  
  
We can also pass variables, try:  
  
FName='Jon';  
greeting('Hello '+FName);  
  
and you will see Hello Jon.

**Multiple Parameters**

We can pass a list of parameters to a function, and we do not need to pass all of them, although we must be precise about their order.  
  
<HTML>  
<HEAD>  
<TITLE>Multiple Parameters</TITLE>  
<SCRIPT>  
function greeting(greetStr,number) {  
if (number!=parseInt(number)||number<=0) number=1;  
for (iter=0;iter<number;iter++) document.write(greetStr+'<BR>');  
}  
greeting('Hello',5);  
greeting('Goodbye');  
greeting('HelloZero',0);  
greeting('HelloThreePointOne',3.1);  
greeting('HelloABC','ABC');  
greeting('HelloMinusTen',-10);  
</SCRIPT>  
</HEAD>  
</HTML>  
  
This program vastly increases the power of our greeting function. We can now tell the function to display a string an exact number of times, according to the number we specify.  
  
But we do need to be careful. As you can see, when we write 'Goodbye', we do not pass a number parameter. The lack of number must be handled from inside the function, and we deal with it using the line:  
  
if (number!=parseInt(number)||number<=0) number=1;  
  
parseInt(number) returns the number in the string up-to the first non-digit (and accepts a minus sign), So,  
  
parseInt(5) equals 5  
parseInt(null) equals NaN  
parseInt(3.1) equals 3  
parseInt(0) equals 0  
parseInt('ABC') equals NaN  
parseInt(-10) equals -10  
  
Note that if we haven't set a number in the parameter list, then number is null. Also that NaN is 'Not a Number'.  
  
The following program demonstrates parsing:  
  
<HTML>  
<HEAD>  
<TITLE>parseInt </TITLE>  
<SCRIPT>  
function displayparseInt(n) {  
document.write('parseInt of '+n+' equals '+parseInt(n)+'<BR>');  
}  
displayparseInt(5);  
displayparseInt(null);  
displayparseInt(3.1);  
displayparseInt(0);  
displayparseInt('ABC');  
displayparseInt(-10);  
</SCRIPT>  
</HEAD>  
</HTML>  
  
Any integer will match its parseInt() value, all other values will fail, including the decimal. We can use this fact to detect whether the number parameter is usable or not, by using the 'NOT EQUAL TO' operator, '!='.  
  
But equally, number<=0 is a problem. Our loop wouldn't do anything, so we catch this condition as well with the OR ('||') logical operator.  
  
If number fails our test, we use an arbitary number=1 for the remainder of the function.  
  
We then pass number to the loop, which causes number of greetStr to be written to the document.  
Returning values from functions  
  
Sometimes we wish to get a result from a function. The function will quite happily calculate x squared, for example, but not tell us what the answer is.  
  
To get the answer, we use the return keyword.  
  
<HTML>  
<HEAD>  
<TITLE>Return</TITLE>  
<SCRIPT>  
function square(x) {  
return x\*x;  
}  
document.write(square(5) +'<BR>');  
ninesquared=square(9);  
document.write(ninesquared);  
</SCRIPT>  
</HEAD>  
</HTML>  
  
square(5) invokes the function, and the function returns x squared. We can either directly act upon this value, or store it in a variable, as we have done with ninesquared.

**Local Variables**

Functions have another quirk, which is that they can house local variables. A local variable is a variable that cannot be seen outside its scope, and behaves independently from other variables of the same name.  
  
This is useful if we are running short of inspired variable names.  
  
To declare a local variable, we use the var keyword.  
  
<HTML>  
<HEAD>  
<TITLE>Local</TITLE>  
<SCRIPT>  
x=5;  
function changex() {  
var x=10;  
document.write('Local x equals '+x+'<BR>');  
}  
document.write('Global x equals '+x+'<BR>');  
changex(x);  
document.write('Global x equals '+x);  
</SCRIPT>  
</HEAD>  
</HTML>  
  
This code gives us the strange output:  
  
Global x equals 5  
Local x equals 10  
Global x equals 5  
  
This occurs because we have two x's in the program, the global x and the local x, and the two variables do not interfere with each other. So, our x starts as 5, and we display this. Then we call the changex() function, which creates a local variable x, and initialises this to 10, and displays the local x value. But outside the function, the global x is unchanged, as we see when we write it to the screen again.  
  
We can refer to the global x from the function as well. Add:  
  
document.write('Global x equals '+window.x+'<BR>');  
  
to the changex() function (see example). The global x is seen as a property of the window object.  
Classes  
  
Classes can seem off-putting at first, but once you see the point of them, their use can be invaluable.  
  
We have already met objects. A computer object is a representation of a real object. For an estate agent the object may be a house, including information about the number of rooms and the price.  
  
An estate agent may have a lot of houses available. These houses all have different characteristics, and as objects they all go through the same processes. They are viewed, surveyed and bought, and so on.  
  
A full estate agent program would be difficult to demonstrate here, but we can introduce the use of classes.  
  
In this example, we have the house class. The house class produces house objects, all with object properties, such as number of rooms and price, and all having access to the same methods, such as sold and bought.  
  
So a class can create objects with a group of properties and methods.  
  
JavaScript doesn't have a keyword specific to class, so we must go back to basics and develop classes in a different way. This isn't very difficult.

**Class Properties**

Let us examine a very small estate agent program.  
  
<HTML>  
<HEAD>  
<TITLE>Estate Agent</TITLE>  
<SCRIPT>  
function House(rooms,price,garage) {  
this.rooms=rooms;  
this.price=price;  
this.garage=garage;  
}  
house1=new House(4,100000,false);  
house2=new House(5,200000,true);  
with (house1) document.write('House 1 has '+rooms+' rooms, '+(garage?'a':'no')+' garage, and costs £'+price+'<BR>');  
with (house2) document.write('House 2 has '+rooms+' rooms, '+(garage?'a':'no')+' garage, and costs £'+price+'<BR>');  
</SCRIPT>  
</HEAD>  
</HTML>  
  
We define a House function that takes three parameters, rooms, price and garage. The function uses the this keyword to create an object.  
  
When we call the House function, we assign the result to our variable, which becomes an object.  
  
So, identical code would be:  
  
house1=new Object();  
house1.rooms=4;  
house1.price=100000;  
house1.garage=false;  
  
We would have to type this in for all houses, which would be very tedious and is why we use the class structure instead.  
  
When we display the details for a house, I have introduced the ternary operator, '?:'. The ternary operator is a compacted version of:  
  
if (garage) str='a'; else str='no';  
  
(garage?'a':'no') means if garage is true, return 'a' else return 'no'. Using the ternary operator removes a line of code, and avoids having to create a new variable.  
Class Methods  
  
The House class we have so far defined only contains object properties. We could add a method to replace the document.write() action we used before. (See example)  
  
<HTML>  
<HEAD>  
<TITLE>Estate Agent 2</TITLE>  
<SCRIPT>  
function House(name,rooms,price,garage) {  
this.name=name;  
this.rooms=rooms;  
this.price=price;  
this.garage=garage;  
this.view=view;  
}  
function view() {  
with (this) document.write(name+' has '+rooms+' rooms, '+(garage?'a':'no')+' garage, and costs £'+price+'<BR>');  
}  
house1=new House('House 1',4,100000,false);  
house2=new House('Big House',5,200000,true);  
house1.view();  
house2.view();  
</SCRIPT>  
</HEAD>  
</HTML>  
  
Much better!  
  
Note how we must add another property, name, so that we can identify the house in question. This offers more flexibility than re-using the variable name, and the variable name is inaccessible anyway, i.e. it is very difficult, if not impossible, to get the view() function to use the string 'house1'.  
  
There is one further improvement that we can make, by using prototype.

**Prototype Methods**

Continuing the Estate Agent example, each House object has to have its own view() function. This can be very expensive on memory, 1000 houses would involve 1000 view() functions.  
  
We can remove the excess functions by using the prototype keyword. Prototype means that if a class is inherited from another class, then it can borrow all of its methods, so we don't need to redefine them.  
  
Any class that we create has a prototype. This prototype is available to all of the objects of the class.  
  
<HTML>  
<HEAD>  
<TITLE>Estate Agent 3</TITLE>  
<SCRIPT>  
function House(name,rooms,price,garage) {  
this.name=name;  
this.rooms=rooms;  
this.price=price;  
this.garage=garage;  
}  
function view() {  
with (this) document.write(name+' has '+rooms+' rooms, '+(garage?'a':'no')+' garage, and costs £'+price+'<BR>');  
}  
House.prototype.view=view;  
house1=new House('House 1',4,100000,false);  
house2=new House('Big House',5,200000,true);  
house1.view();  
house2.view();  
</SCRIPT>  
</HEAD>  
</HTML>  
  
(See example) We can drop the view from the object construction, which saves memory.  
  
We can declare the House.prototype.view method to be that of our view function.  
  
What this ultimately means is that now all objects of the House class have access to the view function, but the view function is only created once.

**Prototype Constants**

We can add prototype constants to classes as well, although these have limited appeal.  
  
If the estate agent program becomes financial active for example, we may decide to define:  
  
House.prototype.salecommission=.01;  
House.prototype.survey=1000;  
  
Which indicate that the commission for a sale is 1%, and that surveying fees are £1000.  
  
These two constants are available to any house object, so adding:  
  
document.write(house1.survey);  
  
displays 1000.  
  
The advantage of this is that it keeps variables of a related type together.  
  
<HTML>  
<HEAD>  
<TITLE>Estate Agent 4</TITLE>  
<SCRIPT>  
function House(name,rooms,price,garage) {  
this.name=name;  
this.rooms=rooms;  
this.price=price;  
this.garage=garage;  
}  
function view() {  
with (this) document.write(name+' has '+rooms+' rooms, '+(garage?'a':'no')+' garage, and costs £'+price+'<BR>');  
}  
function totalcost() {  
with (this) document.write(name+' has a total cost involved of £'+eval(price+survey+price\*salescommission)+'<BR>');  
}  
House.prototype.view=view;  
House.prototype.totalcost=totalcost;  
House.prototype.salescommission=.01;  
House.prototype.survey=1000;  
house1=new House('House 1',4,100000,false);  
house2=new House('Big House',5,200000,true);  
house1.view();  
house2.view();  
house1.totalcost();  
house2.totalcost();  
</SCRIPT>  
</HEAD>  
</HTML>  
  
eval() forces the expressions enclosed in the brackets to be interpreted as numbers. (See example)  
  
Every object of the House class has both constants as properties, and so we can refer to these values through the objects of the class (instances of the class).