|  |
| --- |
| [[https://myetudes.org/etudes-melete-tool/images/printer.png](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385315) Send to Printer](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385315) | [Close Window](https://myetudes.org/portal/tool/4c4d3792-8b10-40ce-8016-d7a5ac569a1c/print_module.jsf?printModuleId=1436385315) |
|  | |
| 4. Functions  4.1. Function Declarations, Parameters & Arguments  *Copyrighted Material - subject to fair use exception*  A function is a block of code that is defined once and is executed every time you call it.  To create a function, we use the 'function' keyword.  When the function keyword is followed by a name, the function will be stored under this name. **JavaScript does not require functions to have a name.**   If a function is not given a name, it is said to be anonymous.  If specified, the  function name must begin with a letter, a $ character or an underscore. Subsequent characters can be letters, digits, underscores, or $.  **The convention is to start all function names with lower case letters.**  When a name includes multiple words, the convention is to begin all words after the first with an uppercase letter such as longFunctionName().  Functions that are supposed to be internal or hidden are given names that begin with an underscore.  Function names are case sensitive so myfunction and myFunction are not the same.  After the name comes a list of parameter names between parentheses, and then finally the body of the function. The body of a function has to be enclosed in curly braces {}.  Below is a **function declaration**for a named function double.  function double(number) {      var result = 2 \* number;      return result;  }  The double function takes one parameter, number. It  returns twice the number that it was given.  The parameters of a function behave like local variables.  However we do not declare them with a var statement inside the function.  We just list them between the parentheses.  Their initial value is determined by the caller.  Once we define and name our function, we can**call** it with different values for the parameter number as follows:  double(2);  double(5);  double(10);  The initial values given to the parameters at the point of call are called the arguments.  In the function calls above,  2, 5 and 10  are **the arguments**to our function.  **They define the value that will be assigned to the parameter, number, inside the function definition.**  4.2. Return Value  *Copyrighted Material - subject to fair use exception*  In JavaScript, a function always has an explicit or implicit return value.  If the function includes a return statement, that statement causes the function to stop executing and to return the value of its expression (if any) to the caller.  If the return statement does not have an associated expression, it returns the undefined value.  If a function does not contain a return statement at all, it executes the code and returns undefined as well.  **The return value of a given function is accessed by writing functionName(arguments)**.  Let's go back to our function  double defined in the previous section:  function double(number) {      var result = 2 \* number;      return result;  }  The function returns a value that is equal to twice the number it is given.  The return value is  accessed by writing **double(5)**.  It can also  be assigned to a variable, and accessed from outside the function.    var amount = double(5);  console.log(amount);  10    We can also combine the two statements above into one and write:  console.log(double(5));  10    Now let's look at a function that takes more than one parameter.  The function longest below returns the longest of two given words.  We access that return value by writing **longest('hi', 'hello')**.  function longest(first, second) {      if (first.length >=  second.length) {          return first;      } else {          return second;      }  }  var greeting = **longest('hi', 'hello')**;  console.log(greeting);  hello  **If the return statement does not have an associated expression, it returns the undefined value.**  Consider the following function, average.   This function includes a return statement with no associated expression.  If we try to display its return value, we’ll see that it is undefined.  function average(first, second) {      var result = (first + second) / 2;      return;  }  var myAverage = average(90, 80);  console.log(myAverage);  undefined  **If a function does not contain a return statement, it executes the code and returns undefined.**  Consider the following function, triple.  This function does not  even include a return statement. Here again if we try to display its return value, we’ll see that it is undefined.  function triple (number) {    var result = 3 \* number;  }  var total = triple(200);  console.log(total);  undefined  4.3. Parameter Checking?  *Copyrighted Material - subject to fair use exception*  JavaScript does not generate any error when the number of arguments and the number of parameters do not match.**If there are too many argument values, the extra argument values will be ignored. If there are too few argument values, the undefined value will be substituted for the missing values.**There is no type checking on the argument values either: any type of value can be passed to any parameter.  Going back to our previous function double, we can call it with 3 arguments as follows:  function double(number) {      var result = 2 \* number;      return result;  }  console.log(double(3, 20, 50));  6  The arguments 20 and 50 are simply ignored.  We can also call our function with no argument as follows:  console.log(double**( )**);  NaN  Since we did not specify any value for the parameter number, JavaScript substituted undefined for its value.  Since 2 \* undefined is not a valid number, we get NaN when we output the return value double().  It is common practice to **use the or (||) operator to provide default values for parameters**in JavaScript functions.  Let's illustrate how to do that with our function double:    function double(number) {  **number = number || 0;** // if number is undefined, default to 0      var result = 2 \* number;      return result;  }    Remember that the or operator || returns the second operand if the first operand is falsy.  When number is undefined (falsy), the expression number || 0 evaluates to 0 so the default value 0 will be used as the new value of number.  And now when we invoke double with no argument, we get 0, instead of NaN:  console.log(double**( )**);  0  **There is no type checking on the argument values**: any type of value can be passed to any parameter.  Let's illustrate that with the following function, hello.  We an call hello with a string, a numeric parameter or a Boolean without generating any error since there is no type associated with the parameter.  function hello(name) {      var greeting = 'Hello ' + name + '!';      return greeting  }    console.log(hello('Alice'));  console.log(hello(5));  console.log(hello(true));  Hello Alice!  Hello 5!  Hello true!  4.4. Function Definition Expressions  *Copyrighted Material - subject to fair use exception*  The function declarations that we have seen so far can only appear as top-level statements. They can appear in global code, or within other functions, but **they cannot appear inside loops or conditionals**.  However functions can also be defined in **function definition expressions**.  **These may appear anywhere that an expression can appear.**  In the example below a function is created with the function keyword and assigned to a variable, opposite.  opposite is now a function and can be called with an argument.  var opposite =**function (number) {**  **var result = -number;**  **return result;**  **};**  console.log(opposite(10));  -10  The code in green above represents a function definition expression.  Note that we can call the typeof operator on the variable opposite and JavaScript will recognize it as a function.  >>> typeof opposite  "function"  Later on in the course we'll see how to use function definition expressions to directly pass a function as an argument to another function.  someFunction ( **function (parameters) { ...}**)    4.5. Variable Scopes  *Copyrighted Material - subject to fair use exception*  **Local JavaScript Variables**  A variable declared (using var) within a function is a local variable and can only be accessed from within that function.  We can have local variables with the same name in different functions, because **local variables are only recognized by the function in which they are declared and by its nested functions**.  Local variables are usually deleted as soon as the function is completed.  **Global JavaScript Variables**  Variables declared outside a function, are global variables, and all code (inside or outside the function) can access them.  The lifetime of JavaScript variables starts when they are declared.  Global variables are deleted when we close the web page or exit the program.  **Assigning Values to Undeclared JavaScript Variables**  If we assign a value to a variable that has not been declared, **the outcome will depend on whether or not we are using strict mode.**  In strict mode, we'll get an exception.  In non-strict mode, the variable will automatically be declared as a global variable.  To avoid this, **it is important to declare all variables that will be used inside a function at the beginning of that function.**  4.6. Hoisting  *Copyrighted Material - subject to fair use exception*  **Variable Hoisting:**  All variables declared within a function are visible throughout the body of the function. It is as if all variable declarations in a function (but not the associated assignments) are "hoisted" (moved up) to the top of the function.  Example:  myVar = 'first';  var myVar;  The two lines of code above are interpreted as:  var myVar;  myVar = 'first';  It is important to note that when the variable declaration and assignment are combined,**only the declaration is  hoisted**.   **The assignment is not.**   As an example, consider the following code:  console.log (myVar);  var myVar = 'Hello';  undefined  The console output here is undefined, not 'Hello' because the two lines of code above are equivalent to:  var myVar;  console.log (myVar);  myVar = 'Hello';    **Function Hoisting:**  **Function declarations** are also hoisted. This allows us to call a function before we declare it, when that function is defined in a function declaration.  console.log(double(5));  // We can call the function before we declare it.  function double(number) {      var result = 2 \* number;      return result;  }  10  **Note that function expressions are not hoisted**.  The following code will result in an error because**only the variable declaration (var opposite) is hoisted not the assignment to the function expression.**  console.log(opposite(10));  var opposite = function(number) {       var result = -number;       return result;  }  /\*  Exception: opposite is not a function  @Scratchpad/1:1    \*/  4.7. Nested Functions  *Copyrighted Material - subject to fair use exception*  Functions may be nested within other functions.   However, the outer function does not have access to the variables and functions defined inside the inner function. This provides a sort of security for the variables of the inner function.  Here’s an example:  function outer() {      var outerVariable = 'variable in outer function';  **function inner() {**  **var innerVariable = 'variable in inner function';**  **console.log('in inner:', outerVariable);  //** produces line 1 below  **console.log('in inner:', innerVariable);  //** produces line 2 below  **}**      inner();      console.log('in outer:', outerVariable);   // produces line 3 below      console.log('in outer:', innerVariable);   // produces the exception below  }  outer();  in inner: variable in outer function                             //  line 1  in inner: variable in inner function                            //   line 2  in outer: variable in outer function                           //    line 3  We get the following exception in Scratchpad because innerVariable is not defined in the function outer.  The exception is caused by the statement: console.log('in outer:', innerVariable);  /\*  Exception: innerVariable is not defined  outer@Scratchpad/1:10  @Scratchpad/1:12  \*/    4.8. Recursion  *Copyrighted Material - subject to fair use exception*  A function can also call itself.  If it does, we call it a recursive function.  Let's take a look at a recursive function that computes the factorial of a number.  The factorial of a number is the product of all numbers up to and including that number.  factorial(3) = 1 x 2 x 3 = 6  factorial (4) = 1 x 2 x 3 x 4 = 24  Note that:  factorial(1) = 1  and  factorial(n) = factorial(n – 1) x n  Those 2 observations allow us to write the following recursive definition of factorial:  function factorial(number) {      if (number <= 1) {          return 1;      } else {          return **factorial(number - 1)** \* number; // a recursive call to factorial inside factorial     }  }  console.log(factorial(5));  120  4.9. Closures  *Copyrighted Material - subject to fair use exception*  When functions are nested in JavaScript, the inner function has access to the scope of the outer function.  As a result, the variables defined in the outer function will live longer than the outer function itself, if the inner function manages to survive beyond the life of the outer function.  A closure is created when the inner function is somehow made available to any scope outside the outer function.  In practice, we can think of a closure as**a function that remembers a state**(without using a global variable.)  Let's work  through an example to see how that works.  function counter() {  **var number = 0;**      function increment() {          number = number + 1;          return number;      }  **return increment;  // a function can return another function**  }  The function counter defines and initializes a number and then **returns another function, increment.** That inner function increment has access to the number initialized in counter.  The variable, number, now survives outside of the function counter.  var count1 = counter();  console.log(count1());   // output 1  console.log(count1());   // output 2  console.log(count1());   // output 3  The result is:  1  2  3  To understand what's going on, let's trace through the above code, line by line:  var count1 = counter();  Here, **we are assigning the return value from counter to count1.**  When counter is called, number is initialized to 0, the **function increment is returned**.  The value of number that increment sees at this point is 0.  So count1 is assigned a copy of the function increment where number is 0. In other words, count1 is now a function too and we can call it as count1().  console.log(count1());  1  count1 is called:  it is really increment with number = 0, so number is updated first (number = number + 1) and then returned.  number is now 1. console.log gives us 1.  console.log(count1());  2  count1 is called:  it is really increment with number= 1, so number is updated first (number = number + 1) and then returned.  number is now 2.  console.log gives us 2.  console.log(count1());  3  count1 is called:  It is really increment with number = 2, so number is updated first (number = number + 1) and then returned..  number is now 3.  console.log gives us 3.  We can even keep a second independent count as follows:  var count2 = counter ();  Here counter is called, number is initialized to 0, the function increment is returned.  The value of number that increment sees at this point is 0.  So **count2 is assigned a different copy of the function increment where number is 0.**  console.log(count2());  1  count2 is called:  **It has a different copy of increment** with number still 0,  so number is updated first (number = number + 1) and then returned.  number is now 1.  console.log gives us 1.  Note that the only way to access these counts is by calling count1() and count2().  The closure here is used to  **remember the counts without the use of a global variable.** |  |