**Download Your Daily Notes**

**Website Application**

 This activity has been applied to our Mzamomtsha Primary School Website and it now allows the user to see the change in the navigation bar when the user is on a specific page by highlighting that section of the nav bar when on said page.

**Daily Notes - Additional Javascript Tasks**

 Calling one function from another function  
Code inside a function behaves just like code anywhere else. This means you can call one function from inside another function. This allows you to "nest" functions so that you can create separate functions, which each perform a specific task, and then run them together as a complete process, one right after the other.  
  
Creating objects with user-defined functions  
JavaScript is based on objects: the window is an object, links are objects, forms are objects, even Netscape itself (or other browser) is an object. Using objects can help make programming easier and more streamlined. You can extend the use of objects in JavaScript by making your own. The process uses functions in a slightly modified way. In fact, you'll be surprised how easy it is to make your own JavaScript objects.  
  
Making a new object entails two steps:  
Define the object in a user-defined function.  
Use the new keyword to create (or instantiate) the object with a call to the object function.  
  
Defining new properties to already-made objects  
After an object has been created you can assign a value to it. But instead of just assigning a value to the object itself, you should define a new property for the object, and assign a value to the property.  
  
myobject is the name of the user-defined object.  
property is the name of the property you want to create.  
value is the value you want to assign.  
  
Defining properties when you create the object  
Another method of defining properties for objects is to include the property names in the object function. You can use this technique to simultaneously create a new object and define the property values.  
  
Operators  
JavaScript has the following types of operators. This section describes the operators and contains information about operator precedence.  
  
Assignment operators  
Comparison operators  
Arithmetic operators  
Bitwise operators  
Logical operators  
String operators  
Conditional (ternary) operator  
Comma operator  
Unary operators  
Relational operators  
JavaScript has both binary and unary operators, and one special ternary operator, the conditional operator.  
  
Assignment Operators  
Return value and chaining  
  
Like most expressions, assignments like x = y have a return value. It can be retrieved by e.g. assigning the expression or logging it:  
  
An assignment operator assigns a value to its left operand based on the value of its right operand. The simple assignment operator is equal (=), which assigns the value of its right operand to its left operand. That is, x = y assigns the value of y to x.  
  
Compound assignment operators  
The return value matches the expression to the right of the = sign in the “Meaning” column of the table above. That means that (x = y) returns y, (x += y) returns the resulting sum x + y, (x \*\*= y) returns the resulting power x \*\* y, and so on.  
  
In the case of logical assignments, (x &&= y), (x ||= y),and (x jQuery224001261423579560672\_1679985275040= y), the return value is that of the logical operation without the assignment, so x && y, x || y, and x ?? y, respectively  
  
Note that the return values are always based on the operands’ values before the operation.  
  
When chaining these expressions, each assignment is evaluated right-to-left. Consider these examples:  
  
w = z = x = y is equivalent to w = (z = (x = y)) or x = y; z = y; w = y  
z += x \*= y is equivalent to z += (x \*= y) or tmp = x \* y; x \*= y; z += tmp (except without the tmp).  
  
Destructuring  
For more complex assignments, the destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects using a syntax that mirrors the construction of array and object literals.  
  
Comparison Operators  
A comparison operator compares its operands and returns a logical value based on whether the comparison is true. The operands can be numerical, string, logical, or object values. Strings are compared based on standard lexicographical ordering, using Unicode values. In most cases, if the two operands are not of the same type, JavaScript attempts to convert them to an appropriate type for the comparison. This behaviour generally results in comparing the operands numerically. The sole exceptions to type conversion within comparisons involve the === and !== operators, which perform strict equality and inequality comparisons. These operators do not attempt to convert the operands to compatible types before checking equality.  
  
Arithmetic operators  
An arithmetic operator takes numerical values (either literals or variables) as their operands and returns a single numerical value. The standard arithmetic operators are addition (+), subtraction (-), multiplication (\*), and division (/). These operators work as they do in most other programming languages when used with floating point numbers (in particular, note that division by zero produces Infinity).  
  
Bitwise Operators  
A bitwise operator treats their operands as a set of 32 bits (zeros and ones), rather than as decimal, hexadecimal, or octal numbers. For example, the decimal number nine has a binary representation of 1001. Bitwise operators perform their operations on such binary representations, but they return standard JavaScript numerical values.  
  
Bitwise Logical Operators  
Conceptually, the bitwise logical operators work as follows:  
  
The operands are converted to thirty-two-bit integers and expressed by a series of bits (zeros and ones). Numbers with more than 32 bits get their most significant bits discarded.  
  
Each bit in the first operand is paired with the corresponding bit in the second operand: first bit to first bit, second bit to second bit, and so on.  
The operator is applied to each pair of bits, and the result is constructed bitwise. For example, the binary representation of nine is 1001, and the binary representation of fifteen is 1111.  
  
Bitwise Shift Operators  
The bitwise shift operators take two operands: the first is a quantity to be shifted, and the second specifies the number of bit positions by which the first operand is to be shifted. The direction of the shift operation is controlled by the operator used.  
  
Shift operators convert their operands to thirty-two-bit integers and return a result of the same type as the left operand.  
  
Logical Operators  
Logical operators are typically used with Boolean (logical) values; when they are, they return a Boolean value. However, the && and || operators return the value of one of the specified operands, so if these operators are used with non-Boolean values, they may return a non-Boolean value.  
  
Short-circuit evaluation  
As logical expressions are evaluated left to right, they are tested for possible "short-circuit" evaluation using the following rules:  
  
false && anything is short-circuit evaluated to false.  
true || anything is short-circuit evaluated to true.  
The rules of logic guarantee that these evaluations are always correct. Note that the anything part of the above expressions is not evaluated, so any side effects of doing so do not take effect.  
Note that for the second case, in modern code you can use the new Nullish coalescing operator (??) that works like ||, but it only returns the second expression, when the first one is "nullish", i.e. null or undefined. It is thus the better alternative to provide defaults, when values like '' or 0 are valid values for the first expression, too.  
  
String Operators  
In addition to the comparison operators, which can be used on string values, the concatenation operator (+) concatenates two string values together, returning another string that is the union of the two operand strings.  
  
Conditional (ternary) Operator  
The conditional operator is the only JavaScript operator that takes three operands. The operator can have one of two values based on a condition.  
  
Comma Operator  
The comma operator (,) simply evaluates both of its operands and returns the value of the last operand. This operator is primarily used inside a for loop, to allow multiple variables to be updated each time through the loop. It is regarded bad style to use it elsewhere, when it is not necessary. Often two separate statements can and should be used instead.  
  
Unary Operator  
A unary operation is an operation with only one operand.  
  
Delete  
The delete operator deletes an object's property.  
  
Typeof  
The typeof operator is used in either of the following ways:  
The typeof operator returns a string indicating the type of the unevaluated operand. operand is the string, variable, keyword, or object for which the type is to be returned. The parentheses are optional.  
  
void  
The void operator is used in either of the following ways:  
The void operator specifies an expression to be evaluated without returning a value. expression is a JavaScript expression to evaluate. The parentheses surrounding the expression are optional, but it is good style to use them.  
  
Relational Operator  
A relational operator compares its operands and returns a Boolean value based on whether the comparison is true.  
in  
The in operator returns true if the specified property is in the specified object.  
  
instanceof  
The instanceof operator returns true if the specified object is of the specified object type. The syntax is:  
where objectName is the name of the object to compare to objectType, and objectType is an object type, such as Date or Array.  
Use instanceof when you need to confirm the type of an object at runtime.  
  
Operator Precedence  
The precedence of operators determines the order they are applied when evaluating an expression. You can override operator precedence by using parentheses.

**Daily Notes - The Document Object Model**

 Event Bubbling  
Bubbling is what the event itself does.  
The event starts at the element that triggered it (Clicking 1. Element button, An Event Listener would be attached to each element). Then, it bubbles up to each of it’s parent elements until it reaches the document. Any listeners on any of those parent elements would get triggered as it bubbles up. Causing all the events to occur on the page possibly slowing down your application.  
  
Using a form field as an example, the event would bubble up to the parent form, then any containers or divs the form was in, then the body, then the html element, then the document, then the window.  
  
Event Propagation  
handleCheck = e => {  
  
e.stopPropagation()  
  
// talk to my API, set the record as "done" or not  
  
}  
  
<span onClick={this.handleCheck}>[]</span>  
  
  
That e.stopPropagation() halts this “bubbling” of events “up” through the DOM. We stop all the events from the parents occurring.  
  
Event Delegation  
Event delegation is a technique for listening to event where you delegate a parent element as the listener for all the events that happen inside it.  
  
So instead of adding Event Listeners to each Button you would just add and event Listener to the parent Element and target the child element.

**Daily Notes - Regular Expressions**

 What Is a Regular Expression?  
A regular expression is a sequence of characters that forms a search pattern.  
  
When you search for data in a text, you can use this search pattern to describe what you are searching for.  
  
A regular expression can be a single character, or a more complicated pattern.  
  
Regular expressions can be used to perform all types of text search and text replace operations.

**Daily Notes - Activity 2 - Website Application**

 This activity has been applied to our Mzamomtsha Primary School Website and it now allows the user to see the change in the navigation bar when the user is on a specific page by highlighting that section of the nav bar when on said page.

**My Views on the Day**

 1. Learning all those additional JavaScript tasks, the documentation object model and implementing the color change in the navigation bar.  
  
2. Activity 2  
  
3. None  
  
4. Additional JavaScript Tasks

**Daily Notes - Day 2 Reflections**

 1. Learning all those additional JavaScript tasks, the documentation object model and implementing the color change in the navigation bar.  
  
2. Activity 2  
  
3. None  
  
4. Additional JavaScript Tasks