# Week1

JavaScript: a programming language that adds complex interactive features to a website

•Developed to handle customer data validation for online commerce

•World Wide Web (web) allows data sharing across a network of linked documents

•Introducing scripting languages

•JavaScript is a type of programming language, specifically a scripting language

•Programming language: a set of instructions directing the actions of the computer

•Typically must be compiledinto machine code by a program called a compiler

•Scripting languages: subcategory of programming languages that run directly from a program or script(are not compiled)

•Must be interpreted (read and scanned line-by-line) by an interpreter.

•Markup languages: languages that define the content, structure, and appearance of a document

•HTML (Hypertext Markup Language) defines the content and structure of a web page

•CSS (Cascading Style Sheets) defines how a web page will appear on a specified device

•Developers struggled to reconcile JavaScript and JScriptin the late 1990s

•JavaScript was submitted as a proposal for a standardized scripting language to the European Computer Manufacturers Association (ECMA) in 1997

•An ECMA committee developed a set of scripting language standards

•The specification for this scripting language, called ECMAScript (ECMA-262), is updated to a new edition annually

•Three foundations of the full implementation of JavaScript:

•ECMAScript (core of the language: syntax, keywords, properties, methods, etc.)

•Document Object Model (DOM): describes how to access the contents of a web page and user actions within that page

•Browser Object Model (BOM): describes how to access the features and behaviors of the browser itself

•DOM and BOM are Application Programming Interfaces (APIs)

•DOM specifications are managed by the World Wide Web Consortium (W3C)

•Current version is DOM Level 4, described as an ongoing “living standard”

•BOM is implemented by each browser application (largely the same among them)

Traditionally, a two-tier system consisting of:

•Server(back end): a device or application from which a client requests information; responsible for data storage, management, communication with external services, and heavy processing

•Client(front end): a device or application that presents an interface to the user and requests information from a server; responsible for gathering information from the user, submitting it to a server, then receiving, formatting, and presenting the results, and sometimes some related data processing

•Web’s two-tier client/server system consists of a web browser and web server that communicate via Hypertext Transfer Protocol (HTTP)

Adding databases and other applications to a web server yields a three-tier client/server system (a.k.a. multitier client/server system, n-tier client/server system): client tier, processing tier (a.k.a. middle tier), and data storage tier

JavaScript and client-side scripting

•HTML produces staticdocuments, whereas client-side scripting provided by JavaScript can respond dynamically to user actions because it runs on a local browser (client)

•Security concerns limit where JavaScript can be used, what files it can access/ commands it can run on the client’s system, and its direct interactions with web servers operating at the processing tier

Server-side scripting: programming using a scripting language (e.g., PHP, ASP.NET, Python, Ruby) that is executed from a web server

•These languages work in the processing tier and can handle communication between the client and data storage tiers, often preparing and processing data

•JavaScript and server-side scripts operate in separate environments but must work together to deliver interactive websites to users

Allow the client to handle the user interface processing and other light processing such as data validation

•Allow the web server to perform intensive calculations and data storage

•Performing some processing on the client is beneficial because:

•Processing power is not limited to the capabilities of the server when processing is distributed among multiple client devices using a web application

•Transfer times across the Internet are minimized, increasing speed

•Server resource requirements (infrastructure, power use) are decreased, reducing costs

Use an Integrated Development Environment (IDE) to manage all of the facets of website development

•Use a code editor to manage coding in HTML, CSS, and JavaScript within a graphical interface

Syntax for embedding JavaScript code in a web page’s HTML file:<script>statements</script>

•Browser stops loading the page and processes the statements when it encounters an embedded script

A JavaScript program is composed of statements(lines of code)

•Sample statement showing optional (but desirable per convention) ending

* semicolon: document.write("<p>Plant choices</>");

JavaScript is considered an object-based programming language

•Object: programming code (including methods) and data (properties) that can be treated as an individual unit or component

•Procedure: logical unit of a computer program consisting of a group of statements that perform a specific task

•Method: a procedure associated with an object

•Sample method call on carLoan

object: carLoan.calcPayments(60)

•Methods often require that you provide data (argument[s]) within the parentheses, which is known as passing arguments

•Property: a piece of data associated with an object

•Sample statement assigning a value for the interestproperty of the carLoanobject: carLoan.interest= .0349;

DOM's Document object represents the entire content of the web page

•The Document object's write()method writes new content to a web page while it is being loaded

•Performs essentially the same function that you perform when you manually add text to the body of a standard web document

•Useful for incorporating constantly-changing data at load time

•Requires a text string (a.k.a. literal string), which is text contained in double or single quotation marks, as an argument

•Will overwrite the entire web page if used after the browser finishes loading the page

•Object names must always be all lowercase

•Incorrect capitalization, e.g. Document.write("Plant choices"); or document.WRITE("Plant choices");, will cause errors

•Comments: lines of code that are not processed by browsers, which you can use to add notes about your code

•Line comment syntax:let apple = "Fuji"; // Variable assignment

•Block comment syntax:

/\*

Here comes a function!

\*/

Variable: specific location in the computer's memory where a program stores a value

•First created and assigned a name, then used to store a value

Rules and conventions for variable names, known as identifiers:

•Must begin with an uppercase or lowercase ASCII letter, dollar sign, or underscore

•Can include numbers but not as the first character

•Cannot include spaces

•Cannot be a reserved word (a.k.a. keyword)—a special word that is part of JavaScript syntax

Best practice is to use camel case, e.g., myVariableName

Declaring a variable creates it and thus is mandatory prior to using the variable

•Initializing a variable assigns it an initial value and is optional

•Syntax for declaring and initializing a variable using the assignment operator:let variable= value;var variable= value;

•Syntax for declaring a variable with a constant (unchangeable) value: const variable= value;

•Sample statements that declare and/or initialize variables:

let salesTotal; // Declares only

let curOrder= 47.58; // Declares and initializes

salesTotal= curOrder;

let orderNumber= "R0218", salesTotal= 47.58, curOrder;

Expression: a literal value or variable, or a combination of literal values, variables, operators, and other expressions, that can be evaluated by a JavaScript interpreter to produce a result

•Expressions are written using:

•Operands: variables and literals(values such as text strings or numbers)

•Operators: symbols used to manipulate operands

•Example uses of the addition (+) and assignment (=) operators:

let salesTotal= 47.58, shippingCost= 10;

let totalCost= salesTotal+ shippingCost;

document.write("<p> Your total costs is $" + totalCost+ "</p>");

Generates the HTML code

<p>Your total cost is $57.58</p>

5 + "2"returns the text string "52"

Expressions can be used to assign new values to variables at any point in a script

•Applies to variables declared with let or var; variables declared with const cannot be modified

•Variables need be declared only once, then can be assigned repeatedly

•Example:

let totalSales= 0;

let item1Sales = 50, item2Sales = 75, item3Sales = 40;

totalSales= item1Sales + item2Sales + item3Sales;

document.write("<p>Total sales = $" + totalSales+ "</p>");Writes the HTML <p>Total sales = $165</p>to the web page

Event: a specific circumstance (such as an action performed by a user or an action performed by the browser) that is monitored by JavaScript and that your script can respond to in some way

Not all events happen with all devices; here are a few examples (part of Figure 1-14)

Events are associated with HTML elements, with each element having its own set of available events

•The clickevent is available for many elements, including the aelement and form controls created with the inputelement

•The loadand unloadevents are available for the bodyelement

•Event handler: code that is executed in response to a specific event occurring on a specific element

•Syntax for an event handler included as an attribute of the initiating element:<elementonevent="JavaScript code">

•Example using the window.alert()method, which displays the string passed to it in a dialog box with an OK button:

<input type="submit" onclick="window.alert

In JavaScript, you look up an element by its idvalue using the getElementById()method of the Document object

•Sample HTML creating an input element with the idvalue firstName:<input type="text" id="firstName" />

•Sample JavaScript creating a variable that references that element:let fName= document.getElementById("firstName");

•Syntax for changing the value assigned to an attribute thus retrieved:

document.getElementById("firstName").value = value;

or fName.value= value;

Including a script element for each code section

•Several script elements can be included in a single HTML file

•Statements in one script section are accessible to subsequent script sections

•E.g., variables declared in a script section can be used in subsequent script sections

•Place script sections containing the document.write() method where the content is to be written

•The DOM is created during page load, and a script that references a part of the page that has not been loaded will cause an error

•Many developers place scripts at the end of the document to avoid this type of error

•The document.getElementById() method can reference page objects only after they are loaded into the DOM

Syntax for attaching a web page to a JavaScript source file from within the HTML file:<script src="url"></script>where url is the JavaScript source file's name and location

•The script element can be used for embedding JavaScript code or referencing a file—not both at once

Default behavior is for commands in an external .jsfile to be loaded when the browser initially encounters the script element in the HTML file

•With the async attribute, the browser parses the HTML and JavaScript code together, only pausing to process the script

•With the defer attribute, the browser parses and loads the HTML, then processes the script

Placing JavaScript code in an external file is preferable because:

•Code can be shared among pages or among team members

•When shared among pages, code need only be downloaded once

•The HTML file is kept neater and cleaner

•Websites are easier to manage when the HTML, CSS, and JavaScript files each focus on one task

Libraries: JavaScript source files that store especially useful generic scripts used on many different websites

•You can incorporate a library into HTML code by creating a script element in the head section and using the src attribute to specify the file name

•Popular libraries include Node.js, jQuery, and Modernizr

•Developers usually create customized versions of large libraries containing only the code they need to limit download time

Well formed web page documents conform to the rules and requirements of HTML

•Validating parser: a program that checks whether a web page is well formed and whether the document conforms to a specific language definition known as DTD

•E.g., W3C Markup Validation Service at http://validator.w3.org/

•Validation: the process of verifying that your document is well formed and checking that the elements in your document are correctly written according to the element definitions in a specific DTD

•Embedding JavaScript in an XHTML document

•Enclose the script element within a CDATA section

•Character data (CDATA): a document section that is not interpreted as markup

•Parsed character data (PCDATA): a document section that is interpreted as markup

Week2

Function: a programming structure consisting of a collection of statements that share a common purpose or calculate a value

•Syntax for a named function: function functionName(parameters) {statements}

•Syntax for an anonymous function: function (parameters) {statements}

Function’s parameters are the variables it uses

Enclosed in a command block (opening and closing curly braces)

JavaScript expression for calling a function:   
functionName(paramValues);

•paramValues passed to a function are the arguments (actual parameters)

Syntax for a function that returns a value: function functionName(parameters) {statementsreturn value;}

•return statement ends execution and returns a single value

Most direct method of associating a function with an event

•Drawback: places JavaScript code in the HTML file

•Syntax for creating an event handler as an attribute of the HTML element:

<elem onevent= "function()">

•Places the event handler within the JavaScript code file

•Can only specify function name, not parameter values

•Only one function can handle an event at a time

•Syntax for an event as an object property:   
object.onevent= function;

•An event listener listens for an event as it propagates through a web page, during either:

•The capture phase (event moves down the object hierarchy) or

•The bubbling phase (event moves back up the object hierarchy)

•Can attach multiple functions to the same event

•Syntax for method that attaches an event listener to an object:   
object.addEventListener("event", function, capture)

•Include entire structure of anonymous function in place of function name in an event handler or event listener

•Can pass in parameter values with this approach

|  |  |
| --- | --- |
| Function | Description |
| decodeURI(string) | Decodes text strings encoded with encodeURI() |
| decodeURIComponent(string) | Decodes text strings encoded with encodeURIComponent() |
| encodeURI(string) | Encodes a text string so it becomes a valid URI |
| encodeURIComponent(string) | Encodes a text string so it becomes a valid URI component |
| eval (string) | Evaluates expressions contained within strings |
| isFinite(number) | Determines whether a number is finite |
| isNaN(number) | Determines whether a value is the special value NaN(Not a Number) |
| parseFloat(string) | Converts string literals to floating-point numbers |
| parseInt(string) | Converts string literals to integers |

Example of using built-in function to verify the socialSecurityNumber variable is not a number:

let socialSecurityNumber= "123-45-6789";

let checkVar= isNaN(socialSecurityNumber);

document.write(checkVar);

Scope: where a variable or function can be called within the program

•Variable/function is only recognized within scope

•Referencing elsewhere results in an error

•Variables declared with let are block scoped: scope is limited to the command block

•Variables declared with var are function scoped: scope is limited to the function

•Variables/functions with local scope (e.g., local variables) are accessible within the command block or function where they are defined

•Includes block scope and function scope

•Those with global scope (e.g., global variables) are defined outside a block/function and thus accessible throughout the program

•Local variable takes precedence when in scope

•Assigning a value to the local variable does not affect the global variable’s value outside the local variable’s scope

•Global variables most useful for small applications and variables used as constants

•Local variables preferable for values used within and changed by functions

let quantityPerBox= 12;

function describeCandy(productName, numberOfBoxes) {

var numberCandies= quantityPerBox\* numberOfBoxes;

document.write("You would like " + numberCandies+ " of our " + productName+ " candies!");

}

quantityPerBox has global scope, unless this code appears within a command block in the larger program, in which case it would have block scope, a type of local scope. numberCandieshas function scope, a type of local scope. You can’t determine the scope of productNameor numberOfBoxesfrom this sample since they aren’t declared here.

Data type: the specific category of information that a variable contains

•Primitive types: data types that can be assigned only a single value

|  |  |
| --- | --- |
| Data Type | Description |
| number | A positive or negative number with or without decimal places, or a number written using exponential notation |
| Boolean | A logical value of trueor false |
| string | Text such as “Hello World!” |
| undefined | An unassigned, undeclared, or nonexistent value |
| null | An empty value |

Strongly typed (statically typed) programming languages require that you declare the type of data that a variable contains and do not allow you to alter that type

•Loosely typed (duck typed, dynamically typed) programming languages do not require you to declare the data type and allow data types to be change

JavaScript is loosely typed

•Data types cannot be declared when variables are created

•JavaScript interpreter determines and assigns or reassigns the variable’s data type based on the type of data stored

•Integer: positive or negative number without decimal places

•Floating point number: positive or negative numbers containing decimal places

•Integer: positive or negative number without decimal places

•Floating point number: positive or negative numbers containing decimal places

•Text string: zero or more characters surrounded by double or single quotation marks

•Empty string: zero-length string value

•Can use quotation marks within strings:

document.write("Welcome to 'Fan Trick Photography’”);

document.write('Welcome to "Fan Trick Photography”’);

•To split a text string onto a new line without causing an error:

•Use two or more strings concatenated by the addition operator (+)

•For some browsers, end a line with the \character to indicate the string continues

•Create a template literal by enclosing the string in backtick characters (`)

•An escape character is placed before characters within strings to indicate that they are to be treated as regular keyboard characters, not as syntax

•JavaScript’s escape character is the backslash (\)

•Escape sequence: combination of an escape character with a specific character, usually to carry out a special function

|  |  |
| --- | --- |
| Escape Sequence | Character |
| \\ | Backslash |
| \b | Backspace |
| \r | Carriage return |
| \" | Double quotation mark |
| \f | Form feed |
| \t | Horizontal tab |
| \n | Newline |
| \0 | Null character |
| \' | Single quotation mark (apostrophe) |
| \v | Vertical tab |
| \xXX | Latin-1 character specified by the XXcharacters, which represent two hexadecimal digits |
| \uXXXX | Unicode character specified by the XXXXcharacters, which represent four hexadecimal |

Two types of JavaScript operators: binary and unary

•Binary operator: requires an operand before and after the operator

•Arithmetic operators: operators used to perform mathematical operations

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Description | Expression | Returns |
| + | Combines or adds two items | 12 + 3 | 15 |
| – | Subtracts one item from another | 12 –3 | 9 |
| \* | Multiplies two items | 12\*3 | 36 |
| / | Divides one item by another | 12/3 | 4 |
| % | Returns the remainder (modulus) after dividing one integer by another integer | 18%5 | 3 |
| \*\* | Raising a value to a power | 3\*\*2 | 9 |

Unary operator: requires just a single operand either before or after the operator

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Description | Expression | Returns |
| ++ | Increases a value by 1 | 12++ | 13 |
| –– | Decreases a value by 1 | 12–– | 11 |
| – | Changes the sign of a value | –12 | –12 |

Two types of unary operators:

•Prefix operators, which are placed before the variable

•Postfix operators, which are placed after the variable

•Prefix operator is applied before assignment operator:let x = 5;let y = ++x // x = 6 and y = 6

•Postfix operator is applied after assignment operator:

let x = 5;

let y = x++ // x = 6 and y = 5

An assignment operator (e.g., =) is used for assigning a value to a variable

•Compound assignment operators both assign a value and perform a calculation

•Interpreter will attempt to convert a nonnumeric to a numeric operand

|  |  |  |
| --- | --- | --- |
| Operator | Example | Equivalent to |
| = | x = y | x = y |
| += | x += y | x = x + y |
| –= | x –= y | x = x –y |
| \*= | x \*= y | x = x \* y |
| /= | x /= y | x = x/y |
| %= | x %= y | x = x % y |
| \*\*= | x \*\*= y | x = x\*\*y |
|  |  |  |

Comparison operators (relational operators): used to compare two operands

•Two nonnumerical operands are compared in lexicographical order

•String plus number: interpreter converts string to number or returns false

|  |  |  |
| --- | --- | --- |
| Operator | Example | Description |
| == | x == y | Tests whether x is equal in value to y |
| === | x === y | Tests whether x is equal in value to y and has the same data type |
| != | x != y | Tests whether x is not equal to y or has a different data type |
| !== | x !== y | Tests whether x is not equal to y and/or doesn’t have the same data type |
| > | x > y | Tests whether x is greater than y |
| >= | x >= y | Tests whether x is greater than or equal to y |
| < | x < y | Tests whether x is less than y |
| <= | x <= y | Tests whether x is less than or equal to y |

Conditional operators (ternary operators) return one of two possible values given the Boolean value of comparison

•Syntax: condition? trueValue: falseValue;

•Condition can be any expression that equals trueor false, including a Boolean variable

•Can return an expression instead of a value

•Falsy values, equivalent to false: ""(empty string), -0, 0, NaN, null, undefined

•Everything else is a truthy value, equivalent to true

•Can often use truthy and falsy values to make comparison operations more compact by omitting the comparison operator

Logical operators

•Used to combine expressions that will result in a Boolean value of trueor false

•Used for negating (swapping) a Boolean value

•Multiple conditions can be grouped within parentheses to create more complicated statements

|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Definition | Example | Description |
| && | and | (x === 5) && (y === 8) | Tests whether x is equal to 5 and y is equal to 8 |
| || | or | (x === 5) || (y === 8) | Test whether x is equal to 5 or y is equal to 8 |
| ! | not | ! (x < 5) | Test whether x is not less than 5 |

|  |  |  |
| --- | --- | --- |
| Name | Special Operator | Description |
| Property access | . | Appends an object, method, or property to another object |
| Array index | [] | Accesses an element of an array |
| Function call | () | Calls up functions or changes the order in which individual operations in an expression are evaluated |
| Comma | , | Separates multiple expressions in the same statement |
| Conditional expression | ?: | Executes one of two expressions based on the results of a conditional expression |
| Delete | delete | Deletes array elements, variables created without the varkeyword, and properties of custom objects |
|  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Special Operator | | Description | |
| Property exists | in | | Returns a value of trueif a specified property is contained within an object |
| Object type | instanceof | | Returns trueif an object is of a specified object type |
| New object | new | | Creates a new instance of a user-defined object type or a predefined JavaScript object type |
| Data type | typeof | | Determines the data type of a variable |
| Void | void | | Evaluates an expression without returning a result |

Operator precedence determines the order in which operations in an expression are evaluated

•Associativitydetermines precedence for operators with equal intrinsic precedence

•Examples:

•5 + 2 \* 8evaluates to 21

•30 / 5 \* 2evaluates to 12

•let x = 3;let y = 2;x = y \*= ++x; // Value of both x and y is 8

•(5 + 2) \* 8evaluates to 56

Accessing the browser console (console) displays error messages from the browser

•Locating an error in your program

•Browser console reports the line where detected each error is located

•Also reports lines that failed to run

•Be sure to make permanent corrections to code within your code editor

# Week 3

* An array is a set of data represented by a single variable name. You use an array when you want to store a group or a list of related information in a single, easily managed location. Lists of names, courses, test scores, and prices are typically stored in arrays.
  + The identifier you use for an array name must follow the same rules as identifiers for variables. It must begin with an uppercase or lowercase ASCII letter, dollar sign ($), or underscore (\_), can include numbers (but not as the first character), cannot include spaces, and cannot be reserved words.
* The most common way to create an array is with an array literal, a single statement that declares a variable and specifies array values as its content. The syntax of the array literal is:
  + let array = [values]
  + Program code. In the code, the words in the variable names are merged. Line 1: let array, equals, left bracket, values, right bracket, semi-colon.
  + where array is the variable name assigned to the array, and values is a comma-separated list of values stored within the array.
  + For example, the following statement creates an array, storing within it the three-letter abbreviations of the months of the year:
    - let months = [“Jan”,”Feb”,”Mar”,”Apr”,”May”,”Jun”,”Jul”,”Aug”,”Sep”,”Oct”,”Nov”,”Dec”];
  + Values stored within an array can involve several different data types. The following dataValues array stores a text string, a numeric value, a Boolean value, and the null value:
    - let dataValues = [“April”, 3, true, null];
  + You can initialize an array with no data values by leaving the contents within the brackets empty as the following statement demonstrates:
    - let dataValues = [];
  + Arrays are objects, so another way to declare and initialize an array is with the following new Array() object constructor:
    - let array= new Array (values)
    - let months = newArray(“Jan”,”Feb”,”Mar”,”Apr”,”May”,”Jun”,”Jul”,”Aug”,”Sep”,”Oct”,”Nov”,”Dec”);
  + The new Object() constructor also defines arrays based on the number items within the array. The general syntax of the statement is:
    - let array= newArray (length)
    - where length is the number of values within the array. The length argument must be entered as an integer between 0 and 232 − 1. The following statement creates the monthName array with 12 elements:
      * let monthName = newArray(12)
      * While the monthName array has been declared, it has not been initialized. The only thing that has happened is that memory for an array of length 12 has been allotted to the monthName variable. You can do the same thing with an array literal if you do not specify any values in the comma-separated list as in the following statement, which creates the dataValues array with four undefined values:

Let dataValues = [,,,];

* Each value stored in an array is called an element, and each element is identified by its position or index within the array. Indexes always start with the number 0, so the first element in any array has an index of 0, the second has an index of 1, and so forth. You can set a specific array value by its index using the expression

Array[index = value;

* + where index is the index number of the array element and value is the value stored at that location within the array. The values of the first several elements in the monthName array could be defined using the following statements:

monthName[0] = “January”;  
monthName[1] = “February”;  
monthName[2] = “March”;

* + Unlike with many other programming languages, arrays in JavaScript are dynamic in that they will automatically expand to allow for new elements. In the following code the dataValues array is declared with four elements. The next statement setting the value of an element with an index of four increases the length the dataValues array to five:

Let dataValues = [10,20,30,40];  
dataValues[4] = 50; //10,20,30,40 and 50 stored in the array

* + JavaScript also allows for the creation of sparse arrays in which some array values are left undefined so that the length of the array is greater than the number of defined values. The following commands create a sparse array with only two defined values out of 100 elements:

Let x = newArray();  
x[0] = “Aaron”;  
x[99] = “Zukov”;

Sparse arrays occur frequently in applications involving customer data where items such as mobile phone numbers or postal codes have not been stored for every individual.

* + JavaScript can treat the entire content of your array as text entries in a comma-separated list. For example, the following statements

Let x = [“Iowa”,”Kansas”,”Illinois”];  
document.write(x);  
will write the text string "Iowa,Kansas,Illinois" into the web page.

* + To determine an array’s current size, use the property:

array.length

* + were array is a reference to the variable storing the array. The value returned by the length property is equal to one more than the highest index number in the array (because array indexes start at 0 rather than 1.) So, if the highest index number is 11, then the value returned by the length property would be 12.
* Many applications store data in a rectangular format known as a matrix, in which the values are arranged in a rectangular grid. The following is an example of a matrix laid out in a grid of three rows and four columns:

4 2 1 2  
1 3 18 6  
3 7 3 4

* + Entries in a matrix are identified by the indexes for the rows and columns. The value 18 from this matrix is referenced using the index pair (2, 3) because that value is placed at the intersection of the second row and third column.
  + JavaScript does not support matrices. However, you can mimic the behavior of matrices by nesting one array within another in a structure called a multidimensional array. The following code recreates that matrix with an array containing three elements, each of which is an array containing four elements:

Var mArray = [  
[4,2,1,2],  
[1,3,18,6],  
[3,7,3,4]  
];

* + Values within a multidimensional array are referenced using the expression:

Array[x][y],

* + where x is the index of the outer array (the row) and y is the index of the inner array (the column.) Thus the expression mArray[1][2] would return the value 18 from the matrix’s second row and third column (remember that indexes start with 0 and not 1.) The expression mArray[2][1] would return the value from the third row and second column, which in this example is the number 7.
  + The number of rows in a multidimensional array is given by the length property. The number of columns can be determined by applying the length property to the first table row. For example, the expressions

mArray.length;  
mArray[0].length

would return values of 3 (the number of rows) and 4 (the number of columns). This assumes that every row has the same number of elements as the first row. You can continue to nest arrays in this fashion to create matrices of even higher dimensions.

* + One reason to use multidimensional arrays is to match values from different arrays within a single variable.
* The Document Object Model organizes HTML elements into collections where each element is an HTML Collection Object. For example, all hyperlinks are part of a collection of links, all input controls are part of a collection of form elements, and so forth. Though these collections are not arrays, they share many of the features of arrays.
  + HTML COLLECTION COLLECTION OF
  + embeds <embed> elements in the document
  + forms <form> elements in the document
  + form.elements Elements within a web form
  + getElementsByClassName(class) Elements in the document with belonging to the class class
  + getElementsByName(name) Elements in the document with a name attribute equal to name
  + getElementsByTagName(tag) Elements in the document with a tag name equal to tag
  + images <img> elements in the document
  + links <a> elements and <area> elements with a href attribute
  + scripts <script> elements in the document
  + styleSheets Stylesheet objects associated with the document
* To reference a specific element within an HTML collection, use either of the following expressions:

Objects[idref] OR objects.idref

* + where objects is a reference to an HTML collection of elements and idref is either an index number representing the position of the element within the collection or the value of the id attribute assigned to the element. As with arrays, the first element in a collection has an index value of 0, the second element has an index value of 1, and so forth. Thus, if the first inline image within a document has the tag: <img src=”logo.png” id=”TiptonLogo” alt=”Tipton Turbines”>

document.images[0]  
document.images[“TiptonLogo”]  
document.images.TiptonLogo

* + Other element collections are referenced in a similar way. As with JavaScript arrays, you can determine the number of elements within a collection with the length property. The following expression will return the number of images within the entire document:

Document.images.length

* + The ordering of the elements within an HTML collection reflects the order of the element tags within the HTML file.
* HTML collections can also be formed by searching through elements within the Document Object Model based on their class attribute, tag name, or name attribute using the following methods:

Document.getElementbyClassName (class)  
document.getElementbyTagName (tag)  
document.getElementbyName (name)

* + where class is the value of the class attribute, tag is the name of the HTML tag and name is the value of the name attribute. For example, to reference the first h1 element within the document, apply the following expression:

document.getElementByTagName(“h1”)[0]

* + Notice that three methods all use the phrase “document.getElements” (plural) as opposed to the document.getElementById() method, which uses the singular form because it returns only one object instead of a collection. A common mistake is to use the singular form, as in document.getElementByName(), which will result in an error.
  + Each of these three expressions returns all matching elements within the entire document. You can also return HTML collections within a specified part of the document by nesting the object references in the following format:

Object.objects

* + where object is an element that contains other elements and objects is a collection within that container. Thus, the following expression returns the collection of paragraphs nested within the first table cell element of the web page:

document.getElementsByTagName(“td”)[0].getElementsByTagName(“p”)

Continuing in this fashion, you can nest one object collection within another and then another to create an element collection specific to one branch of the document hierarchy.

* + Assuming that the first element in an array or an HTML collection has an index number of 1 rather than 0 is a common programming error for beginners. If you are working with an array collection and are seeing results offset by 1 from what you expect, check that your code accounts for 0 as the first index number.
  + Another common mistake is to omit the index number when using properties that should be applied to a specific element within a collection.
  + will result in an error because it attempts to apply the checked status to a collection of input elements. Instead, you must specify only a single element from the collection, as in the following expression that applies the checked status to the first input element:
* In your applications you will often need to repeat the same group of statements several times. Imagine if you had to repeat essentially the same command block dozens, hundreds, or even thousands of times—the code would become unmanageably long. Programmers deal with this kind of situation by creating program loops. A program loop is a command block that executes repeatedly until a stopping condition is met. For example, a program that writes content from an array could be written as a program loop that goes through each array item, writing content as it goes, and stopping only when it has reached the end of the array. There are several different types of program loops. The first to consider is the while loop.
* In a while loop, a command block is executed while a given condition is true but stops once that condition is no longer true. The syntax of a while loop is:

While (condition){  
 statements;  
}

* + where condition is a conditional expression that is either true or false and statements are the statements within the command block that are repeatedly executed as long as that conditional expression is true. Each repetition of the command block is called an iteration.
  + To avoid loops that never end, also known as infinite loops, the command block needs to include at least one statement that eventually results in a falsy value for the condition. Command blocks often use a counter, which is a variable whose value changes with each iteration. Once that counter fails to match the condition, the loop ends. For example, the following code includes a counter variable named i with an initial value of 1. With each iteration, the value of i increases by 1. The loop continues while i is less than or equal to 5.

Let I = 1;  
while (I <=5){  
 document.write(I + “<br>”);  
 i++; //increase the value by 1  
}  
// after the loop ends  
document.write(“<p>The value of I is equal to “ + I +”</p>”);

Through the iterations, the value of i steadily increases. When the counter exceeds a value of 5 the while condition is no longer met and the loop ends, continuing onto the next statement in the program. The following content is written to the web page:

1<br>2<br>3<br>4<br>5<br>  
<p>The value of I is equal to 6 </p>

* + It is common for programmers to use variables named i, j, or k as counters for program loops. This standard practice makes it easy for other programmers to recognize the loop counter without having to read detailed commentary about the code.
  + You can use a wide variety of counters with while loops by varying the initial value, the iteration of the counter, and the conditional expression.
    - INITIAL VALUE ITERATION WHILE CONDITION ITERATED VALUES
    - let i = 5 i++ i <= 10 i = 5, 6, 7, 8, 9, 10
    - let i = 5 i-- i > 0 i = 5, 4, 3, 2, 1
    - let i = 0 i += 60 i <= 180 i = 0, 60, 120, 180
    - let i = 1 i \*= 2 i <= 50 i = 1, 2, 4, 8, 16, 32
    - let i = 90 i /= 3 i > 5 i = 90, 30, 10
* The while loop is an example of a pretest loop in that the condition is evaluated before each iteration of the command block. Because of this, it is possible that command block will be halted before the first iteration. Another type of program loop, called the do while loop, is a posttest loop in which the condition is evaluated after the command block has been executed at least once. The syntax of the do while loop is:

Do {  
 statements;  
} while (condition);

* + Notice that the condition is placed at the end of the loop so that the command block is not tested prior to the first iteration. The following code uses a do while loop to generate a series of numbers and a concluding statement:

Let I = 1;  
do {  
 document.write(I + “<br>”);  
} while (I <=5);  
// after the loop ends  
document.write (“<p>The value of I is equal to “ + I + “</p>”);

1<br>2<br>3<br>4<br>5<br>  
<p>The value of I is equal to 6 </p>

* + Aside from the location of the stopping condition, there is no difference between the while and do while loops. Use the do while loop when you want to ensure that the command block will be executed at least once; use the while loop when your program does not require such a guarantee.
* Another pretest loop is the for loop, in which the initial condition, stopping condition, and iterative expression are placed within a single line of code. The syntax of the for loop is:

For (initial; condition; iteration){  
 statements;  
}

* + where initial is the initial condition before the command block is executed, condition is the condition that must be true for each iteration, and iteration is the change that occurs with each iteration of the command block. For example, the version of the for loop that generates a series of numbers with a concluding statement would be written as:

for (let I = 1; I <=5; i++){  
 document.write (I + “<br>”);  
}  
// after the loop ends  
document.write(“<p>The value of I is equal to “ + I + ”</p>”);

1<br>2<br>3<br>4<br>5<br>  
ReferenceError: 1 is not defined

* + The for loop is simpler and more compact than either the while or do while loops and thus for loops are the preferred method for writing loops. Note that the scope of the i counter is limited to the for loop. Attempting to reference the counter outside of the for loop will produce an error. If you need to reference the final value of the counter variable outside of the loop, you should use either the while or do while loops, but otherwise to avoid confusion use a for loop to limit the scope of your counters.
  + For loops can also be nested within one another to create code that iterates through two sets of counters. The following code demonstrates how to generate a web table by creating an outer loop that iterates through a set of table rows and an inner loop that iterates through a set of table cells within each row:

Document.write(“<table>”);  
for(let i=1; i<=2; i++) {  
 document.write(“<tr>”)  
 for (let j=1; j<=3; j++) {  
 document.write(“<td>” + I + “,” + j + “</td>”);  
 }  
 document.write(“</tr>”);  
}  
document.write(“</table>”);

The resulting web table has two table rows and three table data cells within each of those rows:

<table>  
 <tr>  
 <td>1,1</td><td>1,2</td><td>1,3</td>  
 </tr>  
 <tr>  
 <td>2,1</td><td>2,2</td><td>2,3</td>  
 </tr>  
</table>

* + There is no practical limit to the number of nested for loops you can employ in your program. Nested for loops are often used with multidimensional arrays to loop through each level of the nested arrays.
* To create a for loop that iterates through the contents of an array or HTML collection, apply the following general structure:

For (let i=0; 1<objects.length; i++) {  
 statements;  
}

* + where objects is a reference to either an array or HTML collection. The counter starts with a value of 0 (because 0 is the index of the first element in the list) with the loop continuing if the counter is less than the value of the length property. Recall that the index of the last item in an array or collection will always be one less than the length value. For example, an array with 100 items will have indexes that range from 0 up to 99.
  + Once you have defined a collection, you can work with individual collection objects as you would individual array elements. The following code demonstrates how to apply an event handler to every input element within a document:

Let allInputs = document.getElementsByTagName (“input”);  
for (let i=0; I < allInputs.length; i++){  
 allInputs[i].addEventListener(“click”,checkOrder);  
}

* + Use a for loop to write the game results into cells of the calendar table. The for loop will have the following general structure:

For (let i=0; i<gameDates.length; i++) {  
 write a game result into a table cell   
}

* + with the number of games determined by the gameDates array shown earlier in Figure 3-1. With each iteration of the loop, the following contents will be written into the table cell matching the date on which the game was played:

<p> gameOpponents[i] <br>  
 gameResults[i] : (runsScored[i] – runsAllowed[i])  
</p>

* + where the gameOpponents array provides the opponent for a particular day, the gameResults array provides the result of the game, the runsScored array retrieves the number of runs scored by Tipton, and the runsAllowed array retrieves the number of runs scored by Tipton’s opponent. For example, the information on Tipton’s first game will be written as:

<p>Bettendorf<br>  
 W: (2 - 1)  
</p>

* + To match a game to a table cell, use the date stored in the gamesDate array and match it to the id value of a table cell (recall that each table cell has an id for a specific calendar date.) Thus, the table cell matching a game played on a specific date would be referenced using the expression

Let tableCell = document.getElementById(gameDates[i])

* + Finally, the table cells are not empty, so any content will have to be added to the HTML content already present in the cell instead of overwriting it. JavaScript provides the following insertAdjacentHTML() method to insert additional content into an element:

Element.insertAdjacentHTML(position,text)

* + where element is the element into which the new content is inserted, position is the location of the new content, and text is the text of the content. The position argument has the following values:
    - * "beforeBegin"—to insert new content directly before the element’s opening tag
      * "afterBegin"—to insert new content directly after the element’s opening tag
      * "beforeEnd"—to insert new content directly before the element’s closing tag
      * "afterEnd"—to insert new content directly after the element’s closing tag
      * For this application, you will insert the new content directly before each table cell’s closing tag, using a position value of "beforeEnd".
  + Because arrays, collections, and program loops are so often used together, JavaScript supports several methods to work with array items directly without creating a loop.
  + When designing a program that involves loops, especially a large and complex program, it can be challenging to explain the structure of the program and the relationships between its parts to other team members who might be working with you to create it. It’s common for programmers to create a visual representation to illustrate the parts of a program and how they fit together both before and during development. For loops, such diagrams often take the form of a flowchart, which shows program components as boxes of different shapes, with lines connecting those components that communicate with each other. A flowchart often includes arrows to indicate the direction that information flows between components. Although software is available to create professional-looking flowcharts, most programmers create flowcharts on white boards.
* Array methods that replace program loops are a useful JavaScript feature, and because these methods are built into the language, they are usually faster than program loops and will make your code simpler and more compact. Each of these methods employs a callback function, which is a function passed as a parameter to another function or method. One such method is the forEach() method, which calls a function for each element within an array:

Array.forEach(callback, thisArg)

* + where array is a reference to an array, callback is the function called for each array element, and thisArg is an optional parameter containing a value that can be passed to the callback function. The callback function has the syntax:

function callback (arrValue, index, array){  
 statements;  
}

* + where arrValue is the value of the current array element during each iteration within the array, index is the index of the current array element, and array is the name of the array. Only the arrValue parameter is required; the other two are optional.
  + The following code uses the forEach() method to apply the writeValue() function to each element within the x array:

Let x = [1, 3, 5, 10];  
x.forEach (writeValue);  
  
function writeValue(arrValue){  
document.write(“<td>” + arrValue + “</td>”);  
}

<td>1</td><td>3</td><td>5</td><td>10</td>

* + With the forEach() method, you don’t have to explicitly write the code for the program loop, calculate the size of the array, or worry about iterating past the last array element. The method automatically applies the callback function to each array element for you.
  + You can replace the name of the callback function with the code of an anonymous function, written directly within the forEach() method.
  + The forEach() method can also be used to change array values. The following code calls the stepUp5() function to increase the value of each item in the x array by 5:

Let x = [1,3,5,10];  
x.forEach(stepUp5);  
function stepUp5 (arrValues, I, arr) {  
 arr[i] = arrValue + 5;  
}

* + In this example, the stepUp5() function has three parameters: the arrValue parameter representing the value of the array element at each iteration, i representing the index number at each iteration, and arr representing the name of the array. The result is that value of the x array will be changed from [1, 3, 5, 10] to [6, 8, 10, 15].
  + However, note that none of these methods can be applied to HTML collections, which, though they often act like arrays, are not arrays.
    - ARRAY METHOD DESCRIPTION
    - every(callback, thisArg) Tests whether the value of the callback function is true for all array elements
    - filter(callback, thisArg) Creates a new array populated with the elements of the array that return a value of true from the callback function
    - forEach(callback, thisArg) Applies the callback function to each array element
    - map(callback, thisArg) Creates a new array by passing the original array elements to the callback function, which returns the mapped value of those elements
    - reduce(callback, thisArg) Reduces the array by keeping only those elements returning a true value from the callback function
    - reduceRight(callback, thisArg) Reduces the array starting from the last element by keeping only those elements returning a true value from the callback function
    - some(callback, thisArg) Tests whether the value of callback function is true for at least one array element
    - find(callback, thisArg) Returns the value of the first array element returning a true value from the callback function
    - findIndex(callback, thisArg) Returns the index of the first array element returning a true value from the callback function
  + As your programs increase in size and complexity, the ability to write efficient code becomes essential. Bloated, inefficient code is particularly noticeable with program loops that might repeat the same set of commands hundreds or thousands of times. A millisecond wasted due to one poorly written command can mean an overall loss of several seconds when repeated a thousand times. There are several ways of adding efficiency to your program loops. One is to place all calculations that will not change during the loop, outside of the loop. For example, the expression

Document.getElementsByTagName (“p”).length

* + - * searches through the entire document tree to count the number of paragraphs. The following for statement

for (let I = 0; i<document.getElementsByTagName (“p”).length; i++)

* + - * will perform that search with each iteration. In a long document, this can result is a serious performance hit. Instead, place the length calculation outside the loop as follows:

let pCount = document.getElementsByTagName(“p”).length;  
for (let I = 0; I < pCount; i++)

* Often an application will need to execute a different set of statements depending on varying conditions. A shopping cart application might need to run different code depending on the customer’s choice of shipping or payment. The shopping cart might need to run one set of operations for overnight shipping and different set of operations for standard shipping. A payment using a credit card might require a different set of functions from functions applied to payment using a gift card.
* The process of choosing which code to execute in response to circumstance is known as decision making. The special types of JavaScript statements used for making decisions are called decision-making statements, decision-making structures, or conditional statements. The most common type of decision-making statement is the if statement.
* The syntax of the if statement is:

If (condition){  
 statements;  
}

* + where condition is a Boolean expression that is either true or false and statements are part of the command block that runs when that condition is true. If the command block contains only a single statement you can dispense with the command block and write the if statement as:

if (condition) statement;

* + but it is considered good programming practice to always enclose even a single statement within a command block. The following if statement tests whether the day variable is equal to "Friday" and if that condition is true, displays a special greeting message:

if (day ===”Friday”){  
 window.alert(“Get ready for the weekend!”);  
}

* + A very common error is to use the = symbol in place of the === conditional operator to test for the truth of a condition. The = symbol is an assignment operator and assigns one value to another; it does not test their equality.
* The if statement will only take an action if the condition is true; otherwise it will take no action. To run one command block if the condition is true and a different command block if the condition is not true, use the if else statement:

If (condition){  
 statements if condition is true  
}else {  
 statements if condition is not true  
}

* + The following if else statement displays one greeting if the day variable equals "Friday" and a different greeting if otherwise:

If (day ===”Friday”){  
 window.alert (“Get ready for the weekend!”);  
} else {  
 window.alert(“Have a great day!”);  
}

* + The else command block runs if the condition has any falsy value. Thus, a condition that evaluates to false or null or undefined will trigger the else command block.
* In some applications, there might be several possible conditions to consider. For example, a shopping cart payment might be made with a credit card, a gift card, or an online banking account. For those situations, you can apply multiple if statements in the following structure:

If (condition1){  
 statements if condition1 is true  
} else If (condition2){  
 statements if condition2 is true  
} else {  
 statements if neither condition1 nor condition2 is true  
}

* + In the else if structure, condition1 is tested first. If that condition is true, the corresponding command block executes. Only if it is not true is condition2 tested. If that condition is true, its command block runs. If neither condition1 nor condition2 are true, only then does the final command block run.

If(day===”Friday”){  
 windows.alert (“Get ready for the weekend!”);  
} else if (day===”Monday”) {  
  
 window.alert(“Start of another work week.”)  
} else {  
 window.alert(“Have a great day!”)  
}

* + The else condition is considered the “default” option, applied only when all other possibilities have been tested and rejected. Proceeding in this fashion, you can add as many else if statements as your application requires until you have covered all possible contingencies.
  + Note that because decision-making statements end with the first true condition, you need to order your statements to remove overlapping conditions. In this case, you first test for games that end in less than five innings and then test for games that end in less than nine innings. Switching the order would have treated all games with less than nine innings as shortened but finalized games, even those that lasted a single inning.
* As with program loops, you can nest decision-making statements within one another, creating a series of conditions that all must be true before an action is taken. This type of structure is called a nested decision-making structure. The following code shows an example of nested if statements:

If (day===”Friday”){  
 if (time === “8am”) {  
 window.alert (“Start of the last day of the week.”);  
} else if (time === “5pm”){  
 window.alert (“Time to start the weekend!”);  
} else {  
 window.alert (“A few more hours until the weekend.”);  
}  
} else if (day === “Monday”){  
 window.alert(“Start of another work week.”);  
} else {  
 window.alert(“Have a great day!”);  
}

* + In this example, if the day is "Friday", one of 3 possible messages will be displayed based on the value of the time variable; otherwise two possible messages will be displayed depending on whether the day is "Monday" or another day.
  + With nested statements, it is very easy to lose track of the opening and closing braces. Mismatching the braces will most likely result in an error. To assist you, most code editors will include visual clues matching opening and closing braces.
  + A great challenge for any web developer is ensuring that program code is supported by the browser. Older browser versions may not recognize the latest enhancements made to ECMAScript, and customers running those browsers will be faced with an application that fails due to its lack of support. If you feel that a feature of your code might not be universally supported, you can add a browser test confirming that the feature is recognized by the JavaScript interpreter and providing alternate statements if it is not. The general syntax is:

If (feature){  
 statements that use the feature  
} else {  
 statements that use replacement code  
}

* + where feature is a JavaScript object, property, or method that should be tested for browser support. If the feature returns true, you can apply statements that use the feature; but if the condition returns a falsy value (such as undefined), you can supply an alternate set of commands that use a different feature that is supported.
  + Browser testing is often used to ease the transition into new ECMASCript features, so that the most current features are applied where supported and older features are used where needed. As newer features become more widely supported, developers can simplify their code by removing the browser test and the alternate set of instructions. Throughout the years, many statements using outdated methods have been winnowed away in this fashion, resulting in faster, more efficient code.
  + As the number of possible conditions increases, the entire if else if structure can become large and unwieldy. An alternative to a long list of else if conditions is the following switch statement:

Switch (expression) {  
 case label1: statements; break;  
 case label2: statements; break;  
 case label3: statements; break;  
 default: statements; break;  
}

* + where expression is a statement that returns a value, label1 , label2 , label3 , and so on are possible values of that expression, statements are the commands run with each possible value, and the final default option is run if none of the listed labels match the expression’s value. The following switch statement demonstrates how to run a different set of statements based on the value of the day variable:

switch (day){  
 case “Friday”: alert(“Thank goodness it’s Friday!”); break;  
 case “Monday”: alert(“Blue Monday”); break;  
 case “Saturday”: alert(“Sleep in today.”); break;  
 default: alert(“Today is” + day);  
}

* + Case labels must be discrete values and cannot use operators. Thus, you cannot define a case label based on numeric ranges like < 20 or >= 10. If you need a numeric range, use an else if construction instead of a switch statement.
  + The break statement, marking the end of each case, is an optional keyword that halts the execution of the switch statement once a matching case has been found. For programs in which more than one label might match the expression, omit the break statements and the JavaScript interpreter will continue moving through the case labels, running all statements in which a match has been found. This situation is known as fallthrough.
  + Although you are finished with the calendar, you still should become familiar with some features of program loops and conditional statements for future work with these JavaScript structures. You will examine three features in more detail—the break, continue, and label statements.
* The break statement can be used anywhere within any program loop or conditional statement. When a break statement is encountered, the execution of the code passes to the next set of statements. Breaks are most often used to exit a program loop before the stopping condition is met, as in the following program loop that examines the customerID array for a specific customer ID number:

For (let i=0; i<customerID.length; i++){  
 if (customerID[i] === “C-14281”){  
 window.alert(“C-14281 is found”);  
 break; // stop processing the for loop   
}  
}

* + Once the specific customer ID has been located, there is little point in continuing the for loop. The break command saves the JavaScript interpreter from having to fruitlessly examine the rest of an array that might contain tens of thousands of elements.
  + The continue statement is like the break statement except that instead of stopping a program loop altogether, the continue statement stops only the current iteration and continues on to the next iteration. A continue statement is useful in programs that need to avoid undefined values that can cause the program to fail. In the following code, a for loop is used to examine the contents of an array of customer email addresses. However, the customerEmail array may be sparse with several undefined values that would result in errors if processed. This problem is avoided with an if statement that continues the loop to the next iteration when an undefined value is detected:

For (let i=0; i<customerEmail.length; i++){  
 if (customerEmail [i] === undefined){  
 contine;  
} else{  
 // statements to process the email address  
}  
}

* Statement labels identify statements in the code so that they can be referenced elsewhere in the program. The syntax of the statement label is:

Label: statements

* + where label is the text of the label and statements are the statements identified by the label. You have already seen labels with the switch statement, but labels can also be used with other program loops and conditional statements to provide more control over how statements are processed. Labels often are used with break and continue statements to direct the program flow to a specific set of statements. The syntax to reference a label in such cases is simply

break: label; OR continue: label;

* + For example, the following for loop uses a statement label not only to jump out of the programming loop when the text string “C-14281” is found but also to jump to a location in the script identified by the nextReport label and to continue to process the statements found there:

For (let i=0; i<customerID.length; i++){  
 if (customerID[i] === “C-14281”){  
 window.alert(“C-14281 is found”);  
 break; nextReport// stop processing the for loop   
 }  
}  
  
nextReport: statements

* + Labels are most often used with nested loops when you need to break out of a loop completely, no matter how deeply nested you might be.
  + Spaghetti code is a pejorative programming term that refers to convoluted or poorly written code. One hallmark of spaghetti code is the frequent branching from one section of code to another, making it difficult to track the program line-by-line as it is executed. A change in one part of the program could lead to unpredictable changes in a completely different section of the code. Many developers discourage the use of label statements unless absolutely necessary. They can confuse a programmer trying to fix code in which a program loop can end before its stopping condition, or code in which statements are not processed in the order that they are written in a document. Almost all of the tasks in a program can also be performed by carefully setting up the conditions for program loops without forcing jumps to labeled sections. Even with the best of intentions, spaghetti code can easily occur in environments in which the same code is maintained by several people or passed from one employee to another. A programmer might add a new feature that is needed right away without adequately documenting the changes made to the code or without considering the impact of those changes on other programs. To avoid or at least reduce the occurrence of spaghetti code, always document your code, and develop a structure that is easy for others to follow.  
    if (exam>90){  
     window.alert (“You passed with an A”);  
    } else if (exam>80){  
     window.alert(“You passed with a B”);  
    } else if (exam>70){  
     windows.alert(“You passed with a C”)  
    } else{  
     windows.alert(“You did not pass”)
  + }
* An array contains a set of data represented by a single variable name. You can think of an array as a collection of variables contained within a single variable. Each piece of data contained in an array is called an element. An index is an element’s numeric position within the array.
* A loop statement is a control flow statement that repeatedly executes a statement or a series of statements while a specific condition is true or until a specific condition becomes true. Loop statements in JavaScript include the while, do while, and for statements.
* The while statement is used for repeating a statement or series of statements as long as a given conditional expression evaluates to true.
* Each repetition of a looping statement is called an iteration.
* An infinite loop is a situation in which a loop statement never ends because its conditional expression is never false.
* The do while statement executes a statement or statements once, and then it repeats the execution as long as a given conditional expression evaluates to true.
* The for statement is used to repeat a statement or series of statements as long as a given conditional expression evaluates to true.
* The continue statement halts a looping statement and restarts the loop with a new iteration.
* The process of choosing which code to execute at a given point in an application is known as decision making. In JavaScript, you use the if, if else, else if, and switch statements to create decision-making structures.
* The if statement is used to execute specific programming code if the evaluation of a conditional expression returns a value of true.
* An if statement that includes an else clause is called an if else statement.
* When one decision-making statement is contained within another decision-making statement, they are referred to as nested decision-making structures.
* The switch statement controls program flow by executing a specific set of statements, depending on the value of an expression.
* A break statement is used to exit control statements, such as the switch statement or the while, do while, and for looping statements.

# Week 5 | Chapter 4

* To write a program, you must understand the syntax of the programming language you are using. You must also understand computer-programming logic. The term logic refers to the arrangement of operations within the program to achieve its intended goal.
* Any error in a program that causes it to function incorrectly, whether because of incorrect syntax or flaws in logic, is called a bug. The term debugging refers to the act of tracing and resolving errors in a program. Grace Murray Hopper, a mathematician who was instrumental in developing the Common Business-Oriented Language (COBOL) programming language, is said to have first coined the term. A moth short-circuited a primitive computer that Hopper was using. Removing the moth “debugged” the system and resolved the problem. Today, the term “bug” refers to any sort of problem in the design and operation of a program.
* There are three general types of errors within a program: load-time errors, runtime errors, and logic errors.
* A load-time error, also known as a syntax error, occurs when the program is initially loaded by the browser. One of the tasks of a JavaScript interpreter is to confirm that there are no errors in the syntax. A common syntax error is the misspelling of a JavaScript keyword such as using document.writ() in place of document.write(). Other syntax errors would be forgetting to end a command block with a closing curly brace or forgetting to enclose a text string within a set of quotation marks.
  + Notice, however, that the console says nothing about the syntax error in omitting the closing brace. When the debugger encounters a syntax error, it stops processing the code so that any subsequent errors are not reported. In fixing a syntax error, you might find that fixing one error leads you further down the code to the next error.
* When the interpreter loads the script without finding any syntax errors, it will next attempt to run the code. At this point, a runtime error may appear, which is an error that occurs when the interpreter is unable to run the code. Runtime errors may manifest themselves for several reasons such as attempting to reference a function or variable that has not been declared, using an undefined value in an expression, or performing an illegal mathematical operation such as calculating the square root of a negative number.